Red Hat JBoss Fuse 6.3

Apache Camel コンポーネントリファレンス

Camel コンポーネントの設定リファレンス

Last Updated: 2023-02-26

Red Hat JBoss Fuse 6.3 Apache Camel コンポーネントリファレンス

Camel コンポーネントの設定リファレンス

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Apache_Camel_Component_Reference.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

Apache Camel には 100 を超えるコンポーネントがあり、各コンポーネントは高度な設定が可能です。本ガイドでは、各コンポーネントの設定について説明します。

目次

第1章 コンポーネントの概要 51
1.1. APACHE KARAF の CAMEL コンポーネントのリスト 51
コンポーネントの表 51
1.2. JBOSS EAP の CAMEL コンポーネントのリスト 72
コンポーネントの表 72
1.3. JBOSS EAP でのカスタム CAMEL コンポーネントのデプロイ 83
1.3.1. module.xml 定義の追加 83
1.3.2. 参照の追加 84

第2章 ACTIVEMQ 85
ACTIVEMQ コンポーネント 85
URI 形式 85
オプション 85
CAMEL ON EAP デプロイメント 85
接続ファクトリーの設定 86
SPRING XML を使用した接続ファクトリーの設定 86
接続プールの使用 86
ルートでの MESSAGELISTENER POJO の呼び出し 87
ACTIVEMQ 宛先オプションの使用 88
アドバイザリーメッセージの消費 88
コンポーネント JAR の取得 89

第3章 AHC 90
ASYNC HTTP CLIENT (AHC)コンポーネント 90
URI 形式 90
AHCENDPOINT オプション 90
AHCCOMPONENT オプション 92
メッセージヘッダー 93
メッセージボディー 94
レスポンスコード 94
AHCOPERATIONFAILEDEXCEPTION 94
GET または POST を使用した呼び出し 94
呼び出す URI の設定 94
URI パラメーターの設定 95
HTTP メソッド(GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE)を HTTP プロデューサーに設定する方法 95
CHARSET の設定 96
エンドポイント URI からの URI パラメーター 96
メッセージの URI パラメーター 96
レスポンスコードの取得 96
ASYNCHTTPCLIENT の設定 96
関連項目 97

第4章 AHC-WS 98
ASYNC HTTP CLIENT (AHC) WEBSOCKET クライアントコンポーネント 98
URI 形式 98
AHC-WS オプション 98
WEBSOCKET でのデータの書き込みと読み取り 98
データの書き込みまたは読み取りのための URI の設定 98

第5章 AMQP 100
AMQP 100

URI 形式 100
AMQP オプション 100
使用方法 100
AMQP コンポーネントの設定 100
トピックの使用 101

第6章 APNS 102
APN コンポーネント 102
URI 形式 102
オプション 102
プロデューサー 102
コンシューマー 103
コンポーネント 103
エクスチェンジデータ形式 103
メッセージヘッダー 103
APNSSERVICEFACTORY BUILDER コールバック 104
サンプル 104
CAMEL XML ルート 104
CAMEL JAVA ルート 105
CAMEL コンテキストを作成し、プログラムで APNS コンポーネントを宣言します。 105
APNSPRODUCER - IOS ターゲットデバイスがヘッダー経由で動的に設定された "CAMELAPNSTOKENS" 106
APNSPRODUCER - IOS ターゲットデバイスが URI 経由で静的に設定される 106
APNSCONSUMER 106
関連項目 106

第7章 ATMOSPHERE-WEBSOCKET 107
ATMOSPHERE WEBSOCKET SERVLET コンポーネント 107
URI 形式 107
WEBSOCKET でのデータの読み取りと書き込み 107
データの読み取り/書き込みのための URI の設定 107

第8章 ATOM 109
ATOM コンポーネント 109
URI 形式 109
オプション 109
CAMEL ON EAP デプロイメント 111
エクスチェンジデータ形式 111
メッセージヘッダー 111
サンプル 111

第9章 AVRO 114
AVRO コンポーネント 114
APACHE AVRO の概要 114
AVRO データフォーマットの使用 115
CAMEL での AVRO RPC の使用 115
AVRO RPC URI オプション 116
AVRO RPC ヘッダー 117
例 117

第10章 AWS 119
10.1. AWS コンポーネントの概要 119
Amazon Web Services の Camel コンポーネント 119
10.2. AWS-CW 120
CW コンポーネント 120

URI 形式 120
URI オプション 120
使用方法 121
CW プロデューサーによって評価されるメッセージヘッダー 121
Advanced AmazonCloudWatch configuration 122
Dependencies 122
10.3. AWS-DDB 123
DDB コンポーネント 123
URI 形式 123
URI オプション 123
使用方法 124
DDB プロデューサーによって評価されるメッセージヘッダー 124
BatchGetItems 操作中に設定されたメッセージヘッダー 126
DeleteItem 操作時に設定されたメッセージヘッダー 126
DeleteTable 操作時に設定されたメッセージヘッダー 126
DescribeTable 操作中に設定されたメッセージヘッダー 127
GetItem 操作時に設定されたメッセージヘッダー 127
PutItem 操作中に設定されたメッセージヘッダー 127
Query 操作時に設定されたメッセージヘッダー 128
Scan 操作時に設定されたメッセージヘッダー 128
UpdateItem 操作時に設定されたメッセージヘッダー 128
高度な AmazonDynamoDB 設定 128
Dependencies 129
10.4. AWS-DDBSTREAM 129
DynamoDB ストリームコンポーネント 129
URI 形式 130
URI オプション 130
シーケンス番号 131
バッチコンシューマー 131
AmazonDynamoDBStreamsClient configuration 131
AWS 認証情報の指定 131
Downtime のコッピング 132
Dependencies 132
10.5. AWS-EC2 132
EC2 コンポーネント 132
URI 形式 132
URI オプション 132
EC2 プロデューサーによって評価されるメッセージヘッダー 133
Dependencies 134
10.6. AWS-KINESIS 134
Kinesis コンポーネント 134
URI 形式 134
URI オプション 135
バッチコンシューマー 135
Kinesis コンシューマーによって設定されるメッセージヘッダー 135
AmazonKinesis の設定 136
AWS 認証情報の指定 136
Dependencies 136
10.7. AWS-S3 136
S3 コンポーネント 136
URI 形式 137
URI オプション 137
バッチコンシューマー 139

使用方法 139
S3 プロデューサーによって評価されるメッセージヘッダー 139
S3 プロデューサーによって設定されたメッセージヘッダー 140
S3 コンシューマーによって設定されたメッセージヘッダー 140
高度な AmazonS3 設定 142
Dependencies 142
10.8. AWS-SDB 142
SDB コンポーネント 142
URI 形式 143
URI オプション 143
使用方法 144
SDB プロデューサーによって評価されるメッセージヘッダー 144
DomainMetadata 操作中に設定されたメッセージヘッダー 145
GetAttributes 操作中に設定されたメッセージヘッダー 145
ListDomains 操作中に設定されるメッセージヘッダー 146
Select 操作時に設定されたメッセージヘッダー 146
高度な AmazonSimpleDB 設定 146
Dependencies 147
10.9. AWS-SES 147
SES コンポーネント 147
URI 形式 147
URI オプション 147
使用方法 148
SES プロデューサーによって評価されるメッセージヘッダー 148
SES プロデューサーによって設定されたメッセージヘッダー 149
AmazonSimpleEmailService の高度な設定 149
Dependencies 149
10.10. AWS-SNS 150
SNS コンポーネント 150
URI 形式 150
URI オプション 150
使用方法 151
SNS プロデューサーによって評価されるメッセージヘッダー 151
SNS プロデューサーによって設定されたメッセージヘッダー 151
AmazonSNS の高度な設定 151
Dependencies 152
10.11. AWS-SQS 152
SQS コンポーネント 152
URI 形式 152
URI オプション 152
バッチコンシューマー 157
使用方法 157
SQS プロデューサーによって設定されたメッセージヘッダー 157
SQS コンシューマーによって設定されたメッセージヘッダー 157
AmazonSQS の高度な設定 157
Dependencies 158
JMS スタイルのセレクター 158
10.12. AWS-SWF 158
SWF コンポーネント 158
URI 形式 159
URI オプション 159
使用方法 161
SWF ワークフロープロデューサーによって評価されるメッセージヘッダー 161

SWF ワークフロープロデューサーによって設定されたメッセージヘッダー 162
SWF ワークフローコンシューマーによって設定されたメッセージヘッダー 162
SWF アクティビティープロデューサーによって設定されたメッセージヘッダー 162
SWF Activity コンシューマーによって設定されたメッセージヘッダー 163
高度な amazonSWClient 設定 163
Dependencies 163

第11章 BEAN 164
BEAN コンポーネント 164
URI 形式 164
オプション 164
使用 165
エンドポイントとしての BEAN 166
JAVA DSL BEAN 構文 166
BEAN バインディング 167
BEAN 言語 167

第12章 BEAN バリデーター 168
BEAN バリデーターコンポーネント 168
URI 形式 168
URI オプション 168
OSGI デプロイメント 169
例 169

第13章 BEANSTALK 172
BEANSTALK コンポーネント 172
DEPENDENCIES 172
URI 形式 172
一般的な URI オプション 172
プロデューサー UIR オプション 173
コンシューマー UIR オプション 173
コンシューマーヘッダー 174
例 175

第14章 BINDY 176
BINDY コンポーネント 176
CAMEL ON EAP デプロイメント 176

第15章 BOX 177
BOX コンポーネント 177
URI 形式 177
BOX コンポーネント 178
プロデューサーエンドポイント： 179
エンドポイント接頭辞コラボレーション 179
コラボレーションの URI オプション 180
エンドポイント接頭辞イベント 180
イベントの URI オプション 181
エンドポイント接頭辞グループ 181
グループの URI オプション 182
エンドポイント接頭辞検索 182
検索の URI オプション 182
エンドポイント接頭辞のコメントと共有コメント 183
コメントおよび共有コメントの URI オプション 183
エンドポイント接頭辞ファイルと共有ファイル 184

ファイルおよび SHARED-FILES の URI オプション 184
エンドポイント接頭辞フォルダーおよび共有フォルダー 185
フォルダーまたは共有フォルダーの URI オプション 186
エンドポイント接頭辞 SHARED-ITEMS 186
SHARED-ITEMS の URI オプション 187
エンドポイント接頭辞ユーザー 187
ユーザーの URI オプション 188
コンシューマーエンドポイント： 188
POLL-EVENTS の URI オプション 189
メッセージヘッダー 189
メッセージボディー 189
型コンバーター 189
ユースケース 189

第16章 BRAINTREE 191
BRAINTREE コンポーネント 191
URI 形式 191
BRAINTREECOMPONENT 192
プロデューサーエンドポイント： 192
ENDPOINT PREFIX ADDON 193
エンドポイント接頭辞 アドレス 193
エンドポイント接頭辞 CLIENTTOKEN 194
エンドポイント接頭辞 CREDITCARDVERIFICATION 194
エンドポイント接頭辞 CUSTOMER 195
エンドポイント接頭辞 DISCOUNT 196
エンドポイント接頭辞 MERCHANTACCOUNT 196
エンドポイント接頭辞 PAYMENTMETHOD 196
エンドポイント接頭辞 PAYMENTMETHODNONCE 197
エンドポイント接頭辞 PLAN 198
エンドポイント接頭辞 SETTLEMENTBATCHSUMMARY 198
エンドポイント接頭辞 SUBSCRIPTION 198
エンドポイント接頭辞 TRANSACTION 200
エンドポイント接頭辞 WEBHOOKNOTIFICATION 201
コンシューマーエンドポイント 202
メッセージヘッダー 202
MESSAGE BODY 202
EXAMPLES 202

第17章 参照 204
コンポーネントの参照 204
URI 形式 204
例 204

第18章 CACHE 205
18.1. キャッシュコンポーネント 205
Camel 2.1 以降で利用可能 205
URI 形式 205
オプション 205
キャッシュコンポーネントのオプション 208
Message Headers Camel 2.8 以降 208
キャッシュプロデューサー 209
キャッシュコンシューマー 209
キャッシュプロセッサー 209
例 1: キャッシュの設定 210

例 2: キーのキャッシュへの追加 210
例 2: キャッシュ内の既存のキーの更新 210
例 3: キャッシュ内の既存のキーの削除 210
例 4: キャッシュ内のすべての既存キーの削除 210
例 5: キャッシュに登録されている変更のプロセッサーおよびその他のプロデューサーへの通知 211
例 6: プロセッサーを使用したペイロードをキャッシュ値で選択的に置き換える 211
例 7: キャッシュからのエントリーの取得 212
例 8: キャッシュ内のエントリーの確認 212
EHCache の管理 212
Cache replication Camel 2.8+ 213
例：JMS キャッシュレプリケーション 213
18.2. CACHEREPLICATIONJMSEXAMPLE 213
例：JMS キャッシュレプリケーション 213

第19章 CDI 218
CDI コンポーネント 218
CAMEL ON EAP デプロイメント 218

第20章 CASSANDRA 219
CAMEL CASSANDRA コンポーネント 219
URI 形式 219
エンドポイントオプション 219
メッセージ 220
受信メッセージ 220
送信メッセージ 220
リポジトリー 220
IDEMPOTENT リポジトリー 221
集約リポジトリー 221

第21章 CHUNK 223
チャンクコンポーネント 223
URI 形式 223
オプション 223
チャンクコンテキスト 224
動的テンプレート 225
サンプル 225
電子メールのサンプル 226

第22章 クラス 227
クラスコンポーネント 227
URI 形式 227
オプション 227
使用 227
作成されたインスタンスでのプロパティーの設定 228

第23章 CMIS 229
CMIS コンポーネント 229
URI 形式 229
URI オプション 229
使用方法 230
プロデューサーによって評価されるメッセージヘッダー 230
プロデューサー操作のクエリー中に設定されたメッセージヘッダー 230

第24章 COMETD 232
COMETD コンポーネント 232

URI 形式 232
例 232
オプション 232
認証 234
COMETD コンポーネントでの SSL の設定 234

第25章 コンテキスト 236
コンテキストコンポーネント 236
URI 形式 236
例 236
コンテキストコンポーネントの定義 236
コンテキストコンポーネントの使用 237
エンドポイントの命名 238

第26章 CONTROLBUS コンポーネント 239
CONTROLBUS コンポーネント 239
コマンド 239
オプション 239
サンプル 240
ROUTE コマンドの使用 240
パフォーマンス統計の取得 240
SIMPLE 言語の使用 241

第27章 COUCHDB 242
CAMEL COUCHDB コンポーネント 242
URI 形式 242
オプション 242
HEADERS 243
メッセージボディー 243
サンプル 243

第28章 暗号（デジタル署名） 245
デジタル署名の暗号コンポーネント 245
CAMEL ON EAP デプロイメント 245
はじめに 246
URI 形式 246
オプション 247
1）RAW キー 248
2）キーストアとエイリアス。 248
3）JCE プロバイダーとアルゴリズムの変更 248
4）署名メッジヘッダーの変更 249
5）バッファーサイズの変更 250
6）キーを動的に指定します。 250

第29章 CXF 252
CXF コンポーネント 252
CAMEL ON EAP デプロイメント 252
URI 形式 252
オプション 253
データ形式の説明 258
APACHE ARIES BLUEPRINT での CXF エンドポイントの設定 259
MESSAGE モードで CXF の LOGGINGOUTINTERCEPTOR を有効にする方法 260
RELAYHEADERS オプションの説明 261
POJO モードでのみ利用可能 261

リリース 2.0 以降の変更点 262
SPRING での CXF エンドポイントの設定 263
HOW TO MAKE THE CAMEL-CXF COMPONENT USE LOG4J INSTEAD OF JAVA.UTIL.LOGGING 266
HOW TO LET CAMEL-CXF RESPONSE MESSAGE WITH XML START DOCUMENT 266
POJO データ形式で CAMEL-CXF エンドポイントからメッセージを消費する方法 267
POJO データ形式で CAMEL-CXF エンドポイントのメッセージを準備する方法 268
PAYLOAD データ形式で CAMEL-CXF エンドポイントのメッセージを処理する方法 269
POJO モードで SOAP ヘッダーを取得および設定する方法 269
PAYLOAD モードで SOAP ヘッダーを取得および設定する方法 271
SOAP ヘッダーが MESSAGE モードで利用できない 271
APACHE CAMEL から SOAP FAULT を出力する方法 271
CXF エンドポイントの要求および応答コンテキストを伝播する方法 272
添付ファイルのサポート 273
スタックトレース情報を伝播させる方法 276
PAYLOAD モードのストリーミングサポート 277
汎用 CXF ディスパッチモードの使用 277

第30章 CXF BEAN コンポーネント 278
CXF BEAN コンポーネント(2.0 以降) 278
URI 形式 278
オプション 278
HEADERS 279
作業例 281

第31章 CXFRS 282
CXFRS コンポーネント 282
URI 形式 282
オプション 282
APACHE CAMEL で REST エンドポイントを設定する方法 290
メッセージヘッダーから CXF プロデューサーアドレスを上書きする方法 291
REST リクエストの使用 - シンプルバインディングスタイル 292
シンプルバインディングスタイルの有効化 292
異なるメソッド署名を使用した要求バインディングの例 292
シンプルバインディングスタイルの例 293
REST リクエストの使用 - デフォルトバインディングスタイル 294
HOW TO INVOKE THE REST SERVICE THROUGH CAMEL-CXFRS PRODUCER? 296

第32章 DATAFORMAT コンポーネント 298
データフォーマットのコンポーネント 298
URI 形式 298
サンプル 298

第33章 DATASET 299
データセットコンポーネント 299
URI 形式 299
オプション 299
DATASET の設定 300
例 301
DATASETSUPPORT のプロパティー 301
SIMPLEDATASET 301
LISTDATASET 302
FILEDATASET 302

第34章 DIRECT 303

DIRECT コンポーネント 303
URI 形式 303
オプション 303
サンプル 303

第35章 DIRECT-VM 305
DIRECT VM コンポーネント 305
URI 形式 305
オプション 305
サンプル 306

第36章 DISRUPTOR 307
DISRUPTOR コンポーネント 307
URI 形式 308
オプション 308
待機ストラテジー 310
リクエスト応答の使用 311
同時コンシューマー 311
スレッドプール 311
例 312
MULTIPLECONSUMERS の使用 312
中断情報の抽出 313

第37章 DNS 314
DNS 314
URI 形式 314
オプション 314
HEADERS 314
例 315
IP ルックアップ 315
DNS ルックアップ 315
DNS DIG 315

第38章 DOCKER 317
DOCKER コンポーネント 317
URI 形式 317
ヘッダーストラテジー 317
一般的なオプション 317
コンシューマー操作 318
プロデューサー操作 318
例 321

第39章 DOZER 322
DOZER コンポーネント 322
CAMEL ON EAP デプロイメント 322
URI 形式 322
オプション 322
DOZER でのデータフォーマットの使用 323
DOZER の設定 324
マッピング拡張機能 324
変数マッピング 324
カスタムマッピング 325
式マッピング 326
DOZER 型変換 326

第40章 DROPBOX 327
- CAMEL DROPBOX コンポーネント 327
- URI 形式 327
- 操作 327
- オプション 328
- DEL 操作 328
- サンプル 328
- 結果メッセージヘッダー 328
- 結果メッセージのボディー 328
- GET (DOWNLOAD)操作 329
- サンプル 329
- 結果メッセージヘッダー 329
- 結果メッセージのボディー 329
- 移動操作 330
- サンプル 330
- 結果メッセージヘッダー 330
- 結果メッセージのボディー 330
- PUT (UPLOAD)操作 331
- サンプル 331
- 結果メッセージヘッダー 331
- 結果メッセージのボディー 332
- 検索操作 332
- サンプル 332
- 結果メッセージヘッダー 332
- 結果メッセージのボディー 333

第41章 EJB 334
- EJB COMPONENT 334
- CAMEL ON EAP デプロイメント 334

第42章 ETCD 335
- ETCD コンポーネント 335
- URI 形式 335
- オプション 335
- HEADERS 336
- キー NAMESPACE の例： 336
- 統計 NAMESPACE の例： 337
- NAMESPACE の例を監視します。 337

第43章 ELASTICSEARCH 338
- ELASTICSEARCH コンポーネント 338
- CAMEL ON EAP デプロイメント 338
- URI 形式 338
- エンドポイントオプション 338
- メッセージ操作 339
- インデックスの例 340
- JAVA の例 340
- 詳細は、これらのリソース を参照してください。 341

第44章 ELSQL 342
- ELSQL コンポーネント 342
- オプション 342
- クエリーの結果 347
- ヘッダーの値 347

例 347
その他の参考資料 348

第45章 EVENTADMIN 349
EVENTADMIN コンポーネント 349
DEPENDENCIES 349
URI 形式 349
URI オプション 349
メッセージヘッダー 349
メッセージボディー 349
使用例 349

第46章 EXEC 351
EXEC コンポーネント 351
DEPENDENCIES 351
URI 形式 351
URI オプション 351
メッセージヘッダー 352
メッセージボディー 354
単語数(LINUX)の実行 354
JAVA の実行 354
ANT スクリプトの実行 355
ECHO の実行(WINDOWS) 355

第47章 ファブリックコンポーネント 356
DEPENDENCIES 356
URI 形式 356
URI オプション 357
ファブリックエンドポイントのユースケース 357
ロケーションの検出 357
LOAD-BALANCING クラスター 358
自動再接続機能 358
エンドポイント URI の公開 359
エンドポイント URI の検索 360
負荷分散の例 360
OSGI バンドルプラグインの設定 362

第48章 FACEBOOK 363
FACEBOOK コンポーネント 363
URI 形式 363
FACEBOOKCOMPONENT 363
プロデューサーエンドポイント： 365
コンシューマーエンドポイント： 374
オプションの読み取り 377
メッセージヘッダー 378
メッセージボディー 378
ユースケース 378

第49章 FILE2 379
FILE COMPONENT: APACHE CAMEL 2.0 以降 379
URI 形式 379
CAMEL ON EAP デプロイメント 379
URI オプション 380
コンシューマーのみ 382

ファイルコンシューマーのデフォルト動作 394
プロデューサーのみ 394
ファイルプロデューサーのデフォルト動作 398
移動および削除操作 398
移動および PREMOVE オプションの詳細な制御 399
MOVEFAILED について 400
メッセージヘッダー 400
ファイルプロデューサーのみ 400
ファイルコンシューマーのみ 400
バッチコンシューマー 401
エクスチェンジプロパティー、ファイルコンシューマーのみ 401
フォルダーおよびファイル名を使用した一般的な GOTCHAS 401
ファイル名式 402
他のファイルを直接ドロップするフォルダーからのファイルの使用 402
完了ファイルの使用 402
行われたファイルの作成 403
ディレクトリーから読み取り、別のディレクトリーに書き込みます。 404
オーバールール動的名を使用したディレクトリーからの読み取り、別のディレクトリーへの書き込み 404
ディレクトリーから再帰的に読み取り、別のディレクトリーへの書き込み 404
フラット化の使用 404
ディレクトリーおよびデフォルトの移動操作からの読み取り 404
ディレクトリーから読み取り、JAVA でメッセージを処理します。 405
ディレクトリーからファイルを読み込み、内容を JMS キューに送信します。 405
ファイルへの書き込み 405
EXCHANGE.FILE_NAME を使用してサブディレクトリーに書き込みます。 406
一時ディレクトリーを使用した最終的な宛先へのファイルの書き込み 407
ファイル名の式の使用 407
同じファイルを複数回読み取るのを回避（べき等コンシューマー） 407
ファイルベースのべき等リポジトリーの使用 408
JPA ベースのべき等リポジトリーの使用 409
ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER を使用するフィルター 410
ANT パスマッチャーを使用したフィルターリング 410
COMPARATOR を使用したソート 411
SORTBY を使用したソート 412
GENERICFILEPROCESSSTRATEGY の使用 413
デバッグロギング 414

第50章 FLATPACK 415
FLATPACK コンポーネント 415
URI 形式 415
URI オプション 415
例 416
メッセージヘッダー 416
メッセージボディー 416
ヘッダーおよびトレイルレコード 417
エンドポイントの使用 418

第51章 FOP 419
FOP コンポーネント 419
URI 形式 419
出力形式 419
エンドポイントオプション 420
メッセージ操作 420

例 421

第52章 FREEMARKER 423
FREEMARKER 423
URI 形式 423
オプション 423
FREEMARKER コンテキスト 424
ホットリロード 425
動的テンプレート 425
サンプル 425
電子メールのサンプル 427

第53章 FTP2 429
FTP/SFTP コンポーネント 429
CAMEL ON EAP デプロイメント 429
URI 形式 429
URI オプション 430
その他の URI オプション 439
例 440
ファイルの使用時のデフォルト 440
LIMITATIONS 440
メッセージヘッダー 441
タイムアウトについて 442
ローカルワークディレクトリーの使用 442
ディレクトリーのステップ的な変更 442
STEPWISE=TRUE （デフォルトモード）の使用 443
STEPWISE=FALSE の使用 445
サンプル 446
ルートによってトリガーされるリモート FTP サーバーの使用 447
リモート FTPS サーバー（暗黙的な SSL）およびクライアント認証の使用 447
リモート FTPS サーバー(EXPLICIT TLS)およびカスタムトラストストア設定の使用 447
ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER を使用するフィルター 447
ANT パスマッチャーを使用したフィルターリング 448
SFTP でのプロキシーの使用 449
推奨される SFTP 認証方法の設定 449
固定名を使用した単一ファイルの使用 450
デバッグロギング 450

第54章 GAE 451
54.1. GAE コンポーネントの概要 451
Google App Engine の Apache Camel コンポーネント 451
Camel コンテキスト 452
Apache Camel 2.1 452
Apache Camel 2.2 453
web.xml 454
54.2. GAUTH 455
gauth コンポーネント 455
URI 形式 456
オプション 456
メッセージヘッダー 459
メッセージボディー 459
コンポーネントの設定 459
使用方法 460
GAE の例 462

アクセストークンの使用 463
54.3. GHTTP 464
ghttp コンポーネント 464
URI 形式 465
オプション 465
メッセージヘッダー 466
メッセージボディー 467
メッセージの受信 467
メッセージの送信 468
Dependencies 469
54.4. GLOGIN 469
glogin コンポーネント 469
URI 形式 470
オプション 470
メッセージヘッダー 471
メッセージボディー 471
使用方法 471
54.5. GMAIL 473
Gmail コンポーネント 473
URI 形式 473
オプション 473
メッセージヘッダー 474
メッセージボディー 474
使用方法 474
Dependencies 475
54.6. GSEC 475
Apache Camel GAE アプリケーションのセキュリティー 475
54.7. GTASK 477
gtask コンポーネント 477
URI 形式 477
オプション 477
メッセージヘッダー 478
メッセージボディー 479
使用方法 479
デフォルトのキュー 479
Dependencies 480

第55章 GANGLIA **481**
GANGLIA コンポーネント 481
URI 形式 481
GANGLIA エンドポイントオプション 481
メッセージボディー 483
戻り値/レスポンス 483
文字列メトリクスの送信 484
数値メトリックの送信 484

第56章 GEOCODER **485**
GEOCODER コンポーネント 485
URI 形式 485
オプション 485
PROXY 486
エクスチェンジデータ形式 487
メッセージヘッダー 487

サンプル 488

第57章 GIT 489
GIT コンポーネント 489
URI オプション 489
メッセージヘッダー 490
プロデューサーの例 491
コンシューマーの例 491

第58章 GITHUB 492
GITHUB コンポーネント 492
URI 形式 492
必須オプション： 493
コンシューマーエンドポイント： 493
プロデューサーエンドポイント： 493
URI オプション 494

第59章 GOOGLECALENDAR 495
GOOGLECALENDAR コンポーネント 495
コンポーネントの説明 495
URI 形式 495
GOOGLECALENDARCOMPONENT 496
プロデューサーエンドポイント 497
1.エンドポイント接頭辞 ACL 497
ACLの URI オプション 498
2.エンドポイント接頭辞 カレンダー 498
カレンダーの URI オプション 499
3.エンドポイント接頭辞 チャネル 499
チャネルの URI オプション 499
4.エンドポイント接頭辞の 色 499
色の URI オプション 500
5.エンドポイント接頭辞 イベント 500
イベントの URI オプション 501
6.エンドポイント接頭辞 FREEBUSY 501
FREEBUSYの URI オプション 501
7.ENDPOINT PREFIX LIST 501
LISTの URI オプション 502
8.エンドポイント接頭辞の 設定 502
設定の URI オプション 503
コンシューマーエンドポイント 503
メッセージヘッダー 503
メッセージボディー 503

第60章 GOOGLEDRIVE 504
GOOGLEDRIVE COMPONENT 504
URI 形式 504
GOOGLEDRIVECOMPONENT 505
プロデューサーエンドポイント 506
1.エンドポイント接頭辞 DRIVE-ABOUT 507
DRIVE-ABOUT の URI オプション 507
2.エンドポイント接頭辞 DRIVE-APPS 507
DRIVE-APPS の URI オプション 507
3.エンドポイント接頭辞 DRIVE-CHANGES 508
DRIVE-CHANGES の URI オプション 508

4.エンドポイント接頭辞ドライブチャネル 508
DRIVE-CHANNEL の URI オプション 509
5.エンドポイント接頭辞 DRIVE-CHILDREN 509
DRIVE-CHILDREN の URI オプション 509
6.エンドポイント接頭辞ドライブ 510
DRIVE-COMMENT の URI オプション 510
7.エンドポイント接頭辞ドライブファイル 510
DRIVE-FILES の URI オプション 511
8.エンドポイント接頭辞 DRIVE-PARENTS 512
DRIVE-PARENTS の URI オプション 512
9.エンドポイント接頭辞 DRIVE-PERMISSIONS 512
DRIVE-PERMISSIONS の URI オプション 513
10.エンドポイント接頭辞 DRIVE-PROPERTIES 513
DRIVE-PROPERTIES の URI オプション 514
11.エンドポイント接頭辞 DRIVE-REALTIME 514
DRIVE-REALTIME の URI オプション 515
12.エンドポイント接頭辞ドライブ 515
DRIVE-REPLIES の URI オプション 516
13.エンドポイント接頭辞 DRIVE-REVISIONS 516
DRIVE-REVISIONS の URI オプション 516
コンシューマーエンドポイント 517
メッセージヘッダー 517
メッセージボディー 517

第61章 GOOGLEMAIL 518
GOOGLEMAIL コンポーネント 518
コンポーネントの説明 518
URI 形式 518
GOOGLEMAILCOMPONENT 519
プロデューサーエンドポイント 519
1.エンドポイント接頭辞 アタッチメント 520
添付の URI オプション 520
2.エンドポイント接頭 辞ドラフト 521
ドラフトの URI オプション 521
3.エンドポイント接頭辞の 履歴 521
履歴の URI オプション 522
4.エンドポイント接頭辞の ラベル 522
ラベルの URI オプション 523
5.エンドポイント接頭辞 メッセージ 523
メッセージの URI オプション 524
6.エンドポイント接頭辞 スレッド 524
スレッドの URI オプション 525
7.エンドポイント接頭辞 ユーザー 525
ユーザーの URI オプション 525
コンシューマーエンドポイント 525
メッセージヘッダー 526
メッセージボディー 526

第62章 GUAVA EVENTBUS 527
GUAVA EVENTBUS コンポーネント 527
URI 形式 527
オプション 527
使用方法 528

DEADEVENT に関する考慮事項 529
複数のタイプのイベントの使用 530

第63章 HAWTDB . . . 531
HAWTDB 531
HAWTDBAGGREGATIONREPOSITORY の使用 531
永続化時に保持される内容 533
復元 533
JAVA DSL での HAWTDBAGGREGATIONREPOSITORY の使用 534
SPRING XML での HAWTDBAGGREGATIONREPOSITORY の使用 535
DEPENDENCIES 535

第64章 HAZELCAST コンポーネント . . . 537
HAZELCAST コンポーネント 537
URI 形式 537
オプション 537
セクション 538
マップの使用 538
MAP CACHE PRODUCER - TO("HAZELCAST:MAP:FOO") 539
PUT の例： 540
GET の例： 540
更新のサンプル： 541
削除のサンプル： 541
クエリーのサンプル 542
MAP CACHE CONSUMER - FROM("HAZELCAST:MAP:FOO") 542
マルチマップの使用 544
MULTIMAP CACHE PRODUCER - TO("HAZELCAST:MULTIMAP:FOO") 544
PUT の例： 545
GET の例： 546
更新のサンプル： 546
削除のサンプル： 547
クエリーのサンプル 547
MAP CACHE CONSUMER - FROM("HAZELCAST:MAP:FOO") 548
マルチマップの使用 550
MULTIMAP CACHE PRODUCER - TO("HAZELCAST:MULTIMAP:FOO") 550
PUT の例： 551
REMOVEVALUE の例： 551
GET の例： 552
削除のサンプル： 552
MULTIMAP CACHE CONSUMER - FROM("HAZELCAST:MULTIMAP:FOO") 553
キューの使用 555
キュープロデューサー TO ("HAZELCAST:QUEUE:FOO") 555
追加する例： 555
PUT の例： 555
ポーリングのサンプル： 555
PEEK の例： 555
オファーのサンプル： 556
REMOVEVALUE の例： 556
キューコンシューマー FROM ("HAZELCAST:QUEUE:FOO") 556
トピックの使用 556
トピックプロデューサー - TO ("HAZELCAST:TOPIC:FOO") 556
公開のサンプル 557
TOPIC CONSUMER - FROM ("HAZELCAST:TOPIC:FOO") 557

LIST の使用 557
プロデューサー TO ("HAZELCAST:LIST:FOO")を一覧表示します。 557
追加する例： 557
GET の例： 558
SETVALUE の例： 558
REMOVEVALUE の例： 558
LIST CONSUMER FROM ("HAZELCAST:LIST:FOO") 558
SEDA の使用 559
SEDA PRODUCER TO("HAZELCAST:SEDA:FOO") 559
SEDA CONSUMER FROM("HAZELCAST:SEDA:FOO") 559
ATOMIC NUMBER の使用 560
ATOMIC NUMBER PRODUCER - TO ("HAZELCAST:ATOMICNUMBER:FOO") 561
セットのサンプル： 561
GET の例： 562
INCREMENT のサンプル： 563
デクリメントのサンプル： 563
破棄のサンプル 564
クラスターのサポート 564
INSTANCE CONSUMER - FROM("HAZELCAST:INSTANCE:FOO") 565
HAZELCAST リファレンスの使用 566
名前 566
インスタンス別 567
HAZELCAST インスタンスを OSGI サービスとして公開 568
バンドル A はインスタンスを作成し、OSGI サービスとして公開します。 568
バンドル B は インスタンスを使用します。 568

第65章 HBASE 569
HBASE コンポーネント 569
APACHE HBASE の概要 569
CAMEL および HBASE 569
コンポーネントの設定 570
HBASE プロデューサー 570
プロデューサーでサポートされる URI オプション 571
操作を配置します。 572
操作を取得します。 574
操作を削除します。 574
スキャン操作。 574
HBASE コンシューマー 576
コンシューマーでサポートされている URI オプション 576
HBASE IDEMPOTENT リポジトリー 578
HBASE マッピング 578
HBASE ヘッダーマッピングの例 579
ボディーマッピングの例 580
その他の参考資料 580

第66章 HDFS 582
HDFS コンポーネント 582
URI 形式 582
オプション 583
KEYTYPE および VALUETYPE 584
分割ストラテジー 585
メッセージヘッダー 586
ファイルストリームを閉じるための制御 586

OSGI でのこのコンポーネントの使用 587

第67章 HDFS2 588
HDFS2 コンポーネント 588
URI 形式 588
オプション 589
KEYTYPE および VALUETYPE 590
分割ストラテジー 591
メッセージヘッダー 592
プロデューサーのみ 592
ファイルストリームを閉じるための制御 592
OSGI でのこのコンポーネントの使用 593
手動で定義されたルートでのこのコンポーネントの使用 593
BLUEPRINT コンテナーでのこのコンポーネントの使用 594

第68章 HIPCHAT 595
HIPCHAT コンポーネント 595
URI 形式 595
URI オプション 595
スケジュールされたポーリングコンシューマー 596
HIPCHAT コンシューマーによって設定されたメッセージヘッダー 596
HIPCHAT プロデューサー 597
HIPCHAT プロデューサーによって評価されるメッセージヘッダー 597
HIPCHAT プロデューサーによって設定されたメッセージヘッダー 598
DEPENDENCIES 598

第69章 HL7 600
HL7 コンポーネント 600
CAMEL ON EAP デプロイメント 600
HL7 MLLP プロトコル 601
MINA を使用した HL7 リスナーの公開 601
NETTY を使用した HL7 リスナーの公開(CAMEL 2.15 以降から利用可能) 603
JAVA.LANG.STRING または BYTE[] を使用した HL7 モデル 603
HAPI を使用した HL7V2 モデル 603
HL7 DATAFORMAT 604
メッセージヘッダー 605
オプション 606
DEPENDENCIES 607
TERSER 言語 608
HL7 検証述語 609
HAPICONTEXT (CAMEL 2.14)を使用した HL7 検証述語 609
HL7 確認式 610
追加のサンプル 610

第70章 HTTP 613
HTTP コンポーネント 613
URI 形式 613
例 613
HTTPENDPOINT オプション 614
認証およびプロキシー 617
HTTPCOMPONENT オプション 618
メッセージヘッダー 619
メッセージボディー 620
レスポンスコード 620

HTTPOPERATIONFAILEDEXCEPTION 621
GET または POST を使用した呼び出し 621
HTTPSERVLETREQUEST および HTTPSERVLETRESPONSE にアクセスする方法 622
クライアントタイムアウトの使用 - SO_TIMEOUT 622
プロキシーの設定 622
URI の外部でプロキシー設定の使用 622
CHARSET の設定 622
スケジュールされたポーリングを使用したサンプル 623
レスポンスコードの取得 623
THROWEXCEPTIONONFAILURE=FALSE を使用して応答を返す 623
COOKIE の無効化 624
高度な使用方法 624
MAXCONNECTIONSPERHOST の設定 624
プリエンプション認証の使用 625
リモートサーバーからの自己署名証明書の許可 625
JSSE 設定ユーティリティーの使用 625
APACHE HTTP クライアントを直接設定 626

第71章 HTTP4 628
HTTP4 コンポーネント 628
CAMEL ON EAP デプロイメント 628
URI 形式 628
HTTPCOMPONENT オプション 629
HTTPENDPOINT オプション 630
基本認証およびプロキシーの設定 635
メッセージヘッダー 636
メッセージボディー 637
レスポンスコード 637
HTTPOPERATIONFAILEDEXCEPTION 638
GET または POST を使用した呼び出し 638
HTTPSERVLETREQUEST および HTTPSERVLETRESPONSE にアクセスする方法 639
呼び出す URI の設定 639
URI パラメーターの設定 639
HTTP メソッド(GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE)を HTTP プロデューサーに設定する方法 640
クライアントタイムアウトの使用 - SO_TIMEOUT 641
プロキシーの設定 641
URI の外部でプロキシー設定の使用 641
CHARSET の設定 642
スケジュールされたポーリングを使用したサンプル 642
エンドポイント URI からの URI パラメーター 642
メッセージの URI パラメーター 642
レスポンスコードの取得 642
COOKIE の無効化 643
高度な使用方法 643
JSSE 設定ユーティリティーの使用 643
エンドポイントの SPRING DSL ベースの設定 643
APACHE HTTP クライアントを直接設定 643
HTTPS を使用した GOTCHAS の認証 644
異なる SSLCONTEXTPARAMETERS の使用 645

第72章 IBATIS 646
IBATIS 646

URI 形式 646
オプション 647
メッセージヘッダー 648
メッセージボディー 649
サンプル 649
STATEMENTTYPE を使用した IBATIS の制御の強化 650
スケジュールされたポーリングの例 650
ONCONSUME の使用 651

第73章 IRC **652**
IRC コンポーネント 652
URI 形式 652
オプション 652
JSSE 設定ユーティリティーの使用 654
エンドポイントのプログラムによる設定 654
エンドポイントの SPRING DSL ベースの設定 655
レガシーの基本設定オプションの使用 655
鍵の使用 656

第74章 JASYPT **657**
JASYPT コンポーネント 657
CAMEL ON EAP デプロイメント 657
ツール 657
CAMEL 2.5 および 2.6 のツールの依存関係 658
CAMEL 2.7 以降のツールの依存関係 659
URI オプション 659
マスターパスワードの保護 659
JAVA DSL を使用した例 660
SPRING XML の例 661
関連項目 661

第75章 JAXB **663**
JAXB コンポーネント 663
CAMEL ON EAP デプロイメント 663

第76章 JCACHE **664**
JCACHE コンポーネント 664
URI 形式 664
オプション 664
ヘッダー変数 666
JCACHE ベースのべき等リポジトリーの例： 667
JCACHE ベースの集約リポジトリーの例： 667

第77章 JCLOUDS **670**
JCLOUDS コンポーネント 670
コンポーネントの設定 670
URI 形式 671
BLOBSTORE URI オプション 672
BLOBSTORE のメッセージヘッダー 672
BLOBSTORE 使用例 673
例 1: BLOB に配置 673
例 2: BLOB の取得/読み取り 673
例 3: BLOB の使用 674
COMPUTE サービスの URI オプション 674

コンピュートの使用例 676
例 1: 利用可能なイメージの一覧表示 676
例 2: 新規ノードを作成します。 676
例 3: 実行中のノードでシェルスクリプトを実行します。 676
その他の参考資料 677

第78章 JCR ... 678
JCR コンポーネント 678
URI 形式 678
使用方法 678
プロデューサー 678
コンシューマー 679
例 680

第79章 JDBC ... 682
JDBC コンポーネント 682
URI 形式 682
オプション 682
結果 685
メッセージヘッダー 685
生成されるキー 686
名前付きパラメーターの使用 686
サンプル 687
サンプル：データベースを毎分ポーリングする 688
例：データソース間のデータの移動 688
その他の参考資料 689

第80章 JETTY ... 690
JETTY コンポーネント 690
URI 形式 690
オプション 691
メッセージヘッダー 697
使用方法 697
コンポーネントオプション 697
プロデューサーの例 700
コンシューマーの例 700
セッションサポート 702
JSSE 設定ユーティリティーの使用 702
エンドポイントの SPRING DSL ベースの設定 703
JETTY の直接設定 703
一般的な SSL プロパティーの設定 705
X509CERTIFICATE への参照を取得する方法 706
一般的な HTTP プロパティーの設定 706
OBTAINING X-FORWARDED-FOR HEADER WITH HTTPSERVLETREQUEST.GETREMOTEADDR() 706
HTTP ステータスコードを返すデフォルトの動作 707
CUSTOMIZING HTTPBINDING 707
JETTY ハンドラーおよびセキュリティー設定 708
カスタム HTTP 500 リプライメッセージを返す方法 710
マルチパートフォームのサポート 710
JETTY JMX サポート 711

第81章 JGROUPS ... 713
JGROUPS コンポーネント 713
CAMEL ON EAP デプロイメント 713

URI 形式 713
オプション 713
使用方法 714

第82章 JING 715
JING コンポーネント 715
URI 形式 715
オプション 715
例 716

第83章 JIRA 717
JIRA コンポーネント 717
URI 形式 717
必須オプション： 718
コンシューマーエンドポイント： 718
プロデューサーエンドポイント： 718
URI オプション： 718
JQL: 719

第84章 JMS 720
JMS COMPONENT 720
CAMEL ON EAP デプロイメント 720
URI 形式 721
ACTIVEMQ の使用 721
トランザクションおよびキャッシュレベル 722
永続サブスクリプション 722
メッセージヘッダーのマッピング 722
オプション 723
最も一般的に使用されるオプション 723
その他のすべてのオプション 728
サンプル 741
JMS からの受信 741
JMS に送信する 742
アノテーションの使用 742
SPRING DSL の例 742
その他のサンプル 742
JMS と CAMEL 間のメッセージマッピング 742
JMS メッセージの自動マッピングの無効化 743
カスタム MESSAGECONVERTER の使用 744
選択したマッピングストラテジーの制御 744
送信時のメッセージ形式 744
受信時のメッセージ形式 745
CAMEL を使用したメッセージおよび JMSREPLYTO の送受信 746
JMSPRODUCER 747
JMSCONSUMER 748
エンドポイントを再利用し、実行時に計算された異なる宛先に送信する 749
異なる JMS プロバイダーの設定 750
JNDI を使用した CONNECTIONFACTORY の検索 750
同時消費 751
非同期コンシューマーを使用した同時消費 751
JMS でのリクエスト応答 751
JMS 上でリクエスト応答し、共有固定応答キューを使用する 756
JMS のリクエスト応答と排他的な固定応答キューの使用 757
送信側と受信側のクロックの同期 757

存続期間 758
TRANSACTED CONSUMPTION の有効化 759
応答の遅れに JMSREPLYTO を使用する 760
リクエストタイムアウトの使用 761
JMS をエクスチェンジを保存する DEAD LETTER QUEUE として使用 761
JMS を DEAD LETTER CHANNEL 保存エラーとしてのみ使用 762
INONLY メッセージの送信および JMSREPLYTO ヘッダーの維持 762
宛先での JMS プロバイダーオプションの設定 763
関連項目 763

第85章 JMX **765**
JMX コンポーネント 765
CAMEL ON EAP デプロイメント 765
URI 形式 765
URI オプション 766
OBJECTNAME コンストラクト 767
NAME プロパティーの DOMAIN 767
HASHTABLE のあるドメイン 768
例 768
完全な例 768
MONITOR TYPE CONSUMER 768
例 769
モニタータイプの URI オプション 769

第86章 JOLT **771**
JOLT コンポーネント 771
URI 形式 771
オプション 771
動的仕様 772
サンプル 773

第87章 JPA **774**
JPA コンポーネント 774
CAMEL ON EAP デプロイメント 774
エンドポイントへの送信 774
エンドポイントからの消費 774
URI 形式 775
オプション 775
メッセージヘッダー 779
ENTITYMANAGERFACTORY の設定 779
TRANSACTIONMANAGER の設定 780
名前付きクエリーでのコンシューマーの使用 780
クエリーでのコンシューマーの使用 780
ネイティブクエリーでのコンシューマーの使用 781
例 781
JPA ベースのべき等リポジトリーの使用 781

第88章 JSCH **783**
JSCH 783
URI 形式 783
オプション 783
コンポーネントのオプション 784
制限事項 784

第89章 JT400 786
JT/400 コンポーネント 786
URI 形式 786
URI オプション 786
使用方法 788
接続プール 788
リモートプログラム呼び出し(CAMEL 2.7) 788
例 789
リモートプログラム呼び出しの例(CAMEL 2.7) 789
キーデータキューへの書き込み 789
キーデータキューからの読み取り 790

第90章 KAFKA 791
KAFKA コンポーネント 791
URI 形式 791
オプション 791
プロデューサーオプション 792
コンシューマーオプション 794
サンプル 794
エンドポイント 795
関連項目 795

第91章 KESTREL 796
KESTREL コンポーネント 796
URI 形式 796
オプション 797
SPRING XML を使用した KESTREL コンポーネントの設定 797
使用例 798
例 1: 消費 799
例 2: 生成 799
例 3: SPRING XML 設定 799
DEPENDENCIES 800
SPYMEMCACHED 800

第92章 KRATI 802
KRATI コンポーネント 802
URI 形式 802
KRATI URI オプション 802
データストアのメッセージヘッダー 803
使用例 804
例 1: データストアへの配置。 804
例 2: データストアの取得/読み取り 804
例 3: データストアの使用 805
IDEMPOTENT リポジトリー 805
その他の参考資料 805

第93章 KUBERNETES 807
KUBERNETES コンポーネント 807
URI 形式 807
オプション 807
HEADERS 809

第94章 KURA 812
KURA コンポーネント 812

KURAROUTER ACTIVATOR 812
KURAROUTER のデプロイ 813
KURAROUTER ユーティリティー 814
SLF4J ロガー 814
BUNDLECONTEXT 814
CAMELCONTEXT 814
PRODUCERTEMPLATE 815
CONSUMERTEMPLATE 815
OSGI サービスリゾルバー 815
KURAROUTER アクティベーターコールバック 816
CONFIGURATIONADMIN からの XML ルートの読み込み 816
宣言型 OSGI サービスとして KURA ルーターをデプロイする 817

第95章 言語 818
言語 818
URI 形式 818
URI オプション 818
メッセージヘッダー 819
例 820
リソースからのスクリプトの読み込み 820

第96章 LDAP 822
LDAP コンポーネント 822
URI 形式 822
オプション 822
結果 823
DIRCONTEXT 823
サンプル 823
認証情報を使用したバインディング 824
SSL の設定 825

第97章 LEVELDB 828
LEVELDB 828
LEVELDBAGGREGATIONREPOSITORY の使用 828
永続化時に保持される内容 830
復元 830
JAVA DSL での LEVELDBAGGREGATIONREPOSITORY の使用 831
SPRING XML での LEVELDBAGGREGATIONREPOSITORY の使用 831
DEPENDENCIES 832

第98章 LINKEDIN 834
LINKEDIN コンポーネント 834
URI 形式 834
LINKEDINCOMPONENT 835
プロデューサーエンドポイント： 836
エンドポイント接頭辞のコメント 837
コメントの URI オプション 837
エンドポイント接頭辞企業 837
企業の URI オプション 839
エンドポイント接頭辞グループ 840
グループの URI オプション 840
エンドポイント接頭辞ジョブ 840
ジョブの URI オプション 841
エンドポイント接頭辞 PEOPLE 841

ユーザーの URI オプション 844
エンドポイント接頭辞 POST 846
投稿の URI オプション 846
エンドポイント接頭辞検索 847
検索の URI オプション 848
コンシューマーエンドポイント 849
メッセージヘッダー 849
メッセージボディー 849
ユースケース 849

第99章 リスト 851
コンポーネントの一覧表示 851
URI 形式 851
例 851

第100章 LOG 853
ログコンポーネント 853
URI 形式 853
オプション 853
フォーマット 854
通常のロガーの例 856
フォーマッターサンプルを持つ通常のロガー 857
GROUPSIZE サンプルを使用したスループットロガー 857
GROUPINTERVAL サンプルを使用したスループットロガー 857
ロギング出力の完全なカスタマイズ 858
OSGI でのログコンポーネントの使用 860

第101章 LUCENE 861
LUCENE (INDEXER および SEARCH)コンポーネント 861
CAMEL ON EAP デプロイメント 861
URI 形式 861
オプションの挿入 862
クエリーオプション 862
メッセージヘッダー 863
LUCENE プロデューサー 863
LUCENE プロセッサー 863
例 1: LUCENE インデックスの作成 863
例 2: CAMEL CONTEXT の JNDI レジストリーへのプロパティーのロード 864
例 2: クエリープロデューサーを使用した検索の実行 864
例 3: クエリープロセッサーを使用した検索の実行 864

第102章 MAIL 866
メールコンポーネント 866
CAMEL ON EAP デプロイメント 867
URI 形式 867
サンプルエンドポイント 868
デフォルトのポート 868
オプション 868
SSL サポート 876
JSSE 設定ユーティリティーの使用 876
エンドポイントのプログラムによる設定 876
エンドポイントの SPRING DSL ベースの設定 876
JAVAMAIL の直接設定 877
メールメッセージの内容 877

ヘッダーが事前設定された受信者よりも優先されます。 877
複数の受信者で設定が容易になります。 878
送信者名と電子メールの設定 878
SUN JAVAMAIL 878
サンプル 879
添付ファイルを使用したメールの送信サンプル 879
SSL の例 880
添付サンプルでメールの消費 881
添付ファイルを使用したメールメッセージを分割する方法 882
カスタム SEARCHTERM の使用 883

第103章 マスターコンポーネント 887
DEPENDENCIES 887
URI 形式 887
URI オプション 887
マスターコンポーネントの使用方法 887
JMS ACTIVEMQ ブローカーをポーリングするマスター/スレーブクラスターの例 888
ACTIVEMQ ブローカーからメッセージをポーリングするクラスターを作成する手順 889
OSGI バンドルプラグインの設定 892

第104章 メトリクス 894
メトリクスコンポーネント 894
URI 形式 894
METRIC REGISTRY 894
使用方法 895
HEADERS 895
メトリクスタイプのカウンター 896
オプション 896
HEADERS 896
メトリックのタイプヒストグラム 897
オプション 897
HEADERS 897
メトリックタイプのメーター 898
オプション 898
HEADERS 898
メトリクスタイプタイマー 899
オプション 899
HEADERS 899
METRICSROUTEPOLICYFACTORY 900
METRICSMESSAGEHISTORYFACTORY 901

第105章 MINA2 - 非推奨 904
MINA 2 コンポーネント 904
CAMEL ON EAP デプロイメント 904
URI 形式 905
オプション 905
カスタムコーデックの使用 909
SYNC=FALSE を使用した例 909
SYNC=TRUE の例 910
SPRING DSL を使用した例 910
完了時にセッションを閉じる 911
メッセージの IOSESSION を取得します。 911
MINA フィルターの設定 911

第106章 MLLP 912
MLLP コンポーネント 912
MLLP コンシューマー 912
メッセージヘッダー 913
エクスチェンジプロパティー 913
コンシューマー設定：MLLP プロデューサー 914
メッセージヘッダー 914

第107章 MOCK 915
MOCK コンポーネント 915
URI 形式 916
オプション 916
簡単な例 917
ASSERTPERIOD の使用 917
期待値の設定 917
特定のメッセージへの期待の追加 918
既存エンドポイントのモック化 919
CAMEL-TEST コンポーネントを使用した既存エンドポイントのモック化 921
XML DSL を使用した既存エンドポイントのモック化 922
エンドポイントのモック化および元のエンドポイントへの送信を省略する 923
保持するメッセージ数の制限 925
到着時間でのテスト 926

第108章 MONGODB 927
CAMEL MONGODB コンポーネント 927
URI 形式 927
エンドポイントオプション 928
SPRING XML でのデータベースの設定 934
サンプルルート 934
MONGODB 操作 - プロデューサーエンドポイント 934
クエリー操作 934
FINDBYID 935
FINDONEBYQUERY 935
FINDALL 936
COUNT 938
フィールドフィルターの指定 938
作成/更新の操作 939
INSERT 939
SAVE 940
UPDATE 940
操作の削除 941
REMOVE 942
その他の操作 942
AGGREGATE 942
GETDBSTATS 942
GETCOLSTATS 943
COMMAND 944
動的操作 944
TAILABLE CURSOR コンシューマー 944
テール可能なカーソルコンシューマーの仕組み 945
永続的なテールトラッキング 946
永続的なテールトラッキングの有効化 946
型変換 947

その他の参考資料 948

第109章 MONGODB GRIDFS 950
CAMEL MONGODB GRIDFS コンポーネント 950
URI 形式 950
エンドポイントオプション 950
SPRING XML でのデータベースの設定 954
サンプルルート 954
MONGODB 操作 - プロデューサーエンドポイント 954
GRIDFS CONSUMER 956
その他の参考資料 957

第110章 MQTT 958
MQTT コンポーネント 958
CAMEL ON EAP デプロイメント 958
URI 形式 958
オプション 958
サンプル 960

第111章 MSV 962
MSV コンポーネント 962
URI 形式 962
オプション 962
例 962

第112章 MUSTACHE 964
MUSTACHE 964
URI 形式 964
オプション 964
MUSTACHE コンテキスト 965
動的テンプレート 966
サンプル 966
電子メールのサンプル 967

第113章 MVEL コンポーネント 969
MVEL コンポーネント 969
CAMEL ON EAP デプロイメント 969
URI 形式 969
オプション 969
メッセージヘッダー 970
MVEL コンテキスト 970
ホットリロード 971
動的テンプレート 971
サンプル 972

第114章 MYBATIS 974
MYBATIS 974
URI 形式 974
オプション 974
メッセージヘッダー 977
メッセージボディー 977
サンプル 977
STATEMENTTYPE を使用した MYBATIS の制御の改善 978
INSERTLIST STATEMENTTYPE の使用 979
UPDATELIST STATEMENTTYPE の使用 979

DELETELIST STATEMENTTYPE の使用 980
INSERTLIST、UPDATELIST、および DELETELIST STATEMENTTYPES に関する通知 980
スケジュールされたポーリングの例 981
ONCONSUME の使用 981
トランザクションへの参加 982

第115章 NAGIOS ... 984
NAGIOS 984
URI 形式 984
オプション 984
HEADERS 985
メッセージの送信例 985
NAGIOSEVENTNOTIFER の使用 986

第116章 NAT ... 987
NATS コンポーネント 987
URI 形式 987
オプション 987
HEADERS 988

第117章 NETTY ... 990
NETTY コンポーネント 990
URI 形式 991
オプション 991
レジストリーベースのオプション 998
共有不可能なエンコーダーまたはデコーダーの使用 999
NETTY エンドポイントとの間でメッセージを送信する 999
NETTY プロデューサー 1000
NETTY コンシューマー 1000
HEADERS 1000
REQUEST-REPLY およびシリアライズされたオブジェクトペイロードを使用した UDP NETTY エンドポイント 1001
一方向通信を使用した TCP ベースの NETTY コンシューマーエンドポイント 1001
REQUEST-REPLY 通信を使用する SSL/TCP ベースの NETTY コンシューマーエンドポイント 1001
JSSE 設定ユーティリティーの使用 1001
コンポーネントのプログラムによる設定 1002
エンドポイントの SPRING DSL ベースの設定 1002
JETTY コンポーネントでの基本的な SSL/TLS 設定の使用 1002
SSLSESSION およびクライアント証明書へのアクセス 1003
複数のコーデックの使用 1003
完了したらチャンネルを閉じる 1006
作成されたパイプラインを完全に制御するためのカスタムチャネルパイプラインファクトリーの追加 1006
NETTY BOSS およびワーカースレッドプールの再利用 1008
その他の参考資料 1009

第118章 NETTY4 ... 1010
NETTY4 コンポーネント 1010
CAMEL ON EAP デプロイメント 1010
URI 形式 1010
オプション 1011
レジストリーベースのオプション 1018
共有不可能なエンコーダーまたはデコーダーの使用 1019
NETTY エンドポイントとの間でメッセージを送信する 1019
NETTY プロデューサー 1019

NETTY コンシューマー 1020
使用例 1020
REQUEST-REPLY およびシリアライズされたオブジェクトペイロードを使用した UDP NETTY エンドポイント 1020
一方向通信を使用した TCP ベースの NETTY コンシューマーエンドポイント 1021
複数のコーデックの使用 1021
完了したらチャンネルを閉じる 1023
作成されたパイプラインを完全に制御するためのカスタムチャネルパイプラインファクトリーの追加 1023
NETTY BOSS およびワーカースレッドプールの再利用 1025

第119章 NETTY HTTP 1027
NETTY HTTP コンポーネント 1027
URI 形式 1027
HTTP オプション 1028
メッセージヘッダー 1033
NETTY タイプへのアクセス 1035
例 1035
NETTY がワイルドカードと一致させる方法 1036
同じポートでの複数ルートの使用 1036
複数のルートを持つ同じサーバーブートストラップ設定の再利用 1037
OSGI コンテナー内の複数のバンドル間で複数のルートを持つ同じサーバーブートストラップ設定の再利用 1038
HTTP BASIC 認証の使用 1038
WEB リソースでの ACL の指定 1039

第120章 NETTY4-HTTP 1041
NETTY4 HTTP コンポーネント 1041
URI 形式 1041
HTTP オプション 1042
メッセージヘッダー 1046
NETTY タイプへのアクセス 1048
例 1048
NETTY がワイルドカードと一致させる方法 1049
同じポートでの複数ルートの使用 1049
複数のルートを持つ同じサーバーブートストラップ設定の再利用 1050
OSGI コンテナー内の複数のバンドル間で複数のルートを持つ同じサーバーブートストラップ設定の再利用 1051
HTTP BASIC 認証の使用 1051
WEB リソースでの ACL の指定 1052

第121章 OLINGO2 1054
OLINGO2 コンポーネント 1054
URI 形式 1054
OLINGO2COMPONENT 1054
プロデューサーエンドポイント 1055
ODATA リソースタイプマッピング 1057
URI オプション 1058
コンシューマーエンドポイント 1059
メッセージヘッダー 1059
メッセージボディー 1059
ユースケース 1059

第122章 OPENSHIFT 1061
OPENSHIFT コンポーネント 1061
URI 形式 1061

オプション 1061
例 1064
全アプリケーションの一覧表示 1064
アプリケーションの停止 1064

第123章 PAHO 1066
PAHO コンポーネント 1066
URI 形式 1066
コンポーネントのプロジェクトへの追加 1066
デフォルトのペイロードタイプ 1067
URI オプション 1067
HEADERS 1068

第124章 PAX-LOGGING 1070
PAXLOGGING コンポーネント 1070
DEPENDENCIES 1070
URI 形式 1070
URI オプション 1070
メッセージヘッダー 1071
メッセージボディー 1071
使用例 1071

第125章 PDF 1073
PDF 1073
URI 形式 1073
操作 1073
HEADERS 1074

第126章 PGEVENT 1076
PGEVENT コンポーネント 1076
URI 形式 1076
オプション 1076

第127章 プリンター 1078
プリンターコンポーネント 1078
URI 形式 1078
オプション 1078
プリンタープロデューサー 1079
例 1: アルファベットおよび 1 サイドモードでデフォルトプリンターでテキストベースのペイロードを出力する 1080
例 2: A4 SITESARY および 1 サイドモードでリモートプリンターで GIF ベースのペイロードを出力する 1080
例 3: 日本語の投稿者および 1 サイドモードでリモートプリンターで JPEG ベースのペイロードを出力する 1080

第128章 プロパティー 1081
PROPERTIES コンポーネント 1081
URI 形式 1081
オプション 1081
その他の参考資料 1083

第129章 QUARTZ 1085
QUARTZ コンポーネント 1085
URI 形式 1085
オプション 1085
QUARTZ.PROPERTIES ファイルの設定 1087
JMX での QUARTZ スケジューラーの有効化 1087

QUARTZ スケジューラーの起動 1088
クラスタリング 1088
メッセージヘッダー 1089
CRON トリガーの使用 1089
タイムゾーンの指定 1090

第130章 QUARTZ2 1091
QUARTZ2 COMPONENT 1091
CAMEL ON EAP デプロイメント 1091
URI 形式 1091
オプション 1092
QUARTZ.PROPERTIES ファイルの設定 1094
JMX での QUARTZ スケジューラーの有効化 1095
QUARTZ スケジューラーの起動 1095
クラスタリング 1095
メッセージヘッダー 1096
CRON トリガーの使用 1096
タイムゾーンの指定 1096
QUARTZSCHEDULEDPOLLCONSUMERSCHEDULER の使用 1097

第131章 QUICKFIX 1099
QUICKFIX/J コンポーネント 1099
URI 形式 1099
エンドポイント 1100
エクスチェンジの形式 1100
QUICKFIX/J CONFIGURATION EXTENSIONS 1101
通信コネクター 1101
ロギング 1102
メッセージストア 1102
メッセージファクトリー 1102
JMX 1103
その他のデフォルト 1103
最小限のイニシエーター設定の例 1103
INOUT メッセージ交換パターンの使用 1103
コンシューマーの INOUT エクスチェンジの実装 1103
プロデューサーの INOUT エクスチェンジの実装 1104
例 1104
SPRING の設定 1105
例外処理 1107
修正シーケンス番号管理 1107
ルートの例 1108
QUICKFIX/J COMPONENT PRIOR TO CAMEL 2.5 1109
URI 形式 1109
エクスチェンジデータ形式 1109
LAZY CREATING ENGINES 1110
サンプル 1110

第132章 RABBITMQ 1111
RABBITMQ コンポーネント 1111
URI 形式 1111
オプション 1111
カスタム接続ファクトリー 1116
HEADERS 1116
メッセージボディー 1117

サンプル 1118

第133章 REF 1119
REF コンポーネント 1119
URI 形式 1119
ランタイムルックアップ 1119
例 1119

第134章 REST 1121
REST コンポーネント 1121
URI 形式 1121
URI オプション 1121
PATH および URITEMPLATE 構文 1121
その他の例 1122

第135章 RESTLET 1123
RESTLET コンポーネント 1123
URI 形式 1123
オプション 1124
コンポーネントオプション 1125
メッセージヘッダー 1127
メッセージボディー 1129
認証のある RESTLET エンドポイント 1129
単一の RESTLET エンドポイントから複数のメソッドおよび URI テンプレート(2.0 以降)を提供 1131
RESTLET API を使用した応答の設定 1132
コンポーネントの最大スレッドの設定 1133
WEBAPP 内の RESTLET サーブレットの使用 1133

第136章 RMI 1136
RMI コンポーネント 1136
URI 形式 1136
オプション 1136
使用 1137

第137章 ROUTEBOX 1138
ROUTEBOX コンポーネント 1138
CAMEL ROUTEBOX エンドポイントの要件 1139
URI 形式 1140
オプション 1140
ROUTEBOX へのメッセージの送受信 1142
ステップ 1: レジストリーへの内部ルート詳細の読み込み 1142
ステップ 2: ディスパッチマップの代わりにディスパッチストラテジーを使用するオプション 1143
ステップ 2: ROUTEBOX コンシューマーの起動 1143
ステップ 3: ROUTEBOX プロデューサーの使用 1144

第138章 RSS 1146
RSS コンポーネント 1146
CAMEL ON EAP デプロイメント 1146
URI 形式 1146
オプション 1146
データ型の交換 1148
メッセージヘッダー 1148
RSS DATAFORMAT 1149
エントリーのフィルターリング 1149
その他の参考資料 1150

第139章 SALESFORCE **1151**
SALESFORCE コンポーネント 1151
CAMEL ON EAP デプロイメント 1151
URI 形式 1151
サポートされる SALESFORCE API 1152
REST API 1152
REST BULK API 1153
REST STREAMING API 1154
CONTENTWORKSPACE へのドキュメントのアップロード 1155
CAMEL SALESFORCE MAVEN プラグイン 1155
使用方法 1156

第140章 SAP コンポーネント **1157**
140.1. 概要 1157
Dependencies 1157
SAP コンポーネントの追加プラットフォームの制限 1157
SAP JCo および SAP IDoc ライブラリー 1157
Fuse OSGi コンテナーへのデプロイ (Fabric 以外) 1158
Fuse Fabric でのデプロイ 1159
JBoss EAP コンテナーへのデプロイ 1161
URI 形式 1162
RFC 宛先エンドポイントのオプション 1164
RFC サーバーエンドポイントのオプション 1165
IDoc List Server エンドポイントのオプション 1165
RFC および IDoc エンドポイントの概要 1165
SAP RFC 宛先エンドポイント 1168
SAP RFC サーバーエンドポイント 1168
SAP IDoc および IDoc リスト宛先エンドポイント 1169
SAP IDoc list server endpoint 1169
メタデータリポジトリー 1169
140.2. 設定 1170
140.2.1. 設定の概要 1170
概要 1170
例 1170
140.2.2. 宛先設定 1171
概要 1171
宛先設定のサンプル 1172
tRFC および qRFC 宛先のインターセプター 1172
Logon および認証オプション 1173
接続オプション 1174
接続プールのオプション 1175
セキュアな接続オプション 1175
リポジトリーのオプション 1175
トレース設定オプション 1176
140.2.3. サーバー設定 1176
概要 1176
サーバーの設定例 1177
必須オプション 1178
セキュアな接続オプション 1178
その他のオプション 1179
140.2.4. リポジトリーの設定 1179
概要 1179
リポジトリーデータの例 1180

関数テンプレートのプロパティー 1180
関数テンプレートの例 1181
フィールドメタデータプロパティーを一覧表示します。 1182
elementary list field meta-data example 1183
複雑なリストフィールドのメタデータの例 1184
レコードのメタデータ属性 1184
レコードのメタデータの例 1184
レコードフィールドのメタデータプロパティー 1185
要素のレコードフィールドのメタデータの例 1186
複雑なレコードフィールドのメタデータの例 1186
140.3. RFC のメッセージボディー 1187
要求および応答オブジェクト 1187
構造オブジェクト 1187
フィールドタイプ 1188
要素のフィールドタイプ 1188
文字フィールドタイプ 1189
数値フィールドタイプ 1190
16 進数フィールドタイプ 1191
文字列フィールドタイプ 1191
複雑なフィールドタイプ 1192
構造化フィールドタイプ 1192
テーブルフィールドタイプ 1192
テーブルオブジェクト 1193
140.4. IDOC のメッセージボディー 1193
IDoc メッセージタイプ 1193
IDoc ドキュメントモデル 1194
IDoc が Document オブジェクトにどのように関連しているか 1196
ドキュメントインスタンスの作成例 1197
ドキュメント属性 1198
Java でのドキュメント属性の設定 1201
XML でのドキュメント属性の設定 1202
140.5. トランザクションサポート 1202
BAPI トランザクションモデル 1202
RFC トランザクションモデル 1202
使用するトランザクションモデル 1203
トランザクション RFC 宛先エンドポイント 1203
トランザクション RFC サーバーエンドポイント 1203
140.6. RFC の XML シリアライゼーション 1204
概要 1204
XML namespace 1204
リクエストおよび応答 XML ドキュメント 1204
構造フィールド 1205
テーブルフィールド 1205
要素フィールド 1206
日付と時刻の形式 1207
140.7. IDOC の XML シリアライゼーション 1207
概要 1207
XML namespace 1207
組み込み型コンバーター 1208
XML 形式の IDoc メッセージのボディーの例 1208
140.8. 例 1: SAP からのデータの読み取り 1209
概要 1209
ルートの Java DSL 1209

ルートの XML DSL 1209
createFlightCustomerGetListRequest bean 1210
returnFlightCustomerInfo bean 1210
140.9. 例 2: SAP へのデータの書き込み 1211
概要 1211
ルートの Java DSL 1211
ルートの XML DSL 1212
トランザクションサポート 1212
要求パラメーターの設定 1212
140.10. 例 3: SAP からのリクエストの処理 1212
概要 1212
ルートの Java DSL 1213
ルートの XML DSL 1213
BookFlightRequest bean 1213
BookFlightResponse Bean 1214
FlightInfo ビーン 1215
ConnectionInfoTable Bean 1216
ConnectionInfo bean 1216

第141章 SAP NETWEAVER 1218
SAP NETWEAVER GATEWAY コンポーネント 1218
URI 形式 1218
前提条件 1218
コンポーネントおよびエンドポイントオプション 1219
メッセージヘッダー 1219
例 1220

第142章 スケジューラー 1222
スケジューラーコンポーネント 1222
URI 形式 1222
オプション 1222
補足情報 1225
エクスチェンジプロパティー 1225
例 1225
完了するとスケジューラーが即座にトリガーされるように強制します。 1225
スケジューラーがアイドル状態になる 1226

第143章 SCHEMATRON 1227
SCHEMATRON コンポーネント 1227
URI 形式 1227
URI オプション 1227
HEADERS 1227
URI およびパス構文 1227
SCHEMATRON ルールおよびレポートサンプル 1228

第144章 SEDA 1230
SEDA コンポーネント 1230
URI 形式 1230
オプション 1230
BLOCKINGQUEUE 実装の選択 1233
リクエスト応答の使用 1234
同時コンシューマー 1234
スレッドプールと同時コンシューマーの違い 1235
スレッドプール 1235

例 1235
MULTIPLECONSUMERS の使用 1236
キュー情報の抽出。 1236

第145章 SERVICENOW 1238
SERVICENOW コンポーネント 1238
URI 形式 1238
オプション 1238
HEADERS 1239
使用例 1240

第146章 SERVLET 1242
SERVLET コンポーネント 1242
CAMEL ON EAP デプロイメント 1242
URI 形式 1242
オプション 1242
メッセージヘッダー 1243
使用方法 1243
CAMEL JAR のアプリケーションサーバーブートクラスパスへの配置 1243
例 1245
SPRING 3.X を使用する場合の例 1246
SPRING 2.X を使用する場合の例 1247
OSGI を使用する場合の例 1247

第147章 SERVLETLISTENER COMPONENT 1251
SERVLETLISTENER COMPONENT 1251
使用 1251
オプション 1253
例 1254
ルートの設定 1255
ROUTEBUILDER クラスの使用 1255
パッケージスキャンの使用 1255
XML ファイルの使用 1256
適切なプレースホルダーの設定 1257
JMX の設定 1257
JNDI または CAMEL レジストリーとしての SIMPLE 1257
カスタム CAMELCONTEXTLIFECYCLE の使用 1258

第148章 SHIRO セキュリティー 1260
SHIRO セキュリティーコンポーネント 1260
SHIRO セキュリティーの基本 1260
SHIROSECURITYPOLICY オブジェクトのインスタンス化 1261
SHIROSECURITYPOLICY OPTIONS 1261
CAMEL ルートでの SHIRO 認証の適用 1263
CAMEL ルートでの SHIRO 承認の適用 1263
SHIROSECURITYTOKEN を作成し、メッセージエクスチェンジに注入する 1264
SHIROSECURITYPOLICY でセキュリティーが保護されたルートへのメッセージ送信 1264
SHIROSECURITYPOLICY によってセキュリティーが保護されたルートへのメッセージ送信(CAMEL 2.12 以降により簡単) 1265
SHIROSECURITYTOKEN の使用 1265

第149章 SIP 1267
SIP コンポーネント 1267
URI 形式 1268

オプション 1268
レジストリーベースのオプション 1271
SIP エンドポイントとの間でメッセージを送信 1272
CAMEL SIP パブリッシャーの作成 1272
CAMEL SIP SUBSCRIBER の作成 1273

第150章 SJMS 1275
SJMS コンポーネント 1275
URI 形式 1276
コンポーネントのオプションおよび設定 1277
プロデューサー設定オプション 1279
プロデューサーの使用 1281
INONLY PRODUCER -（デフォルト） 1281
INOUT プロデューサー 1281
コンシューマー設定オプション 1282
コンシューマーの使用 1283
INONLY CONSUMER:（デフォルト） 1283
INOUT コンシューマー 1283
高度な使用方法に関する注意点 1284
プラグイン可能な接続リソース管理 1284
セッション、コンシューマー、プロデューサープーリングおよびキャッシング管理 1285
バッチメッセージのサポート 1285
カスタマイズ可能なトランザクションのコミットストラテジー（ローカル JMS トランザクションのみ） 1286
トランザクションバッチコンシューマーおよびプロデューサー 1286
追記 1287
メッセージヘッダーの形式 1287
メッセージの内容 1288
クラスタリング 1288
トランザクションサポート 1288
SPRINGLESS MEAN I CAN'T USE SPRING? 1289

第151章 SJMS BATCH 1290
SJMS バッチコンポーネント 1290
URI 形式 1291
コンポーネントのオプションおよび設定 1292

第152章 SLACK 1295
SLACK コンポーネント 1295
URI 形式 1295
オプション 1295
SLACKCOMPONENT 1296
例 1296

第153章 SMPP 1297
SMPP コンポーネント 1297
SMS の制限 1297
データコーディング、アルファベット、および国際文字セット 1298
メッセージの分割とスロットリング 1299
URI 形式 1299
URI オプション 1300
プロデューサーメッセージヘッダー 1307
コンシューマーメッセージヘッダー 1313
例外処理 1318
サンプル 1318

デバッグロギング 1319

第154章 SNMP 1321
SNMP コンポーネント 1321
URI 形式 1321
オプション 1321
ポーリングの結果 1322
例 1323

第155章 SOLR 1324
SOLR コンポーネント 1324
URI 形式 1324
エンドポイントオプション 1324
メッセージ操作 1325
例 1326
SOLR のクエリー 1327

第156章 APACHE SPARK 1329
APACHE SPARK コンポーネント 1329
サポート対象のアーキテクチャースタイル 1329
OSGI サーバーでの SPARK の実行 1330
URI 形式 1330
RDD ジョブ 1330
RDD ジョブオプション 1331
VOID RDD コールバック 1332
RDD コールバックの変換 1332
アノテーション付きの RDD コールバック 1332
DATAFRAME ジョブ 1333
DATAFRAME ジョブオプション 1335
HIVE ジョブ 1335
HIVE ジョブオプション 1336

第157章 SPARK REST 1337
SPARK REST コンポーネント 1337
URI 形式 1337
URI オプション 1337
SPARK 構文を使用したパス 1338
CAMEL メッセージへのマッピング 1338
REST DSL 1338
その他の例 1339

第158章 SPLUNK 1340
SPLUNK コンポーネント 1340
URI 形式 1340
プロデューサーエンドポイント： 1340
コンシューマーエンドポイント： 1341
URI オプション 1341
メッセージボディー 1343
ユースケース 1343
その他のコメント 1344
関連項目 1344

第159章 SPRINGBATCH 1346
SPRING BATCH コンポーネント 1346
URI 形式 1346

オプション 1346
使用方法 1347
例 1347
サポートクラス 1348
CAMELITEMREADER 1348
CAMELITEMWRITER 1349
CAMELITEMPROCESSOR 1349
CAMELJOBEXECUTIONLISTENER 1350

第160章 SPRINGINTEGRATION 1351
SPRING INTEGRATION コンポーネント 1351
URI 形式 1351
オプション 1351
使用方法 1352
SPRING インテグレーションエンドポイントの使用 1352
SOURCE アダプターおよび TARGET アダプター 1354

第161章 SPRING イベント 1357
SPRING イベントコンポーネント 1357
URI 形式 1357

第162章 SPRING LDAP 1358
SPRING LDAP コンポーネント 1358
URI 形式 1358
オプション 1358
使用方法 1359
検索 1359
バインド 1359
UNBIND 1360

第163章 SPRING REDIS 1361
SPRING REDIS コンポーネント 1361
URI 形式 1361
URI オプション 1361
使用方法 1362
REDIS プロデューサーによって評価されるメッセージヘッダー 1362
REDIS コンシューマー 1372
DEPENDENCIES 1373

第164章 SPRING WEB SERVICES 1374
SPRING WEB SERVICES コンポーネント 1374
URI 形式 1374
オプション 1375
メッセージヘッダー 1378
WEB サービスへのアクセス 1379
SOAP および WS-ADDRESSING アクションヘッダーの送信 1380
SOAP ヘッダーの使用 1380
ヘッダーおよび添付の伝播 1381
MTOM 添付ファイルの使用方法 1381
カスタムヘッダーおよび添付フィルター 1382
カスタム MESSAGESENDER および MESSAGEFACTORY の使用 1383
WEB サービスの公開 1384
ルートでのエンドポイントマッピング 1385
既存のエンドポイントマッピングを使用した代替設定 1385

POJO (UN) MARSHALLING 1386

第165章 SQL COMPONENT 1388
SQL COMPONENT 1388
CAMEL ON EAP デプロイメント 1388
URI 形式 1389
オプション 1390
メッセージボディーの処理 1399
クエリーの結果 1399
ヘッダーの値 1400
生成されるキー 1400
設定 1401
例 1401
名前付きパラメーターの使用 1402
式パラメーターの使用 1403
動的値による IN クエリーの使用 1403
JDBC ベースのべき等リポジトリーの使用 1404
JDBCMESSAGEIDREPOSITORY のカスタマイズ 1406
JDBC ベースの集約リポジトリーの使用 1407
永続化時に保持される内容 1410
復元 1410
データベース 1411
ボディーとヘッダーをテキストとして保存 1411
CODEC (SERIALIZATION) 1412
TRANSACTION 1413
サービス（開始/停止） 1413
AGGREGATOR の設定 1413
OPTIMISTIC LOCKING 1413

第166章 SQL ストアドプロシージャ 1416
SQL ストアドプロシージャコンポーネント 1416
URI 形式 1416
オプション 1417
ストアドプロシージャーテンプレートの宣言 1418

第167章 SSH 1419
SSH 1419
URI 形式 1419
オプション 1419
コンシューマーのみのオプション 1420
プロデューサーエンドポイントとしての使用 1421
認証 1421
証明書の依存関係 1423
例 1423

第168章 STAX 1424
STAX COMPONENT 1424
URI 形式 1424
コンテンツハンドラーを STAX パーサーとして使用する 1424
JAXB および STAX を使用してコレクションを繰り返し処理します。 1425
XML DSL を使用した前述の例 1427

第169章 STOMP 1428
STOMP コンポーネント 1428

URI 形式 1428
オプション 1428
サンプル 1429

第170章 ストリーム 1430
ストリームコンポーネント 1430
URI 形式 1430
CAMEL ON EAP デプロイメント 1430
オプション 1430
メッセージの内容 1432
サンプル 1432

第171章 STRINGTEMPLATE 1435
STRING TEMPLATE 1435
URI 形式 1435
オプション 1435
HEADERS 1436
ホットリロード 1436
STRINGTEMPLATE 属性 1436
サンプル 1437
電子メールのサンプル 1437

第172章 STUB 1439
スタブコンポーネント 1439
URI 形式 1439
例 1439

第173章 SWAGGER 1441
173.1. 概要 1441
Dependencies 1441
Swagger サーブレットの選択 1441
サーブレット設定パラメーター 1442
CorsFilter の使用 1443
173.2. WAR デプロイメントの設定 1444
Camel 2.15.x 1445
Camel 2.14.x 1446
173.3. OSGI デプロイメントの設定 1447

第174章 SWAGGER JAVA 1450
SWAGGER JAVA コンポーネント 1450
SWAGGER をサーブレットとして使用 1450
REST-DSL での SWAGGER の使用 1451
オプション 1452
CORSFILTER 1454
CONTEXTIDLISTING ENABLED 1455
JSON または YAML 1455
例 1456

第175章 TEST 1457
テストコンポーネント 1457
URI 形式 1457
URI オプション 1457
例 1458

第176章 TIMER 1459

タイマーコンポーネント 1459
URI 形式 1459
オプション 1459
エクスチェンジプロパティー 1460
メッセージヘッダー 1461
例 1461
できるだけ早く実行する 1461
1 度だけ発行 1462

第177章 TWITTER 1463
TWITTER 1463
URI 形式 1463
TWITTERCOMPONENT: 1463
コンシューマーエンドポイント： 1464
プロデューサーエンドポイント： 1464
URI オプション 1465
メッセージヘッダー 1467
メッセージボディー 1468
API 流量制御 1468
TWITTER プロファイル内でステータス更新を作成するには、このプロデューサーを STRING ボディーを送信します。 1468
60 秒ごとに、自宅のタイムラインのすべてのステータスをポーリングするには、以下を実行します。 1468
キーワード 'CAMEL' ですべてのステータスを検索するには、以下を実行します。 1469
静的キーワードでプロデューサーを使用した検索 1469
ヘッダーから動的キーワードを持つプロデューサーを使用した検索 1469
例 1469

第178章 UNDERTOW 1470
UNDERTOW コンポーネント 1470
URI 形式 1470
オプション 1470
メッセージヘッダー 1472
コンポーネントオプション 1472
プロデューサーの例 1472
コンシューマーの例 1473

第179章 検証 1474
検証コンポーネント 1474
URI 形式 1474
オプション 1475
例 1476
ADVANCED: JMX METHOD CLEARCACHEDSCHEMA 1476

第180章 VELOCITY 1478
VELOCITY 1478
URI 形式 1478
オプション 1478
メッセージヘッダー 1479
VELOCITY コンテキスト 1480
ホットリロード 1480
動的テンプレート 1481
サンプル 1481
電子メールのサンプル 1482

第181章 VERTX 1484
VERTX コンポーネント 1484
URI 形式 1484
オプション 1484
既存の VERT.X インスタンスへの接続 1485

第182章 VM 1486
仮想マシンコンポーネント 1486
URI 形式 1486
オプション 1487
サンプル 1487

第183章 VIDEO 1489
VIDEO コンポーネント 1489
CAMEL ON EAP デプロイメント 1489
URI 形式 1489
オプション 1490
エクスチェンジデータ形式 1491
メッセージヘッダー 1491
サンプル 1491

第184章 WEBSOCKET 1493
WEBSOCKET コンポーネント 1493
URI 形式 1493
コンポーネントオプション 1493
エンドポイントオプション 1494
メッセージヘッダー 1496
使用方法 1497
WEBOCKET コンポーネントの SSL の設定 1497
JSSE 設定ユーティリティーの使用 1497
エンドポイントの SPRING DSL ベースの設定 1498
エンドポイントの JAVA DSL ベースの設定 1498

第185章 XMLRPC 1500
XMLRPC COMPONENT 1500
XMLRPC OVERVIEW 1500
URI 形式 1501
オプション 1501
メッセージヘッダー 1503
XMLRPC データフォーマットの使用 1503
クライアントからの XMLRPC サービスの呼び出し 1504
JAVA コードで XMLRPCCLIENT を設定する方法 1504

第186章 XML セキュリティーコンポーネント 1506
XML セキュリティーコンポーネント 1506
XML 署名ラッピングモード 1506
URI 形式 1508
基本的な例 1509
一般的な署名およびオプションの検証 1509
署名オプション 1510
オプションの検証 1517
ENVELOPING XML 署名ケースの出力ノードの決定 1521
署名要素のシブリングとしての分離された XML 署名 1522
SIGNER エンドポイントの XADES-BES/EPES 1524

HEADERS 1528
XADES バージョン 1.4.2 に関する制限 1529
関連項目 1530

第187章 XMPP **1531**
XMPP コンポーネント 1531
URI 形式 1531
オプション 1531
ヘッダーおよび SUBJECT または LANGUAGE の設定 1532
例 1532

第188章 XQUERY エンドポイント **1534**
XQUERY 1534
URI 形式 1534

第189章 XSLT **1535**
XSLT 1535
URI 形式 1535
オプション 1535
XSLT エンドポイントの使用 1538
連携する XSLT へのパラメーターの取得 1539
SPRING XML バージョン 1539
XSL:INCLUDE の使用 1539
XSL:INCLUDE およびデフォルト接頭辞の使用 1540
SAXON 拡張機能の使用 1541
動的スタイルシート 1542
XSLT ERRORLISTENER からの警告、エラー、および致命的なエラーへのアクセス 1542

第190章 XSTREAM **1544**
XSTREAM コンポーネント 1544
CAMEL ON EAP デプロイメント 1544

第191章 YAMMER **1545**
YAMMER 1545
URI 形式 1545
YAMMERCOMPONENT 1545
メッセージの消費 1546
メッセージの消費用の URI オプション 1546
メッセージの形式 1547
メッセージの作成 1550
ユーザー関係の取得 1550
関係を取得するための URI オプション 1550
ユーザーの取得 1551
ユーザーを取得するための URI オプション 1551
ENRICHER の使用 1552

第192章 ZOOKEEPER **1553**
ZOOKEEPER 1553
CAMEL ON EAP デプロイメント 1553
URI 形式 1553
オプション 1554
ユースケース 1554
ZNODE からの読み取り。 1554
ZNODE からの読み取り -（追加の CAMEL 2.10 以降） 1555
ZNODE への書き込み。 1555

ZOOKEEPER が有効な ROUTE ポリシー。 1557

第1章 コンポーネントの概要

概要

本章では、Apache Camel で利用可能なすべてのコンポーネントの概要を説明します。

1.1. APACHE KARAF の CAMEL コンポーネントのリスト

コンポーネントの表

以下の Camel コンポーネントは Apache Karaf (OSGi)コンテナーでサポートされます。

表1.1 Apache Camel のコンポーネント

コンポーネント	エンドポイント URI	アーティファクト ID	説明
ActiveMQ	**activemq:[queue:|topic:]*DestinationName***	**activemq-core**	Apache ActiveMQ を使用した JMS メッセージングの場合。
AHC	**ahc:http[s]://*Hostname*[:*Port*][/*ResourceUri*]**	**camel-ahc**	Async Http Client ライブラリーを使用して外部 HTTP サーバーを呼び出します。
AHC-WS	**ahc-ws[s]://*Hostname*[:*Port*][/*ResourceUri*]**	**camel-ahc-ws**	Async Http Client ライブラリーを使用して外部 WebSocket サーバーを呼び出します。
AMQP	**amqp:[queue:|topic:]*DestinationName*[?*Options*]]**	**camel-amqp**	AMQP プロトコルのあるメッセージングの場合。
APNS	**apns:notify[?*Options*] apns:consumer[?*Options*]**	**camel-apns**	Apple iOS デバイスに通知を送信する場合
Atmosphere-WebSocket	**atmosphere-websocket:///*RelativePath*[?*Options*]**	**camel-atmosphere-websocket**	Atmosphere を使用した外部 WebSocket クライアントからの接続を受け入れます。
Atom	**atom://*AtomUri*[?*Options*]**	**camel-atom**	atom フィードの使用など、atom 統合での Apache Abdera の使用

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Avro	**avro:http://*Hostname*[:*Port*][?*Options*]**	**camel-avro**	データシリアライゼーションのための Apache Avro の使用
AWS-CW	**aws-cw://*Namespace*[?*Options*]**	**camel-aws**	メトリクスを Amazon CloudWatch に送信します。
AWS-DDB	**aws-ddb://*TableName*[?*Options*]**	**camel-aws**	Amazon の DynamoDB (DDB)の 使用向け
AWS-SDB	**aws-sdb://*DomainName*[?*Options*]**	**camel-aws**	Amazon の SimpleDB (SDB) を操作する場合：
AWS-SES	**aws-ses://*From*[?*Options*]**	**camel-aws**	Amazon の Simple Email Service (SES) の使用
AWS-S3	**aws-s3://*BucketName*[?*Options*]**	**camel-aws**	Amazon の Simple Storage Service (S3)の使用向け
AWS-SNS	**aws-sns://*TopicName*[?*Options*]**	**camel-aws**	Amazon の Simple Notification Service (SNS)を使用したメッセージングの 場合。
AWS-SQS	**aws-sqs://*QueueName*[?*Options*]**	**camel-aws**	Amazon の Simple Workflow Service (SWF) を使用したメッセージングの 場合
AWS-SWF	**aws-swf://{*workflow*|*activity*}[?*Options*]**	**camel-aws**	Amazon の Simple Queue Service (SQS) からワークフローを管理する場合。
Bean	**bean:*BeanID*[?methodName=*Method*]**	**camel-core**	Bean バインディングを使用して、メッセージエクスチェンジをレジストリーの Bean にバインドします。は POJO (Plain Old Java Objects)の公開および呼び出しにも使用されます。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Beanstalk	**beanstalk://[*Hostname*[:*port*]][/*tube*][?*options*]**	**camel-beanstalk**	処理後の Beantalk ジョブの取得および後処理用
Bean Validation	**bean-validator:*Something*[?*Options*]**	**camel-bean-validator**	Java Validation API (JSR 303 および JAXP Validation)および参照実装 Hibernate Validator を使用してメッセージのペイロードを検証します。
Bindy	該当なし	**camel-bindy**	非構造化データの解析とバインドを有効にします。
参照	**browse:** *名前*	**camel-core**	テスト、視覚化ツール、またはデバッグに役立つ簡単な **BrowsableEndpoint** を提供します。エンドポイントに送信されたエクスチェンジはすべて参照できます。
Cache	**cache://*CacheName*[?*Options*]**	**camel-cache**	キャッシュコンポーネントを使用すると、EHCache をキャッシュ実装として使用してキャッシュ操作を実行できます。
CDI	該当なし	**camel-cdi**	CDI 統合を提供します。
クラス	**class:*ClassName*[?method=*MethodName*]**	**camel-core**	Bean バインディングを使用して、メッセージエクスチェンジをレジストリーの Bean にバインドします。は POJO の公開および呼び出しにも使用されます(Plain Old Java Objects)。
CMIS	**cmis:*CmisServerUrl*[?*Options*]**	**camel-cmis**	Apache Chemistry クライアント API を使用して、CMIS がサポートする CMS とインターフェイスします。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Cometd	**cometd://*Hostname*[:*Port*]/*ChannelName*[?*Options*]**	**camel-cometd**	cometd/bayeux プロトコルの jetty 実装を使用するトランスポート。
コンテキスト	**context:*CamelContextId*:*LocalEndpointName***	**camel-context**	別の **CamelContext** のエンドポイントを参照します。
ControlBus	**controlbus:*Command*[?*Options*]**	**camel-core**	Camel アプリケーションの管理および監視用にエンドポイントにメッセージを送信できるようにする ControlBus **Enterprise Integration Pattern**。
CouchDB	**couchdb:http://*Hostname*[:*Port*]/*Database*[?*Options*]://*Name*[?*Options*]**	**camel-couchdb**	CouchDB インスタンスをメッセージのプロデューサーまたはコンシューマーとして扱うことができます。
crypto	**crypto:sign:*Name*[?*Options*] crypto:verify:*Name*[?*Options*]**	**camel-crypto**	Java Cryptographic Extension の Signature Service を使用してエクスチェンジに署名し、検証します。
CXF	**cxf://*Address*[?*Options*]**	**camel-cxf**	Web サービスの統合のための Apache CXF の使用
CXF Bean	**cxf:*BeanName***	**camel-cxf**	レジストリーから JAX WS または JAX RS アノテーションが付けられた Bean を使用してエクスチェンジを提案します。
CXFRS	**cxfrs:bean:*RsEndpoint*[?*Options*]**	**camel-cxf**	CXF でホストされる JAX-RS サービスに接続するための Apache CXF との統合を提供します。
DataFormat	**dataformat:*Name*:(marshal|unmarshal)[?*Options*]**	**camel-core**	エンドポイントにメッセージを送信して、標準の Camel データフォーマットのいずれかでメッセージをマーシャリングまたはアンマーシャリングできます。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
DataSet	**dataset:*Name*[?*Options*]**	**camel-core**	ロードおよび soak テストの場合、DataSet はコンポーネントに送信するための大量のメッセージを作成したり、適切に消費されることをアサートしたりする方法を提供します。
Direct	**direct:*EndpointID*[?*Options*]**	**camel-core**	同じ CamelContext からの別のエンドポイントへの同期呼び出し（シングルスレッド）。
Direct-VM	**direct-vm:*EndpointID*[?*Options*]**	**camel-core**	同じ JVM で実行されている別の CamelContext の別のエンドポイントへの同期呼び出し（シングルスレッド）。
Disruptor	**disruptor:*Name*[?*Options*] disruptor-vm:*Name*[?*Options*]**	**camel-disruptor**	SEDA エンドポイントと同様ですが、ブロッキングキューの代わりに Disruptor を使用します。
DNS	**dns:*Operation***	**camel-dns**	ドメイン情報を検索し、DNSJava を使用して DNS クエリーを実行します。
Docker	**docker:*Operation*[?*Options*]**	**camel-docker**	Docker Remote API 経由で docker-java を活用します。
Dozer	**dozer:*EndpointID*[?*Options*]**	**camel-dozer**	Dozer マッピングフレームワークを使用して Java Bean 間のマッピング機能を提供します。
Dropbox	**dropbox://[*Operation*][?*Options*]**	**camel-dropbox**	Dropbox リモートフォルダーからメッセージを送受信します。
ElasticSearch	**elasticsearch:*ClusterName***	**camel-elasticsearch**	ElasticSearch サーバーと対話する場合。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
ElSql	**elsql:*elSqlName*:*resourceUri*[?*Options*]**	**camel-elsql**	ElSql を使用して SQL クエリーを定義する既存の SQL コンポーネントのエクステンション。
etcd	**etcd:*nameSpace*[/*path*][?*Options*]**	**camel-etcd**	Etcd の分散キー/値のストアを使用するために使用します。
EventAdmin	**eventadmin:topic**	**camel-eventadmin**	
exec	**exec://*Executable*[?*Options*]**	**camel-exec**	システムコマンドを実行します。
fabric	**fabric:*ClusterID*[:*PublishedURI*][?*Options*]**	**fabric-camel**	fabric エンドポイントを検索または公開します。
Facebook	**facebook://[*Endpoint*][?*Options*]**	**camel-facebook**	Facebook4J を使用してアクセス可能なすべての Facebook API へのアクセスを提供します。
File2	**file://*DirectoryName*[?*Options*]**	**camel-core**	ファイルへのメッセージを送信するか、ファイルまたはディレクトリーをポーリングします。
flatpack	**flatpack:[fixed|delim]:** *ConfigFile*	**camel-flatpack**	FlatPack ライブラリーを使用した固定幅または区切られたファイルまたはメッセージの処理
FOP	**fop:*OutputFormat***	**camel-fop**	Apache FOP を使用して、メッセージを異なる出力形式にレンダリングします。
FreeMarker	**freemarker:** *TemplateResource*	**camel-freemarker**	Freemarker テンプレートを使用して応答を生成します。
FTP2	**ftp://[*Username*@]*Hostname*[:*Port*]/*Directoryname*[?*Options*]**	**camel-ftp**	FTP でのファイルの送受信

コンポーネント	エンドポイント URI	アーティファクト ID	説明
gauth	**gauth://*Name*[?*Options*]**	**camel-gae**	Google 固有の OAuth コンシューマーを実装するために Web アプリケーションによって使用されます。
GHTTP	**ghttp:///*Path*[?*Options*]** **ghttp://*Hostname*[:*Port*]/*Path*[?*Options*]** **ghttps://*Hostname*[:*Port*]/*Path*[?*Options*]**	**camel-gae**	GAE URL フェッチサービスへの接続を提供し、サーブレットからメッセージを受信するためにも使用できます。
Git	**git://*localRepositoryPath*[?*Options*]**	**camel-git**	一般的な Git リポジトリーと連携できます。
GitHub	**github://*endpointId*[?*Options*]**	**camel-github**	github API との対話用。
GLogin	**glogin://*Hostname*[:*Port*][?*Options*]**	**camel-gae**	GAE アプリケーションにプログラムによるログインを行うために、Google App Engine (GAE)以外の Camel アプリケーションによって使用されます。
Gmail	**gmail://*Username*@gmail.com[?*Options*]** **gmail://*Username*@googlemail.com[?*Options*]**	**camel-gae**	GAE メールサービスを介したメールの送信をサポートします。
GoogleCalendar	**google-calendar://*endpoint-prefix*/*endpointId*[?*Options*]**	**camel-google-calendar**	Google Calendar Web API を使用して Google カレンダーにアクセスする場合。
GoogleMail	**google-mail://*endpoint-prefix*/*endpointId*[?*Options*]**	**camel-google-mail**	Google Mail Web API を使用して Gmail にアクセスする場合。
gtask	**gtask://*QueueName***	**camel-gae**	タスクキューをメッセージキューとして使用して GAE での非同期メッセージ処理をサポートします。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Geocoder	**geocoder:*Address:Name*[?*Options*] geocoder:latlng:*Latitude,Longitude*[?*Options*]**	**camel-geocoder**	指定のアドレスのジオコード(latitude および longitude)を検索するか、リバースルックアップを実行します。
GoogleDrive	**google-drive://*EndpointPrefix*/*Endpoint*[?*Options*]**	**camel-google-drive**	Google ドライブ ファイルストレージサービスへのアクセスを提供します。
Guava EventBus	**guava-eventbus:*BusName*[?*EventClass=ClassName*]**	**camel-guava-eventbus**	Google Guava EventBus は、コンポーネントを相互に明示的に登録しなくても、コンポーネント間のパブリッシュ/サブスクライブスタイルの通信を可能にします（そのため、相互に認識する必要があります）。このコンポーネントは、Camel と Google Guava EventBus インフラストラクチャー間の統合ブリッジを提供します。
Hazelcast	**hazelcast://*StoreType:CacheName*[?*Options*]**	**camel-hazelcast**	Hazelcast は、Java（単一の JAR）に完全に実装されたデータグリッドです。このコンポーネントは、マップ、マルチマップ、seda、queue、set、atomic 番号、および単純なクラスターをサポートします。
HDFS	**hdfs://*Hostname*[:*Port*][/*Path*][?*Options*]**	**camel-hdfs**	HDFS (Hadoop 1.x)を使用して、Hadoop Distributed File System (HDFS) の読み取りと書き込みを行います。
HDFS2	**hdfs2://*Hostname*[:*Port*][/*Path*][?*Options*]**	**camel-hdfs2**	HDFS (Hadoop 2.x)を使用して、Hadoop Distributed File System (HDFS) の読み取りと書き込みを行います。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Hipchat	**hipchat://*Hostname*[:*Port*][?*Options*]**	**camel-hipchat**	Hipchat サービスを介してメッセージを生成および消費する場合。
HL7	**mina:tcp://*Host*[:*Port*]**	**camel-hl7**	HL7 MLLP プロトコルおよび HL7 モデルを使用する場合は、HAPI ライブラリー を使用します。
HTTP	**http://*Hostname*[:*Port*][/*ResourceUri*]**	**camel-http**	Apache HTTP Client 3.x を使用して外部 HTTP サーバーを呼び出す場合。
HTTP4	**http4://*Hostname*[:*Port*][/*ResourceUri*]**	**camel-http4**	Apache HTTP Client 4.x を使用して外部 HTTP サーバーを呼び出す場合。
iBATIS	**ibatis:*OperationName*[?*Options*]**	**camel-ibatis**	Apache iBATIS を使用してリレーショナルデータベースでクエリー、ポーリング、挿入、更新、または削除を実行します。
IMAP	**imap://[*UserName*@]*Host*[:*Port*][?*Options*]**	**camel-mail**	IMap を使用して電子メールを受信する。
IRC	**irc:*Host*[:*Port*]/#*Room***	**camel-irc**	IRC 通信用。
jasypt		**camel-jasypt**	Jasypt と統合し、プロパティー ファイルの機密情報を暗号化できるようにします。
JCache	**cache://*cacheName*[?*Options*]**	**camel-jcache**	JCache (JSR-107)をキャッシュ実装として使用してキャッシュ操作を実行します。
jclouds	**jclouds:[*Blobstore*|*ComputService*]:*Provider***	**camel-jclouds**	JClouds を介してクラウドコンピュートおよび Blobstore サービスを操作する場合

コンポーネント	エンドポイント URI	アーティファクト ID	説明
JCR	**jcr://*UserName*:*Password*@*Repository*/path/to/node**	**camel-jcr**	Apache Jackrabbit などの JCR (JSR-170)準拠のリポジトリーへのメッセージの保存。
JDBC	**jdbc:*DataSourceName*[?*Options*]**	**camel-jdbc**	JDBC クエリーおよび操作を実行する場合。
Jetty	**jetty:http://*Host*[:*Port*][/*ResourceUri*]**	**camel-jetty**	HTTP 経由でサービスを公開する場合。
JGroups	**jgroups:*ClusterName*[?*Options*]**	**camel-jgroups**	JGroups クラスターでメッセージを交換します。
jing	jing:*LocalOrRemoteResource* jing:*LocalOrRemoteResource*?compactSyntax=true	**camel-jing**	RelaxNG または RelaxNG compact 構文を使用してメッセージのペイロードを検証します。
JIRA	**jira://*endpointId*[?*Options*]**	**camel-jira**	JIRA 向けの Atlassian の REST Java Client をカプセル化して JIRA API と対話する場合。
JMS	**jms:[temp:][queue:|topic:]*DestinationName*[?*Options*]**	**camel-jms**	JMS プロバイダーの使用
JMX	**jmx://*Platform*[?*Options*]**	**camel-jmx**	JMX 通知リスナーの使用
Jolt	**jolt:*specName*[?*Options*]**	**camel-jolt**	JOLT 仕様を使用して JSON メッセージを処理できます。
JPA	**jpa:[*EntityClassName*][?*Options*]**	**camel-jpa**	OpenJPA、Hibernate、または TopLink を使用するために JPA 仕様を介してデータベースをキューとして使用する場合。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
jsch	**scp://*Hostname*/*Destination***	**camel-jsch**	**scp** プロトコルのサポート。
JT400	**jt400://*User*:*Pwd*@*System*/*PathToDTAQ***	**camel-jt400**	AS/400 （システム i、IBM i、i5 など）システムでデータキューと統合する場合
Kafka	**kafka://*Hostname*[:*Port*][?*Options*]**	**camel-kafka**	Apache Kafka メッセージブローカーからメッセージを送受信します。
Kestrel	**kestrel://[*AddressList*/]*Queuename*[?*Options*]**	**camel-kestrel**	Kestrel キューから生成または消費されます。
Krati	**krati://[*PathToDatastore*/][?*Options*]**	**camel-krati**	Krati データストアの生成または消費。
Kubernetes	**kubernetes:[*masterUrl*][?*Options*]**	**camel-kubernetes**	アプリケーションを Kubernetes スタンドアロンと統合する場合や、OpenShift 上で統合する場合。
言語	**language://*LanguageName*[:*Script*][?*Options*]**	**camel-core**	言語スクリプトを実行します。
LDAP	**ldap:*Host*[:*Port*]?base=...[&scope=*Scope*]**	**camel-ldap**	LDAP サーバーでの検索の実行(*範囲* は **object|onelevel|subtree**のいずれかでなければなりません)。
LevelDB	該当なし	**camel-leveldb**	非常に軽量で組み込み可能なキーと値のデータベース。
リスト	**list:*ListID***	**camel-core**	テスト、視覚化ツール、またはデバッグに役立つ簡単な BrowsableEndpoint を提供します。エンドポイントに送信されたエクスチェンジはすべて参照できます。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Log	**log:*LoggingCategory*[?level=*LoggingLevel*]**	**camel-core**	Jakarta Commons Logging を使用して、log4j などの基礎となるロギングシステムにメッセージ交換をログに記録します。
Lucene	**lucene:*SearcherName*:insert[?analyzer=*Analyzer*]** **lucene:*SearcherName*:query[?analyzer=*Analyzer*]**	**camel-lucene**	高度な分析/トークン化機能を使用して、Apache Lucene を使用して Java ベースのインデックスと完全なテキストベースの検索を実行します。
マスター	**REVISIT**		
メトリクス	**metrics:[meter|counter|histogram|timer]:*Metricname*[?*Options*]**	**camel-metrics**	Metrics Java ライブラリーを使用して、Camel ルートから直接さまざまなメトリクスを収集できます。
MINA2	**mina2:tcp://*Hostname*[:*Port*][?*Options*]** **mina2:udp://*Hostname*[:*Port*][?*Options*]** **mina2:vm://*Hostname*[:*Port*][?*Options*]**	**camel-mina2**	Apache MINA 2.x の使用
MLLP	該当なし	**camel-mllp**	MLLP プロトコルを使用してシステム間の通信を行う場合このコンポーネントは、簡単な設定 URI と HL7 の自動署名およびインターイメントを提供します。
Mock	**mock:*EndpointID***	**camel-core**	モックを使用してルートおよび仲介ルールをテストする場合。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
MongoDB	**mongodb:*Connection*[?*Options*]**	**camel-mongodb**	MongoDB データベースおよびコレクションと対話します。プロデューサーエンドポイントを提供して、CRUD スタイルの操作や、データベースおよびコレクションに対してより多くの操作を実行し、コレクションおよびディスパッチオブジェクトを Camel ルートにリッスンするコンシューマーエンドポイントを提供します。
MQTT	**mqtt:*Name***	**camel-mqtt**	MQTT M2M メッセージブローカーと通信するためのコンポーネント
MSV	**msv:*LocalOrRemoteResource***	**camel-msv**	MSV ライブラリーを使用してメッセージのペイロードを検証します。
Mustache	**mustache:*TemplateName*[?*Options*]**	**camel-mustache**	Mustache テンプレートを使用してメッセージを処理できます。
MVEL	**mvel:*TemplateName*[?*Options*]**	**camel-mvel**	MVEL テンプレートを使用してメッセージを処理できます。
mybatis	**mybatis:*StatementName***	**camel-mybatis**	MyBatis を使用してリレーショナルデータベースでクエリー、ポーリング、挿入、更新、または削除を実行します。
Nagios	**nagios://*Host*[:*Port*][?*Options*]**	**camel-nagios**	JSendNSCA を使用して Nagios にパッシブチェックを送信します。
Netty	**netty:tcp://localhost:99999[?*Options*]** **netty:udp://*Remotehost*:99999/[?*Options*]**	**camel-netty**	Netty バージョン 3.x によって提供される Java NIO ベースの機能を使用して、TCP プロトコルおよび UDP プロトコルと連携できます。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Netty4	**netty4:tcp://localhost:99999[?*Options*] netty4:udp://*Remote host*:99999/[?*Options*]**	**camel-netty4**	Netty バージョン 4.x によって提供される Java NIO ベースの機能を使用して、TCP プロトコルおよび UDP プロトコルと連携できます。
Netty HTTP	**netty-http:http://*Hostname*[:*Port*][?*Options*]**	**camel-netty-http**	Netty コンポーネントへの拡張機能。Netty バージョン 3.x を使用した HTTP トランスポートを容易にします。
Netty4 HTTP	**netty4-http:http://*Hostname*[:*Port*][?*Options*]**	**camel-netty4-http**	Netty コンポーネントへの拡張機能。Netty バージョン 4.x を使用した HTTP トランスポートを容易にします。
OGNL		**camel-ognl**	OGNL は、Java オブジェクトのプロパティーを取得および設定するための式言語です。
Olingo2	**olingo2://*Endpoint*/*ResourcePath*[?*Options*]**	**camel-olingo2**	Apache Olingo 2.0 を使用して OData 2.0 サービスと通信します。
Paho	**paho:*QName*[?*Options*]**	**camel-paho**	Eclipse Paho ライブラリーを使用して MQTT メッセージングプロトコルのコネクターを提供します。
pax-Logging	**paxlogging:*Appender***	**camel-paxlogging**	OSGi コンテナーのコンテキストで Pax ロギングイベントを受信します。
PDF	**pdf:*Operation*[?*Options*]**	**camel-pdf**	PDF ドキュメントからコンテンツを作成、変更、または抽出する機能を提供します。
PGEvent	**pgevent:*Datasource*[?*Parameters*]** **pgevent://[*HostName*][:*Port*]/*Database*/*Channel*[?*Parameters*]**	**camel-pgevent**	LISTEN/NOTIFY コマンドに対して PostgreSQL イベントを生成および消費する場合。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
POP	**pop3://[*UserName@*]*Host*[:*Port*][?*Options*]**	**camel-mail**	POP3 および JavaMail を使用して電子メールを受信します。
プリンター	**lpr://localhost[:*Port*]/default[?*Options*]** **lpr://*RemoteHost*[:*Port*]/path/to/printer[?*Options*]**	**camel-printer**	ルート上のペイロードをプリンターに転送する方法を提供します。
プロパティー	**properties://*Key*[?*Options*]**	**camel-properties**	エンドポイント URI 定義で直接プロパティープレースホルダーを使用することを容易にします。
Quartz	**quartz://[*GroupName*/]*TimerName*[?*Options*]** **quartz://*GroupName*/*TimerName*/*CronExpression***	**camel-quartz**	Quartz スケジューラーを使用して、スケジュールされたメッセージの配信を提供します。
Quartz2	**quartz2://[*GroupName*/]*TimerName*[?*Options*]** **quartz2://*GroupName*/*TimerName*/*CronExpression***	**camel-quartz2**	Quartz スケジューラー 2.x を使用したスケジュールされたメッセージの配信を提供します。
QuickFix	**quickfix-server:*ConfigFile*** **quickfix-client:*ConfigFile***	**camel-quickfix**	FIX メッセージを送受信できる Java エンジンの QuickFix の実装。
RabbitMQ	**rabbitmq://*Hostname*[:*Port*]/*ExchangeName*[?*Options*]**	**camel-rabbitmq**	RabbitMQ インスタンスからメッセージを生成および消費できます。
Ref	**ref:*EndpointID***	**camel-core**	レジストリーにバインドされる既存エンドポイントを検索するためのコンポーネント。
REST	**rest://*Method*:*Path*[:*UriTemplate*][?*Options*]**	**camel-rest**	Apache Camel Development Guide の Defining Services with REST DSL セクションを使用して REST エンドポイントを定義できます。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Restlet	**restlet:*RestletUrl*[?*Options*]**	**camel-restlet**	Restlet を使用して Restful リソースを消費および生成するコンポーネント。
RMI	**rmi://*RmiRegistryHost*:*RmiRegistryPort*/*RegistryPath***	**camel-rmi**	RMI の操作。
Routebox	**routebox:*routeboxName*[?*Options*]**	**camel-routebox**	
RSS	**rss:*Uri***	**camel-rss**	RSS フィードの使用など、RSS 統合で ROME と連携します。
Salesforce	**salesforce:*Topic*[?*Options*]**	**camel-salesforce**	プロデューサーおよびコンシューマーエンドポイントが Java DTO を使用して Salesforce と通信できるようにします。
SAP	**sap:[destination:*DestinationName*|server:*ServerName*]*rfcName*[?*Options*]**	**camel-sap**	同期リモート関数呼び出し sRFC を使用して、SAP システムへの送受信通信を有効にします。
SAP NetWeaver	**sap-netweaver:https://*Hostname*[:*Port*]/*Path*[?*Options*]**	**camel-sap-netweaver**	HTTP トランスポートを使用して SAP NetWeaver Gateway と統合します。
Schematron	**schematron://*Path*[?*Options*]**	**camel-schematron**	Schematron を使用して XML ドキュメントを検証します。
SEDA	**seda:*EndpointID***	**camel-core**	java.util.concurrent.BlockingQueue にメッセージを配信するために使用されます。これは、同じ CamelContext 内で SEDA スタイルの処理パイプラインを作成する場合に役立ちます。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
SERVLET	**servlet://*RelativePath*[?*Options*]**	**camel-servlet**	HTTP エンドポイントに到達する HTTP リクエストを消費するための HTTP ベースのエンドポイントを提供し、このエンドポイントは公開されたサーブレットにバインドされます。
ServletListener	該当なし	**camel-servletlistener**	Web アプリケーションで Camel アプリケーションのブートストラップに使用されます。
SFTP	**sftp://[*Username*@]*Hostname*[:*Port*]/*Directoryname*[?*Options*]**	**camel-ftp**	SFTP でのファイルの送受信
sip	**sip://User@*Hostname*[:*Port*][?*Options*]** **sips://User@*Hostname*[:*Port*][?*Options*]**	**camel-sip**	har SIP プロトコルを使用して通信機能をパブリッシュ/サブスクライブします。RFC3903 - Session Initiation Protocol (SIP) Extension for Event
SJMS	**sjms:[queue:|topic:]*destinationName*[?*Options*]**	**camel-sjms**	JMS クライアントの作成および設定にベストプラクティスを使用する Camel 用の JMS クライアント。
SJMS-Batch	**sjms-batch:[queue:]*destinationName*[?*Options*]**	**camel-sjms**	JMS キューからの非常に高性能でトランザクションバッチ消費のための特殊なコンポーネント。
Slack	slack:#*Channel*[?*Options*] slack:@*Username*[?*Options*]	**camel-slack**	Slack のインスタンスに接続し、事前確立済みの Slack 受信 Webhook を介してメッセージボディーに含まれるメッセージを配信できます。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
SMPP	**smpp://*UserInfo@Host*[:*Port*][?*Options*]**	**camel-smpp**	JSMPP ライブラリー を使用して Short Messaging Service Center を使用して SMS を送受信するには、以下を行います。
SMTP	**smtp://[*UserName@*]*Host*[:*Port*][?*Options*]**	**camel-mail**	SMTP および JavaMail を使用した電子メールの送信。
SNMP	**snmp://*Hostname*[:*Port*][?*Options*]**	**camel-snmp**	SNMP 対応デバイスをポーリングしたり、トラップを受信したりできます。
Solr	**solr://*Hostname*[:*Port*]/*Solr*[?*Options*]**	**camel-solr**	Solrj クライアント API を使用して、Apache Lucene Solr サーバーと対話します。
Splunk	**splunk://*Endpoint*[?*Options*]**	**camel-splunk**	イベントをパブリッシュし、Splunk でイベントを検索できます。
Spring Batch	**spring-batch:*Job*[?*Options*]**	**camel-spring-batch**	Camel と Spring Batch のブリッジングを行います。
Spring イベント	**spring-event://dummy**	**camel-spring**	Spring コンテキストの Spring ApplicationEvents オブジェクトをパブリッシュまたは消費します。
Spring の統合	**spring-integration:*DefaultChannelName*[?*Options*]**	**camel-spring-integration**	Camel および Spring Integration のブリッジコンポーネント。
Spring LDAP	**spring-ldap:*SpringLdapTemplate*[?*Options*]**	**camel-spring-ldap**	Spring LDAP の Camel ラッパーを提供します。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Spring Redis	**spring-redis://*Hostname*[:*Port*][?*Options*]**	**camel-spring-redis**	Redis からのメッセージの送受信を有効にします。これは高度なキー値ストアで、キーには文字列、ハッシュ、リスト、セット、およびソートされたセットを含めることができます。
Spring Web Services	**spring-ws:[*MappingType*:]*Address*[?*Options*]**	**camel-spring-ws**	Web サービスにアクセスするためのクライアント側のサポート、および Spring Web サービスを使用して独自のコントラクトファースト Web サービスを作成するためのサーバー側のサポート。
SQL	**sql:*SqlQueryString*[?*Options*]**	**camel-sql**	JDBC を使用した SQL クエリーの実行。
SQL ストアドプロシージャ	**sql-stored:*Template*[?*Options*]**	**camel-sql**	Stored Procedure クエリーを使用してデータベースを操作する。
SSH	**ssh:[*Username*[:*Password*]@]*Host*[:*Port*][?*Options*]**	**camel-ssh**	SSH サーバーにコマンドを送信する場合。
StAX	**stax:*ContentHandlerClassName***	**camel-stax**	SAX ContentHandler を介してメッセージを処理します。
STOMP	**stomp:queue:*Destination*[?*Options*]**	**camel-stomp**	Apache ActiveMQ などの Stomp 準拠のブローカーとの間でメッセージを送受信する場合。
ストリーム	**stream:[in|out|err|header][?*Options*]**	**camel-stream**	Unix パイプではなく、input/output/error/file ストリームへの読み取りまたは書き込み。
string Template	**string-template:*TemplateURI*[?*Options*]**	**camel-stringtemplate**	文字列テンプレートを使用して応答を生成します。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Stub	**stub:*SomeOtherCamelUri***	**camel-core**	テストやデバッグを容易にするために、一部の物理ミドルウェアエンドポイントをスタブアウトすることができます。
Swagger	該当なし	**camel-swagger**	CamelContext ファイルで、REST 定義のルートおよびエンドポイントの API ドキュメントを作成できます。
Swagger	該当なし	**camel-swagger**	CamelContext ファイルで REST 定義のルートまたはエンドポイントの API ドキュメントを作成します。
Swagger Java	該当なし	**camel-swagger-java**	REST DSL と統合し、Swagger を使用して REST サービスとその API を公開します。このコンポーネントは、サーブレットとして使用したり、サーブレットなしで REST コンポーネントから直接使用することもできます。
Test	**test:*RouterEndpointUri***	**camel-spring**	指定の基礎となるエンドポイントからポーリングできるすべてのメッセージボディーを受信することを期待する Mock エンドポイントを作成します。
Timer	**timer:*EndpointID*[?*Options*]**	**camel-core**	タイマーエンドポイント。
Twitter	**twitter://[*Endpoint*][?*Options*]**	**camel-twitter**	Twitter エンドポイント。
Undertow	**undertow:http://*Hostname*[:*Port*][/*ResourceUri*][?*Options*]**	**camel-undertow**	HTTP 要求の使用および生成用の HTTP ベースのエンドポイントを提供します。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
検証	**validator:*LocalOrRemoteResource***	**camel-spring**	XML スキーマと JAXP Validation を使用してメッセージのペイロードを検証します。
velocity	**velocity:*TemplateURI*[?*Options*]**	**camel-velocity**	Apache Velocity テンプレートを使用して応答を生成します。
Vertx	**vertx:*ChannelName*[?*Options*]**	**camel-vertx**	Vertx イベントバスの使用
VM	**vm:*EndpointID***	**camel-core**	java.util.concurrent.BlockingQueue にメッセージを配信するために使用されます。これは、同じ JVM 内で SEDA スタイルの処理パイプラインを作成する場合に役立ちます。
video	**weather://*DummyName*[?*Options*]**	**camel-weather**	Open Weather Map からの投票情報をポーリングします。これは、無料のグローバル情報と予測情報を提供するサイトです。
Websocket	**websocket://*Hostname*[:*Port*]/*Path***	**camel-websocket**	Websocket クライアントとの通信。
XML RPC	**xmlrpc://*ServerURI*[?*Options*]**	**camel-xmlrpc**	XML のデータ形式を提供します。これにより、Apache XmlRpc のバインドデータ形式を使用した要求メッセージおよび応答メッセージのシリアライズおよびデシリアライズが可能になります。
XML セキュリティー	該当なし	**camel-xmlsecurity**	W3C 標準の XML 署名構文および 処理で説明されているように、XML 署名を生成および 検証します。
XMPP	**xmpp:*Hostname*[:*Port*][/*Room*]**	**camel-xmpp**	XMPP および Jabber の使用。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
XQuery	**xquery:*TemplateURI***	**camel-saxon**	XQuery テンプレートを使用して応答を生成します。
XSLT	**xslt:*TemplateURI*[?*Options*]**	**camel-spring**	XSLT テンプレートを使用してメッセージを処理できます。
Yammer	**yammer:[*function*][?*Options*]**	**camel-yammer**	Yammer エンタープライズソーシャルネットワークと対話できます。
ZooKeeper	**zookeeper://*Hostname*[:*Port*]/*Path***	**camel-zookeeper**	ZooKeeper クラスターの使用

1.2. JBOSS EAP の CAMEL コンポーネントのリスト

コンポーネントの表

以下の Camel コンポーネントは、Red Hat JBoss Enterprise Application Platform (Java EE)コンテナーでサポートされます。

表1.2 Apache Camel のコンポーネント

コンポーネント	エンドポイント URI	アーティファクト ID	説明
ActiveMQ	**activemq:[queue:|topic:]*DestinationName***	**activemq-core**	Apache ActiveMQ を使用した JMS メッセージングの場合。
Atom	**atom://*AtomUri*[?*Options*]**	**camel-atom**	atom フィードの使用など、atom 統合での Apache Abdera の使用
Bean	**bean:*BeanID*[?methodName=*Method*]**	**camel-core**	Bean バインディングを使用して、メッセージエクスチェンジをレジストリーの Bean にバインドします。は POJO (Plain Old Java Objects)の公開および呼び出しにも使用されます。
Bindy	該当なし	**camel-bindy**	非構造化データの解析とバインドを有効にします。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
参照	**browse:** *名前*	**camel-core**	テスト、視覚化ツール、またはデバッグに役立つ簡単な **BrowsableEndpoint** を提供します。エンドポイントに送信されたエクスチェンジはすべて参照できます。
		camel-castor	**camel-castor** コンポーネントは、Castor XML ライブラリーを使用して XML ペイロードを Java オブジェクトにアンマーシャリングしたり、Java オブジェクトを XML ペイロードにマーシャリングしたりするデータ形式の使用をサポートします。このコンポーネントはエンドポイントファクトリーではありません。詳細は、Apache のドキュメント を参照してください。
CDI	該当なし	**camel-cdi**	CDI 統合を提供します。
クラス	**class:***ClassName***[?method=***MethodName***]**	**camel-core**	Bean バインディングを使用して、メッセージエクスチェンジをレジストリーの Bean にバインドします。は POJO の公開および呼び出しにも使用されます(Plain Old Java Objects)。
ControlBus	**controlbus:***Command***[?***Options***]**	**camel-core**	Camel アプリケーションの管理および監視用にエンドポイントにメッセージを送信できるようにする ControlBus **Enterprise Integration Pattern**。
crypto	**crypto:sign:***Name***[?***Options***]** **crypto:verify:***Name***[?***Options***]**	**camel-crypto**	Java Cryptographic Extension の Signature Service を使用してエクスチェンジに署名し、検証します。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
		camel-csv	**camel-csv** コンポーネントは、メッセージデータでコンマ区切りの値 (CSV)を使用するサポートを提供します。このコンポーネントはエンドポイントファクトリーではありません。詳細は、Apache のドキュメント を参照してください。
CXF	**cxf://*Address*[?*Options*]**	**camel-cxf**	Web サービスの統合のための Apache CXF の使用
CXF Bean	**cxf:*BeanName***	**camel-cxf**	レジストリーから JAX WS または JAX RS アノテーションが付けられた Bean を使用してエクスチェンジを提案します。
CXFRS	**cxfrs:bean:*RsEndpoint*[?*Options*]**	**camel-cxf**	CXF でホストされる JAX-RS サービスに接続するための Apache CXF との統合を提供します。
DataFormat	**dataformat:*Name*:(marshal|unmarshal)[?*Options*]**	**camel-core**	エンドポイントにメッセージを送信して、標準の Camel データフォーマットのいずれかでメッセージをマーシャリングまたはアンマーシャリングできます。
DataSet	**dataset:*Name*[?*Options*]**	**camel-core**	ロードおよび soak テストの場合、DataSet はコンポーネントに送信するための大量のメッセージを作成したり、適切に消費されることをアサートしたりする方法を提供します。
Direct	**direct:*EndpointID*[?*Options*]**	**camel-core**	同じ CamelContext からの別のエンドポイントへの同期呼び出し（シングルスレッド）。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Direct-VM	**direct-vm:*EndpointID*[?*Options*]**	**camel-core**	同じ JVM で実行されている別の CamelContext の別のエンドポイントへの同期呼び出し（シングルスレッド）。
Dozer	**dozer:*EndpointID*[?*Options*]**	**camel-dozer**	Dozer マッピングフレームワークを使用して Java Bean 間のマッピング機能を提供します。
EJB	**ejb:*EjbName*[?method=*MethodName*]**	**camel-ejb**	Bean バインディングを使用して、メッセージエクスチェンジを EJB にバインドします。これは Bean コンポーネントのように動作しますが、EJB にアクセスする場合にのみ機能します。EJB 3.0 以降をサポートします。
ElasticSearch	**elasticsearch:*ClusterName***	**camel-elasticsearch**	ElasticSearch サーバーと対話する場合。
File2	**file://*DirectoryName*[?*Options*]**	**camel-core**	ファイルへのメッセージを送信するか、ファイルまたはディレクトリーをポーリングします。
flatpack	**flatpack:[fixed|delim]:***ConfigFile*	**camel-flatpack**	FlatPack ライブラリーを使用した固定幅または区切られたファイルまたはメッセージの処理
FTP2	**ftp://[*Username*@]*Hostname*[:*Port*]/*Directoryname*[?*Options*]**	**camel-ftp**	FTP でのファイルの送受信
		camel-groovy	**camel-groovy** コンポーネントは、Groovy 言語の使用をサポートします。これはエンドポイントファクトリーではありません。Groovy を参照してください。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
HL7		**camel-hl7**	HL7 MLLP プロトコルおよび HL7 モデルを使用する場合は、HAPI ライブラリー を使用します。
HTTP4	**http4://*Hostname*[:*Port*][/*ResourceUri*]**	**camel-http4**	Apache HTTP Client 4.x を使用して外部 HTTP サーバーを呼び出す場合。
IMAP	**imap://[*UserName@*]*Host*[:*Port*][?*Options*]**	**camel-mail**	IMap を使用して電子メールを受信する。
		camel-jackson	**camel-jackson** コンポーネントは、Jackson を Camel レジストリーで型コンバーターとして統合するためのサポートを提供します。このコンポーネントはエンドポイントファクトリーではありません。camel-jackson を参照してください。
jasypt		**camel-jasypt**	Jasypt と統合し、プロパティー ファイルの機密情報を暗号化できるようにします。
JAXB		**camel-jaxb**	JAXB2 XML マーシャリング標準を使用して XML ペイロードを Java オブジェクトにアンマーシャリングしたり、Java オブジェクトを XML ペイロードにマーシャリングしたりするデータ形式。
JGroups	**jgroups:*ClusterName*[?*Options*]**	**camel-jgroups**	JGroups クラスターでメッセージを交換します。
JMS	**jms:[temp:][queue:|topic:]*DestinationName*[?*Options*]**	**camel-jms**	JMS プロバイダーの使用

コンポーネント	エンドポイント URI	アーティファクト ID	説明
JMX	**jmx://*Platform*[?*Options*]**	**camel-jmx**	JMX 通知リスナーの使用
JPA	**jpa:[*EntityClassName*][?*Options*]**	**camel-jpa**	OpenJPA、Hibernate、または TopLink を使用するために JPA 仕様を介してデータベースをキューとして使用する場合。
Kafka	**kafka://*Hostname*[:*Port*][?*Options*]**	**camel-kafka**	Apache Kafka メッセージブローカーからメッセージを送受信します。
言語	**language://*LanguageName*[:*Script*][?*Options*]**	**camel-core**	言語スクリプトを実行します。
リスト	**list:*ListID***	**camel-core**	テスト、視覚化ツール、またはデバッグに役立つ簡単な BrowsableEndpoint を提供します。エンドポイントに送信されたエクスチェンジはすべて参照できます。
Log	**log:*LoggingCategory*[?level=*LoggingLevel*]**	**camel-core**	Jakarta Commons Logging を使用して、log4j などの基礎となるロギングシステムにメッセージ交換をログに記録します。
Lucene	**lucene:*SearcherName*:insert[?analyzer=*Analyzer*]** **lucene:*SearcherName*:query[?analyzer=*Analyzer*]**	**camel-lucene**	高度な分析/トークン化機能を使用して、Apache Lucene を使用して Java ベースのインデックスと完全なテキストベースの検索を実行します。
MINA2	**mina2:tcp://*Hostname*[:*Port*][?*Options*]** **mina2:udp://*Hostname*[:*Port*][?*Options*]** **mina2:vm://*Hostname*[:*Port*][?*Options*]**	**camel-mina2**	Apache MINA 2.x の使用

コンポーネント	エンドポイント URI	アーティファクト ID	説明
MQTT	**mqtt:*Name***	**camel-mqtt**	MQTT M2M メッセージブローカーと通信するためのコンポーネント
MVEL	**mvel:*TemplateName*[?*Options*]**	**camel-mvel**	MVEL テンプレートを使用してメッセージを処理できます。
Netty4	**netty4:tcp://localhost:99999[?*Options*] netty4:udp://*Remotehost*:99999/[?*Options*]**	**camel-netty4**	Netty バージョン 4.x によって提供される Java NIO ベースの機能を使用して、TCP プロトコルおよび UDP プロトコルと連携できます。
OGNL		**camel-ognl**	OGNL は、Java オブジェクトのプロパティーを取得および設定するための式言語です。
POP	**pop3://[*UserName*@]*Host*[:*Port*][?*Options*]**	**camel-mail**	POP3 および JavaMail を使用して電子メールを受信します。
プロパティー	**properties://*Key*[?*Options*]**	**camel-properties**	エンドポイント URI 定義で直接プロパティープレースホルダーを使用することを容易にします。
		camel-protobuf	このコンポーネントは、Java と Protocol Buffer プロトコルのシリアライズをサポートします。プロトコルバッファーは言語に依存しないプラットフォームに依存しないため、Camel ルートによって生成されたメッセージは他の言語実装によって消費される可能性があります。詳細は、Apache のドキュメント を参照してください。
Quartz2	**quartz2://[*GroupName*/]*TimerName*[?*Options*] quartz2://*GroupName*/*TimerName*/*CronExpression***	**camel-quartz2**	Quartz スケジューラー 2.x を使用したスケジュールされたメッセージの配信を提供します。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
Ref	**ref:*EndpointID***	**camel-core**	レジストリーにバインドされる既存エンドポイントを検索するためのコンポーネント。
REST	**rest://*Method*:*Path*[:*UriTemplate*][?*Options*]**	**camel-rest**	Apache Camel Development Guide の Defining Services with REST DSL セクションを使用して REST エンドポイントを定義できます。
Restlet	**restlet:*RestletUrl*[?*Options*]**	**camel-restlet**	Restlet を使用して Restful リソースを消費および生成するコンポーネント。
RSS	**rss:*Uri***	**camel-rss**	RSS フィードの使用など、RSS 統合で ROME と連携します。
Salesforce	**salesforce:*Topic*[?*Options*]**	**camel-salesforce**	プロデューサーおよびコンシューマーエンドポイントが Java DTO を使用して Salesforce と通信できるようにします。
SAP	**sap:[destination:*DestinationName*|server:*ServerName*]*rfcName*[?*Options*]**	**camel-sap**	同期リモート関数呼び出し sRFC を使用して、SAP システムへの送受信通信を有効にします。
Saxon		**camel-saxon**	Saxon コンポーネントは XQuery をサポートし、Java DSL または XML DSL で式または述語を使用できるようにします。
スクリプト		**camel-script**	Script コンポーネントは、JSR 223 標準に従って式または述語を作成するために使用できるスクリプト言語を複数サポートします。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
SEDA	**seda:*EndpointID***	**camel-core**	java.util.concurrent.BlockingQueue にメッセージを配信するために使用されます。これは、同じ CamelContext 内で SEDA スタイルの処理パイプラインを作成する場合に役立ちます。
SERVLET	**servlet://*RelativePath*[?*Options*]**	**camel-servlet**	HTTP エンドポイントに到達する HTTP リクエストを消費するための HTTP ベースのエンドポイントを提供し、このエンドポイントは公開されたサーブレットにバインドされます。
SFTP	**sftp://[*Username*@]*Hostname*[:*Port*]/*Directoryname*[?*Options*]**	**camel-ftp**	SFTP でのファイルの送受信
SMPP	**smpp://*UserInfo*@*Host*[:*Port*][?*Options*]**	**camel-smpp**	JSMPP ライブラリー を使用して Short Messaging Service Center を使用して SMS を送受信するには、以下を行います。
		camel-soap	**camel-soap** コンポーネントは、SOAP データフォーマットの使用をサポートします。これはエンドポイントファクトリーではありません。詳細は、Apache のドキュメント を参照してください。
		camel-spring	**camel-spring** コンポーネントは、Spring Expression Language (SpEL)の使用のサポートを提供します。これはエンドポイントファクトリーではありません。SpEL を参照してください。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
		camel-spring-security	**camel-spring-security** コンポーネントは、Camel ルートのロールベースの承認を提供します。これはエンドポイントファクトリーではありません。詳細は、Apache のドキュメント を参照してください。
SQL	**sql:*SqlQueryString*[?*Options*]**	**camel-sql**	JDBC を使用した SQL クエリーの実行。
ストリーム	**stream:[in|out|err|header][?*Options*]**	**camel-stream**	Unix パイプではなく、input/output/error/file ストリームへの読み取りまたは書き込み。
Swagger	該当なし	**camel-swagger**	CamelContext ファイルで、REST 定義のルートおよびエンドポイントの API ドキュメントを作成できます。
		camel-tagsoup	このコンポーネントは、HTML の解析と整形された HTML を返すためのサポートを提供します。このコンポーネントはエンドポイントファクトリーではありません。詳細は、Apache のドキュメント を参照してください。
Timer	**timer:*EndpointID*[?*Options*]**	**camel-core**	タイマーエンドポイント。
velocity	**velocity:*TemplateURI*[?*Options*]**	**camel-velocity**	Apache Velocity テンプレートを使用して応答を生成します。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
VM	**vm:*EndpointID***	**camel-core**	java.util.concurrent.BlockingQueue にメッセージを配信するために使用されます。これは、同じ JVM 内で SEDA スタイルの処理パイプラインを作成する場合に役立ちます。
video	**weather://*DummyName*[?*Options*]**	**camel-weather**	Open Weather Map からの投票情報をポーリングします。これは、無料のグローバル情報と予測情報を提供するサイトです。
		camel-xmlbeans	このコンポーネントは、XML ペイロードを Java オブジェクトにアンマーシャリングし、Java オブジェクトを XML ペイロードにマーシャリングするためのサポートを提供します。このコンポーネントはファクトリーエンドポイントではありません。詳細は、Apache のドキュメント を参照してください。
XML セキュリティー	該当なし	**camel-xmlsecurity**	W3C 標準の XML 署名構文および 処理で説明されているように、XML 署名を生成および 検証します。
XQuery	**xquery:*TemplateURI***	**camel-saxon**	XQuery テンプレートを使用して応答を生成します。
XSLT	**xslt:*TemplateURI*[?*Options*]**	**camel-spring**	XSLT テンプレートを使用してメッセージを処理できます。

コンポーネント	エンドポイント URI	アーティファクト ID	説明
XStream		**camel-xstream**	XStream データフォーマットを提供します。これは、XStream ライブラリーを使用して Java オブジェクトを XML との間でマーシャリングおよびアンマーシャリングします。
		camel-zipfile	このコンポーネントは、zip ファイルにメッセージを圧縮し、zip ファイルを元のメッセージに展開するためのサポートを提供します。このコンポーネントはエンドポイントファクトリーではありません。詳細は、Apache のドキュメント を参照してください。
ZooKeeper	**zookeeper://*Hostname*[:*Port*]/*Path***	**camel-zookeeper**	ZooKeeper クラスターの使用

1.3. JBOSS EAP でのカスタム CAMEL コンポーネントのデプロイ

標準の Camel on EAP コンポーネントの他に、JBoss EAP にデプロイする独自のカスタムコンポーネントを追加することもできます。ここでは、JBoss EAP コンテナーに Camel コンポーネントを追加する方法を説明します。

1.3.1. module.xml 定義の追加

module.xml 記述子ファイルは、コンポーネントのクラ出力ディング動作を定義します。

追加のディレクトリーを作成し、**module.xml** ファイルと任意の jar 依存関係を追加できます。たとえば、**modules/system/layers/fuse/org/apache/camel/component/ftp/main** ディレクトリーなどです。

以下は **camel-ftp** コンポーネントの例になります。

```
<module xmlns="urn:jboss:module:1.1" name="org.apache.camel.component.ftp">
  <resources>
    <resource-root path="camel-ftp-2.14.0.jar" />
  </resources>
  <dependencies>
    <module name="com.jcraft.jsch" />
    <module name="javax.xml.bind.api" />
    <module name="org.apache.camel.core" />
    <module name="org.apache.commons.net" />
  </dependencies>
</module>
```

–

1.3.2. 参照の追加

新しいモジュールが Camel デプロイメントに表示されるようにするには、**modules/system/layers/fuse/org/apache/camel/component/main/module.xml** ファイル内にそのモジュールへの参照を追加します。

```
<module xmlns="urn:jboss:module:1.3" name="org.apache.camel.component">
  <dependencies>
    ...
    <module name="org.apache.camel.component.ftp" export="true" services="export"/>
  </dependencies>

</module>
```

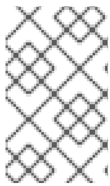
第2章 ACTIVEMQ

ACTIVEMQ コンポーネント

ActiveMQ コンポーネントを使用すると、JMS Queue または Topic にメッセージを送信したり、Apache ActiveMQ を使用して JMS Queue または Topic からメッセージを消費したりできます。

このコンポーネントは JMS コンポーネントをベースとしており、送信に Spring の **JmsTemplate** を使用し、**MessageListenerContainer** を使用して宣言的トランザクションに Spring の JMS サポートを使用します。JMS コンポーネントのすべてのオプションは、このコンポーネントにも適用されます。

このコンポーネントを使用するには、クラスパスに **camel-core.jar**、**camel-spring.jar**、**camel-jms.jar** などの Apache Camel 依存関係とともに **activemq.jar** または **activemq-core.jar** があることを確認してください。



トランザクションおよびキャッシュ

パフォーマンスに影響する可能 **性があるため JMS でトランザクションを使用している場合は、JMS ページの以下のトランザクションレベルとキャッシュ** レベルを参照してください。

URI 形式

```
activemq:[queue:|topic:]destinationName
```

destinationName は ActiveMQ キューまたはトピック名です。デフォルトでは、**destinationName** はキュー名として解釈されます。たとえば、**FOO.BAR** キューに接続するには、以下を使用します。

```
activemq:FOO.BAR
```

必要に応じて、オプションの **queue:** 接頭辞を含めることができます。

```
activemq:queue:FOO.BAR
```

トピックに接続するには、**topic:** 接頭辞を含める必要があります。たとえば、**Stocks.Prices** トピックに接続するには、以下を使用します。

```
activemq:topic:Stocks.Prices
```

オプション

これらのオプションはすべてこのコンポーネントに適用されるため、JMS コンポーネントのオプションを参照してください。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

ActiveMQ Camel コンポーネントを設定して、組み込みブローカーまたは外部ブローカーのいずれかと連携できます。JBoss EAP コンテナーにブローカーを埋め込むには、EAP コンテナー設定ファイルに ActiveMQ リソースアダプターを設定します。詳細は、JBoss Active MQ リソースアダプターのインストール を参照してください。

接続ファクトリーの設定

以下の テストケース は、ActiveMQ への接続に使用される brokerURL を指定する一方で、**activeMQComponent ()** メソッド を使用して ActiveMQComponent を CamelContext に追加する方法を示しています。

```
camelContext.addComponent("activemq", activeMQComponent("vm://localhost?
broker.persistent=false"));
```

SPRING XML を使用した接続ファクトリーの設定

ActiveMQComponent で以下のように ActiveMQ ブローカー URL を設定できます。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <camelContext xmlns="http://camel.apache.org/schema/spring">
  </camelContext>


  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="tcp://somehost:61616"/>
  </bean>

</beans>
```

接続プールの使用

Camel を使用して ActiveMQ ブローカーに送信する場合は、プールされた接続ファクトリーを使用して JMS 接続、セッション、およびプロデューサーの効率的なプーリングを処理することが推奨されます。これは、ActiveMQ Spring Support のページに記載されています。

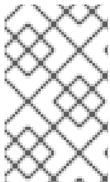
Maven で Jencks AMQ プールを使用できます。

```
    <dependency>
      <groupId>org.apache.activemq</groupId>
      <artifactId>activemq-pool</artifactId>
      <version>_activemq-version_</version>
    </dependency>
```

次に、**activemq** コンポーネントを以下のように設定します。

```
    <bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
```

```
        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>

    <bean id="pooledConnectionFactory"
class="org.apache.activemq.pool.PooledConnectionFactory" init-method="start" destroy-
method="stop">
        <property name="maxConnections" value="8" />
        <property name="connectionFactory" ref="jmsConnectionFactory" />
    </bean>


    <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
        <property name="connectionFactory" ref="pooledConnectionFactory"/>
        <property name="concurrentConsumers" value="10"/>
    </bean>

    <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
        <property name="configuration" ref="jmsConfig"/>
    </bean>
```



注記

プールされた接続ファクトリーの **init** メソッドおよび **destroy** メソッドに注意してください。これは、接続プールが適切に起動およびシャットダウンされるようにすることが重要です。

次に、**PooledConnectionFactory** は、同時に使用する最大 8 つの接続を持つ接続プールを作成します。各接続は、多くのセッションで共有できます。**maxActive** という名前のオプションを使用して、接続ごとのセッションの最大数を設定できます。デフォルト値は **500** です。**ActiveMQ 5.7** 以降では、オプションの名前が **maxActiveSessionPerConnection** という名前になるよう名前が変更されました。**concurrentConsumers** は **maxConnections** よりも高い値に設定されていることに注意してください。これは、各コンシューマーがセッションを使用し、セッションが同じ接続を共有できるため、安全です。この例では、8 * 500 = 4000 のアクティブなセッションを同時に指定できます。

ルートでの MESSAGELISTENER POJO の呼び出し

ActiveMQ コンポーネントは、JMS MessageListener をプロセッサーに変換するヘルパー Type Converter も提供します。つまり、Bean コンポーネントは任意のルート内で JMS MessageListener Bean を直接呼び出すことができます。

たとえば、以下のように JMS で MessageListener を作成できます。

```
public class MyListener implements MessageListener {
    public void onMessage(Message jmsMessage) {
        // ...
    }
}
```

次に、以下のようにルートでこれを使用します。

```
from("file://foo/bar").
  bean(MyListener.class);
```

つまり、Apache Camel コンポーネント を再利用し、JMS **MessageListener** POJO に簡単に統合できます。

ACTIVEMQ 宛先オプションの使用

ActiveMQ 5.6 で利用可能

destination. 接頭辞を使用して、エンドポイント URI で Destination Options を設定できます。たとえば、コンシューマーを排他的としてマークし、その prefetch サイズを 50 に設定するには、次のように実行できます。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://src/test/data?noop=true"/>
    <to uri="activemq:queue:foo"/>
  </route>
  <route>
    <!-- use consumer.exclusive ActiveMQ destination option, notice we have to prefix with destination.
-->
    <from uri="activemq:foo?
destination.consumer.exclusive=true&estination.consumer.prefetchSize=50"/>
    <to uri="mock:results"/>
  </route>
</camelContext>
```

アドバイザリーメッセージの消費

ActiveMQ は、消費できるトピックに含まれる Advisory メッセージ を生成できます。このようなメッセージは、低速なコンシューマーを検出したり、統計値(1 日あたりのメッセージ/生成の数など)を構築する場合にアラートを送信するのに役立ちます。 以下の Spring DSL の例は、トピックからメッセージを読み取る方法を示しています。

```
<route>
 <from uri="activemq:topic:ActiveMQ.Advisory.Connection?mapJmsMessage=false" />
 <convertBodyTo type="java.lang.String"/>
 <transform>
      <simple>${in.body}&#13;</simple>
 </transform>
 <to uri="file://data/activemq/?fileExist=Append&fileName=advisoryConnection-
${date:now:yyyyMMdd}.txt" />
</route>
```

キューでメッセージを消費すると、data/activemq フォルダーの下に以下のファイルが表示されるはずです。

advisoryConnection-20100312.txt advisoryProducer-20100312.txt

文字列を含む :

```
      ActiveMQMessage {commandId = 0, responseRequired = false, messageId = ID:dell-charles-
3258-1268399815140
      -1:0:0:0:221, originalDestination = null, originalTransactionId = null, producerId = ID:dell-charles-
      3258-1268399815140-1:0:0:0, destination = topic://ActiveMQ.Advisory.Connection, transactionId
= null,
```

```
    expiration = 0, timestamp = 0, arrival = 0, brokerInTime = 1268403383468, brokerOutTime =
1268403383468,
    correlationId = null, replyTo = null, persistent = false, type = Advisory, priority = 0, groupID = null,
    groupSequence = 0, targetConsumerId = null, compressed = false, userID = null, content = null,
    marshalledProperties = org.apache.activemq.util.ByteSequence@17e2705, dataStructure =
ConnectionInfo
    {commandId = 1, responseRequired = true, connectionId = ID:dell-charles-3258-1268399815140-
2:50,
    clientId = ID:dell-charles-3258-1268399815140-14:0, userName = , password = *****,
    brokerPath = null, brokerMasterConnector = false, manageable = true, clientMaster = true},
    redeliveryCounter = 0, size = 0, properties = {originBrokerName=master, originBrokerId=ID:dell-
charles-
    3258-1268399815140-0:0, originBrokerURL=vm://master}, readOnlyProperties = true,
readOnlyBody = true,
    droppable = false}
```

コンポーネント JAR の取得

この依存関係が必要です。

- **activemq-camel**

ActiveMQ は、ActiveMQ プロジェクト でリリースされた JMS コンポーネントの拡張機能です。

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
  <version>_activemq-version_</version>
</dependency>
```

第3章 AHC

ASYNC HTTP CLIENT (AHC)コンポーネント

Camel 2.8 から利用可能

ahc: コンポーネントは、(HTTP を使用して外部サーバーを呼び出すクライアントとして)外部 HTTP リソースを消費する HTTP ベースの エンドポイント を提供します。コンポーネントは Async Http Client ライブラリーを使用します。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-ahc</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
ahc:http://hostname[:port][/resourceUri][?options]
ahc:https://hostname[:port][/resourceUri][?options]
```

デフォルトでは、HTTP にポート 80 を使用し、HTTPS には 443 を使用します。

URI にクエリーオプションは **?option=value&option=value&.. の形式で追加できます。**

AHCENDPOINT オプション

名前	デフォルト値	説明
throwExceptionOnFailure	**true**	リモートサーバーからの応答に失敗した場合に **AhcOperationFailedException** を出力することを無効にするオプション。これにより、HTTP ステータスコードに関係なくすべての応答を取得できます。
bridgeEndpoint	**false**	オプションが true の場合、Exchange.HTTP_URI ヘッダーは無視され、リクエストにエンドポイントの URI を使用します。また、**throwExcpetionOnFailure** を false に設定して、AhcProducer がすべての障害応答を返信するようにすることもできます。

transferException	**false**	有効であり、エクスチェンジ がコンシューマー側で処理に失敗した場合、発生した **例外** が **application/x-java-serialized-object** コンテンツタイプとして応答でシリアライズされた場合は (Jetty または Servlet Camel コンポーネントを使用するなど)。プロデューサー側では、例外がデシリアライズされ、**AhcOperationFailedException** ではなくそのまま出力されます。原因となった例外はシリアライズする必要があります。
クライアント	**null**	カスタム **com.ning.http.client.AsyncHttpClient** を使用するには、以下を行います。
clientConfig	**null**	**AsyncHttpClient** がカスタム **com.ning.http.client.AsyncHttpClientConfig** を使用するように設定します。
clientConfig.x	**null**	エンドポイントによって使用される **com.ning.http.client.AsyncHttpClientConfig** インスタンスの追加プロパティーを設定します。このパラメーターを使用して設定された設定オプションは、**clientConfig** パラメーターまたはこのパラメーターを使用して設定されたプロパティーを持つコンポーネントレベルで設定されたインスタンスとマージされます。
clientConfig.realm.x	**null**	Camel 2.11: **com.ning.http.client.AsyncHttpClientConfig** のレルムプロパティーを設定するには、**com.ning.http.client.Realm.RealmBuilder** のオプションを使用できます。たとえば、スキームを設定するには **clientConfig.realm.scheme=DIGEST を設定できます。**

binding	**null**	カスタムの **org.apache.camel.component.ahc.AhcBinding** を使用します。
sslContextParameters	**null**	**Camel 2.9:** CAMEL:Registry の **org.apache.camel.util.jsse.SSLContextParameters** オブジェクトへの参照。この参照は、コンポーネントレベルで設定済みの SSLContextParameters を上書きします。 Security Guide および Using the JSSE Configuration Utility の Configuring Transport Security for Camel Components の章 を参照してください。 このオプションを設定すると、エンドポイントまたはコンポーネントレベルで clientConfig オプションを介して提供される SSL/TLS 設定オプションがすべて上書きされることに注意してください。
bufferSize	**4096**	**Camel 2.10.3:** Camel と AHC クライアント間でデータを転送する際に使用される初期インメモリーバッファーサイズ。

AHCCOMPONENT オプション

名前	デフォルト値	説明
クライアント	**null**	カスタム **com.ning.http.client.AsyncHttpClient** を使用するには、以下を行います。
clientConfig	**null**	**AsyncHttpClient**s を設定するには、カスタムの **com.ning.http.client.AsyncHttpClientConfig** を使用します。
binding	**null**	カスタムの **org.apache.camel.component.ahc.AhcBinding** を使用します。

sslContextParameters	null	Camel 2.9: コンポーネントレベルでカスタム SSL/TLS 設定オプションを設定するには、以下を実行します。 詳細 は、Security Guide の Configuring Transport Security for Camel Components および Using the JSSE Configuration Utility を参照してください。 このオプションを設定すると、エンドポイントまたはコンポーネントレベルで clientConfig オプションを介して提供される SSL/TLS 設定オプションがすべて上書きされることに注意してください。

AhcComponent にオプションのいずれかを設定すると、これらのオプションが作成されている **AhcEndpoint**に伝播されることに注意してください。ただし、**AhcEndpoint** はカスタムオプションを設定/上書きすることもできます。エンドポイントに設定されたオプションは常に **AhcComponent** のオプションよりも優先されます。

メッセージヘッダー

名前	タイプ	説明
Exchange.HTTP_URI	**文字列**	呼び出す URI。エンドポイントに設定された既存の URI を直接上書きします。
Exchange.HTTP_PATH	**文字列**	リクエスト URI のパス。ヘッダーは HTTP_URI でリクエスト URI を構築するために使用されます。パスが "/ で始まる場合、http プロデューサーは Exchange.HTTP_BASE_URI ヘッダーまたは **exchange.getFromEndpoint（）.getEndpointUri（）**;に基づいて相対パスの検索を試みます。
Exchange.HTTP_QUERY	**文字列**	URI パラメーター。エンドポイントで直接設定された既存の URI パラメーターを上書きします。
Exchange.HTTP_RESPONSE_CODE	**int**	外部サーバーからの HTTP 応答コード。OK の場合は 200 です。
Exchange.HTTP_CHARACTER_ENCODING	**文字列**	文字エンコーディング。

Exchange.CONTENT_TYPE	**文字列**	HTTP コンテンツタイプ。は IN メッセージと OUT メッセージの両方で設定され、**text/html** などのコンテンツタイプを提供します。
Exchange.CONTENT_ENCODING	**文字列**	HTTP コンテンツエンコーディング。は IN メッセージと OUT メッセージの両方で設定され、**gzip** などのコンテンツエンコーディングを提供します。

メッセージボディー

Camel は外部サーバーからの HTTP 応答を OUT ボディーに保存します。IN メッセージからのヘッダーはすべて OUT メッセージにコピーされ、ルーティング中にヘッダーが保持されます。さらに、Camel は HTTP 応答ヘッダーと OUT メッセージヘッダーを追加します。

レスポンスコード

Camel は HTTP 応答コードに従って処理されます。

- レスポンスコードは 100..299 の範囲にあり、Camel は応答の成功と見なします。
- 応答コードは 300..399 の範囲にあり、Camel はこれをリダイレクト応答として認識し、その情報と共に **AhcOperationFailedException** を出力します。
- 応答コードは 400+ で、Camel はこれを外部サーバー障害とみなし、その情報と共に **AhcOperationFailedException** を出力します。オプション **throwExceptionOnFailure** を **false** に設定すると、失敗した応答コードに対して **AhcOperationFailedException** が出力されないようになります。これにより、リモートサーバーから応答を取得できるようになります。

AHCOPERATIONFAILEDEXCEPTION

この例外には、以下の情報が含まれます。

- HTTP ステータスコード
- HTTP ステータス行 (ステータスコードのテキスト)
- サーバーがリダイレクトを返した場合は、場所をリダイレクトします
- 応答ボディー(**java.lang.String**)（サーバーがボディーを応答として提供）

GET または POST を使用した呼び出し

以下のアルゴリズムは、**GET** または **POST** HTTP メソッドのいずれかを使用する必要があるかどうかを判断するために使用されます：1。ヘッダーで提供されるメソッドを使用します。2.クエリー文字列がヘッダーで提供される場合は **GET**。3.エンドポイントがクエリー文字列で設定されている場合の **GET**。4.送信するデータがある場合は **POST** します(null ではありません)。5.それ以外の場合は **GET**。

呼び出す URI の設定

HTTP プロデューサーの URI を直接エンドポイント URI として設定できます。以下のルートでは、Camel は HTTP を使用して外部サーバー **oldhost** に呼び出します。

```
from("direct:start")
    .to("ahc:http://oldhost");
```

同等の Spring の例：

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="ahc:http://oldhost"/>
  </route>
</camelContext>
```

メッセージのキー **Exchange.HTTP_URI** でヘッダーを追加することで、HTTP エンドポイント URI を上書きできます。

```
from("direct:start")
    .setHeader(Exchange.HTTP_URI, constant("http://newhost"))
    .to("ahc:http://oldhost");
```

URI パラメーターの設定

ahc プロデューサーは、HTTP サーバーに送信される URI パラメーターをサポートします。URI パラメーターは、エンドポイント URI で直接設定することも、メッセージ上でキー **Exchange.HTTP_QUERY** を持つヘッダーとして設定できます。

```
from("direct:start")
    .to("ahc:http://oldhost?order=123&detail=short");
```

または、ヘッダーで提供されるオプション：

```
from("direct:start")
        .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
    .to("ahc:http://oldhost");
```

HTTP メソッド(GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE)を HTTP プロデューサーに設定する方法

HTTP コンポーネントは、メッセージヘッダーを設定して HTTP リクエストメソッドを設定する方法を提供します。以下に例を示します。

```
from("direct:start")
        .setHeader(Exchange.HTTP_METHOD, constant("POST"))
    .to("ahc:http://www.google.com")
        .to("mock:results");
```

同等の Spring の例：

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
```

```
  <route>
    <from uri="direct:start"/>
    <setHeader headerName="CamelHttpMethod">
        <constant>POST</constant>
    </setHeader>
    <to uri="ahc:http://www.google.com"/>
    <to uri="mock:results"/>
  </route>
</camelContext>
```

CHARSET の設定

POST を使用してデータを送信する場合は、**Exchange** プロパティーを使用して **charset** を設定できます。

```
exchange.setProperty(Exchange.CHARSET_NAME, "iso-8859-1");
```

エンドポイント URI からの URI パラメーター

この例では、完全な URI エンドポイントがあり、これは Web ブラウザーに入力した内容になります。当然ながら、複数の URI パラメーターは、Web ブラウザーと同じように **&** amp; 文字をセパレーターとして使用して設定できます。この場合、Camel は複雑ではありません。

```
// we query for Camel at the Google page
template.sendBody("ahc:http://www.google.com/search?q=Camel", null);
```

メッセージの URI パラメーター

```
Map headers = new HashMap();
headers.put(Exchange.HTTP_QUERY, "q=Camel&lr=lang_en");
// we query for Camel and English language at Google
template.sendBody("ahc:http://www.google.com/search", null, headers);
```

上記のヘッダー値では、先頭に **?** を付け **ず**、通常 **&** amp; 文字でパラメーターを分離することができることに注意してください。

レスポンスコードの取得

AHC コンポーネントから HTTP 応答コードを取得するには、**Exchange.HTTP_RESPONSE_CODE** で Out メッセージヘッダーから値を取得します。

```
Exchange exchange = template.send("ahc:http://www.google.com/search", new Processor() {
            public void process(Exchange exchange) throws Exception {
                exchange.getIn().setHeader(Exchange.HTTP_QUERY, constant("hl=en&q=activemq"));
            }
  });
  Message out = exchange.getOut();
  int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
```

ASYNCHTTPCLIENT の設定

AsyncHttpClient クライアントは **AsyncHttpClientConfig** を使用してクライアントを設定します。詳細は、Async Http Client のドキュメントを参照してください。

以下の例は、ビルダーを使用して **AhcComponent** で設定する **AsyncHttpClientConfig** を作成する方法を示しています。

```
// create a client config builder
AsyncHttpClientConfig.Builder builder = new AsyncHttpClientConfig.Builder();
// use the builder to set the options we want, in this case we want to follow redirects and try
// at most 3 retries to send a request to the host
AsyncHttpClientConfig config = builder.setFollowRedirects(true).setMaxRequestRetry(3).build();

// lookup AhcComponent
AhcComponent component = context.getComponent("ahc", AhcComponent.class);
// and set our custom client config to be used
component.setClientConfig(config);
```

Camel **2.9** では、AHC コンポーネントは Async HTTP ライブラリー 1.6.4 を使用します。この新しいバージョンでは、プレーン Bean スタイルの設定のサポートが追加されました。**AsyncHttpClientConfig クラスは、 AsyncHttpClientConfig** で利用可能な設定オプションの getter および setter を提供します。**AsyncHttpClientConfigBean** のインスタンスは AHC コンポーネントに直接渡すか、**clientConfig** URI パラメーターを使用してエンドポイント URI で参照できます。

Camel **2.9** では、URI で直接設定オプションを設定することも可能です。clientConfig. で始まる URI パラメーターを使用すると、**AsyncHttpClientConfig** のさまざまな設定可能なプロパティーを設定できます。エンドポイント URI で指定されるプロパティーは、clientConfig URI パラメーターによって参照される設定で指定された値とマージされ、clientConfig. パラメーターが優先されるように設定されます。1 つのエンドポイントの設定に依存しないように、参照される **AsyncHttpClientConfig** インスタンスは常にエンドポイントごとにコピーされ、以前に作成されたエンドポイントの設定とは関係ありません。以下の例は、clientConfig. タイプの URI パラメーターを使用して AHC コンポーネントを設定する方法を示しています。

```
from("direct:start")
    .to("ahc:http://localhost:8080/foo?
clientConfig.maxRequestRetry=3&clientConfig.followRedirects=true")
```

関連項目

- Jetty
- HTTP
- HTTP4

第4章 AHC-WS

ASYNC HTTP CLIENT (AHC) WEBSOCKET クライアントコンポーネント

Camel 2.14 から利用可能

ahc-ws コンポーネントは、Websocket 経由で外部サーバーと通信するクライアント用の Websocket ベースの エンドポイント を提供します（外部サーバーへの Websocket 接続を開くクライアントとして）。コンポーネントは、Async Http Client ライブラリーを使用する 3章*AHC* コンポーネントを使用します。

Maven ユーザーは、このコンポーネントの以下の依存関係を **pom.xml** に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-ahc-ws</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
ahc-ws://hostname[:port][/resourceUri][?options]
ahc-wss://hostname[:port][/resourceUri][?options]
```

デフォルトでは、ahc-ws にはポート 80 を使用し、ahc-wss には 443 を使用します。

AHC-WS オプション

AHC-WS コンポーネントは AHC コンポーネントをベースとしているため、AHC コンポーネントのさまざまな設定オプションを使用できます。

WEBSOCKET でのデータの書き込みと読み取り

ahc-ws エンドポイントは、エンドポイントがプロデューサーまたはコンシューマーとしてそれぞれ設定されているかどうかに応じて、ソケットにデータを書き込むか、ソケットから読み取ることができます。

データの書き込みまたは読み取りのための URI の設定

以下のルートでは、Camel は指定された Websocket 接続に書き込みます。

```
from("direct:start")
    .to("ahc-ws://targethost");
```

同等の Spring の例：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
```

```
    <to uri="ahc-ws://targethost"/>
  </route>
</camelContext>
```

以下のルートでは、Camel は指定された WebSocket 接続から読み取ります。

```
from("ahc-ws://targethost")
     .to("direct:next");
```

同等の Spring の例：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="ahc-ws://targethost"/>
    <to uri="direct:next"/>
  </route>
</camelContext>
```

第5章 AMQP

AMQP

AMQP コンポーネントは、Qpid プロジェクトを介して AMQP プロトコル をサポートします。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-amqp</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
amqp:[queue:|topic:]destinationName[?options]
```

宛先名の後に、JMS コンポーネントのさまざまな設定オプションをすべて指定できます。

AMQP オプション

宛先名の後に、JMS コンポーネントのさまざまな設定オプションをすべて指定できます。

使用方法

AMQP コンポーネントは JMS コンポーネントから継承されるため、前者の使用は後者とほぼ同じになります。

```
// Consuming from AMQP queue
from("amqp:queue:incoming").
  to(...);

// Sending message to the AMQP topic
from(...).
  to("amqp:topic:notify");
```

AMQP コンポーネントの設定

Camel 2.16.1 以降では、**AMQPComponent#amqp10Component(String connectionURI)** ファクトリーメソッドを使用して、事前設定されたトピック接頭辞で AMQP 1.0 コンポーネントを返すこともできます。

```
 AMQPComponent amqp =
AMQPComponent.amqp10Component("amqp://guest:guest@localhost:5672");
```

Camel 2.17 以降、**AMQPComponent#amqp10Component(String connectionURI)** は **AMQPComponent#amqpComponent(String connectionURI)** の代わりに ファクトリーメソッドが非推奨になりました。

```
AMQPComponent amqp = AMQPComponent.amqpComponent("amqp://localhost:5672");

AMQPComponent authorizedAmqp = AMQPComponent.amqpComponent("amqp://localhost:5672",
"user", "password");
```

Camel 2.17 以降、AMQP コンポーネントを自動的に設定するために、**org.apache.camel.component.amqp.AMQPConnectionDetails** のインスタンスをレジストリーに追加することもできます。たとえば、Spring Boot の場合、Bean を定義する必要があります。

```
@Bean
AMQPConnectionDetails amqpConnection() {
  return new AMQPConnectionDetails("amqp://lcoalhost:5672");
}

@Bean
AMQPConnectionDetails securedAmqpConnection() {
  return new AMQPConnectionDetails("amqp://lcoalhost:5672", "username", "password");
}
```

Camel プロパティーに依存して AMQP コネクションの詳細を読み取ることもできます。ファクトリーメソッド **AMQPConnectionDetails.discoverAMQP()** は、以下のスニペットが示すように、Kubernetes のような慣例で Camel プロパティーの読み取りを試みます。

```
export AMQP_SERVICE_HOST = "mybroker.com"
export AMQP_SERVICE_PORT = "6666"
export AMQP_SERVICE_USERNAME = "username"
export AMQP_SERVICE_PASSWORD = "password"

...

@Bean
AMQPConnectionDetails amqpConnection() {
  return AMQPConnectionDetails.discoverAMQP();
}
```

トピックの使用

camel-amqp と連携するトピックを使用するには、以下のように **topic://** をトピック接頭辞として使用するようコンポーネントを設定する必要があります。

```
 <bean id="amqp" class="org.apache.camel.component.amqp.AmqpComponent">
        <property name="connectionFactory">
          <bean class="org.apache.qpid.amqp_1_0.jms.impl.ConnectionFactoryImpl" factory-
method="createFromURL">
            <constructor-arg index="0" type="java.lang.String" value="amqp://localhost:5672" />
            <property name="topicPrefix" value="topic://" />  <!-- only necessary when connecting to
ActiveMQ over AMQP 1.0 -->
          </bean>
        </property>
 </bean>
```

AMQPComponent#amqpComponent() メソッドと **AMQPConnectionDetails** の両方で、トピック接頭辞でコンポーネントを事前に設定するため、明示的に設定する必要はありません。

第6章 APNS

APN コンポーネント

Camel 2.8 から利用可能

apns コンポーネントは、iOS デバイスに通知を送信するために使用されます。apns コンポーネントは javapns ライブラリーを使用します。コンポーネントは、Apple Push Notification Servers (APNS)への通知の送信と、サーバーからのフィードバックの消費をサポートします。

コンシューマーは、デフォルトのポーリング時間 3600 秒で設定されます。サーバーのフラッディングを避けるために、Apple Push Notification Server からのフィードバックストリームを間隔で定期的に使用することが推奨されます。

フィードバックストリームは、非アクティブなデバイスに関する情報を提供します。この情報は、モバイルアプリケーションが頻繁に使用されていない場合は、頻繁に（すべて 2 時間または 3 時間）消費できます。

Transport Layer Security (TLS)を使用するように **apns** コンポーネントを設定するには、Security Guide の Configuring Transport Security for Camel Components の章を参照してください 。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-apns</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

通知を送信するには、以下を実行します。

```
apns:notify[?options]
```

フィードバックを利用するには、以下を実行します。

```
apns:consumer[?options]
```

オプション

プロデューサー

プロパティー	デフォルト	説明

トークン		デフォルトでは空です。通知するデバイスに関連するトークンを静的に宣言する場合は、このプロパティーを設定します。トークンはコンマで区切ります。

コンシューマー

プロパティー	デフォルト	説明
delay	**3600**	各ポーリング間の遅延（秒単位）。
initialDelay	**10**	ポーリングが開始されるまでの秒数。
timeUnit	**SECONDS**	ポーリングの時間単位。
userFixedDelay	**true**	**true** の場合は、プール間の固定遅延を使用します。そうでない場合は、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。

URI にクエリーオプションは **?option=value&option=value&.. の形式で追加できます。**

コンポーネント

ApnsComponent は **com.notnoop.apns.ApnsService** で設定する必要があります。このサービスは、**org.apache.camel.component.apns.factory.ApnsServiceFactory** を使用して作成および設定できます。例については、以下を参照してください。詳細は、テストソースコード を参照してください。

エクスチェンジデータ形式

Camel が非アクティブなデバイスに対応するフィードバックデータを取得すると、InactiveDevice オブジェクトのリストを取得します。取得したリストの各 InactiveDevice オブジェクトは In body として設定され、コンシューマーエンドポイントによって処理されます。

メッセージヘッダー

Camel Apns はこれらのヘッダーを使用します。

プロパティー	デフォルト	説明
CamelApnsTokens		デフォルトでは空です。

CamelApnsMessageType	**文字列、ペイロード、APNS_NOTIFICATION**	メッセージタイプとして PAYLOAD を選択すると、メッセージは APNS ペイロードと見なされ、そのまま送信されます。STRING を選択すると、メッセージは APNS ペイロードに変換されます。APNS_NOTIFICATION は、メッセージボディーを **com.notnoop.apns.ApnsNotification** タイプとして送信するために使用されます。

APNSSERVICEFACTORY BUILDER コールバック

ApnsServiceFactory には、デフォルトの **ApnsServiceBuilder** インスタンスの設定または置き換えに使用できる空のコールバックメソッドが含まれています。メソッドの形式は以下のとおりです。

```
protected ApnsServiceBuilder configureServiceBuilder(ApnsServiceBuilder serviceBuilder);
```

これは以下の方法で使用されます。

```
ApnsServiceFactory proxiedApnsServiceFactory = new ApnsServiceFactory(){

  @Override
  protected ApnsServiceBuilder configureServiceBuilder(ApnsServiceBuilder serviceBuilder) {
    return serviceBuilder.withSocksProxy("my.proxy.com", 6666);
  }

};
```

サンプル

CAMEL XML ルート

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

 <!-- Replace by desired values -->
 <bean id="apnsServiceFactory"
class="org.apache.camel.component.apns.factory.ApnsServiceFactory">

  <!-- Optional configuration of feedback host and port -->
  <!-- <property name="feedbackHost" value="localhost" /> -->
  <!-- <property name="feedbackPort" value="7843" /> -->

  <!-- Optional configuration of gateway host and port -->
```

```
  <!-- <property name="gatewayHost" value="localhost" /> -->
  <!-- <property name="gatewayPort" value="7654" /> -->

  <!-- Declaration of certificate used -->
              <!-- from Camel 2.11 onwards you can use prefix: classpath:, file: to refer to load the
certificate from classpath or file. Default it classpath -->
  <property name="certificatePath" value="certificate.p12" />
  <property name="certificatePassword" value="MyCertPassword" />

  <!-- Optional connection strategy - By Default: No need to configure -->
  <!-- Possible options: NON_BLOCKING, QUEUE, POOL or Nothing -->
  <!-- <property name="connectionStrategy" value="POOL" /> -->
  <!-- Optional pool size -->
  <!-- <property name="poolSize" value="15" /> -->

  <!-- Optional connection strategy - By Default: No need to configure -->
  <!-- Possible options: EVERY_HALF_HOUR, EVERY_NOTIFICATION or Nothing (Corresponds to
NEVER javapns option) -->
  <!-- <property name="reconnectionPolicy" value="EVERY_HALF_HOUR" /> -->
 </bean>

 <bean id="apnsService" factory-bean="apnsServiceFactory" factory-method="getApnsService" />

 <!-- Replace this declaration by wanted configuration -->
 <bean id="apns" class="org.apache.camel.component.apns.ApnsComponent">
  <property name="apnsService" ref="apnsService" />
 </bean>

 <camelContext id="camel-apns-test" xmlns="http://camel.apache.org/schema/spring">
         <route id="apns-test">
                 <from uri="apns:consumer?initialDelay=10&elay=3600&imeUnit=SECONDS" />
                 <to uri="log:org.apache.camel.component.apns?showAll=true&ultiline=true" />
                 <to uri="mock:result" />
         </route>
 </camelContext>

</beans>
```

CAMEL JAVA ルート

CAMEL コンテキストを作成し、プログラムで APNS コンポーネントを宣言します。

```
    protected CamelContext createCamelContext() throws Exception {
        CamelContext camelContext = super.createCamelContext();

        ApnsServiceFactory apnsServiceFactory = new ApnsServiceFactory();
        apnsServiceFactory.setCertificatePath("classpath:/certificate.p12");
        apnsServiceFactory.setCertificatePassword("MyCertPassword");

        ApnsService apnsService = apnsServiceFactory.getApnsService(camelContext);

        ApnsComponent apnsComponent = new ApnsComponent(apnsService);
```

```
        camelContext.addComponent("apns", apnsComponent);

        return camelContext;
    }
```

APNSPRODUCER - IOS ターゲットデバイスがヘッダー経由で動的に設定された "CAMELAPNSTOKENS"

```
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() throws Exception {
                from("direct:test")
                    .setHeader(ApnsConstants.HEADER_TOKENS, constant(IOS_DEVICE_TOKEN))
                    .to("apns:notify");
                }
        }
    }
```

APNSPRODUCER - IOS ターゲットデバイスが URI 経由で静的に設定される

```
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() throws Exception {
                from("direct:test").
                to("apns:notify?tokens=" + IOS_DEVICE_TOKEN);
            }
        };
    }
```

APNSCONSUMER

```
from("apns:consumer?initialDelay=10&delay=3600&timeUnit=SECONDS")
    .to("log:com.apache.camel.component.apns?showAll=true&multiline=true")
    .to("mock:result");
```

関連項目

- コンポーネント
- エンドポイント
- APNS の使用に関するブログ(french)

第7章 ATMOSPHERE-WEBSOCKET

ATMOSPHERE WEBSOCKET SERVLET コンポーネント

Camel 2.14 から利用可能

atmosphere-websocket: コンポーネントは、Websocket 経由で外部クライアントと通信するサーブレットに Websocket ベースの エンドポイント を提供します（外部クライアントからの Websocket 接続を受け入れるサーブレットとして）。コンポーネントは 146章*SERVLET* コンポーネントを使用し、Atmosphere ライブラリーを使用して、さまざまなサーブレットコンテナーで Websocket トランスポートをサポートします(Jetty、Tomcat など)。

組み込み Jetty サーバーを起動する 184章*Websocket* コンポーネントとは異なり、このコンポーネントはコンテナーのサーブレットプロバイダーを使用します。

Maven ユーザーは、このコンポーネントの以下の依存関係を **pom.xml** に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-atmosphere-websocket</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
atmosphere-websocket:///relative path[?options]
```

WEBSOCKET でのデータの読み取りと書き込み

atmopshere-websocket エンドポイントは、エンドポイントがプロデューサーまたはコンシューマーとして設定されているかどうかに応じて、ソケットにデータを書き込むか、ソケットから読み取ることができます。

データの読み取り/書き込みのための URI の設定

以下のルートでは、Camel は指定された WebSocket 接続から読み取ります。

```
from("atmosphere-websocket:///servicepath")
    .to("direct:next");
```

同等の Spring の例：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="atmosphere-websocket:///servicepath"/>
    <to uri="direct:next"/>
  </route>
</camelContext>
```

以下のルートでは、Camel は指定された WebSocket 接続から読み取ります。

```
from("direct:next")
    .to("atmosphere-websocket:///servicepath");
```

同等の Spring の例：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:next"/>
    <to uri="atmosphere-websocket:///servicepath"/>
  </route>
</camelContext>
```

第8章 ATOM

ATOM コンポーネント

atom: コンポーネントは、atom フィードのポーリングに使用されます。

Apache Camel は、デフォルトで 500 ミリ秒ごとにフィードをポーリングします。**注記：** コンポーネントは現在、ポーリング（かかる）フィードのみをサポートします。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-atom</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
atom://atomUri[?options]
```

ここでの **atomUri** は、ポーリングする Atom フィードへの URI です。

オプション

プロパティー	デフォルト	説明
splitEntries	**true**	**true** の場合、Apache Camel はフィードをポーリングし、後続のポーリングではポーリングごとに各エントリーが返されます。フィードに 7 エントリーが含まれる場合、Apache Camel は最初のポーリングの最初のエントリー（次のポーリング上の 2 番目のエントリー）を返します。Apache Camel はフィードで新しい更新を行うエントリーがありません。**false** の場合、Apache Camel は呼び出しごとに新しいフィードをポーリングします。

filter	**true**	は、エントリーを返すようにフィルターするために分割されたエントリーでのみ使用されます。Apache Camel はデフォルトで、フィードから新しいエントリーのみを返す **UpdateDateFilter** を使用します。したがって、フィードから消費するクライアントは、同じエントリーを複数回受け取ることはありません。フィルターは最新の最後のエントリーを返します。
lastUpdate	**null**	選択開始タイムスタンプとしてフィルターによってのみ使用されます(**entry.updated** タイムスタンプを使用します)。構文の形式は **yyyy-MM-ddTHH:MM:ss** です。例： **2007-12-24T17:45:59**
throttleEntries	**true**	camel 2.5: 単一のフィードポーリングで特定されたすべてのエントリーを即座に配信するかどうかを設定します。**true** の場合、**consumer.delay** ごとに 1 つのエントリーのみが処理されます。**splitEntries** が **true** に設定されている場合にのみ適用されます。
feedHeader	**true**	Abdera Feed オブジェクトをヘッダーとして追加するかどうかを設定します。
sortEntries	**false**	**splitEntries** が **true** の場合、これらのエントリーを更新された日付でソートするかどうかを設定します。
consumer.delay	**500**	各ポーリングの遅延（ミリ秒単位）。
consumer.initialDelay	**1000**	ポーリングを開始する前にミリス。
consumer.userFixedDelay	**false**	**true** の場合は、プール間の固定遅延を使用します。そうでない場合は、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。

username		HTTP フィードからポーリングする場合の基本認証の場合。
password		HTTP フィードからポーリングする場合の基本認証の場合。

URI にクエリーオプションは **?option=value&option=value&..** **の形式で追加できます。**

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

エクスチェンジデータ形式

Apache Camel は、エントリーを使用して返された **エクスチェンジ** に In ボディーを設定します。**splitEntries** フラグに応じて、Apache Camel は **Entry** または **List<Entry>** **を返します。**

オプション	値	動作
splitEntries	**true**	現在処理中のフィードからのエントリーを 1 つだけ設定します：**exchange.in.body (Entry)**
splitEntries	**false**	フィードのエントリー一覧全体が設定されます：**exchange.in.body (List<Entry>)**

Apache Camel は in ヘッダーに **Feed** オブジェクトを設定できます(**feedHeader** オプションを参照)。これを無効にします。

メッセージヘッダー

Apache Camel atom はこれらのヘッダーを使用します。

ヘッダー	説明
CamelAtomFeed	Apache Camel 2.0: **org.apache.abdera.model.Feed** オブジェクトを使用する場合は、このヘッダーに設定されます。

サンプル

以下の例では、James Strachan のブログをポーリングしています。

```
from("atom://http://macstrac.blogspot.com/feeds/posts/default").to("seda:feeds");
```

この例では、SEDA キューに適したブログのみをフィルターリングしたいです。この例は、コンテナーで実行されていない、または Spring を使用して Apache Camel スタンドアロンを設定する方法も示しています。

```
@Override
protected CamelContext createCamelContext() throws Exception {
    // First we register a blog service in our bean registry
    SimpleRegistry registry = new SimpleRegistry();
    registry.put("blogService", new BlogService());

    // Then we create the camel context with our bean registry
    context = new DefaultCamelContext(registry);

    // Then we add all the routes we need using the route builder DSL syntax
    context.addRoutes(createMyRoutes());

    // And finally we must start Camel to let the magic routing begins
    context.start();

    return context;
}

/**
 * This is the route builder where we create our routes using the Camel DSL syntax
 */
protected RouteBuilder createMyRoutes() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // We pool the atom feeds from the source for further processing in the seda queue
            // we set the delay to 1 second for each pool.
            // Using splitEntries=true will during polling only fetch one Atom Entry at any given time.
            // As the feed.atom file contains 7 entries, using this will require 7 polls to fetch the entire
            // content. When Camel have reach the end of entries it will refresh the atom feed from URI
source
            // and restart - but as Camel by default uses the UpdatedDateFilter it will only deliver new
            // blog entries to "seda:feeds". So only when James Strachan updates his blog with a new
entry
            // Camel will create an exchange for the seda:feeds.
            from("atom:file:src/test/data/feed.atom?
splitEntries=true&consumer.delay=1000").to("seda:feeds");

            // From the feeds we filter each blot entry by using our blog service class
            from("seda:feeds").filter().method("blogService", "isGoodBlog").to("seda:goodBlogs");

            // And the good blogs is moved to a mock queue as this sample is also used for unit testing
            // this is one of the strengths in Camel that you can also use the mock endpoint for your
            // unit tests
            from("seda:goodBlogs").to("mock:result");
        }
    };
}

/**
```

```
 * This is the actual junit test method that does the assertion that our routes is working
 * as expected
 */
@Test
public void testFiltering() throws Exception {
    // create and start Camel
    context = createCamelContext();
    context.start();

    // Get the mock endpoint
    MockEndpoint mock = context.getEndpoint("mock:result", MockEndpoint.class);

    // There should be at least two good blog entries from the feed
    mock.expectedMinimumMessageCount(2);

    // Asserts that the above expectations is true, will throw assertions exception if it failed
    // Camel will default wait max 20 seconds for the assertions to be true, if the conditions
    // is true sooner Camel will continue
    mock.assertIsSatisfied();

    // stop Camel after use
    context.stop();
}

/**
 * Services for blogs
 */
public class BlogService {

    /**
     * Tests the blogs if its a good blog entry or not
     */
    public boolean isGoodBlog(Exchange exchange) {
        Entry entry = exchange.getIn().getBody(Entry.class);
        String title = entry.getTitle();

        // We like blogs about Camel
        boolean good = title.toLowerCase().contains("camel");
        return good;
    }

}
```

第9章 AVRO

AVRO コンポーネント

Camel 2.10 以降で利用可能

このコンポーネントは、avro のデータ形式を提供します。これにより、Apache Avro のバイナリーデータ形式を使用したメッセージのシリアライズおよびデシリアライズが可能になります。さらに、netty または http で avro を使用するためのプロデューサーおよびコンシューマーエンドポイントを提供することで、Apache Avro の rpc に対応します。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-avro</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

APACHE AVRO の概要

Avro では、形式の json を使用してメッセージタイプとプロトコルを定義してから、指定したタイプおよびメッセージの Java コードを生成できます。スキーマは以下のようになります。

```
{"namespace": "org.apache.camel.avro.generated",
 "protocol": "KeyValueProtocol",

 "types": [
     {"name": "Key", "type": "record",
      "fields": [
          {"name": "key",   "type": "string"}
      ]
     },
     {"name": "Value", "type": "record",
      "fields": [
          {"name": "value",   "type": "string"}
      ]
     }
 ],

 "messages": {
     "put": {
         "request": [{"name": "key", "type": "Key"}, {"name": "value", "type": "Value"} ],
         "response": "null"
     },
     "get": {
         "request": [{"name": "key", "type": "Key"}],
         "response": "Value"
     }
 }
}
```

maven や ant などを使用して、スキーマからクラスを簡単に生成できます。詳細は、Apache Avro のドキュメント を参照してください。

ただし、スキーマの最初のアプローチを強制せず、既存のクラスのスキーマを作成できます。**2.12 以降**、既存のプロトコルインターフェイスを使用して RCP 呼び出しを実行できます。プロトコル自体に interface を使用し、パラメーターおよび結果型には POJO Bean またはプリミティブ/String クラスを使用する必要があります。上記のスキーマに対応する クラスの例を以下に示します。

```
package org.apache.camel.avro.reflection;

public interface KeyValueProtocol {
    void put(String key, Value value);
    Value get(String key);
}

class Value {
    private String value;
    public String getValue() { return value; }
    public void setValue(String value) { this.value = value; }
}
```

注記：既存のクラスは、データ形式ではなく RPC（以下を参照）にのみ使用できます。

AVRO データフォーマットの使用

avro データフォーマットの使用は、ルートでマーシャリングまたはアンマーシャリングするクラスを指定するのと同じくらい簡単です。

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:in"/>
        <marshal>
            <avro instanceClass="org.apache.camel.dataformat.avro.Message"/>
        </marshal>
        <to uri="log:out"/>
    </route>
</camelContext>
```

または、コンテキスト内で dataformat を指定し、ルートから参照することもできます。

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
     <dataFormats>
        <avro id="avro" instanceClass="org.apache.camel.dataformat.avro.Message"/>
    </dataFormats>
    <route>
        <from uri="direct:in"/>
        <marshal ref="avro"/>
        <to uri="log:out"/>
    </route>
</camelContext>
```

同様に、avro データフォーマットを使用して、umarshal を実行できます。

CAMEL での AVRO RPC の使用

前述のように、Avro は、http や netty などの複数のトランスポートに対する RPC サポートも提供します。Camel は、これら 2 つのトランスポートにコンシューマーとプロデューサーを提供します。

```
avro:[transport]:[host]:[port][?options]
```

現在、サポートされるトランスポート値は http または netty です。

2.12 以降、URI でメッセージ名を正しく指定できます。

```
avro:[transport]:[host]:[port][/messageName][?options]
```

コンシューマーの場合は、複数のルートを同じソケットに割り当てることができます。正しいルートへのディスパッチは、avro コンポーネントによって自動的に行われます。messageName が指定されていないルート（ある場合）がデフォルトとして使用されます。

avro ipc に camel プロデューサーを使用する場合は、"in" メッセージボディーには、avro プロトコルに指定された操作のパラメーターが含まれている必要があります。応答は out メッセージのボディーに追加されます。

avro ipc に camel avro コンシューマーを使用する場合と同様に、リクエストパラメーターは作成されたエクスチェンジの in メッセージボディー内に配置され、エクスチェンジが処理されると、out メッセージのボディーが応答として送信されます。

注記： デフォルトでは、コンシューマーパラメーターはアレイにラップされます。**2.12** でパラメーターが 1 つしかない場合は、**singleParameter** URI オプションを使用して、配列のラッピングなしで in メッセージボディーでダーティーとして受信できます。

AVRO RPC URI オプション

名前	バージョン	説明
protocolClassName		avro プロトコルのクラス名。
singleParameter	2.12	true の場合、consumer パラメーターは配列にラップされません。プロトコルがメッセージに対してさらにパラメーターを指定すると失敗する
protocol		Avro Procol オブジェクト。複雑なプロトコルを作成する必要がある場合に、**protocolClassName** の代わりに使用できます。#name 表記を使用してレジストリーから Bean を参照できます。
reflectionProtocol	2.12	指定されたプロトコルオブジェクトがリフレクションプロトコルである場合。**protocolClassName** プロトコルタイプは自動検出されるため、**protocol** パラメーターでのみ使用する必要があります。

AVRO RPC ヘッダー

名前	説明
CamelAvroMessageName	送信するメッセージの名前。コンシューマーによる URI からのメッセージ名の上書き（存在する場合）

例

http 経由で camel avro プロデューサーを使用する例：

```
<route>
    <from uri="direct:start"/>
    <to uri="avro:http:localhost:{{avroport}}?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol"/>
    <to uri="log:avro"/>
</route>
```

上記の例では、**CamelAvroMessageName** ヘッダーを入力する必要があります。**2.12 以降**、以下の構文を使用して定数メッセージを呼び出すことができます。

```
<route>
    <from uri="direct:start"/>
    <to uri="avro:http:localhost:{{avroport}}/put?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol"/>
    <to uri="log:avro"/>
</route>
```

Netty 経由で Camel avro コンシューマーを使用したメッセージの消費例：

```
<route>
    <from uri="avro:netty:localhost:{{avroport}}?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol"/>
    <choice>
        <when>
            <el>${in.headers.CamelAvroMessageName == 'put'}</el>
            <process ref="putProcessor"/>
        </when>
        <when>
            <el>${in.headers.CamelAvroMessageName == 'get'}</el>
            <process ref="getProcessor"/>
        </when>
    </choice>
</route>
```

2 **.12 以降**、同じタスクを実行するように 2 つの異なるルートを設定できます。

```
<route>
    <from uri="avro:netty:localhost:{{avroport}}/put?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol">
    <process ref="putProcessor"/>
</route>
```

```
    <route>
        <from uri="avro:netty:localhost:{{avroport}}/get?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol&singleParameter=true"/>
        <process ref="getProcessor"/>
    </route>
```

上記の例では、get は 1 つのパラメーターのみを取るため、**singleParameter** が使用され、**getProcessor** はボディーで Value クラスを直接受信しますが、 **putProcessor** は String キーと Value 値が配列の内容として入力されるサイズ 2 の配列を受け取ります。

第10章 AWS

10.1. AWS コンポーネントの概要

Amazon Web Services の Camel コンポーネント

Amazon Web Services の Camel コンポーネントは、Camel から AWS サービスへの接続を提供します。

AWS サービス	Camel コンポーネント	Camel バージョン	コンポーネントの説明
Simple Queue Service (SQS)	AWS-SQS	2.6	SQS を使用したメッセージの送受信をサポートします。
Simple Notification Service (SNS)	AWS-SNS	2.8	SNS を使用したメッセージの送信をサポートします。
Simple Storage Service (S3)	AWS-S3	2.8	S3 を使用したオブジェクトの保存および取得をサポートします。
Simple Email Service (SES)	AWS-SES	2.8.4	SES を使用したメールの送信をサポートします。
SimpleDB	AWS-SDB	2.8.4	SDB との間でデータの取得をサポートします。
DynamoDB	AWS-DDB	2.10.0	DDB との間でデータの取得をサポートします。
CloudWatch	AWS-CW	2.10.3	CloudWatch へのメトリクスの送信をサポートします。
シンプルなワークフロー	AWS-SWF	2.13.0	SWF を使用したワークフローの管理をサポートします。
EC2	AWS-EC2	2.16.0	EC2 インスタンスの作成、実行、開始、停止、説明、および終了するための AWS EC2 プラットフォームへのメッセージの送信をサポートします。

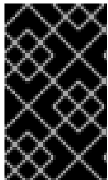
Kinesis Streams	AWS-Kinesis	2.17.0	AWS Kinesis からの受信をサポートします。
DynamoDB Streams	AWS-DDBSTREAM	2.17.0	DynamoDB Streams からの受信をサポートします。

10.2. AWS-CW

CW コンポーネント

Camel 2.11 から利用可能

CW コンポーネントを使用すると、メッセージを Amazon CloudWatch メトリクスに送信できます。Amazon API の実装は AWS SDK によって提供されます。



前提条件

有効な Amazon Web Services 開発者アカウントを持っていて、Amazon CloudWatch を使用するためにサインアップしている必要がある。詳細は、Amazon CloudWatch を参照してください。

URI 形式

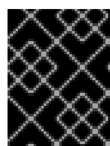
```
aws-cw://namespace[?options]
```

メトリクスが存在しない場合は作成されます。URI にクエリーオプションは **?options=value&option2=value&.. の形式で追加できます。**

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonCwClient	**null**	プロデューサー	レジストリーの **com.amazonaws.services.cloudwatch.AmazonCloudWatch** への参照。
accessKey	**null**	プロデューサー	Amazon AWS Access Key
secretKey	**null**	プロデューサー	Amazon AWS Secret Key

name	null	プロデューサー	メッセージヘッダー 'CamelAwsCwMetricName' が存在しない場合に使用されるメトリクス名。
value	1.0	プロデューサー	メッセージヘッダー 'CamelAwsCwMetricValue' が存在しない場合に使用されるメトリック値。
unit	Count	プロデューサー	メッセージヘッダー 'CamelAwsCwMetricUnit' が存在しない場合に使用されるメトリクスユニット。
namespace	null	プロデューサー	メッセージヘッダー 'CamelAwsCwMetricNamespace' が存在しない場合に使用されるメトリック名前空間。
timestamp	null	プロデューサー	メッセージヘッダー 'CamelAwsCwMetricTimestamp' が存在しない場合に使用されるメトリクスのタイムスタンプ。
amazonCwEndpoint	null	プロデューサー	AWS-CW クライアントが操作するリージョン。
proxyHost	null	プロデューサー	クライアント定義内で使用されるプロキシーホストを指定します。
proxyPort	null	プロデューサー	クライアント定義内で使用されるプロキシーポートを指定します。



必要な CW コンポーネントオプション

Amazon の CloudWatch にアクセスするには、レジストリーまたは accessKey および secretKey で amazonCwClient を指定する必要が あります。

使用方法

CW プロデューサーによって評価されるメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsCwMetricName	**文字列**	Amazon CW メトリクス名。
CamelAwsCwMetricValue	**double**	Amazon CW メトリクス値。
CamelAwsCwMetricUnit	**文字列**	Amazon CW メトリクスユニット。
CamelAwsCwMetricNamespace	**文字列**	Amazon CW メトリクス namespace。
CamelAwsCwMetricTimestamp	**日付**	Amazon CW メトリクスのタイムスタンプ。
CamelAwsCwMetricDimensionName	**文字列**	**Camel 2.12:** Amazon CW メトリクスディメンションの名前。
CamelAwsCwMetricDimensionValue	**文字列**	**Camel 2.12:** Amazon CW metric dimension の値。
CamelAwsCwMetricDimensions	**Map<String, String>**	**Camel 2.12:** ディメンション名とディメンション値のマッピング。

Advanced AmazonCloudWatch configuration

AmazonCloudWatch インスタンス設定をさらに制御する必要がある場合は、独自のインスタンスを作成して、URI から参照することができます。

```
from("direct:start")
.to("aws-cw://namepsace?amazonCwClient=#client");
```

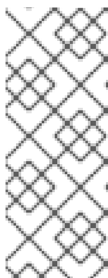
#client はレジストリー内の **AmazonCloudWatch** を参照します。

たとえば、Camel アプリケーションがファイアウォールの背後で実行されている場合は、以下のようになります。

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonCloudWatch client = new AmazonCloudWatchClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



POM.XML

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

${camel-version} は、実際のバージョンの Camel (2.10 以降)に置き換える必要があります。

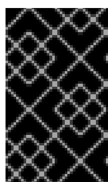
- AWS コンポーネント

10.3. AWS-DDB

DDB コンポーネント

Camel 2.10 以降で利用可能

DynamoDB コンポーネントは、Amazon の DynamoDB サービスからデータの保存および取得をサポートします。



前提条件

有効な Amazon Web Services 開発者アカウントを持っていて、Amazon DynamoDB を使用するためにサインアップしている必要がある。詳細は、Amazon DynamoDB を参照してください。

URI 形式

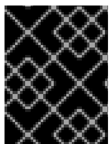
```
aws-ddb://domainName[?options]
```

URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonDDBClient	**null**	プロデューサー	レジストリーの **com.amazonaws.services.dynamodb.AmazonDynamoDB** への参照。
accessKey	**null**	プロデューサー	Amazon AWS Access Key
secretKey	**null**	プロデューサー	Amazon AWS Secret Key

amazonDdbEndpoint	**null**	プロデューサー	AWS-DDB クライアントが動作するリージョン。
tableName	**null**	プロデューサー	現在操作しているテーブルの名前。
readCapacity	**0**	プロデューサー	テーブルからリソースを読み取るために予約するプロビジョニングされたスループット
writeCapacity	**0**	プロデューサー	テーブルにリソースを書き込むために予約するプロビジョニングされたスループット。
consistentRead	**false**	プロデューサー	データの読み取り時に強力な整合性を適用するべきかどうかを決定します。
operation	**PutAttributes**	プロデューサー	有効な値は BatchGetItems、DeleteItem、DeleteTable、DescribeTable、GetItem、PutItem、Query、Scan Scan、UpdateItem、UpdateTable です。
proxyHost	**null**	プロデューサー	クライアント定義内で使用されるプロキシーホストを指定します。
proxyPort	**null**	プロデューサー	クライアント定義内で使用されるプロキシーポートを指定します。



必要な DDB コンポーネントオプション

Amazon の DynamoDB にアクセスするには、レジストリーまたは accessKey および secretKey に amazonDDBClient を指定する必要が あります。

使用方法

DDB プロデューサーによって評価されるメッセージヘッダー

ヘッダー	タイプ	説明

CamelAwsDdbBatchItems	**Map<String, KeysAndAttributes>**	プライマリーキーによって取得するテーブル名と対応する項目のマップ。
CamelAwsDdbTableName	**文字列**	この操作のテーブル名。
CamelAwsDdbKey	**Map<String, AttributeValue>**	テーブル内の各項目を一意に識別するプライマリーキー。
CamelAwsDdbReturnValues	**文字列**	変更前または変更後の属性の名前および値のペアを取得する場合は、このパラメーターを使用します (NONE、ALL_OLD、UPDATED_OLD、ALL_NEW、UPDATED_NEW)。
CamelAwsDdbUpdateCondition	**Map<String, ExpectedAttributeValue>**	条件変更の属性を指定します。
CamelAwsDdbAttributeNames	**Collection<String>**	属性名が指定されていない場合、すべての属性が返されます。
CamelAwsDdbConsistentRead	**ブール値**	true に設定すると、一貫性のある読み取りが発行されます。それ以外の場合は、最終的に一貫性が使用されます。
CamelAwsDdbItem	**Map<String, AttributeValue>**	アイテムの属性のマップ。アイテムを定義するプライマリーキー値を含める必要があります。
CamelAwsDdbKeyConditions	**Map<String, Condition>**	このヘッダーはクエリーの選択基準を指定し、**CamelAwsDdbHashKeyValue** と **CamelAwsDdbScanRangeKeyCondition**の 2 つの古いヘッダーをマージします。
CamelAwsDdbStartKey	**キー**	以前のクエリーを続行するアイテムのプライマリーキー。
CamelAwsDdbLimit	**整数**	返すアイテムの最大数。
CamelAwsDdbScanIndexForward	**ブール値**	インデックスの順方向または逆方向のトラバーサルを指定します。
CamelAwsDdbScanFilter	**Map<String, Condition>**	スキャン結果を評価し、目的の値のみを返します。

CamelAwsDdbUpdateValues	**Map<String, AttributeValueUpdate>**	更新の新しい値とアクションへの属性名のマップ。

BatchGetItems 操作中に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsDdbBatchResponse	**Map<String,BatchResponse>**	テーブル名およびテーブルの各項目属性。
CamelAwsDdbUnprocessedKeys	**Map<String,KeysAndAttributes>**	テーブルのマップと、現在の応答で処理されなかった対応するキーが含まれます。

DeleteItem 操作時に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsDdbAttributes	**Map<String, AttributeValue>**	操作によって返される属性の一覧。

DeleteTable 操作時に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsDdbProvisionedThroughput	**ProvisionedThroughputDescription**	このテーブルの ProvisionedThroughput プロパティーの値
CamelAwsDdbCreationDate	**日付**	このテーブルの DateTime の作成。
CamelAwsDdbTableItemCount	**Long**	このテーブルのアイテム数。
CamelAwsDdbKeySchema	**List<KeySchemaElement>**	このテーブルのプライマリーキーを識別する KeySchema。
CamelAwsDdbTableName	**文字列**	テーブル名。
CamelAwsDdbTableSize	**Long**	テーブルサイズ（バイト単位）。
CamelAwsDdbTableStatus	**文字列**	テーブルのステータス：CREATING、UPDATING、DELETING、ACTIVE

DescribeTable 操作中に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsDdbProvisionedThroughput	**ProvisionedThroughputDescription**	このテーブルの ProvisionedThroughput プロパティーの値
CamelAwsDdbCreationDate	**日付**	このテーブルの DateTime の作成。
CamelAwsDdbTableItemCount	**Long**	このテーブルのアイテム数。
CamelAwsDdbKeySchema	**List<KeySchemaElement>**	このテーブルのプライマリーキーを識別する KeySchema。
CamelAwsDdbTableName	**文字列**	テーブル名。
CamelAwsDdbTableSize	**Long**	テーブルサイズ（バイト単位）。
CamelAwsDdbTableStatus	**文字列**	テーブルのステータス：CREATING、UPDATING、DELETING、ACTIVE
CamelAwsDdbReadCapacity	**Long**	このテーブルの ReadCapacityUnits プロパティー。
CamelAwsDdbWriteCapacity	**Long**	このテーブルの WriteCapacityUnits プロパティー。

GetItem 操作時に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsDdbAttributes	**Map<String, AttributeValue>**	操作によって返される属性の一覧。

PutItem 操作中に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsDdbAttributes	**Map<String, AttributeValue>**	操作によって返される属性の一覧。

Query 操作時に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsDdbItems	**List<java.util.Map<String,AttributeValue>>**	操作によって返される属性の一覧。
CamelAwsDdbLastEvaluatedKey	**キー**	前の結果セットを含む、クエリー操作が停止した項目のプライマリーキー。
CamelAwsDdbConsumedCapacity	**double**	操作中に消費された、テーブルのプロビジョニングされたスループットのキャパシティーユニットの数。
CamelAwsDdbCount	**整数**	応答のアイテム数。

Scan 操作時に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsDdbItems	**List<java.util.Map<String,AttributeValue>>**	操作によって返される属性の一覧。
CamelAwsDdbLastEvaluatedKey	**キー**	前の結果セットを含む、クエリー操作が停止した項目のプライマリーキー。
CamelAwsDdbConsumedCapacity	**double**	操作中に消費された、テーブルのプロビジョニングされたスループットのキャパシティーユニットの数。
CamelAwsDdbCount	**整数**	応答のアイテム数。
CamelAwsDdbScannedCount	**整数**	フィルターが適用される前の完全なスキャン内のアイテムの数。

UpdateItem 操作時に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsDdbAttributes	**Map<String, AttributeValue>**	操作によって返される属性の一覧。

高度な AmazonDynamoDB 設定

AmazonDynamoDB インスタンス設定をさらに制御する必要がある場合は、独自のインスタンスを作成し、URI から参照できます。

```
from("direct:start")
.to("aws-ddb://domainName?amazonDDBClient=#client");
```

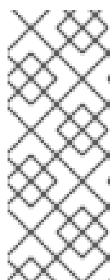
#client は、レジストリー内の **AmazonDynamoDB** を参照します。

たとえば、Camel アプリケーションがファイアウォールの背後で実行されている場合は、以下のようになります。

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonDynamoDB client = new AmazonDynamoDBClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



POM.XML

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

${camel-version} は、実際のバージョンの Camel (2.10 以降)に置き換える必要があります。

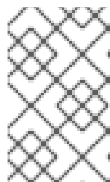
- AWS コンポーネント

10.4. AWS-DDBSTREAM

DynamoDB ストリームコンポーネント

Camel 2.17 以降で利用可能

DynamoDB Stream コンポーネントは、Amazon DynamoDB Stream サービスからメッセージの受信をサポートします。



注記

有効な Amazon Web Services 開発者アカウントがあり、Amazon DynamoDB Streams を使用するようにサインアップする必要があります。詳細は AWS DynamoDBを参照してください。

URI 形式

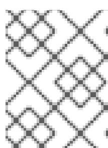
```
aws-ddbstream://table-name[?options]
```

ストリームは、使用する前に作成する必要があります。URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonDynamoDbStreamsClient	**null**	コンシューマー	レジストリーで **com.amazonaws.services.kinesis.AmazonDynamoDBStreams** への参照。
maxMessagesPerPoll	**100**	コンシューマー	各ポーリングで AWS API に返される最大結果。シャードイテレーターがコンシューマーに固有であり、他のコンシューマーに影響を与えないことを示します。
iteratorType	**LATEST**	コンシューマー	**trim_horizon**、**latest**、**after_sequence_number**、または **at_sequence_number** のいずれか。これら 2 つのイテレータータイプの説明は、http://docs.aws.amazon.com/dynamodbstreams/latest/APIReference/API_GetShardIterator.html を参照してください。

sequenceNumberProvider	null	コンシューマー	**org.apache.camel.component.aws.ddbstream.SequenceNumberProvider** の実装またはシーケンス番号を表すリテラル文字列への Bean 参照。このロールは、**after_sequence_number** または **at_sequence_number** イテレータータイプのいずれかを使用する際にストリームの開始場所を決定することです。



注記

プロキシーと関連するクレデンシャルを設定して、レジストリーで amazonDynamoDbStreamsClient を提供する必要があります。

シーケンス番号

リテラル文字列をシーケンス番号として指定したり、レジストリーに Bean を指定したりできます。Bean を使用する例は、現在の位置を変更フィードに保存し、Camel の起動時に復元することです。

AWS 呼び出しが HTTP 400 を返す原因となるため、describe-streams の結果で最も大きなシーケンス番号よりも大きいシーケンス番号を提供するのはエラーです。

バッチコンシューマー

このコンポーネントは、Batch Consumer を実装します。

これにより、たとえば、このバッチに存在するメッセージの数を知ることができ、たとえば、Aggregator にこの数のメッセージを集約させることができます。

AmazonDynamoDBStreamsClient configuration

AmazonDynamoDBStreamsClient のインスタンスを作成し、これをレジストリーにバインドする必要があります。

```
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

Region region = Region.getRegion(Regions.fromName(region));
region.createClient(AmazonDynamoDBStreamsClient.class, null, clientConfiguration);
// the 'null' here is the AWSCredentialsProvider which defaults to an instance of
DefaultAWSCredentialsProviderChain

registry.bind("kinesisClient", client);
```

AWS 認証情報の指定

新しい ClientConfiguration インスタンスの作成時にデフォルトである DefaultAWSCredentialsProviderChain を使用して認証情報を取得することが推奨されますが、createClient (...)の呼び出し時に別の AWSCredentialsProvider を指定できます。

Downtime のコッピング

AWS DynamoDB Streams の 24 時間未満の停止

コンシューマーは(CAMEL-9515に実装されているように)最後に見られたシーケンス番号から再開するため、停止に DynamoDB 自体も含まれていない限り、高速にイベントのフラッディングを受け取る必要があります。

AWS DynamoDB Streams の 24 時間以上停止

AWS は 24 時間の変更のみを保持するため、どのような軽減策がある場合でも、変更イベントを見逃すことができます。

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

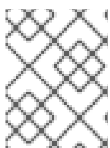
ここで、**${camel-version}** は実際のバージョンの Camel (2.7 以降)に置き換える必要があります。

10.5. AWS-EC2

EC2 コンポーネント

Camel 2.16 以降で利用可能

EC2 コンポーネントは、AWS EC2 インスタンスの作成、実行、起動、停止、および終了をサポートします。



注記

Amazon EC2 を使用するには、有効な Amazon Web Services 開発者アカウントが必要です。詳細は Amazon EC2 を参照してください。

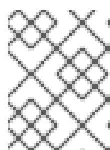
URI 形式

```
aws-ec2://label[?options]
```

URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonEc2Client	**null**	プロデューサー	レジストリーの com.amazonaws.services.ec2.AmazonEC2Client への参照。
accessKey	**null**	プロデューサー	Amazon AWS Access Key
secretKey	**null**	プロデューサー	Amazon AWS Secret Key
amazonEc2Endpoint	**null**	プロデューサー	AWS-EC2 クライアントが操作するリージョン。
operation	**null**	プロデューサー	有効な値は、createAndRunInstances、startInstances、stopInstances、exitInstances、describeInstancesStatus、rebootInstances、monitorInstances、および unmonitorInstances です。
proxyHost	**null**	プロデューサー	**Camel 2.16**: クライアント定義内で使用されるプロキシーホストを指定します。
proxyPort	**null**	プロデューサー	**Camel 2.16**: クライアント定義内で使用されるプロキシーポートを指定します。



注記

Amazon EC2 サービスにアクセスするには、レジストリーまたは accessKey および secretKey で amazonEc2Client を指定する必要があります。

EC2 プロデューサーによって評価されるメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsEC2ImageId	**String**	AWS マーケットプレイスのイメージ ID
CamelAwsEC2InstanceType	com.amazonaws.services.ec2.model.InstanceType	作成して実行するインスタンスタイプ

CamelAwsEC2Operation	**String**	実行する操作
CamelAwsEC2InstanceMinCount	**Int**	実行するインスタンスの最小値数。
CamelAwsEC2InstanceMaxCount	**Int**	実行するインスタンスの最大数。
CamelAwsEC2InstanceMonitoring	**Boolean**	実行中のインスタンスを監視するかどうかを定義します。
CamelAwsEC2InstanceEbsOptimized	**Boolean**	作成インスタンスが EBS I/O に対して最適化されているかどうかを定義します。
CamelAwsEC2InstanceSecurityGroups	**Collection**	インスタンスに関連付けるセキュリティーグループ
CamelAwsEC2InstancesIds	**Collection**	起動、停止、説明、および終了操作を実行するインスタンス IDS のコレクションです。

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

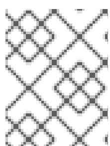
ここで、**${camel-version**} は実際のバージョンの Camel (2.16 以降)に置き換える必要があります。

10.6. AWS-KINESIS

Kinesis コンポーネント

Camel 2.17 以降で利用可能

Kinesis コンポーネントは、Amazon Kinesis サービスからのメッセージの受信および Amazon Kinesis サービスへのメッセージの受信をサポートします。



注記

有効な Amazon Web Services 開発者アカウントを持っていて、Amazon Kinesis を使用するためにサインアップしている必要がある。詳細は AWS Kinesis を参照してください。

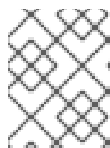
URI 形式

```
aws-kinesis://stream-name[?options]
```

ストリームは、使用する前に作成する必要があります。URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonKinesisClient	**null**	コンシューマー	レジストリーで **com.amazonaws.services.kinesis.AmazonKinesisClient** への参照。
maxMessagesPerPoll	**100**	コンシューマー	各ポーリングで AWS API に返される最大結果。シャードイテレーターがコンシューマーに固有であり、他のコンシューマーに影響を与えないことを示します。
iteratorType	**TRIM_HORIZON**	コンシューマー	trim_horizon または latest のいずれか。これら 2 つのイテレータータイプの説明は、http://docs.aws.amazon.com/kinesis/latest/APIReference/API_GetShardIterator.html を参照してください。



注記

プロキシーと関連するクレデンシャルを設定して、レジストリーで amazonKinesisClient を提供する必要があります。

バッチコンシューマー

このコンポーネントは、Batch Consumer を実装します。

これにより、たとえば、このバッチに存在するメッセージの数を知ることができ、たとえば、Aggregator にこの数のメッセージを集約させることができます。

Kinesis コンシューマーによって設定されるメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsKinesisSequenceNumber	**String**	このレコードのシーケンス番号。これは、サイズが API によって定義されていないため、文字列として表されます。数値型として使用する場合は、次を使用します

CamelAwsKinesisApproximateArrivalTimestamp	**String**	AWS がレコードの到着時間として割り当てた時間。
CamelAwsKinesisPartitionKey	**String**	データレコードが割り当てられているストリーム内のシャードを識別します。

AmazonKinesis の設定

AmazonDynamoDBStreamsClient のインスタンスを作成し、これをレジストリーにバインドする必要があります。

```
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

Region region = Region.getRegion(Regions.fromName(region));
region.createClient(AmazonDynamoDBStreamsClient.class, null, clientConfiguration);
// the 'null' here is the AWSCredentialsProvider which defaults to an instance of
DefaultAWSCredentialsProviderChain

registry.bind("kinesisClient", client);
```

AWS 認証情報の指定

新しい ClientConfiguration インスタンスの作成時にデフォルトである DefaultAWSCredentialsProviderChain を使用して認証情報を取得することが推奨されますが、createClient (...)の呼び出し時に別の AWSCredentialsProvider を指定できます。

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

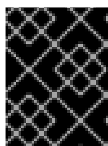
ここで、**${camel-version**} は実際のバージョンの Camel (2.17 以降)に置き換える必要があります。

10.7. AWS-S3

S3 コンポーネント

Camel 2.8 から利用可能

S3 コンポーネントは、Amazon の S3 サービスからの objetc の 保存および取得をサポートします。



前提条件

Amazon S3 を使用するには、有効な Amazon Web Services 開発者アカウントが必要です。詳細は Amazon S3 を参照してください。

URI 形式

```
aws-s3://bucket-name[?options]
```

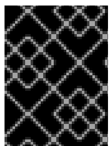
バケットが存在しない場合は作成されます。URI にクエリーオプションは？options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonS3Client	**null**	共有	レジストリーの **com.amazonaws.services.sqs.AmazonS3** への参照。
accessKey	**null**	共有	Amazon AWS Access Key
secretKey	**null**	共有	Amazon AWS Secret Key
amazonS3Endpoint	**null**	共有	AWS-S3 クライアントが操作するリージョン。
region	**null**	プロデューサー	バケットのあるリージョン。このオプションは **com.amazonaws.services.s3.model.CreateBucketRequest** で使用されます。
deleteAfterRead	**true**	コンシューマー	取得後に S3 からオブジェクトを削除します。
deleteAfterWrite	**false**	プロデューサー	S3 ファイルのアップロード後の **Camel 2.11.0** Delete ファイルオブジェクト

maxMessagesPerPoll	10	コンシューマー	1 回のポーリングで取得できるオブジェクトの最大数。**com.amazonaws.services.s3.model.ListObjectsRequest** で使用されています。
policy	**null**	共有	camel 2.8.4: **com.amazonaws.services.s3.AmazonS3#setBucketPolicy （）** メソッドに設定するこのキューのポリシー。
storageClass	**null**	プロデューサー	camel 2.8.4: **com.amazonaws.services.s3.model.PutObjectRequest** リクエストに設定するストレージクラス。
prefix	**null**	コンシューマー	Camel 2.10.1: 対象のオブジェクトのみを消費するために **com.amazonaws.services.s3.model.ListObjectsRequest** で使用される接頭辞。
multiPartUpload	**false**	プロデューサー	Camel 2.15.0: **true** の場合、Camel はマルチパート形式でファイルをアップロードします。ここで、パートサイズは **partSize** オプションで指定できます。
partSize	**25 * 1024 * 1024**	プロデューサー	Camel 2.15.0: マルチパートアップロードで使用される **partSize** を指定します。デフォルトは 25 MB です。
serverSideEncryption	**null**	プロデューサー	camel 2.16: AWS 管理のキーを使用してオブジェクトを暗号化する際にサーバー側の暗号化アルゴリズムを設定します。たとえば、AES256 を使用します。

proxyHost	null	プロデューサー	Camel 2.16: クライアント定義内で使用されるプロキシーホストを指定します。
proxyPort	null	プロデューサー	Camel 2.16: クライアント定義内で使用されるプロキシーポートを指定します。
includeBody	null	コンシューマー	Camel 2.17: **true** の場合、エクスチェンジボディーはファイルの内容にストリームに設定されます。**false** の場合、ヘッダーは S3 オブジェクトメタデータで設定されますが、ボディーは **null** になります。



必要な S3 コンポーネントのオプション

Amazon の S3 にアクセスするには、レジストリーまたは accessKey および secretKey で amazonS3 Client を指定する必要があります。

バッチコンシューマー

このコンポーネントは、バッチコンシューマー を 実装し ます。

これにより、たとえば、このバッチに存在するメッセージの数を把握し、Aggregator はこの数のメッセージを集約できます。

使用方法

S3 プロデューサーによって評価されるメッセージヘッダー

ヘッダー	タイプ
CamelAwsS3Key	**文字列**
CamelAwsS3ContentLength	**Long**
CamelAwsS3ContentType	**文字列**
CamelAwsS3ContentControl	**文字列**
CamelAwsS3ContentDisposition	**文字列**
CamelAwsS3ContentEncoding	**文字列**

CamelAwsS3ContentMD5	**文字列**
CamelAwsS3LastModified	**java.util.Date**
CamelAwsS3StorageClass	**文字列**
CamelAwsS3CannedAcl	**文字列**
CamelAwsS3Acl	**com.amazonaws.services.s3.model.Ac**
CamelAwsS3Headers	**Map<String,String>**
CamelAwsS3ServerSideEncryption	**文字列**

S3 プロデューサーによって設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsS3ETag	**文字列**	新たにアップロードしたオブジェクトの ETag 値。
CamelAwsS3VersionId	**文字列**	新たにアップロードしたオブジェクトの **オプション** のバージョン ID。

S3 コンシューマーによって設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsS3Key	**文字列**	このオブジェクトが保存されるキー。
CamelAwsS3BucketName	**文字列**	このオブジェクトが含まれるバケットの名前。
CamelAwsS3ETag	**文字列**	RFC 1864 に従って、関連付けられたオブジェクトの 16 進エンコードされた 128 ビット MD5 ダイジェスト。このデータは整合性チェックとして使用され、呼び出し元が受信したデータが Amazon S3 によって送信されたデータと同じであることを確認します。

CamelAwsS3LastModified	**日付**	Amazon S3 が最後に関連付けられたオブジェクトへの変更を記録した日時を示す Last-Modified ヘッダーの値。
CamelAwsS3VersionId	**文字列**	関連する Amazon S3 オブジェクトのバージョン ID (ある場合)。バージョン ID は、オブジェクトがオブジェクトのバージョン管理が有効にされている Amazon S3 バケットにアップロードされる場合にのみオブジェクトに割り当てられます。
CamelAwsS3ContentType	**文字列**	関連付けられたオブジェクトに保存されるコンテンツのタイプを示す Content-Type HTTP ヘッダー。このヘッダーの値は標準の MIME タイプです。
CamelAwsS3ContentMD5	**文字列**	RFC 1864 に準拠した、関連付けられたオブジェクト (ヘッダーを含まないコンテンツ) の base64 でエンコードされた 128 ビット MD5 ダイジェスト。このデータは、Amazon S3 が受信したデータが呼び出し元によって送信されたデータと同じであることを確認するために、メッセージ整合性チェックとして使用されます。
CamelAwsS3ContentLength	**Long**	関連付けられたオブジェクトのサイズ (バイト単位) を示す Content-Length HTTP ヘッダー。
CamelAwsS3ContentEncoding	**文字列**	**オプション** の Content-Encoding HTTP ヘッダーには、オブジェクトに適用されたコンテンツエンコーディング、および Content-Type フィールドで参照されるメディアタイプを取得するために適用する必要のあるデコードメカニズムを指定します。
CamelAwsS3ContentDisposition	**文字列**	**オプション** の Content-Disposition HTTP ヘッダー。これは、オブジェクトの保存に使用する推奨ファイル名などの表示情報を指定します。

CamelAwsS3ContentControl	文字列	**オプション** の Cache-Control HTTP ヘッダー。これにより、ユーザーは HTTP 要求/リプライチェーンでキャッシュ動作を指定できます。
CamelAwsS3ServerSideEncryption	文字列	**Camel 2.16:** AWS 管理のキーを使用してオブジェクトを暗号化する際にサーバー側の暗号化アルゴリズム。

高度な AmazonS3 設定

Camel アプリケーションがファイアウォールの背後で実行されている場合、または **AmazonS3** インスタンス設定をより詳細に制御する必要がある場合は、独自のインスタンスを作成できます。

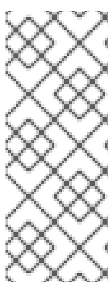
```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");

ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonS3 client = new AmazonS3Client(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

Camel aws-s3 コンポーネント設定でこれを参照します。

```
from("aws-s3://MyBucket?amazonS3Client=#client&delay=5000&maxMessagesPerPoll=5")
.to("mock:result");
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



POM.XML

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

ここで、**$\{camel-version\}** は実際の Camel バージョン(2.8 以降)に置き換える必要があります。

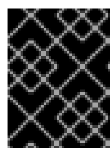
- AWS コンポーネント

10.8. AWS-SDB

SDB コンポーネント

Camel 2.8.4 から利用可能

sdb コンポーネントは 、Amazon の SDB サービスからのデータの保存および取得を サポートします。



前提条件

Amazon SDB を使用するには、有効な Amazon Web Services 開発者アカウントが必要です。詳細は Amazon SDB を 参照してください。

URI 形式

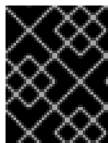
```
aws-sdb://domainName[?options]
```

URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonSDBClient	**null**	プロデューサー	レジストリーの **com.amazonaws.services.simpledb.AmazonSimpleDB** への参照。
accessKey	**null**	プロデューサー	Amazon AWS Access Key
secretKey	**null**	プロデューサー	Amazon AWS Secret Key
amazonSdbEndpoint	**null**	プロデューサー	AWS-SDB クライアントが動作するリージョン。
domainName	**null**	プロデューサー	現在作業中のドメイン名。
maxNumberOfDomains	**100**	プロデューサー	返すドメイン名の最大数。範囲は 1 * から 100 です。
consistentRead	**false**	プロデューサー	データの読み取り時に強力な整合性を適用するべきかどうかを決定します。

operation	**PutAttributes**	プロデューサー	有効な値は BatchDeleteAttributes、BatchPutAttributes、DeleteAttributes、DeleteDomain、DomainMetadata、GetAttributes、ListDomains、PutAttributes、Select です。
proxyHost	**null**	プロデューサー	クライアント定義内で使用されるプロキシーホストを指定します。
proxyPort	**null**	プロデューサー	クライアント定義内で使用されるプロキシーポートを指定します。



必要な SDB コンポーネントのオプション

Amazon の SDB にアクセスするには、レジストリーまたは accessKey および secretKey で amazonSDBClient を指定する必要が あります。

使用方法

SDB プロデューサーによって評価されるメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSdbAttributes	**collection<Attribute>**	処理対象の属性の一覧。
CamelAwsSdbAttributeNames	**Collection<String>**	取得する属性の名前。
CamelAwsSdbConsistentRead	**ブール値**	データの読み取り時に強力な整合性を適用するべきかどうかを決定します。
CamelAwsSdbDeletableItems	**Collection<DeletableItem>**	バッチで削除操作を実行する項目のリスト。
CamelAwsSdbDomainName	**文字列**	現在作業中のドメイン名。
CamelAwsSdbItemName	**文字列**	この項目の一意の鍵
CamelAwsSdbMaxNumberOfDomains	**整数**	返すドメイン名の最大数。範囲は 1* から 100 です。

CamelAwsSdbNextToken	**文字列**	ドメイン/アイテム名の次のリストを開始する場所を指定する文字列。
CamelAwsSdbOperation	**文字列**	URI オプションから操作を上書きするには、以下を実行します。
CamelAwsSdbReplaceableAttributes	**Collection<ReplaceableAttribute>**	Item に配置する属性のリスト。
CamelAwsSdbReplaceableItems	**Collection<ReplaceableItem>**	ドメインに配置する項目のリスト。
CamelAwsSdbSelectExpression	**文字列**	ドメインのクエリーに使用される式。
CamelAwsSdbUpdateCondition	**UpdateCondition**	指定された場合、指定された属性が更新/削除されるかどうかを決定する更新条件。

DomainMetadata 操作中に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSdbTimestamp	**整数**	メタデータの計算時のデータと時間(Epoch (UNIX)秒)。
CamelAwsSdbItemCount	**整数**	ドメインのすべてのアイテム数。
CamelAwsSdbAttributeNameCount	**整数**	ドメイン内の一意の属性名の数。
CamelAwsSdbAttributeValueCount	**整数**	ドメイン内のすべての属性名と値のペアの数。
CamelAwsSdbAttributeNameSize	**Long**	ドメイン内のすべての一意の属性名の合計サイズ（バイト単位）。
CamelAwsSdbAttributeValueSize	**Long**	ドメインのすべての属性値の合計サイズ（バイト単位）。
CamelAwsSdbItemNameSize	**Long**	ドメイン内のすべての項目名の合計サイズ（バイト単位）。

GetAttributes 操作中に設定されたメッセージヘッダー

ヘッダー	タイプ	説明

CamelAwsSdbAttributes	**List<Attribute>**	操作によって返される属性の一覧。

ListDomains 操作中に設定されるメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSdbDomainNames	**List<String>**	式に一致するドメイン名の一覧。
CamelAwsSdbNextToken	**文字列**	指定された MaxNumberOfDomains よりも多くのドメインがあることを示す不透明なトークン。

Select 操作時に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSdbItems	**List<Item>**	select 式に一致するアイテムの一覧。
CamelAwsSdbNextToken	**文字列**	MaxNumberOfItems を超えるアイテム、応答サイズが 1 メガバイト、または実行時間が 5 秒を超えたことを示す不透明なトークン。

高度な AmazonSimpleDB 設定

AmazonSimpleDB インスタンス設定をさらに制御する必要がある場合は、独自のインスタンスを作成して、URI から参照することができます。

```
from("direct:start")
.to("aws-sdb://domainName?amazonSDBClient=#client");
```

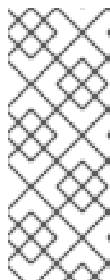
#client はレジストリー内の **AmazonSimpleDB** を参照します。

たとえば、Camel アプリケーションがファイアウォールの背後で実行されている場合は、以下のようになります。

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonSimpleDB client = new AmazonSimpleDBClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



POM.XML

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

${camel-version} は、実際のバージョンの Camel (2.8.4 以降)に置き換える必要があります。

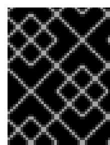
- AWS コンポーネント

10.9. AWS-SES

SES コンポーネント

Camel 2.8.4 から利用可能

ses コンポーネントは、Amazon の SES サービスを使用したメールの 送信をサポートします。



前提条件

Amazon SES を使用するには、有効な Amazon Web Services 開発者アカウントが必要です。詳細は、Amazon SES を参照してください。

URI 形式

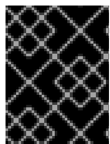
```
aws-ses://from[?options]
```

URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonSESClient	**null**	プロデューサー	レジストリーの **com.amazonaws.services.**
accessKey	**null**	プロデューサー	Amazon AWS Access Key

secretKey	**null**	プロデューサー	Amazon AWS Secret Key
amazonSESEndpoint	**null**	プロデューサー	AWS-SES クライアントが動作するリージョン。
subject	**null**	プロデューサー	メッセージヘッダー 'CamelAwsSesSubject' がRI
to	**null**	プロデューサー	宛先メールアドレスの一覧。'CamelAwsSesTo'
returnPath	**null**	プロデューサー	通知が転送されるメールアドレス。CamelAwsS
replyToAddresses	**null**	プロデューサー	メッセージの返信先メールアドレスのリスト。
proxyHost	**null**	プロデューサー	クライアント定義内で使用されるプロキシーホ
proxyPort	**null**	プロデューサー	クライアント定義内で使用されるプロキシーポ



必要な SES コンポーネントのオプション

Amazon の SES にアクセスするには、レジストリーまたは accessKey および secretKey で amazonS ESClient を指定する必要があります。

使用方法

SES プロデューサーによって評価されるメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSesFrom	**文字列**	送信者のメールアドレス。
CamelAwsSesTo	**List<String>**	このメールの宛先。
CamelAwsSesSubject	**文字列**	メッセージの件名。

CamelAwsSesReplyToAddresses	**List<String>**	メッセージの返信先のメールアドレス。
CamelAwsSesReturnPath	**文字列**	通知の転送先となるメールアドレス。
CamelAwsSesHtmlEmail	**Boolean**	**Camel 2.12.3 以降**、メールコンテンツが HTML かどうかを表示するフラグ。

SES プロデューサーによって設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSesMessageId	**文字列**	Amazon SES メッセージ ID。

AmazonSimpleEmailService の高度な設定

AmazonSimpleEmailService インスタンス設定をさらに制御する必要がある場合は、独自のインスタンスを作成して URI から参照することができます。

```
from("direct:start")
.to("aws-ses://example@example.com?amazonSESClient=#client");
```

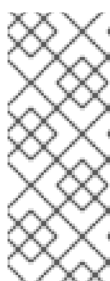
#client はレジストリー内の **AmazonSimpleEmailService** を参照します。

たとえば、Camel アプリケーションがファイアウォールの背後で実行されている場合は、以下のようになります。

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);
AmazonSimpleEmailService client = new AmazonSimpleEmailServiceClient(awsCredentials,
clientConfiguration);

registry.bind("client", client);
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



POM.XML

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

${camel-version} は、実際のバージョンの Camel (2.8.4 以降)に置き換える必要があります。

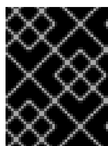
- AWS コンポーネント

10.10. AWS-SNS

SNS コンポーネント

Camel 2.8 から利用可能

SNS コンポーネントを使用すると、メッセージを Amazon Simple Notification Topic に送信できます。Amazon API の実装は AWS SDK によって提供されます。



前提条件

Amazon SNS を使用するには、有効な Amazon Web Services 開発者アカウントが必要です。詳細については、Amazon SNS を参照してください。

URI 形式

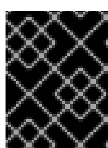
```
aws-sns://topicName[?options]
```

トピックが存在しない場合は作成されます。URI にクエリーオプションは **?options=value&option2=value&.. の形式で追加できます。**

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonSNSClient	**null**	プロデューサー	レジストリーの **com.amazonaws.services.sns.AmazonSNS** への参照。
accessKey	**null**	プロデューサー	Amazon AWS Access Key
secretKey	**null**	プロデューサー	Amazon AWS Secret Key
subject	**null**	プロデューサー	メッセージヘッダー 'CamelAwsSnsSubject' が存在しない場合に使用されるサブジェクト。
amazonSNSEndpoint	**null**	プロデューサー	AWS-SNS クライアントが操作するリージョン。

policy	null	プロデューサー	camel 2.8.4: **com.amazonaws.services.sns.model.SetTopicAttributesRequest** に設定するこのキューのポリシー。
proxyHost	**null**	プロデューサー	クライアント定義内で使用されるプロキシーホストを指定します。
proxyPort	**null**	プロデューサー	クライアント定義内で使用されるプロキシーポートを指定します。



必要な SNS コンポーネントオプション

Amazon の SNS にアクセスするには、レジストリーまたは accessKey および secretKey に amazonSNSClient を指定する必要が あります。

使用方法

SNS プロデューサーによって評価されるメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSnsSubject	**文字列**	Amazon SNS メッセージサブジェクト。設定されていない場合は、**SnsConfiguration** からのサブジェクトが使用されます。

SNS プロデューサーによって設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSnsMessageId	**文字列**	Amazon SNS メッセージ ID。

AmazonSNS の高度な設定

AmazonSNS インスタンス設定をさらに制御する必要がある場合は、独自のインスタンスを作成して、URI から参照することができます。

```
from("direct:start")
.to("aws-sns://MyTopic?amazonSNSClient=#client");
```

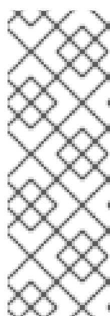
#client は、レジストリー内の **AmazonSNS** を参照します。

たとえば、Camel アプリケーションがファイアウォールの背後で実行されている場合は、以下のようになります。

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);
AmazonSNS client = new AmazonSNSClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



POM.XML

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

ここで、**$\{camel-version\}** は実際の Camel バージョン(2.8 以降)に置き換える必要があります。

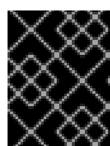
- AWS コンポーネント

10.11. AWS-SQS

SQS コンポーネント

Camel 2.6 以降で利用可能

sqs コンポーネントは、Amazon の SQS サービスへのメッセージの 送受信をサポートします。



前提条件

Amazon SQS を使用するには、有効な Amazon Web Services 開発者アカウントが必要です。詳細は、Amazon SQS を参照してください。

URI 形式

```
aws-sqs://queue-name[?options]
```

キューが存在しない場合は作成されます。URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

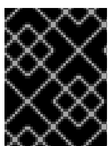
URI オプション

名前	デフォルト値	コンテキスト	説明
amazonSQSClient	**null**	共有	レジストリーの **com.amazonaws.services.sqs.AmazonSQS** への参照。
accessKey	**null**	共有	Amazon AWS Access Key
secretKey	**null**	共有	Amazon AWS Secret Key
amazonSQSEndpoint	**null**	共有	AWS-SQS クライアントが操作するリージョン。Camel が AWS-SQS クライアントを作成する場合にのみ機能します。つまり、amazonSQSClient を明示的に設定した場合、この設定は効果がありません。直接作成するクライアントにこれを設定する必要があります。
attributeNames	**null**	コンシューマー	消費時に受け取る属性名のリスト。 **Camel 2.17:** 複数の名前はコンマで区切ることができます。
messageAttributeNames	**null**	コンシューマー	消費時に受け取るメッセージ属性名のリスト。 **Camel 2.17:** 複数の名前はコンマで区切ることができます。
concurrentConsumers	**1**	コンシューマー	**Camel 2.15.0 では、** 複数のスレッドを使用して SQS キューをポーリングしてスループットを増やすことができます。また、これを正常に機能させるには、**maxMessagesPerPoll** オプションを設定する必要があります。

defaultVisibilityTimeout	**null**	共有	**com.amazonaws.services.sqs.model.CreateQueueRequest** に設定される可視性のタイムアウト（秒単位）。
deleteAfterRead	**true**	コンシューマー	メッセージがルートによって読み取られ、処理された後に SQS からメッセージを削除します。
deleteIfFiltered	**true**	コンシューマー	**Camel 2.12.2**,2.13.0 エクスチェンジがフィルターを通過できない場合、DeleteMessage を SQS キューに送信するかどうか。false およびエクスチェンジがルートのアップストリームで Camel フィルターを作成しない場合は、DeleteMessage を送信しません。
maxMessagesPerPoll	**null**	コンシューマー	**com.amazonaws.services.sqs.model.ReceiveMessageRequest** に設定できる 1 回のポーリングで受信できるメッセージの最大数。
visibilityTimeout	**null**	共有	ReceiveMessage リクエストによって受信メッセージが取得されてから、後続の取得要求から受信したメッセージが非表示になる期間（秒単位）。これは、**defaultVisibilityTimeout** とは異なる場合にのみ有効です。

extendMessageVisibility	**false**	コンシューマー	Camel 2.10: 有効にすると、スケジュールされたバックグラウンドタスクは SQS でのメッセージの可視性を拡張し続けます。これは、メッセージの処理に時間がかかる場合に必要です。**true** に設定すると、**visibilityTimeout** を設定する必要があります。詳細は、Amazon ドキュメント を参照してください。
maximumMessageSize	**null**	共有	Camel 2.8: maximumMessageSize（バイト単位）で、このキューに SQS メッセージを含めることができ、**com.amazonaws.services.sqs.model.SetQueueAttributesRequest** で設定できます。
messageRetentionPeriod	**null**	共有	Camel 2.8: messageRetentionPeriod（秒単位）は、**com.amazonaws.services.sqs.model.SetQueueAttributesRequest** に設定されるこのキューの SQS によってメッセージが保持されます。
policy	**null**	共有	camel 2.8: **com.amazonaws.services.sqs.model.SetQueueAttributesRequest** に設定するこのキューのポリシー。
delaySeconds	**null**	プロデューサー	Camel 2.9.3: 数秒間メッセージの送信を遅延します。

waitTimeSeconds	**0**	プロデューサー	Camel 2.11: ReceiveMessage アクション呼び出しは応答に含めるメッセージがキューにあるまで待機する期間（秒単位(0 から 20)。
receiveMessageWaitTimeSeconds	**0**	共有	Camel 2.11: リクエストで WaitTimeSeconds を指定しない場合、キュー属性 ReceiveMessageWaitTimeSeconds を使用して待機時間を決定します。
queueOwnerAWSAccountId	**null**	共有	Camel 2.12: キューを別のアカウント所有者に接続する必要がある場合は、キュー所有者の aws アカウント ID を指定します。
region	**null**	共有	Camel 2.12.3: サービス URL をビルドするために **queueOwnerAWSAccountId** で使用できるキューリージョンを指定します。
redrivePolicy	**null**	共有	Camel 2.15.0: DeadLetter キューにメッセージを送信するポリシーを指定します。詳細は、Amazon ドキュメント を参照してください。
proxyHost	**null**	プロデューサー	クライアント定義内で使用されるプロキシーホストを指定します。
proxyPort	**null**	プロデューサー	クライアント定義内で使用されるプロキシーポートを指定します。



必要な SQS コンポーネントオプション

Amazon の SQS にアクセスするには、レジストリーまたは accessKey および secretKey に amazonSQSClient を指定する必要が あります。

バッチコンシューマー

このコンポーネントは、バッチコンシューマー を 実装し ます。

これにより、たとえば、このバッチに存在するメッセージの数を把握し、たとえば Aggregator はこの数のメッセージを集約できます。

使用方法

SQS プロデューサーによって設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSqsMD5OfBody	**文字列**	Amazon SQS メッセージの MD5 チェックサム。
CamelAwsSqsMessageId	**文字列**	Amazon SQS メッセージ ID。
CamelAwsSqsDelaySeconds	**整数**	**Camel 2.11** 以降、Amazon SQS メッセージが他から確認できる遅延秒数。

SQS コンシューマーによって設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelAwsSqsMD5OfBody	**文字列**	Amazon SQS メッセージの MD5 チェックサム。
CamelAwsSqsMessageId	**文字列**	Amazon SQS メッセージ ID。
CamelAwsSqsReceiptHandle	**文字列**	Amazon SQS メッセージ受信ハンドル。
CamelAwsSqsAttributes	**Map<String, String>**	Amazon SQS メッセージ属性。

AmazonSQS の高度な設定

Camel アプリケーションがファイアウォールの背後で実行されている場合、または AmazonSQS インスタンス設定をより詳細に制御する必要がある場合は、独自のインスタンスを作成できます。

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");

ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);
```

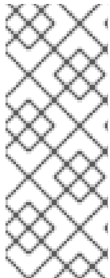
```
AmazonSQS client = new AmazonSQSClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

Camel aws-sqs コンポーネント設定でこれを参照します。

```
from("aws-sqs://MyQueue?amazonSQSClient=#client&delay=5000&maxMessagesPerPoll=5")
.to("mock:result");
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



POM.XML

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

ここで、**$\{camel-version\}** は実際の Camel バージョン(2.6 以降)に置き換える必要があります。

JMS スタイルのセレクター

SQS はセレクターを許可しませんが、Camel Filter EIP を使用し、適切な **visibilityTimeout** を設定することで、これを効果的に実行できます。SQS がメッセージをディスパッチすると、DeleteMessage を受信しない限り、別のコンシューマーにメッセージをディスパッチするまで、可視性のタイムアウトを待機します。デフォルトでは、ルートが失敗しない限り、Camel はルートの最後に DeleteMessage を常に送信します。適切なフィルターリングを行い、ルートが正常に完了する場合でも DeleteMessage を送信しないようにするには、フィルターを使用します。

```
from("aws-sqs://MyQueue?
amazonSQSClient=#client&defaultVisibilityTimeout=5000&deleteIfFiltered=false")
.filter("${header.login} == true")
.to("mock:result");
```

上記のコードでは、エクスチェンジに適切なヘッダーがない場合は、フィルターを介して送信されず、SQS キューから削除されません。5000 ミリ秒が経過すると、メッセージは他のコンシューマーに表示されます。

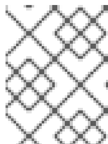
- AWS コンポーネント

10.12. AWS-SWF

SWF コンポーネント

Camel 2.13 で利用可能

Simple Workflow コンポーネントは、Amazon の Simple Workflow サービスからワークフローの 管理をサポートします。



注記

Amazon Simple Workflow を使用するには、有効な Amazon Web Services 開発者アカウントが必要です。詳細は Amazon Simple Workflow を参照してください。

URI 形式

```
aws-swf://<workflow|activity>[?options]
```

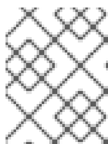
URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
amazonSWClient	**null**	すべて	レジストリーの com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient への参照。
accessKey	**null**	すべて	Amazon AWS Access Key。
secretKey	**null**	すべて	Amazon AWS Secret Key。
sWClient.XXX	**null**	すべて	使用中の AmazonSimpleWorkflowClient に設定するプロパティー。
clientConfiguration.XXX	**null**	すべて	使用中の ClientConfiguration に設定するプロパティー。
startWorkflowOptions.XXX	**null**	ワークフロー/プロデューサー	使用中の useStartWorkflowOptions に設定するプロパティー。
operation	**START**	ワークフロー/プロデューサー	ワークフローで実行する操作。サポートされる操作は次のとおりです。**SIGNAL**、**CANCEL**、**TERMINATE**、**GET_STATE**、NORMAL、NORMAL。**START** **DESCRIBE** **GET_HISTORY**
domainName	**null**	すべて	使用するワークフロードメイン。

activityList	**null**	アクティビティー/コンシューマー	アクティビティーの使用元となるリスト名。
workflowList	**null**	ワークフロー/コンシューマー	ワークフローを使用するリスト名。
eventName	**null**	すべて	使用するワークフローまたはアクティビティーイベント名。
version	**null**	すべて	使用するワークフローまたはアクティビティーイベントバージョン。
signalName	**null**	ワークフロー/プロデューサー	ワークフローに送信するシグナルの名前。
childPolicy	**null**	ワークフロー/プロデューサー	ワークフローを終了する際に子ワークフローで使用するポリシー。
terminationReason	**null**	ワークフロー/プロデューサー	ワークフローを終了する理由。
stateResultType	**Object**	ワークフロー/プロデューサー	ワークフローの状態のクエリー時の結果のタイプ。
terminationDetails	**null**	ワークフロー/プロデューサー	ワークフローを終了するための詳細。
dataConverter	**JsonDataConverter**	すべて	データをシリアライズ/デシリアライズするために使用する **com.amazonaws.services.simpleworkflow.flow.DataConverter** のインスタンス。
activitySchedulingOptions	**null**	アクティビティー/プロデューサー	異なるタイムアウトオプションを指定するために使用される **ActivitySchedulingOptions** のインスタンス。
activityTypeExecutionOptions	**null**	アクティビティー/コンシューマー	**ActivityTypeExecutionOptions** のインスタンス。

activityTypeRegistrationOptions	**null**	アクティビティー/コンシューマー	**ActivityTypeRegistrationOptions** のインスタンス。
workflowTypeRegistrationOptions	**null**	ワークフロー/コンシューマー	**WorkflowTypeRegistrationOptions** のインスタンス。



注記

Amazon の Simple Workflow Service にアクセスするには、レジストリー または accessKey および secretKey に amazonSWClient を指定する必要があります。

使用方法

SWF ワークフロープロデューサーによって評価されるメッセージヘッダー

ワークフロープロデューサーは、ワークフローとの対話を可能にします。新しいワークフロー実行の開始、状態へのクエリー、実行中のワークフローへのシグナル送信、または終了および取り消しを行うことができます。

ヘッダー	タイプ	説明
CamelSWFOperation	**String**	ワークフローで実行する操作。サポートされている操作：SIGNAL、CANCEL、TERMINATE、GET_STATE、START、DESCRIBE、GET_HISTORY。
CamelSWFWorkflowId	**String**	使用するワークフロー ID。
CamelAwsDdbKeyCamelSWFRunId	**String**	使用するワクロー実行 ID。
CamelSWFStateResultType	**String**	ワークフローの状態のクエリー時の結果のタイプ。
CamelSWFEventName	**String**	使用するワークフローまたはアクティビティーイベント名。
CamelSWFVersion	**String**	使用するワークフローまたはアクティビティーイベントバージョン。
CamelSWFReason	**String**	ワークフローを終了する理由。
CamelSWFDetails	**String**	ワークフローを終了するための詳細。

CamelSWFChildPolicy	**String**	ワークフローを終了する際に子ワークフローで使用するポリシー。

SWF ワークフロープロデューサーによって設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelSWFWorkflowId	**String**	使用される Worfklow ID、または新たに生成された ID。
CamelAwsDdbKeyCamelSWFRunId	**String**	使用されるか、または生成される Worfklow run ID。

SWF ワークフローコンシューマーによって設定されたメッセージヘッダー

ワークフローコンシューマーはワークフローロジックを表します。開始時に、ワークフローのデシジョンタスクのポーリングを開始し、これを処理します。ワークフローコンシューマールートは、デシジョンタスクの処理の他に、シグナル（ワークフロープロデューサーから送信）または状態クエリーも受信します。ワークフローコンシューマーの主な目的は、アクティビティープロデューサーを使用して実行するアクティビティータスクをスケジュールすることです。実際のアクティビティータスクは、ワークフローコンシューマーが開始したスレッドからのみスケジュールできます。

ヘッダー	タイプ	説明
CamelSWFAction	**String**	現在のイベントのタイプ (CamelSWFActionExecute、CamelSWFSignalReceivedAction または CamelSWFGetStateAction)を指定します。
CamelSWFWorkflowReplaying	**boolean**	現在のデシジョンタスクが再生されているかどうかを示します。
CamelSWFWorkflowStartTime	**long**	このデシジョンタスクの開始イベントの時間。

SWF アクティビティープロデューサーによって設定されたメッセージヘッダー

アクティビティープロデューサーは、アクティビティータスクをスケジュールできます。アクティビティープロデューサーは、ワークフローコンシューマーが開始したスレッドからのみ使用できます。つまり、ワークフローコンシューマーによって開始される同期エクスチェンジを処理できます。

ヘッダー	タイプ	説明

CamelSWFEventName	**String**	スケジュールするアクティビティー名。
CamelSWFVersion	**String**	スケジュールするアクティビティーバージョン。

SWF Activity コンシューマーによって設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelSWFTaskToken	**String**	手動で完了したタスクのタスク完了を報告するのに必要なタスクトークン。

高度な amazonSWClient 設定

AmazonSimpleWorkflowClient インスタンス設定をさらに制御する必要がある場合は、独自のインスタンスを作成して URI から参照できます。

#client は、レジストリー の AmazonSimpleWorkflowClient を参照し ます。

たとえば、Camel アプリケーションがファイアウォールの背後で実行されている場合は、以下のようになります。

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonSimpleWorkflowClient client = new AmazonSimpleWorkflowClient(awsCredentials,
clientConfiguration);

registry.bind("client", client);
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

ここで、**${camel-version}** は実際のバージョンの Camel (2.13 以降)に置き換える必要があります。

第11章 BEAN

BEAN コンポーネント

bean: コンポーネントは Bean を Apache Camel メッセージエクスチェンジにバインドします。

URI 形式

```
bean:beanID[?options]
```

beanID には、レジストリーで Bean を検索するために使用される任意の文字列を指定できます。

オプション

名前	タイプ	デフォルト	説明
メソッド	文字列	null	呼び出される Bean のメソッド名。指定しない場合、Camel はメソッド自体を判断しようとします。あいまいな場合は、例外が発生します。詳細は、Bean Binding を参照してください。**Camel 2.8 以降** では、タイプ修飾子を指定して、オーバーロードされたメソッドに使用する正確なメソッドを特定することができます。**Camel 2.9** 以降では、メソッド構文でパラメーター値を直接指定できます。
cache	boolean	false	有効にすると、Apache Camel は最初の レジストリー ルックアップの結果をキャッシュします。レジストリー の Bean がシングルトンスコープとして定義されている場合、キャッシュを有効にできます。
multiParameterArray	boolean	false	メッセージボディーから渡されるパラメーターを処理する方法。**true** の場合、In メッセージボディーはパラメーターの配列である必要があります。

bean.xxx		null	Camel 2.17: クラス名から create bean インスタンスで追加オプションを設定します。たとえば、Bean で **foo** オプションを設定するには、**bean.foo=123** を使用します。

URI にクエリーオプションは **?option=value&option=value&.. の形式で追加できます。**

使用

メッセージの消費に使用されるオブジェクトインスタンスは、レジストリー に明示的に登録する必要があり ます。たとえば、Spring を使用している場合は、Spring 設定 **spring.xml** で Bean を定義する必要があり、Spring を使用しない場合は Bean を JNDI に配置します。

```
// lets populate the context with the services we need
// note that we could just use a spring.xml file to avoid this step
JndiContext context = new JndiContext();
context.bind("bye", new SayService("Good Bye!"));

CamelContext camelContext = new DefaultCamelContext(context);
```

エンドポイントが登録されたら、エクスチェンジの処理に使用するルートを構築できます。

```
// lets add simple route
camelContext.addRoutes(new RouteBuilder() {
    public void configure() {
        from("direct:hello").to("bean:bye");
    }
});
```

bean: エンドポイントは、ルートへの入力として定義できません。つまり、消費できません。一部のインバウンドメッセージエンドポイントから Bean エンドポイントに出力としてのみルーティングできます。http://camel.apache.org/endpoint.htmlしたがって、**direct:** または **queue:** エンドポイントを入力として使用することを検討してください。

ProxyHelper で **createProxy()** メソッドを使用して、BeanExchange を生成して任意のエンドポイントに送信するプロキシーを作成できます。

```
Endpoint endpoint = camelContext.getEndpoint("direct:hello");
ISay proxy = ProxyHelper.createProxy(endpoint, ISay.class);
String rc = proxy.say();
assertEquals("Good Bye!", rc);
```

Spring DSL を使用した同じルートの場合:

```
<route>
  <from uri="direct:hello">
  <to uri="bean:bye"/>
</route>
```

エンドポイントとしての BEAN

Apache Camel は、エンドポイントとしての Bean の呼び出しもサポートします。ルートは以下のとおりです。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="myBean"/>
    <to uri="mock:results"/>
  </route>
</camelContext>

<bean id="myBean" class="org.apache.camel.spring.bind.ExampleBean"/>
```

エクスチェンジが **myBean** にルーティングされると、Apache Camel は Bean バインディングを使用して Bean を呼び出します。Bean のソースはプレーン POJO です。

```
public class ExampleBean {

    public String sayHello(String name) {
        return "Hello " + name + "!";
    }
}
```

Apache Camel は Bean バインディング を使用して **sayHello** メソッドを呼び出します。これには、エクスチェンジの In ボディーを **String** 型に変換し、メソッドの出力を Exchange Out ボディーに保存します。

JAVA DSL BEAN 構文

Java DSL には、Bean コンポーネントのシンタックス(sugtactic sugar)が同梱されています。Bean を明示的にエンドポイント (つまり **to ("bean:beanName")**) として指定する代わりに、次の構文を使用できます。

```
// Send message to the bean endpoint
// and invoke method resolved using Bean Binding.
from("direct:start").beanRef("beanName");

// Send message to the bean endpoint
// and invoke given method.
from("direct:start").beanRef("beanName", "methodName");
```

Bean への参照の名前を渡す代わりに (Camel がレジストリーでそれを検索できるようにするため)、Bean 自体を指定できます。

```
// Send message to the given bean instance.
from("direct:start").bean(new ExampleBean());

// Explicit selection of bean method to be invoked.
from("direct:start").bean(new ExampleBean(), "methodName");

// Camel will create the instance of bean and cache it for you.
from("direct:start").bean(ExampleBean.class);
```

BEAN バインディング

呼び出される Bean **メソッド** の選択方法(method パラメーターで明示的に指定されていない場合)と、メッセージ からパラメーター値がどのように構築されるかは、Apache Camel のさまざまな Bean 統合メカニズム全体で使用される Bean バインディング メカニズムによってすべて定義されます。

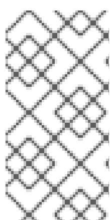
- クラス コンポーネント
- Bean バインディング
- Bean インテグレーション

BEAN 言語

Bean 言語の目的は、Bean で単純なメソッドを使用して式または述語を実装することです。Spring ***ApplicationContext*** などのレジストリーで Bean 名を指定すると、メソッドを呼び出して Expression または Predicate を評価します。ただし、メソッド名を指定しない場合は、以下を使用して選択できます。

- Bean バインディングのルール
- メッセージボディーのタイプ
- Bean メソッドのアノテーション

詳細は Bean 言語 の章を参照してください。



注記

Camel 2.17 以降では、Bean 言語は静的メソッドを純粋な静的クラスで呼び出すことができます。Bean 言語が OGNL メソッドチェーンからメソッドを呼び出す場合、メソッドは null 値を返します。ただし、を提供する追加のメソッド呼び出しを防ぐことができます。 **NullPointerException.**

第12章 BEAN バリデーター

BEAN バリデーターコンポーネント

Apache Camel 2.3 で利用可能

バリデーターコンポーネントは、Java Bean Validation API (JSR 303) を使用してメッセージボディーの Bean 検証を実行します。Camel は Hibernate Validator のリファレンス実装を使用します。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-bean-validator</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
bean-validator:label[?options]
```

または

```
bean-validator://label[?options]
```

label は、エンドポイントを記述する任意のテキスト値です。以下の形式で URI にクエリーオプションを追加できます。 **?option=value&option=value&...**

URI オプション

以下の URI オプションがサポートされています。

オプション	デフォルト	説明
group	**javax.validation.groups.Default**	使用するカスタム検証グループ。
messageInterpolator	**org.hibernate.validator.engine.ResourceBundleMessageInterpolator**	レジストリーのカスタム **javax.validation.MessageInterpolator** への参照。
traversableResolver	**org.hibernate.validator.engine.resolver.DefaultTraversableResolver**	レジストリーのカスタム **javax.validation.TraversableResolver** への参照。
constraintValidatorFactory	**org.hibernate.validator.engine.ConstraintValidatorFactoryImpl**	レジストリーのカスタム **javax.validation.ConstraintValidatorFactory** への参照。

OSGI デプロイメント

OSGi 環境で Hibernate Validator を使用するには、**org.apache.camel.component.bean.validator.HibernateValidationProviderResolver** と同じように、専用の **ValidationProviderResolver** 実装を使用します。以下のスニペットは、このアプローチを示しています。Camel 2.13.0 から **HibernateValidationProviderResolver** を使用できることに注意してください。

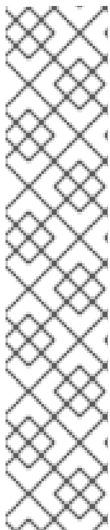
例12.1 Using HibernateValidationProviderResolver

```
from("direct:test")
  .to("bean-validator://ValidationProviderResolverTest?
validationProviderResolver=#myValidationProviderResolver");

...

<bean id="myValidationProviderResolver"
class="org.apache.camel.component.bean.validator.HibernateValidationProviderResolver"/>
```

カスタム **ValidationProviderResolver** が定義されておらず、バリデーターコンポーネントが OSGi 環境にデプロイされている場合、**HibernateValidationProviderResolver** は自動的に使用されます。

例

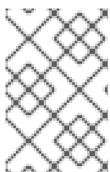
以下のアノテーションを持つ Java Bean があると仮定します。



CAR.JAVA

```
// Java
public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 5, max = 14, groups = OptionalChecks.class)
    private String licensePlate;

    // getter and setter
}
```

カスタムバリデーショングループのインターフェイス定義:



OPTIONALCHECKS.JAVA

```
public interface OptionalChecks {
}
```

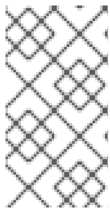
以下の Apache Camel ルートでは、製造元および licensePlate 属性の **@NotNull** 制約のみが検証されます(Apache Camel はデフォルトのグループ **javax.validation.groups.Default**を使用します)。

```
from("direct:start")
.to("bean-validator://x")
.to("mock:end")
```

OptionalChecks グループからの制約を確認する場合は、以下のようなルートを定義する必要があります。

```
from("direct:start")
.to("bean-validator://x?group=OptionalChecks")
.to("mock:end")
```

両方のグループからの制約を確認する場合は、最初に新しいインターフェイスを定義する必要があります。



ALLCHECKS.JAVA

```
@GroupSequence({Default.class, OptionalChecks.class})
public interface AllChecks {
}
```

ルート定義は以下のようになります。

```
from("direct:start")
.to("bean-validator://x?group=AllChecks")
.to("mock:end")
```

また、独自のメッセージインターポレーター、通過可能なリゾルバー、および制約バリデーターファクトリーを提供する必要がある場合は、次のようなルートを記述する必要があります。

```
<bean id="myMessageInterpolator" class="my.ConstraintValidatorFactory" />
<bean id="myTraversableResolver" class="my.TraversableResolver" />
<bean id="myConstraintValidatorFactory" class="my.ConstraintValidatorFactory" />

from("direct:start")
.to("bean-validator://x?
group=AllChecks&messageInterpolator=#myMessageInterpolator&traversableResolver=#myTraversabl
eResolver&constraintValidatorFactory=#myConstraintValidatorFactory")
.to("mock:end")
```

制約は、Java アノテーションではなく XML として記述することもできます。この場合、次のようなファイル **META-INF/validation.xml** を提供する必要があります。

VALIDATION.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
 xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration">
 <default-provider>org.hibernate.validator.HibernateValidator</default-provider>
 <message-
interpolator>org.hibernate.validator.engine.ResourceBundleMessageInterpolator</mess
age-interpolator>
 <traversable-
resolver>org.hibernate.validator.engine.resolver.DefaultTraversableResolver</traversabl
e-resolver>
 <constraint-validator-
factory>org.hibernate.validator.engine.ConstraintValidatorFactoryImpl</constraint-
validator-factory>

 <constraint-mapping>/constraints-car.xml</constraint-mapping>
</validation-config>
```

constraints-car.xml ファイル

CONSTRAINTS-CAR.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/mapping validation-
mapping-1.0.xsd"
 xmlns="http://jboss.org/xml/ns/javax/validation/mapping">
 <default-package>org.apache.camel.component.bean.validator</default-package>

 <bean class="CarWithoutAnnotations" ignore-annotations="true">
  <field name="manufacturer">
   <constraint annotation="javax.validation.constraints.NotNull" />
  </field>

  <field name="licensePlate">
   <constraint annotation="javax.validation.constraints.NotNull" />

   <constraint annotation="javax.validation.constraints.Size">
    <groups>
     <value>org.apache.camel.component.bean.validator.OptionalChecks</value>
    </groups>
    <element name="min">5</element>
    <element name="max">14</element>
   </constraint>
  </field>
 </bean>
</constraint-mappings>
```

第13章 BEANSTALK

BEANSTALK コンポーネント

Camel 2.15 で利用可能

camel-beanstalk プロジェクトは、Beantalk ジョブのジョブの取得および後処理のための Camel コンポーネントを提供します。

Beanstalk ジョブライフサイクルの詳細は、Beanstalk プロトコル を参照してください。

DEPENDENCIES

Maven ユーザーは以下の依存関係を追加する必要があります。 **pom.xml**

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-beanstalk</artifactId>
  <version>${camel-version}</version>
</dependency>
```

ここで、**${camel-version}** は実際のバージョンの Camel (2.15.0 以降)に置き換える必要があります。

URI 形式

```
beanstalk://[host[:port]][/tube][?options]
```

Beanstalk のデフォルトは **port** または **host** と **port** のいずれかを省略できます(localhost と 11300)。**tube** を省略すると、Beanstalk コンポーネントは "default" という名前の tube を使用します。

リッスンしている場合、おそらくいくつかのツーブからのジョブを監視することをお勧めします。たとえば、プラス記号で区切ります。

```
beanstalk://localhost:11300/tube1+tube2
```

Tube 名はデコードされるため、Tube の名前に + や ? などの特殊文字が含まれている場合は、適切に URL エンコードするか、RAW 構文を使用する必要があります。詳細は、を参照してください。

このようにして、ジョブを Beantalk に書いた時に複数の tub を指定できません。

一般的な URI オプション

名前	デフォルト値	説明
jobPriority	1000	ジョブの優先度。(0 は最大で、Beantalk プロトコルを参照)
jobDelay	0	ジョブの遅延（秒単位）。

jobTimeToRun	60	秒単位でのジョブの実行時間(0 の場合)、beanstalkd デーモンがそれを 1 に自動的に引き上げます。Beanstalk プロトコルを参照してください）。

プロデューサー UIR オプション

プロデューサーの動作は **command** パラメーターの影響を受けます。このパラメーターは、ジョブの動作を指示します。

名前	デフォルト値	説明
command	put	• **put** ジョブを Beantalk に配置する手段。ジョブボディーは Camel メッセージボディーに指定されます。ジョブ ID は **beanstalk.jobId** メッセージヘッダーで返されます。 • **delete**、**release**、または **touch** では、メッセージヘッダー **beanstalk.jobId** のジョブ ID が必要です。**bury**操作の結果が **beanstalk.result** メッセージヘッダーで返されます。 • **kick** メッセージボディーで起動するジョブの数を想定し、メッセージヘッダー **beanstalk.result** で実際に開始するジョブの数を返します。

コンシューマー UIR オプション

コンシューマーは、予約された直後にジョブを削除したり、Camel ルートが処理するまで待ちます。最初のシナリオはメッセージキューに似ていますが、2 番目のシナリオは job queue に似ています。この動作は、**consumer.awaitJob** パラメーターによって制御されます。このパラメーターは、デフォルトで **true** に相当します(Beantalkd の性質)。

同期されると、コンシューマーは正常なジョブの完了時に **delete** を呼び出して、失敗時に **bury** を呼び出します。URI で **consumer.onFailure** パラメーターを指定すると、失敗時に実行するコマンドを選択できます。**bury**、**delete** または **release** の値を取ることができます。

JavaBeanstalkClient ライブラリーの同じパラメーターに対応するブール値パラメーター **consumer.useBlockIO** があります。デフォルトでは、**true** です。

release を指定する際には、失敗したジョブが同じ tube ですぐに利用可能になり、コンシューマーが再度取得を試みるため注意してください。**release** を指定し、**jobDelay** を指定できます。

名前	デフォルト値	説明
onFailure	bury	処理に失敗した場合に使用するコマンド。bury、delete、または release のいずれかを選択できます。
useBlockIO	true	blockIO を使用するかどうか。
awaitJob	true	beanstalk からジョブを確認する前にジョブが完了するまで待機するかどうか。

beanstalk コンシューマーはスケジュールされたポーリングコンシューマーです。これは、コンシューマーがポーリングする頻度など、設定可能なオプションが多いことを意味します。詳細は コンシューマーのポーリング を参照してください。

コンシューマーヘッダー

コンシューマーは、Exchange メッセージに複数のジョブヘッダーを保存します。

プロパティー	タイプ	説明
beanstalk.jobId	long	ジョブ ID:
beanstalk.tube	string	このジョブを含む tube の名前
beanstalk.state	string	"ready" または "delayed" または "reserved" または "buried" (予約されている必要があります)
beanstalk.priority	long	優先度値セット
beanstalk.age	int	このジョブを作成した put コマンドからの秒単位の時間
beanstalk.time-left	int	サーバーがこのジョブを準備状態にあるキューに配置するまでの秒数
beanstalk.timeouts	int	予約中にこのジョブがタイムアウトした回数

beanstalk.releases	int	クライアントが予約からこのジョブをリリースした回数
beanstalk.buries	int	このジョブが入念された回数
beanstalk.kicks	int	このジョブが起動した回数

例

この Camel コンポーネントを使用すると、ジョブを処理し、それらを Beanstalkd デーモンに提供することができます。簡単なデモルートは以下のようになります。

```
from("beanstalk:testTube").
  log("Processing job #${property.beanstalk.jobId} with body ${in.body}").
  process(new Processor() {
    @Override
    public void process(Exchange exchange) {
      // try to make integer value out of body
      exchange.getIn().setBody( Integer.valueOf(exchange.getIn().getBody(classOf[String])) );
    }
  }).
  log("Parsed job #${property.beanstalk.jobId} to body ${in.body}");
```

```
from("timer:dig?period=30seconds").
  setBody(constant(10)).log("Kick ${in.body} buried/delayed tasks").
  to("beanstalk:testTube?command=kick");
```

最初のルートでは、tube "testTube" で新しいジョブをリッスンしています。受け取ったら、メッセージボディーの整数値を解析しようとしています。正常に完了するとログを記録し、この正常な交換の完了により、Camel コンポーネントを使用して Beantalk からこのジョブを自動的に **削除** します。一方、ジョブデータを解析できない場合、エクスチェンジは失敗して、Camel コンポーネントはデフォルトでこれをバウズします。これにより、後で処理したり、失敗したジョブを手動で検査したりできます。

そのため、2 つ目のルートは Beanstalk に定期的に要求して、buried および/または delayed 状態から通常のキューに 10 個のジョブを ***開始*** *します。*

第14章 BINDY

BINDY コンポーネント

camel-bindy コンポーネントは、Java Bean を介して構造化されていないデータの解析とバインディングを有効にします。これらの Java Bean はアノテーションで定義されたバインディングマッピングで設定されます。

```
@CsvRecord(separator = ",")
public class Customer {

    @DataField(pos = 1)
    private String firstName;

    @DataField(pos = 2)
    private String lastName;
    ...
}
```

たとえば、データ形式 **BindyCsvDataFormat** unmarshall CSV データをドメインモデルに提供することもできます。

```
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .unmarshal(new BindyCsvDataFormat(Customer.class))
        .to("mock:result");
    }
});
camelctx.start();
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

第15章 BOX

BOX コンポーネント

Camel 2.14 から利用可能

Box コンポーネントは、box-java-sdk-v2 を使用してアクセス可能なすべての Box.com API へのアクセスを提供します。これにより、ファイルのアップロードやダウンロード、フォルダーの作成、編集、管理を行うメッセージの作成、編集、管理を行うことができます。また、ユーザーアカウントへの更新のポーリングや企業アカウントの変更などを可能にする API もサポートしています。

Box.com では、すべてのクライアントアプリケーション認証に OAuth2.0 を使用する必要があります。アカウントで camel-box を使用するには、https://app.box.com/developers/services/edit/ の Box.com 内に新しいアプリケーションを作成する必要があります。Box アプリケーションのクライアント ID およびシークレットにより、現在のユーザーを必要とする Box API にアクセスできます。ユーザーアクセストークンは、エンドユーザーの API によって生成および管理されます。また、Camel アプリケーションは com.box.boxjavalibv2.authorization.IAuthSecureStorage の実装を登録し、com.box.boxjavalibv2.dao.IAuthData OAuth トークン を提供 します。

TLS (Transport Layer Security) を使用するように **camel-box** コンポーネントを設定するには、Security Guide の Configuring Transport Security for Camel Components の章を参照してください。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-box</artifactId>
   <version>${camel-version}</version>
</dependency>
```

URI 形式

```
box://endpoint-prefix/endpoint?[options]
```

エンドポイント接頭辞は以下のいずれかになります。

- コラボレーション
- コメント
- events
- files
- folders
- groups
- poll-events
- search
- 共有コメント

- *shared-files*
- *shared-folders*
- *shared-items*
- *users*

BOX コンポーネント

*Box コンポーネントは、以下のオプションで設定できます。これらのオプションは、**org.apache.camel.component.box.BoxConfiguration** タイプのコンポーネントの Bean プロパティー **configuration** を使用して提供できます。これらのオプションは、エンドポイント URI で指定することもできます。*

オプション	タイプ	説明
authSecureStorage	com.box.boxjavalibv2.authorization.IAuthSecureStorage	OAuth Secure Storage コールバックは、OAuth トークンを提供および保存するために使用できます。コールバックは最初の呼び出し時に null を返し、コンポーネントがアプリケーションのログインおよび承認を行い、OAuth トークンを取得できるようにします。これにより、セキュアなストレージに保存できます。コンポーネントがトークンを自動的に作成できるようにするには、ユーザーパスワードを指定する必要があります。
boxConfig	com.box.boxjavalibv2.IBoxConfig	カスタム Box SDK 設定（通常は必要ありません）
clientId	文字列	Box アプリケーションクライアント ID
clientSecret	文字列	Box アプリケーションクライアントシークレット
connectionManagerBuilder	com.box.boxjavalibv2.BoxConnectionManagerBuilder	カスタム Box 接続マネージャービルダー。基礎となる HttpClient の最大接続などのデフォルト設定を上書きするために使用されます。
httpParams	java.util.Map	プロキシーホストなどの設定用のカスタム HTTP パラメーター
loginTimeout	int	コンポーネントが Box.com からの応答を待つ時間。デフォルトは 30 秒です。
refreshListener	com.box.boxjavalibv2.authorization.OAuthRefreshListener	Camel アプリケーションがルート外のアクセストークンを使用する必要がある場合は、トークン更新の OAuth リスナー

オプション	タイプ	説明
revokeOnShutdown	boolean	ルートシャットダウン時に OAuth 更新トークンを取り消すフラグ。デフォルトは false です。ユーザーパスワードを指定して、カスタム IAuthSecureStorage または自動コンポーネントのログインのいずれかを使用して再起動時に新しい更新トークンが必要になります。
sharedLink	文字列	shared-* エンドポイントのボックス共有リンク。共有コメント、ファイル、またはフォルダーのリンクにすることができます。
sharedPassword	文字列	共有リンクに関連付けられたパスワード。sharedLink で指定する必要があります。
userName	文字列	Box ユーザー名（指定が必要）
userPassword	文字列	Box ユーザーパスワード。authSecureStorage が設定されていない場合や、最初の呼び出しで null を返す必要があります。

プロデューサーエンドポイント：

プロデューサーエンドポイントはエンドポイント接頭辞を使用し、続いてエンドポイント名と以下で説明する関連オプションを使用できます。一部のエンドポイントには、短縮エイリアスを使用できます。エンドポイント URI には 接頭辞が含まれている必要があります。

*必須ではないエンドポイントオプションは **[]** で表されます。エンドポイントに必須のオプションがない場合は、**[]** オプションのセットの1 つを指定する必要があります。プロデューサーエンドポイントは、Camel Exchange In メッセージに含まれる値を持つ endpoint オプションの名前が含まれる必要がある特別なオプション **inBody** を使用することもできます。*

*エンドポイントオプションは、エンドポイント URI またはメッセージヘッダーで動的に指定できます。メッセージヘッダー名は **CamelBox.<option>** の形式で指定する必要があります。**inBody** オプションはメッセージヘッダーを上書きすることに注意してください。つまり、エンドポイントオプション **inBody=option** は **CamelBox.option** ヘッダーを上書きすることに注意してください。*

*エンドポイント URI またはメッセージヘッダーのいずれかで **defaultRequest** オプションに値が指定されていない場合、これは **null** であると想定されます。**null** 値は、他のオプションが一致するエンドポイントを満たさない場合にのみ使用されることに注意してください。*

*Box API エラーが発生すると、エンドポイントは **com.box.restclientv2.exceptions.BoxSDKException** **派生例外原因で** **RuntimeCamelException** を出力します。*

エンドポイント接頭辞コラボレーション

Box のコラボレーションに関する詳細は、https://developers.box.com/docs/#collaborations を参照してください。以下のエンドポイントは、以下のように接頭辞 ***collaborations*** *で呼び出すことができます。*

> *box://collaborations/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
createCollaboration	create	collabRequest, folderId	com.box.boxjavalibv2.dao.BoxCollaboration
deleteCollaboration	delete	collabId, defaultRequest	
getAllCollaborations	allCollaborations	getAllCollabsRequest	java.util.List
getCollaboration	コラボレーション	collabId, defaultRequest	com.box.boxjavalibv2.dao.BoxCollaboration
updateCollaboration	update	collabId, collabRequest	com.box.boxjavalibv2.dao.BoxCollaboration

コラボレーションの URI オプション

名前	タイプ
collabId	文字列
collabRequest	com.box.boxjavalibv2.requests.requestobjects.BoxCollabRequestObject
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject
folderId	文字列
getAllCollabsRequest	com.box.boxjavalibv2.requests.requestobjects.BoxGetAllCollabsRequestObject

エンドポイント接頭辞イベント

Box イベントの詳細は、https://developers.box.com/docs/#events を参照してください。このエンドポイントはプロデューサーで使用できますが、Box イベントは ***poll-events*** *エンドポイント接頭辞を使用してコンシューマーエンドポイントとしてより適しています。以下のエンドポイントは、以下のように接頭辞* ***events*** *で呼び出すことができます。*

> *box://events/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
getEventOptions	eventOptions	defaultRequest	com.box.boxjavalibv2.dao.BoxCollection
getEvents	events	eventRequest	com.box.boxjavalibv2.dao.BoxEventCollection

イベントの URI オプション

名前	タイプ
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject
eventRequest	com.box.boxjavalibv2.requests.requestobjects.BoxEventRequestObject

エンドポイント接頭辞グループ

Box グループの詳細は、https://developers.box.com/docs/#groups を参照してください。以下のエンドポイントは、以下のように接頭辞 ***groups*** *で呼び出すことができます。*

> *box://groups/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
createGroup		[groupRequest], [name]	com.box.boxjavalibv2.dao.BoxGroup
createMembership		[groupId, role, userId], [groupMembershipRequest]	com.box.boxjavalibv2.dao.BoxGroupMembership
deleteGroup	delete	defaultRequest, groupId	
deleteMembership	delete	defaultRequest, membershipId	
getAllCollaborations	allCollaborations	defaultRequest, groupId	com.box.boxjavalibv2.dao.BoxCollection
getAllGroups	allGroups	defaultRequest	com.box.boxjavalibv2.dao.BoxCollection
getMembership	メンバーシップ	defaultRequest, membershipId	com.box.boxjavalibv2.dao.BoxGroupMembership

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
getMemberships	メンバーシップ	defaultRequest, groupId	com.box.boxjavalibv2.dao.BoxCollection
updateGroup	update	groupId、groupRequest	com.box.boxjavalibv2.dao.BoxGroup
updateMembership	update	[groupMembershipRequest], [role], membershipId	com.box.boxjavalibv2.dao.BoxGroupMembership

グループの URI オプション

名前	タイプ
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject
groupId	文字列
groupMembershipRequest	com.box.boxjavalibv2.requests.requestobjects.BoxGroupMembershipRequestObject
groupRequest	com.box.boxjavalibv2.requests.requestobjects.BoxGroupRequestObject
membershipId	文字列
name	文字列
role	文字列
userId	文字列

エンドポイント接頭辞検索

Box 検索 API の詳細は、https://developers.box.com/docs/#search を参照してください。以下のエンドポイントは、以下のように接頭辞 ***search*** *で呼び出すことができます。*

> *box://search/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
search		defaultRequest、searchQuery	com.box.boxjavalibv2.dao.BoxCollection

検索の URI オプション

名前	タイプ
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject
searchQuery	文字列

エンドポイント接頭辞のコメントと共有コメント

*Box コメントの詳細は、https://developers.box.com/docs/#comments を参照してください。以下のエンドポイントは、以下のように接頭辞 **コメント** または **shared-comments** で呼び出すことができます。**shared-comments** 接頭辞には **sharedLink** および **sharedPassword** プロパティーが必要です。*

```
box://comments/endpoint?[options]
box://shared-comments/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
addComment		[commentRequest], [commentedItemId, commentedItemType, message]	com.box.boxjavalibv2.dao.BoxComment
deleteComment	delete	commentId, defaultRequest	
getComment	comment	commentId, defaultRequest	com.box.boxjavalibv2.dao.BoxComment
updateComment	update	commentId, commentRequest	com.box.boxjavalibv2.dao.BoxComment

コメントおよび共有コメントの URI オプション

名前	タイプ
commentId	文字列
commentRequest	com.box.boxjavalibv2.requests.requestobjects.BoxCommentRequestObject
commentedItemId	文字列
commentedItemType	com.box.boxjavalibv2.dao.IBoxType
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject
message	文字列

エンドポイント接頭辞ファイルと共有ファイル

*Box ファイルの詳細は、https://developers.box.com/docs/#files を参照してください。以下のエンドポイントは、以下のように接頭辞 **files** または **shared-files** で呼び出すことができます。**shared-files** 接頭辞には **sharedLink** および **sharedPassword** プロパティーが必要です。*

> *box://files/endpoint?[options]*
> *box://shared-files/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
copyFile		fileId, itemCopyRequest	com.box.boxjavalibv2.dao.BoxFile
createSharedLink	create	fileId, sharedLinkRequest	com.box.boxjavalibv2.dao.BoxFile
deleteFile		defaultRequest, fileId	
downloadFile	ダウンロード	[destination, listener], [listener, outputStreams], defaultRequest, fileId	java.io.InputStream
downloadThumbnail	ダウンロード	extension, fileId, imageRequest	java.io.InputStream
getFile	file	defaultRequest, fileId	com.box.boxjavalibv2.dao.BoxFile
getFileComments	fileComments	defaultRequest, fileId	com.box.boxjavalibv2.dao.BoxCollection
getFileVersions	fileVersions	defaultRequest, fileId	java.util.List
getPreview	preview	extension, fileId, imageRequest	com.box.boxjavalibv2.dao.BoxPreview
getThumbnail	thumbnail	extension, fileId, imageRequest	com.box.boxjavalibv2.dao.BoxThumbnail
updateFileInfo	update	fileId, fileRequest	com.box.boxjavalibv2.dao.BoxFile
uploadFile	upload	fileUploadRequest	com.box.boxjavalibv2.dao.BoxFile
uploadNewVersion	upload	fileId, fileUploadRequest	com.box.boxjavalibv2.dao.BoxFile

ファイルおよび SHARED-FILES の URI オプション

名前	タイプ
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject
destination	java.io.File
extension	文字列
fileId	文字列
fileRequest	com.box.boxjavalibv2.requests.requestobjects.BoxFileRequestObject
fileUploadRequest	com.box.restclientv2.requestsbase.BoxFileUploadRequestObject
imageRequest	com.box.boxjavalibv2.requests.requestobjects.BoxImageRequestObject
itemCopyRequest	com.box.boxjavalibv2.requests.requestobjects.BoxItemCopyRequestObject
listener	com.box.boxjavalibv2.filetransfer.IFileTransferListener
outputStreams	java.io.OutputStream[]
sharedLinkRequest	com.box.boxjavalibv2.requests.requestobjects.BoxSharedLinkRequestObject

エンドポイント接頭辞フォルダーおよび共有フォルダー

Box フォルダーの詳細は、https://developers.box.com/docs/#folders を参照してください。以下のエンドポイントは、以下のように接頭辞 ***folders*** *または* ***shared-folders*** *で呼び出すことができます。接頭辞* ***shared-folders*** *には* ***sharedLink*** *および* ***sharedPassword*** *プロパティーが必要です。*

> *box://folders/endpoint?[options]*
> *box://shared-folders/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
copyFolder		folderId, itemCopyRequest	com.box.boxjavalibv2.dao.BoxFolder
createFolder	create	folderRequest	com.box.boxjavalibv2.dao.BoxFolder
createSharedLink	create	folderId, sharedLinkRequest	com.box.boxjavalibv2.dao.BoxFolder

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
deleteFolder	delete	folderDeleteRequest, folderId	
getFolder	folder	defaultRequest, folderId	com.box.boxjavalibv2.dao.BoxFolder
getFolderCollaborations	folderCollaborations	defaultRequest, folderId	java.util.List
getFolderItems	folderItems	folderId, pagingRequest	com.box.boxjavalibv2.dao.BoxCollection
updateFolderInfo	update	folderId, folderRequest	com.box.boxjavalibv2.dao.BoxFolder

フォルダーまたは共有フォルダーの URI オプション

名前	タイプ
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject
folderDeleteRequest	com.box.boxjavalibv2.requests.requestobjects.BoxFolderDeleteRequestObject
folderId	文字列
folderRequest	com.box.boxjavalibv2.requests.requestobjects.BoxFolderRequestObject
itemCopyRequest	com.box.boxjavalibv2.requests.requestobjects.BoxItemCopyRequestObject
pagingRequest	com.box.boxjavalibv2.requests.requestobjects.BoxPagingRequestObject
sharedLinkRequest	com.box.boxjavalibv2.requests.requestobjects.BoxSharedLinkRequestObject

エンドポイント接頭辞 SHARED-ITEMS

Box 共有項目の詳細は、https://developers.box.com/docs/#shared-items を参照してください。以下のエンドポイントは、以下のように接頭辞 ***shared-items*** *で呼び出すことができます。*

```
box://shared-items/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
getSharedItem	sharedItem	defaultRequest	com.box.boxjavalibv2.dao.BoxItem

SHARED-ITEMS の URI オプション

名前	タイプ
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject

エンドポイント接頭辞ユーザー

Box ユーザーの詳細は、https://developers.box.com/docs/#users を参照してください。以下のエンドポイントは、以下のように接頭辞 ***users*** *で呼び出すことができます。*

> *box://users/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
addEmailAlias		emailAliasRequest, userId	com.box.boxjavalibv2.dao.BoxEmailAlias
createEnterpriseUser	create	userRequest	com.box.boxjavalibv2.dao.BoxUser
deleteEmailAlias		defaultRequest, emailId, userId	
deleteEnterpriseUser		userDeleteRequest, userId	
getAllEnterpriseUser	allEnterpriseUser	defaultRequest, filterTerm	java.util.List
getCurrentUser	currentUser	defaultRequest	com.box.boxjavalibv2.dao.BoxUser
getEmailAliases	emailAliases	defaultRequest, userId	java.util.List
moveFolderToAnotherUser		folderId、simpleUserRequest、userId	com.box.boxjavalibv2.dao.BoxFolder
updateUserInformaiton	update	userId, userRequest	com.box.boxjavalibv2.dao.BoxUser

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
updateUserPrimaryLogin	update	userId, userUpdateLoginRequest	com.box.boxjavalibv2.dao.BoxUser

ユーザーの URI オプション

名前	タイプ
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject
emailAliasRequest	com.box.boxjavalibv2.requests.requestobjects.BoxEmailAliasRequestObject
emailId	文字列
filterTerm	文字列
folderId	文字列
simpleUserRequest	com.box.boxjavalibv2.requests.requestobjects.BoxSimpleUserRequestObject
userDeleteRequest	com.box.boxjavalibv2.requests.requestobjects.BoxUserDeleteRequestObject
userId	文字列
userRequest	com.box.boxjavalibv2.requests.requestobjects.BoxUserRequestObject
userUpdateLoginRequest	com.box.boxjavalibv2.requests.requestobjects.BoxUserUpdateLoginRequestObject

コンシューマーエンドポイント：

Box イベントの詳細は https://developers.box.com/docs/#events および長いポーリングについては https://developers.box.com/docs/#events-long-polling を参照してください。コンシューマーエンドポイントは、次の例に示すように、エンドポイント接頭辞 ***poll-events*** *のみを使用できます。デフォルトでは、コンシューマーは長いポーリングから com.box.boxjavalibv2.dao.BoxEventCollection を分割し、すべての com.box.boxjavalibv2.dao.BoxEvent のエクスチェンジを作成します。コンシューマーが単一のエクスチェンジでコレクション全体を返すようにするには、URI オプション* ***consumer.splitResult=false*** *を使用します。*

```
box://poll-events/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
poll		limit、streamPosition、streamType	デフォルトでは com.box.boxjavalibv2.dao.BoxEvent、または consumer.splitResult=false の場合は com.box.boxjavalibv2.dao.BoxEventCollection です。

POLL-EVENTS の URI オプション

名前	タイプ
limit	整数
streamPosition	Long
streamType	文字列
splitResult	boolean

メッセージヘッダー

*オプションはいずれも、**CamelBox.** 接頭辞を持つプロデューサーエンドポイントのメッセージヘッダーで指定できます。*

メッセージボディー

*すべての結果メッセージ本文は Box Java SDK によって提供されるオブジェクトを使用します。プロデューサーエンドポイントは、**inBody** エンドポイントパラメーターに受信メッセージボディーのオプション名を指定できます。*

型コンバーター

*Box コンポーネントは、GenericFile オブジェクトを File コンポーネントから **com.box.restclientv2.requestsbase.Box File UploadRequestObject に変換して Box.**com にファイルをアップロードするための Camel 型コンバーターも提供します。アップロードのターゲット **folderId** は、エクスチェンジプロパティー **CamelBox.folderId** で指定できます。エクスチェンジプロパティーが指定されていない場合、ルートフォルダー ID の**デフォルト値が 0** になります。*

ユースケース

以下のルートは、新しいファイルをユーザーのルートフォルダーにアップロードします。

```
from("file:...")
 .to("box://files/upload/inBody=fileUploadRequest");
```

以下のルートは、ユーザーのアカウントをポーリングして更新を確認します。

```
from("box://poll-events/poll?streamPosition=-1&streamType=all&limit=100")
 .to("bean:blah");
```

以下のルートは、動的ヘッダーオプションを持つプロデューサーを使用します。fileId プロパティーには Box ファイル ID があるため、以下のようにCamelBox.fileIdヘッダーに割り当てられ ます。

```
from("direct:foo")
 .setHeader("CamelBox.fileId", header("fileId"))
 .to("box://files/download")
 .to("file://...");
```

第16章 BRAINTREE

BRAINTREE コンポーネント

Camel 2.17 以降で利用可能

Braintree コンポーネントは、以下の支払い方法をサポートする Braintree Payments サービスへのアクセスを提供します。

- クレジットカードおよびデビットカード
- Apple Pay
- Android Pay
- Venmo
- PayPal
- Bitcoin （ベータ）

camel-braintree を使用するには、アカウント(サンドボックス または 実稼働)から取得できる API クレデンシャル を指定する必要があります。

Maven ユーザーは、このコンポーネントの以下の依存関係を **pom.xml** に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-braintree</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
braintree://endpoint-prefix/endpoint?[options]
```

エンドポイント接頭辞は以下のいずれかになります。

- addOn
- address
- clientToken
- creditCardverification
- customer
- discount
- merchantAccount
- paymentmethod

- *paymentmethodNonce*
- *plan*
- *settlementBatchSummary*
- *subscription*
- *transaction*
- *webhookNotification*

BRAINTREECOMPONENT

Braintree コンポーネントは、以下のオプションで設定できます。これらのオプションは、org.apache.camel.component.braintree.BraintreeConfiguration タイプのコンポーネントの Bean プロパティー設定 を使用して指定できます。

オプション	タイプ	説明
environment	**String**	要求の転送先を指定する値 - サンドボックスまたは実稼働
merchantId	**String**	使用するゲートウェイアカウントの一意の識別子。これは、実際の業者アカウント ID とは異なります。
publicKey	**String**	ユーザー固有のパブリック ID
privateKey	**String**	共有すべきでないユーザー固有のセキュアな ID。
httpLogLevel	**java.util.logging.Level**	HTTP 呼び出し **の** Camel 2.17.1 ロギングレベル
httpLogName	**String**	http 呼び出しをログに記録するために使用する Camel 2 **.17.1** ログカテゴリー。デフォルトは Braintree です。
httpReadTimeout	**Integer**	HTTP **呼び出しの** Camel 2.17.1 の読み取りタイムアウト

上記のオプションはすべて Braintree Payments によって提供されます。

プロデューサーエンドポイント：

プロデューサーエンドポイントはエンドポイント接頭辞を使用し、続いてエンドポイント名と以下で説明する関連オプションを使用できます。一部のエンドポイントには、短縮エイリアスを使用できます。エンドポイント URI には 接頭辞が含まれている必要があります。

必須ではないエンドポイントオプションは [] で示されます。エンドポイントに必須のオプションがない場合は、[] オプションのセットの 1 つを指定する必要があります。プロデューサーエンドポイントは、Camel Exchange In メッセージに含まれる値を持つ endpoint オプションの名前が含まれる必要がある特別なオプション inBody を使用することもできます。

エンドポイントオプションは、エンドポイント URI またはメッセージヘッダーで動的に指定できます。メッセージヘッダー名は CamelBraintree.<option> 形式である必要があります。inBody オプションはメッセージヘッダーを上書きすることに注意してください。つまり、エンドポイントオプションの inBody=option は CamelBraintree.option ヘッダーを上書きすることに注意してください。

エンドポイントおよびオプションの詳細は、Braintree リファレンス(https://developers.braintreepayments.com/reference/overview)を参照してください。

ENDPOINT PREFIX ADDON

以下のエンドポイントは、以下のように接頭辞 addOn で呼び出すことができます。

> ***braintree://addOn/endpoint***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
all			List<com.braintreegateway.Addon>

エンドポイント接頭辞 アドレス

以下のエンドポイントは、以下のように接頭辞 address で呼び出すことができます。

> ***braintree://address/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
create		**customerId, request**	**com.braintreegateway.Result<com.braintreegateway.Address>**
delete		**customerId, id**	**com.braintreegateway.Result<com.braintreegateway.Address>**
find		**customerId, id**	**com.braintreegateway.Address**
update		**customerId, id, request**	**com.braintreegateway.Result<com.braintreegateway.Address>**

*表16.1 の URI オプション**address***

名前	タイプ
customerId	String
request	com.braintreegateway.AddressRequest
id	String

エンドポイント接頭辞 CLIENTTOKEN

以下のエンドポイントは、以下のように接頭辞 clientToken で呼び出すことができます。

> braintree://clientToken/endpoint?[options]

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
generate		request	String

表16.2 の URI オプションclientToken

名前	タイプ
request	com.braintreegateway.ClientTokenrequest

エンドポイント接頭辞 CREDITCARDVERIFICATION

以下のエンドポイントは、以下のように creditCardverification 接頭辞で呼び出すことができます。

> braintree://creditCardVerification/endpoint?[options]

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
find		id	com.braintreegateway.CreditCardVerification
search		query	com.braintreegateway.ResourceCollection<com.braintreegateway.CreditCardVerification>

表16.3 の URI オプションcreditCardVerification

名前	タイプ
id	**String**
query	**com.braintreegateway.CreditCardVerificationSearchRequest**

エンドポイント接頭辞 CUSTOMER

以下のエンドポイントは、以下のように接頭辞 customer で呼び出すことができます。

> ***braintree://customer/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
all			
create		request	com.braintreegateway.Result<com.braintreegateway.Customer>
delete		id	**com.braintreegateway.Result<com.braintreegateway.Customer>**
find		id	**com.braintreegateway.Customer**
search		query	**com.braintreegateway.ResourceCollection<com.braintreegateway.Customer>**
update		Id、request	**com.braintreegateway.Result<com.braintreegateway.Customer>**

表16.4 の URI オプションcustomer

名前	タイプ
id	文字列
request	**com.braintreegateway.CustomerRequest**

名前	タイプ
query	com.braintreegateway.**CustomerSearchRequest**

エンドポイント接頭辞 DISCOUNT

以下のエンドポイントは、以下のように接頭辞 discount を使用して呼び出すことができます。

> ***braintree://discount/endpoint***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
all			List<com.braintreegateway.Discount>

エンドポイント接頭辞 MERCHANTACCOUNT

以下のエンドポイントは、以下のように接頭辞 merchantAccount で呼び出すことができます。

> ***braintree://merchantAccount/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
create		request	com.braintreegateway.Result<com.braintreegateway.MerchantAccount>
find		id	com.braintreegateway.MerchantAccount
update		Id、request	com.braintreegateway.Result<com.braintreegateway.MerchantAccount>

表16.5 の URI オプションmerchantAccount

名前	タイプ
id	文字列
request	com.braintreegateway.**MerchantAccountRequest**

エンドポイント接頭辞 PAYMENTMETHOD

以下のエンドポイントは、以下のように接頭辞paymentMethod で呼び出すことができます。

> ***braintree://paymentMethod/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
create		request	com.braintreegateway.Result<com.braintreegateway.PaymentMethod>
delete		token	**com.braintreegateway.Result<com.braintreegateway.PaymentMethod>**
find		**token**	com.braintreegateway.PaymentMethod
update		**token**、request	com.braintreegateway.Result<com.braintreegateway.PaymentMethod>

表16.6 の URI オプションpaymentMethod

名前	タイプ
token	文字列
request	com.braintreegateway.PaymentMethodRequest

エンドポイント接頭辞 PAYMENTMETHODNONCE

以下のエンドポイントは、以下のように接頭辞paymentMethodNonce で呼び出すことができます。

> ***braintree://paymentMethodNonce/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
create		paymentMethodToken	com.braintreegateway.Result<com.braintreegateway.PaymentMethodNonce>
find		paymentMethodNonce	com.braintreegateway.PaymentMethodNonce

表16.7 の URI オプションpaymentMethodNonce

名前	タイプ
paymentMethodToken	文字列
paymentMethodNonce	**String**

エンドポイント接頭辞 PLAN

以下のエンドポイントは、以下のように接頭辞 プラン で呼び出すことができます。

> ***braintree://plan/endpoint***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
all			List<com.braintreegateway.Plan>

エンドポイント接頭辞 SETTLEMENTBATCHSUMMARY

以下のエンドポイントは、以下のように接頭辞settlementBatchSummary で呼び出すことができます。

> ***braintree://settlementBatchSummary/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
generate		request	com.braintreegateway.Result<**com.braintreegateway.SettlementBatchSummary>**

表16.8 の URI オプションsettlementBatchSummary

名前	タイプ
settlementDate	カレンダー
groupByCustomField	文字列

エンドポイント接頭辞 SUBSCRIPTION

以下のエンドポイントは、以下のようにsubscription 接頭辞で呼び出すことができます。

> ***braintree://subscription/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
cancel		id	com.braintreegateway.Result<**com.braintreegateway.Subscription>**
create		request	com.braintreegateway.Result<com.braintreegateway.**Subscription**>
delete		customerId, id	com.braintreegateway.Result<com.braintreegateway.**Subscription**>
find		id	**com.braintreegateway.Subscription**
retryCharge		subscriptionId, amount	**com.braintreegateway.Result<com.braintreegateway.Transaction>**
search		searchRequest	com.braintreegateway.ResourceCollection<com.braintreegateway.**Subscription**>
update		Id、request	com.braintreegateway.Result<com.braintreegateway.**Subscription**>

表16.9 の URI オプションsubscription

名前	タイプ
id	文字列
request	com.braintreegateway.SubscriptionRequest
customerId	文字列
subscriptionId	文字列
amount	BigDecimal
searchRequest	com.braintreegateway.SubscriptionSearchRequest.

エンドポイント接頭辞 TRANSACTION

以下のエンドポイントは、以下のように接頭辞 transaction で呼び出すことができます。

> ***braintree://transaction/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
cancelRelease		id	com.braintreegateway.Result<com.braintreegateway.**Transaction**>
cloneTransaction		id, cloneRequest	com.braintreegateway.Result<com.braintreegateway.**Transaction**>
クレジット		request	com.braintreegateway.Result<com.braintreegateway.**Transaction**>
find		id	com.braintreegateway.Transaction
holdInEscrow		id	com.braintreegateway.Result<com.braintreegateway.Transaction>
releaseFromEscrow		**id**	com.braintreegateway.**Result<com.braintreegateway.Transaction>**
refund		id、amount	com.braintreegateway.Result<com.braintreegateway.**Transaction**>
sale		request	**com.braintreegateway.Result<com.braintreegateway.Transaction>**
search		query	**com.braintreegateway.ResourceCollection<com.braintreegateway.Transaction>**
submitForPartialSettlement		id、amount	**com.braintreegateway.Result<com.braintreegateway.Transaction>**

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
submitForSettlement		Id, amount, request	**com.braintreegateway.Result<com.braintreegateway.Transaction>**
voidTransaction		id	**com.braintreegateway.Result<com.braintreegateway.Transaction>**

表16.10 の URI オプションtransaction

名前	タイプ
id	文字列
request	com.braintreegateway.TransactionCloneRequest
cloneRequest	com.braintreegateway.TransactionCloneRequest
amount	BigDecimal
query	com.braintreegateway.TransactionSearchRequest

エンドポイント接頭辞 WEBHOOKNOTIFICATION

以下のエンドポイントは、以下のように接頭辞 webhookNotification で呼び出すことができます。

> *braintree://webhookNotification/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
parse		署名、ペイロード	com.braintreegateway.WebhookNotification
verify		challenge	文字列

表16.11 の URI オプションwebhookNotification

名前	タイプ
signature	**String**
payload	文字列

名前	タイプ
challenge	String

コンシューマーエンドポイント

プロデューサーエンドポイントはいずれもコンシューマーエンドポイントとして使用できます。コンシューマーエンドポイントは、consumer. 接頭辞を持つScheduled Poll Consumer オプションを使用して、エンドポイント呼び出しをスケジュールできます。デフォルトでは、配列またはコレクションを返すコンシューマーエンドポイントは、要素ごとにエクスチェンジを1つ生成し、それらのルートはエクスチェンジごとに1回実行されます。この動作を変更するには、consumer.splitResults=true プロパティーを使用して、リストまたは配列全体の単一のエクスチェンジを返します。

メッセージヘッダー

すべてのURI オプションは、CamelBraintree. 接頭辞を持つプロデューサーエンドポイントのメッセージヘッダーで指定できます。

MESSAGE BODY

すべての結果メッセージ本文は、Braintree Java SDK によって提供されるオブジェクトを使用します。プロデューサーエンドポイントは、inBody エンドポイントパラメーターに受信メッセージボディーのオプション名を指定できます。

EXAMPLES

```
<?xml version="1.0"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
           xsi:schemaLocation="
             http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0
http://aries.apache.org/schemas/blueprint-cm/blueprint-cm-1.0.0.xsd
             http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
             http://camel.apache.org/schema/blueprint
http://camel.apache.org/schema/blueprint/camel-blueprint.xsd">


    <cm:property-placeholder id="placeholder" persistent-id="camel.braintree">
    </cm:property-placeholder>

    <bean id="braintree" class="org.apache.camel.component.braintree.BraintreeComponent">
        <property name="configuration">
            <bean class="org.apache.camel.component.braintree.BraintreeConfiguration">
                <property name="environment" value="${environment}"/>
                <property name="merchantId" value="${merchantId}"/>
                <property name="publicKey" value="${publicKey}"/>
                <property name="privateKey" value="${privateKey}"/>
            </bean>
        </property>
```

```
    </bean>

    <camelContext trace="true" xmlns="http://camel.apache.org/schema/blueprint"
id="braintree-example-context">
        <route id="braintree-example-route">
            <from uri="direct:generateClientToken"/>
            <to uri="braintree://clientToken/generate"/>
            <to uri="stream:out"/>
        </route>
    </camelContext>

</blueprint>
```

第17章 参照

コンポーネントの参照

Apache Camel 2.0 で利用可能

Browse コンポーネントは簡単な BrowsableEndpoint を提供します。これは、テスト、視覚化ツール、またはデバッグに役立ちます。エンドポイントに送信されたエクスチェンジはすべて参照できます。

URI 形式

```
browse:someName
```

someName には、エンドポイントを一意に識別する任意の文字列を指定できます。

例

*以下のルートでは **browse:** コンポーネントを挿入し、パススルーしているエクスチェンジを参照できます。*

```
from("activemq:order.in").to("browse:orderReceived").to("bean:processOrder");
```

これで、Java コード内から受信したエクスチェンジを検査できます。

```
private CamelContext context;

public void inspectRecievedOrders() {
    BrowsableEndpoint browse = context.getEndpoint("browse:orderReceived",
BrowsableEndpoint.class);
    List<Exchange> exchanges = browse.getExchanges();
    ...
    // then we can inspect the list of received exchanges from Java
    for (Exchange exchange : exchanges) {
        String payload = exchange.getIn().getBody();
        ...
    }
}
```

第18章 CACHE

18.1. キャッシュコンポーネント

Camel 2.1 以降で利用可能

キャッシュ コンポーネントを使用すると、EHCache をキャッシュ実装として使用してキャッシュ操作を実行できます。キャッシュ自体はオンデマンドで作成されます。その名前のキャッシュがすでに存在する場合は、単に元の設定で使用されます。

このコンポーネントは、プロデューサーおよびイベントベースのコンシューマーエンドポイントをサポートします。

Cache コンシューマーはイベントベースのコンシューマーであり、特定のキャッシュアクティビティーをリッスンして応答するために使用できます。既存のキャッシュから選択を行う必要がある場合は、キャッシュコンポーネントに定義されたプロセッサーを使用します。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-cache</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
cache://cacheName[?options]
```

URI にクエリーオプションは **?option=value&option=#beanRef&..** の形式で追加できます。

オプション

Cache コンポーネントは以下のオプションをサポートします。

名前	デフォルト値	説明
maxElementsInMemory	**1000**	定義されたキャッシュに格納できる要素の数

memoryStoreEvictionPolicy	**MemoryStoreEvictionPolicy.LFU**	定義されたキャッシュに保存される可能性のある要素の数。オプションは以下のとおりです。 • MemoryStoreEvictionPolicy.LFU - Least frequently used • MemoryStoreEvictionPolicy.LRU: 最近使用されたもの • MemoryStoreEvictionPolicy.FIFO - まずは作成時間で最も古い要素です。
overflowToDisk	**true**	キャッシュがディスクにオーバーフローできるかどうかを指定します。
eternal	**false**	要素が eternal かどうかを設定します。eternal の場合、タイムアウトは無視され、要素が期限切れになりません。
timeToLiveSeconds	**300**	作成時間と要素の有効期限の最大時間。要素が eternal でない場合にのみ使用されます。
timeToIdleSeconds	**300**	要素の有効期限が切れる前のアクセス間の最大時間
diskPersistent	**false**	ディスクストアが仮想マシンの再起動後も持続するかどうか。
diskExpiryThreadIntervalSeconds	**120**	ディスク期限切れスレッドの実行間隔（秒単位）。
cacheManagerFactory	**null**	**Camel 2.8:** EHCache **net.sf.ehcache.CacheManager** をインスタンス化して作成するカスタムファクトリーを使用する場合。**タイプ：** abstract org.apache.camel.component.cache.CacheManagerFactory

eventListenerRegistry	**null**	camel 2.8: 新しいキャッシュすべてに EHCache **net.sf.ehcache.event.CacheEventListener** の一覧を設定します。EHCache xml 設定のキャッシュごとに定義する必要はありません。**Type:** org.apache.camel.component.cache.CacheEventListenerRegistry
cacheLoaderRegistry	**null**	camel 2. **8:** 新しいキャッシュごとに EHCache **net.sf.ehcache.loader.CacheLoader** Wrapper を拡張する **org.apache.camel.component.cache.CacheLoaderWrapper** の一覧を設定します。これは、EHCache xml 設定のキャッシュごとに定義する必要はありません。**タイプ :** org.apache.camel.component.cache.CacheLoaderRegistry
key	**null**	Camel 2.10: デフォルトでは キャッシュキーを使用して を設定します。キーがメッセージヘッダーに提供されている場合は、ヘッダーのキーが優先されます。
operation	**null**	Camel 2.10: デフォルトでは キャッシュ操作を使用して設定します。メッセージヘッダーで操作を行うと、ヘッダーからの操作が優先されます。
objectCache	**false**	Camel 2.10: シリアライズできないオブジェクトをキャッシュに格納できるようにするかどうか。このオプションを有効にすると、ディスクへのオーバーフローも有効にできません。
configurationFile		Camel 2.13/2.12.3: 使用する **ehcache.xml** ファイルの場所を設定します（例：クラスパスからロードする **classpath:com/foo/mycache.xml** など）。設定が指定されていない場合は、EHCache のデフォルト設定が使用されます。

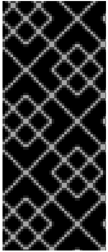
configuration		カスタムの **org.apache.camel.component.cache.CacheConfiguration** 設定を使用します。

キャッシュコンポーネントのオプション

名前	デフォルト値	説明
configuration		カスタムの **org.apache.camel.component.cache.CacheConfiguration** 設定を使用します。
cacheManagerFactory		カスタム **org.apache.camel.component.cache.CacheManagerFactory** を使用するには、を使用します。
configurationFile		Camel 2.13/2.12.3: 使用する **ehcache.xml** ファイルの場所を設定します（例：クラスパスからロードする **classpath:com/foo/mycache.xml** など）。設定が指定されていない場合は、EHCache のデフォルト設定が使用されます。

Message Headers Camel 2.8 以降

ヘッダー	説明
CamelCacheOperation	キャッシュで実行される操作。有効なオプションはです。 • **CamelCacheGet** • **CamelCacheCheck** • **CamelCacheAdd** • **CamelCacheUpdate** • **CamelCacheDelete** • **CamelCacheDeleteAll**
CamelCacheKey	キャッシュに Message を保存するために使用されるキャッシュキー。**CamelCacheOperation** の場合、キャッシュキーは任意になります。 **CamelCacheDeleteAll**



CAMEL 2.8 でのヘッダーの変更

ヘッダー名とサポートされる値は、接頭辞 CamelCache とユースケースが混在するように変更されました。これにより、他のヘッダーから個別のヘッダーを簡単に特定し、維持することができます。CacheConstants の変数名は変更されませんが、値のみが変更されました。また、キャッシュ操作の実行後に、これらのヘッダーがエクスチェンジから削除されるようになりました。

CamelCacheAdd および CamelCacheUpdate 操作は追加のヘッダーをサポートします。

ヘッダー	タイプ	説明
CamelCacheTimeToLive	**整数**	Camel 2.11: 存続時間（秒単位）
CamelCacheTimeToIdle	**整数**	Camel 2.11: アイドル時間（秒単位）
CamelCacheEternal	**ブール値**	Camel 2.11: コンテンツの対象となるかどうか。

キャッシュプロデューサー

データをキャッシュに送信するには、エクスチェンジのペイロードを既存のキャッシュまたは作成されたオンデマンドキャッシュに保存する必要があります。この作業の仕組み

- *上記のメッセージ交換ヘッダーの設定。*
- *メッセージ交換ボディーにキャッシュに送信されるメッセージが含まれていることの確認*

キャッシュコンシューマー

キャッシュからデータを受信するには、CacheConsumer がイベントリスナーを使用して既存または作成されたオンデマンドキャッシュをリッスンし、キャッシュアクティビティー（つまり CamelCacheGet/CamelCacheUpdate/CamelCacheDelete/CamelCacheDelete/CamelCacheDelete/Ca が発生した場合に自動通知を受け取る必要があります。このような作業が行われているとき

- *追加/更新されたばかりのペイロードが含まれる Message Exchange ヘッダーと、メッセージ交換ヘッダーを含むエクスチェンジが配置および送信されます。*
- *CamelCacheDeleteAll 操作の場合、Message Exchange Header CamelCacheKey および Message Exchange Body は入力されません。*

キャッシュプロセッサー

nice プロセッサーのセットには、キャッシュルックアップを実行し、のペイロードコンテンツを選択的に置き換える機能があります。

- *ボディー*
- *token*
- *XPath レベル*

例1: キャッシュの設定

```
from("cache://MyApplicationCache" +
          "?maxElementsInMemory=1000" +
          "&memoryStoreEvictionPolicy=" +
             "MemoryStoreEvictionPolicy.LFU" +
          "&overflowToDisk=true" +
          "&eternal=true" +
          "&timeToLiveSeconds=300" +
          "&timeToIdleSeconds=true" +
          "&diskPersistent=true" +
          "&diskExpiryThreadIntervalSeconds=300")
```

例2: キーのキャッシュへの追加

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
     from("direct:start")
     .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_ADD))
     .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
     .to("cache://TestCache1")
   }
};
```

例2: キャッシュ内の既存のキーの更新

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
     from("direct:start")
     .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_UPDATE))
     .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
     .to("cache://TestCache1")
   }
};
```

例3: キャッシュ内の既存のキーの削除

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
     from("direct:start")
     .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_DELETE))
     .setHeader(CacheConstants.CACHE_KEY", constant("Ralph_Waldo_Emerson"))
     .to("cache://TestCache1")
   }
};
```

例4: キャッシュ内のすべての既存キーの削除

```
RouteBuilder builder = new RouteBuilder() {
```

```
    public void configure() {
     from("direct:start")
     .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_DELETEALL))
     .to("cache://TestCache1");
    }
};
```

例5: キャッシュに登録されている変更のプロセッサーおよびその他のプロデューサーへの通知

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
     from("cache://TestCache1")
     .process(new Processor() {
        public void process(Exchange exchange)
               throws Exception {
           String operation = (String)
exchange.getIn().getHeader(CacheConstants.CACHE_OPERATION);
           String key = (String) exchange.getIn().getHeader(CacheConstants.CACHE_KEY);
           Object body = exchange.getIn().getBody();
           // Do something
        }
     })
   }
};
```

例6: プロセッサーを使用したペイロードをキャッシュ値で選択的に置き換える

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
     //Message Body Replacer
     from("cache://TestCache1")
     .filter(header(CacheConstants.CACHE_KEY).isEqualTo("greeting"))
     .process(new CacheBasedMessageBodyReplacer("cache://TestCache1","farewell"))
     .to("direct:next");

    //Message Token replacer
    from("cache://TestCache1")
    .filter(header(CacheConstants.CACHE_KEY).isEqualTo("quote"))
    .process(new CacheBasedTokenReplacer("cache://TestCache1","novel","#novel#"))
    .process(new CacheBasedTokenReplacer("cache://TestCache1","author","#author#"))
    .process(new CacheBasedTokenReplacer("cache://TestCache1","number","#number#"))
    .to("direct:next");

    //Message XPath replacer
    from("cache://TestCache1").
    .filter(header(CacheConstants.CACHE_KEY).isEqualTo("XML_FRAGMENT"))
    .process(new CacheBasedXPathReplacer("cache://TestCache1","book1","/books/book1"))
    .process (new CacheBasedXPathReplacer("cache://TestCache1","book2","/books/book2"))
    .to("direct:next");
   }
};
```

例7: キャッシュからのエントリーの取得

```
from("direct:start")
    // Prepare headers
    .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_GET))
    .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson")).
    .to("cache://TestCache1").
    // Check if entry was not found
    .choice().when(header(CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull()).
        // If not found, get the payload and put it to cache
        .to("cxf:bean:someHeavyweightOperation").
        .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_ADD))
        .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
        .to("cache://TestCache1")
    .end()
    .to("direct:nextPhase");
```

例8: キャッシュ内のエントリーの確認

注記：CHECK コマンドは、キャッシュにエントリーの存在をテストしますが、本文にメッセージを配置しません。

```
from("direct:start")
    // Prepare headers
    .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_CHECK))
    .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson")).
    .to("cache://TestCache1").
    // Check if entry was not found
    .choice().when(header(CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull()).
        // If not found, get the payload and put it to cache
        .to("cxf:bean:someHeavyweightOperation").
        .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_ADD))
        .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
        .to("cache://TestCache1")
    .end();
```

EHCache の管理

ehcache には、JMX からの独自の統計と管理があります。

以下は、Spring アプリケーションコンテキストで JMX 経由で公開する方法のスニペットです。

```
<bean id="ehCacheManagementService"
class="net.sf.ehcache.management.ManagementService" init-method="init" lazy-init="false">
  <constructor-arg>
    <bean class="net.sf.ehcache.CacheManager" factory-method="getInstance"/>
  </constructor-arg>
  <constructor-arg>
    <bean class="org.springframework.jmx.support.JmxUtils" factory-
method="locateMBeanServer"/>
```

```
  </constructor-arg>
  <constructor-arg value="true"/>
  <constructor-arg value="true"/>
  <constructor-arg value="true"/>
  <constructor-arg value="true"/>
</bean>
```

当然ながら、Java で同じことを実行できます。

```
ManagementService.registerMBeans(CacheManager.getInstance(), mbeanServer, true, true,
true, true);
```

このようにキャッシュヒット、ミス、メモリー内ヒット、ディスクヒット、サイズの統計を取得できます。また、CacheConfiguration パラメーターをオンザフライで変更することもできます。

Cache replication Camel 2.8+

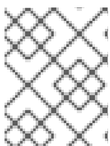
Camel Cache コンポーネントは、RMI、JGroups、JMS、Cache Server などの異なるレプリケーションメカニズムを使用して、サーバーノードにキャッシュを分散できます。

これを機能させるには、以下の 2 つの方法があります。

1. **ehcache.xml** を手動で設定できます。または、
2. 以下の 3 つのオプションを設定できます。
   - **cacheManagerFactory**
   - **eventListenerRegistry**
   - **cacheLoaderRegistry**

最初のオプションを使用した Camel キャッシュレプリケーションの設定は、すべてのキャッシュを個別に設定する必要があるため、もう少し難しい作業になります。したがって、キャッシュの名前がすべて不明な場合は、**ehcache.xml** を使用することは適切ではありません。

2 つ目のオプションは、キャッシュごとにオプションを定義する必要がないため、多くの異なるキャッシュを使用する場合により適しています。これは、レプリケーションオプションが **CacheManager** ごとおよび **CacheEndpoint** ごとに設定されるためです。また、開発フェーズでキャッシュ名がわからない場合は唯一の方法です。



注記

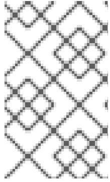
Camel Cache レプリケーションメカニズムをよりよく理解するには、EHCache マニュアル を読むと便利です。

例：JMS キャッシュレプリケーション

JMS レプリケーションは最も強力でセキュアなレプリケーション方法です。Camel キャッシュレプリケーションと併用すると、よりシンプルになります。例と して、別のページ を参照し てください。

18.2. CACHEREPLICATIONJMSEXAMPLE

例：JMS キャッシュレプリケーション



注記

この例ではまだ終了していないことに注意してください。これは、実際のライフサイクル例ではなく OSGi iTest に基づいています。しかし、すべての Camel Cache Riders にとって非常に良好なポイントであるかは問題ありません。

JMS レプリケーションは最も強力でセキュアな方法です。Camel Cache レプリケーションオプションとともに使用する場合は、最も簡単な方法になります。この基本的な例は、キャッシュレプリケーションを機能させるために必要ないくつかの重要な手順に分けられます。

最初のステップでは、CacheManagerFactory の独自の実装を作成します。

```
public class TestingCacheManagerFactory extends CacheManagerFactory {

[...]

    //This constructor is very useful when using Camel with Spring/Blueprint
    public TestingCacheManagerFactory(String xmlName,
            TopicConnection replicationTopicConnection, Topic replicationTopic,
            QueueConnection getQueueConnection, Queue getQueue) {
        this.xmlName = xmlName;
        this.replicationTopicConnection = replicationTopicConnection;
        this.replicationTopic = replicationTopic;
        this.getQueue = getQueue;
        this.getQueueConnection = getQueueConnection;
    }

    @Override
    protected synchronized CacheManager createCacheManagerInstance() {
        //Singleton
        if (cacheManager == null) {
            cacheManager = new
WrappedCacheManager(getClass().getResourceAsStream(xmlName));
        }

        return cacheManager;
    }

    //Wrapping Ehcache's CacheManager to be able to add JMSCacheManagerPeerProvider
    public class WrappedCacheManager extends CacheManager {
        public WrappedCacheManager(InputStream xmlConfig) {
            super(xmlConfig);
            JMSCacheManagerPeerProvider jmsCMPP = new JMSCacheManagerPeerProvider(this,
                    replicationTopicConnection,
                    replicationTopic,
                    getQueueConnection,
                    getQueue,
                    AcknowledgementMode.AUTO_ACKNOWLEDGE,
                    true);
            cacheManagerPeerProviders.put(jmsCMPP.getScheme(), jmsCMPP);
            jmsCMPP.init();
        }
    }
}
```

次に、CacheLoaderWrapper の独自の実装を作成します。最も簡単な方法は以下のとおりです。

```
public class WrappedJMSCacheLoader implements CacheLoaderWrapper {

[...]

    //This constructor is very useful when using Camel with Spring/Blueprint
    public WrappedJMSCacheLoader(QueueConnection getQueueConnection,
            Queue getQueue, AcknowledgementMode acknowledgementMode,
            int timeoutMillis) {
        this.getQueueConnection = getQueueConnection;
        this.getQueue = getQueue;
        this.acknowledgementMode = acknowledgementMode;
        this.timeoutMillis = timeoutMillis;
    }

    @Override
    public void init(Ehcache cache) {
        jmsCacheLoader = new JMSCacheLoader(cache, defaultLoaderArgument,
                getQueueConnection, getQueue, acknowledgementMode,
                timeoutMillis);
    }

    @Override
    public CacheLoader clone(Ehcache arg0) throws CloneNotSupportedException {
        return jmsCacheLoader.clone(arg0);
    }

    @Override
    public void dispose() throws CacheException {
        jmsCacheLoader.dispose();
    }

[...]

}
```

3 番目のステップで、Camel Cache のオプション（値を準備）することができます。

- *cacheManagerFactory*
- *eventListenerRegistry*
- *cacheLoaderRegistry*

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

    <bean id="queueConnection1" factory-bean="amqCF" factory-
method="createQueueConnection" class="javax.jms.QueueConnection" />
```

```
    <bean id="topicConnection1" factory-bean="amqCF" factory-
method="createTopicConnection" class="javax.jms.TopicConnection" />
    <bean id="queue1" class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg ref="getQueue" />
    </bean>
    <bean id="topic1" class="org.apache.activemq.command.ActiveMQTopic">
        <constructor-arg ref="getTopic" />
    </bean>

    <bean id="jmsListener1" class="net.sf.ehcache.distribution.jms.JMSCacheReplicator">
        <constructor-arg index="0" value="true" />
        <constructor-arg index="1" value="true" />
        <constructor-arg index="2" value="true" />
        <constructor-arg index="3" value="true" />
        <constructor-arg index="4" value="false" />
        <constructor-arg index="5" value="0" />
    </bean>

    <bean id="jmsLoader1" class="my.cache.replication.WrappedJMSCacheLoader">
        <constructor-arg index="0" ref="queueConnection1" />
        <constructor-arg index="1" ref="queue1" />
        <constructor-arg index="2" value="AUTO_ACKNOWLEDGE" />
        <constructor-arg index="3" value="30000" />
    </bean>

    <bean id="cacheManagerFactory1"
class="my.cache.replication.TestingCacheManagerFactory">
        <constructor-arg index="0" value="ehcache_jms_test.xml" />
        <constructor-arg index="1" ref="topicConnection1" />
        <constructor-arg index="2" ref="topic1" />
        <constructor-arg index="3" ref="queueConnection1" />
        <constructor-arg index="4" ref="queue1" />
    </bean>

    <bean id="eventListenerRegistry1"
class="org.apache.camel.component.cache.CacheEventListenerRegistry">
        <constructor-arg>
            <list>
                <ref bean="jmsListener1" />
            </list>
        </constructor-arg>
    </bean>

    <bean id="cacheLoaderRegistry1"
class="org.apache.camel.component.cache.CacheLoaderRegistry">
        <constructor-arg>
            <list>
                <ref bean="jmsLoader1"/>
            </list>
        </constructor-arg>
    </bean>
</beans>
```

最後のステップは、Cache コンポーネントを使用して一部のルートを定義することです。

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:camel="http://camel.apache.org/schema/spring"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

    <bean id="getQueue" class="java.lang.String">
        <constructor-arg value="replicationGetQueue" />
    </bean>

    <bean id="getTopic" class="java.lang.String">
        <constructor-arg value="replicationTopic" />
    </bean>

    <!-- Import the xml file explained at step three -->
    <import resource="JMSReplicationCache1.xml"/>

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <camel:endpoint id="fooCache1" uri="cache:foo?
cacheManagerFactory=#cacheManagerFactory1&ventListenerRegistry=#eventListenerRegistr
y1&acheLoaderRegistry=#cacheLoaderRegistry1"/>

        <camel:route>
            <camel:from uri="direct:addRoute"/>
            <camel:setHeader headerName="CamelCacheOperation">
                <camel:constant>CamelCacheAdd</camel:constant>
            </camel:setHeader>
            <camel:setHeader headerName="CamelCacheKey">
                <camel:constant>foo</camel:constant>
            </camel:setHeader>
            <camel:to ref="fooCache1"/>
        </camel:route>

    </camelContext>

    <bean id="amqCF" class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="vm://localhost?broker.persistent=false"/>
    </bean>

    <bean id="activemq" class="org.apache.camel.component.jms.JmsComponent">
        <property name="connectionFactory">
            <ref bean="amqCF"/>
        </property>
    </bean>

</beans>
```

第19章 CDI

CDI コンポーネント

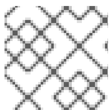
camel-cdi コンポーネントは CDI 統合を提供します。

以下の例は、関連付けられたルートでコンテキストを提供し、使用する方法を示しています。

```
@Startup
@ApplicationScoped
@ContextName("cdi-context")
public class MyRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("direct:start").transform(body().prepend("Hi"));
    }
}
```

以下の例は、Camel コンテキストをプライベートフィールドに注入する方法を示しています。

```
@Inject
@ContextName("cdi-context")
private CamelContext camelctx;
```



注記

CDI 環境の Apache Camel に関する詳細は、Camel CDI を参照してください。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

第20章 CASSANDRA

CAMEL CASSANDRA コンポーネント

Camel 2.15 以降で利用可能

Apache Cassandra は、コモディティーハードウェア上で大量のデータを処理するように設計されたオープンソースの NoSQL データベースです。Amazon の DynamoDB と同様に、Cassandra にはピアツーピアおよびマスターなしのアーキテクチャーがあり、単一障害点を回避し、高可用性を確保します。Google の BigTable と同様に、CQL と呼ばれる SQL のような API からアクセス可能な列ファミリーを使用して Cassandra データを構築します。

このコンポーネントは、(Trift API ではなく) CQL3 API を使用して Cassandra 2.0+ を統合することを目指しています。これは、DataStax が提供する Cassandra Java Driver に基づいています。

*Maven ユーザーは、以下の依存関係を **pom.xml** に追加する必要があります。*

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cassandraql</artifactId>
  <version>x.y.z</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

エンドポイントは Cassandra 接続を開始するか、既存の接続を使用できます。

URI	説明
cql:localhost/keyspace	単一のホスト、デフォルトポート（通常はテスト用）
cql:host1,host2/keyspace	マルチホスト、デフォルトポート
cql:host1,host2:9042/keyspace	マルチホスト、カスタムポート
cql:host1,host2	デフォルトのポートおよびキースペース
cql:bean:sessionRef	提供されるセッション参照
cql:bean:clusterRef/keyspace	提供されるクラスター参照

Cassandra 接続(SSL オプション、プーリングオプション、負荷分散ポリシー、再試行ポリシー、再接続ポリシー)を微調整するには、独自のクラスターインスタンスを作成し、Camel エンドポイントに指定します。

エンドポイントオプション

オプション	デフォルト	説明
clusterName		Cluster name
username and password		セッション認証
cql		CQL クエリー。メッセージヘッダーで上書きできます。
consistencyLevel		**ANY**, **ONE**, **TWO**, **QUORUM**, **LOCAL_QUORUM**…
prepareStatements	**true**	準備済みステートメント（デフォルト）を使用する（デフォルト）かどうか
resultSetConversionStrategy	**ALL**	ResultSet がメッセージボディーに変換および変換する方法 **ALL**、**ONE**、**LIMIT_10**、および **LIMIT_100**…

メッセージ

受信メッセージ

*Camel Cassandra エンドポイントは、クエリーパラメーターとして CQL ステートメントにバインドされる単純なオブジェクト(**Object** または **Object[]** または **Collection<Object>**)があることを想定しています。メッセージボディーが null または空の場合、CQL クエリーはバインディングパラメーターなしで実行されます。*

ヘッダー:

- ***CamelCqlQuery**（オプション、**String**）: プレーンな文字列として、または **RegularStatement**を使用してビルドした CQL クエリー。**QueryBuilder***

送信メッセージ

*Camel Cassandra エンドポイントは、**resultSetConversionStrategy** に応じて 1 つまたは複数の Cassandra Row オブジェクトを生成します。*

- ***List<Row> resultSetConversionStrategy** が **ALL** または の場合 **LIMIT_[0-9]+***
- *single **Row** if **resultSetConversionStrategy** **ONE***
- ***resultSetConversionStrategy** がカスタム実装である場合、それ以外 **ResultSetConversionStrategy***

リポジトリー

Cassandra は、べき等性と集約 EIP のメッセージキーまたはメッセージを格納するために使用できます。

Cassandra はキューイングのユースケースに最適なツールではない場合があります。Cassandra のアンチパターンのキューとデータセットなどのキューをお読みください。これらのテーブルには LeveledCompaction および小さい GC grace 設定を使用し、廃棄した行をすぐに削除できるようにすることが推奨されます。

IDEMPOTENT リポジトリー

***NamedCassandraIdempotentRepository** は、以下のような Cassandra テーブルにメッセージキーを保存します。*

```
CREATE TABLE CAMEL_IDEMPOTENT (
  NAME varchar,   -- Repository name
  KEY varchar,    -- Message key
  PRIMARY KEY (NAME, KEY)
) WITH compaction = {'class':'LeveledCompactionStrategy'}
  AND gc_grace_seconds = 86400;
```

このリポジトリーの実装では、軽量のトランザクション（比較およびセットとも呼ばれます）を使用し、Cassandra 2.0.7+ が必要です。

*または、**CassandraIdempotentRepository** には **NAME** 列がなく、別のデータモデルを使用するように拡張することもできます。*

オプション	デフォルト	説明
table	**CAMEL_IDEMPOTENT**	テーブル名
pkColumns	**NAME**, **KEY**	プライマリーキー列
name		Repository name, value used for **NAME** column （リポジトリー名、列に使用される値）
ttl		キーの有効期間
writeConsistencyLevel		キーの挿入/削除に使用する一貫性レベル： **ANY**, **ONE**, **TWO**, **QUORUM**, -----------|-----… **LOCAL_QUORUM**
readConsistencyLevel		キーの読み取り/チェックに使用する一貫性レベル： **ONE**, **TWO**, **QUORUM**, **LOCAL_QUORUM**…

集約リポジトリー

***NamedCassandraAggregationRepository** は、以下のように相関キーでエクスチェンジを Cassandra テーブルに保存します。*

```
CREATE TABLE CAMEL_AGGREGATION (
```

```
  NAME varchar,       -- Repository name
  KEY varchar,        -- Correlation id
  EXCHANGE_ID varchar, -- Exchange id
  EXCHANGE blob,      -- Serialized exchange
  PRIMARY KEY (NAME, KEY)
) WITH compaction = {'class':'LeveledCompactionStrategy'}
  AND gc_grace_seconds = 86400;
```

または、CassandraAggregationRepository には NAME 列がなく、別のデータモデルを使用するように拡張することもできます。

オプション	デフォルト	説明
table	**CAMEL_AGGREGATION**	テーブル名
pkColumns	**NAME**,**KEY**	プライマリーキー列
exchangeIdColumn	**EXCHANGE_ID**	エクスチェンジ ID 列
exchangeColumn	**EXCHANGE**	コンテンツ列の交換
name		Repository name, value used for **NAME** column （リポジトリー名、列に使用される値）
ttl		存続時間の交換
writeConsistencyLevel		交換の挿入/削除に使用する一貫性レベル： **ANY**, **ONE**, **TWO**, **QUORUM**, ----------|-----… **LOCAL_QUORUM**
readConsistencyLevel		交換の読み取り/チェックに使用する一貫性レベル： **ONE**, **TWO**, **QUORUM**, **LOCAL_QUORUM**…

第21章 CHUNK

チャンクコンポーネント

Camel 2.15 以降で利用可能

chunk: コンポーネントを使用すると、Chunk テンプレートを使用してメッセージを処理できます。これは、Templating を使用してリクエストの応答を生成する場合に理想的です。

*Maven ユーザーは、このコンポーネントの以下の依存関係を **pom.xml** に追加する必要があります。*

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-chunk</artifactId>
<version>x.x.x</version> <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
chunk:templateName[?options]
```

templateName は、呼び出すテンプレートのクラスパスローカル URI に置き換えます。

*以下の形式で URI にクエリーオプションを追加できます。 **?option=value&option=value&...***

オプション

オプション	デフォルト	説明
allowContextMapAll (producer)	**false**	コンテキストマップが前詳細へのアクセスを許可するかどうかを設定します。デフォルトでは、メッセージの本文とヘッダーにのみアクセスできます。このオプションは、現在の Exchange および CamelContext へのフルアクセスに対して有効にできます。これを行うと、CamelContext API の全機能へのアクセスが開かれるため、潜在的なセキュリティーリスクが発生します。

allowTemplateFromHeader (producer)	**false**	ヘッダーのリソーステンプレートの使用を許可するかどうか (デフォルトは false)。**このオプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼されていないコンテンツやユーザー派生コンテンツが含まれている場合、これは最終的にエンドアプリケーションの信頼性および完全性に影響を与える可能性があるため、このオプションは注意して使用してください。**
encoding	**null**	リソースコンテンツの文字エンコーディング。
themesFolder	**null**	テンプレート名をスキャンする代替フォルダー。
themeSubfolder	**null**	themeFolder パラメーターが設定されている場合にテンプレート名をスキャンする代替サブフォルダー。
themeLayer	**null**	テンプレートとして使用するテンプレートファイルの特定のレイヤー。
extension	**null**	themeFolder および themeSubfolder が設定されている場合にテンプレート名をスキャンする代替拡張。

チャンクコンポーネントは、拡張機能 **.chtml** ***または*** **cxml を持つ themes フォルダーで特定のテンプレートを検索します。*別のフォルダーまたは拡張機能を指定する必要がある場合は、上記の特定のオプションを使用する必要があります。***

チャンクコンテキスト

Camel は、Chunk コンテキストでエクスチェンジ情報を提供します（ MapExchange は以下のように転送されます。

key	value
exchange	**Exchange** 自体。
exchange.properties	**Exchange** プロパティー。
headers	In メッセージのヘッダー。
camelContext	Camel コンテキスト。

request	In メッセージ。
body	In メッセージのボディー。
response	Out メッセージ(InOut メッセージ交換パターンにのみ有効)。

動的テンプレート

Camel は 2 つのヘッダーを提供し、テンプレートまたはテンプレートコンテンツ自体に異なるリソースの場所を定義できます。これらのヘッダーのいずれかが設定されている場合、Camel は設定されたエンドポイントでこれを使用します。これにより、ランタイム時に動的テンプレートを指定できます。

ヘッダー	タイプ	説明	サポートバージョン
ChunkConstants.CHUNK_RESOURCE_URI	文字列	設定されたエンドポイントの代わりに使用するテンプレートリソースの URI。	
ChunkConstants.CHUNK_TEMPLATE	文字列	設定されたエンドポイントの代わりに使用するテンプレート。	

サンプル

たとえば、以下のように使用できます。

```
from("activemq:My.Queue").
to("chunk:template");
```

*Chunk テンプレートを使用して InOut メッセージエクスチェンジ(**JMSReplyTo** ヘッダーがある)のメッセージの応答を形成するには、以下を実行します。*

InOnly を使用してメッセージを消費し、別の宛先に送信する場合は、以下を使用します。

```
from("activemq:My.Queue").
to("chunk:template").
to("activemq:Another.Queue");
```

以下のように、ヘッダーを介してコンポーネントを動的に使用するテンプレートを指定できます。

```
from("direct:in").
setHeader(ChunkConstants.CHUNK_RESOURCE_URI).constant("template").
to("chunk:dummy?allowTemplateFromHeader=true");
```

警告

allowTemplateFromHeader オプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼できないコンテンツまたはユーザー派生コンテンツが含まれる場合、これは最終的に、エンドアプリケーションの確実性と整合性に及ぼす可能性があるため、このオプションを使用してください。

Chunk コンポーネントオプションの例を以下に示します。

```
from("direct:in").
to("chunk:file_example?
themeFolder=template&themeSubfolder=subfolder&extension=chunk");
```

この例では、Chunk コンポーネントは template/subfolder フォルダーの file_example.chunk ファイルを検索します。

電子メールのサンプル

この例では、注文確認メールに Chunk テンプレートを使用します。メールテンプレートは、以下のように Chunk でレイアウトされます。

```
Dear {$headers.lastName}, {$headers.firstName}

Thanks for the order of {$headers.item}.

Regards Camel Riders Bookstore
{$body}
```

第22章 クラス

クラスコンポーネント

Apache Camel 2.4 から利用可能

class: コンポーネントは Bean をメッセージエクスチェンジにバインドします。これはBean コンポーネントと同じように機能しますが、レジストリーから Bean を検索する代わりに、クラス名に基づいて Bean を作成します。

URI 形式

```
class:className[?options]
```

className は、Bean として作成および使用する完全修飾クラス名です。

オプション

名前	タイプ	デフォルト	説明
メソッド	**文字列**	**null**	Bean が呼び出されるメソッド名。指定のない場合は、Apache Camel はメソッド自体を選択しようとします。曖昧さが発生すると、例外が発生します。詳細は、Bean Binding を参照してください。
multiParameterArray	**boolean**	**false**	メッセージボディーから渡されるパラメーターを処理する方法。**true** の場合、In メッセージボディーはパラメーターの配列である必要があります。
bean.xxx		**null**	**Camel 2.17:** クラス名から create bean インスタンスで追加オプションを設定します。たとえば、Bean で **foo** オプションを設定するには、**bean.foo=123** を使用します。

URI にクエリーオプションは ***?option=value&option=value&..*** *の形式で追加できます。*

使用

クラス コンポーネントを Bean コンポーネントとして使用するだけで、代わりに完全修飾クラス名を指定してクラスコンポーネントを使用します。たとえば、MyFooBean を使用するには、次のようにする必要があります。

```
from("direct:start").to("class:org.apache.camel.component.bean.MyFooBean").to("mock:result");
```

MyFooBean で呼び出すメソッドを指定することもできます（例： hello ）。

```
from("direct:start").to("class:org.apache.camel.component.bean.MyFooBean?method=hello").to("mock:result");
```

作成されたインスタンスでのプロパティーの設定

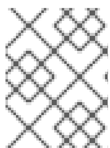
エンドポイント URI では、作成されたインスタンスに設定するプロパティーを指定できます。たとえば、setPrefix メソッドがある場合です。

```
// Camel 2.17 onwards
from("direct:start")
   .to("class:org.apache.camel.component.bean.MyPrefixBean?bean.prefix=Bye")
   .to("mock:result");

// Camel 2.16 and older
from("direct:start")
   .to("class:org.apache.camel.component.bean.MyPrefixBean?prefix=Bye")
   .to("mock:result");
```

また、# 構文を使用して、レジストリーで検索するプロパティーを参照することもできます。

```
// Camel 2.17 onwards
from("direct:start")
   .to("class:org.apache.camel.component.bean.MyPrefixBean?bean.cool=#foo")
   .to("mock:result");

// Camel 2.16 and older
from("direct:start")
   .to("class:org.apache.camel.component.bean.MyPrefixBean?cool=#foo")
   .to("mock:result");
```

これは、ID foo でレジストリーから Bean を検索し、MyPrefixBean クラスの作成されたインスタンスで setCool メソッドを呼び出します。



注記

クラス コンポーネントがほぼ同じように機能する Bean コンポーネントの詳細を参照してください。

- Bean
- Bean バインディング
- Bean インテグレーション

第23章 CMIS

CMIS コンポーネント

cmis コンポーネントは Apache Chemistry クライアント API を使用し、CMIS 準拠のコンテンツリポジトリーからノードを追加/読み取りできます。

URI 形式

> ***cmis://cmisServerUrl[?options]***

URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
queryMode	**false**	プロデューサー	true の場合、はメッセージボディーから cmis クエリーを実行し、結果を返します。そうでない場合は、cmis リポジトリーにノードを作成します。
query	**文字列**	コンシューマー	リポジトリーに対して実行する cmis クエリー。指定しない場合、コンシューマーはコンテンツツリーを再帰的に反復することで、コンテンツリポジトリーからすべてのノードを取得します。
username	**null**	両方	cmis リポジトリーのユーザー名
password	**null**	両方	cmis リポジトリーのパスワード
repositoryId	**null**	両方	使用するリポジトリーの ID。指定しない場合、最初に利用可能なリポジトリーが使用されます。
pageSize	**100**	両方	ページごとに取得するノード数
readCount	**0**	両方	読み取るノードの最大数

readContent	**false**	両方	true に設定すると、ドキュメントノードのコンテンツはプロパティーに加えて取得されます。

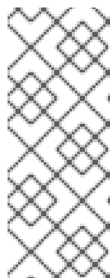
使用方法

プロデューサーによって評価されるメッセージヘッダー

ヘッダー	デフォルト値	説明
CamelCMISFolderPath	/	実行時に使用する現在のフォルダー。指定しない場合はルートフォルダーを使用します。
CamelCMISRetrieveContent	**false**	**queryMode** では、このヘッダーはプロデューサーがドキュメントノードの内容を取得するように強制します。
CamelCMISReadSize	**0**	読み取るノードの最大数。
cmis:path	**null**	**CamelCMISFolderPath** が設定されていない場合、はこの cmis プロパティーからノードのパスを見つけようとします。これは名前になります。
cmis:name	**null**	**CamelCMISFolderPath** が設定されていない場合、はこの cmis プロパティーからノードのパスを見つけようとします。これは path です。
cmis:objectTypeId	**null**	ノードのタイプ
cmis:contentStreamMimeType	**null**	ドキュメントに設定する mimetype

プロデューサー操作のクエリー中に設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelCMISResultCount	**整数**	クエリーから返されるノード数。



POM.XML

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cmis</artifactId>
  <version>${camel-version}</version>
</dependency>
```

${camel-version} *は、実際のバージョンの Camel (2.11 以降)に置き換える必要があります。*

第24章 COMETD

COMETD コンポーネント

cometd: コンポーネントは、Extras td/bayeux プロトコル のjetty 実装と連携するためのトランスポートです。このコンポーネントを dojo ツールキットライブラリーと組み合わせて使用すると、AJAX ベースのメカニズムを使用して Apache Camel メッセージを直接ブラウザーにプッシュできます。

URI 形式

```
cometd://host:port/channelName[?options]
```

channelName は、Apache Camel エンドポイントによってサブスクライブできるトピックを表します。

例

```
cometd://localhost:8080/service/mychannel
cometds://localhost:8443/service/mychannel
```

cometds: *は SSL が設定されたエンドポイントを表します。*

オプション

名前	デフォルト値	説明
resourceBase		Web リソースまたはクラスパスのルートディレクトリー。コンポーネントがファイルシステムまたはクラスパスからリソースを読み込むかどうかに応じて、プロトコル file: または classpath: を使用します。クラスパスは、リソースが jar にパッケージ化されている OSGI デプロイメントに必要です。
baseResource		**Camel 2.7:** Web リソースまたはクラスパスのルートディレクトリー。コンポーネントがファイルシステムまたはクラスパスからリソースを読み込むかどうかに応じて、プロトコル file: または classpath: を使用します。クラスパスは、リソースが jar にパッケージ化されている OSGI デプロイメントに必要です。
timeout	**240000**	サーバー側のポーリングのタイムアウト（ミリ秒単位）。これは、応答前にサーバーが再接続要求を保持する期間です。

interval	**0**	クライアント側のポーリングのタイムアウト（ミリ秒単位）。クライアントが再接続間で待機する時間
maxInterval	**30000**	クライアント側の最大ポーリングタイムアウト（ミリ秒単位）。接続がこの時間内に受信されない場合、クライアントは削除されます。
multiFrameInterval	**1500**	同じブラウザーから複数の接続が検出されると、クライアント側のポーリングのタイムアウト。
jsonCommented	**true**	**true** の場合、サーバーはコメントでラップされた JSON を受け入れ、コメントでラップされた JSON を生成します。これは、Hijacking に対する defence です。
logLevel	**1**	**0**=none, **1**=info, **2**=debug.
sslContextParameters		Camel 2.9: レジストリー の **org.apache.camel.util.jsse.SSLContextParameters** オブジェクトへの 参照。 この参照は、コンポーネントレベルで設定済みの SSLContextParameters を上書きします。 Security Guide および Using the JSSE Configuration Utility の Configuring Transport Security for Camel Components の章 を参照してください。
crossOriginFilterOn	**false**	Camel 2.10: **true** の場合、サーバーはドメイン間のフィルターリングをサポートします
allowedOrigins	*	Camel 2.10: **crosssOriginFilterOn** が **true**の場合、クロスをサポートするオリジンドメイン
filterPath		Camel 2.10: **crosssOriginFilterOn** が **true**の場合、filterPath は CrossOriginFilter によって使用されます。

disconnectLocalSession	false	Camel 2.10.5/2.11.1: （プロデューサーのみ）: メッセージをチャネルに公開した後にローカルセッションを切断するかどうか。デフォルトでは CometD によって調整されないため、ローカルセッションの切断が必要です。そのため、メモリー不足になります。

*URI にクエリーオプションは **?option=value&option=value&..** の形式で追加できます。*

パラメーターを渡す方法の例を以下に示します。

*ファイルの場合(Webapp リソースが Web アプリケーションディレクトリーにある場合) **cometd://localhost:8080?resourceBase=file./webapp** クラスパスの場合(Web リソースが Webapp フォルダー内にパッケージ化される場合) **cometd://localhost:8080?resourceBase=classpath:webapp***

認証

Camel 2.8 から利用可能

*カスタムの **SecurityPolicy** および **Extension** の を **CometdComponent** に設定できます。これにより、ここに記載されているように認証を使用できます。*

COMETD コンポーネントでの SSL の設定

*Camel 2.9 の時点で、Cometd コンポーネントは Camel JSSE 設定ユーティリティーを介して SSL/TLS 設定をサポートします。このユーティリティーは、作成する必要のあるコンポーネント固有のコードの量を大幅に減らし、エンドポイントおよびコンポーネントレベルで設定可能です。以下の例は、エンドポイントの Spring DSL ベースの設定を示しています。**CometdComponent** クラスで SSL を設定する必要があります。*

Security Guide の Configuring Transport Security for Camel Components の章を参照してください。

```
...
 <camel:sslContextParameters
   id="sslContextParameters">
  <camel:keyManagers
    keyPassword="keyPassword">
   <camel:keyStore
     resource="/users/home/server/keystore.jks"
     password="keystorePassword"/>
  </camel:keyManagers>
  <camel:trustManagers>
   <camel:keyStore
     resource="/users/home/server/keystore.jks"
     password="keystorePassword"/>
  </camel:keyManagers>
 </camel:sslContextParameters>

 <bean id="cometd" class="org. apache. camel. component.cometd.CometdComponent">
  <property name="sslContextParameters" ref="sslContextParameters"/>
```

```
</bean>
...
<to uri="cometds://127.0.0.1:443/service/test?baseResource=file:./target/test-classes/webapp&timeout=240000&interval=0&maxInterval=30000&multiFrameInterval=1500&jsonCommented=true&logLevel=2&sslContextParameters=#sslContextParameters"/>...
```

第25章 コンテキスト

コンテキストコンポーネント

Camel 2.7 以降で利用可能

context コンポーネントを使用すると、複数のルートを持つ CamelContext から新しい Camel コンポーネントを作成できます。これは、ラックボックスとして扱われるため、他の CamelContext のコンポーネント内のローカルエンドポイントを参照できます。

これは、コンテキストコンポーネントはエンドユーザーにとって非常に簡単ですが、Routes コンポーネントは、CamelContext コンポーネント内のローカルエンドポイントを参照するための設定アプローチよりもシンプルな規則になります。???

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-context</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
context:camelContextId:localEndpointName[?options]
```

または、context: 接頭辞を省略することもできます。

```
camelContextId:localEndpointName[?options]
```

- camelContextId は、CamelContext をレジストリーに登録するのに使用した ID です。
- localEndpointName は、ブラックリストボックス CamelContext 内で評価される有効な Camel URI にすることができます。または、ローカルエンドポイントにマップされる論理名を使用することもできます。たとえば、ローカルで direct:invoices やseda:purchaseOrders などのエンドポイントが id supplyChain の CamelContext 内にある場合は、URIsupplyChain:invoices または supplyChain:purchaseOrders を使用して物理エンドポイントの種類を省略し、純粋な論理 URI を使用できます。

URI にクエリーオプションは **?option=value&option=value&..** の形式で追加できます。

例

この例では、ブラックボックスコンテキストを作成し、別の CamelContext から使用します。

コンテキストコンポーネントの定義

まず CamelContext を作成してそのルートをいくつか追加し、そのルートを起動してから CamelContext をレジストリー(JNDI、Spring、Guice、OSGi など)に登録する必要があります。

これは、この テストケース の通常の Camel 方法で実行できます(createRegistry ()メソッドを参照)。この例では、使用されている Java および JNDI を示しています。

```
// lets create our black box as a camel context and a set of routes
DefaultCamelContext blackBox = new DefaultCamelContext(registry);
blackBox.setName("blackBox");
blackBox.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        // receive purchase orders, lets process it in some way then send an invoice
        // to our invoice endpoint
        from("direct:purchaseOrder").
          setHeader("received").constant("true").
          to("direct:invoice");
    }
});
blackBox.start();

registry.bind("accounts", blackBox);
```

上記のルートでは、純粋なローカルエンドポイント(direct およびseda)を使用していることに注意してください。また、Accounts ID を使用してこの CamelContext を公開することに 注意 してください。Spring では、

```
<camelContext id="accounts" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:purchaseOrder"/>
    ...
    <to uri="direct:invoice"/>
  </route>
</camelContext>
```

コンテキストコンポーネントの使用

その後、別の CamelContext では、アカウントに送信するだけで、この "accounts black box" を参照できます。purchaseOrder および consuming from accounts:invoice です。

より詳細で明示的な場合は、context:accounts:purchaseOrder またはさらに context:accounts:direct://purchaseOrder を使用できます。ただし、実装の詳細を非表示にし、単純な論理命名スキームを提供するため、論理エンドポイント URI の使用が推奨されます。

たとえば、一部のミドルウェア（ブラックボックス外）でこのアカウントブラックボックスを公開する場合は、以下を行うことができます。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <!-- consume from an ActiveMQ into the black box -->
    <from uri="activemq:Accounts.PurchaseOrders"/>
    <to uri="accounts:purchaseOrders"/>
```

```
  </route>
  <route>
    <!-- lets send invoices from the black box to a different ActiveMQ Queue -->
    <from uri="accounts:invoice"/>
    <to uri="activemq:UK.Accounts.Invoices"/>
  </route>
</camelContext>
```

エンドポイントの命名

コンテキストコンポーネントインスタンスには、その CamelContext 外からアクセスできる多くのパブリック入出力エンドポイントを含めることができます。多くの場合は、上記のようにミドルウェアを非表示にするために論理名を使用することが推奨されます。

ただし、コンポーネントに入力、出力、またはエラー/dead 文字のエンドポイントが 1 つしかない場合は、、、および err で共通の posix シェル名を使用すること が推奨されます。

第26章 CONTROLBUS コンポーネント

CONTROLBUS コンポーネント

Camel 2.11 から利用可能

controlbus: コンポーネントは、 Control Bus EIP パターンに基づいて Camel アプリケーションを簡単に管理できます。たとえば、メッセージを エンドポイント に送信することで、ルートのライフサイクルを制御したり、パフォーマンス統計を収集したりできます。

> ***controlbus:command[?options]***

ここでの command には、使用するコマンドのタイプを識別する任意の文字列を指定できます。

コマンド

コマンド	説明
route	**routeId** および **action** パラメーターを使用してルートを制御します。特別なキーワード **current** は現在のルートを示します。
言語	メッセージボディーの評価に 使用する言語 を指定できます。評価の結果がある場合は、結果はメッセージボディーに配置されます。

オプション

名前	デフォルト値	説明
routeId	**null**	ID でルートを指定し **ます**。

action	**null**	**start**、**stop**、または **status** のいずれかのアクションを示します。ルートを開始または停止する場合、またはメッセージボディーの出力としてルートのステータスを取得します。Camel 2.11.1 以降の **suspend** および **resume** を使用して、ルートを一時停止または再開できます。Camel 2.11.1 以降では、統計を使用して XML 形式で返されるパフォーマンスの静的を取得できます。**routeId** オプションを使用して、パフォーマンス統計を取得するルートを定義できます。**routeId** が定義されていない場合は、CamelContext 全体の統計を取得できます。
async	**false**	コントロールバスタスクを非同期で実行するかどうか。**重要：** このオプションが有効な場合には、タスクの結果は Exchange に設定されて **いません**。これは、タスクを同期的に実行する場合にのみ可能です。
loggingLevel	**INFO**	タスクの完了時またはタスクの処理中に例外が発生した場合に使用されるロギングレベル。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

サンプル

ROUTE コマンドの使用

route コマンドを使用すると、ルートを起動するなど、特定のルートで共通のタスクを簡単に実行できます。このエンドポイントに空のメッセージを送信できます。

```
template.sendBody("controlbus:route?routeId=foo&action=start", null);
```

ルートのステータスを取得するには、以下を実行できます。

```
String status = template.requestBody("controlbus:route?routeId=foo&action=status", null,
String.class);
```

パフォーマンス統計の取得

Camel 2.11.1 以降で利用可能

これには、JMX を有効にする必要があります（デフォルトは）。その後、ルートまたは CamelContext ごとにパフォーマンススティックを取得できます。たとえば、foo という名前のルートの静的を取得するには、以下を行います。

```
String xml = template.requestBody("controlbus:route?routeId=foo&action=stats", null,
String.class);
```

返される静的は XML 形式になります。ManagedRouteMBean で dumpRouteStatsAsXml 操作で JMX から取得できるものと同じデータ。

CamelContext 全体の静的を取得するには、以下のように routeId パラメーターを省略します。

```
String xml = template.requestBody("controlbus:route?action=stats", null, String.class);
```

SIMPLE 言語の使用

たとえば、制御バスで Simple 言語を使用すると、特定のルートを停止することができます。以下のメッセージが含まれる "controlbus:language:simple" エンドポイントにメッセージを送信することができます。

```
template.sendBody("controlbus:language:simple", "${camelContext.stopRoute('myRoute')}");
```

これは void 操作であるため、結果が返されません。ただし、ルートステータスが必要な場合は、以下を実行できます。

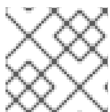
```
String status = template.requestBody("controlbus:language:simple",
"${camelContext.getRouteStatus('myRoute')}", String.class);
```

注記： route コマンドを使用してルートのライフサイクルを制御するのが簡単になります。language コマンドを使用すると、Groovy などの強力な、または Simple 言語を拡張できる言語スクリプトを実行することができます。

たとえば、Camel 自体をシャットダウンするには、次のようにします。

```
template.sendBody("controlbus:language:simple?async=true", "${camelContext.stop()}");
```

async=true を使用して、Camel を非同期的に停止することに注意してください。それ以外の場合は、制御バスコンポーネントに送信されたメッセージの処理中に Camel を停止しようとします。



注記

Groovy などの他の言語を使用することもできます。

- ControlBus EIP
- JMX コンポーネント
- Camel での JMX の使用

第27章 COUCHDB

CAMEL COUCHDB コンポーネント

Camel 2.11 から利用可能

couchdb: コンポーネントを使用すると、CouchDB インスタンスをメッセージのプロデューサーまたはコンシューマーとして扱うことができます。軽量の LightCouch API を使用すると、この Camel コンポーネントには以下の機能があります。

- *コンシューマーとして、は、挿入、更新、および削除をメッセージとして Camel ルートに公開するために couch changeset を監視します。*
- *プロデューサーとして、はドキュメントを保存または更新できます。*
- *複数のインスタンスにまたがる複数のデータベースなど、必要な数だけエンドポイントをサポートできます。*
- *削除時にのみイベントトリガーを持つことができ、挿入/更新または all （デフォルト）のみを挿入します。*
- *sequenceId、ドキュメントリビジョン、ドキュメント ID、および HTTP メソッドタイプに設定されたヘッダー。*

*Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。*

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-couchdb</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
couchdb:http://hostname[:port]/database?[options]
```

hostname は、実行中の couchdb インスタンスのホスト名です。port はオプションであり、指定されていない場合、デフォルトは 5984 に設定されます。

オプション

プロパティー	デフォルト	説明
削除	**true**	ドキュメントの削除はイベントとして公開されます。
updates	**true**	ドキュメントの挿入/更新がイベントとして公開される

heartbeat	30000	ソケットを存続させるための空のメッセージを送信する頻度
createDatabase	true	データベースが存在しない場合は作成します。
username	null	認証されたデータベースの場合のユーザー名
password	null	認証されたデータベースのパスワード

HEADERS

以下のヘッダーは、メッセージトランスポート中にエクスチェンジに設定されます。

プロパティー	値
CouchDbDatabase	メッセージの元となったデータベース
CouchDbSeq	更新/削除メッセージの couchdb 変更セットシーケンス番号
CouchDbId	couchdb ドキュメント ID
CouchDbRev	couchdb ドキュメントリビジョン
CouchDbMethod	メソッド（削除/更新）

メッセージが受信されると、ヘッダーはコンシューマーによって設定されます。また、プロデューサーは挿入/更新が行われるとダウンストリームプロセッサーのヘッダーも設定します。プロデューサーの前に設定されたヘッダーは無視されます。たとえば、CouchDbId をヘッダーとして設定すると、挿入の ID として使用されず、ドキュメントの ID は引き続き使用されます。

メッセージボディー

コンポーネントは、挿入するドキュメントとしてメッセージボディーを使用します。ボディーが String のインスタンスである場合は、挿入前に GSON オブジェクトにマーシャリングされます。つまり、文字列が有効な JSON である必要があります。そうでないと、挿入/更新に失敗します。ボディーが com.google.gson.JsonElement のインスタンスである場合は、そのまま挿入されます。そうしないと、プロデューサーはサポートされていないボディータイプの例外を出力します。

サンプル

たとえば、ローカルで実行している CouchDB インスタンスからすべての挿入、更新、および削除を 9999 ポートで使用する場合は、以下を使用できます。

```
from("couchdb:http://localhost:9999").process(someProcessor);
```

削除のみに関心がある場合は、以下を使用できます。

```
from("couchdb:http://localhost:9999?updates=false").process(someProcessor);
```

メッセージをドキュメントとして挿入する場合は、エクスチェンジのボディーが使用されます。

```
from("someProducingEndpoint").process(someProcessor).to("couchdb:http://localhost:9999"
)
```

第28章 暗号（デジタル署名）

デジタル署名の暗号コンポーネント

Apache Camel 2.3 で利用可能

Apache Camel 暗号化エンドポイントと Java の Cryptographic エクステンションを使用すると、エクスチェンジのデジタル署名を簡単に作成できます。Apache Camel は、エクスチェンジのワークフローの 1 つの部分で交換用の署名を作成し、ワークフローの後半に署名を検証するために使用する柔軟なエンドポイントのペアを提供します。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-crypto</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

Camel 暗号化エンドポイントおよび Java の暗号化エクステンションを使用すると、エクスチェンジのデジタル署名を簡単に作成できます。ただし、Camel は柔軟なエンドポイントのペアも提供します。最初に、エクスチェンジの署名を作成してから、ワークフローの後半部分で署名を検証します。

たとえば、以下のように、JNDI にバインドするためのキーストアを読み込み、Camel が Bean レジストリーから検索できるようにします。

```
// Java
KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
InputStream in = getClass().getResourceAsStream("/my-keystore.ks");
keystore.load(in, "my-keystore-password".toCharArray());
Certificate cert = keystore.getCertificate("my-certificate-alias");

KeyStoreParameters keystoreParameters = new KeyStoreParameters();
keystoreParameters.setPassword("my-keystore-password");
keystoreParameters.setResource("./my-keystore.ks");

InitialContext initialContext = new InitialContext();
initialContext.bind("signatureParams", keystoreParameters);
initialContext.bind("keystore", keystore);
initialContext.bind("myPublicKey", cert.getPublicKey());
initialContext.bind("myCert", cert);
initialContext.bind("myPrivateKey", keystore.getKey("my-certificate-alias", "my-keystore-password".toCharArray()));
```

以下は、エクスチェンジに署名し、検証する camel ルートです。

```
// Java
CamelContext camelContext = new DefaultCamelContext();

camelContext.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:sign")
            .to("crypto:sign://basic?privateKey=#myPrivateKey")
            .to("direct:verify");
        from("direct:verify")
            .to("crypto:verify://basic?publicKey=#myPublicKey")
            .to("mock:result");
    }
});
```

はじめに

デジタル署名は、非対称 Cryptographic 技術を使用してメッセージに署名します。（非常に）高レベルから、アルゴリズムは、特別なプロパティーと複合キーのペアを使用します。これは、1 つの鍵で暗号化されたデータは他のキーでのみ復号化できます。秘密鍵の 1 つは、密接に保護され、他の公開鍵がメッセージの検証に関心のある人に共有されている間、メッセージに署名するために使用されます。メッセージは、秘密鍵を使用してメッセージのダイジェストを暗号化することによって署名されます。この暗号化されたダイジェストは メッセージと共に送信されます。一方、ベリファイアはメッセージダイジェストを再計算し、公開鍵を使用して署名のダイジェストを復号化します。両方のダイジェストが一致する場合、ベリファイアは秘密鍵の所有者のみが署名を作成できることを認識します。

Apache Camel は、Java Cryptographic Extension からの Signature サービスを使用して、交換署名の作成に必要なすべての大きな暗号化レイテンシーを実行します。以下は、Cryptography、Message digests、および Digital Signatures のメカニズム、メッセージダイジェスト、および Digital Signatures の使用方法を説明する優れたソースです。

- Bruce Schneier's Applied Cryptography
- David Hook による Java の暗号開始
- The ever insightful, Wikipedia Digital_signatures

URI 形式

前述のように、Apache Camel は署名を作成および検証するための暗号化エンドポイントのペアを提供します。

```
crypto:sign:name[?options]
crypto:verify:name[?options]
```

- **crypto:sign** は署名を作成し、定数 **org.apache.camel.component.crypto.DigitalSignatureConstants.SIGNATURE**、つまり Header キーに保存します。**"CamelDigitalSignature"**.
- **crypto:verify** はこのヘッダーの内容で読み取り、検証の計算を行います。

正しく機能させるには、署名して鍵のペアを共有し、**PrivateKey** に署名し、**PublicKey** （またはそれを含む 証明書 ）を検証する必要があります。JCE の使用は、これらのキーペアの生成が非常に簡単ですが、通常は KeyStore を使用してキーを格納し、共有することが最も安全です。DSL は、キーがどのように提供されるかについて非常に柔軟であり、さまざまなメカニズムを提供します。

crypto:sign エンドポイントは通常、あるルートと補完的な crypto:verify で定義されます。ただし、もう1つのルートの後に表示される例を簡潔にしています。記号と検証の両方を同時に設定すべきという意味はありません。

オプション

名前	タイプ	デフォルト	説明
algorithm	**文字列**	**DSA**	使用される JCE 署名アルゴリズムの名前。
alias	**文字列**	**null**	キーストアからキーを選択するために使用されるエイリアス名。
bufferSize	**整数**	**2048**	署名プロセスで使用されるバッファーサイズ
certificate	**証明書**	**null**	エクスチェンジのペイロードの署名の検証に使用される証明書。このキーまたは公開鍵のいずれかが必要です。
keystore	**keystore**	**null**	署名および検証に使用されるキーおよび証明書を格納する JCE キーストアへの参照。
provider	**文字列**	**null**	使用する必要がある JCE セキュリティープロバイダーの名前。
privateKey	**PrivatKey**	**null**	エクスチェンジのペイロードの署名に使用される秘密鍵。
publicKey	**PublicKey**	**null**	エクスチェンジのペイロードの署名の検証に使用される公開鍵。
secureRandom	**secureRandom**	**null**	署名サービスの初期化に使用される **SecureRandom** オブジェクトへの参照。
password	**char[]**	**null**	キーストアのパスワード。

clearHeaders	文字列	true	検証操作後にメッセージから camel 暗号化ヘッダーを削除します（値は **"true"**/{{"false"}}）できます。

1）RAW キー

交換に署名して検証する最も基本的な方法は、以下のように KeyPair を使用することです。

```
from("direct:keypair").to("crypto:sign://basic?privateKey=#myPrivateKey",
"crypto:verify://basic?publicKey=#myPublicKey", "mock:result");
```

キーへの参照を使用して Spring XML エクステンション でも同じことができます。

```
<route>
   <from uri="direct:keypair"/>
   <to uri="crypto:sign://basic?privateKey=#myPrivateKey" />
   <to uri="crypto:verify://basic?publicKey=#myPublicKey" />
   <to uri="mock:result"/>
</route>
```

2）キーストアとエイリアス。

JCE は、PrivateKeys と Certificates のペアを格納するために非常に汎用の KeyStore を提供し、それらを暗号化およびパスワードで保護します。これらは、エイリアスを取得 apis に適用して、そこから取得できます。キーストアにキーと証明書を取得する方法は複数あります。ほとんどの場合、これは外部の keytool アプリケーションで行います。これ は、keytool を使用して自己署名証明書と秘密鍵で KeyStore を作成するのに適した例です。

この例では、キーと bob によってエイリアスがついた証明書を持つキーストアを使用します。キーストアおよびキーのパスワードは 'letmein' です。

以下は、Fluent ビルダーを介してキーストアを使用する方法を示しています。また、キーストアを読み込み、初期化する方法も示しています。

```
from("direct:keystore").to("crypto:sign://keystore?
keystore=#keystore&alias=bob&password=letmein", "crypto:verify://keystore?
keystore=#keystore&alias=bob", "mock:result");
```

Spring ではも、ref を使用して実際のキーストアインスタンスを検索します。

```
<route>
   <from uri="direct:keystore"/>
   <to uri="crypto:sign://keystore?keystore=#keystore&lias=bob&assword=letmein" />
   <to uri="crypto:verify://keystore?keystore=#keystore&lias=bob" />
   <to uri="mock:result"/>
</route>
```

3）JCE プロバイダーとアルゴリズムの変更

署名アルゴリズムまたはセキュリティープロバイダーの変更は、名前を指定するための簡単です。選択したアルゴリズムと互換性のあるキーも使用する必要があります。

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(512, new SecureRandom());
keyPair = keyGen.generateKeyPair();
PrivateKey privateKey = keyPair.getPrivate();
PublicKey publicKey = keyPair.getPublic();

// we can set the keys explicitly on the endpoint instances.
context.getEndpoint("crypto:sign://rsa?algorithm=MD5withRSA",
DigitalSignatureEndpoint.class).setPrivateKey(privateKey);
context.getEndpoint("crypto:verify://rsa?algorithm=MD5withRSA",
DigitalSignatureEndpoint.class).setPublicKey(publicKey);
from("direct:algorithm").to("crypto:sign://rsa?algorithm=MD5withRSA", "crypto:verify://rsa?
algorithm=MD5withRSA", "mock:result");
```

```
from("direct:provider").to("crypto:sign://provider?privateKey=#myPrivateKey&provider=SUN",
"crypto:verify://provider?publicKey=#myPublicKey&provider=SUN", "mock:result");
```

または

```
<route>
   <from uri="direct:algorithm"/>
   <to uri="crypto:sign://rsa?algorithm=MD5withRSA&rivateKey=#rsaPrivateKey" />
   <to uri="crypto:verify://rsa?algorithm=MD5withRSA&ublicKey=#rsaPublicKey" />
   <to uri="mock:result"/>
</route>
```

```
<route>
   <from uri="direct:provider"/>
   <to uri="crypto:sign://provider?privateKey=#myPrivateKey&rovider=SUN" />
   <to uri="crypto:verify://provider?publicKey=#myPublicKey&rovider=SUN" />
   <to uri="mock:result"/>
</route>
```

4）署名メッジヘッダーの変更

署名の保存に使用するメッセージヘッダーを変更することが望ましい場合があります。以下のように、ルート定義で異なるヘッダー名を指定できます。

```
from("direct:signature-header").to("crypto:sign://another?
privateKey=#myPrivateKey&signatureHeader=AnotherDigitalSignature",
                              "crypto:verify://another?
publicKey=#myPublicKey&signatureHeader=AnotherDigitalSignature", "mock:result");
```

または

```
<route>
   <from uri="direct:signature-header"/>
   <to uri="crypto:sign://another?
privateKey=#myPrivateKey&ignatureHeader=AnotherDigitalSignature" />
   <to uri="crypto:verify://another?
```

```
publicKey=#myPublicKey&ignatureHeader=AnotherDigitalSignature" />
   <to uri="mock:result"/>
</route>
```

5）バッファーサイズの変更

バッファーサイズを更新する必要がある場合...

```
from("direct:buffersize").to("crypto:sign://buffer?
privateKey=#myPrivateKey&buffersize=1024", "crypto:verify://buffer?
publicKey=#myPublicKey&buffersize=1024", "mock:result");
```

または

```
<route>
   <from uri="direct:buffersize" />
   <to uri="crypto:sign://buffer?privateKey=#myPrivateKey&uffersize=1024" />
   <to uri="crypto:verify://buffer?publicKey=#myPublicKey&uffersize=1024" />
   <to uri="mock:result"/>
</route>
```

6）キーを動的に指定します。

Recipient List または同様の EIP を使用する場合、エクスチェンジの受信者は動的に異なる場合があります。すべての受信者で同じキーを使用することは実行可能でも望ましい場合もあります。交換ごとに署名キーを動的に指定できると便利です。エクスチェンジは、署名前にターゲット受信者のキーで動的にエンリッチできます。これを容易にするために、署名メカニズムにより、以下のメッセージヘッダーを介してキーを動的に提供できます。

- Exchange.SIGNATURE_PRIVATE_KEY, "CamelSignaturePrivateKey"
- Exchange.SIGNATURE_PUBLIC_KEY_OR_CERT, "CamelSignaturePublicKeyOrCert"

```
from("direct:headerkey-sign").to("crypto:sign://alias");
from("direct:headerkey-verify").to("crypto:verify://alias", "mock:result");
```

または

```
<route>
   <from uri="direct:headerkey-sign"/>
   <to uri="crypto:sign://headerkey" />
</route>
<route>
   <from uri="direct:headerkey-verify"/>
   <to uri="crypto:verify://headerkey" />
   <to uri="mock:result"/>
</route>
```

再度、キーストアエイリアスを動的に提供することをお勧めします。ここでも、エイリアスをメッセージヘッダーに提供できます。

- Exchange.KEYSTORE_ALIAS, "CamelSignatureKeyStoreAlias"

```
from("direct:alias-sign").to("crypto:sign://alias?keystore=#keystore");
from("direct:alias-verify").to("crypto:verify://alias?keystore=#keystore", "mock:result");
```

または

```
<route>
   <from uri="direct:alias-sign"/>
   <to uri="crypto:sign://alias?keystore=#keystore" />
</route>
<route>
   <from uri="direct:alias-verify"/>
   <to uri="crypto:verify://alias?keystore=#keystore" />
   <to uri="mock:result"/>
</route>
```

ヘッダーは以下のように設定されます。

```
Exchange unsigned = getMandatoryEndpoint("direct:alias-sign").createExchange();
unsigned.getIn().setBody(payload);
unsigned.getIn().setHeader(DigitalSignatureConstants.KEYSTORE_ALIAS, "bob");
unsigned.getIn().setHeader(DigitalSignatureConstants.KEYSTORE_PASSWORD,
"letmein".toCharArray());
template.send("direct:alias-sign", unsigned);
Exchange signed = getMandatoryEndpoint("direct:alias-sign").createExchange();
signed.getIn().copyFrom(unsigned.getOut());
signed.getIn().setHeader(KEYSTORE_ALIAS, "bob");
template.send("direct:alias-verify", signed);
```

以下も参照してください。

- 暗号化は データ形式としても利用できます。

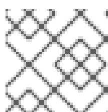
第29章 CXF

CXF コンポーネント

cxf: コンポーネントは、CXF でホストされる JAX-WS サービスに接続するためのApache CXF との統合を提供します。

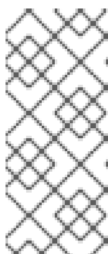
Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-cxf</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```



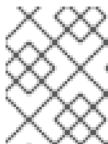
注記

CXF の依存関係については、 **WHICH-JARS** テキストファイルを参照してください。



注記

CXF をコンシューマーとして使用する場合、CAMEL:CXF Bean コンポーネント を使用すると、処理からメッセージペイロードを RESTful または SOAP Web サービスとして受信する方法を計算できます。これは、多量のトランスポートを使用して Web サービスを消費する可能性があります。Bean コンポーネントの設定も簡単で、Camel および CXF を使用して Web サービスを実装する最も高速な方法を提供します。



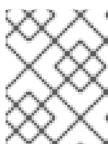
注記

ストリーミングモードで CXF を使用する場合(DataFormat オプションを参照)、Stream caching も読み取ります。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

CXF コンポーネントは、Apache CXF も使用する JBoss EAP **webservices** サブシステムと統合します。詳細は、JAX-WS を参照してください。



注記

現在、Camel on EAP サブシステムは CXF または Restlet コンシューマーをサポートしません。ただし、**CamelProxy** を使用して CXF コンシューマーの動作を模倣できます。

URI 形式

```
cxf:bean:cxfEndpoint[?options]
```

cxfEndpoint は、Spring Bean レジストリーの Bean を参照する Bean ID を表します。この URI 形式では、ほとんどのエンドポイントの詳細を Bean 定義で指定します。

> ***cxf://someAddress[?options]***

someAddress は CXF エンドポイントのアドレスを指定します。この URI 形式では、ほとんどのエンドポイントの詳細は オプションを使用して指定されます。

上記のいずれかのスタイルでは、以下のように URI にオプションを追加できます。

> ***cxf:bean:cxfEndpoint?wsdlURL=wsdl/hello_world.wsdl&dataFormat=PAYLOAD***

オプション

名前	必須	説明
wsdlURL	いいえ	WSDL の場所。WSDL はデフォルトでエンドポイントアドレスから取得されます。以下に例を示します。 **file://local/wsdl/hello.wsdl** または **wsdl/hello.wsdl**

serviceClass	はい	SEI（サービスエンドポイントインターフェイス）クラスの名前。このクラスは JSR181 アノテーションを持つことができますが、必須ではありません。 **2.0 以降、**このオプションは POJO モードでのみ必要です。wsdlURL オプションを指定すると、PAYLOAD モードおよび MESSAGE モードに serviceClass は必要ありません。serviceClass なしで wsdlURL オプションを使用する場合は、serviceName および portName (Spring 設定の endpointName)オプションを指定する **必要** があります。 2.0 以降、**#** 表記を使用してレジストリーから **serviceClass** オブジェクトインスタンスを参照できます。 **Spring AOP プロキシー以外の場合は Object.getClass（）.getName（）メソッドに依存するため、参照されるオブジェクトは Proxy (Spring AOP Proxy は OK)にすること** はできません。 **2.8 以降、** PAYLOAD モードと MESSAGE モードの wsdlURL オプションと serviceClass オプションの両方を省略できます。省略すると、CXF Dispatch モードを容易にするために、任意の XML 要素を PAYLOAD モードで CxfPayload のボディーに配置できます。 例： **org.apache.camel.Hello**
serviceName	WSDL に複数の **serviceName** が存在する場合のみ	このサービスが実装しているサービス名。**wsdl:service@name** にマップされます。以下に例を示します。 **{http://org.apache.camel}ServiceName**
endpointName	**serviceName** の下に複数の **portName** が存在し、Camel 2.2 以降の camel-cxf コンシューマーに必要です。	このサービスが実装しているポート名は、**wsdl:port@name** にマップされます。以下に例を示します。 **{http://org.apache.camel}PortName**

dataFormat	いいえ	CXF エンドポイントがサポートするメッセージデータ形式。使用できる値は **POJO（デフォルト）**、**PAYLOAD**、**MESSAGE** です。
relayHeaders	いいえ	このオプションについては、**relayHeadersオプション の説明**を参照してください。ルートに CXF エンドポイントリレーヘッダーが必要です。現在、**dataFormat=POJOデフォルト**:**true**の**例**:**true**、**false**
wrapped	いいえ	CXF エンドポイントプロデューサーが呼び出す操作の種類。使用できる値は **true**、**false（デフォルト）** です。
wrappedStyle	いいえ	**2.5.0 以降**、パラメーターが SOAP ボディーでどのように表現されるかを記述する WSDL スタイル。値が **false** の場合、CXF はドキュメントリテラルのアンラップスタイルを選択します。値が **true** の場合、CXF はドキュメントリテラルラップスタイルを選択します。
setDefaultBus	いいえ	**非推奨：** このエンドポイントにデフォルトの CXF バスを使用するかどうかを指定します。使用できる値は **true**、**false（デフォルト）** です。このオプションは非推奨となり、Camel 2.16 以降では **defaultBus** を使用する必要があります。
defaultBus	いいえ	**非推奨：** このエンドポイントにデフォルトの CXF バスを使用するかどうかを指定します。使用できる値は **true**、**false（デフォルト）** です。このオプションは非推奨となり、Camel 2.16 以降では **defaultBus** を使用する必要があります。

bus	いいえ	**#** 表記を使用してレジストリーからバスオブジェクトを参照します（例： **bus=#*busName***）。参照オブジェクトは **org.apache.cxf.Bus** のインスタンスである必要があります。 デフォルトでは、は CXF Bus Factory によって作成されたデフォルトのバスを使用します。
cxfBinding	いいえ	**#** 表記を使用して、レジストリーから CXF バインディングオブジェクトを参照します（例：**cxfBinding=#*bindingName*** ）。参照されるオブジェクトは **org.apache.camel.component.cxf.CxfBinding** のインスタンスである必要があります。
headerFilterStrategy	いいえ	**#** 表記を使用して、レジストリーからヘッダーフィルターストラテジーオブジェクトを参照します（例：**headerFilterStrategy=#strategyName** ）。参照されるオブジェクトは **org.apache.camel.spi.HeaderFilterStrategy** のインスタンスである必要があります。
loggingFeatureEnabled	いいえ	新しい 2.3 では、このオプションは、インバウンドおよびアウトバウンド SOAP メッセージをログに書き込む CXF Logging Feature を有効にします。使用できる値は **true**、**false（デフォルト）** です。
defaultOperationName	いいえ	2.4 の新機能として、このオプションはリモートサービスを呼び出す **CxfProducer** によって使用されるデフォルトの **operationName** を設定します。以下に例を示します。 **defaultOperationName=greetMe**

defaultOperationNamespace	いいえ	2.4 の新機能として、このオプションはリモートサービスを呼び出す CxfProducer によって使用されるデフォルトの operationNamespace を設定します。以下に例を示します。 **defaultOperationNamespace = http://apache.org/hello_world_soap_http**
同期	いいえ	2.5 の新機能として、このオプションにより、CXF エンドポイントで sync または async API を使用して基礎となる作業を実行できます。デフォルト値は **false** です。つまり、**camel-cxf** エンドポイントはデフォルトで非同期 API の使用を試行します。
publishedEndpointUrl	いいえ	2.5 の新機能で、このオプションは、サービスアドレス URL と **?wsdl** を使用してアクセスされる公開された WSDL に表示されるエンドポイント URL を上書きします。以下に例を示します。 **publshedEndpointUrl=http://example.com/service**
properties.*propName*	いいえ	Camel 2.8: エンドポイント URI でカスタム CXF プロパティーを設定できます。たとえば、**properties.mtom-enabled=true** を設定して MTOM を有効にします。呼び出しの開始時に CXF がスレッドを切り替えないようにするには、**properties.org.apache.cxf.interceptor.OneWayProcessorInterceptor.USE_ORIGINAL_THREAD=true** を設定します。
allowStreaming	いいえ	2.8.2 の新機能このオプションは、PAYLOAD モード（以下を参照）で実行しているときに CXF コンポーネントを制御するか、またはペイロードを DOM 要素に解析するか、場合によってはストリーミングを可能にする javax.xml.transform.Source オブジェクトとしてペイロードを保持するかどうかを制御します。

skipFaultLogging	いいえ	2.11 の新機能このオプションは、PhaseInterceptorChain がキャッチする Fault のロギングをスキップするかどうかを制御します。
cxfEndpointConfigurer	いいえ	Camel 2.11 の新機能このオプションは、**org.apache.camel.component.cxf.CxfEndpointConfigurer which** supports の実装を適用して、プログラム的に CXF エンドポイントを設定できます。**Camel 2.15.0 以降、**ユーザーは **CxfEndpointConfigurer** の configure{Server|Client} メソッドを実装して CXF サーバーおよびクライアントを設定できます。
username	いいえ	Camel 2.12.3 の新機能。このオプションは、CXF クライアントのユーザー名の基本認証情報を設定するために使用されます。
password	いいえ	Camel 2.12.3 の新機能。このオプションは、CXF クライアントのパスワードの基本認証情報を設定するために使用されます。
continuationTimeout	いいえ	Camel 2.14.0 の新機能は、CXF サーバーが Jetty または Servlet トランスポートを使用している場合にデフォルトで CxfConsumer で使用できる CXF 継続タイムアウトを設定するために使用されます。(**Camel 2.14.** 0 よりも前のバージョンでは、CxfConsumer は継続タイムアウトを 0 に設定するだけで、継続一時停止操作がタイムアウトしないことを意味します)。 デフォルト: 30000 例: continuation=80000

serviceName および portName は QNames であるため、上記の例のようにそれらを {namespace} の接頭辞を指定するようにしてください。

データ形式の説明

DataFormat	説明
POJO	POJO（旧称の Java オブジェクト）は、ターゲットサーバーで呼び出されるメソッドの Java パラメーターです。Protocol および Logical JAX-WS ハンドラーの両方がサポートされます。

PAYLOAD	**PAYLOAD** は、CXF エンドポイントのメッセージ設定の適用後のメッセージペイロード(**soap:body**の内容)です。プロトコル JAX-WS ハンドラーのみがサポートされます。論理 JAX-WS ハンドラーはサポートされていません。
MESSAGE	**MESSAGE** は、トランスポート層から受信される生のメッセージです。Stream をタッチまたは変更することは意図されていません。このような DataFormat を使用している場合は、CXF インターセプターの一部が削除されるため、camel-cxf コンシューマーと JAX-WS ハンドラーがサポートされていない後に soap ヘッダーは表示されません。
CXF_MESSAGE	Camel 2.8.2 の新機能の**CXF_MESSAGE** では、メッセージをトランスポート層から raw SOAP メッセージに変換して CXF インターセプターの完全な機能を呼び出すことができます。

エクスチェンジのプロパティー CamelCXFDataFormat を取得することで、エクスチェンジのデータフォーマットモードを決定できます。Exchange キー定数は org.apache.camel.component.cxf.CxfConstants.DATA_FORMAT_PROPERTY で定義されます。

APACHE ARIES BLUEPRINT での CXF エンドポイントの設定

Camel 2.8 以降、CXF エンドポイントに Aries Blueprint dependency injection を使用するためのサポートがあります。スキーマは Spring スキーマと非常に似ているため、移行は非常に透過的です。

以下に例を示します。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
          xmlns:camel-cxf="http://camel.apache.org/schema/blueprint/cxf"
    xmlns:cxfcore="http://cxf.apache.org/blueprint/core"
          xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

      <camel-cxf:cxfEndpoint id="routerEndpoint"
                   address="http://localhost:9001/router"
                   serviceClass="org.apache.servicemix.examples.cxf.HelloWorld">
        <camel-cxf:properties>
            <entry key="dataFormat" value="MESSAGE"/>
        </camel-cxf:properties>
     </camel-cxf:cxfEndpoint>

     <camel-cxf:cxfEndpoint id="serviceEndpoint"
   address="http://localhost:9000/SoapContext/SoapPort"
                  serviceClass="org.apache.servicemix.examples.cxf.HelloWorld">
    </camel-cxf:cxfEndpoint>

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <route>
```

```
        <from uri="routerEndpoint"/>
        <to uri="log:request"/>
      </route>
   </camelContext>

</blueprint>
```

現在、endpoint 要素は最初にサポートされている CXF namespacehandler です。

Spring の場合と同様に Bean 参照を使用することもできます。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
         xmlns:jaxws="http://cxf.apache.org/blueprint/jaxws"
         xmlns:cxf="http://cxf.apache.org/blueprint/core"
         xmlns:camel="http://camel.apache.org/schema/blueprint"
         xmlns:camelcxf="http://camel.apache.org/schema/blueprint/cxf"
         xsi:schemaLocation="
           http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
           http://cxf.apache.org/blueprint/jaxws
http://cxf.apache.org/schemas/blueprint/jaxws.xsd
           http://cxf.apache.org/blueprint/core http://cxf.apache.org/schemas/blueprint/core.xsd
           ">

   <camelcxf:cxfEndpoint id="reportIncident"
                address="/camel-example-cxf-blueprint/webservices/incident"
                wsdlURL="META-INF/wsdl/report_incident.wsdl"

serviceClass="org.apache.camel.example.reportincident.ReportIncidentEndpoint">
   </camelcxf:cxfEndpoint>

   <bean id="reportIncidentRoutes"
class="org.apache.camel.example.reportincident.ReportIncidentRoutes" />

   <camelContext xmlns="http://camel.apache.org/schema/blueprint">
      <routeBuilder ref="reportIncidentRoutes"/>
   </camelContext>

</blueprint>
```

MESSAGE モードで CXF の LOGGINGOUTINTERCEPTOR を有効にする方法

CXF の **LoggingOutInterceptor** は、ロギングシステム（）に対応するアウトバウンドメッセージを出力します。**java.util.loggingLoggingOutInterceptor** は **PRE_STREAM** フェーズであるため（ただし、**PRE_STREAM** フェーズは **MESSAGE** モードで削除）、**WRITE** フェーズ中に実行する **LoggingOutInterceptor** を設定する必要があります。以下は例です。

```
  <bean id="loggingOutInterceptor"
class="org.apache.cxf.interceptor.LoggingOutInterceptor">
      <!--  it really should have been user-prestream but CXF does have such phase! -->
      <constructor-arg value="target/write"/>
  </bean>

<cxf:cxfEndpoint id="serviceEndpoint" address="http://localhost:9002/helloworld"
 serviceClass="org.apache.camel.component.cxf.HelloService">
 <cxf:outInterceptors>
    <ref bean="loggingOutInterceptor"/>
 </cxf:outInterceptors>
 <cxf:properties>
  <entry key="dataFormat" value="MESSAGE"/>
 </cxf:properties>
</cxf:cxfEndpoint>
```

RELAYHEADERS オプションの説明

JAXWS WSDL ファースト開発者 の視点から、帯域 内 および帯域外 ヘッダーがあります。

インバウンド ヘッダーは、SOAP ヘッダーなどのエンドポイントの WSDL バインディングコントラクトの一部として明示的に定義されるヘッダーです。

帯域外 ヘッダーはネットワーク経由でシリアライズされるヘッダーですが、WSDL バインディングコントラクトの一部ではありません。

ヘッダーのリレー/フィルターリングは双方向です。

ルートに CXF エンドポイントがあり、開発者がSOAP ヘッダーなどのオンワイヤヘッダーを別の JAXWS エンドポイントによって消費されるルートにリレーする必要がある場合、**relayHeaders** はデフォルト値の **true** に設定する必要があります。

POJO モードでのみ利用可能

relayHeaders=true 設定は、ヘッダーをリレーする意図を表します。指定されたヘッダーがリレーされるかどうかに関する実際の決定は、**MessageHeadersRelay** インターフェイスを実装するプラグ可能なインスタンスに委譲されます。**MessageHeadersRelay** の具体的な実装は、ヘッダーをリレーする必要があるかどうかを判断するために参照されます。よく知られている SOAP 名前空間にバインドする **SoapMessageHeadersRelay** の実装があります。現在、帯域外ヘッダーのみがフィルターされ、**relayHeaders=true** の場合に帯域内ヘッダーが常にリレーされます。ネットワーク上でヘッダーがあり、その名前空間がランタイムに認識されていない場合は、フォールバック **DefaultMessageHeadersRelay** が使用され、単純にすべてのヘッダーをリレーできます。

relayHeaders=false 設定は、すべてのヘッダー(in-band および out-of-band)が破棄されることをアサートします。

独自の **MessageHeadersRelay** 実装を上書きするか、リレーのリストに追加の実装を追加できます。事前に読み込んだリレーインスタンスをオーバーライドするには、**MessageHeadersRelay** 実装サービスが、上書きするものと同じ名前空間であることを確認します。また、オーバーライドするリレーは、上書きする名前空間としてすべての名前空間に対応する必要があります。そうしないと、インスタンスマッピングをリレーする名前空間に曖昧さが発生するため、ルート起動時にランタイム例外が出力されます。

```
<cxf:cxfEndpoint ...>
```

```
    <cxf:properties>
      <entry key="org.apache.camel.cxf.message.headers.relays">
        <list>
          <ref bean="customHeadersRelay"/>
        </list>
      </entry>
    </cxf:properties>
  </cxf:cxfEndpoint>
  <bean id="customHeadersRelay"
class="org.apache.camel.component.cxf.soap.headers.CustomHeadersRelay"/>
```

ここで、ヘッダーをリレー/ドロップする方法を示すテストを確認してください。

https://svn.apache.org/repos/asf/camel/branches/camel-1.x/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest.ja

リリース 2.0 以降の変更点

- POJO モードおよび PAYLOAD モードがサポートされます。POJO モードでは、帯域内ヘッダーが処理され、CXF によってヘッダーリストから削除されたため、帯域外メッセージヘッダーのみがフィルターリングできます。帯域内ヘッダーは、POJO モードで MessageContentList に組み込まれます。camel-cxf コンポーネントは、MessageContentList から帯域内ヘッダーの削除を試行しようとします。帯域内ヘッダーのフィルターリングが必要な場合、PAYLOAD モードを使用するか、CXF インターセプター/JAXWS ハンドラーのプラグインを CXF エンドポイントに使用してください。

- Message Header Relay メカニズムは CxfHeaderFilterStrategy に統合されました。relayHeaders オプション、そのセマンティクス、およびデフォルト値は同じですが、CxfHeaderFilterStrategy のプロパティーです。以下は、設定例です。

```
<bean id="dropAllMessageHeadersStrategy"
class="org.apache.camel.component.cxf.common.header.CxfHeaderFilterStrategy">

    <!--  Set relayHeaders to false to drop all SOAP headers -->
    <property name="relayHeaders" value="false"/>

</bean>
```

  次に、エンドポイントは CxfHeaderFilterStrategy を参照できます。

```
<route>
    <from uri="cxf:bean:routerNoRelayEndpoint?
headerFilterStrategy=#dropAllMessageHeadersStrategy"/>
    <to uri="cxf:bean:serviceNoRelayEndpoint?
headerFilterStrategy=#dropAllMessageHeadersStrategy"/>
</route>
```

- MessageHeadersRelay インターフェイスは若干変更され、MessageHeaderFilter に変更されました。これは CxfHeaderFilterStrategy のプロパティーです。以下は、ユーザー定義のメッセージヘッダーフィルターを設定する例です。

```
<bean id="customMessageFilterStrategy"
class="org.apache.camel.component.cxf.common.header.CxfHeaderFilterStrategy">
    <property name="messageHeaderFilters">
```

```
        <list>
            <!--  SoapMessageHeaderFilter is the built in filter.  It can be removed by
omitting it. -->
            <bean
class="org.apache.camel.component.cxf.common.header.SoapMessageHeaderFilter"/
>

            <!--  Add custom filter here -->
            <bean
class="org.apache.camel.component.cxf.soap.headers.CustomHeaderFilter"/>
        </list>
    </property>
</bean>
```

- *relayHeaders 以外に、CxfHeaderFilterStrategy で設定できる新しいプロパティーがあります。*

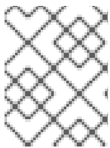
名前	説明	type	必須？	デフォルト値
relayHeaders	すべてのメッセージヘッダーがメッセージヘッダーフィルターによって処理されます。	**boolean**	いいえ	**true** (1.6.1 動作)
relayAllMessageHeaders	すべてのメッセージヘッダーが伝播されます（メッセージヘッダーフィルターによる処理なし）。	**boolean**	いいえ	**false** (1.6.1 の動作)
allowFilterNamespaceClash	2 つのフィルターがアクティベーション namespace で重複する場合、プロパティーはその処理方法を制御します。値が **true** の場合、最後の値が優先されます。値が **false** の場合、例外が発生します。	**boolean**	いいえ	**false** (1.6.1 の動作)

SPRING での CXF エンドポイントの設定

*以下の Spring 設定ファイルで CXF エンドポイントを設定できます。また、エンドポイントを **camelContext** タグに埋め込むこともできます。サービスエンドポイントを呼び出す場合、**operationName** および **operationNamespace** ヘッダーを、呼び出す操作を明示的に設定できます。*

*注記 Camel 2.x では、**http://camel.apache.org/schema/cxf** を CXF エンドポイントのターゲット namespace として使用するよう変更します。*

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://camel.apache.org/schema/cxf"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd     ">
 ...
```



注記

*Apache Camel 2.x では、**http://activemq.apache.org/camel/schema/cxfEndpoint** 名前空間が **http://camel.apache.org/schema/cxf** に変更されました。*

*ルート beans 要素に指定された JAX-WS **schemaLocation** 属性を含めるようにしてください。これにより、CXF はファイルを検証でき、必須です。また、**< cxf:cxfEndpoint/ >** タグの末尾にある namespace 宣言にも注意してください。これらの宣言は、組み合わせた **{namespace}localName** 構文がこのタグの属性値ではサポートされていないためです。*

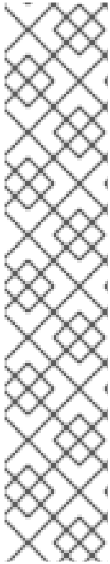
***cxf:cxfEndpoint** 要素は、多くの追加属性をサポートします。*

名前	値
PortName	このサービスが実装しているエンドポイント名は、**wsdl:port@name** にマップされます。**ns:PORT_NAME** の形式で、**ns** はこのスコープで有効な namespace 接頭辞になります。
serviceName	このサービスが実装しているサービス名。**wsdl:service@name** にマップされます。**ns:SERVICE_NAME** の形式で、**ns** はこのスコープで有効な namespace 接頭辞になります。
wsdlURL	WSDL の場所。クラスパス、ファイルシステム上、またはリモートでホストできます。
bindingId	使用するサービスモデルの **bindingId**。
address	サービスの公開アドレス。
bus	JAX-WS エンドポイントで使用されるバス名。
serviceClass	JSR181 アノテーションを持つことができる SEI (Service Endpoint Interface)クラスのクラス名。

また、多くの子要素もサポートします。

名前	値
cxf:inInterceptors	このエンドポイントの受信インターセプター。<bean> **; または < ref> の** リスト。
cxf:inFaultInterceptors	このエンドポイントの受信障害インターセプター。<bean> **; または < ref> の** リスト。
cxf:outInterceptors	このエンドポイントの送信インターセプター。<bean> **; または < ref> の** リスト。
cxf:outFaultInterceptors	このエンドポイントの送信障害インターセプター。<bean> **; または < ref> の** リスト。
cxf:properties	JAX-WS エンドポイントに提供する必要があるプロパティーマップ。以下を参照してください。
cxf:handlers	JAX-WS エンドポイントに提供する必要がある JAX-WS ハンドラーリスト。以下を参照してください。
cxf:dataBinding	エンドポイントで使用する **DataBinding** を指定できます。これは、Spring < **bean class="MyDataBinding"**/ > 構文を使用して指定できます。
cxf:binding	このエンドポイントで使用する **BindingFactory** を指定できます。これは、Spring < **bean class="MyBindingFactory"**/ > 構文を使用して指定できます。
cxf:features	このエンドポイントのインターセプターを保持する機能。< **bean> s または**< **ref> s**の一覧
cxf:schemaLocations	使用するエンドポイントのスキーマの場所。< **schemaLocation> の**一覧
cxf:serviceFactory	このエンドポイントで使用するサービスファクトリー。これは、Spring < **bean class="MyServiceFactory"/> 構文を使用して指定でき** ます。

インターセプター、プロパティー、およびハンドラーの提供方法を示す高度な例は、以下を参照してください。 *http://cwiki.apache.org/CXF20DOC/jax-ws-configuration.html*



注記

CXF:properties を使用して、以下のように Spring 設定ファイルの CXF エンドポイントの **dataFormat** および **setDefaultBus** プロパティーを設定できます。

```
<cxf:cxfEndpoint id="testEndpoint" address="http://localhost:9000/router"
    serviceClass="org.apache.camel.component.cxf.HelloService"
    endpointName="s:PortName"
    serviceName="s:ServiceName"
    xmlns:s="http://www.example.com/test">
    <cxf:properties>
      <entry key="dataFormat" value="MESSAGE"/>
      <entry key="setDefaultBus" value="true"/>
    </cxf:properties>
  </cxf:cxfEndpoint>
```

HOW TO MAKE THE CAMEL-CXF COMPONENT USE LOG4J INSTEAD OF JAVA.UTIL.LOGGING

CXF のデフォルトロガーは **java.util.logging** です。**log4j** に変更するには、以下の手順を実行します。**META-INF/cxf/org.apache.cxf.logger** という名前のクラスパスにファイルを作成します。このファイルには、1 行にコメントのない **org.apache.cxf.common.logging.Log4jLogger** クラスの完全修飾名が含まれている必要があります。

HOW TO LET CAMEL-CXF RESPONSE MESSAGE WITH XML START DOCUMENT

PHP などの SOAP クライアントを使用している場合は、CXF は XML 開始ドキュメント **<?xml version="1.0" encoding="utf-8"?>** を追加しないため、この種類のエラーが発生します。

```
Error:sendSms: SoapFault exception: [Client] looks like we got no XML document in [...]
```

この問題を解決するには、XML 開始ドキュメントを作成するように StaxOutInterceptor に指示するだけです。

```
public class WriteXmlDeclarationInterceptor extends
AbstractPhaseInterceptor<SoapMessage> {
    public WriteXmlDeclarationInterceptor() {
        super(Phase.PRE_STREAM);
        addBefore(StaxOutInterceptor.class.getName());
    }

    public void handleMessage(SoapMessage message) throws Fault {
        message.put("org.apache.cxf.stax.force-start-document", Boolean.TRUE);
    }

}
```

このようなカスタマーインターセプターを追加し、**camel-cxf** エンドユーザーに設定できます。

```
<cxf:cxfEndpoint id="routerEndpoint"
address="http://localhost:${CXFTestSupport.port2}/CXFGreeterRouterTest/CamelContext/Rou
terPort"
```

```
    serviceClass="org.apache.hello_world_soap_http.GreeterImpl"
    skipFaultLogging="true">
      <cxf:outInterceptors>
          <!-- This interceptor will force the CXF server send the XML start document to client -->
          <bean class="org.apache.camel.component.cxf.WriteXmlDeclarationInterceptor"/>
      </cxf:outInterceptors>
      <cxf:properties>
          <!-- Set the publishedEndpointUrl which could override the service address from
generated WSDL as you want -->
          <entry key="publishedEndpointUrl" value="http://www.simple.com/services/test" />
      </cxf:properties>
  </cxf:cxfEndpoint>
```

または、Camel 2.4 を使用している場合は、このようなメッセージヘッダーを追加し ます。

```
// set up the response context which force start document
Map<String, Object> map = new HashMap<String, Object>();
map.put("org.apache.cxf.stax.force-start-document", Boolean.TRUE);
exchange.getOut().setHeader(Client.RESPONSE_CONTEXT, map);
```

POJO データ形式で CAMEL-CXF エンドポイントからメッセージを消費する方法

camel-cxf エンドポイントコンシューマー**POJO** データ形式は cxf invoker をベースとしているため、メッセージヘッダーには **CxfConstants.OPERATION_NAME** という名前の プロパティーがあり、メッセージボディーは SEI メソッドパラメーターのリストです。

```
public class PersonProcessor implements Processor {

    private static final transient Logger LOG =
LoggerFactory.getLogger(PersonProcessor.class);

    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        LOG.info("processing exchange in camel");

        BindingOperationInfo boi =
(BindingOperationInfo)exchange.getProperty(BindingOperationInfo.class.toString());
        if (boi != null) {
            LOG.info("boi.isUnwrapped" + boi.isUnwrapped());
        }
        // Get the parameters list which element is the holder.
        MessageContentsList msgList = (MessageContentsList)exchange.getIn().getBody();
        Holder<String> personId = (Holder<String>)msgList.get(0);
        Holder<String> ssn = (Holder<String>)msgList.get(1);
        Holder<String> name = (Holder<String>)msgList.get(2);

        if (personId.value == null || personId.value.length() == 0) {
            LOG.info("person id 123, so throwing exception");
            // Try to throw out the soap fault message
            org.apache.camel.wsdl_first.types.UnknownPersonFault personFault =
                new org.apache.camel.wsdl_first.types.UnknownPersonFault();
            personFault.setPersonId("");
            org.apache.camel.wsdl_first.UnknownPersonFault fault =
```

```
            new org.apache.camel.wsdl_first.UnknownPersonFault("Get the null value of person
name", personFault);
        // Since camel has its own exception handler framework, we can't throw the exception
to trigger it
        // We just set the fault message in the exchange for camel-cxf component handling and
return
        exchange.getOut().setFault(true);
        exchange.getOut().setBody(fault);
        return;
      }

      name.value = "Bonjour";
      ssn.value = "123";
      LOG.info("setting Bonjour as the response");
      // Set the response message, first element is the return value of the operation,
      // the others are the holders of method parameters
      exchange.getOut().setBody(new Object[] {null, personId, ssn, name});
   }

}
```

POJO データ形式で CAMEL-CXF エンドポイントのメッセージを準備する方法

camel-cxf エンドポイントプロデューサーは cxf クライアント API に基づいています。最初にメッセージヘッダーで操作名を指定し、メソッドパラメーターをリストに追加し、このパラメーターリストでメッセージを初期化する必要があります。応答メッセージのボディーは messageContentsList で、その結果をリストから取得できます。

メッセージヘッダーで操作名を指定しない場合、CxfProducer は defaultOperationName from CxfEndpoint の使用を試行します。CxfEndpoint に defaultOperationName が設定されていない場合、操作リストから最初の操作名が選択されます。

メッセージボディーからオブジェクトアレイを取得する場合は、以下のように message.getbody (Object[].class) を使用してボディーを取得できます。

```
Exchange senderExchange = new DefaultExchange(context, ExchangePattern.InOut);
final List<String> params = new ArrayList<String>();
// Prepare the request message for the camel-cxf procedure
params.add(TEST_MESSAGE);
senderExchange.getIn().setBody(params);
senderExchange.getIn().setHeader(CxfConstants.OPERATION_NAME, ECHO_OPERATION);

Exchange exchange = template.send("direct:EndpointA", senderExchange);

org.apache.camel.Message out = exchange.getOut();
// The response message's body is an MessageContentsList which first element is the return
value of the operation,
// If there are some holder parameters, the holder parameter will be filled in the reset of List.
// The result will be extract from the MessageContentsList with the String class type
MessageContentsList result = (MessageContentsList)out.getBody();
LOG.info("Received output text: " + result.get(0));
Map<String, Object> responseContext = CastUtils.cast((Map<?, ?
>)out.getHeader(Client.RESPONSE_CONTEXT));
assertNotNull(responseContext);
```

```
assertEquals("We should get the response context here", "UTF-8",
responseContext.get(org.apache.cxf.message.Message.ENCODING));
assertEquals("Reply body on Camel is wrong", "echo " + TEST_MESSAGE, result.get(0));
```

PAYLOAD データ形式で CAMEL-CXF エンドポイントのメッセージを処理する方法

Apache Camel 2.0: **CxfMessage.getBody （）** は、SOAP メッセージヘッダーおよび Body 要素のゲッターを持つ **org.apache.camel.component.cxf.CxfPayload** オブジェクトを返します。この変更により、Apache Camel メッセージからネイティブ CXF メッセージを切り離すことができます。

```
protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
            from(SIMPLE_ENDPOINT_URI + "&dataFormat=PAYLOAD").to("log:info").process(new
Processor() {
                @SuppressWarnings("unchecked")
                public void process(final Exchange exchange) throws Exception {
                    CxfPayload<SoapHeader> requestPayload =
exchange.getIn().getBody(CxfPayload.class);
                    List<Source> inElements = requestPayload.getBodySources();
                    List<Source> outElements = new ArrayList<Source>();
                    // You can use a customer toStringConverter to turn a CxfPayLoad message into
String as you want
                    String request = exchange.getIn().getBody(String.class);
                    XmlConverter converter = new XmlConverter();
                    String documentString = ECHO_RESPONSE;

                    Element in = new XmlConverter().toDOMElement(inElements.get(0));
                    // Just check the element namespace
                    if (!in.getNamespaceURI().equals(ELEMENT_NAMESPACE)) {
                        throw new IllegalArgumentException("Wrong element namespace");
                    }
                    if (in.getLocalName().equals("echoBoolean")) {
                        documentString = ECHO_BOOLEAN_RESPONSE;
                        checkRequest("ECHO_BOOLEAN_REQUEST", request);
                    } else {
                        documentString = ECHO_RESPONSE;
                        checkRequest("ECHO_REQUEST", request);
                    }
                    Document outDocument = converter.toDOMDocument(documentString);
                    outElements.add(new DOMSource(outDocument.getDocumentElement()));
                    // set the payload header with null
                    CxfPayload<SoapHeader> responsePayload = new CxfPayload<SoapHeader>(null,
outElements, null);
                    exchange.getOut().setBody(responsePayload);
                }
            });
        }
    };
}
```

POJO モードで SOAP ヘッダーを取得および設定する方法

POJO は、CXF エンドポイントが Camel エクスチェンジを生成または消費した場合に、データフォーマットが Java オブジェクトのリスト であることを意味します。Apache Camel はメッセージボディーをこのモードで POJO として公開しますが、CXF コンポーネントは引き続き SOAP ヘッダーの読み取りおよび書き込みへのアクセスを提供します。ただし、CXF インターセプターは処理後にヘッダーリストから帯域外 SOAP ヘッダーを削除するため、POJO モードで使用できるのは帯域外 SOAP ヘッダーのみです。

以下の例は、SOAP ヘッダーの取得/設定方法を示しています。ある CXF エンドポイントから別の CXF エンドポイントに転送するルートがあるとします。つまり、SOAP Client -> Apache Camel -> CXF service です。要求が CXF サービスに送信される前に(1)で SOAP ヘッダーの取得/挿入に 2 つのプロセッサーを添付し、応答がSOAP クライアントに戻る前に(2)。この例では プロセッサー(1)および(2)は InsertRequestOutHeaderProcessor および InsertResponseOutHeaderProcessor です。ルートは以下のようになります。

```
<route>
    <from uri="cxf:bean:routerRelayEndpointWithInsertion"/>
    <process ref="InsertRequestOutHeaderProcessor" />
    <to uri="cxf:bean:serviceRelayEndpointWithInsertion"/>
    <process ref="InsertResponseOutHeaderProcessor" />
</route>
```

*2.x で は、SOAP ヘッダーは Apache Camel Message ヘッダーとの間で伝播されます。Apache Camel メッセージヘッダー名は **org.apache.cxf.headers.Header.list** で、CXF (**org.apache.cxf.headers.Header.HEADER_LIST**)で定義された定数です。ヘッダーの値は、CXF **SoapHeader** オブジェクト(**org.apache.cxf.binding.soap.SoapHeader**)の **List<>** です。以下のスニペットは **InsertResponseOutHeaderProcessor** です（応答メッセージに新しい SOAP ヘッダーを挿入します）。**InsertResponseOutHeaderProcessor** と **InsertRequestOutHeaderProcessor** の両方で SOAP ヘッダーにアクセスする方法は、実際には同じです。2 つのプロセッサーの唯一の違いは、挿入された SOAP ヘッダーの方向を設定することです。*

```
public static class InsertResponseOutHeaderProcessor implements Processor {

    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        // You should be able to get the header if exchange is routed from camel-cxf endpoint
        List<SoapHeader> soapHeaders = CastUtils.cast((List<?
>)exchange.getIn().getHeader(Header.HEADER_LIST));
        if (soapHeaders == null) {
            // we just create a new soap headers in case the header is null
            soapHeaders = new ArrayList<SoapHeader>();
        }

        // Insert a new header
        String xml = "<?xml version=\"1.0\" encoding=\"utf-8\"?><outofbandHeader "
            + "xmlns=\"http://cxf.apache.org/outofband/Header\" hdrAttribute=\"testHdrAttribute\" "
            + "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\"
soap:mustUnderstand=\"1\">"
            + "<name>New_testOobHeader</name><value>New_testOobHeaderValue</value>
</outofbandHeader>";
        SoapHeader newHeader = new SoapHeader(soapHeaders.get(0).getName(),
                    DOMUtils.readXml(new StringReader(xml)).getDocumentElement());
        // make sure direction is OUT since it is a response message.
        newHeader.setDirection(Direction.DIRECTION_OUT);
        //newHeader.setMustUnderstand(false);
        soapHeaders.add(newHeader);
```

```
    }

}
```

PAYLOAD モードでSOAP ヘッダーを取得および設定する方法

PAYLOAD モードでSOAP メッセージ(**CxfPayload** オブジェクト)にアクセスする方法はすでに説明されています(「PAYLOAD データ形式で camel-cxf エンドポイントのメッセージを処理する方法」を参照してください)。

CxfPayload オブジェクトを取得したら、DOM Elements (SOAP ヘッダー)の**List** を返す **CxfPayload.getHeaders()** メソッドを呼び出すことができます。

```
from(getRouterEndpointURI()).process(new Processor() {
    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        CxfPayload<SoapHeader> payload = exchange.getIn().getBody(CxfPayload.class);
        List<Source> elements = payload.getBodySources();
        assertNotNull("We should get the elements here", elements);
        assertEquals("Get the wrong elements size", 1, elements.size());

        Element el = new XmlConverter().toDOMElement(elements.get(0));
        elements.set(0, new DOMSource(el));
        assertEquals("Get the wrong namespace URI", "http://camel.apache.org/pizza/types",
                el.getNamespaceURI());

        List<SoapHeader> headers = payload.getHeaders();
        assertNotNull("We should get the headers here", headers);
        assertEquals("Get the wrong headers size", headers.size(), 1);
        assertEquals("Get the wrong namespace URI",
                ((Element)(headers.get(0).getObject())).getNamespaceURI(),
                "http://camel.apache.org/pizza/types");
    }

})
.to(getServiceEndpointURI());
```

Camel 2.16.0 以降、「POJO モードでSOAP ヘッダーを取得および設定する方法」で説明されているのと同じアプローチを使用してSOAP ヘッダーを設定または取得できます。**org.apache.cxf.headers.Header.list** ヘッダーを使用して、SOAP ヘッダーの一覧を取得および設定できるようになりました。つまり、ある Camel CXF エンドポイントから別の(SOAP Client -> Camel -> CXF サービス)に転送するルートがある場合、SOAP クライアントによって送信された SOAP ヘッダーも CXF サービスに転送されるようになりました。ヘッダーの転送を希望しない場合は、**org.apache.cxf.headers.Header.list** Camel ヘッダーから削除します。

SOAP ヘッダーがMESSAGE モードで利用できない

SOAP 処理はスキップされるため、SOAP ヘッダーは **MESSAGE** モードでは利用できません。

APACHE CAMEL からSOAP FAULT を出力する方法

CXF エンドポイントを使用してSOAP リクエストを消費する場合、Camel コンテキストからSOAP Fault を出力する必要がある場合があります。基本的に、**throwFault** DSL を使用してこれを実行できま

す。POJO、PAYLOAD、および MESSAGE データフォーマットで機能します。以下のように soap 障害を定義できます。

```
SOAP_FAULT = new SoapFault(EXCEPTION_MESSAGE, SoapFault.FAULT_CODE_CLIENT);
Element detail = SOAP_FAULT.getOrCreateDetail();
Document doc = detail.getOwnerDocument();
Text tn = doc.createTextNode(DETAIL_TEXT);
detail.appendChild(tn);
```

次に、以下のようにこれを出力します。

```
from(routerEndpointURI).setFaultBody(constant(SOAP_FAULT));
```

CXF エンドポイントが MESSAGE データ形式で機能している場合は、メッセージボディーに SOAP Fault メッセージを設定し、メッセージヘッダーに応答コードを設定できます。

```
from(routerEndpointURI).process(new Processor() {

    public void process(Exchange exchange) throws Exception {
        Message out = exchange.getOut();
        // Set the message body with the
        out.setBody(this.getClass().getResourceAsStream("SoapFaultMessage.xml"));
        // Set the response code here
        out.setHeader(org.apache.cxf.message.Message.RESPONSE_CODE, new Integer(500));
    }

});
```

POJO データフォーマットでも同様です。Out ボディーに SOAP Fault を設定し、以下のように Message.setFault(true) を呼び出すことで障害を示すこともできます。

```
from("direct:start").onException(SoapFault.class).maximumRedeliveries(0).handled(true)
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            SoapFault fault = exchange
                .getProperty(Exchange.EXCEPTION_CAUGHT, SoapFault.class);
            exchange.getOut().setFault(true);
            exchange.getOut().setBody(fault);
        }

    }).end().to(serviceURI);
```

CXF エンドポイントの要求および応答コンテキストを伝播する方法

CXF クライアント API は、リクエストおよび応答コンテキストで操作を呼び出す方法を提供します。CXF エンドポイントプロデューサーを使用して外部 Web サービスを呼び出す場合は、リクエストコンテキストを設定し、以下のコードで応答コンテキストを取得できます。

```
        CxfExchange exchange = (CxfExchange)template.send(getJaxwsEndpointUri(), new
Processor() {
             public void process(final Exchange exchange) {
                 final List<String> params = new ArrayList<String>();
                 params.add(TEST_MESSAGE);
```

```
            // Set the request context to the inMessage
            Map<String, Object> requestContext = new HashMap<String, Object>();
            requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
JAXWS_SERVER_ADDRESS);
            exchange.getIn().setBody(params);
            exchange.getIn().setHeader(Client.REQUEST_CONTEXT , requestContext);
            exchange.getIn().setHeader(CxfConstants.OPERATION_NAME,
GREET_ME_OPERATION);
          }
        });
        org.apache.camel.Message out = exchange.getOut();
        // The output is an object array, the first element of the array is the return value
        Object\[\] output = out.getBody(Object\[\].class);
        LOG.info("Received output text: " + output\[0\]);
        // Get the response context form outMessage
        Map<String, Object> responseContext =
CastUtils.cast((Map)out.getHeader(Client.RESPONSE_CONTEXT));
        assertNotNull(responseContext);
        assertEquals("Get the wrong wsdl opertion name", "
{http://apache.org/hello_world_soap_http}greetMe",
                    responseContext.get("javax.xml.ws.wsdl.operation").toString());
```

添付ファイルのサポート

POJO モード： Attachment および MTOM を備えた SOAP の両方がサポートされます(MTOM を有効にする場合は Payload モードの例を参照)。 ただし、添付付きの SOAP はテストされていません。 アタッチメントは POJO にマーシャリングおよびアンマーシャリングされるため、通常は添付ファイルを処理する必要はありません。 添付ファイルは、2.1 以降 Camel メッセージの添付に伝播されます。 そのため、Camel Message API による再添付が可能です。

```
DataHandler Message.getAttachment(String id)
```

をクリックします。

ペイロードモード： MTOM は 2.1 以降でサポートされています。添付ファイルは、上記の Camel Message API で取得できます。このモードでは SOAP 処理がないため、Attachment を使用した SOAP はサポートされません。

MTOM を有効にするには、CXF エンドポイントプロパティー mtom_enabled を true に設定します。(Spring でのみ実行できます)

```
<cxf:cxfEndpoint id="routerEndpoint"
address="http://localhost:${CXFTestSupport.port1}/CxfMtomRouterPayloadModeTest/jaxws-mtom/hello"
        wsdlURL="mtom.wsdl"
        serviceName="ns:HelloService"
        endpointName="ns:HelloPort"
        xmlns:ns="http://apache.org/camel/cxf/mtom_feature">

    <cxf:properties>
        <!--  enable mtom by setting this property to true -->
        <entry key="mtom-enabled" value="true"/>
```

```
        <!--  set the camel-cxf endpoint data fromat to PAYLOAD mode -->
        <entry key="dataFormat" value="PAYLOAD"/>
    </cxf:properties>
```

attachment で Camel メッセージを生成し、Payload モードで CXF エンドポイントに送信できます。

```
Exchange exchange = context.createProducerTemplate().send("direct:testEndpoint", new
Processor() {

    public void process(Exchange exchange) throws Exception {
        exchange.setPattern(ExchangePattern.InOut);
        List&lt;Source> elements = new ArrayList&lt;Source>();
        elements.add(new DOMSource(DOMUtils.readXml(new
StringReader(MtomTestHelper.REQ_MESSAGE)).getDocumentElement()));
        CxfPayload<SoapHeader> body = new CxfPayload<SoapHeader>(new
ArrayList<SoapHeader>(),
            elements, null);
        exchange.getIn().setBody(body);
        exchange.getIn().addAttachment(MtomTestHelper.REQ_PHOTO_CID,
            new DataHandler(new ByteArrayDataSource(MtomTestHelper.REQ_PHOTO_DATA,
"application/octet-stream")));

        exchange.getIn().addAttachment(MtomTestHelper.REQ_IMAGE_CID,
            new DataHandler(new ByteArrayDataSource(MtomTestHelper.requestJpeg,
"image/jpeg")));

    }

});

// process response

CxfPayload<SoapHeader> out = exchange.getOut().getBody(CxfPayload.class);
Assert.assertEquals(1, out.getBody().size());

Map<String, String> ns = new HashMap<String, String>();
ns.put("ns", MtomTestHelper.SERVICE_TYPES_NS);
ns.put("xop", MtomTestHelper.XOP_NS);

XPathUtils xu = new XPathUtils(ns);
Element oute = new XmlConverter().toDOMElement(out.getBody().get(0));
Element ele = (Element)xu.getValue("//ns:DetailResponse/ns:photo/xop:Include", oute,
                                   XPathConstants.NODE);
String photoId = ele.getAttribute("href").substring(4); // skip "cid:"

ele = (Element)xu.getValue("//ns:DetailResponse/ns:image/xop:Include", oute,
                                   XPathConstants.NODE);
String imageId = ele.getAttribute("href").substring(4); // skip "cid:"

DataHandler dr = exchange.getOut().getAttachment(photoId);
Assert.assertEquals("application/octet-stream", dr.getContentType());
MtomTestHelper.assertEquals(MtomTestHelper.RESP_PHOTO_DATA,
IOUtils.readBytesFromStream(dr.getInputStream()));

dr = exchange.getOut().getAttachment(imageId);
Assert.assertEquals("image/jpeg", dr.getContentType());
```

```
BufferedImage image = ImageIO.read(dr.getInputStream());
Assert.assertEquals(560, image.getWidth());
Assert.assertEquals(300, image.getHeight());
```

Payload モードで CXF エンドポイントから受信した Camel メッセージを使用することもできます。

```
public static class MyProcessor implements Processor {

   @SuppressWarnings("unchecked")
   public void process(Exchange exchange) throws Exception {
      CxfPayload<SoapHeader> in = exchange.getIn().getBody(CxfPayload.class);

      // verify request
      assertEquals(1, in.getBody().size());

      Map<String, String> ns = new HashMap<String, String>();
      ns.put("ns", MtomTestHelper.SERVICE_TYPES_NS);
      ns.put("xop", MtomTestHelper.XOP_NS);

      XPathUtils xu = new XPathUtils(ns);
      Element body = new XmlConverter().toDOMElement(in.getBody().get(0));
      Element ele = (Element)xu.getValue("//ns:Detail/ns:photo/xop:Include", body,
                              XPathConstants.NODE);
      String photoId = ele.getAttribute("href").substring(4); // skip "cid:"
      assertEquals(MtomTestHelper.REQ_PHOTO_CID, photoId);

      ele = (Element)xu.getValue("//ns:Detail/ns:image/xop:Include", body,
                              XPathConstants.NODE);
      String imageId = ele.getAttribute("href").substring(4); // skip "cid:"
      assertEquals(MtomTestHelper.REQ_IMAGE_CID, imageId);

      DataHandler dr = exchange.getIn().getAttachment(photoId);
      assertEquals("application/octet-stream", dr.getContentType());
      MtomTestHelper.assertEquals(MtomTestHelper.REQ_PHOTO_DATA,
IOUtils.readBytesFromStream(dr.getInputStream()));

      dr = exchange.getIn().getAttachment(imageId);
      assertEquals("image/jpeg", dr.getContentType());
      MtomTestHelper.assertEquals(MtomTestHelper.requestJpeg,
IOUtils.readBytesFromStream(dr.getInputStream()));

      // create response
      List&lt;Source> elements = new ArrayList&lt;Source>();
      elements.add(new DOMSource(DOMUtils.readXml(new
StringReader(MtomTestHelper.RESP_MESSAGE)).getDocumentElement()));
      CxfPayload&lt;SoapHeader> sbody = new CxfPayload&lt;SoapHeader>(new
ArrayList&lt;SoapHeader>(),
         elements, null);
      exchange.getOut().setBody(sbody);
      exchange.getOut().addAttachment(MtomTestHelper.RESP_PHOTO_CID,
         new DataHandler(new ByteArrayDataSource(MtomTestHelper.RESP_PHOTO_DATA,
"application/octet-stream")));

      exchange.getOut().addAttachment(MtomTestHelper.RESP_IMAGE_CID,
         new DataHandler(new ByteArrayDataSource(MtomTestHelper.responseJpeg,
```

```
"image/jpeg")));

    }
}
```

メッセージモード： 添付ファイルはメッセージをまったく処理しないため、サポートされていません。

CXF_MESSAGE モード： MTOM はサポートされ、添付ファイルは上記の Camel Message API で取得できます。マルチパート（つまり MTOM）メッセージを受信すると、デフォルトの **SOAPMessage** から **String** コンバーターは、ボディーの完全なマルチパートペイロードを提供することに注意してください。SOAP XML を **String** として要求する場合は、**message.getSOAPPart()** でメッセージボディーを設定し、Camel 変換で残りの作業を行うことができます。

スタックトレース情報を伝播させる方法

Java の例外がサーバー側で発生したときに、例外のスタックトレースがフォールトメッセージにマーシャリングされてクライアントに返されるように、CXF エンドポイントを設定することができます。この機能を有効にするには、**dataFormat** を **PAYLOAD** に設定し、以下のように **faultStackTraceEnabled** プロパティーを **true** に設定します。**cxfEndpoint**

```
<cxf:cxfEndpoint id="router" address="http://localhost:9002/TestMessage"
    wsdlURL="ship.wsdl"
    endpointName="s:TestSoapEndpoint"
    serviceName="s:TestService"
    xmlns:s="http://test">
  <cxf:properties>
    <!-- enable sending the stack trace back to client; the default value is false-->
    <entry key="faultStackTraceEnabled" value="true" />
    <entry key="dataFormat" value="PAYLOAD" />
  </cxf:properties>
</cxf:cxfEndpoint>
```

セキュリティー上の理由から、スタックトレースには原因となる例外（つまり、**Caused by**に続くスタックトレースの一部）は含まれません。スタックトレースに原因となる例外を含める場合は、以下のように **cxfEndpoint** 要素で **exceptionMessageCauseEnabled** プロパティーを **true** に設定します。

```
<cxf:cxfEndpoint id="router" address="http://localhost:9002/TestMessage"
    wsdlURL="ship.wsdl"
    endpointName="s:TestSoapEndpoint"
    serviceName="s:TestService"
    xmlns:s="http://test">
  <cxf:properties>
    <!-- enable to show the cause exception message and the default value is false -->
   <entry key="exceptionMessageCauseEnabled" value="true" />
    <!-- enable to send the stack trace back to client,  the default value is false-->
    <entry key="faultStackTraceEnabled" value="true" />
    <entry key="dataFormat" value="PAYLOAD" />
  </cxf:properties>
</cxf:cxfEndpoint>
```

警告

exceptionMessageCauseEnabled フラグは、テストおよび診断の目的でのみ有効にする必要があります。サーバーにおいて例外の元の原因を隠すことで、敵対的なユーザーがサーバーを調査しにくくするのが、通常の実践的なやり方です。

PAYLOAD モードのストリーミングサポート

2.8.2 では、PAYLOAD モードを使用する場合に camel-cxf コンポーネントで受信メッセージのストリーミングがサポートされるようになりました。以前は、受信メッセージは完全に DOM 解析されていました。メッセージが大きい場合、これには時間がかかり、大量のメモリーを使用します。2.8.2 以降、受信メッセージはルーティング時に javax.xml.transform.Source として残ることができます。また、ペイロードを何も変更しないと、ターゲットの宛先に直接ストリーミングできます。一般的な単純なプロキシーのユースケース（例：from ("cxf:...").to ("cxf:...")）では、パフォーマンスが大幅に向上し、メモリー要件が大幅に低下する可能性があります。

ただし、ストリーミングが適切でない、または望ましくない場合もあります。ストリーミングの性質上、無効な受信 XML は処理チェーンで後でキャッチされないことがあります。また、特定のアクションでは、メッセージを DOM で解析する必要があります(WS-Security やメッセージトレースなど)。この場合、ストリーミングの利点は制限されます。この時点で、ストリーミングを制御する方法は 2 つあります。

- endpoint プロパティー："allowStreaming=false" をエンドポイントプロパティーとして追加し、ストリーミングをオン/オフにすることができます。
- Component プロパティー：CxfComponent オブジェクトには、そのコンポーネントから作成されたエンドポイントのデフォルトを設定できる allowStreaming プロパティーもあります。
- グローバルシステムプロパティー：org.apache.camel.component.cxf.streaming のシステムプロパティーを false に追加して、オフにできます。これにより、グローバルのデフォルトが設定されますが、上記の endpoint プロパティーを設定すると、そのエンドポイントに対してこの値が上書きされます。

汎用 CXF ディスパッチモードの使用

2.8.0 以降、camel-cxf コンポーネントは汎用 CXF ディスパッチモード をサポートします。これは、任意の構造のメッセージを転送できます（つまり、特定の XML スキーマにバインドされていません）。このモードを使用するには、CXF エンドポイントの wsdlURL および serviceClass 属性の指定を省略します。

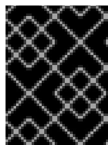
```
<cxf:cxfEndpoint id="testEndpoint" address="http://localhost:9000/SoapContext/SoapAnyPort">
  <cxf:properties>
    <entry key="dataFormat" value="PAYLOAD"/>
  </cxf:properties>
</cxf:cxfEndpoint>
```

デフォルトの CXF ディスパッチクライアントは特定の SOAPAction ヘッダーを送信しません。そのため、ターゲットサービスが特定の SOAPAction 値を必要とする場合、キー SOAPAction （大文字小文字の区別なし）を使用して Camel ヘッダーに提供されます。

第30章 CXF BEAN コンポーネント

CXF BEAN コンポーネント(2.0 以降)

cxfbean: コンポーネントを使用すると、他の Camel エンドポイントがエクスチェンジを送信し、Web サービス Bean オブジェクトを呼び出すことができます。現在、JAXRS および JAXWS (Camel 2.1 の新機能)アノテーション付きサービス Bean のみをサポートしています。



重要

CxfBeanEndpoint は **ProcessorEndpoint** であるため、コンシューマーはありません。Bean コンポーネントと同様に機能します。

URI 形式

```
cxfbean:serviceBeanRef
```

serviceBeanRef は、サービス Bean オブジェクトを検索するためのレジストリーキーです。**serviceBeanRef** が **List** オブジェクトを参照する場合、**List** の要素はエンドポイントによって許可されるサービス Bean オブジェクトです。

オプション

名前	説明	例	必須？	デフォルト値
bus	# 表記で指定された CXF バス参照。参照オブジェクトは **org.apache.cxf.Bus** のインスタンスである必要があります。	**bus=#busName**	いいえ	CXF Bus Factory によって作成されたデフォルトのバス
cxfBeanBinding	# 表記で指定された CXF Bean バインディング。参照されるオブジェクトは、**org.apache.camel.component.cxf.cxfbean.CxfBeanBinding** のインスタンスである必要があります。	**cxfBinding=#bindingName**	いいえ	**DefaultCxfBeanBinding**

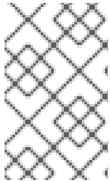
headerFilterStrategy	# 表記で指定されたヘッダーフィルターストラテジー。参照されるオブジェクトは **org.apache.camel.spi.HeaderFilterStrategy** のインスタンスである必要があります。	**headerFilterStrategy=#strategyName**	いいえ	**CxfHeaderFilterStrategy**
populateFromClass	2.3 以降、このオプションが **false に設定されていない限り、POJO のアノテーションが付けられた wsdlLocation は無視されます（デフォルト）。**2.3 より前のバージョンでは、POJO のアノテーションが付けられた wsdlLocation は常に尊重され、無視することはできません。	**true、false**	いいえ	**true**
providers	2.5 以降、CXFRS エンドポイントのプロバイダーを設定します。	**providers=#providerRef1,#providerRef2**	いいえ	**null**
setDefaultBus	CXF エンドポイントがバスを単独で作成すると、デフォルトのバスを設定します。	**true、false**	いいえ	**false**

HEADERS

名前	説明	タイプ	必須？	デフォルト値	In/Out	例

CamelHttpCharacterEncoding (before 2.0-m2: **CamelCxfBeanCharacterEncoding**)	文字エンコーディング	**文字列**	いいえ	なし	In	ISO-8859-1
CamelContentType (before 2.0-m2: **CamelCxfBeanContentType**)	コンテンツタイプ	**文字列**	いいえ	**/**	In	**text/xml**
CamelHttpBaseUri (2.0-m3 以前：**CamelCxfBeanRequestBasePath**)	このヘッダーの値は、CXF メッセージで **Message.BASE_PATH** プロパティーとして設定されます。これは CXF JAX-RS 処理に必要です。基本的には、要求 URI のスキーム、ホスト、およびポートの部分です。	**文字列**	はい	Camel エクスチェンジのソースエンドポイントのエンドポイント URI	In	http://localhost:9000
CamelHttpPath (before 2.0-m2: **CamelCxfBeanRequestPat{}**h)	要求 URI のパス	**文字列**	はい	なし	In	**consumer/123**

CamelHttpMethod (before 2.0-m2: **CamelCxfBeanVerb**)	RESTful リクエストの動詞	**文字列**	はい	なし	In	を **取得** します。 を配置、 投稿、**削除**
CamelHttpResponseCode	HTTP 応答コード	**整数**	いいえ	なし	Out	200



注記

現在、CXF Bean コンポーネントは Jetty HTTP コンポーネントでテストされており、変換を必要とせずに Jetty HTTP コンポーネントからのヘッダーを理解することができます。

作業例

この例は、Jetty HTTP サーバーを起動するルートを作成する方法を示しています。ルートは CXF Bean にリクエストを送信し、JAXRS アノテーション付きサービスを呼び出します。

*まず、以下のようにルートを作成します。**from** エンドポイントは、ポート 9000 でリッスンする Jetty HTTP エンドポイントです。RESTful リクエスト URI はエンドポイントの URI http://localhost:9000 と完全に一致しないため、**matchOnUriPrefix** オプションを **true** に設定する必要があります。*

```
<route>
 <from uri="jetty:http://localhost:9000?matchOnUriPrefix=true" />
 <to uri="cxfbean:customerServiceBean" />
 <to uri="mock:endpointA" />
</route>
```

***to** エンドポイントは、Bean が**customerServiceBean** という名前の CXF Bean です。名前はレジストリーから検索されます。次に、サービス Bean が Spring レジストリーで利用可能であることを確認します。Spring 設定で Bean 定義を作成します。この例では、（要素が 1 つ）サービス Bean のリストを作成します。List なしで単一の Bean のみを作成できます。*

```
<util:list id="customerServiceBean">
 <bean class="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService" />
</util:list>

<bean class="org.apache.camel.wsdl_first.PersonImpl" id="jaxwsBean" />
```

それがそれです。ルートが起動すると、Web サービスはビジネスに準備が整います。HTTP クライアントは要求および応答を受信できます。

```
url = new URL("http://localhost:9000/customerservice/orders/223/products/323");
in = url.openStream();
assertEquals("{\"Product\":{\"description\":\"product 323\",\"id\":323}}",
CxfUtils.getStringFromInputStream(in));
```

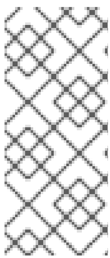
第31章 CXFRS

CXFRS コンポーネント

cxfrs: コンポーネントは、CXF でホストされる JAX-RS サービスに接続するためのApache CXF との統合を提供します。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cxf</artifactId>
  <version>x.x.x</version>  <!-- use the same version as your Camel core version -->
</dependency>
```



注記

CXF をコンシューマーとして使用する場合、CXF Bean コンポーネントを使用すると、処理からメッセージペイロードを受け取る方法を RESTful または SOAP Web サービスとして消費できます。これは、多量のトランスポートを使用して Web サービスを消費する可能性があります。Bean コンポーネントの設定も簡単で、Camel および CXF を使用して Web サービスを実装する最も高速な方法を提供します。

URI 形式

```
cxfrs://address?options
```

ここでの address は CXF エンドポイントのアドレスを表します。

```
cxfrs:bean:rsEndpoint
```

rsEndpoint は、CXFRS クライアントまたはサーバーを表す Spring Bean の名前を表します。

上記のいずれかのスタイルでは、以下のように URI にオプションを追加できます。

```
cxfrs:bean:cxfEndpoint?resourceClasses=org.apache.camel.rs.Example
```

オプション

名前	説明	例	必須？	デフォルト値
resourceClasses	REST サービスとしてエクスポートするリソースクラス。複数のクラスはコンマで区切ることができます。	resourceClasses=org.apache.camel.rs.Example1,org.apache.camel.rs.Exchange2	いいえ	なし

httpClientAPI	**Apache Camel 2.1 の新** 機能。true の場合、CxfRsProducer は HttpClientAPI を使用してサービスを呼び出します。	**httpClientAPI=true**	No	**true**
synchronous	2.5 の新機能として、このオプションにより、CxfRsConsumer が sync または async API を使用して基礎となる作業を行うよう決定します。デフォルト値は false です。これは、デフォルトで非同期 API の使用を試みることを意味します。	**synchronous=true**	No	**false**
throwExceptionOnFailure	2.6 の新機能として、このオプションは CxfRsProducer に対してリターンコードを検査するように指示し、リターンコードが 207 を超える場合は Exception を生成します。	**throwExceptionOnFailure=true**	No	**true**

maxClientCacheSize	2.6 の新機能として、In メッセージヘッダー **CamelDestinationOverrideUrl** を設定して、ルートに定義されたターゲットの宛先 Web サービスまたは REST サービスを動的に上書きできるようになりました。この実装は、CXF クライアントまたは **CxfProvider** および **CxfRsProvider** で **ClientFactoryBean** をキャッシュします。このオプションを使用すると、キャッシュの最大サイズを設定できます。	**maxClientCacheSize=5**	いいえ	10
setDefaultBus	**非推奨：** Camel 2.16 以降では **defaultBus** オプションを使用します。CXF エンドポイントがバスを単独で作成すると、デフォルトのバスを設定します。	**setDefaultBus=true**	いいえ	**false**
defaultBus	CXF エンドポイントがバスを単独で作成すると、デフォルトのバスを設定します。	**defaultBus=true**	いいえ	**false**

bus	2.9.0 の新機能 CXF Bus Factory によって作成されるデフォルトのバス。レジストリーからバスオブジェクトを参照するには、**\#** 表記を使用します。参照オブジェクトは **org.apache.cxf.Bus** のインスタンスである必要があります。	**bus=#busName**	いいえ	**なし**

bindingStyle	**2.11 以降。**Camel との間でリクエストと応答をマッピングする方法を設定します。以下の 2 つの値を使用できます。 ● **SimpleConsumer** => Consuming a REST Request with the Simple Binding Style below を参照してください。 ● **Default** => （デフォルトスタイル）。コンシューマーの場合、これは **MessageContentsList** でルートに渡され、ルートに低レベルの処理が必要になります。 ● custom => では、**binding** オプションを使用してカスタムバインディングを指定できます。	**bindingStyle=SimpleConsumer**	いいえ	デフォルト

binding	カスタムの **CxfRsBinding** 実装を指定して、未加工の CXF 要求および応答オブジェクトの低レベルの処理を実行できます。実装は Camel レジストリーにバインドする必要があり、ハッシュ(#)表記を使用して参照する必要があります。	**binding=#myBinding**	No	**DefaultCxfRsBinding**
providers	**Camel 2.12.2 以降**、カスタムの JAX-RS プロバイダーリストを CxfRs エンドポイントに設定します。	**providers=#MyProviders**	いいえ	なし
schemaLocations	**Camel 2.12.2 以降**、受信 XML または JAXB 駆動型 JSON の検証に使用できるスキーマの場所を設定します。	**schemaLocations=#MySchemaLocations**	いいえ	なし
features	**Camel 2.12.3 以降**、機能リストを CxfRs エンドポイントに設定します。	**features=#MyFeatures**	いいえ	なし
properties	**Camel 2.12.4 以降**、プロパティーを CxfRs エンドポイントに設定します。	**properties=#MyProperties**	いいえ	なし
inInterceptors	**Camel 2.12.4 以降**、inInterceptors を CxfRs エンドポイントに設定します。	**inInterceptors=#MyInterceptors**	いいえ	なし
outInterceptors	**Camel 2.12.4 以降**、outInterceptor を CxfRs エンドポイントに設定します。	**outInterceptors=#MyInterceptors**	いいえ	なし

inFaultInterceptors	**Camel 2.12.4 以降**、inFaultInterceptors を CxfRs エンドポイントに設定します。	**inFaultInterceptors=#MyInterceptors**	いいえ	なし
outFaultIntercetpros	**Camel 2.12.4 以降**、outFaultInterceptors を CxfRs エンドポイントに設定します。	**outFaultInterceptors=#MyInterceptors**	いいえ	なし
continuationTimeout	**Camel 2.14.0 以降**、このオプションは CXF サーバーが Jetty または Servlet トランスポートを使用している場合にデフォルトで CxfConsumer で使用できる CXF 継続タイムアウトを設定するために使用されます。(**Camel 2.14.** 0 よりも前のバージョンでは、CxfConsumer は継続タイムアウトを 0 に設定するだけで、継続一時停止操作がタイムアウトしないことを意味します)。	**continuationTimeout=800000**	いいえ	30000
ignoreDeleteMethodMessageBody	**Camel 2.14.1 以降**、このオプションは HTTP API の使用時に **DELETE** メソッドのメッセージボディーを無視するように **CxfRsProducer** に指示するために使用されます。	**ignoreDeleteMethodMessageBody=true**	いいえ	false

modelRef	**Camel 2.14.2 以降**、このオプションは、アノテーションのないリソースクラスに役立つ モデルファイル を指定するために使用されます。 **Camel 2.15 以降**、このオプションは、ドキュメントのみのエンドポイントをエミュレートするためにサービスクラスを指定せずにモデルファイルを参照できます。	**modelRef=classpath:/CustomerServiceModel.xml**	いいえ	なし
performInvocation	Camel 2.15 の場合、オプションが true の場合、Camel **は** リソースクラスインスタンスの呼び出しを実行し、応答オブジェクトをエクスチェンジに配置してさらに処理します。	**performInvocation=true**	いいえ	false
loggingFeatureEnabled	このオプションは、インバウンドおよびアウトバウンド REST メッセージをログに書き込む CXF ロギング機能を有効にします。	-	いいえ	**false**
skipFaultLogging	このオプションは、***PhaseInterceptorChain*** がキャッチする Fault のロギングをスキップするかどうかを制御します。	-	いいえ	**false**
loggingSizeLimit	ロギング機能を有効にすると、このオプションを使用してロガーが出力するバイト数を制限します。	-	いいえ	**0**

propagateContexts	Camel 2.15 When **true 以降**、JAXRS UriInfo、HttpHeaders、Request、および SecurityContext コンテキストは、型指定された Camel エクスチェンジプロパティーとしてカスタム CXFRS プロセッサーで利用できます。これらのコンテキストは、JAX-RS API を使用して現在のリクエストを分析するために使用できます。			
loggingFeatureEnabled	このオプションは、インバウンドおよびアウトバウンド REST メッセージをログに書き込む CXF Logging 機能を有効にします。		いいえ	**false**
skipFaultLogging	このオプションは、PhaseInterceptor Chain がキャッチする Fault のロギングをスキップするかどうかを制御します。		いいえ	**false**
loggingSizeLimit	ロギング機能が有効な場合、ロガーが出力されるバイト数の合計サイズを制限します。		いいえ	**0**

Spring 設定を介して CXF REST エンドポイントを設定することもできます。CXF REST クライアントと CXF REST Server の間には多くの違いがあるため、それらに異なる設定が提供されます。詳細は、スキーマファイル と CXF JAX-RS ドキュメント を参照してください。

APACHE CAMEL で REST エンドポイントを設定する方法

camel-cxf スキーマファイル には、REST エンドポイント定義に 2 つの要素があります。cxf:rsServer for REST コンシューマー、cxf:rsClient (REST プロデューサーの場合) ここでは、Apache Camel REST サービスルート設定の例があります。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://camel.apache.org/schema/cxf"
    xmlns:jaxrs="http://cxf.apache.org/jaxrs"
```

```
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd
        http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd
    ">
  <!-- Defined the real JAXRS back end service  -->
  <jaxrs:server id="restService"
         address="http://localhost:9002/rest"
         staticSubresourceResolution="true">
    <jaxrs:serviceBeans>
      <ref bean="customerService"/>
    </jaxrs:serviceBeans>
  </jaxrs:server>

  <!--bean id="jsonProvider" class="org.apache.cxf.jaxrs.provider.JSONProvider"/-->

  <bean id="customerService"
class="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService" />

  <!-- Defined the server endpoint to create the cxf-rs consumer -->
  <cxf:rsServer id="rsServer" address="http://localhost:9000/route"
    serviceClass="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService"
      loggingFeatureEnabled="true" loggingSizeLimit="20" skipFaultLogging="true"/>

  <!-- Defined the client endpoint to create the cxf-rs consumer -->
  <cxf:rsClient id="rsClient" address="http://localhost:9002/rest"
    serviceClass="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService"
      loggingFeatureEnabled="true" skipFaultLogging="true"/>

  <!-- The camel route context -->
  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
       <from uri="cxfrs://bean://rsServer"/>
       <!-- We can remove this configure as the CXFRS producer is using the HttpAPI by default
-->
       <setHeader headerName="CamelCxfRsUsingHttpAPI">
         <constant>True</constant>
       </setHeader>
       <to uri="cxfrs://bean://rsClient"/>
    </route>
  </camelContext>

</beans>
```

メッセージヘッダーから CXF プロデューサーアドレスを上書きする方法

camel-cxfrs プロデューサーは、CamelDestinationOverrideUrl のキーでメッセージを設定することで、サービスアドレスを上書きすることをサポートします。

```
// set up the service address from the message header to override the setting of CXF endpoint
exchange.getIn().setHeader(Exchange.DESTINATION_OVERRIDE_URL,
constant(getServiceAddress()));
```

REST リクエストの使用 - シンプルバインディングスタイル

Camel 2.11 から利用可能

Default バインディングスタイルは代わりに低レベルであるため、ユーザーはルートに送信される **MessageContentsList** オブジェクトを手動で処理する必要があります。そのため、ルートロジックと JAX-RS 操作のメソッド署名およびパラメーターインデックスが密接に結合されます。悪意、難易度、エラーが発生しやすくなります。

一方、**SimpleConsumer** バインディングスタイルは以下のマッピングを実行し、要求データが Camel Message 内でアクセスしやすく します。

- JAX-RS パラメーター(@HeaderParam、@QueryParam など)は IN メッセージヘッダーとして注入されます。ヘッダー名は アノテーションの値と一致します。
- リクエストエンティティー(POJO またはその他のタイプ)は IN メッセージボディーになります。JAX-RS メソッド署名で単一のエンティティーを識別できない場合は、元の **MessageContentsList** にフォールバックします。
- バイナリー **@Multipart** ボディーの部分は IN message attachments、**DataHandler**、**InputStream DataSource**、および CXF の **Attachment** クラスをサポートします。
- バイナリー以外の **@Multipart** ボディーの部分は IN メッセージヘッダーとしてマッピングされます。ヘッダー名は Body Part 名と一致します。

さらに、以下のルールは Response マッピングに適用されます。

- メッセージボディータイプが **javax.ws.rs.core.Response**（ユーザー構築応答）と異なる場合、新しい **Response** が作成され、メッセージボディーはエンティティーとして設定されます(null 以外)。応答ステータスコードは **Exchange.HTTP_RESPONSE_CODE** ヘッダーから取得されます。存在しない場合には、デフォルトは 200 OK に設定されます。
- メッセージボディータイプが **javax.ws.rs.core.Response** と等しい場合、ユーザーはカスタム応答を構築するため、尊重され、最終的な応答になります。
- いずれの場合も、カスタムまたはデフォルトの **HeaderFilterStrategy** で許可されている Camel ヘッダーが HTTP 応答に追加されます。

シンプルバインディングスタイルの有効化

このバインディングスタイルを有効にするには、コンシューマーエンドポイントの **bindingStyle** パラメーターを **SimpleConsumer** の値に設定します。

```
from("cxfrs:bean:rsServer?bindingStyle=SimpleConsumer")
 .to("log:TEST?showAll=true");
```

異なるメソッド署名を使用した要求バインディングの例

以下は、Simple バインディングの想定される結果と共にメソッド署名の一覧です。

public Response doAction (BusinessObject request); Request payload is placed in IN メッセージボディーは IN メッセージボディーに配置され、元の MessageContentsList を置き換えます。

public Response doAction (BusinessObject request, @HeaderParam ("abcd") String abcd,

@QueryParam ("defg") String defg); *Request payload placed in IN メッセージボディーは、元の MessageContentsList を置き換えます。どちらのリクエストパラメーターも、abcd と defg という名前の IN メッセージヘッダーとしてマッピングされます。*

public Response doAction (@HeaderParam ("abcd") String abcd, @QueryParam ("defg") String defg); *Both request params mapped as IN メッセージヘッダー with names abcd and defg.元の MessageContentsList には 2 つのパラメーターのみが含まれている場合でも保持されます。*

public Response doAction (@Multipart (value="body1") BusinessObject request, @Multipart (value="body2") BusinessObject request2); *最初のパラメーターは body1 という名前のヘッダーとして転送され、2 番目のパラメーターはヘッダー body2 としてマッピングされます。元の MessageContentsList は IN メッセージボディーとして保持されます。*

public Response doAction (InputStream abcd); *The InputStream is unwrapped from the MessageContentsList and preserved as the IN message body.*

public Response doAction (DataHandler abcd); *DataHandler は MessageContentsList からラップ解除され、IN メッセージボディーとして保持されます。*

シンプルバインディングスタイルの例

このメソッドに JAX-RS リソースクラスがあるとします。

```
    @POST @Path("/customers/{type}")
    public Response newCustomer(Customer customer, @PathParam("type") String type,
@QueryParam("active") @DefaultValue("true") boolean active) {
        return null;
    }
```

以下のルートでサービスを提供します。

```
    from("cxfrs:bean:rsServer?bindingStyle=SimpleConsumer")
        .recipientList(simple("direct:${header.operationName}"));

    from("direct:newCustomer")
        .log("Request: type=${header.type}, active=${header.active}, customerData=${body}");
```

XML ペイロードを使用した以下の HTTP 要求(Customer DTO が JAXB-annotated の場合)。

```
POST /customers/gold?active=true

Payload:
<Customer>
  <fullName>Raul Kripalani</fullName>
  <country>Spain</country>
  <project>Apache Camel</project>
</Customer>
```

メッセージを出力します。

```
Request: type=gold, active=true, customerData=<Customer.toString() representation>
```

要求および書き込み応答の処理方法の例は、を参照して ください。

REST リクエストの使用 - デフォルトバインディングスタイル

CXF JAX-RS フロントエンドは JAX-RS (JSR-311) API を実装しているため、リソースクラスを REST サービスとしてエクスポートできます。また、CXF Invoker API を使用して、REST リクエストを通常の Java オブジェクトメソッド呼び出しに変換します。camel-restlet コンポーネントとは異なり、エンドポイント内で URI テンプレートを指定する必要はありません。CXF は JSR-311 仕様に従って、REST 要求 URI をリソースクラスメソッドマッピングに処理します。Apache Camel で行う必要があるのは、このメソッドリクエストを適切なプロセッサーまたはエンドポイントに委譲することです。

以下は CXFRS ルートの例です。

```
private static final String CXF_RS_ENDPOINT_URI = "cxfrs://http://localhost:" + CXT + "/rest?
resourceClasses=org.apache.camel.component.cxf.jaxrs.testbean.CustomerServiceResource
";

 protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() {
            errorHandler(new NoErrorHandlerBuilder());
            from(CXF_RS_ENDPOINT_URI).process(new Processor() {

                public void process(Exchange exchange) throws Exception {
                    Message inMessage = exchange.getIn();
                    // Get the operation name from in message
                    String operationName = inMessage.getHeader(CxfConstants.OPERATION_NAME,
String.class);
                    if ("getCustomer".equals(operationName)) {
                        String httpMethod = inMessage.getHeader(Exchange.HTTP_METHOD,
String.class);
                        assertEquals("Get a wrong http method", "GET", httpMethod);
                        String path = inMessage.getHeader(Exchange.HTTP_PATH, String.class);
                        // The parameter of the invocation is stored in the body of in message
                        String id = inMessage.getBody(String.class);
                        if ("/customerservice/customers/126".equals(path)) {
                            Customer customer = new Customer();
                            customer.setId(Long.parseLong(id));
                            customer.setName("Willem");
                            // We just put the response Object into the out message body
                            exchange.getOut().setBody(customer);
                        } else {
                            if ("/customerservice/customers/400".equals(path)) {
                                // We return the remote client IP address this time
                                org.apache.cxf.message.Message cxfMessage =
inMessage.getHeader(CxfConstants.CAMEL_CXF_MESSAGE,
org.apache.cxf.message.Message.class);
                                ServletRequest request = (ServletRequest)
cxfMessage.get("HTTP.REQUEST");
                                String remoteAddress = request.getRemoteAddr();
                                Response r = Response.status(200).entity("The remoteAddress is " +
remoteAddress).build();
                                exchange.getOut().setBody(r);
                                return;
                            }
                            if ("/customerservice/customers/123".equals(path)) {
                                // send a customer response back
                                Response r = Response.status(200).entity("customer response
```

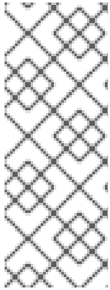
```
back!").build();
                                exchange.getOut().setBody(r);
                                return;
                            }
                            if ("/customerservice/customers/456".equals(path)) {
                                Response r = Response.status(404).entity("Can't found the customer with
uri " + path).build();
                                throw new WebApplicationException(r);
                            } else {
                                throw new RuntimeCamelException("Can't found the customer with uri " +
path);
                            }
                        }
                    }
                    if ("updateCustomer".equals(operationName)) {
                        assertEquals("Get a wrong customer message header", "header1;header2",
inMessage.getHeader("test"));
                        String httpMethod = inMessage.getHeader(Exchange.HTTP_METHOD,
String.class);
                        assertEquals("Get a wrong http method", "PUT", httpMethod);
                        Customer customer = inMessage.getBody(Customer.class);
                        assertNotNull("The customer should not be null.", customer);
                        // Now you can do what you want on the customer object
                        assertEquals("Get a wrong customer name.", "Mary", customer.getName());
                        // set the response back
                        exchange.getOut().setBody(Response.ok().build());
                }
                }

            });
        }
    };
}
```

エンドポイントの設定に使用される対応するリソースクラスはインターフェイスとして定義されます。

```
@Path("/customerservice/")
public interface CustomerServiceResource {

    @GET
    @Path("/customers/{id}/")
    Customer getCustomer(@PathParam("id") String id);

    @PUT
    @Path("/customers/")
    Response updateCustomer(Customer customer);
 }
```

重要

デフォルトでは、JAX-RS リソースクラスは JAX-RS プロパティーのみ を設定するために使用されます。メソッドは、エンドポイントへのメッセージのルーティング中に実行されず、ルート自体はすべての処理を行います。



注記

*Camel 2.15 以降では、デフォルトモードの no-op サービス実装クラスではなく、インターフェイスのみを提供するだけで十分です。Camel 2.15 以降では、**performInvocation** オプションが有効になっていると、サービス実装が最初に呼び出されます。応答は Camel エクスチェンジに設定され、ルートの実行は通常どおり継続されます。これは、既存の JAX-RS 実装を Camel ルートに統合する場合や、カスタムプロセッサーでの JAX-RS 応答の後処理に役立ちます。*

HOW TO INVOKE THE REST SERVICE THROUGH CAMEL-CXFRS PRODUCER?

CXF JAXRS フロントエンドは プロキシーベースのクライアント API を実装し、この API はプロキシーを介してリモート REST サービスを呼び出すことができます。**camel-cxfrs** プロデューサーはこの プロキシー API に基づいています。メッセージヘッダーで操作名を指定し、メッセージボディーにパラメーターを準備するだけで、**camel-cxfrs** プロデューサーは適切な REST 要求を生成します。

以下に例を示します。

```
Exchange exchange = template.send("direct://proxy", new Processor() {

    public void process(Exchange exchange) throws Exception {
        exchange.setPattern(ExchangePattern.InOut);
        Message inMessage = exchange.getIn();
        setupDestinationURL(inMessage);
        // set the operation name
        inMessage.setHeader(CxfConstants.OPERATION_NAME, "getCustomer");
        // using the proxy client API
        inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_USING_HTTP_API,
Boolean.FALSE);
        // set a customer header
        inMessage.setHeader("key", "value");
        // set the parameters , if you just have one parameter
        // camel will put this object into an Object[] itself
        inMessage.setBody("123");
    }

});

// get the response message
Customer response = (Customer) exchange.getOut().getBody();

assertNotNull("The response should not be null ", response);
assertEquals("Get a wrong customer id ", String.valueOf(response.getId()), "123");
assertEquals("Get a wrong customer name", response.getName(), "John");
assertEquals("Get a wrong response code", 200,
exchange.getOut().getHeader(Exchange.HTTP_RESPONSE_CODE));
assertEquals("Get a wrong header value", "value", exchange.getOut().getHeader("key"));
```

CXF JAXRS フロントエンドは http 中心のクライアント API も提供し、**camel-cxfrs** プロデューサーからこの API を呼び出すこともできます。HTTP_PATH および Http メソッドを指定し、URI オプション httpClientAPI を使用するか、**CxfConstants.CAMEL_CXF_RS_USING_HTTP_API** でメッセージヘッダーを設定して、プロデューサーに HTTP 中心のクライアントの使用を知らせる必要があります。応答オブジェクトは、**CxfConstants.CAMEL_CXF_RS_RESPONSE_CLASS** で指定するタイプクラスに変換できます。

```
Exchange exchange = template.send("direct://http", new Processor() {

    public void process(Exchange exchange) throws Exception {
        exchange.setPattern(ExchangePattern.InOut);
        Message inMessage = exchange.getIn();
        setupDestinationURL(inMessage);
        // using the http central client API
        inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_USING_HTTP_API, Boolean.TRUE);
        // set the Http method
        inMessage.setHeader(Exchange.HTTP_METHOD, "GET");
        // set the relative path
        inMessage.setHeader(Exchange.HTTP_PATH, "/customerservice/customers/123");
        // Specify the response class , cxfrs will use InputStream as the response object type
        inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_RESPONSE_CLASS,
Customer.class);
        // set a customer header
        inMessage.setHeader("key", "value");
        // since we use the Get method, so we don't need to set the message body
        inMessage.setBody(null);
    }

});

// get the response message
Customer response = (Customer) exchange.getOut().getBody();

assertNotNull("The response should not be null ", response);
assertEquals("Get a wrong customer id ", String.valueOf(response.getId()), "123");
assertEquals("Get a wrong customer name", response.getName(), "John");
assertEquals("Get a wrong response code", 200,
exchange.getOut().getHeader(Exchange.HTTP_RESPONSE_CODE));
assertEquals("Get a wrong header value", "value", exchange.getOut().getHeader("key"));
```

Apache Camel 2.1 以降、CXFRS HTTP 中心クライアントの CXFRS URI からクエリーパラメーターを指定することもサポートします。

```
Exchange exchange = template.send("cxfrs://http://localhost:" + getPort2() + "/" +
getClass().getSimpleName() + "/testQuery?httpClientAPI=true&q1=12&q2=13"
```

Dynamical ルーティングをサポートするには、**CxfConstants.CAMEL_CXF_RS_QUERY_MAP** ヘッダーを使用してパラメーターマップを設定して URI のクエリーパラメーターを上書きすることができます。

```
Map<String, String> queryMap = new LinkedHashMap<String, String>();
queryMap.put("q1", "new");
queryMap.put("q2", "world");
inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_QUERY_MAP, queryMap);
```

第32章 DATAFORMAT コンポーネント

データフォーマットのコンポーネント

Camel 2.12 以降で利用可能

dataformat: コンポーネントでは、Data Format を Camel コンポーネントとして使用できます。

URI 形式

```
dataformat:name:(marshal|unmarshal)[?options]
```

name は、データフォーマット の名前です。その後、操作の後に、を マーシャリング または アンマーシャリング する必要があります。オプションは、使用中の データフォーマット の設定に使用されます。サポートされるオプションは、Data Format のドキュメントを参照してください。

サンプル

たとえば、JAXB Data Format を使用するには、以下を実行します。

```
from("activemq:My.Queue").
  to("dataformat:jaxb:unmarshal?contextPath=com.acme.model").
  to("mqseries:Another.Queue");
```

XML DSL では以下を行います。

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="activemq:My.Queue"/>
    <to uri="dataformat:jaxb:unmarshal?contextPath=com.acme.model"/>
    <to uri="mqseries:Another.Queue"/>
  </route>
</camelContext>
```

- データ形式

第33章 DATASET

データセットコンポーネント

DataSet コンポーネント(1.3.0 以降で利用可能)は、システムの負荷およびソークテストを簡単に実行するメカニズムを提供します。これは、メッセージのソースとして、データセットを受け取ることをアサートする方法の両方で、DataSet インスタンス を作成することを可能にすることで機能します。

Apache Camel は、データセットの送信時に スループットロガー を使用します。

URI 形式

> ***dataset:name[?options]***

name は、レジストリーでDataSet インスタンス を見つけるために使用されます。

*Apache Camel には、**org.apache.camel.component.dataset.DataSet** のサポート実装が同梱されており、独自の DataSet を実装するためのベースとして使用できる **org.apache.camel.component.dataset.DataSetSupport** クラスです。Apache Camel には、テストに使用できる実装(**org.apache.camel.component.dataset.SimpleDataSet**、**org.apache.camel.component.dataset.ListDataSet**、および **org.apache.camel.component.dataset.FileDataSet**)も同梱されます。これらはすべて **DataSetSupport** を拡張します。*

オプション

オプション	デフォルト	説明
produceDelay	3	遅延をミリ秒単位で指定できます。これにより、プロデューサーが一時停止し、低速なプロデューサーをシミュレートします。このオプションを -1 に設定し、遅延なしを強制しない限り、最低 3 ミリ秒の遅延を使用します。
consumeDelay	0	ミリ秒で遅延を指定できるようにします。これにより、コンシューマーは低速なコンシューマーをシミュレートするために一時停止します。
preloadSize	0	ルートが初期化を完了する前に事前ロードするメッセージ数を設定します（送信）。
initialDelay	1000	Camel 2.1: メッセージの送信を開始する前に待機する期間（ミリ秒単位）。

minRate	0	DataSet にこの数のメッセージが含まれるまで待ちます。
dataSetIndex	**lenient**	Camel 2.17: **CamelDataSetIndex** ヘッダーの動作を制御します。サポートされる値は **strict**、**lenient** および **off** です。Camel 2.17 より前のデフォルトの動作は、**dataSetIndex=strict** を設定して復元できます。 コンシューマーの場合： **strict** または、 **lenient** **CamelDataSetIndex** ヘッダーは常に設定されます。 **off** **CamelDataSetIndex** ヘッダーは設定されません。 プロデューサーの場合： **strict** **CamelDataSetIndex** ヘッダーが存在し、ヘッダーの値を確認します。 **lenient** **CamelDataSetIndex** ヘッダーが存在する場合は、ヘッダーの値が検証されます。ヘッダーが存在しない場合は設定されます。 **off** **CamelDataSetIndex** ヘッダーがある場合、ヘッダーの値は検証されません。ヘッダーが存在しない場合は設定されません。

*URI にクエリーオプションは **?option=value&option=value&..** の形式で追加できます。*

DATASET の設定

Apache Camel は、DataSet インターフェイスを実装する Bean のレジストリーでルックアップします。そのため、以下のように独自の DataSet を登録できます。

```
<bean id="myDataSet" class="com.mycompany.MyDataSet">
  <property name="size" value="100"/>
</bean>
```

例

たとえば、メッセージのセットがキューに送信され、メッセージを失うことなくキューから消費されることをテストするには、以下を実行します。

```
// send the dataset to a queue
from("dataset:foo").to("activemq:SomeQueue");

// now lets test that the messages are consumed correctly
from("activemq:SomeQueue").to("dataset:foo");
```

上記はレジストリーを検索して、メッセージの作成に使用される foo DataSet インスタンスを見つけます。

*次に、以下に示すように **SimpleDataSet** を使用するなど、DataSet 実装を作成し、データセットのサイズやメッセージがどのように見えるかなどを設定します。*

DATASETSUPPORT のプロパティー

プロパティー	タイプ	デフォルト	説明
defaultHeaders	**Map<String,Object>**	**null**	デフォルトのメッセージボディーを指定します。SimpleDataSet の場合、これは一定のペイロードです。ただし、メッセージごとにカスタムペイロードを作成する場合は、**DataSetSupport** の独自の導出を作成します。
outputTransformer	**org.apache.camel.Processor**	null	
size	**long**	**10**	送信/消費するメッセージの数を指定します。
reportCount	**long**	**-1**	進捗を報告する前に受信するメッセージの数を指定します。大規模な負荷テストの進捗を表示するのに便利です。< 0 の場合、**size** / 5、が 0 の場合は **size**、それ以外の場合は **reportCount** 値に設定されます。

SIMPLEDATASET

***SimpleDataSet** は **DataSetSupport** を拡張し、デフォルトのボディーを追加します。*

表33.1 SimpleDataSet の追加プロパティー

プロパティー	タイプ	デフォルト	説明
defaultBody	**Object**	**<hello>world!</hello>**	デフォルトのメッセージボディーを指定します。SimpleDataSet の場合、これは一定のペイロードです。ただし、メッセージごとにカスタムペイロードを作成する場合は、**DataSetSupport** の独自の導出を作成します。

LISTDATASET

ListDataSet は DataSetSupport を拡張し、デフォルトの本文の一覧を追加します。

表33.2 ListDataSet の追加プロパティー

プロパティー	タイプ	デフォルト	説明
defaultBodies	**Object**	**<hello>world!</hello>**	
size	**long**	**10**	

FILEDATASET

SimpleDataSet は ListDataSet を拡張し、ファイルから本文を読み込むサポートを追加します。

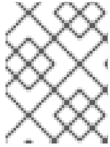
表33.3 FileDataSet の追加プロパティー

プロパティー	タイプ	デフォルト	説明
sourceFile	**String**	null	
delimiter	**String**	\z	

第34章 DIRECT

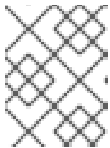
DIRECT コンポーネント

direct: プロデューサーがメッセージエクスチェンジを送信する際に、コンポーネントはコンシューマーを直接同期呼び出しを提供します。このエンドポイントは、同じ Camel コンテキストの既存ルートを接続するために使用できます。



注記

SEDA コンポーネントは、プロデューサーがメッセージエクスチェンジを送信するときに、コンシューマーの非同期呼び出しを提供します。



注記

VM コンポーネントは、同じ JVM で実行されている限り、Camel コンテキスト間の接続を提供します。

URI 形式

```
direct:someName[?options]
```

someName には、エンドポイントを一意に識別する任意の文字列を指定できます。

オプション

名前	デフォルト値	説明
block	false	Camel 2.11.1: アクティブなコンシューマーのないダイレクトエンドポイントにメッセージを送信する場合は、プロデューサーにブロックし、コンシューマーがアクティブになるのを待つようにプロデューサーに指示することができます。
timeout	30000	Camel 2.11.1: ブロックが有効な場合に使用するタイムアウト値。
failIfNoConsumers	true	Camel 2.16.0: アクティブなコンシューマーのない DIRECT エンドポイントに送信するときにプロデューサーが例外を出力して失敗するかどうかを示します。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

サンプル

以下のルートでは、direct コンポーネントを使用して 2 つのルートをリンクします。

```
from("activemq:queue:order.in")
    .to("bean:orderServer?method=validate")
    .to("direct:processOrder");

from("direct:processOrder")
    .to("bean:orderService?method=process")
    .to("activemq:queue:order.out");
```

Spring DSL を使用した例:

```
 <route>
   <from uri="activemq:queue:order.in"/>
   <to uri="bean:orderService?method=validate"/>
   <to uri="direct:processOrder"/>
</route>

<route>
   <from uri="direct:processOrder"/>
   <to uri="bean:orderService?method=process"/>
   <to uri="activemq:queue:order.out"/>
</route>
```

SEDA コンポーネントの例、どのように併用できるかも併せて参照してください。

- *SEDA*
- *VM*

第35章 DIRECT-VM

DIRECT VM コンポーネント

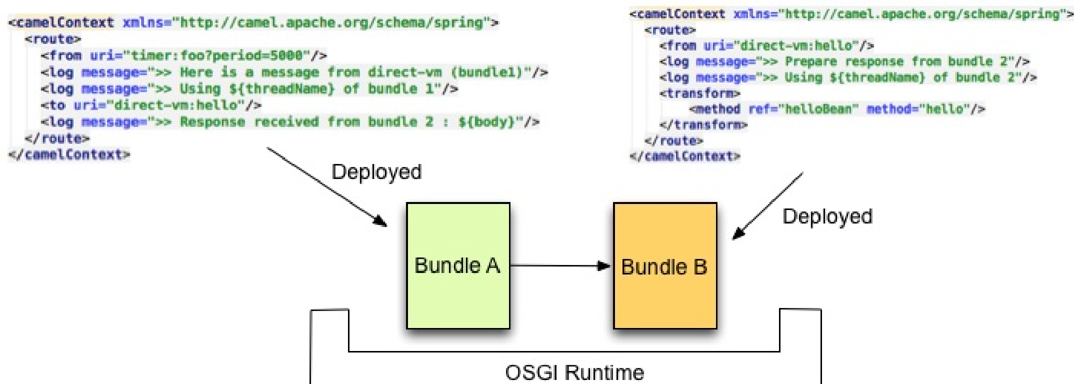
Camel 2.10 以降で利用可能

direct-vm: コンポーネントは、プロデューサーがメッセージエクスチェンジを送信するときに JVM のコンシューマーを直接同期呼び出しを提供します。このエンドポイントは、同じ Camel コンテキスト内の既存のルートと、同じ JVM の他の Camel コンテキストから接続するために使用できます。

このコンポーネントは、 Direct -VM が CamelContext インスタンス全体の通信をサポートする点で Direct コンポーネントとは異なります。そのため、このメカニズムを使用して Web アプリケーション全体で通信できます(camel-core.jar がシステム/ブートクラスパス上にある場合)。

ランタイム時に、既存のコンシューマーを停止し、新しいコンシューマーを開始することで、新しいコンシューマーにスワップできます。ただし、ある時点では、特定のエンドポイントに対して最大 1 つのアクティブなコンシューマーしか存在できません。

このコンポーネントを使用すると、以下の後に確認できるように、異なる OSGI バンドルにデプロイされたルートを接続することもできます。異なるバンドルで実行されている場合でも、Camel ルートは同じスレッドを使用します。トランザクションを使用してアプリケーションを開発するための自動スリア (Tx)。



URI 形式

```
direct-vm:someName
```

someName には、エンドポイントを一意に識別する任意の文字列を指定できます。

オプション

名前	デフォルト値	説明

block	false	Camel 2.11.1: アクティブなコンシューマーのないダイレクトエンドポイントにメッセージを送信する場合は、プロデューサーにブロックし、コンシューマーがアクティブになるのを待つようにプロデューサーに指示することができます。
timeout	30000	Camel 2.11.1: ブロックが有効な場合に使用するタイムアウト値。
failIfNoConsumers	true	Camel 2.16.0: アクティブなコンシューマーのない DIRECT エンドポイントに送信するときにプロデューサーが例外を出力して失敗するかどうかを示します。

サンプル

以下のルートでは、direct コンポーネントを使用して 2 つのルートをリンクします。

```
from("activemq:queue:order.in")
   .to("bean:orderServer?method=validate")
   .to("direct-vm:processOrder");
```

別の CamelContext （別の OSGi バンドルなど）で

```
from("direct-vm:processOrder")
   .to("bean:orderService?method=process")
   .to("activemq:queue:order.out");
```

Spring DSL を使用した例:

```
 <route>
   <from uri="activemq:queue:order.in"/>
   <to uri="bean:orderService?method=validate"/>
   <to uri="direct-vm:processOrder"/>
</route>

<route>
   <from uri="direct-vm:processOrder"/>
   <to uri="bean:orderService?method=process"/>
   <to uri="activemq:queue:order.out"/>
</route>
```

- *Direct*
- *SEDA*
- *VM*

第36章 DISRUPTOR

DISRUPTOR コンポーネント

Camel 2.12 以降で利用可能

中断： コンポーネントは、標準のSEDA コンポーネントと同様に非同期SEDA 機能を提供しますが、標準のSEDA で使用されるBlockingQueue の代わりにDisruptor を使用します。または、このコンポーネントで disruptor-vm: エンドポイントをサポートし、標準の 仮想マシン の代替手段を提供します。SEDA コンポーネントと同様に、中断のバッファー： エンドポイントは単一 のCamelContext 内でのみ表示され、永続性またはリカバリーに対するサポートは提供されません。*disruptor-vm:* エンドポイントは、CamelContexts インスタンス間の通信もサポートします。そのため、このメカニズムを使用して Web アプリケーション全体で通信を行うことができます(camel-disruptor.jar がシステム/ブート クラスパス上にある場合)。

SEDA または VM コンポーネントで Disruptor コンポーネントを使用することを選択する主な利点は、プロデューサーとマルチキャストされたコンシューマーまたは同時コンシューマーの間の競合が高いユースケースのパフォーマンスです。このような場合、スループットが大幅に増加し、レイテンシーが減少しています。競合のないシナリオのパフォーマンスは、SEDA および VM コンポーネントと類似しています。

Disruptor は、可能な限り SEDA および VM コンポーネントの動作を模倣して実装されます。それらの相違点は、以下のとおりです。

- 使用されるバッファーは、常にサイズでバインドされます（デフォルトの 1024 エクスチェンジ）。
- バッファーは常に禁止されるため、例外を出力する代わりにバッファーが満杯である間に Disruptor のデフォルト動作がブロックされます。このデフォルトの動作は コンポーネントに設定できます（オプションを参照）。
- Disruptor 予約は BrowsableEndpoint インターフェイスを実装しません。そのため、現在 Disruptor にあるエクスチェンジを取得できず、交換の量のみを取得できます。
- Disruptor では、コンシューマー（マルチキャストの有無）を静的に設定する必要があります。オンザフライでコンシューマーを追加または削除するには、Diruptor の保留中のすべてのエクスチェンジを完全にフラッシュする必要があります。
- 再設定の結果：Disruptor で送信されるデータは直接処理され、コンシューマーが 1 つ以上ある場合は 'gone' になり、結合後に新しいエクスチェンジのみを取得します。
- pollTimeout オプションは Disruptor コンポーネントではサポートされません。
- プロデューサーが完全な Disruptor をブロックすると、スレッド割り込みに応答しません。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-disruptor</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

> ***disruptor:someName[?options]***

または

> ***disruptor-vm:someName[?options]***

**someName* には、現在の CamelContext 内のエンドポイントを一意に識別する任意の文字列（または *disruptor-vm:* の場合のコンテキスト全体で）を指定できます。URI に、以下の形式でクエリーオプションを追加できます。*

> ***?option=value&option=value&...***

オプション

*以下のオプションはすべて、*disruptor:* および *disruptor-vm: *コンポーネントの両方に対して有効です。*

名前	デフォルト	説明
size	1024	Disruptors リングバッファーの最大容量。2 の最も近い累乗に事実上増加します。**注：** このオプションを使用する場合、キュー名で最初のエンドポイントが作成され、サイズを決定するのは最小限です。すべてのエンドポイントが同じサイズを使用するようにするには、すべてのエンドポイントで size オプションを設定するか、作成される最初のエンドポイントを設定します。
bufferSize		**コンポーネントのみ：** Disruptors リングバッファーのデフォルトサイズ（保持可能なメッセージ数の容量）。このオプションは、サイズが使用されていない場合に使用されます。
queueSize		**コンポーネントのみ：** SEDA コンポーネントとの最大互換性を維持するために bufferSize を指定する追加のオプション。
concurrentConsumers	1	同時スレッド処理エクスチェンジの数。

waitForTaskToComplete	IfReplyExpected	非同期タスクが完了するまで呼び出し元が待機するかどうかを指定するオプション。**Always**、**Never**、または **IfReplyExpected** の 3 つのオプションがサポートされます。最初の 2 つの値は自己説明です。最後の値 **IfReplyExpected** は、メッセージが RequestReply-based の場合にのみ待機します。非同期メッセージングの詳細は、非同期メッセージング を参照してください。
timeout	30000	プロデューサーが非同期タスクが完了するまで待機するタイムアウト（ミリ秒単位）。詳細は、**waitForTaskToComplete** および Async を参照してください。0 または負の値を使用して、タイムアウトを無効にすることができます。
defaultMultipleConsumers		**コンポーネントのみ：multipleConsumers** が指定されていない場合に使用されるこのコンポーネントによって作成されるエンドポイントに、複数のコンシューマーのデフォルト許可を設定できます。
multipleConsumers	false	複数のコンシューマーを許可するかどうかを指定します。有効にすると、Publish-Subscribe メッセージングに Disruptor を使用できます。つまり、SEDA キューにメッセージを送信し、各コンシューマーがメッセージのコピーを受け取ることができます。有効にすると、すべてのコンシューマーエンドポイントでこのオプションを指定する必要があります。
limitConcurrentConsumers	true	concurrentConsumers 数を最大 500 に制限するかどうか。デフォルトでは、Disruptor エンドポイントがより大きな数値で設定されている場合、例外が発生します。このオプションをオフにすると、そのチェックを無効にできます。

blockWhenFull	true	メッセージを full Disruptor に送信するスレッドが、リングバッファーの容量が枯渇しなくなるまでブロックされるかどうか。デフォルトでは、呼び出し元のスレッドは、メッセージが受け入れられるまでブロックおよび待機します。このオプションを無効にすると、キューが満杯であることを示す例外が出力されます。
defaultBlockWhenFull		**Component only: blockWhenFull** が指定されていない場合に使用されるこのコンポントによって作成されたエンドポイントに対してリングバッファーが満杯になると、デフォルトのプロデューサー動作を設定できます。
waitStrategy	Blocking	新しいエクスチェンジが公開されるのを待つコンシューマースレッドによって使用されるストラテジーを定義します。許可されるオプションは、**Blocking**、**Sleeping**、**BusySpin**、および **Yielding** です。この件に関する詳細は、以下のセクションを参照してください。
defaultWaitStrategy		**Component only: waitStrategy** が指定されていない場合に使用されるこのコンポントによって作成されたエンドポイントにデフォルトの待機ストラテジーを設定できます。
producerType	multi	Disruptor で許可されるプロデューサーを定義します。許可されるオプションは、複数のプロデューサーと **Single** が、特定の最適化を有効にすることができることです(1 つのスレッドまたは他のスレッド上、または同期されている場合)。

待機ストラテジー

待機ストラテジーは、次のエクスチェンジの公開を待機しているコンシューマースレッドによって実行される待機の種類に影響します。以下のストラテジーを選択できます。

名前	説明	アドバイス

Blocking	バリアで待機している Consumers に lock および condition 変数を使用するブロッキングストラテジー。	このストラテジーは、CPU リソースほどスループットと低レイテンシーが重要ではない場合に使用できます。
スリープ状態	最初にスピンアップし、Thread.yield（）を使用し、最終的には OS と JVM の最小数で、コンシューマーがバリアで待機している間に許可されるスリープストラテジー。	このストラテジーは、パフォーマンスと CPU リソース間の妥協です。レイテンシーの急増は、quiet 期間の後に発生する可能性があります。
BusySpin	バリアで待機している Consumers にビジースピンループを使用する busy Spin ストラテジー。	このストラテジーは、CPU リソースを使用して、レイテンシージッターを引き起こす可能性のある syscall を回避します。スレッドを特定の CPU コアにバインドできる場合に最適です。
yielding	初回の回転後にバリアで待機した Consumers に Thread.yield（）を使用するストラテジーを生成します。	このストラテジーは、レイテンシーが大幅に急増することなく、パフォーマンスと CPU リソース間の妥協です。

リクエスト応答の使用

Disruptor コンポーネントは RequestReply の使用をサポートします。ここでは、呼び出し元は Async ルートが完了するまで待機します。以下に例を示します。

```
from("mina:tcp://0.0.0.0:9876?textline=true&sync=true").to("disruptor:input");
from("disruptor:input").to("bean:processInput").to("bean:createResponse");
```

上記のルートでは、受信リクエストを受け入れるポート 9876 の TCP リスナーがあります。リクエストは disruptor:input バッファーにルーティングされます。RequestReply メッセージであるため、応答を待ちます。disruptor:input バッファーのコンシューマーが完了すると、応答が元のメッセージの応答にコピーされます。

同時コンシューマー

デフォルトでは、Diruptor エンドポイントは単一のコンシューマースレッドを使用しますが、同時コンシューマースレッドを使用するように設定できます。そのため、スレッドプールの代わりに以下を使用できます。

```
from("disruptor:stageName?concurrentConsumers=5").process(...)
```

2 つの違いは、スレッドプールは負荷に応じて動的に拡大/縮小する可能性がある一方で、同時コンシューマーの数は常に内部的に固定され、サポートされるため、パフォーマンスは高くなります。

スレッドプール

以下のような手順を実行して、スレッドプールを Disruptor エンドポイントに追加することに注意してください。

```
from("disruptor:stageName").thread(5).process(...)
```

は、Diruptor を使用してパフォーマンスの一部を効果的に否定して、通常の BlockingQueue を追加して Disruptor と組み合わせて使用することで、優先できます。代わりに、concurrentConsumers オプションを使用して Disruptor エンドポイントでメッセージを処理するスレッド数を直接設定することが推奨されます。

例

以下のルートでは、Diruptor を使用してこの非同期キューにリクエストを送信し、別のスレッドでさらに処理するために fire-and-forget メッセージを送信でき、このスレッドの定数応答を元の呼び出し元に返します。

```
public void configure() throws Exception {
    from("direct:start")
        // send it to the disruptor that is async
        .to("disruptor:next")
        // return a constant response
        .transform(constant("OK"));

    from("disruptor:next").to("mock:result");
}
```

ここでは、Hello World メッセージを送信し、応答が OK であることが想定されます。

```
Object out = template.requestBody("direct:start", "Hello World");
assertEquals("OK", out);
```

Hello World メッセージは、さらに処理するために、別のスレッドから Disruptor から取得されます。これはユニットテストからのものであるため、ユニットテストでアサーションを実行できるモックエンドポイントに送信されます。

MULTIPLECONSUMERS の使用

この例では、2 つのコンシューマーを定義し、Spring Bean として登録しました。

```
<!-- define the consumers as spring beans -->
<bean id="consumer1" class="org.apache.camel.spring.example.FooEventConsumer"/>

<bean id="consumer2"
class="org.apache.camel.spring.example.AnotherFooEventConsumer"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- define a shared endpoint which the consumers can refer to instead of using url -->
    <endpoint id="foo" uri="disruptor:foo?multipleConsumers=true"/>
</camelContext>
```

Disruptor foo エンドポイントで multipleConsumers=true を指定しているため、これらの 2 つ以上のコンシューマーは、種類の pub-sub スタイルのメッセージングとしてメッセージの独自のコピーを受け取ることができます。Bean はユニットテストの一部であるため、単にモックエンドポイントにメッセー

ジを送信しますが、@Consume を使用して Disruptor から消費する方法に注意してください。

```
public class FooEventConsumer {

    @EndpointInject(uri = "mock:result")
    private ProducerTemplate destination;

    @Consume(ref = "foo")
    public void doSomething(String body) {
        destination.sendBody("foo" + body);
    }

}
```

中断情報の抽出

必要な場合は、以下のように JMX を使用せずにバッファーサイズなどの情報を取得できます。

```
DisruptorEndpoint disruptor = context.getEndpoint("disruptor:xxxx");
int size = disruptor.getBufferSize();
```

第37章 DNS

DNS

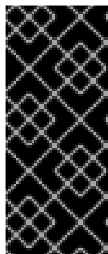
Camel 2.7 以降で利用可能

これは、DNSJava を使用して DNS クエリーを実行する Camel の追加コンポーネントです。コンポーネントは、DNSJava 上のシンレイヤーです。コンポーネントは以下の操作を提供します。

ip
: ドメインを IP アドレスで解決します。

lookup
: ドメインに関する情報を検索します。

dig
: DNS クエリーを実行します。



SUN JVM が必要です

DNSJava ライブラリーは SUN JVM で実行する必要があります。Apache ServiceMix または Apache Karaf を使用する場合は、**etc/jre.properties** ファイルを調整して、**sun.net.spi.nameservice** をエクスポートされた Java プラットフォームパッケージの一覧に追加する必要があります。この変更を反映するには、サーバーを再起動する必要があります。

Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-dns</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

DNS コンポーネントの URI スキームは以下のとおりです。

```
dns://operation
```

このコンポーネントはプロデューサーのみをサポートします。

オプション

なし。

HEADERS

ヘッダー	タイプ	操作	説明
dns.domain	**String**	**ip**	ドメイン名。必須。
dns.name	**String**	**lookup**	検索する名前。必須。
dns.type	-	**lookup**, **dig**	ルックアップのタイプ。**org.xbill.dns.Type** の値と一致する必要があります。(オプション)
dns.class	-	**lookup**, **dig**	ルックアップの DNS クラス。**org.xbill.dns.DClass** の値と一致する必要があります。(オプション)
dns.query	**String**	**dig**	クエリー自体。必須。
dns.server	**String**	**dig**	クエリーに関する特定のサーバー。指定がない場合は、OS で指定されたデフォルトのものが使用されます。(オプション)

例

IP ルックアップ

```
<route id="IPCheck">
   <from uri="direct:start"/>
   <to uri="dns:ip"/>
</route>
```

*これにより、ドメインの IP が検索されます。たとえば、www.example.com は 192.0.32.10 に解決されます。検索する IP アドレスは、キー **dns.domain** を含むヘッダーに指定する必要があります。*

DNS ルックアップ

```
<route id="IPCheck">
   <from uri="direct:start"/>
   <to uri="dns:lookup"/>
</route>
```

*これにより、ドメインに関連付けられた DNS レコードのセットが返されます。検索する名前は、ヘッダーに **dns.name** キーを指定する必要があります。*

DNS DIG

dig は、DNS クエリーを実行する Unix コマンドラインユーティリティーです。

```
<route id="IPCheck">
   <from uri="direct:start"/>
   <to uri="dns:dig"/>
</route>
```

クエリーは、キー **dns.query** を含むヘッダーに提供する必要があります。

第38章 DOCKER

DOCKER コンポーネント

Camel 2.15 以降で利用可能

Docker と通信するための Camel コンポーネント。

Docker Camel コンポーネントは、Docker Remote API を介して docker-java を利用します。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-docker</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
docker://[operation]?[options]
```

ここでの operation は、Docker で実行する固有のアクションです。

ヘッダーストラテジー

*すべての URI オプションは、Header プロパティーとして渡すことができます。メッセージヘッダーにある値は URI パラメーターよりも優先されます。ヘッダープロパティーは、以下に示すように *CamelDocker* で始まる URI オプションの形式を取ります。*

URI オプション	ヘッダープロパティー
containerId	CamelDockerContainerId

一般的なオプション

以下のパラメーターは、コンポーネントの呼び出しと合わせて使用できます。

オプション	ヘッダー	説明	デフォルト値
host	CamelDockerHost	**必須：** Docker ホスト	localhost
port	CamelDockerPort	**必須：**Docker ポート	2375
username	CamelDockerUserName	認証に使用するユーザー名	

password	CamelDockerPassword	認証するパスワード	
email	CamelDockerEmail	ユーザーに関連付けられたメールアドレス	
secure	CamelDockerSecure	HTTPS 通信の使用	false
requestTimeout	CamelDockerRequestTimeout	応答の要求タイムアウト（秒単位）	30
certPath	CamelDockerCertPath	SSL 証明書チェーンを含む場所	

コンシューマー操作

コンシューマーは以下の操作をサポートします。

操作	オプション	説明	生成されるアイテム
events	initialRange	Docker イベントの監視（ストリーム）	イベント

プロデューサー操作

以下のプロデューサー操作を使用できます。

その他操作	オプション	説明	戻り値
auth		認証設定の確認	
info		システム全体の情報	Info
ping		Docker サーバーに ping します。	
version		Docker バージョン情報を表示します。	バージョン

イメージ操作	オプション	説明	本文の内容	戻り値
image/list	filter, showAll	イメージを一覧表示します。		List<Image>
image/create	**repository**	イメージを作成する	InputStream	CreateImageResponse

image/build	noCache、quiet、remove、tag	stdin で Dockerfile からイメージを構築する	InputStream または File	InputStream
image/pull	**リポジトリー**、レジストリー、タグ	レジストリーからイメージをプルします。		InputStream
image/push	**name**	イメージをレジストリーにプッシュします。		InputStream
image/search	**用語**	イメージの検索		List<SearchItem>
image/remove	**imageId**	イメージの削除		
image/tag	**imageID、repository、tag、force**	イメージをリポジトリーにタグ付けします。		
image/inspect	**imageId**	イメージの検証		InspectImageResponse

コンテナー操作	オプション	説明	本文の内容	戻り値
container/list	showSize、showAll、before、since、limit、List コンテナー	initialRange		List<Container>
container/create	**imageID**、name、exposeedPorts、workDir、disableNetwork、hostname、user、tty、stdInOpen、stdInOnce、memoryLimit、memorySwap、cpuShares、attachStdIn、attachStdOut、attachStdErr、env、cmd、dns、image、volumesFrom	コンテナーを作成します。		CreateContainerResponse

container/start	**containerId**, bind, links, lxcConf, portBindings, privileged, publishAllPorts, dns, dnsSearch, volumesFrom, networkMode, devices, restartPolicy, capAdd, capDrop	コンテナーの起動		
container/inspect	**containerId**	コンテナーの検査		InspectContainerResponse
container/wait	**containerId**	コンテナーの待機	整数	
container/log	**containerId**, stdOut, stdErr, timestamps, followStream, tailAll, tail	コンテナーログの取得		InputStream
container/attach	**containerId**、stdOut、stdErr、タイムスタンプ、logs、followStream	コンテナーへの割り当て		InputStream
container/stop	**containerId**, timeout	コンテナーの停止		
container/restart	**containerId**, timeout	コンテナーの再起動		
container/diff	**containerId**	コンテナーの変更の検証		ChangeLog
container/kill	**containerId**、signal	コンテナーの強制終了		
container/top	**containerId**, psArgs	コンテナーで実行しているプロセスを一覧表示します。		TopContainerResponse
container/pause	**containerId**	コンテナーの一時停止		
container/unpause	**containerId**	コンテナーの一時停止解除		

container/commit	**containerId**, repository, message, tag, attachStdIn, attachStdOut, attachStdErr, cmd, disableNetwork, pause, env, exposedPorts, hostname, memory, memorySwap, openStdIn, portSpecs, stdInOnce, tty, user, volumes, hostname	コンテナーの変更から新規イメージを作成します。	文字列	
container/copyfile	**containerId**, **resource**, hostPath	コンテナーからファイルまたはフォルダーをコピーする	InputStream	
container/remove	**containerId**、force、removeVolumes	コンテナーの削除		

例

以下の例では、Docker からのイベントを使用します。

```
from("docker://events?host=192.168.59.103&port=2375").to("log:event");
```

以下の例では、Docker に対してシステム全体の情報をクエリーします。

```
from("docker://info?host=192.168.59.103&port=2375").to("log:info");
```

第39章 DOZER

DOZER コンポーネント

dozer: コンポーネントは、Dozer マッピングフレームワークを使用して Java Bean 間のマッピング機能を提供します。Camel は、Dzer マッピングを 型コンバーター として トリガーする機能もサポートします。Dozer エンドポイントと Dozer コンバーターの使用の主な相違点は次のとおりです。

- コンバーターレジストリーを介して、エンドポイントごとに Dozer マッピング設定とグローバル設定を管理する機能。
- Dozer エンドポイントは、Camel データ形式を使用して入出力データをマーシャリングまたはアンマーシャリングし、単一の任意の変換エンドポイントをサポートするように設定できます。
- Dozer コンポーネントでは、Dozer のより詳細な統合と拡張を追加機能（たとえば、リテラル値のマッピング、マッピングの式の使用など）することができます。

Dozer コンポーネントを使用するには、Maven ユーザーは以下の依存関係を **pom.xml** に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-dozer</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

Dozer コンポーネントはプロデューサーエンドポイントのみをサポートします。

```
dozer:endpointId[?options]
```

ここで、endpointId は、Dozer エンドポイント設定を一意に識別するために使用される名前です。

例：Dozer エンドポイント URI:

```
from("direct:orderInput").
  to("dozer:transformOrder?
mappingFile=orderMapping.xml&targetModel=example.XYZOrder").
  to("direct:orderOutput");
```

オプション

名前	デフォルト	説明
mappingFile	dozerBeanMapping.xml	Dozer 設定ファイルの場所。デフォルトでは、ファイルはクラスパスから読み込まれますが、**file:**、**classpath:**、または **http:** を使用して、特定の場所から設定を読み込むことができます。
unmarshalId	none	Java 以外のタイプからマッピング入力をアンマーシャリングするために使用する Camel Context 内で定義された dataFormat の ID。
marshalId	none	マッピング出力を Java 以外のタイプにマーシャリングするために使用する Camel Context 内で定義された dataFormat の ID。
sourceModel	none	マッピングで使用されるソースタイプの完全修飾クラス名。これが指定されている場合、マッピングへの入力は、Dozer でマッピングされる前に指定された型に変換されます。
targetModel	none	マッピングで使用されるターゲットタイプの完全修飾クラス名。このオプションは必須です。
mappingConfiguration	none	Dozer マッピングの設定に使用する Camel レジストリーの DozerBeanMapperConfiguration Bean の名前。これは、Dzer の設定方法を詳細に制御するために使用可能な mappingFile オプションの代替手段です。値に "#" 接頭辞を使用して、Bean が Camel レジストリーにあることを示します（例：#myDozerConfig）。

DOZER でのデータフォーマットの使用

Dozer はマッピングの Java 以外のソースおよびターゲットをサポートしないため、XML ドキュメントを独自の Java オブジェクトにマッピングすることはできません。Camel は Java と データ 形式を使用したさまざまなフォーマットのマーシャリングを広範囲にサポートしています。Dozer コンポーネントは、Dozer を介して処理される前に入出力データをデータ形式で渡すように指定できるため、このサポートを利用します。Dozer への呼び出し外で常にこれを実行できますが、Dozer コンポーネントで直接サポートすることで、単一のエンドポイントを使用して Camel 内で任意の変換を設定できます。

たとえば、Dozer コンポーネントを使用して XML データ構造と JSON データ構造をマッピングしたとします。Camel コンテキストで以下のデータ形式が定義されている場合：

```
<dataFormats>
  <json library="Jackson" id="myjson"/>
  <jaxb contextPath="org.example" id="myjaxb"/>
</dataFormats>
```

その後、Jackson データ形式を使用して入力 XML をアンマーシャリングし、Jackson を使用してマッピング出力をマーシャリングするように Dozer エンドポイントを設定できます。

```
<endpoint uri="dozer:xml2json?
marshalId=myjson&amp;unmarshalId=myjaxb&amp;targetModel=org.example.Order"/>
```

DOZER の設定

すべての Dozer エンドポイントには、ソースおよびターゲットオブジェクト間のマッピングを定義する Dozer マッピング設定ファイルが必要です。mappingFile または mappingConfiguration オプションがエンドポイントで指定されていない場合、コンポーネントは META-INF/dozerBeanMapping.xml の場所にデフォルト設定されます。1 つのエンドポイントに複数のマッピング設定ファイルを提供する必要がある場合、または追加の設定オプション（イベントリスナー、カスタムコンバーターなど）を指定する必要がある場合は、**org.apache.camel.converter.dozer.DozerBeanMapperConfiguration** のインスタンスを使用できます。

```
<bean id="mapper"
class="org.apache.camel.converter.dozer.DozerBeanMapperConfiguration">
  <property name="mappingFiles">
    <list>
      <value>mapping1.xml</value>
      <value>mapping2.xml</value>
    </list>
  </property>
</bean>
```

マッピング拡張機能

Dozer コンポーネントは、Dozer マッピングフレームワークにカスタムコンバーターとして多くの拡張機能を実装します。これらのコンバーターは、Dzer 自体で直接サポートされていないマッピング関数を実装します。

変数マッピング

変数のマッピングにより、Dozer 設定内の変数定義の値をソースフィールドの値を使用する代わりに、ターゲットフィールドにマップできます。これは、他のマッピングフレームワークの定数マッピングと同等で、はリテラル値をターゲットフィールドに割り当てることができます。変数マッピングを使用するには、マッピング設定で変数を定義し、VariableMapper クラスから選択したターゲットフィールドにマップします。

```
<mappings xmlns="http://dozer.sourceforge.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://dozer.sourceforge.net
http://dozer.sourceforge.net/schema/beanmapping.xsd">
  <configuration>
```

```
    <variables>
      <variable name="CUST_ID">ACME-SALES</variable>
    </variables>
  </configuration>
  <mapping>
    <class-a>org.apache.camel.component.dozer.VariableMapper</class-a>
    <class-b>org.example.Order</class-b>
    <field custom-converter-id="_variableMapping" custom-converter-param="${CUST_ID}">
      <a>literal</a>
      <b>custId</b>
    </field>
  </mapping>
</mappings>
```

カスタムマッピング

カスタムマッピングにより、ソースフィールドがターゲットフィールドにマップされる方法についての独自のロジックを定義できます。Dozer のカスタマーコンバーターと機能的に似ており、以下の 2 つの相違点があります。

- カスタムマッピングを持つ単一のクラスに複数のコンバーターメソッドを設定できます。
- カスタムマッピングを使用して Dozer 固有のインターフェイスを実装する必要はありません。

カスタムマッピングは、マッピング設定で組み込みの '_customMapping' コンバーターを使用して宣言されます。このコンバーターの パラメーターの構文は以下のとおりです。

```
[class-name][,method-name]
```

メソッド名はオプションです。Dozer コンポーネントは、マッピングに必要な入出力タイプに一致するメソッドを検索します。カスタムマッピングと設定の例を以下に示します。

```
public class CustomMapper {
    // All customer ids must be wrapped in "[ ]"
    public Object mapCustomer(String customerId) {
        return "[" + customerId + "]";
    }
}
```

```
<mappings xmlns="http://dozer.sourceforge.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://dozer.sourceforge.net
http://dozer.sourceforge.net/schema/beanmapping.xsd">
  <mapping>
    <class-a>org.example.A</class-a>
    <class-b>org.example.B</class-b>
    <field custom-converter-id="_customMapping"
     custom-converter-param="org.example.CustomMapper,mapCustomer">
      <a>header.customerNum</a>
      <b>custId</b>
    </field>
  </mapping>
</mappings>
```

式マッピング

式マッピングを使用すると、Camel の強力な 言語 機能を使用して式を評価し、結果をマッピングのターゲットフィールドに割り当てることができます。Camel がサポートする言語は、式マッピングで使用できます。式の基本的な例には、Camel メッセージヘッダーまたはエクスチェンジプロパティーをターゲットフィールドにマップしたり、複数のソースフィールドをターゲットフィールドに連結したりする機能が含まれます。マッピング式の構文は次のとおりです。

```
[language]:[expression]
```

メッセージヘッダーをターゲットフィールドにマップする例：

```
<mappings xmlns="http://dozer.sourceforge.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://dozer.sourceforge.net
http://dozer.sourceforge.net/schema/beanmapping.xsd">
  <mapping>
    <class-a>org.apache.camel.component.dozer.ExpressionMapper</class-a>
    <class-b>org.example.B</class-b>
    <field custom-converter-id="_expressionMapping" custom-converter-
param="simple:\${header.customerNumber}">
      <a>expression</a>
      <b>custId</b>
    </field>
  </mapping>
</mappings>
```

Dozer が EL を使用して定義された変数値を解決しようとするとエラーを防ぐために、式内のプロパティーを "\" でエスケープする必要があります。

DOZER 型変換

Dozer コンポーネントは、Dozer Mapping フレームワークを使用して Java Bean 間のマッピング機能を提供します。ただし、Camel は Dozer マッピングを型コンバーターとしてトリガーする機能もサポートします。

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").convertBodyTo(CustomerB.class);
    }
});

DozerBeanMapperConfiguration mconfig = new DozerBeanMapperConfiguration();
mconfig.setMappingFiles(Arrays.asList(new String[] { "mappings.xml" }));
new DozerTypeConverterLoader(camelctx, mconfig);
```

第40章 DROPBOX

CAMEL DROPBOX コンポーネント

Camel 2.14 から利用可能

dropbox: コンポーネントを使用すると、Dropbox リモートフォルダーをメッセージのプロデューサーまたはコンシューマーとして扱うことができます。Dropbox Java Core API（このコンポーネントの参照バージョンは 1.7.x）を使用する場合、この Camel コンポーネントには以下の機能があります。

- コンシューマーとして、ファイルをダウンロードし、クエリーでファイルを検索します。
- プロデューサーとしてファイルのダウンロード、リモートディレクトリー間でのファイルの移動、ファイル/ディレクトリーの削除、ファイルのアップロード、クエリーによるファイルの検索

Maven ユーザーは、このコンポーネントの以下の依存関係を **pom.xml** に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-dropbox</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
dropbox://[operation]?[options]
```

ここでの operation は、Dropbox リモートフォルダーで実行する特定のアクション（通常は CRUD アクション）です。

操作

操作	説明
del	Dropbox のファイルまたはディレクトリーを削除します。
get	Dropbox からファイルをダウンロードする
move	Dropbox のフォルダーからファイルの移動
put	Dropbox でのファイルのアップロード
search	文字列クエリーに基づいて Dropbox でファイルを検索する

操作 には追加のオプションが必要です。特定の操作には一部は必須です。

オプション

Dropbox API と連携するには、accessToken およびclientIdentifier を取得する必要があります。取得する方法は、Dropbox のドキュメント を参照してください。

以下は、すべての操作に必要なオプションの一覧です。

プロパティー	Mandatory	説明
accessToken	**true**	特定の Dropbox ユーザーの API リクエストを行うためのアクセストークン
clientIdentifier	**true**	API リクエストを行うために登録されたアプリケーションの名前

DEL 操作

Dropbox のファイルを削除します。

Camel プロデューサーとしてのみ動作します。

以下は、この操作のオプションの一覧です。

プロパティー	Mandatory	説明
remotePath	**true**	Dropbox で削除するフォルダーまたはファイル

サンプル

from ("direct:start").to ("dropbox://del?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1").to ("mock:result");

from ("direct:start").to ("dropbox://del?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1/file1.tar.gz").to ("mock:result");

結果メッセージヘッダー

メッセージ結果に以下のヘッダーが設定されます。

プロパティー	値
DELETED_PATH	ドロップボックスで削除されたパスの名前

結果メッセージのボディー

以下のオブジェクトはメッセージボディーの結果に設定されます。

オブジェクトタイプ	説明
String	ドロップボックスで削除されたパスの名前

GET (DOWNLOAD)操作

Dropbox からファイルをダウンロードします。

Camel プロデューサーまたは Camel コンシューマーとして機能します。

以下は、この操作のオプションの一覧です。

プロパティー	Mandatory	説明
remotePath	**true**	Dropbox からダウンロードするフォルダーまたはファイル

サンプル

from ("direct:start").to ("dropbox://get?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1/file1.tar.gz").to ("file:///home/kermit/?fileName=file1.tar.gz");

from ("direct:start").to ("dropbox://get?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1").to ("mock:result");

from ("dropbox://get?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1").to ("file:///home/kermit/");

結果メッセージヘッダー

メッセージ結果に以下のヘッダーが設定されます。

プロパティー	値
DOWNLOADED_FILE	1 つのファイルのダウンロードの場合、ダウンロードしたリモートファイルのパス
DOWNLOADED_FILES	ファイルが複数ダウンロードされている場合は、ダウンロードしたリモートファイルのパス

結果メッセージのボディー

以下のオブジェクトはメッセージボディーの結果に設定されます。

オブジェクトタイプ	説明
ByteArrayOutputStream	単一ファイルのダウンロードの場合、ダウンロードしたファイルを表すストリーム。
Map<String, ByteArrayOutputStream>	複数のファイルをダウンロードすると、がキーとして指定されたリモートファイルのパスと、ダウンロードしたファイルを表すストリームが値として、マップがダウンロードされます。

移動操作

Dropbox のファイルをあるフォルダーから別のフォルダーに移動します。

Camel プロデューサーとしてのみ動作します。

以下は、この操作のオプションの一覧です。

プロパティー	Mandatory	説明
remotePath	**true**	移動する元のファイルまたはディレクトリー
newRemotePath	**true**	宛先ファイルまたはフォルダー

サンプル

from ("direct:start").to ("dropbox://move?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1&newRemotePath=/root/folder2")("mock:result");

結果メッセージヘッダー

メッセージ結果に以下のヘッダーが設定されます。

プロパティー	値
MOVED_PATH	ドロップボックスで移動したパスの名前

結果メッセージのボディー

以下のオブジェクトはメッセージボディーの結果に設定されます。

オブジェクトタイプ	説明

String	ドロップボックスで移動したパスの名前

PUT (UPLOAD)操作

Dropbox でファイルをアップロードします。

Camel プロデューサーとして機能します。

以下は、この操作のオプションの一覧です。

プロパティー	Mandatory	説明
uploadMode	**true**	このオプションを追加または強制すると、ドロップボックスにファイルを保存する方法を指定します。add の場合、同じ名前のファイルがすでにドロップボックスに存在する場合、新しいファイルの名前が変更されます。同じ名前のファイルがすでに存在する場合は、これは上書きされます。
localPath	**true**	ローカルファイルシステム から Dropbox にアップロードするフォルダーまたはファイル。
remotePath	**false**	Dropbox のフォルダー宛先。プロパティーが設定されていない場合、コンポーネントはローカルパスと同等のリモートパス上のファイルをアップロードします。

サンプル

from ("direct:start").to ("dropbox://put?accessToken=XXX&clientIdentifier=XXX&uploadMode=add&localPath=/root/folder1").to ("mock:result");

from ("direct:start").to ("dropbox://put?accessToken=XXX&clientIdentifier=XXX&uploadMode=add&localPath=/root/folder1&remotePath=/roo ("mock:result");

結果メッセージヘッダー

メッセージ結果に以下のヘッダーが設定されます。

プロパティー	値
UPLOADED_FILE	単一ファイルのアップロードの場合は、アップロードされたリモートパスのパス。

UPLOADED_FILES	複数のファイルのアップロードの場合は、リモートパスと共に文字列がアップロードされます。

結果メッセージのボディー

以下のオブジェクトはメッセージボディーの結果に設定されます。

オブジェクトタイプ	説明
String	単一ファイルのアップロードの場合は、アップロード操作の結果、OK、または KO
Map<String, DropboxResultCode>	複数のファイルのアップロードの場合、がリモートファイルのパスとしてアップロードされるマップ。アップロード操作の結果、OK または KO の値

検索操作

サブディレクトリーを含むリモート Dropbox フォルダー内で検索します。

Camel プロデューサーおよび Camel コンシューマーとして機能します。

以下は、この操作のオプションの一覧です。

プロパティー	Mandatory	説明
remotePath	**true**	検索先の Dropbox のフォルダー。
query	**false**	検索するサブ文字列のスペース区切りリスト。ファイルは、すべてのサブ文字列が含まれる場合にのみ一致します。このオプションが設定されていない場合、すべてのファイルが一致します。

サンプル

from ("dropbox://search?accessToken=XXX&clientIdentifier=XXX&remotePath=/XXX&query=XXX").to ("mock:result");

from ("direct:start").to ("dropbox://search?accessToken=XXX&clientIdentifier=XXX&remotePath=/XXX").to ("mock:result");

結果メッセージヘッダー

メッセージ結果に以下のヘッダーが設定されます。

プロパティー	値
FOUNDED_FILES	検出されたファイルパスの一覧

結果メッセージのボディー

以下のオブジェクトはメッセージボディーの結果に設定されます。

オブジェクトタイプ	説明
List<DbxEntry>	検出されたファイルパスの一覧。このオブジェクトの詳細は、Dropbox のドキュメント http://dropbox.github.io/dropbox-sdk-java/api-docs/v1.7.x/com/dropbox/core/DbxEntry.htmlを参照してください。

第41章 EJB

EJB COMPONENT

EJB コンポーネントは Java Enterprise Java Beans を Camel メッセージエクスチェンジにバインドします。以下に例を示します。

```
CamelContext camelctx = new DefaultCamelContext();
    camelctx.addRoutes(new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:start").to("ejb:java:module/HelloBean");
        }
    });
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

第42章 ETCD

ETCD コンポーネント

etcd は、マシンのクラスター全体でデータを保存する信頼できる方法を提供します。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-etcd</artifactId>
   <!-- use the same version as your Camel core version -->
   <version>x.y.z</version>
</dependency>
```

URI 形式

```
etcd:namespace[/path][?options]
```

namespace は etcd-component が操作する必要のある etcd コンテキストを表し、path は影響を受けるノードを定義するためのオプションの属性です。

サポートされている名前空間は次のとおりです。

- keys
- watch
- stats

オプション

名前	デフォルト値	説明
uris	http://localhost:2379,http://localhost:4001	コンポーネントが接続する必要のある URI を定義します。
sslContextParameters	**null**	カスタム **org.apache.camel.util.jsse.SSLContextParameters** を使用するには、を使用します。JSSE 設定ユーティリティーの使用 を参照してください。
userName	**null**	Basic 認証に使用するユーザー名
password	**null**	Basic 認証に使用するパスワード

sendEmptyExchangeOnTimeout	**false**	キーを監視するタイムアウトが発生した場合に空のメッセージを送信する（コンシューマーのみ）
recursive	**false**	アクションを再帰的に適用する
timeToLive	**null**	キーのライフサイクルをミリ秒単位で設定します。
timeout	**null**	アクションが完了するまでにかかる最大時間を設定します。

HEADERS

名前	タイプ	説明
CamelEtcdAction	**java.lang.String**	実行するアクション。サポートされる値は set、delete、deleteDir、get です。
CamelEtcdNamespace	**java.lang.String**	エクスチェンジが生成/処理された etcd コンテキスト
CamelEtcdPath	**java.lang.String**	キー namespace の場合、これは URI エンドポイントからのパスが使用される場合に、アクションの対象となるノードを判別するために使用されます。統計および監視名前空間の場合、処理されたノード beign のパスを含む
CamelEtcdTimeout	**java.lang.Long**	アクションが完了するまでにかかる最大時間を設定します。存在しない場合は、**timeout** オプションが考慮されます。
CamelEtcdTtl	**java.lang.Integer**	キーのライフサイクルをミリ秒単位で設定します。存在しない場合は、**timeToLive** オプションが考慮されます。
CamelEtcdRecursive	**java.lang.Boolean**	アクションを再帰的に適用するには、以下を行います。存在しない場合は、**recursive** オプションが考慮されます。

キー NAMESPACE の例：

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new RouteBuilder() {
```

```
    public void configure() {
       from("direct:keys-set")
            .to("etcd:keys")
                .to("log:camel-etcd?level=INFO");
    }
});

Map<String, Object> headers = new HashMap<>();
headers.put(EtcdConstants.ETCD_ACTION, EtcdConstants.ETCD_KEYS_ACTION_SET);
headers.put(EtcdConstants.ETCD_PATH, "/camel/etcd/myKey");

ProducerTemplate template = context.createProducerTemplate();
template.sendBodyAndHeaders("direct:keys-set", "camel-etcd", headers);
```

統計 NAMESPACE の例：

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new RouteBuilder() {
    public void configure() {
       from("etcd:stats/leader?consumer.delay=50&consumer.initialDelay=0")
           .to("log:etcd-leader-stats?level=INFO");
       from("etcd:stats/self?consumer.delay=50&consumer.initialDelay=0")
           .to("log:etcd-self-stats?level=INFO");
       from("etcd:stats/store?consumer.delay=50&consumer.initialDelay=0")
           .to("log:etcd-store-stats?level=INFO");
    }
});
```

NAMESPACE の例を監視します。

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new RouteBuilder() {
    public void configure() {
       from("etcd:watch/recursive?recursive=true")
        .marshall().json()
        .to("log:etcd-event?level=INFO")
    }
});
```

第43章 ELASTICSEARCH

ELASTICSEARCH コンポーネント

Camel 2.11 から利用可能

ElasticSearch コンポーネントを使用すると、ElasticSearch サーバーとのインターフェイスが可能になります。

*Maven ユーザーは、このコンポーネントの **pom.xml** に以下の依存関係を追加する必要があります。*

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-elasticsearch</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
elasticsearch://clusterName?[options]
```

ヒント

*ローカル(JVM/classloader) ElasticSearch サーバーに対して実行する場合は、URI の clusterName の値を **local** に設定します。詳細は、クライアントガイド を参照してください。*

エンドポイントオプション

以下のオプションは ElasticSearch エンドポイントで設定できます。すべては、エンドポイント URI パラメーターまたはヘッダー（ヘッダーオーバーライドエンドポイントプロパティー）として設定する必要があります。

name	description
operation	必須。実行する操作を示します。
indexName	動作させるインデックスの名前。
indexType	動作させるインデックスのタイプ。
ip	Camel 2.12 を使用する TransportClient リモートホスト IP

port	使用する TransportClient リモートポート（デフォルトは
transportAddresses	**Camel 2.16:** 使用するリモートトランスポートアドレス ***ip*** りに考慮するには、オプション ***ip*** および ***port*** を空白のま
consistencyLevel	**Camel 2.16:** INDEX および BULK 操作で使用する書き込み れかです)。
replicationType	**Camel 2.16:** INDEX および BULK 操作で使用するレプリケ です)。Elasticsearch 2.0.0 から非同期レプリケーションが は削除されました。
parent	**Camel 2.16.1 / 2.17.0:** 親レコードの ID を指定するために ションです。
clientTransportSniff	**Camel 2.17:** クライアントが残りのクラスターをスニッフ

メッセージ操作

以下の ElasticSearch 操作は現在サポートされています。operation のキーでエンドポイント URI オプションまたはエクスチェンジヘッダーを設定し、値は以下のいずれかに設定されるだけです。一部の操作では、他のパラメーターやメッセージボディーも設定する必要があります。

operation	メッセージボディー	description
INDEX	**Map**、**String**、または **byte[]** コンテンツでインデックス化するコンテンツ **XContentBuilder**	インデックスにコンテンツを追加し、本文でコンテンツの **indexId** を返します。
GET_BY_ID	取得するコンテンツのインデックス ID	指定されたインデックスを取得し、ボディーで **GetResult** オブジェクトを返します。
DELETE	削除するコンテンツのインデックス ID	指定された indexId を削除し、ボディーで **DeleteResult** オブジェクトを返します。
BULK_INDEX	すでに受け入れられたタイプの **リスト** または **コレクション** (**XContentBuilder**、**Map**、**byte[]**、または **String**)	**Camel 2.14 は、** コンテンツをインデックスに追加し、ボディーで正常にインデックス化されたドキュメントの ID の List を返します。
BULK	すでに受け入れられたタイプの **リスト** または **コレクション** (**XContentBuilder**、**Map**、**byte[]**、または **String**)	**Camel 2.15 では、** コンテンツをインデックスに追加し、ボディーで BulkResponse オブジェクトを返します。
SEARCH	**Map** または、 **SearchRequest**	**Camel 2.15:** クエリー文字列のマップでコンテンツを検索します。

MULTIGET	一覧 **MultigetRequest.Item**	Camel 2.17: **MultigetRequest** で指定したインデックスやタイプなどを取得し、ボディーで **MultigetResponse** オブジェクトを返します。
MULTISEARCH	一覧 **SearchRequest**	Camel 2.17: **MultiSearchRequest** に指定されたパラメーターを検索し、ボディーで **MultiSearchResponse** オブジェクトを返します。
EXISTS	ヘッダーとしてのインデックス名	Camel 2.17: ボディー内のブール値オブジェクトを返します。
UPDATE	**Map**更新するコンテンツ、**String**、または **byte[]** **XContentBuilder**	Camel 2.17: コンテンツをインデックスに更新し、ボディーでコンテンツの **indexId** を返します。

インデックスの例

以下は簡単な INDEX の例になります。

```
from("direct:index")
   .to("elasticsearch://local?operation=INDEX&indexName=twitter&indexType=tweet");
```

```
<route>
   <from uri="direct:index" />
   <to uri="elasticsearch://local?operation=INDEX&indexName=twitter&indexType=tweet"/>
</route>
```

クライアントは単に、Map が含まれるボディーメッセージをルートに渡す必要があります。結果ボディーには、作成された indexId が含まれます。

```
Map<String, String> map = new HashMap<String, String>();
map.put("content", "test");
String indexId = template.requestBody("direct:index", map, String.class);
```

JAVA の例

以下の例は、Java で定義された Camel ルートから ElasticSearch コンポーネントを使用する方法を示しています。

```
CamelContext camelContext = new DefaultCamelContext();
camelContext.addRoutes(new RouteBuilder() {
   @Override
   public void configure() throws Exception {
      from("direct:index")
        .to("elasticsearch://local?operation=INDEX&indexName=twitter&indexType=tweet");
   }
});
```

```
Map<String, String> indexedData = new HashMap<>();
indexedData.put("content", "test");

ProducerTemplate template = camelContext.createProducerTemplate();
template.sendBody("direct:index", indexedData);
```

詳細は、これらのリソース を参照してください。

ElasticSearch Main Site

ElasticSearch Java API

第44章 ELSQL

ELSQL コンポーネント

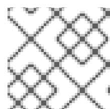
Camel 2.16 以降で利用可能

elsql: コンポーネントは、ElSql を使用して SQL クエリーを定義する既存の SQL コンポーネントへの拡張です。

このコンポーネントは、実際の SQL 処理のために **spring-jdbc** を背後で使用します。

Maven ユーザーは、このコンポーネントの以下の依存関係を **pom.xml** に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-elsql</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```



注記

このコンポーネントは、Transactional Client として使用できます。

ElSql コンポーネントは、以下のエンドポイント URI 表記を使用します。

```
elsql:elSqlName:resourceUri[?options]
```

以下の形式で URI にクエリーオプションを追加できます。 **?option=value&option=value&...**

SQL クエリーへのパラメーターは、elsql マッピングファイルのパラメーターに名前を付け、指定の優先順位で Camel メッセージから対応するキーにマップします。

1.Camel 2.16.1: Simple 式の場合はメッセージボディーから。

2. メッセージボディーから **java.util.Map**

3. メッセージヘッダーから

名前付きパラメーターを解決できない場合は、例外が発生します。

オプション

オプション	型	デフォルト	説明

resourceUri	文字列	null	**必須**: 使用する elsql SQL ステートメントが含まれるリソースファイル。複数のリソースはコンマで区切って指定できます。リソースはデフォルトでクラスパスにロードされ、file: の接頭辞を指定してファイルシステムからロードすることができます。このオプションはコンポーネントで設定してから、エンドポイントで設定する必要がないことに注意してください。
elSqlConfig		null	特定の設定された ElSqlConfig を使用します。代わりに databaseVendor オプションを使用することが推奨されます。
databaseVendor		デフォルト	ベンダー固有の ElSqlConfig を使用するには、以下を実行します。使用できる値は、デフォルト、Postgres、HSql、MySql、Oracle、SqlServer 2008、Veritca です。
batch	**boolean**	**false**	SQL バッチ更新ステートメントを実行します。**true** に設定されている場合は、インバウンドメッセージのボディーがどのように変化するかについての以下の注記を参照してください。
dataSource	**String**	**null**	レジストリーで検索するための **DataSource** への参照。

template.<xxx>		**null**	クエリーを実行するために背後で使用される Spring NamedParameterJdbcTemplate に追加のオプションを設定します。たとえば、**template.maxRows=10** です。詳細なドキュメントは、NamedParameterJdbcTemplate javadoc のドキュメント を参照してください。
consumer.delay	**long**	**500**	各ポーリングの遅延（ミリ秒単位）。
consumer.initialDelay	**long**	**1000**	ポーリングが開始するまでの時間（ミリ秒単位）。
consumer.useFixedDelay	**boolean**	**false**	ポーリング間の固定遅延を使用するには、**true** に設定します。それ以外の場合は、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。
maxMessagesPerPoll	**int**	**0**	ポーリングごとに収集するメッセージの最大数を定義する整数値。デフォルトでは最大値は設定されていません。
consumer.useIterator	**boolean**	**true**	**true** ポーリングが個別に処理されるときに返された各行の場合。**false** の場合、データ **java.util.List** 全体が IN ボディーとして設定されます。
consumer.routeEmptyResultSet	**boolean**	**false**	ポーリングするデータがない場合に、単一の空のエクスチェンジをルーティングするかどうか。

consumer.onConsume	**String**	**null**	各行を処理した後、このクエリーを実行できます。エクスチェンジが正常に処理された場合、たとえば、行を processed とマークします。クエリーには パラメーターを指定できます。
consumer.onConsumeFailed	**String**	**null**	各行を処理した後、このクエリーを実行できます。エクスチェンジが失敗した場合、たとえば行が失敗したとマークされます。クエリーには パラメーターを指定できます。
consumer.onConsumeBatchComplete	**String**	**null**	バッチ全体の処理後、このクエリーを実行して行を一括更新することができます。クエリーにはパラメーターを指定できません。
consumer.breakBatchOnConsumeFail	**boolean**	**false**	**consumer.onConsume** を使用して失敗すると、このオプションはバッチを分割するか、バッチからの次の行の処理を継続するかどうかを制御します。

outputType	**String**	**SelectList**	コンシューマーまたはプロデューサーの出力を Map of Map として **SelectList** にするか、以下のように単一の Java オブジェクトとして **SelectOne** を作成します。1）クエリーに列が 1 つしかない場合は、その JDBC Column オブジェクトが返されます。(**SELECT COUNT(*) FROM PROJECT** など) Long オブジェクトが返されます。クエリーに複数の列がある場合は、その結果の Map を返します。c） **outputClass** が設定されている場合は、その結果の Map を返します。次に、列名に一致するすべてのセッターを呼び出して、クエリー結果を Java Bean オブジェクトに変換します。クラスにインスタンスを作成するデフォルトのコンストラクターがあるとします。d）クエリーによって複数の行が発生した場合は、一意でない結果例外が出力されます。 また、SelectList は SelectOne として各行を Java オブジェクトにマッピングしています（ステップ c のみ）。
outputClass	**String**	**null**	**outputType=SelectOne** の場合に変換として使用する完全なパッケージおよびクラス名を指定します。
outputHeader	**String**	**null**	メッセージボディーではなくヘッダーとして結果を保存するには、以下を行います。これにより、既存のメッセージボディーをそのまま保存できます。
noop	**boolean**	**false**	これが設定されている場合、は SQL クエリーの結果を無視し、処理を継続するために OUT メッセージとして既存の IN メッセージを使用します。

transacted	boolean	false	Camel 2.16.2: SQL コンシューマーのみ：トランザクションを有効または無効にします。有効にすると、エクスチェンジの処理が失敗した場合、コンシューマーは追加のエクスチェンジを処理しなくなり、ロールバックの Eager が発生します。

クエリーの結果

select 操作の場合、結果は JdbcTemplate.queryForList （）メソッドによって返される List<Map<String, Object>> タイプのインスタンスになります。update 操作では、結果は更新行数で、Integer として返されます。

デフォルトでは、結果はメッセージのボディーに配置されます。outputHeader パラメーターが設定されている場合、結果はヘッダーに配置されます。これは、完全な Message Enrichment パターンを使用してヘッダーを追加する代わりに、シーケンスやその他の小さい値をヘッダーにクエリーするための簡潔な構文を提供します。outputHeader と outputType を一緒に使用すると便利です。

ヘッダーの値

update 操作の実行時に、SQL コンポーネントは以下のメッセージヘッダーに更新数を保存します。

ヘッダー	説明
CamelSqlUpdateCount	**Integer** オブジェクトとして返される **update** 操作で更新される行数。
CamelSqlRowCount	**Integer** オブジェクトとして返される **select** 操作に対して返される行数。

例

以下のルートでは、Projects テーブルからすべてのプロジェクトを取得します。SQL クエリーには、：#lic と :#min の 2 つの名前付きパラメーターがあることに注意してください。

その後、Camel はメッセージボディーまたはメッセージヘッダーからこれらのパラメーターを検索します。上記の例では、名前付きパラメーターに定数値で 2 つのヘッダーを設定することに注意してください。

```
from("direct:projects")
  .setHeader("lic", constant("ASF"))
  .setHeader("min", constant(123))
  .to("elsql:projects:com/foo/orders.elsql")
```

および elsql マッピングファイル

```
@NAME(projects)
  SELECT *
  FROM projects
  WHERE license = :lic AND id > :min
  ORDER BY id
```

メッセージボディーが **java.util.Map** の場合、名前付きパラメーターはボディーから取得されます。

```
from("direct:projects")
  .to("elsql:projects:com/foo/orders.elsql")
```

Camel 2.16.1 以降では、Simple 式も使用できます。これにより、メッセージボディーの表記のような OGNL を使用できます。ここでは、**getLicense** および **getMinimum** メソッドがあることを前提としています。

```
@NAME(projects)
  SELECT *
  FROM projects
  WHERE license = :${body.license} AND id > :${body.minimum}
  ORDER BY id
```

その他の参考資料

- SQL Component
- MyBatis
- JDBC

第45章 EVENTADMIN

EVENTADMIN コンポーネント

Camel 2.6 で利用可能

eventadmin コンポーネントは OSGi 環境で使用することで、OSGi EventAdmin イベントを受信して処理できます。

DEPENDENCIES

Maven ユーザーは以下の依存関係を **pom.xml** に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-eventadmin</artifactId>
  <version>${camel-version}</version>
</dependency>
```

${camel-version} は、実際のバージョンの Camel (2.6.0 以降)に置き換える必要があります。

URI 形式

```
eventadmin:topic[?options]
```

topic はリッスンするトピックの名前です。

URI オプション

名前	デフォルト値	説明
send	**false**	send または synchronous 配信を使用するかどうか。デフォルト false （非同期配信）

メッセージヘッダー

名前	タイプ	メッセージ	説明

メッセージボディー

in メッセージボディーは受信した Event に設定されます。

使用例

```
<route>
   <from uri="eventadmin:*"/>
```

```
    <to uri="stream:out"/>
</route>
```

第46章 EXEC

EXEC コンポーネント

Apache Camel 2.3 で利用可能

***exec** コンポーネントは、システムコマンドを実行するために使用できます。*

DEPENDENCIES

*Maven ユーザーは以下の依存関係を **pom.xml** に追加する必要があります。*

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-exec</artifactId>
  <version>${camel-version}</version>
</dependency>
```

***${camel-version}** は、実際のバージョンの Apache Camel (2.3.0 以降)に置き換える必要があります。*

URI 形式

```
exec://executable[?options]
```

*ここでの **executable** は、実行されるシステムコマンドの名前またはファイルパスです。実行可能ファイル名を使用する場合（例： **exec:java**）、実行ファイルはシステムパスになければなりません。*

URI オプション

名前	デフォルト値	説明
args	**null**	実行可能ファイルの引数。引数は、**"** で引用できる1つまたは複数のスペース区切りトークンにすることができます。たとえば、**args="arg 1" arg2** は２つの引数 **arg 1** と **arg2** を使用します。引用符を含めるには、**""** を使用します。例： **args=""arg 1"" arg2** は引数 **"arg 1"** および **arg2** を使用します。
workingDir	**null**	コマンドを実行するディレクトリー。**null** の場合、現在のプロセスの作業ディレクトリーが使用されます。

timeout	**Long.MAX_VALUE**	実行可能ファイルを終了するまでの時間（ミリ秒単位）。タイムアウト内に実行が終了していない場合、コンポーネントは終了リクエストを送信します。
outFile	**null**	実行ファイルによって作成されたファイルの名前。出力と見なす必要があります。**outFile** が設定されていない場合、実行可能ファイルの標準出力(stdout)は出力とみなされます。
binding	**DefaultExecBinding** インスタンス	レジストリー の **org.apache.commons.exec.ExecBinding** への 参照。
commandExecutor	**DefaultCommandExecutor** インスタンス	コマンドの実行をカスタマイズする レジストリー の **org.apache.commons.exec.ExecCommandExecutor** への参照。デフォルトのコマンドエグゼキューターは、commons-exec ライブラリー を使用します。実行したすべてのコマンドにシャットダウンフックを追加します。
useStderrOnEmptyStdout	**false**	**stdout** が空の場合、このコンポーネントは Camel Message Body に **stderr** が設定されることを示すブール値。この動作は、デフォルトでは無効になっています (**false**)。

メッセージヘッダー

サポートされるヘッダーは org.apache.camel.component.exec.ExecBinding で定義されています。

名前	タイプ	メッセージ	説明
ExecBinding.EXEC_COMMAND_EXECUTABLE	**文字列**	**in**	実行する system コマンドの名前。URI の **実行ファイル** を上書きします。
ExecBinding.EXEC_COMMAND_ARGS	**java.util.List<String>**	**in**	実行可能ファイルの引数。引数は文字通りに使用され、引用は適用されません。URI の既存 **引数** を上書きします。

ExecBinding.EXEC_COMMAND_ARGS	**文字列**	**in**	Camel 2.5: 実行可能ファイルの引数は、各引数が空白文字で区切られた Single 文字列として指定されます(URI オプションの引数を参照)。**引数は文字通りに使用され、引用は適用されません。URI の既存 引数 を上書きします。**
ExecBinding.EXEC_COMMAND_OUT_FILE	**文字列**	**in**	実行ファイルによって作成されたファイルの名前。実行可能ファイルの出力と見なす必要があります。URI 内の既存の **outFile** を上書きします。
ExecBinding.EXEC_COMMAND_TIMEOUT	**long**	**in**	実行可能ファイルを終了するまでの時間（ミリ秒単位）。URI の既存の **タイムアウト** を上書きします。
ExecBinding.EXEC_COMMAND_WORKING_DIR	**文字列**	**in**	コマンドを実行するディレクトリー。URI 内の既存の **workingDir** を上書きします。
ExecBinding.EXEC_EXIT_VALUE	**int**	**out**	このヘッダーの値は、実行ファイルの **終了値** です。ゼロ以外の終了値は、通常、異常な終了を示します。終了値は OS に依存することに注意してください。
ExecBinding.EXEC_STDERR	**java.io.InputStream**	**out**	このヘッダーの値は、実行可能ファイルの標準エラーストリーム(stderr)を参照します。stderr が書き込まれていない場合、値は **null** になります。

ExecBinding.EXEC_USE_STDERR_ON_EMPTY_STDOUT	boolean	in	**stdout** が空の場合、このコンポーネントは Camel Message Body に **stderr** が設定されることを示します。この動作は、デフォルトでは無効になっています (**false**)。

メッセージボディー

Exec コンポーネントが、java.io.InputStream に 変換できるメッセージボディーの を受信する場合、標準入力(stdin)を介して実行可能ファイルに入力を提供するために使用されます。実行後、メッセージボディー は実行の結果です。つまり、stdout、stderr、終了値、および out ファイルが含まれる org.apache.camel.components.exec.ExecResult インスタンスです。このコンポーネントは、便宜上、以下の ExecResult 型コンバーター をサポートします。

From	終了
ExecResult	**java.io.InputStream**
ExecResult	**文字列**
ExecResult	**byte []**
ExecResult	**org.w3c.dom.Document**

単語数(LINUX)の実行

以下の例では、wc （単語数、Linux）を実行して /usr/share/dict/words ファイルの単語をカウントします。単語 count （出力）は、wc のスタンドアロン出力ストリームで書かれています。

```
from("direct:exec")
.to("exec:wc?args=--words /usr/share/dict/words")
.process(new Processor() {
    public void process(Exchange exchange) throws Exception {
      // By default, the body is ExecResult instance
      assertIsInstanceOf(ExecResult.class, exchange.getIn().getBody());
      // Use the Camel Exec String type converter to convert the ExecResult to String
      // In this case, the stdout is considered as output
      String wordCountOutput = exchange.getIn().getBody(String.class);
      // do something with the word count
    }
});
```

JAVA の実行

以下の例では、java がシステムパスにあると、-server と -version の 2 つの引数を指定して java を実行します。

```
from("direct:exec")
.to("exec:java?args=-server -version")
```

以下の例では、-server、-version、および sytem プロパティー user.name の 3 つの引数を使用して、c:/temp で java を実行します。

```
from("direct:exec")
.to("exec:c:/program files/jdk/bin/java?args=-server -version -
Duser.name=Camel&workingDir=c:/temp")
```

ANT スクリプトの実行

以下の例では、ant.bat がシステムパスにあり、CamelExecBuildFile.xml が現在のディレクトリーにあると、CamelExecBuildFile.xml ビルドファイルで Apache Ant (Windows のみ) を実行します。

```
from("direct:exec")
.to("exec:ant.bat?args=-f CamelExecBuildFile.xml")
```

以下の例では、ant.bat コマンドは出力を -l を使用する CamelExecOutFile.txt にリダイレクトします。CamelExecOutFile.txt ファイルは、outFile=CamelExecOutFile.txt ファイルとして使用されます。この例では、ant.bat がシステムパスにあり、CamelExecBuildFile.xml が現在のディレクトリーにあることを前提としています。

```
from("direct:exec")
.to("exec:ant.bat?args=-f CamelExecBuildFile.xml -l
CamelExecOutFile.txt&outFile=CamelExecOutFile.txt")
.process(new Processor() {
    public void process(Exchange exchange) throws Exception {
      InputStream outFile = exchange.getIn().getBody(InputStream.class);
      assertIsInstanceOf(InputStream.class, outFile);
      // do something with the out file here
    }
  });
```

ECHO の実行(WINDOWS)

echo や dir などのコマンドは、オペレーティングシステムのコマンドインタープリターでのみ実行できます。この例は、Windows でこのようなコマンド echo を実行する方法を示しています。

```
from("direct:exec").to("exec:cmd?args=/C echo echoString")
```

第47章 ファブリックコンポーネント

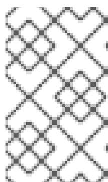
概要

Fabric コンポーネントは、Apache Camel エンドポイントの場所検出メカニズムを実装します。このメカニズムを使用して、エンドポイントのクラスター上で負荷分散を提供することもできます。クライアント側(プロデューサーエンドポイント)では、エンドポイントは抽象 ID で表され、実行時に ID は特定のエンドポイント URI に解決されます。URI は(Fuse Fabric によって提供される)分散レジストリーに格納されるため、デプロイ時にトポロジーを動的に指定できる柔軟なアプリケーションを作成できます。

DEPENDENCIES

Fabric コンポーネントは、ファブリックが有効な Red Hat JBoss Fuse コンテナーのコンテキストでのみ使用できます。**fabric-camel** 機能がインストールされていることを確認する必要があります。Fabric のコンテキストでは、該当するプロファイルに追加して機能をインストールします。たとえば、**my-master-profile** というプロファイルを使用している場合は、以下のコンソールコマンドを入力して **fabric-camel** 機能を追加します。

```
karaf@root> fabric:profile-edit --features fabric-camel my-master-profile
```



注記

fabric コンポーネントを適切に使用するには、**fabric-zookeeper** や **fabric-commands** などのプロファイルを追加してください。ただし、jetty エンドポイントを使用している場合は、**camel-jetty** 機能を含めます。

URI 形式

ファブリックエンドポイントには、以下の URI 形式があります。

```
fabric:ClusterID[:PublishedURI[?Options]]
```

URI の形式は、コンシューマーエンドポイントまたはプロデューサーエンドポイントを指定するために使用されるかどうかによって異なります。

Fabric コンシューマーエンドポイント の場合、URI 形式は次のとおりです。

```
fabric:ClusterID:PublishedURI[?Options]
```

指定した URI **PublishedURI** がファブリックレジストリーに公開され、**ClusterId** クラスターに関連付けられます。オプション **Options** はコンシューマーエンドポイントインスタンスの作成時に使用されますが、オプションはファブリックレジストリーの **PublishedURI** で公開されません。

Fabric プロデューサーエンドポイント の場合、URI 形式は次のとおりです。

```
fabric:ClusterID
```

クライアントはファブリックレジストリーで ID **ClusterId** を検索し、接続する URI を検出します。

URI オプション

Fabric コンポーネント自体は URI オプションをサポートしません。ただし、公開された URI にオプションを指定することもできます。これらのオプションは URI の一部としてファブリックレジストリーに保存され、以下のように使用されます。

- サーバーにのみ適用されるサーバーのみのオプション - オプションは、実行時にサーバーエンドポイント（コンシューマーエンドポイント）に適用されます。
- クライアントにのみ適用されるクライアントのみのオプション - オプションは、実行時にクライアントエンドポイント（プロデューサーエンドポイント）に適用されます。
- クライアントとサーバーに共通する一般的なオプション - クライアントとサーバーの両方に適用されます。

ファブリックエンドポイントのユースケース

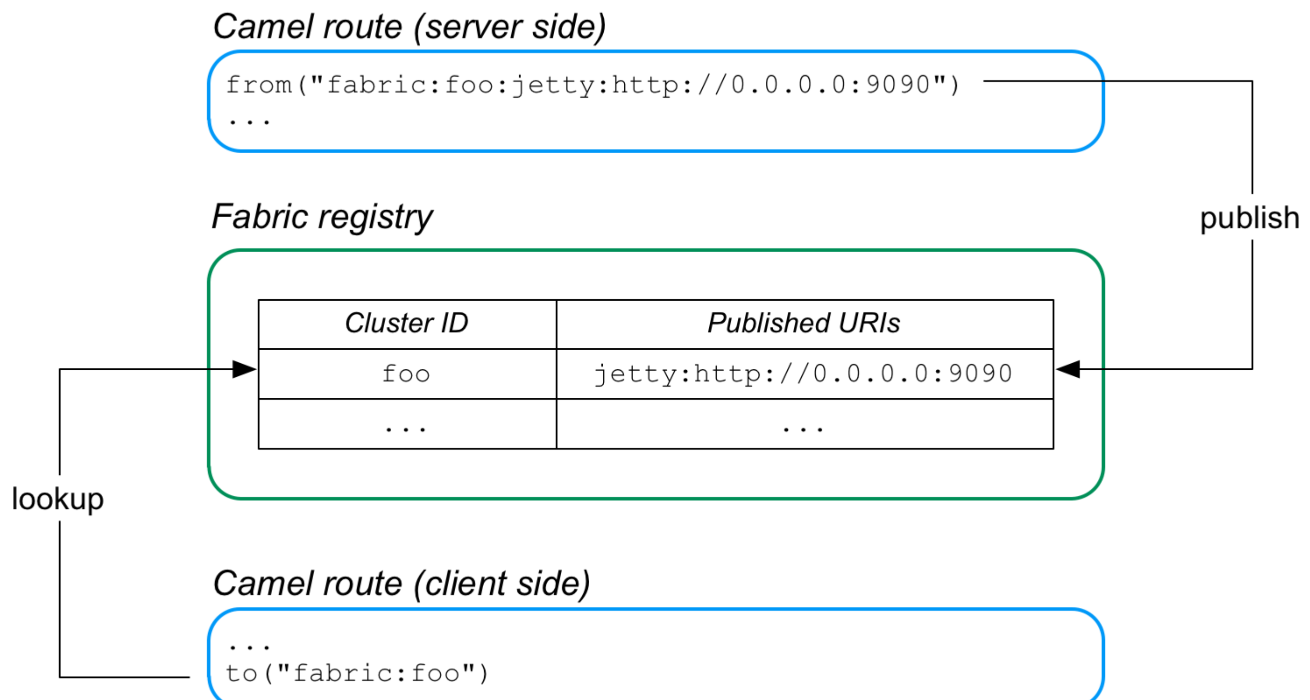
Fabric エンドポイントは基本的に Apache Camel エンドポイントの検出メカニズムを提供します。たとえば、以下の基本的なユースケースをサポートします。

- 「ロケーションの検出」 の形式にする必要があります。
- 「Load-balancing クラスター」 の形式にする必要があります。

ロケーションの検出

図47.1「Fabric によるロケーションの検出」 では、Fabric エンドポイントが実行時にロケーションの検出を有効にする方法の概要を説明します。

図47.1 Fabric によるロケーションの検出



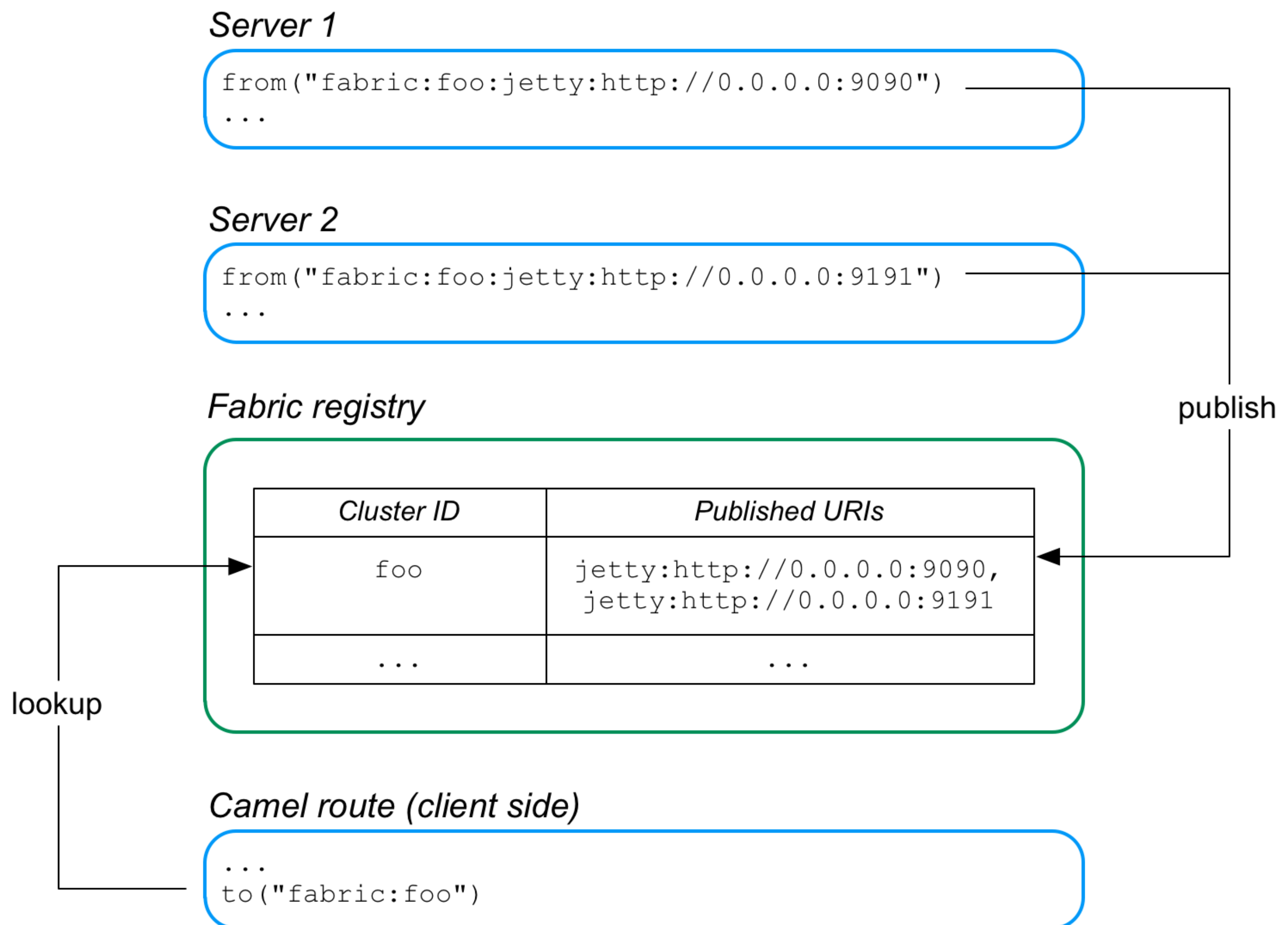
このアプリケーションのサーバー側は、Fabric エンドポイントで始まるルートによって定義され、Fabric エンドポイントは URI **jetty:http://0.0.0.0:9090** を公開します。このルートが起動すると、クラスター ID **foo** の下にある fabric レジストリーの Jetty URI が自動的に登録されます。

アプリケーションのクライアント側は、Fabric エンドポイント **fabric:foo** で終わるルートによって定義されます。クライアントルートが起動すると、ファブリックレジストリーで ID **foo** が自動的に検索され、関連付けられた Jetty エンドポイント URI が取得されるようになりました。その後、クライアントは検出された Jetty URI を使用してプロデューサーエンドポイントを作成し、対応するサーバーポートに接続します。

LOAD-BALANCING クラスター

図47.2「Fabric による負荷分散」 では、Fabric エンドポイントが負荷分散クラスターを作成できる方法の概要を説明します。

図47.2 Fabric による負荷分散



この場合、URI (**jetty:http://0.0.0.0:9090**) および **jetty:http://0.0.0.0:9191** で 2 つの Jetty サーバーが作成されます。これらの公開された URI には **fabric:foo:** の接頭辞が付けられるため、両方の Jetty URI はファブリックレジストリーの同じ クラスター ID **foo** の下に登録されます。

クライアントルートが起動すると、ファブリックレジストリーで ID **foo** が自動的に検索されるようになりました。**foo** ID は複数のエンドポイント URI に関連付けられているため、ファブリックはランダム負荷分散アルゴリズムを実装し、利用可能な URI のいずれかを選択します。その後、クライアントは選択した URI を使用してプロデューサーエンドポイントを作成します。

自動再接続機能

fabric エンドポイントは自動再接続をサポートします。そのため、クライアントエンドポイント (producer エンドポイント)がそのサーバーエンドポイントへの接続を失うと、自動的にファブリックレジストリーに戻り、別の URI を要求してから、新しい URI に接続します。

エンドポイント URI の公開

ファブリックレジストリーでエンドポイント URI **PublishedURI** を公開するには、パブリッシャー構文 **FabricScheme:ClusterID:PublishedURI** で fabric エンドポイントを定義します。この構文は、コンシューマーエンドポイント（つまり、**from** DSL コマンドに表示されるエンドポイント）でのみ使用できることに注意してください。

例47.1「URI の公開」 は Jetty HTTP サーバーを実装するルートを示しています。ここでは、Jetty URI は ID **cluster** の下のファブリックレジストリーに公開されます。ルートは、HTTP 応答の本文で定数メッセージ **Response from Zookeeper agent** を返す単純な HTTP サーバーです。

例47.1 URI の公開

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="
           http://www.osgi.org/xmlns/blueprint/v1.0.0
           https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

 <bean id="fabric-camel" class="io.fabric8.camel.FabricComponent"/>

   <camelContext id="camel" trace="false"
xmlns="http://camel.apache.org/schema/blueprint">
      <route id="fabric-server">
         <from uri="fabric-camel:cluster:jetty:http://0.0.0.0:9090/fabric"/>
         <log message="Request received : ${body}"/>
         <setHeader headerName="karaf.name">
            <simple>${sys.karaf.name}</simple>
         </setHeader>
         <transform>
            <simple>Response from Zookeeper agent</simple>
         </transform>
      </route>
   </camelContext>

</blueprint>
```

前述の例について、以下の点に注意してください。

- Fabric コンポーネントは、**CuratorFramework** オブジェクトを使用して ZooKeeper サーバー (Fabric レジストリー)に接続します。ここで、**CuratorFramework** オブジェクトへの参照が自動的に提供されます。
- **from** DSL コマンドは、ファブリック URI **fabric-camel:cluster:jetty:http://0.0.0.0:9090/fabric** を定義します。ランタイム時に、以下の 2 つのことが発生します。
  - 指定された **jetty** URI はクラスター ID **cluster** の下のファブリックレジストリーに公開されます。
  - Jetty エンドポイントはアクティベートされ、ルートのコンシューマーエンドポイントとして使用されます(**fabric-camel:cluster:** 接頭辞なしで指定された場合のみ)。

ルートは Blueprint XML に実装されているため、通常、このコードを含むファイルを Maven プロジェクトの **src/main/resources/OSGI-INF/blueprint** ディレクトリーに追加します。

エンドポイント URI の検索

fabric レジストリーで URI を検索する場合は、**FabricScheme:ClusterID** の形式で、ID で fabric エンドポイント URI を指定するだけです。この構文は、プロデューサーエンドポイントで使用されます（例：**to** DSL コマンドに表示されるエンドポイント）。

例47.2「URI の検索」は、ファブリックレジストリーで指定された ID **cluster** を検索して、実行時に HTTP エンドポイントが動的に検出される HTTP クライアントを実装するルートを示しています。

例47.2 URI の検索

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="
           http://www.osgi.org/xmlns/blueprint/v1.0.0
           https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

  <bean id="fabric-camel" class="io.fabric8.camel.FabricComponent"/>

  <camelContext id="camel" trace="false"
xmlns="http://camel.apache.org/schema/blueprint">

    <route id="fabric-client">
      <from uri="timer://foo?fixedRate=true&amp;period=10000"/>
      <setBody>
          <simple>Hello from Zookeeper server</simple>
      </setBody>
      <to uri="fabric-camel:cluster"/>
      <log message=">>> ${body} : ${header.karaf.name}"/>
    </route>

  </camelContext>

  <reference interface="org.apache.camel.spi.ComponentResolver"
             filter="(component=jetty)"/>

</blueprint>
```

ルートは Blueprint XML に実装されているため、通常、このコードを含むファイルを Maven プロジェクトの **src/main/resources/OSGI-INF/blueprint** ディレクトリーに追加します。

負荷分散の例

基本的には、ファブリックエンドポイントを使用すると、負荷分散の実装が簡単になります。必要なのは、同じ クラスター ID の下に複数のエンドポイント URI を公開することだけです。クライアントがそのクラスター ID を検索すると、利用可能なエンドポイント URI の一覧からランダムに選択できるようになりました。

負荷分散クラスターのサーバーはほぼ同じ設定になります。基本的に、これら間の唯一の違いは、異な

るホスト名や IP ポートを使用してエンドポイント URI を公開することです。ただし、負荷分散クラスター内の各サーバーに個別の OSGi バンドルを作成する代わりに、設定変数を使用してホストまたはポートを指定できるようにするテンプレートを定義することが推奨されます。

例47.3「ロードバランシングクラスターのサーバーテンプレート」 は、負荷分散クラスターでサーバーを定義するテンプレートアプローチを示しています。

例47.3 ロードバランシングクラスターのサーバーテンプレート

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
        xsi:schemaLocation="
        http://www.osgi.org/xmlns/blueprint/v1.0.0
        https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

    <!-- osgi blueprint property placeholder -->
    <cm:property-placeholder
      id="myConfig"
      persistent-id="io.fabric8.examples.camel.loadbalancing.server"/>

    <bean id="fabric-camel" class="io.fabric8.camel.FabricComponent"/>

    <camelContext id="camel" trace="false"
xmlns="http://camel.apache.org/schema/blueprint">
        <!-- using Camel properties component and refer
           to the blueprint property placeholder by its id -->
        <propertyPlaceholder id="properties"
                    location="blueprint:myConfig"
                    prefixToken="[[" suffixToken="]]"/>

        <route id="fabric-server">
          <from uri="fabric-camel:cluster:jetty:http://0.0.0.0:[[portNumber]]/fabric"/>
          <log message="Request received : ${body}"/>
          <setHeader headerName="karaf.name">
            <simple>${sys.karaf.name}</simple>
          </setHeader>
          <transform>
            <simple>Response from Zookeeper agent</simple>
          </transform>
        </route>
    </camelContext>

</blueprint>
```

*まず、OSGi Blueprint プロパティープレースホルダーを初期化する必要があります。プロパティープレースホルダーのメカニズムを使用すると、OSGi Config Admin サービスからプロパティー設定を読み取り、Blueprint 設定ファイルのプロパティーを置き換えることができます。この例では、プロパティープレースホルダーは **io.fabric8.examples.camel.loadbalancing.server** 永続 ID からプロパティーにアクセスします。OSGi Config Admin サービスの永続 ID は、関連するプロパティー設定のコレクションを識別します。プロパティープレースホルダーの初期化後に、構文を使用して永続 ID から任意のプロパティー値(**[[PropName]]**)にアクセスできます。*

*Fabric endpont URI はプロパティープレースホルダーのメカニズムを利用して、実行時に Jetty ポート **[[portNumber]]** の値を置き換えます。デプロイ時に、**portName** プロパティーの値を指定できます。たとえば、カスタム機能を使用する場合は、機能定義で プロパティーを指定できます(Add OSGi configurations to the featureを参照してください)。または、Fuse Management Console でデプロイメントプロファイルを定義するときに設定プロパティーを指定することもできます。*

OSGI バンドルプラグインの設定

*Fabric エンドポイントを使用する OSGi バンドルを定義する場合は、以下の Java パッケージをインポートするように **Import-Package** バンドルヘッダーを設定する必要があります。*

> ***io.fabric8.zookeeper***

たとえば、Maven を使用してアプリケーションをビルドする場合、例47.4「Maven バンドルプラグインの設定」 は、必要なパッケージをインポートするように Maven バンドルプラグインを設定する方法を示しています。

例47.4 Maven バンドルプラグインの設定

```
<project ... >
  ...
  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${project.groupId}.${project.artifactId}</Bundle-
SymbolicName>
            <Import-Package>
              io.fabric8.zookeeper,
              *
            </Import-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

第48章 FACEBOOK

FACEBOOK コンポーネント

Camel 2.12 以降で利用可能

Facebook コンポーネントは、Facebook4J を使用してアクセス可能なすべての Facebook API へのアクセスを提供します。これにより、メッセージを生成、コメント、写真、アルバム、ビデオ、写真、リンクなど、投稿を取得、追加、および削除できます。また、投稿、ユーザー、グループなどをポーリングできる API もサポートしています。

Facebook では、すべてのクライアントアプリケーション認証に OAuth を使用する必要があります。アカウントで camel-facebook を使用するには、https://developers.facebook.com/apps で Facebook 内に新しいアプリケーションを作成し、アプリケーションにアカウントへのアクセスを許可する必要があります。Facebook アプリケーションの id およびシークレットを使用すると、現在のユーザーを必要としない Facebook API にアクセスできます。ログインユーザーを必要とする API には、ユーザーアクセストークンが必要です。ユーザーアクセストークンの取得に関する詳細は、https://developers.facebook.com/docs/facebook-login/access-tokens/ を参照してください。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-facebook</artifactId>
    <version>${camel-version}</version>
</dependency>
```

URI 形式

```
facebook://[endpoint]?[options]
```

FACEBOOKCOMPONENT

facebook コンポーネントは、以下の Facebook アカウント設定を使用して設定できますが、これは必須となります。org.apache.camel.component.facebook.config.FacebookConfiguration タイプの Bean プロパティー設定 を使用して、値をコンポーネントに提供できます。oAuthAccessToken オプションは省略できますが、アプリケーション API へのアクセスのみを許可します。

これらのオプションは、エンドポイント URI で直接設定することもできます。

オプション	説明
oAuthAppId	アプリケーション ID
oAuthAppSecret	アプリケーションシークレット
oAuthAccessToken	ユーザーアクセストークン

上記の設定に加えて、以下の必須ではないオプションを使用して、コンポーネントの 設定プロパティー またはエンドポイント URI のいずれかを使用して、基礎となる Facebook4J ランタイムを設定 できます。

オプション	説明	デフォルト値
oAuthAuthorizationURL	OAuth 認証 URL	https://www.facebook.com/dialog/oauth
oAuthPermissions	デフォルトの OAuth パーミッション。コンマ区切りのパーミッション名。詳細は、https://developers.facebook.com/docs/reference/login/#permissions を参照してください。	null
oAuthAccessTokenURL	OAuth アクセストークンの URL	https://graph.facebook.com/oauth/access_token
debugEnabled	デバッジ出力を有効にします。組み込みロガーでのみ有効	false
gzipEnabled	Facebook GZIP エンコーディングの使用	true
httpConnectionTimeout	HTTP 接続のタイムアウト（ミリ秒単位）	20000
httpDefaultMaxPerRoute	ルートごとの HTTP 最大接続数	2
httpMaxTotalConnections	HTTP 最大合計接続数	20
httpProxyHost	HTTP プロキシーサーバーのホスト名	null
httpProxyPassword	HTTP プロキシーサーバーのパスワード	null
httpProxyPort	HTTP プロキシーサーバーポート	null
httpProxyUser	HTTP プロキシーサーバーのユーザー名	null
httpReadTimeout	HTTP の読み取りタイムアウト（ミリ秒単位）	120000
httpRetryCount	HTTP の再試行回数	0
httpRetryIntervalSeconds	HTTP 再試行の間隔（秒単位）	5

httpStreamingReadTimeout	HTTP ストリーミングの読み取りタイムアウト（ミリ秒単位）	40000
jsonStoreEnabled	true に設定すると、生の JSON フォームは DataObjectFactory に保存されます。	false
mbeanEnabled	true に設定すると、Facebook4J mbean が登録されます。	false
prettyDebugEnabled	true に設定されている場合に JSON デバッグ出力を事前送信	false
restBaseURL	API ベース URL	https://graph.facebook.com/
useSSL	SSL の使用	true
videoBaseURL	ビデオ API ベース URL	https://graph-video.facebook.com/
clientURL	Facebook4J API クライアント URL	http://facebook4j.org/en/facebook4j-<version>xml
clientVersion	Facebook4J クライアント API バージョン	1.1.12

プロデューサーエンドポイント：

プロデューサーエンドポイントは、以下の表のエンドポイント名とオプションを使用できます。エンドポイントは、get またはsearch 接頭辞なしで短縮名を使用することもできます。.必須ではないエンドポイントオプションは [] で示されます。

プロデューサーエンドポイントは、特殊なオプション *inBody* を使用することもできます。これには、値が Camel Exchange In メッセージに含まれる endpoint オプションの名前が含まれる必要があります。たとえば、以下のルートの facebook エンドポイントは、受信メッセージボディーのユーザー ID 値のアクティビティーを取得します。

> ***from("direct:test").to("facebook://activities?inBody=userId")...***

エンドポイントオプションは、エンドポイント URI またはメッセージヘッダーで動的に指定できます。メッセージヘッダー名は CamelFacebook.option の形式でなければなりません。たとえば、前のルートの userId オプション値は、CamelFacebook.userId メッセージヘッダーに別の方法で提供できます。inBody オプションはメッセージヘッダーを上書きすることに注意してください。たとえば、inBody=user は CamelFacebook.userId ヘッダーを上書きすることに注意してください。

文字列を返すエンドポイントは、作成または変更されたエンティティーの Id を返します。たとえば、addAlbumPhoto は新しいアルバム ID を返します。ブール値を返すエンドポイントは、成功の場合は true を返し、それ以外の場合は false を返します。Facebook API エラーが発生すると、エンドポイントは facebook4j.FacebookException が原因で RuntimeCamelException を出力します。

エンドポイント	ショートネーム	オプション	ボディタイプ
アカウント			
getAccounts	アカウント	[reading],[userId]	facebook4j.ResponseList<facebook4j.Account>
アクティビティー			
getActivities	アクティビティー	[reading],[userId]	facebook4j.ResponseList<facebook4j.Activity>
Albums			
addAlbumPhoto	addAlbumPhoto	albumId,source,[message]	文字列
commentAlbum	commentAlbum	albumId,message	文字列
createAlbum	createAlbum	albumCreate,[userId]	文字列
getAlbum	album	albumId,[reading]	facebook.Album
getAlbumComments	albumComments	albumId,[reading]	facebook4j.ResponseList<facebook4j.Comment>
getAlbumCoverPhoto	albumCoverPhoto	albumId	java.net.URL
getAlbumLikes	albumLikes	albumId,[reading]	facebook4j.ResponseList<facebook4j.Like>
getAlbumPhotos	albumPhotos	albumId,[reading]	facebook4j.ResponseList<facebook4j.Photos>
getAlbums	albums	[reading],[userId]	facebook4j.ResponseList<facebook4j.Album>
コメント			
deleteComment	deleteComment	commentId	boolean
getComment	comment	commentId	facebook4j.Comment
getCommentLikes	commentLikes	commentId,[reading]	facebook4j.ResponseList<facebook4j.Like>
イベント			

getEvent	event	eventId,[reading]	facebook4j.Event
getEventPhotos	eventPhotos	eventId,[reading]	facebook4j.ResponseList<facebook4j.Photo>
getEventPictureURL	eventPictureURL	eventId,[size]	java.net.URL
getEvents	events	[reading],[userId]	facebook4j.ResponseList<facebook4j.Event>
getEventVideos	eventVideos	eventId,[reading]	facebook4j.ResponseList<facebook4j.Video>
getRSVPStatusAsNoreply	rSVPStatusAsNoreply	eventId,[userId]	facebook4j.ResponseList<facebook4j.RSVPStatus>
getRSVPStatusInAttending	rSVPStatusInAttending	eventId,[userId]	facebook4j.ResponseList<facebook4j.RSVPStatus>
getRSVPStatusInDeclined	rSVPStatusInDeclined	eventId,[userId]	facebook4j.ResponseList<facebook4j.RSVPStatus>
getRSVPStatusInMaybe	rSVPStatusInMaybe	eventId,[userId]	facebook4j.ResponseList<facebook4j.RSVPStatus>
postEventFeed	postEventFeed	eventId,postUpdate	文字列
postEventLink	postEventLink	eventId,link,[message]	文字列
postEventPhoto	postEventPhoto	eventId,source,[message]	文字列
postEventStatusMessage	postEventStatusMessage	eventId,message	文字列
postEventVideo	postEventVideo	eventId,source,[title,description]	文字列
rsvpEventAsAttending	rsvpEventAsAttending	eventId	boolean
rsvpEventAsDeclined	rsvpEventAsDeclined	eventId	boolean
rsvpEventAsMaybe	rsvpEventAsMaybe	eventId	boolean
ファミリー			

getFamily	family	[reading],[userId]	facebook4j.ResponseList<facebook4j.Family>
お気に入り			
getBooks	書籍	[reading],[userId]	facebook4j.ResponseList<facebook4j.Book>
getGames	games	[reading],[userId]	facebook4j.ResponseList<facebook4j.Game>
getMovies	videos	[reading],[userId]	facebook4j.ResponseList<facebook4j.Movie>
getMusic	music	[reading],[userId]	facebook4j.ResponseList<facebook4j.Music>
getTelevision	television	[reading],[userId]	facebook4j.ResponseList<facebook4j.Television>
FQL (Facebook Query Language)			
executeFQL	executeFQL	query,[locale]	facebook4j.internal.org.json.JSONArray
executeMultiFQL	executeMultiFQL	queries,[locale]	java.util.Map<tring,facebook4j.internal.org.json.JSONArray>
friends			
addFriendlistMember	addFriendlistMember	friendlistId,userId	boolean
createFriendlist	createFriendlist	friendlistName,[userId]	文字列
deleteFriendlist	deleteFriendlist	friendlistId	boolean
getBelongsFriend	belongsFriend	friendId,[reading],[userId]	facebook4j.ResponseList<facebook4j.Friend>
getFriendlist	friendlist	friendlistId,[reading]	facebook4j.FriendList
getFriendlistMembers	friendlistMembers	friendlistId	facebook4j.ResponseList<facebook4j.Friend>
getFriendlists	friendlists	[reading],[userId]	facebook4j.ResponseList<facebook4j.FriendList>

getFriends	friends	[reading],[userId]	facebook4j.ResponseList<facebook4j.Friend>
removeFriendlistMember	removeFriendlistMember	friendlistId,userId	boolean
games			
deleteAchievement	deleteAchievement	achievementURL,[userId]	boolean
deleteScore	deleteScore	[userId]	boolean
getAchievements	達成度	[reading],[userId]	facebook4j.ResponseList<facebook4j.Achievement>
getScores	スコア	[reading],[userId]	facebook4j.ResponseList<facebook4j.Score>
postAchievement	postAchievement	achievementURL,[userId]	文字列
postScore	postScore	scoreValue,[userId]	文字列
グループ			
getGroup	group	groupId,[reading]	facebook4j.Group
getGroupDocs	groupDocs	groupId,[reading]	facebook4j.ResponseList<facebook4j.GroupDoc>
getGroupFeed	groupFeed	groupId,[reading]	facebook4j.ResponseList<facebook4j.Post>
getGroupMembers	groupMembers	groupId,[reading]	facebook4j.ResponseList<facebook4j.GroupMember>
getGroupPictureURL	groupPictureURL	groupId	java.net.URL
getGroups	groups	[reading],[userId]	facebook4j.ResponseList<facebook4j.Group>
postGroupFeed	postGroupFeed	groupId,postUpdate	文字列
postGroupLink	postGroupLink	groupId,link,[message]	文字列

postGroupStatusMessage	postGroupStatusMessage	groupId,message	文字列
Insights			
getInsights	Insights	objectId,metric,[reading]	facebook4j.ResponseList<facebook4j.Insight>
likes			
getUserLikes	userLikes	[reading],[userId]	facebook4j.ResponseList<facebook4j.Like>
リンク			
commentLink	commentLink	linkId,message	文字列
getLink	リンク	linkId,[reading]	facebook4j.Link
getLinkComments	linkComments	linkId,[reading]	facebook4j.ResponseList<facebook4j.Comment>
getLinkLikes	linkLikes	linkId,[reading]	facebook4j.ResponseList<facebook4j.Like>
メッセージ			
getInbox	inbox	[reading],[userId]	facebook4j.InboxResponseList<facebook4j.Inbox>
getMessage	message	messageId,[reading]	facebook4j.Message
getOutbox	送信トレイ	[reading],[userId]	facebook4j.ResponseList<facebook4j.Message>
getUpdates	updates	[reading],[userId]	facebook4j.ResponseList<facebook4j.Message>
通知			
getNotifications	通知	[includeRead],[reading],[userId]	facebook4j.ResponseList<facebook4j.Notification>
markNotificationAsRead	markNotificationAsRead	notificationId	boolean
パーミッション			

getPermissions	permissions	[userId]	java.util.List<acebook4j.Permission>
revokePermission	revokePermission	permissionName, [userId]	boolean
写真			
addTagToPhoto	addTagToPhoto	photoId,[toUserId],[toUserIds],[tagUpdate]	boolean
commentPhoto	commentPhoto	photoId,message	文字列
deletePhoto	deletePhoto	photoId	boolean
getPhoto	photo	photoId,[reading]	facebook4j.Photo
getPhotoComments	photoComments	photoId,[reading]	facebook4j.ResponseList<facebook4j.Comment>
getPhotoLikes	photoLikes	photoId,[reading]	facebook4j.ResponseList<facebook4j.Like>
getPhotos	写真	[reading],[userId]	facebook4j.ResponseList<facebook4j.Photo>
getPhotoURL	photoURL	photoId	java.net.URL
getTagsOnPhoto	tagsOnPhoto	photoId,[reading]	facebook4j.ResponseList<facebook4j.Tag>
postPhoto	postPhoto	source,[message],[place],[noStory],[userId]	文字列
updateTagOnPhoto	updateTagOnPhoto	photoId,[toUserId],[tagUpdate]	boolean
Pokes			
getPokes	pokes	[reading],[userId]	facebook4j.ResponseList<facebook4j.Poke>
posts			
commentPost	commentPost	postId,message	文字列
deletePost	deletePost	postId	boolean

getFeed	フィード	[reading],[userId]	facebook4j.ResponseList<facebook4j.Post>
getPost	post	postId,[reading]	facebook4j.Post
getPostComments	postComments	postId,[reading]	facebook4j.ResponseList<facebook4j.Comment>
getPostLikes	postLikes	postId,[reading]	facebook4j.ResponseList<facebook4j.Like>
getPosts	posts	[reading],[userId]	facebook4j.ResponseList<facebook4j.Post>
getTagged	tagged	[reading],[userId]	facebook4j.ResponseList<facebook4j.Post>
postFeed	postFeed	postUpdate,[userId]	文字列
postLink	postLink	link,[message],[userId]	文字列
postStatusMessage	postStatusMessage	message,[userId]	文字列
getSubscribedto	subscribedto	[reading],[userId]	facebook4j.ResponseList<facebook4j.Subscribedto>
getSubscribers	加入者	[reading],[userId]	facebook4j.ResponseList<facebook4j.Subscriber>
ユーザーのテスト			
createTestUser	createTestUser	appId,[name],[userLocale],[permissions]	facebook4j.TestUser
deleteTestUser	deleteTestUser	testUserId	boolean
getTestUsers	testUsers	appId	java.util.List<acebook4j.TestUser>
makeFriendTestUser	makeFriendTestUser	testUser1,testUser2	boolean
Users			
getMe	me	[reading]	facebook4j.User

getPictureURL	pictureURL	[size],[userId]	java.net.URL
getUser	user	userId,[reading]	facebook4j.User
getUsers	users	ids	java.util.List<acebook4j.User>
動画			
commentVideo	commentVideo	videoId,message	文字列
getVideo	video	videoId,[reading]	facebook4j.Video
getVideoComments	videoComments	videoId,[reading]	facebook4j.ResponseList<facebook4j.Comment>
getVideoCover	videoCover	videoId	java.net.URL
getVideoLikes	videoLikes	videoId,[reading]	facebook4j.ResponseList<facebook4j.Like>
getVideos	動画	[reading],[userId]	facebook4j.ResponseList<facebook4j.Video>
postVideo	postVideo	source, [title,description], [userId]	文字列
検索			
search	search	query,[reading]	facebook4j.ResponseList<facebook4j.internal.org.json.JSONObject>
searchEvents	events	query,[reading]	facebook4j.ResponseList<facebook4j.Event>
searchGroups	groups	query,[reading]	facebook4j.ResponseList<facebook4j.Group>
searchPlaces	場所	query,[reading], [center,distance]	facebook4j.ResponseList<facebook4j.Place>
searchPosts	posts	query,[reading]	facebook4j.ResponseList<facebook4j.Post>
SearchUsers	users	query,[reading]	facebook4j.ResponseList<facebook4j.User>

コンシューマーエンドポイント：

read #Reading Options パラメーターを取るプロデューサーエンドポイントはいずれも、コンシューマーエンドポイントとして使用できます。ポーリングコンシューマーは、since および until フィールドを使用して、ポーリング間隔内の応答を取得します。他の読み取りフィールドに加えて、最初のポーリングのエンドポイントで値の最初の値を指定できます。

単一のルートエクスチェンジで List （または facebook4j.ResponseList）を返すエンドポイントではなく、camel-facebook は返されたオブジェクトごとに 1 つのルートエクスチェンジを作成します。たとえば、facebook://home が 5 つの投稿になる場合、ルートは 5 回実行されます(Post ごとに 1 回)。

URI オプション

名前	タイプ	説明
achievementURL	java.net.URL	達成の一意の URL
albumCreate	facebook4j.AlbumCreate	作成する Facebook Album
albumId	文字列	アルバム ID
allowNewOptions	boolean	他のユーザーが新しいオプションを追加できる場合は True
appId	文字列	Facebook アプリケーションの ID
中央	facebook4j.GeoLocation	場所 latitude および longitude
commentId	文字列	コメント ID
description	文字列	説明テキスト
distance	int	名の距離
eventId	文字列	イベント ID
eventUpdate	facebook4j.EventUpdate	作成または更新するイベント
friendId	文字列	friend ID
friendUserId	文字列	friend ユーザー ID
friendlistId	文字列	フリンドリスト ID
friendlistName	文字列	先頭リスト名
groupId	文字列	グループ ID
ids	String[]	ユーザーの ID

includeRead	boolean	未読の通知に加えて、ユーザーがすでに読んでいるという通知を有効にします。
リンク	java.net.URL	リンク URL
linkId	文字列	リンク ID
locale	java.util.Locale	目的の FQL ロケール
message	文字列	メッセージのテキスト
messageId	文字列	メッセージ ID
メトリクス	文字列	メトリクス名
name	文字列	ユーザー名をテストします。最初は last の形式でなければなりません
noStory	boolean	true に設定すると、アプリケーションを使用して写真をアップロードする際に、ユーザーのプロファイルで自動的に生成されるフィードストーリーが抑制されます。
noteId	文字列	ノート ID
notificationId	文字列	通知 ID
objectId	文字列	Insights オブジェクト ID
optionDescription	文字列	質問の回答オプションの説明
options	java.util.List<String>	質問の回答オプション
permissionName	文字列	パーミッション名
permissions	文字列	perm1,perm2,... の形式でユーザーパーミッションをテストします。
photoId	文字列	写真 ID
配置	文字列	写真に関連付けられた場所の Facebook ID
placeId	文字列	場所 ID

postId	文字列	投稿 ID
postUpdate	facebook4j.PostUpdate	作成または更新の投稿
クエリー	java.util.Map<tring>	FQL クエリー
query	文字列	FQL クエリーまたは検索用語で検索*エンドポイント
質問	文字列	質問テキスト
questionId	文字列	質問 ID
読み取り	facebook4j.Reading	オプションの読み取りパラメーター。Reading Options(#Reading Options)を参照してください。
scoreValue	int	値の数値スコア
size	facebook4j.PictureSize	図のサイズ、大型、通常の、小、または角の 1 つ
source	facebook4j.Media	java.io.File または java.io.Inputstream のいずれかからのメディアコンテンツ
subject	文字列	サブジェクトの注記
tagUpdate	facebook4j.TagUpdate	写真タグ情報
testUser1	facebook4j.TestUser	ユーザーのテスト
testUser2	facebook4j.TestUser	ユーザーのテスト
testUserId	文字列	テストユーザーの ID
title	文字列	タイトルテキスト
toUserId	文字列	タグを付けるユーザーの ID
toUserIds	java.util.List<tring>	タグを付けるユーザーの ID
userId	文字列	Facebook ユーザー ID
userId1	文字列	ユーザーの ID
userId2	文字列	ユーザーの ID

userIds	String[]	イベントに招待するユーザーの ID
userLocale	文字列	テストユーザーロケール
videoId	文字列	ビデオ ID

オプションの読み取り

facebook4j.Reading タイプの read オプションでは、パラメーターの読み取りがサポートされるようになりました。これにより、特定のフィールドの選択や結果数の制限などが可能になります。詳細は、Facebook Developers のGraph API を参照してください。

また、ポーリング全体で重複したメッセージが送信されないように、Facebook データをポーリングするためにコンシューマーエンドポイントによっても使用されます。

読み取りオプションは、facebook4j.Reading タイプの参照または値、またはCamelFacebook. 接頭辞を持つエンドポイント URI またはエクスチェンジヘッダーのいずれかで以下の読み取りオプションを使用して指定できます。

オプション	説明
reading.fields	field1,field2,.. の形式で取得するフィールド名。
reading.limit	リスト結果に対して返す項目数を制限します。たとえば、10 の制限により、1 から 10 が返されます。
reading.offset	リスト結果の開始オフセット（制限が 10 など）、10 のオフセットはアイテム 11 から 20 を返します。
reading.until	時間ベースのデータの範囲の最後を参照する Unix タイムスタンプまたは strtotime データ値
reading.since	時間ベースのデータの範囲の開始を示す Unix タイムスタンプまたは strtotime データ値
reading.locale	特定のロケールでローカライズされたコンテンツを取得します。言語が文字列として指定される [,country][,variant]
reading.with	ロケーション情報がアタッチされているオブジェクトに関する情報を取得し、true に設定します。
reading.metadata	Facebook Graph API Introspection を使用してオブジェクトメタデータを取得し、true に設定します。
reading.filter	ユーザーのストリームフィルターキー。Facebook stream_filterを参照してください。

メッセージヘッダー

「コンシューマーエンドポイント：」 はいずれも、CamelFacebook. 接頭辞を持つプロデューサーエンドポイントのメッセージヘッダーに提供できます。

メッセージボディー

すべての結果メッセージ本文は、Facebook4J API が提供するオブジェクトを使用します。プロデューサーエンドポイントは、inBody エンドポイントパラメーターに受信メッセージボディーのオプション名を指定できます。

配列を返すエンドポイント、または facebook4j.ResponseList または java.util.List の場合、コンシューマーエンドポイントはリスト内のすべての要素を個別のメッセージにマップします。

ユースケース

Facebook プロファイル内に投稿を作成するには、このプロデューサーを facebook4j.PostUpdate ボディーに送信します。

```
from("direct:foo")
 .to("facebook://postFeed/inBody=postUpdate);
```

自宅フィードのすべてのステータスを 5 秒ごとにポーリングするには、次のコマンドを実行します。

```
from("facebook://home?consumer.delay=5000")
 .to("bean:blah");
```

ヘッダーから動的オプションでプロデューサーを使用した検索。

バーヘッダーには公開投稿で実行する Facebook 検索文字列があるため、この値を CamelFacebook.query ヘッダーに割り当てる必要があります。

```
from("direct:foo")
 .setHeader("CamelFacebook.query", header("bar"))
 .to("facebook://posts");
```

第49章 FILE2

FILE COMPONENT: APACHE CAMEL 2.0 以降

File コンポーネントはファイルシステムへのアクセスを提供します。これにより、他の Apache Camel コンポーネントや、 他のコンポーネントからのメッセージをディスクに保存できます。

URI 形式

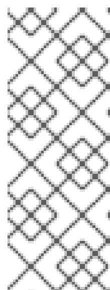
```
file:directoryName[?options]
```

または

```
file://directoryName[?options]
```

directoryName は基礎となるファイルディレクトリーを表します。

URI にクエリーオプションは **?option=value&option=value&..** の形式で追加できます。



注記

Apache Camel は、開始ディレクトリーで設定されたエンドポイントのみをサポートします。そのため、directoryName はディレクトリーである必要があります。1 つのファイルのみを使用する場合は、fileName オプションを使用できます（例：**fileName=thefilename**）。また、開始ディレクトリーには ${ } プレースホルダーの動的式を含めることはできません。ここでも、**fileName** オプションを使用してファイル名の動的部分を指定します。

別のアプリケーションによって現在書き込まれているファイルの読み取りを回避する

JDK File IO API は、別のアプリケーションが現在ファイルを書き込み/コピーしているかどうかの検出に少し制限されていることに注意してください。実装は、OS プラットフォームによっても異なる場合があります。これにより、Apache Camel が別のプロセスでロックされていないと判断し、消費を開始する可能性があります。したがって、お使いの環境に応じて独自の調査を行う必要があります。これを支援するために、Apache Camel はさまざまな **readLock** オプションと、使用できる **doneFileName** オプションを提供します。「他のファイルを直接ドロップするフォルダーからのファイルの使用」 のセクションも参照してください。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI オプション

名前	デフォルト値	説明
autoCreate	**true**	ファイルのパス名に不足しているディレクトリーを自動的に作成します。ファイルコンシューマーの場合は、開始ディレクトリーを作成することを意味します。ファイルプロデューサーの場合は、ファイルが書き込まれるディレクトリーを意味します。
bufferSize	128kb	書き込みバッファーのサイズ (バイト単位)。

fileName	**null**	File Language などの式 を使用して、ファイル名を動的に設定します。コンシューマーの場合は、ファイル名フィルターとして使用されます。プロデューサーの場合、書き込むファイル名を評価するために使用されます。式が設定されている場合、**CamelFileName** ヘッダーよりも優先されます。(**注：** ヘッダー自体も Expressionにすることができます)。式オプションは **String** 型と **Expression** タイプの両方をサポートします。式が **String** タイプである場合、これは **常に** File Language を使用して評価されます。式が **Expression** タイプである場合、指定された **Expression** タイプが使用されます。これにより、たとえば OGNL 式を使用できます。コンシューマーの場合、これを使用してファイル名をフィルターリングできるため、たとえば、ファイル 言語 構文 **mydata-${date:now:yyyyMMdd}.txt** を使用して現在のファイルを使用できます。Camel 2.11 以降では、プロデューサーは既存の **CamelFileName** ヘッダーよりも優先される **CamelOverruleFileName** ヘッダーをサポートします。**CamelOverruleFileName** は一度だけ使用されるヘッダーであり、**CamelFileName** の一時的な保存を回避し、後で復元する必要があるため、簡単になります。
フラット化	**false**	flatten は、ファイル名パスをフラット化して先頭のパスを削除するため、ファイル名のみになります。これにより、サブディレクトリーに再帰的に使用できますが、たとえばファイルを別のディレクトリーに書き込む場合、ファイルは単一のディレクトリーに書き込まれます。これをプロデューサーで **true** に設定すると、**CamelFileName** ヘッダーで認識されたファイル名はすべて先頭パスから取り除かれます。

charset	**null**	Camel 2.5: このオプションはファイルのエンコーディングを指定するために使用されます。camel は Exchange.CHARSET_NAME で Exchange プロパティーをこのオプションの値に設定します。
copyAndDeleteOnRenameFail	**true**	Camel 2.9: ファイルを直接変更できなかった場合に、フォールバックしてファイルのコピーと削除を行うかどうか。このオプションは FTP コンポーネントでは使用できません。
renameUsingCopy	**false**	Camel 2.13.1: コピーおよび削除ストラテジーを使用して名前変更操作を実行します。これは主に、通常の名前変更操作が信頼できない環境で使用されます (たとえば、異なるファイルシステムまたはネットワーク間)。このオプションは、**copyAndDeleteOnRenameFail** パラメーターよりも優先されますが、これは追加の遅延後にのみ自動的にコピーおよび削除ストラテジーにフォールバックします。

コンシューマーのみ

名前	デフォルト値	説明
initialDelay	**1000**	ファイル/ディレクトリーの開始をポーリングするまでのミリ秒。
delay	**500**	次にファイル/ディレクトリーをポーリングするまでのミリ秒。
useFixedDelay	**true**	プール間の固定遅延を使用するには、**true** に設定します。それ以外の場合、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。
runLoggingLevel	**TRACE**	Camel 2.8: コンシューマーはポーリング時に開始/完了のログ行をログに記録します。このオプションを使用すると、ログレベルを設定できます。

再帰	false	ディレクトリーの場合は、すべてのサブディレクトリー内のファイルも検索します。
delete	false	**true** の場合、ファイルは正常に処理されると削除されます。
noop	false	**true** の場合、ファイルは移動または削除されません。このオプションは、読み取り専用データまたは ETL タイプの要件に適しています。**noop=true** の場合、Apache Camel は **idempotent=true** も設定し、同じファイルを何度も消費しないようにします。
preMove	null	File Language などの式 を使用して、処理 **前** に移動する際にファイル名を動的に設定します。たとえば、進行中のファイルを **order** ディレクトリーに移動するには、この値を **order** に設定します。
move	.camel	File Language などの式 を使用して、処理 **後** に移動する際にファイル名を動的に設定します。ファイルを **.done** サブディレクトリーに移動するには、**.done** と入力します。
moveFailed	null	File Language などの式 を使用して、処理 **後に失敗したファイルを移動する際にファイル** 名を動的に設定します。ファイルを **エラー** サブディレクトリーに移動するには、**error** を実行します。**注記：** ファイルを別の場所に移動すると、ファイルを別の場所に移動するとエラーを **処理** できるため、Apache Camel はファイルを再度取得できません。
include	null	ファイル名が正規表現パターンに一致する場合にファイルを含めるために使用されます（照合は **Camel 2.17 以降と大文字と小文字を区別し** ます）。

exclude	**null**	ファイル名が正規表現パターンに一致する場合にファイルを除外するために使用されます（一致は **Camel 2.17 以降と大文字と小文字を区別し** ます）。
antInclude	**null**	**Camel 2.10:** Ant スタイルのフィルターが含まれます（例：**antInclude=*{}*/*{}.txt**）。コンマ区切り形式で複数の組み込みを指定できます。ant パスフィルターの詳細は、「ANT パスマッチャーを使用したフィルターリング」 を参照してください。
antExclude	**null**	**Camel 2.10:** Ant style filter exclusion.**antInclude** と **antExclude** の両方を使用する場合は、**antInclude** よりも **antExclude** が優先されます。コンマ区切り形式で複数の除外を指定できます。ant パスフィルターの詳細は、「ANT パスマッチャーを使用したフィルターリング」 を参照してください。
antFilterCaseSensitive	**true**	**Camel 2.11:** 大文字/小文字を区別しない Ant スタイルのフィルター。
idempotent	**false**	Apache Camel がすでに処理されたファイルをスキップできるように、Idempotent Consumer EIP パターンを使用するオプション。デフォルトでは、1000 エントリーを保持するメモリーベースの LRUCache を使用します。**noop=true** の場合、同じファイルを何度も使用することを避けるために、べき等性も有効になります。
idempotentKey	**式**	**Camel 2.11:** カスタムのべき等キーを使用するには、以下を実行します。デフォルトでは、ファイルの絶対パスが使用されます。File Language を使用すると、ファイル名やファイルサイズを使用することができます（例：idempotentKey=$-$）。

idempotentRepository	**null**	Pluggable repository as a org.apache.camel.processor.idempotent.MessageIdRepository class.指定がない場合はデフォルトで **MemoryMessageIdRepository** を使用し、**べき等** 性が **true** になります。
inProgressRepository	**memory**	org.apache.camel.processor.idempotent.MessageIdRepository クラスとしてのプラグ可能な in-progress リポジトリー。in-progress リポジトリーは、現在進行中のファイルが消費されていることを示すために使用されます。デフォルトでは、メモリーベースのリポジトリーが使用されます。
filter	**null**	**org.apache.camel.component.file.GenericFileFilter** クラスとしてのプラグ可能なフィルター。フィルターがその **accept ()** メソッドで **false** を返す場合、ファイルをスキップします。Apache Camel には、**camel-spring** コンポーネントの **ANT パスマッチャー** フィルターも同梱されています。詳細は、以下のセクションを参照してください。
shuffle	**false**	**Camel 2.16:** ファイルのリストをシャッフルするには（ランダムな順序でのソート）
sorter	**null**	java.util.Comparator<org.apache.camel.component.file.GenericFile > クラスとしてのプラグ可能なソーター。
sortBy	**null**	File 言語 を使用した組み込みソート。ネストされたソートをサポートしているため、ファイル名でのソートと、2 つ目のグループとして変更日でソートできます。詳細は、以下のソートセクションを参照してください。
readLock	**none**	ファイルに排他的な読み取りロックがある (つまり、ファイルが進行中または書き込み中ではない) 場合にのみファイルをポーリングするために、コンシューマーが使

		用します。Apache Camel は、ファイルロックが許可されるまで待機します。 **readLock** オプションは、以下の組み込みストラテジーをサポートします。 ● **none** は、読み取りロックはまったくありません。 ● **changed** は、長さ/変更のタイムスタンプを使用して、ファイルが現在コピーされているかどうかを検出します。これを判断するために少なくとも 1 秒待機します。したがって、このオプションはファイルを他のほど速く消費できませんが、JDK IO API はファイルが別のプロセスで現在使用されているかどうかを判断できないため、信頼性を高めることができます。**readLockCheckInterval** オプションを使用して、チェック頻度を設定できます。 ● **markerFile** Camel はマーカーファイル ***fileName*.camelLock** を作成し、その後にロックを保持します。このオプションは FTP コンポーネントでは使用できません。 ● **fileLock** は **java.nio.channels.FileLock** を使用します。マウント/共有を介してリモートファイルシステムにアクセスする場合は、そのファイルシステムが分散ファイルロックをサポートしている場合を除き、このアプローチは回避する必要があります。 ● **rename** は、排他的な読み取りロックを取得できる場合は、テストとしてファイルの名前変更を試みるために使用します。 ● **idempotent** (Camel 2.16) は、**idempotentRepository** を読み取りロックとして使用するために使用します。これにより、

		べき等性リポジトリーの実装がサポートする場合に、クラスターリングをサポートする読み取りロックを使用できます。 **注記：** 外部プログラムがファイルを書き込む場合は、Red Hat は **readLock** オプションの代わりに **doneFileName** オプションを使用することを推奨します。 **警告：** 読み取りロックストラテジーのほとんどは、クラスターモードでの使用には適していません。つまり、同じディレクトリーで同じファイルを読み取ろうとしている複数のコンシューマーを含めることはできません。この場合、読み取りロックは確実に機能しません。Hazelcast コンポーネントから などのクラスター対応べき等リポジトリー実装を使用する場合、べき等読み取りロックは、クラスター化された信頼性をサポートします。
readLockTimeout	**0** (FTP の場合は **2000**)	読み取りロックでサポートされている場合、読み取りロックのオプションのタイムアウト（ミリ秒単位）。読み取りロックを許可できず、タイムアウトがトリガーされた場合、Apache Camel はファイルをスキップします。Apache Camel が次にポーリングすると、がファイルを再度試行します。この場合、読み取りロックが許可される可能性があります。現在、**fileLock**、**changed**、および **rename** がタイムアウトに対応します。
readLockCheckInterval	**1000** (FTP の場合、**5000**)	**Camel 2.6:** 読み取りロックでサポートされている場合、読み取りロックの間隔（ミリ秒単位）。この間隔は、読み取りロックを取得する試行間のスリープに使用されます。たとえば、**changed** 読み取りロックを使用する場合は、**低速な書き込み** に対応するために間隔を長く設定できます。デフォルトの 1 秒。プロデューサーによるファイルの書き込みが非常に遅い場合は **短すぎる** 可能性があります。

readLockMinLength	**1**	**Camel 2.10.1:** このオプションは、**readLock=changed** にのみ適用されます。このオプションを使用すると、最小ファイルの長さを設定できます。デフォルトで Camel はファイルにデータが含まれていると想定するため、デフォルト値は 1 です。このオプションをゼロに設定すると、長さがゼロのファイルを使用できます。
readLockLoggingLevel	**WARN**	**Camel 2.12:** 読み取りロックを取得できなかったときに使用されるログレベル。デフォルトでは、WARN がログに記録されます。このレベルを変更できます。たとえば、ログを記録しないように OFF に設定できます。このオプションは、changed、fileLock、rename タイプの readLock にのみ適用されます。
readLockMarkerFile	**true**	**Camel 2.14:** changed、rename、または exclusive 読み取りロックタイプでマーカーファイルを使用するかどうか。デフォルトでは、他のプロセスが同じファイルを取得するのを防ぐために、マーカーファイルも使用されます。このオプションを false に設定すると、この動作をオフにできます。たとえば、Camel アプリケーションによってマーカーファイルをファイルシステムに書き込みたくない場合などです。
readLockRemoveOnRollback	**true**	**Camel 2.16:** このオプションは、**readLock=idempotent** にのみ適用されます。ファイル処理が失敗し、ロールバックが行われるときに、べき等性リポジトリーからファイル名エントリーを削除するかどうかを指定します。**false** の場合、ファイル名のエントリーが確認されます（ファイルがコミットしたかのように）。

readLockRemoveOnCommit	**false**	Camel 2.16: このオプションは、**readLock=idempotent** にのみ適用されます。ファイル処理に成功し、コミットが行われるときに、べき等性リポジトリーからファイル名エントリーを削除するかどうかを指定します。デフォルトでは、エントリーは削除され **ない** ため、別のアクティブなノードがファイルを取得しようとすると競合状態が発生しなくなります。この場合、（競合状態のリスクが引き継がった後に）**X** 分後にファイル名エントリーがエビクトされるように、べき等リポジトリーでエビクションストラテジーを設定することが推奨されます。
readLockDeleteOrphanLockFiles	**true**	Camel 2.16: このオプションは、**readLock=markerFile** にのみ適用されます。起動時に、孤立した読み取りロックファイルを削除するかどうかを指定します。このファイルは、Camel が適切にシャットダウンされなかった場合にファイルシステムに残っている可能性があります(JVM クラッシュ)。**false** の場合、孤立したロックファイルが Camel による対応するファイルの取得をブロックしなくなる可能性があります。
directoryMustExist		Camel 2.5: **startingDirectoryMustExist** と似ていますが、これは再帰的なサブディレクトリーのポーリングに適用されます。
doneFileName	**null**	Camel 2.6: 提供されている場合、Camel は **完了** ファイルが存在する場合にのみファイルを消費します。このオプションは、使用するファイル名を設定します。固定の名前を指定できます。または、動的プレースホルダーを使用することもできます。**完了** ファイルは、**常に元のファイルと同じフォルダーに** 想定されます。例は、**完了ファイルの使用と、実行ファイル の書き込みセクションを参照して** ください。注記：外部プログラムがファイルを書き込む場合には、Red Hat は **doneFileName** オプションを使用することを推奨します。

exclusiveReadLockStrategy	**null**	**org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy** 実装としてのプラグ可能な読み取りロック。
maxMessagesPerPoll	**0**	ポーリングごとに収集するメッセージの最大数を定義する整数。デフォルトでは最大値は設定されていません。たとえば 1000 などの制限を設定して、サーバーが起動時に数千のファイルを読み込まないようにすることができます。無効にするには、0 または負の値を設定します。
eagerMaxMessagesPerPoll	**true**	Camel 2.9.3: **maxMessagesPerPoll** の制限が Eager であるかどうかを制御できます。eager の場合、ファイルのスキャン中に制限されます。false の場合、すべてのファイルをスキャンし、並び替えを実行します。このオプションを false に設定すると、すべてのファイルを最初にソートしてからポーリングを制限できます。ソートのためにすべてのファイルの詳細がメモリー内にあるため、メモリー使用量が大きくなることに注意してください。
minDepth	0	Camel 2.8: ディレクトリーを再帰的に処理する際に処理を開始する最小深度。**minDepth=1 は** ベースディレクトリーを意味します。**minDepth=2** は最初のサブディレクトリーを意味します。このオプションは FTP コンシューマーではサポートされて **いません**。
maxDepth	**Integer.MAX_VALUE**	Camel 2.8: ディレクトリーを再帰的に処理する際にトラバースする最大深度。このオプションは FTP コンシューマーではサポートされて **いません**。

processStrategy	**null**	プラグ可能な **org.apache.camel.component.file.GenericFileProcessStrategy** により、独自の **readLock** オプションまたは同様のものを実装できます。特別な **準備完了** ファイルが存在するなど、ファイルを使用する前に特別な条件を満たす必要がある場合にも使用できます。このオプションを設定すると、**readLock** オプションは適用されません。
startingDirectoryMustExist	**false**	開始ディレクトリーの存在が必要かどうか。**autoCreate** オプションはデフォルトで有効になっていることに注意してください。つまり、開始ディレクトリーが存在しない場合は通常は自動作成されます。**autoCreate** を無効にし、これを有効にして開始ディレクトリーが存在する必要があることを確認できます。ディレクトリーが存在しない場合は例外が発生します。
pollStrategy	**null**	プラグ可能な **org.apache.camel.spi.PollingConsumerPollStrategy** により、**Exchange** が作成され、Camel でルーティングされる前に、通常ポーリング操作中に発生するエラー処理を制御するカスタム実装を提供できます。つまり、ポーリングで情報を収集している間にエラーが発生したため、たとえばファイルネットワークへのアクセスに失敗し、Camel がこれにアクセスしてファイルをスキャンできません。デフォルトの実装は、**WARN** レベルで原因となった例外をログに記録し、無視します。
sendEmptyMessageWhenIdle	**false**	Camel 2.9: ポーリングコンシューマーがファイルをポーリングしなかった場合は、このオプションを有効にして、代わりに空のメッセージ（ボディーなし）を送信できます。

consumer.bridgeErrorHandler	**false**	Camel 2.10: コンシューマーを Camel ルーティングエラーハンドラーにブリッジングできるようにします。つまり、ファイルの取得を試みる際に発生した例外はメッセージとして処理され、ルーティングエラーハンドラーによって処理されます。デフォルトでは、コンシューマーは **org.apache.camel.spi.ExceptionHandler** を使用して例外に対応し、デフォルトでは **WARN**/**ERROR** レベルでログに記録され、無視されます。詳細は、How to use the Camel error handler to handling with exception triggered outside the routing engine を参照してください。
scheduledExecutorService	**null**	Camel 2.10: コンシューマーに使用するカスタム/共有スレッドプールを設定できます。デフォルトでは、各コンシューマーに独自の単一スレッドのスレッドプールがあります。このオプションを使用すると、複数のファイルコンシューマー間でスレッドプールを共有できます。
scheduler	**null**	Camel 2.12: カスタムスケジューラーを使用して、コンシューマーの実行をトリガーします。詳細はコンシューマーの ポーリング を参照してください。たとえば、CRON 式をサポートする Quartz2 および Springベースのスケジューラーがあります。
backoffMultiplier	**0**	Camel 2.12: 後続のアイドル状態/エラーが連続して発生した場合は、スケジュールされたポーリングコンシューマーのバックオフを解除します。乗数は、次の試行が行われる前にスキップされるポーリングの数になります。このオプションが設定されている場合は、**backoffIdleThreshold** や **backoffErrorThreshold** も設定する必要があります。詳細は、Polling Consumer を参照してください。
backoffIdleThreshold	**0**	Camel 2.12: **backoffMultipler** が開始する前に発生する必要のある後続のアイドルポーリングの数。

backoffErrorThreshold	**0**	Camel 2.12: **backoffMultipler** が開始する前に発生すべき後続のエラーポーリングの数（エラーにより失敗）。
onCompletionExceptionHandler		Camel 2.16: カスタムの **org.apache.camel.spi.ExceptionHandler** を使用して、コンシューマーがコミットまたはロールバックを実行する完了プロセスのファイル中に発生する例外を処理します。デフォルトの実装は、WARN レベルですべての例外をログに記録し、無視します。
probeContentType	**false**	Camel 2.17: コンテンツタイプのプローブを有効にするかどうか。有効にすると、コンシューマーは **Files#probeContentType(java.nio.file.Path)** を使用してファイルの Content-Type を判別し、キー **Exchange#FILE_CONTENT_TYPE** のヘッダーとしてメッセージに保存します。 **注記：** Camel 2.15 以降では、この関数が導入され、常に有効になっていました。しかし、**Camel 2.17 以降**、一部のファイルシステムで問題が発生する可能性があるため、デフォルトではオフになっています。

extendedAttributes	**null**	Camel 2.17: オプションの値に応じて **Files.getAttribute(ava.nio.file.Path, java.lang.String attribute)** または **Files.readAttributes(ava.nio.file.Path, java.lang.String attributes)** を使用して **java.nio.file.attribute** クラスで拡張ファイル属性の収集を有効にするには、以下を実行します。このオプションは、収集する属性のコンマ区切りリスト（例：**basic:creationTime**, **posix:group**-or a simple wildcard-or **posix:*** など）をサポートします。属性名の前には、基本属性が照会されます。結果は、**CamelFileExtendedAttributes** タイプのキー **Map<String, Object>** のヘッダーとして保存されます。**キー** は属性の名前（例：**posix:group**）で、**値** は **Files.getAttribute()** または ----------|----- への呼び出しによって返された属性になります。**Files.readAttributes**

ファイルコンシューマーのデフォルト動作

- ***デフォルトでは、ファイルは処理期間中ロックされません。***
- ***ルートが完了すると、ファイルは .camel サブディレクトリーに移動し、それらが削除されるように表示されます。***
- *File Consumer* ***は、..、.camel、.m2、 . groovy など、名前がドットで始まるファイルを常にスキップします。***
- ***includeNamePrefix、includeNamePostfix、excludeNamePrefix、excludeNamePostfix、およびregexPattern などのオプションが使用される場合、ファイル（ディレクトリーではない）のみが有効なファイル名に一致します。***

プロデューサーのみ

名前	デフォルト値	説明

fileExist	オーバーライド	同じ名前のファイルがすでに存在する場合のアクション。以下の値を指定できます：Override、Append、Fail、Ignore、Move、および TryRename (Camel 2.11.1)。デフォルトの を **上書き** し、既存のファイルを置き換えます。**append** は、既存のファイルにコンテンツを追加します。**Fail** は **GenericFileOperationException** を出力し、既存のファイルがあることを示します。ignore は問題をサイレントに無視し、既存のファイルは上書きし **ません** が、問題があるとします。**Move オプションには Camel 2.10.1 以降が必要で、対応する moveExisting オプションも設定する必要があります。オプション eagerDeleteTargetFile を使用して、ファイルの移動と既存ファイルがすでに存在する場合の動作を制御できます。そうしないと、移動操作が失敗します。Move オプションは、ターゲットファイルを書き込む前に既存のファイルを移動します。TryRenameCamel 2.11.1 は、tempFileName オプションが使用されている場合にのみ適用されます。これにより、存在チェックを実行せずに、一時的なファイル名から実際のファイル名への変更を試みることができます。このチェックは、一部のファイルシステム、特に FTP サーバーでは高速になる場合があります。**
tempPrefix	**null**	このオプションは、一時的な名前を使用してファイルを書き込み、書き込みが完了した後に、その名前を実際の名前に変更するために使用されます。書き込み中のファイルを識別し、(排他的読み取りロックを使用せずに) コンシューマーが進行中のファイルを読み取らないようにするために使用できます。大きなファイルをアップロードするときに FTP でよく使用されます。

tempFileName	**null**	Camel 2.1: **tempPrefix** オプションと **同じ** ですが、File 言語を使用する一時的なファイル名の命名をより詳細に制御できます。
keepLastModified	**false**	Camel 2.2: ソースファイル（存在する場合）から最後に変更されたタイムスタンプを保持します。**Exchange.FILE_LAST_MODIFIED** ヘッダーを使用してタイムスタンプを見つけます。このヘッダーには、**java.util.Date** またはタイムスタンプ付きの **long** を含めることができます。タイムスタンプが存在し、オプションが有効な場合は、書き込まれたファイルにこのタイムスタンプが設定されます。**注記：** このオプションは、**ファイル** プロデューサーにのみ適用されます。このオプションは、ftp プロデューサーでは使用**できません**。

eagerDeleteTargetFile	**true**	Camel 2.3: 既存のターゲットファイルを完全に削除するかどうか。このオプションは、**fileExists=Override** および **tempFileName** オプションも使用する場合にのみ適用されます。これを使用して、一時ファイルが書き込まれる前にターゲットファイルを削除することを無効化 (false に設定) できます。たとえば、大きなファイルを書き込んで、一時ファイルの書き込み中にターゲットファイルを存在させたい場合があります。これにより、一時ファイルの名前がターゲットファイル名に変更される直前まで、ターゲットファイルは削除されません。このオプションの Camel 2.10.1 から、**fileExist=Move** が有効で、既存のファイルが存在する場合に、既存のファイルを削除するかどうかを制御するためにも使用されます。オプション **copyAndDeleteOnRenameFail** が **false** の場合、既存のファイルが存在する場合は例外が出力されます。**true** の場合、既存のファイルは移動操作の前に削除されます。
doneFileName	**null**	Camel 2.6: 指定された場合、元のファイルが書き込まれると、Camel は 2 番目の **完了** ファイルを書き込みます。**完了** ファイルは空になります。このオプションは、使用するファイル名を設定します。固定の名前を指定できます。または、動的プレースホルダーを使用することもできます。**完了** ファイルは、**常に** 元のファイルと同じフォルダーに書き込まれます。例は、writing done file セクションを参照してください。注記：外部プログラムがファイルを書き込む場合には、Red Hat は **doneFileName** オプションを使用することを推奨します。

allowNullBody	**false**	**Camel 2.10.1:** ファイルの書き込み中に null ボディーが許可されるかどうかを指定するために使用されます。true に設定すると空のファイルが作成され、false に設定して null の本文をファイルコンポーネントに送信しようとすると、Cannot write null body to file.という GenericFileWriteException が出力されます。fileExist オプションが Override に設定されている場合、ファイルは切り捨てられ、append に設定するとファイルは変更されません。
forceWrites	**true**	**Camel 2.10.5/2.11:** ファイルシステムへの書き込みを強制的に同期するかどうか。たとえばログや監査ログへの書き込みなど、このレベルの保証が必要ない場合はオフにすることでパフォーマンスが向上します。
chmod	**null**	**Camel 2.15.0:** プロデューサーによって送信されるファイルパーミッションを指定します。ここで、**chmod** の値は **000** から **777** の間でなければなりません。先頭の数字（例：**0755**-are）は無視されます。
chmodDirectory	**null**	**Camel 2.17.0:** プロデューサーが不足しているディレクトリーを作成する場合に使用されるディレクトリーパーミッションを指定します。ここで、**chmod** の値は **000** から **777** の間でなければなりません。先頭の数字（例：**0755**-are）は無視されます。

ファイルプロデューサーのデフォルト動作

- *デフォルトでは、同じ名前の既存ファイルが存在する場合は、既存のファイルを上書きします。Apache Camel 1.x では、**Append** がファイルプロデューサーのデフォルトです。これは **java.io.File** を使用したデフォルトのファイル操作であるため、Apache Camel 2.0 で**Override** に変更しました。また、camel-ftp コンポーネントで使用する FTP ライブラリーのデフォルトです。*

移動および削除操作

移動または削除操作は、ルーティングが完了した後（コマンド後）で実行されます。したがって、エクスチェンジ の処理中に、ファイルは引き続き inbox フォルダーにあります。

以下に例を示します。

```
from("file://inbox?move=.done").to("bean:handleOrder");
```

inbox フォルダーにファイルがドロップされると、ファイルコンシューマーはこれを通知し、**handleOrder** Bean にルーティングされる新しい **FileExchange** を作成します。その後、Bean は **File** オブジェクトを処理します。この時点で、ファイルは **inbox** フォルダーに残ります。Bean が完了し、ルートが完了すると、ファイルコンシューマーは move 操作を実行し、ファイルを **.done** サブフォルダーに移動します。

move オプションおよびpreMove オプションはディレクトリー名とみなされます(File Language や Simple などの式を使用する場合、式の評価の結果は使用されるファイル名になります)。たとえば、を設定すると、以下のようになります。

```
move=../backup/copy-of-${file:name}
```

次に、使用する File 言語 を使用して、使用するファイル名を返します。これは、relative または absolute のいずれかです。相対的な場合、ディレクトリーは、ファイルが消費されたフォルダー内からのサブフォルダーとして作成されます。

デフォルトでは、Apache Camel は、ファイルが消費されたディレクトリーとの関連で、消費されたファイルを **.camel** サブフォルダーに移動します。

処理後にファイルを削除する場合は、ルートは以下のようになります。

```
from("file://inobox?delete=true").to("bean:handleOrder");
```

ファイルが処理される 前 に移動する 事前 移動操作が導入されました。これにより、処理前にこのサブフォルダーに移動する際にスキャンされたファイルをマークできます。

```
from("file://inbox?preMove=inprogress").to("bean:handleOrder");
```

pre move と通常の移動を組み合わせることができます。

```
from("file://inbox?preMove=inprogress&move=.done").to("bean:handleOrder");
```

したがって、このような状況では、処理中および処理された後にファイルが 進行中 のフォルダーにある場合、このファイルは **.done** フォルダーに移動します。

移動および PREMOVE オプションの詳細な制御

move オプションおよびpreMove オプションは Expression-based であるため、ディレクトリーおよび名前パターンの高度な設定を行うために File Language の全機能を使用できます。実際、Apache Camel は、入力したディレクトリー名を File Language 式に内部的に変換します。そのため、**move=.done** を入力すると、Apache Camel がこれを **${file:parent}/.done/${file:onlyname}** に変換します。これは、Apache Camel がオプション値に **${ }** を指定していないことを検知した場合にのみ行われます。そのため、**${ }** を含む式を入力すると、式は File Language 式として解釈されます。

したがって、パターンとして現在の日付を持つバックアップフォルダーにファイルを移動する場合は、以下を行うことができます。

```
move=backup/${date:now:yyyyMMdd}/${file:name}
```

MOVEFAILED について

***moveFailed** オプションを使用すると、処理 できない ファイルを、選択したエラーフォルダーなどの別の場所に移動できます。たとえば、タイムスタンプ付きのエラーフォルダー内のファイルを移動するには、**moveFailed=/error/${file:name.noext}-${date:now:yyyyMMddHHmmssSSS}.${file:name.ext}** を使用できます。*

その他の例については、File Language を参照してください。

メッセージヘッダー

このコンポーネントでは、以下のヘッダーがサポートされます。

ファイルプロデューサーのみ

ヘッダー	説明
CamelFileName	書き込むファイルの名前を指定します（エンドポイントディレクトリーへの相対パス）。名前は **String**、File Language または Simple 式を持つ **String**、または Expression オブジェクトになります。**null** の場合、Apache Camel はメッセージ固有の ID に基づいてファイル名を自動生成します。
CamelFileNameProduced	書き込まれた出力ファイルの実際の絶対パス（パス + 名）。このヘッダーは Camel によって設定され、その目的は書き込まれたファイルの名前でエンドユーザーを提供します。
CamelOverruleFileName	Camel 2.11: **CamelFileName** ヘッダーのオーバーラップに使用され、代わりに値を使用します（ただし、プロデューサーは ファイルの書き込み後にこのヘッダーを削除するため、1 回のみ）。値は String のみにすることができます。**fileName** オプションが設定されている場合、これは引き続き評価されていることに注意してください。

ファイルコンシューマーのみ

ヘッダー	説明
CamelFileName	消費されたファイルの名前（エンドポイントに設定された開始ディレクトリーからのオフセットを含む相対パス）。
CamelFileNameOnly	ファイル名のみ (先行パスを含まない名前)。

CamelFileAbsolute	消費されたファイルが絶対パスを表すかどうかを指定する **ブール値** オプション。通常、相対パスの場合は **false** である必要があります。絶対パスは通常使用しないでくださいが、ファイルを絶対パスに移動できるように move オプションに追加しました。ただし、他の場所でも使用することができます。
CamelFileAbsolutePath	ファイルへの絶対パス。相対ファイルの場合、このパスは代わりに相対パスを保持します。
CamelFilePath	ファイルパス。相対ファイルの場合、これは開始ディレクトリー + 相対ファイル名になります。絶対ファイルの場合、これは絶対パスです。
CamelFileRelativePath	相対パス。
CamelFileParent	親パス。
CamelFileLength	ファイルサイズを含む **long** 値。
CamelFileLastModified	ファイルの最終変更のタイムスタンプを含む **long** 値。

バッチコンシューマー

このコンポーネントは、バッチコンシューマー を 実装し ます。

エクスチェンジプロパティー、ファイルコンシューマーのみ

*ファイルコンシューマーは **BatchConsumer** であるため、ポーリングするファイルのバッチ処理をサポートします。バッチ処理により、Apache Camel はいくつかのプロパティーを エクスチェンジ に追加し、現在のインデックスをポーリングしたファイルの数を把握します。*

プロパティー	説明
CamelBatchSize	このバッチでポーリングされたファイルの合計数。
CamelBatchIndex	バッチの現在のインデックス。0 から始まります。
CamelBatchComplete	バッチの最後の エクスチェンジ を示す **ブール** 値。最後のエントリーに対してのみ **true** になります。

これにより、たとえば、このバッチに存在するファイルの数を把握し、Aggregator はこの数のファイルを集約できます。

フォルダーおよびファイル名を使用した一般的な GOTCHAS

Apache Camel がファイルを生成する場合（ファイルの書き込み）、選択したファイル名の設定方法に

影響を及ぼします。デフォルトでは、Apache Camel はメッセージID をファイル名として使用し、メッセージID は通常一意の生成された ID であるため、**ID-MACHINENAME-2443-1211718892437-1-0** などのファイル名になります。このようなファイル名が必要ない場合は、**CamelFileName** メッセージヘッダーにファイル名を指定する必要があります。定数 **Exchange.FILE_NAME** も使用できます。

以下のサンプルコードは、メッセージID をファイル名として使用してファイルを作成します。

```
from("direct:report").to("file:target/reports");
```

report.txt をファイル名として使用するには、以下を実行します。

```
from("direct:report").setHeader(Exchange.FILE_NAME, constant("report.txt")).to(
"file:target/reports");
```

または上記と同じですが、**CamelFileName** の場合：

```
from("direct:report").setHeader("CamelFileName", constant("report.txt")).to(
"file:target/reports");
```

fileName URI オプションを使用してエンドポイントにファイル名を設定する構文。

```
from("direct:report").to("file:target/reports/?fileName=report.txt");
```

ファイル名式

ファイル名は、expression オプションを使用するか、**CamelFileName** ヘッダーの文字列ベースのFile Language 式のいずれかを使用して設定できます。構文およびサンプルについては、File Language を参照してください。

他のファイルを直接ドロップするフォルダーからのファイルの使用

他のアプリケーションがファイルを直接書き込むフォルダーからファイルを消費する場合は注意してください。さまざまな **readLock** オプションを見て、ユースケースに適したものを確認します。ただし、最善の方法は、別のフォルダーに書き込み、書き込み後にドロップフォルダーにファイルを移動することです。ただし、ドロップフォルダーに直接ファイルを書き込むと、ファイルが 変更 されたアルゴリズムを使用して、ファイルのサイズや変更が一定期間変更されるかどうかを確認するため、変更されたファイルが現在書き込みまたはコピーされているかどうかをより適切に検出できます。他の読み取りロックオプションは、これを検出するときに必ずしも適切ではない Java File API に依存します。また、**doneFileName** オプションも確認します。これは、ファイルが完了して使用できるときにマーカーファイル(done)を使用してシグナルを送ります。

完了ファイルの使用

Camel 2.6 以降で利用可能

また、以下の実行ファイルの書き込み のセクション を参照してください。

完了ファイルが存在する場合にのみファイルを消費する場合は、エンドポイントで **doneFileName** オプションを使用できます。

```
from("file:bar?doneFileName=done");
```

ターゲットファイルと同じディレクトリーにファイル名が存在する場合にのみ、bar フォルダーからファイルを消費します。Camel は、ファイルの使用が完了すると、完了ファイルを自動的に削除します。

ただし、ターゲットファイルごとに 1 つのファイルを配置するのが一般的です。これは、相関が 1:1 であることを意味します。これを実行するには、**doneFileName** オプションで動的プレースホルダーを使用する必要があります。現在、Camel は、file:name および **file:name.noext** の 2 つの動的トークンをサポートします。これは ${ } で囲む必要があります。コンシューマーは、接頭辞または接尾辞（両方ではない）として実行されたファイル名の静的部分のみをサポートします。

```
from("file:bar?doneFileName=${file:name}.done");
```

この例では、ファイル名が .done の完了したファイルが存在する場合にのみポーリングされます。以下に例を示します。

- **hello.txt** - 消費されるファイルです。
- **hello.txt.done** - 関連する完了ファイルです。

以下のように、完了ファイルの接頭辞を使用することもできます。

```
from("file:bar?doneFileName=ready-${file:name}");
```

- **hello.txt** - 消費されるファイルです。
- **ready-hello.txt** - 関連する完了ファイルです。

行われたファイルの作成

Camel 2.6 以降で利用可能

af ファイルを書き込んだ後は、ファイルが完了 して書き込まれたことを他の人に示すために、追加の完了ファイルをマーカーとして書き出すことができます。これには、ファイルプロデューサーエンドポイントで **doneFileName** オプションを使用できます。

```
.to("file:bar?doneFileName=done");
```

は単に、target ファイルと同じディレクトリーに **done** という名前のファイルを作成します。

ただし、ターゲットファイルごとに 1 つのファイルを配置するのが一般的です。これは、相関が 1:1 であることを意味します。これを実行するには、**doneFileName** オプションで動的プレースホルダーを使用する必要があります。現在 Camel は、file:name および **file:name.noext** の 2 つの動的トークンをサポートします。これは、**${ }** で囲む必要があります。

```
.to("file:bar?doneFileName=done-${file:name}");
```

たとえば、ターゲットファイルがターゲットファイルと同じディレクトリーに **foo.txt** の場合は、**done-foo.txt** という名前のファイルを作成します。

```
.to("file:bar?doneFileName=${file:name}.done");
```

たとえば、ターゲットファイルがターゲットファイルと同じディレクトリーに **foo.txt.done** であった場合、**foo.txt**.done という名前のファイルを作成します。

```
.to("file:bar?doneFileName=${file:name.noext}.done");
```

たとえば、ターゲットファイルが foo.txt の場合に、ターゲットファイルと同じディレクトリーに foo.done という名前のファイルを作成します。

ディレクトリーから読み取り、別のディレクトリーに書き込みます。

```
from("file://inputdir/?delete=true").to("file://outputdir")
```

オーバールール動的名を使用したディレクトリーからの読み取り、別のディレクトリーへの書き込み

```
from("file://inputdir/?delete=true").to("file://outputdir?overruleFile=copy-of-${file:name}")
```

ディレクトリーでリッスンし、そこにドロップされる各ファイルのメッセージを作成します。コンテンツを outputdir にコピーし、inputdir のファイルを削除します。

ディレクトリーから再帰的に読み取り、別のディレクトリーへの書き込み

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

ディレクトリーでリッスンし、そこにドロップされる各ファイルのメッセージを作成します。コンテンツを outputdir にコピーし、inputdir のファイルを削除します。はサブディレクトリーに再帰的にスキャンします。は、サブディレクトリーを含む inputdir と同じディレクトリー構造のファイルを outputdir に配置します。

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

以下の出力レイアウトが生成されます。

```
outputdir/foo.txt
outputdir/sub/bar.txt
```

フラット化の使用

ファイルを同じディレクトリーの outputdir ディレクトリーに保存する場合は、ソースディレクトリーのレイアウトを無視する（たとえば、パスをフラット化するなど）、ファイルプロデューサー側で flatten=true オプションを追加するだけです。

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir?flatten=true")
```

以下の出力レイアウトが生成されます。

```
outputdir/foo.txt
outputdir/bar.txt
```

ディレクトリーおよびデフォルトの移動操作からの読み取り

Apache Camel はデフォルトで、処理されたファイルを、ファイルが消費されたディレクトリー内の .camel サブディレクトリーに移動します。

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

は、以前と同様にレイアウト に影響します。

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

after

```
inputdir/.camel/foo.txt
inputdir/sub/.camel/bar.txt
outputdir/foo.txt
outputdir/sub/bar.txt
```

ディレクトリーから読み取り、JAVA でメッセージを処理します。

```
from("file://inputdir/").process(new Processor() {
  public void process(Exchange exchange) throws Exception {
    Object body = exchange.getIn().getBody();
    // do some business logic with the input body
  }
});
```

ボディーは、inputdir ディレクトリーにドロップされたばかりのファイルを参照する File オブジェクトです。

ディレクトリーからファイルを読み込み、内容を JMS キューに送信します。

```
from("file://inputdir/").convertBodyTo(String.class).to("jms:test.queue")
```

デフォルトでは、File エンドポイントは File オブジェクトが含まれるFileMessage をボディーとして送信します。これを JMS コンポーネントに直接送信すると、JMS メッセージには File オブジェクトのみが含まれますが、コンテンツは含まれません。File を String に変換することで、メッセージにはファイルの内容（おそらく必要なもの）が含まれます。

Spring DSL を使用した上記のルート：

```
<route>
  <from uri="file://inputdir/"/>
  <convertBodyTo type="java.lang.String"/>
  <to uri="jms:test.queue"/>
</route>
```

ファイルへの書き込み

Apache Camel は当然ながらファイルを書き込むこともできます。つまり、ファイルが生成されます。以下の例では、ディレクトリーに書き込まれる前に処理する SEDA キューに関するレポートの一部を受け取ります。

```
public void testToFile() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedFileExists("target/test-reports/report.txt");

    template.sendBody("direct:reports", "This is a great report");

    assertMockEndpointsSatisfied();
}

protected JndiRegistry createRegistry() throws Exception {
    // bind our processor in the registry with the given id
    JndiRegistry reg = super.createRegistry();
    reg.bind("processReport", new ProcessReport());
    return reg;
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // the reports from the seda queue is processed by our processor
            // before they are written to files in the target/reports directory
            from("direct:reports").processRef("processReport").to("file://target/test-reports",
"mock:result");
        }
    };
}

private static class ProcessReport implements Processor {

    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        // do some business logic here

        // set the output to the file
        exchange.getOut().setBody(body);

        // set the output filename using java code logic, notice that this is done by setting
        // a special header property of the out exchange
        exchange.getOut().setHeader(Exchange.FILE_NAME, "report.txt");
    }

}
```

EXCHANGE.FILE_NAME を使用してサブディレクトリーに書き込みます。

単一ルートを使用すると、任意の数のサブディレクトリーにファイルを書き込むことができます。以下のようなルート設定がある場合：

```
<route>
  <from uri="bean:myBean"/>
```

```
  <to uri="file:/rootDirectory"/>
</route>
```

myBean をヘッダー Exchange.FILE_NAME を以下のような値に設定できます。

```
Exchange.FILE_NAME = hello.txt => /rootDirectory/hello.txt
Exchange.FILE_NAME = foo/bye.txt => /rootDirectory/foo/bye.txt
```

これにより、1 つのルートを使用してファイルを複数の宛先に書き込むことができます。

一時ディレクトリーを使用した最終的な宛先へのファイルの書き込み

時折、宛先ディレクトリーに関連する一部のディレクトリーにファイルを一時的に書き込む必要があります。このような状況は、通常、書き込み先のディレクトリーから一部のフィルター機能を持つ外部プロセスを読み取ると発生します。以下のファイルの例では、/var/myapp/filesInProgress ディレクトリーに書き込まれ、データ転送が完了すると、アトミックで /var/myapp/finalDirectory ディレクトリーに移動します。

```
from("direct:start").
  to("file:///var/myapp/finalDirectory?tempPrefix=/../filesInProgress/");
```

ファイル名の式の使用

この例では、現在の日付をサブフォルダー名として使用して、消費したファイルをバックアップフォルダーに移動します。

```
from("file://inbox?move=backup/${date:now:yyyyMMdd}/${file:name}").to("...");
```

その他のサンプルについては、File Language を参照してください。

同じファイルを複数回読み取るのを回避（べき等コンシューマー）

Apache Camel はコンポーネント内 で直接べき等コンシューマー をサポートするため、すでに処理されたファイルを省略します。この機能は、idempotent=true オプションを設定することで有効にできます。

```
from("file://inbox?idempotent=true").to("...");
```

Camel は絶対ファイル名をべき等キーとして使用し、重複ファイルを検出します。Camel 2.11 以降では、IdempotentKey オプションの式を使用してこのキーをカスタマイズできます。たとえば、名前とファイルサイズの両方をキーとして使用するには、以下を実行します。

```
<route>
  <from uri="file://inbox?idempotent=true&dempotentKey=${file:name}-${file:size}"/>
  <to uri="bean:processInbox"/>
</route>
```

デフォルトでは、Apache Camel は消費されたファイルを追跡するためにインメモリーベースのストアを使用し、最大 1000 エントリーを保持する最も最近使用されたキャッシュを使用します。指定された ID を持つレジストリー内の Bean の参照を示すために、値に # 記号を使用して、このストアの独自の実装をプラグインでき ます。

```
<!-- define our store as a plain spring bean -->
<bean id="myStore" class="com.mycompany.MyIdempotentStore"/>

<route>
  <from uri="file://inbox?idempotent=true&dempotentRepository=#myStore"/>
  <to uri="bean:processInbox"/>
</route>
```

Apache Camel は、以前に使用されたためファイルをスキップすると DEBUG レベルでログを記録します。

```
DEBUG FileConsumer is idempotent and the file has been consumed before. Will skip this
file: target\idempotent\report.txt
```

ファイルベースのべき等リポジトリーの使用

このセクションでは、デフォルトとして使用されるメモリー内ベースではなく、ファイルベースのべき等リポジトリー org.apache.camel.processor.idempotent.FileIdempotentRepository を使用します。このリポジトリーは、ファイルリポジトリーを読み取らないために 1 つのレベルキャッシュを使用します。これは、file リポジトリーを使用して 1 レベルのキャッシュの内容を保存します。これにより、リポジトリーはサーバーの再起動後も維持されます。これは、起動時にファイルの内容を 1 レベルのキャッシュに読み込みます。ファイル構造は、キーを ファイルの別々の行に保存するため、非常に簡単です。デフォルトでは、ファイルストアのサイズ制限は 1mb で、ファイルが大きくなると、Apache Camel はファイルストアを切り捨て、1 番目のレベルのキャッシュを新しい空のファイルにフラッシュしてコンテンツを再ビルドします。

Spring XML を使用してファイルべき等リポジトリーを作成し、\# 記号を使用して idempotentRepository でリポジトリーを使用するようにファイルコンシューマーを定義し、レジストリー検索を示します。

```
<!-- this is our file based idempotent store configured to use the .filestore.dat as file -->
<bean id="fileStore"
class="org.apache.camel.processor.idempotent.FileIdempotentRepository">
   <!-- the filename for the store -->
   <property name="fileStore" value="target/fileidempotent/.filestore.dat"/>
   <!-- the max filesize in bytes for the file. Apache Camel will trunk and flush the cache
       if the file gets bigger -->
   <property name="maxFileStoreSize" value="512000"/>
   <!-- the number of elements in our store -->
   <property name="cacheSize" value="250"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
   <route>
      <from uri="file://target/fileidempotent/?
idempotent=true&dempotentRepository=#fileStore&ove=done/${file:name}"/>
```

```
        <to uri="mock:result"/>
    </route>
</camelContext>
```

JPA ベースのべき等リポジトリーの使用

このセクションでは、デフォルトとして使用されるインメモリーベースの代わりに、JPA ベースのべき等リポジトリーを使用します。

最初に、org.apache.camel.processor.idempotent.jpa.MessageProcessed クラスをモデルとして使用する必要がある META-INF/persistence.xml に persistence-unit が必要です。

```
<persistence-unit name="idempotentDb" transaction-type="RESOURCE_LOCAL">
  <class>org.apache.camel.processor.idempotent.jpa.MessageProcessed</class>

  <properties>
    <property name="openjpa.ConnectionURL"
value="jdbc:derby:target/idempotentTest;create=true"/>
    <property name="openjpa.ConnectionDriverName"
value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema"/>
    <property name="openjpa.Log" value="DefaultLevel=WARN, Tool=INFO"/>
  </properties>
</persistence-unit>
```

次に、Spring XML ファイルで Spring jpaTemplate を設定する必要があります。

```
<!-- this is standard spring JPA configuration -->
<bean id="jpaTemplate" class="org.springframework.orm.jpa.JpaTemplate">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <!-- we use idempotentDB as the persitence unit name defined in the persistence.xml file -->
    <property name="persistenceUnitName" value="idempotentDb"/>
</bean>
```

最後に、Spring XML ファイルで JPA idempotent リポジトリーを作成することもできます。

```
<!-- we define our jpa based idempotent repository we want to use in the file consumer -->
<bean id="jpaStore"
class="org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository">
    <!-- Here we refer to the spring jpaTemplate -->
    <constructor-arg index="0" ref="jpaTemplate"/>
    <!-- This 2nd parameter is the name  (= a cateogry name).
```

```
      You can have different repositories with different names -->
    <constructor-arg index="1" value="FileConsumer"/>
</bean>
```

次に、idempotentRepository オプションと # 構文を使用して、ファイルコンシューマーエンドポイントで jpaStore Bean のみを参照する必要があります。

```
<route>
  <from uri="file://inbox?idempotent=true&dempotentRepository=#jpaStore"/>
  <to uri="bean:processInbox"/>
</route>
```

ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER を使用するフィルター

Apache Camel はプラグ可能なフィルターリングストラテジーをサポートします。その後、このようなフィルターでエンドポイントを設定し、処理中の特定のファイルを省略できます。

この例では、ファイル名の skip で始まるファイルを スキップ する独自のフィルターを構築します。

```
public class MyFileFilter implements GenericFileFilter {
    public boolean accept(GenericFile pathname) {
        // we dont accept any files starting with skip in the name
        return !pathname.getFileName().startsWith("skip");
    }
}
```

次に、filter 属性を使用して、Spring XML ファイルで定義したフィルター(# 表記を使用)を参照するルートを設定できます。

```
<!-- define our filter as a plain spring bean -->
<bean id="myFilter" class="com.mycompany.MyFileFilter"/>

<route>
  <from uri="file://inbox?filter=#myFilter"/>
  <to uri="bean:processInbox"/>
</route>
```

ANT パスマッチャーを使用したフィルターリング

ANT パスマッチャーは、camel-spring jar で追加設定なしで提供されます。そのため、Maven を使用している場合は camel-spring に依存する必要があります。理由は、Spring の AntPathMatcher を使用して実際のマッチングを行うことです。

ファイルパスは、以下のルールと一致します。

- ? 1 文字に一致します。
- * ゼロ以上の文字に一致します。
- ** パス内の 0 個以上のディレクトリーに一致します。

以下の例は、その使用方法を示しています。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
   <template id="camelTemplate"/>

   <!-- use myFilter as filter to allow setting ANT paths for which files to scan for -->
   <endpoint id="myFileEndpoint" uri="file://target/antpathmatcher?
recursive=true&ilter=#myAntFilter"/>

   <route>
      <from ref="myFileEndpoint"/>
      <to uri="mock:result"/>
   </route>
</camelContext>

<!-- we use the antpath file filter to use ant paths for includes and exlucde -->
<bean id="myAntFilter"
class="org.apache.camel.component.file.AntPathMatcherGenericFileFilter">
   <!-- include and file in the subfolder that has day in the name -->
   <property name="includes" value="**/subfolder/**/*day*"/>
   <!-- exclude all files with bad in name or .xml files. Use comma to seperate multiple
excludes -->
   <property name="excludes" value="**/*bad*,**/*.xml"/>
</bean>
```

COMPARATOR を使用したソート

Apache Camel はプラグ可能なソートストラテジーをサポートします。Java の java.util.Comparator でビルドを使用するこのストラテジーです。その後、このようなコンパレーターでエンドポイントを設定し、Apache Camel でファイルをソートしてから処理できます。

この例では、ファイル名でソートする独自のコンパレーターを構築します。

```
public class MyFileSorter implements Comparator<GenericFile> {
    public int compare(GenericFile o1, GenericFile o2) {
        return o1.getFileName().compareToIgnoreCase(o2.getFileName());
    }
}
```

次に、sorter オプションを使用してルートを設定し、Spring XML ファイルで定義したソーター (mySorter)を参照できます。

```
 <!-- define our sorter as a plain spring bean -->
 <bean id="mySorter" class="com.mycompany.MyFileSorter"/>

<route>
  <from uri="file://inbox?sorter=#mySorter"/>
  <to uri="bean:processInbox"/>
</route>
```

URI オプションは # 構文を使用して BEAN を参照できます。

Spring DSL ルートでは、id の前に # を付けることで、レジストリー 内の Bean を参照できます。そのため、sorter=#mySorter を記述すると、Apache Camel に対して ID mySorter の Bean の Bean を検索するよう指示されます。

SORTBY を使用したソート

Apache Camel はプラグ可能なソートストラテジーをサポートします。このストラテジーでは、File 言語 を使用してソートを設定します。sortBy オプションは以下のように設定されます。

```
sortBy=group 1;group 2;group 3;...
```

各グループはセミコロンで区切ります。簡単な状況では、1 つのグループのみを使用するので、簡単な例を以下に示します。

```
sortBy=file:name
```

ファイル名でソートされ、グループの前に reverse: を付けることで順序を元に戻すことができるため、ソートは Z..A になります。

```
sortBy=reverse:file:name
```

File Language の完全なパワーがあるので、他のいくつかのパラメーターを使用できるので、ファイルサイズでソートしたい場合は以下を行います。

```
sortBy=file:length
```

文字列比較には ignoreCase: を使用してケースを無視するように設定できます。したがって、ファイル名のソートを使用し、ケースを無視する場合は、以下を行います。

```
sortBy=ignoreCase:file:name
```

ignore case と reverse を組み合わせることができますが、最初に逆を指定する必要があります。

```
sortBy=reverse:ignoreCase:file:name
```

以下の例では、最後に変更されたファイルで、以下を行います。

```
sortBy=file:modified
```

次に、2 番目のオプションとして名前でグループ化するため、同じ変更を持つファイルを名前でソートします。

```
sortBy=file:modified;file:name
```

ここで問題が存在しますが、これを見つけることができますか ?ファイルの変更されたタイムスタンプはミリ秒単位であるため問題になりすぎますが、日付のみで、その後にサブグループを名前でソートしたい場合はどうすればよいですか ?また、ファイル言語の本当の力が あれば、パターンをサポートする date コマンドを使用できます。そのため、以下のように解決できます。

```
sortBy=date:file:yyyyMMdd;file:name
```

Yeah は非常に強力で、グループごとに reverse を使用することもできます。そのため、ファイル名を元に戻すことができます。

```
sortBy=date:file:yyyyMMdd;reverse:file:name
```

GENERICFILEPROCESSSTRATEGY の使用

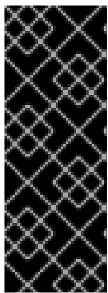
processStrategy オプションは、カスタムの GenericFileProcessStrategy を使用できます。これにより、独自の 開始、コミット、および ロールバック ロジックを実装できます。たとえば、システムが

使用するフォルダーにファイルを書き込むことを前提とします。ただし、別の準備完了 ファイルも書き込まれる前に、このファイルの使用を開始することはできません。

*そのため、独自の **GenericFileProcessStrategy** を実装することで、以下のように実装できます。*

- ***begin** （）メソッドでは、特別な準備 済みファイルが存在するかどうかをテストできます。**begin** メソッドは ブール値 を返し、ファイルを使用できるかどうかを示します。*

- ***commit** （）メソッドでは、実際のファイルを移動し、準備完了 ファイルを削除することもできます。*



使用における重要 CONSUMER.BRIDGEERRORHANDLER

***consumer.bridgeErrorHandler** を使用する場合、インターセプターのOnCompletion は適用さ れません。Exchange は **Camel** Error Handler によって直接処理され、インターセプターなどの以前のアクションがCompletion のアクションを実行することを許可しません。*

デバッグロギング

*このコンポーネントにはログレベル **TRACE** があり、問題がある場合に役立ちます。*

以下も参照してください。

- *ファイル言語*

- ***FTP2***

第50章 FLATPACK

FLATPACK コンポーネント

Flatpack コンポーネントは、FlatPack ライブラリー を使用して固定幅と区切られたファイル解析をサポートします。注記： このコンポーネントは、flatpack ファイルからオブジェクトモデルへの消費のみをサポートします。オブジェクトモデルから flatpack 形式に書き込みを行うことは(yet)できません。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-flatpack</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
flatpack:[delim|fixed]:flatPackConfig.pzmap.xml[?options]
```

または、設定ファイルのない区切りファイルハンドラーには、以下を使用します。

```
flatpack:someName[?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	説明
delimiter	,	区切りファイルのデフォルトの文字区切り文字。
textQualifier	"	区切りファイルのテキスト修飾子。

ignoreFirstRecord	**true**	最初の行が区切られたファイル（列ヘッダー用）で無視されるかどうか。
splitRows	**true**	Apache Camel 1.5 では、コンポーネントは各行を 1 つずつ、またはコンテンツ全体を一度に処理できます。
allowShortLines	**false**	*Camel 2.9.3:* 行を予想よりも短くし、追加の文字を無視します。
ignoreExtraColumns	**false**	*Camel 2.9.3:* 行を予想よりも長くし、追加文字を無視します。

例

- ***flatpack:fixed:foo.pzmap.xml は、foo.pzmap.xml ファイル設定を使用して固定幅エンドポイントを作成します。***

- ***flatpack:delim:bar.pzmap.xml ファイル設定を使用して、bar.pzmap.xml ファイル設定を使用して区切られたエンドポイントを作成します。***

- ***flatpack:foo は、ファイル設定なしで foo という区切りエンドポイントを作成します。***

メッセージヘッダー

Apache Camel は以下のヘッダーを IN メッセージに保存します。

ヘッダー	説明
camelFlatpackCounter	現在の行インデックス。**splitRows=false** の場合、カウンターは行の合計数です。

メッセージボディー

コンポーネントは、java.util.Map または java.util.List のコンバーターを持つ org.apache.camel.component.flatpack.DataSetList オブジェクトとして IN メッセージのデータを提供します。一度に 1 行を処理する場合は、通常 マップ が必要です(splitRows=true)。コンテンツ全体

に List を使用します(splitRows=false)。リストの各要素は Map です。各 Map には、列名とそれに対応する値のキーが含まれます。

たとえば、以下の例から firstname を取得するには、次のコマンドを実行します。

```
Map row = exchange.getIn().getBody(Map.class);
String firstName = row.get("FIRSTNAME");
```

ただし、これを常に List として取得することもできます(splitRows=trueも同様です)。同じ例：

```
List data = exchange.getIn().getBody(List.class);
Map row = (Map)data.get(0);
String firstName = row.get("FIRSTNAME");
```

ヘッダーおよびトレイルレコード

Flatpack のヘッダーおよびトレイル概念がサポートされます。ただし、固定されたレコード ID を使用する 必要があり ます。

- ヘッダー レコードのヘッダー（小文字でなければなりません）
- trailer レコードのトレイル（小文字でなければなりません）

以下の例は、ヘッダーとトレイルがあるこのファクトを示しています。必要でない場合は、それらのいずれかまたは両方を省略することができます。

```
<RECORD id="header" startPosition="1" endPosition="3" indicator="HBT">
   <COLUMN name="INDICATOR" length="3"/>
   <COLUMN name="DATE" length="8"/>
</RECORD>

<COLUMN name="FIRSTNAME" length="35" />
<COLUMN name="LASTNAME" length="35" />
<COLUMN name="ADDRESS" length="100" />
<COLUMN name="CITY" length="100" />
<COLUMN name="STATE" length="2" />
<COLUMN name="ZIP" length="5" />

<RECORD id="trailer" startPosition="1" endPosition="3" indicator="FBT">
```

```
    <COLUMN name="INDICATOR" length="3"/>
    <COLUMN name="STATUS" length="7"/>
  </RECORD>
```

エンドポイントの使用

一般的なユースケースでは、別のルートでさらなる処理のためにファイルをこのエンドポイントに送信します。以下に例を示します。

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="file://someDirectory"/>
    <to uri="flatpack:foo"/>
  </route>

  <route>
    <from uri="flatpack:foo"/>
    ...
  </route>
</camelContext>
```

Bean 統合を容易にするために、作成された各メッセージのペイロードを マップ に変換することもできます。

第51章 FOP

FOP コンポーネント

Camel 2.10 以降で利用可能

FOP コンポーネントを使用すると、Apache FOP を使用してメッセージを異なる出力形式にレンダリングできます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-fop</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
fop://outputFormat?[options]
```

出力形式

プライマリー出力形式は PDF ですが、他の出力形式もサポートされます。http://xmlgraphics.apache.org/fop/0.95/output.html

名前	出力形式	説明
PDF	**application/pdf**	移植可能なドキュメント形式
PS	**application/postscript**	Adobe Postscript
PCL	**application/x-pcl**	プリンター制御言語
PNG	**image/png**	PNG イメージ

JPEG	**image/jpeg**	JPEG イメージ
SVG	**image/svg+xml**	スケーラブルなベクトルグラフ
XML	**application/X-fop-areatree**	エリアツリー表現
MIF	**application/mif**	FrameMaker's MIF
RTF	**application/rtf**	リッチテキスト形式
TXT	**text/plain**	テキスト

有効な出力形式の完全なリストは、こちらを参照してください。

エンドポイントオプション

name	デフォルト値	description
outputFormat		上記の表を参照してください。
userConfigURL	none	以下の 構造 を持つ設定ファイルの場所。Camel 2.12 以降では、ファイルはデフォルトでクラスパスから読み込まれます。**file:** または **classpath:** を接頭辞として使用し、ファイルまたはクラスパスからリソースを読み込むことができます。以前のリリースでは、ファイルは常にファイルシステムからロードされていました。
fopFactory		**org.apache.fop.apps.FopFactory** のカスタム設定またはカスタム実装を使用できます。

メッセージ操作

name	デフォルト値	description

CamelFop.Output.Format		そのメッセージの出力形式を上書きします。
CamelFop.Encrypt.userPassword		PDF ユーザーのパスワード
CamelFop.Encrypt.ownerPassword		PDF 所有者の乗車語
CamelFop.Encrypt.allowPrint	**true**	PDF の印刷が可能
CamelFop.Encrypt.allowCopyContent	**true**	PDF の内容のコピーを許可します。
CamelFop.Encrypt.allowEditContent	**true**	PDF の内容を編集できます。
CamelFop.Encrypt.allowEditAnnotations	**true**	PDF のアノテーションを編集可能
CamelFop.Render.producer	Apache FOP	ドキュメントを生成するシステム/ソフトウェアのメタデータ要素
CamelFop.Render.creator		ドキュメントを作成したユーザーのメタデータ要素
CamelFop.Render.creationDate		作成日
CamelFop.Render.author		ドキュメントのコンテンツの作成
CamelFop.Render.title		ドキュメントのタイトル
CamelFop.Render.subject		ドキュメントの主体
CamelFop.Render.keywords		本書に適用されるキーワードのセット

例

以下は、XML データおよび XSLT テンプレートから PDF をレンダリングし、ターゲットフォルダーに PDF ファイルを保存するルートの例です。

```
from("file:source/data/xml")
    .to("xslt:xslt/template.xsl")
```

```
.to("fop:application/pdf")
.to("file:target/data");
```

第52章 FREEMARKER

FREEMARKER

freemarker: コンポーネントを使用すると、FreeMarker テンプレートを使用してメッセージを処理できます。これは、Templating を使用してリクエストの応答を生成する場合に理想的です。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-freemarker</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
freemarker:templateName[?options]
```

templateName は、呼び出すテンプレートのクラスパスローカル URI、またはリモートテンプレートの完全な URL （例： file://folder/myfile.ftl）に置き換えます。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	デフォルト	説明
allowContextMapAll (producer)	**false**	コンテキストマップが前詳細へのアクセスを許可するかどうかを設定します。デフォルトでは、メッセージの本文とヘッダーにのみアクセスできます。このオプションは、現在の Exchange および CamelContext へのフルアクセスに対して有効にできます。これを行うと、CamelContext API の全機能へのアクセスが開かれるため、潜在的なセキュリティーリスクが発生します。

allowTemplateFromHeader (producer)	**false**	ヘッダーのリソーステンプレートの使用を許可するかどうか (デフォルトは false)。**このオプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼されていないコンテンツやユーザー派生コンテンツが含まれている場合、これは最終的にエンドアプリケーションの信頼性および完全性に影響を与える可能性があるため、このオプションは注意して使用してください。**
contentCache	**true**	読み込み時のリソースコンテンツのキャッシュ。注記： **Camel 2.9** でキャッシュされたリソースコンテンツは、エンドポイントの **clearContentCache** 操作を使用して JMX 経由でクリアできます。
encoding	**null**	リソースコンテンツの文字エンコーディング。
templateUpdateDelay	**5**	*Camel 2.9:* 読み込んだテンプレートリソースがキャッシュに留まる秒数。

FREEMARKER コンテキスト

Apache Camel は FreeMarker コンテキストで交換情報を提供します（マップのみ）。Exchange は以下のように転送されます。

キー	値
exchange	**Exchange** 自体。
exchange.properties	**Exchange** プロパティー。
ヘッダー	In メッセージのヘッダー。
camelContext	Camel コンテキスト。
request	In メッセージ。
ボディー	In メッセージのボディー。

response	Out メッセージ(InOut メッセージ交換パターンにのみ有効)。

Camel 2.14 以降では、以下のようにキー CamelFreemarkerDataModel を使用して、メッセージヘッダーにカスタム FreeMarker コンテキストを設定できます。

```
Map<String, Object> variableMap = new HashMap<String, Object>();
variableMap.put("headers", headersMap);
variableMap.put("body", "Monday");
variableMap.put("exchange", exchange);
exchange.getIn().setHeader("CamelFreemarkerDataModel", variableMap);
```

ホットリロード

FreeMarker テンプレートリソースは、デフォルトでは、ファイルとクラスパスリソース（展開形式の jar）の両方でホットリロードでき ません。contentCache=false を設定すると、Apache Camel はリソースをキャッシュせず、ホットリロードが有効になります。このシナリオは開発中に使用できます。

動的テンプレート

Camel 2 .1 Camel では利用可能な 2 つの ヘッダーで、テンプレートまたはテンプレートコンテンツ自体の異なるリソースの場所を定義できます。これらのヘッダーのいずれかが設定されている場合、Camel は設定されたエンドポイントでこれを使用します。これにより、ランタイム時に動的テンプレートを指定できます。

ヘッダー	タイプ	説明	サポートバージョン
FreemarkerConstants.FREEMARKER_RESOURCE	**org.springframework.core.io.Resource**	テンプレートリソース	<= 1.6.2, <= 2.1
FreemarkerConstants.FREEMARKER_RESOURCE_URI	**String**	設定されたエンドポイントの代わりに使用するテンプレートリソースの URI。	>= 2.1
FreemarkerConstants.FREEMARKER_TEMPLATE	**String**	設定されたエンドポイントの代わりに使用するテンプレート。	>= 2.1

サンプル

たとえば、以下のようなルートを定義できます。

```
from("activemq:My.Queue").
  to("freemarker:com/acme/MyResponse.ftl");
```

FreeMarker テンプレートを使用して InOut メッセージエクスチェンジへの応答(JMSReplyTo ヘッダーがある場所)を形成するには、以下を実行します。

InOnly エクスチェンジを処理する場合は、FreeMarker テンプレートを使用してメッセージを別のエンドポイントに送信する前に変換できます。

```
from("activemq:My.Queue").
  to(ExchangePattern.InOut,"freemarker:com/acme/MyResponse.ftl").
  to("activemq:Another.Queue");
```

また、コンテンツキャッシュを無効にするには（たとえば、.ftl テンプレートをホットリロードする必要がある開発使用の場合）：

```
from("activemq:My.Queue").
  to(ExchangePattern.InOut,"freemarker:com/acme/MyResponse.ftl?contentCache=false").
  to("activemq:Another.Queue");
```

ファイルベースのリソースの場合：

```
from("activemq:My.Queue").
  to(ExchangePattern.InOut,"freemarker:file://myfolder/MyResponse.ftl?contentCache=false").
  to("activemq:Another.Queue");
```

Camel 2.1 では、以下のように、コンポーネントがヘッダーを介して動的に使用するテンプレートを指定できます。

```
from("direct:in").

setHeader(FreemarkerConstants.FREEMARKER_RESOURCE_URI).constant("path/to/my/temp
late.ftl").
  to("freemarker:dummy?allowTemplateFromHeader=true");
```

警告

allowTemplateFromHeader オプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼できないコンテンツまたはユーザー派生コンテンツが含まれる場合、これは最終的に、エンドアプリケーションの確実性と整合性に及ぼす可能性があるため、このオプションを使用してください。

電子メールのサンプル

この例では、FreeMarker テンプレートを使用し、注文確認メールを使用します。メールテンプレートは、以下のように FreeMarker に記載されています。

```
Dear ${headers.lastName}, ${headers.firstName}

Thanks for the order of ${headers.item}.

Regards Camel Riders Bookstore
${body}
```

Java コード：

```
private Exchange createLetter() {
    Exchange exchange = context.getEndpoint("direct:a").createExchange();
    Message msg = exchange.getIn();
    msg.setHeader("firstName", "Claus");
    msg.setHeader("lastName", "Ibsen");
    msg.setHeader("item", "Camel in Action");
    msg.setBody("PS: Next beer is on me, James");
    return exchange;
}

@Test
public void testFreemarkerLetter() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedBodiesReceived("Dear Ibsen, Claus\n\nThanks for the order of Camel in
Action."
        + "\n\nRegards Camel Riders Bookstore\nPS: Next beer is on me, James");

    template.send("direct:a", createLetter());

    mock.assertIsSatisfied();
}
```

```
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:a")
                .to("freemarker:org/apache/camel/component/freemarker/letter.ftl")
                .to("mock:result");
        }
    };
}
```

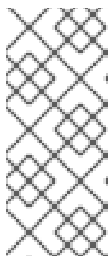
第53章 FTP2

FTP/SFTP コンポーネント

このコンポーネントは、FTP プロトコルおよび SFTP プロトコルを介してリモートファイルシステムへのアクセスを提供します。

リモート FTP サーバーからの消費

ファイルの使用に関する詳細については、以下のファイルを使用する際の Default というセクションを必ずお読みください。



注記

絶対パスはサポートされていません。Camel 2.16 は、directoryname から先頭のスラッシュをすべてトリミングすることにより、絶対パスを相対に変換します。WARN メッセージがログに出力されます。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
ftp://[username@]hostname[:port]/directoryname[?options]
sftp://[username@]hostname[:port]/directoryname[?options]
ftps://[username@]hostname[:port]/directoryname[?options]
```

directoryname は基礎となるディレクトリーを表します。ネストされたフォルダーを含めることができます。ディレクトリー名は相対パスです。絶対パスはサポートされていません。相対パスには、ネストされたフォルダーを含めることができます（例： /inbox/usCamel 2.16 では、autoCreate オプションがサポートされます。コンシューマーが起動すると（ポーリングがスケジュールされる前）、指定されたディレクトリーが自動的に作成されます。autoCreate のデフォルト値は true です。

ユーザー 名を指定しないと、パスワードなしで 匿名 ログインが試行されます。ポート 番号が指定されていない場合、Apache Camel はプロトコルに従ってデフォルト値を提供します(ftp = 21、sftp = 22、ftps = 21)。

このコンポーネントは、実際の FTP 作業に 2 つの異なるライブラリーを使用します。FTP および FTPS は Apache Commons Net を使用し、SFTP は JCraft JSCH を使用します。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

URI オプション

以下のオプションは FTP コンポーネント専用です。

名前	デフォルト値	説明
username	**null**	リモートファイル systen へのログインに使用するユーザー名を指定します。
password	**null**	リモートファイルシステムへのログインに使用するパスワードを指定します。
アカウント	**null**	**Camel 2.15.2:** リモート FTP サーバーへのログインに使用するアカウントを指定します(FTP および FTP Secure のみ)。
binary	**false**	ファイル転送モードを BINARY または ASCII で指定します。デフォルトは ASCII (**false**)です。
disconnect	**false**	**Camel 2.2:** 使用直後にリモート FTP サーバーから切断するかどうか。コンシューマーとプロデューサーの両方に使用できます。disconnect は、FTP サーバーへの現在の接続を切断するだけです。停止するコンシューマーがある場合は、代わりにコンシューマー/ルートを停止する必要があります。

localWorkDirectory	**null**	使用する場合、ローカルの作業ディレクトリーを使用して、リモートファイルのコンテンツをローカルファイルに直接保存し、コンテンツがメモリーに読み込まれないようにできます。これは、非常に大きなリモートファイルを使用している場合に、メモリーを節約するために役立ちます。詳細は、こちらを参照してください。
passiveMode	**false**	**FTP のみ**: パッシブモード接続を使用するかどうかを指定します。デフォルトはアクティブモード **{false**）です。
securityProtocol	**TLS**	**FTPS のみ：** 基礎となるセキュリティープロトコルを設定します。**TLS**: Transport Layer Security **SSL**: Secure Sockets Layer の値を定義します。
disableSecureDataChannelDefaults	**false**	**Camel 2.4: FTPS のみ**: セキュアなデータ転送を使用する場合に **execPbsz** および **execProt** のデフォルト値の使用を無効にするかどうか。このオプションを **true** に設定するには、**execPbsz** オプションおよび **execProt** オプションを明示的に設定する必要があります。
ダウンロード	**true**	**Camel 2.11:** FTP コンシューマーがファイルをダウンロードするかどうか。このオプションを **false** に設定すると、メッセージボディーは **null** になりますが、コンシューマーはファイル名、ファイルサイズなどのファイルの詳細が含まれる Camel Exchange をトリガーします。ファイルがダウンロードされないだけです。

streamDownload	false	Camel 2.11: コンシューマーが前もってファイル全体をダウンロードするか、デフォルトの動作では、インメモリーアレイではなくリモートリソースから InputStream エドを amel Exchange の本文として渡す必要があるかどうかは無視されます。ダウンロードが false r が localWorkDirectory の場合は無視されます。このオプションは大きなリモートファイルで作業する場合に便利です。
execProt	**null**	**Camel 2.4: FTPS のみ**: セキュアなデータチャネルのデフォルトが無効でない場合は、デフォルトでオプション **P** を使用します。以下の値が使用できます。 ● **C**: Clear ● **s**: Safe (TLS プロトコルのみ) ● **e**: 機密(TLS プロトコルのみ) ● **P**: プライベート
execPbsz	**null**	**Camel 2.4: FTPS のみ**: このオプションは、セキュアなデータチャネルのバッファーサイズを指定します。**useSecureDataChannel** オプションが有効で、このオプションが明示的に設定されていない場合、値 **0** が使用されます。
isImplicit	**false**	**FTPS のみ - セキュリティーモードを設定します** (implicit/explicit)。デフォルトは明示的です(**false**)。
knownHostsFile	**null**	**SFTP のみ： known_hosts** ファイルを設定して、SFTP エンドポイントがホストキーの検証を実行できるようにします。
knownHostsUri	**null**	**SFTP のみ：Camel 2.11.1: known_hosts** ファイル（デフォルトではクラスパスからロード）を設定し、SFTP エンドポイントがホストキーの検証を実行できるようにします。

keyPair	**null**	**SFTP のみ：Camel 2.12.0:** SSH 公開鍵認証用の Java KeyPair を設定し、DSA キーまたは RSA 鍵をサポートします。
privateKeyFile	**null**	**SFTP のみ：** SFTP エンドポイントが秘密鍵を検証できるように秘密鍵ファイルを設定します。
privateKeyUri	**null**	**SFTP のみ：Camel 2.11.1:** 秘密鍵ファイル（デフォルトではクラスパスからロードされる）を SFTP エンドポイントが秘密鍵の検証を実行できるように設定します。
privateKey	**null**	**SFTP のみ：Camel 2.11.1:** SFTP エンドポイントが秘密鍵を検証することができるように、秘密鍵を byte[] に設定します。
privateKeyFilePassphrase	**null**	**SFTP のみ：** 秘密鍵ファイルのパスフレーズを SFTP エンドポイントが秘密鍵の検証を実行できるように設定します。
privateKeyPassphrase	**null**	**SFTP のみ：Camel 2.11.1:** SFTP エンドポイントが秘密鍵の検証を実行できるように、秘密鍵ファイルのパスフレーズを設定します。
preferredAuthentications	**null**	**SFTP のみ：Camel 2.10.7、2.11.2、2.12.0:** SFTP エンドポイントが使用される優先認証を設定します。たとえば、password,publickey などです。指定しない場合は、JSCH のデフォルトのリストが使用されます。
暗号化	**null**	**Camel 2.8.2、2.9: SFTP は、優先順に使用される暗号のコンマ区切りリストのみ** を設定します。可能な暗号名は、JCraft JSCH で定義されています。aes128-ctr,aes128-cbc,3des-ctr,3des-cbc,blowfish-cbc,aes192-cbc,aes256-cbc,aes256-cbc などがあります。指定しない場合は、JSCH のデフォルトのリストが使用されます。

fastExistsCheck	**false**	Camel 2.8.2、2.9: このオプションを true に設定すると、camel-ftp はリストファイルを直接使用してファイルが存在するかどうかを確認します。一部の FTP サーバーは直接ファイルの一覧表示をサポートしない可能性があるため、オプションが false の場合、camel-ftp は古い方法を使用してディレクトリーを一覧表示し、ファイルが存在するかどうかを確認します。**Camel 2.10.1** 以降では、このオプションが **readLock=changed** に影響し、ファイル情報の更新に高速チェックを実行するかどうかを制御します。FTP サーバーに多くのファイルがある場合に、これを使用してプロセスを迅速化できます。
strictHostKeyChecking	**いいえ**	SFTP のみ：Camel 2.2: 厳密なホストキーチェックを使用するかどうかを設定します。使用できる値は **no**、**yes**、および **ask** です。Camel は人間の介入を想定して質問に回答できないため、使用は理にかなっています。**注記：Camel 2.1 のデフォルトおよび以下では、が 求め られます。**
maximumReconnectAttempts	3	リモート FTP サーバーに接続しようとすると、Apache Camel の実行の最大再接続試行を指定します。この動作を無効にするには、0 を使用します。
reconnectDelay	1000	Apache Camel が再接続を試みる前に待機する遅延（ミリ秒単位）。
connectTimeout	**10000**	Camel 2.4: 接続タイムアウト（ミリ秒単位）です。これは FTP/FTPS の **ftpClient.connectTimeout** の使用に対応します。SFTP では、接続の試行時にもこのオプションが使用されます。
soTimeout	**30000**	FTP および FTPS のみ：Camel 2.4: **SocketOptions.SO_TIMEOUT** の値はミリ秒単位です。

timeout	**30000**	**FTP および FTPS のみ：Camel 2.4:** データのタイムアウトはミリ秒単位です。これは FTP/FTPS の **ftpClient.dataTimeout** の使用に対応します。SFTP の場合、データタイムアウトはありません。
throwExceptionOnConnectFailed	**false**	**Camel 2.5:** 接続に成功し、ログインが確立できない場合に例外を出力するかどうか。これにより、カスタムの **pollStrategy** は例外に対応できます。たとえば、コンシューマーやのような機能を停止できます。
siteCommand	**null**	**FTP および FTPS のみ：Camel 2.5:** ログインに成功した後にサイトコマンドを実行する。複数のサイトコマンドは、改行文字(\n)を使用して区切ることができます。**help site** を使用して、FTP サーバーがサポートするサイトコマンドを確認します。
stepwise	**true**	ディレクトリーを使用する場合は、ディレクトリーツリーをトラバースするためにステップ単位のモードを使用するかどうかを指定します。Stepwise は、一度に 1 つのディレクトリーが CD になることを意味します。詳細は、「ディレクトリーのステップ的な変更」を参照してください。
separator	**UNIX**	**Camel 2.6:** ファイルのアップロード時に使用するパスセパレーター **char** を変更します。**auto** は、変更せずに指定されたパスを使用することを意味します。**UNIX** は、UNIX スタイルのパス区切り文字を使用することを意味します。**Windows** は、Windows スタイルのパス区切り文字を使用することを意味します。
chmod	**null**	**SFTP プロデューサーのみ：Camel 2.9:** 保存したファイルに chmod を設定できます。たとえば、**chmod=640** です。

圧縮	**0**	**SFTP のみ： Camel 2.8.3/2.9:** 圧縮を使用します。レベルを 1 から 10 に指定します。**重要：** 圧縮サポートのために必要な JSCH zlib JAR をクラスパスに手動で追加する必要があります。
receiveBufferSize	**32768**	**FTP/FTPS のみ： Camel 2.15.1:** ファイルのダウンロード用のバッファーサイズ。デフォルトのサイズは 32KB です。
ftpClient	**null**	**FTP および FTPS のみ：Camel 2.1:** カスタムの **org.apache.commons.net.ftp.FTPClient** インスタンスを使用できます。
ftpClientConfig	**null**	**FTP および FTPS のみ：Camel 2.1:** カスタムの **org.apache.commons.net.ftp.FTPClientConfig** インスタンスを使用できます。
serverAliveInterval	**0**	**SFTP のみ：Camel 2.8** では、sftp セッションの serverAliveInterval を設定できます。
serverAliveCountMax	**1**	**SFTP のみ：Camel 2.8** を使用すると、sftp セッションの serverAliveCountMax を設定できます。
ftpClient.trustStore.file	**null**	**FTPS のみ：** トラストストアファイルを設定し、FTPS クライアントが信頼された証明書を検索できるようにします。
ftpClient.trustStore.type	**JKS**	**FTPS のみ：トラスト** ストアのタイプを設定します。
ftpClient.trustStore.algorithm	**SunX509**	**FTPS Only:** トラストストアアルゴリズムを設定します。
ftpClient.trustStore.password	**null**	**FTPS のみ：トラスト** ストアのパスワードを設定します。
ftpClient.keyStore.file	**null**	**FTPS のみ：** FTPS クライアントがプライベート証明書を検索できるようにキーストアファイルを設定します。

ftpClient.keyStore.type	**JKS**	FTPS Only: キーストアタイプを設定します。
ftpClient.keyStore.algorithm	**SunX509**	FTPS Only: キーストアアルゴリズムを設定します。
ftpClient.keyStore.password	**null**	FTPS Only: キーストアのパスワードを設定します。
ftpClient.keyStore.keyPassword	**null**	FTPS Only: 秘密鍵のパスワードを設定します。
sslContextParameters	**null**	**FTPS のみ：Camel 2.9:** レジストリーの **org.apache.camel.util.jsse.SSLContextParameters** への参照。 この参照は、ftpClient で設定された SSL 関連のオプションと、FtpsConfiguration に設定された securityProtocol (SSL、TLS など)を上書きします。 Security Guide および JSSE ユーティリティーの Configuring Transport Security for Camel Components を参照してください。
proxy	**null**	**SFTP のみ：Camel 2.10.7、2.11.1:** レジストリーで **com.jcraft.jsch.Proxy** への参照。 このプロキシーは、ターゲット SFTP ホストからのメッセージの消費/送信に使用されます。
useList	**true**	**FTP/FTPS のみ：Camel 2.12.1:** コンシューマーが FTP LIST コマンドを使用してディレクトリーの一覧を取得し、どのファイルが存在するかを確認します。このオプションを **false** に設定すると、**stepwise=false** を設定し、**fileName** も固定名に設定する必要があります。そのため、コンシューマーは取得するファイルの名前を認識します。これを行うと、1 つのファイルのみを取得できます。詳細については以下をご覧ください

ignoreFileNotFoundOrPermissionError	**false**	Camel 2.12.1: ファイルの取得を試みたにも拘らず、存在せず（何らかの理由で）ファイルの取得を試みたときにコンシューマーが無視するかどうか、またはファイルパーミッションが不十分であるために失敗するかどうか。
sendNoop	**true**	Camel 2.16: プロデューサーのみ。FTP サーバーにファイルをアップロードする前に、noop コマンドを書き込み前チェックとして送信するかどうか。このオプションは、接続を検証し、サイレント再接続を有効にするためにデフォルトで有効になります。ただし、これで何らかの問題が発生した場合には、このオプションを無効にすることができます。
jschLoggingLevel	**WARN**	SFTP のみ：Camel 2.15.3/2.16: JSCH アクティビティーロギングに使用するログレベル。JSCH はデフォルトで詳細であるため(**INFO** レベル)、しきい値はデフォルトで **WARN** に設定されます。

FTPS コンポーネントのデフォルトのトラストストア

FTPS コンポーネントで SSL に関連する ftpClient. プロパティーを使用する場合、トラストストアはすべての証明書を受け入れます。信頼の選択証明書のみが必要な場合は、ftpClient.trustStore.xxx オプションを使用するか、カスタム ftpClient を設定してトラストストアを設定する必要があります。

sslContextParameters を使用する場合、トラストストアは提供される SSLContextParameters インスタンスの設定によって管理されます。

その他のオプション

File のすべてのオプションは FTP2 によって継承されるため、その他のオプションについては、File を参照してください。

ftpClient. または ftpClientConfig. 接頭辞を使用すると、URI から ftpClientConfig および ftp ClientConfig に直接追加オプションを設定できます。

たとえば、FTPClient の setDataTimeout を 30 秒に設定するには、以下を実行できます。

```
from("ftp://foo@myserver?password=secret&ftpClient.dataTimeout=30000")
   .to("bean:foo");
```

たとえば、日付形式やタイムゾーンを設定するために、組み合わせて一致させ、両方の接頭辞を使用できます。

```
from("ftp://foo@myserver?
password=secret&ftpClient.dataTimeout=30000&ftpClientConfig.serverLanguageCode=fr")
   .to("bean:foo");
```

これらのオプションはいくつでも使用できます。

可能なオプションと詳細は、Apache Commons FTPClientConfig のドキュメントを参照してください。Apache Commons FTPClient でも同様です。

URL に多くの長い設定がない場合は、レジストリーで Camel ルックアップを許可することで、使用する ftpClient または ftpClientConfig を 参照 できます。

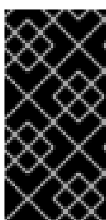
以下に例を示します。

```
<bean id="myConfig" class="org.apache.commons.net.ftp.FTPClientConfig">
   <property name="lenientFutureDates" value="true"/>
   <property name="serverLanguageCode" value="fr"/>
</bean>
```

URL で # 表記を使用すると、Camel がこの Bean をルックアップします。

```
from("ftp://foo@myserver?password=secret&ftpClientConfig=#myConfig").to("bean:foo");
```

その他の URI オプション



重要

このコンポーネントにも適用されるすべてのオプションについては、File2 を参照してください。

例

以下は、FTP エンドポイント URI の例です。

```
ftp://someone@someftpserver.com/public/upload/images/holiday2008?
password=secret&binary=true
ftp://someoneelse@someotherftpserver.co.uk:12049/reports/2008/password=secret&binary=fa
lse
ftp://publicftpserver.com/download
```

FTP コンシューマーが同時実行をサポートしない

FTP コンシューマー（同じエンドポイント）は同時実行をサポートしません（バッキング FTP クライアントはスレッドセーフではありません）。複数の FTP コンシューマーを使用して、異なるエンドポイントからポーリングできます。これは、同時コンシューマーをサポートしない単一のエンドポイントのみです。

FTP プロデューサーにはこの問題が ない ため、同時実行をサポートします。

補足情報

このコンポーネントは、File2 コンポーネントの拡張機能です。そのため、File2 コンポーネントページには、他のサンプルと詳細があります。

ファイルの使用時のデフォルト

FTP コンシューマーは、デフォルトで、消費されたファイルがリモート FTP サーバー上で変更されないままにします。ファイルを削除したり、別の場所に移動したりする場合は、明示的に設定する必要があります。たとえば、delete=true を使用してファイルを削除するか、move=.done を使用してファイルを非表示にしたサブディレクトリーに移動できます。

通常の File コンシューマーは、デフォルトでファイルを .camel サブディレクトリーに移動します。FTP コンシューマーに対して Camel はデフォルトでこれを行わないのは、ファイルを移動または削除するためにデフォルトでパーミッションがない可能性があるからです。

LIMITATIONS

readLock オプションを使用すると、Apache Camel で現在書き込まれているファイルが消費され ない ようにすることができます。ただし、ユーザーに書き込みアクセスが必要であるため、このオプションはデフォルトでオフになっています。FTP 経由で現在書き込まれているファイルを使用しないようにするための他の解決策があります。たとえば、一時的な宛先に書き込みを行い、ファイルを書き込んだ後に移動することができます。

ftp プロデューサーは、既存のファイルへの追加をサポートし ません。リモートサーバー上の既存のファイルは、ファイルが書き込まれる前に削除されます。

メッセージヘッダー

以下のメッセージヘッダーを使用して、コンポーネントの動作に影響を与えることができます。

ヘッダー	説明
CamelFileName	エンドポイントに送信するときに出力メッセージに使用される出力ファイル名（エンドポイントディレクトリーに対する相対）を指定します。これがなく、式がない場合は、生成されたメッセージ ID がファイル名として使用されます。
CamelFileNameProduced	書き込まれた出力ファイルのパス名。このヘッダーは Apache Camel によって自動的に設定されます。
CamelFileBatchIndex	このバッチで使用されるファイルの合計数からの現在のインデックス。
CamelFileBatchSize	このバッチで消費されるファイルの合計数。
CamelFileHost	リモートホスト名。
CamelFileLocalWorkPath	ローカルの作業ディレクトリーが使用されている場合は、ローカルのワークファイルへのパス。

さらに、FTP/FTPS コンシューマーおよびプロデューサーは、以下のヘッダーで Camel Message を強化します。

ヘッダー	説明
CamelFtpReplyCode	**Camel 2.11.1:** FTP クライアントの応答コード（タイプは整数）

CamelFtpReplyString	**Camel 2.11.1:** FTP クライアントの応答文字列

タイムアウトについて

2 つのライブラリーセット（上記を参照）には、タイムアウトを設定するために異なる API があります。両方の connectTimeout オプションを使用してタイムアウトをミリ秒単位で設定し、ネットワーク接続を確立できます。FTP/FTPS で個別の soTimeout を設定することもできます。これは ftpClient.soTimeout の使用に対応します。SFTP は connectTimeout を soTimeout として自動的に使用することに注意してください。timeout オプションは、ftpClient.dataTimeout 値に対応するデータタイムアウトとして FTP/FTSP にのみ適用されます。すべてのタイムアウト値はミリ秒単位です。

ローカルワークディレクトリーの使用

Apache Camel は、リモートの FTP サーバーからの消費と、ローカルの作業ディレクトリーへのファイルのダウンロードをサポートします。これにより、FileOutputStream を使用してローカルファイルに直接ストリーミングされるため、リモートファイルの内容全体がメモリーに読み取られなくなります。

Apache Camel は、ファイルのダウンロード中に .inprogress を拡張子とし、リモートファイルと同じ名前のローカルファイルに保存します。その後、ファイルの名前が変更され、.inprogress 接尾辞が削除されます。最後に、エクスチェンジ が完了すると、ローカルファイルが削除されます。

そのため、リモートの FTP サーバーからファイルをダウンロードしてファイルとして保存する場合は、以下のようなファイルエンドポイントにルーティングする必要があります。

```
from("ftp://someone@someserver.com?
password=secret&localWorkDirectory=/tmp").to("file://inbox");
```

ワークファイルの名前変更による最適化

上記のルートは、ファイルの内容全体をメモリーに読み込まないようにするため、効率的です。リモートファイルをローカルファイルストリームに直接ダウンロードします。その後、java.io.File ハンドルが エクスチェンジ ボディーとして使用されます。ファイルプロデューサーはこのファクトを活用し、ワークファイル java.io.File ハンドルで直接動作し、java.io.File.rename をターゲットファイル名に対して実行できます。Apache Camel はローカルのワークファイルを認識しているため、作業ファイルはいつでも削除されることが意図されているため、ファイルコピーの代わりに名前変更を最適化および使用できます。

ディレクトリーのステップ的な変更

Camel FTP は、ファイルの使用時（ダウンロードなど）やファイルの生成（アップロードなど）に関

する 2 つのモードで動作できます。

- stepwise
- 非ステップ(stepwise)

状況やセキュリティーの問題に応じて、どちらかを選択できます。一部の Camel エンドユーザーは、ステップベースを使用している場合のみファイルをダウンロードできますが、ダウンロードできない場合に限りダウンロードが可能です。少なくとも、選択する選択肢があります。

ディレクトリーのステップ的な変更は、多くの場合、ユーザーがそのホームディレクトリーに制限され、ホームディレクトリーが / として報告される場合にのみ機能します。

これら 2 つの違いは、例で説明するのが最適です。リモート FTP サーバーで以下のディレクトリー構造がある場合は、ファイルをトラバースしてダウンロードする必要があります。

```
/
/one
/one/two
/one/two/sub-a
/one/two/sub-b
```

また、sub-a (a.txt)と sub-b (b.txt)の各フォルダーにファイルがあります。

STEPWISE=TRUE （デフォルトモード）の使用

以下のログは、FTP エンドポイントがステップ単位 のモードで動作している場合の FTP エンドポイントとリモート FTP サーバー間の対話を示しています。

```
TYPE A
200 Type set to A
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
SYST
215 UNIX emulated by FileZilla
PORT 127,0,0,1,17,94
```

```
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CWD sub-a
250 CWD successful. "/one/two/sub-a" is current directory.
PORT 127,0,0,1,17,95
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CDUP
200 CDUP successful. "/one/two" is current directory.
CWD sub-b
250 CWD successful. "/one/two/sub-b" is current directory.
PORT 127,0,0,1,17,96
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CDUP
200 CDUP successful. "/one/two" is current directory.
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
PORT 127,0,0,1,17,97
200 Port command successful
RETR foo.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
CWD sub-a
250 CWD successful. "/one/two/sub-a" is current directory.
PORT 127,0,0,1,17,98
200 Port command successful
RETR a.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
```

```
CWD two
250 CWD successful. "/one/two" is current directory.
CWD sub-b
250 CWD successful. "/one/two/sub-b" is current directory.
PORT 127,0,0,1,17,99
200 Port command successful
RETR b.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
QUIT
221 Goodbye
disconnected.
```

ステップが有効にされると、CD xxx を使用してディレクトリー構造を通過します。

STEPWISE=FALSE の使用

以下のログは、FTP エンドポイントがステップ以外のモードで動作している場合の FTP エンドポイントとリモート FTP サーバー間の対話を示して います。

```
230 Logged on
TYPE A
200 Type set to A
SYST
215 UNIX emulated by FileZilla
PORT 127,0,0,1,4,122
200 Port command successful
LIST one/two
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,123
200 Port command successful
LIST one/two/sub-a
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,124
200 Port command successful
LIST one/two/sub-b
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,125
200 Port command successful
RETR one/two/foo.txt
150 Opening data channel for file transfer.
226 Transfer OK
PORT 127,0,0,1,4,126
200 Port command successful
RETR one/two/sub-a/a.txt
```

```
150 Opening data channel for file transfer.
226 Transfer OK
PORT 127,0,0,1,4,127
200 Port command successful
RETR one/two/sub-b/b.txt
150 Opening data channel for file transfer.
226 Transfer OK
QUIT
221 Goodbye
disconnected.
```

ステップ以外の場合に分かるように、CD 操作は呼び出されません。

サンプル

以下の例では、Apache Camel を設定して、FTP サーバーからすべてのレポートを 1 時間(60 分)に BINARY コンテンツとしてダウンロードし、ローカルファイルシステムにファイルとして保存するように設定します。

```
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // we use a delay of 60 minutes (eg. once pr. hour we poll the FTP server
            long delay = 60 * 60 * 1000L;

            // from the given FTP server we poll (= download) all the files
            // from the public/reports folder as BINARY types and store this as files
            // in a local directory. Apache Camel will use the filenames from the FTPServer

            // notice that the FTPConsumer properties must be prefixed with "consumer." in the
URL
            // the delay parameter is from the FileConsumer component so we should use
consumer.delay as
            // the URI parameter name. The FTP Component is an extension of the File Component.
            from("ftp://tiger:scott@localhost/public/reports?binary=true&consumer.delay=" +
delay).
                to("file://target/test-reports");
        }
    };
}
```

Spring DSL を使用したルート：

```
<route>
   <from uri="ftp://scott@localhost/public/reports?password=tiger&inary=true&elay=60000"/>
   <to uri="file://target/test-reports"/>
</route>
```

ルートによってトリガーされるリモート FTP サーバーの使用

FTP コンシューマーは、from ルートで使用されるスケジュールされたコンシューマーとして構築されます。ただし、ルート内でトリガーされた FTP サーバーから消費を開始する場合は、以下のようなルートを使用します。

```
from("seda:start")
  // define the file name so that only a single file is polled and deleted once retrieved
  .pollEnrich("ftp://admin@localhost:21/getme?
password=admin&binary=false&fileName=myFile.txt&delete=true")
  .to("mock:result");
```

リモート FTPS サーバー（暗黙的な SSL）およびクライアント認証の使用

```
from("ftps://admin@localhost:2222/public/camel?
password=admin&securityProtocol=SSL&isImplicit=true
      &ftpClient.keyStore.file=./src/test/resources/server.jks
      &ftpClient.keyStore.password=password&ftpClient.keyStore.keyPassword=password")
  .to("bean:foo");
```

リモート FTPS サーバー(EXPLICIT TLS)およびカスタムトラストストア設定の使用

```
from("ftps://admin@localhost:2222/public/camel?
password=admin&ftpClient.trustStore.file=./src/test/resources/server.jks&ftpClient.trustStore.
password=password")
  .to("bean:foo");
```

ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER を使用するフィルター

Apache Camel はプラグ可能なフィルターリングストラテジーをサポートします。Java で org.apache.camel.component.file.GenericFileFilter インターフェイスを実装してフィルターストラテジーを定義します。次に、フィルターでエンドポイントを設定し、特定のファイルをスキップできます。

以下の例では、report で始まるファイル名のファイルのみを許可するフィルターを定義します。

```
public class MyFileFilter<T> implements GenericFileFilter<T> {
```

```
    public boolean accept(GenericFile<T> file) {
        // we only want report files
        return file.getFileName().startsWith("report");
    }
}
```

次に、filter 属性を使用して、Spring XML ファイルで定義したフィルター(# 表記を使用)を参照するルートを設定できます。

```
<!-- define our sorter as a plain spring bean -->
<bean id="myFilter" class="com.mycompany.MyFileFilter"/>

<route>
  <from uri="ftp://someuser@someftpserver.com?password=secret&filter=#myFilter"/>
  <to uri="bean:processInbox"/>
</route>
```

ANT パスマッチャーを使用したフィルターリング

ANT パスマッチャーは、camel-spring jar で追加設定なしで同梱されるフィルターです。そのため、Maven を使用している場合は camel-spring に依存する必要があります。理由は、Spring の AntPathMatcher を使用して実際のマッチングを行うためです。

ファイルパスは、以下のルールと一致します。

- ? 1 文字に一致します。
- * ゼロ以上の文字に一致します。
- ** パス内の 0 個以上のディレクトリーに一致します。

以下の例は、その使用方法を示しています。

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>
<camelContext xmlns="http://camel.apache.org/schema/spring">

  <template id="camelTemplate"/>

  <!-- use myFilter as filter to allow setting ANT paths for which files to scan for -->
  <endpoint id="myFTPEndpoint"
```

```
uri="ftp://admin@localhost:${SpringFileAntPathMatcherRemoteFileFilterTest.ftpPort}/antpath
?password=admin&ecursive=true&elay=10000&nitialDelay=2000&ilter=#myAntFilter"/>

    <route>
        <from ref="myFTPEndpoint"/>
        <to uri="mock:result"/>
    </route>
</camelContext>

<!-- we use the AntPathMatcherRemoteFileFilter to use ant paths for includes and exclude -->
<bean id="myAntFilter"
class="org.apache.camel.component.file.AntPathMatcherGenericFileFilter">
    <!-- include any files in the sub-folder that have day in the name -->
    <property name="includes" value="**/subfolder/**/*day*"/>
    <!-- exclude all files with bad in name or .xml files. Use comma to seperate multiple
excludes -->
    <property name="excludes" value="**/*bad*,**/*.xml"/>
</bean>
```

SFTP でのプロキシーの使用

HTTP プロキシーを使用してリモートホストに接続するには、以下のようにルートを設定できます。

```
<!-- define our sorter as a plain spring bean -->
<bean id="proxy" class="com.jcraft.jsch.ProxyHTTP">
  <constructor-arg value="localhost"/>
  <constructor-arg value="7777"/>
</bean>

<route>
  <from uri="sftp://localhost:9999/root?username=admin&password=admin&proxy=#proxy"/>
  <to uri="bean:processFile"/>
</route>
```

必要に応じて、ユーザー名とパスワードをプロキシーに割り当てることもできます。すべてのオプションを検出するには、com.jcraft.jsch.Proxy のドキュメントを参照してください。

推奨される SFTP 認証方法の設定

sftp コンポーネントが使用する認証方法の一覧を明示的に指定する場合は、preferredAuthentications オプションを使用します。たとえば、Camel が秘密鍵/パブリック SSH 鍵で認証を試み、公開鍵が利用できない場合にユーザー/パスワード認証にフォールバックします。以下のルート設定を使用します。

```
from("sftp://localhost:9999/root?
username=admin&password=admin&preferredAuthentications=publickey,password")
  .to("bean:processFile");
```

固定名を使用した単一ファイルの使用

1 つのファイルをダウンロードし、ファイル名を知っている場合は、fileName=myFileName.txt を使用して、ダウンロードするファイルの名前を Camel に指示できます。デフォルトでは、コンシューマーは FTP LIST コマンドを実行してディレクトリーの一覧を実行し、fileName オプションに基づいてこれらのファイルをフィルターリングします。このユースケースでは、useList=false を設定してディレクトリー一覧をオフにすることが推奨されます。たとえば、FTP サーバーへのログインに使用されるユーザーアカウントには、FTP LIST コマンドを実行する権限がない場合があります。そのため、useList=false を使用してこれをオフにしてから、fileName=myFileName.txt でダウンロードするファイルの固定名を指定します。その後、FTP コンシューマーはファイルをダウンロードできます。何らかの理由で ファイルが存在しない場合は、Camel はデフォルトで例外を出力し、ignoreFileNotFoundOrPermissionError=true を設定してこれを無視します。

たとえば、単一ファイルを選択し、使用後に削除する Camel ルートを設定するには、次のようにします。

```
from("ftp://admin@localhost:21/nolist/?
password=admin&stepwise=false&useList=false&ignoreFileNotFoundOrPermissionError=true
&fileName=report.txt&delete=true")
  .to("activemq:queue:report");
```

上記で説明したすべてのオプションを使用していることに注意してください。

ConsumerTemplate で使用することもできます。たとえば、単一のファイル（存在する場合）をダウンロードし、ファイルの内容を String タイプとして取得するには、以下を実行します。

```
String data = template.retrieveBodyNoWait("ftp://admin@localhost:21/nolist/?
password=admin&stepwise=false&useList=false&ignoreFileNotFoundOrPermissionError=true
&fileName=report.txt&delete=true", String.class);
```

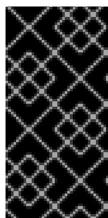
デバッグロギング

このコンポーネントにはログレベル TRACE があり、問題がある場合に役立ちます。

第54章 GAE

54.1. GAE コンポーネントの概要

Google App Engine の Apache Camel コンポーネント



重要

GAE コンポーネントは非推奨となり、JBoss Fuse の今後のリリースで削除される予定です。

チュートリアル

- *Apache Camel on GAE を使用するためのスタート地点は、Google App Engine 上の Camel のチュートリアルです。*

- *OAuth チュートリアル は、Web アプリケーションで OAuth を実装する方法を示しています。*

Google App Engine (GAE)の Apache Camel コンポーネントは camel-gae プロジェクトの一部で、GAE の クラウドコンピューティングサービス への接続を提供します。これにより、Apache Camel インターフェイスを介してアプリケーションが GAE クラウドコンピューティング環境にアクセスできるようになります。他のクラウドコンピューティング環境のこのパターンに従うと、Apache Camel アプリケーションをあるクラウドプロバイダーから別のクラウドコンピューティングプロバイダーに移植しやすくなります。以下の表は、Google App Engine によって提供されるクラウドコンピューティングサービスとサポートする Apache Camel コンポーネントを示しています。各コンポーネントのドキュメントは、Camel Component 列のリンクに従って表示されます。

GAE サービス	Camel コンポーネント	コンポーネントの説明
URL フェッチサービス	ghttp	GAE URL フェッチサービスへの接続を提供しますが、サーブレットからメッセージを受信する場合にも使用できます。
タスクキューサービス	gtask	タスクキューをメッセージキューとして使用して、GAE での非同期メッセージ処理をサポートします。

メールサービス	gmail	GAE メールサービスを介したメールの送信をサポートします。メールの受信はまだサポートされていませんが、後で追加されます。
memcache サービス		サポートされていません。
XMPP サービス		サポートされていません。
イメージサービス		サポートされていません。
データストアサービス		サポートされていません。
Accounts サービス	gauth glogin	これらのコンポーネントは、認証および承認のために Google Accounts API と対話します。Google アカウントは Google App Engine に固有のものではありませんが、セキュリティーを実装するために GAE アプリケーションによって使用されます。gauth コンポーネントは、Google 固有の OAuth コンシューマーを実装するために Web アプリケーションによって使用されます。このコンポーネントは、GAE 以外の Web アプリケーションを有効にするためにも使用できます。glogin コンポーネントは、GAE アプリケーションにプログラムによるログインのために Java クライアント (GAE 外)によって使用されます。GAE アプリケーションを不正アクセスから保護する方法は、セキュリティー ページを参照してください。

Camel コンテキスト

Google App Engine での SpringCamelContext の設定は、Camel 2.1 以降のバージョンによって異なります。この問題は、http://camel.apache.org/schema/spring の Camel 固有の Spring 設定 XML スキーマを使用するには JAXB が必要で、Camel 2.1 は JAXB をサポートしない Google App Engine SDK バージョンに依存することです。この制限は、Camel 2.2 以降から削除されました。

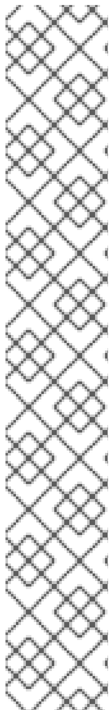
javax.management パッケージは App Engine JRE のホワイトリストに含まれていないため、JMX は必ず無効にする必要があります。

Apache Camel 2.1

camel-gae 2.1 には、以下の CamelContext 実装が同梱されています。

- org.apache.camel.component.gae.context.GaeDefaultCamelContext (org.apache.camel.impl.DefaultCamelContextの拡張)
- org.apache.camel.component.gae.context.GaeSpringCamelContext (org.apache.camel.spring.SpringCamelContextを拡張)

いずれも起動前に JMX を無効にします。GaeSpringCamelContext は、以下の例のようにルートビルダーを追加するセッターメソッドを追加で提供します。



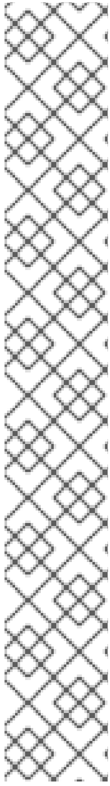
APPCTX.XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="camelContext"

class="org.apache.camel.component.gae.context.GaeSpringCamelContext">
        <property name="routeBuilder" ref="myRouteBuilder" />
    </bean>

    <bean id="myRouteBuilder"
        class="org.example.MyRouteBuilder">
    </bean>

</beans>
```

または、ルートビルダーの一覧を設定するには、GaeSpringCamelContext の routeBuilders プロパティーを使用します。この方法では、JAXB を必要とせずに SpringCamelContext を GAE に設定できます。

Apache Camel 2.2

Camel 2.2 以降では、アプリケーションは SpringCamelContext を設定するために http://camel.apache.org/schema/spring namespace を使用できますが、JMX を無効にする必要があります。以下は例です。



APPCTX.XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

    <camel:camelContext id="camelContext">
        <camel:jmxAgent id="agent" disabled="true" />
        <camel:routeBuilder ref="myRouteBuilder"/>
    </camel:camelContext>

    <bean id="myRouteBuilder"
        class="org.example.MyRouteBuilder">
    </bean>

</beans>
```

web.xml

GAE で Apache Camel を実行するには、camel-servlet から CamelHttpTransportServlet を使用する必要があります。以下の例は、Spring アプリケーションコンテキスト XML ファイルとともにこのサーブレットを設定する方法を示しています。

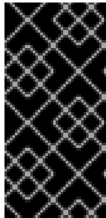
WEB.XML

```
<web-app
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="
http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

    <servlet>
        <servlet-name>CamelServlet</servlet-name>
        <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet
-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>appctx.xml</param-value>
        </init-param>
    </servlet>

    <!--
        Mapping used for external requests
     -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/camel/*</url-pattern>
    </servlet-mapping>

    <!--
        Mapping used for web hooks accessed by task queueing service.
     -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/worker/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Spring アプリケーションコンテキスト XML ファイルの場所は、contextConfigLocation init パラメーターによって指定されます。appctx.xml ファイルはクラスパス上にある必要があります。サーブレットマッピングは、Google App Engine にデプロイする際に、http://<appname>.appspot.com/camel/... で Apache Camel アプリケーションにアクセスできるようにします。< appname > は実際の GAE アプリケーション名に置き換えます。2 つ目のサーブレットマッピングは、Web フック を介してバックグラウンド処理のためにタスクキューサービスによって内部的に使用されます。このマッピングは gtask コンポーネントに関連し、詳細に説明されています。

54.2. GAUTH

gauth コンポーネント



重要

GAE コンポーネントは非推奨となり、JBoss Fuse の今後のリリースで削除される予定です。

Apache Camel 2.3 で利用可能

gauth コンポーネントは、Google 固有の OAuth コンシューマーを実装するために Web アプリケーションによって使用されます。他の OAuth プロバイダーもサポートするように、後で拡張されます。このコンポーネントは Google App Engine (GAE)の Camel コンポーネントに属していますが、OAuth-enable 非 GAE Web アプリケーションにも使用することができます。Google の OAuth 実装の詳細については、Google OAuth API リファレンス を参照してください。

URI 形式

```
gauth://name[?options]
```

エンドポイント 名 は、を 承認 または アップグレード できます。承認 エンドポイントは、Google から承認されていないリクエストトークンを取得し、ユーザーを承認ページにリダイレクトするために使用されます。upgrade エンドポイントは、Google から OAuth コールバックを処理し、承認されたリクエストトークンを有効期限の長いアクセストークンにアップグレードするために使用されます。例については、使用方法のセクション を参照してください。

オプション

名前	デフォルト値	必須	説明
callback	**null**	True (**GAuthAuthorizeBinding.GAUTH_CALLBACK** メッセージヘッダーで設定することもできます)	アクセスの許可または拒否後にユーザーをリダイレクトする URL。

scope	**null**	true (**GAuthAuthorizeBinding.GAUTH_SCOPE** メッセージヘッダーで設定することもできます)	アクセスするサービスを識別する URL。スコープは各 Google サービスによって定義されます。正しい値については、サービスのドキュメントを参照してください。複数のスコープを指定するには、それぞれをコンマで区切って一覧表示します。例：**http://www.google.com/calendar/feeds/**
consumerKey	**null**	True (コンポーネントレベルでも設定できます)。	Web アプリケーションを識別するドメイン。これは、アプリケーションを Google に登録する際に使用されるドメインです。例：**camelcloud.appspot.com**登録されていないアプリケーションでは、**匿名** を使用します。
consumerSecret	**null**	**consumerSecret** または **keyLoaderRef** のいずれかが必要です（または、コンポーネントレベルの で設定できます）。	Web アプリケーションのコンシューマーシークレット。コンシューマーシークレットは、アプリケーションを Google に登録する際に生成されます。HMAC-SHA1 署名方式を使用する場合は、これが必要です。登録されていないアプリケーションでは、**匿名** を使用します。

keyLoaderRef	**null**	**consumerSecret** または **keyLoaderRef** のいずれかが必要です（代わりに コンポーネントレベルの に設定できます）。	レジストリー内の秘密鍵ローダーへの参照。**camel-gae** の一部は 2 つの主要なローダーです。PKCS#8 ファイルから秘密鍵を読み込む **GAuthPk8Loader** と、Java キーストアから秘密鍵を読み込む **GAuthJksLoader** です。これは、RSA-SHA1 署名メソッドを使用する場合は必要になります。これらのクラスは **org.apache.camel.component.gae.auth** パッケージで定義されます。
authorizeBindingRef	**GAuthAuthorizeBinding**への参照	false	**エクスチェンジ** が **GoogleOAuthParameters** **にバインドされる方法をカスタマイズするための** **OutboundBinding<GAuthEndpoint, GoogleOAuthParameters, GoogleOAuthParameters** > への参照。このバインディングは、teh 承認フェーズに使用されます。ほとんどのアプリケーションはデフォルト値を変更しません。
upgradeBindingRef	**GAuthAuthorizeBinding**への参照	false	レジストリーでの **OutboundBinding<GAuthEndpoint、GoogleOAuthParameters、GoogleOAuthParameters** > への参照。**エクスチェンジ** が **GoogleOAuthParameters** にバインドされる方法をカスタマイズするためのものです。このバインディングは、teh トークンのアップグレードフェーズに使用されます。ほとんどのアプリケーションはデフォルト値を変更しません。

メッセージヘッダー

名前	タイプ	エンドポイント	メッセージ	説明
GAuthAuthorizeBinding.GAUTH_CALLBACK	**文字列**	**gauth:authorize**	in	**コールバック** オプションを上書きします。
GAuthAuthorizeBinding.GAUTH_SCOPE	**文字列**	**gauth:authorize**	in	**scope** オプションを上書きします。
GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN	**文字列**	**gauth:upgrade**	out	有効期間の長いアクセストークンが含まれています。このトークンは、アプリケーションによってユーザーのコンテキストに保存する必要があります。
GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN_SECRET	**文字列**	**gauth:upgrade**	out	アクセストークンのシークレットが含まれます。このトークンシークレットは、アプリケーションによってユーザーのコンテキストに保存する必要があります。

メッセージボディー

gauth コンポーネントはメッセージの本文を読み書きしません。

コンポーネントの設定

consumerKey、consumerSecret、keyLoader などの一部のエンドポイントオプションは、通常 gauth:authorize および gauth:upgrade エンドポイントで同じ値に設定されます。gauth コンポーネントは、コンポーネントレベルでそれらを設定できるようにします。これらの設定は gauth エンドポイントによって継承され、エンドポイント URI で冗長的に設定する必要はありません。以下は、いくつかの設定例になります。

HMAC-SHA1 署名方法を使用した登録済み WEB アプリケーションのコンポーネント設定

```
<bean id="gauth"
class="org.apache.camel.component.gae.auth.GAuthComponent">
    <property name="consumerKey" value="example.appspot.com" />
    <property name="consumerSecret" value="QAtA...HfQ" />
</bean>
```

HMAC-SHA1 署名方法を使用した未登録の WEB アプリケーションのコンポーネント設定

```
<bean id="gauth"
class="org.apache.camel.component.gae.auth.GAuthComponent">
    <!-- Google will display a warning message on the authorization page -->
    <property name="consumerKey" value="anonymous" />
    <property name="consumerSecret" value="anonymous" />
</bean>
```

RSA-SHA1 署名メソッドを使用した登録済み WEB アプリケーションのコンポーネント設定

```
<bean id="gauth"
class="org.apache.camel.component.gae.auth.GAuthComponent">
    <property name="consumerKey" value="ipfcloud.appspot.com" />
    <property name="keyLoader" ref="jksLoader" />
    <!--<property name="keyLoader" ref="pk8Loader" />-->
</bean>

<!-- Loads the private key from a Java key store -->
<bean id="jksLoader"
    class="org.apache.camel.component.gae.auth.GAuthJksLoader">
    <property name="keyStoreLocation" value="myKeytore.jks" />
    <property name="keyAlias" value="myKey" />
    <property name="keyPass" value="myKeyPassword" />
    <property name="storePass" value="myStorePassword" />
</bean>

<!-- Loads the private key from a PKCS#8 file -->
<bean id="pk8Loader"
    class="org.apache.camel.component.gae.auth.GAuthPk8Loader">
    <property name="keyStoreLocation" value="myKeyfile.pk8" />
</bean>
```

使用方法

以下は、(GAE 以外の) Web アプリケーションに OAuth を追加するための最低限の設定です。以下の例では、Web アプリケーションが gauth.example.org で実行されていることを前提としています。

GAUTHROUTEBUILDER.JAVA

```
import java.net.URLEncoder;
import org.apache.camel.builder.RouteBuilder;

public class GAuthRouteBuilder extends RouteBuilder {

   @Override
   public void configure() throws Exception {

      // Calback URL to redirect user from Google Authorization back to the web
application
      String encodedCallback =
URLEncoder.encode("https://gauth.example.org:8443/handler", "UTF-8");
      // Application will request for authorization to access a user's Google
Calendar
      String encodedScope =
URLEncoder.encode("http://www.google.com/calendar/feeds/", "UTF-8");

      // Route 1: A GET request to http://gauth.example.org/authorize will trigger
the the OAuth
      // sequence of interactions. The gauth:authorize endpoint obtains an
unauthorized request
      // token from Google and then redirects the user (browser) to a Google
authorization page.
      from("jetty:http://0.0.0.0:8080/authorize")
         .to("gauth:authorize?callback=" + encodedCallback + "&scope=" +
encodedScope);

      // Route 2: Handle callback from Google. After the user granted access to
Google Calendar
      // Google redirects the user to https://gauth.example.org:8443/handler (see
callback) along
      // with an authorized request token. The gauth:access endpoint exchanges
the authorized
      // request token against a long-lived access token.
      from("jetty:https://0.0.0.0:8443/handler")
         .to("gauth:upgrade")
         // The access token can be obtained from
         //
exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN)
         // The access token secret can be obtained from
         //
exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN_
SECRET)
         .process(/* store the tokens in context of the current user ... */);
   }

}
```

OAuth シーケンスは、 http://gauth.example.org/authorize に GET リクエストを送信することで

トリガーされます。その後、ユーザーは Google 承認ページにリダイレクトされます。このページへのアクセスを許可した後、Google はユーザーをコールバックを処理する Web アプリケーションにリダイレクトすると、最後に Google から有効期限の長いアクセストークンを取得します。

これら 2 つのルートは、他の Web アプリケーションフレームワークと完全に共存できます。フレームワークは、Web アプリケーション固有の機能の基盤を提供しますが、OAuth サービスプロバイダーのインテグレーションは Apache Camel で実行されます。OAuth 統合部分は、jetty コンポーネントの代わりにサーブレットコンポーネントを使用して、既存の サーブレット コンテナーのリソースを使用することもできます。

OAUTH アクセストークンについて

- アプリケーションは、現在のユーザーのコンテキストでアクセストークンを保存する必要があります。ユーザーが次回ログインする場合、OAuth の dance を再度実行せずに、アクセストークンをデータベースから直接読み込むことができます。

- その後、アクセストークンを使用して、ユーザーの代わりに Google カレンダー API などの Google サービスへのアクセスを取得します。Java アプリケーションは多くの場合、GData Java ライブラリー を使用する可能性が高くなります。ユーザーのカレンダーフィードを読み取るために GData Java ライブラリーでアクセストークンを使用する方法の 例 は、以下を参照してください。

- ユーザーは、Google Accounts ページからアクセストークンをいつでも取り消すことができます。この場合、対応する Google サービスにアクセスすると、承認例外が発生します。Web アプリケーションは、保存されたアクセストークンを削除し、別のトークンを作成するためにユーザーを Google 承認ページにリダイレクトする必要があります。

上記の例は、以下のコンポーネント設定に依存します。

```
<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
    <property name="consumerKey" value="anonymous" />
    <property name="consumerSecret" value="anonymous" />
</bean>
```

Google が承認ページに警告メッセージを表示したくない場合は、Web アプリケーションを 登録 して consumerKey および consumerSecret 設定を変更する必要があります。

GAE の例

Google App Engine アプリケーションを有効にするには、ルートビルダーでいくつかの小さな変更

のみが必要になります。GAE アプリケーションのホスト名が camelcloud.appspot.com であるとすると、設定は以下のようになります。ここでは、ghttp コンポーネントは、jetty コンポーネントの代わりに HTTP (S)要求を処理するために使用されます。

GAUTHROUTEBUILDER

```
import java.net.URLEncoder;
import org.apache.camel.builder.RouteBuilder;

public class TutorialRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        String encodedCallback =
URLEncoder.encode("https://camelcloud.appspot.com/handler", "UTF-8");
        String encodedScope =
URLEncoder.encode("http://www.google.com/calendar/feeds/", "UTF-8");

        from("ghttp:///authorize")
            .to("gauth:authorize?callback=" + encodedCallback + "&scope=" +
encodedScope);

        from("ghttp:///handler")
            .to("gauth:upgrade")
            .process(/* store the tokens in context of the current user ... */);
    }

}
```

アクセストークンの使用

以下の例は、アクセストークンを使用して、GData Java ライブラリー でユーザーの Google カレンダーデータにアクセスする方法を示しています。サンプルアプリケーションは、ユーザーのパブリックおよびプライベートカレンダーのタイトルを stdout に書き込みます。

アクセストークンの使用

```
import com.google.gdata.client.authn.oauth.OAuthHmacSha1Signer;
import com.google.gdata.client.authn.oauth.OAuthParameters;
import com.google.gdata.client.calendar.CalendarService;
import com.google.gdata.data.calendar.CalendarEntry;
import com.google.gdata.data.calendar.CalendarFeed;

import java.net.URL;

public class AccessExample {

    public static void main(String... args) throws Exception {
        String accessToken = ...
        String accessTokenSecret = ...

        CalendarService myService = new CalendarService("exampleCo-
exampleApp-1.0");
        OAuthParameters params = new OAuthParameters();
        params.setOAuthConsumerKey("anonymous");
        params.setOAuthConsumerSecret("anonymous");
        params.setOAuthToken(accessToken);
        params.setOAuthTokenSecret(accessTokenSecret);
        myService.setOAuthCredentials(params, new OAuthHmacSha1Signer());

        URL feedUrl = new URL("http://www.google.com/calendar/feeds/default/");
        CalendarFeed resultFeed = myService.getFeed(feedUrl,
CalendarFeed.class);

        System.out.println("Your calendars:");
        System.out.println();

        for (int i = 0; i < resultFeed.getEntries().size(); i++) {
            CalendarEntry entry = resultFeed.getEntries().get(i);
            System.out.println(entry.getTitle().getPlainText());
        }
    }
}
```

54.3. GHTTP

ghttp コンポーネント

重要

GAE コンポーネントは非推奨となり、JBoss Fuse の今後のリリースで削除される予定です。

ghttp コンポーネントは、Google App Engine (GAE)の Camel コンポーネントに提供します。GAE URL フェッチサービス への接続を提供しますが、サーブレットからメッセージを受信するためにも使

用できます(GAE で HTTP リクエストを受信する唯一の方法)。これは、Servlet コンポーネント を拡張することで実現されます。そのため、ghttp URI 形式およびオプションは、コンシューマー側の（からの）およびプロデューサー側の(から)とは異なります。

URI 形式

形式	コンテキスト	Comment
ghttp:///path[?options]	コンシューマー	Servlet コンポーネントも参照してください。
ghttp://hostname[:port][/path][?options] ghttps://hostname[:port][/path][?options]	プロデューサー	Http コンポーネントも参照してください。

オプション

名前	デフォルト値	コンテキスト	説明
bridgeEndpoint	**true**	プロデューサー	**true** に設定すると、**Exchange.HTTP_URI** ヘッダーは無視されます。**Exchange.HTTP_URI** ヘッダーでデフォルトのエンドポイント URI を上書きするには、このオプションを **false** に設定します。
throwExceptionOnFailure	**true**	プロデューサー	応答コードが >= 400 の場合は、**org.apache.camel.component.gae.http** を出力します。例外の出力を無効にするには、このオプションを **false** に設定します。

inboundBindingRef	GHttpBindingへの参照	コンシューマー	**エクスチェンジ** のサーブレット API へのバインディングをカスタマイズするための、**レジストリー内の InboundBinding<GHttpEndpoint、HttpServletRequest、HttpServletResponse** > への参照。参照バインディングは、**org.apache.camel.component.http.HttpBinding** への後プロセッサーとして使用されます。
outboundBindingRef	**GHttpBinding**への参照	プロデューサー	**エクスチェンジ** の **URLFetchService** へのバインディングをカスタマイズするための Registry の **OutboundBinding<GHttpEndpoint, HTTPRequest, HTTPResponse** > への参照。

コンシューマー側では、[Servlet コンポーネント](#) のすべてのオプションがサポートされます。

メッセージヘッダー

プロデューサー側では、[Http コンポーネント](#) の以下のヘッダーがサポートされます。

名前	タイプ	説明
Exchange.CONTENT_TYPE	**文字列**	HTTP コンテンツタイプ。は、**in** および **out** メッセージの両方で設定され、**text/html** などのコンテンツタイプを提供します。
Exchange.CONTENT_ENCODING	**文字列**	HTTP コンテンツエンコーディング。は、**gzip** などのコンテンツエンコーディングを提供するために **in** および **out** メッセージの両方に設定されます。

Exchange.HTTP_METHOD	文字列	実行する HTTP メソッド。**GET**、**POST**、**PUT**、**DELETE** のいずれか。設定されていない場合は、メッセージのボディーが **null** でない場合は **POST** が使用され、それ以外の場合は**GET** が使用されます。
Exchange.HTTP_QUERY	文字列	エンドポイント URI のクエリー部分または **Exchange.HTTP_URI** のクエリー部分（定義されている場合）を上書きします。クエリー文字列はデコードされた形式である必要があります。
Exchange.HTTP_URI	文字列	**bridgeEndpoint** オプションが **false** に設定されている場合には、デフォルトのエンドポイント URI を上書きします。URI 文字列はデコード形式である必要があります。
Exchange.RESPONSE_CODE	**int**	URL からの HTTP 応答コードは、サービス応答をフェッチします。

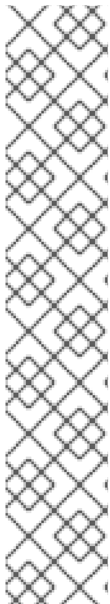
コンシューマー側で Servlet コンポーネントコンポーネント のすべてのヘッダーがサポートされます。

メッセージボディー

*プロデューサー側で、のメッセージボディーは **byte[]** に変換されます。**out** メッセージ本文は **InputStream** として利用できます。リポジトリーサイズが 1 メガバイトを超える場合、URL フェッチサービスによって **ResponseTooLargeException** が出力されます(クォータおよび制限を参照)。*

メッセージの受信

*ghttp コンポーネント経由でメッセージを受信するには、**CamelHttpTransportServlet** を設定し、アプリケーションの **web.xml** でマッピングする必要があります(「web.xml」を参照してください)。たとえば、**http://<appname>.appspot.com/camel/*** または **http://localhost/camel/***（ローカル開発サーバーを使用する場合）でターゲットとする要求を処理するには、以下のサーブレットマッピングを定義する必要があります。*



WEB.XML

```
...
<servlet>
    <servlet-name>CamelServlet</servlet-name>
    <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet
-class>
    ...
</servlet>
...
<servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/camel/*</url-pattern>
</servlet-mapping>
...
```

エンドポイント URI パス定義は、このサーブレットマッピングに相対的です（例：ルート）。

```
from("ghttp:///greeting").transform().constant("Hello")
```

http://<appname>.appspot.com/camel/greeting でターゲットとなる要求を処理します。この例では、リクエストボディーは無視され、応答ボディーが Hello に設定されています。http://<appname>.appspot.com/camel/greeting/* でターゲットとする要求は、デフォルトでは処理されません。これには、オプション matchOnUriPrefix を true に設定する必要があります。

```
from("ghttp:///greeting?matchOnUriPrefix=true").transform().constant("Hello")
```

メッセージの送信

外部 HTTP サービスに再起動を送信する場合、ghttp コンポーネントは URL フェッチサービス を使用します。たとえば、Apache Camel のホームページは、プロデューサー側で以下のエンドポイント定義で取得できます。

```
from(...)
...
.to("ghttp://camel.apache.org")
...
```

使用される HTTP メソッドは、Exchange.HTTP_METHOD メッセージヘッダーまたはインメッセージの本文の存在によって異なります(null の場合はGET、POST の場合は POST)。GAE アプリケーションを介した Camel ホームページの取得は、

```
from("ghttp:///home")
.to("ghttp://camel.apache.org")
```

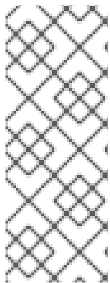
–

GET リクエストを http://<appname>.appspot.com/camel/home に送信すると、Camel ホームページが返されます。外部サービスとの HTTPS ベースの通信は、ghttps スキームで有効にできます。

```
from(...)
...
.to("ghttps://svn.apache.org/repos/asf/camel/trunk/")
...
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



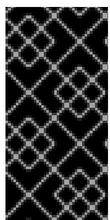
POM.XML

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-gae</artifactId>
   <version>${camel-version}</version>
</dependency>
```

${camel-version} は、実際のバージョンの Apache Camel (2.1.0 以降)に置き換える必要があります。

54.4. GLOGIN

glogin コンポーネント



重要

GAE コンポーネントは非推奨となり、JBoss Fuse の今後のリリースで削除される予定です。

Apache Camel 2.3 （または最新の 開発スナップショット）で利用 できます。

glogin コンポーネントは、GAE アプリケーションにプログラムによるログインを行うために、Google App Engine (GAE)以外の Apache Camel アプリケーションによって使用されます。これは、54章GAE の一部です。セキュリティー対応の GAE アプリケーション は通常、ユーザーをログインページにリダイレクトします。認証用のユーザー名とパスワードを送信すると、ユーザーはアプリケーションにリダイレクトされます。これは、クライアントがブラウザーであるアプリケーションで適

切に機能します。その他のすべてのアプリケーションでは、ログインプロセスをプログラムで行う必要があります。プログラムによるログインに 必要なすべての手順 は、glogin コンポーネントによって実装されます。これらは以下のとおりです。

1. *ClientLogin API 経由で Google アカウント から認証トークンを取得します。*
2. *Google App Engine のログイン API から承認クッキーを取得します。*

その後、承認クッキーは後続の HTTP リクエストを GAE アプリケーションに送信する必要があります。24 時間後に期限切れになり、更新する必要があります。

URI 形式

> ***glogin://hostname[:port][?options]***

ホスト 名 は、GAE アプリケーションのインターネットホスト名（例： camelcloud.appspot.com）または 開発サーバー が実行されているホストの名前（例： localhost）です。この ポート は、開発サーバー(devMode=true の場合)に接続する場合にのみ使用され、オプションを参照してください。デフォルトは 8080 です。

オプション

名前	デフォルト値	必須	説明
clientName	**apache-camel-2.x**	false	**< organization>\-<appname>\-<version> の形式が推奨されるクライアント名** （必須ではありません）。
userName	**null**	true (**GLoginBinding.GLOGIN_USER_NAME** メッセージヘッダーで設定できます)	ログインユーザー名（メールアドレス）。
password	**null**	true (**GLoginBinding.GLOGIN_PASSWORD** メッセージヘッダーで設定することもできます)	ログインパスワード。

devMode	**false**	false	**true** に設定すると、開発サーバーへのログインが試行されます。
devAdmin	**false**	false	**true** に設定すると、admin ロールで開発サーバーへのログインが試行されます。

メッセージヘッダー

名前	タイプ	メッセージ	説明
GLoginBinding.GLOGIN_HOST_NAME	**文字列**	in	エンドポイント URI で定義されたホスト名を上書きします。
GLoginBinding.GLOGIN_USER_NAME	**文字列**	in	**userName** オプションを上書きします。
GLoginBinding.GLOGIN_PASSWORD	**文字列**	in	**パスワード** オプションを上書きします。
GLoginBinding.GLOGIN_TOKEN	**文字列**	out	Google アカウント から取得した認証トークンが含まれます。開発サーバーにログインしても、このヘッダーは設定されません。
GLoginBinding.GLOGIN_COOKIE	**文字列**	out	Google App Engine （または開発サーバー）から取得したアプリケーション固有の承認クッキーが含まれます。

メッセージボディー

***glogin** コンポーネントはメッセージの本文を読み書きしません。*

使用方法

*以下の JUnit テストは、開発サーバーと、**http://camelcloud.appspot.com** にあるデプロイされた GAE アプリケーションにログインする方法を示しています。*

GLOGINTEST.JAVA

```
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.ProducerTemplate;
import org.junit.Ignore;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import static org.apache.camel.component.gae.login.GLoginBinding.*;
import static org.junit.Assert.*;

public class GLoginTest {

    private ProducerTemplate template = ...

    @Test
    public void testDevLogin() {
        Exchange result = template.request("glogin://localhost:8888?
userName=test@example.org&devMode=true", null);
        assertNotNull(result.getOut().getHeader(GLOGIN_COOKIE));
    }

    @Test
    public void testRemoteLogin() {
        Exchange result = template.request("glogin://camelcloud.appspot.com",
new Processor() {
            public void process(Exchange exchange) throws Exception {
                exchange.getIn().setHeader(GLOGIN_USER_NAME,
"replaceme@gmail.com");
                exchange.getIn().setHeader(GLOGIN_PASSWORD, "replaceme");
            }
        });
        assertNotNull(result.getOut().getHeader(GLOGIN_COOKIE));
    }

}
```

開発サーバーにログイン時の承認クッキーは次のようになります。

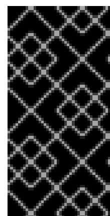
```
ahlogincookie=test@example.org:false:11223191102230730701;Path=/
```

デプロイされた GAE アプリケーションにログインした認証クッキーは、以下のように表示されます。

```
ACSID=AJKiYcE...XxhH9P_jR_V3; expires=Sun, 07-Feb-2010 15:14:51 GMT; path=/
```

54.5. GMAIL

Gmail コンポーネント



重要

GAE コンポーネントは非推奨となり、JBoss Fuse の今後のリリースで削除される予定です。

gmail コンポーネントは、Google App Engine (GAE)の Camel コンポーネントに提供します。GAE メール サービス を介したメールの送信をサポートします。メールの受信はまだサポートされていませんが、後で追加されます。現在、アプリケーション管理者がメールを送信できる Google アカウントのみ。

URI 形式

```
gmail://user@gmail.com[?options]
gmail://user@googlemail.com[?options]
```

オプション

名前	デフォルト値	コンテキスト	説明
上記を以下のように変更します。	null	プロデューサー	メールの to-receiver。これは、単一のレシーバーまたはコンマ区切りの受信側の一覧にすることができます。
cc	null	プロデューサー	メールの cc-receiver。これは、単一のレシーバーまたはコンマ区切りの受信側の一覧にすることができます。
bcc	null	プロデューサー	メールの bcc-receiver。これは、単一のレシーバーまたはコンマ区切りの受信側の一覧にすることができます。
subject	null	プロデューサー	メールの件名。

outboundBindingRef	**GMailBinding**への参照	プロデューサー	**エクスチェンジ** のメールサービスへのバインディングをカスタマイズするための レジストリー 内の **OutboundBinding<GMailEndpoint, MailService.Message, void** > への参照。

メッセージヘッダー

名前	タイプ	コンテキスト	説明
GMailBinding.GMAIL_SUBJECT	**文字列**	プロデューサー	メールの件名。**サブジェクト** エンドポイントオプションを上書きします。
GMailBinding.GMAIL_SENDER	**文字列**	プロデューサー	電子メールの送信者。エンドポイント URI の送信者定義を上書きします。
GMailBinding.GMAIL_TO	**文字列**	プロデューサー	メールの to-receiver (s) エンドポイントオプション **を上書きし** ます。
GMailBinding.GMAIL_CC	**文字列**	プロデューサー	メールの cc-receiver (s)**cc** endpoint オプションを上書きします。
GMailBinding.GMAIL_BCC	**文字列**	プロデューサー	メールの bcc-receiver (s)**bcc** エンドポイントオプションを上書きします。

メッセージボディー

プロデューサー側では、の メッセージボディーは String に変換されます。

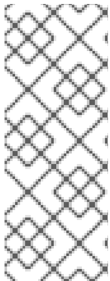
使用方法

```
...
.setHeader(GMailBinding.GMAIL_SUBJECT, constant("Hello"))
.setHeader(GMailBinding.GMAIL_TO, constant("account2@somewhere.com"))
.to("gmail://account1@gmail.com");
```

account1@gmail.com から account2@somewhere.com に、件名 Hello の付いたメールを送信します。メールメッセージのボディーは、メッセージ本文の から取得されます。account1@gmail.com は、現在の GAE アプリケーションの管理者アカウントである必要があることに注意してください。

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



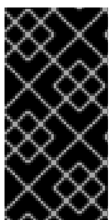
POM.XML

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-gae</artifactId>
   <version>${camel-version}</version>
</dependency>
```

ここで、$\{camel-version\} は Apache Camel の実際のバージョン(2.1.0 以降)に置き換える必要があります。

54.6. GSEC

Apache Camel GAE アプリケーションのセキュリティー



重要

GAE コンポーネントは非推奨となり、JBoss Fuse の今後のリリースで削除される予定です。

承認されていないアクセスからの GAE アプリケーションのセキュリティー保護は、Google App Engine ドキュメントの Security and Authentication セクションを参照してください。承認制約は web.xml ファイルで宣言されます(「web.xml」を参照してください)。これは Apache Camel アプリケーションにも適用されます。以下の例では、アプリケーションは、認証されたユーザー（任意のロール）のみがアプリケーションにアクセスできるように設定されます。さらに、/worker/* URL へのアクセスは、admin ロールを持つユーザーのみが実行できます。デフォルトでは、gtask コンポーネントによってインストールされた Web フック URL は /worker/* パターンに一致し、通常のユーザーがアクセスすることはできません。この認可制約では、タスクキューイングサービス（常に admin ロール）のみが Web フックにアクセスできます。カスタムで宣言的ではない承認ロジックを実装するには、Apache Camel GAE アプリケーションは Google Accounts Java API を使用する必要があります。

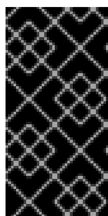
例54.1 web.xml （承認制約あり）

```
<web-app
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="
http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

    <servlet>
        <servlet-name>CamelServlet</servlet-name>
        <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>appctx.xml</param-value>
        </init-param>
    </servlet>

    <!--
        Mapping used for external requests
     -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/camel/*</url-pattern>
    </servlet-mapping>

    <!--
        Mapping used for web hooks accessed by task queueing service.
     -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/worker/*</url-pattern>
    </servlet-mapping>

    <!--
        By default allow any user who is logged in to access the whole
        application.
     -->
    <security-constraint>
        <web-resource-collection>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>*</role-name>
        </auth-constraint>
    </security-constraint>

    <!--
        Allow only admin users to access /worker/* URLs e.g. to prevent
        normal user to access gtask web hooks.
     -->
    <security-constraint>
        <web-resource-collection>
            <url-pattern>/worker/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
```

```
            <role-name>admin</role-name>
        </auth-constraint>
    </security-constraint>

</web-app>
```

54.7. GTASK

gtask コンポーネント



重要

GAE コンポーネントは非推奨となり、JBoss Fuse の今後のリリースで削除される予定です。

gtask コンポーネントは、Google App Engine (GAE)の Camel コンポーネントに提供します。これは、タスク キューをメッセージキューとして使用することで、GAE での非同期メッセージ処理をサポートします。キューへメッセージを追加するには、タスクキュー API を使用します。キューからメッセージを受信するには、HTTP コールバックハンドラーをインストールします。ハンドラーは、タスクキューサービスによって開始される HTTP POST コールバック(Web フック)によって呼び出されます。新しいタスクがキューに追加されるたびに、コールバックが送信されます。gtask コンポーネントはこれらの詳細から抽象化され、JMS または SEDA を使用したメッセージキューのように GAE でのメッセージキューを容易にするエンドポイント URI をサポートします。

URI 形式

```
gtask://queue-name
```

オプション

名前	デフォルト値	コンテキスト	説明

workerRoot	**worker**	プロデューサー	コールバックハンドラーのサーブレットマッピング。デフォルトでは、このコンポーネントには **/worker/* のコールバックサーブレットマッピングが必要です**。別のサーブレットマッピングを使用する場合（例： **/myworker/*** ）は、プロデューサー側でオプションとして設定する必要があります（ **to ("gtask:myqueue?workerRoot=myworker")**。
inboundBindingRef	**GTaskBinding**への参照	コンシューマー	**エクスチェンジ** のサーブレット API へのバインディングをカスタマイズするための **InboundBinding<GTaskEndpoint、HttpServletRequest、HttpServletResponse>** への参照。参照バインディングは、**org.apache.camel.component.http.HttpBinding** への後プロセッサーとして使用されます。
outboundBindingRef	**GTaskBinding**への参照	プロデューサー	**エクスチェンジ** のタスクキューサービスへのバインディングをカスタマイズするための Registry の **OutboundBinding<GTaskEndpoint, TaskOptions, void>** への参照。

コンシューマー側では、[Servlet コンポーネント] のすべてのオプションがサポートされます。

メッセージヘッダー

コンシューマサイドでは、[Servlet コンポーネントコンポーネント] のすべてのヘッダーと以下がサポートされます。

名前	タイプ	コンテキスト	説明
GTaskBinding.GTASK_QUEUE_NAME	**文字列**	コンシューマー	タスクキューの名前。
GTaskBinding.GTASK_TASK_NAME	**文字列**	コンシューマー	タスクの名前（生成された値）。
GTaskBinding.GTASK_RETRY_COUNT	**int**	コンシューマー	コールバックの再試行回数。

メッセージボディー

プロデューサー側では、メッセージボディーは byte[] に変換され、content-type application/octet-stream としてコールバックハンドラーに POST されます。

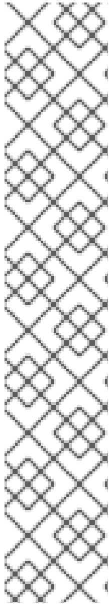
使用方法

タスクキューの設定は、Google App Engine の管理タスクです。事前設定されたキューは 1 つだけで、デフォルト のキューは追加設定なしで参照できます。このキューは以下の例で使用されます。ローカル開発サーバーでタスクキューを使用する場合は、開発 者コンソール からタスクを手動で実行する必要があります。

デフォルトのキュー

```
...
.to(gtask:default) // add message to default queue

from(gtask:default) // receive message from default queue (via a web hook)
...
```

この例では、以下のサーブレットマッピングが必要です。

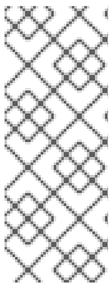


WEB.XML

```
...
<servlet>
    <servlet-name>CamelServlet</servlet-name>
    <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet
-class>
    ...
</servlet>
...
<servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/worker/*</url-pattern>
</servlet-mapping>
...
```

Dependencies

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



POM.XML

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-gae</artifactId>
    <version>${camel-version}</version>
</dependency>
```

${camel-version} は、実際のバージョンの Apache Camel (2.1.0 以降)に置き換える必要があります。

第55章 GANGLIA

GANGLIA コンポーネント

Camel 2.15.0 から利用可能

Ganglia コンポーネントは、gmetric4j ライブラリーを使用して、Ganglia モニターリングシステムに値（メッセージボディー）をメトリックとして送信するメカニズムを提供します。標準の Ganglia および JMXetric と併用して、1 つのプラットフォームでオペレーティングシステム、JVM、およびビジネスプロセスからメトリクスを監視できます。

JVM と同じマシン上で Ganglia gmond エージェントを実行している必要があります。エージェントは、現在 camel-ganglia が使用できない Ganglia インフラストラクチャーにハートビートを送信します。

ほとんどの Linux システム(EPEL、Fedora、Debian、および Ubuntu を持つ RHEL および CentOS) では、Ganglia エージェントパッケージのみをインストールできます。マルチキャストモードで自動的に実行されますが、通常の UDP ユニキャストモードで実行するように設定できます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-ganglia</artifactId>
   <version>2.17.0.redhat-630xxx</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
ganglia:address:port[?options]
```

URI にクエリーオプションを追加するには、?option=value&option=value&.. 形式を使用します。

GANGLIA エンドポイントオプション

名前	デフォルト	説明	w/ ヘッダーを上書きします。

<table>
<tr><td>mode</td><td>MULTICAST</td><td>UDP メトリクスパケットの送信に使用するアドレス指定モードを指定します。有効な値は MULTICAST または UNICAST です。</td><td></td></tr>
<tr><td>ttl</td><td>5</td><td>MULTICAST を使用する場合は、パケットの存続期間を指定します。</td><td></td></tr>
<tr><td>wireFormat31x</td><td>true</td><td>使用するワイヤ形式を指定します。true Wire 形式を Ganglia v3.1.0 以降に設定します。false Wire 形式を Ganglia v3.0.x 以前に設定します。</td><td></td></tr>
<tr><td>groupName</td><td>Java</td><td>メトリックが属するグループを指定します。</td><td></td></tr>
<tr><td>prefix</td><td></td><td>[オプション] metricName に追加する文字列接頭辞を指定します。接頭辞にアンダースコアが自動的に追加されます。</td><td></td></tr>
<tr><td>metricName</td><td>メトリクス</td><td>メトリクスに使用する名前を指定します。</td><td>GangliaConstants.METRIC_NAME</td></tr>
<tr><td>type</td><td>STRING</td><td>メトリックのタイプを指定します(STRING, INT8|16|32, ----------
|-----|----------|-----|
splunk, NORMAL, NORMAL) UINT81632 FLOAT DOUBLE</td><td>GangliaConstants.METRIC_TYPE</td></tr>
</table>

slope	**BOTH**	メトリックのデータの保存方法を決定するメトリックの有効期間のスライプを指定します。有効な値は以下のとおりです。 ● **ZERO**- 値が変更されない ● **POSITIVE**- 値を増やすか、同じままにする ● **NEGATIVE**- 値は減少するか、同じままにする ● **BOTH**- 値は増減できます。	**GangliaConstants.METRIC_SLOPE**
units		[オプション] メトリックの値をクォレートする測定単位を指定します（バイト、秒、チェンス、スレーターなど）。 その他のツールは後で行う可能性があるため、接頭辞(**k** [kilo]、**m** [milli] など)を使用してユニットをスケーリングすることはできません。値もスケーリングする必要があります。	**GangliaConstants.METRIC_UNITS**
tmax	**60**	ギーメトリック呼び出し間の最大時間を秒単位で指定します。**tmax** 以降、Ganglia は現在の値の有効期限が切れていると見なします。	**GangliaConstants.METRIC_TMAX**
dmax	**0**	指定されたメトリックの有効期間を秒単位で指定します。	**GangliaConstants.METRIC_DMAX**

メッセージボディー

メッセージボディーの文字列や数値のタイプなどの値は、Ganglia モニターリングシステムに送信されます。

戻り値/レスポンス

Ganglia は、一方向 UDP またはマルチキャストを使用してメトリックを送信します。メッセージのボディーには応答や変更はありません。

文字列メトリクスの送信

メッセージ本文は String に変換され、メトリクス値として送信されます。数値のメトリクスとは異なり、String の値は表示できませんが、Ganglia によりレポートが利用できるようになります。すべての Ganglia ホストページの上部にある os_version 文字列は、String メトリクスの例です。

```
from("direct:string.for.ganglia")
    .setHeader(GangliaConstants.METRIC_NAME, simple("my_string_metric"))
    .setHeader(GangliaConstants.METRIC_TYPE, GMetricType.STRING)
    .to("direct:ganglia.tx");

from("direct:ganglia.tx")
    .to("ganglia:239.2.11.71:8649?mode=MULTICAST&prefix=test");
```

数値メトリックの送信

```
from("direct:value.for.ganglia")
    .setHeader(GangliaConstants.METRIC_NAME, simple("widgets_in_stock"))
    .setHeader(GangliaConstants.METRIC_TYPE, GMetricType.UINT32)
    .setHeader(GangliaConstants.METRIC_UNITS, simple("widgets"))
    .to("direct:ganglia.tx");

from("direct:ganglia.tx")
    .to("ganglia:239.2.11.71:8649?mode=MULTICAST&prefix=test");
```

第56章 GEOCODER

GEOCODER コンポーネント

Camel 2.12 以降で利用可能

geocoder: コンポーネントは、特定のアドレスまたは逆引きルックアップの地理コード(latitude および longitude)を検索するために使用されます。コンポーネントは、Google Geocoder ライブラリーに Java API を使用します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-geocoder</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
geocoder:address:name[?options]
geocoder:latlng:latitude,longitude[?options]
```

オプション

プロパティー	デフォルト	説明
言語	**en**	使用する言語。

headersOnly	**false**	ヘッダーで エクスチェンジのみを補完 し、ボディーをそのまま残すかどうか。
clientId		このクライアント ID で google Premium を使用するには、以下を実行します。
clientKey		このクライアントキーで google Premium を使用するには、以下を行います。
httpClientConfigurer	**null**	Camel 2.17: レジストリーの **org.apache.camel.component.geocoder.http.HttpClientConfigurer** への参照。
clientConnectionManager	**null**	Camel 2.17: カスタム **org.apache.http.conn.ClientConnectionManager** を使用します。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

PROXY

以下のプロキシーオプションを Geocoder エンドポイントに設定することもできます。

プロパティー	デフォルト	説明
proxyHost	**null**	Camel 2.17: プロキシーのホスト名。
proxyPort	**null**	Camel 2.17: プロキシーポート番号。
proxyAuthMethod	**null**	Camel 2.17: プロキシーの認証メソッド（**Basic** または **Digest** のいずれか。**NTLM**
proxyAuthUsername	**null**	Camel 2.17: プロキシー認証のユーザー名。
proxyAuthPassword	**null**	Camel 2.17: プロキシー認証のパススルー。

proxyAuthDomain	**null**	Camel 2.17: プロキシー NTML 認証のドメイン。
proxyAuthHost	**null**	Camel 2.17: プロキシー NTML 認証用のオプションのホスト。

エクスチェンジデータ形式

Camel はボディーを com.google.code.geocoder.model.GeocodeResponse タイプとして配信します。また、アドレスが current の場合、応答は現在 の場所の JSON 表現を持つ String 型になります。

オプション headersOnly が true に設定されている場合、メッセージボディーはそのまま残り、ヘッダーのみが Exchange に追加されます。

メッセージヘッダー

ヘッダー	説明
CamelGeoCoderStatus	必須。ジオコーダーライブラリーのステータスコード。ステータスが **GeocoderStatus.OK** の場合、追加のヘッダーが補完されます。
CamelGeoCoderAddress	フォーマットされたアドレス
CamelGeoCoderLat	場所のお気に入り。
CamelGeoCoderLng	場所が長くなります。
CamelGeoCoderLatlng	場所のお気に入りと長い場所です。コンマで区切ります。
CamelGeoCoderCity	都市の長い名前。
CamelGeoCoderRegionCode	リージョンコード。
CamelGeoCoderRegionName	リージョン名。

CamelGeoCoderCountryLong	国の長い名前。
CamelGeoCoderCountryShort	国の短縮名。

使用中の利用可能なデータおよびモード（アドレス対 latlng）によっては、すべてのヘッダーが提供されるわけではないことに注意してください。

サンプル

以下の例では、Paris、France の latitude と longitude を取得しています。

```
from("direct:start")
  .to("geocoder:address:Paris, France")
```

CamelGeoCoderAddress でヘッダーを指定すると、エンドポイント設定が上書きされるため、Copenhagen の場所を取得するには、以下のようにヘッダーでメッセージを送信できます。

```
template.sendBodyAndHeader("direct:start", "Hello", GeoCoderConstants.ADDRESS,
"Copenhagen, Denmark");
```

latitude と longitude のアドレスを取得するには、以下を実行できます。

```
from("direct:start")
  .to("geocoder:latlng:40.714224,-73.961452")
  .log("Location ${header.CamelGeocoderAddress} is at lat/lng:
${header.CamelGeocoderLatlng} and in country ${header.CamelGeoCoderCountryShort}")
```

ログを作成する

```
Location 285 Bedford Avenue, Brooklyn, NY 11211, USA is at lat/lng: 40.71412890,-
73.96140740 and in country US
```

現在の場所を取得するには、以下のように current をアドレスとして使用します。

```
from("direct:start")
  .to("geocoder:address:current")
```

第57章 GIT

GIT コンポーネント

Camel 2.16 以降で利用可能

Git コンポーネントを使用すると、汎用 Git リポジトリーを操作することができます。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-git</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
git://localRepositoryPath[?options]
```

URI オプション

プロデューサーは特定のリポジトリーで操作を行うことができます。コンシューマーは特定のリポジトリーでコミット、タグ、およびブランチを使用できます。

名前	デフォルト値	タイプ	コンテキスト	説明
localPath	**null**	文字列	共有	ローカル Git リポジトリーへのパス
remotePath	**null**	文字列	共有	リモート Git リポジトリーへのパス

operation	**null**	文字列	プロデューサー	実行する操作。現在、以下の値をサポートしています。 **clone, init, add, remove, commit, commitAll, createBranch, deleteBranch, createTag, deleteTag, status, log, push, pull, showBranches, cherryPick.**
branchName	**null**	文字列	プロデューサー	作業するブランチの名前
tagName	**null**	文字列	プロデューサー	作業するブランチのタグ
username	**null**	文字列	プロデューサー	git リポジトリーの認証フェーズで使用するユーザー名
password	**null**	文字列	プロデューサー	git リポジトリーの認証フェーズで使用するパスワード
type	**null**	文字列	コンシューマー	コンシューマーのタイプ。現在、以下の値をサポートしています。 **commit, tag, branch**

メッセージヘッダー

名前	デフォルト値	タイプ	コンテキスト	説明
CamelGitOperation	**null**	文字列	プロデューサー	エンドポイントオプションとして指定されていない場合に、リポジトリーで実行する操作
CamelGitFilename	**null**	文字列	プロデューサー	add 操作のファイル名

CamelGitCommitMessage	**null**	文字列	プロデューサー	コミット操作に関連するコミットメッセージ
CamelGitCommitUsername	**null**	文字列	プロデューサー	コミット操作のコミットユーザー名
CamelGitCommitEmail	**null**	文字列	プロデューサー	コミット操作のメール
CamelGitCommitId	**null**	文字列	プロデューサー	コミット ID

プロデューサーの例

以下は、ファイル test.java をローカルリポジトリーに追加し、master ブランチに特定のメッセージでコミットしてからリモートリポジトリーにプッシュするプロデューサーのルート例です。

```
from("direct:start")
        .setHeader(GitConstants.GIT_FILE_NAME, constant("test.java"))
        .to("git:///tmp/testRepo?operation=add")
        .setHeader(GitConstants.GIT_COMMIT_MESSAGE, constant("first commit"))
        .to("git:///tmp/testRepo?operation=commit")
        .to("git:///tmp/testRepo?
operation=push&remotePath=https://foo.com/test/test.git&username=xxx&password=xxx")
```

コンシューマーの例

以下は、コミットを使用するコンシューマーのルート例です。

```
from("git:///tmp/testRepo?type=commit")
                        .to(....)
```

第58章 GITHUB

GITHUB コンポーネント

Camel 2.15 以降で利用可能

GitHub コンポーネントは、egit-github をカプセル化して GitHub API と対話します。現在、新しいプル要求、プル要求のコメント、タグ、およびコミットのポーリングが行われています。プル要求にコメントを作成したり、プルリクエストを完全に閉じることもできます。

Webhook ではなく、このエンドポイントは単純なポーリングに依存します。理由は次のとおりです。

- 信頼性/安定性の懸念
- ポーリングしているペイロードのタイプは通常大きくありません（上向き、ページングは API で利用可能です）。
- Webhook が失敗した場合にパブリックにアクセスできない一部のアプリケーションをサポートする必要があります。

GitHub API はかなり大きくなることに注意してください。そのため、このコンポーネントは簡単に拡張でき、追加の対話を提供できます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-github</artifactId>
   <version>${camel-version}</version>
</dependency>
```

URI 形式

```
github://endpoint[?options]
```

必須オプション：

これらは エンドポイントで直接設定できることに注意してください。

オプション	説明
username	GitHub ユーザー名。**oauthToken** が指定されていない場合に必要です。
password	GitHub パスワード(**oauthToken** が提供されない限り必要)
oauthToken	GitHub OAuth トークン。**username** および **password** が指定されていない場合に必要です。
repoOwner	GitHub リポジトリーの所有者（組織）。
repoName	GitHub リポジトリー名。

コンシューマーエンドポイント：

エンドポイント	コンテキスト	ボディタイプ
pullRequest	ポーリング	org.eclipse.egit.github.core.PullRequest
pullRequestComment	ポーリング	org.eclipse.egit.github.core.Comment （一般的なプルリクエストに関する議論）または org.eclipse.egit.github.core.CommitComment （プルリクエスト差分のインラインコメント）
tag	ポーリング	org.eclipse.egit.github.core.RepositoryTag
commit	ポーリング	org.eclipse.egit.github.core.RepositoryCommit

プロデューサーエンドポイント：

エンドポイント	本文	メッセージヘッダー

pullRequestComment	string （コメントテキスト）	• GitHubPullRequest （整数）(REQUIRED): プル要求番号。 • GitHubInResponseTo （整数）: プル要求の差分で別のインラインコメントに応答する場合に必須です。オフにすると、プル要求の議論に関する一般的なコメントが想定されます。
closePullRequest	none	• GitHubPullRequest （整数）(REQUIRED): プル要求番号。

URI オプション

名前	デフォルト値	説明
delay	**60**	秒単位

第59章 GOOGLECALENDAR

GOOGLECALENDAR コンポーネント

Camel 2.15 以降で利用可能

コンポーネントの説明

Google Calendar コンポーネントは、Google カレンダー Web API 経由で Google カレンダー へのアクセスを提供します。

Google カレンダーは、OAuth 2.0 プロトコル を使用して Google アカウントを認証し、ユーザーデータへのアクセスを承認します。このコンポーネントを使用する前に、アカウントを作成し、OAuth クレデンシャル を生成 する必要があります。認証情報は、clientId、clientSecret、および refreshToken で設定されます。有効期間の長い refreshToken を生成するための便利なリソースは OAuth プレイグラウンドです。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-google-calendar</artifactId>
        <version>2.17.0.redhat-630xxx</version>
</dependency>
```

URI 形式

GoogleCalendar コンポーネントは以下の URI 形式を使用します。

```
google-calendar://endpoint-prefix/endpoint?[options]
```

エンドポイント接頭辞は以下のいずれかになります。

- acl
- calendars

- *channels*
- *colors*
- *events*
- *freebusy*
- *list*
- *settings*

GOOGLECALENDARCOMPONENT

GoogleCalendar コンポーネントは、以下のオプションで設定できます。これらのオプションは、タイプ org.apache.camel.component.google.calendar.GoogleCalendarConfiguration のコンポーネントの Bean プロパティー configuration を使用して提供できます。

オプション	タイプ	
accessToken	**String**	OAuth 2 アクセストークン通常、これ が推奨されます。
applicationName	**String**	Google カレンダーアプリケーション
clientId	**String**	カレンダーアプリケーションのクラ
clientSecret	**String**	カレンダーアプリケーションのクラ
refreshToken	**String**	OAuth2 更新トークン。このトークン 切れるたびに新しい **accessToken** ます。
scopes	**List<String>**	カレンダーアプリケーションでユー は、https://developers.google.com/
emailAddress	**String**	**Camel 2.16.0:** Google Service Accou

p12FileName	String	Camel 2.16.0: Google サービスアカ

プロデューサーエンドポイント

プロデューサーエンドポイントはエンドポイント接頭辞を使用し、続いてエンドポイント名と以下で説明する関連オプションを使用できます。一部のエンドポイントには、短縮エイリアスを使用できます。エンドポイント URI には 接頭辞が含まれている必要があります。

必須ではないエンドポイントオプションは [] で示されます。エンドポイントに必須のオプションがない場合は、[] オプションのセットの 1 つを指定する必要があります。プロデューサーエンドポイントは、Camel Exchange In メッセージに含まれる値を持つ endpoint オプションの名前が含まれる必要がある特別なオプション inBody を使用することもできます。

エンドポイントオプションは、エンドポイント URI またはメッセージヘッダーで動的に指定できます。メッセージヘッダー名は CamelGoogleCalendar.<option> の形式である必要があります。inBody オプションはメッセージヘッダーを上書きすることに注意してください。つまり、エンドポイントオプション inBody=option は CamelGoogleCalendar.option ヘッダーを上書きすることに注意してください。

1.エンドポイント接頭辞 ACL

以下のエンドポイントは、以下のように接頭辞 acl で呼び出すことができます。

> *google-calendar://acl/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		calendarId, ruleId	
get		calendarId, ruleId	com.google.api.services.calendar.model.AclRule
insert		calendarId, content	com.google.api.services.calendar.model.AclRule
list		calendarId	**com.google.api.services.calendar.model.Acl**
patch		calendarId、 content、ruleId	com.google.api.services.calendar.model.AclRule

update		calendarId、content、ruleId	com.google.api.services.calendar.model.AclRule
watch		calendarId, contentChannel	com.google.api.services.calendar.model.Channel

ACLの URI オプション

名前	タイプ
calendarId	文字列
content	com.google.api.services.calendar.model.AclRule
contentChannel	com.google.api.services.calendar.model.Channel
ruleId	文字列

2.エンドポイント接頭辞 カレンダー

以下のエンドポイントは、以下のように接頭辞 calendars で呼び出すことができます。

> ***google-calendar://calendars/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
clear		calendarId	
delete		calendarId	
get		calendarId	**com.google.api.services.calendar.Calendar**
insert		content	**com.google.api.services.calendar.Calendar**
patch		calendarId, content	**com.google.api.services.calendar.Calendar**

update		calendarId, content	**com.google.api.services.calendar.Calendar**

カレンダーの URI オプション

名前	タイプ
calendarId	文字列
content	com.google.api.services.calendar.model.Calendar

3.エンドポイント接頭辞 チャネル

以下のエンドポイントは、以下のように接頭辞 channels で呼び出すことができます。

google-calendar://channels/endpoint?[options]

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
stop		contentChannel	

チャネルの URI オプション

名前	タイプ
contentChannel	com.google.api.services.calendar.model.Channel

4.エンドポイント接頭辞の 色

以下のエンドポイントは、以下のように接頭辞 colors で呼び出すことができます。

google-calendar://colors/endpoint?[options]

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
get			**com.google.api.services.calendar.model.Colors**

色の URI オプション

名前	タイプ

5.エンドポイント接頭辞 イベント

以下のエンドポイントは、以下のように接頭辞 events で呼び出すことができます。

```
google-calendar://events/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
calendarImport		calendarId, content	com.google.api.services.calendar.model.Event
delete		calendarId, eventId	
get		calendarId, eventId	com.google.api.services.calendar.model.Event
insert		calendarId, content	com.google.api.services.calendar.model.Event
instances		calendarId, eventId	**com.google.api.services.calendar.model.Events**
list		calendarId	**com.google.api.services.calendar.model.Events**
move		calendarId、destination、eventId	com.google.api.services.calendar.model.Event
patch		calendarId、content、eventId	com.google.api.services.calendar.model.Event
quickAdd		calendarId, text	com.google.api.services.calendar.model.Event
update		calendarId、content、eventId	com.google.api.services.calendar.model.Event
watch		calendarId, contentChannel	com.google.api.services.calendar.model.Channel

イベントの URI オプション

名前	タイプ
calendarId	文字列
content	com.google.api.services.calendar.model.Event
contentChannel	com.google.api.services.calendar.model.Channel
destination	文字列
eventId	文字列
text	文字列

6.エンドポイント接頭辞 FREEBUSY

*以下のエンドポイントは、以下のように接頭辞 **freebusy** で呼び出すことができます。*

> ***google-calendar://freebusy/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
query		content	com.google.api.services.calendar.model.FreeBusyResponse

FREEBUSY*の URI オプション*

名前	タイプ
content	com.google.api.services.calendar.model.FreeBusyRequest

7.ENDPOINT PREFIX LIST

*以下のエンドポイントは、以下のように接頭辞 **list** で呼び出すことができます。*

> ***google-calendar://list/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		calendarId	
get		calendarId	**com.google.api.services.calendar.model.CalendarListEntry**
insert		content	**com.google.api.services.calendar.model.CalendarListEntry**
list			**com.google.api.services.calendar.model.CalendarList**
patch		calendarId, content	**com.google.api.services.calendar.model.CalendarListEntry**
update		calendarId, content	**com.google.api.services.calendar.model.CalendarListEntry**
watch		contentChannel	com.google.api.services.calendar.model.Channel

LIST*の URI オプション*

名前	タイプ
calendarId	文字列
content	com.google.api.services.calendar.model.CalendarListEntry
contentChannel	com.google.api.services.calendar.model.Channel

***8.エンドポイント接頭辞の* 設定**

以下のエンドポイントは、以下のように接頭辞* settings *で呼び出すことができます。

> ***google-calendar://settings/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ

get		設定	**com.google.api.services.calendar.model.Setting**
list			**com.google.api.services.calendar.model.Settings**
watch		contentChannel	com.google.api.services.calendar.model.Channel

設定の *URI オプション*

名前	タイプ
contentChannel	com.google.api.services.calendar.model.Channel
設定	文字列

コンシューマーエンドポイント

プロデューサーエンドポイントはいずれもコンシューマーエンドポイントとして使用できます。コンシューマーエンドポイントは、consumer. 接頭辞が付いた Scheduled Poll Consumer オプション を使用して、エンドポイントの呼び出しをスケジュールできます。配列またはコレクションを返すコンシューマーエンドポイントは、要素ごとにエクスチェンジを 1 つ生成し、それらのルートはエクスチェンジごとに 1 回実行されます。

メッセージヘッダー

URI オプションは、CamelGoogleCalendar. 接頭辞が付いたプロデューサーエンドポイントのメッセージヘッダーで指定できます。

メッセージボディー

すべての結果メッセージ本文は、GoogleCalendarComponent によって使用される基礎となる API によって提供されるオブジェクトを使用します。プロデューサーエンドポイントは、inBody エンドポイント URI パラメーターに受信メッセージボディーのオプション名を指定できます。配列またはコレクションを返すエンドポイントの場合、コンシューマーエンドポイントはすべての要素を個別のメッセージにマップします。

第60章 GOOGLEDRIVE

GOOGLEDRIVE COMPONENT

Camel 2.14 から利用可能

Google Drive コンポーネントは、Google Drive Web API 経由で Google ドライブファイルストレージサービス へのアクセスを提供します。

Google ドライブは、OAuth 2.0 プロトコル を使用して Google アカウントを認証し、ユーザーデータへのアクセスを承認します。このコンポーネントを使用する前に、アカウントを作成し、OAuth クレデンシャル を生成 する必要があります。認証情報は、clientId、clientSecret、および refreshToken で設定されます。有効期間の長い refreshToken を生成するための便利なリソースは OAuth プレイグラウンドです。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-google-drive</artifactId>
        <version>2.14-SNAPSHOT</version>
</dependency>
```

URI 形式

GoogleDrive コンポーネントは以下の URI 形式を使用します。

```
google-drive://endpoint-prefix/endpoint?[options]
```

エンドポイント接頭辞は以下のいずれかになります。

- drive-about
- drive-apps

- *drive-changes*
- *drive-channels*
- *drive-children*
- *drive-comments*
- *drive-files*
- *drive-parents*
- *drive-permissions*
- *drive-properties*
- *drive-realtime*
- *drive-replies*
- *drive-revisions*

GOOGLEDRIVECOMPONENT

GoogleDrive コンポーネントは、以下のオプションで設定できます。これらのオプションは、タイプ org.apache.camel.component.google.drive.GoogleDriveConfiguration のコンポーネントの Bean プロパティー configuration を使用して提供できます。

オプション	タイプ	説明

accessToken	String	OAuth 2 アクセストークン通常、これは 1 時間後に期限切れになるため、長期間の使用には **refreshToken** が推奨されます。
applicationName	String	Google ドライブアプリケーション名。たとえば、**camel-google-drive/1.0** です。
clientId	String	ドライブアプリケーションのクライアント ID
clientSecret	String	ドライブアプリケーションのクライアントシークレット
refreshToken	String	OAuth 2 トークンの更新これを使用すると、現在の有効期限が切れるたびに Google ドライブコンポーネントが新しい **accessToken** を取得することができます。アプリケーションが長い期間であれば必要です。
scopes	List<String>	ドライブアプリケーションでユーザーアカウントに必要なパーミッションのレベルを指定します。詳細は、https://developers.google.com/drive/web/scopes を参照してください。

プロデューサーエンドポイント

プロデューサーエンドポイントはエンドポイント接頭辞を使用し、続いてエンドポイント名と以下で説明する関連オプションを使用できます。一部のエンドポイントには、短縮エイリアスを使用できます。エンドポイント URI には 接頭辞が含まれている必要があります。

必須ではないエンドポイントオプションは [] で示されます。エンドポイントに必須のオプションがない場合は、[] オプションのセットの 1 つを指定する必要があります。プロデューサーエンドポイントは、Camel Exchange In メッセージに含まれる値を持つ endpoint オプションの名前が含まれる必要がある特別なオプション inBody を使用することもできます。

エンドポイントオプションは、エンドポイント URI またはメッセージヘッダーで動的に指定できます。メッセージヘッダー名は CamelGoogleDrive.<option> の形式である必要があります。inBody オプションはメッセージヘッダーを上書きすることに注意してください。つまり、エンドポイントオプ

ション inBody=option は CamelGoogleDrive.option ヘッダーを上書きすることに注意してください。

エンドポイントおよびオプションの詳細は、API ドキュメント(https://developers.google.com/drive/v2/reference/)を参照してください。

1.エンドポイント接頭辞 DRIVE-ABOUT

以下のエンドポイントは、以下のように接頭辞 drive-about で呼び出すことができます。

```
google-drive://drive-about/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
get			com.google.api.services.drive.model.About

DRIVE-ABOUT の URI オプション

名前	タイプ

2.エンドポイント接頭辞 DRIVE-APPS

以下のエンドポイントは、以下のように接頭辞 drive-apps で呼び出すことができます。

```
google-drive://drive-apps/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
get		appId	com.google.api.services.drive.model.App
list			com.google.api.services.drive.model.AppList

DRIVE-APPS の URI オプション

名前	タイプ
appId	String

3.エンドポイント接頭辞 DRIVE-CHANGES

以下のエンドポイントは、以下のように接頭辞 drive-changes で呼び出すことができます。

```
google-drive://drive-changes/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
get		changeId	com.google.api.services.drive.model.Change
list			com.google.api.services.drive.model.ChangeList
watch		contentChannel	com.google.api.services.drive.model.Channel

DRIVE-CHANGES の URI オプション

名前	タイプ
changeId	String
contentChannel	com.google.api.services.drive.model.Channel

4.エンドポイント接頭辞ドライブチャネル

以下のエンドポイントは、以下のように接頭辞 drive-channels で呼び出すことができます。

```
google-drive://drive-channels/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ

stop		contentChannel	

DRIVE-CHANNEL の URI オプション

名前	タイプ
contentChannel	com.google.api.services.drive.model.Channel

5.エンドポイント接頭辞 DRIVE-CHILDREN

以下のエンドポイントは、以下のように接頭辞 drive-children で呼び出すことができます。

> *google-drive://drive-children/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		childId, folderId	
get		childId, folderId	com.google.api.services.drive.model.ChildReference
insert		content, folderId	com.google.api.services.drive.model.ChildReference
list		folderId	com.google.api.services.drive.model.ChildList

DRIVE-CHILDREN の URI オプション

名前	タイプ
childId	String
content	com.google.api.services.drive.model.ChildReference
folderId	String

6.エンドポイント接頭辞ドライブ

*以下のエンドポイントは、以下のように接頭辞 **drive-comments** で呼び出すことができます。*

> *google-drive://drive-comments/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		commentId, fileId	
get		commentId, fileId	com.google.api.services.drive.model.Comment
insert		content, fileId	com.google.api.services.drive.model.Comment
list		fileId	com.google.api.services.drive.model.CommentList
patch		commentId, content, fileId	com.google.api.services.drive.model.Comment
update		commentId, content, fileId	com.google.api.services.drive.model.Comment

DRIVE-COMMENT の URI オプション

名前	タイプ
commentId	String
content	com.google.api.services.drive.model.Comment
fileId	String

7.エンドポイント接頭辞ドライブファイル

*以下のエンドポイントは、以下のように接頭辞 **drive-files** で呼び出すことができます。*

google-drive://drive-files/endpoint?[options]

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
copy		content, fileId	com.google.api.services.drive.model.File
delete		fileId	
emptyTrash			
get		fileId	com.google.api.services.drive.model.File
insert		[mediaContent], content	com.google.api.services.drive.model.File
list			com.google.api.services.drive.model.FileList
patch		content, fileId	com.google.api.services.drive.model.File
touch		fileId	com.google.api.services.drive.model.File
trash		fileId	com.google.api.services.drive.model.File
untrash		fileId	com.google.api.services.drive.model.File
update		[mediaContent], content, fileId	com.google.api.services.drive.model.File
watch		contentChannel, fileId	com.google.api.services.drive.model.Channel

DRIVE-FILES の URI オプション

名前	タイプ
content	com.google.api.services.drive.model.File

contentChannel	com.google.api.services.drive.model.Channel
fileId	String
mediaContent	com.google.api.client.http.AbstractInputStreamContent

8.エンドポイント接頭辞 DRIVE-PARENTS

以下のエンドポイントは、以下のように接頭辞 drive-parents で呼び出すことができます。

```
google-drive://drive-parents/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		fileId, parentId	
get		fileId, parentId	com.google.api.services.drive.model.ParentReference
insert		content, fileId	com.google.api.services.drive.model.ParentReference
list		fileId	com.google.api.services.drive.model.ParentList

DRIVE-PARENTS の URI オプション

名前	タイプ
content	com.google.api.services.drive.model.ParentReference
fileId	String
parentId	String

9.エンドポイント接頭辞 DRIVE-PERMISSIONS

*以下のエンドポイントは、以下のように接頭辞 **drive-permissions** で呼び出すことができます。*

> ***google-drive://drive-permissions/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		**fileId, permissionId**	
get		**fileId, permissionId**	**com.google.api.services.drive.model.Permission**
getIdForEmail		**email**	**com.google.api.services.drive.model.PermissionId**
insert		**content, fileId**	**com.google.api.services.drive.model.Permission**
list		**fileId**	**com.google.api.services.drive.model.PermissionList**
patch		**content, fileId, permissionId**	**com.google.api.services.drive.model.Permission**
update		**content, fileId, permissionId**	**com.google.api.services.drive.model.Permission**

DRIVE-PERMISSIONS の URI オプション

名前	タイプ
content	**com.google.api.services.drive.model.Permission**
email	**String**
fileId	**String**
permissionId	**String**

10.エンドポイント接頭辞 DRIVE-PROPERTIES

以下のエンドポイントは、以下のように接頭辞 drive-properties で呼び出すことができます。

```
google-drive://drive-properties/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		fileId, propertyKey	
get		fileId, propertyKey	com.google.api.services.drive.model.Property
insert		content, fileId	com.google.api.services.drive.model.Property
list		fileId	com.google.api.services.drive.model.PropertyList
patch		content, fileId, propertyKey	com.google.api.services.drive.model.Property
update		content, fileId, propertyKey	com.google.api.services.drive.model.Property

DRIVE-PROPERTIES の URI オプション

名前	タイプ
content	com.google.api.services.drive.model.Property
fileId	String
propertyKey	String

11.エンドポイント接頭辞 DRIVE-REALTIME

以下のエンドポイントは、以下のように接頭辞 drive-realtime で呼び出すことができます。

```
google-drive://drive-realtime/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
get		fileId	
update		[mediaContent], fileId	

DRIVE-REALTIME の URI オプション

名前	タイプ
fileId	String
mediaContent	com.google.api.client.http.AbstractInputStreamContent

12.エンドポイント接頭辞ドライブ

以下のエンドポイントは、以下のように接頭辞 drive-replies で呼び出すことができます。

> *google-drive://drive-replies/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		commentId, fileId, replyId	
get		commentId, fileId, replyId	com.google.api.services.drive.model.CommentReply
insert		commentId, content, fileId	com.google.api.services.drive.model.CommentReply
list		commentId, fileId	com.google.api.services.drive.model.CommentReplyList
patch		commentId, content, fileId, replyId	com.google.api.services.drive.model.CommentReply

update		commentId, content, fileId, replyId	com.google.api.services.drive.model.CommentReply

DRIVE-REPLIES の URI オプション

名前	タイプ
commentId	String
content	com.google.api.services.drive.model.CommentReply
fileId	String
replyId	String

13.エンドポイント接頭辞 DRIVE-REVISIONS

以下のエンドポイントは、以下のように接頭辞 drive-revisions で呼び出すことができます。

```
google-drive://drive-revisions/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		fileId, revisionId	
get		fileId, revisionId	com.google.api.services.drive.model.Revision
list		fileId	com.google.api.services.drive.model.RevisionList
patch		content, fileId, revisionId	com.google.api.services.drive.model.Revision
update		content, fileId, revisionId	com.google.api.services.drive.model.Revision

DRIVE-REVISIONS の URI オプション

名前	タイプ
content	com.google.api.services.drive.model.Revision
fileId	String
revisionId	String

コンシューマーエンドポイント

*プロデューサーエンドポイントはいずれもコンシューマーエンドポイントとして使用できます。コンシューマーエンドポイントは、**consumer.** 接頭辞が付いた Scheduled Poll Consumer オプション を使用して、エンドポイントの呼び出しをスケジュールできます。配列またはコレクションを返すコンシューマーエンドポイントは、要素ごとにエクスチェンジを 1 つ生成し、それらのルートはエクスチェンジごとに 1 回実行されます。*

メッセージヘッダー

*URI オプションは、**CamelGoogleDrive.** 接頭辞が付いたプロデューサーエンドポイントのメッセージヘッダーで指定できます。*

メッセージボディー

*すべての結果メッセージ本文は、GoogleDriveComponent が使用する基礎となる API によって提供されるオブジェクトを使用します。プロデューサーエンドポイントは、**inBody** エンドポイント URI パラメーターに受信メッセージボディーのオプション名を指定できます。配列またはコレクションを返すエンドポイントの場合、コンシューマーエンドポイントはすべての要素を個別のメッセージにマップします。*

第61章 GOOGLEMAIL

GOOGLEMAIL コンポーネント

Camel 2.15 以降で利用可能

コンポーネントの説明

Google Mail コンポーネントは、Google Mail Web API 経由で Gmail へのアクセスを提供します。

Google Mail は OAuth 2.0 プロトコル を使用して Google アカウントを認証し、ユーザーデータへのアクセスを承認します。このコンポーネントを使用する前に、アカウントを作成し、OAuth クレデンシャル を生成 する必要があります。認証情報は、clientId、clientSecret、および refreshToken で設定されます。有効期間の長い refreshToken を生成するための便利なリソースは OAuth プレイグラウンドです。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-google-mail</artifactId>
        <version>2.17.0.redhat-630xxx</version>
</dependency>
```

URI 形式

GoogleMail コンポーネントは以下の URI 形式を使用します。

```
google-mail://endpoint-prefix/endpoint?[options]
```

エンドポイント接頭辞は以下のいずれかになります。

- attachments
- ドラフト

- *history*
- *labels*
- *messages*
- *threads*
- *users*

GOOGLEMAILCOMPONENT

GoogleMail コンポーネントは、以下のオプションで設定できます。これらのオプションは、タイプ org.apache.camel.component.google.mail.GoogleMailConfiguration のコンポーネントの Bean プロパティー configuration を使用して提供できます。

オプション	タイプ	
accessToken	**String**	OAuth2 アクセストークン。通常、これは 1 時間 ます。
applicationName	**String**	Google ドライブアプリケーション名。たとえば
clientId	**String**	ドライブアプリケーションのクライアント ID。
clientSecret	**String**	ドライブアプリケーションのクライアントシーク
refreshToken	**String**	OAuth2 更新トークン。このトークンを使用する **accessToken** を取得できます。
scopes	**List<String>**	ドライブアプリケーションでユーザーアカウント は、https://developers.google.com/gmail/api/a
user	**String**	**Camel 2.16.0:** サービスアカウントフローでアプ
p12FileName	**String**	**Camel 2.16.0:** Google サービスアカウントで使用

プロデューサーエンドポイント

プロデューサーエンドポイントはエンドポイント接頭辞を使用し、続いてエンドポイント名と以下で説明する関連オプションを使用できます。一部のエンドポイントには、短縮エイリアスを使用できます。エンドポイント URI には 接頭辞が含まれている必要があります。

必須ではないエンドポイントオプションは [] で示されます。エンドポイントに必須のオプションがない場合は、[] オプションのセットの 1 つを指定する必要があります。プロデューサーエンドポイントは、Camel Exchange In メッセージに含まれる値を持つ endpoint オプションの名前が含まれる必要がある特別なオプション inBody を使用することもできます。

エンドポイントオプションは、エンドポイント URI またはメッセージヘッダーで動的に指定できます。メッセージヘッダー名は CamelGoogleMail.<option> の形式である必要があります。inBody オプションはメッセージヘッダーを上書きすることに注意してください。つまり、エンドポイントオプション inBody=option は CamelGoogleMail.option ヘッダーを上書きすることに注意してください。

エンドポイントおよびオプションの詳細は、API ドキュメント(https://developers.google.com/gmail/api/v1/reference/)を参照してください。

1.エンドポイント接頭辞 アタッチメント

以下のエンドポイントは、以下のように接頭辞 attachments で呼び出すことができます。

```
google-mail://attachments/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
get		id、messageId、userId	com.google.api.services.gmail.model.MessagePartBody

添付の URI オプション

名前	タイプ
id	文字列
messageId	文字列
userId	文字列

2.エンドポイント接頭辞ドラフト

以下のエンドポイントは、以下のように接頭辞 drafts で呼び出すことができます。

> ***google-mail://drafts/endpoint?[options]***

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
create		[mediaContent], content, userId	**com.google.api.services.gmail.model.Draft**
delete		id、userId	
get		id、userId	**com.google.api.services.gmail.model.Draft**
list		userId	**com.google.api.services.gmail.model.ListDraftsResponse**
send		[mediaContent], content, userId	**com.google.api.services.gmail.model.Message**
update		[mediaContent]、content、id、userId	**com.google.api.services.gmail.model.Draft**

ドラフトの *URI オプション*

名前	タイプ
content	com.google.api.services.gmail.model.Draft
id	文字列
mediaContent	com.google.api.client.http.AbstractInputStreamContent
userId	文字列

3.エンドポイント接頭辞の 履歴

以下のエンドポイントは、以下のように接頭辞 history で呼び出すことができます。

> *google-mail://history/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
list		userId	**com.google.api.services.gmail.model.ListHistoryResponse**

履歴の URI オプション

名前	タイプ
userId	文字列

4.エンドポイント接頭辞の ラベル

以下のエンドポイントは、以下のように接頭辞 labels で呼び出すことができます。

> *google-mail://labels/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
create		content, userId	**com.google.api.services.gmail.model.Label**
delete		id、userId	
get		id、userId	**com.google.api.services.gmail.model.Label**
list		userId	**com.google.api.services.gmail.model.ListLabelsResponse**
patch		content、id、userId	**com.google.api.services.gmail.model.Label**

update		content、id、userId	**com.google.api.services.gmail.model.Label**

ラベルの URI オプション

名前	タイプ
content	com.google.api.services.gmail.model.Label
id	文字列
userId	文字列

5.エンドポイント接頭辞 メッセージ

以下のエンドポイントは、以下のように接頭辞 messages で呼び出すことができます。

> *google-mail://messages/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		id、userId	
get		id、userId	**com.google.api.services.gmail.model.Message**
gmailImport		[mediaContent], content, userId	**com.google.api.services.gmail.model.Message**
insert		[mediaContent], content, userId	**com.google.api.services.gmail.model.Message**
list		userId	**com.google.api.services.gmail.model.ListMessagesResponse**
modify		id, modifyMessageRequest, userId	**com.google.api.services.gmail.model.Message**

send		[mediaContent], content, userId	**com.google.api.services.gmail.model.Message**
trash		id、userId	
untrash		id、userId	

メッセージの URI オプション

名前	タイプ
content	com.google.api.services.gmail.model.Message
id	文字列
mediaContent	com.google.api.client.http.AbstractInputStreamContent
modifyMessageRequest	com.google.api.services.gmail.model.ModifyMessageRequest
userId	文字列

6.エンドポイント接頭辞 スレッド

以下のエンドポイントは、以下のように接頭辞 threads で呼び出すことができます。

```
google-mail://threads/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
delete		id、userId	
get		id、userId	**com.google.api.services.gmail.model.Thread**
list		userId	**com.google.api.services.gmail.model.ListThreadsResponse**

modify		content、id、userId	**com.google.api.services.gmail.model.Thread**
trash		id、userId	
untrash		id、userId	

スレッドの URI オプション

名前	タイプ
content	com.google.api.services.gmail.model.ModifyThreadRequest
id	文字列
userId	文字列

7.エンドポイント接頭辞 ユーザー

以下のエンドポイントは、以下のように接頭辞 users で呼び出すことができます。

```
google-mail://users/endpoint?[options]
```

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
getProfile		userId	com.google.api.services.gmail.model.Profile

ユーザーの URI オプション

名前	タイプ
userId	文字列

コンシューマーエンドポイント

プロデューサーエンドポイントはいずれもコンシューマーエンドポイントとして使用できます。コンシューマーエンドポイントは、consumer. 接頭辞が付い た Scheduled Poll Consumer オプション を使用して、エンドポイントの呼び出しをスケジュールできます。配列またはコレクションを返すコン

シューマーエンドポイントは、要素ごとにエクスチェンジを 1 つ生成し、それらのルートはエクスチェンジごとに 1 回実行されます。

メッセージヘッダー

URI オプションは、CamelGoogleMail. 接頭辞が付いたプロデューサーエンドポイントのメッセージヘッダーで指定できます。

メッセージボディー

すべての結果メッセージ本文は、GoogleMailComponent が使用する基礎となる API によって提供されるオブジェクトを使用します。プロデューサーエンドポイントは、inBody エンドポイント URI パラメーターに受信メッセージボディーのオプション名を指定できます。配列またはコレクションを返すエンドポイントの場合、コンシューマーエンドポイントはすべての要素を個別のメッセージにマップします。

第62章 GUAVA EVENTBUS

GUAVA EVENTBUS コンポーネント

Camel 2.10.0 以降で利用可能

Google Guava EventBus は、コンポーネントを相互に明示的に登録しなくても、コンポーネント間のパブリッシュ/サブスクライブスタイルの通信を可能にします（そのため、相互に認識する必要があります）。guava-eventbus: コンポーネントは、Camel と Google Guava EventBus インフラストラクチャー間の統合ブリッジを提供します。後者のコンポーネントでは、Guava EventBus で交換されるメッセージは、Camel ルートに透過的に転送できます。EventBus コンポーネントを使用すると、Camel エクスチェンジのボディーを Guava EventBus にルーティングすることもできます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-guava-eventbus</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
guava-eventbus:busName[?options]
```

busName は、Camel レジストリーにある com.google.common.eventbus.EventBus インスタンスの名前を表します。

オプション

名前	デフォルト値	説明

eventClass	null	Camel 2.10: ルートのコンシューマー側で使用された場合、は **EventBus** から受信したイベントを、**eventClass** の クラスおよびスーパークラス にフィルターします。このオプションの null 値は、これを **java.lang.Object** に設定するのと同じです。つまり、コンシューマーはイベントバスに受信するすべてのメッセージを取得します。このオプションは **listenerInterface** オプションと併用できません。
listenerInterface	null	Camel 2.11: **@Subscribe** アノテーションが付けられたメソッドを持つインターフェイス。動的プロキシーはインターフェイスで作成され、**EventBus** リスナーとして登録できるようにします。特に、マルチイベントリスナーを作成し、**DeadEvent** を適切に処理する際に役立ちます。このオプションは **eventClass** オプションと併用できません。

使用方法

ルートのコンシューマー側で guava-eventbus コンポーネントを使用すると、Guava EventBus に送信されたメッセージをキャプチャーし、Camel ルートに転送します。Guava EventBus コンシューマーは受信メッセージを 非同期的 に処理します。

```
SimpleRegistry registry = new SimpleRegistry();
EventBus eventBus = new EventBus();
registry.put("busName", eventBus);
CamelContext camel = new DefaultCamelContext(registry);

from("guava-eventbus:busName").to("seda:queue");

eventBus.post("Send me to the SEDA queue.");
```

ルートのプロデューサー側で guava-eventbus コンポーネントを使用すると、Camel エクスチェンジのボディーが Guava EventBus インスタンスに転送されます。

```
SimpleRegistry registry = new SimpleRegistry();
EventBus eventBus = new EventBus();
registry.put("busName", eventBus);
CamelContext camel = new DefaultCamelContext(registry);
```

```
from("direct:start").to("guava-eventbus:busName");

ProducerTemplate producerTemplate = camel.createProducerTemplate();
producer.sendBody("direct:start", "Send me to the Guava EventBus.");

eventBus.register(new Object(){
  @Subscribe
  public void messageHander(String message) {
    System.out.println("Message received from the Camel: " + message);
  }
});
```

DEADEVENT に関する考慮事項

Guava EventBus の設計によって生じる制限により、@Subscribe メソッドでアノテーションが付けられたクラスを作成せずにリスナーによって受信されるイベントクラスを指定できないことに注意してください。この制限は、eventClass オプションが指定されたエンドポイントが可能なすべてのイベント(java.lang.Object)をリッスンし、ランタイム時に適切なメッセージをプログラムでフィルターすることを意味します。以下のスニッシングは、Camel コードベースからの適切な抜粋を示しています。

```
@Subscribe
public void eventReceived(Object event) {
  if (eventClass == null || eventClass.isAssignableFrom(event.getClass())) {
    doEventReceived(event);
...
```

このアプローチの欠点は、Camel が使用する EventBus インスタンスが com.google.common.eventbus.DeadEvent 通知を生成しないことです。Camel が正確に指定されたイベントのみをリッスンするようにするには（そのため、DeadEvent サポートを有効にする）、listenerInterface endpoint オプションを使用します。Camel は、後者のオプションで指定したインターフェイスで動的プロキシーを作成し、インターフェイスハンドラーメソッドによって指定されたメッセージのみをリッスンします。SpecificEvent インスタンスのみを処理する単一のメソッドインターフェイスの例を以下に示します。

```
package com.example;

public interface CustomListener {

  @Subscribe
  void eventReceived(SpecificEvent event);

}
```

上記のリスナーは、以下のようにエンドポイント定義で使用できます。

```
from("guava-eventbus:busName?listenerInterface=com.example.CustomListener").to("seda:queue");
```

複数のタイプのイベントの使用

Guava EventBus コンシューマーが消費する複数のタイプのイベントを定義するには、リスナーインターフェイスが @Subscribe アノテーションが付けられた複数のメソッドを提供できるため、listenerInterface エンドポイントオプションを使用します。

```
package com.example;

public interface MultipleEventsListener {

  @Subscribe
  void someEventReceived(SomeEvent event);

  @Subscribe
  void anotherEventReceived(AnotherEvent event);

}
```

上記のリスナーは、以下のようにエンドポイント定義で使用できます。

```
from("guava-eventbus:busName?
listenerInterface=com.example.MultipleEventsListener").to("seda:queue");
```

第63章 HAWTDB

HAWTDB

Apache Camel 2.3 で利用可能

HawtDB は、非常に軽量で組み込み可能なキー値データベースです。これにより、Apache Camel とともに、Aggregator などのさまざまな Apache Camel 機能の永続的なサポートが提供されます。

非推奨

HawtDB プロジェクトは非推奨となり、軽量で組み込み可能なキー値データベースとして leveldb に置き換えられました。leveldb の使用を容易にするには、leveldbjni プロジェクトがあります。Apache ActiveMQ プロジェクトは、今後、leveldb を主要なファイルベースのメッセージストアとして使用して kahadb を置き換える予定です。

camel-leveldb コンポーネントとして、これの代わりに を使用することが推奨されます。

現在の機能では、以下が提供されます。

- HawtDBAggregationRepository

HAWTDBAGGREGATIONREPOSITORY の使用

HawtDBAggregationRepository は AggregationRepository で、オンザフライで集約されたメッセージを永続化します。これにより、デフォルトのアグリゲーターはメモリーの AggregationRepository のみを使用するので、メッセージを失わないようにします。

これには、以下のオプションがあります。

オプション	タイプ	説明
repositoryName	文字列	必須リポジトリー名。複数のリポジトリーに共有 **HawtDBFile** を使用できます。
persistentFileName	文字列	永続ストレージのファイル名。起動時にファイルが存在しない場合は、新しいファイルが作成されます。
bufferSize	int	ファイルストアにマップされるメモリーセグメントバッファーのサイズ。デフォルトでは 8mb です。値はバイト単位です。
sync	boolean	**HawtDBFile** が書き込み時に同期されるかどうか。デフォルトは **true** です。書き込み時に同期することで、すべての書き込みがディスクにスプールされるのを待つため、更新は失われません。このオプションを無効にすると、多くの書き込みがバッチ処理されると HawtDB が自動的に同期されます。
pageSize	short	メモリーページのサイズ。デフォルトでは 512 バイトになります。値はバイト単位です。
hawtDBFile	HawtDBFile	既存の設定された **org.apache.camel.component.hawtdb.HawtDBFile** インスタンスを使用します。
returnOldExchange	boolean	get 操作が存在する場合は、get 操作によって古い既存のエクスチェンジが返されるかどうか。デフォルトでは、集計時に古いエクスチェンジを必要としないため、このオプションは **false** で最適化されます。
useRecovery	boolean	リカバリーが有効になっているかどうか。このオプションは、デフォルトで **true** です。有効にすると、Apache Camel Aggregator は失敗した集約されたエクスチェンジを自動的にリカバリーし、再送信を行います。

recoveryInterval	long	recovery が有効になっている場合、バックグラウンドタスクは x 度ごとに実行され、失敗したエクスチェンジをスキャンしてリカバリーし、再送信します。デフォルトでは、この間隔は 5000 ミリ秒です。
maximumRedeliveries	int	リカバリーされたエクスチェンジの再配信試行の最大数を制限できます。有効にすると、すべての再配信試行に失敗すると、エクスチェンジはデッドレターチャネルに移動します。デフォルトでは、このオプションは無効です。このオプションを使用する場合は、**deadLetterUri** オプションも指定する必要があります。
deadLetterUri	文字列	リカバリーされたエクスチェンジが使い切られる Dead Letter Channel のエンドポイント URI。このオプションを使用する場合は、**maximumRedeliveries** オプションも指定する必要があります。
optimisticLocking	**false**	Camel 2.12: 複数の Camel アプリケーションが同じ HawtDB ベースの集約リポジトリーを共有するクラスター環境で必要になる、楽観的ロックを有効にします。

***repositoryName** オプションを指定する必要があります。次に、**persistentFileName** または **hawtDBFile** のいずれかを指定する必要があります。*

永続化時に保持される内容

***HawtDBAggregationRepository** は、**Serializable** と互換性のあるデータタイプのみを保持します。データ型がそのようなタイプの場合はドロップされ、**WARN** がログに記録されます。また、メッセージ本文とメッセージ ヘッダーのみを保持 し ます。**Exchange** プロパティーは永続化され ません。*

復元

***HawtDBAggregationRepository** は、デフォルトで失敗したエクスチェンジ を復元し ます。これは、永続ストアで失敗した エクスチェンジをスキャンするバックグラウンドタスクを持つことで行われます。**checkInterval** オプションを使用して、このタスクの実行頻度を設定できます。リカバリーはト*

ランザクションとして機能し、Apache Camel が失敗したエクスチェンジのリカバリーと再配信を試みます。リカバリーされたエクスチェンジは永続ストアから復元され、再送信されて再度送信されます。

エクスチェンジ のリカバリー/再配信時に、以下のヘッダーが設定されます。

ヘッダー	タイプ	説明
Exchange.REDELIVERED	ブール値	エクスチェンジ が再配信されていることを示すために が true に設定されます。
Exchange.REDELIVERY_COUNTER	整数	1 から始まる再配信の試行。

エクスチェンジ が正常に処理された場合のみ、完了とマークされます。これは、AggregationRepository で confirm メソッドが呼び出されたときに発生します。つまり、同じ エクスチェンジ が再び失敗すると、成功するまで再試行されます。

maximumRedeliveries オプションを使用して、特定のリカバリーエクスチェンジの再配信試行の最大数を制限でき ます。また、adLetter Uri オプションも設定する 必要があります。これにより、Apache Camel は maximumRedeliveries に達したときに エクスチェンジ を送信する場所を認識します。

camel-hawtdb のユニットテストには、このテスト などのいくつかの例を確認できます。

JAVA DSL での HAWTDBAGGREGATIONREPOSITORY の使用

この例では、target/data/hawtdb.dat ファイルで集約されたメッセージを永続化します。

```
public void configure() throws Exception {
    // create the hawtdb repo
    HawtDBAggregationRepository repo = new HawtDBAggregationRepository("repo1",
"target/data/hawtdb.dat");

    // here is the Camel route where we aggregate
    from("direct:start")
        .aggregate(header("id"), new MyAggregationStrategy())
            // use our created hawtdb repo as aggregation repository
            .completionSize(5).aggregationRepository(repo)
            .to("mock:aggregated");
}
```

SPRING XML での HAWTDBAGGREGATIONREPOSITORY の使用

同じ例になりますが、代わりに Spring XML を使用します。

```
<!-- a persistent aggregation repository using camel-hawtdb -->
<bean id="repo"
class="org.apache.camel.component.hawtdb.HawtDBAggregationRepository">
   <!-- store the repo in the hawtdb.dat file -->
   <property name="persistentFileName" value="target/data/hawtdb.dat"/>
   <!-- and use repo2 as the repository name -->
   <property name="repositoryName" value="repo2"/>
</bean>

<!-- aggregate the messages using this strategy -->
<bean id="myAggregatorStrategy"
class="org.apache.camel.component.hawtdb.HawtDBSpringAggregateTest$MyAggregationSt
rategy"/>

<!-- this is the camel routes -->
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">

   <route>
      <from uri="direct:start"/>
      <!-- aggregate using our strategy and hawtdb repo, and complete when we have 5
messages aggregated -->
      <aggregate strategyRef="myAggregatorStrategy" aggregationRepositoryRef="repo"
completionSize="5">
         <!-- correlate by header with the key id -->
         <correlationExpression><header>id</header></correlationExpression>
         <!-- send aggregated messages to the mock endpoint -->
         <to uri="mock:aggregated"/>
      </aggregate>
   </route>

</camelContext>
```

DEPENDENCIES

Apache Camel ルートで HawtDB を使用するには、camel-hawtdb の依存関係を追加する必要があります。

maven を使用する場合は、以下を pom.xml に追加し、バージョン番号を最新および最大のリリースに置き換えます(最新バージョンのダウンロードページを参照してください)。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hawtdb</artifactId>
```

```
  <version>2.3.0</version>
</dependency>
```

以下も参照してください。

- *Apache Camel Development Guideのセクション Aggregator*
- *SQL*
- *コンポーネント*

第64章 HAZELCAST コンポーネント

HAZELCAST コンポーネント

Apache Camel 2.7 で利用可能

hazelcast: コンポーネントを使用すると、Hazelcast 分散データグリッド/キャッシュを操作することができます。Hazelcast は、Java（単一の jar）で完全に書き込まれたメモリーデータグリッドの です。マップ、マルチマップ（同じキー、n 値）、queue、list、atomic 番号など、さまざまなデータストアの優れたパレットを提供します。Hazelcast を使用する主な理由は、単純なクラスターサポートです。ネットワーク上でマルチキャストを有効にしている場合は、追加設定なしで、数百のノードを持つクラスターを実行できます。Hazelcast は、ノード間の n コピー（デフォルトは 1）、キャッシュ永続性、ネットワーク設定（必要な場合）、ニアキャッシュ、eviction などの機能を追加するように簡単に設定できます。詳細は、Hazelcast のドキュメント(http://www.hazelcast.com/docs.jsp)を参照してください。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-hazelcast</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
hazelcast:[ map | multimap | queue | topic | seda | set | atomicvalue | instance |
list]:cachename[?options]
```

オプション

名前	必須	説明
hazelcastInstance	いいえ	**Camel 2.14:** hazelcast エンドポイントに使用できる hazelcast インスタンス参照。インスタンス参照を指定しない場合、Camel は camel-hazelcast インスタンスからデフォルトの hazelcast インスタンスを使用します。
hazelcastInstanceName	No	

defaultOperation	-1	Camel 2.15: 操作ヘッダーが提供されていない場合に使用するデフォルトの操作を指定します。

警告

2 番目の接頭辞を使用して、使用するデータストアのタイプを定義する必要があります。

セクション

1. *マップの使用*
2. *マルチマップの使用*
3. *キューの使用*
4. *トピックの使用*
5. *リストの使用*
6. *sedaの使用*
7. *アトミック番号の使用*
8. *クラスター サポート（インスタンス）の使用*

マップの使用

MAP CACHE PRODUCER - TO("HAZELCAST:MAP:FOO")

マップに値を保存する場合は、マップキャッシュプロデューサーを使用できます。マップキャッシュプロデューサーは、5 つの操作(put、get、update、delete、query)を提供します。最初の 4 では、hazelcast.operation.type ヘッダー変数内に操作を提供する必要があります。Java DSL では、org.apache.camel.component.hazelcast.HazelcastConstants からの定数を使用できます。

リクエストメッセージのヘッダー変数：

名前	タイプ	説明
hazelcast.operation.type	文字列	有効な値は put、delete、get、update、query です。
hazelcast.objectId	文字列	キャッシュ内にオブジェクトを保存/検索するオブジェクト ID（クエリー操作には必要ありません）

警告

Apache Camel 2.8 でヘッダー変数が変更されました。

名前	タイプ	説明
CamelHazelcastOperationType	文字列	有効な値は put、delete、get、update、query Version 2.8です。
CamelHazelcastObjectId	文字列	キャッシュ内にオブジェクトを保存/検索するオブジェクト ID（クエリー操作には必要ありません）バージョン 2.8

以下を使用してサンプルを呼び出すことができます。

```
template.sendBodyAndHeader("direct:[put|get|update|delete|query]", "my-foo",
HazelcastConstants.OBJECT_ID, "4711");
```

PUT の例：

Java DSL の場合

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.PUT_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:put" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>put</constant>
 </setHeader>
 <to uri="hazelcast:map:foo" />
</route>
```

GET の例：

Java DSL の場合

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
 <from uri="direct:get" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>get</constant>
 </setHeader>
```

```
 <to uri="hazelcast:map:foo" />
 <to uri="seda:out" />
</route>
```

更新のサンプル：

Java DSL の場合

```
from("direct:update")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.UPDATE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:update" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>update</constant>
 </setHeader>
 <to uri="hazelcast:map:foo" />
</route>
```

削除のサンプル：

Java DSL の場合

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.DELETE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:delete" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>delete</constant>
 </setHeader>
 <to uri="hazelcast:map:foo" />
</route>
```

-

クエリーのサンプル

Java DSL の場合

```
from("direct:query")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.QUERY_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
 <from uri="direct:query" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>query</constant>
 </setHeader>
 <to uri="hazelcast:map:foo" />
 <to uri="seda:out" />
</route>
```

クエリー操作 Hazelcast は、分散マップをクエリーする構文のような SQL を提供します。

```
String q1 = "bar > 1000";
template.sendBodyAndHeader("direct:query", null, HazelcastConstants.QUERY, q1);
```

MAP CACHE CONSUMER - FROM("HAZELCAST:MAP:FOO")

Hazelcast は、データグリッドでイベントリスナーを提供します。キャッシュを操作する場合に通知する場合は、map コンシューマーを使用できます。、update、delete、evict の 4 つのイベントがあります。イベントタイプはhazelcast.listener.actionヘッダー変数に保存されます。map コンシューマーは、これらの変数内に追加情報を提供します。

応答メッセージ内のヘッダー変数：

名前	タイプ	説明
hazelcast.listener.time	Long	イベントの時間（ミリ秒単位）

hazelcast.listener.type	文字列	マップコンシューマーは"cachelistener"をここで設定します。
hazelcast.listener.action	文字列	イベントのタイプ：ここでは、**更新**、**エンコーデント**、**および削除され** ました。
hazelcast.objectId	文字列	オブジェクトの oid
hazelcast.cache.name	文字列	キャッシュの名前（例：foo）
hazelcast.cache.type	文字列	キャッシュのタイプ - マップ

警告

Apache Camel 2.8 でヘッダー変数が変更されました。

名前	タイプ	説明
CamelHazelcastListenerTime	**Long**	イベントの時間（ミリス **バージョン** 2.8 でのイベント時間）
CamelHazelcastListenerType	文字列	map consumer sets here "cachelistener" **Version** 2.8
CamelHazelcastListenerAction	文字列	イベントのタイプ：ここで、**更新**、**エンコーデーション**、**および削除** さ れました。 **バージョン** 2.8
CamelHazelcastObjectId	文字列	The oid of the object **Version** 2.8
CamelHazelcastCacheName	文字列	キャッシュの名前（例：foo **バージョン** 2.8）
CamelHazelcastCacheType	文字列	キャッシュのタイプ：ここでは、マップ **バージョン** 2.8

オブジェクト値は、メッセージボディー内に put および update アクションに保存されます。

以下に例を示します。

```
fromF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.log("object...")
.choice()

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDE
D))
        .log("...added")
        .to("mock:added")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVIC
TED))
        .log("...envicted")
        .to("mock:envicted")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.UPDAT
ED))
        .log("...updated")
        .to("mock:updated")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMO
VED))
        .log("...removed")
        .to("mock:removed")
    .otherwise()
        .log("fail!");
```

マルチマップの使用

MULTIMAP CACHE PRODUCER - TO("HAZELCAST:MULTIMAP:FOO")

マルチマップは、n の値を 1 つのキーに格納できるキャッシュです。マルチマッププロデューサーは 4 つの操作(put、get、removevalue、delete)を提供します。

リクエストメッセージのヘッダー変数：

名前	タイプ	説明

hazelcast.operation.type	文字列	有効な値は put、get、removevalue、delete です。
hazelcast.objectId	文字列	キャッシュ内にオブジェクトを保存/検索するオブジェクト ID

警告

Apache Camel 2.8 でヘッダー変数が変更されました。

Apache Camel 2.8 でのリクエストメッセージのヘッダー変数：

名前	タイプ	説明
CamelHazelcastOperationType	文字列	有効な値は put、delete、get、update、query **Available as Apache Camel 2.8**です。
CamelHazelcastObjectId	文字列	キャッシュ内にオブジェクトを保存/検索するオブジェクト ID（クエリー操作には必要ありません）**バージョン 2.8**

以下を使用してサンプルを呼び出すことができます。

```
template.sendBodyAndHeader("direct:[put|get|update|delete|query]", "my-foo",
HazelcastConstants.OBJECT_ID, "4711");
```

PUT の例：

Java DSL の場合

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.PUT_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:put" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>put</constant>
 </setHeader>
 <to uri="hazelcast:map:foo" />
</route>
```

GET の例：

Java DSL の場合

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
 <from uri="direct:get" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>get</constant>
 </setHeader>
 <to uri="hazelcast:map:foo" />
 <to uri="seda:out" />
</route>
```

更新のサンプル：

Java DSL の場合

```
from("direct:update")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.UPDATE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:update" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>update</constant>
 </setHeader>
 <to uri="hazelcast:map:foo" />
</route>
```

削除のサンプル：

Java DSL の場合

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.DELETE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:delete" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>delete</constant>
 </setHeader>
 <to uri="hazelcast:map:foo" />
</route>
```

クエリーのサンプル

Java DSL の場合

```
from("direct:query")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.QUERY_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
 <from uri="direct:query" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>query</constant>
 </setHeader>
 <to uri="hazelcast:map:foo" />
 <to uri="seda:out" />
</route>
```

クエリー操作 Hazelcast は、分散マップをクエリーする構文のような SQL を提供します。

```
String q1 = "bar > 1000";
template.sendBodyAndHeader("direct:query", null, HazelcastConstants.QUERY, q1);
```

MAP CACHE CONSUMER - FROM("HAZELCAST:MAP:FOO")

Hazelcast は、データグリッドでイベントリスナーを提供します。キャッシュを操作する場合に通知する場合は、map コンシューマーを使用できます。、update、delete、evict の 4 つのイベントがあります。イベントタイプはhazelcast.listener.actionヘッダー変数に保存されます。map コンシューマーは、これらの変数内に追加情報を提供します。

応答メッセージ内のヘッダー変数：

名前	タイプ	説明
hazelcast.listener.time	Long	イベントの時間（ミリ秒単位）
hazelcast.listener.type	文字列	マップコンシューマーは"cachelistener"をここで設定します。
hazelcast.listener.action	文字列	イベントのタイプ：ここでは、**更新**、**エンコーデント**、**および 削除され** ました。
hazelcast.objectId	文字列	オブジェクトの oid
hazelcast.cache.name	文字列	キャッシュの名前（例：foo）

hazelcast.cache.type	文字列	キャッシュのタイプ - マップ

警告

Apache Camel 2.8 でヘッダー変数が変更されました。

名前	タイプ	説明
CamelHazelcastListenerTime	**Long**	イベントの時間（ミリス バージョン 2.8 でのイベント時間）
CamelHazelcastListenerType	文字列	map consumer sets here "cachelistener" Version 2.8
CamelHazelcastListenerAction	文字列	イベントのタイプ：ここで、更新、エンコーデーション、および削除 さ れました。 バージョン 2.8
CamelHazelcastObjectId	文字列	The oid of the object Version 2.8
CamelHazelcastCacheName	文字列	キャッシュの名前（例：foo バージョン 2.8）
CamelHazelcastCacheType	文字列	キャッシュのタイプ：ここでは、マップ バージョン 2.8

オブジェクト値は、メッセージボディー内に put および update アクションに保存されます。

以下に例を示します。

```
fromF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.log("object...")
.choice()

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDE
D))
        .log("...added")
        .to("mock:added")
```

```
.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVICTED))
        .log("...envicted")
        .to("mock:envicted")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.UPDATED))
        .log("...updated")
        .to("mock:updated")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
        .log("...removed")
        .to("mock:removed")
    .otherwise()
        .log("fail!");
```

マルチマップの使用

MULTIMAP CACHE PRODUCER - TO("HAZELCAST:MULTIMAP:FOO")

マルチマップは、n の値を 1 つのキーに格納できるキャッシュです。マルチマッププロデューサーは 4 つの操作(put、get、removevalue、delete)を提供します。

リクエストメッセージのヘッダー変数：

名前	タイプ	説明
hazelcast.operation.type	文字列	有効な値は put、get、removevalue、delete です。
hazelcast.objectId	文字列	キャッシュ内にオブジェクトを保存/検索するオブジェクト ID

警告

Apache Camel 2.8 でヘッダー変数が変更されました。

名前	タイプ	説明
CamelHazelcastOperationType	文字列	有効な値は put、get、removevalue、および delete Version 2.8です。
CamelHazelcastObjectId	文字列	キャッシュ バージョン 2.8内でオブジェクトを保存/検索するオブジェクト ID

PUT の例：

Java DSL の場合

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.PUT_OPERATION))
.to(String.format("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX));
```

Spring DSL:

```
<route>
 <from uri="direct:put" />
 <log message="put.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>put</constant>
 </setHeader>
 <to uri="hazelcast:multimap:foo" />
</route>
```

REMOVEVALUE の例：

Java DSL の場合

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.REMOVEVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:removevalue" />
 <log message="removevalue..."/>
      <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>removevalue</constant>
 </setHeader>
 <to uri="hazelcast:multimap:foo" />
</route>
```

値を削除するには、メッセージのボディー内で削除する値を指定する必要があります。マルチマップオブジェクト } がある場合は、メッセージボディー内に my-foo の値を削除する必要があります。

GET の例：

Java DSL の場合

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
 <from uri="direct:get" />
 <log message="get.."/>
      <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>get</constant>
 </setHeader>
 <to uri="hazelcast:multimap:foo" />
 <to uri="seda:out" />
</route>
```

削除のサンプル：

Java DSL の場合

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.DELETE_OPERATION))
```

```
.toF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:delete" />
 <log message="delete.."/>
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>delete</constant>
 </setHeader>
 <to uri="hazelcast:multimap:foo" />
</route>
```

以下を使用して、テストクラスで呼び出しできます。

```
template.sendBodyAndHeader("direct:[put|get|removevalue|delete]", "my-foo",
HazelcastConstants.OBJECT_ID, "4711");
```

MULTIMAP CACHE CONSUMER - FROM("HAZELCAST:MULTIMAP:FOO")

マルチマップキャッシュの場合、このコンポーネントはマップキャッシュコンシューマーと同じリスナー/変数を提供します(update および enviction リスナーを除く)。唯一の違いは、URI 内の multimap 接頭辞です。以下は例です。

```
fromF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.log("object...")
.choice()
 .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDE
D))
  .log("...added")
               .to("mock:added")

//.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVI
CTED))
       //       .log("...envicted")
       //       .to("mock:envicted")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMO
VED))
               .log("...removed")
               .to("mock:removed")
       .otherwise()
               .log("fail!");
```

応答メッセージ内のヘッダー変数：

名前	タイプ	説明
hazelcast.listener.time	**Long**	イベントの時間（ミリ秒単位）
hazelcast.listener.type	**文字列**	マップコンシューマーは"cachelistener"をここで設定します。
hazelcast.listener.action	**文字列**	イベントのタイプ - ここでは **追加** および **削除** （および **近い動作**）
hazelcast.objectId	**文字列**	オブジェクトの oid
hazelcast.cache.name	**文字列**	キャッシュの名前（例：foo）
hazelcast.cache.type	**文字列**	キャッシュのタイプ（ここではマルチマップ）

エビクションは機能として追加されますが、すぐに追加されます（これは Hazelcast の問題です）。

警告

Apache Camel 2.8 でヘッダー変数が変更されました。

名前	タイプ	説明
CamelHazelcastListenerTime	**Long**	イベントの時間（ミリス バージョン 2.8 でのイベント時間）
CamelHazelcastListenerType	**文字列**	map consumer sets here "cachelistener" Version 2.8
CamelHazelcastListenerAction	**文字列**	イベントのタイプ：ここでは **追加** および **削除** （および近い **動作**） バージョン 2.8
CamelHazelcastObjectId	**文字列**	The oid of the object Version 2.8

CamelHazelcastCacheName	**文字列**	キャッシュの名前（例：foo **バージョン** 2.8）
CamelHazelcastCacheType	**文字列**	キャッシュのタイプ（ここではマルチマップ **バージョン 2.8）**

キューの使用

キュープロデューサー TO ("HAZELCAST:QUEUE:FOO")

キュープロデューサーは 6 つの操作(add、put、poll、peek、provided、removevalue)を提供します。

追加する例：

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.ADD_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

PUT の例：

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.PUT_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

ポーリングのサンプル：

```
from("direct:poll")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.POLL_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

PEEK の例：

```
from("direct:peek")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.PEEK_OPERATION))
```

```
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

オファーのサンプル：

```
from("direct:offer")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.OFFER_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

REMOVEVALUE の例：

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.REMOVEVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

キューコンシューマー FROM ("HAZELCAST:QUEUE:FOO")

キューコンシューマーは、2 つの操作(add、remove)を提供します。

```
fromF("hazelcast:%smm", HazelcastConstants.QUEUE_PREFIX)
   .log("object...")
   .choice()
 .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDE
D))
        .log("...added")
  .to("mock:added")
 .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMO
VED))
  .log("...removed")
  .to("mock:removed")
 .otherwise()
  .log("fail!");
```

トピックの使用

トピックプロデューサー - TO ("HAZELCAST:TOPIC:FOO")

トピックプロデューサーは 1 つの操作(publish)のみを提供します。

公開のサンプル

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.PUBLISH_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.PUBLISH_OPERATION);
```

TOPIC CONSUMER - FROM ("HAZELCAST:TOPIC:FOO")

トピックコンシューマーは 1 つの操作（受信）のみを提供します。このコンポーネントは、トピックの場合のように、複数の消費をサポートすることが予想されます。そのため、同じハザーキャストのトピックに必要な数のコンシューマーを自由に使用できます。

```
fromF("hazelcast:%sfoo", HazelcastConstants.TOPIC_PREFIX)
  .choice()

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.RECEI
VED))
     .log("...message received")
    .otherwise()
     .log("...this should never have happened")
```

LIST の使用

プロデューサー TO ("HAZELCAST:LIST:FOO")を一覧表示します。

リストプロデューサーは、4 つの操作(add、set、get、removevalue)を提供します。

追加する例：

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.ADD_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

GET の例：

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX)
.to("seda:out");
```

SETVALUE の例：

```
from("direct:set")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.SETVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

REMOVEVALUE の例：

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.REMOVEVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

警告

hazelcast でまだサポートされておらず、set、get、および removevalue は今後追加されることに注意してください。

LIST CONSUMER FROM ("HAZELCAST:LIST:FOO")

リストコンシューマーは、2 つの操作(add、remove)を提供します。

```
fromF("hazelcast:%smm", HazelcastConstants.LIST_PREFIX)
 .log("object...")
 .choice()

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDE
D))
   .log("...added")
                    .to("mock:added")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMO
VED))
```

```
    .log("...removed")
                .to("mock:removed")
            .otherwise()
                .log("fail!");
```

SEDA の使用

SEDA コンポーネントは、提供される REST コンポーネントとは異なります。これは、コア "SEDA" コンポーネントと同様に、非同期 SEDA アーキテクチャーをサポートするためにワークキューを実装します。

SEDA PRODUCER TO("HAZELCAST:SEDA:FOO")

SEDA プロデューサーは操作を提供しません。指定のキューにデータのみを送信します。

名前	デフォルト値	説明
transferExchange	false	Apache Camel 2.8.0: true に設定すると、エクスチェンジ全体が転送されます。ヘッダーまたはボディーにシリアライズ可能なオブジェクトが含まれる場合、それらはスキップされます。

Java DSL:

```
from("direct:foo")
.to("hazelcast:seda:foo");
```

Spring DSL の場合：

```
<route>
  <from uri="direct:start" />
  <to uri="hazelcast:seda:foo" />
</route>
```

SEDA CONSUMER FROM("HAZELCAST:SEDA:FOO")

SEDA コンシューマーは操作を提供しません。指定のキューからデータのみを取得します。

名前	デフォルト値	説明
pollInterval	**1000**	Camel 2.15 以降非推奨になりました。代わりに **pollTimeout** を使用します。
pollTimeout	**1000**	SEDA キューから消費する際に使用するタイムアウト。タイムアウトが発生すると、コンシューマーは実行を継続できるかどうかを確認できます。値を低く設定すると、シャットダウン時にコンシューマーがより迅速に対応できるようになります。
concurrentConsumers	**1**	SEDA キューからの同時コンシューマーのポーリングを使用します。
transferExchange	**false**	**Camel 2.8.0:** true に設定すると、エクスチェンジ全体が転送されます。ヘッダーまたはボディーにシリアライズ可能なオブジェクトが含まれる場合、それらはスキップされます。
transacted	**false**	**Camel 2.10.4:** true に設定すると、コンシューマーはトランザクションモードで実行されます。

Java DSL:

```
from("hazelcast:seda:foo")
.to("mock:result");
```

Spring DSL:

```
<route>
  <from uri="hazelcast:seda:foo" />
  <to uri="mock:result" />
</route>
```

ATOMIC NUMBER の使用

警告

このエンドポイントにはコンシューマーがありません\!

ATOMIC NUMBER PRODUCER - TO ("HAZELCAST:ATOMICNUMBER:FOO")

アトミック番号は、単にグリッドワイド番号(long)を提供するオブジェクトです。このプロデューサーの操作は setvalue （指定の値で数値を設定）、get、増加(+1)、縮小(-1)、および destroy です。

リクエストメッセージのヘッダー変数：

名前	タイプ	説明
hazelcast.operation.type	文字列	有効な値は、setvalue、get、increase、decrease、destroy です。

警告

Apache Camel 2.8 でヘッダー変数が変更されました。

名前	タイプ	説明
CamelHazelcastOperationType	文字列	有効な値は、setvalue、get、grade、destroy Available (Apache Camel バージョン 2.8 の場合)です。

セットのサンプル：

Java DSL の場合

–

```
from("direct:set")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.SETVALUE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:set" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>setvalue</constant>
 </setHeader>
 <to uri="hazelcast:atomicvalue:foo" />
</route>
```

メッセージボディー内で設定する値を指定します（ここでは 10）。template.sendBody("direct:set", 10);

GET の例：

Java DSL の場合

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:get" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>get</constant>
 </setHeader>
 <to uri="hazelcast:atomicvalue:foo" />
</route>
```

long body = template.requestBody("direct:get", null, Long.class); で数値を取得できます。

INCREMENT のサンプル：

Java DSL の場合

```
from("direct:increment")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.INCREMENT_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:increment" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>increment</constant>
 </setHeader>
 <to uri="hazelcast:atomicvalue:foo" />
</route>
```

実際の値（インクリメント後）はメッセージのボディー内に提供されます。

デクリメントのサンプル：

Java DSL の場合

```
from("direct:decrement")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.DECREMENT_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:decrement" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>decrement</constant>
 </setHeader>
 <to uri="hazelcast:atomicvalue:foo" />
</route>
```

実際の値（デクリメント後）はメッセージのボディー内に提供されます。

破棄のサンプル

警告

Hazelcast にはバグがあります。そのため、この機能は適切に機能しない可能性があります。1.9.3 で修正されます。

Java DSL の場合

```
from("direct:destroy")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.DESTROY_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
 <from uri="direct:destroy" />
       <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" --
>
 <setHeader headerName="hazelcast.operation.type">
  <constant>destroy</constant>
 </setHeader>
 <to uri="hazelcast:atomicvalue:foo" />
</route>
```

クラスターのサポート

警告

このエンドポイントはプロデューサーを提供しません。

INSTANCE CONSUMER - FROM("HAZELCAST:INSTANCE:FOO")

Hazelcast は 1 つのサーバーノードで理にかなっていますが、クラスター環境では非常に強力です。インスタンスコンシューマーは、新しいキャッシュインスタンスがクラスターに参加したり、クラスターから離脱したりする場合に実行されます。

以下に例を示します。

```
fromF("hazelcast:%sfoo", HazelcastConstants.INSTANCE_PREFIX)
.log("instance...")
.choice()
 .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
  .log("...added")
  .to("mock:added")
 .otherwise()
  .log("...removed")
  .to("mock:removed");
```

各イベントは、メッセージヘッダー内で以下の情報を提供します。

応答メッセージ内のヘッダー変数:

名前	タイプ	説明
hazelcast.listener.time	**Long**	イベントの時間（ミリ秒単位）
hazelcast.listener.type	**文字列**	map consumer sets here "instancelistener"
hazelcast.listener.action	**文字列**	イベントのタイプ：ここで **追加** または **削除され**ています。
hazelcast.instance.host	**文字列**	インスタンスのホスト名

hazelcast.instance.port	**整数**	インスタンスのポート番号

警告

Apache Camel 2.8 でヘッダー変数が変更されました。

名前	タイプ	説明
CamelHazelcastListenerTime	**Long**	イベントの時間（ミリス バージョン 2.8 でのイベント時間）
CamelHazelcastListenerType	**文字列**	map consumer sets here "instancelistener" Version 2.8
CamelHazelcastListenerActionn	**文字列**	イベントのタイプ：ここでは を **追加** または **削除** します。 バージョン 2.8
CamelHazelcastInstanceHost	**文字列**	インスタンス バージョン 2.8のホスト名
CamelHazelcastInstancePort	**整数**	インスタンスの バージョン 2.8 のポート番号

HAZELCAST リファレンスの使用

名前

```
<bean id="hazelcastLifecycle" class="com.hazelcast.core.LifecycleService"
    factory-bean="hazelcastInstance" factory-method="getLifecycleService"
    destroy-method="shutdown" />

<bean id="config" class="com.hazelcast.config.Config">
  <constructor-arg type="java.lang.String" value="HZ.INSTANCE" />
```

```
</bean>

<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast" factory-
method="newHazelcastInstance">
   <constructor-arg type="com.hazelcast.config.Config" ref="config"/>
</bean>
<camelContext xmlns="http://camel.apache.org/schema/spring">
   <route id="testHazelcastInstanceBeanRefPut">
      <from uri="direct:testHazelcastInstanceBeanRefPut"/>
      <setHeader headerName="CamelHazelcastOperationType">
         <constant>put</constant>
      </setHeader>
      <to uri="hazelcast:map:testmap?hazelcastInstanceName=HZ.INSTANCE"/>
   </route>

   <route id="testHazelcastInstanceBeanRefGet">
      <from uri="direct:testHazelcastInstanceBeanRefGet" />
      <setHeader headerName="CamelHazelcastOperationType">
         <constant>get</constant>
      </setHeader>
      <to uri="hazelcast:map:testmap?hazelcastInstanceName=HZ.INSTANCE"/>
      <to uri="seda:out" />
   </route>
</camelContext>
```

インスタンス別

```
<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast"
      factory-method="newHazelcastInstance" />
<bean id="hazelcastLifecycle" class="com.hazelcast.core.LifecycleService"
      factory-bean="hazelcastInstance" factory-method="getLifecycleService"
      destroy-method="shutdown" />

<camelContext xmlns="http://camel.apache.org/schema/spring">
   <route id="testHazelcastInstanceBeanRefPut">
      <from uri="direct:testHazelcastInstanceBeanRefPut"/>
      <setHeader headerName="CamelHazelcastOperationType">
         <constant>put</constant>
      </setHeader>
      <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
   </route>

   <route id="testHazelcastInstanceBeanRefGet">
      <from uri="direct:testHazelcastInstanceBeanRefGet" />
      <setHeader headerName="CamelHazelcastOperationType">
         <constant>get</constant>
      </setHeader>
      <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
      <to uri="seda:out" />
   </route>
</camelContext>
```

HAZELCAST インスタンスを OSGI サービスとして公開

OSGI コンテナーで動作し、同じコンテナーのすべてのバンドルで hazelcast のインスタンスを 1 つ使用する場合。キャッシュの必要なキャッシュは、hazelcast エンドポイントでサービスを参照することで、インスタンスを OSGI サービスおよびバンドルとして公開できます。

バンドル A はインスタンスを作成し、OSGI サービスとして公開します。

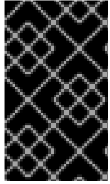
```
<bean id="config" class="com.hazelcast.config.FileSystemXmlConfig">
   <argument type="java.lang.String" value="${hazelcast.config}"/>
</bean>

<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast" factory-
method="newHazelcastInstance">
   <argument type="com.hazelcast.config.Config" ref="config"/>
</bean>

<!-- publishing the hazelcastInstance as a service -->
<service ref="hazelcastInstance" interface="com.hazelcast.core.HazelcastInstance" />
```

バンドル B は インスタンスを使用します。

```
<!-- referencing the hazelcastInstance as a service -->
<reference ref="hazelcastInstance" interface="com.hazelcast.core.HazelcastInstance" />

<camelContext xmlns="http://camel.apache.org/schema/blueprint">
   <route id="testHazelcastInstanceBeanRefPut">
      <from uri="direct:testHazelcastInstanceBeanRefPut"/>
      <setHeader headerName="CamelHazelcastOperationType">
         <constant>put</constant>
      </setHeader>
      <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
   </route>

   <route id="testHazelcastInstanceBeanRefGet">
      <from uri="direct:testHazelcastInstanceBeanRefGet" />
      <setHeader headerName="CamelHazelcastOperationType">
         <constant>get</constant>
      </setHeader>
      <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
      <to uri="seda:out" />
   </route>
</camelContext>
```

第65章 HBASE

HBASE コンポーネント



重要

Camel HBase は Apache Karaf ではサポートされません。

Camel 2.10 以降で利用可能

このコンポーネントは、Apache HBase のべき等リポジトリー、プロデューサー、およびコンシューマーを提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-hbase</artifactId>
   <version>2.17.0.redhat-630xxx</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

APACHE HBASE の概要

HBase は、Google の Bigtable: A Distributed Storage System for Structured Data の後にモデル化された、オープンソースの分散型バージョン管理された列指向のストアです。Big Data へのランダムでリアルタイム読み取り/書き込みアクセスが必要な場合は、HBase を使用できます。詳細は、Apache HBase を参照してください。

CAMEL および HBASE

camel ルート内で datasotre を使用する場合は、Camel メッセージがデータストアにどのように保存されるかを指定する常にチャーンがあります。ドキュメントベースのストアでは、メッセージボディーをドキュメントに直接マッピングできるため、より簡単になります。リレーショナルデータベースでは、ORM ソリューションを使用してプロパティーを列などにマップできます。列ベースのストアには、このようなマッピングを実行する標準的な方法がないため、より困難です。

HBase は、さらに 2 つの課題を追加します。

- *HBase は列をファミリーにグループ化するため、命名規則を使用してプロパティーを列にマッピングするだけでは不十分です。*

- *HBase には 型の概念がありません。つまり、すべてを byte[] として格納し、byte[] が String、Number、シリアル化された Java オブジェクト、またはバイナリーデータのみを表しているかどうかは認識されません。*

これらの課題を解決するために、camel-hbase はメッセージヘッダーを使用して、メッセージの HBase 列へのマッピングを指定します。また、HBase データをモデル化し、xml/json との間で簡単に変換できる camel-hbase 提供クラスを使用する機能も提供します。最後に、ユーザーが独自のマッピングストラテジーを実装して使用する機能を提供します。

camel-hbase マッピングストラテジーに関係なく、メッセージを org.apache.camel.component.hbase.model.HBaseData オブジェクトに変換し、そのオブジェクトを内部操作に使用します。

コンポーネントの設定

HBase コンポーネントは、カスタム HBaseConfiguration オブジェクトをプロパティーとして指定するか、クラスパスにある HBase 関連のリソースに基づいて、独自の HBase 設定オブジェクトを作成できます。

```
<bean id="hbase" class="org.apache.camel.component.hbase.HBaseComponent">
   <property name="configuration" ref="config"/>
</bean>
```

コンポーネントに設定オブジェクトが指定されていない場合、コンポーネントはこれを作成します。作成した設定は、設定を取り出す hbase-site.xml ファイルのクラスパスを検索します。HBase クライアントの設定方法の詳細は、HBase client configuration and dependenciesを参照してください。

HBASE プロデューサー

上記のように、Camel は HBase の produers エンドポイントを提供します。これにより、Camel ルートを使用して HBase からデータを保存、削除、取得、またはクエリーできます。

```
hbase://table[?options]
```

table はテーブル名です。

サポートされる操作は以下のとおりです。

- ***Put***
- ***Get***
- ***削除***
- ***スキャン***

プロデューサーでサポートされる URI オプション

名前	デフォルト値	説明
operation	**CamelHBasePut**	実行する HBase 操作。**サポートされる値**:**CamelHBasePut**、**CamelHBaseGet**、**CamelHBaseDelete**、および **CamelHBaseScan**。
maxResults	**100**	スキャンする行の最大数。**サポートされる操作**:**CamelHBaseScan**
mappingStrategyName	**header**	Camel メッセージを HBase 列にマッピングするために使用するストラテジー。サポートされる値は、**ヘッダー**、または **ボディー** です。
mappingStrategyClassName	**null**	カスタムマッピングストラテジー実装のクラス名。
filters	**null**	フィルターの一覧。**サポートされる操作**:**CamelHBaseScan**。

userGroupInformation	**UserGroupInformation**	**Camel 2.17:** Kerberos を使用する場合など、HBase と通信する権限を定義します。
row.xxx	**null**	**Camel 2.17:** キー/値を HBaseRow モデルにマッピングするのに使用されます。**Camel 2.17 以降** では、マッピングでは **row.** 接頭辞を使用する必要があります。キーは、ヘッダーマッピングテーブルに以下に一覧表示されます。たとえば、以下のようになります。**row.family=info&row.qualifier=firstName&row.family2=birthdate&row.qualifier2=year**

ヘッダーマッピングのオプション:

名前	デフォルト値	説明
rowId		行の ID。これは、通常 Exchange ごとの行の変更として使用が制限されています。
rowType	文字列	行 ID に対応するタイプ。**サポートされる操作**:**CamelHBaseScan**。
family		列ファミリー。複数の列を参照するための数字接尾辞を **サポート** します。
修飾子		列修飾子。複数の列を参照するための数字接尾辞を **サポート** します。
value		値。複数の列を参照するための数字接尾辞を **サポート** します。
valueType	文字列	値のタイプ。複数の列を参照するための数字接尾辞をサポートします。**サポートされる操作**:**CamelHBaseGet** および **CamelHBaseScan**。

操作を配置します。

HBase は列ベースのストアで、特定の行の特定の列にデータを保存できます。列はファミリーにグループ化されるため、列を指定するには、列ファミリーとその列の修飾子を指定する必要があります。

特定の列にデータを保存するには、列と行の両方を指定する必要があります。

camel ルートから HBase にデータを保存する最も単純なシナリオでは、メッセージボディーの一部を指定の HBase 列に保存します。

```
<route>
  <from uri="direct:in"/>
  <!-- Set the HBase Row -->
  <setHeader headerName="CamelHBaseRowId">
    <el>${in.body.id}</el>
  </setHeader>
  <!-- Set the HBase Value -->
  <setHeader headerName="CamelHBaseValue">
    <el>${in.body.value}</el>
  </setHeader>
  <to uri="hbase:mytable?
operation=CamelHBasePut&amily=myfamily&ualifier=myqualifier"/>
</route>
```

上記のルートは、メッセージボディーに id および value プロパティーを持つオブジェクトが含まれ、値の内容を id で指定された行の HBase 列 myfamily:myqualifier に保存します。複数の列と値のペアを指定する必要がある場合は、追加の列マッピングを指定できます。2 番目のヘッダーからの数字（例： RowId2, RowId3, RowId4 など）を使用する必要があります。最初のヘッダーのみには数字 1 がありません。

```
<route>
  <from uri="direct:in"/>
  <!-- Set the HBase Row 1st column -->
  <setHeader headerName="CamelHBaseRowId">
    <el>${in.body.id}</el>
  </setHeader>
  <!-- Set the HBase Row 2nd column -->
  <setHeader headerName="CamelHBaseRowId2">
    <el>${in.body.id}</el>
  </setHeader>
  <!-- Set the HBase Value for 1st column -->
  <setHeader headerName="CamelHBaseValue">
    <el>${in.body.value}</el>
  </setHeader>
  <!-- Set the HBase Value for 2nd column -->
  <setHeader headerName="CamelHBaseValue2">
    <el>${in.body.othervalue}</el>
  </setHeader>
  <to uri="hbase:mytable?
operation=CamelHBasePut&amily=myfamily&ualifier=myqualifier&amily2=myfamily&ualifier2=
myqualifier2"/>
</route>
```

uri オプション、メッセージヘッダー、またはその両方の組み合わせを使用できることを覚えておく

ことが重要です。定数を URI の一部として指定し、動的な値をヘッダーとして指定することが推奨されます。何かがヘッダーとして定義され、uri の一部として定義されている場合、ヘッダーが使用されます。

操作を取得します。

Get Operation は、指定された HBase 行から 1 つ以上の値を取得するために使用される操作です。取得する値を指定するには、uri の一部として指定するか、メッセージヘッダーとして指定できます。

```
<route>
    <from uri="direct:in"/>
    <!-- Set the HBase Row of the Get -->
    <setHeader headerName="CamelHBaseRowId">
        <el>${in.body.id}</el>
    </setHeader>
    <to uri="hbase:mytable?
operation=CamelHBaseGet&amily=myfamily&ualifier=myqualifier&alueType=java.lang.Long"/
>
    <to uri="log:out"/>
</route>
```

上記の例では、get 操作の結果は CamelHBaseValue という名前のヘッダーとして保存されます。

操作を削除します。

camel-hbase で HBase delete 操作を実行することもできます。delete 操作は、行全体を削除します。指定する必要があるのは、メッセージヘッダーの一部として 1 つ以上の行になります。

```
<route>
    <from uri="direct:in"/>
    <!-- Set the HBase Row of the Get -->
    <setHeader headerName="CamelHBaseRowId">
        <el>${in.body.id}</el>
    </setHeader>
    <to uri="hbase:mytable?operation=CamelHBaseDelete"/>
</route>
```

スキャン操作。

スキャン操作は HBase のクエリーと同等です。スキャン操作を使用して、複数の行を取得できます。結果の一部となる列を指定し、値をオブジェクトに変換する方法を指定するには、uri オプションまたはヘッダーを使用します。

```
<route>
```

```
        <from uri="direct:in"/>
        <to uri="hbase:mytable?
operation=CamelHBaseScan&amily=myfamily&ualifier=myqualifier&alueType=java.lang.Long
&owType=java.lang.String"/>
        <to uri="log:out"/>
    </route>
```

この場合、結果を制限するためにフィルターの一覧も指定する必要があります。フィルターのリストを uri の一部として指定できます。Camel はフィルターすべてを満たす行のみ を 返します。メッセージの一部である情報を認識するフィルターを設定するには、Camel は ModelAwareFilter を定義します。これにより、フィルターがメッセージとマッピングストラテジーで定義されるモデルを考慮することができます。ModelAwareFilter camel-hbase を使用すると、選択したマッピングストラテジーが in メッセージに適用され、マッピングをモデル化する オブジェクトが作成され、そのオブジェクトを Filter に渡します。

たとえば、メッセージヘッダーとして を使用してスキャンを実行するには、以下のように ModelAwareColumnMatchingFilter を使用します。

```
    <route>
        <from uri="direct:scan"/>
        <!-- Set the Criteria -->
        <setHeader headerName="CamelHBaseFamily">
            <constant>name</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseQualifier">
            <constant>first</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseValue">
            <el>in.body.firstName</el>
        </setHeader>
        <setHeader headerName="CamelHBaseFamily2">
            <constant>name</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseQualifier2">
            <constant>last</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseValue2">
            <el>in.body.lastName</el>
        </setHeader>
        <!-- Set additional fields that you want to be return by skipping value -->
        <setHeader headerName="CamelHBaseFamily3">
            <constant>address</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseQualifier3">
            <constant>country</constant>
        </setHeader>
        <to uri="hbase:mytable?operation=CamelHBaseScan&ilters=#myFilterList"/>
    </route>

    <bean id="myFilters" class="java.util.ArrayList">
        <constructor-arg>
            <list>
```

```
                <bean
class="org.apache.camel.component.hbase.filters.ModelAwareColumnMatchingFilter"/>
            </list>
        </constructor-arg>
    </bean>
```

上記のルートは、pojo プロパティーが firstName で、lastName プロパティーがメッセージボディーとして渡されると仮定し、これらのプロパティーを取得し、それらをメッセージヘッダーの一部として追加します。デフォルトのマッピングストラテジーは、ヘッダーを HBase 列にマップするモデルオブジェクトを作成し、そのモデルを ModelAwareColumnMatchingFilter に渡します。フィルターは、モデルに一致する列が含まれていない行を除外します。サンプルによるクエリーに似ています。

HBASE コンシューマー

Camel HBase Consumer は、指定された HBase テーブルで繰り返しスキャンを実行し、メッセージの一部としてスキャン結果を返します。ヘッダーマッピング（デフォルト）またはボディーマッピングのいずれかを指定できます。後ほど、org.apache.camel.component.hbase.model.HBaseData をメッセージボディーの一部として追加します。

```
hbase://table[?options]
```

uri オプションの一部として、返す列とそのタイプを指定できます。

```
hbase:mutable?
family=name&qualifer=first&valueType=java.lang.String&family=address&qualifer=number&v
alueType2=java.lang.Integer&rowType=java.lang.Long
```

上記の例では、指定されたフィールドで設定されるモデルオブジェクトを作成し、スキャン結果によりモデルオブジェクトに値が設定されます。最後に、マッピングストラテジーは、このモデルを Camel メッセージにマッピングするために使用されます。

コンシューマーでサポートされている URI オプション

名前	デフォルト値	説明
initialDelay	1000	最初のポーリングが開始されるまでの時間（ミリ秒単位）。
delay	500	次のポーリングまでの時間 (ミリ秒単位)。

useFixedDelay	**true**	固定遅延または固定レートを使用するかどうかを制御します。詳細は、JDK の ScheduledExecutorService を参照してください。
timeUnit	**TimeUnit.MILLISECONDS**	**initialDelay** および **delay** オプションの時間単位。
runLoggingLevel	**TRACE**	**Camel 2.8:** コンシューマーはポーリング時に開始/完了のログ行をログに記録します。このオプションを使用すると、ログレベルを設定できます。
operation	**CamelHBasePut**	実行する HBase 操作。**サポートされる値:CamelHBasePut**、**CamelHBaseGet**、**CamelHBaseDelete**、および **CamelHBaseScan**。
maxResults	**100**	スキャンする行の最大数。**サポートされる操作：CamelHBaseScan**
mappingStrategyName	**header**	Camel メッセージを HBase 列にマッピングするために使用するストラテジー。サポートされる値は、**ヘッダー**、または **ボディー** です。
mappingStrategyClassName	**null**	カスタムマッピングストラテジー実装のクラス名。
filters	**null**	フィルターの一覧。**サポートされる操作:CamelHBaseScan**
remove	**true**	**true** の場合、Camel HBase Consumer は処理する行を削除します。
userGroupInformation	**UserGroupInformation**	**Camel 2.17:** Kerberos を使用する場合など、HBase と通信する権限を定義します。

ヘッダーマッピングのオプション：

名前	デフォルト値	説明

rowId		行の ID。これは、通常 Exchange ごとの行の変更として使用が制限されています。
rowType	文字列	行 ID に対応するタイプ。**サポートされる操作**: CamelHBaseScan
family		列ファミリー。**upports には、複数の列を参照するための数字の接尾辞が含まれます。
修飾子		列修飾子。*複数の列を参照するための数字接尾辞をサポートします。
value		値。複数の列を参照するための数字接尾辞をサポートします。
rowModel	文字列	各行のモデル化方法を記述する org.apache.camel.component.hbase.model.HBaseRow のインスタンス

***rowModel** のロールが明確でない場合、URI オプション（ファミリー、修飾子、タイプなど）を使用して "describ" ではなく、HBaseRow modle をプログラムで構築できます。*

HBASE IDEMPOTENT リポジトリー

***camel-hbase** コンポーネントは、各メッセージが一度だけ処理されるようにするときに使用できるべき等リポジトリーも提供します。HBase idempotent リポジトリーは、テーブル、列ファミリー、列修飾子で設定され、そのテーブルにメッセージごとに行を作成します。*

```
HBaseConfiguration configuration = HBaseConfiguration.create();
HBaseIdempotentRepository repository = new HBaseIdempotentRepository(configuration,
tableName, family, qualifier);

from("direct:in")
  .idempotentConsumer(header("messageId"), repository)
  .to("log:out);
```

HBASE マッピング

前述のように、デフォルトのマッピングストラテジーが ヘッダー および ボディー マッピングであることが分かります。以下は、各マッピングストラテジーの仕組みの詳細な例になります。

HBASE ヘッダーマッピングの例

ヘッダーマッピングはデフォルトのマッピングです。値 myvalue を HBase 行 myrow と列 myfamily:mycolum に指定するには、メッセージには以下のヘッダーが含まれている必要があります。

ヘッダー	値
CamelHBaseRowId	myrow
CamelHBaseFamily	myfamily
CamelHBaseQualifier	myqualifier
CamelHBaseValue	myvalue

異なる列および / または異なる行により多くの値を配置するには、ヘッダーのインデックスで接尾辞が付いたヘッダーを追加で指定できます。以下に例を示します。

ヘッダー	値
CamelHBaseRowId	myrow
CamelHBaseFamily	myfamily
CamelHBaseQualifier	myqualifier
CamelHBaseValue	myvalue
CamelHBaseRowId2	myrow2
CamelHBaseFamily2	myfamily
CamelHBaseQualifier2	myqualifier
CamelHBaseValue2	myvalue2

get や scan などの取得操作では、データの変換先となるタイプごとに、各列に を指定することもできます。以下に例を示します。

ヘッダー	値

CamelHBaseFamily	myfamily
CamelHBaseQualifier	myqualifier
CamelHBaseValueType	Long

すべてのメッセージに対して定数とみなされるボイラープレートヘッダーを回避するために、以下のようにエンドポイント URI の一部として指定することもできます。

ボディーマッピングの例

ボディーマッピングストラテジーを使用するには、URI の一部としてオプション mappingStrategy を指定する必要があります。以下に例を示します。

```
hbase:mytable?mappingStrategy=body
```

ボディーマッピングストラテジーを使用するには、ボディーに org.apache.camel.component.hbase.model.HBaseData のインスタンスが含まれている必要があります。構築できます

```
HBaseData data = new HBaseData();
HBaseRow row = new HBaseRow();
row.setId("myRowId");
HBaseCell cell = new HBaseCell();
cell.setFamily("myfamily");
cell.setQualifier("myqualifier");
cell.setValue("myValue");
row.getCells().add(cell);
data.addRows().add(row);
```

上記のオブジェクトは put 操作で使用することができます。たとえば、ID myRowId で行を作成または更新し、値 myvalue を列 myfamily:myqualifier に追加します。ボディーマッピングストラテジーが最初に認識されない可能性があります。ヘッダーマッピングストラテジーに対する利点は、HBaseData オブジェクトを xml/json との間で簡単に変換できることです。

その他の参考資料

- Polling Consumer

Apache HBase

第66章 HDFS

HDFS コンポーネント

Camel 2.8 から利用可能

hdfs コンポーネントを使用すると、HDFS ファイルシステムとの間でメッセージを読み書きできます。HDFS は、Hadoop の中心となる分散ファイルシステムです。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-hdfs</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
hdfs://hostname[:port][/path][?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。パスは以下のように処理されます。

1. コンシューマーとして、ファイルの場合、ファイルを読み取るだけです。それ以外の場合は、設定されたパターンを満たすパス下のすべてのファイルをスキャンするディレクトリーを表します。このディレクトリー下のすべてのファイルは同じタイプである必要があります。

2. プロデューサーとして、少なくとも 1 つの分割ストラテジーが定義されている場合、パスはディレクトリーと見なされ、そのディレクトリーの下に、プロデューサーは設定された UuidGenerator を使用して分割された名前ごとに異なるファイルを作成します。

注記

通常モードで HDFS から消費する場合、ファイルはチャンクに分割され、チャンクごとにメッセージを生成します。chunkSize オプションを使用して、チャンクのサイズを設定できます。File コンポーネントを使用して HDFS から読み取り、通常のファイルに書き込む場合は、fileMode=Append を設定してチャンクを連結できます。

オプション

名前	デフォルト値	説明
overwrite	**true**	ファイルは上書きできる
append	**false**	既存のファイルに追加します。すべての HDFS ファイルシステムが append オプションをサポートしているわけではないことに注意してください。
bufferSize	**4096**	HDFS が使用するバッファーサイズ
レプリケーション	**3**	HDFS レプリケーション係数
blockSize	**67108864**	HDFS ブロックのサイズ
fileType	**NORMAL_FILE**	SEQUENCE_FILE、MAP_FILE、ARRAY_FILE、または BLOOMMAP_FILE を指定できます。Hadoop を参照してください。
fileSystemType	**HDFS**	ローカルファイルシステムの LOCAL にすることができます。
keyType	**NULL**	シーケンスまたはマップファイルの場合のキーのタイプ。以下を参照してください。
valueType	**TEXT**	シーケンスまたはマップファイルの場合のキーのタイプ。以下を参照してください。
splitStrategy		異なる基準に基づいてファイルを分割する方法に関するストラテジーを記述する文字列。以下を参照してください。
openedSuffix	**開放済み**	読み取り/書き込み用にファイルが開かれると、書き込みフェーズでファイルを読み取らないように、この接尾辞でファイルの名前が変更されます。
readSuffix	**read**	ファイルを読み込んだら、再度読み取るのを回避するために、この接尾辞で名前を変更します。
initialDelay	**0**	コンシューマーの場合、ディレクトリーのスキャンを開始する前に待機する時間（ミリ秒単位）。

delay	**0**	ディレクトリースキャンの間隔（ミリ秒単位）。
pattern	*	ディレクトリーのスキャンに使用されるパターン
chunkSize	**4096**	通常のファイルを読み取る場合、これはチャンクごとにメッセージを生成するチャンクに分割されます。
connectOnStartup	**true**	**Camel 2.9.3/** 2.10.1: プロデューサー/コンシューマーの開始時に HDFS ファイルシステムに接続するかどうか。**false** の場合、接続はオンデマンドで作成されます。HDFS は、45 x 20 秒再配信をハードコーディングしているため、接続を確立するために最大 15 分かかる可能性があることに注意してください。このオプションを **false** に設定すると、アプリケーションは起動できるようになり、最大 15 分間ブロックされません。
owner		**Camel 2.13/2.12.4:** コンシューマーがファイルを取得するには、ファイルの所有者はこの所有者と一致する必要があります。それ以外の場合は、ファイルはスキップされます。

KEYTYPE および VALUETYPE

- ***NULL これは、キーまたは値が存在しないことを意味します。***
- ***バイトを書き込む BYTE では、java バイトクラスは BYTE にマッピングされます。***
- ***バイトのシーケンスを書き込む BYTES。java ByteBuffer クラスをマッピングします。***
- ***java 整数を書き込む INT***
- ***java 浮動小数点を書き込む FLOAT***

- java long を記述する LONG
- Java の二重書き込み用の DOUBLE
- java 文字列を書き込む TEXT

BYTES は他のすべてでも使用されます。たとえば、Camel ではファイルが InputStream として送信されます。この例では、シーケンスファイルまたはマップファイルでバイトのシーケンスとして記述されます。

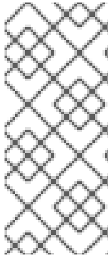
分割ストラテジー

現行バージョンでは、addment モードでファイルを開くと、信頼性が十分にないため、無効になっています。そのため、新しいファイルしか作成できません。Camel HDFS エンドポイントは、以下の方法でこの問題の解決を試みます。

- split strategy オプションが定義されている場合、hdfs パスがディレクトリーとして使用され、設定された UuidGenerator を使用してファイルが作成されます。
- 分割条件が満たされるたびに、新しいファイルが作成されます。splitStrategy オプションは、以下の構文で文字列として定義されます。 splitStrategy=<ST>:<value>,<ST>:<value>,*

<ST> は以下のとおりです。

- BYTES は新しいファイルが作成され、書き込まれたバイト数が <value> を超えると古いファイルが閉じられます。
- MESSAGES は新しいファイルが作成され、書き込まれたメッセージの数が <value> を超える場合に古いファイルが閉じられます。
- IDLE 新しいファイルが作成され、最後の <value> ミリ秒で書き込みが行われなかった場合に古いファイルが閉じられます。



注記

*現在、このストラテジーでは、**IDLE** 値を設定するか、**HdfsConstants.HDFS_CLOSE** ヘッダーを **false** に設定して **BYTES/MESSAGES** 設定を使用する必要があります。そうしないと、ファイルは各メッセージで閉じられます。*

以下に例を示します。

```
hdfs://localhost/tmp/simple-file?splitStrategy=IDLE:1000,BYTES:5
```

*つまり、新しいファイルは 1 秒以上アイドル状態であるか、または 5 バイト以上書き込まれている場合に作成されます。そのため、**hadoop fs ls /tmp/simplefile** を実行すると、複数のファイルが作成されていることがわかります。*

メッセージヘッダー

このコンポーネントでは、以下のヘッダーがサポートされます。

プロデューサーのみ

ヘッダー	説明
CamelFileName	Camel 2.13: 書き込むファイルの名前を指定します（エンドポイントパスに相対的）。名前は **String** または Expression オブジェクトにすることができます。分割ストラテジーを使用しない場合のみ関連します。

ファイルストリームを閉じるための制御

Camel 2.10.4 以降で利用可能

*分割ストラテジー なし で HDFS プロデューサーを使用する場合、ファイル出力ストリームはデフォルトで書き込み後に閉じられます。ただし、ストリームを開いたままにし、後でストリームを明示的に閉じることもできます。そのため、ヘッダー **HdfsConstants.HDFS_CLOSE (value = "CamelHdfsClose")**を使用してこれを制御することができます。この値をブール値に設定すると、ストリームを閉じるべきかどうかを明示的に制御できます。*

これは、スプリットストラテジーを使用する場合には適用されないことに注意してください。これは、ストリームが閉じられるタイミングを制御できるさまざまなストラテジーがあるためです。

OSGI でのこのコンポーネントの使用

このコンポーネントは OSGi 環境で完全に機能しますが、ユーザーからのいくつかのアクションが必要になります。Hadoop は、リソースをロードするためにスレッドコンテキストクラ出力ダーを使用します。通常、スレッドコンテキストクラ出力ダーは、ルートを含むバンドルのバンドルクラ出力ダーになります。そのため、デフォルトの設定ファイルはバンドルクラ出力ダーから見える必要があります。これに対処する一般的な方法は、バンドルルートに core-default.xml のコピーを保持することです。このファイルは hadoop-common.jar にあります。

第67章 HDFS2

HDFS2 コンポーネント

Camel 2.13 で利用可能

hdfs2 コンポーネントを使用すると、Hadoop 2.x を使用して HDFS ファイルシステムとの間でメッセージを読み書きできます。HDFS は、Hadoop の中心となる分散ファイルシステムです。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-hdfs2</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

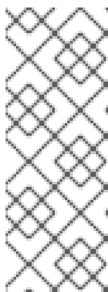
URI 形式

```
hdfs2://hostname[:port][/path][?options]
```

?option=value&option=value&... の形式で、クエリーオプションを URI に追加できます。パスは以下の方法で処理されます。

1. コンシューマーとして、ファイルの場合、ファイルを読み取るだけです。それ以外の場合は、設定されたパターンを満たすパス下のすべてのファイルをスキャンするディレクトリーを表します。このディレクトリー下のすべてのファイルは同じタイプである必要があります。

2. プロデューサーとして、少なくとも 1 つの分割ストラテジーが定義されている場合、パスはディレクトリーと見なされ、そのディレクトリー下で、プロデューサーは設定された UuidGenerator を使用して、分割ごとに異なるファイルを作成します。



注記

通常モードで HDFS から消費する場合、ファイルはチャンクに分割され、チャンクごとにメッセージを生成します。chunkSize オプションを使用して、チャンクのサイズを設定できます。File コンポーネントを使用して HDFS から読み取り、通常のファイルに書き込む場合は、fileMode=Append を設定してチャンクを連結できます。

オプション

名前	デフォルト値	説明
overwrite	**true**	ファイルは上書きできる
append	**false**	既存のファイルに追加します。すべての HDFS ファイルシステムが append オプションをサポートしているわけではないことに注意してください。
bufferSize	**4096**	HDFS が使用するバッファーサイズ
replication	**3**	HDFS レプリケーション係数
blockSize	**67108864**	HDFS ブロックのサイズ
fileType	**NORMAL_FILE**	SEQUENCE_FILE、MAP_FILE、ARRAY_FILE、または BLOOMMAP_FILE を指定できます。Hadoop を参照してください。
fileSystemType	**HDFS**	ローカルファイルシステムの LOCAL にすることができます。
keyType	**NULL**	シーケンスまたはマップファイルの場合のキーのタイプ。以下を参照してください。
valueType	**TEXT**	シーケンスまたはマップファイルの場合のキーのタイプ。以下を参照してください。
splitStrategy		異なる基準に基づいてファイルを分割する方法に関するストラテジーを記述する文字列。以下を参照してください。

openedSuffix	**opened**	読み取り/書き込みのためにファイルを開くと、書き込みフェーズでファイルを読み取らないように、この接尾辞でファイルの名前が変更されます。
readSuffix	**read**	ファイルを読み込んだら、再度読み取るのを回避するために、この接尾辞で名前を変更します。
initialDelay	**0**	コンシューマーの場合、ディレクトリーのスキャンを開始する前に待機する時間（ミリ秒単位）。
delay	**0**	ディレクトリースキャンの間隔（ミリ秒単位）。
pattern	*	ディレクトリーのスキャンに使用されるパターン
chunkSize	**4096**	通常のファイルを読み取る場合、これはチャンクごとにメッセージを生成するチャンクに分割されます。
connectOnStartup	**true**	**Camel 2.9.3/** 2.10.1: プロデューサー/コンシューマーの開始時に HDFS ファイルシステムに接続するかどうか。**false** の場合、接続はオンデマンドで作成されます。HDFS は、45 x 20 秒再配信をハードコーディングしているため、接続を確立するために最大 15 分かかる可能性があることに注意してください。このオプションを **false** に設定すると、アプリケーションは起動でき、最大 15 分間ブロックされません。
owner		コンシューマーがファイルを選択できるようにするには、ファイルの所有者はこの所有者と一致する必要があります。それ以外の場合は、ファイルはスキップされます。

KEYTYPE および VALUETYPE

- *NULL これは、キーまたは値が存在しないことを意味します。*

- *バイトを書き込む BYTE では、java バイトクラスは BYTE にマッピングされます。*

- *バイトのシーケンスを書き込む **BYTES**。java ByteBuffer クラスをマッピングします。*
- *java 整数を書き込む **INT***
- *java 浮動小数点を書き込む **FLOAT***
- *java long を記述する **LONG***
- ***Java の二重書き込み用の DOUBLE***
- *java 文字列を書き込む **TEXT***

BYTES は他のすべてでも使用されます。たとえば、Camel ではファイルが InputStream として送信されます。この例では、シーケンスファイルまたはマップファイルでバイトのシーケンスとして記述されます。

分割ストラテジー

現行バージョンでは、addment モードでファイルを開くことは、信頼性が非常にないため、無効になっています。そのため、新しいファイルしか作成できません。Camel HDFS エンドポイントは、以下の方法でこの問題の解決を試みます。

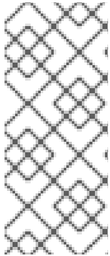
- *split strategy オプションが定義されている場合、hdfs パスがディレクトリーとして使用され、設定された UuidGeneratorを使用してファイルが作成されます。*
- *分割条件が満たされるたびに、新しいファイルが作成されます。splitStrategy オプションは、splitStrategy=<ST>:<value>,<ST>:<value>,* 構文で文字列として定義されます。*

ここで、<ST> は以下のようになります。

- *BYTES は新しいファイルが作成され、書き込まれたバイト数が <value> を超えると古いファイルが閉じられます。*

- *MESSAGES は新しいファイルが作成され、書き込まれたメッセージの数が <value> を超える場合に古いファイルが閉じられます。*

- *IDLE 新しいファイルが作成され、最後の <value> ミリ秒で書き込みが行われなかった場合に古いファイルが閉じられます。*



注記

現在、このストラテジーでは、IDLE 値を設定するか、HdfsConstants.HDFS_CLOSE ヘッダーを false に設定して BYTES/MESSAGES 設定を使用する必要があります。その他の場合は、各メッセージでファイルが閉じられます。

以下に例を示します。

```
hdfs2://localhost/tmp/simple-file?splitStrategy=IDLE:1000,BYTES:5
```

つまり、新しいファイルは 1 秒以上アイドル状態であるか、または 5 バイト以上書き込まれている場合に作成されます。hadoop fs -ls /tmp/simple-file を実行すると、複数のファイルが作成されていることがわかります。

メッセージヘッダー

このコンポーネントでは、以下のヘッダーがサポートされます。

プロデューサーのみ

ヘッダー	説明
CamelFileName	Camel 2.13: 書き込むファイルの名前を指定します（エンドポイントパスに相対的）。名前は **String** または **Expression** オブジェクトにすることができます。分割ストラテジーを使用しない場合のみ関連します。

ファイルストリームを閉じるための制御

分割ストラテジー なし で HDFS2 プロデューサーを使用する場合、ファイル出力ストリームはデフォルトで書き込み後に閉じられます。ただし、ストリームを開いたままにし、後でストリームを明示的に閉じることもできます。そのため、ヘッダー HdfsConstants.HDFS_CLOSE（値 = "CamelHdfsClose"）を使用して制御することができます。この値をブール値に設定すると、ストリームを閉じるべきかどうかを明示的に制御できます。

これは、スプリットストラテジーを使用する場合には適用されないことに注意してください。これは、ストリームが閉じられるタイミングを制御できるさまざまなストラテジーがあるためです。

OSGI でのこのコンポーネントの使用

さまざまな org.apache.hadoop.fs.FileSystem 実装の検出に際して、Hadoop 2.x が使用するメカニズムに関連する OSGi 環境でこのコンポーネントを実行する場合には、いくつかの quirk があります。Hadoop 2.x は java.util.ServiceLoader を使用します。これは、利用可能なファイルシステムタイプと実装を定義する /META-INF/services/org.apache.hadoop.fs.FileSystem ファイルを探します。これらのリソースは、OSGi 内で実行すると利用できません。

camel-hdfs コンポーネントと同様に、デフォルトの設定ファイルはバンドルクラ出力ダーから見える必要があります。これに対処する一般的な方法は、バンドルルートに core-default.xml のコピー（例： hdfs-default.xml）を保持することです。

手動で定義されたルートでのこのコンポーネントの使用

以下の 2 つのオプションがあります。

1. ルートを定義するバンドルで /META-INF/services/org.apache.hadoop.fs.FileSystem リソースをパッケージ化します。このリソースには、必要なHadoop 2.xファイルシステムの実装がすべて含まれているはずです。

2. org.apache.hadoop.fs.FileSystem クラス内の内部かつ静的キャッシュを設定するボイラープレート初期化コードを提供します。

```
org.apache.hadoop.conf.Configuration conf = new org.apache.hadoop.conf.Configuration();
conf.setClass("fs.file.impl", org.apache.hadoop.fs.LocalFileSystem.class, FileSystem.class);
conf.setClass("fs.hdfs.impl", org.apache.hadoop.hdfs.DistributedFileSystem.class,
FileSystem.class);
...
FileSystem.get("file:///", conf);
FileSystem.get("hdfs://localhost:9000/", conf);
...
```

BLUEPRINT コンテナーでのこのコンポーネントの使用

2 つのオプション:

1. *Blueprint 定義を含むバンドルを使用して /META-INF/services/org.apache.hadoop.fs.FileSystem リソースをパッケージ化します。*

2. *Blueprint 定義ファイルに以下を追加します。*

```
<bean id="hdfsOsgiHelper" class="org.apache.camel.component.hdfs2.HdfsOsgiHelper">
  <argument>
    <map>
      <entry key="file:///" value="org.apache.hadoop.fs.LocalFileSystem"  />
      <entry key="hdfs://localhost:9000/"
value="org.apache.hadoop.hdfs.DistributedFileSystem" />
      ...
    </map>
  </argument>
</bean>

<bean id="hdfs2" class="org.apache.camel.component.hdfs2.HdfsComponent" depends-
on="hdfsOsgiHelper" />
```

このようにして、Hadoop 2.x は URI スキームをファイルシステム実装に正しくマッピングします。

第68章 HIPCHAT

HIPCHAT コンポーネント

Camel 2.15.0 から利用可能

Hipchat コンポーネントは、Hipchat サービスからのメッセージの生成と消費をサポートします。

有効な Hipchat ユーザーアカウントがあり、メッセージの生成/消費に使用できる 個人アクセストークン を取得する必要があります。

URI 形式

> ***hipchat://[host][:port]?options***

URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	必須	producer/Consumer	説明
protocol	**null**	共有	はい	両方	Hipchat サーバーに接続するためのデフォルトのプロトコル
host	**null**	共有	はい	両方	接続する Hipchat の API ホスト
port	80	共有	いいえ	両方	Hipchat ホストで接続するポート
authToken	**null**	共有	はい	両方	Hipchatから取得した認証トークン（パーソナルアクセストークン）

delay	5000	共有	いいえ	コンシューマー	提供された consumeUsers からのメッセージを消費するための millisec のポーリング間隔。これを減らす前に、流量制御 についてお読みください。
consumeUsers	**null**	共有	いいえ	コンシューマー	**authToken** の所有者にメッセージを消費する必要があるユーザー @Mentions またはメールのコンマ区切りリスト。

スケジュールされたポーリングコンシューマー

このコンポーネントは、ScheduledPollConsumer を実装します。指定された 'consumeUsers' の最後のメッセージのみが取得され、エクスチェンジボディーとして送信されます。次のポーリングに新しいメッセージがない場合に、同じメッセージを再度取得しない場合は、以下のように べき等コンシューマー を追加できます。ScheduledPollConsumer のすべてのオプションは、コンシューマーでの制御にも使用できます。

```
@Override
public void configure() throws Exception {
 String hipchatEndpointUri = "hipchat://?authToken=XXXX&consumeUsers=@Joe,@John";
 from(hipchatEndpointUri)
  .idempotentConsumer(
    simple("${in.header.HipchatMessageDate} ${in.header.HipchatFromUser}"),
    MemoryIdempotentRepository.memoryIdempotentRepository(200)
  )
  .to("mock:result");
}
```

HIPCHAT コンシューマーによって設定されたメッセージヘッダー

ヘッダー	Constant	タイプ	説明
HipchatFromUser	HipchatConstants.FROM_USER	**文字列**	ボディーには、このユーザーから authToken の所有者に送信されたメッセージがあります。
HipchatMessageDate	HipchatConstants.MESSAGE_DATE	**文字列**	日付メッセージが送信された。形式は、Hipchat 応答 にある ISO-8601 です。
HipchatFromUserResponseStatus	HipchatConstants.FROM_USER_RESPONSE_STATUS	**StatusLine**	受信した API 応答のステータス。

HIPCHAT プロデューサー

プロデューサーは、Room と User の両方へ同時にメッセージを送信できます。エクスチェンジのボディーはメッセージとして送信されます。使用例を以下に示します。適切なヘッダーを設定する必要があります。

```
@Override
 public void configure() throws Exception {
  String hipchatEndpointUri = "hipchat://?authToken=XXXX";
  from("direct:start")
   .to(hipchatEndpointUri)
   .to("mock:result");
 }
```

HIPCHAT プロデューサーによって評価されるメッセージヘッダー

ヘッダー	Constant	タイプ	説明
HipchatToUser	HipchatConstants.TO_USER	**文字列**	メッセージを送信する必要がある Hipchat ユーザー。
HipchatToRoom	HipchatConstants.TO_ROOM	**文字列**	メッセージを送信する必要のある Hipchat 部屋。
HipchatMessageFormat	HipchatConstants.MESSAGE_FORMAT	文字列	有効な形式は 'text' または 'html' です。**デフォルト：'text'**
HipchatMessageBackgroundColor	HipchatConstants.MESSAGE_BACKGROUND_COLOR	**文字列**	有効な色の値は 'yellow'、'green'、'red'、'purple'、'gray'、'random' です。**デフォルト：'yellow'(Room のみ)**
HipchatTriggerNotification	HipchatConstants.TRIGGER_NOTIFY	**文字列**	有効な値は true または false です。このメッセージがユーザー通知をトリガーするかどうか（タブの色の変更、サウンドの再生、モバイルフォンなどへの通知など）。**デフォルト：'false'(Room のみ)**

HIPCHAT プロデューサーによって設定されたメッセージヘッダー

ヘッダー	Constant	タイプ	説明
HipchatToUserResponseStatus	HipchatConstants.TO_USER_RESPONSE_STATUS	**StatusLine**	メッセージがユーザーに送信されたときに受信した API 応答のステータス。
HipchatFromUserResponseStatus	HipchatConstants.TO_ROOM_RESPONSE_STATUS	**StatusLine**	部屋に送信されたメッセージ時に受信した API 応答のステータス。

DEPENDENCIES

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-hipchat</artifactId>
    <version>${camel-version}</version>
</dependency>
```

ここで、${camel-version} は実際のバージョンの Camel (2.15.0 以降)に置き換える必要があります。

第69章 HL7

HL7 コンポーネント

HL7 コンポーネントは、HAPI ライブラリー を使用して HL7 MLLP プロトコルおよび HL7 v2 メッセージ を操作するために使用されます。

このコンポーネントは以下をサポートします。

- HL7 MLLP codec (Mina)
- Camel 2.15 以降の Netty4 の HL7 MLLP コーデック
- HAPI および文字列間の 型コンバーター
- HAPI ライブラリーを使用した HL7 DataFormat
- さらに使いやすいため、105章MINA2 - 非推奨 コンポーネントと統合されています。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-hl7</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

HL7 MLLP プロトコル

HL7 は、テキストベースの TCP ソケットベースのプロトコルである HL7 MLLP プロトコルでよく使用されます。このコンポーネントには、MLLP プロトコルに準拠する Mina および Netty4 Codec が同梱されるため、TCP トランスポート層で HL7 リクエストを受け入れる HL7 リスナーを簡単に公開できます。

HL7 リスナーサービスを公開するには、camel-mina2 または camel-netty4 コンポーネントが HL7MLLPCodec (mina2)または HL7MLLPNettyDecoder/HL7MLLPNettyEncoder (Netty4)とともに使用されます。

HL7 MLLP コーデックには、以下のオプションがあります。

名前	デフォルト値	説明
startByte	**0x0b**	HL7 ペイロードにまたがる開始バイト。
endByte1	**0x1c**	HL7 ペイロードにまたがる最初のエンドバイト。
endByte2	**0x0d**	HL7 ペイロードにまたがる 2 番目の終了バイト。
charset	JVM のデフォルト	codec に使用するエンコーディング(文字セット名)。指定しない場合、Camel は JVM のデフォルト Charset を 使用します。
produceString	**true**	Camel 2.14.1: **true** の場合、コーデックは定義された charset を使用して文字列を作成します。**false** の場合、コーデックはプレーンなバイトアレイをルートに送信し、HL7 Data Format が HL7 メッセージコンテンツから実際の文字セットを決定できるようにします。
convertLFtoCR	**false**	は、**\n** を **\r** (**0x0d**、13 進数)に変換します。HL7 は、セグメントターミネーターとして **\r** を使用します。HAPI ライブラリーには **\r** を使用する必要があります。

MINA を使用した HL7 リスナーの公開

Spring XML ファイルでは、ポート 8888 で TCP を使用して HL7 要求をリッスンするように Mina2 エンドポイントを設定します。

```
<endpoint id="hl7MinaListener" uri="mina2:tcp://localhost:8888?
sync=true&amp;codec=#hl7codec"/>
```

sync=true は、このリスナーが同期されているため、呼び出し元に HL7 応答を返すことを示します。HL7 コーデックが codec=#hl7codec で設定されている。hl7codec は Spring Bean ID であるため、mygreatcodecforhl7 または任意の名前を付けることができます。codec も Spring XML ファイルで設定されます。

```
<bean id="hl7codec" class="org.apache.camel.component.hl7.HL7MLLPCodec">
    <property name="charset" value="iso-8859-1"/>
</bean>
```

この Java DSL の例が示すように、エンドポイント hl7MinaListener はルートでコンシューマーとして使用できます。

```
from("hl7MinaListener").beanRef("patientLookupService");
```

これは、HL7 をリッスンし、HL7 という名前のサービスにルーティングする非常に単純 なルートです。これは Spring Bean ID で、以下のように Spring XML で設定されます。

```
    <bean id="patientLookupService"
class="com.mycompany.healthcare.service.PatientLookupService"/>
```

Camel のもう 1 つの強力な機能は、以下のように Camel に関連付けられていない POJO クラスにビジネスロジックを持つことができることです。

```
import ca.uhn.hl7v2.HL7Exception;
import ca.uhn.hl7v2.model.Message;
import ca.uhn.hl7v2.model.v24.segment.QRD;

public class PatientLookupService {
    public Message lookupPatient(Message input) throws HL7Exception {
        QRD qrd = (QRD)input.get("QRD");
        String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue();

        // find patient data based on the patient id and create a HL7 model object with the
response
        Message response = ... create and set response data
        return response
    }
```

このクラスは Camel からではなく HAPI ライブラリーからのインポートのみを使用することに注意してください。

NETTY を使用した HL7 リスナーの公開(CAMEL 2.15 以降から利用可能)

Spring XML ファイルでは、ポート 8888 で TCP を使用して HL7 リクエストをリッスンするように Netty4 エンドポイントを設定します。

```
<endpoint id="hl7NettyListener" uri="netty4:tcp://localhost:8888?
sync=true&amp;encoder=#hl7encoder&amp;decoder=#hl7decoder"/>
```

sync=true このリスナーが同期されているため、呼び出し元に HL7 応答を返すことを示します。HL7 コーデックは、encoder=#hl7encoder および decoder=#hl7decoder で設定されます。hl7encoder および hl7decoder は Bean ID のみであるため、名前が異なる可能性があることに注意してください。Bean は Spring XML ファイルで設定できます。

```
<bean id="hl7decoder"
class="org.apache.camel.component.hl7.HL7MLLPNettyDecoderFactory"/>
<bean id="hl7encoder"
class="org.apache.camel.component.hl7.HL7MLLPNettyEncoderFactory"/>
```

この Java DSL の例が示すように、hl7NettyListener エンドポイントはコンシューマーとしてルートで使用できます。

```
from("hl7NettyListener").beanRef("patientLookupService");
```

JAVA.LANG.STRING または BYTE[] を使用した HL7 モデル

HL7 MLLP コーデックは、プレーン String をデータ形式として使用します。Camel は Type Converter を使用して文字列を HAPI HL7 モデルオブジェクトに変換しますが、データを独自に解析する場合は、プレーン String オブジェクトを使用できます。

Camel 2.14.1 の時点で、produceString プロパティーを false に設定すると、Mina および Netty コーデックの両方がプレーン byte[] をデータ形式として使用することもできます。Type Converter は、byte[] を HAPI HL7 モデルオブジェクトとの間で変換することもできます。

HAPI を使用した HL7V2 モデル

HL7v2 モデルは、HAPI ライブラリーの Java オブジェクトを使用します。このライブラリーを使用すると、HL7v2 で主に使用される EDI 形式(ER7)からエンコードおよびデコードできます。

以下の例は、リフォルト ID 0101701234 で発行者を検索するリクエストです。

```
MSH|^~\\&|MYSENDER|MYRECEIVER|MYAPPLICATION||200612211200||QRY^A19|1234|P|2.4
QRD|200612211200|R|I|GetPatient|||1^RD|0101701234|DEM||
```

HL7 モデルを使用すると、ca.uhn.hl7v2.model.Message オブジェクトを使用して作業できます。 たとえば、音声 ID を取得することができます。

```
Message msg = exchange.getIn().getBody(Message.class);
QRD qrd = (QRD)msg.get("QRD");
String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue();  // 0101701234
```

byte[], String またはその他の単純なオブジェクト形式を使用する必要がないため、HL7 リスナーと組み合わせると便利です。HAPI HL7v2 モデルオブジェクトのみを使用できます。事前にメッセージタイプが分かっている場合は、よりタイプセーフになります。

```
QRY_A19 msg = exchange.getIn().getBody(QRY_A19.class);
String patientId = msg.getQRD().getWhoSubjectFilter(0).getIDNumber().getValue();
```

HL7 DATAFORMAT

HL7 コンポーネントには、HL7 モデルオブジェクトのマーシャリングまたはアンマーシャリングに使用できる HL7 データフォーマットが同梱されています。

- marshal = メッセージからバイトストリームへ(HL7 MLLP コーデックを使用して応答する場合に使用できます)
- unmarshal = バイトストリームからメッセージへ(HL7 MLLP からストリームデータを受信する場合に使用できます)

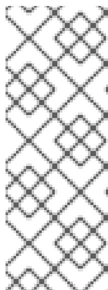
データフォーマットを使用するには、インスタンスをインスタンス化し、ルートビルダーで marshal または unmarshal 操作を呼び出します。

```
DataFormat hl7 = new HL7DataFormat();
...
from("direct:hl7in").marshal(hl7).to("jms:queue:hl7out");
```

上記の例では、HL7 は HAPI Message オブジェクトからバイトストリームにマーシャリングされ、JMS キューに配置されます。以下の例は、逆になります。

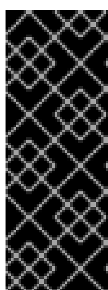
```
DataFormat hl7 = new HL7DataFormat();
...
from("jms:queue:hl7out").unmarshal(hl7).to("patientLookupService");
```

ここでは、プロキシールックアップサービスに渡される HAPI Message オブジェクトにバイトストリームをアンマーシャリングします。



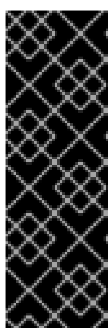
注記

HAPI 2.0 (Camel 2.11 で使用される)の時点で、HL7v2 モデルクラスは完全にシリアライズ可能です。そのため、HL7v2 メッセージを JMS キューに直接配置できます（例： marshal() を呼び出しずに）、キューから直接再度読み取ることができます（例： unmarshal()を呼び出さずに）。



重要

Camel 2.11 の時点で、unmarshal は \n から \r に変換してセグメント区切り文字を自動的に修正しません。この変換が必要な場合は、org.apache.camel.component.hl7.HL7#convertLFToCR で便利な Expression が提供されます。



重要

Camel 2.14.1 の時点で、marshal および unmarshal の両方が MSH-18 フィールドで提供された charset を評価します。このフィールドが空の場合、デフォルトでは、対応する Camel charset プロパティー/ヘッダーに含まれる charset が想定されます。HL7DataFormat クラスから継承する際に guessCharsetName メソッドを上書きすると、このデフォルトの動作を変更することもできます。

Camel には、よく知られているデータ形式用の簡略化された構文があります。HL7DataFormat オブジェクトのインスタンスを作成する必要はありません。

```
from("direct:hl7in").marshal().hl7().to("jms:queue:hl7out");
from("jms:queue:hl7out").unmarshal().hl7().to("patientLookupService");
```

メッセージヘッダー

unmarshal 操作は、MSH セグメントから以下のフィールドを Camel メッセージのヘッダーとして追

加します。

キー	MSH フィールド	例
CamelHL7SendingApplication	**MSH-3**	**MYSERVER**
CamelHL7SendingFacility	**MSH-4**	**MYSERVERAPP**
CamelHL7ReceivingApplication	**MSH-5**	**MYCLIENT**
CamelHL7ReceivingFacility	**MSH-6**	**MYCLIENTAPP**
CamelHL7Timestamp	**MSH-7**	**20071231235900**
CamelHL7Security	**MSH-8**	**null**
CamelHL7MessageType	**MSH-9-1**	**ADT**
CamelHL7TriggerEvent	**MSH-9-2**	**A01**
CamelHL7MessageControl	**MSH-10**	**1234**
CamelHL7ProcessingId	**MSH-11**	**P**
CamelHL7VersionId	**MSH-12**	**2.4**
CamelHL7Context	**-**	Camel 2.14: メッセージの解析に使用された **HapiContext** が含まれます。
CamelHL7Charset	**MSH-18**	Camel 2.14.1: Unicode UTF-8

***CamelHL7Context* 以外のヘッダーはすべて *String* 型です。ヘッダー値がない場合、その値は *null* になります。**

オプション

***HL7* データ形式は以下のオプションをサポートします。**

オプション	デフォルト	説明

validate	**true**	デフォルトの検証ルールを使用して HAPI Parser がメッセージを検証するかどうか。parser オプションまたは hapiContext オプションを使用して、目的の HAPI ValidationContext で初期化することが推奨されます。
parser	**ca.uhn.hl7v2.parser.GenericParser**	使用されるカスタムパーサー。タイプ ca.uhn.hl7v2.parser.Parser である必要があります。GenericParser では、XML でエンコードされた HL7v2 メッセージの解析も許可されることに注意してください。
hapiContext	**ca.uhn.hl7v2.DefaultHapiContext**	Camel 2.14: カスタムパーサー、カスタム ValidationContext などを定義できるカスタム HAPI コンテキスト。これにより、HL7 の解析およびレンダリングプロセスを完全に制御できます。

DEPENDENCIES

Camel ルートで HL7 を使用するには、上記の camel-hl7 の依存関係を追加して、このデータ形式を実装する必要があります。

HAPI ライブラリーは、HL7v2 メッセージバージョンごとに 1 つずつ、ベースライブラリー と複数の構造ライブラリーに分割されました。

- ***v2.1 構造ライブラリー***
- ***v2.2 構造ライブラリー***
- ***v2.3 構造ライブラリー***
- ***v2.3.1 構造ライブラリー***

- v2.4 構造ライブラリー
- v2.5 構造ライブラリー
- v2.5.1 構造ライブラリー
- v2.6 構造ライブラリー

デフォルトでは、camel-hl7 は HAPI ベースライブラリー のみを参照します。アプリケーションは、構造ライブラリー自体を含めます。たとえば、アプリケーションが HL7v2 メッセージバージョン 2.4 および 2.5 で機能する場合は、以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>ca.uhn.hapi</groupId>
    <artifactId>hapi-structures-v24</artifactId>
    <version>2.2</version>
    <!-- use the same version as your hapi-base version -->
</dependency>
<dependency>
    <groupId>ca.uhn.hapi</groupId>
    <artifactId>hapi-structures-v25</artifactId>
    <version>2.2</version>
    <!-- use the same version as your hapi-base version -->
</dependency>
```

または、ベースライブラリーを含む OSGi バンドル、すべての構造ライブラリーおよび必要な依存関係（バンドルクラスパス上）は、中央の Maven リポジトリー からダウンロードできます。

```
<dependency>
    <groupId>ca.uhn.hapi</groupId>
    <artifactId>hapi-osgi-base</artifactId>
    <version>2.2</version>
</dependency>
```

TERSER 言語

HAPI は、一般的に使用される場所の仕様構文を使用してフィールドへのアクセスを提供する Terser クラスを提供します。Terser 言語を使用すると、この構文を使用してメッセージから値を抽出し、フィルターリング、コンテンツベースのルーティングなどの式および述語として使用できます。

例：

```
import static org.apache.camel.component.hl7.HL7.terser;
...

  // extract patient ID from field QRD-8 in the QRY_A19 message above and put into message
header
  from("direct:test1")
    .setHeader("PATIENT_ID",terser("QRD-8(0)-1"))
    .to("mock:test1");
  // continue processing if extracted field equals a message header
  from("direct:test2")
    .filter(terser("QRD-8(0)-1").isEqualTo(header("PATIENT_ID"))
    .to("mock:test2");
```

HL7 検証述語

多くの場合、HL7v2 メッセージを解析し、別のステップで HAPI ValidationContext に対して検証することが推奨されます。

例：

```
import static org.apache.camel.component.hl7.HL7.messageConformsTo;
import ca.uhn.hl7v2.validation.impl.DefaultValidation;
...

  // Use standard or define your own validation rules
  ValidationContext defaultContext = new DefaultValidation();

  // Throws PredicateValidationException if message does not validate
  from("direct:test1").validate(messageConformsTo(defaultContext)).to("mock:test1");
```

HAPICONTEXT (CAMEL 2.14)を使用した HL7 検証述語

HAPI コンテキストは常に ValidationContext （または ValidationRuleBuilder）で設定されているため、検証ルールに間接的にアクセスできます。さらに、HL7DataFormat のマーシャリングを解除すると、CamelHL7Context header で設定された HAPI コンテキストが転送され、このコンテキストの検証ルールを簡単に再利用できます。

```
import static org.apache.camel.component.hl7.HL7.messageConformsTo;
import static org.apache.camel.component.hl7.HL7.messageConforms
...

  HapiContext hapiContext = new DefaultHapiContext();
  hapiContext.getParserConfiguration().setValidating(false); // don't validate during parsing
```

```
 // customize HapiContext some more ... e.g. enforce that PID-8 in ADT_A01 messages of
version 2.4 is not empty
 ValidationRuleBuilder builder = new ValidationRuleBuilder() {
   @Override
   protected void configure() {
     forVersion(Version.V24)
         .message("ADT", "A01")
         .terser("PID-8", not(empty()));
     }
   };
 hapiContext.setValidationRuleBuilder(builder);

 HL7DataFormat hl7 = new HL7DataFormat();
 hl7.setHapiContext(hapiContext);

 from("direct:test1")
   .unmarshal(hl7)              // uses the GenericParser returned from the HapiContext
   .validate(messageConforms())   // uses the validation rules returned from the HapiContext
                          // equivalent with .validate(messageConformsTo(hapiContext))
   // route continues from here
```

HL7 確認式

HL7v2 処理の一般的なタスクは、たとえば検証結果に基づいて、受信 HL7v2 メッセージへの応答として確認応答メッセージを生成することです。ack 式は、この処理を非常に重要に達成できます。

```
import static org.apache.camel.component.hl7.HL7.messageConformsTo;
import static org.apache.camel.component.hl7.HL7.ack;
import ca.uhn.hl7v2.validation.impl.DefaultValidation;
...

 // Use standard or define your own validation rules
  ValidationContext defaultContext = new DefaultValidation();

  from("direct:test1")
    .onException(Exception.class)
      .handled(true)
      .transform(ack()) // auto-generates negative ack because of exception in Exchange
      .end()
    .validate(messageConformsTo(defaultContext))
    // do something meaningful here
    ...
    // acknowledgement
    .transform(ack())
```

追加のサンプル

以下の例では、プレーンの String HL7 リクエストが応答を返す HL7 リスナーに送信されます。

■

```
String line1 =
"MSH|^~\\&|MYSENDER|MYRECEIVER|MYAPPLICATION||200612211200||QRY^A19|1234|P|2.4";

String line2 = "QRD|200612211200|R|I|GetPatient|||1^RD|0101701234|DEM||";

StringBuilder in = new StringBuilder();
in.append(line1);
in.append("\n");
in.append(line2);

String out = (String)template.requestBody("mina2:tcp://127.0.0.1:8888?
sync=true&codec=#hl7codec", in.toString());
```

次の例では、HL7 リスナーからの HL7 リクエストはビジネスロジックにルーティングされ、これはレジストリーに登録されているプレーン POJO として hl7service として実装されます。

```
public class MyHL7BusinessLogic {

    // This is a plain POJO that has NO imports whatsoever on Apache Camel.
    // its a plain POJO only importing the HAPI library so we can much easier work with the HL7
format.

    public Message handleA19(Message msg) throws Exception {
        // here you can have your business logic for A19 messages
        assertTrue(msg instanceof QRY_A19);
        // just return the same dummy response
        return createADR19Message();
    }

    public Message handleA01(Message msg) throws Exception {
        // here you can have your business logic for A01 messages
        assertTrue(msg instanceof ADT_A01);
        // just return the same dummy response
        return createADT01Message();
    }
}
```

その後、RouteBuilder を使用する Camel ルートは以下のようになります。

```
DataFormat hl7 = new HL7DataFormat();
// we setup or HL7 listener on port 8888 (using the hl7codec) and in sync mode so we can
return a response
from("mina2:tcp://127.0.0.1:8888?sync=true&codec=#hl7codec")
    // we use the HL7 data format to unmarshal from HL7 stream to the HAPI Message model
    // this ensures that the camel message has been enriched with hl7 specific headers to
    // make the routing much easier (see below)
    .unmarshal(hl7)
    // using choice as the content base router
    .choice()
        // where we choose that A19 queries invoke the handleA19 method on our hl7service
bean
```

```
        .when(header("CamelHL7TriggerEvent").isEqualTo("A19"))
            .beanRef("hl7service", "handleA19")
            .to("mock:a19")
        // and A01 should invoke the handleA01 method on our hl7service bean
        .when(header("CamelHL7TriggerEvent").isEqualTo("A01")).to("mock:a01")
            .beanRef("hl7service", "handleA01")
            .to("mock:a19")
        // other types should go to mock:unknown
        .otherwise()
            .to("mock:unknown")
    // end choice block
    .end()
    // marshal response back
    .marshal(hl7);
```

HL7 DataFormat を使用すると、Camel メッセージヘッダーに MSH セグメントからのフィールドが入力されることに注意してください。ヘッダーは、上記の例のように、フィルターリングまたはコンテンツベースのルーティングに特に便利です。

第70章 HTTP

HTTP コンポーネント

http: コンポーネントは HTTP ベースの エンドポイント を提供し、外部の HTTP リソース(HTTP を使用して外部サーバーを呼び出すクライアントとして)を消費します。

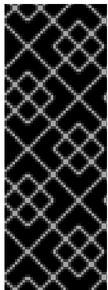
Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-http</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
http:hostname[:port][/resourceUri][?param1=value1][&param2=value2]
```

デフォルトでは、HTTP にポート 80 を使用し、HTTPS には 443 を使用します。



CAMEL-HTTP VS CAMEL-JETTY

生成できるのは、HTTP コンポーネントによって生成されたエンドポイントのみです。そのため、Camel ルートへの入力としては使用しないでください。HTTP エンドポイントを Camel ルートへの入力として HTTP サーバー経由でバインド/公開するには、Jetty コンポーネントまたは Servlet コンポーネント を使用できます。

例

POST を使用してボディーで url を呼び出して、応答を out メッセージとして返します。if body が GET を使用して null 呼び出し URL で、応答を out メッセージとして返します。

Java DSL	Spring DSL
`from("direct:start")` `  .to("http://myhost/mypath");`	`<from uri="direct:start"/>` `<to uri="http://oldhost"/>`

ヘッダーを追加することで、HTTP エンドポイント URI を上書きできます。Camel は http://newhost を呼び出します。これは、REST urls などに非常に便利です。

Java DSL

```
from("direct:start")
  .setHeader(Exchange.HTTP_URI, simple("http://myserver/orders/${header.orderId}"))
  .to("http://dummyhost");
```

URI パラメーターは、エンドポイント URI またはヘッダーとして直接設定できます。

Java DSL

```
from("direct:start")
  .to("http://oldhost?order=123&detail=short");
from("direct:start")
  .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
  .to("http://oldhost");
```

HTTP リクエストメソッドを POST に設定します。

Java DSL

```
from("direct:start")
  .setHeader(Exchange.HTTP_METHOD,
constant("POST"))
  .to("http://www.google.com");
```

Spring DSL

```
<from uri="direct:start"/>
<setHeader
headerName="CamelHttpMethod">
  <constant>POST</constant>
</setHeader>
<to uri="http://www.google.com"/>
<to uri="mock:results"/>
```

HTTPENDPOINT オプション

名前	デフォルト値	説明

throwExceptionOnFailure	**true**	リモートサーバーからの応答が失敗した場合に **HttpOperationFailedException** を出力することを無効にするオプション。これにより、HTTP ステータスコードに関係なくすべての応答を取得できます。
bridgeEndpoint	**false**	オプションが true の場合、HttpProducer は Exchange.HTTP_URI ヘッダーを無視し、リクエストにエンドポイントの URI を使用します。また、**throwExceptionOnFailure** を false に設定して、HttpProducer がすべての障害応答を返信するようにすることもできます。**Camel 2.3:** オプションが true の場合、content-encoding が gzip の場合、HttpProducer および CamelServlet は gzip 処理をスキップします。
disableStreamCache	**false**	DefaultHttpBinding は、要求入力ストリームをストリームキャッシュにコピーし、このオプションが 2 回読み取りをサポートする場合はメッセージボディーに配置します。それ以外の場合は、DefaultHttpBinding は要求入力ストリームをメッセージボディーに設定します。**Camel 2.17:** このオプションは、デフォルトでストリームキャッシングではなく、応答ストリームを直接使用できるようにプロデューサーによってもサポートされるようになりました。
httpBinding	**null**	レジストリーの **org.apache.camel.component.http.HttpBinding** への参照。
httpClientConfigurer	**null**	レジストリーの **org.apache.camel.component.http.HttpClientConfigurer** への参照。
httpClient.XXX	**null**	HttpClientParams でオプションの設定たとえば、**httpClient.soTimeout=5000** は **SO_TIMEOUT** を 5 秒に設定します。

clientConnectionManager	**null**	カスタムの **org.apache.http.conn.ClientConnectionManager** を使用するには、以下を行います。
transferException	**false**	Camel 2.6: 有効で、エクスチェンジ がコンシューマー側で処理に失敗した場合、発生した **例外** が **application/x-java-serialized-object** コンテンツタイプとして応答でシリアライズされた場合は (Jetty または Servlet Camel コンポーネントを使用)。プロデューサー側では、例外がデシリアライズされ、**HttpOperationFailedException** ではなくそのまま出力されます。原因となった例外はシリアライズする必要があります。
headerFilterStrategy	**null**	Camel 2.11: レジストリーの **org.apache.camel.spi.HeaderFilterStrategy** のインスタンスへの参照。これは、新しい create HttpEndpoint にカスタム headerFilterStrategy を適用するために使用されます。
eagerCheckContentAvailable	**false**	Camel 2.15.3/2.16: コンシューマーのみ。Content-Length ヘッダーが 0 の場合、HTTP リクエストにコンテンツがあるかどうかを即時にチェックするかどうか。HTTP クライアントがストリームデータを送信しない場合に、この機能を有効にできます。
copyHeaders	**true**	Camel 2.16: **true** の場合、コピーストラテジーに従って OUT エクスチェンジヘッダーにコピーされます。**false** の場合、HTTP 応答からのヘッダーのみが含まれます (IN ヘッダーはコピーされません)。
okStatusCodeRange	**200-299**	Camel 2.16: 正常な応答と見なされるステータスコード。値は含まれます。範囲は、構文 ***from-to*** を使用して定義する必要があります。

ignoreResponseBody	**false**	Camel 2.16: **true** の場合、http プロデューサーは応答ボディーを読み取りせず、入力ストリームをキャッシュします。

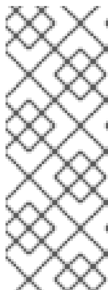
認証およびプロキシー

以下の認証オプションを HttpEndpoint に設定できます。

名前	デフォルト値	説明
authMethod	**null**	**Basic**、**Digest**、または **NTLM** のいずれかの認証方法。
authMethodPriority	**null**	認証方法の優先度。はコンマで区切られたリストです。例： **NTLM** を除外する **Basic,Digest**。
authUsername	**null**	認証用のユーザー名
authPassword	**null**	認証のパスワード
authDomain	**null**	NTML 認証のドメイン
authHost	**null**	NTML 認証のオプションのホスト
proxyHost	**null**	プロキシーのホスト名
proxyPort	**null**	プロキシーポート番号
proxyAuthMethod	**null**	プロキシーの認証方法(**Basic** **Digest** または **NTLM** のいずれか)。
proxyAuthUsername	**null**	プロキシー認証用のユーザー名
proxyAuthPassword	**null**	プロキシー認証のパスワード
proxyAuthDomain	**null**	プロキシー NTML 認証のドメイン
proxyAuthHost	**null**	プロキシー NTML 認証のオプションホスト

認証を使用する場合は、authMethod オプションまたは authProxyMethod オプションのメソッドを選択する 必要 があります。プロキシーおよび認証の詳細は、HttpComponent または HttpEndoint のいずれかに設定できます。HttpEndpoint で指定される値は、HttpComponent よりも優先されます。これを 1 回実行できる HttpComponent で設定する最も適しています。

HTTP コンポーネントは設定よりも規則を使用することを意味します。つまり、authMethodPriority を明示的に設定していない場合は、select (ed) authMethod を priority としても使用します。そのため、authMethod.Basic を使用する場合、auhtMethodPriority は Basic のみになります。



注記

Camel HTTP コンポーネントは HttpClient v3.x をベースとしているため、NTLM プロトコルの初期バージョンである NTLMv1 と呼ばれる サポートのみに限定 されます。NTLMv2 をサポートしていません。Camel HTTP4 コンポーネントには、NTLMv2 のサポートがあります。

HTTPCOMPONENT オプション

名前	デフォルト値	説明
httpBinding	**null**	カスタムの **org.apache.camel.component.http.HttpBinding** を使用するには、以下を行います。
httpClientConfigurer	**null**	カスタムの **org.apache.camel.component.http.HttpClientConfigurer** を使用するには、以下を行います。
httpConnectionManager	**null**	カスタムの **org.apache.commons.httpclient.HttpConnectionManager** を使用するには、以下を行います。
httpConfiguration	**null**	カスタムの **org.apache.camel.component.http.HttpConfiguration** を使用するには、以下を行います。

allowJavaSerializedObject	**false**	Camel 2.16.1/2.15.5: **リクエストが context-type=application/x-java-serialized-object を使用している場合に Java のシリアル化を許可するかどうか。** このオプションはデフォルトでオフになっています。**警告：** このオプションを有効にすると、Java はリクエストから Java に受信データをデシリアライズし、セキュリティーリスクとなる可能性があることに注意してください。

メッセージヘッダー

名前	タイプ	説明
Exchange.HTTP_URI	**文字列**	呼び出す URI。エンドポイントに設定された既存の URI を直接上書きします。この URI は、呼び出す HTTP サーバーの URI です。セキュリティーなどのエンドポイントオプションを設定できる Camel エンドポイント URI とは異なります。このヘッダーは、HTTP サーバーの UTI のみをサポートしません。
Exchange.HTTP_METHOD	**文字列**	使用する HTTP メソッド/Verb (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE)
Exchange.HTTP_PATH	**文字列**	リクエスト URI のパス。ヘッダーは HTTP_URI でリクエスト URI を構築するために使用されます。**Camel 2.3.0:** パスが "/ で始まる場合、http プロデューサーは Exchange.HTTP_BASE_URI ヘッダーまたは exchange.getFromEndpoint（）.getEndpointUri（）; に基づいて相対パスの検索を試みます。
Exchange.HTTP_QUERY	**文字列**	URI パラメーター。エンドポイントで直接設定された既存の URI パラメーターを上書きします。

Exchange.HTTP_RESPONSE_CODE	**int**	外部サーバーからの HTTP 応答コード。OK の場合は 200 です。
Exchange.HTTP_CHARACTER_ENCODING	**文字列**	文字エンコーディング。
Exchange.CONTENT_TYPE	**文字列**	HTTP コンテンツタイプ。は IN メッセージと OUT メッセージの両方で設定され、**text/html** などのコンテンツタイプを提供します。
Exchange.CONTENT_ENCODING	**文字列**	HTTP コンテンツエンコーディング。は IN メッセージと OUT メッセージの両方で設定され、**gzip** などのコンテンツエンコーディングを提供します。
Exchange.HTTP_SERVLET_REQUEST	**HttpServletRequest**	**HttpServletRequest** オブジェクト。
Exchange.HTTP_SERVLET_RESPONSE	**HttpServletResponse**	**HttpServletResponse** オブジェクト。
Exchange.HTTP_PROTOCOL_VERSION	**文字列**	**Camel 2.5:** このヘッダーで http プロトコルバージョンを設定できます（例：）。"HTTP/1.0"。ヘッダーを指定しないと、HttpProducer はデフォルト値 HTTP/1.1 を使用します。

上記のヘッダー名は定数です。Spring DSL では、名前の代わりに定数の値を使用する必要があります。

メッセージボディー

Camel は外部サーバーからの HTTP 応答を OUT ボディーに保存します。IN メッセージからのヘッダーはすべて OUT メッセージにコピーされ、ルーティング中にヘッダーが保持されます。さらに、Camel は HTTP 応答ヘッダーと OUT メッセージヘッダーを追加します。

レスポンスコード

Camel は HTTP 応答コードに従って処理されます。

- レスポンスコードは 100..299 の範囲にあり、Camel は応答の成功と見なします。

- 応答コードは 300..399 の範囲にあり、Camel はこれをリダイレクト応答とみなし、その情報とともに HttpOperationFailedException を出力します。

- 応答コードは 400+ で、Camel はこれを外部サーバーの障害と見なし、この情報とともに HttpOperationFailedException を出力します。

THROWEXCEPTIONONFAILURE

オプション throwExceptionOnFailure を false に設定すると、失敗したレスポンスコードに対して HttpOperationFailedException が出力されないようにすることができます。これにより、リモートサーバーから応答を取得できるようになります。デモには、以下の例があります。

HTTPOPERATIONFAILEDEXCEPTION

この例外には、以下の情報が含まれます。

- HTTP ステータスコード

- HTTP ステータス行 (ステータスコードのテキスト)

- サーバーがリダイレクトを返した場合は、場所をリダイレクトします

- 応答ボディー(java.lang.String)（サーバーがボディーを応答として提供）

GET または POST を使用した呼び出し

以下のアルゴリズムは、GET または POST HTTP メソッドのいずれかを使用する必要があるかどうかを判断するために使用されます：1。ヘッダーで提供されるメソッドを使用します。2.クエリー文字列がヘッダーで提供される場合は GET。3.エンドポイントがクエリー文字列で設定されている場合の GET。4.送信するデータがある場合は POST します(null ではありません)。5.それ以外の場合は GET。

HTTPSERVLETREQUEST および HTTPSERVLETRESPONSE にアクセスする方法

これらの 2 つにアクセスするには、を使用すると Camel 型コンバーターシステムを使用できます。

```
HttpServletRequest request = exchange.getIn().getBody(HttpServletRequest.class);
HttpServletRequest response = exchange.getIn().getBody(HttpServletResponse.class);
```

クライアントタイムアウトの使用 - SO_TIMEOUT

このリンクのユニットテストを参照してください。

プロキシーの設定

Java DSL

```
from("direct:start")
  .to("http://oldhost?proxyHost=www.myproxy.com&proxyPort=80");
```

proxyUsername および proxyPassword オプションを使用したプロキシー認証にも対応しています。

URI の外部でプロキシー設定の使用

Java DSL	Spring DSL

`context.getProperties().put("http.proxyHost", "172.168.18.9");`
`context.getProperties().put("http.proxyPort" "8080");`
	`<camelContext>`
	`<properties>`
	`<property key="http.proxyHost" value="172.168.18.9"/>`
	`<property key="http.proxyPort" value="8080"/>`
	`</properties>`
	`</camelContext>`

Endpoint のオプションは、コンテキストのオプションを上書きします。

CHARSET の設定

POST を使用してデータを送信する場合は、charsetを設定できます。

```
setProperty(Exchange.CHARSET_NAME, "iso-8859-1");
```

スケジュールされたポーリングを使用したサンプル

この例は、Google ホームページを 10 秒ごとにポーリングし、そのページを file. html ファイルに書き込みます。

```
from("timer://foo?fixedRate=true&delay=0&period=10000")
    .to("http://www.google.com")
    .setHeader(FileComponent.HEADER_FILE_NAME, "message.html").to("file:target/google");
```

レスポンスコードの取得

HTTP コンポーネントから HTTP 応答コードを取得するには、Out メッセージヘッダーと Exchange.HTTP_RESPONSE_CODE の値を取得します。

```
   Exchange exchange = template.send("http://www.google.com/search", new Processor() {
            public void process(Exchange exchange) throws Exception {
                exchange.getIn().setHeader(Exchange.HTTP_QUERY,
constant("hl=en&q=activemq"));
            }
   });
   Message out = exchange.getOut();
   int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
```

THROWEXCEPTIONONFAILURE=FALSE を使用して応答を返す

以下のルートでは、リモート HTTP 呼び出しから返されたデータで 補完 するメッセージをルーティングします。リモートサーバーからの応答が必要な場合、throwExceptionOnFailure オプションを false に設定して、AggregationStrategy で応答を取得します。コードは HTTP ステータスコード 404 をシミュレートするユニットテストをベースにしているため、アサーションコードなどがいくつかあります。

```
// We set throwExceptionOnFailure to false to let Camel return any response from the remove
HTTP server without thrown
// HttpOperationFailedException in case of failures.
// This allows us to handle all responses in the aggregation strategy where we can check the
HTTP response code
// and decide what to do. As this is based on an unit test we assert the code is 404
from("direct:start").enrich("http://localhost:{{port}}/myserver?
throwExceptionOnFailure=false&user=Camel", new AggregationStrategy() {
    public Exchange aggregate(Exchange original, Exchange resource) {
```

```
        // get the response code
        Integer code = resource.getIn().getHeader(Exchange.HTTP_RESPONSE_CODE,
Integer.class);
        assertEquals(404, code.intValue());
        return resource;
    }
}).to("mock:result");

// this is our jetty server where we simulate the 404
from("jetty://http://localhost:{{port}}/myserver")
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                exchange.getOut().setBody("Page not found");
                exchange.getOut().setHeader(Exchange.HTTP_RESPONSE_CODE, 404);
            }
        });
```

COOKIE の無効化

Cookie を無効にするには、URI オプション httpClient.cookiePolicy=ignoreCookiesを追加して HTTP Client が Cookie を無視するように設定します。

高度な使用方法

HTTP プロデューサーをより詳細に制御する必要がある場合は、HttpComponent を使用する必要があります。ここで、さまざまなクラスを設定してカスタム動作を提供できます。

MAXCONNECTIONSPERHOST の設定

HTTP コンポーネントには org.apache.commons.httpclient.HttpConnectionManager があります。ここでは、指定のコンポーネントの各種のグローバル設定を設定できます。グローバルでは、コンポーネントによって作成されるエンドポイントに同じ HttpConnectionManager があることを意味します。したがって、ホストごとに max 接続に異なる値を設定する場合は、通常使用するエンドポイント URI では なく、HTTP コンポーネントで定義する必要があります。そのため、以下のようになります。

まず、Spring XML で http コンポーネントを定義します。はい、同じスキーム名 http を使用します。そうしないと、Camel はデフォルト設定でコンポーネントを自動検出して作成します。必要なものは、このオプションを設定できるように、これをオーバールールすることです。以下の例では、max 接続をデフォルトの 2 ではなく 5 に設定します。

```
<bean id="http" class="org.apache.camel.component.http.HttpComponent">
   <property name="camelContext" ref="camel"/>
   <property name="httpConnectionManager" ref="myHttpConnectionManager"/>
</bean>
```

```
<bean id="myHttpConnectionManager"
class="org.apache.commons.httpclient.MultiThreadedHttpConnectionManager">
   <property name="params" ref="myHttpConnectionManagerParams"/>
</bean>

<bean id="myHttpConnectionManagerParams"
class="org.apache.commons.httpclient.params.HttpConnectionManagerParams">
   <property name="defaultMaxConnectionsPerHost" value="5"/>
</bean>
```

次に、ルート内で通常通り使用できます。

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring" trace="true">
   <route>
      <from uri="direct:start"/>
      <to uri="http://www.google.com"/>
      <to uri="mock:result"/>
   </route>
</camelContext>
```

プリエンプション認証の使用

エンドユーザーは、HTTPS での認証に問題があることを報告していました。この問題は最終的に、HTTPS サーバーが HTTP コード 401 Authorization Required を返しなかったときに解決されました。解決策は、httpClient.authenticationPreemptive=trueの URI オプションを設定することでした。

リモートサーバーからの自己署名証明書の許可

Apache Commons HTTP API でこれを行う方法については、いくつかのコードに関するメーリングリストの リンク を参照してください。

JSSE 設定ユーティリティーの使用

Camel 2.8 以降、HTTP4 コンポーネントは JSSE ユーティリティー を介して SSL/TLS 設定をサポートします。このユーティリティーは、作成する必要のあるコンポーネント固有のコードの量を大幅に減らし、エンドポイントおよびコンポーネントレベルで設定可能です。 以下の例は、HTTP4 コンポーネントで ユーティリティーを使用する方法を示しています。Security Guide の Configuring Transport Security for Camel Components の章を参照してください。

このコンポーネントで使用される Apache HTTP クライアントのバージョンは、グローバルプロトコルレジストリーから SSL/TLS 情報を解決します。 このコンポーネントは、Camel JSSE Configuration ユーティリティーの使用をサポートするために、HTTP クライアントのプロトコルソケットファクトリーの実装

org.apache.camel.component.http.SSLContextParametersSecureProtocolSocketFactory を提供します。 以下の例は、プロトコルレジストリーを設定し、登録されたプロトコル情報をルートで使用する方法を示しています。

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

ProtocolSocketFactory factory =
   new SSLContextParametersSecureProtocolSocketFactory(scp);

Protocol.registerProtocol("https",
       new Protocol(
       "https",
       factory,
       443));

from("direct:start")
       .to("https://mail.google.com/mail/").to("mock:results");
```

APACHE HTTP クライアントを直接設定

基本的に、HTTP コンポーネントは Apache HTTP クライアントの上に構築され、カスタム org.apache.camel.component.http.HttpClientConfigurer を実装し、完全な制御が必要な場合に http クライアントで一部の設定を行うことができます。

ただし、キーストアとトラストストアを指定する だけ の場合は、Apache HTTP HttpClientConfigurer でこれを行うことができます。以下に例を示します。

```
Protocol authhttps = new Protocol("https", new AuthSSLProtocolSocketFactory(
  new URL("file:my.keystore"), "mypassword",
  new URL("file:my.truststore"), "mypassword"), 443);

Protocol.registerProtocol("https", authhttps);
```

次に、HttpClientConfigurer を実装するクラスを作成し、上記の例ごとにキーストアまたはトラストストアを提供する https プロトコルを登録する必要があります。次に、Camel ルートビルダークラスから、以下のようにフックできます。

```
HttpComponent httpComponent = getContext().getComponent("http", HttpComponent.class);
httpComponent.setHttpClientConfigurer(new MyHttpClientConfigurer());
```

Spring DSL を使用してこれを実行する場合は、URI を使用して HttpClientConfigurer を指定できます。以下に例を示します。

```
<bean id="myHttpClientConfigurer"
 class="my.https.HttpClientConfigurer">
</bean>

<to uri="https://myhostname.com:443/myURL?
httpClientConfigurerRef=myHttpClientConfigurer"/>
```

上記のように HttpClientConfigurer を実装し、キーストアとトラストストアを設定すると問題なく機能します。

- Jetty

第71章 HTTP4

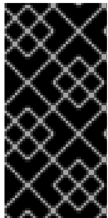
HTTP4 コンポーネント

Camel 2.3 の時点で利用可能

http4: コンポーネントは、(HTTP を使用して外部サーバーを呼び出すクライアントとして)外部 HTTP リソースを呼び出す HTTP ベースの エンドポイント を提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-http4</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```



CAMEL-HTTP4 VS CAMEL-HTTP

camel-http4 は Apache HttpClient 4.x を使用し、camel-http は Apache HttpClient 3.x を使用します。

CAMEL ON EAP デプロイメント

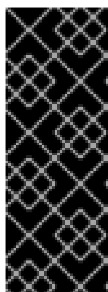
このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
http4:hostname[:port][/resourceUri][?options]
```

デフォルトでは、HTTP にポート 80 を使用し、HTTPS には 443 を使用します。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。



CAMEL-HTTP4 VS CAMEL-JETTY

生成できるのは、HTTP4 コンポーネントによって生成されたエンドポイントのみです。そのため、Camel ルートへの入力としては使用しないでください。HTTP エンドポイントを Camel ルートへの入力として、HTTP サーバー経由でバインド/公開するには、Jetty コンポーネント を使用してください。

HTTPCOMPONENT オプション

名前	デフォルト値	説明
maxTotalConnections	**200**	接続の最大数。
connectionsPerRoute	**20**	ルートごとの最大接続数。
cookieStore	**null**	Camel 2.11.2/2.12.0: カスタムの **org.apache.http.client.CookieStore** を使用するには、以下を行います。デフォルトでは、**org.apache.http.impl.client.BasicCookieStore** が使用されます。これはインメモリーのみの Cookie ストアです。**bridgeEndpoint=true** の場合、クッキーストアは単にブリッジ（プロキシーとして機能するなど）であるため、Cookie ストアを noop クッキーストアとして強制的に保存しないでください。
httpClientConfigurer	**null**	レジストリーの **org.apache.camel.component.http.HttpClientConfigurer** への参照。
clientConnectionManager	**null**	カスタムの **org.apache.http.conn.ClientConnectionManager** を使用するには、以下を行います。
httpBinding	**null**	カスタムの **org.apache.camel.component.http.HttpBinding** を使用するには、以下を行います。

httpContext	**null**	Camel 2.9.2: リクエストの実行時にカスタムの **org.apache.http.protocol.HttpContext** を使用します。
sslContextParameters	**null**	Camel 2.8: カスタムの **org.apache.camel.util.jsse.SSLContextParameters** を使用するには、以下を行います。Security Guide の Configuring Transport Security for Camel Components の章を参照してください。「JSSE 設定ユーティリティーの使用」
x509HostnameVerifier	**BrowserCompatHostnameVerifier**	Camel 2.7: **org.apache.http.conn.ssl.X509HostnameVerifier** インスタンスを別の **org.apache.http.conn.ssl.StrictHostnameVerifier** または **org.apache.http.conn.ssl.AllowAllHostnameVerifier** 等のレジストリーで参照できます。
connectionTimeToLive	**-1**	Camel 2.11.0: 接続の有効期間。時間単位はミリ秒で、デフォルト値は常に存続します。
allowJavaSerializedObject	**false**	Camel 2.16.1/2.15.5: **リクエストが context-type=application/x-java-serialized-object を使用している場合に Java のシリアル化を許可するかどうか。**このオプションはデフォルトでオフになっています。**警告：**このオプションを有効にすると、Java はリクエストから Java に受信データをデシリアライズし、セキュリティーリスクとなる可能性があることに注意してください。

HTTPENDPOINT オプション

名前	デフォルト値	説明

throwExceptionOnFailure	**true**	リモートサーバーからの応答が失敗した場合に **HttpOperationFailedException** を出力することを無効にするオプション。これにより、HTTP ステータスコードに関係なくすべての応答を取得できます。
bridgeEndpoint	**false**	true の場合、HttpProducer は Exchange.HTTP_URI ヘッダーを無視し、リクエストにエンドポイントの URI を使用します。また、**throwExcpetionOnFailure** を false に設定して、HttpProducer がすべての障害応答を返信するようにすることもできます。また、true HttpProducer に設定すると、content-encoding が gzip の場合、CamelServlet は gzip 処理をスキップします。
clearExpiredCookies	**true**	**Camel 2.11.2/2.12.0:** HTTP リクエストを送信する前に期限切れのクッキーを消去するかどうか。これにより、クッキーストアは、有効期限が切れたときに削除される新しいクッキーを追加して拡張し続けません。
cookieStore	**null**	**Camel 2.11.2/2.12.0:** カスタムの **org.apache.http.client.CookieStore** を使用するには、以下を行います。デフォルトでは、**org.apache.http.impl.client.BasicCookieStore** が使用されます。これはインメモリーのみの Cookie ストアです。**bridgeEndpoint=true** の場合、クッキーストアは単にブリッジ（プロキシーとして機能するなど）であるため、Cookie ストアを noop クッキーストアとして強制的に保存しないでください。

disableStreamCache	**false**	DefaultHttpBinding は、要求入力ストリームをストリームキャッシュにコピーし、このオプションが複数の読み取りをサポートする場合はメッセージボディーに配置します。それ以外の場合は、DefaultHttpBinding は要求入力ストリームをメッセージボディーに直接設定します。**Camel 2.17:** このオプションは、デフォルトでストリームキャッシングの代わりに応答ストリームを直接使用できるようにするため、プロデューサーによってもサポートされるようになりました。
headerFilterStrategy	**null**	**Camel 2.10.4:** レジストリーの **org.apache.camel.spi.HeaderFilterStrategy** のインスタンスへの参照。t は、新しい create HttpEndpoint にカスタム headerFilterStrategy を適用するために使用されます。
httpBindingRef	**null**	**非推奨および は Camel 3.0: Registry の org.apache.camel.component.http.HttpBinding への参照で削除され** ます。代わりに **httpBinding** オプションを使用してください。
httpBinding	**null**	カスタムの **org.apache.camel.component.http.HttpBinding** を使用するには、以下を行います。
httpClientConfigurer	**null**	レジストリーの **org.apache.camel.component.http.HttpClientConfigurer** への参照。
httpContext	**null**	**Camel 2.9.2:** リクエストの実行時にカスタムの **org.apache.http.protocol.HttpContext** を使用します。

httpClient.XXX	**null**	BasicHttpParams でオプションの設定たとえば、**httpClient.soTimeout=5000** は **SO_TIMEOUT** を 5 秒に設定します。以下のパラメーター bean のセッターメソッドを確認します。AuthParamBean、ClientParamBean、ConnConnectionParamBean、ConnRouteParamBean、CookieSpecParamBean、HttpConnectionParamBean、HttpConnectionParamBean、および HttpProtocolParamBean Camel 2.13.0: **httpClient は HttpClientBuilder および RequestConfig.Builder の設定に変更されているため**、完全な参考として API ドキュメントを確認してください。たとえば、**httpClient.socketTimeout=5000** を使用して、ソケットのタイムアウトを 5 秒に設定します。
clientConnectionManager	**null**	カスタムの **org.apache.http.conn.ClientConnectionManager** を使用するには、以下を行います。
transferException	**false**	有効化され、エクスチェンジ がコンシューマー側で処理に失敗した場合、発生した **例外** が **application/x-java-serialized-object** コンテンツタイプとして応答でシリアライズされた場合は (Jetty または SERVLET Camel コンポーネントを使用)。プロデューサー側では、例外がデシリアライズされ、**HttpOperationFailedException** ではなくそのまま出力されます。原因となった例外はシリアライズする必要があります。

sslContextParameters	**null**	Camel 2.11.1: レジストリーの **org.apache.camel.util.jsse.SSLContextParameters** への参照。**重要：** HttpComponent ごとに、**org.apache.camel.util.jsse.SSLContextParameters** の 1 つのインスタンスのみがサポートされます。2 つ以上の異なるインスタンスを使用する必要がある場合は、インスタンスごとに新しい HttpComponent を定義する必要があります。詳細については以下をご覧くださいSecurity Guide および 「JSSE 設定ユーティリティーの使用」 の Configuring Transport Security for Camel Components も参照してください。
x509HostnameVerifier	**BrowserCompatHostnameVerifier**	Camel 2.7: **org.apache.http.conn.ssl.X509HostnameVerifier** インスタンスを別の **org.apache.http.conn.ssl.StrictHostnameVerifier** または **org.apache.http.conn.ssl.AllowAllHostnameVerifier** 等のレジストリーで参照できます。
maxTotalConnections	**null**	Camel 2.14: コネクションマネージャーが持つ合計接続数。このオプションが設定されていない場合、Camel は代わりにコンポーネントの 設定を使用します。
connectionsPerRoute	**null**	Camel 2.14: ルートごとの最大接続数。このオプションが設定されていない場合、Camel は代わりにコンポーネントの 設定を使用します。
authenticationPreemptive	**false**	Camel 2.11.3/2.12.2: このオプションが true の場合、camel-http4 はプリエンプティブな Basic 認証をサーバーに送信します。

eagerCheckContentAvailable	**false**	Camel 2.15.3/2.16: コンシューマーのみ。Content-Length ヘッダーが 0 の場合、HTTP リクエストにコンテンツがあるかどうかを即時にチェックするかどうか。HTTP クライアントがストリームデータを送信しない場合に、この機能を有効にできます。
copyHeaders	**true**	Camel 2.16: **true** の場合、コピーストラテジーに従って OUT エクスチェンジヘッダーにコピーされます。**false** の場合、HTTP 応答からのヘッダーのみが含まれます (IN ヘッダーはコピーされません)。
okStatusCodeRange	**200-299**	Camel 2.16: 正常な応答と見なされるステータスコード。値は含まれます。範囲は、構文 ***from-to*** を使用して定義する必要があります。
ignoreResponseBody	**false**	Camel 2.16: **true** の場合、http プロデューサーは応答ボディーを読み取りせず、入力ストリームをキャッシュします。

以下の認証オプションを HttpEndpoint に設定できます。

基本認証およびプロキシーの設定

Before Camel 2.8.0

名前	デフォルト値	説明
username	**null**	認証用のユーザー名。
password	**null**	認証のパスワード。
domain	**null**	認証用のドメイン名。
host	**null**	ホスト名の認証。
proxyHost	**null**	プロキシーのホスト名

proxyPort	null	プロキシーポート番号
proxyUsername	null	プロキシー認証用のユーザー名
proxyPassword	null	プロキシー認証のパスワード
proxyDomain	null	プロキシードメイン名
proxyNtHost	null	プロキシー Nt ホスト名
名前	**デフォルト値**	**説明**
authUsername	null	認証用のユーザー名
authPassword	null	認証のパスワード
authDomain	null	認証のドメイン名
authHost	null	ホスト名の認証
proxyAuthHost	null	プロキシーのホスト名
proxyAuthPort	null	プロキシーポート番号
proxyAuthScheme	null	プロキシースキーム は、設定されていない場合は、エンドポイントからスキームをフォールバックし、使用します。
proxyAuthUsername	null	プロキシー認証用のユーザー名
proxyAuthPassword	null	プロキシー認証のパスワード
proxyAuthDomain	null	プロキシードメイン名
proxyAuthNtHost	null	プロキシー Nt ホスト名

メッセージヘッダー

名前	タイプ	説明

Exchange.HTTP_URI	文字列	呼び出す URI。エンドポイントに設定された既存の URI を直接上書きします。この URI は、呼び出す HTTP サーバーの URI です。セキュリティーなどのエンドポイントオプションを設定できる Camel エンドポイント URI とは異なります。このヘッダーは、HTTP サーバーの UTI のみをサポートしません。
Exchange.HTTP_PATH	文字列	リクエスト URI のパス。ヘッダーは HTTP_URI でリクエスト URI を構築するために使用されます。
Exchange.HTTP_QUERY	文字列	URI パラメーター。エンドポイントで直接設定された既存の URI パラメーターを上書きします。
Exchange.HTTP_RESPONSE_CODE	int	外部サーバーからの HTTP 応答コード。OK の場合は 200 です。
Exchange.HTTP_RESPONSE_TEXT	文字列	外部サーバーからの HTTP 応答テキスト。
Exchange.HTTP_CHARACTER_ENCODING	文字列	文字エンコーディング。
Exchange.CONTENT_TYPE	文字列	HTTP コンテンツタイプ。は IN メッセージと OUT メッセージの両方で設定され、**text/html** などのコンテンツタイプを提供します。
Exchange.CONTENT_ENCODING	文字列	HTTP コンテンツエンコーディング。は IN メッセージと OUT メッセージの両方で設定され、**gzip** などのコンテンツエンコーディングを提供します。

メッセージボディー

Camel は外部サーバーからの HTTP 応答を OUT ボディーに保存します。IN メッセージからのヘッダーはすべて OUT メッセージにコピーされ、ルーティング中にヘッダーが保持されます。さらに、Camel は HTTP 応答ヘッダーと OUT メッセージヘッダーを追加します。

レスポンスコード

Camel は HTTP 応答コードに従って処理されます。

- *レスポンスコードは 100..299 の範囲にあり、Camel は応答の成功と見なします。*
- *応答コードは 300..399 の範囲にあり、Camel はこれをリダイレクト応答とみなし、その情報とともに HttpOperationFailedException を出力します。*
- *応答コードは 400+ で、Camel はこれを外部サーバーの障害と見なし、この情報とともに HttpOperationFailedException を出力します。*

THROWEXCEPTIONONFAILURE

オプション throwExceptionOnFailure を false に設定すると、失敗したレスポンスコードに対して HttpOperationFailedException が出力されないようにすることができます。これにより、リモートサーバーから応答を取得できるようになります。デモには、以下の例があります。

HTTPOPERATIONFAILEDEXCEPTION

この例外には、以下の情報が含まれます。

- *HTTP ステータスコード*
- *HTTP ステータス行 (ステータスコードのテキスト)*
- *サーバーがリダイレクトを返した場合は、場所をリダイレクトします*
- *応答ボディー(java.lang.String)（サーバーがボディーを応答として提供）*

GET または POST を使用した呼び出し

以下のアルゴリズムは、GET または POST HTTP メソッドを使用する必要があるかどうかを判断するために使用されます：1。ヘッダーで提供されるメソッドを使用します。2.クエリー文字列がヘッ

ダーで提供される場合は GET。3.エンドポイントがクエリー文字列で設定されている場合の GET。4. 送信するデータがある場合は POST します(null ではありません)。5.それ以外の場合は GET。

HTTPSERVLETREQUEST および HTTPSERVLETRESPONSE にアクセスする方法

注記 を使用すると、Camel 型コンバーターシステムを使用してこれら 2 つにアクセスできます。camel-jetty または camel-cxf エンドポイントの後には、プロセッサーからリクエストおよび応答を取得できます。

```
HttpServletRequest request = exchange.getIn().getBody(HttpServletRequest.class);
HttpServletRequest response = exchange.getIn().getBody(HttpServletResponse.class);
```

呼び出す URI の設定

HTTP プロデューサーの URI を直接エンドポイント URI として設定できます。以下のルートでは、Camel は HTTP を使用して外部サーバー oldhost に呼び出します。

```
from("direct:start")
    .to("http4://oldhost");
```

同等の Spring の例：

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="http4://oldhost"/>
  </route>
</camelContext>
```

メッセージのキー Exchange.HTTP_URI でヘッダーを追加することで、HTTP エンドポイント URI を上書きできます。

```
from("direct:start")
  .setHeader(Exchange.HTTP_URI, constant("http://newhost"))
  .to("http4://oldhost");
```

上記のサンプルでは、エンドポイントが http4://oldhost で設定されていても、http://newhost を呼び出します。http4 エンドポイントがブリッジモードで動作している場合、Exchange.HTTP_URI のメッセージヘッダーを無視します。

URI パラメーターの設定

http プロデューサーは、HTTP サーバーに送信される URI パラメーターをサポートします。URI パラメーターは、エンドポイント URI で直接設定することも、メッセージ上でキー Exchange.HTTP_QUERY を持つヘッダーとして設定できます。

```
from("direct:start")
 .to("http4://oldhost?order=123&detail=short");
```

または、ヘッダーで提供されるオプション：

```
from("direct:start")
  .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
  .to("http4://oldhost");
```

HTTP メソッド(GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE)を HTTP プロデューサーに設定する方法

HTTP4 コンポーネントは、メッセージヘッダーを設定して HTTP リクエストメソッドを設定する方法を提供します。以下に例を示します。

```
from("direct:start")
  .setHeader(Exchange.HTTP_METHOD,
constant(org.apache.camel.component.http4.HttpMethods.POST))
  .to("http4://www.google.com")
  .to("mock:results");
```

メソッドは、文字列定数を使用して少し短くすることができます。

```
.setHeader("CamelHttpMethod", constant("POST"))
```

同等の Spring の例：

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <setHeader headerName="CamelHttpMethod">
        <constant>POST</constant>
    </setHeader>
    <to uri="http4://www.google.com"/>
    <to uri="mock:results"/>
  </route>
</camelContext>
```

クライアントタイムアウトの使用 - SO_TIMEOUT

HttpSOTimeoutTest ユニットテストを参照してください。

プロキシーの設定

HTTP4 コンポーネントは、プロキシーを設定する方法を提供します。

```
from("direct:start")
  .to("http4://oldhost?proxyAuthHost=www.myproxy.com&proxyAuthPort=80");
```

proxyAuthUsername および proxyAuthPassword オプションを使用したプロキシー認証にも対応しています。

URI の外部でプロキシー設定の使用

システムプロパティーの競合を回避するには、CamelContext または URI からのみプロキシー設定を設定できます。Java DSL:

```
context.getProperties().put("http.proxyHost", "172.168.18.9");
context.getProperties().put("http.proxyPort" "8080");
```

Spring XML

```
<camelContext>
    <properties>
        <property key="http.proxyHost" value="172.168.18.9"/>
        <property key="http.proxyPort" value="8080"/>
    </properties>
</camelContext>
```

Camel は最初に Java System または CamelContext プロパティーから設定を設定し、指定した場合はエンドポイントプロキシーオプションを設定します。そのため、エンドポイントオプションでシステムプロパティーを上書きできます。

Camel 2.8 には、使用するスキームを明示的に設定できる http.proxyScheme プロパティーもあります。

CHARSET の設定

POST を使用してデータを送信する場合は、Exchange プロパティーを使用して charset を設定できます。

```
exchange.setProperty(Exchange.CHARSET_NAME, "ISO-8859-1");
```

スケジュールされたポーリングを使用したサンプル

この例では、Google ホームページを 10 秒ごとにポーリングし、そのページをファイル message.html に書き込みます。

```
from("timer://foo?fixedRate=true&delay=0&period=10000")
    .to("http4://www.google.com")
    .setHeader(FileComponent.HEADER_FILE_NAME, "message.html")
    .to("file:target/google");
```

エンドポイント URI からの URI パラメーター

この例では、完全な URI エンドポイントがあり、これは Web ブラウザーに入力した内容になります。当然ながら、複数の URI パラメーターは、Web ブラウザーと同じように & amp; 文字をセパレーターとして使用して設定できます。この場合、Camel は複雑ではありません。

```
// we query for Camel at the Google page
template.sendBody("http4://www.google.com/search?q=Camel", null);
```

メッセージの URI パラメーター

```
Map headers = new HashMap();
headers.put(Exchange.HTTP_QUERY, "q=Camel&lr=lang_en");
// we query for Camel and English language at Google
template.sendBody("http4://www.google.com/search", null, headers);
```

上記のヘッダー値では、先頭に ? を付け ず、通常 & amp; 文字でパラメーターを分離することができることに注意してください。

レスポンスコードの取得

HTTP4 コンポーネントから HTTP 応答コードを取得するには、Exchange.HTTP_RESPONSE_CODE で Out メッセージヘッダーの値を取得します。

```
Exchange exchange = template.send("http4://www.google.com/search", new Processor() {
  public void process(Exchange exchange) throws Exception {
    exchange.getIn().setHeader(Exchange.HTTP_QUERY, constant("hl=en&q=activemq"));
  }
});
Message out = exchange.getOut();
int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
```

COOKIE の無効化

Cookie を無効にするには、URI オプション httpClient.cookiePolicy=ignoreCookiesを追加して HTTP Client が Cookie を無視するように設定します。

高度な使用方法

HTTP プロデューサーをより詳細に制御する必要がある場合は、HttpComponent を使用する必要があります。ここで、さまざまなクラスを設定してカスタム動作を提供できます。

JSSE 設定ユーティリティーの使用

Camel 2.8 以降、HTTP4 コンポーネントは Camel JSSE 設定ユーティリティーを介して SSL/TLS 設定をサポートします。このユーティリティーは、作成する必要のあるコンポーネント固有のコードの量を大幅に減らし、エンドポイントおよびコンポーネントレベルで設定可能です。 以下の例は、HTTP4 コンポーネントの設定方法を示しています。Security Guide の Configuring Transport Security for Camel Components も参照してください。

エンドポイントの SPRING DSL ベースの設定

```
...
  <camel:sslContextParameters
      id="sslContextParameters">
    <camel:keyManagers
        keyPassword="keyPassword">
      <camel:keyStore
          resource="/users/home/server/keystore.jks"
          password="keystorePassword"/>
    </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="https4://127.0.0.1/mail/?sslContextParametersRef=sslContextParameters"/>...
```

APACHE HTTP クライアントを直接設定

基本的に camel-http4 コンポーネントは Apache HttpClient 上に構築されています。詳細は

SSL/TLS のカスタマイズ を参照するか、org.apache.camel.component.http4.HttpsServerTestSupport unit test ベースクラスを確認してください。また、カスタムの org.apache.camel.component.http4.HttpClientConfigurer を実装して、完全な制御が必要な場合は、http クライアントでいくつかの設定を行うこともできます。

ただし、キーストアとトラストストアを指定する だけ の場合は、Apache HTTP HttpClientConfigurer でこれを行うことができます。以下に例を示します。

```
KeyStore keystore = ...;
KeyStore truststore = ...;

SchemeRegistry registry = new SchemeRegistry();
registry.register(new Scheme("https", 443, new SSLSocketFactory(keystore, "mypassword",
truststore)));
```

次に、HttpClientConfigurer を実装するクラスを作成し、上記の例ごとにキーストアまたはトラストストアを提供する https プロトコルを登録する必要があります。次に、Camel ルートビルダークラスから、以下のようにフックできます。

```
HttpComponent httpComponent = getContext().getComponent("http4", HttpComponent.class);
httpComponent.setHttpClientConfigurer(new MyHttpClientConfigurer());
```

Spring DSL を使用してこれを実行する場合は、URI を使用して HttpClientConfigurer を指定できます。以下に例を示します。

```
<bean id="myHttpClientConfigurer"
 class="my.https.HttpClientConfigurer">
</bean>

<to uri="https4://myhostname.com:443/myURL?
httpClientConfigurer=myHttpClientConfigurer"/>
```

上記のように HttpClientConfigurer を実装し、キーストアとトラストストアを設定すると問題なく機能します。

HTTPS を使用した GOTCHAS の認証

エンドユーザーは、HTTPS での認証に問題があることを報告していました。この問題は最終的に、カスタム設定済みの org.apache.http.protocol.HttpContext を指定して解決されました。

1.HttpContexts の(Spring)ファクトリーを作成します。

```
public class HttpContextFactory {

  private String httpHost = "localhost";
  private String httpPort = 9001;

  private BasicHttpContext httpContext = new BasicHttpContext();
  private BasicAuthCache authCache = new BasicAuthCache();
  private BasicScheme basicAuth = new BasicScheme();

  public HttpContext getObject() {
    authCache.put(new HttpHost(httpHost, httpPort), basicAuth);

    httpContext.setAttribute(ClientContext.AUTH_CACHE, authCache);

    return httpContext;
  }

  // getter and setter
}
```

2.Spring アプリケーションコンテキストファイルで HttpContext を宣言します。

```
<bean id="myHttpContext" factory-bean="httpContextFactory" factory-method="getObject"/>
```

3.http4 URL のコンテキストを参照します。

```
<to uri="https4://myhostname.com:443/myURL?httpContext=myHttpContext"/>
```

異なる SSLCONTEXTPARAMETERS の使用

HTTP4 コンポーネントは、コンポーネントごとに org.apache.camel.util.jsse.SSLContextParameters の 1 つのインスタンスのみをサポートします。2 つ以上の異なるインスタンスを使用する必要がある場合は、以下のように複数の HTTP4 コンポーネントを設定する必要があります。この例は、それぞれ sslContextParameters プロパティーの独自のインスタンスを使用する 2 つのコンポーネントを示しています。

```
<bean id="http4-foo" class="org.apache.camel.component.http4.HttpComponent">
   <property name="sslContextParameters" ref="sslContextParams1"/>
   <property name="x509HostnameVerifier" ref="hostnameVerifier"/>
</bean>

<bean id="http4-bar" class="org.apache.camel.component.http4.HttpComponent">
   <property name="sslContextParameters" ref="sslContextParams2"/>
   <property name="x509HostnameVerifier" ref="hostnameVerifier"/>
</bean>
```

第72章 IBATIS

IBATIS

ibatis: コンポーネントを使用すると、Apache iBATIS を使用して、リレーショナルデータベースでデータのクエリー、ポーリング、挿入、更新、および削除を行うことができます。

PREFER MYBATIS

Apache iBatis プロジェクトがアクティブではなくなりました。プロジェクトは Apache の外部に移動し、MyBatis プロジェクトとして知られています。したがって、代わりに MyBatis を使用することが推奨されます。この camel-ibatis コンポーネントは Camel 3.0 で削除されます。

ibatis は Spring 4.x をサポートしません。そのため、Spring 3.x 以前のみを iBatis で使用できます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-ibatis</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
ibatis:statementName[?options]
```

statementName は、評価するクエリー、挿入、更新、または削除の操作にマップする iBATIS XML 設定ファイルの名前です。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

このコンポーネントはデフォルトで、クラスパスのルートから iBatis SqlMapConfig ファイルをロードし、SqlMapConfig.xml という名前が付けられたことが予想されます。Spring リソースの読み込みを使用するため、クラスパス、ファイル、または http を接頭辞として使用し、これらのスキームでリソースを読み込むことができます。Camel 2.2 では、setSqlMapConfig (String) メソッドを使用して iBatisComponent でこれを設定できます。

オプション

オプション	型	デフォルト	説明
consumer.onConsume	**文字列**	**null**	消費後に実行するステートメント。たとえば、Apache Camel で消費および処理された後に行を更新するために使用できます。後ほどサンプルを参照してください。複数のステートメントはコンマで区切ることができます。
consumer.useIterator	**boolean**	**true**	**true** の場合、ポーリングが個別に処理されるときに返される各行。**false** の場合、データの **リスト** 全体が IN ボディーとして設定されます。
consumer.routeEmptyResultSet	**boolean**	**false**	Apache Camel 2.0: 空の結果セットをルーティングするかどうかを設定します。デフォルトでは、空の結果セットはルーティングされません。
statementType	**StatementType**	**null**	Apache Camel 1.6.1/2.0: 呼び出す iBatisis **SqlMapClient** メソッドを制御するために IbatisProducer に指定する必要があります。enum 値は **QueryForObject**、**QueryForList**、**Insert**、**Update**、**Delete** です。

maxMessagesPerPoll	**int**	**0**	Apache Camel 2.0: ポーリングごとに収集する最大メッセージを定義する整数。デフォルトでは最大値は設定されていません。たとえば制限を 1000 などに設定して、数千のファイルがあるサーバーの起動を回避できます。無効にするには、0 または負の値を設定します。
分離	**文字列**	**TRANSACTION_REPEATABLE_READ**	*Camel 2.9:* 文字列 は、のトランザクション分離レベルを定義します。使用できる値は TRANSACTION_NONE、TRANSACTION_READ_UNCOMMITTED、TRANSACTION_READ_COMMITTED、TRANSACTION_REPEATABLE_READ、TRANSACTION_SERIALIZABLE です。

分離	**文字列**	**TRANSACTION_REPEATABLE_READ**	*Camel 2.9:* 文字列 は、のトランザクション分離レベルを定義します。使用できる値は TRANSACTION_NONE、TRANSACTION_READ_UNCOMMITTED、TRANSACTION_READ_COMMITTED、TRANSACTION_REPEATABLE_READ、TRANSACTION_SERIALIZABLE です。

メッセージヘッダー

Apache Camel は、IN または OUT のいずれかの結果を、使用する operationName のヘッダーで入力します。

ヘッダー	タイプ	説明

CamelIBatisStatementName	**文字列**	Apache Camel 2.0: 使用される **statementName** （例：insertAccount）。
CamelIBatisResult	**オブジェクト**	Apache Camel 1.6.2/2.0: いずれかの操作で iBatis から返される **応答**。たとえば、**INSERT** は自動生成キーや行数などを返すことができます。

メッセージボディー

Apache Camel 1.6.2/2.0: iBatis からの応答は、SELECT ステートメントである場合にのみボディーとして設定されます。つまり、たとえば、Apache Camel は INSERT ステートメントではボディーを置き換えません。これにより、ルーティングを継続し、元のボディーを維持することができます。iBatis からの応答は、常にキー CamelIBatisResult を持つヘッダーに保存されます。

サンプル

たとえば、JMS キューから Bean を使用し、それらをデータベースに挿入する場合は、以下を実行できます。

```
from("activemq:queue:newAccount").
  to("ibatis:insertAccount?statementType=Insert");
```

statementType を指定する必要があります。これは、Apache Camel に対して SqlMapClient 操作を呼び出すよう指示します。

insertAccount は、SQL マップファイルの iBatis ID に置き換えます。

```
<!-- Insert example, using the Account parameter class -->
<insert id="insertAccount" parameterClass="Account">
  insert into ACCOUNT (
    ACC_ID,
    ACC_FIRST_NAME,
    ACC_LAST_NAME,
    ACC_EMAIL
  )
  values (
    #id#, #firstName#, #lastName#, #emailAddress#
  )
</insert>
```

STATEMENTTYPE を使用した IBATIS の制御の強化

Apache Camel 1.6.1/2.0 で利用可能: iBatis エンドポイントへのルーティングでは、より詳細な制御が必要です。これにより、実行する SQL ステートメントが SELEECT、UPDATE、DELETE、または INSERT であるかを制御できます。これは Apache Camel 1.6.1/2.0 で実行できるようになりました。たとえば、IN ボディーに SELECT ステートメントへのパラメーターが含まれる iBatis エンドポイントにルーティングする場合は、以下を実行できます。

```
from("direct:start")
    .to("ibatis:selectAccountById?statementType=QueryForObject")
    .to("mock:result");
```

上記のコードでは、iBatis ステートメント selectAccountById を呼び出し、IN 本文に 整数 タイプなどの取得するアカウント ID が含まれる必要があります。

QueryForList などの他の操作でも同じことができます。

```
from("direct:start")
    .to("ibatis:selectAllAccounts?statementType=QueryForList")
    .to("mock:result");
```

また UPDATE と同じです。ここで、Account オブジェクトを IN ボディーとして iBatis に送信できます。

```
from("direct:start")
    .to("ibatis:updateAccount?statementType=Update")
    .to("mock:result");
```

スケジュールされたポーリングの例

このコンポーネントはスケジュールされたポーリングをサポートしないため、Timer コンポーネントや Quartz コンポーネントなどのスケジュールされたポーリングをトリガーする別のメカニズムを使用する必要があります。

以下の例では、Timer コンポーネントを使用して 30 秒ごとにデータベースをポーリングし、データを JMS キューに送信します。

```
from("timer://pollTheDatabase?delay=30000").to("ibatis:selectAllAccounts?
statementType=QueryForList").to("activemq:queue:allAccounts");
```

また、使用されている iBatis SQL マップファイルは次のとおりです。

```
<!-- Select with no parameters using the result map for Account class. -->
<select id="selectAllAccounts" resultMap="AccountResult">
  select * from ACCOUNT
</select>
```

ONCONSUME の使用

このコンポーネントは、Apache Camel によってデータが消費および処理された 後 のステートメントの実行をサポートします。これにより、データベースで更新後の更新を行うことができます。すべてのステートメントは UPDATE ステートメントである必要があることに注意してください。Apache Camel は、名前をコンマで区切る必要がある複数のステートメントの実行をサポートします。

以下のルートは、consumeAccount ステートメントデータが処理されることを示しています。これにより、データベースの行のステータスを processed に変更できるため、2 回以上消費しないようにします。

```
from("ibatis:selectUnprocessedAccounts?
consumer.onConsume=consumeAccount").to("mock:results");
```

および sqlmap ファイルのステートメントは、以下のようになります。

```
<select id="selectUnprocessedAccounts" resultMap="AccountResult">
    select * from ACCOUNT where PROCESSED = false
</select>
```

```
<update id="consumeAccount" parameterClass="Account">
    update ACCOUNT set PROCESSED = true where ACC_ID = #id#
</update>
```

第73章 IRC

IRC コンポーネント

irc コンポーネントは IRC (Internet Relay Chat)トランスポートを実装します。

URI 形式

```
irc:nick@host[:port]/#room[?options]
```

Apache Camel 2.0 では、以下の形式を使用することもできます。

```
irc:nick@host[:port]?channels=#channel1,#channel2,#channel3[?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	説明	例	デフォルト値
channels	2.0 の新機能。参加する IRC チャネルのコンマ区切りリスト。	channels=#channel1,#channel2	null
ニックネーム	チャットで使用されるニックネーム。	irc:MyNick@irc.server.org#channel or irc:irc.server.org#channel?nickname=MyUser	null
username	IRC サーバーのユーザー名。	irc:MyUser@irc.server.org#channel or irc:irc.server.org#channel?username=MyUser	ニックネームと同じです。
password	IRC サーバーパスワード。	password=somepass	なし

realname	IRC ユーザーの実際の名前。	**realname=MyName**	なし
colors	サーバーが色コードをサポートするかどうか。	**true, false**	**true**
onReply	コマンドまたは情報メッセージへの一般的な応答を処理するかどうか。	**true, false**	**false**
onNick	ニックネーム変更イベントを処理します。	**true, false**	**true**
onQuit	ユーザーの終了イベントを処理します。	**true, false**	**true**
onJoin	ユーザーの参加イベントを処理します。	**true, false**	**true**
onKick	開始イベントを処理します。	**true, false**	**true**
onMode	モード変更イベントを処理します。	**true, false**	**true**
onPart	ユーザー部分イベントを処理します。	**true, false**	**true**
onTopic	トピック変更イベントを処理します。	**true, false**	**true**
onPrivmsg	メッセージイベントを処理します。	**true, false**	**true**
trustManager	2.0 の新機能として、SSL サーバーの証明書の検証に使用されるトラストマネージャーです。	**trustManager=#referenceToTrustManagerBean**	**すべて** の証明書を受け入れるデフォルトのトラストマネージャーが使用されます。
keys	**Camel 2.2:** IRC チャネルキーのコンマ区切りリスト。重要な点は、チャネルと同じ順序でリストされます。必要なキーのみで複数のチャネルに参加する場合は、そのチャネルに空の値を挿入するだけで済みます。	**irc:MyNick@irc.server.org/#channel?keys=chankey**	**null**

sslContextParameters	*Camel 2.9:* レジストリー内の **org.apache.camel.util.jsse.SSLContextParameters** オブジェクトへの 参照。 この参照は、コンポーネントレベルで設定済みの SSLContextParameters を上書きします。 Security Guide および「JSSE 設定ユーティリティーの使用」 の Configuring Transport Security for Camel Components を参照してください。この設定は、**trustManager** オプションを上書きすることに注意してください。	\#mySslContextParameters	**null**

JSSE 設定ユーティリティーの使用

Camel 2.9 の時点で、IRC コンポーネントは を介して SSL/TLS 設定をサポートし http://camel.apache.org/http4.html#HTTP4-UsingtheJSSEConfigurationUtility ます。このユーティリティーは、作成する必要のあるコンポーネント固有のコードの量を大幅に減らし、エンドポイントおよびコンポーネントレベルで設定可能です。以下の例では、IRC コンポーネントで ユーティリティーを使用する方法を説明します。

Security Guide の Configuring Transport Security for Camel Components の章を参照してください。

エンドポイントのプログラムによる設定

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/truststore.jks");
ksp.setPassword("keystorePassword");
```

```
TrustManagersParameters tmp = new TrustManagersParameters();
tmp.setKeyStore(ksp);

SSLContextParameters scp = new SSLContextParameters();
scp.setTrustManagers(tmp);

Registry registry = ...
registry.bind("sslContextParameters", scp);

...

from(...)
    .to("ircs://camel-prd-user@server:6669/#camel-test?nickname=camel-
prd&password=password&sslContextParameters=#sslContextParameters");
```

エンドポイントの SPRING DSL ベースの設定

```
...
  <camel:sslContextParameters
      id="sslContextParameters">
    <camel:trustManagers>
      <camel:keyStore
          resource="/users/home/server/truststore.jks"
          password="keystorePassword"/>
    </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="ircs://camel-prd-user@server:6669/#camel-test?nickname=camel-
prd&password=password&sslContextParameters=#sslContextParameters"/>...
```

レガシーの基本設定オプションの使用

以下のように、SSL 対応の IRC サーバーに接続することもできます。

```
ircs:host[:port]/#room?username=user&password=pass
```

デフォルトでは、IRC トランスポートは SSLDefaultTrustManager を使用します。独自のカスタムトラストマネージャーを提供する必要がある場合は、以下のように trustManager パラメーターを使用します。

```
ircs:host[:port]/#room?
username=user&password=pass&trustManager=#referenceToMyTrustManagerBean
```

鍵の使用

Camel 2.2 で利用可能一部の irc 部屋では、そのチャネルに参加できるようにキーを指定する必要があります。キーは秘密の単語です。

たとえば、チャネル 1 と 3 のみがキーを使用する 3 つのチャネルに参加します。

```
irc:nick@irc.server.org?channels=#chan1,#chan2,#chan3&keys=chan1Key,,chan3key
```

第74章 JASYPT

JASYPT コンポーネント

Camel 2.5 で利用可能

Jasypt は、暗号化と復号化を容易にする簡略化された暗号化ライブラリーです。Camel は Jasypt と統合し、プロパティー ファイルの機密情報を暗号化できるようにします。camel-jasypt をクラスパスでドロップすると、暗号化された値は Camel によって自動的に復号化されます。これにより、人間はユーザー名とパスワードなどの機密情報を簡単に特定できなくなります。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jasypt</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

ツール

Jasypt コンポーネントは、値を暗号化または復号化するためのコマンドラインツールを少し提供します。

コンソールは構文と提供するオプションを出力します。

```
Apache Camel Jasypt takes the following options

 -h or -help = Displays the help screen
 -c or -command <command> = Command either encrypt or decrypt
```

```
-p or -password <password> = Password to use
-i or -input <input> = Text to encrypt or decrypt
-a or -algorithm <algorithm> = Optional algorithm to use
```

たとえば、値 tiger を暗号化するには、以下のパラメーターで実行します。apache camel kit で、lib フォルダーに移動し、以下の java cmd を実行します。<CAMEL_HOME> は、Camel ディストリビューションをダウンロードして展開します。

```
$ cd <CAMEL_HOME>/lib
$ java -jar camel-jasypt-2.5.0.jar -c encrypt -p secret -i tiger
```

以下の結果を出力します。

```
Encrypted text: qaEEacuW7BUti8LcMgyjKw==
```

つまり、暗号化された表現の qaEEacuW7BUti8LcMgyjKw== は、秘密 のマスターパスワードを知っている場合に tiger に復号化できます。ツールを再度実行すると、暗号化された値は別の結果を返します。ただし、値を復号すると、常に正しい元の値が返されます。

そのため、以下のパラメーターを使用して ツールを実行してテストできます。

```
$ cd <CAMEL_HOME>/lib
$ java -jar camel-jasypt-2.5.0.jar -c decrypt -p secret -i qaEEacuW7BUti8LcMgyjKw==
```

以下の結果を出力します。

```
Decrypted text: tiger
```

次に、プロパティー ファイルで暗号化された値を使用するのが理想です。パスワード値が暗号化され、値に ENC (value here)を囲むトークンがあることに注意してください。

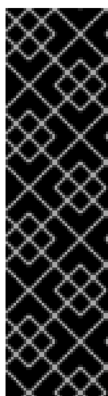
```
# refer to a mock endpoint name by that encrypted password
cool.result=mock:{{cool.password}}

# here is a password which is encrypted
cool.password=ENC(bsW9uV37gQ0QHFu7KO03Ww==)
```

CAMEL 2.5 および 2.6 のツールの依存関係

ツールには、接頭辞が optional/ の camel-jasypt の MANIFEST.MF ファイルに登録されたクラスパ

スに以下の JAR が必要です。したがって、上記の java cmd が、オプション の ディレクトリーの Apache Distribution から必要な JAR を選択できる理由です。

> jasypt-1.6.jar commons-lang-2.4.jar commons-codec-1.4.jar icu4j-4.0.1.jar



JAVA 1.5 ユーザー

icu4j-4.0.1.jar は、JDK 1.5 で実行する場合にのみ必要になります。

この JAR は Apache Camel によって配布されず、手動でダウンロードして Camel ディストリビューションの lib/optional ディレクトリーにコピーします。Apache Central Maven リポジトリー からダウンロードできます。

CAMEL 2.7 以降のツールの依存関係

jasypt 1.7 以降は完全にスタンドアロンになったため、追加の JAR は必要ありません。

URI オプション

以下のオプションは Jasypt コンポーネント専用です。

名前	デフォルト値	型	Description
password	null	文字列	復号化に使用するマスターパスワードを指定します。このオプションは必須です。詳細は、こちらを参照してください。
algorithm	null	文字列	使用する任意のアルゴリズムの名前。

マスターパスワードの保護

Jasypt が使用するマスターパスワードを提供する必要があります。これにより、値を復号できます。ただし、このマスターパスワードを開いた状態にすることは理想的な解決策ではない場合があります。そのため、JVM システムプロパティーまたは OS 環境設定として指定できます。これを選択すると、パスワード オプションはこれを指示する接頭辞をサポートします。sysenv: 指定されたキーを使用して OS システム環境を検索することを意味します。sys: JVM システムプロパティーを検索することを意味します。

たとえば、アプリケーションを起動する前にパスワードを指定できます。

```
$ export CAMEL_ENCRYPTION_PASSWORD=secret
```

次に、起動スクリプトの実行など、アプリケーションを起動します。

アプリケーションが稼働している場合、環境の設定を解除できます。

```
$ unset CAMEL_ENCRYPTION_PASSWORD
```

パスワード オプションは、password=sysenv:CAMEL_ENCRYPTION_PASSWORD のように定義することが関係します。

JAVA DSL を使用した例

Java DSL では、 Jasypt PropertiesParser インスタンスとして設定し、以下のように Properties コンポーネントに設定する必要があります。

```
// create the jasypt properties parser
JasyptPropertiesParser jasypt = new JasyptPropertiesParser();
// and set the master password
jasypt.setPassword("secret");

// create the properties component
PropertiesComponent pc = new PropertiesComponent();
pc.setLocation("classpath:org/apache/camel/component/jasypt/myproperties.properties");
// and use the jasypt properties parser so we can decrypt values
pc.setPropertiesParser(jasypt);

// add properties component to camel context
context.addComponent("properties", pc);
```

次に、プロパティーファイル myproperties.properties には、以下のような暗号化された値が含まれます。パスワード値が暗号化され、値に ENC (value here)を囲むトークンがあることに注意してください。

```
# refer to a mock endpoint name by that encrypted password
cool.result=mock:{{cool.password}}

# here is a password which is encrypted
cool.password=ENC(bsW9uV37gQ0QHFu7KO03Ww==)
```

SPRING XML の例

Spring XML では、以下に示す JasyptPropertiesParser を設定する必要があります。次に、Camel Properties コンポーネントはプロパティーパーサーとして jasypt を使用するように指示されます。つまり、Jasypt はプロパティーで検索された値を復号化する機会を持ちます。

```
<!-- define the jasypt properties parser with the given password to be used -->
<bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <property name="password" value="secret"/>
</bean>

<!-- define the camel properties component -->
<bean id="properties"
class="org.apache.camel.component.properties.PropertiesComponent">
    <!-- the properties file is in the classpath -->
    <property name="location"
value="classpath:org/apache/camel/component/jasypt/myproperties.properties"/>
    <!-- and let it leverage the jasypt parser -->
    <property name="propertiesParser" ref="jasypt"/>
</bean>
```

Properties コンポーネントは、以下に示す < camelContext> タグ内で インライン化することもできます。propertiesParserRef 属性を使用して Jasypt を参照する方法に注意してください。

```
<!-- define the jasypt properties parser with the given password to be used -->
<bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <!-- password is mandatory, you can prefix it with sysenv: or sys: to indicate it should use
         an OS environment or JVM system property value, so you dont have the master
password defined here -->
    <property name="password" value="secret"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- define the camel properties placeholder, and let it leverage jasypt -->
    <propertyPlaceholder id="properties"

location="classpath:org/apache/camel/component/jasypt/myproperties.properties"
                         propertiesParserRef="jasypt"/>
    <route>
        <from uri="direct:start"/>
        <to uri="{{cool.result}}"/>
    </route>
</camelContext>
```

関連項目

- セキュリティー

- *プロパティー*
- ***ActiveMQ で暗号化されたパスワード: ActiveMQ はこの camel-jasypt コンポーネントと同様の機能を持ちます。***

第75章 JAXB

JAXB コンポーネント

JAXB データフォーマットは JAXB サポートを提供します。Camel は、XML データの JAXB アノテーションの付けられたクラスへのアンマーシャリング、およびクラスから XML へのマーシャリングをサポートします。詳細は、JAXB を参照してください。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

第76章 JCACHE

JCACHE コンポーネント

Camel 2.17 以降で利用可能

jcache コンポーネントを使用すると、JCache (JSR-107)をキャッシュ実装として使用してキャッシュ操作を実行できます。キャッシュ自体はオンデマンドで作成されます。その名前のキャッシュがすでに存在する場合は、単に元の設定で使用されます。

このコンポーネントは、プロデューサーおよびイベントベースのコンシューマーエンドポイントをサポートします。

Cache コンシューマーはイベントベースのコンシューマーであり、特定のキャッシュアクティビティーをリッスンして応答するために使用できます。既存のキャッシュから選択を行う必要がある場合は、キャッシュコンポーネントに定義されたプロセッサーを使用します。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-jcache</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
cache://cacheName[?options]
```

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=#beanRef&...

オプション

名前	デフォルト値	説明

cachingProvider	**null**	javax.cache.spi.CachingProvider の完全修飾クラス名。OSGI 環境で必須です。
cacheConfiguration	**null**	javax.cache.configuration.Configuration インスタンスへの参照
cacheConfigurationProperties	**null**	javax.cache.CacheManager を作成するための **javax.cache.spi.CachingProvider** の java.util.Properties への参照。
configurationUri	**null**	の実装固有の URI **javax.cache.CacheManager**
cacheLoaderFactory	**null**	**javax.cache.configuration.Factory** への参照 **javax.cache.integration.CacheLoader**
cacheWriterFactory	**null**	**javax.cache.configuration.Factory** への参照 **javax.cache.integration.CacheWriter**
expiryPolicyFactory	**null**	**javax.cache.configuration.Factory** への参照 **javax.cache.expiry.ExpiryPolicy**
readThrough	**false**	read-through モードが必要であるかどうかを示すフラグ
writeThrough	**false**	write-through モードが必要であるかどうかを示すフラグ
storeByValue	true	キャッシュが store-by-value または store-by-reference であるかどうかを示すフラグ
statisticsEnabled	**fasle**	統計収集が有効になっているかどうかを示すフラグ
managementEnabled	**false**	管理が有効になっているかどうかを示すフラグ
filteredEvents	**null**	フィルターするイベントタイプのコンマ区切りリスト。それらが一緒に指定されると eventFilters によって上書きされます。

eventFilters	**null**	javax.cache.event.CacheEntryEventFilter 参照のコンマ区切りリスト。一緒に指定された場合に filteredEvents を上書きします。
oldValueRequired	**false**	イベントに古い値が必要であるかどうかを示すフラグ。サポートされる値は CREATED、UPDATED、REMOVED、EXPIRED です。
同期	false	イベントリスナーがイベントの原因となるスレッドをブロックするかどうかを示すフラグ
action	null	適用するデフォルトのアクション。ヘッダーの値には優先度があります。
createCacheIfNotExists	true	cacheName で識別されるキャッシュが存在しない場合に作成する必要があるかどうかを設定します。

ヘッダー変数

名前	タイプ	説明
CamelJCacheAction	**java.lang.String**	サポートされる値は PUT、PUTALL、PUTIFABSENT、GET、GETALL、GETANDREMOVE、GETANDREPLACE、GETANDPUT、REPLACE、REMOVE、REMOVEALL、INVOKE、CLEAR です。
CamelJCacheResult	**java.lang.Object**	アクションの結果、つまり PUT、REMOVE、REPLACE のブール値
CamelJCacheEventType	**java.lang.String**	イベントのタイプ **javax.cache.event.EventType**
CamelJCacheKey	**java.lang.Object**	アクションを適用するキー
CamelJCacheKeys	java.util.Set<java-lang.Object>	GETALL、REMOVEALL、INVOKE に使用されるアクションを適用するキーのセット

CamelJCacheOldValue	**java.lang.Object**	コンシューマー側では、ヘッダー値にはキーに関連付けられた古い値が含まれます。プロデューサー側で、操作などの CAS を使用するには、ヘッダーに予想される古い値が含まれている必要があります。
CamelJCacheEntryProcessor	javax.cache.processor.EntryProcessor	**INVOKE** アクションに使用するエントリープロセッサー
CamelJCacheEntryArgs	**java.util.collection<java.lang.Object>**	に渡す追加の引数 **javax.cache.processor.EntryProcessor**

JCACHE ベースのべき等リポジトリーの例 :

```
JCacheIdempotentRepository idempotentRepo = new JCacheIdempotentRepository();
idempotentRepo.setCacheName("idempotent-cache")

from("direct:in")
    .idempotentConsumer(header("messageId"), idempotentRepo)
    .to("mock:out");
```

JCACHE ベースの集約リポジトリーの例 :

```
package org.apache.camel.component.jcache.processor.aggregate;

import org.apache.camel.EndpointInject;
import org.apache.camel.Exchange;
import org.apache.camel.Produce;
import org.apache.camel.ProducerTemplate;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.mock.MockEndpoint;
import org.apache.camel.processor.aggregate.AggregationStrategy;
import org.junit.Test;

public class JCacheAggregationRepositoryRoutesTest extends
JCacheAggregationRepositoryTestSupport {
    private static final String MOCK_GOTCHA = "mock:gotcha";
    private static final String DIRECT_ONE = "direct:one";
    private static final String DIRECT_TWO = "direct:two";
    @EndpointInject(uri = MOCK_GOTCHA)
```

```
private MockEndpoint mock;
@Produce(uri = DIRECT_ONE)
private ProducerTemplate produceOne;
@Produce(uri = DIRECT_TWO)
private ProducerTemplate produceTwo;
@Test
public void checkAggregationFromTwoRoutes() throws Exception {
    final JCacheAggregationRepository repoOne = createRepository(false);
    final JCacheAggregationRepository repoTwo = createRepository(false);
    final int completionSize = 4;
    final String correlator = "CORRELATOR";
    RouteBuilder rbOne = new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from(DIRECT_ONE).routeId("AggregatingRouteOne")
                .aggregate(header(correlator))
                .aggregationRepository(repoOne)
                .aggregationStrategy(new MyAggregationStrategy())
                .completionSize(completionSize)
                .to(MOCK_GOTCHA);
        }
    };
    RouteBuilder rbTwo = new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from(DIRECT_TWO).routeId("AggregatingRouteTwo")
                .aggregate(header(correlator))
                .aggregationRepository(repoTwo)
                .aggregationStrategy(new MyAggregationStrategy())
                .completionSize(completionSize)
                .to(MOCK_GOTCHA);
        }
    };
    context().addRoutes(rbOne);
    context().addRoutes(rbTwo);
    context().start();
    mock.expectedMessageCount(1);
    mock.expectedBodiesReceived(1 + 2 + 3 + 4);
    produceOne.sendBodyAndHeader(1, correlator, correlator);
    produceTwo.sendBodyAndHeader(2, correlator, correlator);
    produceOne.sendBodyAndHeader(3, correlator, correlator);
    produceOne.sendBodyAndHeader(4, correlator, correlator);
    mock.assertIsSatisfied();
}
private class MyAggregationStrategy implements AggregationStrategy {
    @Override
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        if (oldExchange == null) {
            return newExchange;
        } else {
            Integer n = newExchange.getIn().getBody(Integer.class);
            Integer o = oldExchange.getIn().getBody(Integer.class);
            Integer v = (o == null ? 0 : o) + (n == null ? 0 : n);
            oldExchange.getIn().setBody(v, Integer.class);
            return oldExchange;
        }
```

```
        }
    }
    protected JCacheAggregationRepository createRepository(boolean optimistic) throws
Exception {
        JCacheAggregationRepository repository = new JCacheAggregationRepository();
        repository.setConfiguration(new JCacheConfiguration());
        repository.setCacheName("aggregation-repository");
        repository.setOptimistic(optimistic);
        return repository;
    }
}
```

第77章 JCLOUDS

JCLOUDS コンポーネント

Camel 2.9 以降で利用可能

このコンポーネントにより、クラウドプロバイダーのキー/値エンジン(blob ストア)および Compute サービスとの対話が可能になります。コンポーネントは jclouds を使用します。これは、Blobstores および Compute サービスの抽象化を提供するライブラリーです。

ComputeService は、クラウド内でマシンを管理するタスクを簡素化します。たとえば、ComputeService を使用して 5 つのマシンを起動し、それらにソフトウェアをインストールできます。blobstore は、Amazon S3 などのキーと値プロバイダーの処理を簡素化します。たとえば、BlobStore はコンテナーの簡単な Map ビューを提供できます。

camel jclouds コンポーネントを使用すると、JcloudsBlobStoreEndpoint と JcloudsComputeEndpoint の 2 種類のエンドポイントを指定するため、両方の抽象化を使用できます。Blobstore エンドポイントにはプロデューサーとコンシューマーの両方を設定できますが、コンピュートエンドポイントにはプロデューサーのみを使用できます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-jclouds</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

コンポーネントの設定

camel jclouds コンポーネントは、初期化中にコンポーネントに渡される限り、複数の jclouds blobstores および Compute サービスを利用します。コンポーネントは、リスト blobstores および Compute サービスを受け入れます。以下は、その設定方法を示しています。

```
<bean id="jclouds" class="org.apache.camel.component.jclouds.JcloudsComponent">
   <property name="computeServices">
```

```
            <list>
                <ref bean="computeService"/>
            </list>
        </property>
        <property name="blobStores">
            <list>
                <ref bean="blobStore"/>
            </list>
        </property>
    </bean>

    <!-- Creating a blobstore from spring / blueprint xml -->
    <bean id="blobStoreContextFactory"
class="org.jclouds.blobstore.BlobStoreContextFactory"/>

    <bean id="blobStoreContext" factory-bean="blobStoreContextFactory" factory-
method="createContext">
        <constructor-arg name="provider" value="PROVIDER_NAME"/>
        <constructor-arg name="identity"  value="IDENTITY"/>
        <constructor-arg name="credential" value="CREDENTIAL"/>
    </bean>

    <bean id="blobStore" factory-bean="blobStoreContext" factory-method="getBlobStore"/>

    <!-- Creating a compute service from spring / blueprint xml -->
    <bean id="computeServiceContextFactory"
class="org.jclouds.compute.ComputeServiceContextFactory"/>

    <bean id="computeServiceContext" factory-bean="computeServiceContextFactory" factory-
method="createContext">
        <constructor-arg name="provider" value="PROVIDER_NAME"/>
        <constructor-arg name="identity"  value="IDENTITY"/>
        <constructor-arg name="credential" value="CREDENTIAL"/>
    </bean>

    <bean id="computeService" factory-bean="computeServiceContext" factory-
method="getComputeService"/>
```

ご覧のとおり、コンポーネントは複数の blobstores および Compute サービスを処理できます。各エンドポイントによって使用される実際の実装は、URI 内でプロバイダーを渡すことで指定されます。

URI 形式

```
jclouds:blobstore:[provider id][?options]
jclouds:compute:[provider id][?options]
```

プロバイダー ID は、ターゲットサービスを提供するクラウドプロバイダーの名前です(例：aws-s3 または aws_ec2)。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

BLOBSTORE URI オプション

名前	デフォルト値	説明
operation	PUT	*producer Only*.blobstore に送信される操作のタイプを指定します。許可される値は PUT、GET です。
container	null	blob コンテナーの名前。
blobName	null	Blob の名前。

これらのオプションはいくつでも使用できます。

> ***jclouds:blobstore:aws-s3?operation=CamelJcloudsGet&container=mycontainer&blobName=someblob***

プロデューサーエンドポイントの場合、適切なヘッダーをメッセージに渡すことで、上記の URI オプションをすべて上書きできます。

BLOBSTORE のメッセージヘッダー

ヘッダー	説明

CamelJcloudsOperation	Blob で実行される操作。有効なオプションは です。 • PUT • GET
CamelJcloudsContainer	blob コンテナーの名前。
CamelJcloudsBlobName	Blob の名前。

BLOBSTORE 使用例

例 1: BLOB に配置

この例では、jclouds コンポーネントを使用して、Blob 内にメッセージを格納する方法を説明します。

```
from("direct:start")
    .to("jclouds:blobstore:aws-s3" +
        "?operation=PUT" +
        "&container=mycontainer" +
        "&blobName=myblob");
```

上記の例では、メッセージのヘッダーを使用して、任意の URI パラメーターを上書きできます。上記の例は、xml を使用してルートを定義する方法を示しています。

```
<route>
    <from uri="direct:start"/>
    <to uri="jclouds:blobstore:aws-s3?
operation=PUT&container=mycontainer&blobName=myblob"/>
</route>
```

例 2: BLOB の取得/読み取り

この例では、jclouds コンポーネントを使用して Blob の contnet を読み取る方法を説明します。

```
from("direct:start")
    .to("jclouds:blobstore:aws-s3" +
        "?operation=GET" +
```

```
        "&container=mycontainer" +
        "&blobName=myblob");
```

上記の例では、メッセージのヘッダーを使用して、任意の URI パラメーターを上書きできます。上記の例は、xml を使用してルートを定義する方法を示しています。

```
<route>
    <from uri="direct:start"/>
    <to uri="jclouds:blobstore:aws-s3?
operation=PUT&container=mycontainer&blobName=myblob"/>
</route>
```

例 3: BLOB の使用

この例では、指定されたコンテナーにあるすべての blob を消費します。生成されたエクスチェンジには、Blob のペイロードをボディーとして含まれます。

```
    from("jclouds:blobstore:aws-s3" +
        "?container=mycontainer")
        .to("direct:next");
```

以下に示すように、xml を使用して同じゴールを実現できます。

```
<route>
    <from uri="jclouds:blobstore:aws-s3?
operation=GET&container=mycontainer&blobName=myblob"/>
    <to uri="direct:next"/>
</route>
```

COMPUTE サービスの URI オプション

名前	デフォルト値	説明

operation	CamelJcloudsPut	コンピュートサービスで実行される操作のタイプを指定します。許可される値は、CamelJcloudsCreateNode、CamelJcloudsRunScript、CamelJcloudsDestroyNode、CamelJcloudsListNodes、CamelJCloudsListImages、CamelJCloudsListHardware です。
imageId	null	*CamelJcloudsCreateNode 操作のみ* ノードの作成に使用される imageId。値は実際のクラウドプロバイダーによって異なります。
locationId	null	*CamelJcloudsCreateNode 操作のみ* ノードの作成に使用する場所。値は実際のクラウドプロバイダーによって異なります。
hardwareId	null	*CamelJcloudsCreateNode 操作のみ* ノードの作成に使用されるハードウェア。値は実際のクラウドプロバイダーによって異なります。
group	null	*CamelJcloudsCreateNode 操作のみ* 新規作成されたノードに割り当てられるグループ。値は実際のクラウドプロバイダーによって異なります。
nodeId	null	*CamelJcloudsRunScript & CamelJcloudsDestroyNode 操作のみ* スクリプトを実行するノードの ID、または破棄されるノードの ID。
user	null	*CamelJcloudsRunScript 操作のみ* スクリプトを実行するターゲットノードのユーザー。

Compute サービスで使用するパラメーターの組み合わせは、操作によって異なります。

> ***jclouds:compute:aws-ec2?operation=CamelJcloudsCreateNode&imageId=AMI_XXXXX&locationId=eu-west-1&group=mygroup***

コンピュートの使用例

以下の例は、java dsl および spring/blueprint xml での jclouds コンピュートプロデューサーの使用を示しています。

例 1: 利用可能なイメージの一覧表示

```
from("jclouds:compute:aws-ec2" +
   "&operation=CamelJCloudsListImages")
   .to("direct:next");
```

これにより、本文内にイメージの一覧が含まれるメッセージが作成されます。xml を使用して同じ操作を行うこともできます。

```
<route>
   <from uri="jclouds:compute:aws-ec2?operation=CamelJCloudsListImages"/>
   <to uri="direct:next"/>
</route>
```

例 2: 新規ノードを作成します。

```
from("direct:start").
to("jclouds:compute:aws-ec2" +
   "?operation=CamelJcloudsCreateNode" +
   "&imageId=AMI_XXXXX" +
   "&locationId=XXXXX" +
   "&group=myGroup");
```

これにより、クラウドプロバイダーに新しいノードが作成されます。この場合の out メッセージは、新規に作成されたノードに関する情報が含まれるメタデータのセットになります（例：ip、hostname）。以下は、spring xml を使用した同じです。

```
<route>
   <from uri="direct:start"/>
   <to uri="jclouds:compute:aws-ec2?
operation=CamelJcloudsCreateNode&imageId=AMI_XXXXX&locationId=XXXXX&group=myGr
oup"/>
</route>
```

例 3: 実行中のノードでシェルスクリプトを実行します。

```
from("direct:start").
to("jclouds:compute:aws-ec2" +
   "?operation=CamelJcloudsRunScript" +
```

```
        "?nodeId=10" +
        "&user=ubuntu");
```

上記のサンプルは in メッセージのボディーを取得します。これは、実行するシェルスクリプトが含まれることが予想されます。スクリプトが取得されると、ノードに送信され、指定したユーザー下で実行できるようになります（大文字の場合は）。ターゲットノードは、nodeId を使用して指定されます。nodeId は、ノードの作成時に取得できます。これは、結果のメタデータの一部となるか、または CamelJcloudsListNodes 操作を実行して取得できます。

これ は、コンポーネントに渡される Compute サービスを、適切な JClouds SSH 対応モジュール (jsch や sshj など)で初期化する必要があります。

以下は、spring xml を使用した同じです。

```
<route>
   <from uri="direct:start"/>
   <to uri="jclouds:compute:aws-ec2?operation=CamelJcloudsRunScript&?
nodeId=10&user=ubuntu"/>
</route>
```

その他の参考資料

jclouds の詳細については、Jclouds BlobStore Guide Jclouds Compute Guideを参照してください。

第78章 JCR

JCR コンポーネント

jcr コンポーネントを使用すると、プロデューサーで JCR 準拠のコンテンツリポジトリー(Apache Jackrabbitなど)との間でノードを追加/読み取るか、コンシューマーに EventListener を登録できます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jcr</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
jcr://user:password@repository/path/to/node
```

コンシューマーの追加

Camel 2.10 以降では、コンシューマーを JCR の EventListener またはプロデューサーとして使用して、識別子でノードを読み取ることができます。

使用方法

URI の repository 要素は、Camel コンテキストレジストリーで JCR Repository オブジェクトを検索するために使用されます。

プロデューサー

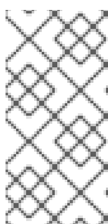
名前	デフォルト値	説明
CamelJcrOperation	CamelJcrInsert	使用する CamelJcrInsert または CamelJcrGetById 操作
CamelJcrNodeName	null	使用するノード名を決定するために使用されます。

CamelJcrNodeType	null	Camel 2.16: 新規ノードの追加時にプライマリーノードタイプを指定します。

メッセージが JCR プロデューサーエンドポイントに送信される場合：

- *操作が CamelJcrInsert: コンテンツリポジトリーに新しいノードが作成されると、IN メッセージのすべてのメッセージヘッダーが javax.jcr.Value インスタンスに変換され、新規ノードに追加され、ノードの UUID が OUT メッセージで返されます。*

- *操作が CamelJcrGetById の場合：メッセージボディーをノード識別子として使用して、新しいノードがリポジトリーから取得されます。*



注記

JCR Producer は、2.12.3 より前の Camel バージョンのメッセージヘッダーではなく、メッセージプロパティーを使用していました。

コンシューマー

コンシューマーは定期的に JCR に接続し、メッセージボディーで List<javax.jcr.observation.Event> を返します。

名前	デフォルト値	説明
eventTypes	0	javax.jcr.observation.Event.NODE_ADDED、javax.jcr.observation.Event.NODE_REMOVED などのビットマスク値としてエンコードされた 1 つ以上のイベントタイプの組み合わせ。
deep	false	true の場合、関連付けられた親ノードが現在のパスまたはそのサブグラフ内にあるイベントを受け取ります。
UUID	null	関連する親ノードにコンマ区切りの uuid 一覧に識別子の 1 つがあるイベントのみが受信されます。

nodeTypeNames	**null**	関連付けられた親ノードにノードタイプ（またはノード種別のいずれかのサブタイプ）のいずれかがあるイベントのみが受信されます。
noLocal	**false**	**noLocal** が **true** の場合、リスナーが登録されたセッションによって生成されたイベントは無視されます。そうでない場合は、無視されません。
sessionLiveCheckInterval	**60000**	各セッションがライブチェックするまで待機する間隔（ミリ秒単位）。
sessionLiveCheckIntervalOnStart	**3000**	最初のセッションがライブチェックされるまで待機する間隔（ミリ秒単位）。
username		**Camel 2.15:** URI の authority セクションではなく、URI パラメーターとしてユーザー名を指定できます。
password		**Camel 2.15:** URI の authority セクションではなく、URI パラメーターとしてパスワードを指定できます。
workspaceName	**null**	**Camel 2.16:** デフォルトとは異なるワークスペースを指定できます。

例

以下のスニペットは、content リポジトリーの /home/test ノードに node という名前のノードを作成します。1 つの追加プロパティーもノードに追加されます。my.contents.property には送信されるメッセージのボディーが含まれます。

```
from("direct:a").setHeader(JcrConstants.JCR_NODE_NAME, constant("node"))
    .setHeader("my.contents.property", body())
    .to("jcr://user:pass@repository/home/test");
```

以下のコードは、Event.NODE_ADDED および Event.NODE_REMOVED イベント（イベントタイプ 1 と 2 の両方）のパス import-application/inbox の下に EventListener を登録し、すべての子についてディープをリッスンします。

_

```
<route>
    <from uri="jcr://user:pass@repository/import-application/inbox?eventTypes=3&deep=true"
/>
    <to uri="direct:execute-import-application" />
</route>
```

第79章 JDBC

JDBC コンポーネント

JDBC コンポーネントを使用すると、JDBC 経由でデータベースにアクセスできます。ここでは、SQL クエリー(SELECT)および操作(INSERT、UPDATE など)がメッセージボディーに送信されます。このコンポーネントは、spring-jdbc を使用する SQL Component コンポーネントとは異なり、標準の JDBC API を使用します。

警告

このコンポーネントはプロデューサーエンドポイントを定義する場合にのみ使用できます。つまり、from （）ステートメントで JDBC コンポーネントを使用することはできません。

URI 形式

```
jdbc:dataSourceName[?options]
```

このコンポーネントはプロデューサーエンドポイントのみをサポートします。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
readSize	0	ポーリングクエリーで読み取り可能なデフォルトの最大行数。

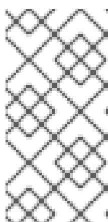
statement.<xxx>	**null**	Apache Camel 2.1: クエリーを実行するために背後で使用される **java.sql.Statement** に追加のオプションを設定します。たとえば、**statement.maxRows=10 を使用します**。詳細なドキュメントは、**java.sql.Statement** javadoc のドキュメント を参照してください。
useJDBC4ColumnNameAndLabelSemantics	**true**	JDBC 4/3 列ラベル/名前セマンティクスを使用するかどうかを設定します。JDBC ドライバーでデータを選択する場合に、このオプションを使用して **false** にすることができます。これは、エイリアスを使用し て **SQL SELECT を使用する場合にのみ適用されます（例：SQL SELECT id を識別子、persons から given_name のよう** な名前）。
resetAutoCommit	**true**	Camel 2.9: **true** の場合、Camel は JDBC 接続の **autoCommit** を **false** に設定し、ステートメントを実行した後に変更をコミットし、最後に接続の **autoCommit** フラグをリセットします。JDBC 接続が **autoCommit** フラグのリセットをサポートしていない場合は、これを **false** に設定します。XA トランザクションと併用する場合は、トランザクションマネージャーがこの tx のコミットを担当するように、ほとんどの場合 false に設定する必要があります。
allowNamedParameters	**true**	Camel 2.12: クエリーで名前付きパラメーターの使用を許可するかどうか。
prepareStatementStrategy		Camel 2.12: プラグインがカスタムの **org.apache.camel.component.jdbc.JdbcPrepareStatementStrategy** を使用して、クエリーおよび準備済みステートメントの準備を制御できます。

useHeadersAsParameters	**false**	Camel 2.12: 名前付きパラメーターで **prepareStatementStrategy** を使用するには、このオプションを **true** に設定します。これにより、名前付きプレースホルダーでクエリーを定義し、クエリープレースホルダーの動的な値でヘッダーを使用できます。
outputType	**SelectList**	**Camel 2.12.1:** producer to SelectList as List of Map, or SelectOne as single Java object in the following way: a）クエリーに列が 1 つしかない場合は、その JDBC Column オブジェクトが返されます。SELECT COUNT (*) FROM PROJECT (SELECT COUNT (*) FROM PROJECT など)-Long オブジェクトを返します。b （クエリーに複数の列がある場合） その結果の Map を返します。c）outputClass が設定されている場合には、列名に一致するすべてのセッターを呼び出すことで、クエリー結果を Java Bean オブジェクトに変換します。クラスにインスタンスを作成するデフォルトのコンストラクターがあるとします。**Camel 2.14** 以降では、**SelectList** もサポートされます。d）クエリーによって複数の行が発生した場合、一意でない結果例外が出力されます。 **Camel 2.14.0:** 新しい **StreamList** 出力タイプ値。**Iterator<Map<String, Object>>** を使用してクエリーの結果をストリーミングします。Splitter EIP と併用できます。
outputClass	**null**	**Camel 2.12.1:** outputType=SelectOne の場合に変換として使用する完全なパッケージおよびクラス名を指定します。**Camel 2.14** 以降では、**SelectList** もサポートされます。

beanRowMapper		Camel 2.12.1: **outputClass** の使用時にカスタム **org.apache.camel.component.jdbc.BeanRowMapper** を使用するには、以下を行います。デフォルトの実装では、行名が低くなり、アンダースコアやダッシュをスキップします。たとえば、**CUST_ID は cust Id としてマッピングされます。**
useGetBytesForBlob	**false**	Camel 2.16: BLOB 列を文字列データではなくバイトとして読み取る。これは、BLOB 列をバイトとして読み取る必要がある Oracle などの特定のデータベースで必要になる場合があります。

結果

デフォルトでは、結果は OUT ボディーに ArrayList<HashMap<String, Object>> として返されます。List オブジェクトには行のリストが含まれ、Map オブジェクトには、String キーを持つ各行が列名として含まれます。



注記

このコンポーネントは ResultSetMetaData を取得し、列名を Map のキーとして返すことができます。

メッセージヘッダー

ヘッダー	説明
CamelJdbcRowCount	クエリーが **SELECT** の場合、行数はこの OUT ヘッダーで返されます。
CamelJdbcUpdateCount	クエリーが **UPDATE** の場合、更新数はこの OUT ヘッダーで返されます。
CamelGeneratedKeysRows	Camel 2.10: 生成された kets が含まれる行。
CamelGeneratedKeysRowCount	Camel 2.10: 生成されたキーが含まれるヘッダーの行数。
CamelJdbcColumnNames	Camel 2.11.1: ResultSet からの列名(**java.util.Set** 型)

CamelJdbcParametes	Camel 2.12: **useHeadersAsParameters** が有効な場合に使用されるヘッダーを持つ **java.util.Map**。

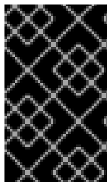
生成されるキー

Camel 2.10 以降で利用可能

SQL INSERT を使用してデータを挿入すると、----------|----- は自動生成されたキーをサポートする可能性があります。JDBC プロデューサーに対して、ヘッダーで生成されたキーを返すように指示できます。これには、ヘッダー CamelRetrieveGeneratedKeys=true を設定します。次に、生成された鍵が上記の表に記載されているキーが含まれるヘッダーとして提供されます。

```
from("direct:insert")
  .setHeader("CamelRetrieveGeneratedKeys", constant(true))
  .setBody(constant("INSERT INTO projects (project) VALUES ('Camel')"))
  .to("jdbc:myDataSource");
```

詳細は、この ユニットテスト を参照してください。



重要

生成された鍵を使用することは、名前付きパラメーターと併用できません。

名前付きパラメーターの使用

Camel 2.12 以降で利用可能

以下のルートでは、Projects テーブルからすべてのプロジェクトを取得します。SQL クエリーには、 : ?lic と :?min の 2 つの名前付きパラメーターがあることに注意してください。Camel はメッセージヘッダーからこれらのパラメーターを検索します。上記の例では、名前付きパラメーターに定数値で 2 つのヘッダーを設定することに注意してください。

```
from("direct:projects")
   .setHeader("lic", constant("ASF"))
   .setHeader("min", constant(123))
```

```
.setBody(constant("select * from projects where license = :?lic and id > :?min order by id"))
.to("jdbc:myDataSource?useHeadersAsParameters=true")
```

ヘッダー値を java.util.Map に保存し、キー CamelJdbcParameters を使用してヘッダーにマッピングすることもできます。

サンプル

以下の例では、customers テーブルから行を取得します。

最初に、データソースを testdb として Apache Camel レジストリーに登録します。

```
JndiRegistry reg = super.createRegistry();
reg.bind("testdb", db);
return reg;
```

次に、JDBC コンポーネントにルーティングするルートを設定し、SQL が実行されます。前の手順でバインドされた testdb データソースを参照する方法に注意してください。

```
// lets add simple route
public void configure() throws Exception {
    from("direct:hello").to("jdbc:testdb?readSize=100");
}
```

または、以下のように Spring で DataSource を作成することもできます。

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
  <!-- trigger every second -->
     <from uri="timer://kickoff?period=1s"/>
     <setBody>
       <constant>select * from customer</constant>
     </setBody>
     <to uri="jdbc:testdb"/>
     <to uri="mock:result"/>
  </route>
</camelContext>
<!-- Just add a demo to show how to bind a date source for camel in Spring-->
<jdbc:embedded-database id="testdb" type="DERBY">
 <jdbc:script location="classpath:sql/init.sql"/>
</jdbc:embedded-database>
```

エンドポイントを作成し、SQL クエリーを IN メッセージのボディーに追加してから、エクスチェン

ジを送信します。クエリーの結果は OUT ボディーで返されます。

```
// first we create our exchange using the endpoint
Endpoint endpoint = context.getEndpoint("direct:hello");
Exchange exchange = endpoint.createExchange();
// then we set the SQL on the in body
exchange.getIn().setBody("select * from customer order by ID");

// now we send the exchange to the endpoint, and receives the response from Camel
Exchange out = template.send(endpoint, exchange);

// assertions of the response
assertNotNull(out);
assertNotNull(out.getOut());
List<Map<String, Object>> data = out.getOut().getBody(List.class);
assertNotNull(data);
assertEquals(3, data.size());
Map<String, Object> row = data.get(0);
assertEquals("cust1", row.get("ID"));
assertEquals("jstrachan", row.get("NAME"));
row = data.get(1);
assertEquals("cust2", row.get("ID"));
assertEquals("nsandhu", row.get("NAME"));
```

ResultSet 全体ではなく、一度に 1 行ずつ作業する場合は、以下のような Splitter EIP を使用する必要があります。

```
from("direct:hello")
// here we split the data from the testdb into new messages one by one
// so the mock endpoint will receive a message per row in the table
// the StreamList option allows to stream the result of the query without creating a List of rows
// and notice we also enable streaming mode on the splitter
.to("jdbc:testdb?outputType=StreamList")
  .split(body()).streaming()
  .to("mock:result");
```

サンプル：データベースを毎分ポーリングする

JDBC コンポーネントを使用してデータベースをポーリングする場合は、Timer や Quartz などのポーリングスケジューラーと組み合わせる必要があります。以下の例では、60 秒ごとにデータベースからデータを取得します。

```
from("timer://foo?period=60000").setBody(constant("select * from
customer")).to("jdbc:testdb").to("activemq:queue:customers");
```

例：データソース間のデータの移動

一般的なユースケースは、データをクエリーし、処理して別のデータソース(ETL 操作)に移動することです。以下の例では、1 時間ごとに新しい顧客レコードを取得し、それらをフィルターリング/変換し、宛先テーブルに移動します。

```
from("timer://MoveNewCustomersEveryHour?period=3600000")
    .setBody(constant("select * from customer where create_time > (sysdate-1/24)"))
    .to("jdbc:testdb")
    .split(body())
  .process(new MyCustomerProcessor()) //filter/transform results as needed
        .setBody(simple("insert into processed_customer
values('${body[ID]}','${body[NAME]}')"))
        .to("jdbc:testdb");
```

その他の参考資料

- SQL

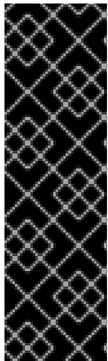
第80章 JETTY

JETTY コンポーネント

警告

プロデューサーは非推奨になりました。使用しないでください。jetty をコンシューマーとしてのみ使用することが推奨されます（例：jetty）。

jetty コンポーネントは、HTTP 要求を使用し、生成するための HTTP ベースの エンドポイント を提供します。つまり、Jetty コンポーネントは単純な Web サーバーとして動作します。Jetty は http クライアントとしても使用でき、Camel でプロデューサーとして使用することもできます。



ストリーム

Jetty はストリームベースであり、受信する入力がストリームとして Camel に送信されます。つまり、ストリームのコンテンツを 1 度 だけ読み取ることができます。メッセージボディーが空である、または Exchange.HTTP_RESPONSE_CODE データを複数回アクセスする必要がある場合（たとえば、マルチキャストまたは再配信エラー処理）、Stream Caching を使用するか、メッセージボディーを複数回再読み取りできる String に変換する必要があります。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jetty</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
jetty:http://hostname[:port][/resourceUri][?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
sessionSupport	**false**	Jetty のサーバー側でセッションマネージャーを有効にするかどうかを指定します。
httpClient.XXX	**null**	Jetty の HttpClient の設定。たとえば、**httpClient.idleTimeout=30000** を設定すると、アイドルタイムアウトは 30 秒に設定されます。
httpClient	**null**	このエンドポイントによって作成されたすべてのプロデューサーに共有 **org.eclipse.jetty.client.HttpClient** を使用します。このオプションは、特別な状況でのみ使用してください。
httpClientMinThreads	**null**	Camel 2.11: ***プロデューサーのみ***: **HttpClient** スレッドプールの最小スレッド数に値を設定します。この設定は、コンポーネントレベルに設定された設定を上書きします。最小サイズと最大サイズの両方を設定する必要があることに注意してください。設定しない場合は、Jetty のスレッドプールで使用される min 8 スレッドにデフォルト設定されます。
httpClientMaxThreads	**null**	Camel 2.11: ***プロデューサーのみ***: **HttpClient** スレッドプールの最大スレッド数に値を設定します。この設定は、コンポーネントレベルに設定された設定を上書きします。最小サイズと最大サイズの両方を設定する必要があることに注意してください。設定しない場合は、Jetty のスレッドプールで使用される最大 16 スレッドにデフォルト設定されます。

httpBindingRef	**null**	レジストリーの **org.apache.camel.component.http.HttpBinding** への参照。**HttpBinding** を使用して、コンシューマーに応答を書き込む方法をカスタマイズできます。
jettyHttpBindingRef	**null**	Camel 2.6.0+: レジストリーの **org.apache.camel.component.jetty.JettyHttpBinding** への参照。**JettyHttpBinding** を使用すると、プロデューサーに対して応答の作成方法をカスタマイズできます。
matchOnUriPrefix	**false**	完全に一致するものが見つからない場合に、**CamelServlet** が URI 接頭辞と一致することでターゲットコンシューマーの検索を試みるかどうか。How do I let Jetty match wildcards を参照してください。
handlers	**null**	レジストリー(Spring **ApplicationContext**など)の **org.mortbay.jetty.Handler** インスタンスのコンマ区切りの一覧を指定します。これらのハンドラーは Jetty サーブレットコンテキストに追加されます（セキュリティーを追加するためなど）。
chunked	**true**	Camel 2.2: このオプションが false の場合、Jetty サーブレットが HTTP ストリーミングを無効にし、応答に content-length ヘッダーを設定します。
enableJmx	**false**	Camel 2.3: このオプションが true の場合、このエンドポイントに対して Jetty JMX サポートが有効になります。

disableStreamCache	**false**	**Camel 2.3:** Jetty からの raw 入力ストリームがキャッシュされているかどうかを判断します(Camel はストリームをファイル、ストリームキャッシュ)キャッシュにストリームします。http://camel.apache.org/stream-caching.htmlデフォルトでは、Camel は Jetty 入力ストリームをキャッシュして複数回読み取りし、Camel がストリームからすべてのデータを取得できるようにします。ただし、ファイルや他の永続ストアに直接ストリーミングするなど、raw ストリームにアクセスする必要がある場合などにこのオプションを **true** に設定できます。DefaultHttpBinding は、要求入力ストリームをストリームキャッシュにコピーし、このオプションが **false** の場合、ストリームを複数回読み取るようにメッセージボディーに配置します。Jetty を使用してエンドポイントをブリッジ/プロキシーする場合は、メッセージペイロードを複数回読み取る必要がない場合は、このオプションを有効にしてパフォーマンスを向上することを検討してください。
throwExceptionOnFailure	**true**	リモートサーバーからの応答が失敗した場合に **HttpOperationFailedException** を出力することを無効にするオプション。これにより、HTTP ステータスコードに関するすべての応答を取得できます。
transferException	**false**	**Camel 2.6:** 有効で、エクスチェンジ がコンシューマー側で処理に失敗した場合、発生した例外が応答で **application/x-java-serialized-object** コンテンツタイプとしてシリアライズされた場合は、以下を行います。プロデューサー側では、例外がデシリアライズされ、**HttpOperationFailedException** ではなくそのまま出力されます。原因となった例外はシリアライズする必要があります。

bridgeEndpoint	**false**	> **Camel 2.1:** オプションが true の場合、HttpProducer は Exchange.HTTP_URI ヘッダーを無視し、リクエストにエンドポイントの URI を使用します。また、**throwExceptionOnFailure** を false に設定して、HttpProducer がすべての障害応答を返信するようにすることもできます。**Camel 2.3:** オプションが true の場合、content-encoding が gzip の場合、HttpProducer および CamelServlet は gzip 処理をスキップします。ブリッジ時に最適化するために **disableStreamCache** を true に設定することも検討してください。
enableMultipartFilter	**true**	**Camel 2.5:** Jetty **org.eclipse.jetty.servlets.MultiPartFilter** が有効かどうか。マルチパートリクエストもプロキシー/ブリッジされるように、エンドポイントをブリッジする場合にはこの値を **false** に設定する必要があります。
multipartFilterRef	**null**	**Camel 2.6:** カスタムの multipart フィルターの使用を許可します。注記： **multipartFilterRef** を設定すると、**enableMultipartFilter** の値が **true** に強制されます。
filterInit.xxx	**null**	**Camel 2.17:** フィルターの設定 **InitParameters**たとえば、**filterInit.parameter=value** を設定すると、フィルター **init** メソッドを呼び出す際に、パラメーターを使用することができます。
filtersRef	**null**	**Camel 2.9:** リストに格納され、レジストリーで検索できるカスタムフィルターを使用できます。

continuationTimeout	null	Camel 2.6: Jetty をコンシューマー（サーバー）として使用する場合にタイムアウトをミリ秒単位で設定できます。デフォルトでは Jetty は 30000 を使用します。<= **0 の値を使用すると有効期限** が切れることはありません。タイムアウトが発生すると、リクエストは期限切れになり、Jetty は http エラー 503 をクライアントに返します。このオプションは、非同期ルーティングエンジンで Jetty を使用する場合のみ使用されます。
useContinuation	true	Camel 2.6: Jetty サーバーの Jetty 継続を 使用するかどうか。
sslContextParametersRef	null	**非推奨 : Camel** 2.8: レジストリーの **org.apache.camel.util.jsse.SSLContextParameters** への参照。この参照は、コンポーネントレベルで設定済みの SSLContextParameters を上書きします。Security Guide および「JSSE 設定ユーティリティーの使用」 の Configuring Transport Security for Camel Components を参照して ください。
sslContextParameters	null	Camel 2.17: レジストリーの **org.apache.camel.util.jsse.SSLContextParameters** への参照。この参照は、コンポーネントレベルで設定済みの SSLContextParameters を上書きします。Security Guide および「JSSE 設定ユーティリティーの使用」 の Configuring Transport Security for Camel Components を参照して ください。
traceEnabled	false	この Jetty コンシューマーに対して HTTP TRACE を有効にするかどうかを指定します。デフォルトでは、TRACE はオフになっています。
optionsEnabled	false	Camel 2.17: この Jetty コンシューマーに対して HTTP OPTIONS を有効にするかどうかを指定します。デフォルトでは、OPTIONS はオフになっています。

headerFilterStrategy	**null**	Camel 2.11: レジストリー の **org.apache.camel.spi.HeaderFilterStrategy** のインスタンスへの 参照。これは、新しい create HttpJettyEndpoint にカスタム headerFilterStrategy を適用するために使用されます。
httpMethodRestrict	**null**	**Camel 2.11: コンシューマーのみ**: **GET/POST/PUT** など、**HttpMethod** が一致する場合にのみ消費を許可するために使用されます。**Camel 2.15 以降** では、複数のメソッドをコンマで区切って指定できます。
responseBufferSize	**null**	Camel 2.12: **javax.servlet.ServletResponse** でカスタムバッファーサイズを使用するには、以下を行います。
proxyHost	**null**	**Camel 2.11: Jetty クライアントによって使用される http プロキシーホスト URL のみ。**
proxyPort	**null**	**Camel 2.11: Jetty クライアントによって使用される http プロキシーポートのみプロデューサー。**
sendServerVersion	**true**	**Camel 2.13:** オプションが true の場合、jetty は要求を送信するクライアントに jetty バージョン情報のあるサーバーヘッダーを送信します。なお、他の camel-jetty エンドポイントが同じポートを共有していないことを確認してください。それ以外の場合は、このオプションが期待どおりに機能しない可能性があります。
sendDateHeader	**false**	**Camel 2.14:** オプションが true の場合、jetty サーバーはリクエストを送信するクライアントに日付ヘッダーを送信します。なお、他の camel-jetty エンドポイントが同じポートを共有していないことを確認してください。それ以外の場合は、このオプションが期待どおりに機能しない可能性があります。
enableCORS	**false**	**Camel 2.15:** オプションが **true** の場合、Jetty サーバーは、すぐに使用できる CORS をサポートする **CrossOriginFilter** を設定します。

okStatusCodeRange	200-299	Camel 2.16: プロデューサーのみ。正常な応答と見なされるステータスコード。値は含まれます。範囲は、構文 **from-to** を使用して定義する必要があります。

メッセージヘッダー

Camel は HTTP コンポーネントと同じメッセージヘッダーを使用します。Camel 2.2 では、(Exchange.HTTP_CHUNKED,CamelHttpChunked)ヘッダーを使用して camel-jetty コンシューマーでチェンドエンコーディングをオンまたはオフにします。

Camel は すべて の request.parameter および request.headers も設定します。たとえば、URL http://myserver/myserver?orderid=123 を持つクライアントリクエストの場合、エクスチェンジには orderid という名前のヘッダー（値が 123）が含まれます。

Camel 2.2.0 以降では、Get Method だけでなく、他の HTTP メソッドからもメッセージヘッダーから request.parameter を取得できます。

使用方法

Jetty コンポーネントは、コンシューマーエンドポイントとプロデューサーエンドポイントの両方をサポートします。他の HTTP エンドポイントへ生成するもう 1 つのオプションは、HTTP コンポーネントを使用することです。

コンポーネントオプション

JettyHttpComponent は、以下のオプションを提供します。

名前	デフォルト値	説明
enableJmx	false	Camel 2.3: このオプションが true の場合、このエンドポイントに対して Jetty JMX サポートが有効になります。
sslKeyPassword	null	**コンシューマーのみ**: SSL 使用時のキーストアのパスワード。
sslPassword	null	**コンシューマーのみ**: SSL を使用する場合のパスワード。

sslKeystore	**null**	**コンシューマーのみ**: キーストアへのパス。
minThreads	**null**	**Camel 2 .5 コンシューマーのみ**: サーバースレッドプールの最小スレッド数に値を設定します。
maxThreads	**null**	**Camel 2 .5 コンシューマーのみ**: サーバースレッドプールの最大スレッド数に値を設定します。
threadPool	**null**	**Camel 2 .5 コンシューマーのみ**: サーバーにカスタムスレッドプールを使用します。
sslSocketConnectors	**null**	**Camel 2.3 Consumer のみ：** ポート番号固有の SSL コネクターごとに含まれるマップ。詳細は、**SSL サポート** のセクションを参照してください。
socketConnectors	**null**	**Camel 2. 5 コンシューマーのみ：** ポート番号固有の HTTP コネクターごとに含まれるマップ。**sslSocketConnectors** と同じ原則を使用するため、**SSL サポート** のセクションを参照してください。
sslSocketConnectorProperties	**null**	**Camel 2. 5 コンシューマーのみ**一般的な SSL コネクタープロパティーが含まれるマップ。
socketConnectorProperties	**null**	**Camel 2. 5 コンシューマーのみ**一般的な HTTP コネクタープロパティーが含まれるマップ。**sslSocketConnectorProperties** と同じ principle を使用します。
httpClient	**null**	**非推奨: プロデューサーのみ**:jetty プロデューサーでカスタム **HttpClient** を使用します。このオプションは Camel 2.11 以降から削除されます。代わりに、エンドポイントに オプションを設定できます。

httpClientMinThreads	**null**	**プロデューサーのみ**: **HttpClient** スレッドプールの最小スレッド数に値を設定します。最小サイズと最大サイズの両方を設定する必要があることに注意してください。
httpClientMaxThreads	**null**	**プロデューサーのみ**: **HttpClient** スレッドプールの最大スレッド数に値を設定します。最小サイズと最大サイズの両方を設定する必要があることに注意してください。
httpClientThreadPool	**null**	**非推奨: プロデューサーのみ**: クライアントにカスタムスレッドプールを使用します。このオプションは Camel 2.11 以降から削除されます。
sslContextParameters	**null**	**Camel 2.8:** コンポーネントレベルでカスタム SSL/TLS 設定オプションを設定するには、以下を実行します。詳細は、「JSSE 設定ユーティリティーの使用」 を参照してください。
requestBufferSize	**null**	**Camel 2.11.** 2: Jetty コネクターでリクエストバッファーサイズのカスタム値を設定できます。
requestHeaderSize	**null**	**Camel 2.11.** 2: Jetty コネクターでリクエストヘッダーサイズのカスタム値を設定します。
responseBufferSize	**null**	**Camel 2.11.** 2: Jetty コネクターで応答バッファーサイズのカスタム値を設定できます。
responseHeaderSize	**null**	**Camel 2.11.** 2: Jetty コネクターで応答ヘッダーサイズのカスタム値を設定します。
proxyHost	**null**	**Camel 2.12.2/2.11.3** http プロキシーを使用します。
proxyPort	**null**	**Camel 2.12.2/2.11.3:** http プロキシーを使用するには、以下を実行します。
errorHandler	**null**	**Camel 2.15:** このオプションは、Jetty サーバーが使用する ErrorHandler を設定するために使用されます。

allowJavaSerializedObject	**false**	Camel 2.16.1/2.15.5: **リクエストが context-type=application/x-java-serialized-object を使用している場合に Java のシリアル化を許可するかどうか。**これはデフォルトでオフになっています。**警告**: このオプションを有効にすると、Java はリクエストから受信したデータを Java にデシリアライズし、セキュリティーリスクとなる可能性があることに注意してください。

プロデューサーの例

警告

プロデューサーは非推奨になりました。使用しないでください。jetty をコンシューマーとしてのみ使用することが推奨されます（例：jetty）。

以下は、HTTP リクエストを既存の HTTP エンドポイントに送信する方法の基本的な例です。

Java DSL で

```
from("direct:start").to("jetty://http://www.google.com");
```

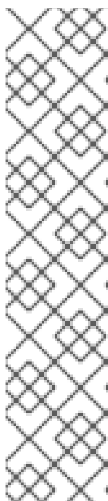
Spring XML で または を使用します。

```
<route>
   <from uri="direct:start"/>
   <to uri="jetty://http://www.google.com"/>
<route>
```

コンシューマーの例

以下の例では、 http://localhost:8080/myapp/myservice で HTTP サービスを公開するルートを定義します。

```
from("jetty:http://localhost:{{port}}/myapp/myservice").process(new MyBookService());
```



LOCALHOST の使用

URL で localhost を指定すると、Camel はローカルの TCP/IP ネットワークインターフェイスでのみエンドポイントを公開するため、操作するマシンからアクセスすることはできません。

特定のネットワークインターフェイスで Jetty エンドポイントを公開する必要がある場合は、このインターフェイスの数値の IP アドレスをホストとして使用する必要があります。すべてのネットワークインターフェイスで Jetty エンドポイントを公開する必要がある場合は、0.0.0.0 アドレスを使用する必要があります。

ヒント

URI 接頭辞全体をリッスンするに は、Jetty がワイルドカードと一致させる方法 を参照してください。

ヒント

実際には HTTP によってルートを公開し、すでにサーブレットがある場合は、代わりに Servlet Transport を参照する必要があります。

このビジネスロジックは MyBookService クラスに実装され、HTTP リクエストの内容にアクセスし、応答を返します。注記： コードがユニットテストの一部であるため、この例では assert 呼び出しが表示されます。

```
public class MyBookService implements Processor {
    public void process(Exchange exchange) throws Exception {
        // just get the body as a string
        String body = exchange.getIn().getBody(String.class);

        // we have access to the HttpServletRequest here and we can grab it if we need it
        HttpServletRequest req = exchange.getIn().getBody(HttpServletRequest.class);
        assertNotNull(req);

        // for unit testing
        assertEquals("bookid=123", body);

        // send a html response
        exchange.getOut().setBody("<html><body>Book 123 is Camel in Action</body></html>");
    }
}
```

以下の例は、URI パラメーター 1 つを含むすべての要求をエンドポイント mock: one にルーティングし、他のすべてのリクエストを mock:other にルーティングするコンテンツベースのルートを示しています。

```
from("jetty:" + serverUri)
    .choice()
    .when().simple("${header.one}").to("mock:one")
    .otherwise()
    .to("mock:other");
```

そのため、クライアントが HTTP リクエスト http://serverUri?one=hello を送信すると、Jetty コンポーネントは HTTP リクエストパラメーターをエクスチェンジの in.header にコピーします。次に、Simple 言語を使用して、このヘッダーを含むエクスチェンジを特定のエンドポイントへルーティングし、他のすべてのエクスチェンジを別のエンドポイントにルーティングすることができます。Simple\-\-- など、 El や OGNL\--we よりも強力な言語を使用している場合は、パラメーター値をテストし、ヘッダー値に基づいてルーティングを行うこともできます。

セッションサポート

セッションサポートオプション sessionSupport を使用すると、HttpSession オブジェクトを有効にし、エクスチェンジの処理中にセッションオブジェクトにアクセスできます。たとえば、以下のルートはセッションを有効にします。

```
<route>
    <from uri="jetty:http://0.0.0.0/myapp/myservice/?sessionSupport=true"/>
    <processRef ref="myCode"/>
<route>
```

myCode プロセッサー は、Spring bean 要素でインスタンス化できます。

```
<bean id="myCode"class="com.mycompany.MyCodeProcessor"/>
```

プロセッサーの実装は、以下のように HttpSession にアクセスできます。

```
public void process(Exchange exchange) throws Exception {
    HttpSession session = exchange.getIn(HttpMessage.class).getRequest().getSession();
    ...
}
```

JSSE 設定ユーティリティーの使用

Camel 2.8 以降、Jetty コンポーネントは Camel JSSE 設定ユーティリティーを介して SSL/TLS 設

定をサポートします。このユーティリティーは、作成する必要のあるコンポーネント固有のコードの量を大幅に減らし、エンドポイントおよびコンポーネントレベルで設定可能です。 たとえば、Security GuideのConfiguring Transport Security for Camel Componentsの章を参照してください。Security Guide のSecuring the Camel Jetty Component の章を参照してください。

エンドポイントの SPRING DSL ベースの設定

```
...
  <camel:sslContextParameters
      id="sslContextParameters">
    <camel:keyManagers
        keyPassword="keyPassword">
      <camel:keyStore
          resource="/users/home/server/keystore.jks"
          password="keystorePassword"/>
    </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="jetty:https://127.0.0.1/mail/?sslContextParametersRef=sslContextParameters"/>
...
```

JETTY の直接設定

Jetty は、追加設定なしで SSL サポートを提供します。Jetty が SSL モードで実行できるようにするには、https:// prefix--- で URI をフォーマットします。以下に例を示します。

```
<from uri="jetty:https://0.0.0.0/myapp/myservice/"/>
```

また、jetty は、正しい SSL 証明書を読み込むためにキーストアをロードする場所と、使用するパスワードを把握しておく必要があります。以下の JVM システムプロパティーを設定します。

Camel 2.2 まで

- jetty.ssl.keystore は、Jetty サーバー自身の X.509 証明書をキー エントリー に含める Java キー ストアファイルの場所を指定します。キーエントリーは、X.509 証明書（実際には 公開鍵）と関連する秘密鍵を保存します。

- jetty.ssl.password キーストアファイルへのアクセスに必要なストアパスワード（これはキー ストア コマンドの \-storepass オプションに指定されたパスワードと同じです）。

- jetty.ssl.keypassword キーストアの証明書のキーエントリーにアクセスするために使用されるキーパスワード（これはキー ストア コマンドの \-keypass オプションに指定されたパス

ワードと同じです）。

Camel 2.3 以降

- org.eclipse.jetty.ssl.keystore は、キーエントリー に Jetty サーバー自体の X.509 証明書が含まれる Java キー ストアファイルの場所を指定します。キーエントリーは、X.509 証明書（実際には 公開鍵）と関連する秘密鍵を保存します。

- org.eclipse.jetty.ssl.password キーストアファイルへのアクセスに必要なストアパスワード（これはキー ストア コマンドの \-storepass オプションに指定されたパスワードと同じです）。

- org.eclipse.jetty.ssl.keypassword キーストアの証明書のキーエントリーにアクセスするために使用されるキーパスワード（これはキー ストア コマンドの \-keypass オプションに指定されたパスワードと同じです）。

Jetty エンドポイントに SSL を設定する方法は、Jetty サイト(http://docs.codehaus.org/display/JETTY/How+to+configure+SSL)を参照してください。

一部の SSL プロパティーは Camel によって直接公開されませんが、Camel は基礎となる SslSocketConnector を公開します。これにより、クライアント証明書を必要とする相互認証の needClientAuth や、クライアントが証明書を必要としない相互認証には wantClientAuth などのプロパティーを設定できます。さまざまな Camel バージョンには若干の違いがあります。

Camel 2.2 まで

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
    <property name="sslSocketConnectors">
        <map>
            <entry key="8043">
                <bean class="org.mortbay.jetty.security.SslSocketConnector">
                    <property name="password"value="..."/>
                    <property name="keyPassword"value="..."/>
                    <property name="keystore"value="..."/>
                    <property name="needClientAuth"value="..."/>
                    <property name="truststore"value="..."/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

Camel 2.3, 2.4

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
    <property name="sslSocketConnectors">
        <map>
            <entry key="8043">
                <bean class="org.eclipse.jetty.server.ssl.SslSocketConnector">
                    <property name="password"value="..."/>
                    <property name="keyPassword"value="..."/>
                    <property name="keystore"value="..."/>
                    <property name="needClientAuth"value="..."/>
                    <property name="truststore"value="..."/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

** from Camel 2.5 we switch to use SslSelectChannelConnector **

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
    <property name="sslSocketConnectors">
        <map>
            <entry key="8043">
                <bean class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
                    <property name="password"value="..."/>
                    <property name="keyPassword"value="..."/>
                    <property name="keystore"value="..."/>
                    <property name="needClientAuth"value="..."/>
                    <property name="truststore"value="..."/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

上記のマップでキーとして使用する値は、Jetty がリッスンするように設定するポートです。

一般的な SSL プロパティーの設定

Camel 2.5 で利用可能

（上記のように）ポート番号固有の SSL ソケットコネクターではなく、すべての SSL ソケットコネクターに適用される一般的なプロパティーを設定できるようになりました（ポート番号をエントリーとして追加しない）。

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
   <property name="sslSocketConnectorProperties">
      <map>
         <entry key="password"value="..."/>
         <entry key="keyPassword"value="..."/>
         <entry key="keystore"value="..."/>
         <entry key="needClientAuth"value="..."/>
         <entry key="truststore"value="..."/>
      </map>
   </property>
</bean>
```

X509CERTIFICATE への参照を取得する方法

Jetty は、以下のようにコードからアクセスできる HttpServletRequest に証明書への参照を保存します。

```
HttpServletRequest req = exchange.getIn().getBody(HttpServletRequest.class);
X509Certificate cert = (X509Certificate)
req.getAttribute("javax.servlet.request.X509Certificate")
```

一般的な HTTP プロパティーの設定

Camel 2.5 で利用可能

（上記のように）ポート番号固有の HTTP ソケットコネクターではなく、すべての HTTP ソケットコネクターに適用される一般的なプロパティーを設定できるようになりました（ポート番号をエントリーとして追加しない）。

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
   <property name="socketConnectorProperties">
      <map>
         <entry key="acceptors" value="4"/>
         <entry key="maxIdleTime" value="300000"/>
      </map>
   </property>
</bean>
```

OBTAINING X-FORWARDED-FOR HEADER WITH HTTPSERVLETREQUEST.GETREMOTEADDR()

HTTP 要求が Apache サーバーで処理され、mod_proxy で Jetty に転送された場合、元のクライアント IP アドレスは X-Forwarded-For ヘッダーに、HttpServletRequest.getRemoteAddr() は Apache プロキシーのアドレスを返します。

Jetty には転送されたプロパティーがあり、X-Forwarded-For から値を取得し、HttpServletRequest remoteAddr プロパティーに配置します。このプロパティーはエンドポイント設定から直接は利用できませんが、socketConnectors プロパティーを使用して簡単に追加できます。

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
  <property name="socketConnectors">
    <map>
      <entry key="8080">
        <bean class="org.eclipse.jetty.server.nio.SelectChannelConnector">
          <property name="forwarded" value="true"/>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

これは、既存の Apache サーバーがドメインの TLS 接続を処理し、それらを内部でアプリケーションサーバーにプロキシーする場合に便利です。

HTTP ステータスコードを返すデフォルトの動作

HTTP ステータスコードのデフォルト動作は、応答の書き込み方法を処理し、HTTP ステータスコードも設定する org.apache.camel.component.http.DefaultHttpBinding クラスによって定義されます。

エクスチェンジが正常に処理されると、200 HTTP ステータスコードが返されます。例外でエクスチェンジが失敗すると、500 HTTP ステータスコードが返され、スタックトレースがボディーで返されます。返す HTTP ステータスコードを指定する場合は、OUT メッセージの Exchange.HTTP_RESPONSE_CODE ヘッダーにコードを設定します。

CUSTOMIZING HTTPBINDING

デフォルトでは、Camel は org.apache.camel.component.http.DefaultHttpBinding を使用して応答の作成方法を処理します。必要に応じて、独自の HttpBinding クラスを実装するか、DefaultHttpBinding を拡張し、適切なメソッドを上書きすることで、この動作をカスタマイズできます。

以下の例は、例外が返される方法を変更するために DefaultHttpBinding をカスタマイズする方法を示しています。

```
public class MyHttpBinding extends DefaultHttpBinding {

    @Override
```

```
    public void doWriteExceptionResponse(Throwable exception, HttpServletResponse response) throws IOException {
        // we override the doWriteExceptionResponse as we only want to alter the binding how exceptions is
        // written back to the client.

        // we just return HTTP 200 so the client thinks its okay
        response.setStatus(200);
        // and we return this fixed text
        response.getWriter().write("Something went wrong but we dont care");
    }
}
```

次に、バインディングのインスタンスを作成し、以下のように Spring レジストリーに登録することができます。

```
<bean id="mybinding"class="com.mycompany.MyHttpBinding"/>
```

次に、ルートを定義するときにこのバインディングを参照できます。

```
<route><from uri="jetty:http://0.0.0.0:8080/myapp/myservice?httpBindingRef=mybinding"/><to uri="bean:doSomething"/></route>
```

JETTY ハンドラーおよびセキュリティー設定

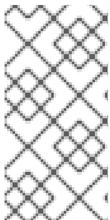
エンドポイントで Jetty ハンドラーの一覧を設定できます。これは、高度な Jetty セキュリティー機能を有効にするのに役立ちます。これらのハンドラーは、以下のように Spring XML で設定されます。

```
<-- Jetty Security handling -->
<bean id="userRealm" class="org.mortbay.jetty.plus.jaas.JAASUserRealm">
    <property name="name" value="tracker-users"/>
    <property name="loginModuleName" value="ldaploginmodule"/>
</bean>

<bean id="constraint" class="org.mortbay.jetty.security.Constraint">
    <property name="name" value="BASIC"/>
    <property name="roles" value="tracker-users"/>
    <property name="authenticate" value="true"/>
</bean>

<bean id="constraintMapping" class="org.mortbay.jetty.security.ConstraintMapping">
    <property name="constraint" ref="constraint"/>
    <property name="pathSpec" value="/*"/>
</bean>

<bean id="securityHandler" class="org.mortbay.jetty.security.SecurityHandler">
```

```
    <property name="userRealm" ref="userRealm"/>
    <property name="constraintMappings" ref="constraintMapping"/>
</bean>
```

Camel 2.3 以降では、 以下のように Jetty ハンドラーのリストを設定できます。

```
<-- Jetty Security handling -->
<bean id="constraint" class="org.eclipse.jetty.util.security.Constraint">
    <property name="name" value="BASIC"/>
    <property name="roles" value="tracker-users"/>
    <property name="authenticate" value="true"/>
</bean>

<bean id="constraintMapping" class="org.eclipse.jetty.security.ConstraintMapping">
    <property name="constraint" ref="constraint"/>
    <property name="pathSpec" value="/*"/>
</bean>

<bean id="securityHandler" class="org.eclipse.jetty.security.ConstraintSecurityHandler">
    <property name="authenticator">
        <bean class="org.eclipse.jetty.security.authentication.BasicAuthenticator"/>
    </property>
    <property name="constraintMappings">
        <list>
            <ref bean="constraintMapping"/>
        </list>
    </property>
</bean>
```



注記

Blueprint XML 構文(Apache Karaf コンテナー)では、ref 要素を <ref component-id="constraintMapping"/> として指定する必要があります。

次に、エンドポイントを以下のように定義できます。

```
from("jetty:http://0.0.0.0:9080/myservice?handlers=securityHandler")
```

より多くのハンドラーが必要な場合は、ハンドラー オプションを Bean ID のコンマ区切りリストに設定します。

このサンプルを Apache Karaf コンテナーにデプロイする場合、以下の Java パッケージをインポートするように Bundle ヘッダーを設定する必要があります。

```
org.eclipse.jetty.security
org.eclipse.jetty.util.security
org.eclipse.jetty.security.authentication
```

たとえば、Maven POM では、felix-maven-plugin プラグインを以下の Import-Package 要素で設定できます。

```
<Import-Package>
    org.eclipse.jetty.security,
    org.eclipse.jetty.util.security,
    org.eclipse.jetty.security.authentication
</Import-Package>
```

カスタム HTTP 500 リプライメッセージを返す方法

Camel Jetty が応答するデフォルトのリプライメッセージの代わりに、問題が発生した場合にカスタムリプライメッセージを返す必要がある場合があります。カスタム HttpBinding を使用してメッセージマッピングを制御できますが、多くの場合、Camel の Exception Clause を使用してカスタムリプライメッセージを構築する方が簡単です。たとえば、ここに示すとおり、Dude something wrong with HTTP error code 500 が返されます。

```
from("jetty://http://localhost:{{port}}/myserver")
    // use onException to catch all exceptions and return a custom reply message
    .onException(Exception.class)
        .handled(true)
        // create a custom failure response
        .transform(constant("Dude something went wrong"))
        // we must remember to set error code 500 as handled(true)
        // otherwise would let Camel thing its a OK response (200)
        .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500))
    .end()
    // now just force an exception immediately
    .throwException(new IllegalArgumentException("I cannot do this"));
```

マルチパートフォームのサポート

Camel 2.3.0 以降、camel-jetty は追加設定なしで、multipart フォームへのサポートになります。送信されたフォームデータはメッセージヘッダーにマッピングされます。camel-jetty は、アップロードされたファイルごとに添付ファイルを作成します。ファイル名は、添付の名前にマッピングされます。コンテンツタイプは、添付ファイル名のコンテンツタイプとして設定されます。この例では、こちらを参照してください。

```
// Set the jetty temp directory which store the file for multi part form
// camel-jetty will clean up the file after it handled the request.
// The option works rightly from Camel 2.4.0
getContext().getProperties().put("CamelJettyTempDir", "target");
```

```
from("jetty://http://localhost:{{port}}/test").process(new Processor() {

    public void process(Exchange exchange) throws Exception {
        Message in = exchange.getIn();
        assertEquals("Get a wrong attachement size", 1, in.getAttachments().size());
        // The file name is attachment id
        DataHandler data = in.getAttachment("NOTICE.txt");

        assertNotNull("Should get the DataHandle NOTICE.txt", data);
        // This assert is wrong, but the correct content-type (application/octet-stream)
        // will not be returned until Jetty makes it available - currently the content-type
        // returned is just the default for FileDataHandler (for the implentation being used)
        //assertEquals("Get a wrong content type", "text/plain", data.getContentType());
        assertEquals("Got the wrong name", "NOTICE.txt", data.getName());

        assertTrue("We should get the data from the DataHandle", data.getDataSource()
            .getInputStream().available() > 0);

        // The other form date can be get from the message header
        exchange.getOut().setBody(in.getHeader("comment"));
    }

});
```

JETTY JMX サポート

Camel 2.3.0 以降、camel-jetty はエンドポイント設定のコンポーネントおよびエンドポイントレベルでの Jetty の JMX 機能の有効化をサポートします。コンポーネントが Camel コンテキストに登録されている MBeanServer への参照を提供するため、このコンポーネントで JMX サポートを有効にするには、JMX を Camel コンテキスト内で有効にする必要があります。camel-jetty コンポーネントは特定のプロトコル/ホスト/ポートペアリングの Jetty リソースをキャッシュして再利用するため、この設定オプションは、プロトコル/ホスト/ポートのペアを使用するために最初のエンドポイントの作成時にのみ評価されます。たとえば、以下の XML フラグメントから作成された 2 つのルートがある場合、JMX サポートは https://0.0.0.0 でリッスンするすべてのエンドポイントに対して有効のままになります。

```
<from uri="jetty:https://0.0.0.0/myapp/myservice1/?enableJmx=true"/>
```

```
<from uri="jetty:https://0.0.0.0/myapp/myservice2/?enableJmx=false"/>
```

camel-jetty コンポーネントは、Jetty MBeanContainer の直接設定も提供します。Jetty は MBean 名を動的に作成します。Camel コンテキストの外部で Jetty の別のインスタンスを実行し、インスタンス間で同じ MBeanServer を共有する場合は、Jetty MBean を登録するときに名前が競合しないように、両方のインスタンスを同じ MBeanContainer への参照で提供できます。

- *HTTP*

第81章 JGROUPS

JGROUPS コンポーネント

Camel 2.10.0 以降で利用可能

JGroups は、信頼できるマルチキャスト通信のためのツールキットです。jgroups: コンポーネントは、Camel インフラストラクチャーと JGroups クラスター間のメッセージの交換を提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache-extra.camel</groupId>
   <artifactId>camel-jgroups</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
jgroups:clusterName[?options]
```

clusterName は、コンポーネントが接続する必要のある JGroups クラスターの名前を表します。

オプション

名前	デフォルト値	説明
channelProperties	**null**	*Camel 2.10.0:* エンドポイントによって使用される JChannel の設定プロパティーを指定します。

使用方法

ルートのコンシューマー側で jgroups コンポーネントを使用すると、エンドポイントに関連付けられた JChannel によって受信されるメッセージを取得し、Camel ルートに転送します。JGroups コンシューマーは受信メッセージを 非同期的 に処理します。

```
// Capture messages from cluster named
// 'clusterName' and send them to Camel route.
from("jgroups:clusterName").to("seda:queue");
```

ルートのプロデューサー側で jgroups コンポーネントを使用すると、Camel エクスチェンジのボディーがエンドポイントによって管理される JChannel インスタンスに転送されます。

```
// Send message to the cluster named 'clusterName'
from("direct:start").to("jgroups:clusterName");
```

第82章 JING

JING コンポーネント

Jing コンポーネントは Jing Library を使用して、以下のいずれかを使用してメッセージボディーの XML 検証を実行します。

- RelaxNG XML 構文
- RelaxNG Compact 構文

MSV コンポーネントは、RelaxNG XML 構文にも対応できることに注意してください。

URI 形式

> jing:someLocalOrRemoteResource

Camel 2.16 から、コンポーネントは URI スキームとして jing を使用し、compactSyntax オプションを使用して RNG モードまたは RNC モードのいずれかを選択できます。RNG モードは RelaxNG XML 構文を使用し、RNC モードは RelaxNG Compact 構文を使用します。以下の例は、可能な URI 値を示しています。

例	説明
jing:foo/bar.rng	クラスパスの XML ファイル **foo/bar.rng** を参照します。
jing:http://foo.com/bar.rnc?compactSyntax=true	URL **http://foo.com/bar.rnc** から RelaxNG Compact Syntax ファイルを参照します。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	デフォルト	説明

compactSyntax	false	RelaxNG compact 構文(RNC)を使用して検証するかどうか。

例

以下の 例 は、エンドポイント direct:start からのルートを設定する方法を示しています。これは、XML が指定の RelaxNG Compact Syntax スキーマ（クラスパスで提供される）と一致するかどうかに基づいて、mock:valid または mock:invalid のいずれかのエンドポイントに送信されます。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
   <route>
      <from uri="direct:start"/>
      <doTry>
         <to uri="jing:org/apache/camel/component/validator/jing/schema.rnc?
compactSyntax=true"/>
         <to uri="mock:valid"/>

         <doCatch>
            <exception>org.apache.camel.ValidationException</exception>
            <to uri="mock:invalid"/>
         </doCatch>
         <doFinally>
            <to uri="mock:finally"/>
         </doFinally>
      </doTry>
   </route>
</camelContext>
```

第83章 JIRA

JIRA コンポーネント

Camel 2.15 以降で利用可能

JIRA コンポーネントは、JIRA の Atlassian の REST Java Client をカプセル化することで JIRA API と対話し ます。現在、新しい問題と新しいコメントをポーリングしています。また、新たな問題を作成することもできます。

Webhook ではなく、このエンドポイントは単純なポーリングに依存します。理由は次のとおりです。

- 信頼性/安定性の懸念
- ポーリングしているペイロードのタイプは通常大きくありません（上向き、ページングは API で利用可能です）。
- Webhook が失敗した場合にパブリックにアクセスできない一部のアプリケーションをサポートする必要があります。

JIRA API はかなり大きくなることに注意してください。そのため、このコンポーネントは簡単に拡張でき、追加の対話を提供できます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jira</artifactId>
    <version>${camel-version}</version>
</dependency>
```

URI 形式

```
jira://endpoint[?options]
```

必須オプション：

これらは エンドポイントで直接設定できることに注意してください。

オプション	説明
serverUrl	JIRA ホストサーバーの URL
username	JIRA ユーザー名
password	JIRA password

コンシューマーエンドポイント：

エンドポイント	コンテキスト	ボディタイプ
newIssue	ポーリング	com.atlassian.jira.rest.client.domain.BasicIssue
newComment	ポーリング	com.atlassian.jira.rest.client.domain.Comment

プロデューサーエンドポイント：

エンドポイント	本文	必要なメッセージヘッダー
pullRequestComment	文字列（説明を発行）	• ProjectKey （文字列）: プロジェクトキー • IssueTypeId (long): 問題タイプ ID （例：Bug は通常ほとんどのデフォルト設定では 1 です） • IssueSummary （文字列）: 問題の概要（タイトル）

URI オプション：

名前	デフォルト値	説明

delay	**60**	秒単位
jql		コンシューマーエンドポイントによって使用されます。詳細は以下を参照してください。

JQL:

JQL URI オプションは、両方のコンシューマーエンドポイントによって使用されます。理論的には、プロジェクトキーなどのアイテムは URI オプション自体である可能性があります。ただし、JQL の使用を要求すると、コンシューマーはより柔軟で強力になります。

少なくとも、コンシューマーには以下が必要です。

> ***jira://[endpoint]?[required options]&jql=project=[project key]***

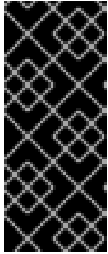
注意すべき重要なことは、newIssue コンシューマーが JQL に "ORDER BY key desc" を自動的に追加することです。これは、プロジェクト内のすべての問題をインデックス化する必要なく、起動処理を最適化するために行われます。

別の注意点として、newComment コンシューマーはすべての問題をインデックス化し、プロジェクトのコメント を 送る必要があります。そのため、大規模なプロジェクトでは、JQL 式を可能 な 限り最適化することが重要です。たとえば、JIRA Toolkit プラグインには、"Number of comments" カスタムフィールド -- use '"Number of comments" > 0' がクエリーに含まれます。また、状態 (status=Open)に基づいて最小化を試み、ポーリングの遅延などを増やします。以下に例を示します。

> ***jira://[endpoint]?[required options]&jql=RAW(project=[project key] AND status in (Open, \"Coding In Progress\") AND \"Number of comments\">0)"***

第84章 JMS

JMS COMPONENT



ACTIVEMQ の使用

Apache ActiveMQ を使用している場合は、ActiveMQ で最適化されているため、ActiveMQ コンポーネントを使用することが推奨されます。このページのすべてのオプションとサンプルも ActiveMQ コンポーネントに対して有効です。

トランザクションおよびキャッシュ

JMS でトランザクションを使用している場合は、パフォーマンスに影響する可能性があるため、以下のトランザクション レベルとキャッシュ レベルを参照してください。

JMS でのリクエスト/応答

Camel にはパフォーマンスおよびクラスター環境に設定するオプションを多数提供するため、このページの Request-reply over JMS セクションは、リクエスト/リプライに関する重要なメモについて、さらに JMS で確認してください。

JMS コンポーネントを使用すると、JMS キューまたはトピックとの間でメッセージを送信（または消費）できます。宣言的トランザクションには Spring の JMS サポートを使用し、送信に Spring の JmsTemplate を使用し、消費するために MessageListenerContainer を使用します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jms</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークに

よってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
jms:[queue:|topic:]destinationName[?options]
```

destinationName は JMS キューまたはトピック名です。デフォルトでは、destinationName はキュー名として解釈されます。たとえば、キューに接続するには、FOO.BAR を次のように使用します。

```
jms:FOO.BAR
```

必要に応じて、オプションの queue: 接頭辞を含めることができます。

```
jms:queue:FOO.BAR
```

トピックに接続するには、topic: 接頭辞を含める 必要があり ます。たとえば、Stocks.Prices トピックに接続するには、以下を使用します。

```
jms:topic:Stocks.Prices
```

?option=value&option=value&.. という形式を使用して、URI にクエリーオプションを追加します。

ACTIVEMQ の使用

JMS コンポーネントは、メッセージ送信に Spring 2 の JmsTemplate を再利用します。これは、J2EE 以外のコンテナーでの使用に理想的ではなく、パフォーマンスが低下 するのを防ぐために、通常 JMS プロバイダーでキャッシュを必要とします。

Apache ActiveMQ をメッセージブローカーとして使用する場合は、以下のいずれかを推奨します。

- ActiveMQ を効率的に使用するように最適化されている ActiveMQ コンポーネントを使用する
- ActiveMQ の PoolingConnectionFactory を使用します。

トランザクションおよびキャッシュレベル

メッセージを使用し、トランザクション(transacted=true)を使用している場合、キャッシュレベルのデフォルト設定はパフォーマンスに影響を与える可能性があります。XA トランザクションを使用している場合は、XA トランザクションが適切に機能しなくなる可能性があるため、キャッシュできません。

XA を使用してい ない 場合は、cacheLevelName=CACHE_CONSUMER の設定など、パフォーマンスの速度を上げるようにキャッシュを検討してください。

Camel 2.7.x より、cacheLevelName のデフォルト設定は CACHE_CONSUMER です。cacheLevelName=CACHE_NONE を明示的に設定する必要があります。Camel 2.8 以降では、cacheLevelName のデフォルト設定は CACHE_AUTO です。このデフォルトは、モードを自動的に検出し、それに応じてキャッシュレベルを設定します。

- *CACHE_CONSUMER = if transacted=false*
- *CACHE_NONE = if transacted=true*

そのため、デフォルト設定が保存されているとすることができます。非 XA トランザクションを使用している場合は、cacheLevelName=CACHE_CONSUMER の使用を検討してください。

永続サブスクリプション

永続トピックサブスクリプションを使用する場合は、clientId と durableSubscriptionName の両方を指定する必要があります。clientId の値は一意でなければならず、ネットワーク全体の単一の JMS 接続インスタンスによってのみ使用できます。この制限を回避するために、代わり に仮想トピック を使用することが推奨されます。ここ での永続メッセージングに関する詳細な背景。

メッセージヘッダーのマッピング

メッセージヘッダーを使用する場合、JMS 仕様ではヘッダー名が有効な Java 識別子である必要があります。そのため、ヘッダーに有効な Java 識別子に名前を付けてみてください。これを行う利点の 1 つは、JMS セレクター内でヘッダーを使用できることです(SQL92 構文でヘッダーの Java 識別子構文が義務付けられます)。

デフォルトでは、ヘッダー名をマッピングする簡単なストラテジーが使用されます。ストラテジーは、以下に示すようにヘッダー名のドットとハイフンを置き換え、ヘッダー名がネットワーク経由で送

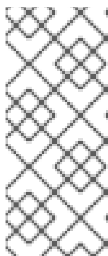
信される JMS メッセージから復元されるときに置換を元に戻すことです。意味を確認するBean コンポーネントで呼び出すメソッド名を失うことなく、File Component の filename ヘッダーを失うことはありません。

Camel でヘッダー名を受け入れる現在のヘッダー名ストラテジーは次のとおりです。

- ドットは _DOT_ に置き換えられ、Camel がメッセージを消費すると置換が逆になります。
- ハイフンは _HYPHEN_ に置き換えられ、Camel がメッセージを消費すると置換が逆になります。

オプション

JMSConfiguration POJO でプロパティーにマップする JMS エンドポイントで多くの異なるプロパティーを設定できます。



SPRING JMS へのマッピング

これらのプロパティーの多くは、Camel がメッセージの送受信に使用する Spring JMS のプロパティーにマップされます。そのため、関連する Spring ドキュメント を参照して、これらのプロパティーの詳細情報を取得できます。

オプションは 2 つのテーブルに分割され、最初のテーブルは使用される最も一般的なオプションになります。後者には残りが含まれます。

最も一般的に使用されるオプション

オプション	デフォルト値	説明

clientId	**null**	使用する JMS クライアント ID を設定します。この値は、指定されている場合は一意である必要があり、単一の JMS 接続インスタンスでのみ使用できることに注意してください。通常、これは永続トピックサブスクリプションにのみ必要です。代わりに 仮想トピックを使用することも できます。
concurrentConsumers	**1**	同時コンシューマーのデフォルトの数を指定します。**Camel 2.10.3** 以降では、JMS でリクエスト/リプライを行うときにこのオプションを使用することもできます。スレッドの動的なスケールアップ/ダウンを制御するには、**maxMessagesPerTask** オプションも参照してください。
replyToConcurrentConsumers	**1**	Camel 2.16: JMS 上でリクエスト/リプライを行う際の同時コンシューマーのデフォルトの数を指定します。
disableReplyTo	**false**	**true** の場合、プロデューサーは **JMSReplyTo** ヘッダーが送信され、InOnly の場合のように抑制されない例外を持つ **InOnly** エクスチェンジのように動作します。**InOnly** と同様に、プロデューサーは応答を待ちません。このフラグを持つコンシューマーは **InOnly** のように動作します。この機能は、**InOut** 要求を別のキューにブリッジするために使用できます。これにより、他のキューのルートが応答を直接元の **JMSReplyTo** に送信することができます。
durableSubscriptionName	**null**	永続トピックサブスクリプションを指定するための永続サブスクライバー名。**clientId** オプションも設定 **する必要があり** ます。

maxConcurrentConsumers	**1**	同時コンシューマーの最大数を指定します。**Camel 2.10.3** 以降では、JMS でリクエスト/リプライを行うときにこのオプションを使用することもできます。スレッドの動的なスケールアップ/ダウンを制御するには、**maxMessagesPerTask** オプションも参照してください。スレッドをスケールダウンするには、**maxMessagesPerTask** オプションを 0 より大きい整数に設定する必要があります。そうしないと、スレッドの数はシャットダウンまで **maxConcurrentConsumers** に残ります。
replyToMaxConcurrentConsumers	**1**	**Camel 2.16:** JMS 上でリクエスト/リプライを行う際の同時コンシューマーの最大数を指定します。スレッドの動的なスケールアップ/ダウンを制御するオプションも参照してください。**maxMessagesPerTask**
maxMessagesPerTask	**-1**	タスクごとのメッセージ数。\-1 は無制限です。同時コンシューマーの範囲（例：min < max）を使用する場合は、このオプションを使用して値を **100** に設定し、必要な作業が少ない場合にコンシューマーを縮小する方法を制御できます。

preserveMessageQos	**false**	JMS エンドポイントの QoS 設定の代わりに、メッセージで指定された QoS 設定を使用してメッセージを送信する場合は **true** に設定します。以下の 3 つのヘッダーは **JMSPriority**、**JMSDeliveryMode**、および **JMSExpiration** とみなされます。提供できるのはすべてまたは一部のみです。指定のない場合は、Camel はフォールバックしてエンドポイントからの値を使用します。そのため、このオプションを使用すると、ヘッダーはエンドポイントからの値を上書きします。これとは対照的に、**explicitQosEnabled** オプションは、エンドポイントに設定されたオプションのみを使用し、メッセージヘッダーからの値は使用しません。
replyTo	**null**	明示的な ReplyTo 宛先を提供します。**これは、Message.getJMSReplyTo（）の受信値を上書きします。** JMS 経由で リクエスト応答 を行う場合は、以下の JMS でさらに **Request-reply** で確認して詳細を確認し、**replyToType** オプションも **必ず** 読んでください。
replyToOverride	**null**	**Camel 2.15:** JMS メッセージの明示的な ReplyTo 宛先を提供します。これは replyTo の設定を上書きします。メッセージをリモートキューへ転送し、ReplyTo 宛先からリプライメッセージを受信する場合に便利です。

replyToType	**null**	**Camel 2.9:** JMS 上でリクエスト/リプライを行うときに、replyTo キューに使用するストラテジーを明示的に指定できます。使用できる値は、**一時的な**、**共有**、または **排他的** です。デフォルトでは、Camel は一時キューを使用します。ただし、**replyTo** が設定されている場合は、デフォルトで **Shared** が使用されます。このオプションを使用すると、共有キューの代わりに排他的キューを使用できます。詳細は、以下を参照してください。特に、クラスター環境で実行されたかどうかに関する注意点と、**Shared** 応答キューのパフォーマンスは、alternative **Temporary** および **Exclusive** よりも低いという事実を参照してください。
requestTimeout	**20000**	**プロデューサーのみ：** InOut Exchange Pattern（ミリ秒単位）を使用する場合の応答を待機するタイムアウト。デフォルトは 40 秒です。**Camel 2.13/2.12.3** 以降では、ヘッダー **CamelJmsRequestTimeout** を追加してこのエンドポイントが設定されたタイムアウト値を上書きし、メッセージごとに個別のタイムアウト値を設定できます。詳細は、**About time to live** のセクションを参照してください。**requestTimeoutCheckerInterval** オプションも参照してください。
selector	**null**	ブローカー内のメッセージのフィルターに使用される SQL 92 述語である JMS セレクターを設定します。Camel **2.3.0 よりも前のバージョンでは、= などの特殊文字をエンコードしないといけない場合があります。Camel** ConsumerTemplate ではこのオプションをサポートしません。
timeToLive	**null**	メッセージを送信する場合は、メッセージの存続期間（ミリ秒単位）を指定します。詳細は、**About time to live** のセクションを参照してください。

transacted	**false**	InOnly Exchange Pattern を使用したメッセージの送受信にトランザクションモードを使用するかどうかを指定します。
testConnectionOnStartup	**false**	**Camel 2.1:** 起動時に接続をテストするかどうかを指定します。これにより、Camel が起動すると、すべての JMS コンシューマーが JMS ブローカーに有効な接続を持つようになります。接続を付与できない場合、Camel は起動時に例外を出力します。これにより、Camel が失敗したコネクションで開始されなくなります。**Camel 2.8** 以降では、JMS プロデューサーもテストされています。

その他のすべてのオプション

オプション	デフォルト値	説明
acceptMessagesWhileStopping	**false**	停止中にコンシューマーがメッセージを受け入れるかどうかを指定します。キューに送信されたメッセージは、実行時に JMS ルートを開始および停止する場合、このオプションを有効にすることを検討してください。このオプションが **false** で、JMS ルートを停止すると、メッセージが拒否され、JMS ブローカーは再配信を試行する必要がありますが、再度拒否され、最終的にメッセージは JMS ブローカーのデッドレターキューに移動される可能性があります。これを回避するには、このオプションを有効にすることが推奨されます。
acknowledgementModeName	**AUTO_ACKNOWLEDGE**	**SESSION_TRANSACTED**、**CLIENT_ACKNOWLEDGE**、**AUTO_ACKNOWLEDGE**、**DUPS_OK_ACKNOWLEDGE**のいずれかである JMS の確認名

acknowledgementMode	\-1	整数として定義される JMS 確認モード。ベンダー固有の拡張機能を確認モードに設定できます。通常のモードでは、代わりに **acknowledgementModeName** を使用することが推奨されます。
allowNullBody	**true**	Camel 2.9.3/2.10.1: ボディーのないメッセージの送信を許可するかどうか。このオプションが **false** でメッセージボディーが null の場合、**JMSException** が出力されます。
alwaysCopyMessage	**false**	**true** の場合、送信のためにプロデューサーに渡されるときに、Camel は常にメッセージの JMS メッセージコピーを作成します。**replyToDestinationSelectorName** が設定されている場合など、一部の状況でメッセージのコピーが必要になる場合（誤って Camel は **replyToDestinationSelectorName** が設定されている場合、Camel は **alwaysCopyMessage** オプションを **true** に設定します）
asyncConsumer	**false**	Camel 2.9: **JmsConsumer** が エクスチェンジ を 非同期的 に処理するかどうか。有効にすると、**JmsConsumer** は JMS キューから次のメッセージを選択し、以前のメッセージは非同期で処理されます(Asynchronous Routing Engine)。つまり、メッセージは 100% ではなく、順番に処理できます。無効（デフォルト）の場合、**JmsConsumer** が JMS キューから次のメッセージを取得する前に、エクスチェンジ は完全に処理されます。トランザクションが有効になっている場合は、トランザクションを同期的に実行する必要があるため、**asyncConsumer=true** は非同期的に実行しません(Camel 3.0 は非同期トランザクションをサポートする可能性があるため)。

asyncStartListener	**false**	Camel 2.10: ルートの開始時に **JmsConsumer** メッセージリスナーを非同期に起動するかどうか。たとえば、**JmsConsumer** がリモート JMS ブローカーへの接続を取得できない場合、再試行中やフェイルオーバー中にブロックされる可能性があります。これにより、ルートの開始時に Camel がブロックされます。このオプションを **true** に設定すると、ルートの起動を許可します。一方、**JmsConsumer** は非同期モードで専用のスレッドを使用して JMS ブローカーに接続します。このオプションを使用する場合は、接続を確立できない場合は例外が **WARN** レベルでログに記録され、コンシューマーはメッセージを受信できず、ルートを再起動して再試行できます。
asyncStopListener	**false**	Camel 2.10: ルートを停止するときに **JmsConsumer** メッセージリスナーを非同期に停止するかどうか。
autoStartup	**true**	コンシューマーコンテナーを自動起動させるかどうかを指定します。
cacheLevelName	CACHE_AUTO (Camel >= 2.8.0) CACHE_CONSUMER (Camel <= 2.7.1)	基礎となる JMS リソースの名前でキャッシュレベルを設定します。使用できる値は、**CACHE_AUTO**、**CACHE_CONNECTION**、**CACHE_CONSUMER**、**CACHE_NONE**、および **CACHE_SESSION** です。Camel 2.8 以降のデフォルト設定は **CACHE_AUTO** です。Camel 2.7.1 以前では、デフォルトは **CACHE_CONSUMER** です。詳細は、Spring のドキュメント および トランザクションキャッシュレベル を 参照してください。
cacheLevel		基礎となる JMS リソースの ID でキャッシュレベルを設定します。詳細は、**cacheLevelName** オプションを参照してください。

consumerType	デフォルト	使用するコンシューマータイプ。**Simple**、**Default**、または **Custom** のいずれかです。コンシューマータイプは、使用する Spring JMS リスナーを決定します。**デフォルト** では **org.springframework.jms.listener.DefaultMessageListenerContainer** を使用します。**Simple** は **org.springframework.jms.listener.SimpleMessageListenerContainer** を使用します。**Custom** が指定されている場合、**messageListenerContainerFactoryRef** オプションで**定義される MessageListenerContainerFactory は、使用する org.springframework.jms.listener.AbstractMessageListenerContainer を決定します (Camel 2.11 および 2.10.2 の新しいオプション)**。このオプションは、Camel 2.7 および 2.8 で一時的に削除されました。ただし、Camel 2.9 以降は再び追加されました。
connectionFactory	**null**	指定がない場合は、**listenerConnectionFactory** および **templateConnectionFactory** に使用するデフォルトの JMS 接続ファクトリー。

defaultTaskExecutorType	（説明を参照）	Camel 2.10.4: プロデューサーエンドポイントのコンシューマーエンドポイントと ReplyTo コンシューマーの両方に対して、DefaultMessageListenerContainer で使用するデフォルトの TaskExecutor タイプを指定します。使用できる値： **SimpleAsync** (Spring の SimpleAsyncTaskExecutorを使用) または **ThreadPool** （最適な値で Spring の ThreadPoolTaskExecutor を使用）- キャッシュされた threadpool-like を使用します。設定されていない場合、デフォルトで以前の動作に設定されます。これはコンシューマーエンドポイントにキャッシュされたスレッドプールを使用し、応答コンシューマーに SimpleAsync を使用します。**ThreadPool** の使用は、同時コンシューマーを動的に増減することで、Elastic 設定でスレッドゴミ箱を減らすことが推奨されます。
deliveryMode	null	Camel 2.12.2/2.13: 使用する配信モードを指定します。Possibles 値は、**javax.jms.DeliveryMode** で定義される値になります。
deliveryPersistent	**true**	永続的な配信がデフォルトで使用されるかどうかを指定します。
destination	**null**	このエンドポイントで使用する JMS Destination オブジェクトを指定します。
destinationName	**null**	このエンドポイントで使用する JMS 宛先名を指定します。
destinationResolver	**null**	独自のリゾルバーを使用できるプラグ可能な **org.springframework.jms.support.destination.DestinationResolver**。JNDI レジストリーで実際の宛先を検索するなど。

disableTimeToLive	**false**	camel **2.8:** このオプションを使用して、時間のライブを無効にします。たとえば、JMS でリクエスト/リプライを行う場合、Camel はデフォルトで **requestTimeout** 値を送信されるメッセージに対して存続する時間として使用します。この問題は、送信者と受信者のシステムでクロックを同期して同期する必要があることです。これは必ずしも簡単にアーカイブできるとは限りません。そのため、**disableTimeToLive=true** を使用して、送信されたメッセージに time to live 値を設定でき **ません**。その後、メッセージは受信側システムで期限切れになりません。詳細は、**About time to live** のセクションを参照してください。
eagerLoadingOfProperties	**false**	JMS プロパティーが必要ないため、メッセージ受信直後に JMS プロパティーの Eager 読み込みを有効にします。ただし、この機能は基礎となる JMS プロバイダーの問題や JMS プロパティーの使用の初期段階でキャッチすることがあります。この機能はテスト目的でも使用でき、JMS プロパティーが正しく認識および処理されるようにします。
exceptionListener	**null**	基礎となる JMS 例外を通知する JMS 例外リスナーを指定します。
errorHandler	**null**	>**Camel 2.8.2、2.9:** メッセージの処理中に検出されない例外が発生した場合に **org.springframework.util.ErrorHandler** を呼び出し **ます**。デフォルトでは、**errorHandler** が設定されていない場合、これらの例外は WARN レベルでログに記録されます。**Camel 2.9.1** 以降、以下の 2 つのオプションを使用して、ロギングレベルおよびスタックトレースをログに記録するかどうかを設定できます。これにより、カスタム **errorHandler** をコーディングする必要がなく、設定が非常に簡単になります。

errorHandlerLoggingLevel	**WARN**	Camel 2.9.1: キャッチされない例外をログに記録するために、デフォルトの **errorHandler** ロギングレベルを設定できます。
errorHandlerLogStackTrace	**true**	Camel 2.9.1: デフォルトの **errorHandler** で、スタックトレースをログに記録するかどうかを制御できます。
explicitQosEnabled	**false**	メッセージの送信時に **deliveryMode**、**priority**、または **timeToLive** のサービス品質を使用する場合を設定します。このオプションは Spring の **JmsTemplate** に基づいています。**deliveryMode** オプション、**priority** オプション、および **timeToLive** オプションが現在のエンドポイントに適用されます。これは、メッセージ粒度で動作し、Camel In メッセージヘッダーからのみ QoS プロパティーを読み取る **preserveMessageQos** オプションとは対照的です。
exposeListenerSession	**true**	メッセージを消費するときにリスナーセッションを公開するかどうかを指定します。
forceSendOriginalMessage	**false**	>Camel 2.7: **mapJmsMessage=false** を使用すると、ルート中にヘッダー（取得または設定）にアクセスすると、Camel は新しい JMS 宛先に送信する新しい JMS メッセージを作成します。このオプションを **true** に設定すると、Camel は受信された元の JMS メッセージを送信するようにします。
idleTaskExecutionLimit	**1**	実行内でメッセージを受信せずに、受信タスクのアイドル実行の制限を指定します。この制限に達すると、タスクはシャットダウンし、他の実行中のタスクまで受信したままになります（動的スケジューリングの場合は、**maxConcurrentConsumers** の設定を参照してください）。

idleConsumerLimit	**1**	**Camel 2.8.2、2.9:** 指定した時間にアイドル状態にできるコンシューマーの数の制限を指定します。
includeSentJMSMessageID	**false**	**Camel 2.10.3:** InOnly（例：fire および forget）を使用して JMS 宛先に送信する場合にのみ適用されます。このオプションを有効にすると、メッセージが JMS 宛先に送信されたときに JMS クライアントによって使用される実際の JMSMessageID で Camel Exchange を補完します。
includeAllJMSXProperties	**false**	**Camel 2.11.2/2.12:** JMS から Camel Message にマッピングする際にすべての JMSXxxx プロパティーを含めるかどうか。これを **true** に設定すると、**JMSXAppID** や **JMSXUserID** などのプロパティーが含まれます。**注記：** カスタムの **headerFilterStrategy** を使用している場合、このオプションは適用されません。
jmsMessageType	**null**	JMS メッセージの送信に特定の **javax.jms.Message** 実装の使用を強制できます。使用できる値は、**バイト**、**マップ**、**オブジェクト**、**ストリーム**、**テキストです**。デフォルトでは、Camel は In ボディータイプから使用する JMS メッセージタイプを決定します。このオプションを使用すると指定できます。

jmsKeyFormatStrategy	**default**	JMS 仕様に準拠できるように、JMS キーをエンコードおよびデコードするためのプラグ可能な戦略。Camel は、追加設定なしで、**default** と **passthrough** の 2 つの実装を提供します。**デフォルト** のストラテジーは、ドットとハイフン(**.** および **\-**)を安全にマーシャリングします。**パススルー** ストラテジーは、キーをそのまま残します。JMS ヘッダーキーに不正な文字が含まれているかどうかは問題にならない JMS ブローカーに使用できます。**org.apache.camel.component.jms.JmsKeyFormatStrategy** の独自の実装を提供し、**\#** 表記を使用して参照できます。
jmsOperations	**null**	**org.springframework.jms.core.JmsOperations** インターフェイスの独自の実装を使用できます。Camel はデフォルトで **JmsTemplate** を使用します。テスト目的に使用できますが、Spring API ドキュメントに記載されているあまり使用されません。
lazyCreateTransactionManager	**true**	**true** の場合、オプション **transacted=true** 時に **transactionManager** が注入されない場合、Camel は **JmsTransactionManager** を作成します。
listenerConnectionFactory	**null**	メッセージの消費に使用される JMS 接続ファクトリー。
mapJmsMessage	**true**	Camel が受信した JMS メッセージを **javax.jms.TextMessage** などの適切なペイロードタイプに自動的にマップするかどうかを指定 **し** ます。詳細は、マッピングの仕組みについて以下のセクションを参照してください。
maximumBrowseSize	**\-1**	Browse または JMX API を使用してエンドポイントを参照する際に、最もフェッチされたメッセージの数を制限します。

messageConverter	**null**	カスタムの Spring **org.springframework.jms.support.converter.MessageConverter** を使用して、javax.jms.Message のマッピング方法や、**javax.jms.Message** からのマッピング方法を制御することができます。
messageIdEnabled	**true**	送信時に、メッセージ ID を追加するかどうかを指定します。
messageListenerContainerFactoryRef	**null**	Camel 2.10.2: メッセージの消費に使用する **org.springframework.jms.listener.AbstractMessageListenerContainer** を決定するために使用される **MessageListenerContainerFactory** のレジストリー ID。これを設定すると、**consumerType** が自動的に **Custom** に設定されます。
messageTimestampEnabled	**true**	メッセージの送信時にタイムスタンプをデフォルトで有効にするかどうかを指定します。
password	**null**	コネクターファクトリーのパスワード。
priority	**4**	1 より大きい値は、送信時のメッセージの優先度を指定します(0 は最も低い優先度で、9 は最も高い値です)。このオプションを有効にするには、**explicitQosEnabled** オプションも有効にする **必要** があります。
pubSubNoLocal	**false**	独自の接続によってパブリッシュされたメッセージの配信を禁止するかどうかを指定します。
receiveTimeout	**1000**	メッセージの受信のタイムアウト（ミリ秒単位）。
recoveryInterval	**5000**	リカバリーの試行の間隔を指定します。つまり、接続が更新されるタイミング（ミリ秒単位）を指定します。デフォルトは 5000 ミリ秒(5 秒)です。

replyToSameDestinationAllowed	**false**	Camel 2.16: コンシューマーのみ：JMS コンシューマーが消費に使用するのと同じ宛先にリプライメッセージを送信できるかどうか。これにより、同じメッセージを消費して自身に送信することで、無限ループを防ぐことができます。
replyToCacheLevelName	**CACHE_CONSUMER**	camel 2.9.1: JMS 上でリクエスト/リプライを行うときに応答コンシューマーの名前でキャッシュレベルを設定します。このオプションは、（一時的ではなく）Fixed reply queue を使用する場合にのみ適用されます。Camel はデフォルトで を使用します。排他的または共有 w/ **replyToSelectorName** に **CACHE_CONSUMER** を使用します。**replyToSelectorName** なしで共有される **CACHE_SESSION**。IBM WebSphere などの一部の JMS ブローカーは、**replyToCacheLevelName =CACHE_NONE** を設定して機能させる必要がある場合があります。
replyToDestinationSelectorName	**null**	固定名を使用して JMS セレクターを設定し、共有キューの使用時に（つまり一時的な応答キューを使用していない場合）、他の者からのリプライをフィルターリングできます。
replyToDeliveryPersistent	**true**	応答に永続配信をデフォルトで使用するかどうかを指定します。
requestTimeoutCheckerInterval	**1000**	Camel 2.9.2: JMS でリクエスト/リプライを行うときに Camel がタイムアウトした エクスチェンジをチェックする頻度を設定します。デフォルトの Camel は 1 秒あたり 1 回チェックします。ただし、タイムアウトの発生時に迅速な反応が必要な場合は、この間隔を下げてより頻繁にチェックできます。タイムアウトは、requestTimeout オプションによって決定されます。

subscriptionDurable	**false**	**@非推奨： durableSubscriptionName** および **clientId** を指定するとデフォルトで有効になります。
taskExecutor	**null**	メッセージを消費するためにカスタムタスクエグゼキューターを指定できます。
taskExecutorSpring2	**null**	**Camel 2.6:** Camel で Spring 2.x を使用する場合に使用する。メッセージを消費するためにカスタムタスクエグゼキューターを指定できます。
templateConnectionFactory	**null**	メッセージの送信に使用される JMS 接続ファクトリー。
transactedInOut	**false**	**@非推奨：** InOut Exchange パターン を使用してメッセージを送信するためにトランザクションモードを使用するかどうかを指定し ます。プロデューサーエンドポイントにのみ適用されます。詳細は、Transacted Consumption の有効化 を参照してください。
transactionManager	**null**	使用する Spring トランザクションマネージャー。
transactionName	**"JmsConsumer[destinationName]"**	使用するトランザクションの名前。
transactionTimeout	**null**	トランザクションモードを使用する場合のトランザクションのタイムアウト値（秒単位）。

transferException	**false**	有効にすると、リクエスト 応答メッセージング(InOut)を使用し、エクスチェンジ がコンシューマー側で失敗した場合、原因となった **例外** は応答として **javax.jms.ObjectMessage** として送り返されます。クライアントが Camel の場合、返される **例外が再出力** されます。これにより、Camel JMS をルーティングのブリッジとして使用できます。たとえば、永続キューを使用して堅牢なルーティングを有効にすることができます。transferExchange も有効にされている場合、このオプションは優先されることに注意してください。キャッチされた例外はシリアライズ可能である必要があります。コンシューマー側の元の例外は、プロデューサーに返されるときに **org.apache.camel.RuntimeCamel Exception** などの外部例外でラップできます。
transferFault	**false**	Camel 2.17: が有効で、リクエスト応答メッセージング(InOut)を使用し、エクスチェンジがコンシューマー側で SOAP 障害（例外ではない）で失敗した場合、**org.apache.camel.Message.isFault()** の fault フラグは、キー **JmsConstants.JMS_TRANSFER_FAULT** を持つ JMS ヘッダーとして応答として送り返されます。クライアントが Camel の場合、返される fault フラグが **org.apache.camel.Message.setFault(boolean)** に設定されます。CXF や Spring-WS などの SOAP ベースをサポートする Camel コンポーネントを使用する場合は、これを有効にできます。

transferExchange	false	ボディーとヘッダーだけでなく、ネットワーク上でエクスチェンジを転送することができます。以下のフィールドは転送されます：ボディー、Out body、Fault ボディー、In headers、Out ヘッダー、Fault ヘッダー、エクスチェンジプロパティー、エクスチェンジ例外。これには、オブジェクトがシリアライズ可能である必要があります。Camel はシリアライズできないオブジェクトを除外し、**WARN** レベルでログに記録します。このオプションはプロデューサーとコンシューマーの両方で有効に **する必要があり** ます。したがって、Camel はペイロードが通常のペイロードではなく Exchange であることを認識します。
username	null	コネクターファクトリーのユーザー名。
useMessageIDAsCorrelationID	false	**JMSMessageID** を常に **InOut** メッセージの **JMSCorrelationID** として使用するかどうかを指定します。
useVersion102	false	**@廃止予定(Camel 2.5 以降から削除):** 古い JMS API を使用するかどうかを指定します。

サンプル

JMS は、他のコンポーネントにも多くの例で使用されます。ただし、開始するためのいくつかのサンプルを以下に示します。

JMS からの受信

以下の例では、JMS メッセージを受信し、そのメッセージを POJO にルーティングするルートを設定します。

```
from("jms:queue:foo").
  to("bean:myBusinessLogic");
```

ルートがコンテキストベースになるように、任意の EIP パターンを使用することができます。たとえ

ば、大きな費ナーに対して注文トピックをフィルターリングする方法を以下に示します。

```
from("jms:topic:OrdersTopic").
  filter().method("myBean", "isGoldCustomer").
    to("jms:queue:BigSpendersQueue");
```

JMS に送信する

以下の例では、ファイルフォルダーをポーリングし、ファイルの内容を JMS トピックに送信します。BytesMessage ではなく TextMessage としてファイルの内容が必要な場合は、ボディーを String に変換する必要があります。

```
from("file://orders").
  convertBodyTo(String.class).
  to("jms:topic:OrdersTopic");
```

アノテーションの使用

Camel にはアノテーションも含まれるため、POJO Consuming および POJO Producing を使用できます。

SPRING DSL の例

上記の例では Java DSL を使用しています。Camel は Spring XML DSL もサポートします。以下は、Spring DSL を使用した高額支出者のサンプルです。

```
<route>
  <from uri="jms:topic:OrdersTopic"/>
  <filter>
    <method bean="myBean" method="isGoldCustomer"/>
    <to uri="jms:queue:BigSpendersQueue"/>
  </filter>
</route>
```

その他のサンプル

JMS は、他のコンポーネントおよび EIP パターンの例の多くに表示され、この Camel ドキュメントにも記載されています。したがって、ドキュメントを自由に参照してください。

JMS と CAMEL 間のメッセージマッピング

Camel は javax.jms.Message と org.apache.camel.Message の間でメッセージを自動的にマッピングします。

JMS メッセージを送信する場合、Camel はメッセージボディーを以下の JMS メッセージボディーに変換します。

ボディタイプ	JMS Message	Comment
文字列	**javax.jms.TextMessage**	
org.w3c.dom.Node	**javax.jms.TextMessage**	DOM は **String** に変換されます。
マップ	**javax.jms.MapMessage**	
java.io.Serializable	**javax.jms.ObjectMessage**	
byte[]	**javax.jms.BytesMessage**	
java.io.File	**javax.jms.BytesMessage**	
java.io.Reader	**javax.jms.BytesMessage**	
java.io.InputStream	**javax.jms.BytesMessage**	
java.nio.ByteBuffer	**javax.jms.BytesMessage**	

JMS メッセージを受信すると、Camel は JMS メッセージを以下のボディー型に変換します。

JMS Message	ボディタイプ
javax.jms.TextMessage	文字列
javax.jms.BytesMessage	**byte[]**
javax.jms.MapMessage	**Map<String, Object>**
javax.jms.ObjectMessage	オブジェクト

JMS メッセージの自動マッピングの無効化

mapJmsMessage オプションを使用して、上記の自動マッピングを無効にすることができます。無効にすると、Camel は受信した JMS メッセージをマップせず、ペイロードとして直接使用します。これにより、マッピングのオーバーヘッドを回避し、Camel が JMS メッセージを通過させることができます。たとえば、クラスパス上に ない クラスで javax.jms.ObjectMessage JMS メッセージをルーティングすることもできます。

カスタム MESSAGECONVERTER の使用

messageConverter オプションを使用して、Spring org.springframework.jms.support.converter.MessageConverter クラスでマッピングを実行できます。

たとえば、以下のルートでは、JMS 注文キューにメッセージを送信するときにカスタムメッセージコンバーターを使用します。

```
from("file://inbox/order").to("jms:queue:order?messageConverter=#myMessageConverter");
```

JMS 宛先から消費するときに、カスタムメッセージコンバーターを使用することもできます。

選択したマッピングストラテジーの制御

エンドポイント URL で jmsMessageType オプションを使用して、すべてのメッセージに対して特定のメッセージタイプを強制的に実行できます。以下のルートでは、ファイルをフォルダーからポーリングし、javax.jms.TextMessage として送信します。これは、JMS プロデューサーエンドポイントがテキストメッセージを使用するように強制したためです。

```
from("file://inbox/order").to("jms:queue:order?jmsMessageType=Text");
```

ヘッダーに CamelJmsMessageType キーを設定して、各メッシングに使用するメッセージタイプを指定することもできます。以下に例を示します。

```
from("file://inbox/order").setHeader("CamelJmsMessageType",
JmsMessageType.Text).to("jms:queue:order");
```

使用できる値は enum クラス org.apache.camel.jms.JmsMessageType で定義されます。

送信時のメッセージ形式

JMS ネットワーク上で送信されるエクスチェンジは、JMS Message 仕様 に準拠する必要があります。

exchange.in.header の場合、ヘッダー キー に以下のルールが適用されます。

- JMS または JMSX で始まるキーは予約されています。
- exchange.in.headers キーはリテラルで、すべて有効な Java 識別子である必要があります（キー名にドットを使用しないでください）。
- Camel は、JMS メッセージを消費するときにドットとハイフンと逆順を置き換えます。. は _DOT_ に置き換え、Camel がメッセージを消費するときに逆の置換を行います。\- は _HYPHEN_ に置き換え、Camel がメッセージを消費するときに逆の置換を行います。
- 鍵のフォーマットに独自のカスタムストラテジーを使用できるオプション jmsKeyFormatStrategy も併せて参照してください。

exchange.in.header の場合、以下のルールがヘッダー 値 に適用されます。

- 値はプリミティブまたはカウンターオブジェクト(Integer、Long、Characterなど)である必要があります。型、String、CharSequence、Date、BigDecimal、および BigInteger はすべて、その toString （）表現に変換されます。その他のタイプはすべて破棄されます。

Camel は、特定のヘッダー値をドロップすると、DEBUG レベルでカテゴリー org.apache.camel.component.jms.JmsBinding でログに記録されます。以下に例を示します。

```
2008-07-09 06:43:04,046 [main           ] DEBUG JmsBinding
  - Ignoring non primitive header: order of class:
org.apache.camel.component.jms.issues.DummyOrder with value: DummyOrder{orderId=333,
itemId=4444, quantity=2}
```

受信時のメッセージ形式

Camel は、メッセージを受信すると、以下のプロパティーを エクスチェンジ に追加します。

プロパティー	タイプ	説明
org.apache.camel.jms.replyDestination	**javax.jms.Destination**	応答先。

Camel は、JMS メッセージを受信すると、以下の JMS プロパティーを In メッセージヘッダーに追加します。

ヘッダー	タイプ	説明
JMSCorrelationID	**文字列**	JMS 相関 ID。
JMSDeliveryMode	**int**	JMS 配信モード。
JMSDestination	**javax.jms.Destination**	JMS 宛先。
JMSExpiration	**long**	JMS の有効期限。
JMSMessageID	**文字列**	JMS 一意のメッセージ ID。
JMSPriority	**int**	JMS の優先度（最も低い優先度は 0、最も高い優先順位が 9）
JMSRedelivered	**boolean**	は、JMS メッセージ再配信です。
JMSReplyTo	**javax.jms.Destination**	JMS 応答から宛先。
JMSTimestamp	**long**	JMS タイムスタンプ。
JMSType	**文字列**	JMS タイプ。
JMSXGroupID	**文字列**	JMS グループ ID。

CAMEL を使用したメッセージおよび JMSREPLYTO の送受信

JMS コンポーネントは複雑で、場合によってはその動作に細心の注意を払う必要があります。そのため、これは検索する一部のエリア/動作の簡単な概要です。

Camel が JMSProducer を使用してメッセージを送信する場合、以下の条件をチェックします。

- *メッセージ交換パターン*
- ***JMSReplyTo** がエンドポイントまたはメッセージヘッダーに設定されているかどうか。*
- *次のオプションが JMS エンドポイントに設定されているかどうか：**disableReplyTo**、**procaitMessageQos**、**explicitQos**。*

これらはすべて複雑で、ユースケースをサポートするように理解し、設定できます。

JMSPRODUCER

***JmsProducer** は設定に応じて以下のように動作します。*

交換パターン	その他のオプション	説明
InOut	\-	Camel は応答を想定し、一時的な **JMSReplyTo** を設定し、メッセージの送信後に一時キューでリプライメッセージをリッスンするようになります。
InOut	**JMSReplyTo** が設定されている	Camel は応答を想定し、メッセージの送信後に、指定された **JMSReplyTo** キューでリプライメッセージをリッスンし始めます。
InOnly	\-	Camel はメッセージを送信し、応答を想定し **ません**。

InOnly	**JMSReplyTo** が設定されている	デフォルトでは、Camel は **JMSReplyTo** 宛先を破棄し、メッセージを送信する前に **JMSReplyTo** ヘッダーをクリアします。Camel はメッセージを送信し、応答を想定し **ません**。Camel はこれを **WARN** レベルでログに記録します(Camel 2.6 以降から **DEBUG** レベルに変更します)。**preserveMessageQuo=true** を使用して、**JMSReplyTo** を保持するよう Camel に指示することができます。すべての状況では、**JmsProducer** は応答を期待し **ない** ため、メッセージの送信後に続行します。

JMSCONSUMER

JmsConsumer は設定に応じて以下のように動作します。

交換パターン	その他のオプション	説明
InOut	\-	Camel は応答を **JMSReplyTo** キューに送信します。
InOnly	\-	パターンが InOnly であるため、Camel は応答を返しません。
\-	**disableReplyTo=true**	このオプションは応答を抑制します。

そのため、エクスチェンジに設定されたメッセージ交換パターンに注意してください。

ルートの途中で JMS 宛先にメッセージを送信する場合は、使用する交換パターンを指定できます。詳細は、Request Reply を参照してください。これは、InOnly メッセージを JMS トピックに送信する場合に便利です。

```
from("activemq:queue:in")
  .to("bean:validateOrder")
  .to(ExchangePattern.InOnly, "activemq:topic:order")
  .to("bean:handleOrder");
```

エンドポイントを再利用し、実行時に計算された異なる宛先に送信する

多くの異なる JMS 宛先にメッセージを送信する必要がある場合は、JMS エンドポイントを再利用し、メッセージヘッダーで実際の宛先を指定することが理にかなっています。これにより、Camel は同じエンドポイントを再利用できますが、異なる宛先に送信することができます。これにより、メモリーおよびスレッドリソースで作成されたエンドポイント数が大幅に削減されます。

以下のヘッダーで宛先を指定できます。

ヘッダー	タイプ	説明
CamelJmsDestination	**javax.jms.Destination**	宛先オブジェクト。
CamelJmsDestinationName	**文字列**	宛先名。

たとえば、以下のルートは、実行時に宛先を計算し、それを使用して JMS URL に表示される宛先を上書きする方法を示しています。

```
from("file://inbox")
  .to("bean:computeDestination")
  .to("activemq:queue:dummy");
```

キュー名 dummy はプレースホルダーです。JMS エンドポイント URL の一部として提供される必要がありますが、この例では無視されます。

computeDestination Bean で、以下のように CamelJmsDestinationName ヘッダーを設定して実際の宛先を指定します。

```
public void setJmsHeader(Exchange exchange) {
   String id = ....
   exchange.getIn().setHeader("CamelJmsDestinationName", "order:" + id");
}
```

その後、Camel はこのヘッダーを読み取り、エンドポイントに設定されたヘッダーではなく宛先として使用します。そのため、この例では、Camel は id の値が 2 であると仮定して、activemq:queue:order:2 にメッセージを送信します。

CamelJmsDestination と CamelJmsDestination Name ヘッダーの両方が設定されている場合、CamelJmsDestination が優先されます。JMS プロデューサーは、エクスチェンジから CamelJmsDestination ヘッダーと CamelJmsDestinationName ヘッダーの両方を削除し、ルート内

の誤ってループを回避するために作成された JMS メッセージに伝播しないことに注意してください（メッセージが別の JMS エンドポイントに転送されるシナリオ）。

異なる JMS プロバイダーの設定

以下のように、Spring XML で JMS プロバイダーを設定できます。

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <jmxAgent id="agent" disabled="true"/>
</camelContext>

<bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost?
broker.persistent=false&roker.useJmx=false"/>
    </bean>
  </property>
</bean>
```

基本的には、JMS コンポーネントインスタンスをいくつでも設定し、id 属性を 使用して一意の名前を指定できます。上記の例では、activemq コンポーネントを設定します。MQSeries、TibCo、BEA、Sonic などを設定するのも同様です。

名前付きの JMS コンポーネントを取得したら、URI を使用してそのコンポーネント内のエンドポイントを参照できます。たとえば、コンポーネント名 activemq の場合は、URI 形式 activemq:[queue:|topic:]destinationName を使用して宛先を参照できます。他のすべての JMS プロバイダーに同じ方法を使用できます。

これは、エンドポイント URI に使用するスキーム名の SpringContext からコンポーネントを遅延的にフェッチし、Component が エンドポイント URI を解決することで機能します。

JNDI を使用した CONNECTIONFACTORY の検索

J2EE コンテナーを使用している場合は、Spring で通常の < bean > メカニズムを使用するのではなく、JNDI を検索して JMS ConnectionFactory を見つける必要がある場合があります。これは、Spring のファクトリー Bean または新しい Spring XML namespace を使用して実行できます。以下に例を示します。

```
<bean id="weblogic" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory" ref="myConnectionFactory"/>
</bean>
```

```
<jee:jndi-lookup id="myConnectionFactory" jndi-name="jms/connectionFactory"/>
```

JNDI ルックアップの詳細 は、Spring リファレンスドキュメントの jee スキーマ を参照してください。

同時消費

JMS の一般的な要件は、アプリケーションの応答性を高めるために、複数のスレッドでメッセージを同時に消費することです。以下のように、concurrentConsumers オプションを設定して、JMS エンドポイントを提供するスレッドの数を指定できます。

```
from("jms:SomeQueue?concurrentConsumers=20").
  bean(MyClass.class);
```

このオプションは、以下のいずれかの方法で設定できます。

- JmsComponent で以下を行います。
- エンドポイント URI または
- JmsEndpoint で setConcurrentConsumers （） を直接呼び出す。

非同期コンシューマーを使用した同時消費

各同時コンシューマーは、現在のメッセージが完全に処理されたときに、JMS ブローカーから次に利用可能なメッセージのみを選択することに注意してください。オプション asyncConsumer=true を設定すると、コンシューマーが JMS キューから次のメッセージをピックアップし、以前のメッセージが非同期的に処理されます(Asynchronous Routing Engine)。詳細は、asyncConsumer のページ上部にある表を参照してください。

```
from("jms:SomeQueue?concurrentConsumers=20&asyncConsumer=true").
  bean(MyClass.class);
```

JMS でのリクエスト応答

Camel は JMS で のリクエスト応答 をサポートします。JMS キューにメッセージを送信する場合、エクスチェンジの MEP は InOut である必要があります。

Camel には、パフォーマンスとクラスター環境に影響を与える JMS 上でリクエスト/応答を設定するオプションが複数あります。以下の表は、オプションの概要を示しています。

オプション	パフォーマンス	Cluster	説明
Temporary	高速	はい	一時キューは応答キューとして使用され、Camel によって自動作成されます。これを使用するには、replyTo キュー名を指定し **ません**。また、任意で **replyToType=Temporary** を設定して、一時キューが使用されていることを示すことができます。

共有	Slow	はい	共有永続キューが応答キューとして使用されます。キューは事前に作成する必要がありますが、一部のブローカーは Apache ActiveMQ などのオンザフライで作成できます。これを使用するには、replyTo キュー名を指定する必要があります。また、オプションで **replyToType=Shared** を設定して、共有キューが使用されていることを示すことができます。共有キューは、この Camel アプリケーションを同時に実行する複数のノードを持つクラスター環境で使用できます。すべては同じ共有応答キューを使用します。これは、JMS メッセージセレクターが想定されるリプライメッセージの関連付けに使用されるため可能です。ただし、これはパフォーマンスに影響します。JMS メッセージセレクターは遅くなるため、**Temporary** または **Exclusive** キューほど高速ではありません。パフォーマンスを向上させるために、これを微調整する方法は、以下を参照してください。

排他的	高速	いいえ	排他的な永続キューは応答キューとして使用されます。キューは事前に作成する必要がありますが、一部のブローカーは Apache ActiveMQ などのオンザフライで作成できます。これを使用するには、replyTo キュー名を指定する必要があります。また、**replyTo** キュー名 **が** 設定されている場合は、**Shared** がデフォルトで使用されるため、Camel に対して排他的キューを使用するように **replyToType=Exclusive** を設定する必要があります。排他的な応答キューを使用する場合、JMS メッセージセレクターは使用され **ない** ため、他のアプリケーションはこのキューを使用しないでください。排他的キューは、この Camel アプリケーションが同時に実行しているクラスター環境で使用することは **できません**。応答キューが要求メッセージを送信したのと同じノードに戻るかどうかを制御することができないため、共有キューが JMS メッセージセレクターを使用してこれを確実に行うためです。各排他的応答キューをノードごとに一意の名前で設定している場合 **は**、クラスター環境でこれを実行できます。そのため、リプライメッセージは指定されたノードのそのキューに戻り、リプライメッセージを待機します。

concurrentConsumers	高速	はい	Camel 2.10.3: 使用中の同時メッセージリスナーを使用してメッセージを同時に処理できるようにします。**concurrentConsumers** および **maxConcurrentConsumers** オプションを使用して範囲を指定できます。**注記： Shared** 応答キューを使用すると、同時リスナーでは適切に機能しない可能性があるため、このオプションを注意して使用してください。
maxConcurrentConsumers	高速	はい	Camel 2.10.3: 使用中の同時メッセージリスナーを使用してメッセージを同時に処理できるようにします。**concurrentConsumers** および **maxConcurrentConsumers** オプションを使用して範囲を指定できます。**注記： Shared** 応答キューを使用すると、同時リスナーでは適切に機能しない可能性があるため、このオプションを注意して使用してください。

***JmsProducer** は **InOut** を検出し、使用される応答宛先を持つ **JMSReplyTo** ヘッダーを提供します。デフォルトでは、Camel は一時キューを使用しますが、エンドポイントで **replyTo** オプションを使用して、固定された応答キューを指定できます（以下の未指定の応答キューについて参照）。*

*Camel は応答キューをリッスンするコンシューマーを自動的に設定するため、何もし ない でください。このコンシューマーは、応答をリッスンする Spring **DefaultMessageListenerContainer** です。ただし、同時コンシューマー 1 つに固定されています。つまり、返信を処理するスレッドが 1 つしかないため、応答が順番に処理されます。応答をより迅速に処理する場合は、同時実行を使用する必要があります。ただし、**concurrentConsumer** オプション は使用しません。以下のルートに示されるように、代わりに Camel DSL からの スレッド を使用する必要があります。*

```
from(xxx)
.inOut().to("activemq:queue:foo")
.threads(5)
```

```
.to(yyy)
.to(zzz);
```

このルートでは、5 つのスレッドを持つスレッドプールを使用してリプライを 非同期 にルーティングするように Camel に指示します。

ヒント

スレッドを使用する代わりに、Camel 2.10.3 以上を使用する場合は concurrentConsumers オプションを使用します。詳細は以下を参照してください。

Camel 2.10.3 以降では、concurrentConsumers および maxConcurrentConsumers オプションを使用して、リスナーが同時スレッドを使用するように設定できます。これにより、以下のように Camel でこの設定を簡単に設定できます。

```
from(xxx)
.inOut().to("activemq:queue:foo?concurrentConsumers=5")
.to(yyy)
.to(zzz);
```

JMS 上でリクエスト応答し、共有固定応答キューを使用する

以下の例のように Request Reply over JMS を実行する際に、固定応答キューを使用する場合は注意してください。

```
from(xxx)
.inOut().to("activemq:queue:foo?replyTo=bar")
.to(yyy)
```

この例では、bar という名前の固定応答キューが使用されます。デフォルトでは、Camel は固定応答キューの使用時にキューが共有されていると仮定するため、JMSSelector を使用して想定されるリプライメッセージのみを選択します（例： JMSCorrelationID）。排他的な固定応答キューについては、次のセクションを参照してください。つまり、一時キューほど高速ではありません。receiveTimeout オプションを使用して、Camel が応答メッセージに対してプルする頻度をスピードアップできます。デフォルトでは 1000 ミリ秒です。そのため、これをより迅速に設定するには、以下のように 250 ミリ秒に設定して、1 秒あたり 4 回プルできます。

```
from(xxx)
.inOut().to("activemq:queue:foo?replyTo=bar&receiveTimeout=250")
.to(yyy)
```

これにより、Camel がメッセージブローカーにプル要求を送信するため、より多くのネットワーク

ラフィックが必要になることに注意してください。通常、可能な場合は一時キューを使用することが推奨されます。

JMS のリクエスト応答と排他的な固定応答キューの使用

Camel 2.9 以降で利用可能

上記の例では、Camel は bar という名前の固定応答キューが共有されていることを想定するため、JMSSelector を使用して想定するリプライメッセージのみを消費します。ただし、JMS selectos が遅くなるため、これを行う欠点があります。また、応答キューのコンシューマーは、新しい JMS セレクター ID での更新に時間がかかります。実際、receiveTimeout オプションがタイムアウトした場合にのみ更新されます。デフォルトは 1 秒です。そのため、理論的にはリプライメッセージの検出に約 1 秒かかる可能性があります。一方、固定応答キューが Camel 応答コンシューマーに排他的である場合は、JMS セレクターを使用しないようにすることができます。実際、一時キューを使用するほど高速です。そのため、Camel 2.9 以降では、以下の例のように応答キューが排他的であることを Camel に指示するために設定できる ReplyToType オプションが導入されました。

```
from(xxx)
.inOut().to("activemq:queue:foo?replyTo=bar&replyToType=Exclusive")
.to(yyy)
```

キューは、各エンドポイントおよびすべてのエンドポイント専用である必要があることに注意してください。そのため、2 つのルートがある場合は、以下の例のようにそれぞれ固有の応答キューが必要になります。

```
from(xxx)
.inOut().to("activemq:queue:foo?replyTo=bar&replyToType=Exclusive")
.to(yyy)

from(aaa)
.inOut().to("activemq:queue:order?replyTo=order.reply&replyToType=Exclusive")
.to(bbb)
```

クラスター環境で実行する場合は、同じことが当てはまります。次に、クラスターの各ノードは一意の応答キュー名を使用する必要があります。そうしないと、クラスター内の各ノードは、別のノードへの応答として意図されたメッセージが選択される可能性があります。クラスター環境では、代わりに共有応答キューを使用することが推奨されます。

送信側と受信側のクロックの同期

システム間でメッセージングを実行する場合は、システムがクロックを同期していることが望ましいです。たとえば、JMS メッセージを送信する場合、メッセージに時間(time to live)を設定できます。次に、受信側はこの値を検査し、メッセージがすでに期限切れであるかどうかを判断できるため、メッセージを消費して処理するのではなくドロップできます。ただし、これには、送信者と受信側の両方が

クロックを同期している必要があります。ActiveMQ を使用している場合は、タイムスタンププラグイン を使用してクロックを同期できます。

存続期間

同期されたクロックについては、上記の最初の読み取り。

Camel で JMS でリクエスト/リプライ(InOut)を実行すると、Camel は送信者側でタイムアウトを使用します。これは、requestTimeout オプションのデフォルト 20 秒です。より高い/小さい値を設定することで、これを制御できます。ただし、ライブタイムは、送信する JMS メッセージに引き続き設定されます。では、がシステム間でクロックを同期する必要があります。設定されていない場合、ライブ値を設定する時間を無効にすることができます。これは、Camel 2.8 以降から disableTimeToLive オプションを使用して実行できるようになりました。そのため、このオプションを disableTimeToLive=true に設定すると、Camel は JMS メッセージの送信時に存続時間を設定し ません。ただし、 リクエストのタイムアウトはアクティブな状態です。たとえば、JMS でリクエスト/リプライをライブで行なっても、Camel は引き続きタイムアウトを 20 秒(requestTimeout オプション)使用します。当然ながら、このオプションを設定することもできます。そのため、requestTimeout と disableTimeToLive の 2 つのオプションにより、リクエスト/リプライを行うときに詳細な制御が可能になります。

Camel 2.13/2.12.3 以降では、メッセージに ヘッダーを指定して、エンドポイントで設定された値の代わりに、リクエストタイムアウト値として上書きおよび使用できます。以下に例を示します。

```
from("direct:someWhere")
  .to("jms:queue:foo?replyTo=bar&requestTimeout=30s")
  .to("bean:processReply");
```

上記のルートでは、requestTimeout を 30 秒に設定したエンドポイントがあります。そのため、Camel は、その応答メッセージがバーキューに戻るまで 30 秒待機します。リプライメッセージが受信されない場合、エクスチェンジ に org.apache.camel.ExchangeTimedOutException が設定され、Camel はメッセージのルーティングを続行します。これは例外によって失敗し、Camel のエラーハンドラーが反応します。

メッセージのタイムアウト値ごとに使用する場合は、定数値 "CamelJmsRequestTimeout" のキー org.apache.camel.component.jms.JmsConstants#JMS_REQUEST_TIMEOUT でヘッダーを設定し、タイムアウト値を long 型に設定します。

たとえば、以下のように Bean を使用して、サービス Bean で whatIsTheTimeout メソッドを呼び出すなど、個別のメッセージごとにタイムアウト値を計算することができます。

```
from("direct:someWhere")
  .setHeader("CamelJmsRequestTimeout", method(ServiceBean.class, "whatIsTheTimeout"))
```

```
.to("jms:queue:foo?replyTo=bar&requestTimeout=30s")
.to("bean:processReply");
```

Camel を使用して JMS 経由で forget (InOut)を実行し、取得(InOut)を実行する場合、デフォルトで Camel はメッセージのライブ値を設定し ません。timeToLive オプションを使用して値を設定できます。たとえば、5 秒を指定するには、timeToLive=5000 を設定します。disableTimeToLive オプションを使用して、InOnly メッセージングでも存続時間を強制的に無効にすることができます。requestTimeout オプションは、InOnly メッセージングには使用されません。

TRANSACTED CONSUMPTION の有効化

一般的な要件は、トランザクションのキューから消費してから、Camel ルートを使用してメッセージを処理することです。これを実行するには、コンポーネント/エンドポイントに以下のプロパティーを設定するようにしてください。

- transacted = true
- TransactionManager = Transsaction Manager \- 通常は JmsTransactionManager

詳細は、Transactional Client EIP パターンを参照してください。

JMS 上のトランザクションおよびリクエスト応答

JMS で Request Reply を使用すると、1 つのトランザクションは使用できません。JMS はコミットが実行されるまでメッセージを送信しないため、サーバー側はトランザクションのコミットまで何も受信しません。そのため、Request Reply を使用するには、リクエストの送信後にトランザクションをコミットし、応答を受信するために別のトランザクションを使用する必要があります。

この問題に対処するために、JMS コンポーネントは異なるプロパティーを使用して一方向メッセージングおよび要求応答メッセージングに使用するトランザクションを指定します。

transacted プロパティーは、InOnly メッセージ交換 パターン (MEP)に のみ 適用されます。

transactedInOut プロパティーは、InOut (Request Reply)メッセージ交換 パターン (MEP)に適用されます。

Camel 2.10 以降で利用可能

コンポーネント/エンドポイントで以下のプロパティーを使用して、DMLC トランザクションセッション API を利用できます。

- *transacted = true*
- *lazyCreateTransactionManager = false*

これを実行する利点は、TransactionManager を設定せずにローカルトランザクションを使用する場合に cacheLevel 設定を有効にすることです。TransactionManager が設定されている場合、DMLC レベルでキャッシングは発生せず、プールされた接続ファクトリーに依存する必要があります。このような設定の詳細は、こちらおよび こちら を参照してください。

応答の遅れに JMSREPLYTO を使用する

Camel を JMS リスナーとして使用する場合、キー ReplyTo を持つ ReplyTo javax.jms.Destination オブジェクトの値で Exchange プロパティーを設定します。この 宛先 は、以下のように取得できま

す。

```
Destination replyDestination =
exchange.getIn().getHeader(JmsConstants.JMS_REPLY_DESTINATION, Destination.class);
```

その後、それを使用して通常の JMS または Camel を使用して応答を送信します。

```
// we need to pass in the JMS component, and in this sample we use ActiveMQ
JmsEndpoint endpoint = JmsEndpoint.newInstance(replyDestination, activeMQComponent);
// now we have the endpoint we can use regular Camel API to send a message to it
template.sendBody(endpoint, "Here is the late reply.");
```

応答を送信する別のソリューションは、送信時に同じエクスチェンジプロパティーに replyDestination オブジェクトを提供することです。その後、Camel はこのプロパティーを取得し、実際の宛先に使用します。ただし、エンドポイント URI にはダミーの宛先が含まれている必要があります。以下に例を示します。

```
// we pretend to send it to some non existing dummy queue
template.send("activemq:queue:dummy, new Processor() {
    public void process(Exchange exchange) throws Exception {
        // and here we override the destination with the ReplyTo destination object so the
message is sent to there instead of dummy
        exchange.getIn().setHeader(JmsConstants.JMS_DESTINATION, replyDestination);
        exchange.getIn().setBody("Here is the late reply.");
    }
}
```

リクエストタイムアウトの使用

以下の例では、リクエスト応答スタイルのメッセージ Exchange (requestBody method = InOutを使用)を Camel でさらに処理するために低速なキューに送り、返信応答を待ちます。http://camel.apache.org/request-reply.html

```
// send a in-out with a timeout for 5 sec
Object out = template.requestBody("activemq:queue:slow?requestTimeout=5000", "Hello
World");
```

JMS をエクスチェンジを保存する DEAD LETTER QUEUE として使用

通常、JMS をトランスポートとして使用する場合、ボディーとヘッダーのみをペイロードとして転送します。Dead Letter Channel で JMS を使用する場合は、JMS キューを Dead Letter Queue として使用する場合、通常は原因となる例外が JMS メッセージに保存されません。ただし、JMS デッドレターキューで transferExchange オプションを使用して、org.apache.camel.impl.DefaultExchangeHolder を保持する javax.jms.ObjectMessage としてエクスチェンジ 全体をキューに保存するよう Camel に指示することができます。これにより、Dead

Letter Queue から消費でき、キー Exchange.EXCEPTION_CAUGHT を使用してエクスチェンジプロパティーから例外を取得できます。以下のデモでは、以下のようになります。

```
// setup error handler to use JMS as queue and store the entire Exchange
errorHandler(deadLetterChannel("jms:queue:dead?transferExchange=true"));
```

その後、JMS キューから消費し、問題を分析できます。

```
from("jms:queue:dead").to("bean:myErrorAnalyzer");

// and in our bean
String body = exchange.getIn().getBody();
Exception cause = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class);
// the cause message is
String problem = cause.getMessage();
```

JMS を DEAD LETTER CHANNEL 保存エラーとしてのみ使用

JMS を使用して原因のエラーメッセージを保存したり、初期化できるカスタムボディーを保存したりできます。以下の例では、Message Translator EIP を使用して、JMS デッドレターキューに移動する前に、失敗したエクスチェンジの変換を行います。

```
// we sent it to a seda dead queue first
errorHandler(deadLetterChannel("seda:dead"));

// and on the seda dead queue we can do the custom transformation before its sent to the
JMS queue
from("seda:dead").transform(exceptionMessage()).to("jms:queue:dead");
```

ここでは、元の原因エラーメッセージのみを変換に保存します。ただし、任意の式を使用して 任意のものを送信できます。たとえば、Bean でメソッドを呼び出すか、カスタムプロセッサーを使用することができます。

INONLY メッセージの送信および JMSREPLYTO ヘッダーの維持

camel-jms を使用して JMS 宛先に送信する場合、プロデューサーは MEP を使用して InOnly または InOut メッセージングを検出します。ただし、InOnly メッセージを送信し、JMSReplyTo ヘッダーを保持する場合があります。これを実行するには、Camel に対してこれを保持するよう指示する必要があります。そうでないと、JMSReplyTo ヘッダーがドロップされます。

たとえば、InOnly メッセージを foo キューに送信し、bar キューを持つ JMSReplyTo を使用するには、以下を実行します。

```
template.send("activemq:queue:foo?preserveMessageQos=true", new Processor() {
  public void process(Exchange exchange) throws Exception {
    exchange.getIn().setBody("World");
    exchange.getIn().setHeader("JMSReplyTo", "bar");
  }
});
```

preserveMessageQos=true を使用して、JMSReplyTo ヘッダーを保持するよう Camel に指示することに注意してください。

宛先での JMS プロバイダーオプションの設定

IBM の WebSphere MQ などの一部の JMS プロバイダーには、JMS 宛先でオプションを設定する必要があります。たとえば、targetClient オプションを指定する必要がある場合があります。targetClient は Camel URI オプションではなく WebSphere MQ オプションであるため、以下のように JMS 宛先名で設定する必要があります。

```
...
.setHeader("CamelJmsDestinationName", constant("queue:///MY_QUEUE?targetClient=1"))
.to("wmq:queue:MY_QUEUE?useMessageIDAsCorrelationID=true");
```

一部のバージョンの WMQ は、宛先名でこのオプションを受け入れないため、以下のような例外が発生します。

```
com.ibm.msg.client.jms.DetailedJMSException: JMSCC0005: 指定された値
'MY_QUEUE?targetClient=1' is not allowed for 'XMSC_DESTINATION_NAME'
```

回避策として、カスタム DestinationResolver を使用します。

```
JmsComponent wmq = new JmsComponent(connectionFactory);

wmq.setDestinationResolver(new DestinationResolver(){
   public Destination resolveDestinationName(Session session, String destinationName,
boolean pubSubDomain) throws JMSException {
       MQQueueSession wmqSession = (MQQueueSession) session;
       return wmqSession.createQueue("queue:///" + destinationName + "?targetClient=1");
   }
});
```

関連項目

- ***Bean インテグレーション***

- ***JmsTemplate gotchas***

第85章 JMX

JMX コンポーネント

JMX コンポーネントを使用すると、コンシューマーは MBean の通知をサブスクライブできます。コンポーネントは、エクスチェンジを介して Notification オブジェクトを直接渡すか、このプロジェクトで提供されるスキーマに従って XML にシリアライズすることをサポートします。これはコンシューマーのみのコンポーネントです。プロデューサーの作成を試みると、例外が発生します。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

JBoss EAP コンテナーのコンテキストでは、JMX コンポーネントは以下のように JBoss EAP JMX サブシステムと統合します。

```
CamelContext camelctx =
contextFactory.createWildflyCamelContext(getClass().getClassLoader());
camelctx.addRoutes(new RouteBuilder() {
   @Override
   public void configure() throws Exception {
      String host = InetAddress.getLocalHost().getHostName();
      from("jmx:platform?format=raw&objectDomain=org.apache.camel&key.context=" + host +
"/system-context-1&key.type=routes&key.name=\"route1\"" +
      "&monitorType=counter&observedAttribute=ExchangesTotal&granularityPeriod=500").
      to("direct:end");
   }
});
camelctx.start();

ConsumerTemplate consumer = camelctx.createConsumerTemplate();
MonitorNotification notifcation = consumer.receiveBody("direct:end",
MonitorNotification.class);
Assert.assertEquals("ExchangesTotal", notifcation.getObservedAttribute());
```

URI 形式

コンポーネントは、以下の URI を使用してローカルプラットフォームの MBean サーバーに接続できます。

```
jmx://platform?options
```

リモート MBean サーバーの URL は、jmx: スキーム接頭辞の後に以下のように指定できます。

```
jmx:service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi?options
```

URI にクエリーオプションを追加するには、?option=value&option=value&.. の形式を使用します。

URI オプション

プロパティー	必須	デフォルト	説明
format		**xml**	メッセージのボディーの形式。**xml** または **raw** のいずれか。**xml** の場合、通知は XML にシリアライズされます。**raw** の場合、raw java オブジェクトはボディーとして設定されます。
password			リモート接続を確立するための認証情報。
objectDomain	はい		接続している MBean のドメイン。
objectName			接続している MBean の name キー。キーのリストのこのプロパティーのいずれかを指定する必要があります（ただし両方ではありません）。詳細は、「ObjectName コンストラクト」 を参照してください。
notificationFilter			**NotificationFilter** インターフェイスを実装する Bean への参照。***#beanID*** 構文は、レジストリー内の Bean を参照するために使用されます。

handback			通知の受信時にリスナーに渡す値。この値は **jmx.handback** メッセージヘッダーに配置されます。
testConnectionOnStartup		**true**	*Camel 2.11* true の場合、起動時に JMX 接続を確立できない場合にコンシューマーは例外を出力します。false の場合、コンシューマーは接続が確立されるまで x 秒ごとに JMX 接続を確立しようとします。ここで、x は設定された **reconnectDelay** です。
reconnectOnConnectionFailure		**false**	*Camel 2.11* true の場合、接続障害が発生したときにコンシューマーは JMX サーバーへの再接続を試みます。コンシューマーは接続が確立されるまで x 秒ごとに JMX 接続を再確立しようとします。x は設定された **reconnectDelay** に置き換えます。
reconnectDelay		**10**	*Camel 2.11* 初期接続の作成を再試行する前、または失われた接続を再接続するまで待機する秒数。

OBJECTNAME コンストラクト

URI には常に objectDomain プロパティーが必要です。さらに、URI には objectName または key で始まる 1 つ以上のプロパティーが含まれている必要があります。

NAME プロパティーの DOMAIN

objectName プロパティーが指定されると、以下のコンストラクターを使用して MBean の ObjectName インスタンスを構築します。

```
ObjectName(String domain, String key, String value)
```

前述のコンストラクターの key の値は name で、値は objectName プロパティーの値になります。

HASHTABLE のあるドメイン

```
ObjectName(String domain, Hashtable<String,String> table)
```

Hashtable は、key で始まるプロパティーを抽出することで構築されます。プロパティーには、Hashtable をビルドする前に、key 接頭辞が取り除かれます。これにより、URI に変数番号が含まれるプロパティーが含まれ、MBean を識別できるようになります。

例

```
from("jmx:platform?objectDomain=jmxExample&key.name=simpleBean").
        to("log:jmxEvent");
```

完全な例

JMX コンポーネントを使用した完全な例は、examples/camel-example-jmx ディレクトリーにあります。

MONITOR TYPE CONSUMER

Camel 2.8 では、JMX の一般的な ユースケースが、デプロイされた Bean の属性を監視するモニター Bean を作成することです。これには、JMX モニターを作成してデプロイするためにいくつかの Java コードを記述する必要があります。以下に例を示します。

```
CounterMonitor monitor = new CounterMonitor();
monitor.addObservedObject(makeObjectName("simpleBean"));
monitor.setObservedAttribute("MonitorNumber");
monitor.setNotify(true);
monitor.setInitThreshold(1);
monitor.setGranularityPeriod(500);
registerBean(monitor, makeObjectName("counter"));
monitor.start();
```

2.8 バージョンには、指定された objectName および属性のモニター Bean を自動的に作成し、登録する新しいタイプのコンシューマーが導入されました。追加のエンドポイント属性を使用すると、ユーザーは監視する属性、作成するモニターのタイプ、およびその他の必要なプロパティーを指定できま

す。上記のコードスニペットは、エンドポイントプロパティーのセットに分散されます。コンシューマーはこれらのプロパティーを使用して CounterMonitor を作成し、登録してから変更にサブスクライブします。すべての JMX モニタータイプがサポートされます。

例

```
from("jmx:platform?objectDomain=myDomain&objectName=simpleBean&" +
    "monitorType=counter&observedAttribute=MonitorNumber&initThreshold=1&" +
    "granularityPeriod=500").to("mock:sink");
```

上記の例では、新しい Monitor Bean が作成され、MonitorNumber 属性を監視するローカルの mbean サーバーに置かれます。simpleBeanモニター Bean とオプションの追加タイプを以下に示します。新たにデプロイされたモニター Bean は、コンシューマーが停止すると自動的にアンデプロイされます。

モニタータイプの URI オプション

プロパティー	type	適用先	description
monitorType	**enum**	all	カウンターの 1 つ (guage、string)
observedAttribute	**string**	all	観察される属性
granularityPeriod	**long**	all	監視される属性の粒度（ミリ秒単位）。JMX によると、デフォルトは 10 秒です。
initThreshold	**number**	counter	初期しきい値
offset	**number**	counter	オフセット値
modulus	**number**	counter	modulus 値
differenceMode	**boolean**	counter、gage	違いを報告する必要がある場合は true、実際の値の場合は false
notifyHigh	**boolean**	ゲージ	高通知のオン/オフスイッチ
notifyLow	**boolean**	ゲージ	通知の低/オフスイッチ
highThreshold	**number**	ゲージ	レポートが高い通知のしきい値

lowThreshold	**number**	ゲージ	低い notificaton レポートのしきい値
notifyDiffer	**boolean**	string	文字列が異なる場合に通知を発生させる場合は true
notifyMatch	**boolean**	string	true: 文字列が一致するときに通知を実行する
stringToCompare	**string**	string	属性値と比較する文字列

モニタースタイルのコンシューマーは、ローカルの mbean サーバーでのみサポートされます。現在、JMX は、プロキシーのデプロイメントを容易にするために、リモートでデプロイされたクラスや、クライアントとサーバーの両方にアダプターライブラリーを持たずに mbeans のリモートデプロイメントをサポートしていません。

第86章 JOLT

JOLT コンポーネント

Camel 2.16 以降で利用可能

Jolt コンポーネントを使用すると、JOLT 仕様を使用して JSON メッセージを処理できます。JSON 変換に JSON を実行する場合に適しています。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jolt</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
jolt:specName[?options]
```

specName は、呼び出す仕様のクラスパスローカル URI、またはリモート仕様の完全な URL （例：file://folder/myfile.json）です。

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

オプション

オプション	デフォルト	説明

allowContextMapAll (producer)	**false**	コンテキストマップが前詳細へのアクセスを許可するかどうかを設定します。デフォルトでは、メッセージの本文とヘッダーにのみアクセスできます。このオプションは、現在の Exchange および CamelContext へのフルアクセスに対して有効にできます。これを行うと、CamelContext API の全機能へのアクセスが開かれるため、潜在的なセキュリティーリスクが発生します。
allowTemplateFromHeader (producer)	**false**	ヘッダーのリソーステンプレートの使用を許可するかどうか (デフォルトは false)。**このオプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼されていないコンテンツやユーザー派生コンテンツが含まれている場合、これは最終的にエンドアプリケーションの信頼性および完全性に影響を与える可能性があるため、このオプションは注意して使用してください。**
contentCache	**true**	ロード時のリソースコンテンツのキャッシュ。注記：Camel 2.9 のキャッシュされたリソースコンテンツは、エンドポイントの **clearContentCache** 操作を使用して JMX 経由でクリアできます。
outputType	**Hydrated**	変換の出力を、List または Maps、または JSON を含む String オブジェクトとして Hydrated に設定します。
inputType	**Hydrated**	変換の入力を List または Maps、または JSON を含む String オブジェクトとして Hydrated に設定します。
transformDsl	**Chainr**	提供された仕様の読み込みに使用する 変換 DSL。利用可能な値 Chainr、Shiftr、Defaultr、Removr、および Sortr。

動的仕様

Camel は ヘッダーを提供し、仕様に異なるリソースの場所を定義できます。このヘッダーが設定されている場合、Camel は設定されたエンドポイントでこれを使用します。これにより、実行時に動的仕様を指定できます。

ヘッダー	タイプ	説明
CamelJoltResourceUri	文字列	設定されたエンドポイントの代わりに使用する仕様リソースの URI。

サンプル

たとえば、以下のようなものを使用できます。

```
from("activemq:My.Queue").
  to("jolt:com/acme/MyResponse.json");
```

また、ファイルベースのリソースは以下のようになります。

```
from("activemq:My.Queue").
  to("jolt:file://myfolder/MyResponse.json?contentCache=true").
  to("activemq:Another.Queue");
```

また、以下のように、ヘッダーを使用してコンポーネントを動的に使用する仕様を指定することもできます。

```
from("direct:in").
  setHeader("CamelJoltResourceUri").constant("path/to/my/spec.json").
  to("jolt:dummy?allowTemplateFromHeader=true");
```

警告

allowTemplateFromHeader オプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼できないコンテンツまたはユーザー派生コンテンツが含まれる場合、これは最終的に、エンドアプリケーションの確実性と整合性に及ぼす可能性があるため、このオプションを使用してください。

第87章 JPA

JPA コンポーネント

jpa コンポーネントを使用すると、EJB 3 の Java Persistence Architecture (JPA)を使用して永続ストレージから Java オブジェクトを保存および取得できます。JPA は、OpenJPA、Hibernate、TopLink などの Object/Relational Mapping (ORM)製品をラップする標準のインターフェイスレイヤーです。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

JBoss EAP コンテナーでの JPA コンポーネントの使用に関する詳細は、JPA との統合 を参照してください。

エンドポイントへの送信

Java エンティティー Bean を JPA プロデューサーエンドポイントに送信すると、Java エンティティー Bean をデータベースに保存できます。In メッセージのボディーは、エンティティー Bean （つまり、@Entity アノテーションを持つ POJO）またはコレクションまたはエンティティー Bean の配列であると想定されます。

ボディーがエンティティーのリストである場合は、entityType=java.util.ArrayList をプロデューサーエンドポイントに渡される設定として使用するようにしてください。

ボディーに前述のタイプの 1 つが含まれていない場合は、最初にエンドポイントの前に Message Translator を追加して、最初に必要な変換を実行します。

エンドポイントからの消費

JPA コンシューマーエンドポイントからメッセージを消費すると、データベース内のエンティティー Bean が削除される（または更新）されます。これにより、データベーステーブルを論理キューとして使用できます。コンシューマーはキューからメッセージを取得してから、それらを削除/更新してキューから論理的に削除できます。

エンティティー Bean が処理されたときに(ルーティングが完了したら)エンティティー Bean を削除したくない場合は、URI で consumeDelete=false を指定できます。これにより、エンティティーはポーリングごとに処理されます。

エンティティーでいくつかの更新を実行して処理済みとしてマークする場合(将来のクエリーから除外するなど)、メソッドに @Consumed のアノテーションを付け、エンティティー Bean の処理時にエンティティー Bean で呼び出されます(ルーティングの完了時)。

Camel 2.13 以降では、@PreConsumed を使用できます。これは、(ルーティングの前に)処理される前にエンティティー Bean で呼び出されます。

URI 形式

> *jpa:entityClassName[?options]*

エンドポイントに送信する場合、Entity ClassName *は任意です。指定した場合、Type Converter はボディーが正しいタイプであることを確認するのに役立ちます。*

*使用するには、*EntityClassName *は必須です。*

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
entityType	entityClassName	URI から entityClassName を上書きします。
persistenceUnit	**camel**	デフォルトで使用される JPA 永続ユニット。
consumeDelete	**true**	**JPA コンシューマーのみ: true** の場合、エンティティーは消費後に削除されます。**false** の場合、エンティティーは削除されません。

consumeLockEntity	**true**	JPA consumer only: ポーリングによる結果の処理中に、各エンティティー Bean に排他ロックを設定するかどうかを指定します。
flushOnSend	**true**	**JPA プロデューサーのみ：** エンティティー Bean が永続化された後に EntityManager をフラッシュします。
maximumResults	**-1**	**JPA コンシューマーのみ：** クエリーで取得する結果の最大数を設定し ます。
transactionManager	**null**	このオプションはレジストリーベースであり、指定された **transactionManager** を正しく検索できるように **#** 表記が必要です（例：**transactionManager=#myTransactionManager**使用するトランザクションマネージャーを指定します。指定のない場合、Apache Camel はデフォルトで **JpaTransactionManager** を使用します。JTA トランザクションマネージャーの設定に使用できます(EJB コンテナーとの統合)。
consumer.delay	**500**	**JPA コンシューマーのみ：** 各ポーリング間の遅延（ミリ秒単位）。
consumer.initialDelay	**1000**	**JPA コンシューマーのみ：** ポーリングの開始前の Milliseconds。
consumer.useFixedDelay	**false**	**JPA コンシューマーのみ：true** に設定すると、ポーリング間の固定遅延が使用されます。それ以外の場合は、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。

maxMessagesPerPoll	**0**	Apache Camel 2.0:JPA コンシューマーのみ： ポーリングごとに収集するメッセージの最大数を定義する整数値。デフォルトでは最大値は設定されていません。サーバーの起動時に多数のメッセージをポーリングしないようにするために使用できます。無効にするには 0 または負の値を設定します。
consumer.query		JPA コンシューマーのみ： データを消費するときにカスタムクエリーを使用します。
consumer.namedQuery		JPA コンシューマーのみ： データを消費する場合に名前付きクエリーを使用するには、以下を実行します。
consumer.nativeQuery		JPA コンシューマーのみ： データを消費するときにカスタムネイティブクエリーを使用するには、以下を実行します。
consumer.parameters		Camel 2.12: JPA コンシューマーのみ： クエリーの構築に使用されるパラメーターマップ。パラメーターは Map のインスタンスで、キーは String で、値は Object です。これは一般的なタイプ **java.util.Map<String, Object>** であることが期待されます。ここで、キーは指定の JPA クエリーの名前付きパラメーターで、値は選択する対応する有効な値です。
consumer.resultClass		Camel 2.7: JPA consumer only: 返されたペイロードのタイプを定義します(**entityManager.createNativeQuery (nativeQuery, resultClass)** は **entityManager.createNativeQuery (nativeQuery)**)。このオプションを指定しないと、オブジェクトアレイが返されます。データを消費するときにネイティブクエリーと併用する場合にのみ影響があります。

consumer.transacted	**false**	Camel 2.7.5/2.8.3/2.9: **JPA コンシューマーのみ：** トランザクションモードでコンシューマーを実行するかどうか。これにより、バッチ全体が処理されたときに、すべてのメッセージがコミットまたはロールバックされます。デフォルトの動作(false)は、以前に処理されたすべてのメッセージをコミットし、最後に失敗したメッセージのみをロールバックすることです。
consumer.lockModeType	**WRITE**	Camel 2.11.2/2.12: コンシューマーにロックモードを設定するには、以下を行います。使用できる値は enum **javax.persistence.LockModeType** で定義されます。Camel 2.13 **以降、デフォルト値は PESSIMISTIC_WRITE に変更されてい** ます。
consumer.SkipLockedEntity	**false**	Camel 2.13: ロック時に NOWAIT を使用し、エンティティーをサイレントにスキップするかどうかを設定します。
usePersist	**false**	Camel 2.5: JPA producer only: **entityManager.merge (entity)** の代わりに **entityManager.persist (entity)** を使用することを示します。注記：**entityManager.persist (entity)** は分離されたエンティティーに対して機能しません (EntityManager は INSERT クエリーの代わりに UPDATE を実行する必要があります)。
joinTransaction	**true**	Camel 2.12.3: camel-jpa は、Camel 2.12 以降からデフォルトでトランザクションに参加します。このオプションを使用してこれをオフにすることができます。たとえば、**LOCAL_RESOURCE** を使用し、トランザクションが JPA プロバイダーでは機能しません。このオプションは、すべてのエンドポイントに設定する代わりに、**JpaComponent** でグローバルに設定することもできます。

usePassedInEntityManager	**false**	**Camel 2.12.4/2.13.1:** JPA producer only。**true** の場合、Camel はコンポーネント/エンドポイントで設定されたエンティティーマネージャーではなく、ヘッダー **JpaConstants.ENTITYMANAGER** からの **EntityManager** を使用します。これにより、エンドユーザーは、使用するエンティティーマネージャーを制御できます。
sharedEntityManager	**false**	**Camel 2.16:** コンシューマー/プロデューサーに Spring の **SharedEntityManager** を使用するかどうか。エンティティーマネージャーを共有し、トランザクションの組み合わせが適切ではないため、このオプションが **true** の場合は、**joinTransaction=false** を設定することが推奨されます。

メッセージヘッダー

Apache Camel は以下のメッセージヘッダーをエクスチェンジに追加します。

ヘッダー	タイプ	説明
CamelEntityManager	**EntityManager**	**Camel 2.12: JPA consumer / Camel 2.12.2: JPA producer:** JpaConsumer または JpaProducer によって使用される JPA EntityManager オブジェクト。

ENTITYMANAGERFACTORY の設定

特定の EntityManagerFactory インスタンスを使用するように JPA コンポーネントを設定することを強く推奨します。これを行わないと、各 JpaEndpoint は独自の EntityManagerFactory インスタンスを自動作成します。たとえば、以下のように myEMFactory エンティティーマネージャーファクトリーを参照する JPA コンポーネントをインスタンス化できます。

```
<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent">
  <property name="entityManagerFactory" ref="myEMFactory"/>
</bean>
```

Camel 2.3 では、JpaComponent はレジストリーから EntityManagerFactory を自動的に検索します。つまり、上記のように JpaComponent でこれを設定する必要はありません。あいまいさがある場

合にのみこれを実行する必要があります。この場合、Camel は WARN をログに記録します。

TRANSACTIONMANAGER の設定

Camel 2.3 以降、JpaComponent はレジストリーから TransactionManager を自動的に検索します。Camel が登録された TransactionManager インスタンスが見つからない場合は、TransactionTemplate を検索し、そこから TransactionManager の抽出を試みます。レジストリーで利用可能な TransactionTemplate がない場合、JpaEndpoint は TransactionManager の独自のインスタンスを自動作成します。

TransactionManager の複数のインスタンスが見つかった場合、Camel は WARN メッセージをログに記録します。このような場合は、以下のように myTransactionManager トランザクションマネージャーを参照する JPA コンポーネントをインスタンス化して明示的に設定する必要がある場合があります。

```
<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent">
  <property name="entityManagerFactory" ref="myEMFactory"/>
  <property name="transactionManager" ref="myTransactionManager"/>
</bean>
```

名前付きクエリーでのコンシューマーの使用

選択したエンティティーのみを使用する場合は、consumer.namedQuery URI クエリーオプションを使用できます。まず、JPA Entity クラスで名前付きクエリーを定義する必要があります。

```
@Entity
@NamedQuery(name = "step1", query = "select x from MultiSteps x where x.step = 1")
public class MultiSteps {
  ...
}
```

その後、以下のようにコンシューマー URI を定義できます。

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.namedQuery=step1")
.to("bean:myBusinessLogic");
```

クエリーでのコンシューマーの使用

選択したエンティティーのみを使用する場合は、consumer.query URI クエリーオプションを使用できます。クエリーオプションを定義するだけで済みます。

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.query=select o from
org.apache.camel.examples.MultiSteps o where o.step = 1")
.to("bean:myBusinessLogic");
```

ネイティブクエリーでのコンシューマーの使用

選択したエンティティーのみを使用する場合は、consumer.nativeQuery URI クエリーオプションを使用できます。ネイティブクエリーオプションを定義するだけで済みます。

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.nativeQuery=select * from
MultiSteps where step = 1")
.to("bean:myBusinessLogic");
```

native クエリーオプションを使用すると、メッセージボディーにオブジェクトアレイを受け取ります。

例

JPA を使用してトレースされたメッセージをデータベースに保存する例は、トレーサーの例を参照してください。

JPA ベースのべき等リポジトリーの使用

このセクションでは、JPA ベースのべき等リポジトリーを使用します。

まず、persistence.xml ファイルで persistence-unit を設定する必要があります。

```
<persistence-unit name="idempotentDb" transaction-type="RESOURCE_LOCAL">
  <class>org.apache.camel.processor.idempotent.jpa.MessageProcessed</class>

  <properties>
    <property name="openjpa.ConnectionURL"
value="jdbc:derby:target/idempotentTest;create=true"/>
    <property name="openjpa.ConnectionDriverName"
value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema"/>
    <property name="openjpa.Log" value="DefaultLevel=WARN, Tool=INFO"/>
  </properties>
 </persistence-unit>
```

次に、org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository によって使用される org.springframework.orm.jpa.JpaTemplate を設定する必要があります。

```
<!-- this is standard spring JPA configuration -->
 <bean id="jpaTemplate" class="org.springframework.orm.jpa.JpaTemplate">
     <property name="entityManagerFactory" ref="entityManagerFactory"/>
 </bean>

 <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
     <!-- we use idempotentDB as the persitence unit name defined in the persistence.xml file --
>
     <property name="persistenceUnitName" value="idempotentDb"/>
 </bean>
```

その後、org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository を設定できます。

```
<!-- we define our jpa based idempotent repository we want to use in the file consumer -->
 <bean id="jpaStore"
class="org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository">
     <!-- Here we refer to the spring jpaTemplate -->
     <constructor-arg index="0" ref="jpaTemplate"/>
     <!-- This 2nd parameter is the name  (= a cateogry name).
          You can have different repositories with different names -->
     <constructor-arg index="1" value="FileConsumer"/>
 </bean>
```

最後に、Spring XML ファイルで JPA idempotent リポジトリーを作成することもできます。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="JpaMessageIdRepositoryTest">
        <from uri="direct:start" />
        <idempotentConsumer messageIdRepositoryRef="jpaStore">
            <header>messageId</header>
            <to uri="mock:result" />
        </idempotentConsumer>
    </route>
</camelContext>
```

第88章 JSCH

JSCH

camel-jsch コンポーネントは、Jsch プロジェクトの Client API を使用して SCP プロトコル をサポートします。jsch は、sftp: プロトコルの FTP コンポーネントによって camel ですでに使用されています。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jsch</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
scp://host[:port]/destination[?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

ファイル名は、URI の <path> 部分、またはメッセージの CamelFileName ヘッダーのいずれかで指定できます（コードで使用される場合はExchange.FILE_NAME ）。

オプション

名前	説明	例	デフォルト値
username	リモートファイルシステムへのログインに使用するユーザー名を指定します。		null

password	リモートファイルシステムへのログインに使用するパスワードを指定します。		**null**
knownHostsFile	scp エンドポイントがホストキーの検証を実行できるように **known_hosts** ファイルを設定します。		**null**
strictHostKeyChecking	厳密なホストキーチェックを使用するかどうかを設定します。使用できる値は **no**、**yes**です。		**いいえ**
chmod	保存されたファイルに chmod を設定できます。たとえば、**chmod=664** です。		**null**
useUserKnownHostsFile	Camel 2.15: **knownHostFile** が明示的に設定されていない場合は、**System.getProperty("user.home") + "/.ssh/known_hosts"** からのホストファイルを使用します。		**true**

コンポーネントのオプション

JschComponent は以下のオプションをサポートします。

名前	説明	デフォルト値
verboseLogging	Camel 2.15: JSCH は、追加設定なしで詳細なロギングです。したがって、デフォルトで **DEBUG** ロギングまでロギングを下げます。	**true**

制限事項

現在、camel-jsch は Producer （つまり、ファイルを別のホストにコピーする）のみをサポートします。

第89章 JT400

JT/400 コンポーネント

jt400 コンポーネントを使用すると、データキューを使用して AS/400 システムでメッセージを交換できます。

URI 形式

```
jt400://user:password@system/QSYS.LIB/LIBRARY.LIB/QUEUE.DTAQ[?options]
```

リモートプログラムを呼び出す(Camel 2.7)

```
jt400://user:password@system/QSYS.LIB/LIBRARY.LIB/program.PGM[?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

URI オプション

データキューのメッセージ交換の場合：

名前	デフォルト値	説明
ccsid	Default system CCSID	AS/400 システムとの接続に使用する CCSID を指定します。
format	**text**	有効なオプションを送信するためのデータ形式を指定します。**text** (**String**で表現)および **バイナリー** (**byte[]**で表されます)です。
consumer.delay	**500**	各ポーリングの遅延（ミリ秒単位）。
consumer.initialDelay	**1000**	ポーリングが開始するまでの時間（ミリ秒単位）。

consumer.userFixedDelay	**false**	**true**: ポーリング間の固定遅延を使用します。そうでない場合は、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。
guiAvailable	**false**	**Camel 2.8:** Camel を実行している環境で AS/400 プロンプトが有効かどうかを指定します。
キー付き	**false**	*Camel 2.10:* キー化されたデータキューまたはキー以外のデータキューを使用するかどうか。
searchKey	**null**	*Camel 2.10:* キーで主要なデータキューのキーを検索します。
searchType	**EQ**	*Camel 2.10:* **EQ**、**NE**、**LT**、**LE**、**GT**、または **GE** の値である検索タイプ。
connectionPool	**AS400ConnectionPool** インスタンス	*Camel 2.10:* レジストリーの com.ibm.as400.access.AS400ConnectionPool インスタンスへの参照。これは、AS/400 システムへの接続を取得するために使用されます。検索表記法('#' 文字)を使用する必要があります。
secured	**false**	**Camel 2.16:** AS/400 への SSL 接続を使用するかどうか。

リモートプログラム呼び出しの場合(Camel 2.7):

名前	デフォルト値	説明
outputFieldsIdx		出力パラメーターであるフィールド（プログラムパラメーター）を指定します。
fieldsLength		フィールド（プログラムパラメーター）の長さを AS/400 プログラム定義で指定します。

format	**text**	*Camel 2.10:* メッセージを送信するためのデータ形式を指定します： **text** (**文字列**で表される)および **バイナリー** (**byte[]**で表されます)
guiAvailable	**false**	*Camel 2.8:* Camel を実行している環境で AS/400 プロンプトが有効かどうかを指定します。
connectionPool	**AS400ConnectionPool** インスタンス	*Camel 2.10:* レジストリーの com.ibm.as400.access.AS400ConnectionPool インスタンスへの参照。これは、AS/400 システムへの接続を取得するために使用されます。検索表記法('#' 文字)を使用する必要があります。

使用方法

コンシューマーエンドポイントとして設定されている場合、エンドポイントはリモートシステムのデータキューをポーリングします。データキューのすべてのエントリーについて、フォーマットに応じて、In メッセージのボディーのエントリーのデータで新しい エクスチェンジ が送信され、String または byte[] に分類されます。プロバイダーエンドポイントの場合、In メッセージボディーの内容は raw バイトまたはテキストとしてデータキューに配置されます。

接続プール

Camel 2.10 以降で利用可能

接続プールは Camel 2.10 以降で使用されます。Jt400Component で接続プールを明示的に設定することも、エンドポイントの uri オプションとして明示的に設定できます。

リモートプログラム呼び出し(CAMEL 2.7)

このエンドポイントは、入力が String 配列または byte[] 配列のいずれかであることを想定し、ネイティブ jt400 ライブラリーメカニズムを介してすべての CCSID 処理を処理します。パラメーターは、位置の値として null を渡すと 省略 できます（リモートプログラムではサポートする必要があります）。プログラムの実行後、エンドポイントはプログラムによって返された値で String 配列または

byte[] 配列を返します（入力のみのパラメーターのみのパラメーターには、呼び出しの最初のデータと同じデータが含まれます）。このエンドポイントはプロバイダーエンドポイントを実装しません。

例

以下のスニペットでは、direct:george エンドポイントに送信されたエクスチェンジのデータは、LIVERPOOL という名前のシステムのライブラリー BEATLES のデータキュー PENNYLANE に配置されます。別のユーザーは同じデータキューに接続してデータキューから情報を受け取り、これを mock:ringo エンドポイントに転送します。

```
public class Jt400RouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

from("direct:george").to("jt400://GEORGE:EGROEG@LIVERPOOL/QSYS.LIB/BEATLES.LIB/PE
NNYLANE.DTAQ");

from("jt400://RINGO:OGNIR@LIVERPOOL/QSYS.LIB/BEATLES.LIB/PENNYLANE.DTAQ").to("m
ock:ringo");
    }
}
```

リモートプログラム呼び出しの例(CAMEL 2.7)

以下のスニペットでは、direct:work エンドポイントに送信されたデータ Exchange には、ライブラリー assets のプログラム compute の引数として使用される 3 つの文字列が含まれます。このプログラムは、2nd パラメーターおよび 3rd パラメーターに出力値を書き込みます。すべてのパラメーターは direct:play エンドポイントに送信されます。

```
public class Jt400RouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {

from("direct:work").to("jt400://GRUPO:ATWORK@server/QSYS.LIB/assets.LIB/compute.PGM?
fieldsLength=10,10,512&ouputFieldsIdx=2,3").to("direct:play");
    }
}
```

キーデータキューへの書き込み

```
from("jms:queue:input")
.to("jt400://username:password@system/lib.lib/MSGINDQ.DTAQ?keyed=true");
```

キーデータキューからの読み取り

```
from("jt400://username:password@system/lib.lib/MSGOUTDQ.DTAQ?
keyed=true&searchKey=MYKEY&searchType=GE")
.to("jms:queue:output");
```

第90章 KAFKA

KAFKA コンポーネント

Camel 2.13 で利用可能

kafka: コンポーネントは、Apache Kafka メッセージブローカーとの通信に使用されます。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-kafka</artifactId>
   <version>2.17.0.redhat-630xxx</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

Camel 2.17 以降： Kafka Java クライアントを使用するため、Scala は使用されなくなりました。

Camel 2.16 以前： 選択した Scala ライブラリーの Maven 依存関係も追加する必要があります。camel-kafka はその依存関係を含みませんが、提供を想定します。たとえば、Scala 2.10.4 を使用するには、以下を追加します。

```
<dependency>
   <groupId>org.scala-lang</groupId>
   <artifactId>scala-library</artifactId>
   <version>2.10.4</version>
</dependency>
```

URI 形式

```
kafka:server:port[?options]
```

オプション

プロパティー	デフォルト	説明
zookeeperHost		使用する zookeeper ホスト

zookeeperPort	2181	使用する zookeeper ポート
zookeeperConnect		Camel 2.13.3/2.14.1: 使用している場合は、zookeeperHost/zookeeperPort は使用されません。
topic		使用するトピック
groupId		
partitioner		
consumerStreams	10	
clientId		
zookeeperSessionTimeoutMs		
zookeeperConnectionTimeoutMs		
zookeeperSyncTimeMs		
consumersCount	**1**	Camel 2.15.0: Kafka サーバーに接続するコンシューマーの数。
batchSize	**100**	Camel 2.15.0: **BatchingConsumerTask** が 1 度処理する **batchSize**。
barrierAwaitTimeoutMs	**10000**	Camel 2.15.0: **BatchingConsumerTask** のエクスチェンジが **batchSize** を超える場合は、**barrierAwaitTimeoutMs** まで待機します。
bridgeEndpoint	**false**	Camel 2.16.0: **bridgeEndpoint** が **true** の場合、プロデューサーはメッセージのトピックヘッダー設定を無視します。

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

プロデューサーオプション

プロパティー	デフォルト	説明
producerType	**sync**	以下の値を使用できます。 ● **sync** : メッセージ/バッチを即座に送信し、応答を受け取るまで待機します。 ● **async** : 送信するメッセージ/バッチをキューに入れます。**queueBufferingMaxMs** または **batchNumMessages** でこのキューからポーリングするブローカー (Kafka ノード)ごとにスレッドがあります。
compressionCodec		
compressedTopics		
messageSendMaxRetries		
retryBackoffMs		
topicMetadataRefreshIntervalMs		
sendBufferBytes		
requestRequiredAcks		
requestTimeoutMs		
queueBufferingMaxMs		
queueBufferingMaxMessages		
queueEnqueueTimeoutMs		
batchNumMessages		
serializerClass		
keySerializerClass		

コンシューマーオプション

プロパティー	デフォルト	説明
consumerId		
socketTimeoutMs		
socketReceiveBufferBytes		
fetchMessageMaxBytes		
autoCommitEnable		
autoCommitIntervalMs		
queuedMaxMessages		
rebalanceMaxRetries		
fetchMinBytes		
fetchWaitMaxMs		
rebalanceBackoffMs		
refreshLeaderBackoffMs		
autoOffsetReset		
consumerTimeoutMs		

サンプル

メッセージの使用：

```
from("kafka:localhost:9092?
topic=test&zookeeperHost=localhost&zookeeperPort=2181&groupId=group1").to("log:input");
```

メッセージを生成します。

その他の例は、camel-kafka のユニットテスト を参照してください。

エンドポイント

Camel は、Endpoint インターフェイスを使用した Message Endpoint パターンをサポートします。エンドポイントは通常コンポーネントによって 作成 され、エンドポイントは通常 URI を介して DSL で参照され ます。

エンドポイントから以下のメソッドを使用できます。

- *createProducer （） は、メッセージエクスチェンジをエンドポイントに送信するために Producer を作成します。*
- *createConsumer （） は、Consumer の作成時に Processor 経由でエンドポイントからメッセージエクスチェンジを消費するために Event Driven Consumerパターンを実装します。*
- *createPollingConsumer （） は、PollingConsumer経由でエンドポイントからメッセージエクスチェンジを消費するために Polling Consumer パターンを実装します。*

関連項目

- *Configuring Camel (Camel の設定)*
- *Message Endpoint パターン*
- *URI*
- *コンポーネントの作成*

第91章 KESTREL

KESTREL コンポーネント

Kestrel コンポーネントを使用すると、Kestrel キューへメッセージを送信したり、Kestrel キューからメッセージを消費したりできます。このコンポーネントは、Kestrel サーバーとの memcached プロトコル通信に spymemcached クライアントを使用します。

警告

そのため、Kestrel プロジェクトは非アクティブであるため、このコンポーネントは が 非推奨になりました。

URI 形式

```
kestrel://[addresslist/]queuename[?options]
```

queuename は Kestrel 上のキューの名前です。URI の addresslist 部分には、1 つ以上の host:port ペアが含まれる場合があります。たとえば、kserver01:22133 のキュー foo に接続するには、以下を使用します。

```
kestrel://kserver01:22133/foo
```

addresslist を省略すると、localhost:22133 が想定されます。つまり、以下のようになります。

```
kestrel://foo
```

同様に、addresslist の host:port ペアからポートを省略すると、デフォルトのポート 22133 が想定されます。以下に例を示します。

```
kestrel://kserver01/foo
```

*以下は、クラスター化されたキューの生成に使用される **Kestrel** エンドポイント **URI** の例です。*

> ***kestrel://kserver01:22133,kserver02:22133,kserver03:22133/massive***

*以下は、キューから同時に消費するために使用される **Kestrel** エンドポイント **URI** の例です。*

> ***kestrel://kserver03:22133/massive?concurrentConsumers=25&waitTimeMs=500***

オプション

*各 **Kestrel** エンドポイントでプロパティーを個別に設定するには、エンドポイント **URI** の **?parameters** の部分でプロパティーを指定します。省略された **?parameters** はデフォルトで **KestrelComponent** のベース **KestrelelConfiguration** で設定される内容に設定されます。以下のプロパティーは **KestrelConfiguration** や各エンドポイントに設定できます。*

オプション	デフォルト値	説明
concurrentConsumers	**1**	同時コンシューマースレッドの数を指定します。
waitTimeMs	**100**	**/t=...** を指定します。GET リクエストで Kestrel に渡される待機時間を指定します。

*注記： **waitTimeMs** がゼロ（または負の値）に設定されている場合、**/t=...** 指定子は **GET** 要求時にサーバーに渡され ません。キューが空の場合、**GET** 呼び出しは値なしで即座に返します。ポーリングフェーズで **"tight looping"** が発生しないように、このコンポーネントは、**GET** リクエストから何も返されない場合は常に **Thread.sleep (100)**を実行します（何も返されない場合のみ）。**waitTimeMs** に正の値以外の値を設定することを強く推奨 します。*

***SPRING XML** を使用した **KESTREL** コンポーネントの設定*

明示的な設定の最も単純な形式は次のとおりです。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <bean id="kestrel" class="org.apache.camel.component.kestrel.KestrelComponent"/>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
  </camelContext>

</beans>
```

これにより、すべてのデフォルト設定で Kestrel コンポーネントが有効になります。つまり、デフォルトで localhost:22133、100ms の待機時間、および同時でないコンシューマー 1 つを使用します。

ベース設定で特定のオプション(?properties が指定されていないエンドポイントに設定を提供する) を使用するには、以下のように KestrelConfiguration POJO を設定します。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <bean id="kestrelConfiguration"
class="org.apache.camel.component.kestrel.KestrelConfiguration">
    <property name="addresses" value="kestrel01:22133"/>
    <property name="waitTimeMs" value="100"/>
    <property name="concurrentConsumers" value="1"/>
  </bean>

  <bean id="kestrel" class="org.apache.camel.component.kestrel.KestrelComponent">
    <property name="configuration" ref="kestrelConfiguration"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
  </camelContext>

</beans>
```

使用例

例 1: 消費

```
from("kestrel://kserver02:22133/massive?concurrentConsumers=10&waitTimeMs=500")
 .bean("myConsumer", "onMessage");
```

```
public class MyConsumer {
    public void onMessage(String message) {
        ...
    }
}
```

例 2: 生成

```
public class MyProducer {
    @EndpointInject(uri = "kestrel://kserver01:22133,kserver02:22133/myqueue")
    ProducerTemplate producerTemplate;

    public void produceSomething() {
        producerTemplate.sendBody("Hello, world.");
    }
}
```

例 3: SPRING XML 設定

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="kestrel://ks01:22133/sequential?concurrentConsumers=1&waitTimeMs=500"/>
    <bean ref="myBean" method="onMessage"/>
  </route>
  <route>
    <from uri="direct:start"/>
    <to uri="kestrel://ks02:22133/stuff"/>
  </route>
</camelContext>
```

```
public class MyBean {
    public void onMessage(String message) {
        ...
    }
}
```

DEPENDENCIES

Kestrel コンポーネントには、以下の依存関係があります。

- Spymemcached 2.5 （以上）

SPYMEMCACHED

クラスパスに spymemcached jar が 必要 です。以下は、pom.xml で使用できるスニペットです。

```
<dependency>
  <groupId>spy</groupId>
  <artifactId>memcached</artifactId>
  <version>2.5</version>
</dependency>
```

または、jar を直接ダウンロード できます。

制限事項

注記： JVM アサーションが有効になっていると、spymemcached クライアントライブラリーが kestrel で適切に動作し ません。アサーションが有効で、要求されたキーに /t=... エクステンションが含まれる場合に spymemcached には既知の問題があります（例：エンドポイント URI で waitTimeMs オプションを使用している場合は、強く推奨されます）。

ただし、JVM アサーションを 明示的に有効にしない限り、JVM アサーションはデフォルトで無効 に なっているため、通常の状況では問題は発生しません。

注意すべき点は、Maven の Surefire テストプラグインがアサーションを 有効にすることです。Maven テスト環境でこのコンポーネントを使用している場合は、enableAssertions を false に設定する必要がある場合があります。詳細は、surefire:test を参照 してください。

第92章 KRATI

KRATI コンポーネント

Camel 2.9 以降で利用可能

このコンポーネントでは、Camel 内で krati データストアとデータセットを使用できます。Krati は、レイテンシーが非常に低く、スループットが高いシンプルな永続データストアです。これは、設定、パフォーマンス、および JVM ガベージコレクションの調整をほとんどせずに、読み取り/書き込み集約型アプリケーションとの統合を容易にするように設計されています。

Camel は、krati datastore_(key/value engine)_ のプロデューサーとコンシューマーを提供します。また、重複したメッセージをフィルターリングするべき等リポジトリーも提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-krati</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
krati:[the path of the datastore][?options]
```

データストアのパス は、krati がデータストアに使用するフォルダーの相対パスです。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

KRATI URI オプション

名前	デフォルト値	説明
operation	CamelKratiPut	**プロデューサーのみ**データストアに対して実行される操作のタイプを指定します。許可される値は CamelKratiPut、CamelKratiGet、CamelKratiDelete、および CamelKratiDeleteAll です。
initialCapacity	100	ストアの inital capcity。
keySerializer	KratiDefaultSerializer	キーのシリアライズに使用されるシリアライザーシリアライザー。
valueSerializer	KratiDefaultSerializer	値をシリアライズするために使用されるシリアライザーシリアライザー。
segmentFactory	ChannelSegmentFactory	使用するセグメントファクトリー。許可されるインスタンスクラス：ChannelSegmentFactory、MemorySegmentFactory、MappedSegmentFactory および WriteBufferSegmentFactory。
hashFunction	FnvHashFunction	使用するハッシュ関数。許可されるインスタンスクラス：FnvHashFunction,Fnv1Hash32,FnvHash64,Fnv1aHash32,Fnv1aHash64, JenkisHashFunction, MurmurHashFunctiom
maxMessagesPerPoll		**Camel 2.10.5/2.11.1:** 1 回のポーリングで受信できるメッセージの最大数。これを使用して、大量のデータの読み取りを回避し、メモリーを過剰に消費することができます。

プロデューサーエンドポイントの場合、適切なヘッダーをメッセージに渡すことで、上記の URI オプションをすべて上書きできます。

データストアのメッセージヘッダー

ヘッダー	説明
CamelKratiOperation	データストアで実行される操作。有効なオプションは です。 • **CamelKratiAdd** • **CamelKratiGet** • **CamelKratiDelete** • **CamelKratiDeleteAll**
CamelKratiKey	キー。
CamelKratiValue	値。

使用例

例 1: データストアへの配置。

この例では、データストア内に任意のメッセージを保存する方法を示します。

```
from("direct:put").to("krati:target/test/producertest");
```

上記の例では、メッセージのヘッダーを使用して、任意の URI パラメーターを上書きできます。上記の例は、xml を使用してルートを定義する方法を示しています。

```
<route>
    <from uri="direct:put"/>
    <to uri="krati:target/test/producerspringtest"/>
</route>
```

例 2: データストアの取得/読み取り

この例では、データストアのコントラネットを読み取る方法を説明します。

```
from("direct:get")
    .setHeader(KratiConstants.KRATI_OPERATION,
constant(KratiConstants.KRATI_OPERATION_GET))
    .to("krati:target/test/producertest");
```

上記の例では、メッセージのヘッダーを使用して、任意の URI パラメーターを上書きできます。上記の例は、xml を使用してルートを定義する方法を示しています。

```
<route>
     <from uri="direct:get"/>
     <to uri="krati:target/test/producerspringtest?operation=CamelKratiGet"/>
</route>
```

例 3: データストアの使用

この例では、指定されたデータストアの下にあるすべてのアイテムを消費します。

```
    from("krati:target/test/consumertest")
        .to("direct:next");
```

以下に示すように、xml を使用して同じゴールを実現できます。

```
<route>
    <from uri="krati:target/test/consumerspringtest"/>
    <to uri="mock:results"/>
</route>
```

IDEMPOTENT リポジトリー

前述のように、このコンポーネントは重複メッセージのフィルターリングに使用できるリポジトリーとべき等リポジトリーも提供します。

```
from("direct://in").idempotentConsumer(header("messageId"), new
KratiIdempotentRepositroy("/tmp/idempotent").to("log://out");
```

その他の参考資料

Krati Websitre

第93章 KUBERNETES

KUBERNETES コンポーネント

Camel 2.17 以降で利用可能

Kubernetes コンポーネントは、アプリケーションを Kubernetes スタンドアロンまたは Openshift 上に統合するためのコンポーネントです。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-kubernetes</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
kubernetes:masterUrl[?options]
```

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

オプション

名前	デフォルト値	説明
masterUrl	**null**	**必須：** Kubernetes マスター URL
category		**必須**: プロデューサー/コンシューマーのカテゴリー。可能な値：**namespaces**, **services**, **replicationControllers**, **pods**, ----------|-----, NORMAL, --------|-----, NORMAL, --------|-----, NORMAL, **persistentVolumes** **persistentVolumesClaims** **secrets** **resourcesQuota** **serviceAccounts** **nodes** **builds** **buildConfigs**

username		Kubernetes クラスターにログインするためのユーザー名
password		Kubernetes クラスターにログインするためのパスワード
operation		**プロデューサーのみ：** プロデューサーが実行する操作。以下の値を使用できます。 **listNamespaces**, **listNamespacesByLabels**, **getNamespace**, **createNamespace**, **deleteNamespace**, **listServices**, **listServicesByLabels**, **getService**, **createService**, **deleteService**, **listReplicationControllers**, **listReplicationControllersByLabels**, **getReplicationController**, **createReplicationController**, **deleteReplicationController**, **listPods**, **listPodsByLabels**, **getPod**, **createPod**, **deletePod**, **listPersistentVolumes**, **listPersistentVolumesByLabels**, **getPersistentVolume**, **listPersistentVolumesClaims**, **listPersistentVolumesClaimsByLabels**, **getPersistentVolumeClaim**, **createPersistentVolumeClaim**, **deletePersistentVolumeClaim**, **listSecrets**, **listSecretsByLabels**, **getSecret**, **createSecret**, **deleteSecret**, **listResourcesQuota**, **listResourcesQuotaByLabels**, **getResourceQuota**, **createResourceQuota**, **deleteResourceQuota**, **listServiceAccounts**, **listServiceAccountsByLabels**, **getServiceAccount**, **createServiceAccount**, **deleteServiceAccount**, **listNodes**, **listNodesByLabels**, **getNode**, **listBuilds**, **listBuildsByLabels**, **getBuild**, **listBuildConfigs**, **listBuildConfigsByLabels**, **getBuildConfig**
apiVersion		使用する API バージョン

caCertFile		CA 証明書ファイルへのパス
caCertData		CA 証明書データへのパス
clientCertFile		クライアント証明書ファイルへのパス
clientCertData		クライアント証明書データへのパス
clientKeyAlgo		クライアントが使用する鍵アルゴリズム
clientKeyFile		クライアントキーファイルへのパス
clientKeyData		クライアントキーデータへのパス
clientKeyPassphrase		クライアント鍵のパスフレーズ
oauthToken		認証トークン
trustCerts		証明書がデフォルトで信頼されるかどうかを定義します。
namespaceName		**コンシューマーのみ：** コンシューマーが監視する namespace
poolSize		**コンシューマーのみ：** Kubernetes コンシューマーの Threadpool サイズ

HEADERS

名前	タイプ	説明
CamelKubernetesOperation	**String**	Producer 操作
CamelKubernetesNamespaceName	文字列	namespace 名
CamelKubernetesNamespaceLabels	マップ	namespace ラベル
CamelKubernetesServiceLabels	マップ	サービスラベル
CamelKubernetesServiceName	文字列	サービス名
CamelKubernetesServiceSpec	io.fabric8.kubernetes.api.model.ServiceSpec	サービスの仕様

CamelKubernetesReplicationControllersLabels	マップ	レプリケーションコントローラーラベル
CamelKubernetesReplicationControllerName	文字列	レプリケーションコントローラー名
CamelKubernetesReplicationControllerSpec	io.fabric8.kubernetes.api.model.ReplicationControllerSpec	レプリケーションコントローラーの仕様
CamelKubernetesPodsLabels	マップ	Pod のラベル
CamelKubernetesPodName	文字列	Pod の名前
CamelKubernetesPodSpec	**io.fabric8.kubernetes.api.model.PodSpec**	Pod の仕様
CamelKubernetesPersistentVolumesLabels	マップ	永続ボリュームのラベル
CamelKubernetesPersistentVolumesName	文字列	永続ボリューム名
CamelKubernetesPersistentVolumesClaimsLabels	マップ	Persistent Volume Claim （永続ボリューム要求、PVC）ラベル
CamelKubernetesPersistentVolumesClaimsName	文字列	永続ボリューム要求 (PVC) の名前
CamelKubernetesPersistentVolumesClaimsSpec	**io.fabric8.kubernetes.api.model.PersistentVolumeClaimSpec**	Persistent Volume Claim （永続ボリューム要求、PVC）の仕様
CamelKubernetesSecretsLabels	マップ	シークレットラベル
CamelKubernetesSecretsName	文字列	Secret 名
CamelKubernetesSecret	io.fabric8.kubernetes.api.model.Secret	Secret オブジェクト
CamelKubernetesResourcesQuotaLabels	マップ	リソースクォータのラベル
CamelKubernetesResourcesQuotaName	文字列	リソースクォータ名
CamelKubernetesResourceQuotaSpec	**io.fabric8.kubernetes.api.model.ResourceQuotaSpec**	リソースクォータの仕様
CamelKubernetesServiceAccountsLabels	マップ	サービスアカウントラベル
CamelKubernetesServiceAccountName	文字列	サービスアカウント名

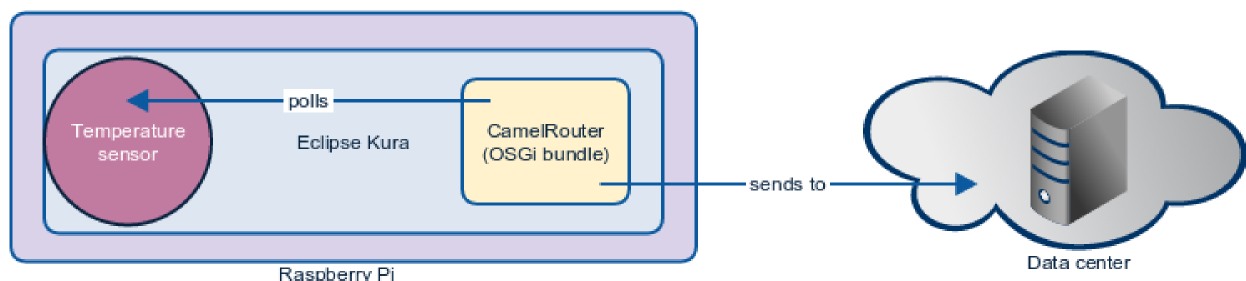
CamelKubernetesServiceAccount	**io.fabric8.kubernetes.api.model.ServiceAccount**	Service Account オブジェクト
CamelKubernetesNodesLabels	マップ	ノードラベル
CamelKubernetesNodeName	**String**	ノード名
CamelKubernetesBuildsLabels	**Map**	OpenShift ビルドラベル
CamelKubernetesBuildName	**String**	OpenShift ビルド名
CamelKubernetesBuildConfigsLabels	**Map**	OpenShift ビルド設定ラベル
CamelKubernetesBuildConfigName	文字列	OpenShift ビルド設定名
CamelKubernetesEventAction	io.fabric8.kubernetes.client.Watcher.Action	コンシューマーによって監視されるアクション
CamelKubernetesEventTimestamp	文字列	コンシューマーによって監視されるアクションのタイムスタンプ

第94章 KURA

KURA コンポーネント

Kura コンポーネントは、Camel 2.15 以降で利用できます。

このドキュメントページでは、Camel と Eclipse Kura M2M ゲートウェイの統合オプションについて説明します。Camel ルートを Eclipse Kura にデプロイする一般的な理由は、エンタープライズ統合パターンと Camel コンポーネントをメッセージング M2M ゲートウェイに提供することです。たとえば、Rusberry PI に Kura をインストールし、Kura サービスを使用してその Raspberry PI に接続されているセンサーから温度を読み、最後に Camel EIP およびコンポーネントを使用して現在の温度値をデータセンターサービスに転送する場合があります。



KURAROUTER ACTIVATOR

Eclipse Kura にデプロイされたバンドルは通常、バンドルアクティベーターとして開発され ます。そのため、Apache Camel ルートを Kura にデプロイする最も簡単な方法は、拡張 org.apache.camel.kura.KuraRouter クラスを含む OSGi バンドルを作成することです。

```
public class MyKuraRouter extends KuraRouter {

  @Override
  public void configure() throws Exception {
    from("timer:trigger").
      to("netty-http:http://app.mydatacenter.com/api");
  }

}
```

KuraRouter は org.osgi.framework.BundleActivator インターフェイスを実装しているため、Kura バンドルコンポーネントクラス start および stop のライフサイクルメソッドを登録する必要があります。

Kura ルーターは独自の OSGi 対応の CamelContext を起動します。これは、KuraRouter を拡張するすべてのクラスに対して、専用の CamelContext インスタンスがあることを意味します。理想的には、OSGi バンドルごとに KuraRouter をデプロイすることが推奨されます。

KURAROUTER のデプロイ

Kura ルータークラスを含むバンドルは、OSGi マニフェストに以下のパッケージをインポートする必要があります。

```
Import-Package: org.osgi.framework;version="1.3.0",
  org.slf4j;version="1.6.4",

org.apache.camel,org.apache.camel.impl,org.apache.camel.core.osgi,org.apache.camel.builde
r,org.apache.camel.model,
  org.apache.camel.component.kura
```

Camel コンポーネントはランタイムレベルでサービスとして解決されるため、ルートで使用する予定のすべての Camel コンポーネントバンドルをインポートする必要がないことに注意してください。

ルーターバンドルをデプロイする前に、以下の Camel コアバンドルをデプロイ(Kura GoGo シェルを使用)していることを確認してください。

```
install file:///home/user/.m2/repository/org/apache/camel/camel-core/2.15.0/camel-core-
2.15.0.jar
start <camel-core-bundle-id>
install file:///home/user/.m2/repository/org/apache/camel/camel-core-osgi/2.15.0/camel-core-
osgi-2.15.0.jar
start <camel-core-osgi-bundle-id>
install file:///home/user/.m2/repository/org/apache/camel/camel-kura/2.15.0/camel-kura-
2.15.0.jar
start <camel-kura-bundle-id>
```

さらに、ルートで使用する予定のすべてのコンポーネント：

```
install file:///home/user/.m2/repository/org/apache/camel/camel-stream/2.15.0/camel-stream-
2.15.0.jar
start <camel-stream-bundle-id>
```

最後に、ルーターバンドルをデプロイします。

```
install file:///home/user/.m2/repository/com/example/myrouter/1.0/myrouter-1.0.jar
start <your-bundle-id>
```

KURAROUTER ユーティリティー

Kura ルーターベースクラスは多くの便利なユーティリティーを提供します。本セクションでは、各項目を取り上げます。

SLF4J ロガー

Kura はロギング目的で SLF4J ファサードを使用します。protected member log は、指定の Kura ルーターに関連付けられた SLF4J ロガーインスタンスを返します。

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
  log.info("Configuring Camel routes!");
        ...
    }

}
```

BUNDLECONTEXT

protected member bundleContext は、指定の Kura ルーターに関連付けられたバンドルコンテキストを返します。

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
  ServiceReference<MyService> serviceRef =
bundleContext.getServiceReference(LogService.class.getName());
  MyService myService = bundleContext.getService(serviceRef);
        ...
    }

}
```

CAMELCONTEXT

protected member camelContext は、指定の Kura ルーターに関連付けられた CamelContext です。

```
public class MyKuraRouter extends KuraRouter {
```

```
    @Override
    public void configure() throws Exception {
  camelContext.getStatus();
        ...
    }

}
```

PRODUCERTEMPLATE

protected member producerTemplate は、指定の Camel コンテキストに関連付けられた ProducerTemplate インスタンスです。

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        producerTemplate.sendBody("jms:temperature", 22.0);
        ...
    }

}
```

CONSUMERTEMPLATE

protected member consumerTemplate は、指定の Camel コンテキストに関連付けられた ConsumerTemplate インスタンスです。

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        producerTemplate.sendBody("jms:temperature", 22.0);
        ...
    }

}
```

OSGI サービスリゾルバー

OSGi サービスリゾルバー(service(Class<T> serviceType))を使用して、OSGi バンドルコンテキストからタイプでサービスを簡単に取得できます。

```
public class MyKuraRouter extends KuraRouter {

    @Override
```

```
    public void configure() throws Exception {
  MyService myService = service(MyService.class);
      ...
    }

}
```

service が見つからない場合は、null 値が返されます。サービスが利用できない場合にアプリケーションが失敗する場合は、代わりに requiredService(Class) メソッドを使用します。サービスが見つからない場合、requiredService は IllegalStateException を出力します。

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        MyService myService = requiredService(MyService.class);
        ...
    }

}
```

KURAROUTER アクティベーターコールバック

Kura ルーターには、Camel ルーターの動作方法をカスタマイズするために使用できるライフサイクルコールバックが含まれています。たとえば、以前の起動直前にルーターに関連付けられた CamelContext インスタンスを設定するには、KuraRouter クラスの beforeStart メソッドを上書きします。

```
public class MyKuraRouter extends KuraRouter {

  ...

  protected void beforeStart(CamelContext camelContext) {
    OsgiDefaultCamelContext osgiContext = (OsgiCamelContext) camelContext;
    osgiContext.setName("NameOfTheRouter");
  }

}
```

CONFIGURATIONADMIN からの XML ルートの読み込み

サーバー設定からルートの XML 定義を読み取る必要がある場合があります。この IoT ゲートウェイでは、無線の再デプロイメントコストが大きくなる可能性があるという一般的なシナリオになります。この要件に対応するには、KuraRouter が OSGi ConfigurationAdmin を使用して、kura.camel PID から kura.camel.BUNDLE-SYMBOLIC-NAME.route プロパティーを検索します。この方法では、デプロイされた KuraRouter ごとに Camel XML ルートファイルを定義できます。ルートを更新するには、適切な設定プロパティーを編集し、関連するバンドルを再起動するだけです。kura.camel.BUNDLE-

***SYMBOLIC-NAME.route* プロパティーの内容は、Camel XML ルートファイルになります。以下に例を示します。**

```
<routes xmlns="http://camel.apache.org/schema/spring">
    <route id="loaded">
        <from uri="direct:bar"/>
        <to uri="mock:bar"/>
    </route>
</routes>
```

宣言型 OSGI サービスとして KURA ルーターをデプロイする

Kura ルーターを宣言型 OSGi サービスとしてデプロイする場合は、KuraRouter で提供される activate および deactivate メソッドを使用できます。

```
<scr:component name="org.eclipse.kura.example.camel.MyKuraRouter" activate="activate"
deactivate="deactivate" enabled="true" immediate="true">
  <implementation class="org.eclipse.kura.example.camel.MyKuraRouter"/>
</scr:component>
```

第95章 言語

言語

Camel 2.5 で利用可能

言語コンポーネントを使用すると、Camel でサポートされる言語によってスクリプトを実行するエンドポイントに エクスチェンジ を送信できます。http://camel.apache.org/language.html言語スクリプトを実行するコンポーネントを使用すると、より動的なルーティング機能が可能になります。たとえば、Routing SlipRouting Slip または Dynamic RouterDynamic Router EIP を使用すると、スクリプトが動的に定義される 言語 エンドポイントにメッセージを送信することができます。

このコンポーネントは camel-core の追加設定なしで提供されるため、追加の JAR は必要ありません。Groovy や JavaScript 言語の使用など、選択の言語が義務付けられている場合にのみ、追加の Camel コンポーネントを含める必要があります。

Camel 2.11 以降では、Camel の他の 言語 でサポートされるのと同じ表記を使用するスクリプトの外部リソースを参照できます。

```
language://languageName:resource:scheme:location][?options]
```

URI 形式

```
language://languageName[:script][?options]
```

URI オプション

コンポーネントは以下のオプションをサポートします。

名前	デフォルト値	型	説明
languageName	null	文字列	単純な、groovy、javascript など、使用する 言語 の名前。このオプションは必須です。
script	null	文字列	実行するスクリプト。

変換	**true**	**boolean**	スクリプトの結果を新しいメッセージボディーとして使用するかどうか。**false** に設定するとスクリプトが実行されますが、スクリプトの結果は破棄されます。
contentCache	**true**	**boolean**	**Camel 2.9:** リソースからロードされた場合にスクリプトをキャッシュするかどうか。注記：**Camel 2.10.3** から、キャッシュされたスクリプトは clearContentCache 操作を使用して JMX 経由でランタイム時にリロードを強制できます。
cacheScript	**false**	**boolean**	**Camel 2.13/2.12.2/2.11.3:** コンパイルされたスクリプトをキャッシュするかどうか。このオプションを有効にすると、スクリプトはコンパイル/作成が 1 回のみ行われ、Camel メッセージの処理時に再利用されるため、パフォーマンスが向上します。ただし、これにより、以前の評価で残ったデータに対する副次的な影響が引き起こされ、同時実行の問題にも影響が及ぶ可能性があります。評価中のスクリプトがべき等である場合、このオプションはオンにすることができます。
binary	**false**	**boolean**	**Camel 2.14.1:** スクリプトがバイナリーコンテンツであるかどうか。これは、バイナリーファイルの読み込みなど、Constant 言語を使用してリソースを読み込むために使用されます。

メッセージヘッダー

以下のメッセージヘッダーを使用して、コンポーネントの動作に影響を与えることができます。

ヘッダー	説明
CamelLanguageScript	ヘッダーで提供される実行するスクリプト。エンドポイントに設定されたスクリプトよりも優先されます。

例

たとえば、Simple 言語を Message TranslatorMessage Translator a メッセージに使用できます。

```
String script = URLEncoder.encode("Hello ${body}", "UTF-8");
from("direct:start").to("language:simple:" + script).to("mock:result");
```

メッセージボディーの型を変換する場合は、これも実行できます。

```
String script = URLEncoder.encode("${mandatoryBodyAs(String)}", "UTF-8");
from("direct:start").to("language:simple:" + script).to("mock:result");
```

また、この例では、入力メッセージを 2 を乗算して、Groovy 言語を使用することもできます。

```
from("direct:start").to("language:groovy:request.body * 2").to("mock:result");
```

以下に示すように、スクリプトをヘッダーとして提供することもできます。ここでは、XPath 言語を使用して < foo > タグからテキストを抽出します。

```
Object out = producer.requestBodyAndHeader("language:xpath", "<foo>Hello World</foo>",
Exchange.LANGUAGE_SCRIPT, "/foo/text()");
assertEquals("Hello World", out);
```

リソースからのスクリプトの読み込み

Camel 2.9 以降で利用可能

エンドポイント URI または Exchange.LANGUAGE_SCRIPT ヘッダーのいずれかでロードするスクリプトのリソース URI を指定できます。uri は、file:、classpath:、または http のいずれかのスキームで開始する必要があります。

たとえば、クラスパスからスクリプトを読み込むには、次のコマンドを実行します。

```
from("direct:start")
    // load the script from the classpath

.to("language:simple:classpath:org/apache/camel/component/language/mysimplescript.txt")
    .to("mock:result");
```

デフォルトでは、スクリプトは一度ロードされてキャッシュされます。ただし、contentCache オプションを無効にして、評価ごとにスクリプトを読み込むことができます。たとえば、ディスク上のファイル myscript.txt が変更された場合は、更新されたスクリプトが使用されます。

```
from("direct:start")
    // the script will be loaded on each message, as we disabled cache
    .to("language:simple:file:target/script/myscript.txt?contentCache=false")
    .to("mock:result");
```

Camel 2.11 以降では、以下のように resource：と接頭辞を付けると、Camel の他の 言語 と同様のリソース を参照できます。

```
from("direct:start")
    // load the script from the classpath

.to("language:simple:resource:classpath:org/apache/camel/component/language/mysimplesc
ript.txt")
    .to("mock:result");
```

- *言語*
- *Routing SlipRouting Slip*
- *Dynamic RouterDynamic Router*

第96章 LDAP

LDAP コンポーネント

ldap コンポーネントを使用すると、フィルターを使用してメッセージペイロードとして LDAP サーバーで検索を実行できます。このコンポーネントは、標準の JNDI (javax.naming パッケージ)を使用してサーバーにアクセスします。

URI 形式

```
ldap:ldapServerBean[?options]
```

*URI の **ldapServerBean** 部分は、レジストリーの DirContext Bean を参照します。LDAP コンポーネントはプロデューサーエンドポイントのみをサポートします。つまり、ldap URI はルートの開始時に from に表示されることができません。*

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
base	**ou=system**	検索用のベース DN。
scope	**subtree**	ベース DN から始まるエントリーのツリーを詳しく検索する方法を指定します。値は、**オブジェクト**、**1** 次、または **サブツリー** にすることができます。
pageSize	ページングは使用されません。	LDAP モジュールはページングを使用してすべての結果を取得します（ほとんどの LDAP サーバーは 1 つのクエリーで複数のエントリーを取得しようとすると例外を出力します）。これを使用できるようにするには、**LdapContext** (**DirContext**のサブクラス)を **ldapServerBean** として渡す必要があります（そうでない場合は例外が発生します）。

returnedAttributes	LDAP サーバー（すべてまたはなし）に依存します。	結果の各エントリーに設定する必要がある属性のコンマ区切りリスト

結果

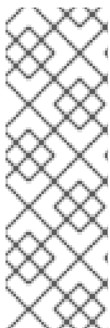
結果は、ArrayList<javax.naming.directory.SearchResult > オブジェクトとして Out ボディーで返されます。

DIRCONTEXT

URI ldap:ldapserver は、ldapserver という ID を持つ Spring Bean を参照します。ldapserver Bean は以下のように定義できます。

```
<bean id="ldapserver" class="javax.naming.directory.InitialDirContext" scope="prototype">
  <constructor-arg>
    <props>
      <prop key="java.naming.factory.initial">com.sun.jndi.ldap.LdapCtxFactory</prop>
      <prop key="java.naming.provider.url">ldap://localhost:10389</prop>
      <prop key="java.naming.security.authentication">none</prop>
    </props>
  </constructor-arg>
</bean>
```

上記の例では、匿名でローカルでホストされる LDAP サーバーに接続する通常の Sun ベースの LDAP DirContext を宣言します。



注記

DirContext オブジェクトは、コントラクトによる同時実行をサポートする必要は ありません。そのため、ディレクトリーコンテキストは Bean 定義で設定 scope="prototype" で宣言されるか、コンテキストが同時実行をサポートすることが重要です。Spring フレームワークでは、プロトタイプ スコープオブジェクトは検索時に毎回インスタンス化されます。

サンプル

上記の Spring 設定から、以下のコードサンプルは LDAP 要求を送信し、メンバーのグループを検索します。次に、Common Name が応答から抽出されます。

```
ProducerTemplate<Exchange> template = exchange
```

```
 .getContext().createProducerTemplate();

Collection<?> results = (Collection<?>) (template
  .sendBody(
    "ldap:ldapserver?base=ou=mygroup,ou=groups,ou=system",
    "(member=uid=huntc,ou=users,ou=system)"));

if (results.size() > 0) {
  // Extract what we need from the device's profile

  Iterator<?> resultIter = results.iterator();
  SearchResult searchResult = (SearchResult) resultIter
      .next();
  Attributes attributes = searchResult
      .getAttributes();
  Attribute deviceCNAttr = attributes.get("cn");
  String deviceCN = (String) deviceCNAttr.get();

  ...
```

特定のフィルターが必要ない場合（たとえば、単一のエントリーのみ）。ワイルドカードフィルター式を指定します。たとえば、LDAP エントリーに Common Name がある場合は、以下のようなフィルター式を使用します。

```
(cn=*)
```

認証情報を使用したバインディング

Camel エンドユーザーは、クレデンシャルを使用して ldap サーバーにバインドするのに使用したこのサンプルコードをしました。

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.ldap.LdapCtxFactory");
props.setProperty(Context.PROVIDER_URL, "ldap://localhost:389");
props.setProperty(Context.URL_PKG_PREFIXES, "com.sun.jndi.url");
props.setProperty(Context.REFERRAL, "ignore");
props.setProperty(Context.SECURITY_AUTHENTICATION, "simple");
props.setProperty(Context.SECURITY_PRINCIPAL, "cn=Manager");
props.setProperty(Context.SECURITY_CREDENTIALS, "secret");

SimpleRegistry reg = new SimpleRegistry();
reg.put("myldap", new InitialLdapContext(props, null));

CamelContext context = new DefaultCamelContext(reg);
context.addRoutes(
    new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:start").to("ldap:myldap?base=ou=test");
        }
```

```
    }
);
context.start();

ProducerTemplate template = context.createProducerTemplate();

Endpoint endpoint = context.getEndpoint("direct:start");
Exchange exchange = endpoint.createExchange();
exchange.getIn().setBody("(uid=test)");
Exchange out = template.send(endpoint, exchange);

Collection<SearchResult> data = out.getOut().getBody(Collection.class);
assert data != null;
assert !data.isEmpty();

System.out.println(out.getOut().getBody());

context.stop();
```

SSL の設定

以下の例のように、InitialDirContext Bean でカスタムソケットファクトリーを作成し、参照することのみが必要になります。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
                 http://camel.apache.org/schema/blueprint http://camel.apache.org/schema/blueprint/camel-
blueprint.xsd">


    <sslContextParameters xmlns="http://camel.apache.org/schema/blueprint"
                          id="sslContextParameters">
        <keyManagers
                keyPassword="{{keystore.pwd}}">
            <keyStore
                    resource="{{keystore.url}}"
                    password="{{keystore.pwd}}"/>
        </keyManagers>
    </sslContextParameters>

    <bean id="customSocketFactory" class="zotix.co.util.CustomSocketFactory">
        <argument ref="sslContextParameters" />
    </bean>
    <bean id="ldapserver" class="javax.naming.directory.InitialDirContext" scope="prototype">
        <argument>
            <props>
                <prop key="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory"/>
                <prop key="java.naming.provider.url" value="ldaps://lab.zotix.co:636"/>
                <prop key="java.naming.security.protocol" value="ssl"/>
                <prop key="java.naming.security.authentication" value="simple" />
```

```
            <prop key="java.naming.security.principal" value="cn=Manager,dc=example,dc=com"/>
            <prop key="java.naming.security.credentials" value="passw0rd"/>
            <prop key="java.naming.ldap.factory.socket"
                value="zotix.co.util.CustomSocketFactory"/>
        </props>
    </argument>
  </bean>
</blueprint>
```

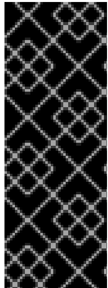
CustomSocketFactory クラスは以下のように実装されます。

```
import org.apache.camel.util.jsse.SSLContextParameters;

import javax.net.SocketFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;
import java.io.IOException;
import java.net.InetAddress;
import java.net.Socket;
import java.security.KeyStore;

/**
 * The CustomSocketFactory. Loads the KeyStore and creates an instance of
SSLSocketFactory
 */
public class CustomSocketFactory extends SSLSocketFactory {

  private static SSLSocketFactory socketFactory;

  /**
   * Called by the getDefault() method.
   */
  public CustomSocketFactory() {

  }

  /**
   * Called by Blueprint DI to initialise an instance of SocketFactory
   *
   * @param sslContextParameters
   */
  public CustomSocketFactory(SSLContextParameters sslContextParameters) {
    try {
      KeyStore keyStore =
sslContextParameters.getKeyManagers().getKeyStore().createKeyStore();
      TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
      tmf.init(keyStore);
      SSLContext ctx = SSLContext.getInstance("TLS");
      ctx.init(null, tmf.getTrustManagers(), null);
      socketFactory = ctx.getSocketFactory();
    } catch (Exception ex) {
      ex.printStackTrace(System.err);  /* handle exception */
    }
```

```
    }

    /**
     * Getter for the SocketFactory
     *
     * @return
     */
    public static SocketFactory getDefault() {
        return new CustomSocketFactory();
    }

    @Override
    public String[] getDefaultCipherSuites() {
        return socketFactory.getDefaultCipherSuites();
    }

    @Override
    public String[] getSupportedCipherSuites() {
        return socketFactory.getSupportedCipherSuites();
    }

    @Override
    public Socket createSocket(Socket socket, String string, int i, boolean bln) throws
IOException {
        return socketFactory.createSocket(socket, string, i, bln);
    }

    @Override
    public Socket createSocket(String string, int i) throws IOException {
        return socketFactory.createSocket(string, i);
    }

    @Override
    public Socket createSocket(String string, int i, InetAddress ia, int i1) throws IOException {
        return socketFactory.createSocket(string, i, ia, i1);
    }

    @Override
    public Socket createSocket(InetAddress ia, int i) throws IOException {
        return socketFactory.createSocket(ia, i);
    }

    @Override
    public Socket createSocket(InetAddress ia, int i, InetAddress ia1, int i1) throws IOException
{
        return socketFactory.createSocket(ia, i, ia1, i1);
    }
}
```

第97章 LEVELDB

LEVELDB



重要

JBoss Fuse 6.3 以降、Camel LevelDB (camel-leveldb)コンポーネントは、Red Hat Enterprise Linux 以外のすべてのオペレーティングシステムで 非推奨 になりました。今後、Camel LevelDB コンポーネントは Red Hat Enterprise Linux でのみサポートされます。

Camel 2.10 以降で利用可能

Leveldb は、非常に軽量で組み込み可能なキー値のデータベースです。Camel では、Aggregator などのさまざまな Camel 機能の永続的なサポートを提供します。

現在の機能では、以下が提供されます。

- LevelDBAggregationRepository

LEVELDBAGGREGATIONREPOSITORY の使用

LevelDBAggregationRepository は AggregationRepository で、オンザフライで集約されたメッセージを永続化します。これにより、デフォルトのアグリゲーターはメモリーの AggregationRepository のみを使用するので、メッセージを失わないようにします。

これには、以下のオプションがあります。

オプション	タイプ	説明

repositoryName	文字列	必須リポジトリー名。複数のリポジトリーに共有 **LevelDBFile** を使用できます。
persistentFileName	文字列	永続ストレージのファイル名。起動時にファイルが存在しない場合は、新しいファイルが作成されます。
levelDBFile	LevelDBFile	既存の設定された **org.apache.camel.component.leveldb.LevelDBFile** インスタンスを使用します。
sync	boolean	**Camel 2.12:** LevelDBFile が書き込み時に同期されるかどうか。デフォルトは false です。書き込み時に同期することで、すべての書き込みがディスクにスプールされるのを待つため、更新は失われません。非同期書き込みと同期 書き込みの詳細は、LevelDB ドキュメント を参照してください。
returnOldExchange	boolean	get 操作が存在する場合は、get 操作によって古い既存のエクスチェンジが返されるかどうか。デフォルトでは、集計時に古いエクスチェンジを必要としないため、このオプションは **false** で最適化されます。
useRecovery	boolean	リカバリーが有効になっているかどうか。このオプションは、デフォルトで **true** です。有効にすると、Camel Aggregator は失敗した集約されたエクスチェンジを自動的にリカバリーし、再送信を行います。
recoveryInterval	long	recovery が有効になっている場合、バックグラウンドタスクは x 度ごとに実行され、失敗したエクスチェンジをスキャンしてリカバリーし、再送信します。デフォルトでは、この間隔は 5000 ミリ秒です。

maximumRedeliveries	int	リカバリーされたエクスチェンジの再配信試行の最大数を制限できます。有効にすると、すべての再配信試行に失敗すると、エクスチェンジはデッドレターチャネルに移動します。デフォルトでは、このオプションは無効です。このオプションを使用する場合は、**deadLetterUri** オプションも指定する必要があります。
deadLetterUri	文字列	Dead Letter Channel のエンドポイント URI。使い切られるリカバリーされたエクスチェンジが移動されます。このオプションを使用する場合は、**maximumRedeliveries** オプションも指定する必要があります。

repositoryName *オプションを指定する必要があります。次に、**persistentFileName** または **levelDBFile** のいずれかを指定する必要があります。*

永続化時に保持される内容

LevelDBAggregationRepository *は、**Serializable** と互換性のあるメッセージボディーデータタイプのみを保持します。メッセージヘッダーはプリミティブ、文字列、数字または同様のものである必要があります。データ型がそのようなタイプの場合はドロップされ、**WARN** がログに記録されます。また、メッセージ本文とメッセージ ヘッダーのみを保持 し ます。**Exchange** プロパティーは永続化されません。*

復元

LevelDBAggregationRepository *は、デフォルトで失敗したエクスチェンジ を復元し ます。これは、永続ストアで失敗した エクスチェンジをスキャンするバックグラウンドタスクを持つことで行われます。**checkInterval** オプションを使用して、このタスクの実行頻度を設定できます。リカバリーはトランザクションとして機能し、**Camel** が失敗したエクスチェンジのリカバリーおよび再配信を試み ます。リカバリー さ れたエクスチェンジは永続ストアから復元され、再送信されて再度送信されます。*

エクスチェンジ のリカバリー/再配信時に、以下のヘッダーが設定されます。

ヘッダー	タイプ	説明

Exchange.REDELIVERED	ブール値	エクスチェンジ が再配信されていることを示すために が true に設定されます。
Exchange.REDELIVERY_COUNTER	整数	1 から始まる再配信の試行。

エクスチェンジ が正常に処理された場合のみ、完了とマークされます。これは、AggregationRepository で confirm メソッドが呼び出されたときに発生します。つまり、同じ エクスチェンジ が再び失敗すると、成功するまで再試行されます。

maximumRedeliveries オプションを使用して、特定のリカバリーエクスチェンジの再配信試行の最大数を制限でき ます。maximumRedeliveries に達したときに エクスチェンジ を送信する場所を Camel が認識できるように deadLetterUri オプションも設定する必要があります。

camel-leveldb のユニットテスト（例： このテスト ）にいくつかの例を確認できます。

JAVA DSL での LEVELDBAGGREGATIONREPOSITORY の使用

この例では、target/data/leveldb.dat ファイルで集約されたメッセージを永続化します。

```
public void configure() throws Exception {
    // create the leveldb repo
    LevelDBAggregationRepository repo = new LevelDBAggregationRepository("repo1",
"target/data/leveldb.dat");

    // here is the Camel route where we aggregate
    from("direct:start")
        .aggregate(header("id"), new MyAggregationStrategy())
            // use our created leveldb repo as aggregation repository
            .completionSize(5).aggregationRepository(repo)
            .to("mock:aggregated");
}
```

SPRING XML での LEVELDBAGGREGATIONREPOSITORY の使用

同じ例になりますが、代わりに Spring XML を使用します。

```
<!-- a persistent aggregation repository using camel-leveldb -->
<bean id="repo"
class="org.apache.camel.component.leveldb.LevelDBAggregationRepository">
    <!-- store the repo in the leveldb.dat file -->
    <property name="persistentFileName" value="target/data/leveldb.dat"/>
    <!-- and use repo2 as the repository name -->
    <property name="repositoryName" value="repo2"/>
</bean>

<!-- aggregate the messages using this strategy -->
<bean id="myAggregatorStrategy"
class="org.apache.camel.component.leveldb.LevelDBSpringAggregateTest$MyAggregationSt
rategy"/>

<!-- this is the camel routes -->
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">

    <route>
        <from uri="direct:start"/>
        <!-- aggregate using our strategy and leveldb repo, and complete when we have 5
messages aggregated -->
        <aggregate strategyRef="myAggregatorStrategy" aggregationRepositoryRef="repo"
completionSize="5">
            <!-- correlate by header with the key id -->
            <correlationExpression><header>id</header></correlationExpression>
            <!-- send aggregated messages to the mock endpoint -->
            <to uri="mock:aggregated"/>
        </aggregate>
    </route>

</camelContext>
```

DEPENDENCIES

camel ルートで LevelDB を使用するには、camel-leveldb の依存関係を追加する必要があります。

maven を使用する場合は、以下を pom.xml に追加し、バージョン番号を最新および最大のリリースに置き換えます(最新バージョンのダウンロードページを参照してください)。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-leveldb</artifactId>
  <version>2.17.0.redhat-630xxx</version>
</dependency>
```

- *アグリゲーター*
- *HawtDB*
- *コンポーネント*

第98章 LINKEDIN

LINKEDIN コンポーネント

Camel 2.14 から利用可能

LinkedIn コンポーネントは、https://developer.linkedin.com/rest に記載されている LinkedIn REST API へのアクセスを提供します。

LinkedIn は、すべてのクライアントアプリケーション認証に OAuth2.0 を使用します。アカウントで camel-linkedin を使用するには、https://www.linkedin.com/secure/developer で LinkedIn に新しいアプリケーションを作成する必要があります。LinkedIn アプリケーションのクライアント ID およびシークレットにより、現在のユーザーを必要とする LinkedIn REST API へのアクセスが許可されます。ユーザーアクセストークンは、エンドユーザーのコンポーネントによって生成および管理されます。また、Camel アプリケーションは org.apache.camel.component.linkedin.api.OAuthSecureStorage の実装を登録し、org.apache.camel.component.linkedin.apiOAuthToken OAuth トークンを提供することもできます。

TLS (Transport Layer Security)を使用するように camel-linkedin コンポーネントを設定するには、Security Guide の Configuring Transport Security for Camel Components の章を参照してください。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-linkedin</artifactId>
   <version>${camel-version}</version>
</dependency>
```

URI 形式

```
linkedin://endpoint-prefix/endpoint?[options]
```

endpoint-prefix は以下のいずれかになります。

- comments

- *companies*
- *groups*
- *jobs*
- *people*
- *posts*
- *search*

LINKEDINCOMPONENT

LinkedIn コンポーネントは、以下のオプションで設定できます。これらのオプションは、org.apache.camel.component.linkedin.LinkedInConfiguration タイプのコンポーネントの Bean プロパティー 設定 を使用して指定できます。

オプション	タイプ	説明
clientId	**String**	LinkedIn アプリケーションクライアント ID
clientSecret	**String**	LinkedIn アプリケーションクライアントシークレット
httpParams	**java.util.Map**	プロキシーホストおよびポートなどのカスタム HTTP パラメーター。AllClientPNamesからの定数を使用します。
lazyAuth	**boolean**	Lazy OAuth を有効または無効にするフラグ。デフォルトは true です。有効にすると、OAuth トークンの取得または生成は最初の REST 呼び出しまで実行されません。

redirectUri	**String**	アプリケーションのリダイレクト URI は、機能するリダイレクトサーバーを用意しなくても、コンポーネントはこのページにリダイレクトされません。テストには https://localhost を使用できます。
scopes	**org.apache.camel.component.linkedin.api.OAuthScope[]**	https://developer.linkedin.com/documents/authentication#grantingで指定される LinkedIn スコープの一覧
secureStorage	**org.apache.camel.component.linkedin.api.OAuthSecureStorage**	OAuth トークンを提供するコールバックインターフェイス、またはコンポーネントによって生成されたトークンを保存するコールバックインターフェイス。コールバックは最初の呼び出しで **null** を返し、作成されたトークンを saveToken （）コールバックに保存します。コールバックが初めて null を返す場合は、**userPassword** を指定する必要があります。
userName	**String**	LinkedIn ユーザーアカウント名を指定する必要があります。提供する必要があります
userPassword	**String**	LinkedIn アカウントのパスワード

プロデューサーエンドポイント：

*プロデューサーエンドポイントはエンドポイント接頭辞を使用し、続いてエンドポイント名と以下で説明する関連オプションを使用できます。一部のエンドポイントには、短縮エイリアスを使用できます。エンドポイント **URI** には 接頭辞が含まれている必要があります。*

*必須ではないエンドポイントオプションは **[]** で示されます。エンドポイントに必須のオプションがない場合は、**[]** オプションのセットの 1 つを指定する必要があります。プロデューサーエンドポイントは、**Camel Exchange In** メッセージに含まれる値を持つ **endpoint** オプションの名前が含まれる必要がある特別なオプション **inBody** を使用することもできます。*

*エンドポイントオプションは、エンドポイント **URI** またはメッセージヘッダーで動的に指定できます。メッセージヘッダー名は **CamelLinkedIn.<option>** の形式である必要があります。**inBody** オプションはメッセージヘッダーを上書きすることに注意してください。つまり、エンドポイントオプションの **inBody=option** は **CamelLinkedIn.option** ヘッダーを上書きすることに注意してください。*

エンドポイントおよびオプションの詳細は、https://developer.linkedin.com/rest の LinkedIn REST API ドキュメントを参照してください。

エンドポイント接頭辞のコメント

以下のエンドポイントは、以下のように接頭辞 コメント で呼び出すことができます。

> *linkedin://comments/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
getComment	comment	comment_id, fields	org.apache.camel.component.linkedin.api.model.Comment
removeComment	comment	comment_id	

コメントの URI オプション

名前	タイプ
comment_id	String
fields	String

エンドポイント接頭辞企業

以下のエンドポイントは、以下のように接頭辞 企業 で呼び出すことができます。

> *linkedin://companies/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
addCompanyUpdateComment	companyUpdateComment	company_id, update_key, updatecomment	
addCompanyUpdateCommentAsCompany	companyUpdateCommentAsCompany	company_id, update_key, updatecomment	

addShare	share	company_id, share	
getCompanies	companies	email_domain, fields, is_company_admin	org.apache.camel.component.linkedin.api.model.Companies
getCompanyById	companyById	company_id, fields	org.apache.camel.component.linkedin.api.model.Company
getCompanyByName	companyByName	fields, universal_name	org.apache.camel.component.linkedin.api.model.Company
getCompanyUpdateComments	companyUpdateComments	company_id, fields, secure_urls, update_key	org.apache.camel.component.linkedin.api.model.Comments
getCompanyUpdateLikes	companyUpdateLikes	company_id, fields, secure_urls, update_key	org.apache.camel.component.linkedin.api.model.Likes
getCompanyUpdates	companyUpdates	company_id, count, event_type, fields, start	org.apache.camel.component.linkedin.api.model.Updates
getHistoricalFollowStatistics	historicalFollowStatistics	company_id, end_timestamp, start_timestamp, time_granularity	org.apache.camel.component.linkedin.api.model.HistoricalFollowStatistics
getHistoricalStatusUpdateStatistics	historicalStatusUpdateStatistics	company_id, end_timestamp, start_timestamp, time_granularity, update_key	org.apache.camel.component.linkedin.api.model.HistoricalStatusUpdateStatistics
getNumberOfFollowers	numberOfFollowers	companySizes, company_id, geos, industries, jobFunc, seniorities	org.apache.camel.component.linkedin.api.model.NumFollowers
getStatistics	statistics	company_id	org.apache.camel.component.linkedin.api.model.CompanyStatistics
isShareEnabled		company_id	org.apache.camel.component.linkedin.api.model.IsCompanyShareEnabled

isViewerShareEnabled		company_id	org.apache.camel.component.linkedin.api.model.IsCompanyShareEnabled
likeCompanyUpdate		company_id, isliked, update_key	

企業の URI オプション

[s） [companySizes, count, email_domain, end_timestamp, event_type, geos, industry, is_company_admin, jobFunc, secure_urls, depthities, start, start_timestamp, time_granularity] のいずれかのオプションに値が指定されていない場合、null と見なされます。null 値は、他のオプションが一致するエンドポイントを満たさない場合にのみ使用されることに注意してください。

名前	タイプ
companySizes	java.util.List
company_id	Long
count	Long
email_domain	String
end_timestamp	Long
event_type	org.apache.camel.component.linkedin.api.Eventtype
fields	String
geos	java.util.List
industries	java.util.List
is_company_admin	Boolean
isliked	org.apache.camel.component.linkedin.api.model.IsLiked
jobFunc	java.util.List
secure_urls	Boolean

seniorities	java.util.List
share	org.apache.camel.component.linkedin.api.model.Share
start	Long
start_timestamp	Long
time_granularity	org.apache.camel.component.linkedin.api.Timegranularity
universal_name	String
update_key	String
updatecomment	org.apache.camel.component.linkedin.api.model.UpdateComment

エンドポイント接頭辞グループ

以下のエンドポイントは、以下のように接頭辞 グループ で呼び出すことができます。

> *linkedin://groups/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
addPost	post	group_id, post	
getGroup	group	group_id	org.apache.camel.component.linkedin.api.model.Group

グループの URI オプション

名前	タイプ
group_id	Long
post	org.apache.camel.component.linkedin.api.model.Post

エンドポイント接頭辞ジョブ

以下のエンドポイントは、以下のように接頭辞 ジョブ で呼び出すことができます。

> *linkedin://jobs/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
addJob	job	job	
editJob		job, partner_job_id	
getJob	job	fields, job_id	org.apache.camel.component.linkedin.api.model.Job
removeJob	job	partner_job_id	

ジョブの URI オプション

名前	タイプ
fields	String
job	org.apache.camel.component.linkedin.api.model.Job
job_id	Long
partner_job_id	Long

エンドポイント接頭辞 PEOPLE

以下のエンドポイントは、以下のように接頭辞 people で呼び出すことができます。

> *linkedin://people/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
addActivity	activity	activity	
addGroupMembership	groupMembership	groupmembership	

addInvite	invite	mailboxitem	
addJobBookmark	jobBookmark	jobbookmark	
addUpdateComment	updateComment	update_key, updatecomment	
followCompany		company	
getConnections	connections	fields, secure_urls	org.apache.camel.component.linkedin.api.model.Connections
getConnectionsById	connectionsById	fields, person_id, secure_urls	org.apache.camel.component.linkedin.api.model.Connections
getConnectionsByUrl	connectionsByUrl	fields, public_profile_url, secure_urls	org.apache.camel.component.linkedin.api.model.Connections
getFollowedCompanies	followedCompanies	fields	org.apache.camel.component.linkedin.api.model.Companies
getGroupMembershipSettings	groupMembershipSettings	count, fields, group_id, start	org.apache.camel.component.linkedin.api.model.GroupMemberships
getGroupMemberships	groupMemberships	count, fields, membership_state, start	org.apache.camel.component.linkedin.api.model.GroupMemberships
getJobBookmarks	jobBookmarks		org.apache.camel.component.linkedin.api.model.JobBookmarks
getNetworkStats	networkStats		org.apache.camel.component.linkedin.api.model.NetworkStats
getNetworkUpdates	networkUpdates	after, before, count, fields, scope, secure_urls, show_hidden_members, start, type	org.apache.camel.component.linkedin.api.model.Updates

getNetworkUpdatesById	networkUpdatesById	after, before, count, fields, person_id, scope, secure_urls, show_hidden_members, start, type	org.apache.camel.component.linkedin.api.model.Updates
getPerson	person	fields, secure_urls	org.apache.camel.component.linkedin.api.model.Person
getPersonById	personById	fields, person_id, secure_urls	org.apache.camel.component.linkedin.api.model.Person
getPersonByUrl	personByUrl	fields, public_profile_url, secure_urls	org.apache.camel.component.linkedin.api.model.Person
getPosts	posts	category, count, fields, group_id, modified_since, order, role, start	org.apache.camel.component.linkedin.api.model.Posts
getSuggestedCompanies	suggestedCompanies	fields	org.apache.camel.component.linkedin.api.model.Companies
getSuggestedGroupPosts	suggestedGroupPosts	category, count, fields, group_id, modified_since, order, role, start	org.apache.camel.component.linkedin.api.model.Posts
getSuggestedGroups	suggestedGroups	fields	org.apache.camel.component.linkedin.api.model.Groups
getSuggestedJobs	suggestedJobs	fields	org.apache.camel.component.linkedin.api.model.JobSuggestions
getUpdateComments	updateComments	fields, secure_urls, update_key	org.apache.camel.component.linkedin.api.model.Comments
getUpdateLikes	updateLikes	fields, secure_urls, update_key	org.apache.camel.component.linkedin.api.model.Likes
likeUpdate		isliked, update_key	

removeGroupMembership	groupMembership	group_id	
removeGroupSuggestion	groupSuggestion	group_id	
removeJobBookmark	jobBookmark	job_id	
share		share	org.apache.camel.component.linkedin.api.model.Update
stopFollowingCompany		company_id	
updateGroupMembership		group_id, groupmembership	

ユーザーの URI オプション

[after, before, category, count, membership_state, modified_since, order, public_profile_url, role, scope, secure_urls, show_hidden_members, start, type] のいずれかのオプションに値が指定されていない場合は、null と見なされます。null 値は、他のオプションが一致するエンドポイントを満たさない場合にのみ使用されることに注意してください。

名前	タイプ
activity	org.apache.camel.component.linkedin.api.model.Activity
after	Long
before	Long
category	org.apache.camel.component.linkedin.api.Category
company	org.apache.camel.component.linkedin.api.model.Company
company_id	Long
count	Long
fields	String

group_id	Long
groupmembership	org.apache.camel.component.linkedin.api.model.GroupMembership
isliked	org.apache.camel.component.linkedin.api.model.IsLiked
job_id	Long
jobbookmark	org.apache.camel.component.linkedin.api.model.JobBookmark
mailboxitem	org.apache.camel.component.linkedin.api.model.MailboxItem
membership_state	org.apache.camel.component.linkedin.api.model.MembershipState
modified_since	Long
order	org.apache.camel.component.linkedin.api.Order
person_id	String
public_profile_url	String
role	org.apache.camel.component.linkedin.api.Role
scope	String
secure_urls	Boolean
share	org.apache.camel.component.linkedin.api.model.Share
show_hidden_members	Boolean
start	Long
type	org.apache.camel.component.linkedin.api.Type
update_key	String

updatecomment	org.apache.camel.component.linkedin.api.model.UpdateComment

エンドポイント接頭辞 POST

以下のエンドポイントは、以下のように posts 接頭辞で呼び出すことができます。

linkedin://posts/endpoint?[options]

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
addComment	comment	comment, post_id	
flagCategory		post_id, postcategorycode	
followPost		isfollowing, post_id	
getPost	post	count, fields, post_id, start	org.apache.camel.component.linkedin.api.model.Post
getPostComments	postComments	count, fields, post_id, start	org.apache.camel.component.linkedin.api.model.Comments
likePost		isliked, post_id	
removePost	post	post_id	

投稿の URI オプション

エンドポイント URI またはメッセージヘッダーのいずれかで値が提供されない [count, start] オプションの場合は、null と見なされます。null 値は、他のオプションが一致するエンドポイントを満たさない場合にのみ使用されることに注意してください。

名前	タイプ
comment	org.apache.camel.component.linkedin.api.model.Comment
count	Long

fields	String
isfollowing	org.apache.camel.component.linkedin.api.model.IsFollowing
isliked	org.apache.camel.component.linkedin.api.model.IsLiked
post_id	String
postcategorycode	org.apache.camel.component.linkedin.api.model.PostCategoryCode
start	Long

エンドポイント接頭辞検索

以下のエンドポイントは、以下のように接頭辞 search で呼び出すことができます。

> *linkedin://search/endpoint?[options]*

エンドポイント	短縮形エイリアス	オプション	結果ボディーのタイプ
searchCompanies	companies	count, facet, facets, fields, hq_only, keywords, sort, start	org.apache.camel.component.linkedin.api.model.CompanySearch
searchJobs	jobs	company_name, count, country_code, distance, facet, facets, fields, job_title, keywords, postal_code, sort, start	org.apache.camel.component.linkedin.api.model.JobSearch

searchPeople	people	company_name, count, country_code, current_company, current_school, current_title, distance, facet, facets, fields, first_name, keywords, last_name, postal_code, school_name, sort, start, title	org.apache.camel.component.linkedin.api.model.PeopleSearch

検索の URI オプション

[s）[company_name, count, country_code, current_company, current_school, current_title, distance, facet, facets, first_name, hq_only, job_title, keywords, last_name, postal_code, school_name, sort, start, title] のいずれかのオプションに指定しないと、null と見なされます。null 値は、他のオプションが一致するエンドポイントを満たさない場合にのみ使用されることに注意してください。

名前	タイプ
company_name	String
count	Long
country_code	String
current_company	String
current_school	String
current_title	String
distance	org.apache.camel.component.linkedin.api.model.Distance
facet	String
facets	String
fields	String
first_name	String

hq_only	String
job_title	String
keywords	String
last_name	String
postal_code	String
school_name	String
sort	String
start	Long
title	String

コンシューマーエンドポイント

プロデューサーエンドポイントはいずれもコンシューマーエンドポイントとして使用できます。コンシューマーエンドポイントは、consumer. 接頭辞を持つ Scheduled Poll Consumer オプション を使用して、エンドポイント呼び出しをスケジュールできます。デフォルトでは、配列またはコレクションを返すコンシューマーエンドポイントは、要素ごとにエクスチェンジを 1 つ生成し、それらのルートはエクスチェンジごとに 1 回実行されます。この動作を変更するには、consumer.splitResults=true プロパティーを使用して、リストまたは配列全体の単一のエクスチェンジを返します。

メッセージヘッダー

すべての URI オプションは、CamelLinkedIn. 接頭辞を持つプロデューサーエンドポイントのメッセージヘッダーで指定できます。

メッセージボディー

すべての結果メッセージ本文は、Apache CXF JAX-RS を使用してビルドされる Camel LinkedIn API SDK によって提供されるオブジェクトを使用します。プロデューサーエンドポイントは、inBody エンドポイントパラメーターに受信メッセージボディーのオプション名を指定できます。

ユースケース

以下のルートはユーザーのプロファイルを取得します。

```
from("direct:foo")
 .to("linkedin://people/person");
```

以下のルートは、30 秒ごとにユーザーの接続をポーリングします。

```
from("linkedin://people/connections?consumer.timeUnit=SECONDS&consumer.delay=30")
 .to("bean:foo");
```

*以下のルートは、動的ヘッダーオプションを持つプロデューサーを使用します。***personId** *ヘッダーには LinkedIn person ID があるため、以下のように* **CamelLinkedIn.person_id** *ヘッダーに割り当てられます。*

```
from("direct:foo")
 .setHeader("CamelLinkedIn.person_id", header("personId"))
 .to("linkedin://people/connectionsById")
 .to("bean://bar");
```

第99章 リスト

コンポーネントの一覧表示

非推奨: は Apache Camel 2.0 の Browse コンポーネントに名前が変更されました。

List コンポーネントは、テスト、視覚化ツール、またはデバッグに役立つシンプルな BrowsableEndpoint を提供します。エンドポイントに送信されたエクスチェンジはすべて参照できます。

URI 形式

```
list:someName
```

someName には、エンドポイントを一意に識別する任意の文字列を指定できます。

例

以下のルートには、渡されるエクスチェンジを閲覧できる list コンポーネントがあります。

```
from("activemq:order.in").to("list:orderReceived").to("bean:processOrder");
```

その後、java コードから受信したエクスチェンジを検査できます。

```
private CamelContext context;

public void inspectRecievedOrders() {
    BrowsableEndpoint browse = context.getEndpoint("list:orderReceived",
BrowsableEndpoint.class);
    List<Exchange> exchanges = browse.getExchanges();
    ...
    // then we can inspect the list of received exchanges from Java
    for (Exchange exchange : exchanges) {
        String payload = exchange.getIn().getBody();
        ...
    }
}
```

以下も参照してください。

-
  参照

第100章 LOG

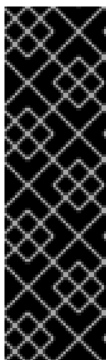
ログコンポーネント

log: コンポーネントは、メッセージエクスチェンジを基礎となるロギングメカニズムに記録します。

URI 形式

```
log:loggingCategory[?options]
```

loggingCategory は使用するロギングカテゴリーの名前です。URI にクエリーオプションは **?option=value&option=value&..** の形式で追加できます。



レジストリーからの LOGGER インスタンスの使用

Camel 2.12.4/2.13.1 以降、レジストリーに **org.slf4j.Logger** の単一インスタンスがある場合、**loggingCategory** はロガーインスタンスの作成に使用されなくなりました。代わりに、登録されたインスタンスが使用されます。また、**?logger=#myLogger** URI パラメーターを使用して特定の **Logger** インスタンスを参照することもできます。最終的には、登録済みおよび URI **logger** パラメーターがない場合、**loggingCategory** を使用してロガーインスタンスが作成されます。

たとえば、ログエンドポイントは通常、以下のように **level** オプションを使用してログ レベル を指定します。

```
log:org.apache.camel.example?level=DEBUG
```

デフォルトのロガーは、すべてのエクスチェンジ(通常のロギング)をログに記録します。しかし、Apache Camel には **Throughput** ロガーも含まれています。これは、**groupSize** オプションが指定されているたびに使用されます。

DSL へのログイン

DSL には直接 ログイン がありますが、目的が異なります。これは、軽量ログおよびヒューマンログを対象としています。詳細は、LogEIP を参照してください。

オプション

オプション	デフォルト	タイプ	Description
level	**INFO**	**文字列**	使用するロギングレベル。使用できる値は、**ERROR**、**WARN**、**INFO**、**DEBUG**、**TRACE**、**OFF**です。
marker	**null**	**文字列**	Camel 2.9: 使用 するオプションのマーカー名。
groupSize	**null**	**整数**	スループットロギングのグループサイズを指定する整数。
groupInterval	**null**	**整数**	指定した場合、この時間間隔（ミリ秒単位）でメッセージ統計をグループ化します。
groupDelay	**0**	**整数**	統計の初期遅延を設定します（ミリ秒単位）。
groupActiveOnly	**true**	**boolean**	true の場合、新しいメッセージが時間間隔で受信されない場合に統計を非表示にします。false の場合は、メッセージトラフィックに関係なく統計を表示します。
logger		**ロガー**	Camel 2.12.4/2.13.1: 使用するレジストリーから **org.slf4j.Logger** へのオプションの参照。

フォーマット

ログにより、ログ行へのエクスチェンジの実行がフォーマットされます。デフォルトでは、ログは LogFormatter を使用してログ出力をフォーマットします。LogFormatter には以下のオプションが含まれます。

オプション	デフォルト	説明

showAll	**false**	すべてのオプションをオンにするクイックオプション（マルチライン、maxChars を使用する場合は手動で設定する必要があります）。
showExchangeId	**false**	一意のエクスチェンジ ID を表示します。
showExchangePattern	**true**	は、Message Exchange Pattern（略して MEP）を示しています。
showProperties	**false**	エクスチェンジプロパティーを表示します。
showHeaders	**false**	In メッセージヘッダーを表示します。
skipBodyLineSeparator	**true**	**Camel 2.12.2:** メッセージボディーをログに記録するときに行セパレーターをスキップするかどうか。これにより、はメッセージボディーを 1 行でログに記録できます。このオプションを **false** に設定すると、本文から行区切り文字が保持され、ボディー **は として ログに記録され** ます。
showBodyType	**true**	In body Java タイプを表示します。
showBody	**true**	In ボディーを表示します。
showOut	**false**	エクスチェンジに Out メッセージがある場合は、Out メッセージが表示されます。
showException	**false**	**Apache Camel 2.0:** エクスチェンジに例外がある場合は、例外メッセージを表示します（スタックトレースなし）。
showCaughtException	**false**	**Apache Camel 2.0:** エクスチェンジにキャッチされた例外がある場合は、例外メッセージを表示します（スタックトレースなし）。キャッチされた例外はエクスチェンジのプロパティーとして保存され、たとえば **doCatch** は例外をキャッチできます。Try Catch Finally を参照してください。

showStackTrace	**false**	Apache Camel 2.0: エクスチェンジに例外がある場合にスタックトレースを表示します。**showAll**、**showException**、または **showCaughtException** のいずれかが有効な場合にのみ有効です。
showFiles	**false**	**Camel 2.9:** Camel がファイルの本文を表示するかどうか(java.io.File など)。
showFuture	**false**	Camel で **java.util.concurrent.Future** ボディーを表示するかどうか。有効な場合、Camel は **Future** タスクが実行されるまで待機できる可能性があります。デフォルトでは待機しません。
showStreams	**false**	**Camel 2.8:** Camel がストリーム本文を表示するかどうか (java.io.InputStream など)。このオプションを有効にすると、このロガーによってストリームがすでに読み込まれているため、後でメッセージボディーにアクセスできないことがあります。これを修正するには、Stream Caching を使用する必要があります。
複数行	**false**	**true** の場合、各情報が新しい行に記録されます。
maxChars		1 行に記録される文字数を制限します。デフォルト値は、**Camel** 2.9 以降の **10000** です。

ロギングストリーム本文

*上記の **showFiles** または **showStreams** プロパティーをサポートしない古いバージョンの **Camel** では、代わりに CamelContext で以下のプロパティーを設定して、ストリームとファイル本文の両方をログに記録できます。*

```
camelContext.getProperties().put(Exchange.LOG_DEBUG_BODY_STREAMS, true);
```

通常のロガーの例

以下のルートでは、注文の処理前に DEBUG レベルで受信注文をログに記録します。

```
from("activemq:orders").to("log:com.mycompany.order?
level=DEBUG").to("bean:processOrder");
```

または、Spring XML を使用してルートを定義します。

```
<route>
  <from uri="activemq:orders"/>
  <to uri="log:com.mycompany.order?level=DEBUG"/>
  <to uri="bean:processOrder"/>
</route>
```

フォーマッターサンプルを持つ通常のロガー

以下のルートでは、注文が処理される前に INFO レベルで受信注文をログに記録します。

```
from("activemq:orders").
    to("log:com.mycompany.order?showAll=true&multiline=true").to("bean:processOrder");
```

GROUPSIZE サンプルを使用したスループットロガー

以下のルートでは、10 メッセージ別にグループ化された DEBUG レベルで受信注文のスループットをログに記録します。

```
from("activemq:orders").
    to("log:com.mycompany.order?level=DEBUG&groupSize=10").to("bean:processOrder");
```

GROUPINTERVAL サンプルを使用したスループットロガー

このルートは、メッセージ統計が 10 ごとにログに記録され、最初の 60 秒の遅延と統計がメッセージトラフィックがなくても表示されます。

```
from("activemq:orders")
    .to("log:com.mycompany.order?
level=DEBUG&groupInterval=10000&groupDelay=60000&groupActiveOnly=false")
    .to("bean:processOrder");
```

以下がログに記録されます。

> *"Received: 1000 new messages, with total 2000 so far. Last group took: 10000 millis which is: 100 messages per second. average: 100"*

ロギング出力の完全なカスタマイズ

Camel 2.11 から利用可能

フォーマット セクションで説明されているオプションを使用すると、ロガーの出力の多くを制御できます。ただし、ログの行は常に以下の構造に従います。

> *Exchange[Id:ID-machine-local-50656-1234567901234-1-2, ExchangePattern:InOut, Properties:{CamelToEndpoint=log://org.apache.camel.component.log.TEST?showAll=true, CamelCreatedTimestamp=Thu Mar 28 00:00:00 WET 2013}, Headers:{breadcrumbId=ID-machine-local-50656-1234567901234-1-1}, BodyType:String, Body:Hello World, Out: null]*

この形式は、おそらく必要なため、場合によっては適さない場合があります。

- *... 出力されたヘッダーとプロパティーをフィルターリングすると、知見と詳細度のバランスが取れます。*

- *... 最も読み取り可能と見なされるものに合わせてログメッセージを調整します。*

- *...Splunk などのログマイニングシステムによるダイジェストのためにログメッセージを調整します。*

-

  ... 特定のボディータイプの出力が異なります。

-

  ... etc.

絶対カスタマイズが必要な場合は、ExchangeFormatter インターフェイスを実装するクラスを作成できます。完全なエクスチェンジにアクセスできる format (Exchange) メソッド内で、必要な正確な情報を選択して抽出し、カスタム方式でフォーマットして返すことができます。戻り値は最終的なログメッセージになります。

Log コンポーネントには、以下のいずれかの方法でカスタム ExchangeFormatter を選択できます。

レジストリーで LogComponent を明示的にインスタンス化します。

```
<bean name="log" class="org.apache.camel.component.log.LogComponent">
  <property name="exchangeFormatter" ref="myCustomFormatter" />
</bean>
```

設定に対する規則：

単に Bean を logFormatter という名前で登録します。Log コンポーネントは、自動的に取得できるほどインテリジェントなものです。

```
<bean name="logFormatter" class="com.xyz.MyCustomExchangeFormatter" />
```

注記： Exchange フォーマッター は Camel コンテキスト内のすべてのログエンドポイントに適用されます。異なるエンドポイントに異なる ExchangeFormatters が必要な場合は、必要に応じて LogComponent をインスタンス化し、関連する Bean 名をエンドポイント接頭辞として使用します。

カスタムログフォーマッターの使用時に Camel 2.11.2/2.12 以降では、カスタムログフォーマッターに設定されるログ URI でパラメーターを指定できます。ただし、logFormatter をプロトタイプスコー

プとして定義し、異なるパラメーターがある場合は共有しないようにします。以下に例を示します。

```
<bean name="logFormatter" class="com.xyz.MyCustomExchangeFormatter"
scope="prototype"/>
```

その後、さまざまなオプションでログ URI を使用して Camel ルートを設定できます。

```
<to uri="log:foo?param1=foo&aram2=100"/>
...
<to uri="log:bar?param1=bar&aram2=200"/>
```

OSGI でのログコンポーネントの使用

Camel 2.12.4/2.13.1 以降の改善

OSGi 内で Log コンポーネントを使用する場合(Karaf でなど)、基礎となるロギングメカニズムは PAX ロギングによって提供されます。org.slf4j.LoggerFactory.getLogger() メソッドを呼び出すバンドルを検索し、バンドルをロガーインスタンスに関連付けます。カスタム org.sfl4j.Logger インスタンスを指定しない場合、Log コンポーネントによって作成されたロガーは camel-core バンドルに関連付けられます。

シナリオによっては、ロガーに関連付けられたバンドルがルート定義が含まれるバンドルである必要があります。これを実行するには、レジストリーに org.slf4j.Logger の単一インスタンスを登録するか、または logger URI パラメーターを使用して参照します。

第101章 LUCENE

LUCENE (INDEXER および SEARCH)コンポーネント

Apache Camel 2.2 で利用可能

lucene コンポーネントは Apache Lucene プロジェクトに基づいています。Apache Lucene は、Java で完全に記述された、高性能でフル機能のテキスト検索エンジンライブラリーです。Lucene の詳細は、以下のリンクを参照してください。

- http://lucene.apache.org/java/docs/
- http://lucene.apache.org/java/docs/features.html

camel の lucene コンポーネントは、エンタープライズ統合パターンおよびシナリオで Lucene エンドポイントの統合および使用状況を容易にします。lucene コンポーネントは以下を行います。

- ペイロードが Lucene エンドポイントに送信される際にドキュメントの検索可能なインデックスを構築します。
- Apache Camel でのインデックス検索の実行を容易にする

このコンポーネントはプロデューサーエンドポイントのみをサポートします。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
lucene:searcherName:insert[?options]
lucene:searcherName:query[?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプションの挿入

名前	デフォルト値	説明
アナライザー	**StandardAnalyzer**	Analyzer は、テキストを分析する TokenStreams をビルドします。そのため、テキストからインデックス用語を抽出するポリシーを表します。analyzer の値は、抽象クラス org.apache.lucene.analysis.Analyzer を拡張する任意のクラスにすることができます。Lucene は、追加設定なしで豊富なアナライザーのセットも提供します。
indexDir	**./indexDirectory**	指定されたアナライザーによるドキュメントの分析時にインデックスファイルが作成されるファイルシステムディレクトリー
srcDir	**null**	プロデューサー起動時にインデックスに分析および追加するために使用されるファイルを含むオプションのディレクトリーです。

クエリーオプション

名前	デフォルト値	説明
アナライザー	**StandardAnalyzer**	Analyzer は、テキストを分析する TokenStreams をビルドします。そのため、テキストからインデックス用語を抽出するポリシーを表します。analyzer の値は、抽象クラス org.apache.lucene.analysis.Analyzer を拡張する任意のクラスにすることができます。Lucene は、追加設定なしで豊富なアナライザーのセットも提供します。
indexDir	**./indexDirectory**	指定されたアナライザーによるドキュメントの分析時にインデックスファイルが作成されるファイルシステムディレクトリー

maxHits	10	検索操作の結果セットを制限する整数値

メッセージヘッダー

ヘッダー	説明
QUERY	インデックスで実行される Lucene クエリー。クエリーにはワイルドカードとフレーズが含まれる場合があります。
RETURN_LUCENE_DOCS	Camel 2.15: ヒット情報を返すときに実際の Lucene ドキュメントを含めるには、このヘッダーを true に設定します。

LUCENE プロデューサー

このコンポーネントは、2 つのプロデューサーエンドポイントをサポートします。

- *insert - 挿入プロデューサーは、受信エクスチェンジのボディーを分析し、それをトークン (content)に関連付けることで、検索可能なインデックスを構築します。*
- *query: クエリープロデューサーは、事前に作成されたインデックスで検索を実行します。クエリーは、検索可能なインデックスを使用してスコアおよび関連性ベースの検索を実行します。受信エクスチェンジを介してクエリーを送信すると、QUERY というヘッダープロパティー名が含まれます。ヘッダープロパティー QUERY の値は Lucene クエリーです。Lucene クエリーを作成する方法は、 http://lucene.apache.org/java/3_0_0/queryparsersyntax.html を参照してください。*

LUCENE プロセッサー

プロデューサーを作成せずに lucene に対してクエリーを実行できる LuceneQueryProcessor と呼ばれるプロセッサーがあります。

例 1: LUCENE インデックスの作成

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      from("direct:start").
          to("lucene:whitespaceQuotesIndex:insert?
analyzer=#whitespaceAnalyzer&indexDir=#whitespace&srcDir=#load_dir").
```

```
            to("mock:result");
    }
};
```

例 2: CAMEL CONTEXT の JNDI レジストリーへのプロパティーのロード

```
@Override
protected JndiRegistry createRegistry() throws Exception {
  JndiRegistry registry =
         new JndiRegistry(createJndiContext());
  registry.bind("whitespace", new File("./whitespaceIndexDir"));
  registry.bind("load_dir",
        new File("src/test/resources/sources"));
  registry.bind("whitespaceAnalyzer",
        new WhitespaceAnalyzer());
  return registry;
}
...
CamelContext context = new DefaultCamelContext(createRegistry());
```

例 2: クエリープロデューサーを使用した検索の実行

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
       from("direct:start").
          setHeader("QUERY", constant("Seinfeld")).
          to("lucene:searchIndex:query?
analyzer=#whitespaceAnalyzer&indexDir=#whitespace&maxHits=20").
          to("direct:next");

       from("direct:next").process(new Processor() {
          public void process(Exchange exchange) throws Exception {
             Hits hits = exchange.getIn().getBody(Hits.class);
             printResults(hits);
          }

          private void printResults(Hits hits) {
              LOG.debug("Number of hits: " + hits.getNumberOfHits());
              for (int i = 0; i < hits.getNumberOfHits(); i++) {
                 LOG.debug("Hit " + i + " Index Location:" + hits.getHit().get(i).getHitLocation());
                 LOG.debug("Hit " + i + " Score:" + hits.getHit().get(i).getScore());
                 LOG.debug("Hit " + i + " Data:" + hits.getHit().get(i).getData());
              }
           }
       }).to("mock:searchResult");
   }
};
```

例 3: クエリープロセッサーを使用した検索の実行

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
```

```
        try {
            from("direct:start").
                setHeader("QUERY", constant("Rodney Dangerfield")).
                process(new LuceneQueryProcessor("target/stdindexDir", analyzer, null, 20)).
                to("direct:next");
        } catch (Exception e) {
            e.printStackTrace();
        }

        from("direct:next").process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                Hits hits = exchange.getIn().getBody(Hits.class);
                printResults(hits);
            }

            private void printResults(Hits hits) {
                LOG.debug("Number of hits: " + hits.getNumberOfHits());
                for (int i = 0; i < hits.getNumberOfHits(); i++) {
                    LOG.debug("Hit " + i + " Index Location:" + hits.getHit().get(i).getHitLocation());
                    LOG.debug("Hit " + i + " Score:" + hits.getHit().get(i).getScore());
                    LOG.debug("Hit " + i + " Data:" + hits.getHit().get(i).getData());
                }
            }
        }).to("mock:searchResult");
    }
};
```

第102章 MAIL

メールコンポーネント

mail コンポーネントは、Spring のメールサポートと基盤の JavaMail システムを介して電子メールへのアクセスを提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-mail</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

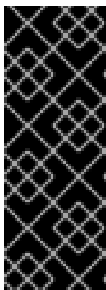
GERONIMO MAIL .JAR

添付ファイルのあるメールをポーリングする際に、geronimo メール .jar (v1.6) にバグがあることが発見されました。Content-Type を正しく特定できません。したがって、.jpeg ファイルをメールに添付し、ポーリングすると、Content-Type は image/jpeg ではなく text/plain として解決されます。このため、org.apache.camel.component.ContentTypeResolver SPI インターフェイスが追加され、独自の実装を提供でき、ファイル名に基づいて正しい Mime タイプを返すことでこのバグを修正できるようになりました。そのため、ファイル名が jpeg/jpg で終わる場合は、image/jpeg を返すことができます。

MailComponent インスタンスまたは MailEndpoint インスタンスでカスタムリゾルバーを設定できます。

POP3 または IMAP

POP3 にはいくつかの制限があり、エンドユーザーは可能であれば IMAP を使用することが推奨されます。



*テストでの **MOCK-MAIL** の使用*

*ユニットテストにはモックフレームワークを使用できます。これにより、実際のメールサーバーを必要とせずにテストできます。ただし、実稼働環境などにメールを実際のメールサーバーに送信する必要がある場合、モックメールを含めないでください。クラスパスに **mock-javamail.jar** が存在するだけで、メールの送信が回避されます。*

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

デフォルトでは、Camel は独自のメールセッションを作成し、このセッションを使用してメールサーバーと対話します。しかし、JBoss EAP にはすでに mail サブシステムがあり、セキュアな接続に関連するすべてのサポート、ユーザー名とパスワードの暗号化などが提供されます。JBoss EAP 設定内でメールセッションを設定し、JNDI を使用して Camel エンドポイントに接続することが推奨されます。

詳細は、Camel-Mail Configuration を参照してください。

URI 形式

メールエンドポイントには、以下の URI 形式のいずれかを使用できます（プロトコル、SMTP、POP3、または IMAP 用）。

```
smtp://[username@]host[:port][?options]
pop3://[username@]host[:port][?options]
imap://[username@]host[:port][?options]
```

mail コンポーネントは、これらのプロトコルのセキュアなバリアント(SSL で階層化)にも対応しています。スキームに s を追加して、セキュアなプロトコルを有効にできます。

```
smtps://[username@]host[:port][?options]
pop3s://[username@]host[:port][?options]
imaps://[username@]host[:port][?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

サンプルエンドポイント

通常、以下のようにログイン認証情報で URI を指定します（例：SMTP を使用）。

```
smtp://[username@]host[:port][?password=somepwd]
```

または、クエリーオプションとしてユーザー名とパスワードの両方を指定することもできます。

```
smtp://host[:port]?password=somepwd&username=someuser
```

以下に例を示します。

```
smtp://mycompany.mailserver:30?password=tiger&username=scott
```

デフォルトのポート

デフォルトのポート番号がサポートされています。ポート番号を省略すると、Camel はプロトコルに基づいて使用するポート番号を決定します。

Protocol	デフォルトのポート番号
SMTP	**25**
SMTPS	**465**
POP3	**110**
POP3S	**995**
IMAP	**143**
IMAPS	**993**

オプション

プロパティー	デフォルト	説明

host		接続するホスト名または IP アドレス。
port	「デフォルトのポート」を参照してください。	接続先の TCP ポート番号。
username		メールサーバーのユーザー名。
password	**null**	メールサーバーのパスワード。
ignoreUriScheme	**false**	**false** の場合、Camel はスキームを使用してトランスポートプロトコル(POP、IMAP、SMTP など)を決定します。
contentType	**text/plain**	メールメッセージのコンテンツタイプ。HTML メールには **text/html** を使用します。
folderName	**INBOX**	ポーリングするフォルダー。
destination	**username@host**	**@非推奨**: 代わりに **to** オプションを使用してください。**TO** 受信者（電子メールの受信側）。
上記を以下のように変更します。	**username@host**	TO 受信者（メールの受信側）。複数のメールアドレスはコンマで区切ります。**&** などの特殊文字を含む電子メールアドレスは、**RAW()** 関数で囲む必要があります。これは、URI エンコーディングをバイパスするために使用できます。
replyTo	**alias@host**	**Camel 2.8.4 以降、2.9.1\+** では、Reply-To 受信者（応答メールの受信側）です。複数のメールアドレスはコンマで区切ります。
CC	**null**	CC の受信者（メールの受信側）。複数のメールアドレスはコンマで区切ります。
BCC	**null**	BCC 受信者（メールの受信側）。複数のメールアドレスはコンマで区切ります。
from	**camel@localhost**	FROM メールアドレス。

subject		Camel 2.3 の時点で、送信されるメッセージの Subject。注記：ヘッダーにサブジェクトを設定することは、このオプションよりも優先されます。
peek	**true**	Camel 2.11.3/2.12.2: コンシューマーのみ。メールメッセージを処理する前に、**javax.mail.Message** に peeked のマークを付けます。これは、**IMAPMessage** メッセージタイプにのみ適用されます。peek を使用すると、メールはメールサーバーで **SEEN** とマークされません。これにより、Camel でエラー処理がある場合にメールメッセージをロールバックできます。
delete	**false**	処理後にメッセージを削除します。これには、メールメッセージに **DELETED** フラグを設定します。**false** の場合、**SEEN** フラグが代わりに設定されます。Camel 2.10 以降では、ヘッダーにキー **delete** を設定して、メールを削除するかどうかを決定します。
Unseen	**true**	コンシューマーエンドポイントを設定して、未確認メッセージ（新しいメッセージ）またはすべてのメッセージのみを処理することができます。Camel は常に削除されたメッセージを省略することに注意してください。**true** のデフォルトオプションは、未確認のメッセージだけに絞り込みます。POP3 は **SEEN** フラグをサポートしないため、このオプションは POP3 ではサポートされていません。代わりに IMAP を使用してください。**重要： searchTerm** オプションも使用する場合、このオプションは使用され **ません**。代わりに、**searchTerm** の使用時の未確認を無効にし、**searchTerm.unseen=false** を用語として追加します。

copyTo	**null**	Camel 2.10: コンシューマーのみ。メールメッセージの処理後、指定した名前のメールフォルダーにコピーできます。この設定値を上書きするには、キー **copyTo** のヘッダーを使用します。これにより、実行時に設定されたフォルダー名にメッセージをコピーできます。
fetchSize	**\-1**	ポーリング中に消費するメッセージの最大数を設定します。メールボックスフォルダーに多くのメッセージが含まれる場合、これを使用してメールサーバーの過負荷を回避することができます。デフォルト値の **\-1** はフェッチサイズがなく、すべてのメッセージが消費されることを意味します。値を 0 に設定すると、Camel はメッセージをまったく消費しません。
alternativeBodyHeader	**CamelMailAlternativeBody**	代替メールボディーが含まれる IN メッセージヘッダーへのキーを指定します。たとえば、**テキスト/html** 形式でメールを送信し、HTML 以外のメールクライアント用に代替メール本文を提供する場合は、別のメール本文をヘッダーとして設定します。
debugMode	**false**	基礎となるメールフレームワークでデバッグモードを有効にします。SUN Mail フレームワークは、デフォルトでデバッグメッセージを **System.out** に記録します。
connectionTimeout	**30000**	接続のタイムアウト（ミリ秒単位）。デフォルトは 30 秒です。
consumer.initialDelay	**1000**	ポーリングが開始するまでの時間（ミリ秒単位）。
consumer.delay	**60000**	Camel は、メールサーバーのオーバーロードを防ぐために、デフォルトで 1 分ごとにメールボックスをポーリングします。

consumer.useFixedDelay	**false**	**true** に設定すると、ポーリング間の固定遅延が使用されます。それ以外の場合には、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。
disconnect	**false**	camel **2.8.3/2.9:** ポーリング後にコンシューマーが切断されるかどうか。これを有効にすると、Camel がポーリングごとに強制的に接続します。
closeFolder	**true**	**Camel 2.10.4:** ポーリング後にコンシューマーがフォルダーを閉じるかどうか。このオプションを **false** に設定して **disconnect=false** も持つと、コンシューマーはポーリング間でフォルダーを開いたままにします。
mail.XXX	**null**	追加の java メールプロパティーを設定します。たとえば、POP3 を使用する際に特別なプロパティーを設定する場合は、**mail.pop3.forgettopheaders=true** などの URI にオプションを直接提供できるようになりました。たとえば、mail **.pop3.forgettopheaders=true&mail.mime.encodefilename=true** などの複数のオプションを設定できます。
mapMailMessage	**true**	camel **2.8:** Camel が受信したメールメッセージを Camel ボディー/ヘッダーにマップするかどうかを指定します。true に設定すると、メールメッセージのボディーは Camel IN メッセージのボディーにマッピングされ、メールヘッダーは IN ヘッダーにマッピングされます。このオプションを false に設定すると、IN メッセージには生の **javax.mail.Message** が含まれます。**exchange.getIn（）.getBody(javax.mail.Message.class)** を呼び出すことで、この未加工メッセージを取得できます。

maxMessagesPerPoll	**0**	ポーリングごとに収集するメッセージの最大数を指定します。デフォルトでは最大値は設定されていません。たとえば、1000 などの制限を設定して、サーバーの起動時に数千のファイルがダウンロードされないようにすることができます。このオプションを無効にするには、0 または負の値を設定します。
javaMailSender	**null**	カスタムメール実装を使用するために、プラグ可能な **org.apache.camel.component.mail.JavaMailSender** インスタンスを指定します。
ignoreUnsupportedCharset	**false**	Camel がメールの送信時にローカル JVM でサポートされていない charset を無視するようにするオプション。charset がサポートされていない場合は、**charset=XXX** (**XXX** はサポート対象外の charset を表します)が **コンテンツタイプ** から削除され、代わりにプラットフォームのデフォルトに依存します。
sslContextParameters	**null**	**Camel 2.10:** Registry の **org.apache.camel.util.jsse.SSLContextParameters** オブジェクトへの参照。 この参照は、コンポーネントレベルで設定済みの SSLContextParameters を上書きします。Security Guide および Using the JSSE Configuration Utility の Configuring Transport Security for Camel Components の章 を参照してください。
searchTerm	**null**	**Camel 2.11:** **javax.mail.search.SearchTerm** を参照します。これにより、特定の日付の後に送信されるサブジェクト、ボディー、from などの検索条件に基づいてメールをフィルターリングできます。例については、以下を参照してください。

searchTerm.xxx	**null**	Camel 2.11: **org.apache.camel.component.mail.SimpleSearchTerm** クラスで定義される制限数の用語をサポートするエンドポイント URI から直接検索用語を設定します。例については、以下を参照してください。
sortTerm	**null**	**Camel 2.15:** IMAP が検索したメールをソートするために使用する sortTerms を設定します。最初にレジストリーで**com.sun.mail.imap.sortTerm** の配列を定義し、この URI オプションで参照するには **#name** が必要になる場合があります。 **Camel 2.16:** Camel が内部で変換する URI でソート用語のコンマ区切りリストを指定することもできます。たとえば、日付で降順をソートするには、**sortTerm=reverse,date** を使用します。**com.sun.mail.imap.SortTerm** で定義されているソート用語のいずれかを使用できます。
postProcessAction	**null**	**Camel 2.15:** 通常の処理が終了した後にメールボックスで後処理タスクを実行する場合は、**org.apache.camel.component.mail. MailBoxPostProcessAction** を参照してください。
skipFailedMessage	**false**	**Camel 2.15:** メールコンシューマーが指定のメールメッセージを取得できない場合、このオプションを使用するとメッセージをスキップして次のメールメッセージを取得できます。デフォルトの動作では、コンシューマーは例外を出力し、バッチからのメールは Camel によってルーティングできません。

handleFailedMessage	**false**	**Camel 2.15:** メールコンシューマーが指定のメールメッセージを取得できない場合、このオプションを使用すると、コンシューマーのエラーハンドラーによって原因となった例外を処理できるようになります。コンシューマーでブリッジエラーハンドラーを有効にすると、Camel ルーティングエラーハンドラーは代わりに例外を処理できます。デフォルトの動作では、コンシューマーは例外を出力し、バッチからのメールは Camel によってルーティングできません。
dummyTrustManager	**false**	**Camel 2.17:** すべての証明書を信頼するためにダミーのセキュリティー設定を使用するには、以下を実行します。実稼働ではなく、開発モードにのみ使用してください。
idempotentRepository	**null**	**Camel 2.17:** プラグ可能なリポジトリー org.apache.camel.spi.IdempotentRepository。同じメールボックスからのクラスター消費を許可し、コンシューマーの処理にメールメッセージが有効かどうかを調整します。
idempotentRepositoryRemoveOnCommit	**true**	**Camel 2.17:** idempotent リポジトリーを使用する場合、メールメッセージが正常に処理され、コミットされると、メッセージ ID がべき等リポジトリー（デフォルト）から削除されるか、リポジトリーに保存されます。デフォルトでは、メッセージ ID は一意であり、リポジトリーに格納される値がないことが想定されます。これは、メールメッセージが再度消費されるのを防ぐために表示/移動または削除としてマークされるためです。そのため、メッセージ ID がべき等リポジトリーに保存されるため、値はほとんどありません。ただし、このオプションを使用すると、任意の理由でメッセージ ID を保存できます。

mailUidGenerator		Camel 2.17: カスタムロジックを使用してメールメッセージの UUID を生成できるようにするプラグ可能な **MailUidGenerator**。

SSL サポート

基盤となるメールフレームワークは、SSL サポートを提供します。 必要な Java Mail API 設定オプションを完全に指定して SSL/TLS サポートを設定するか、コンポーネントまたはエンドポイント設定を介して設定された SSLContextParameters オブジェクトを提供することもできます。Security Guide の Configuring Transport Security for Camel Components の章を参照してください。

JSSE 設定ユーティリティーの使用

Camel 2.10 の時点で、mail コンポーネントは Camel JSSE 設定ユーティリティーを介して SSL/TLS 設定をサポートします。 このユーティリティーは、作成する必要のあるコンポーネント固有のコードの量を大幅に減らし、エンドポイントおよびコンポーネントレベルで設定可能です。 以下の例は、mail コンポーネントで ユーティリティーを使用する方法を示しています。

エンドポイントのプログラムによる設定

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/truststore.jks");
ksp.setPassword("keystorePassword");
TrustManagersParameters tmp = new TrustManagersParameters();
tmp.setKeyStore(ksp);
SSLContextParameters scp = new SSLContextParameters();
scp.setTrustManagers(tmp);
Registry registry = ...
registry.bind("sslContextParameters", scp);
...
from(...)
  .to("smtps://smtp.google.com?
username=user@gmail.com&password=password&sslContextParameters=#sslContextParam
eters");
```

エンドポイントの SPRING DSL ベースの設定

```
...
<camel:sslContextParameters id="sslContextParameters">
  <camel:trustManagers>
    <camel:keyStore resource="/users/home/server/truststore.jks"
password="keystorePassword"/>
  </camel:trustManagers>
</camel:sslContextParameters>...
...
```

```
<to uri="smtps://smtp.google.com?
username=user@gmail.com&password=password&sslContextParameters=#sslContextParam
eters"/>...
```

JAVAMAIL の直接設定

Camel は、既知の認証局（デフォルトの JVM 信頼設定）によって発行された証明書のみを信頼する SUN JavaMail を使用します。独自の証明書を発行する場合は、CA 証明書を JVM の Java 信頼/キーストアファイルにインポートする必要があります。デフォルトの JVM 信頼/キーストアファイルを上書きします（詳細は、JavaMail の SSLNOTES.txt を参照してください）。

メールメッセージの内容

Camel はメッセージエクスチェンジの IN ボディーを MimeMessage テキストコンテンツとして使用します。ボディーは String.class に変換されます。

Camel はすべてのエクスチェンジの IN ヘッダーを MimeMessage ヘッダーにコピーします。

MimeMessage のサブジェクトは、IN メッセージのヘッダープロパティーを使用して設定できます。以下のコードはこれを示しています。

```
from("direct:a").setHeader("subject", constant(subject)).to("smtp://james2@localhost");
```

recipients などの他の MimeMessage ヘッダーにも適用されるため、以下のようにヘッダープロパティー を 使用できます。

```
Map<String, Object> map = new HashMap<String, Object>();
map.put("To", "davsclaus@apache.org");
map.put("From", "jstrachan@apache.org");
map.put("Subject", "Camel rocks");

String body = "Hello Claus.\nYes it does.\n\nRegards James.";
template.sendBodyAndHeaders("smtp://davsclaus@apache.org", body, map);
```

Camel 2.11 から MailProducer をサーバーに送信する場合、Camel メッセージヘッダーから CamelMailMessageId キーを持つ MimeMessage のメッセージ ID を取得できるはずです。

ヘッダーが事前設定された受信者よりも優先されます。

メッセージヘッダーに指定された受信者は常にエンドポイント URI で事前設定された受信者よりも優

先されます。メッセージヘッダーに受信者を提供する場合は、取得する内容になります。エンドポイント URI で事前設定された受信者はフォールバックとして処理されます。

以下のコード例では、メールメッセージは davsclaus@apache.org に送信されます。これは、事前設定された受信者 info@mycompany.com よりも優先されます。エンドポイント URI の CC および BCC 設定も無視され、これらの受信者はメールを受信しません。ヘッダーと事前設定された設定の選択は all または nothing です。mail コンポーネントはヘッダーのみから受信者を取得する か、事前設定された設定から排他的に取得します。ヘッダーと事前設定された設定を混在させることはできません。

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("to", "davsclaus@apache.org");

template.sendBodyAndHeaders("smtp://admin@localhost?to=info@mycompany.com",
"Hello World", headers);
```

複数の受信者で設定が容易になります。

コンマ区切りまたはセミコロンで区切られたリストを使用して、複数の受信者を設定できます。これは、ヘッダー設定とエンドポイント URI の設定の両方に適用されます。以下に例を示します。

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("to", "davsclaus@apache.org ; jstrachan@apache.org ;
ningjiang@apache.org");
```

上記の例では、セミコロン ; を区切り文字として使用します。

送信者名と電子メールの設定

受信者は < email> の形式で指定して、受信者の名前とメールアドレスの両方を含めることができます。

たとえば、Message で以下のヘッダーを定義します。

```
Map headers = new HashMap();
map.put("To", "Claus Ibsen <davsclaus@apache.org>");
map.put("From", "James Strachan <jstrachan@apache.org>");
map.put("Subject", "Camel is cool");
```

SUN JAVAMAIL

SUN JavaMail は、メールの消費と生成のフーッドで使用されます。POP3 プロトコルまたは IMAP プロトコルのいずれかを使用する際に、エンドユーザーがこれらの参照を参照することが推奨されます。特に、POP3 の機能は IMAP よりも限定されている点にご留意ください。

- *SUN POP3 API*
- *SUN IMAP API*
- *一般的に MAIL フラグについて*

サンプル

JMS キューから受信したメッセージを電子メールとして送信する単純なルートから始めます。email アカウントは、mymailserver.com の 管理者 アカウントです。

```
from("jms://queue:subscription").to("smtp://admin@mymailserver.com?password=secret");
```

次のサンプルでは、1 分ごとに新しい電子メールのメールボックスをポーリングします。ポーリング間隔 consumer.delay を 60000 ミリ秒 = 60 秒として設定するのに特別なコンシューマーオプションを使用していることに注意してください。

```
from("imap://admin@mymailserver.com
    password=secret&unseen=true&consumer.delay=60000")
   .to("seda://mails");
```

この例では、複数の受信者にメールを送信します。

```
// all the recipients of this mail are:
// To: camel@riders.org , easy@riders.org
// CC: me@you.org
// BCC: someone@somewhere.org
String recipients =
"&To=camel@riders.org,easy@riders.org&CC=me@you.org&BCC=someone@somewhere.org
";

from("direct:a").to("smtp://you@mymailserver.com?
password=secret&From=you@apache.org" + recipients);
```

添付ファイルを使用したメールの送信サンプル

添付ファイルはすべての CAMEL コンポーネントによってサポートされない

接続 API は Java アクティベーションキーフレームワークをベースとしており、一般的に Mail API でのみ使用されます。他の Camel コンポーネントの多くは添付をサポートしないため、添付がルートとともに伝播されるため、アタッチメントが失われる可能性があります。したがって、サムのルールは、メッセージをメールエンドポイントに送信する前に添付ファイルを追加することです。

mail コンポーネントは添付ファイルをサポートします。以下の例では、ロゴファイルアタッチメントのあるプレーンテキストメッセージが含まれるメールメッセージを送信します。

```
// create an exchange with a normal body and attachment to be produced as email
Endpoint endpoint = context.getEndpoint("smtp://james@mymailserver.com?
password=secret");

// create the exchange with the mail message that is multipart with a file and a Hello World
text/plain message.
Exchange exchange = endpoint.createExchange();
Message in = exchange.getIn();
in.setBody("Hello World");
in.addAttachment("logo.jpeg", new DataHandler(new
FileDataSource("src/test/data/logo.jpeg")));

// create a producer that can produce the exchange (= send the mail)
Producer producer = endpoint.createProducer();
// start the producer
producer.start();
// and let it go (processes the exchange by sending the email)
producer.process(exchange);
```

SSL の例

この例では、メールの Google メール受信トレイをポーリングします。メールをローカルメールクライアントにダウンロードするには、Google メールで SSL を有効にして設定する必要があります。これには、Google メールアカウントにログインし、IMAP アクセスを許可するように設定を変更します。Google には、これを実行するための詳細なドキュメントがあります。

```
from("imaps://imap.gmail.com?
username=YOUR_USERNAME@gmail.com&password=YOUR_PASSWORD"
    + "&delete=false&unseen=true&consumer.delay=60000").to("log:newmail");
```

上記のルートは、新しいメールを 1 分ごとに新しいメールの Google メールインボックスをポーリン

グし、受信したメッセージを newmail ロガーカテゴリーに記録します。DEBUG ロギングを有効にしてサンプルを実行すると、ログの進捗をモニターできます。

```
2008-05-08 06:32:09,640 DEBUG MailConsumer - Connecting to MailStore
imaps//imap.gmail.com:993 (SSL enabled), folder=INBOX
2008-05-08 06:32:11,203 DEBUG MailConsumer - Polling mailfolder:
imaps//imap.gmail.com:993 (SSL enabled), folder=INBOX
2008-05-08 06:32:11,640 DEBUG MailConsumer - Fetching 1 messages. Total 1 messages.
2008-05-08 06:32:12,171 DEBUG MailConsumer - Processing message: messageNumber=
[332], from=[James Bond <007@mi5.co.uk>], to=YOUR_USERNAME@gmail.com], subject=[...
2008-05-08 06:32:12,187 INFO  newmail - Exchange[MailMessage: messageNumber=[332],
from=[James Bond <007@mi5.co.uk>], to=YOUR_USERNAME@gmail.com], subject=[...
```

添付サンプルでメールの消費

この例では、メールボックスをポーリングし、メールからのすべての添付ファイルをファイルとして保存します。まず、メールボックスをポーリングするルートを定義します。このサンプルは google メールをベースとしているため、SSL の例と同じルートを使用します。

```
from("imaps://imap.gmail.com?
username=YOUR_USERNAME@gmail.com&password=YOUR_PASSWORD"
  + "&delete=false&unseen=true&consumer.delay=60000").process(new MyMailProcessor());
```

メールをログに記録する代わりに、java コードからのメールを処理できるプロセッサーを使用します。

```
    public void process(Exchange exchange) throws Exception {
        // the API is a bit clunky so we need to loop
        Map<String, DataHandler> attachments = exchange.getIn().getAttachments();
        if (attachments.size() > 0) {
            for (String name : attachments.keySet()) {
                DataHandler dh = attachments.get(name);
                // get the file name
                String filename = dh.getName();

                // get the content and convert it to byte[]
                byte[] data = exchange.getContext().getTypeConverter()
                                  .convertTo(byte[].class, dh.getInputStream());

                // write the data to a file
                FileOutputStream out = new FileOutputStream(filename);
                out.write(data);
                out.flush();
                out.close();
            }
        }
    }
```

ご覧のとおり、添付を処理する API は少し明確ですが、javax.activation.DataHandler を取得して標準 API を使用して添付を処理できるようにすることができます。

添付ファイルを使用したメールメッセージを分割する方法

この例では、複数の添付ファイルがあるメールメッセージを消費します。実行する内容は、個別の添付ファイルごとに Splitter EIP を使用して、添付ファイルを個別に処理することです。たとえば、メールメッセージに 5 つの添付がある場合、Splitter は 1 つの添付を持つ 5 つのメッセージを処理します。これを行うには、Splitter にカスタム 式 を提供する必要があります。ここでは、1 つの添付で 5 つのメッセージが含まれる List<Message> を提供します。

このコードは、camel-mail コンポーネントで Camel 2.10 以降は追加設定なしで提供されます。コードはクラス org.apache.camel.component.mail.SplitAttachmentsExpression にあり、ソースコードは ここにあります。

Camel ルートでは、以下のようにルートでこの 式 を使用する必要があります。

```
from("pop3://james@mymailserver.com?password=secret&consumer.delay=1000")
    .to("log:email")
    // use the SplitAttachmentsExpression which will split the message per attachment
    .split(new SplitAttachmentsExpression())
        // each message going to this mock has a single attachment
        .to("mock:split")
    .end();
```

XML DSL を使用する場合は、以下のように Splitter でメソッド呼び出し式を宣言する必要があります。

```
<split>
  <method beanType="org.apache.camel.component.mail.SplitAttachmentsExpression"/>
  <to uri="mock:split"/>
</split>
```

Camel 2.16 以降では、添付ファイルを byte[] として分割してメッセージボディーとして保存することもできます。これは、ブール値 true で式を作成して行います。以下に例を示します。

```
SplitAttachmentsExpression split = SplitAttachmentsExpression(true);
```

次に、Splitter EIP で式を使用します。

カスタム SEARCHTERM の使用

Camel 2.11 から利用可能

MailEndpoint で searchTerm を設定すると、不要なメールをフィルターリングできます。

たとえば、Subject または Text のいずれかに Camel が含まれるようにメールをフィルターするには、以下のように実行できます。

```
<route>
  <from uri="imaps://mymailseerver?
username=foo&password=secret&searchTerm.subjectOrBody=Camel"/>
  <to uri="bean:myBean"/>
</route>
```

"searchTerm.subjectOrBody" をパラメーターキーとして使用して、メールの件名またはボディーで検索して、Camel という単語を含めることに注意してください。org.apache.camel.component.mail.SimpleSearchTerm クラスには、設定可能な数多くのオプションがあります。

または、新しい未確認のメールが 24 時間後に戻ってきます。now-24h 構文に注意してください。詳細は以下の表を参照してください。

```
<route>
  <from uri="imaps://mymailseerver?
username=foo&password=secret&searchTerm.fromSentDate=now-24h"/>
  <to uri="bean:myBean"/>
</route>
```

エンドポイント URI 設定に複数の searchTerm を指定できます。これらは AND 演算子を使用して組み合わせられるため、両方の条件が一致する必要があります。たとえば、メールの件名に Camel がある 24 時間後に最後に確認されていないメールを取得するには、以下を実行できます。

```
<route>
  <from uri="imaps://mymailseerver?
username=foo&password=secret&searchTerm.subject=Camel&searchTerm.fromSentDate=no
w-24h"/>
  <to uri="bean:myBean"/>
</route>
```

オプション	デフォルト	説明
unseen	**true**	未確認のメールのみで制限するかどうか。
subjectOrBody	**null**	単語を含む対象またはボディーで制限します。
subject	**null**	サブジェクトには という単語が含まれている必要があります。
body	**null**	本文には という単語が含まれている必要があります。
from	**null**	メールは、指定した電子メールパターンから取得する必要があります。
to	**null**	メールは特定のメールパターンにする必要があります。
fromSentDate	**null**	メールは、指定した日付または等しい(GE)後に送信する必要があります。日付パターンは **yyyy-MM-dd HH:mm:SS** です。たとえば、2012-01- **01 00:00:00 を 2012** 年以降の年から使用します。現在のタイムスタンプ **には now** を使用できます。now 構文はオプションのオフセットをサポートします。これは、数値で + または - のいずれかとして指定できます。たとえば、過去 24 時間の場合は、**now - 24h** を使用するか、スペース **now-24h なしで使用できます**。Camel は時間、分、秒の省略形をサポートすることに注意してください。

toSentDate	null	メールは、指定した日付の前または等しい(BE)に送信する必要があります。日付パターンは **yyyy-MM-dd HH:mm:SS** です。たとえば、2012-01- **01 00:00:00 を 2012** 年前に使用します。現在のタイムスタンプ **にはnow** を使用できます。now 構文はオプションのオフセットをサポートします。これは、数値で + または - のいずれかとして指定できます。たとえば、過去 24 時間の場合は、**now - 24h** を使用するか、スペース **now-24h なしで使用できます**。Camel は時間、分、秒の省略形をサポートすることに注意してください。

SimpleSearchTerm は POJO から簡単に設定できるように設計されています。そのため、XML で <bean> スタイルを使用して設定することもできます。

```
<bean id="mySearchTerm" class="org.apache.camel.component.mail.SimpleSearchTerm">
  <property name="subject" value="Order"/>
  <property name="to" value="acme-order@acme.com"/>
  <property name="fromSentDate" value="now"/>
 </bean>
```

その後、以下のように Camel ルートの #beanId を使用して、この Bean を参照できます。

```
<route>
  <from uri="imaps://mymailseerver?
username=foo&password=secret&searchTerm=#mySearchTerm"/>
  <to uri="bean:myBean"/>
</route>
```

Java には、org.apache.camel.component.mail. SearchTermBuilder クラスを使用して複合 SearchTerm を構築するビルダークラスがあります。これにより、次のような複雑な用語を作成できます。

```
// we just want the unseen mails which is not spam
SearchTermBuilder builder = new SearchTermBuilder();

builder.unseen().body(Op.not, "Spam").subject(Op.not, "Spam")
  // which was sent from either foo or bar
  .from("foo@somewhere.com").from(Op.or, "bar@somewhere.com");
```

```
  // .. and we could continue building the terms

SearchTerm term = builder.build();
```

第103章 マスターコンポーネント

概要

マスターコンポーネントは、クラスターの単一のコンシューマーのみが指定のエンドポイントから消費されるようにする方法を提供します。その JVM が停止した場合に自動フェイルオーバーを使用します。この機能は、同時消費をサポートしていないレガシーバックエンドから消費する必要がある場合や、商用または安定性の理由から、いつでもバックエンドへの接続を 1 つだけ持つことができる場合に役立ちます。

DEPENDENCIES

マスターコンポーネントは、ファブリックが有効な Red Hat JBoss Fuse コンテナーのコンテキストでのみ使用できます。fabric-camel 機能がインストールされていることを確認する必要があります。

Fabric のコンテキストでは、該当するプロファイルに追加して機能をインストールします。たとえば、my-master-profile というプロファイルを使用している場合は、以下のコンソールコマンドを入力して fabric-camel 機能を追加します。

```
karaf@root> fabric:profile-edit --features fabric-camel my-master-profile
```

URI 形式

マスターエンドポイントは、コンシューマーエンドポイント としてのみ使用できます。以下の URI 形式があります。

```
master:ClusterID:EndpointURI[?Options]
```

URI EndpointURI がファブリックレジストリーに公開され、ClusterId クラスターに関連付けられます。

URI オプション

マスターコンポーネント自体は URI オプションをサポートし ません。そのため、URI のオプションは指定されたコンシューマーエンドポイント EndpointURI に適用されます。

マスターコンポーネントの使用方法

マスターコンポーネントは、エンドポイントからメッセージをポーリングする必要がある場合に便利ですが、そのエンドポイントへの 1 つのコネクション のみが可能です。この場合、マスターコンポーネントを使用して、コンシューマーエンドポイントのフェイルオーバークラスターを定義できます。クラスターの各マスターエンドポイントは指定のエンドポイントからメッセージを消費できますが、マスターエンドポイントの 1 つ だけがアクティブになりますが、他のマスターエンドポイントは待機しています（スレーブ）。

*たとえば、**seda:bar** エンドポイントから消費できる **Master** エンドポイントのクラスターを設定するには、以下の手順を実行します。*

1. *以下の **URI** でマスターエンドポイントを定義します（クラスター内の各エンドポイントはまったく 同じ **URI** を使用します）。*

   ```
   master:mysedalock:seda:bar
   ```

   *クラスター内の各マスターエンドポイントは、**mysedalock** ロックを取得しようとします (**Zookeeper** レジストリーのキーとして実装されます)。ロックの取得に成功するマスターエンドポイントはアクティブ（マスター）になり、**seda:bar** エンドポイントからメッセージの使用を開始します。他の **Master** エンドポイントは待機状態に入り、ロック（スレーブ）の試行を続行します。*

2. *マスターエンドポイントをデプロイするプロファイルには、**fabric-camel** 機能を含めるようにしてください。*

3. ***Blueprint XML** では、以下のように **Camel** ルートの開始時に **Master** エンドポイントを定義できます。*

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     ...
     <camelContext id="camel" xmlns="http://camel.apache.org/schema/blueprint">
       <route>
         <from uri="master:mysedalock:seda:bar"/>
         ...
       </route>
     </camelContext>
     ...
   </blueprint>
   ```

JMS ACTIVEMQ ブローカーをポーリングするマスター/スレーブクラスターの例

たとえば、マスターコンポーネントを使用する一般的な方法は、JMS キューからメッセージを消費するための排他的コンシューマーのクラスターを作成することです。マスターエンドポイントの 1 つのみがキューからいつでも消費され、そのマスターエンドポイントがダウンした場合、他のマスターエンドポイントの 1 つが引き継ぎます（新規マスターになります）。この例では、2 つの Camel ルートのクラスターを作成します。ここでは、各ルートは、指定されたキュー FABRIC.DEMO から消費できる Master エンドポイントで始まります。

ACTIVEMQ ブローカーからメッセージをポーリングするクラスターを作成する手順

マスターコンポーネントに基づいて ActiveMQ ブローカーからメッセージをポーリングするマスター/スレーブクラスターを作成するには、以下の手順を実行します。

1. ファブリックがない場合は、以下のコンソールコマンドを入力して作成します。

```
JBossFuse:karaf@root> fabric:create --new-user AdminUser --new-user-password
AdminPass
  --zookeeper-password ZooPass --wait-for-provisioning
```

--new-user および --new-user-password オプションは、新しい管理者ユーザーの認証情報を指定します。Zookeeper パスワードは、Fabric レジストリーサービス(/fabric下のすべてのノード)の機密データを保護するために使用されます。

注記

ローカルマシンで VPN（仮想プライベートネットワーク）を使用する場合は、ローカルコンテナーの使用中にファブリックを作成し、ログオフしたままに VPN をログに記録することが推奨されます。ローカル Fabric Server は、固定 IP アドレスまたはホスト名に永続的に関連付けられます。ファブリックの作成時に VPN が有効になっている場合、基礎となる Java ランタイムは、永続的なローカルホスト名ではなく VPN ホスト名を検出して使用することができます。これは、マルチホームマシンでも問題になる可能性があります。ホスト名について絶対的に確認するには、IP アドレスを明示的に指定できます。Fabric Guide の Creating a New Fabric の章を参照 してください。

2. この例では、Apache ActiveMQ ブローカーの実行中のインスタンスにアクセスでき、ブローカーの OpenWire コネクターの TCP ポートを知っている必要があります。たとえば、以下のいずれかの方法で ActiveMQ ブローカーにアクセスできます。

   - JBoss Fuse のクリーンインストールでファブリックを作成したばかりです（コールド再起動後）。この場合、root コンテナーはデフォルトで jboss-fuse-full プロファイルを含

める必要があります。以下のように fabric:container-list コンソールコマンドを入力して、このケースを確認できます。

```
JBossFuse:karaf@root> fabric:container-list
[id]    [version] [connected] [profiles]                              [provision status]
root*   1.0       true        fabric, fabric-ensemble-0000-1, jboss-fuse-full    success
```

デフォルトでは、jboss-fuse-full プロファイルはポート 61616 でリッスンする ActiveMQ ブローカーをインスタンス化します。現在の例ではこのブローカーを使用できます。

- ルートコンテナー（またはその他のコンテナー）で実行されているブローカーがない場合、コンソールプロンプトで以下の fabric コマンドを入力して、ブローカーを新しいファブリック子コンテナー broker1 にすばやくインストールできます。

```
JBossFuse:karaf@root> fabric:container-create-child --profile mq-default root
broker1
```

この場合、ブラウザーベースの Fuse 管理コンソールを使用して、ブローカーの OpenWire コネクターの TCP ポートを検出できます。

3. Master コンポーネントを使用する単純な Apache Camel ルートをデプロイするために使用される master-example プロファイルを作成します。以下のコンソールコマンドを実行してプロファイルを作成します。

```
JBossFuse:karaf@root> fabric:profile-create --parents default master-example
```

4. 必要な Karaf 機能を master-example プロファイルに追加します。以下のコンソールコマンドを入力します。

```
fabric:profile-edit --features fabric-camel master-example
fabric:profile-edit --features activemq-camel master-example
```

5. master-example プロファイルで、簡単な Camel ルートをリソースとして定義します。ビルトインテキストエディターを呼び出して、以下のように camel.xml の新しいリソースを作成します。

```
fabric:profile-edit --resource camel.xml master-example
```

以下の内容を組み込みテキストエディターにコピーして貼り付けます。

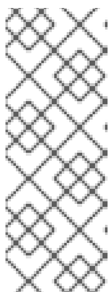
```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <camelContext id="camel" xmlns="http://camel.apache.org/schema/blueprint">
      <route id="fabric-server">
         <from uri="master:lockhandle:activemq:queue:FABRIC.DEMO"/>
         <log message="Message received : ${body}"/>
      </route>
   </camelContext>

   <bean id="activemq"
        class="org.apache.activemq.camel.component.ActiveMQComponent">
      <property name="brokerURL" value="tcp://localhost:OpenWirePort"/>
      <property name="userName" value="UserName"/>
      <property name="password" value="Password"/>
   </bean>

</blueprint>
```

OpenWirePort をブローカーの OpenWire コネクターのポート番号に置き換え、UserName および Password をコンテナー上の有効な JAAS クレデンシャルに置き換えることで、ルート設定をカスタマイズするようにしてください（たとえば、これらの手順の 手順 1 で作成した AdminUser および AdminPass クレデンシャルを置き換えることができます）。

テキストエディターを保存して終了するには、Ctrl-S, Ctrl-X と入力します。

6. camel.xml リソースを OSGi バンドルとしてデプロイするように master-example プロファイルを設定します。以下のコンソールコマンドを入力し、master-example エージェントプロパティーに新しいエントリーを作成します。

```
fabric:profile-edit --bundles blueprint:profile:camel.xml master-example
```



注記

blueprint: 接頭辞は Fabric に対して、指定されたリソースを Blueprint XML ファイルとしてデプロイするように指示します。profile: 接頭辞は Fabric に対して、リソース（現在のプロファイルの現在のバージョン）の場所を指示します。

7. master-example プロファイルをクラスターとしてデプロイできるように、新しい子コンテ

ナー 2 つを作成します(1 つのマスターとスレーブ 1 つ)。以下のコンソールコマンドを入力します。

```
fabric:container-create-child root child 2
```

8. 以下のように、master-example プロファイルと mq-client プロファイルの両方を各子コンテナーにデプロイします。

```
fabric:container-change-profile child1 master-example mq-client
fabric:container-change-profile child2 master-example mq-client
```

9. メッセージをブローカーの FABRIC.DEMO キューに送信すると、メッセージはデプロイされたマスターエンドポイントの 1 つ（および 1 つだけ）によって消費されます。たとえば、ブラウザーベースの Fuse Management コンソールを使用して、メッセージをブローカーに簡単に作成および送信できます。

10. 現在のマスターをホストするコンテナーを停止する場合（最初に child1 コンテナー）、スレーブは新規マスター(child2 コンテナー内)に昇格され、FABRIC.DEMO キューからのメッセージの消費を開始します。たとえば、child2 に現在のマスターが含まれていると仮定すると、以下のコンソールコマンドを入力して停止できます。

```
fabric:container-stop child2
```

OSGI バンドルプラグインの設定

マスターエンドポイントを使用する OSGi バンドルを定義する場合は、以下の Java パッケージをインポートするように Import-Package バンドルヘッダーを設定する必要があります。

```
io.fabric8.zookeeper
```

たとえば、Maven を使用してアプリケーションをビルドする場合、例103.1「Maven バンドルプラグインの設定」 は、必要なパッケージをインポートするように Maven バンドルプラグインを設定する方法を示しています。

例103.1 Maven バンドルプラグインの設定

```
<project ... >
  ...
  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
```

```
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${project.groupId}.${project.artifactId}</Bundle-
SymbolicName>
            <Import-Package>
              io.fabric8.zookeeper,
              *
            </Import-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

第104章 メトリクス

メトリクスコンポーネント

Camel 2.14 から利用可能

metrics: コンポーネントを使用すると、Camel ルートからさまざまなメトリクスを直接収集できます。サポートされるメトリックタイプは、カウンター、ヒストグラム、メーター、タイマー です。メトリクス を使用すると、アプリケーションの動作を測定するための簡単な方法が提供されます。設定可能なレポートバックエンドは、統計を収集し、視覚化するためのさまざまな統合オプションを有効にします。コンポーネントは、MetricsRoutePolicyFactory も提供します。これにより、codehale メトリクスを使用してルート統計を公開できます。詳細は、ページ下部を参照してください。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-metrics</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
metrics:[ meter | counter | histogram | timer ]:metricname[?options]
```

METRIC REGISTRY

Camel Metrics Component はデフォルトで MetricRegistry を使用し、Slf4jReporter と 60 秒のレポート間隔を使用します。デフォルトレジストリーは、Camel レジストリーで metricRegistry という名前の Bean を提供することで、カスタムレジストリーに置き換えることができます。たとえば、Spring Java Configuration を使用します。

```
@Configuration
public static class MyConfig extends SingleRouteCamelConfiguration {

    @Bean
    @Override
    public RouteBuilder route() {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                // define Camel routes here
            }
```

```
        };
    }

    @Bean(name = MetricsComponent.METRIC_REGISTRY_NAME)
    public MetricRegistry getMetricRegistry() {
        MetricRegistry registry = ...;
        return registry;
    }
}
```

警告

MetricRegistry レポートに内部スレッドを使用します。バージョン 3.0.1 には、ユーザーが終了時にクリーンアップするパブリック API はありません。そのため、Camel Metrics Component を使用すると、Java クラ出力ダーがリークし、OutOfMemoryErrors が発生する可能性があります。

使用方法

各メトリクスには type および name があります。サポートされるタイプは、カウンター、ヒストグラム、メーター、タイマー です。メトリック名は単純な文字列です。メトリックタイプが指定されていない場合は、デフォルトでタイプ meter が使用されます。

HEADERS

URI で定義されたメトリクス名は、ヘッダーと名前 CamelMetricsName を使用して上書きできます。

以下に例を示します。

```
from("direct:in")
    .setHeader(MetricsConstants.HEADER_METRIC_NAME, constant("new.name"))
    .to("metrics:counter:name.not.used")
    .to("direct:out");
```

name.not.used ではなく、new.name という名前のカウンターを更新します。

Metrics エンドポイントがエクスチェンジの処理を終了すると、Metrics 固有のヘッダーはすべてメッセージから削除されます。エクスチェンジのエンドポイントの処理中に、Metrics エンドポイントはすべての例外をキャッチし、レベル warn を使用してログエントリーを書き込みます。

メトリクスタイプのカウンター

```
metrics:counter:metricname[?options]
```

オプション

名前	デフォルト	説明
Increment	-	カウンターに追加する長い値
decrement	-	カウンターから減算する長い値

increment または decrement が定義されていない場合、カウンター値は 1 つずつ増えます。increment と decrement の両方が定義されている場合は、インクリメント操作のみが呼び出されます。

```
// update counter simple.counter by 7
from("direct:in")
    .to("metric:counter:simple.counter?increment=7")
    .to("direct:out");
```

```
// increment counter simple.counter by 1
from("direct:in")
    .to("metric:counter:simple.counter")
    .to("direct:out");
```

```
// decrement counter simple.counter by 3
from("direct:in")
    .to("metric:counter:simple.counter?decrement=3")
    .to("direct:out");
```

HEADERS

メッセージヘッダーは、Metrics コンポーネント URI で指定される increment および decrement の値を上書きするために使用できます。

名前	説明	想定されるタイプ
CamelMetricsCounterIncrement	URI のインクリメント値を上書きします。	Long
CamelMetricsCounterDecrement	URI のデクリメント値の上書き	Long

```
// update counter simple.counter by 417
from("direct:in")
    .setHeader(MetricsConstants.HEADER_COUNTER_INCREMENT, constant(417L))
    .to("metric:counter:simple.counter?increment=7")
    .to("direct:out");
```

```
// updates counter using simple language to evaluate body.length
from("direct:in")
    .setHeader(MetricsConstants.HEADER_COUNTER_INCREMENT, simple("${body.length}"))
    .to("metrics:counter:body.length")
    .to("mock:out");
```

メトリックのタイプヒストグラム

```
metrics:histogram:metricname[?options]
```

オプション

名前	デフォルト	説明
value	-	ヒストグラムで使用する値

value が設定されていない場合、ヒストグラムには何も追加されず、警告が記録されます。

```
// adds value 9923 to simple.histogram
from("direct:in")
    .to("metric:histogram:simple.histogram?value=9923")
    .to("direct:out");
```

```
// nothing is added to simple.histogram; warning is logged
from("direct:in")
    .to("metric:histogram:simple.histogram")
    .to("direct:out");
```

HEADERS

メッセージヘッダーを使用して、Metrics コンポーネント URI に指定された値をオーバーライドでき

ます。

名前	説明	想定されるタイプ
CamelMetricsHistogramValue	URI のヒストグラム値の上書き	Long

```
// adds value 992 to simple.histogram
from("direct:in")
    .setHeader(MetricsConstants.HEADER_HISTOGRAM_VALUE, constant(992L))
    .to("metric:histogram:simple.histogram?value=700")
    .to("direct:out")
```

メトリックタイプのメーター

```
metrics:meter:metricname[?options]
```

オプション

名前	デフォルト	説明
mark	-	マークとして使用する長い値

mark が設定されていない場合、meter.mark() は引数なしで呼び出されます。

```
// marks simple.meter without value
from("direct:in")
    .to("metric:simple.meter")
    .to("direct:out");
```

```
// marks simple.meter with value 81
from("direct:in")
    .to("metric:meter:simple.meter?mark=81")
    .to("direct:out");
```

HEADERS

メッセージヘッダーを使用して、Metrics コンポーネント URI に指定された mark 値を上書きできます。

名前	説明	想定されるタイプ
CamelMetricsMeterMark	URI のマーク値の上書き	Long

```
// updates meter simple.meter with value 345
from("direct:in")
    .setHeader(MetricsConstants.HEADER_METER_MARK, constant(345L))
    .to("metric:meter:simple.meter?mark=123")
    .to("direct:out");
```

メトリクスタイプタイマー

```
metrics:timer:metricname[?options]
```

オプション

名前	デフォルト	説明
action	-	start または stop

action または無効な値が指定されていない場合、警告はタイマー更新なしでログに記録されます。実行中のタイマーまたは stop でアクション start が呼び出されている場合は、何も更新されず、警告が記録されます。

```
// measure time taken by route "calculate"
from("direct:in")
    .to("metrics:timer:simple.timer?action=start")
    .to("direct:calculate")
    .to("metrics:timer:simple.timer?action=stop");
```

TimerContext オブジェクトは、異なる Metrics コンポーネント呼び出し間で Exchange プロパティーとして保存されます。

HEADERS

メッセージヘッダーは、Metrics コンポーネント URI に指定されたアクション値を上書きするために使用できます。

名前	説明	想定されるタイプ
CamelMetricsTimerAction	URI でのタイマーアクションの上書き	**org.apache.camel.component.metrics.timer.TimerEndpoint.TimerAction**

```
// sets timer action using header
from("direct:in")
    .setHeader(MetricsConstants.HEADER_TIMER_ACTION, TimerAction.start)
```

```
    .to("metric:timer:simple.timer")
    .to("direct:out");
```

METRICSROUTEPOLICYFACTORY

このファクトリーを使用すると、コード化されたメトリクスを使用してルートの使用状況の統計を公開する各ルートに RoutePolicy を追加できます。このファクトリーは、以下の例のように Java および XML で使用できます。

ヒント

MetricsRoutePolicyFactory を使用する代わりに、選択したいくつかのルートのみをインストルメント化する場合に、インストルメント化するルートごとに MetricsRoutePolicy を定義できます。

Java の場合は、以下のようにファクトリーを CamelContext に追加します。

```
context.addRoutePolicyFactory(new MetricsRoutePolicyFactory());
```

XML DSL の場合は、以下のように <bean> を定義します。

```
<!-- use camel-metrics route policy to gather metrics for all routes -->
<bean id="metricsRoutePolicyFactory"
class="org.apache.camel.component.metrics.routepolicy.MetricsRoutePolicyFactory"/>
```

MetricsRoutePolicyFactory および MetricsRoutePolicy は以下のオプションをサポートします。

名前	デフォルト	説明
useJmx	**false**	**com.codahale.metrics.JmxReporter** を使用して、詳細な統計を JMX に報告するかどうか。CamelContext で JMX が有効になっている場合、JMX ツリーのサービスタイプの下に **MetricsRegistryService** mbean が登録されていることに注意してください。この mbean には、統計を JSON 出力する 1 つのオペレーションがあります。**useJmx** を true に設定することは、統計タイプごとに詳細な mbeans が必要な場合にのみ必要です。

名前	デフォルト	説明
jmxDomain	**org.apache.camel.metrics**	JMX ドメイン名
prettyPrint	**false**	統計を json 形式で出力する際に pretty print を使用するかどうか。
metricsRegistry		共有 **com.codahale.metrics.MetricRegistry** の使用を許可します。指定しない場合は、Camel はこの CamelContext によって使用される共有インスタンスを作成します。
rateUnit	**TimeUnit.SECONDS**	メトリクスレポーターまたは統計を json 出力するときのレートに使用する単位。
durationUnit	**TimeUnit.MILLISECONDS**	メトリクスレポーターまたは統計を json 出力するときの期間に使用する単位。
namePattern	**##name##.##routeId##.##type##**	Camel 2.17: 使用する名前パターン。区切り文字としてドットを使用しますが、変更できます。**##name##**、**##routeId##**、および **##type##** の値は実際の値に置き換えられます。**###name###** は CamelContext の名前で、**###routeId###** はルートの名前、**###type###** は応答の値になります。

*Java コード tou からは、以下のように **org.apache.camel.component.metrics.routepolicy.MetricsRegistryService** から **com.codahale.metrics.MetricRegistry** を保持できます。*

```
MetricRegistryService registryService = context.hasService(MetricsRegistryService.class);
if (registryService != null) {
  MetricsRegistry registry = registryService.getMetricsRegistry();
  ...
}
```

METRICSMESSAGEHISTORYFACTORY

このファクトリーを使用すると、メッセージのルーティング中にメトリクスを使用してメッセージ履歴のパフォーマンス統計をキャプチャーできます。これは、すべてのルートの各ノードにメトリクスタイマーを使用して機能します。このファクトリーは、以下の例のように Java および XML で使用できます。

Java の場合は、以下のようにファクトリーを CamelContext に設定します。

```
context.setMessageHistoryFactory(new MetricsMessageHistoryFactory());
```

XML DSL の場合は、以下のように <bean> を定義します。

```
<!-- use camel-metrics message history to gather metrics for all messages being routed -->
<bean id="metricsMessageHistoryFactory"
class="org.apache.camel.component.metrics.messagehistory.MetricsMessageHistoryFactory"/>
```

以下のオプションはファクトリーでサポートされます。

名前	デフォルト	説明
useJmx	**false**	**com.codahale.metrics.JmxReporter** を使用して、詳細な統計を JMX に報告するかどうか。CamelContext で JMX が有効になっている場合、**MetricsRegistryService** MBean は JMX ツリーのサービスタイプの下に登録されることに注意してください。この MBean には、JSon を使用して統計を出力する 1 つの操作があります。**useJmx** を **true** に設定する必要があるのは、統計タイプごとに詳細な MBean が必要な場合のみです。
jmxDomain	**org.apache.camel.metrics**	JMX ドメイン名。
prettyPrint	**false**	JSon 形式で統計を出力するときに pretty print を使用するかどうか。
metricsRegistry		共有 **com.codahale.metrics.MetricRegistry** を有効にします。指定がない場合は、Camel はこの CamelContext の共有インスタンスを作成します。
rateUnit	**TimeUnit.SECONDS**	メトリクスレポーターのレートに使用する単位、または統計を JSon としてダンプする場合。
durationUnit	**TimeUnit.MILLISECONDS**	メトリクスレポーターまたは統計を JSon としてダンプするときの期間に使用する単位。
namePattern	**##name##.##routeId##.###id###.##type##**	使用する名前パターン。区切り文字としてドットを使用しますが、変更できます。**##name##**、**##routeId##**、**##type##**、および **###id###** の値は、実際の値に置き換えられます。**###name###** は CamelContext の名前で、**###routeId###** パターンはルートの名前、**###id###** パターンはノード ID を表し、**###type###** は履歴の値になります。

実行時に、Java API または JMX からメトリクスにアクセスできます。これにより、データを JSon 出力として収集できます。Java コードから、以下のように CamelContext からサービスを取得できます。

```
MetricsMessageHistoryService service =
context.hasService(MetricsMessageHistoryService.class);
String json = service.dumpStatisticsAsJson();
```

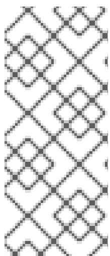
また、MBean は name=MetricsMessageHistoryService を使用して type=services ツリーに登録される JMX API です。

第105章 MINA2 - 非推奨

MINA 2 コンポーネント

非推奨

MINA2 コンポーネントは非推奨になりました。代わりに Netty を使用してください。



注記

コンシューマーエンドポイントでは、sync=false に注意してください。camel-mina2 以降、すべてのコンシューマーエクスチェンジは InOut です。これは camel-mina とは異なります。

Camel 2.10 以降で利用可能

mina2: コンポーネントは Apache MINA 2.xを使用するためのトランスポートです。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mina2</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
mina2:tcp://hostname[:port][?options]
mina2:udp://hostname[:port][?options]
mina2:vm://hostname[:port][?options]
```

codec オプションを使用して、レジストリーに codec を指定できます。TCP を使用し、codec を指定しない場合は、テキスト行 ベースのコーデックまたはオブジェクトシリアル化を代わりに使用する必要があるかどうかを判断するためにテキストラインフラグが使用されます。デフォルトでは、オブジェクトのシリアライズが使用されます。

codec が指定されていない場合、UDP の場合、デフォルトは基本的な ByteBuffer ベースのコーデックを使用します。

仮想マシンプロトコルは、同じ JVM の直接転送メカニズムとして使用されます。

Mina プロデューサーのデフォルトのタイムアウト値は 30 秒で、リモートサーバーからの応答を待ちます。

通常の使用では、camel-mina はボディー content-essage ヘッダーとエクスチェンジプロパティーは送信されません。ただし、オプション transferExchange を使用すると、エクスチェンジ自体をネットワーク経由で転送できます。以下のオプションを参照してください。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	デフォルト値	説明

codec	**null**	Spring **ApplicationContext** などの名前付きの **ProtocolCodecFactory** インスタンスをレジストリーで参照できます。これはマーシャリングに使用されます。
disconnect	**false**	使用直後に Mina セッションから切断するかどうか。コンシューマーとプロデューサーの両方に使用できます。
textline	**false**	TCP にのみ使用されます。codec が指定されていない場合、このフラグを使用してテキスト行ベースのコーデックを示すことができます。指定されていない場合、または値が **false** の場合、Object Serialization は TCP 経由で想定されます。
textlineDelimiter	**DEFAULT**	TCP および **textline=true** にのみ使用されます。使用するテキスト行区切り文字を設定します。使用できる値は、**DEFAULT**、**AUTO**、**WINDOWS**、**UNIX** または **MAC** です。指定がない場合は、Camel は **DEFAULT** を使用します。この区切り文字は、テキストの最後をマークするために使用されます。
sync	**true**	endpoint を one-way または request-response として設定するための設定。
lazySessionCreation	**true**	Camel プロデューサーの開始時にリモートサーバーが稼働していない場合、セッションは遅延して例外を回避するために作成できます。
timeout	**30000**	リモートサーバーからの応答を待つ時間を指定するタイムアウトを設定できます。タイムアウト単位はミリ秒単位であるため、60000 は 60 秒です。タイムアウトは Mina プロデューサーにのみ使用されます。

encoding	JVM のデフォルト	TCP テキストラインコーデックおよび UDP プロトコルに使用するようにエンコーディング(charset 名)を設定できます。指定しない場合、Camel は JVM のデフォルト Charset を 使用します。
transferExchange	**false**	TCP にのみ使用されます。ボディーだけでなく、ネットワーク経由でエクスチェンジを転送することができます。以下のフィールドは転送されます：ボディー、Out body、Fault ボディー、In headers、Out ヘッダー、Fault ヘッダー、エクスチェンジプロパティー、エクスチェンジ例外。これには、オブジェクトが **シリアライズ可能** である必要があります。Camel はシリアライズできないオブジェクトを除外し、**WARN** レベルでログに記録します。
minaLogger	**false**	Apache MINA ロギングフィルターを有効にできます。Apache MINA は、**INFO** レベルで **slf4j** ロギングを使用して、すべての入出力をログに記録します。
filters	**null**	Mina IoFilters のリストを設定して登録できます。**フィルター** は Bean 参照のコンマ区切りリストとして指定できます（例：**\#filterBean1,#filterBean2)** **。各 Bean**のタイプは **org.apache.mina.common.IoFilter** である必要があります。
encoderMaxLineLength	**\-1**	テキストプロトコルエンコーダーの最大行の長さを設定します。デフォルトでは、Mina 自体のデフォルト値が使用されます。これは **Integer.MAX_VALUE** です。
decoderMaxLineLength	**\-1**	テキストラインプロトコルデコーダーの最大行の長さを設定します。デフォルトでは、Mina itself のデフォルト値が使用されます。これは 1024 です。

maximumPoolSize	16	TCP プロデューサーはスレッドセーフであり、同時実行性をはるかに向上します。このオプションを使用すると、同時プロデューサーのスレッドプールにスレッド数を設定できます。**注記：** Camel にはプールされたサービスがあり、すでにスレッドセーフであり、サポートされる同時実行性が保証されています。
allowDefaultCodec	**true**	mina コンポーネントは、両方が null の場合、codec が **null** で、**テキストライン** が **false** の場合にデフォルトの **コーデック** をインストールします。**allowDefaultCodec** を **false** に設定すると、mina コンポーネントがフィルターチェーンの最初の要素としてデフォルトのコーデックをインストールできなくなります。これは、SSL フィルターなど、フィルターチェーンの最初のフィルターを設定する必要があるシナリオで役立ちます。
disconnectOnNoReply	**true**	sync が有効になっている場合、返答がない場所で切断される必要がある場合は、このオプションは MinaConsumer を指示します。
noReplyLogLevel	**WARN**	sync が有効になっている場合、このオプションは MinaConsumer を決定し、送信する応答がない場合に使用するログレベルを指定します。値は **FATAL、ERROR、INFO、DEBUG、OFF** です。
orderedThreadPoolExecutor	**true**	順序付けられたスレッドプールを使用して、イベントが同じチャネルで順番に処理されるかどうか。
sslContextParameters	**null**	**org.apache.camel.util.jsse.SSLContextParameters** インスタンスを使用した SSL 設定。Using the JSSE Configuration Utility を参照してください。
autoStartTls	**true**	SSL ハンドシェイクを自動起動するかどうか。

cachedAddress	**true**	**Camel 2.14:** InetAddress を一度作成して再利用するかどうか。これを false に設定すると、ネットワークで DNS の変更を取得できます。
clientMode	**false**	**Camel 2.15:** コンシューマーのみ。clientMode が true の場合、mina コンシューマーはアドレスを TCP クライアントとして接続します。

カスタムコーデックの使用

Mina how to write your own codec を参照してください。camel-mina でカスタムコーデックを使用するには、Spring XML ファイルに Bean を作成するなどして、codec をレジストリーに登録する必要があります。次に、codec オプションを使用してコーデックの Bean ID を指定します。カスタムコーデックを持つ HL7 を参照してください。

SYNC=FALSE を使用した例

この例では、Camel はポート 6200 で TCP 接続をリッスンするサービスを公開します。テキストライン コーデックを使用します。ルートでは、ポート 6200 をリッスンする Mina コンシューマーエンドポイントを作成します。

```
from("mina2:tcp://localhost:" + port1 + "?textline=true&sync=false").to("mock:result");
```

サンプルはユニットテストの一部であるため、ポート 6200 で一部のデータを送信することでテストします。

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedBodiesReceived("Hello World");

template.sendBody("mina2:tcp://localhost:" + port1 + "?textline=true&sync=false", "Hello
World");

assertMockEndpointsSatisfied();
```

SYNC=TRUE の例

次のサンプルでは、ポート 6201 で TCP サービスを公開する一般的なユースケースも、テキストコーデックを使用します。ただし、今回は応答を返すため、コンシューマーで sync オプションを true に設定します。

```
from("mina2:tcp://localhost:" + port2 + "?textline=true&sync=true").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        exchange.getOut().setBody("Bye " + body);
    }
});
```

次に、template.requestBody （）メソッドを使用してデータを送信し、応答を取得し、サンプルをテストします。応答が String であることが分かっているため、 String にキャストし、応答がプロセッサーコードロジックで動的に設定されているものであることをアサートできます。

```
String response = (String)template.requestBody("mina2:tcp://localhost:" + port2 + "?textline=true&sync=true", "World");
assertEquals("Bye World", response);
```

SPRING DSL を使用した例

当然ながら、Spring DSL は MINA2 にも使用することができます。以下の例では、ポート 5555 で TCP サーバーを公開します。

```
 <route>
   <from uri="mina2:tcp://localhost:5555?textline=true"/>
   <to uri="bean:myTCPOrderHandler"/>
</route>
```

上記のルートでは、テキストのコーデックを使用して、ポート 5555 で TCP サーバーを公開します。ID myTCPOrderHandler を持つ Spring Bean がリクエストを処理し、応答を返します。たとえば、ハンドラー Bean は以下のように実装できます。

```
    public String handleOrder(String payload) {
        ...
```

```
        return "Order: OK"
    }
```

完了時にセッションを閉じる

サーバーとして動作する場合、クライアントの変換が終了する場合などにセッションを閉じることがあります。Camel に対してセッションを閉じるよう指示するには、CamelMinaCloseSessionWhenComplete キーを持つヘッダーをブール値 true に追加する必要があります。

たとえば、以下の例では、bye メッセージをクライアントに書き戻した後にセッションを閉じます。

```
        from("mina2:tcp://localhost:8080?sync=true&textline=true").process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                String body = exchange.getIn().getBody(String.class);
                exchange.getOut().setBody("Bye " + body);

exchange.getOut().setHeader(Mina2Constants.MINA_CLOSE_SESSION_WHEN_COMPLETE,
true);
            }
        });
```

メッセージの IOSESSION を取得します。

このキー Mina2Constants.MINA_IOSESSION を使用してメッセージヘッダーから IoSession を取得し、キー Mina2Constants.MINA_LOCAL_ADDRESS キーでローカルホストアドレスとリモートホストアドレスを Mina2Constants.MINA_REMOTE_ADDRESS キーで取得することもできます。

MINA フィルターの設定

フィルターを使用すると、SslFilter などの一部の Mina フィルターを使用できます。カスタマイズされたフィルターを実装することもできます。codec および logger は、タイプ IoFilter の Mina フィルターとしても実装されることに注意してください。定義できるフィルターは、フィルターチェーンの最後に追加されます。つまり、codec および logger の後です。

- 以下も参照してください。

  Netty

第106章 MLLP

MLLP コンポーネント

Camel 2.17 以降で利用可能

MLLP コンポーネントは、MLLP プロトコルの Nuances を処理し、MLLP プロトコルを使用して他のシステムと通信するために Healthcare プロバイダーが必要とする機能を提供します。MLLP コンポーネントは、簡単な設定 URI、自動 HL7 承認生成、および自動確認応答を提供します。

MLLP プロトコルは通常、多数の同時 TCP 接続を使用しません。単一のアクティブな TCP 接続は通常のケースです。したがって、MLLP コンポーネントは、標準の Java ソケットをベースとする単純なスレッドごとのモデルを使用します。これにより、実装がシンプルになり、Camel 自体以外の依存関係がなくなります。

コンポーネントは以下をサポートします。

- TCP サーバーを使用した Camel コンシューマー
- TCP クライアントを使用した Camel プロデューサー

MLLP コンポーネントは byte[] ペイロードを使用し、Camel Type Conversion に依存して byte[] を他のタイプに変換します。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel<groupId>
  <artifactId>camel-mllp<artifactId>
  <version>x.x.x<version>
  < use the same version as your Camel core version -->
<dependency>
```

MLLP コンシューマー

MLLP コンシューマーは、MLLP フレームメッセージの受信と HL7 確認応答の送信をサポートします。MLLP コンシューマーは自動的に HL7 Acknowledgement (HL7 アプリケーション確認応答)

(AA、AE、AR)を自動的に生成するか、CamelMllpAcknowledgement 交換プロパティーを使用して確認応答を指定できます。さらに、生成される確認応答のタイプは、CamelMllpAcknowledgementType エクスチェンジプロパティーを設定することで制御できます。

メッセージヘッダー

MLLP コンシューマーは、以下のヘッダーを Camel メッセージに追加します。

キー	MSH フィールド	例
CamelMllpLocalAddress		
CamelMllpRemoteAddress		
CamelMllpSendingApplication	**MSH-3**	**APPA**
CamelMllpSendingFacility	**MSH-4**	**FACA**
CamelMllpReceivingApplication	**MSH-5**	**CAMEL**
CamelMllpReceivingFacility	**MSH-6**	**FACB**
CamelMllpTimestamp	**MSH-7**	**20150106235900**
CamelMllpSecurity	**MSH-8**	
CamelMllpMessageType	MSH-9	**ADT^A04**
CamelMllpEventType	MSH-9-1	**AD4**
CamelMllpTriggerEvent	MSH-9-2	**A04**
CamelMllpMessageControlId	MSH-10	12345
CamelMllpProcessingId	MSH-11	P
CamelMllpVersionId	MSH-12	2.3.1
CamelMllpCharset	MSH-18	

すべてのヘッダーは String タイプです。ヘッダーの値がない場合、その値は null になります。

エクスチェンジプロパティー

MLLP コンシューマーが生成する確認のタイプは、Camel エクスチェンジでこれらのプロパティーによって制御できます。

キー		例
CamelMllpAcknowledgement		
CamelMllpAcknowledgement Type		AR

すべてのヘッダーは String タイプです。ヘッダーの値がない場合、その値は null になります。

コンシューマー設定：MLLP プロデューサー

MLLP プロデューサーは、MLLP フレームメッセージの送信と HL7 確認応答の受信をサポートします。MLLP Producer は HL7 Acknowledgment を干渉し、負の確認を受け取ると例外を発生させます。受信した確認応答がイントラクションされ、確認応答が負の場合に例外が発生します。

メッセージヘッダー

MLLP Producer は、Camel メッセージに以下のヘッダーを追加します。

キー	MSH フィールド	例
CamelMllpLocalAddress		
CamelMllpRemoteAddress		
CamelMllpAcknowledgement		
CamelMllpAcknowledgement Type		AA

すべてのヘッダーは String タイプです。ヘッダーの値がない場合、その値は null になります。

第107章 MOCK

MOCK コンポーネント

Mock コンポーネントは、強力な宣言型テストメカニズムを提供します。これは、テストの開始前に Mock エンドポイントで宣言的期待を作成できるという点で jMock と似ています。次に、テストが実行されます。これは通常、1 つ以上のエンドポイントに対してメッセージを実行し、最終的にはシステムが期待どおりに機能するようにテストケースで期待をアサートできます。

これにより、以下のようなさまざまなことをテストできます。

- 正しい数のメッセージが各エンドポイントで受信されます。
- 正しいペイロードが適切な順序で受信されます。
- メッセージは、一部の 式 を使用して注文テスト関数を作成し、順番にエンドポイントに到達します。
- メッセージは、特定のヘッダーに特定の値がある場合や、そのメッセージのパーツが XPath または XQuery Expression を評価するなど、一部の述語と一致するなど、何らかの述語に一致します。http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/Predicate.html

Mock エンドポイントである Test エンドポイント もありますが、2 番目のエンドポイントを使用して想定されるメッセージ本文のリストを提供し、Mock エンドポイントアサーションを自動的に設定します。つまり、File または データベース の一部のサンプルメッセージからアサーションを自動的に設定する Mock エンドポイントです。以下に例を示します。

MOCK エンドポイントは受信したエクスチェンジをメモリーに無期限に保持します。

Mock はテスト用に設計されていることに注意してください。Mock エンドポイントをルートに追加すると、エンドポイントに送信された各 エクスチェンジ は、明示的にリセットするか JVM が再起動するまで、メモリーに保存されます（後続の検証を許可します）。高いボリュームや大きなメッセージを送信する場合は、メモリーを過剰に使用する可能性があります。目的が deployable ルートをインラインでテストする場合は、Mock エンドポイントをルートに直接追加するのではなく、テストで NotifyBuilder または AdviceWith を使用することを検討してください。

Camel 2.10 以降では、2 つの新しいオプションが retainFirst と retainLast の 2 つあり、Mock エンドポイントがメモリー内に保持するメッセージの数を制限するために使用できます。

URI 形式

```
mock:someName[?options]
```

someName には、エンドポイントを一意に識別する任意の文字列を指定できます。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	デフォルト	説明
reportGroup	null	レポートに スループットロガー を使用するサイズ
retainFirst		Camel 2.10: メモリーの最初の X 個のメッセージのみを保持します。
retainLast		Camel 2.10: 最後の X 個のメッセージのみをメモリーに保持します。

簡単な例

以下は、使用中の Mock エンドポイントの例です。まず、エンドポイントがコンテキストで解決されます。その後、期待値を設定し、テストの実行後に、期待値が満たされていることをアサートします。

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class);

resultEndpoint.expectedMessageCount(2);

// send some messages
...

// now lets assert that the mock:foo endpoint received 2 messages
resultEndpoint.assertIsSatisfied();
```

通常 は assertIsSatisfied （）メソッドを呼び出して、テスト実行後に期待が満たされていることをテストします。

Apache Camel はデフォルトで assertIsSatisfied （） が呼び出されると 10 秒待機します。これは setResultWaitTime (millis) メソッドを設定することで設定できます。

ASSERTPERIOD の使用

Camel 2.7 で利用可能 アサーションが満たされると、Camel は assertIsSatisfied メソッドの 待機を停止し、続行します。つまり、新しいメッセージがモックエンドポイント（後でのみ）に到達すると、アサーションの結果には影響を与えません。その後の期間後に新しいメッセージが到達しないことをテストする場合は、以下のように setAssertPeriod メソッドを設定してこれを実行できます。

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class);
resultEndpoint.setAssertPeriod(5000);
resultEndpoint.expectedMessageCount(2);

// send some messages
...

// now lets assert that the mock:foo endpoint received 2 messages
resultEndpoint.assertIsSatisfied();
```

期待値の設定

MockEndpoint の javadoc から、期待を設定するために使用できるさまざまなヘルパーメソッドを確認できます。主なメソッドは以下のとおりです。

メソッド	説明
expectedMessageCount(int)	エンドポイントで想定されるメッセージ数を定義します。
expectedMinimumMessageCount(int)	エンドポイントで想定されるメッセージの最小数を定義します。
expectedBodiesReceived(...)	受信すべき想定されるボディーを定義します（順番に）。
expectedHeaderReceived(...)	受信する必要のあるヘッダーを定義する
expectsAscending(Expression)	メッセージの順序を追加するには、指定の 式 を使用してメッセージを比較することで、メッセージが順番に受信されることを想定します。
expectsDescending(Expression)	メッセージの順序を追加するには、指定の 式 を使用してメッセージを比較することで、メッセージが順番に受信されることを想定します。
expectsNoDuplicates(Expression)	重複メッセージが受信されないことを期待するには、式 を使用して各メッセージの一意の識別子を計算します。JMS を使用している場合は **JMSMessageID** やメッセージ内の一意の参照番号などになります。

以下は別の例です。

```
resultEndpoint.expectedBodiesReceived("firstMessageBody", "secondMessageBody", "thirdMessageBody");
```

特定のメッセージへの期待の追加

さらに、message (int messageIndex) メソッドを使用して、受信される特定のメッセージに関するアサーションを追加できます。

たとえば、最初のメッセージのヘッダーまたはボディーに期待値を追加するには(java.util.Listなどのゼロベースのインデックスを使用)、以下のコードを使用できます。

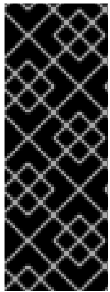
```
resultEndpoint.message(0).header("foo").isEqualTo("bar");
```

camel-core プロセッサーテスト で使用される Mock エンドポイントの例がいくつかあります。

既存エンドポイントのモック化

Camel 2.7 以降で利用可能

Camel では、Camel ルート内の既存のエンドポイントを自動的にモック化できるようになりました。



仕組み

重要： エンドポイントは動作中ですが、Mock エンドポイントがインジェクトされ、最初にメッセージを受信してから、メッセージをターゲットエンドポイントに委譲することです。これは、インターセプトおよびデリゲートまたはエンドポイントリスナーとして表示できます。

指定のルートがある場合には、以下を実行します。

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:start").to("direct:foo").to("log:foo").to("mock:result");

            from("direct:foo").transform(constant("Bye World"));
        }
    };
}
```

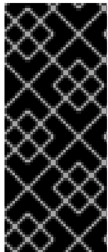
その後、以下のように Camel の adviceWith 機能を使用して、ユニットテストから指定のルートにあるすべてのエンドポイントをモック化できます。

```
public void testAdvisedMockEndpoints() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0).adviceWith(context, new AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            // mock all endpoints
            mockEndpoints();
        }
```

```
    });

    getMockEndpoint("mock:direct:start").expectedBodiesReceived("Hello World");
    getMockEndpoint("mock:direct:foo").expectedBodiesReceived("Hello World");
    getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");
    getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // additional test to ensure correct endpoints in registry
    assertNotNull(context.hasEndpoint("direct:start"));
    assertNotNull(context.hasEndpoint("direct:foo"));
    assertNotNull(context.hasEndpoint("log:foo"));
    assertNotNull(context.hasEndpoint("mock:result"));
    // all the endpoints was mocked
    assertNotNull(context.hasEndpoint("mock:direct:start"));
    assertNotNull(context.hasEndpoint("mock:direct:foo"));
    assertNotNull(context.hasEndpoint("mock:log:foo"));
}
```

モックエンドポイントには uri mock:<endpoint> が指定されることに 注意してください（例：mock:direct:foo ）。INFO レベルの Camel ログで、モックされるエンドポイントは以下のようになります。

```
INFO  Adviced endpoint [direct://foo] with mock endpoint [mock:direct:foo]
```



モックされたエンドポイントはパラメーターがありません。

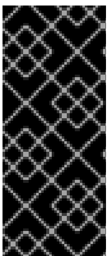
モックされたエンドポイントには、パラメーターが取り除かれます。たとえば、エンドポイント "log:foo?showAll=true" は、次のエンドポイントである "mock:log:foo" にモックされます。パラメーターが削除されたことに注意してください。

パターンを使用して特定のエンドポイントのみをモックすることもできます。たとえば、次のように行うすべての ログ エンドポイントをモック化するには、以下を実行します。

```
public void testAdvisedMockEndpointsWithPattern() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0).adviceWith(context, new AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            // mock only log endpoints
            mockEndpoints("log*");
        }
    });
```

```
    // now we can refer to log:foo as a mock and set our expectations
    getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");

    getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // additional test to ensure correct endpoints in registry
    assertNotNull(context.hasEndpoint("direct:start"));
    assertNotNull(context.hasEndpoint("direct:foo"));
    assertNotNull(context.hasEndpoint("log:foo"));
    assertNotNull(context.hasEndpoint("mock:result"));
    // only the log:foo endpoint was mocked
    assertNotNull(context.hasEndpoint("mock:log:foo"));
    assertNull(context.hasEndpoint("mock:direct:start"));
    assertNull(context.hasEndpoint("mock:direct:foo"));
}
```

サポートされるパターンはワイルドカードまたは正規表現になります。詳細は、Camel で使用される一致関数として Intercept を参照してください。



重要

エンドポイントをモックすると、メッセージがモックに到達するとコピーされることに注意してください。つまり、Camel はより多くのメモリーを使用します。これは、多くのメッセージで を送信する場合、適切ではない可能性があります。

CAMEL-TEST コンポーネントを使用した既存エンドポイントのモック化

adviceWith を使用して Camel にエンドポイントをモックするように指示する代わりに、camel-test Test Kit を使用する際にこの動作を簡単に有効にできます。同じルートを以下のようにテストできます。isMockEndpoints メソッドから "*" を返すことに注意してください。これは、Camel にすべてのエンドポイントをモックするように指示します。すべての ログ エンドポイントのみをモック化したい場合は、代わりに log*" を返すことができます。

```
public class IsMockEndpointsJUnit4Test extends CamelTestSupport {

    @Override
    public String isMockEndpoints() {
        // override this method and return the pattern for which endpoints to mock.
        // use * to indicate all
        return "*";
    }

    @Test
    public void testMockAllEndpoints() throws Exception {
```

```
        // notice we have automatic mocked all endpoints and the name of the endpoints is
"mock:uri"
        getMockEndpoint("mock:direct:start").expectedBodiesReceived("Hello World");
        getMockEndpoint("mock:direct:foo").expectedBodiesReceived("Hello World");
        getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");
        getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

        template.sendBody("direct:start", "Hello World");

        assertMockEndpointsSatisfied();

        // additional test to ensure correct endpoints in registry
        assertNotNull(context.hasEndpoint("direct:start"));
        assertNotNull(context.hasEndpoint("direct:foo"));
        assertNotNull(context.hasEndpoint("log:foo"));
        assertNotNull(context.hasEndpoint("mock:result"));
        // all the endpoints was mocked
        assertNotNull(context.hasEndpoint("mock:direct:start"));
        assertNotNull(context.hasEndpoint("mock:direct:foo"));
        assertNotNull(context.hasEndpoint("mock:log:foo"));
    }

    @Override
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                from("direct:start").to("direct:foo").to("log:foo").to("mock:result");

                from("direct:foo").transform(constant("Bye World"));
            }
        };
    }
}
```

XML DSL を使用した既存エンドポイントのモック化

ユニットテストに camel-test コンポーネントを使用しない場合（上記のように）、ルートに XML ファイルを使用する場合に別のアプローチを使用できます。解決策は、ユニットテストで使用する新しい XML ファイルを作成し、テストするルートを持つ目的の XML ファイルを含めることです。

camel-route.xml ファイルにルートがあるとします。

```
<!-- this camel route is in the camel-route.xml file -->
<camelContext xmlns="http://camel.apache.org/schema/spring">

    <route>
        <from uri="direct:start"/>
        <to uri="direct:foo"/>
        <to uri="log:foo"/>
        <to uri="mock:result"/>
```

```
    </route>

    <route>
        <from uri="direct:foo"/>
        <transform>
            <constant>Bye World</constant>
        </transform>
    </route>

</camelContext>
```

次に、以下のように新しい XML ファイルを作成します。ここで、camel-route.xml ファイルが含まれ、org.apache.camel.impl.InterceptSendToMockEndpointStrategy クラスで Spring Bean を定義します。これは、Camel に対してすべてのエンドポイントをモック化するように指示します。

```
<!-- the Camel route is defined in another XML file -->
<import resource="camel-route.xml"/>

<!-- bean which enables mocking all endpoints -->
<bean id="mockAllEndpoints"
class="org.apache.camel.impl.InterceptSendToMockEndpointStrategy"/>
```

その後、ユニットテストで camel-route.xml の代わりに新しい XML ファイル(test- camel-route.xml)を読み込みます。

すべての ログ エンドポイントのみをモック化するには、Bean のコンストラクターでパターンを定義できます。

```
<bean id="mockAllEndpoints"
class="org.apache.camel.impl.InterceptSendToMockEndpointStrategy">
    <constructor-arg index="0" value="log*"/>
</bean>
```

エンドポイントのモック化および元のエンドポイントへの送信を省略する

Camel 2.10 以降で利用可能

特定のエンドポイントへのモック化や送信を簡単にスキップしたい場合があります。そのため、メッセージはデーストされ、モックエンドポイントのみに送信されます。Camel 2.10 以降で

は、AdviceWith または Test Kit を使用して mockEndpointsAndSkip メソッドを使用できるようになりました。以下の例では、direct:foo と direct:bar の 2 つのエンドポイントへ の送信を省略します。

```
public void testAdvisedMockEndpointsWithSkip() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0).adviceWith(context, new AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            // mock sending to direct:foo and direct:bar and skip send to it
            mockEndpointsAndSkip("direct:foo", "direct:bar");
        }
    });

    getMockEndpoint("mock:result").expectedBodiesReceived("Hello World");
    getMockEndpoint("mock:direct:foo").expectedMessageCount(1);
    getMockEndpoint("mock:direct:bar").expectedMessageCount(1);

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // the message was not send to the direct:foo route and thus not sent to the seda endpoint
    SedaEndpoint seda = context.getEndpoint("seda:foo", SedaEndpoint.class);
    assertEquals(0, seda.getCurrentQueueSize());
}
```

Test Kitを使用した同じ例

```
public class IsMockEndpointsAndSkipJUnit4Test extends CamelTestSupport {

    @Override
    public String isMockEndpointsAndSkip() {
        // override this method and return the pattern for which endpoints to mock,
        // and skip sending to the original endpoint.
        return "direct:foo";
    }

    @Test
    public void testMockEndpointAndSkip() throws Exception {
        // notice we have automatic mocked the direct:foo endpoints and the name of the
endpoints is "mock:uri"
        getMockEndpoint("mock:result").expectedBodiesReceived("Hello World");
        getMockEndpoint("mock:direct:foo").expectedMessageCount(1);

        template.sendBody("direct:start", "Hello World");

        assertMockEndpointsSatisfied();

        // the message was not send to the direct:foo route and thus not sent to the seda
endpoint
        SedaEndpoint seda = context.getEndpoint("seda:foo", SedaEndpoint.class);
```

```
        assertEquals(0, seda.getCurrentQueueSize());
    }

    @Override
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                from("direct:start").to("direct:foo").to("mock:result");

                from("direct:foo").transform(constant("Bye World")).to("seda:foo");
            }
        };
    }
}
```

保持するメッセージ数の制限

Camel 2.10 以降で利用可能

Mock エンドポイントは、デフォルトで受信したすべての エクスチェンジ のコピーを保持します。したがって、大量のメッセージをテストすると、メモリーを消費します。Camel 2.10 以降では、2 つのオプション retainFirst および retainLast を導入しました。これを使用して、最初と最後の エクスチェンジの N 分のみを保持するために使用できます。

以下のコードでは、最初の 5 つと最後の 5 エクスチェンジのコピーを保持するだけです。

```
MockEndpoint mock = getMockEndpoint("mock:data");
mock.setRetainFirst(5);
mock.setRetainLast(5);
mock.expectedMessageCount(2000);

...

mock.assertIsSatisfied();
```

この使用にはいくつかの制限があります。MockEndpoint の getExchanges ( )メソッドおよび getReceivedExchanges （） メソッドは、エクスチェンジの保持コピーのみを 返し ます。上記の例では、リストには 10 エクスチェンジ（最初の 5 つ、および最後の 5）が含まれます。retainFirst オプションおよび retainLast オプションには、使用可能な想定されるメソッドの制限もあります。たとえば、メッセージボディーやヘッダーなどで機能する予定のXXX メソッドは、保持済みメッセージでのみ動作します。上記の例では、10 個の保持済みメッセージのみについてテストできます。

到着時間でのテスト

Camel 2.7 以降で利用可能

Mock エンドポイントは、メッセージの到着時間を Exchange のプロパティーとして保存します。

```
Date time = exchange.getProperty(Exchange.RECEIVED_TIMESTAMP, Date.class);
```

この情報を使用して、メッセージがモックに到達するタイミングを確認することができます。ただし、以前のメッセージと次のメッセージがモックに到達する間隔を把握するための基盤も提供します。これを使用して、Mock エンドポイントで DSL に 到達 して期待を設定できます。

たとえば、最初のメッセージが、次に行う前に 0-2 秒の間に到着するように設定するには、次のコマンドを実行します。

```
mock.message(0).arrives().noLaterThan(2).seconds().beforeNext();
```

また、これを 2 番目のメッセージ(0 インデックスベース)が前のメッセージから 0-2 秒より後に到達しないように定義することもできます。

```
mock.message(1).arrives().noLaterThan(2).seconds().afterPrevious();
```

between を使用して下限を設定することもできます。たとえば、1-4 秒の間にあるとします。

```
mock.message(1).arrives().between(1, 4).seconds().afterPrevious();
```

たとえば、メッセージ間のギャップが最大 1 秒であることを示すために、すべてのメッセージに期待値を設定することもできます。

```
mock.allMessages().arrives().noLaterThan(1).seconds().beforeNext();
```

時間単位

上記の例では、時間単位として 秒 を使用しますが、Camel には ミリ 秒、および 分 もあります。

第108章 MONGODB

CAMEL MONGODB コンポーネント

Camel 2.10 以降で利用可能

Wikipedia: "NoSQL is a moving promoting a loosely defined class of loosely defined class of non-elational databases and ACID guarantee." 過去数年、NoSQL ソリューションが人気が増し、Facebook、LinkedIn、Twitter などの主要な非常に使用されているサイトやサービスは、それらを広範囲に使用することが知られています。これにより、スケーラビリティーと調整性が広範囲に活用されています。

基本的に、NoSQL ソリューションは、SQL をクエリー言語として使用しておらず、通常は ACID と同様のトランザクション動作やリレーショナルデータベースを提供しないという点で、従来の NORMAL (Relational Database Management Systems)とは異なります。代わりに、柔軟なデータ構造とスキーマの概念を中心としています（つまり、固定されたスキーマを持つデータベーステーブルの概念が破棄され、コモディティーハードウェアおよびプラジング処理における非常に高いスケーラビリティーがあります）。

MongoDB は非常に人気の高い NoSQL ソリューションであり、camel-mongodb コンポーネントは Camel と MongoDB を統合するため、MongoDB コレクションをプロデューサー（コレクションでの操作）およびコンシューマー(MongoDB コレクションからのドキュメント)として対話できます。

MongoDB はドキュメントの概念に関するものです（オフィスのドキュメントではなく、JSON/BSON で定義された階層データ）。このコンポーネントページは、それらに精通していることを前提としています。それ以外の場合は、http://www.mongodb.org/ にアクセスします。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-mongodb</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

> ***mongodb:connectionBean?database=databaseName&collection=collectionName&operation=operationName[&moreOptions...]***

エンドポイントオプション

MongoDB エンドポイントは、以下のオプションをサポートします。これらのオプションは、Producer または Consumer のどちらとして動作するかによって異なります（オプションはコンシューマータイプによって変わります）。

名前	デフォルト値	説明	プロデューサー	Tailable Cursor コンシューマー
database	none	**必須。** このエンドポイントがバインドされるデータベースの名前。動的性が有効で、 **CamelMongoDbDatabase** ヘッダーが設定されていない限り、すべての操作はこのデータベースに対して実行されます。	(/)	(/)
コレクション	none	**必須(getDbStats およびコマンド操作の検証)。** このエンドポイントがバインドされるコレクションの名前（指定されたデータベース内）。動的性が有効で、 **CamelMongoDbDatabase** ヘッダーが設定されていない限り、ll 操作はこのデータベースに対して実行されます。	(/)	(/)

collectionIndex	none	Camel 2.12: 新しいコレクションを挿入する際に作成するオプションの 単一フィールド インデックス または複合インデックス。	(/)	
operation	none	**プロデューサーに必要です。** このエンドポイントが実行する操作の ID。以下を選択します。 ● クエリー操作：**findById**、**findOneByQuery**、**findAll**、**count** ● 書き込み操作：、**を保存**、**更新** ● 削除操作：**remove** ● 他の操作：**getDbStats**、**getColStats**、**command**	(/)	
createCollection	true	エンドポイントの初期中にコレクションが存在しない場合に、そのコレクションを MongoDB データベースに自動作成するかどうかを決定します。このオプションが **false** でコレクションが存在しない場合は、初期例外が出力されます。	(/)	

writeConcern	none （ドライバーのデフォルト）	MongoDB のパラメーター化された値から操作に **WriteConcern** を設定します。WriteConcern.valueOf (String) を参照してください。	(/)	
writeConcernRef	none	レジストリーに存在するカスタム **WriteConcern** を設定します。Bean 名を指定します。	(/)	
readPreference	none	**Camel 2.12.4、2.13.1、および 2.14.0 で利用可能：接続に** **ReadPreference** **を設定し** ます。許可される値は、ReadPreference#valueOf （）パブリック API でサポートされる値です。現在、MongoDB-Java-Driver バージョン 2.12.0 の時点で、サポートされる値は **primary**、**primaryPreferred**、**secondary**、および **secondaryPreferred** です。**nearest**このオプションの詳細については、の ドキュメント を参照してください。	(/)	

dynamicity	false	true に設定すると、エンドポイントは受信メッセージの **CamelMongoDbDatabase** ヘッダーおよび **CamelMongoDbCollection** ヘッダーを検査し、それらのいずれかが存在する場合は、その特定の操作に対してターゲットコレクションやデータベースが上書きされます。この機能が必要ない場合にすべてのエクスチェンジでルックアップがトリガーされないようにするには、デフォルトで false に設定します。	(/)	
writeResultAsHeader	false	**Camel 2.10.3 および 2.11 で利用可能：書き込み操作（保存、更新、挿入など）**で、ボディーを MongoDB によって返される WriteResult オブジェクトに置き換えるのではなく、入力ボディーを変更せず、**CamelMongoWriteResult** ヘッダー(constant **MongoDbConstants.WRITERESULT**)に WriteResult を配置します。	(/)	

outputType	**DBObjectList** の場合： **findAll** **DBObject** その他のすべての操作の場合	Camel 2.16: プロデューサーの出力を **DBObjectList**、**DBObject** または **DBCursor** の選択したタイプに変換します。 **DBObjectList** または **DBCursor**（出力をストリーミングするのに便利な場合があります）は、**findAll** に適用されます。**DBObject** その他のすべての操作に適用されます。	(/)	
persistentTailTracking	false	Tailable Cursor コンシューマーの永続的なテールトラッキングを有効または無効にします。詳細は、以下を参照してください。		(/)
persistentId	none	**永続的なテールトラッキングが有効な場合は必須です。**この永続テールトラッカーの ID。これは、テールトラッキングコレクションでレコードを残りから分離します。		(/)

tailTrackingIncreasingField	none	**永続的なテールトラッキングが有効な場合は必須です。**増加する性質の着信レコードの相関フィールドであり、生成されるたびに tail カーソルを配置するために使用されます。カーソルは type: tailTrackIncreasingField > lastValue のクエリーで作成されます(lastValue は永続的なテールトラッキングから復元される可能性があります)。型は Integer、Date、String などにすることができます。注記：現在の時点でドット表記はサポートされていません。そのため、フィールドはドキュメントの最上位に置かれます。		(/)
cursorRegenerationDelay	1000ms	MongoDB サーバーによって強制終了されてから、エンドポイントがカーソルの再生成を待機する時間を指定します（通常の動作）。		(/)
tailTrackDb	エンドポイントのと同じです。	永続的なテールトラッカーがランタイム情報を保存するデータベース。		(/)
tailTrackCollection	camelTailTracking	永続テールトラッカーがランタイム情報を保存するコレクション。		(/)

tailTrackField	lastTrackingValue	永続的なテールトラッカーが最後に追跡された値を保存するフィールド。		(/)

SPRING XML でのデータベースの設定

以下の Spring XML は、MongoDB インスタンスへのコネクションを定義する Bean を作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="mongoBean" class="com.mongodb.Mongo">
      <constructor-arg name="host" value="${mongodb.host}" />
      <constructor-arg name="port" value="${mongodb.port}" />
   </bean>
</beans>
```

サンプルルート

Spring XML で定義された以下のルートは、コレクションで操作 dbStats を実行します。

```
<route>
  <from uri="direct:start" />
  <!-- using bean 'mongoBean' defined above -->
  <to uri="mongodb:mongoBean?
database=${mongodb.database}&amp;collection=${mongodb.collection}&amp;operation=getDbStats"
/>
  <to uri="direct:result" />
</route>
```

MONGODB 操作 - プロデューサーエンドポイント

クエリー操作

FINDBYID

この操作は、_id フィールドが IN メッセージボディーの内容と一致するコレクションから 1 つの要素のみを取得します。受信オブジェクトは、BSON タイプと同等のものにすることができます。http://bsonspec.org/#/specification および http://www.mongodb.org/display/DOCS/Java+Types を参照してください。

```
from("direct:findById")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=findById")
    .to("mock:resultFindById");
```

SUPPORTS FIELDS FILTER（フィールドフィルターに対応）

この操作は、フィールドフィルターの指定をサポートします。「フィールドフィルターの指定」を参照してください。

FINDONEBYQUERY

この操作を使用して、MongoDB クエリーに一致するコレクションから 1 つの要素のみを取得します。クエリーオブジェクトは IN メッセージボディー から抽出され ます。つまり、DBObject 型であるか、DBObject に変換できる必要があります。JSON 文字列または Hashmap にすることができます。詳細は 、タイプ変換 を参照してください。

クエリーのない例（コレクションの任意のオブジェクトを返します）。

```
from("direct:findOneByQuery")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=findOneByQuery")
    .to("mock:resultFindOneByQuery");
```

クエリーの例（一致する結果 1 つを返す）:

```
from("direct:findOneByQuery")
    .setBody().constant("{ \"name\": \"Raul Kripalani\" }")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=findOneByQuery")
    .to("mock:resultFindOneByQuery");
```

SUPPORTS FIELDS FILTER（フィールドフィルターに対応）

この操作は、フィールドフィルターの指定をサポートします。「フィールドフィルターの指定」を参照してください。

FINDALL

findAll 操作は、クエリーに一致するすべてのドキュメント、またはまったく none を返します。この場合、コレクションに含まれるすべてのドキュメントが返されます。クエリーオブジェクトは IN メッセージボディー から抽出され ます。つまり、DBObject 型であるか、DBObject に変換できる必要があります。JSON 文字列または Hashmap にすることができます。詳細は 、タイプ変換 を参照してください。

クエリーのない例（コレクション内のすべてのオブジェクトを返します）。

```
from("direct:findAll")
   .to("mongodb:myDb?database=flights&collection=tickets&operation=findAll")
   .to("mock:resultFindAll");
```

クエリーの例（一致する結果をすべて返す）：

```
from("direct:findAll")
   .setBody().constant("{ \"name\": \"Raul Kripalani\" }")
   .to("mongodb:myDb?database=flights&collection=tickets&operation=findAll")
   .to("mock:resultFindAll");
```

ページングおよび効率的な取得は、以下のヘッダーでサポートされます。

ヘッダーのキー	クイック定数	説明 (MongoDB API ドキュメントから抜粋)	想定されるタイプ
CamelMongoDbNumToSkip	**MongoDbConstants.NUM_TO_SKIP**	カーソルの先頭にある特定数の要素を破棄します。	int/Integer
CamelMongoDbLimit	**MongoDbConstants.LIMIT**	返された要素の数を制限します。	int/Integer

CamelMongoDbBatchSize	**MongoDbConstants.BATCH_SIZE**	1 つのバッチで返される要素の数を制限します。カーソルは通常、結果オブジェクトのバッチを取得してローカルに保存します。batchSize が正の場合、取得したオブジェクトの各バッチのサイズを表します。これを調整して、パフォーマンスを最適化し、データ転送を制限することができます。batchSize が負の値の場合、最大バッチサイズ制限（通常は 4MB）内に一致する、返されるオブジェクト数に制限し、カーソルは閉じられます。たとえば、batchSize が -10 の場合、サーバーは最大 10 ドキュメントを返し、最大で 4MB に該当するドキュメントを返し、カーソルを閉じます。この機能は、ドキュメントが最大サイズ内に収まる必要がある点で limit （）とは異なるため、カーソルサーバー側を閉じるために要求を送信する必要がなくなります。バッチサイズは、カーソルの反復後にも変更できます。この場合、設定は次のバッチ取得に適用されます。	int/Integer

さらに、CamelMongoDbSortBy ヘッダーでソートを記述する関連する DBObject を配置することで sortBy 条件を設定することもできます。MongoDbConstants.SORT_BY.

findAll 操作は以下の OUT ヘッダーも返し、ページングを使用している場合に結果ページを反復できるようにします。

ヘッダーのキー	クイック定数	説明 (MongoDB API ドキュメントから抜粋)	データのタイプ

CamelMongoDbResultTotalSize	**MongoDbConstants.RESULT_TOTAL_SIZE**	クエリーに一致するオブジェクトの数。これは、制限/スキップを考慮していません。	int/Integer
CamelMongoDbResultPageSize	**MongoDbConstants.RESULT_PAGE_SIZE**	クエリーに一致するオブジェクトの数。これは、制限/スキップを考慮していません。	int/Integer

SUPPORTS FIELDS FILTER（フィールドフィルターに対応）

この操作は、フィールドフィルターの指定をサポートします。「フィールドフィルターの指定」を参照してください。

COUNT

Long を Out メッセージボディーとして返すコレクション内のオブジェクトの合計数を返します。以下の例では、dynamicCollectionName コレクションのレコード数をカウントします。動的性が有効になっている方法、その結果、操作は notableScientists コレクションに対しては実行されず、dynamicCollectionName コレクションに対しては実行されません。

```
// from("direct:count").to("mongodb:myDb?database=tickets&collection=flights&operation=count&dynamicity=true");
Long result = template.requestBodyAndHeader("direct:count", "irrelevantBody",
MongoDbConstants.COLLECTION, "dynamicCollectionName");
assertTrue("Result is not of type Long", result instanceof Long);
```

Camel 2.14 以降では、メッセージボディーに com.mongodb.DBObject オブジェクトをクエリーとして提供でき、操作はこの基準に一致するドキュメントの数を返します。

```
DBObject query = ...
Long count = template.requestBodyAndHeader("direct:count", query,
MongoDbConstants.COLLECTION, "dynamicCollectionName");
```

フィールドフィルターの指定

クエリー操作は、デフォルトでは、（フィールドすべてとともに）一致するオブジェクトを完全に返します。ドキュメントのサイズが大きく、フィールドのサブセットのみを取得する必要がある場合は、CamelMongoDbFields .FIELDS_FILTER ヘッダーで関連する DBObject （または JSON String や Map など）を DBObject に変換するだけで、すべてのクエリー操作でフィールドフィルターを指定できます。

以下は、MongoDB の BasicDBObjectBuilder を使用して DBObject の作成を簡素化する例です。_id および boringField を除くすべてのフィールドを取得します。

```
// route: from("direct:findAll").to("mongodb:myDb?
database=flights&collection=tickets&operation=findAll")
DBObject fieldFilter = BasicDBObjectBuilder.start().add("_id", 0).add("boringField", 0).get();
Object result = template.requestBodyAndHeader("direct:findAll", (Object) null,
MongoDbConstants.FIELDS_FILTER, fieldFilter);
```

作成/更新の操作

INSERT

IN メッセージボディーから取得した MongoDB コレクションに新しいオブジェクトを挿入します。型変換は、DBObject または List に変換しようとします。単一の挿入と複数の挿入の 2 つのモードがサポートされます。複数の挿入の場合、エンドポイントは、- の場合、または - DBObject に変換できる限り、任意のタイプのオブジェクトの List、Array、または Collections を想定します。すべてのオブジェクトは一度に挿入されます。エンドポイントは、入力に応じて呼び出すバックエンド操作(1 つまたは複数の挿入)をインテリジェントに決定します。

以下に例を示します。

```
from("direct:insert")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=insert");
```

操作は WriteResult を返し、WriteConcern または invokeGetLastError オプションの値によっては、getLastError （） はすでに呼び出されます。書き込み操作の最終結果にアクセスする場合は、WriteResult で getLastError （）または getCachedLastError （） を呼び出して CommandResult を取得する必要があります。次に、CommandResult. ok （）、CommandResult. getErrorMessage （）、および/または CommandResult.getException ( )を呼び出すことで結果を確認できます。

新規オブジェクトの _id はコレクションで一意である必要があることに注意してください。値を指定しないと、MongoDB が自動的に生成されます。しかし、それを指定して一意でない場合は、挿入操作は失敗します(Camel が通知する場合は、callGetLastError を有効にするか、書き込み結果を待つ WriteConcern を設定する必要があります)。

これはコンポーネントの制限ではありませんが、スループットを高めるために MongoDB でどのように機能するかです。カスタムの _id を使用している場合は、アプリケーションレベルが一意であること

を確認する必要があります（これはグッドプラクティスでもあります）。

Camel 2.15 以降、挿入されたレコードの OID は、CamelMongoOid キー下のメッセージヘッダー (MongoDbConstants.OID 定数)に保存されます。1 つの挿入の場合は org.bson.types.ObjectId を保存する値は java.util.List<org.bson.types.ObjectId> で、複数のレコードが挿入されている場合は になります。

SAVE

save 操作は upsert (UPdate、inSERT)操作と同等で、レコードが更新され、存在しない場合は、すべて 1 つのアトミック操作に挿入されます。MongoDB は _id フィールドに基づいてマッチングを実行します。

更新の場合、オブジェクトは完全に置き換えられ、MongoDB の $modifiers の 使用が許可されないことに注意してください。そのため、オブジェクトがすでに存在する場合は、2 つのオプションがあります。

1. クエリーを実行して、オブジェクト全体をそのフィールドすべてと共に最初に取得します（効率的ではない場合もあります）、Camel 内で変更し、保存します。

2. $modifiers とともに更新操作を使用してください。この操作は、代わりにサーバー側で更新を実行します。upsert フラグを有効にすることができます。この場合、挿入が必要な場合、MongoDB は $modifiers をフィルタークエリーオブジェクトに適用し、結果を挿入します。

以下に例を示します。

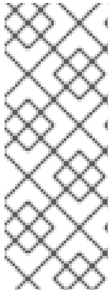
```
from("direct:insert")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=save");
```

UPDATE

コレクションの 1 つまたは複数のレコードを更新します。正確に 2 つの要素が含まれる IN メッセージボディーとして List<DBObject> が必要です。

- 要素 1（インデックス 0）=> フィルタークエリー => は、通常のクエリーオブジェクトと同様に、影響を受けるオブジェクトを決定します。

- *element 2 (index 1)=> update rules => update rules => how matched objects will be updated.MongoDB からの 修飾子操作 はすべてサポートされます。*



MULTIUPDATES

デフォルトでは、複数のオブジェクトがフィルタークエリーと一致する場合でも、MongoDB は 1 オブジェクトのみを更新します。一致する すべて のレコードを更新するよう MongoDB に指示するには、CamelMongoDbMultiUpdate IN メッセージヘッダーを true に設定します。

キーの CamelMongoDbRecordsAffected のあるヘッダーは、更新されたレコードの数(WriteResult.getN　()からコピー）を含むレコードの数とともに (MongoDbConstants.RECORDS_AFFECTED 定数)を返します。

以下の IN メッセージヘッダーをサポートします。

ヘッダーのキー	クイック定数	説明 (MongoDB API ドキュメントから抜粋)	想定されるタイプ
CamelMongoDbMultiUpdate	**MongoDbConstants.MULTIUPDATE**	更新を一致するすべてのオブジェクトに適用する必要がある場合。http://www.mongodb.org/display/DOCS/Atomic+Operations を参照してください。	boolean/Boolean
CamelMongoDbUpsert	**MongoDbConstants.UPSERT**	データベースが存在しない場合には、要素を作成する必要がある	boolean/Boolean

たとえば、以下は "scientist" フィールドの値を "Darwin" に設定することにより、filterField フィールドが true である すべて のレコードを更新します。

```
// route: from("direct:update").to("mongodb:myDb?database=science&collection=notableScientists&operation=update");
DBObject filterField = new BasicDBObject("filterField", true);
DBObject updateObj = new BasicDBObject("$set", new BasicDBObject("scientist", "Darwin"));
Object result = template.requestBodyAndHeader("direct:update", new Object[] {filterField, updateObj}, MongoDbConstants.MULTIUPDATE, true);
```

操作の削除

REMOVE

コレクションから一致するレコードを削除します。IN メッセージ本文は削除フィルタークエリーとして機能し、タイプ DBObject または型として変換可能であることが予想されます。以下の例では、サイエンスデータベース、notableScientists コレクションで、フィールド 'conditionField' が true であるすべてのオブジェクトを削除します。

```
// route: from("direct:remove").to("mongodb:myDb?
database=science&collection=notableScientists&operation=remove");
DBObject conditionField = new BasicDBObject("conditionField", true);
Object result = template.requestBody("direct:remove", conditionField);
```

キーの CamelMongoDbRecordsAffected のあるヘッダーが返され、タイプ int のあるMongoDbConstants.RECORDS_AFFECTED 定数とともに返されます。これには、削除されたレコードの数(WriteResult.getN （)からコピー）が含まれます。

その他の操作

AGGREGATE

Camel 2.14: ボディーに含まれる指定のパイプラインで集約を実行します。集約は長く重い操作になる可能性があります。注意して使用してください。

```
// route: from("direct:aggregate").to("mongodb:myDb?
database=science&collection=notableScientists&operation=aggregate");
from("direct:aggregate")
    .setBody().constant("[{ $match : {$or : [{\"scientist\" : \"Darwin\"},{\"scientist\" :
\"Einstein\"}]}},{ $group: { _id: \"$scientist\", count: { $sum: 1 }} } ]")
    .to("mongodb:myDb?
database=science&collection=notableScientists&operation=aggregate")
    .to("mock:resultAggregate");
```

GETDBSTATS

MongoDB シェルで db.stats （） コマンドを実行するのと同等のもので、データベースに関する有用な統計情報が表示されます。以下に例を示します。

```
> db.stats();
{
 "db" : "test",
 "collections" : 7,
 "objects" : 719,
 "avgObjSize" : 59.73296244784423,
 "dataSize" : 42948,
 "storageSize" : 1000058880,
 "numExtents" : 9,
 "indexes" : 4,
 "indexSize" : 32704,
 "fileSize" : 1275068416,
 "nsSizeMB" : 16,
 "ok" : 1
}
```

使用例：

```
// from("direct:getDbStats").to("mongodb:myDb?
database=flights&collection=tickets&operation=getDbStats");
Object result = template.requestBody("direct:getDbStats", "irrelevantBody");
assertTrue("Result is not of type DBObject", result instanceof DBObject);
```

操作は、OUT メッセージボディーの DBObject の形式で、シェルに表示されるデータ構造と同様のデータ構造を返します。

GETCOLSTATS

MongoDB シェルで db.collection.stats （） コマンドを実行するのと同等のもので、コレクションに関する有用な統計図を表示します。以下に例を示します。

```
> db.camelTest.stats();
{
 "ns" : "test.camelTest",
 "count" : 100,
 "size" : 5792,
 "avgObjSize" : 57.92,
 "storageSize" : 20480,
 "numExtents" : 2,
 "nindexes" : 1,
 "lastExtentSize" : 16384,
 "paddingFactor" : 1,
 "flags" : 1,
 "totalIndexSize" : 8176,
 "indexSizes" : {
  "_id_" : 8176
```

```
 },
 "ok" : 1
}
```

使用例：

```
// from("direct:getColStats").to("mongodb:myDb?
database=flights&collection=tickets&operation=getColStats");
Object result = template.requestBody("direct:getColStats", "irrelevantBody");
assertTrue("Result is not of type DBObject", result instanceof DBObject);
```

操作は、OUT メッセージボディーの DBObject の形式で、シェルに表示されるデータ構造と同様のデータ構造を返します。

COMMAND

Camel 2.15: ボディーをデータベースでコマンドとして実行します。ホスト情報、レプリケーション、またはシャーディングのステータスを取得する際に、管理操作に役立ちます。コレクションパラメーターは、この操作には使用されません。

```
// route: from("command").to("mongodb:myDb?database=science&operation=command");
DBObject commandBody = new BasicDBObject("hostInfo", "1");
Object result = template.requestBody("direct:command", commandBody);
```

動的操作

Exchange は、MongoDbConstants.OPERATION_HEADER 定数によって定義される CamelMongoDbOperation ヘッダーを設定することで、エンドポイントの固定操作を上書きできます。サポートされる値は MongoDbOperation の列挙によって決定され、エンドポイント URI の operation パラメーターの許可される値と一致します。

以下に例を示します。

```
// from("direct:insert").to("mongodb:myDb?
database=flights&collection=tickets&operation=insert");
Object result = template.requestBodyAndHeader("direct:insert", "irrelevantBody",
MongoDbConstants.OPERATION_HEADER, "count");
assertTrue("Result is not of type Long", result instanceof Long);
```

TAILABLE CURSOR コンシューマー

*MongoDB は、*nix システムの tail -f コマンドと同様に、カーソルを開いた状態に維持することで、コレクションから継続中のデータを即座に消費するメカニズムを提供します。このメカニズムは、新しいデータを取得するためにクライアントがスケジュールされた間隔でクライアントに新しいデータをプッシュするので、スケジュールされたポーリングよりも効率的です。また、その他の冗長ネットワークトラフィックも軽減します。*

テール可能なカーソルを使用する必要があるのは 1 つだけです。コレクションはキャプチャーコレクションである必要があります。つまり、このコレクションは N オブジェクトのみを保持し、制限に達すると MongoDB は、最初に挿入された順序で古いオブジェクトをフラッシュします。詳細は、http://www.mongodb.org/display/DOCS/Tailable+Cursors を参照してください。

Camel MongoDB コンポーネントはテール可能なカーソルコンシューマーを実装し、この機能を Camel ルートで使用できるようにします。新しいオブジェクトが挿入されると、MongoDB は DBObjects として、テール可能なカーソルコンシューマーに DBObject としてプッシュします。これにより、エクスチェンジに変換され、ルートロジックがトリガーされます。

テール可能なカーソルコンシューマーの仕組み

カーソルをテール可能なカーソルに移動するには、最初にカーソルを生成する際に、いくつかの特別なフラグが MongoDB に通知されます。作成が完了すると、カーソルは開いたままになり、新しいデータが到着するまで DBCursor.next （） メソッドを呼び出すとブロックされます。ただし、MongoDB サーバーは、新しいデータが不確定な期間の後に表示されない場合に、カーソルを終了する権利を留保します。新しいデータを引き続き使用する場合は、カーソルを再生成する必要があります。これには、停止した位置を覚えておく必要があります。そうでないと、最初からやり直し始めます。

Camel MongoDB テール可能なカーソルコンシューマーは、これらすべてのタスクを処理します。増加する性のあるデータ内の一部のフィールドにキーを指定する必要があります。これは、タイムスタンプ、順次 ID など、再生成されるたびにカーソルを配置するためのマーカーとして機能します。MongoDB でサポートされている任意のデータ型にすることができます。日付、文字列、および整数が適切に機能します。このメカニズムには、このコンポーネントのコンテキストで追跡が必要です。

コンシューマーはこのフィールドの最後の値を記憶し、カーソルを再生成するたびに、increasingField > lastValue のようなフィルターでクエリーを実行し、未読のデータのみが消費されるようにします。

増加フィールドの設定：エンドポイント URI tailTrackingIncreasingField オプションの increasing フィールドのキーを設定 します。Camel 2.10 では、このフィールドのネストされたナビゲーションはまだサポートされていないため、データの最上位フィールドである必要があります。つまり、timestamp フィールドは問題ありませんが、nested.timestamp は機能しません。ネストされた増加フィールドのサポートが必要な場合は、Camel JIRA でチケットを作成してください。

カーソルの再生成遅延： 初期化時に新しいデータがまだ利用できない場合、MongoDB はすぐにカー

ソルを強制終了します。この場合、サーバーに圧倒をかけないため、（デフォルト値 1000ms.）cursorRegenerationDelay オプションが導入されました。これは、ニーズに合わせて変更できます。

以下に例を示します。

```
from("mongodb:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime")
    .id("tailableCursorConsumer1")
    .autoStartup(false)
    .to("mock:test");
```

上記のルートはflights.cancellations の上限付きコレクションから消費されます。これは、増加するフィールドとして departureTime を使用し、デフォルトの再生成カーソル遅延は 1000ms です。

永続的なテールトラッキング

標準のテールトラッキングは揮発性で、最後の値はメモリーにのみ保持されます。ただし、実際には、Camel コンテナーを毎回再起動する必要がありますが、最後の値は失われ、テール可能なカーソルコンシューマーは上部から消費を開始し、重複したレコードをルートに送信する可能性が非常に高くなります。

この状況に対処するために、永続テールトラッキング 機能を有効にして、MongoDB データベース内の特別なコレクションで最後に消費された値を追跡することもできます。コンシューマーが再び初期化されると、最後に追跡された値を復元し、何も起こらなかったかのように続行します。

最後の読み取り値は、2 つの状況で永続化されます。将来的にも一定間隔で永続化することを検討し (5 秒ごとにフラッシュ)、需要がある場合は堅牢性を高めることができます。この機能を要求するには、Camel JIRA でチケットを作成してください。

永続的なテールトラッキングの有効化

この関数を有効にするには、エンドポイント URI に少なくとも以下のオプションを設定します。

- persistentTailTracking オプションを trueに設定

- このコンシューマー の 一意の識別子への PersistentID オプション。これにより、多くのコンシューマーで同じコレクションを再利用できます。

*さらに、**tailTrackDb** オプション、**tailTrackCollection** オプション、および **tailTrackField** オプションを設定して、ランタイム情報が保存される場所をカスタマイズすることができます。各オプションの説明は、このページの上部にあるエンドポイントオプションの表を参照してください。*

*たとえば、以下のルートは**flights.cancellations** の上限付きコレクションから消費され、増加するフィールドとして **departureTime** を使用します。デフォルトの再生成カーソル遅延は **1000ms** で、永続的なテールトラッキングがオンになり、**flights.camelTailTracking** の **id** の下で永続化されます。**lastTrackingValue** フィールドに最後に処理された値を保存する(**camelTailTracking** および **lastTrackingValue** がデフォルトです)。*

```
from("mongodb:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime&persiste
ntTailTracking=true" +
     "&persistentId=cancellationsTracker")
 .id("tailableCursorConsumer2")
 .autoStartup(false)
 .to("mock:test");
```

*以下は上記と同じ例ですが、**"lastProcessedDepartureTime"** フィールドの **"trackers.camelTrackers"** コレクションの下に永続的なテールトラッキングランタイム情報が格納されます。*

```
from("mongodb:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime&persiste
ntTailTracking=true" +

"&persistentId=cancellationsTracker"&tailTrackDb=trackers&tailTrackCollection=camelTracke
rs" +
     "&tailTrackField=lastProcessedDepartureTime")
 .id("tailableCursorConsumer3")
 .autoStartup(false)
 .to("mock:test");
```

型変換

***camel-mongodb** コンポーネントに含まれる **MongoDbBasicConverters** 型コンバーターは、以下の変換を提供します。*

名前	タイププから	以下を入力します。	どのように変更を加えればよいですか？

fromMapToDBObject	**マップ**	**DBObject**	新しい **BasicDBObject(Map m)コンストラクターを使用して新しい BasicDBObject** を構築します。
fromBasicDBObjectToMap	**BasicDBObject**	**マップ**	**BasicDBObject** はすでに **マップ**を実装しています
fromStringToDBObject	**文字列**	**DBObject**	**com.mongodb.util.JSON.parse(String s)** を使用する
fromAnyObjectToDBObject	**オブジェクト**	**DBObject**	Jackson ライブラリー を使用してオブジェクトを **Map** に変換します。これは、新しい **BasicDBObject**の初期化に使用されます。

この型コンバーターは自動検出されるため、手動で設定する必要はありません。

その他の参考資料

- ***MongoDB の Web サイト***
- ***NoSQL Wikipedia の記事***
-

MongoDB Java ドライバー API ドキュメント - 現行バージョン

-

その他の使用例での ユニットテスト

第109章 MONGODB GRIDFS

CAMEL MONGODB GRIDFS コンポーネント

Camel 2.17 以降で利用可能

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-mongodb-gridfs</artifactId>
   <version>2.17.0.redhat-630xxx</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

Camel バージョン 2.19 以降：

```
mongodb-gridfs:connectionBean?
database=databaseName&bucket=bucketName[&moreOptions...]
```

Camel バージョン 2.17 および 2.18。

```
gridfs:connectionBean?database=databaseName&bucket=bucketName[&moreOptions...]
```

エンドポイントオプション

gridfs エンドポイントは、プロデューサーとして動作するか、コンシューマーとして動作するかによって、以下のオプションをサポートします。（コンシューマーの場合、オプションはコンシューマーのタイプによっても異なります。）

名前	デフォルト値	説明	プロデューサー	コンシューマー

database	none	**必須。** このエンドポイントをバインドするデータベースの名前を指定します。すべての操作は、このデータベースに対して実行されます。	Y	Y
bucket	**fs**	指定されたデータベース内の GridFS バケットの名前を指定します。デフォルトは **GridFS.DEFAULT_BUCKET** の値です。	Y	Y
operation	**create**	このエンドポイントが実行する操作の ID を指定します。有効な値は以下のとおりです。 • query operations: **findOne**, **listAll**, **count** • 書き込み操作：**create** • delete operation: **remove**	Y	N
query	none	**queryStrategy** オプションと併用して、新規ファイルの検索に使用されるクエリーを作成します。	N	Y
queryStrategy	**TimeStamp**	新規ファイルの検索に使用されるストラテジーを指定します。有効な値は以下のとおりです。 • **TimeStamp** コンシュー	N	Y

		マーの起動後にアップロードされるファイルを処理します。 ● **PersistentTimestamp** **TimeStamp** と同様に、コレクションに使用された最後のタイムスタンプを保持するため、再起動時にコンシューマーは停止した場所を再開できます。 ● **FileAttribute** **fileAtttributeName** で指定された属性がないファイルを見つけます(fileAttributeName を参照してください)。処理後、**fileAttributeName** で指定された属性がファイルに追加されます。 ● **TimestampAndFileAttribute** **TimeStamp** より新しいファイル			

		を見つけ、**fileAtttributeName** で指定された属性がありません。 ● **PersistentTimestampAndFileAttribute**		
persistentTSCollection	**camel-timestamps**	**PersistentTimestamp** と併用されます。タイムスタンプが保存されるコレクションを指定します。	N	Y
persistentTSObject	**camel-timestamp**	**PersistentTimestamp** と併用されます。タイムスタンプオブジェクトの ID を指定します。 これにより、各コンシューマーは共通のコレクションに独自のタイムスタンプ ID を保存できます。	N	Y
fileAttributeName	**camel-processed**	**FileAttribute** と併用されます。使用する属性の名前を指定します。 ファイルが処理されると、指定した属性は の **処理** に設定されます。ファイル処理が完了すると、指定された属性は **done** に設定されます。	N	Y
delay	**500** (ms)	新規ファイルの GridFS の後続のポーリングの間隔をミリ秒単位で指定します。	N	Y

initialDelay	1000 (ms)	新しいファイルを初めてポーリングするまでの遅延をミリ秒単位で指定します。	N	Y

SPRING XML でのデータベースの設定

以下の Spring XML は、MongoDB インスタンスへのコネクションを定義する Bean を作成します。

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="mongoBean" class="com.mongodb.Mongo">
      <constructor-arg name="host" value="${mongodb.host}" />
      <constructor-arg name="port" value="${mongodb.port}" />
   </bean>
</beans>
```

サンプルルート

Spring XML で定義された以下のルートは、コレクションで操作 findOne を実行します。

```
<route>
  <from uri="direct:start" />
  <!-- using bean 'mongoBean' defined above -->
  <to uri="mongodb-gridfs:mongoBean?database=${mongodb.database}&amp;operation=findOne" />
  <to uri="direct:result" />
</route>
```

MONGODB 操作 - プロデューサーエンドポイント

- *カウント*

  コレクション内のファイルの合計数を返し、OUT メッセージのボディーとして整数を返します。

  ```
  // from("direct:count").to("mongodb-gridfs?database=tickets&operation=count");
  Integer result = template.requestBodyAndHeader("direct:count", "irrelevantBody");
  assertTrue("Result is not of type Long", result instanceof Integer);
  ```

  filename ヘッダーを使用して、指定したファイル名に一致するファイルの数を指定できま

す。

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(Exchange.FILE_NAME, "filename.txt");
Integer count = template.requestBodyAndHeaders("direct:count", query, headers);
```

- listAll

  タブで区切られたストリームに、ファイル名とその ID を一覧表示する Reader を返します。

```
// from("direct:listAll").to("mongodb-gridfs?database=tickets&operation=listAll");
Reader result = template.requestBodyAndHeader("direct:listAll", irrelevantBody");

filename1.txt     1252314321
filename2.txt     2897651254
```

- findOne

  受信ヘッダーから Exchange.FILE_NAME を使用して、GridFS システムで一致するファイルを見つけ、ボディーをコンテンツの InputStream に設定し、メタデータをヘッダーとして提供します。

```
// from("direct:findOne").to("mongodb-gridfs?database=tickets&operation=findOne");
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(Exchange.FILE_NAME, "filename.txt");
InputStream result = template.requestBodyAndHeaders("direct:findOne",
"irrelevantBody", headers);
```

- create

  ファイルコンテンツの受信ヘッダーとボディーコンテンツの受信ヘッダー Exchange.FILE_NAME を使用して、GridFS データベースに新しいファイルを作成します。InputStream

```
// from("direct:create").to("mongodb-gridfs?database=tickets&operation=create");
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(Exchange.FILE_NAME, "filename.txt");
InputStream result = ...the data for the file...
template.requestBodyAndHeaders("direct:create", stream, headers);
```

- remove

  GridFS データベースからファイルを削除します。

```
// from("direct:remove").to("mongodb-gridfs?database=tickets&operation=remove");
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(Exchange.FILE_NAME, "filename.txt");
template.requestBodyAndHeaders("direct:remove", "", headers);
```

GRIDFS CONSUMER

MongoDB GridFS コンポーネントは、新しいファイルを処理するために GridFS を定期的にポーリングします。delay と initialDelay の 2 つのパラメーターがこの動作を制御します。delay は、バックグラウンドスレッドがポーリングの試行間でスリープする時間を指定します（デフォルトは 500ミリ秒）。initialDelay は、最初に GridFS をポーリングするまでコンシューマーの起動を待機する時間を指定します。これは、バックエンドサービスが利用可能になるのにもう少し時間が必要な場合に役立ちます。

グリッド内のどのファイルがまだ処理されていないかを判断するために、コンシューマーがいくつかの戦略を使用できます。

- TimeStamp-[default] 起動時に、コンシューマーは現在の時刻を開始点として使用します。コンシューマーの起動後に追加されたファイルのみが処理されます。コンシューマー起動前で、グリッド内のすべてのファイルは無視されます。ポーリング後、コンシューマーは最後に処理されたファイルのタイムスタンプでタイムスタンプを更新します。

- PersistentTimestamp-on は起動時に、persistentTSObject によって提供されるオブジェクトの persistentTSCollection で指定されたコレクションをクエリーし、これを開始タイムスタンプとして使用します。そのオブジェクトが存在しない場合、コンシューマーは現在の時間を使用してオブジェクトを作成します。ファイルが処理されるたびに、コレクションのタイムスタンプが更新されます。

- FileAttribute- タイムスタンプを使用する代わりに、コンシューマーは fileAttributeName で指定された属性がないファイルについて GridFS にクエリーを実行します。コンシューマーがファイルの処理を開始すると、この属性は GridFS のファイルに追加されます。

  使用例：

```
from("mongodb-gridfs?database=tickets&queryStrategy=FileAttribute").process(...);
```

- *TimestampAndFileAttribute- 2 つのストラテジーを分解すると、コンシューマーは、fileAttributeName で指定された属性がない TimeStamp より新しいファイルを検索します。ファイルの処理中に、不足している属性が GridFS のファイルに追加されます。*

- *PersistentTimestampAndFileAttribute- 2 つのストラテジーを分解すると、コンシューマーは、fileAttributeName で指定された属性がない TimeStamp より新しいファイルを検索します。ファイルの処理中に、不足している属性が GridFS のファイルに追加され、コレクションのタイムスタンプが更新されます。*

使用例：

```
from("mongodb-gridfs?database=myData&queryStrategy=PersistentTimestamp&
persistentTSCollection=CamelTimestamps&persistentTSObject=myDataTS).process(..
.);
```

その他の参考資料

- *MongoDB の Web サイト*
- *NoSQL Wikipedia の記事*
- *MongoDB Java ドライバー API ドキュメント - 現行バージョン*
- *その他の使用例での ユニットテスト*

第110章 MQTT

MQTT コンポーネント

Camel 2.10 以降で利用可能

mqtt: コンポーネントは、Apache ActiveMQ や Mosquittoなどの MQTT 準拠のメッセージブローカーとの通信に使用されます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-mqtt</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
mqtt://name[?options]
```

name は、コンポーネントを割り当てる名前です。

オプション

プロパティー	デフォルト
host	tcp://127.0.0.1:1883
localAddress	
username	
password	
connectAttemptsMax	-1
reconnectAttemptsMax	-1
reconnectDelay	10
reconnectBackOffMultiplier	2.0
reconnectDelayMax	30000
qualityOfService	AtLeastOnce
subscribeTopicName	
subscribeTopicNames	
publishTopicName	camel/mqtt/test
byDefaultRetain	false

mqttTopicPropertyName	_MQTTTopicPropertyName+
mqttRetainPropertyName	MQTTRetain
mqttQosPropertyName	MQTTQos
connectWaitInSeconds	10
disconnectWaitInSeconds	5
sendWaitInSeconds	5
clientId	
cleanSession	true

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

サンプル

メッセージの送信：

```
from("direct:foo")
    .to("mqtt:cheese?publishTopicName=test.mqtt.topic");
```

メッセージの使用：

```
from("mqtt:bar?subscribeTopicName=test.mqtt.topic")
   .transform(body().convertToString())
   .to("mock:result")
```

第111章 MSV

MSV コンポーネント

MSV コンポーネントは、MSV ライブラリー および XML スキーマや RelaxNG XML Syntax などのサポートされる XML スキーマ言語を使用して、メッセージボディー の XML 検証を実行します。

Jing コンポーネントは RelaxNG Compact 構文もサポートすることに注意してください。

URI 形式

```
msv:someLocalOrRemoteResource[?options]
```

someLocalOrRemoteResource は、クラスパス上のローカルリソースへの URL、またはファイルシステムのリモートリソースまたはリソースへの完全な URL です。以下に例を示します。

```
msv:org/foo/bar.rng
msv:file:../foo/bar.rng
msv:http://acme.com/cheese.rng
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	デフォルト	説明
useDom	true	DOMSource/DOMResult または SaxSource/SaxResult をバリデーターによって使用されるかどうか。**注記：** MSV コンポーネントで DOM を使用する必要があります。

例

以下の 例 は、エンドポイント direct:start からのルートを設定する方法を示しています。これは、XML が指定の RelaxNG XML Schema （クラスパスで提供される）と一致するかどうかに基づいて、mock:valid または mock:invalid のいずれかのエンドポイントに送信されます。

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
```

```
    <route>
        <from uri="direct:start"/>
        <doTry>
            <to uri="msv:org/apache/camel/component/validator/msv/schema.rng"/>
            <to uri="mock:valid"/>

            <doCatch>
                <exception>org.apache.camel.ValidationException</exception>
                <to uri="mock:invalid"/>
            </doCatch>
            <doFinally>
                <to uri="mock:finally"/>
            </doFinally>
        </doTry>
    </route>
</camelContext>
```

第112章 MUSTACHE

MUSTACHE

Camel 2.12 以降で利用可能

mustache: コンポーネントは Mustache テンプレートを使用してメッセージを処理できるようにします。これは、Templating を使用してリクエストの応答を生成する場合に理想的です。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-mustache</artifactId>
<version>x.x.x</version> <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
mustache:templateName[?options]
```

templateName は、呼び出すテンプレートのクラスパスローカル URI、またはリモートテンプレートの完全な URL （例：file://folder/myfile.mustache）です。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	デフォルト	説明
allowContextMapAll (producer)	**false**	コンテキストマップが前詳細へのアクセスを許可するかどうかを設定します。デフォルトでは、メッセージの本文とヘッダーにのみアクセスできます。このオプションは、現在の Exchange および CamelContext へのフルアクセスに対して有効にできます。これを行うと、CamelContext API の全機能へのアクセスが開かれるため、潜在的なセキュリティーリスクが発生します。
allowTemplateFromHeader (producer)	**false**	ヘッダーのリソーステンプレートの使用を許可するかどうか (デフォルトは false)。**このオプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼されていないコンテンツやユーザー派生コンテンツが含まれている場合、これは最終的にエンドアプリケーションの信頼性および完全性に影響を与える可能性があるため、このオプションは注意して使用してください。**
encoding	**null**	リソースコンテンツの文字エンコーディング。
startDelimiter	{{	テンプレートコードの開始を示すために使用される文字。
endDelimiter	}}	テンプレートコードの終わりのマークに使用される文字。

MUSTACHE コンテキスト

Camel は Mustache コンテキストでエクスチェンジ情報を提供します（マップのみ）。Exchange は以下のように転送されます。

key	value

exchange	**Exchange** 自体。
exchange.properties	**Exchange** プロパティー。
ヘッダー	In メッセージのヘッダー。
camelContext	Camel コンテキスト。
request	In メッセージ。
ボディー	In メッセージのボディー。
response	Out メッセージ(InOut メッセージ交換パターンにのみ有効)。

動的テンプレート

Camel は 2 つのヘッダーを提供し、テンプレートまたはテンプレートコンテンツ自体に異なるリソースの場所を定義できます。これらのヘッダーのいずれかが設定されている場合、Camel は設定されたエンドポイントでこれを使用します。これにより、ランタイム時に動的テンプレートを指定できます。

ヘッダー	タイプ	説明	サポートバージョン
MustacheConstants.MUSTACHE_RESOURCE_URI	文字列	設定されたエンドポイントの代わりに使用するテンプレートリソースの URI。	
MustacheConstants.MUSTACHE_TEMPLATE	文字列	設定されたエンドポイントの代わりに使用するテンプレート。	

サンプル

たとえば、以下のように使用できます。

```
from("activemq:My.Queue").
to("mustache:com/acme/MyResponse.mustache");
```

Mustache テンプレートを使用して InOut メッセージエクスチェンジ(JMSReplyTo ヘッダーがある)のメッセージの応答を形成。

InOnly を使用してメッセージを消費し、別の宛先に送信する場合は、以下を使用します。

```
from("activemq:My.Queue").
to("mustache:com/acme/MyResponse.mustache").
to("activemq:Another.Queue");
```

以下のように、ヘッダーを介してコンポーネントを動的に使用するテンプレートを指定できます。

```
from("direct:in").
setHeader(MustacheConstants.MUSTACHE_RESOURCE_URI).constant("path/to/my/template.
mustache").
to("mustache:dummy?allowTemplateFromHeader=true");
```

警告

allowTemplateFromHeader オプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼できないコンテンツまたはユーザー派生コンテンツが含まれる場合、これは最終的に、エンドアプリケーションの確実性と整合性に及ぼす可能性があるため、このオプションを使用してください。

電子メールのサンプル

この例では、注文確認メールに Mustache テンプレートを使用します。メールテンプレートは、以下のように Mustache に記載されています。

```
Dear {{headers.lastName}}, {{headers.firstName}}

Thanks for the order of {{headers.item}}.
```

> ***Regards Camel Riders Bookstore***
> ***{{body}}***

第113章 MVEL コンポーネント

MVEL コンポーネント

Camel 2.12 以降で利用可能

mvel: コンポーネントを使用すると、MVEL テンプレートを使用してメッセージを処理できます。これは、Templating を使用してリクエストの応答を生成する場合に理想的です。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-mvel</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
mvel:templateName[?options]
```

templateName は、呼び出すテンプレートのクラスパスローカル URI、またはリモートテンプレートの完全な URL （例：file://folder/myfile.mvel）です。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	デフォルト	説明
allowContextMapAll (producer)	**false**	コンテキストマップが前詳細へのアクセスを許可するかどうかを設定します。デフォルトでは、メッセージの本文とヘッダーにのみアクセスできます。このオプションは、現在の Exchange および CamelContext へのフルアクセスに対して有効にできます。これを行うと、CamelContext API の全機能へのアクセスが開かれるため、潜在的なセキュリティーリスクが発生します。
allowTemplateFromHeader (producer)	**false**	ヘッダーのリソーステンプレートの使用を許可するかどうか (デフォルトは false)。**このオプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼されていないコンテンツやユーザー派生コンテンツが含まれている場合、これは最終的にエンドアプリケーションの信頼性および完全性に影響を与える可能性があるため、このオプションは注意して使用してください。**
contentCache	**true**	ロード時のリソースコンテンツのキャッシュ。キャッシュされたリソースコンテンツは、エンドポイントの **clearContentCache** 操作を使用して JMX 経由でクリアできます。
encoding	**null**	リソースコンテンツの文字エンコーディング。

メッセージヘッダー

***mvel* コンポーネントはメッセージに 2 つのヘッダーを設定します。**

ヘッダー	説明
CamelMvelResourceUri	templateName (**String** オブジェクトとして)

***MVEL* コンテキスト**

Camel は MVEL コンテキストで交換情報を提供します（マップのみ）。Exchange は以下のように転送されます。

key	value
exchange	**Exchange** 自体。
exchange.properties	**Exchange** プロパティー。
ヘッダー	In メッセージのヘッダー。
camelContext	Camel コンテキストの意図。
request	In メッセージ。
in	In メッセージ。
ボディー	In メッセージのボディー。
out	Out メッセージ(InOut メッセージ交換パターンにのみ有効)。
response	Out メッセージ(InOut メッセージ交換パターンにのみ有効)。

ホットリロード

mvel テンプレートリソースは、デフォルトではファイルとクラスパスリソース（展開形式の jar）の両方に対してホットリロードが可能です。contentCache=true を設定すると、Camel はリソースを 1 度だけロードするため、ホットリロードはできません。このシナリオは、リソースが変更されない場合に実稼働環境で使用することができます。

動的テンプレート

Camel は 2 つのヘッダーを提供し、テンプレートまたはテンプレートコンテンツ自体に異なるリソースの場所を定義できます。これらのヘッダーのいずれかが設定されている場合、Camel は設定されたエンドポイントでこれを使用します。これにより、ランタイム時に動的テンプレートを指定できます。

ヘッダー	タイプ	説明
CamelMvelResourceUri	文字列	設定されたエンドポイントの代わりに使用するテンプレートリソースの URI。

CamelMvelTemplate	文字列	設定されたエンドポイントの代わりに使用するテンプレート。

サンプル

たとえば、以下のようなものを使用できます。

```
from("activemq:My.Queue").
  to("mvel:com/acme/MyResponse.mvel");
```

MVEL テンプレートを使用して InOut メッセージエクスチェンジ(JMSReplyTo ヘッダーがある)のメッセージへの応答を形成するには、以下を実行します。

以下のように、ヘッダーを使用してコンポーネントを動的に使用するテンプレートを指定するには、以下を実行します。

```
from("direct:in").
  setHeader("CamelMvelResourceUri").constant("path/to/my/template.mvel").
  to("mvel:dummy");
```

ヘッダーとしてテンプレートを直接指定するには、以下のように、コンポーネントはヘッダーを介して動的に使用する必要があります。

```
from("direct:in").
  setHeader("CamelMvelTemplate").constant("@{\"The result is \" + request.body * 3}\" }").
  to("velocity:dummy?allowTemplateFromHeader=true");
```

警告

***allowTemplateFromHeader** オプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼できないコンテンツまたはユーザー派生コンテンツが含まれる場合、これは最終的に、エンドアプリケーションの確実性と整合性に及ぼす可能性があるため、このオプションを使用してください。*

第114章 MYBATIS

MYBATIS

Camel 2.7 以降で利用可能

mybatis: コンポーネントを使用すると、MyBatis を使用して、リレーショナルデータベースでデータのクエリー、ポーリング、挿入、更新、および削除を行うことができます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-mybatis</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
mybatis:statementName[?options]
```

ここでの statementName は、評価するクエリー、挿入、更新、または削除の操作にマップする MyBatis XML マッピングファイルのステートメント名です。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

このコンポーネントはデフォルトで、想定される名前 SqlMapConfig.xml でクラスパスのルートから MyBatis SqlMapConfig ファイルをロードします。ファイルが別の場所にある場合は、MyBatisComponent コンポーネントで configurationUri オプションを設定する必要があります。

オプション

オプション	型	デフォルト	説明

consumer.onConsume	**文字列**	**null**	消費後に実行するステートメント。たとえば、Camel で消費および処理された後に行を更新するために使用できます。後ほどサンプル を参照してください。複数のステートメントはコンマで区切ることができます。
consumer.useIterator	**boolean**	**true**	**true** の場合、ポーリングが個別に処理されるときに返される各行。**false** の場合、データの **リスト** 全体が IN ボディーとして設定されます。
consumer.routeEmptyResultSet	**boolean**	**false**	空の結果セットをルーティングすべきかどうかを設定します。
statementType	**StatementType**	**null**	呼び出す操作の種類を制御するためにプロデューサーに指定する必要があります。enum 値は **SelectOne**、**SelectList**、**Insert List**、**Update**、**UpdateList**、**Delete**、および **DeleteList** です。**注記：InsertList** は Camel 2.10 で利用可能で、**UpdateList は** Camel 2.11 で利用できます。

maxMessagesPerPoll	**int**	**0**	このオプションは、データベースプールによって返される結果をバッチに分割し、それらを複数のエクスチェンジで配信することを目的としています。この整数は、単一のエクスチェンジで配信する最大メッセージを定義します。デフォルトでは最大値は設定されていません。たとえば制限を 1000 などに設定して、数千のファイルがあるサーバーの起動を回避できます。無効にするには、0 または負の値を設定します。
executorType	**文字列**	**null**	Camel 2.11: ステートメントの実行中に使用されるエグゼキューターのタイプ。サポートされる値は、simple、re reuse、batch です。デフォルトでは、値は指定されず、MyBatis が使用するもの（つまり **simple** ）と同じです。**シンプルな** エクゼキューターは特別なことを行いません。エグゼキューターを **再利用** すると、準備済みステートメントが再利用されます。**バッチ** エグゼキューターは、ステートメントとバッチの更新を再利用します。
outputHeader	**文字列**	**null**	Camel 2.15: メッセージのボディーではなくヘッダーとして結果を保存します。これにより、既存のメッセージボディーをそのまま保存できます。
inputHeader	**文字列**	**null**	Camel 2.15: ボディーの代わりにコンポーネントへの入力としてヘッダー値を使用します。

transacted	ブール値	false	Camel 2.16.2: SQL コンシューマーのみ。トランザクションを有効または無効にします。有効にすると、エクスチェンジの処理に失敗し、コンシューマーは追加のエクスチェンジを処理しなくなり、即時ロールバックが発生します。

メッセージヘッダー

Camel は結果メッセージ(IN または OUT のいずれかのヘッダーに、使用する ステートメントが含まれるヘッダー)を入力します。

ヘッダー	タイプ	説明
CamelMyBatisStatementName	**文字列**	使用される statementName（例：insertAccount）。
CamelMyBatisResult	**オブジェクト**	いずれの操作でも MtBatis から返される **応答**。たとえば、**INSERT** は自動生成キーや行数などを返すことができます。

メッセージボディー

MyBatis からの応答は、SELECT ステートメントの場合のみボディーとして設定されます。たとえば、INSERT ステートメントの場合、Camel はボディーを置き換えません。これにより、ルーティングを継続し、元のボディーを維持することができます。MyBatis からの応答は、常にキー CamelMyBatisResult を持つヘッダーに保存されます。

サンプル

たとえば、JMS キューから Bean を使用し、それらをデータベースに挿入する場合は、以下を実行できます。

```
from("activemq:queue:newAccount").
  to("mybatis:insertAccount?statementType=Insert");
```

呼び出す操作の種類を Camel に指示する必要があるため、statementType を指定する必要があることに注意してください。

insertAccount は、SQL マッピングファイルの MyBatis ID に置き換えます。

```
<!-- Insert example, using the Account parameter class -->
<insert id="insertAccount" parameterType="Account">
  insert into ACCOUNT (
    ACC_ID,
    ACC_FIRST_NAME,
    ACC_LAST_NAME,
    ACC_EMAIL
  )
  values (
    #{id}, #{firstName}, #{lastName}, #{emailAddress}
  )
</insert>
```

STATEMENTTYPE を使用した MYBATIS の制御の改善

MyBatis エンドポイントにルーティングする場合、実行する SQL ステートメントが SELECT、UPDATE、DELETE、または INSERT などであるかどうかを制御するために、より詳細な制御が必要になります。たとえば、IN ボディーに SELECT ステートメントへのパラメーターが含まれる MyBatis エンドポイントにルーティングする場合は、以下を実行できます。

```
from("direct:start")
    .to("mybatis:selectAccountById?statementType=SelectOne")
    .to("mock:result");
```

上記のコードでは、MyBatis ステートメント selectAccountById を呼び出し、IN 本文に 整数 タイプなどの取得するアカウント ID が含まれる必要があります。

SelectList などの他の操作でも同じことができます。

```
from("direct:start")
    .to("mybatis:selectAllAccounts?statementType=SelectList")
    .to("mock:result");
```

また UPDATE と同じです。ここでは、Account オブジェクトを IN ボディーとして MyBatis に送信できます。

```
from("direct:start")
    .to("mybatis:updateAccount?statementType=Update")
    .to("mock:result");
```

INSERTLIST STATEMENTTYPE の使用

Camel 2.10 以降で利用可能

mybatis では、for-each batch ドライバーを使用して複数の行を挿入できます。これを使用するには、マッパー XML ファイルで <foreach> を使用する必要があります。以下に例を示します。

```
<!-- Batch Insert example, using the Account parameter class -->
<insert id="batchInsertAccount" parameterType="java.util.List">
    insert into ACCOUNT (
    ACC_ID,
    ACC_FIRST_NAME,
    ACC_LAST_NAME,
    ACC_EMAIL
    )
    values (
    <foreach item="Account" collection="list" open="" close="" separator="),(">
        #{Account.id}, #{Account.firstName}, #{Account.lastName}, #{Account.emailAddress}
    </foreach>
    )
</insert>
```

次に、以下のように InsertList ステートメントタイプを使用する mybatis エンドポイントに Camel メッセージを送信することで、複数の行を挿入できます。

```
from("direct:start")
    .to("mybatis:batchInsertAccount?statementType=InsertList")
    .to("mock:result");
```

UPDATELIST STATEMENTTYPE の使用

Camel 2.11 から利用可能

mybatis では、for-each batch ドライバーを使用して複数の行を更新できます。これを使用するには、マッパー XML ファイルで <foreach> を使用する必要があります。以下に例を示します。

```
<update id="batchUpdateAccount" parameterType="java.util.Map">
    update ACCOUNT set
    ACC_EMAIL = #{emailAddress}
    where
```

```
    ACC_ID in
    <foreach item="Account" collection="list" open="(" close=")" separator=",">
      #{Account.id}
    </foreach>
</update>
```

次に、以下のように UpdateList ステートメントタイプを使用する mybatis エンドポイントに Camel メッセージを送信することで、複数の行を更新できます。

```
from("direct:start")
    .to("mybatis:batchUpdateAccount?statementType=UpdateList")
    .to("mock:result");
```

DELETELIST STATEMENTTYPE の使用

Camel 2.11 から利用可能

mybatis では、for-each batch ドライバーを使用して複数の行を削除できます。これを使用するには、マッパー XML ファイルで <foreach> を使用する必要があります。以下に例を示します。

```
<delete id="batchDeleteAccountById" parameterType="java.util.List">
    delete from ACCOUNT
    where
    ACC_ID in
    <foreach item="AccountID" collection="list" open="(" close=")" separator=",">
      #{AccountID}
    </foreach>
</delete>
```

次に、以下のように DeleteList ステートメントタイプを使用する mybatis エンドポイントに Camel メッセージを送信することにより、複数の行を削除できます。

```
from("direct:start")
    .to("mybatis:batchDeleteAccount?statementType=DeleteList")
    .to("mock:result");
```

INSERTLIST、UPDATELIST、および DELETELIST STATEMENTTYPES に関する通知

すべてのタイプのパラメーター(List、Map など)を mybatis に渡すことができ、エンドユーザーは、mybatis 動的クエリー 機能を利用して必要に応じて処理します。

スケジュールされたポーリングの例

このコンポーネントはスケジュールされたポーリングをサポートするため、Polling Consumer として使用できます。たとえば、データベースを 1 分ごとにポーリングするには、以下を実行します。

```
from("mybatis:selectAllAccounts?delay=60000").to("activemq:queue:allAccounts");
```

その他のオプションは、ポーリング コンシューマーの ScheduledPollConsumer Options を参照してください。

または、Timer や Quartz コンポーネントなどのスケジュールされたポーリングをトリガーする別のメカニズムを使用することもできます。

以下の例では、Timer コンポーネントを使用して 30 秒ごとにデータベースをポーリングし、データを JMS キューに送信します。

```
from("timer://pollTheDatabase?
delay=30000").to("mybatis:selectAllAccounts").to("activemq:queue:allAccounts");
```

MyBatis SQL マッピングファイルが使用される：

```
<!-- Select with no parameters using the result map for Account class. -->
<select id="selectAllAccounts" resultMap="AccountResult">
  select * from ACCOUNT
</select>
```

ONCONSUME の使用

このコンポーネントは、データが Camel によって消費および処理された 後 のステートメントの実行をサポートします。これにより、データベースで更新後の更新を行うことができます。すべてのステートメントは UPDATE ステートメントである必要があることに注意してください。Camel は、名前をコンマで区切る必要がある複数のステートメントの実行をサポートします。

以下のルートは、consumeAccount ステートメントデータが処理されることを示しています。これにより、データベースの行のステータスを processed に変更できるため、2 回以上消費しないようにします。

```
from("mybatis:selectUnprocessedAccounts?
consumer.onConsume=consumeAccount").to("mock:results");
```

および sqlmap ファイルのステートメントは、以下のようになります。

```
<select id="selectUnprocessedAccounts" resultMap="AccountResult">
  select * from ACCOUNT where PROCESSED = false
</select>
```

```
<update id="consumeAccount" parameterType="Account">
  update ACCOUNT set PROCESSED = true where ACC_ID = #{id}
</update>
```

トランザクションへの参加

camel-mybatis でトランザクションマネージャーを設定するには、標準の MyBatis SqlMapConfig.xml ファイル外でデータベース設定を外部化する必要があるため、少し時間がかかる場合があります。

最初の部分では、DataSource を設定する必要があります。通常、これは Spring プロキシーでラップする必要があるプール(DBCP または c3p0)です。このプロキシーは、Spring 以外の DataSource を使用して Spring トランザクションに参加できるようにします(MyBatis SqlSessionFactory はこれのみを行います)。

```
  <bean id="dataSource"
class="org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy">
    <constructor-arg>
      <bean class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="org.postgresql.Driver"/>
        <property name="jdbcUrl" value="jdbc:postgresql://localhost:5432/myDatabase"/>
        <property name="user" value="myUser"/>
        <property name="password" value="myPassword"/>
      </bean>
    </constructor-arg>
  </bean>
```

これには、プロパティープレースホルダーを使用してデータベース設定を外部化できるようにする追加の利点があります。

その後、トランザクションマネージャーは、外部 DataSource を管理するように設定されます。

```
  <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
```

```
    <property name="dataSource" ref="dataSource"/>
  </bean>
```

mybatis-spring SqlSessionFactoryBean は、同じ DataSource をラップします。

```
  <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <!-- standard mybatis config file -->
<property name="configLocation" value="/META-INF/SqlMapConfig.xml"/>
    <!-- externalised mappers -->
<property name="mapperLocations" value="classpath*:META-INF/mappers/**/*.xml"/>
  </bean>
```

その後、camel-mybatis コンポーネントはそのファクトリーで設定されます。

```
  <bean id="mybatis" class="org.apache.camel.component.mybatis.MyBatisComponent">
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
  </bean>
```

最後に、トランザクション ポリシーはトランザクションマネージャーの上部に定義され、通常どおり使用できます。

```
  <bean id="PROPAGATION_REQUIRED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager"/>
    <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED"/>
  </bean>

  <camelContext id="my-model-context" xmlns="http://camel.apache.org/schema/spring">
    <route id="insertModel">
      <from uri="direct:insert"/>
      <transacted ref="PROPAGATION_REQUIRED"/>
      <to uri="mybatis:myModel.insert?statementType=Insert"/>
    </route>
  </camelContext>
```

第115章 NAGIOS

NAGIOS

Apache Camel 2.3 で利用可能

Nagios コンポーネントを使用すると、パッシブチェックを Nagios に送信できます。

URI 形式

```
nagios://host[:port][?Options]
```

Apache Camel は、Nagios コンポーネントとともに 2 つの機能を提供します。メッセージをエンドポイントに送信することで、パッシブチェックメッセージを送信できます。Apache Camel は、通知を Nagios に送信できる EventNotifer も提供します。

オプション

名前	デフォルト値	説明
host	none	これは、チェックを送信する必要のある Nagios ホストのアドレスです。
port		ホストのポート番号。
password		チェックを Nagios に送信する際に認証されるパスワード。
connectionTimeout	5000	接続タイムアウト（ミリ秒単位）。
timeout	5000	タイムアウトをミリ秒単位で送信。

nagiosSettings		設定済みの **com.googlecode.jsendnsca.core.NagiosSettings** オブジェクトを使用するには、以下を実行します。これを使用する場合、他のオプションは使用されません。
sendSync	**true**	パッシブチェックの送信時に同期を使用するかどうか。**false** に設定すると、Apache Camel はメッセージのルーティングを継続し、パッシブチェックメッセージは非同期で送信されます。
encryptionMethod	**いいえ**	*Camel 2.9:* 暗号化方法を指定します。使用できる値は、**No**、**Xor**、または **TripleDes** です。

HEADERS

名前	説明
CamelNagiosHostName	これは、チェックを送信する必要のある Nagios ホストのアドレスです。このヘッダーは、エンドポイントに設定された既存のホスト名を上書きします。
CamelNagiosLevel	これは重大度レベルです。値 **CRITICAL**、**WARNING**、**OK** を使用できます。Apache Camel はデフォルトで **OK** を使用します。
CamelNagiosServiceName	サーボイの名前。デフォルトでは CamelContext 名を使用します。

メッセージの送信例

メッセージペイロードにメッセージが含まれる Nagios にメッセージを送信することができます。デフォルトでは、OK レベルになり、CamelContext 名をサービス名として使用します。これらの値は、上記のように ヘッダーを使用して詳細化できます。

たとえば、以下のように Hello Nagios メッセージを Nagios に送信します。

```
template.sendBody("direct:start", "Hello Nagios");

from("direct:start").to("nagios:127.0.0.1:5667?password=secret").to("mock:result");
```

CRITICAL メッセージを送信するには、以下のようなヘッダーを送信できます。

```
Map headers = new HashMap();
headers.put(NagiosConstants.LEVEL, "CRITICAL");
headers.put(NagiosConstants.HOST_NAME, "myHost");
headers.put(NagiosConstants.SERVICE_NAME, "myService");
template.sendBodyAndHeaders("direct:start", "Hello Nagios", headers);
```

NAGIOSEVENTNOTIFER の使用

Nagios コンポーネントは、イベントを Nagios に送信するために使用できる EventNotifer も提供します。たとえば、以下のように Java からこれを有効にすることができます。

```
NagiosEventNotifier notifier = new NagiosEventNotifier();
notifier.getConfiguration().setHost("localhost");
notifier.getConfiguration().setPort(5667);
notifier.getConfiguration().setPassword("password");

CamelContext context = ...
context.getManagementStrategy().addEventNotifier(notifier);
return context;
```

Spring XML では、EventNotifier タイプで Spring Bean を定義するだけです。Apache Camel は、Spring を使用した CamelContext の高度な設定 を参照してください。

第116章 NAT

NATS コンポーネント

NATS は、高速で信頼性の高いメッセージングプラットフォームです。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-nats</artifactId>
    <!-- use the same version as your Camel core version -->
    <version>x.y.z</version>
</dependency>
```

URI 形式

```
nats:servers[?options]
```

ここで、サーバー は NATS サーバーの一覧を表します。

オプション

名前	デフォルト値	説明
servers	null	コンポーネントが接続する必要のあるサーバーを定義します。
topic	null	サブスクライブ/パブリッシュするトピック。
reconnect	true	再接続機能を使用するかどうか。
pedantic	false	pedantic モードで実行するかどうか（これはパフォーマンスに影響します）。
verbose	false	詳細モードで実行するかどうか
ssl	false	SSL を使用するかどうか

reconnectTimeWait	**2000**	再接続を試みるまでの待機時間（ミリ秒単位）
maxReconnectAttempts	**3**	接続が失われた場合の再接続試行の最大数を設定します。
pingInterval	**4000**	接続がまだ存続している場合を認識する ping 間隔（ミリ秒単位）
noRandomizeServers	**false**	接続の試行のためにサーバーの順序をランダム化するかどうか。
queueName	**null**	キュー設定（コンシューマー）に NATS を使用している場合は、キュー名。
maxMessages	**null**	maxMessages （コンシューマー）の後にサブスクライブするトピックからメッセージの受信を停止します。
poolSize	**10**	コンシューマーワーカー（コンシューマー）のプールサイズ。

HEADERS

名前	タイプ	説明
CamelNatsMessageTimestamp	**long**	消費されたメッセージのタイムスタンプ。
CamelNatsSubscriptionId	**Integer**	コンシューマーのサブスクリプション ID。

プロデューサーの例：

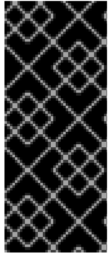
```
from("direct:send").to("nats://localhost:4222?topic=test");
```

コンシューマーの例：

```
from("nats://localhost:4222?
topic=test&maxMessages=5&queueName=test").to("mock:result");
```

第117章 NETTY

NETTY コンポーネント



重要

Camel Netty コンポーネントは JBoss Fuse 6.3 以降非推奨となり、今後のリリースで削除されます。代わりに新しい Camel Netty4 コンポーネントの使用に切り替える必要があります。

Camel 2.3 の時点で利用可能

Camel の Netty コンポーネントは、Netty プロジェクトに基づくソケット通信コンポーネントです。Netty は NIO クライアントサーバーフレームワークです。これにより、プロトコルサーバーやクライアントなどのネットワークアプリケーションを迅速かつ簡単に開発できます。Netty は、TCP や UDP ソケットサーバーなどのネットワークプログラミングを大幅に簡素化および合理化します。

ヒント

新しい Netty 4 を使用する Netty4 コンポーネントがあります。このコンポーネントは古い Netty 3 ライブラリーを使用するため、このコンポーネントを使用することが推奨されます。

この Camel コンポーネントは、プロデューサーとコンシューマーエンドポイントの両方をサポートします。

Netty コンポーネントには複数のオプションがあり、多くの TCP/UDP 通信パラメーター（バッファーサイズ、keepAlive、tcpNoDelay など）を詳細に制御し、Camel ルートでの In-Only および In-Out の両方の通信を容易にします。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-netty</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

netty コンポーネントの URI スキームは以下のとおりです。

```
netty:tcp://localhost:99999[?options]
netty:udp://remotehost:99999/[?options]
```

このコンポーネントは、TCP と UDP の両方のプロデューサーおよびコンシューマーエンドポイントをサポートします。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
keepAlive	**true**	非アクティブのためにソケットが閉じられないように設定
tcpNoDelay	**true**	TCP プロトコルパフォーマンスを改善するための設定
backlog		**Camel 2.9.6/2.10.4/2.11:** netty コンシューマー(server)のバックログを設定できます。バックログは、OS に応じてベストエフォートであることに注意してください。このオプションを **200**、**500**、**1000** などの値に設定すると、TCP スタックは accept キューが何であるかを示します。このオプションが設定されていない場合、バックログは OS の設定によって異なります。
broadcast	**false**	UDP でマルチキャストを選択するための設定
connectTimeout	**10000**	ソケット接続が利用可能になるまでの待機時間。値はミリ秒単位です。

reuseAddress	**true**	ソケット多重化を容易にするための設定
sync	**true**	エンドポイントを一方向または request-response として設定するための設定
同期	**false**	**Camel 2.10:** 非同期ルーティングエンジンが使用さ れていないかどうか。その **後**、非同期ルーティングエンジンが使用さ れ、**true** は同期処理を強制的に実行します。
ssl	**false**	SSL 暗号化をこのエンドポイントに適用するかどうかを指定するための設定
sslClientCertHeaders	**false**	**Camel 2.12:** 有効および SSL モードの場合、Netty コンシューマーはサブジェクト名、発行者名、シリアル番号、有効な日付範囲などのクライアント証明書が含まれるヘッダーで Camel メッセージ を強化します。
sendBufferSize	**65536 バイト**	アウトバウンド通信中に使用される TCP/UDP バッファーサイズ。サイズはバイトです。
receiveBufferSize	**65536 バイト**	インバウンド通信中に使用される TCP/UDP バッファーサイズ。サイズはバイトです。
option.XXX	**null**	**Camel 2.11/2.10.4:** 接頭辞として option. を使用して追加の netty オプションを設定できます。たとえば、"option.child.keepAlive=false" は netty オプション "child.keepAlive=false" を設定します。使用可能なオプションについては、Netty のドキュメントを参照してください。
corePoolSize	**10**	コンポーネントの起動時に割り当てられるスレッドの数。デフォルトは 10 です。**注記：** このオプションは Camel 2.9.2 以降から削除されます。Netty のデフォルト設定に依存するため、

maxPoolSize	**100**	このエンドポイントに割り当てることができるスレッドの最大数。デフォルトは 100 です。**注記：** このオプションは Camel 2.9.2 以降から削除されます。Netty のデフォルト設定に依存するため、
disconnect	**false**	使用直後に Netty チャネルから切断（閉じる）するかどうか。コンシューマーとプロデューサーの両方に使用できます。
lazyChannelCreation	**true**	チャネルは、Camel プロデューサーの開始時にリモートサーバーが稼働していない場合に例外を回避するために遅延して作成できます。
transferExchange	**false**	TCP にのみ使用されます。ボディーだけでなく、ネットワーク経由でエクスチェンジを転送することができます。以下のフィールドは転送されます：ボディー、Out body、Fault ボディー、In headers、Out ヘッダー、Fault ヘッダー、エクスチェンジプロパティー、エクスチェンジ例外。これには、オブジェクトがシリアライズ可能である必要があります。Camel はシリアル化できないオブジェクトを除外し、WARN レベルでログに記録します。
disconnectOnNoReply	**true**	sync が有効になっている場合、このオプションは NettyConsumer を指示します。この場合、返信の応答がない場合、このオプションは NettyConsumer を切断します。
noReplyLogLevel	**WARN**	sync が有効になっている場合、このオプションは NettyConsumer を決定し、送信応答がない場合に使用するロギングレベルを指定します。値は **FATAL**、**ERROR**、**INFO**、**DEBUG**、**OFF** です。
serverExceptionCaughtLogLevel	**WARN**	**Camel 2.11.1:** サーバー (NettyConsumer)が例外をキャッチする場合、このロギングレベルを使用してログに記録されます。

serverClosedChannelExceptionCaughtLogLevel	**DEBUG**	**Camel 2.11.1:** サーバー (NettyConsumer)が **java.nio.channels.ClosedChannelException** を取得する場合、このロギングレベルを使用してログに記録されます。これは、クライアントが突然切断され、Netty サーバーで閉じられた例外が発生する可能性があるため、閉じられたチャンネル例外を記録しないようにするために使用されます。
allowDefaultCodec	**true**	**Camel 2.4:** netty コンポーネントは、エンコーダー/ダンダーの両方が null で、テキストラインが false の場合にデフォルトのコーデックをインストールします。allowDefaultCodec を false に設定すると、netty コンポーネントがフィルターチェーンの最初の要素としてデフォルトの codec をインストールできなくなります。
textline	**false**	**Camel 2.4:** TCP にのみ使用されます。codec が指定されていない場合、このフラグを使用してテキスト行ベースのコーデックを示すことができます。指定されていない場合、または値が false の場合、Object Serialization は TCP 経由で想定されます。
delimiter	**LINE**	**Camel 2.4:** テキストラインコーデックに使用する区切り文字。使用できる値は **LINE** および **NULL** です。
decoderMaxLineLength	**1024**	**Camel 2.4:** テキストラインコーデックに使用する最大行の長さ。
autoAppendDelimiter	**true**	**Camel 2.4:** テキストラインコーデックを使用して送信する際に、不足している終了区切り文字を自動追加するかどうか。
encoding	**null**	**Camel 2.4:** テキストラインコーデックに使用するエンコーディング（文字セット名）。指定しない場合、Camel は JVM のデフォルト Charset を使用します。

workerCount	**null**	**Camel 2.9:** netty が nio モードで動作している場合、Netty は cpu_core_threads*2 である Netty のデフォルトの workerCount パラメーターを使用します。ユーザーはこの操作を使用して Netty からデフォルトの workerCount を上書きできます。
sslContextParameters	**null**	**Camel 2.9:** **org.apache.camel.util.jsse.SSLContextParameters** インスタンスを使用した SSL 設定。Using the JSSE Configuration Utility を参照してください。
receiveBufferSizePredictor	**null**	**Camel 2.9:** バッファーサイズ予測を設定します。詳細は Jetty のドキュメント および この メールスレッド を参照してください。
requestTimeout	**0**	**Camel 2.11.1:** リモートサーバーを呼び出すときに Netty プロデューサーのタイムアウトを使用できます。デフォルトでは、タイムアウトは使用されません。値はミリ秒単位です。**requestTimeout** オプションは Netty の **ReadTimeoutHandler** を使用してタイムアウトをトリガーします。**Camel 2.16、2.12.1.3:** は、**CamelNettyRequestTimeout** ヘッダーを設定してこの設定を上書きすることもできます。
needClientAuth	**false**	**Camel 2.11:** SSL の使用時にサーバーがクライアント認証を必要とするかどうかを設定します。
orderedThreadPoolExecutor	**true**	**Camel 2.10.2:** 順序付けされたスレッドプールを使用して、イベントが同じチャネルで順番に処理されるかどうか。詳細は、**org.jboss.netty.handler.execution.OrderedMemoryAwareThreadPoolExecutor** の netty javadoc の詳細を参照してください。
maximumPoolSize	**16**	**Camel 2.10.2:** 順序付けされたスレッドプールが使用されている場合のコアプールサイズ。

producerPoolEnabled	**true**	Camel 2.10.4/Camel 2.11: プロデューサーのみ。プロデューサープールが有効かどうか。**重要：** 同時実行性と信頼できるリクエスト/リプライを処理するのにプーリングが必要になるため、この機能をオフにしないでください。
producerPoolMaxActive	**-1**	Camel 2.10.3: プロデューサーのみ。プールが割り当てることができるオブジェクト数の上限を設定します（クライアントに対してチェックするか、チェックアウトをアイドリングします）。制限なしには負の値を使用してください。
producerPoolMinIdle	**0**	Camel 2.10.3: プロデューサーのみ。エビクタースレッド（アクティブな場合）が新しいオブジェクトを生成する前に、プロデューサープールで許可されるインスタンスの最小数を設定します。
producerPoolMaxIdle	**100**	Camel 2.10.3: プロデューサーのみ。プール内のアイドリングインスタンスの数の上限を設定します。
producerPoolMinEvictableIdle	**300000**	Camel 2.10.3: プロデューサーのみ。アイドルオブジェクトのエビクターによるエビクションの対象となる前に、オブジェクトがプールでアイドル状態にある可能性のある最小時間（ミリ秒単位）を設定します。
bootstrapConfiguration	**null**	Camel 2.12: コンシューマーのみ。**org.apache.camel.component.netty.NettyServerBootstrapConfiguration** インスタンスを使用して Netty ServerBootstrap オプションを設定できます。これは、複数のコンシューマーで同じ設定を再利用するために使用できます。これにより、設定をより簡単に調整できます。

bossGroup	**null**	Camel 2.12: 明示的な **org.jboss.netty.channel.socket.nio.BossPool** を boss スレッドプールとして使用する。たとえば、スレッドプールを複数のコンシューマーと共有するには、以下を実行します。デフォルトでは、各コンシューマーには 1 コアスレッドを持つ独自の boss プールがあります。
workerGroup	**null**	Camel 2.12: 明示的な **org.jboss.netty.channel.socket.nio.WorkerPool** をワーカースレッドプールとして使用する。たとえば、スレッドプールを複数のコンシューマーと共有するには、以下を実行します。デフォルトでは、各コンシューマーには 2 x cpu count コアスレッドを持つ独自のワーカープールがあります。
channelGroup	**null**	Camel 2.17: 明示的な **io.netty.channel.group.ChannelGroup**を使用して、メッセージを複数のチャネルに広げるには。
networkInterface	**null**	Camel 2.12: コンシューマーのみ。UDP を使用する場合、このオプションを使用して、マルチキャストグループに参加する **eth0** などの名前でネットワークインターフェイスを指定できます。
udpConnectionlessSending	**false**	Camel 2.15: プロデューサーのみ。このオプションは、接続なしの UDP 送信をサポートします。これは genuine fire-and-forget です。UDP 送信試行は、**PortUnreachableException** 例外を受け取ります（受信ポートでリッスンしているものがない場合）。
clientMode	**false**	Camel 2.15: コンシューマーのみ。**clientMode** が **true** の場合、Netty コンシューマーは TCP クライアントとしてアドレスに接続します。

useChannelBuffer	**false**	Camel 2.16: プロデューサーのみ。**useChannelBuffer** が **true** の場合、Netty プロデューサーはメッセージ本文を **channelBuffer** に切り替えてから送信します。

レジストリーベースのオプション

Codec ハンドラーおよび SSL キーストアは、Spring XML ファイルの など、レジストリー に登録できます。渡すことができる値は次のとおりです。

名前	説明
passphrase	SSH を使用して送信されたペイロードの暗号化/復号化に使用するパスワード設定
keyStoreFormat	ペイロードの暗号化に使用されるキーストア形式。設定されていない場合、デフォルトは JKS です。
securityProvider	ペイロードの暗号化に使用するセキュリティープロバイダー。設定されていない場合、デフォルトは SunX509 です。
keyStoreFile	**非推奨：** 暗号化に使用されるクライアント側の証明書キーストア
trustStoreFile	**非推奨：** 暗号化に使用されるサーバー側の証明書キーストア
keyStoreResource	Camel 2.11.1: 暗号化に使用されるクライアント側の証明書キーストア。デフォルトではクラスパスからロードされますが、**"classpath:"**、**"file:"**、または **"http:"** をプレフィックスとして指定して、異なるシステムからリソースをロードすることもできます。
trustStoreResource	Camel 2.11.1: 暗号化に使用されるサーバー側の証明書キーストア。デフォルトではクラスパスからロードされますが、**"classpath:"**、**"file:"**、または **"http:"** をプレフィックスとして指定して、異なるシステムからリソースをロードすることもできます。
sslHandler	SSL ハンドラーを返すために使用できるクラスへの参照

encoder	アウトバウンドペイロードの特別なマーシャリングを実行するために使用できるカスタム **ChannelHandler** クラス。**org.jboss.netty.channel.ChannelDownStreamHandler** をオーバーライドする必要があります。
encorders	使用するエンコーダーのリスト。コンマ区切りの値を持つ文字列を使用し、値をレジストリー で検索でき ます。Camel がルックアップする必要があることを認識できるように、値の前に # を付けることを忘れないでください。
decoder	インバウンドペイロードの特別なマーシャリングを実行するために使用できるカスタム **ChannelHandler** クラス。Must override **org.jboss.netty.channel.ChannelUpStreamHandler**.decoder が定義されていない場合、Netty はデフォルトで ObjectDecoder クラスを介してシリアライズされた Java オブジェクトに設定されます。別の形式が想定される場合は、デコーダーを指定する必要があります。
decoders	使用するデコーダーのリスト。コンマ区切りの値を持つ文字列を使用し、値をレジストリー で検索できます。Camel がルックアップする必要があることを認識できるように、値の前に # を付けることを忘れないでください。

重要： 共有不可能なエンコーダー/デコーダーの使用について以下をお読みください。

共有不可能なエンコーダーまたはデコーダーの使用

エンコーダーまたはデコーダーが共有できない場合（たとえば、@Shareable クラスアノテーションがある場合）、エンコーダー/デコーダーは org.apache.camel.component.netty.ChannelHandlerFactory インターフェイスを実装し、newChannelHandler メソッドで新規インスタンスを返します。これにより、エンコーダー/デコーダーを安全に使用できるようになります。そうでない場合、Netty コンポーネントはエンドポイントの作成時に WARN をログに記録します。

Netty コンポーネントは、一般的に使用されるメソッドが多数含まれる org.apache.camel.component.netty.ChannelHandlerFactories ファクトリークラスを提供します。

NETTY エンドポイントとの間でメッセージを送信する

NETTY プロデューサー

Producer モードでは、コンポーネントは TCP または UDP プロトコル（オプションの SSL サポートあり）を使用して、ソケットエンドポイントにペイロードを送信する機能を提供します。

プロデューサーモードは、一方向および要求応答ベースの操作の両方をサポートします。

NETTY コンシューマー

Consumer モードでは、コンポーネントは以下を行う機能を提供します。

- *TCP または UDP プロトコル（任意の SSL サポートあり）を使用して、指定したソケットでリッスンします。*
- *text/xml、バイナリー、およびシリアライズされたオブジェクトベースのペイロードを使用して、ソケットで要求を受信します。*
- *それらをメッセージ交換としてルートに送信します。*

コンシューマーモードは、一方向および要求応答ベースの操作の両方をサポートします。

HEADERS

Netty コンシューマーによって作成されたエクスチェンジについて、以下のヘッダーが入力されます。

ヘッダーのキー	クラス	説明
NettyConstants.NETTY_CHANNEL_HANDLER_CONTEXT / CamelNettyChannelHandlerContext	**org.jboss.netty.channel.ChannelHandlerContext**	**ChannelHandlerContext** Netty によって受信される接続に関連付けられたインスタンス。

ヘッダーのキー	クラス	説明
NettyConstants.NETTY_MESSAGE_EVENT / CamelNettyMessageEvent	**org.jboss.netty.channel.MessageEvent**	**MessageEvent** Netty によって受信される接続に関連付けられたインスタンス。
NettyConstants.NETTY_REMOTE_ADDRESS / CamelNettyRemoteAddress	**java.net.SocketAddress**	着信ソケット接続のリモートアドレス。
NettyConstants.NETTY_LOCAL_ADDRESS / CamelNettyLocalAddress	**java.net.SocketAddress**	着信ソケット接続のローカルアドレス。

REQUEST-REPLY およびシリアライズされたオブジェクトペイロードを使用した UDP NETTY エンドポイント

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
    from("netty:udp://localhost:5155?sync=true")
      .process(new Processor() {
         public void process(Exchange exchange) throws Exception {
           Poetry poetry = (Poetry) exchange.getIn().getBody();
           poetry.setPoet("Dr. Sarojini Naidu");
           exchange.getOut().setBody(poetry);
         }
       }
    }
};
```

一方向通信を使用した TCP ベースの NETTY コンシューマーエンドポイント

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
       from("netty:tcp://localhost:5150")
           .to("mock:result");
  }
};
```

REQUEST-REPLY 通信を使用する SSL/TCP ベースの NETTY コンシューマーエンドポイント

JSSE 設定ユーティリティーの使用

Camel 2.9 の時点で、Netty コンポーネントは を介して SSL/TLS 設定をサポートし

http://camel.apache.org/camel-configuration-utilities.html ます。このユーティリティーは、作成する必要のあるコンポーネント固有のコードの量を大幅に減らし、エンドポイントおよびコンポーネントレベルで設定可能です。 以下の例は、Netty コンポーネントで ユーティリティーを使用する方法を示しています。

コンポーネントのプログラムによる設定

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

NettyComponent nettyComponent = getContext().getComponent("netty",
NettyComponent.class);
nettyComponent.setSslContextParameters(scp);
```

エンドポイントの SPRING DSL ベースの設定

```
...
  <camel:sslContextParameters
      id="sslContextParameters">
    <camel:keyManagers
        keyPassword="keyPassword">
      <camel:keyStore
          resource="/users/home/server/keystore.jks"
          password="keystorePassword"/>
    </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="netty:tcp://localhost:5150?
sync=true&ssl=true&sslContextParameters=#sslContextParameters"/>
...
```

JETTY コンポーネントでの基本的な SSL/TLS 設定の使用

```
JndiRegistry registry = new JndiRegistry(createJndiContext());
registry.bind("password", "changeit");
registry.bind("ksf", new File("src/test/resources/keystore.jks"));
registry.bind("tsf", new File("src/test/resources/keystore.jks"));

context.createRegistry(registry);
context.addRoutes(new RouteBuilder() {
  public void configure() {
      String netty_ssl_endpoint =
         "netty:tcp://localhost:5150?sync=true&ssl=true&passphrase=#password"
```

```
            + "&keyStoreFile=#ksf&trustStoreFile=#tsf";
          String return_string =
            "When You Go Home, Tell Them Of Us And Say,"
            + "For Your Tomorrow, We Gave Our Today.";

          from(netty_ssl_endpoint)
          .process(new Processor() {
             public void process(Exchange exchange) throws Exception {
               exchange.getOut().setBody(return_string);
             }
          }
     }
   });
```

SSLSESSION およびクライアント証明書へのアクセス

Camel 2.12 以降で利用可能

たとえば、クライアント証明書の詳細を取得する必要がある場合は、javax.net.ssl.SSLSession にアクセスできます。ssl=true の場合、Netty コンポーネントは以下のように SSLSession を Camel Message のヘッダーとして保存します。

```
SSLSession session = exchange.getIn().getHeader(NettyConstants.NETTY_SSL_SESSION,
SSLSession.class);
// get the first certificate which is client certificate
javax.security.cert.X509Certificate cert = session.getPeerCertificateChain()[0];
Principal principal = cert.getSubjectDN();
```

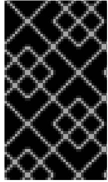
needClientAuth=true を設定してクライアントを認証します。設定しないと、SSLSession はクライアント証明書に関する情報にアクセスできず、例外 javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated が発生する可能性があります。クライアント証明書の有効期限が切れている場合や無効な場合は、この例外が発生する場合もあります。

ヒント

sslClientCertHeaders オプションは true に設定でき、クライアント証明書に関する詳細が含まれるヘッダーで Camel Message を補完します。たとえば、サブジェクト名は、ヘッダー CamelNettySSLClientCertSubjectName で簡単に利用できます。

複数のコーデックの使用

特定のケースでは、エンコーダーとデコーダーのチェーンを netty パイプラインに追加する必要がある場合があります。multpile codecs を camel netty エンドポイントに追加するには、encoders およびdecoders URI パラメーターを使用する必要があります。encoder パラメーターおよびdecoder パラメーターと同様に、パイプラインに追加する必要のある ChannelUpstreamHandlers および ChannelDownstreamHandlers のリストへの参照を提供するために使用されます。エンコーダーを指定すると、デコーダーやデコーダーパラメーターと同様にエンコーダーパラメーターが無視されます。



重要

共有不可能なエンコーダー/デコーダーの使用について、上記を参照してください。

エンドポイントの作成時に解決できるように、コーデックのリストを Camel のレジストリーに追加する必要があります。

```
ChannelHandlerFactory lengthDecoder =
ChannelHandlerFactories.newLengthFieldBasedFrameDecoder(1048576, 0, 4, 0, 4);

StringDecoder stringDecoder = new StringDecoder();
registry.bind("length-decoder", lengthDecoder);
registry.bind("string-decoder", stringDecoder);

LengthFieldPrepender lengthEncoder = new LengthFieldPrepender(4);
StringEncoder stringEncoder = new StringEncoder();
registry.bind("length-encoder", lengthEncoder);
registry.bind("string-encoder", stringEncoder);

List<ChannelHandler> decoders = new ArrayList<ChannelHandler>();
decoders.add(lengthDecoder);
decoders.add(stringDecoder);

List<ChannelHandler> encoders = new ArrayList<ChannelHandler>();
encoders.add(lengthEncoder);
encoders.add(stringEncoder);

registry.bind("encoders", encoders);
registry.bind("decoders", decoders);
```

Spring のネイティブコレクションサポートを使用して、アプリケーションコンテキストでコーデックリストを指定できます。

```
    <util:list id="decoders" list-class="java.util.LinkedList">
        <bean class="org.apache.camel.component.netty.ChannelHandlerFactories" factory-
method="newLengthFieldBasedFrameDecoder">
            <constructor-arg value="1048576"/>
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
```

```
        </bean>
        <bean class="org.jboss.netty.handler.codec.string.StringDecoder"/>
    </util:list>

    <util:list id="encoders" list-class="java.util.LinkedList">
        <bean class="org.jboss.netty.handler.codec.frame.LengthFieldPrepender">
            <constructor-arg value="4"/>
        </bean>
        <bean class="org.jboss.netty.handler.codec.string.StringEncoder"/>
    </util:list>

    <bean id="length-encoder"
class="org.jboss.netty.handler.codec.frame.LengthFieldPrepender">
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-encoder" class="org.jboss.netty.handler.codec.string.StringEncoder"/>

    <bean id="length-decoder"
class="org.apache.camel.component.netty.ChannelHandlerFactories" factory-
method="newLengthFieldBasedFrameDecoder">
        <constructor-arg value="1048576"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-decoder" class="org.jboss.netty.handler.codec.string.StringDecoder"/>

</beans>
```

Bean 名は、コンマ区切りリストとして、または List に含めるか、netty エンドポイント定義で使用できます。

```
                from("direct:multiple-codec").to("netty:tcp://localhost:{{port}}?
encoders=#encoders&sync=false");

                from("netty:tcp://localhost:{{port}}?decoders=#length-decoder,#string-
decoder&sync=false").to("mock:multiple-codec");
            }
        };
    }
}
```

または spring を使用します。

```
<camelContext id="multiple-netty-codecs-context"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:multiple-codec"/>
        <to uri="netty:tcp://localhost:5150?encoders=#encoders&ync=false"/>
    </route>
```

```
  <route>
    <from uri="netty:tcp://localhost:5150?decoders=#length-decoder,#string-
decoder&ync=false"/>
    <to uri="mock:multiple-codec"/>
  </route>
</camelContext>
```

完了したらチャンネルを閉じる

サーバーとして動作する場合は、クライアントの変換が終了する場合などにチャネルを閉じることがあります。これを行うには、エンドポイントオプション disconnect=true を設定します。

ただし、以下のようにメッセージごとに Camel に指示することもできます。Camel に対してチャネルを閉じるよう指示するには、CamelNettyCloseChannelWhenComplete キーを持つヘッダーをブール値 true に追加する必要があります。たとえば、以下の例では、bye メッセージをクライアントに書き戻した後にチャンネルを閉じます。

```
    from("netty:tcp://localhost:8080").process(new Processor() {
      public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        exchange.getOut().setBody("Bye " + body);
        // some condition which determines if we should close
        if (close) {

exchange.getOut().setHeader(NettyConstants.NETTY_CLOSE_CHANNEL_WHEN_COMPLETE,
true);
        }
      }
    });
```

作成されたパイプラインを完全に制御するためのカスタムチャネルパイプラインファクトリーの追加

Camel 2.5 で利用可能

カスタムチャネルパイプラインは、カスタムハンドラーを挿入してハンドラー/インターセプターチェーンを完全に制御します。これは、Netty Endpoint URL で指定しなくても、エンコーダーとデコーダーを簡単に指定します。

カスタムパイプラインを追加するには、カスタムチャネルパイプラインファクトリーを作成し、コンテキストレジストリー(JNDIRegistry、または camel-spring ApplicationContextRegistry など)でコンテキストに登録する必要があります。

カスタムパイプラインファクトリーは以下のように構築する必要があります。

- Producer リンクされたチャネルパイプラインファクトリーは、ClientPipelineFactory の抽象クラスを拡張する必要があります。

- Consumer リンクされたチャネルパイプラインファクトリーは、ServerPipelineFactory の抽象クラスを拡張する必要があります。

- クラスは、カスタムハンドラー、エンコーダー、およびデコーダーを挿入するために、getPipeline （）メソッドを上書きする必要があります。getPipeline （）メソッドを上書きしない場合、ハンドラー、エンコーダー、またはデコーダーのないパイプラインがパイプラインに有線されます。

以下の例は、ServerChannel Pipeline ファクトリーを作成する方法を示しています。

カスタムパイプラインファクトリーの使用

```
public class SampleServerChannelPipelineFactory extends
ServerPipelineFactory {
    private int maxLineSize = 1024;

    public ChannelPipeline getPipeline() throws Exception {
        ChannelPipeline channelPipeline = Channels.pipeline();

        channelPipeline.addLast("encoder-SD", new
StringEncoder(CharsetUtil.UTF_8));
        channelPipeline.addLast("decoder-DELIM", new
DelimiterBasedFrameDecoder(maxLineSize, true, Delimiters.lineDelimiter()));
        channelPipeline.addLast("decoder-SD", new
StringDecoder(CharsetUtil.UTF_8));
        // here we add the default Camel ServerChannelHandler for the consumer,
to allow Camel to route the message etc.
        channelPipeline.addLast("handler", new
ServerChannelHandler(consumer));

        return channelPipeline;
    }
}
```

次に、カスタムチャネルパイプラインファクトリーをレジストリーに追加し、以下のように Camel ルートでインスタンス化/活用できます。

```
Registry registry = camelContext.getRegistry();
```

```
serverPipelineFactory = new TestServerChannelPipelineFactory();
registry.bind("spf", serverPipelineFactory);
context.addRoutes(new RouteBuilder() {
  public void configure() {
      String netty_ssl_endpoint =
         "netty:tcp://localhost:5150?serverPipelineFactory=#spf"
      String return_string =
         "When You Go Home, Tell Them Of Us And Say,"
         + "For Your Tomorrow, We Gave Our Today.";

      from(netty_ssl_endpoint)
       .process(new Processor() {
          public void process(Exchange exchange) throws Exception {
            exchange.getOut().setBody(return_string);
          }
       }
  }
});
```

NETTY BOSS およびワーカースレッドプールの再利用

Camel 2.12 以降で利用可能

Netty には boss と worker の 2 種類のスレッドプールがあります。デフォルトでは、各 Netty コンシューマーおよびプロデューサーにはプライベートスレッドプールがあります。複数のコンシューマーまたはプロデューサー間でこれらのスレッドプールを再利用する場合は、スレッドプールを作成し、レジストリー に登録する必要があり ます。

たとえば、Spring XML を使用する場合は、以下のように 2 つのワーカースレッドを持つ NettyWorkerPoolBuilder を使用して共有ワーカースレッドプールを作成できます。

```
<!-- use the worker pool builder to create to help create the shared thread pool -->
<bean id="poolBuilder"
class="org.apache.camel.component.netty.NettyWorkerPoolBuilder">
  <property name="workerCount" value="2"/>
</bean>

<!-- the shared worker thread pool -->
<bean id="sharedPool" class="org.jboss.netty.channel.socket.nio.WorkerPool"
     factory-bean="poolBuilder" factory-method="build" destroy-method="shutdown">
</bean>
```

ヒント

boss スレッドプールの場合は、Netty コンシューマーには org.apache.camel.component.netty.NettyServerBossPoolBuilder ビルダーと、Netty 生成用の org.apache.camel.component.netty.NettyClientBossPoolBuilder があります。

Camel ルートでは、以下のように URI で workerPool オプションを設定することで、このワーカープールを参照できます。

```
<route>
  <from uri="netty:tcp://localhost:5021?
textline=true&ync=true&orkerPool=#sharedPool&rderedThreadPoolExecutor=false"/>
  <to uri="log:result"/>
  ...
</route>
```

別のルートがある場合は、共有ワーカープールを参照できます。

```
<route>
  <from uri="netty:tcp://localhost:5022?
textline=true&ync=true&orkerPool=#sharedPool&rderedThreadPoolExecutor=false"/>
  <to uri="log:result"/>
  ...
</route>
```

... など。

その他の参考資料

- *Netty HTTP*
- *Mina2*

第118章 NETTY4

NETTY4 コンポーネント

Camel の Netty4 コンポーネントは、Netty プロジェクトバージョン 4 に基づくソケット通信コンポーネントです。Netty は NIO クライアントサーバーフレームワークです。これにより、プロトコルサーバーやクライアントなどの netwServerInitializerFactoryork アプリケーションを迅速かつ簡単に開発できます。Netty4 は、TCP や UDP ソケットサーバーなどのネットワークプログラミングを大幅に簡素化および合理化します。

この Camel コンポーネントは、プロデューサーとコンシューマーエンドポイントの両方をサポートします。

Netty コンポーネントには複数のオプションがあり、多くの TCP/UDP 通信パラメーター（バッファーサイズ、keepAlive、tcpNoDelay など）を詳細に制御し、Camel ルートでの In-Only および In-Out の両方の通信を容易にします。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty4</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

netty コンポーネントの URI スキームは以下のとおりです。

```
netty4:tcp://localhost:99999[?options]
netty4:udp://remotehost:99999/[?options]
```

このコンポーネントは、TCP と UDP の両方のプロデューサーおよびコンシューマーエンドポイントをサポートします。

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

オプション

名前	デフォルト値	説明
keepAlive	**true**	非アクティブのためにソケットが閉じられないように設定
tcpNoDelay	**true**	TCP プロトコルパフォーマンスを改善するための設定
backlog		netty コンシューマー（サーバー）にバックログを設定できます。バックログは、OS に応じてベストエフォートであることに注意してください。このオプションを **200**、**500** または **1000** などの値に設定すると、TCP スタックは accept キューの長さを指定します。このオプションが設定されていない場合、バックログは OS の設定によって異なります。
broadcast	**false**	UDP でマルチキャストを選択するための設定
connectTimeout	**10000**	ソケット接続が利用可能になるまでの待機時間。値はミリ秒単位です。
reuseAddress	**true**	ソケット多重化を容易にするための設定
sync	**true**	エンドポイントを一方向または request-response として設定するための設定
synchronous	**false**	Asynchronous Routing Engine が使用されないかどうか。**false** 次に、非同期ルーティングエンジン (**true**)を使用して同期の処理を強制します。
ssl	**false**	SSL 暗号化をこのエンドポイントに適用するかどうかを指定するための設定

sslClientCertHeaders	**false**	有効および SSL モードの場合、Netty コンシューマーはサブジェクト名、発行者名、シリアル番号、有効な日付範囲などのクライアント証明書に関する情報で Camel Message を補完します。
sendBufferSize	**65536 bytes**	アウトバウンド通信中に使用される TCP/UDP バッファーサイズ。サイズはバイトです。
receiveBufferSize	**65536 bytes**	インバウンド通信中に使用される TCP/UDP バッファーサイズ。サイズはバイトです。
option.XXX	**null**	"option." を接頭辞として使用して、追加の netty オプションを設定できます。たとえば、"option.child.keepAlive=false" は netty オプション "child.keepAlive=false" を設定します。使用可能なオプションについては、Netty のドキュメントを参照してください。
corePoolSize	**10**	コンポーネントの起動時に割り当てられるスレッドの数。デフォルトは 10 です。**注記：** このオプションは Camel 2.9.2 以降から削除されます。Netty のデフォルト設定に依存するため、
maxPoolSize	**100**	このエンドポイントに割り当てることができるスレッドの最大数。デフォルトは 100 です。**注記：** このオプションは Camel 2.9.2 以降から削除されます。Netty のデフォルト設定に依存するため、
disconnect	**false**	使用直後に Netty チャネルから切断（閉じる）するかどうか。コンシューマーとプロデューサーの両方に使用できます。
lazyChannelCreation	**true**	チャネルは、Camel プロデューサーの開始時にリモートサーバーが稼働していない場合に例外を回避するために遅延して作成できます。

transferExchange	**false**	TCP にのみ使用されます。ボディーだけでなく、ネットワーク経由でエクスチェンジを転送することができます。以下のフィールドは転送されます：ボディー、Out body、Fault ボディー、In headers、Out ヘッダー、Fault ヘッダー、エクスチェンジプロパティー、エクスチェンジ例外。これには、オブジェクトがシリアライズ可能である必要があります。Camel はシリアル化できないオブジェクトを除外し、WARN レベルでログに記録します。
disconnectOnNoReply	**true**	sync が有効になっている場合、このオプションは NettyConsumer を指示します。この場合、返信の応答がない場合、このオプションは NettyConsumer を切断します。
noReplyLogLevel	**WARN**	sync が有効になっている場合、このオプションは NettyConsumer を決定し、送信応答がない場合に使用するロギングレベルを指定します。値： **FATAL, ERROR, INFO, DEBUG, OFF**
serverExceptionCaughtLogLevel	**WARN**	サーバー(NettyConsumer)が例外をキャッチする場合、このロギングレベルを使用してログに記録されます。
serverClosedChannelExceptionCaughtLogLevel	**DEBUG**	サーバー(NettyConsumer)が **java.nio.channels.ClosedChannelException** をキャッチすると、このログレベルを使用してログに記録されます。これは、クライアントが突然切断され、Netty サーバーで閉じられた例外が発生する可能性があるため、閉じられたチャンネル例外を記録しないようにするために使用されます。
allowDefaultCodec	**true**	netty コンポーネントは、エンコーダー/ダンダーが null で、テキストラインが false の場合にデフォルトのコーデックをインストールします。allowDefaultCodec を false に設定すると、netty コンポーネントがフィルターチェーンの最初の要素としてデフォルトの codec をインストールできなくなります。

textline	**false**	TCP にのみ使用されます。codec が指定されていない場合、このフラグを使用してテキスト行ベースのコーデックを示すことができます。指定されていない場合、または値が false の場合、Object Serialization は TCP 経由で想定されます。
delimiter	**LINE**	テキストラインコーデックに使用する区切り文字。使用できる値は **LINE** および **NULL** です。
decoderMaxLineLength	**1024**	テキストラインコーデックに使用する最大行の長さ。
autoAppendDelimiter	**true**	テキストラインコーデックを使用して送信する際に、不足している終了区切り文字を自動追加するかどうか。
encoding	**null**	テキストラインコーデックに使用するエンコーディング（文字セット名）。指定しない場合、Camel は JVM のデフォルト Charset を使用します。
workerCount	**null**	netty が nio モードで機能する場合、Netty は cpu_core_threads*2 のデフォルトの workerCount パラメーターを使用します。ユーザーはこの操作を使用して Netty からデフォルトの workerCount を上書きできます。
sslContextParameters	**null**	**org.apache.camel.util.jsse.SSLContextParameters** インスタンスを使用した SSL 設定。Security Guide および Using the JSSE Configuration Utility の Configuring Transport Security for Camel Components の章 を参照してください。
receiveBufferSizePredictor	**null**	バッファーサイズ予測を設定します。詳細は Jetty のドキュメントおよび この メールスレッド を参照してください。

requestTimeout	**0**	リモートサーバーを呼び出すときに Netty プロデューサーのタイムアウトを使用できます。デフォルトでは、タイムアウトは使用されません。値はミリ秒単位です。**requestTimeout** は Netty の **ReadTimeoutHandler** を使用してタイムアウトをトリガーします。**Camel 2.16、2.12.1.3:** は、**CamelNettyRequestTimeout** ヘッダーを設定してこの設定を上書きすることもできます。
needClientAuth	**false**	SSL の使用時にサーバーがクライアント認証を必要とするかどうかを設定します。
usingExecutorService	**true**	executorService を使用して camel ルート内でメッセージを処理するかどうかは、executorService を NettyComponent から設定できます。
maximumPoolSize	**16**	順序付けされたスレッドプールが使用されている場合のコアプールサイズ。**注記： これは、Camel 2.15 から 2.14.1 以降 NettyComponent レベルに設定できます。**
producerPoolEnabled	**true**	プロデューサーのみ。プロデューサープールが有効かどうか。**重要：** 同時実行性と信頼できるリクエスト/リプライを処理するのにプーリングが必要になるため、この機能をオフにしないでください。
producerPoolMaxActive	**-1**	プロデューサーのみ。プールが割り当てることができるオブジェクト数の上限を設定します（クライアントに対してチェックするか、チェックアウトをアイドリングします）。制限なしには負の値を使用してください。
producerPoolMinIdle	**0**	プロデューサーのみ。エビクタースレッド（アクティブな場合）が新しいオブジェクトを生成する前に、プロデューサープールで許可されるインスタンスの最小数を設定します。
producerPoolMaxIdle	**100**	プロデューサーのみ。プール内のアイドリングインスタンスの数の上限を設定します。

producerPoolMinEvictableIdle	**300000**	プロデューサーのみ。アイドルオブジェクトのエビクターによるエビクションの対象となる前に、オブジェクトがプールでアイドル状態にある可能性のある最小時間（ミリ秒単位）を設定します。
bootstrapConfiguration	**null**	コンシューマーのみ。**org.apache.camel.component.netty4.NettyServerBootstrapConfiguration** インスタンスを使用して Netty ServerBootstrap オプションを設定できます。これは、複数のコンシューマーで同じ設定を再利用するために使用できます。これにより、設定をより簡単に調整できます。
bossGroup	**null**	boss スレッドプールとして明示的な **io.netty.channel.EventLoopGroup** を使用します。たとえば、スレッドプールを複数のコンシューマーと共有するには、以下を実行します。デフォルトでは、各コンシューマーには 1 コアスレッドを持つ独自の boss プールがあります。
workerGroup	**null**	明示的な **io.netty.channel.EventLoopGroup** をワーカースレッドプールとして使用するには、以下を実行します。たとえば、スレッドプールを複数のコンシューマーと共有するには、以下を実行します。デフォルトでは、各コンシューマーには 2 x cpu count コアスレッドを持つ独自のワーカープールがあります。
channelGroup	**null**	Camel 2.17: 明示的な **io.netty.channel.group.ChannelGroup**を使用して、メッセージを複数のチャネルに広げるには。
networkInterface	**null**	コンシューマーのみ。UDP を使用する場合、このオプションを使用して、マルチキャストグループに参加する **eth0** などの名前でネットワークインターフェイスを指定できます。

clientInitializerFactory	**null**	Camel 2.15: カスタムクライアントイニシャライザーファクトリーを使用して、チャネルのパイプラインを制御します。詳細については以下をご覧ください
serverInitializerFactory	**null**	Camel 2.15: カスタムサーバーイニシャライザーファクトリーを使用して、チャネルのパイプラインを制御します。詳細については以下をご覧ください
clientPipelineFactory	**null**	**非推奨**: 代わりに **clientInitializerFactory** を使用してください。
serverPipelineFactory	**null**	**非推奨**: 代わりに **serverInitializerFactory** を使用してください。
udpConnectionlessSending	**false**	Camel 2.15: プロデューサーのみ。このオプションは、接続なしの UDP 送信をサポートします。これは genuine fire-and-forget です。UDP 送信試行は、**PortUnreachableException** 例外を受け取ります（受信ポートでリッスンしているものがない場合）。
clientMode	**false**	Camel 2.15: コンシューマーのみ。**clientMode** が **true** の場合、Netty コンシューマーは TCP クライアントとしてアドレスに接続します。
reconnect	**true**	Camel 2.16: コンシューマーのみ。コンシューマーで **clientMode** でのみ使用され、コンシューマーは切断時に自動的に再接続を試みます。
reconnectInterval	**10000**	Camel 2.16: コンシューマーのみ。**reconnect** および **clientMode** が有効な場合に使用されます。再接続を試行する間隔（ミリ秒単位）。

useByteBuf	**false**	**Camel 2.16:** プロデューサーのみ。**useByteBuf** が **true** の場合、Netty プロデューサーはメッセージ本文を **ByteBuf** に変換してから送信してください。
udpByteArrayCodec	**false**	**Camel 2.16:** UDP プロトコルを使用すると、このオプションを **true** にすると、デフォルトのオブジェクトシリアル化コーデックではなく、バイト配列としてデータが送信されます。
reuseChannel	**false**	**Camel 2.17:** プロデューサーのみ。このオプションを使用すると、プロデューサーはエクスチェンジの処理ライフサイクルで同じ Netty チャネルを再利用できます。これは、Camel ルートでサーバーを複数回呼び出す必要があり、同じネットワーク接続を使用する場合に使用できます。これを使用する場合、エクスチェンジが完了するまでチャンネルは接続プールに戻されません。disconnect オプションが true に設定されている場合は切断されます。再利用された Channel は、鍵 **NettyConstants#NETTY_CHANNEL** を使用してエクスチェンジプロパティーとしてエクスチェンジプロパティーとして保存されます。これにより、ルーティング中にチャネルを取得して使用することができます。

レジストリーベースのオプション

Codec ハンドラーおよび SSL キーストアは、Spring XML ファイルの など、レジストリー に登録できます。渡すことができる値は次のとおりです。

名前	説明
passphrase	SSH を使用して送信されたペイロードの暗号化/復号化に使用するパ
keyStoreFormat	ペイロードの暗号化に使用されるキーストア形式。設定されていない
securityProvider	ペイロードの暗号化に使用するセキュリティープロバイダー。設定さ

keyStoreFile	**非推奨：** 暗号化に使用されるクライアント側の証明書キーストア
trustStoreFile	**非推奨：** 暗号化に使用されるサーバー側の証明書キーストア
keyStoreResource	Camel 2.11.1: 暗号化に使用されるクライアント側の証明書キーストア。 リソースを読み込むには、**"classpath:"**、**"file:"**、または **"http:"** の
trustStoreResource	Camel 2.11.1: 暗号化に使用されるサーバー側の証明書キーストア。は ソースを読み込むには、**"classpath:"**、**"file:"**、または **"http:"** の接
sslHandler	SSL ハンドラーを返すために使用できるクラスへの参照
encoder	アウトバウンドペイロードの特別なマーシャリングを実行するために ス。**io.netty.channel.ChannelInboundHandlerAdapter** を上書
encoders	使用するエンコーダーのリスト。コンマ区切りの値を持つ文字列を使 必要があることを認識できるように、値の前に # を付けることを忘れ
decoder	インバウンドペイロードの特別なマーシャリングを実行するために使 ス。**io.netty.channel.ChannelOutboundHandlerAdapter** を上書
decoders	使用するデコーダーのリスト。コンマ区切りの値を持つ文字列を使用 要があることを認識できるように、値の前に # を付けることを忘れな

重要： 共有不可能なエンコーダー/デコーダーの使用について以下をお読みください。

共有不可能なエンコーダーまたはデコーダーの使用

エンコーダーまたはデコーダーが共有できない場合（たとえば、@Shareable クラスアノテーションがある場合）、エンコーダー/デコーダーは org.apache.camel.component.netty.ChannelHandlerFactory インターフェイスを実装し、newChannelHandler メソッドで新規インスタンスを返します。これにより、エンコーダー/デコーダーを安全に使用できるようになります。そうでない場合、Netty コンポーネントはエンドポイントの作成時に WARN をログに記録します。

Netty コンポーネントは、org.apache.camel.component.netty.ChannelHandlerFactories ファクトリークラスを提供します。このクラスには、一般的に使用されるメソッドが多数あります。

NETTY エンドポイントとの間でメッセージを送信する

NETTY プロデューサー

Producer モードでは、コンポーネントは TCP または UDP プロトコル（オプションの SSL サポートあり）を使用して、ソケットエンドポイントにペイロードを送信する機能を提供します。

プロデューサーモードは、一方向および要求応答ベースの操作の両方をサポートします。

NETTY コンシューマー

Consumer モードでは、コンポーネントは以下を行う機能を提供します。

- TCP または UDP プロトコル（任意の SSL サポートあり）を使用して、指定したソケットでリッスンします。
- text/xml、バイナリー、およびシリアライズされたオブジェクトベースのペイロードを使用して、ソケットで要求を受信します。
- それらをメッセージ交換としてルートに送信します。

コンシューマーモードは、一方向および要求応答ベースの操作の両方をサポートします。

使用例

REQUEST-REPLY およびシリアライズされたオブジェクトペイロードを使用した UDP NETTY エンドポイント

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
    from("netty4:udp://localhost:5155?sync=true")
      .process(new Processor() {
         public void process(Exchange exchange) throws Exception {
           Poetry poetry = (Poetry) exchange.getIn().getBody();
           poetry.setPoet("Dr. Sarojini Naidu");
           exchange.getOut().setBody(poetry);
         }
```

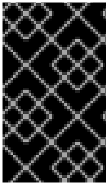
```
        }
    }
};
```

一方向通信を使用した TCP ベースの NETTY コンシューマーエンドポイント

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
      from("netty4:tcp://localhost:5150")
          .to("mock:result");
  }
};
```

複数のコーデックの使用

特定のケースでは、エンコーダーとデコーダーのチェーンを netty パイプラインに追加する必要がある場合があります。multpile codecs を camel netty エンドポイントに追加するには、encoders およびdecoders URI パラメーターを使用する必要があります。encoder パラメーターおよびdecoder パラメーターと同様に、パイプラインに追加する必要のある ChannelUpstreamHandlers および ChannelDownstreamHandlers のリストへの参照を提供するために使用されます。エンコーダーを指定すると、デコーダーやデコーダーパラメーターと同様にエンコーダーパラメーターが無視されます。



重要

共有不可能なエンコーダー/デコーダーの使用について、上記を参照してください。

エンドポイントの作成時に解決できるように、コーデックのリストを Camel のレジストリーに追加する必要があります。

```
ChannelHandlerFactory lengthDecoder =
ChannelHandlerFactories.newLengthFieldBasedFrameDecoder(1048576, 0, 4, 0, 4);

StringDecoder stringDecoder = new StringDecoder();
registry.bind("length-decoder", lengthDecoder);
registry.bind("string-decoder", stringDecoder);

LengthFieldPrepender lengthEncoder = new LengthFieldPrepender(4);
StringEncoder stringEncoder = new StringEncoder();
registry.bind("length-encoder", lengthEncoder);
registry.bind("string-encoder", stringEncoder);

List<ChannelHandler> decoders = new ArrayList<ChannelHandler>();
decoders.add(lengthDecoder);
decoders.add(stringDecoder);

List<ChannelHandler> encoders = new ArrayList<ChannelHandler>();
encoders.add(lengthEncoder);
```

```
encoders.add(stringEncoder);

registry.bind("encoders", encoders);
registry.bind("decoders", decoders);
```

Spring のネイティブコレクションサポートを使用して、アプリケーションコンテキストでコーデックリストを指定できます。

```
<util:list id="decoders" list-class="java.util.LinkedList">
        <bean class="org.apache.camel.component.netty4.ChannelHandlerFactories" factory-
method="newLengthFieldBasedFrameDecoder">
            <constructor-arg value="1048576"/>
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
        </bean>
        <bean class="io.netty.handler.codec.string.StringDecoder"/>
    </util:list>

    <util:list id="encoders" list-class="java.util.LinkedList">
        <bean class="io.netty.handler.codec.LengthFieldPrepender">
            <constructor-arg value="4"/>
        </bean>
        <bean class="io.netty.handler.codec.string.StringEncoder"/>
    </util:list>

    <bean id="length-encoder" class="io.netty.handler.codec.LengthFieldPrepender">
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-encoder" class="io.netty.handler.codec.string.StringEncoder"/>

    <bean id="length-decoder"
class="org.apache.camel.component.netty4.ChannelHandlerFactories" factory-
method="newLengthFieldBasedFrameDecoder">
        <constructor-arg value="1048576"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-decoder" class="io.netty.handler.codec.string.StringDecoder"/>
```

Bean 名は、コンマ区切りリストとして、または List に含めるか、netty エンドポイント定義で使用できます。

```
 from("direct:multiple-codec").to("netty4:tcp://localhost:{{port}}?
encoders=#encoders&sync=false");
```

```
from("netty4:tcp://localhost:{{port}}?decoders=#length-decoder,#string-
decoder&sync=false").to("mock:multiple-codec");
```

または spring を使用します。

```
<camelContext id="multiple-netty-codecs-context"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:multiple-codec"/>
        <to uri="netty4:tcp://localhost:5150?encoders=#encoders&amp;sync=false"/>
    </route>
    <route>
        <from uri="netty4:tcp://localhost:5150?decoders=#length-decoder,#string-
decoder&amp;sync=false"/>
        <to uri="mock:multiple-codec"/>
    </route>
</camelContext>
```

完了したらチャンネルを閉じる

サーバーとして動作する場合は、クライアントの変換が終了する場合などにチャネルを閉じることがあります。これは、エンドポイントのオプション disconnect=true を設定するだけで実行できます。

ただし、以下のようにメッセージごとに Camel に指示することもできます。Camel に対してチャネルを閉じるよう指示するには、CamelNettyCloseChannelWhenComplete キーが true のヘッダーをブール値 値に追加する必要があります。たとえば、以下の例では、bye メッセージをクライアントに書き戻した後にチャンネルを閉じます。

```
from("netty4:tcp://localhost:8080").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        exchange.getOut().setBody("Bye " + body);
        // some condition which determines if we should close
        if (close) {

exchange.getOut().setHeader(NettyConstants.NETTY_CLOSE_CHANNEL_WHEN_COMPLETE,
true);
        }
    }
});
```

作成されたパイプラインを完全に制御するためのカスタムチャネルパイプラインファクトリーの追加

カスタムチャネルパイプラインは、カスタムハンドラーを挿入してハンドラー/インターセプターチェーンを完全に制御します。これは、Netty Endpoint URL で指定しなくても、エンコーダーとデ

コーダーを簡単に指定します。

カスタムパイプラインを追加するには、カスタムチャネルパイプラインファクトリーを作成し、コンテキストレジストリー(JNDIRegistry、または camel-spring ApplicationContextRegistry など)でコンテキストに登録する必要があります。

カスタムパイプラインファクトリーは以下のように構築する必要があります。

- Producer リンクされたチャネルパイプラインファクトリーは、抽象クラス ClientInitializerFactory を拡張する必要があります。
- Consumer linked チャネルパイプラインファクトリーは、抽象クラス ServerInitializerFactory を拡張する必要があります。
- クラスは、initChannel() メソッドをオーバーライドして、カスタムハンドラー、エンコーダー、およびデコーダーを挿入する必要があります。initChannel() メソッドをオーバーライドしない場合は、ハンドラー、エンコーダー、またはデコーダーがないパイプラインを作成し、パイプラインにワイヤリングします。

以下の例は、ServerInitializerFactory ファクトリーを作成する方法を示しています。

例118.1 サーバーイニシャライザーファクトリーの使用

```
public class SampleServerInitializerFactory extends ServerInitializerFactory {
    private int maxLineSize = 1024;

    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline channelPipeline = ch.pipeline();

        channelPipeline.addLast("encoder-SD", new StringEncoder(CharsetUtil.UTF_8));
        channelPipeline.addLast("decoder-DELIM", new
DelimiterBasedFrameDecoder(maxLineSize, true, Delimiters.lineDelimiter()));
        channelPipeline.addLast("decoder-SD", new StringDecoder(CharsetUtil.UTF_8));
        // here we add the default Camel ServerChannelHandler for the consumer, to allow
Camel to route the message etc.
        channelPipeline.addLast("handler", new ServerChannelHandler(consumer));
    }
}
```

カスタムサーバーイニシャライザーファクトリーをレジストリーに追加し、以下のように Camel ルートでインスタンス化/活用できます。

```
Registry registry = camelContext.getRegistry();
ServerInitializerFactory factory = new TestServerInitializerFactory();
registry.bind("spf", factory);
context.addRoutes(new RouteBuilder() {
  public void configure() {
      String netty_ssl_endpoint =
         "netty4:tcp://localhost:5150?serverInitializerFactory=#spf"
      String return_string =
         "When You Go Home, Tell Them Of Us And Say,"
         + "For Your Tomorrow, We Gave Our Today.";

      from(netty_ssl_endpoint)
       .process(new Processor() {
          public void process(Exchange exchange) throws Exception {
            exchange.getOut().setBody(return_string);
          }
       }
  }
});
```

NETTY BOSS およびワーカースレッドプールの再利用

Camel 2.12 以降で利用可能

Netty には boss と worker の 2 種類のスレッドプールがあります。デフォルトでは、各 Netty コンシューマーおよびプロデューサーにはプライベートスレッドプールがあります。複数のコンシューマーまたはプロデューサー間でこれらのスレッドプールを再利用する場合は、スレッドプールを作成し、レジストリー に登録する必要があり ます。

たとえば、Spring XML を使用する場合は、以下のように 2 つのワーカースレッドを持つ NettyWorkerPoolBuilder を使用して共有ワーカースレッドプールを作成できます。

```
<!-- use the worker pool builder to create to help create the shared thread pool -->
<bean id="poolBuilder" class="org.apache.camel.component.netty.NettyWorkerPoolBuilder">
  <property name="workerCount" value="2"/>
</bean>

<!-- the shared worker thread pool -->
<bean id="sharedPool" class="org.jboss.netty.channel.socket.nio.WorkerPool"
      factory-bean="poolBuilder" factory-method="build" destroy-method="shutdown">
</bean>
```

ヒント

***boss** スレッドプールの場合、**Netty** コンシューマーの **org.apache.camel.component.netty4.NettyServerBossPoolBuilder** ビルダーがあり、**Netty** が生成する **org.apache.camel.component.netty4.NettyClientBossPoolBuilder** があります。*

***Camel** ルートでは、以下のように **URI** で **workerPool** オプションを設定することで、このワーカープールを参照できます。*

```
<route>
  <from uri="netty4:tcp://localhost:5021?textline=true&amp;sync=true&amp;workerPool=#sharedPool&amp;usingExecutorService=false"/>
  <to uri="log:result"/>
  ...
</route>
```

別のルートがある場合は、共有ワーカープールを参照できます。

```
<route>
  <from uri="netty4:tcp://localhost:5022?textline=true&amp;sync=true&amp;workerPool=#sharedPool&amp;usingExecutorService=false"/>
  <to uri="log:result"/>
  ...
</route>
```

... など。

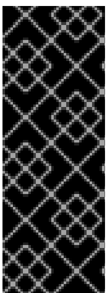
第119章 NETTY HTTP

NETTY HTTP コンポーネント

Camel 2.12 以降で利用可能

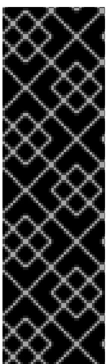
netty-http コンポーネントは Netty コンポーネントへの拡張であり、Netty で HTTP トランスポートを容易にします。

この Camel コンポーネントは、プロデューサーとコンシューマーエンドポイントの両方をサポートします。



NETTY 4.0 の計画アップグレード

このコンポーネントは、camel-netty4 コンポーネントのアップグレードが完了したときに Netty 4.0 を使用するようにアップグレードすることが意図されています。現時点では、このコンポーネントは Netty 3.x をベースとしています。アップグレードは、可能な限り後方互換性があることを目的としています。



ストリーム

Netty はストリームベースであり、受信する入力がストリームとして Camel に送信されることを意味します。つまり、ストリームのコンテンツを 1 度 だけ読み取ることができます。メッセージボディーが空であるように見える場合や、データに複数回アクセスする必要がある場合（例：マルチキャストの実行、再配信エラー処理）は、Stream Caching を使用するか、メッセージボディーを複数回再読み取りできる 文字列 に変換する必要があります。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty-http</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

netty コンポーネントの URI スキームは以下のとおりです。

> *netty-http:http://localhost:8080[?options]*

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

> *クエリーパラメーターとエンドポイントオプションの比較*
>
> *Camel が URI クエリーパラメーターおよびエンドポイントオプションを認識する方法に気づくかもしれません。たとえば、netty-http:http//example.com?myParam=myValue&compression=true のようにエンドポイント URI を作成できます。この例では、myParam は HTTP パラメーターですが、compression は Camel エンドポイントオプションです。このような状況で Camel によって使用されるストラテジーは、利用可能なエンドポイントオプションを解決し、URI から削除することです。これは、前述の例では、Netty HTTP プロデューサーによってエンドポイントに送信される HTTP リクエストは、http//example.com?myParam=myValue のようになります。compression エンドポイントオプションはターゲット URL から解決され、削除されます。*
>
> *また、動的ヘッダー(CamelHttpQueryなど)を使用してエンドポイントオプションを指定できないことに注意してください。エンドポイントオプションは、エンドポイント URI 定義レベルでのみ指定できます(to または from DSL 要素など)。*

HTTP オプション

> *その他のオプション*
>
> *重要： このコンポーネントは、Netty からすべてのオプションを継承します。そのため、Netty のドキュメントも確認してください。Netty の一部のオプションは、UDP トランスポートに関連するオプションなど、この Netty HTTP コンポーネントを使用する場合に適用されないことに注意してください。*

名前	デフォルト値	説明

chunkedMaxContentLength	**1mb**	Netty HTTP サーバーで受信されるチャンクごとの最大コンテンツ長（バイト単位）。
圧縮	**false**	クライアントが HTTP ヘッダーからサポートしている場合、Netty HTTP サーバーの圧縮に gzip/deflate の使用を許可します。
headerFilterStrategy		カスタムの **org.apache.camel.spi.HeaderFilterStrategy** を使用してヘッダーをフィルターリングします。
httpMethodRestrict		Netty HTTP コンシューマーで HTTP メソッドを無効にするには、以下を行います。コンマで区切って複数指定できます。
mapHeaders	**true**	このオプションを有効にすると、Netty から Camel Message へのバインド中にヘッダーもマッピングされます（たとえば、ヘッダーとして Camel Message に追加されます）。このオプションをオフにして無効にすることができます。ヘッダーには、Netty HTTP リクエスト **org.jboss.netty.handler.codec.http.HttpRequest** インスタンスを返す **getHttpRequest （)** メソッドが含まれる **org.apache.camel.component.netty.http.NettyHttpMessage** メッセージから引き続きアクセスできます。
matchOnUriPrefix	**false**	完全一致が見つからない場合に Camel が URI 接頭辞と一致することでターゲットコンシューマーの検索を試みるかどうか。詳細については以下をご覧ください
nettyHttpBinding		Netty および Camel Message API との間でバインドするためにカスタム **org.apache.camel.component.netty.http.NettyHttpBinding** を使用するには、以下を行います。

bridgeEndpoint	**false**	オプションが **true** の場合、プロデューサーは **Exchange.HTTP_URI** ヘッダーを無視し、リクエストにエンドポイントの URI を使用します。また、**throwExceptionOnFailure** を **false** に設定して、プロデューサーがすべての障害応答を返信するようにすることもできます。ブリッジモードで動作しているコンシューマーは、gzip 圧縮および WWW URL フォームエンコーディングを省略します(**Exchange.SKIP_GZIP_ENCODING** ヘッダーおよび **Exchange.SKIP_WWW_FORM_URLENCODED** ヘッダーを消費されたエクスチェンジに追加します)。
throwExceptionOnFailure	**true**	リモートサーバーからの応答が失敗した場合に **HttpOperationFailedException** を出力することを無効にするオプション。これにより、HTTP ステータスコードに関するすべての応答を取得できます。
traceEnabled	**false**	この Netty HTTP コンシューマーに対して HTTP TRACE を有効にするかどうかを指定します。デフォルトでは、TRACE はオフになっています。
transferException	**false**	有効にすると、エクスチェンジ がコンシューマー側で処理に失敗し、発生した例外が **application/x-java-serialized-object** コンテンツタイプとして応答でシリアライズされた場合に、例外がシリアライズされました。プロデューサー側では、例外がデシリアライズされ、**HttpOperationFailedException** ではなくそのまま出力されます。原因となった例外はシリアライズする必要があります。

urlDecodeHeaders	**false**	このオプションを有効にすると、Netty から Camel Message へのバインディング時に、ヘッダーの値は URL デコードされます（例：%20 は空白文字になります）。このオプションは、デフォルトの **org.apache.camel.component.netty.http.NettyHttpBinding** によって使用されるため、カスタム **org.apache.camel.component.netty.http.NettyHttpBinding** を実装する場合は、このオプションに応じてヘッダーをデコードする必要があります。
nettySharedHttpServer	**null**	共有 Netty HTTP サーバーを使用するには、以下を行います。詳細は Netty HTTP Server Exampleを参照してください。
disableStreamCache	**false**	Netty **HttpRequest#getContent** （）からの raw 入力ストリームがキャッシュされるかどうかを決定します(Camel はストリームを軽量メモリーベースの Stream キャッシュに読み取ります)。デフォルトでは、Camel は Netty 入力ストリームをキャッシュして複数回読み取りし、Camel がストリームからすべてのデータを取得できるようにします。ただし、ファイルや他の永続ストアに直接ストリーミングするなど、raw ストリームにアクセスする必要がある場合などにこのオプションを **true** に設定できます。このオプションを有効にすると、Netty ストリームが追加設定なしで複数回読み取ることができないため、Netty raw ストリームで reader インデックスを手動でリセットする必要があります。
securityConfiguration	**null**	**コンシューマーのみ**。セキュアな Web リソースを設定するための **org.apache.camel.component.netty.http.NettyHttpSecurityConfiguration** を参照します。

send503whenSuspended	**true**	**コンシューマーのみ**。コンシューマーが一時停止されたときに HTTP ステータスコード 503 を送信するかどうか。オプションが **false** の場合、コンシューマーが一時停止されると Netty Acceptor がバインド解除されるため、クライアントは接続できなくなります。
maxHeaderSize	**8192**	Camel 2.15.3 : **コンシューマーのみ**。すべてのヘッダーの最大長。各ヘッダーの長さの合計がこの値を超えると、**TooLongFrameException** が発生します。
okStatusCodeRange	**200-299**	Camel 2.16: 正常な応答と見なされるステータスコード。値は含まれます。範囲は、構文 ***from-to*** を使用して定義する必要があります。
useRelativePath	**false**	Camel 2.16 : **プロデューサーのみ**。リクエスト行または絶対 URI (**http://0.0.0.0:8080/myapp**)でパス(/**myapp**)を使用するかどうか（デフォルト）。

***NettyHttpSecurityConfiguration* には以下のオプションがあります。**

名前	デフォルト値	説明
認証	**true**	認証が有効であるかどうか。を使用して、この機能を迅速にオフにできます。
constraint	**Basic**	サポートされる制約。現在、**Basic** のみが実装され、サポートされています。
realm	**null**	JAAS セキュリティーレルムの名前。このオプションは必須です。
securityConstraint	**null**	ACL を Web リソースに定義できるセキュリティー制約マッパーをプラグインできるようにします。

securityAuthenticator	**null**	認証を実行するオーセンティケーターをプラグインできるようにします。設定されていない場合、**org.apache.camel.component.netty.http.JAASSecurityAuthenticator** はデフォルトで使用されます。
loginDeniedLoggingLevel	**DEBUG**	ログイン試行の失敗時に使用されるロギングレベル。これにより、ログインに失敗した理由の詳細を確認できます。
roleClassName	**null**	ユーザーロールを含む **Principal** 実装の FQN クラス名を指定するには、以下を実行します。何も指定されていない場合、Netty HTTP コンポーネントはデフォルトで、FQN クラス名に FQN クラス名に小文字の単語ロールがある場合に **Principal** が **ロール** ベースであると想定します。複数のクラス名をコンマで区切って指定できます。

メッセージヘッダー

以下のヘッダーをプロデューサーで使用して、HTTP リクエストを制御できます。

名前	タイプ	説明
CamelHttpMethod	**文字列**	GET、POST、TRACE など、使用する HTTP メソッドを制御できます。タイプは **org.jboss.netty.handler.codec.http.HttpMethod** インスタンスでも指定できます。
CamelHttpQuery	**文字列**	エンドポイント設定を上書きする **String** 値として URI クエリーパラメーターを指定できるようにします。& 記号を使用して、複数のパラメーターを区切ります。例： **foo=bar&beer=yes**

CamelHttpPath	文字列	Camel 2.13.1/2.12.4: エンドポイント設定をオーバーライドする **String** の値として URI コンテキストパスおよびクエリーパラメーターを指定できます。これにより、同じリモート HTTP サーバーを呼び出すために同じプロデューサーを再利用できますが、動的コンテキストパスおよびクエリーパラメーターを使用できます。
Content-Type	文字列	HTTP ボディーのコンテンツタイプを設定するには、以下を行います。例： **text/plain; charset="UTF-8"**。
CamelHttpResponseCode	int	使用する HTTP ステータスコードを設定できます。デフォルトでは、成功には 200、失敗には 500 が使用されます。

***Netty HTTP* エンドポイントからルートが起動すると、以下のヘッダーが meta-data として提供されます。**

テーブルの説明は、from ("netty-http:http:0.0.0.0:8080/myapp")... を持つルートのオフセットを取ります。

名前	タイプ	説明
CamelHttpMethod	文字列	GET、POST、TRACE などの HTTP メソッド。
CamelHttpUrl	文字列	プロトコル、ホストおよびポートなどの URL。
CamelHttpUri	文字列	プロトコル、ホストおよびポートのない URI (/myapp など)
CamelHttpQuery	文字列	**foo=bar&beer=yes** などのクエリーパラメーター

CamelHttpRawQuery	文字列	Camel 2.13.0: **foo=bar&beer=yes** などのクエリーパラメーター。コンシューマー(URL デコード前)に到達する際に、raw 形式で保存されます。
CamelHttpPath	文字列	追加のコンテキストパス。クライアントが context-path **/myapp** と呼ばれる場合、この値は空になります。クライアントが **/myapp/mystuff** を呼び出す場合、このヘッダー値は **/mystuff** になります。つまり、ルートエンドポイントに設定された context-path の後に値になります。
CamelHttpCharacterEncoding	文字列	content-type ヘッダーからの charset。
CamelHttpAuthentication	文字列	ユーザーが HTTP **Basic** を使用して認証された場合、このヘッダーは Basic の値で追加されます。
Content-Type	文字列	提供された場合はコンテンツタイプ。例： **text/plain; charset="UTF-8"**。

NETTY タイプへのアクセス

*このコンポーネントは、**org.apache.camel.component.netty.http.NettyHttpMessage** を Exchange のメッセージ実装として使用します。これにより、エンドユーザーは以下のように元の **Netty** 要求/応答インスタンスにアクセスできます。元の応答は常にアクセスできない可能性があることに注意してください。*

```
org.jboss.netty.handler.codec.http.HttpRequest request =
exchange.getIn(NettyHttpMessage.class).getHttpRequest();
```

例

*以下のルートでは Netty HTTP を **HTTP** サーバーとして使用し、ハードコーディングされた **Bye World** メッセージを返します。*

```
from("netty-http:http://0.0.0.0:8080/foo")
  .transform().constant("Bye World");
```

*また、以下のように ProducerTemplate で **Camel** を使用してこの **HTTP** サーバーを呼び出すことも*

できます。

```
String out = template.requestBody("netty-http:http://localhost:8080/foo", "Hello World", String.class);
System.out.println(out);
```

また、Bye World を出力として返します。

NETTY がワイルドカードと一致させる方法

デフォルトでは、Netty HTTP は正確な URI の場合にのみ一致します。ただし、接頭辞に一致するよう Netty に指示することができます。以下に例を示します。

```
from("netty-http:http://0.0.0.0:8123/foo").to("mock:foo");
```

上記のルートでは、Netty HTTP は URI が完全に一致する場合にのみ一致するため、http://0.0.0.0:8123/foo を入力し、http://0.0.0.0:8123/foo/bar を使用しない場合は一致しません。

そのため、ワイルドカードの一致を有効にするには、以下を実行します。

```
from("netty-http:http://0.0.0.0:8123/foo?matchOnUriPrefix=true").to("mock:foo");
```

そのため、Netty は foo で始まるすべてのエンドポイントに一致するようになりました。

任意のエンドポイントに一致させる には、以下を実行できます。

```
from("netty-http:http://0.0.0.0:8123?matchOnUriPrefix=true").to("mock:foo");
```

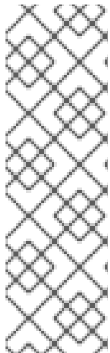
同じポートでの複数ルートの使用

同じ CamelContext で、同じポート(org.jboss.netty.bootstrap.ServerBootstrap インスタンスなど)を共有する Netty HTTP からの複数のルートを持つことができます。これを実行するには、ルートが同じ org.jboss.netty.bootstrap.ServerBootstrap インスタンスを共有するため、複数のブートストラップオプションをルートで同一にする必要があります。インスタンスは、最初に作成したルートからのオプションで設定されます。

ルートが同じでなければならないオプション

は、org.apache.camel.component.netty.NettyServerBootstrapConfiguration 設定クラスで定義されたすべてのオプションです。別のオプションで別のルートを設定した場合、Camel は起動時に例外を出力し、オプションが同一ではないことを示します。これを軽減するには、すべてのオプションが同一であることを確認します。

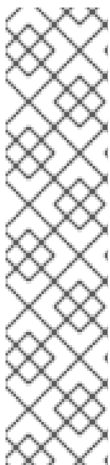
以下は、同じポートを共有する 2 つのルートを持つ例です。



同じポートを共有する 2 つのルート

```
from("netty-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

from("netty-http:http://0.0.0.0:{{port}}/bar")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

以下は、1st ルートと同じ org.apache.camel.component.netty.NettyServerBootstrapConfiguration オプションを持たない、誤って設定された 2 番目のルートの例です。これにより、起動時に Camel が失敗します。



2 つのルートが同じポートを共有しますが、2 番目のルートは設定が間違っているため、開始時に失敗します。

```
from("netty-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

// we cannot have a 2nd route on same port with SSL enabled, when the 1st
route is NOT
from("netty-http:http://0.0.0.0:{{port}}/bar?ssl=true")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

複数のルートを持つ同じサーバーブートストラップ設定の再利用

org.apache.camel.component.netty.NettyServerBootstrapConfiguration タイプの単一のインスタンスに共通のサーバーブートストラップオプションを設定すると、Netty HTTP コンシューマーで bootstrapConfiguration オプションを使用して、すべてのコンシューマーで同じオプションを参照および再利用できます。

```
<bean id="nettyHttpBootstrapOptions"
class="org.apache.camel.component.netty.NettyServerBootstrapConfiguration">
  <property name="backlog" value="200"/>
```

```
  <property name="connectTimeout" value="20000"/>
  <property name="workerCount" value="16"/>
</bean>
```

以下のように、このオプションを参照するルートで、

```
<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/foo?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/bar?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/beer?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>
```

OSGI コンテナー内の複数のバンドル間で複数のルートを持つ同じサーバーブートストラップ設定の再利用

詳細は、Netty HTTP Server Example を参照してください。

HTTP BASIC 認証の使用

Netty HTTP コンシューマーは、以下のように使用するセキュリティーレルム名を指定して HTTP Basic 認証をサポートします。

```
<route>
   <from uri="netty-http:http://0.0.0.0:{{port}}/foo?securityConfiguration.realm=karaf"/>
   ...
</route>
```

Basic 認証を有効にするには、レルム名が必須です。デフォルトでは、JAAS ベースのオーセンティケーターが使用されます。これは、指定されたレルム名（上記の例では karaf）を使用し、認証に JAAS レルムと JAAS LoginModuleを使用します。

Apache Karaf / ServiceMix のエンドユーザーには、追加設定なしで karaf レルムがあるため、上記

の例はこれらのコンテナーのすぐに機能しなくなります。

WEB リソースでの ACL の指定

org.apache.camel.component.netty.http.SecurityConstraint を使用すると、Web リソースに制限を定義できます。また、org.apache.camel.component.netty.http.SecurityConstraintMapping は追加設定なしで提供され、ロールへの包含と除外を簡単に定義できます。

たとえば、XML DSL で以下のように制約 Bean を定義します。

```
<bean id="constraint"
class="org.apache.camel.component.netty.http.SecurityConstraintMapping">
  <!-- inclusions defines url -> roles restrictions -->
  <!-- a * should be used for any role accepted (or even no roles) -->
  <property name="inclusions">
    <map>
      <entry key="/*" value="*"/>
      <entry key="/admin/*" value="admin"/>
      <entry key="/guest/*" value="admin,guest"/>
    </map>
  </property>
  <!-- exclusions is used to define public urls, which requires no authentication -->
  <property name="exclusions">
    <set>
      <value>/public/*</value>
    </set>
  </property>
</bean>
```

上記の制約は、次のように を定義します。

- */* へのアクセスは制限され、すべてのロールが受け入れられます（ユーザーにもロールがない場合も同様です）。*
- */admin/* へのアクセスには admin ロールが必要です。*
- */guest/* へのアクセスには、admin ロールまたは guest ロールが必要です。*
- */public/* へのアクセスは、認証が不要であることを意味します。そのため、ログインせずに誰でも公開されます。*

この制約を使用するには、以下のように Bean ID を参照する必要があります。

```
<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/foo?
matchOnUriPrefix=true&ecurityConfiguration.realm=karaf&ecurityConfiguration.securityCons
traint=#constraint"/>
  ...
</route>
```

- *Netty*
- *Netty HTTP サーバーの例*
- *Jetty*

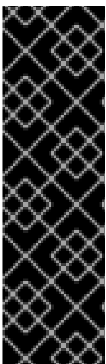
第120章 NETTY4-HTTP

NETTY4 HTTP コンポーネント

Camel 2.14 から利用可能

netty4-http コンポーネントは Netty4 コンポーネントへの拡張であり、Netty4 で HTTP トランスポートを容易にします。

この Camel コンポーネントは、プロデューサーとコンシューマーエンドポイントの両方をサポートします。



ストリーム

Netty はストリームベースであり、受信する入力がストリームとして Camel に送信されることを意味します。つまり、ストリームのコンテンツを 1 度 だけ読み取ることができます。メッセージボディーが空である、またはデータに複数回アクセスする必要がある場合（例：マルチキャストの実行、再配信エラー処理）は、Stream キャッシュ を使用するか、メッセージボディーを複数回再読み取りできる String に変換する必要があります。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty4-http</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

netty コンポーネントの URI スキームは以下のとおりです。

```
netty4-http:http://localhost:8080[?options]
```

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

クエリーパラメーターとエンドポイントオプションの比較

Camel が URI クエリーパラメーターおよびエンドポイントオプションを認識する方法に気づくかもしれません。たとえば、netty4-http:http//example.com?myParam=myValue&compression=true のようにエンドポイント URI を作成できます。この例では、myParam は HTTP パラメーターですが、compression は Camel エンドポイントオプションです。このような状況で Camel によって使用されるストラテジーは、利用可能なエンドポイントオプションを解決し、URI から削除することです。これは、前述の例では、Netty HTTP プロデューサーによってエンドポイントに送信される HTTP リクエストは、http//example.com?myParam=myValue のようになります。compression エンドポイントオプションはターゲット URL から解決され、削除されます。

また、動的ヘッダー(CamelHttpQueryなど)を使用してエンドポイントオプションを指定できないことに注意してください。エンドポイントオプションは、エンドポイント URI 定義レベルでのみ指定できます(to または from DSL 要素など)。

HTTP オプション

その他のオプション

重要: このコンポーネントは、Netty4 からすべてのオプションを継承します。そのため、Netty4 のドキュメントも確認してください。Netty4 の一部のオプションは、UDP トランスポートに関連するオプションなど、この Netty4 HTTP コンポーネントを使用する場合に適用されないことに注意してください。

名前	デフォルト値	説明
chunkedMaxContentLength	**1mb**	Netty HTTP サーバーで受信されるチャンクごとの最大コンテンツ長（バイト単位）。
compression	**false**	クライアントが HTTP ヘッダーからサポートしている場合、Netty HTTP サーバーの圧縮に gzip/deflate の使用を許可します。
headerFilterStrategy		カスタム **org.apache.camel.spi.HeaderFilterStrategy** を使用してヘッダーをフィルターリングします。

httpMethodRestrict		Netty HTTP コンシューマーで HTTP メソッドを無効にするには、以下を行います。コンマで区切って複数指定できます。
mapHeaders	**true**	このオプションが有効な場合、Netty から Camel Message へのバインド中にヘッダーもマッピングされます（ヘッダーが Camel Message に追加されます）。このオプションをオフにして無効にすることができます。ヘッダーには、**org.apache.camel.component.netty4.http.NettyHttpMessage** メッセージから引き続きアクセスできます。このメソッドは **getHttpRequest()** で Netty HTTP リクエスト **io.netty.handler.codec.http.HttpRequest** インスタンスを返します。
matchOnUriPrefix	**false**	完全一致が見つからない場合に Camel が URI 接頭辞と一致することでターゲットコンシューマーの検索を試みるかどうか。詳細については以下をご覧ください
nettyHttpBinding		Netty および Camel Message API との間でバインドするためにカスタム **org.apache.camel.component.netty4.http.NettyHttpBinding** を使用します。
bridgeEndpoint	**false**	オプションが **true** の場合、プロデューサーは **Exchange.HTTP_URI** ヘッダーを無視し、リクエストにエンドポイントの URI を使用します。また、**throwExceptionOnFailure** を **false** に設定して、プロデューサーがすべての障害応答を返信するようにすることもできます。
throwExceptionOnFailure	**true**	リモートサーバーからの応答に失敗した場合、**HttpOperationFailedException** の出力を無効にするオプション。これにより、HTTP ステータスコードに関するすべての応答を取得できます。
traceEnabled	**false**	この Netty HTTP コンシューマーに対して HTTP TRACE を有効にするかどうかを指定します。デフォルトでは、TRACE はオフになっています。

transferException	**false**	有効にすると、エクスチェンジ がコンシューマー側で処理に失敗し、発生した例外が応答で **application/x-java-serialized-object** コンテンツタイプとしてシリアライズされた場合は、以下を行います。プロデューサー側では、**HttpOperationFailedException** ではなく、例外がデシリアライズされ、そのまま出力されます。原因となった例外はシリアライズする必要があります。
urlDecodeHeaders		このオプションを有効にすると、Netty から Camel Message へのバインド中にヘッダーの値は URL デコードされます（例：%20 は空白文字になります）。このオプションはデフォルトの **org.apache.camel.component.netty4.http.NettyHttpBinding** で使用されているため、カスタム **org.apache.camel.component.netty4.http.NettyHttpBinding** を実装する場合は、このオプションに応じてヘッダーをデコードする必要があります。**注記：** このオプションはデフォルト **false** です。
nettySharedHttpServer	**null**	共有 Netty4 HTTP サーバーを使用するには、以下を行います。詳細は Netty HTTP Server Example を参照してください。
disableStreamCache	**false**	Netty **HttpRequest#getContent()** からの raw 入力ストリームがキャッシュされるかどうかを決定します (Camel はストリームを軽量メモリーベースの Stream キャッシュ) キャッシュに読み取ります。デフォルトでは、Camel は Netty 入力ストリームをキャッシュして複数回読み取りし、Camel がストリームからすべてのデータを取得できるようにします。ただし、このオプションを **true** に設定することができます。たとえば、ファイルや他の永続ストアに直接ストリーミングするなど、raw ストリームにアクセスする必要がある場合などです。このオプションを有効にすると、Netty ストリームが追加設定なしで複数回読み取ることができないため、Netty raw ストリームで reader インデックスを手動でリセットする必要があります。

securityConfiguration	**null**	**コンシューマーのみ**。セキュアな Web リソースを設定するには、**org.apache.camel.component.netty4.http.NettyHttpSecurityConfiguration** を参照してください。
send503whenSuspended	**true**	**コンシューマーのみ**。コンシューマーが一時停止されたときに HTTP ステータスコード 503 を送信するかどうか。オプションが **false** の場合、コンシューマーが一時停止されると Netty Acceptor がバインド解除されるため、クライアントは接続できなくなります。
maxHeaderSize	**8192**	Camel 2.15.3 : **コンシューマーのみ**。すべてのヘッダーの最大長。各ヘッダーの長さの合計がこの値を超えると、**TooLongFrameException** が発生します。
okStatusCodeRange	**200-299**	Camel 2.16: 正常な応答と見なされるステータスコード。値は含まれます。範囲は、構文 ***from-to*** を使用して定義する必要があります。
useRelativePath	**false**	Camel 2.16 : **プロデューサーのみ**。リクエスト行または絶対 URI (**http://0.0.0.0:8080/myapp**)でパス(/**myapp**)を使用するかどうか（デフォルト）。

NettyHttpSecurityConfiguration *には以下のオプションがあります。*

名前	デフォルト値	説明
authenticate	**true**	認証が有効であるかどうか。を使用して、この機能を迅速にオフにできます。
constraint	**Basic**	サポートされる制約。現在、**Basic** のみが実装され、サポートされています。
realm	**null**	JAAS セキュリティーレルムの名前。このオプションは必須です。

securityConstraint	**null**	ACL を Web リソースに定義できるセキュリティー制約マッパーをプラグインできるようにします。
securityAuthenticator	**null**	認証を実行するオーセンティケーターをプラグインできるようにします。何も設定されていない場合は、**org.apache.camel.component.netty4.http.JAASSecurityAuthenticator** がデフォルトで使用されます。
loginDeniedLoggingLevel	**DEBUG**	ログイン試行の失敗時に使用されるロギングレベル。これにより、ログインに失敗した理由の詳細を確認できます。
roleClassName	**null**	ユーザーロールを含む **Principal** 実装の FQN クラス名を指定するには、以下を実行します。指定がない場合は、Netty4 HTTP コンポーネントはデフォルトで **Principal** を、FQN クラス名に **role** の小文字の という単語が小文字の場合、ロールベースであると想定します。複数のクラス名をコンマで区切って指定できます。

メッセージヘッダー

以下のヘッダーをプロデューサーで使用して、HTTP リクエストを制御できます。

名前	タイプ	説明
CamelHttpMethod	**String**	GET、POST、TRACE など、使用する HTTP メソッドを制御できます。タイプは **io.netty.handler.codec.http.HttpMethod** インスタンスにも指定できます。
CamelHttpQuery	**String**	エンドポイント設定を上書きする **String** の値として URI クエリーパラメーターを提供できるようにします。& 記号を使用して、複数のパラメーターを区切ります。例： **foo=bar&beer=yes**

CamelHttpPath	**String**	エンドポイント設定を上書きする **String** の値として URI コンテキストパスおよびクエリーパラメーターを指定できます。これにより、同じリモート http サーバーを呼び出すために同じプロデューサーを再利用できますが、動的コンテキストパスおよびクエリーパラメーターを使用できます。
Content-Type	**String**	HTTP ボディーのコンテンツタイプを設定するには、以下を行います。例： **text/plain; charset="UTF-8"**
CamelHttpResponseCode	**int**	使用する HTTP ステータスコードを設定できます。デフォルトでは、成功には 200、失敗には 500 が使用されます。

Netty4 HTTP エンドポイントからルートが起動すると、以下のヘッダーが meta-data として提供されます。

テーブルの説明は、以下を持つルート内のオフセットを取ります。 from("netty4-http:http:0.0.0.0:8080/myapp")...

名前	タイプ	説明
CamelHttpMethod	**String**	GET、POST、TRACE などの HTTP メソッド。
CamelHttpUrl	**String**	プロトコル、ホストおよびポートなどの URL。 http://0.0.0.0:8080/myapp
CamelHttpUri	**String**	プロトコル、ホストおよびポートなどの URI のない URI。 /myapp
CamelHttpQuery	**String**	などのクエリーパラメーター **foo=bar&beer=yes**
CamelHttpRawQuery	**String**	クエリーパラメーター(**foo=bar&beer=yes** など)コンシューマー(URL デコード前)に到達する際に、raw 形式で保存されます。

CamelHttpPath	**String**	追加のコンテキストパス。クライアントが context-path **/myapp** と呼ばれると、この値は空になります。クライアントが **/myapp/mystuff** を呼び出す場合、このヘッダーの値は **/mystuff** になります。つまり、ルートエンドポイントに設定された context-path の後に値になります。
CamelHttpCharacterEncoding	**String**	content-type ヘッダーからの charset。
CamelHttpAuthentication	**String**	ユーザーが HTTP Basic を使用して認証された場合、このヘッダーは **Basic** の値で追加されます。
Content-Type	**String**	提供された場合はコンテンツタイプ。例： **text/plain; charset="UTF-8"**

NETTY タイプへのアクセス

*このコンポーネントは、**org.apache.camel.component.netty4.http.NettyHttpMessage** をエクスチェンジのメッセージ実装として使用し ます。これにより、エンドユーザーは以下のように元の **Netty** 要求/応答インスタンスにアクセスできます。元の応答は常にアクセスできない可能性があることに注意してください。*

```
io.netty.handler.codec.http.HttpRequest request =
exchange.getIn(NettyHttpMessage.class).getHttpRequest();
```

例

*以下のルートでは **Netty4 HTTP** を **HTTP** サーバーとして使用し、ハードコーディングされた **Bye World** メッセージを返します。*

```
    from("netty4-http:http://0.0.0.0:8080/foo")
      .transform().constant("Bye World");
```

*また、以下のように **ProducerTemplate** で **Camel** を使用してこの **HTTP** サーバーを呼び出すこともできます。*

```
String out = template.requestBody("netty4-http:http://localhost:8080/foo", "Hello World",
String.class);
System.out.println(out);
```

また、Bye World を出力として返します。

NETTY がワイルドカードと一致させる方法

デフォルトでは Netty4 HTTP は正確な URI の場合にのみ一致します。ただし、接頭辞に一致するよう Netty に指示することができます。以下に例を示します。

```
from("netty4-http:http://0.0.0.0:8123/foo").to("mock:foo");
```

上記のルートでは、Netty4 HTTP は URI が完全に一致する場合にのみ一致するため、http://0.0.0.0:8123/foo を入力すると一致しますが、http://0.0.0.0:8123/foo/bar を使用しない場合は一致しません。

そのため、ワイルドカードの一致を有効にするには、以下を実行します。

```
from("netty4-http:http://0.0.0.0:8123/foo?matchOnUriPrefix=true").to("mock:foo");
```

そのため、Netty は foo で始まるすべてのエンドポイントに一致するようになりました。

任意のエンドポイントに一致させる には、以下を実行できます。

```
from("netty4-http:http://0.0.0.0:8123?matchOnUriPrefix=true").to("mock:foo");
```

同じポートでの複数ルートの使用

同じ CamelContext で、同じポート（例： io.netty.bootstrap.ServerBootstrap インスタンス）を共有する Netty4 HTTP からの複数のルートを持つことができます。これを実行するには、ルートが同じ io.netty.bootstrap.ServerBootstrap インスタンスを共有するため、複数のブートストラップオプションをルートで同一でなければなりません。インスタンスは、最初に作成したルートからのオプションで設定されます。

ルートが同じでなければならないオプションは、org.apache.camel.component.netty4.NettyServerBootstrapConfiguration 設定クラスで定義されているすべてのオプションです。別のオプションで別のルートを設定した場合、Camel は起動時に例

外を出力し、オプションが同一ではないことを示します。これを軽減するには、すべてのオプションが同一であることを確認します。

以下は、同じポートを共有する 2 つのルートを持つ例です。

例120.1 同じポートを共有する 2 つのルート

```
from("netty4-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

from("netty4-http:http://0.0.0.0:{{port}}/bar")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

以下は、1st ルートと同じ org.apache.camel.component.netty4.NettyServerBootstrapConfiguration オプションがない、誤って設定された 2 番目のルートの例です。これにより、起動時に Camel が失敗します。

例120.2 2 つのルートが同じポートを共有しますが、2 番目のルートは設定が間違っているため、開始時に失敗します。

```
from("netty4-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

// we cannot have a 2nd route on same port with SSL enabled, when the 1st route is NOT
from("netty4-http:http://0.0.0.0:{{port}}/bar?ssl=true")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

複数のルートを持つ同じサーバーブートストラップ設定の再利用

org.apache.camel.component.netty4.NettyServerBootstrapConfiguration タイプの単一のインスタンスに共通のサーバーブートストラップオプションを設定することで、Netty4 HTTP コンシューマーで bootstrapConfiguration オプションを使用して、すべてのコンシューマーで同じオプションを参照および再利用できます。

```
<bean id="nettyHttpBootstrapOptions"
class="org.apache.camel.component.netty4.NettyServerBootstrapConfiguration">
  <property name="backlog" value="200"/>
  <property name="connectionTimeout" value="20000"/>
  <property name="workerCount" value="16"/>
</bean>
```

以下のように、このオプションを参照するルートで、

```
<route>
  <from uri="netty4-http:http://0.0.0.0:{{port}}/foo?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty4-http:http://0.0.0.0:{{port}}/bar?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty4-http:http://0.0.0.0:{{port}}/beer?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>
```

OSGI コンテナー内の複数のバンドル間で複数のルートを持つ同じサーバーブートストラップ設定の再利用

詳細は、Netty HTTP Server Example を参照してください。

HTTP BASIC 認証の使用

Netty HTTP コンシューマーは、以下のように使用するセキュリティーレルム名を指定して HTTP Basic 認証をサポートします。

```
<route>
   <from uri="netty4-http:http://0.0.0.0:{{port}}/foo?securityConfiguration.realm=karaf"/>
   ...
</route>
```

Basic 認証を有効にするには、レルム名が必須です。デフォルトでは、JAAS ベースのオーセンティケーターが使用されます。これは、指定されたレルム名（上記の例ではkaraf）を使用し、認証に JAAS レルムと JAAS {{LoginModule}}s を使用します。

Apache Karaf / ServiceMix のエンドユーザーには、追加設定なしで karaf レルムがあるため、上記の例はこれらのコンテナーのすぐに機能しなくなります。

WEB リソースでの ACL の指定

org.apache.camel.component.netty4.http.SecurityConstraint では、Web リソースに制限を定義できます。また、org.apache.camel.component.netty.http.SecurityConstraintMapping はすぐに使用できるので、ロールで包含と除外を簡単に定義できます。

たとえば、XML DSL で以下のように制約 Bean を定義します。

```
<bean id="constraint" class="org.apache.camel.component.netty4.http.SecurityConstraintMapping">
  <!-- inclusions defines url -> roles restrictions -->
  <!-- a * should be used for any role accepted (or even no roles) -->
  <property name="inclusions">
    <map>
      <entry key="/*" value="*"/>
      <entry key="/admin/*" value="admin"/>
      <entry key="/guest/*" value="admin,guest"/>
    </map>
  </property>
  <!-- exclusions is used to define public urls, which requires no authentication -->
  <property name="exclusions">
    <set>
      <value>/public/*</value>
    </set>
  </property>
</bean>
```

上記の制約は、次のように を定義します。

- ***/* へのアクセスは制限され、すべてのロールが受け入れられます（ユーザーにもロールがない場合も同様です）。***

- ***/admin/* へのアクセスには admin ロールが必要です。***

- ***/guest/* へのアクセスには、admin ロールまたは guest ロールが必要です。***

- ***/public/* へのアクセスは、認証が不要であることを意味します。そのため、ログインせずに誰でも公開されます。***

この制約を使用するには、以下のように Bean ID を参照する必要があります。

```
<route>
  <from uri="netty4-http:http://0.0.0.0:{{port}}/foo?
matchOnUriPrefix=true&amp;securityConfiguration.realm=karaf&amp;securityConfiguration.securityCon
straint=#constraint"/>
  ...
</route>
```

第121章 OLINGO2

OLINGO2 コンポーネント

Camel 2.14 から利用可能

Olingo2 コンポーネントは Apache Olingo バージョン 2.0 API を使用して OData 2.0 準拠のサービスと対話します。数多くの一般的な商用およびエンタープライズベンダーおよび製品が OData プロトコルをサポートします。サポートする製品のサンプルリストは、OData の Web サイト を参照してください。

Olingo2 コンポーネントは、カスタムおよび OData システムクエリーパラメーターを使用したフィード、デルタフィード、エンティティー、単純および複雑なプロパティー、リンク、カウントをサポートします。エンティティー、プロパティー、および関連リンクの更新をサポートします。また、単一の OData バッチ操作としてクエリーおよび変更リクエストの送信もサポートします。

コンポーネントは、OData サービスコネクションの HTTP 接続パラメーターおよびヘッダーの設定をサポートします。これにより、ターゲット OData サービスによって必要に応じて SSL、OAuth2.0 などを使用できます。

TLS (Transport Layer Security)を使用するように camel-olingo2 コンポーネントを設定するには、Security Guide の Configuring Transport Security for Camel Components の章を参照してください。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-olingo2</artifactId>
  <version>${camel-version}</version>
</dependency>
```

URI 形式

```
olingo2://endpoint/<resource-path>?[options]
```

OLINGO2COMPONENT

Olingo2 コンポーネントは、以下のオプションで設定できます。これらのオプション

*は、**org.apache.camel.component.olingo2.Olingo2Configuration** タイプのコンポーネントの **Bean** プロパティー **configuration** を使用して提供できます。*

オプション	タイプ	説明
serviceUri	文字列	ターゲット OData サービスベース URI（例：http://services.odata.org/OData/OData.svc）
contentType	文字列	Content-Type ヘッダーの値を使用して JSON または XML メッセージ形式を指定できます。デフォルトは **application/json;charset=utf-8**です。
connectTimeout	int	HTTP 接続作成のタイムアウト（ミリ秒単位）。デフォルトは 30,000 (30 秒)です。
socketTimeout	int	HTTP 要求のタイムアウト（ミリ秒単位）。デフォルトは 30,000 (30 秒)です。
httpHeaders	java.util.Map<String, String>	すべてのリクエストに挿入するカスタム HTTP ヘッダー。これには OAuth トークンが含まれる場合があります。
proxy	org.apache.http.HttpHost	HTTP プロキシーサーバーの設定
sslContext	javax.net.ssl.SSLContext	HTTP SSL 設定
httpAsyncClientBuilder	org.apache.http.impl.nio.client.HttpAsyncClientBuilder	より複雑な HTTP クライアント設定のカスタム HTTP 非同期クライアントビルダーは、connectionTimeout、socketTimeout、proxy、および sslContext を上書きします。socketTimeout はビルダーで指定する **必要** があります。指定しないと、OData リクエストが永久にブロックされる可能性があることに注意してください。

プロデューサーエンドポイント

*プロデューサーエンドポイントは、次に一覧表示されるエンドポイント名とオプションを使用できます。プロデューサーエンドポイントは、**Camel Exchange In** メッセージに含まれる値を持つ **endpoint** オプションの名前が含まれる必要がある特別なオプション **inBody** を使用することもできます。**inBody** オプションのデフォルトは、このオプションを取るエンドポイントの **data** に設定されます。*

*エンドポイントオプションは、エンドポイント URI またはメッセージヘッダーで動的に指定できます。メッセージヘッダー名は **CamelOlingo2.<option>** の形式で指定する必要があります。**inBody** オプションはメッセージヘッダーを上書きすることに注意してください。つまり、エンドポイントオプション **inBody=option** は **CamelOlingo2.option** ヘッダーを上書きすることに注意してください。さらに、クエリーパラメーターを指定することもできます。*

***resourcePath** オプションは、URI パスの一部として URI に指定されるか、エンドポイントオプション **?resourcePath=<resource-path>** として指定するか、ヘッダー値 **CamelOlingo2.resourcePath** として指定できることに注意してください。**OData** エンティティーキー述語はリソースパスの一部にすることができます（例： **Manufacturer s ('1')**、**'1 '** はキー述語です。または、リソースパス **Manufacturers** および **keyPredicate** オプション **'1'** で個別に指定できます。*

エンドポイント	オプション	HTTP メソッド	結果ボディーのタイプ
batch	data	**POST with multipart/mixed batch request**	**java.util.List<org.apache.camel.component.olingo2.api.batch.Olingo2BatchResponse>**
create	data、resourcePath	**POST**	**org.apache.olingo.odata2.api.ep.entry.ODataEntry for new entries org.apache.olingo.odata2.api.commons.HttpStatusCodes for other OData resources**
delete	resourcePath	**DELETE**	**org.apache.olingo.odata2.api.commons.HttpStatusCodes**
merge	data、resourcePath	**MERGE**	**org.apache.olingo.odata2.api.commons.HttpStatusCodes**
patch	data、resourcePath	**PATCH**	**org.apache.olingo.odata2.api.commons.HttpStatusCodes**
read	queryParams, resourcePath	**GET**	**Depends on OData resource being queried as described next**

エンドポイント	オプション	HTTP メソッド	結果ボディーのタイプ
update	data、resourcePath	**PUT**	**org.apache.olingo.odata2.api.commons.HttpStatusCodes**

ODATA リソースタイプマッピング

データオプションの 読み取り エンドポイントおよびデータタイプの結果は、クエリー、作成、または変更を行う OData リソースによって異なります。

OData リソースタイプ	resourcePath および keyPredicate からのリソース URI	in または Out Body タイプ
エンティティーデータモデル	$metadata	**org.apache.olingo.odata2.api.edm.Edm**
サービスドキュメント	/	**org.apache.olingo.odata2.api.servicedocument.ServiceDocument**
OData フィード	<entity-set>	**org.apache.olingo.odata2.api.ep.feed.ODataFeed**
OData エントリー	<entity-set>(<key-predicate>)	**org.apache.olingo.odata2.api.ep.entry.ODataEntry** Out body （応答）の場合、**java.util.Map<String, Object>** は In body （要求）です。
単純なプロパティー	<entity-set>(<key-predicate>)/<simple-property>	**Appropriate Java data type as described by <link xlink:href="http://olingo.apache.org/javadoc/odata2/index.html?org/apache/olingo/odata2/api/edm/class-use/EdmProperty.html">Olingo EdmProperty</link>**

OData リソースタイプ	resourcePath および keyPredicate からのリソース URI	in または Out Body タイプ
単純なプロパティー値	<entity-set>(<key-predicate>)/<simple-property>/$value	**Appropriate Java data type as described by <link xlink:href="http://olingo.apache.org/javadoc/odata2/index.html?org/apache/olingo/odata2/api/edm/class-use/EdmProperty.html" >Olingo EdmProperty</link>**
複雑なプロパティー	<entity-set>(<key-predicate>)/<complex-property>	**java.util.Map<String, Object>**
ゼロまたは 1 つの関連付けリンク	<entity-set>(<key-predicate>/$link/<one-to-one-entity-set-property>	**String for response java.util.Map<String, Object> with key property names and values for request**
リンクの 0 個または多数	<entity-set>(<key-predicate>/$link/<one-to-many-entity-set-property>	**java.util.List<String> for response java.util.List<java.util.Map<String, Object>> containing list of key property names and values for request**
Count	<resource-uri>/$count	**java.lang.Long**

URI オプション

エンドポイント URI またはメッセージヘッダーのいずれかで queryParams に値が指定されていない場合、これは null であると想定されます。null 値は、他のオプションが一致するエンドポイントを満たさない場合にのみ使用されることに注意してください。

名前	タイプ	説明
data	**Object**	OData リソースの作成または変更に使用される適切なタイプを持つデータ
keyPredicate	**String**	パラメーター化された OData リソースエンドポイントを作成するためのキー述語。キー述語の値がヘッダーに動的に提供される作成/更新操作に役立ちます。

名前	タイプ	説明
queryParams	**java.util.Map<String, String>**	OData システムオプションおよびカスタムクエリーオプション。詳細は OData 2.0 URI Conventions を参照してください。
resourcePath	**String**	OData リソースパス（キー述語が含まれるか、または含まれない場合あり）
*	**String**	その他の URI オプションはクエリーパラメーターとして扱われ、クエリーパラメーターマップに追加され、**queryParams** オプションのエントリーを上書きします（これも指定されている場合）。

コンシューマーエンドポイント

*コンシューマーエンドポイントとして使用できる **read** エンドポイントのみです。コンシューマーエンドポイントは、**consumer.** 接頭辞が付いた **Scheduled Poll Consumer オプション** を使用して、エンドポイントの呼び出しをスケジュールできます。デフォルトでは、配列またはコレクションを返すコンシューマーエンドポイントは、要素ごとにエクスチェンジを 1 つ生成し、それらのルートはエクスチェンジごとに 1 回実行されます。この動作は、エンドポイントプロパティー **consumer.splitResult=false** を設定して無効にできます。*

メッセージヘッダー

*URI オプションは、**CamelOlingo2.** 接頭辞が付いたプロデューサーエンドポイントのメッセージヘッダーで指定できます。*

メッセージボディー

*すべての結果メッセージ本文は、Olingo2Component によって使用される基盤となる **Apache Olingo 2.0 API** によって提供されるオブジェクトを使用します。プロデューサーエンドポイントは、**inBody** エンドポイント URI パラメーターに受信メッセージボディーのオプション名を指定できます。配列またはコレクションを返すエンドポイントでは、**consumer.splitResult** が **false** に設定されていない限り、コンシューマーエンドポイントはすべての要素を個別のメッセージにマップします。*

ユースケース

以下のルートは、Mufacturer フィードから上位 5 つのエントリーを昇順で読み取ります。

```
from("direct:...")
    .setHeader("CamelOlingo2.$top","5")
    .to("olingo2://read/Manufacturers?orderBy=Name%20asc");
```

以下のルートは、受信 id ヘッダーの key プロパティーの値を使用して Manufacturer エントリーを読み取ります。

```
from("direct:...")
    .setHeader("CamelOlingo2.keyPredicate", header("id"))
    .to("olingo2://read/Manufacturers");
```

以下のルートは、ボディーメッセージの java.util.Map<String, Object> を使用して Manufacturer エントリーを作成します。

```
from("direct:...")
    .to("olingo2://create/Manufacturers");
```

以下のルートは、30 秒ごとに Manufacturer デルタフィード をポーリングします。Bean blah は Bean paramsBean を更新し、ODataDeltaFeed の結果で返された値で更新された !deltatoken プロパティーを追加します。最初のデルタトークンが不明なため、コンシューマーエンドポイントは最初に ODataFeed 値を生成し、後続のポーリングで ODataDeltaFeed を生成します。

```
from("olingo2://read/Manufacturers?
queryParams=#paramsBean&consumer.timeUnit=SECONDS&consumer.delay=30")
    .to("bean:blah");
```

第122章 OPENSHIFT

OPENSHIFT コンポーネント

Camel 2.14 から利用可能

openshift コンポーネントは、OpenShift アプリケーションを管理するためのコンポーネントです。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-openshift</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
openshift:clientId[?options]
```

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

オプション

名前	デフォルト値	説明
domain	null	ドメイン名。指定のない場合は、デフォルトのドメインが使用されます。
username		**必須**: openshift サーバーにログインするためのユーザー名。
password		**必須：** openshift サーバーにログインするためのパスワード。

server		openshift サーバーへの URL。指定しない場合、ローカルの openshift 設定ファイル **~/.openshift/express.conf** からのデフォルト値が使用されます。それも失敗する場合は、openshift.redhat.com が使用されます。
delay	**10s**	**コンシューマーのみ：** アプリケーションの状態変更をポーリングする頻度。デフォルトでは、10 秒ごとにポーリングを行います。

operation	**list**	**プロデューサーのみ**: 実行する操作： **list**、**start**、**stop**、**restart**、および ----------|----- などです。**statelist** 操作は、すべてのアプリケーションに関する情報を json 形式で返します。**state** 操作は、started、stopped などの状態を返します。他の操作は値を返しません。 Camel 2.16: 以下の操作を追加します。 ● **getStandaloneCartridge** スタンドアロンカートリッジの表示名を返します。 ● **getEmbeddedCartridges** インストールされた埋め込みカートリッジの一覧を返します。 ● **addEmbeddedCartridge** 操作は、ヘッダー **CamelOpenShiftEmbeddedCartridgeName** のインターフェイス **IEmbeddableCartridge** で名前を指定して、埋め込み可能なカートリッジの最新バージョンを追加し、カートリッジの表示名を返します。 ● **removeEmbeddedCartridge** 操作は、ヘッダー **CamelOpenShiftEmbeddedCartridgeName** のインターフェイス **IEmbeddableCartridge** でその名前を指定し、カートリッジの表示名を返すことで、埋め込みカートリッジを削除します。
application		**プロデューサーのみ**: アプリケーション名は **start**、**stop**、**restart**、または **state** を取得します。
mode		**プロデューサーのみ**: メッセージボディーを pojo または json として出力するかどうか。pojo の場合、メッセージは **List<com.openshift.client.IApplication>** タイプです。

例

全アプリケーションの一覧表示

```
// sending route
from("direct:apps")
   .to("openshift:myClient?username=foo&password=secret&operation=list");
   .to("log:apps");
```

この場合、すべてのアプリケーションに関する情報が pojo として返されます。json 応答が必要な場合は、mode=json を設定します。

アプリケーションの停止

```
// stopping the foobar application
from("direct:control")
   .to("openshift:myClient?
username=foo&password=secret&operation=stop&application=foobar");
```

上記の例では、foobar という名前のアプリケーションを停止します。

ギア状態変更のポーリング

コンシューマーは、ギアの状態の変更をポーリングするために使用されます。たとえば、新規ギアの追加/削除、またはライフサイクルの変更時（起動、停止など）などです。

```
// trigger when state changes on our gears
from("openshift:myClient?username=foo&password=secret&delay=30s")
   .log("Event ${header.CamelOpenShiftEventType} on application ${body.name} changed
state to ${header.CamelOpenShiftEventNewState}");
```

コンシューマーがエクスチェンジを出力すると、本文には com.openshift.client.IApplication がメッセージボディーとして含まれます。また、以下のヘッダーが含まれます。

ヘッダー	null にすることができます	説明

ヘッダー	null にすることができます	説明
CamelOpenShiftEventType	いいえ	イベントのタイプは、追加、削除、または変更のいずれかになります。
CamelOpenShiftEventOldState	はい	イベントタイプが変更される場合の古い状態。
CamelOpenShiftEventNewState	いいえ	イベントタイプの新しい状態

第123章 PAHO

PAHO コンポーネント

Paho コンポーネントは、Eclipse Paho ライブラリーを使用して MQTT メッセージングプロトコルのコネクターを提供します。Paho は最も人気の高い MQTT ライブラリーの 1 つであるため、Java プロジェクトと統合する場合は Camel Paho コネクターを使用します。

URI 形式

```
paho:queueName[?options]
```

たとえば、以下のスニペットは、Camel ルーターと同じホストにインストールされている MQTT ブローカーからメッセージを読み取ります。

```
from("paho:some/queue").
  to("mock:test");
```

以下のスニペットは、MQTT ブローカーにメッセージを送信します。

```
from("direct:test").
  to("paho:some/target/queue");
```

?option=value&option=value&... の形式で、URI にクエリーオプションを追加できます。たとえば、リモート MQTT ブローカーからメッセージを読み取る方法は次のとおりです。

```
from("paho:some/queue?brokerUrl=tcp://iot.eclipse.org:1883").
  to("mock:test");
```

コンポーネントのプロジェクトへの追加

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-paho</artifactId>
    <version>x.y.z</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

Paho アーティファクトは Maven Central でホストされていないため、Eclipse Paho リポジトリーを POM xml ファイルに追加する必要があります。

```
<repositories>
  <repository>
    <id>eclipse-paho</id>
    <url>https://repo.eclipse.org/content/repositories/paho-releases</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

デフォルトのペイロードタイプ

デフォルトでは、Camel Paho コンポーネントは MQTT メッセージから抽出（または配置）されたバイナリーペイロードで動作します。

```
// Receive payload
byte[] payload = (byte[]) consumerTemplate.receiveBody("paho:topic");

// Send payload
byte[] payload = "message".getBytes();
producerTemplate.sendBody("paho:topic", payload);
```

当然ながら、Camel ビルドタイプ変換 API は自動データ型変換を実行できます。以下の例では、Camel は自動的にバイナリーペイロードを String （および逆順）に変換します。

```
// Receive payload
String payload = consumerTemplate.receiveBody("paho:topic", String.class);

// Send payload
String payload = "message";
producerTemplate.sendBody("paho:topic", payload);
```

URI オプション

オプション	デフォルト	説明
clientId	**camel-<timestamp>**	MQTT クライアント識別子。
brokerUrl	**tcp://localhost:1883**	MQTT ブローカーの URL。
persistence	**memory**	使用されるクライアントの永続性 - **memory** または **file**

filePersistenceDirectory	現行ディレクトリー	(Camel 2.16.1 および 2.17) ファイルの永続性で使用されるベースディレクトリー。ファイル以外の永続性が使用されている場合は有効になりません。
qos	2	クライアントの QoS (Quality of Service)レベル(**0**-**2**)
connectOptions	none	Camel レジストリーにある org.eclipse.paho.client.mqttv3.**MqttConnectOptions** インスタンスへの参照。参照 **MqttConnectOptions** インスタンスは、エンドポイントによって接続を初期化するために使用されます。たとえば、**connectOptions=#my**ConnectOptions という名前の Spring Bean を参照する場合、表記法を使用できます。**ConnectOptions my**レジストリーに **MqttConnectOptions** のインスタンスが 1 つしかない場合は、エンドポイントによって自動的に取得されます。

たとえば、Camel で使用される規則オーバー設定方法は、ほとんどの状況で非常に便利ですが、MQTT クライアント接続をより詳細に制御したい場合があります。このような状況に対応するために、タイプ org.eclipse.paho.client.mqttv3.MqttConnectOptions の Bean を Camel レジストリーに追加します。Spring アプリケーションでは、Bean をアプリケーションコンテキストに追加することを意味します。以下のスニペットは、パスワードベースの認証を使用して MQTT ブローカーに接続します。

```
@Bean
MqttConnectOptions connectOptions() {
  MqttConnectOptions connectOptions = new MqttConnectOptions();
  connectOptions.setUserName("henry");
  connectOptions.setPassword("secret".toCharArray());
  return connectOptions;
}
```

HEADERS

次のヘッダーは、Paho コンポーネントによって認識されます。

ヘッダー	Java 定数	エンドポイントタイプ	値のタイプ	説明

PahoOriginalMessage	**PahoConstants.HEADER_ORIGINAL_MESSAGE**	コンシューマー	**org.eclipse.paho.client.mqttv3.MqttMessage**	クライアントが受信した元の Paho メッセージインスタンス。 **非推奨：** Camel 2.17 以降では、元の MqttMessage はヘッダーとして保存されませんが、ゲッター **getMqttMessage** を持つ **org.apache.camel.component.paho.PahoMessage** メッセージには保存されません。
CamelMqttTopic	PahoConstants.MQTT_TOPIC	コンシューマー	文字列	**Camel 2.17:** トピック

第124章 PAX-LOGGING

PAXLOGGING コンポーネント

Camel 2.6 で利用可能

paxlogging コンポーネントを OSGi 環境で使用すると、PaxLogging イベントを受信して処理できます。

DEPENDENCIES

Maven ユーザーは以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-paxlogging</artifactId>
  <version>${camel-version}</version>
</dependency>
```

ここで、$\{camel-version\} は実際の Camel バージョン(2.6.0 以降)に置き換える必要があります。

URI 形式

```
paxlogging:appender
```

appender は、PaxLogging サービス設定で設定する必要がある pax アペンダーの名前です。

URI オプション

名前	デフォルト値	説明

メッセージヘッダー

名前	タイプ	メッセージ	説明

メッセージボディー

in メッセージボディーは受信した PaxLoggingEvent に設定されます。

使用例

```
<route>
   <from uri="paxlogging:camel"/>
   <to uri="stream:out"/>
</route>
```

–

設定:

> ***log4j.rootLogger=INFO, out, osgi:VmLogAppender, osgi:camel***

第125章 PDF

PDF

Camel 2.16.0 から利用可能

PDF コンポーネントは、PDF ドキュメントからコンテンツを作成、変更、または抽出する機能を提供します。このコンポーネントは、Apache PDFBox を基礎となるライブラリーとして使用して PDF ドキュメントと連携します。

PDF コンポーネントを使用するには、Maven ユーザーは以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-pdf</artifactId>
   <version>2.17.0.redhat-630xxx</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

PDF コンポーネントはプロデューサーエンドポイントのみをサポートします。

```
pdf:operation[?options]
```

operation は、PDF ドキュメントで実行する固有のアクションです。

操作

名前	デフォルト値	説明
marginTop	20	ページのテキストとトップエッジ間の pixel にマージンサイズを設定します。
marginBottom	20	ページのテキストと下線間の pixel にマージンサイズを設定します。

marginLeft	20	ページのテキストと左エッジ間のピクセルにマージンサイズを設定します。
marginRight	40	ページのテキストと右端間のピクセルにマージンサイズを設定します。このオプションは、textProcessingFactory オプションが lineTermination の場合は無視されます。
fontSize	14	ピクセルにフォントのサイズを設定します。
pageSize	**PAGE_SIZE_A4**	ページのサイズを設定します。使用できる値 • **PAGE_SIZE_A0** • **PAGE_SIZE_A1** • **PAGE_SIZE_A2** • **PAGE_SIZE_A3** • **PAGE_SIZE_A4** • **PAGE_SIZE_A5** • **PAGE_SIZE_A6** • **PAGE_SIZE_LETTER**
font	Helvetica	PDFBox のベースフォントの 1 つ。
textProcessingFactory	**lineTermination**	テキスト処理ファクトリーを設定します。**lineTermination** : 行中断書き込みストラテジーのクラスセットを構築します。行終了記号でスライスされるテキストは、行に適合するかどうかに関係なく記述されます。**autoFormatting** - テキストは単語でスライスされ、行に適合する最大単語数が PDF ドキュメントに書き込まれます。このストラテジーでは、行に収まらないすべての単語が新しい行に移動されます。

HEADERS

ヘッダー

pdf-document	**Mandatory append** 操作のヘッダー、および他のすべての操作では無視されます。想定されるタイプは PDDocument です。追加操作に使用される PDF ドキュメントを保存します。
protection-policy	想定されるタイプは ProtectionPolicy です。これが指定されている場合、PDF ドキュメントがこれで暗号化されます。
decryption-material	想定されるタイプは DecryptionMaterial です。PDF ドキュメントが暗号化されている場合は **必須** ヘッダーです。

第126章 PGEVENT

PGEVENT コンポーネント

これは Apache Camel のコンポーネントで、PostgreSQL 8.3 以降に追加された LISTEN/NOTIFY コマンドに関連する PostgreSQL イベントの生成/消費を可能にします。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-pgevent</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

pgevent コンポーネントは、以下の 2 つのスタイルのエンドポイント URI 表記を使用します。

```
pgevent:datasource[?parameters]
pgevent://host:port/database/channel[?parameters]
```

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

オプション

オプション	型	デフォルト	説明
datasource	文字列		使用するレジストリーから検索するデータソースの名前
hostname	文字列	localhost	データソースを使用する代わりに、このホスト名とポートを使用して PostgresSQL データベースに接続します。

port	int	5432	データソースを使用する代わりに、このホスト名とポートを使用して PostgresSQL データベースに接続します。
database	文字列		データベース名
channel	文字列		チャネル名
user	文字列	postgres	Username
pass	文字列		Password

第127章 プリンター

プリンターコンポーネント

Apache Camel 2.1 から利用可能

プリンター コンポーネントは、ルート上のペイロードをプリンターに転送する方法を提供します。当然ながら、ペイロードは、コンポーネントが適切に出力するためにペイロードをフォーマットする必要があります。目的は、特定のペイロードをジョブとして Apache Camel フローのラインプリンターに転送することです。

このコンポーネントはプロデューサーエンドポイントのみをサポートします。

この機能により、javax 印刷 API を使用して local、remote、またはワイヤレスリンクされたプリンターという名前のデフォルトのプリンターでペイロードを出力できます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-printer</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

プリンターの URI スキームは標準化されていないため（標準が IETF 印刷標準となるため、ベンダーによって均一に適用されるものではない）、スキームとして lpr が選択されています。

```
lpr://localhost/default[?options]
lpr://remotehost:port/path/to/printer[?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
mediaSize	**NA_LETTER**	javax.print.attribute.standard.MediaSizeName API の列挙名で定義されるように companyary を設定します。デフォルト設定では North American Letter sized sitesary を使用します。値のケースは無視されます。たとえば、**iso_a4** および **ISO_A4** の値を使用できます。
コピー	**1**	**javax.print.attribute.standard.Copies** API に基づいてコピー数を設定します。
sides	**Sides.ONE_SIDED**	**javax.print.attribute.standard.Sides** API に基づいてサイドまたは 2 つのサイド印刷を設定します。
flavor	**DocFlavor.BYTE_ARRAY**	**javax.print.DocFlavor** API に基づいて **DocFlavor** を設定します。
mimeType	**AUTOSENSE**	**javax.print.DocFlavor** API でサポートされる **mimeTypes** を設定します。
mediaTray	**AUTOSENSE**	Camel 2.11.x は javax.print.DocFlavor API によってサポートされる MediaTray を設定するため、
printerPrefix	**null**	Camel 2.11.x はプリンターの接頭辞名を設定するため、プリンター名が //hostname/printer で開始されていない場合に便利です。
sendToPrinter	**true**	このオプションを **false** に設定すると、印刷データ がプリンターに送信されなくなります。
オリエンテーション	**portrait**	Camel 2.13.x 以降、ページ指向を設定します。設定可能な値：**portrait**、**landscape**、**reverse-portrait** または **reverse-landscape** **javax.print.attribute.standard.OrientationRequested**

プリンタープロデューサー

プリンターへのデータ送信は非常に簡単で、ルートの にメッセージ交換を送信することができるプロデューサーエンドポイントを作成します。

例 1: アルファベットおよび 1 サイドモードでデフォルトプリンターでテキストベースのペイロードを出力する

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
      from(file://inputdir/?delete=true)
      .to("lpr://localhost/default?copies=2" +
          "&flavor=DocFlavor.INPUT_STREAM&" +
          "&mimeType=AUTOSENSE" +
          "&mediaSize=NA_LETTER" +
          "&sides=one-sided")
    }};
```

例 2: A4 SITESARY および 1 サイドモードでリモートプリンターで GIF ベースのペイロードを出力する

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
      from(file://inputdir/?delete=true)
      .to("lpr://remotehost/sales/salesprinter" +
          "?copies=2&sides=one-sided" +
          "&mimeType=GIF&mediaSize=ISO_A4" +
          "&flavor=DocFlavor.INPUT_STREAM")
   }};
```

例 3: 日本語の投稿者および 1 サイドモードでリモートプリンターで JPEG ベースのペイロードを出力する

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
      from(file://inputdir/?delete=true)
      .to("lpr://remotehost/sales/salesprinter" +
          "?copies=2&sides=one-sided" +
          "&mimeType=JPEG" +
          "&mediaSize=JAPANESE_POSTCARD" +
          "&flavor=DocFlavor.INPUT_STREAM")
    }};
```

第128章 プロパティー

PROPERTIES コンポーネント

Apache Camel 2.3 で利用可能

URI 形式

> ***properties:key[?options]***

*ここでの **key** は、検索するプロパティーのキーです。*

オプション

名前	タイプ	デフォルト	説明
cache	**boolean**	**true**	ロードされたプロパティーをキャッシュするかどうか。
ロケーション	**文字列**	**null**	プロパティーをロードする場所の一覧。複数の場所を分離する場合はコンマで区切ることができます。このオプションはデフォルトの場所を上書きし、このオプションからのロケーション ***のみ*** を使用します。
encoding	**文字列**	**null**	**Camel 2.14.3/2.15.1:** 特定の文字セットを使用して UTF-8 などのプロパティーを読み込む。デフォルトでは、ISO-8859-1 (latin1)が使用されます。
ignoreMissingLocation	**boolean**	**false**	**Camel 2.10:** プロパティーファイルが見つからない場合など、ロケーションが見つからないかどうかを警告せずに無視するかどうか。

propertyPrefix	**文字列**	**null**	Camel 2.9: 解決前にプロパティー名の前に追加された任意の接頭辞。
propertySuffix	**文字列**	**null**	Camel 2.9: 解決前にプロパティー名に追加される任意の接尾辞。
fallbackToUnaugmentedProperty	**boolean**	**true**	Camel 2.9: **true** の場合、指定したプレーンプロパティー名をフォールバックする前に **propertyPrefix** および **propertySuffix** で拡張されたプロパティー名の解決を最初に試みます。**false** の場合、拡張されたプロパティー名のみが検索されます。
prefixToken	**文字列**	**{{**	Camel 2.9: プロパティートークンの開始を示すトークン。
suffixToken	**文字列**	**}}**	Camel 2.9: プロパティートークンの最後を示すトークン。

systemPropertiesMode	**int**	**2**	Camel 2.16: システムプロパティーを解決し、使用するかどうかに使用するモード。 ● **0** = never (JVM システムプロパティーは使用されません) ● **1** = fallback (JVM システムプロパティーは、キーのある通常のプロパティーが存在しない場合にのみフォールバックとして使用されます) ● **2** = override (JVM システムプロパティーが存在する場合は使用され、そうでない場合は通常のプロパティーが使用されます) これを org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer で Spring のプロパティープレースホルダーにブリッジする場合、BridgePropertyPlaceholderConfigurer の設定は PropertiesComponent の設定よりも優先されます。

JAVA コードからのプロパティーの解決

CamelContext で resolvePropertyPlaceholders メソッドを使用して、任意の Java コードからプロパティーを解決できます。

その他の参考資料

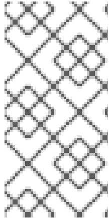
- ***Apache Camel Development Guide の section "Property Placeholders"***

- *プロパティーで暗号化された値（パスワードなど）を使用するための* ***jasypt*** *???*

第129章 QUARTZ

QUARTZ コンポーネント

quartz: コンポーネントは、Quartz スケジューラー 1.x を使用してスケジュールされたメッセージの配信を提供します。各エンドポイントは、異なるタイマー(Quartz 用語、Trigger および JobDetail)を表します。



注記

Quartz 2.x を使用している場合は、Camel 2.12 以降では Quartz2 コンポーネントを使用する必要があります。

URI 形式

```
quartz://timerName?options
quartz://groupName/timerName?options
quartz://groupName/timerName?cron=expression
quartz://timerName?cron=expression
```

コンポーネントは CronTrigger または SimpleTrigger のいずれかを使用します。cron 式が指定されていない場合、コンポーネントは単純なトリガーを使用します。groupName が指定されていない場合、quartz コンポーネントは Camel グループ名を使用します。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

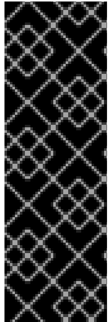
パラメーター	デフォルト	説明
cron	なし	**cron** 式を指定します(**trigger.*** または **job.*** オプションと互換性がありません)。
trigger.repeatCount	**0**	SimpleTrigger: タイマーを繰り返す回数はどれくらいですか?
trigger.repeatInterval	**0**	SimpleTrigger: 繰り返されるトリガーの間隔(ミリ秒単位)。
job.name	**null**	ジョブ名を設定します。

ジョブ。*XXX*	**null**	ジョブオプションを ***XXX*** セッター名で設定します。
trigger.*XXX*	**null**	***XXX*** セッター名で trigger オプションを設定します。
stateful	**false**	デフォルトのジョブの代わりに Quartz **StatefulJob** を使用します。
fireNow	**false**	Camel 2.2.0: **true** の場合は、**SimpleTrigger** の使用時にルートを起動したときにトリガーを実行します。
deleteJob	**true**	Camel 2.12: **true** の場合、ルートが停止するとトリガーが自動的に削除されます。**false** の場合、スケジューラーは残り、ユーザーは Camel URI で事前設定されたトリガーを再利用できます。名前が一致するだけです。**deleteJob** と **true** の両方を設定することはできません。
pauseJob	**false**	Camel 2.12: **true** の場合、トリガーはルートが停止すると自動的に一時停止します。**false** の場合、スケジューラーは残り、ユーザーは Camel URI で事前設定されたトリガーを再利用できます。名前が一致するだけです。**deleteJob** と **pauseJob** は **true** の両方に設定することはできない点に注意してください。
usingFixedCamelContextName	**false**	Camel 2.15.0: **true** の場合、**JobDataMap** は **CamelContext** 名を直接使用して Camel コンテキストを参照します。**false** の場合は、deployed がデプロイ中に変更される可能性のある管理名を使用します。**JobDataMap** **CamelContext**

*たとえば、以下のルーティングルールは、**mock:results** エンドポイントに 2 つのタイマーイベントを実行します。*

```
from("quartz://myGroup/myTimerName?
trigger.repeatInterval=2&trigger.repeatCount=1").routeId("myRoute").to("mock:result");
```

StatefulJob を使用する場合、JobDataMap はジョブが実行されるたびに再永続化されるため、次の実行の状態を保持します。



OSGI での実行と QUARTZ ルートを持つ複数のバンドルがある

Apache ServiceMix や Apache Karaf などの OSGi で を実行し、Quartz エンドポイントから開始する Camel ルートを持つ複数のバンドルがある場合は、OSGi コンテナーの QuartzScheduler で必要となるため、この ID が一意である <camelContext> に ID を割り当てるようにしてください。<camelContext> に id を設定しないと、一意の ID が自動的に割り当てられ、問題はありません。

QUARTZ.PROPERTIES ファイルの設定

デフォルトでは、Quartz はクラスパスの org/quartz ディレクトリーで quartz.properties ファイルを検索します。WAR デプロイメントを使用している場合は、WEB-INF/classes/org/quartz の quartz.properties をドロップするだけです。

ただし、Camel Quartz コンポーネントでは、プロパティーを設定することもできます。

パラメーター	デフォルト	タイプ	説明
properties	null	プロパティー	Camel 2.4: **java.util.Propoperties** インスタンスを設定できます。
propertiesFile	null	文字列	Camel 2.4: クラスパスから読み込むプロパティーのファイル名

これには、Spring XML で以下のように設定します。

```
<bean id="quartz" class="org.apache.camel.component.quartz.QuartzComponent">
  <property name="propertiesFile" value="com/mycompany/myquartz.properties"/>
</bean>
```

JMX での QUARTZ スケジューラーの有効化

JMX を有効にするには、quartz スケジューラープロパティーを設定する必要があります。これは通常、org.quartz.scheduler.jmx.export オプションを設定ファイルの true 値に設定します。

Camel 2.13 以降では、明示的に無効にしない限り、Camel はこのオプションを true に自動的に設定します。

QUARTZ スケジューラーの起動

Camel 2.4 以降で利用可能

Quartz コンポーネントは、Quartz スケジューラーの遅延、または自動起動を行わないオプションを提供します。

パラメーター	デフォルト	タイプ	説明
startDelayedSeconds	0	int	Camel 2.4: quartz スケジューラーを起動するまで待機する秒数。
autoStartScheduler	true	boolean	Camel 2.4: スケジューラーを自動的に起動するかどうか。

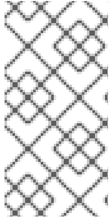
これには、Spring XML で以下のように設定します。

```
<bean id="quartz" class="org.apache.camel.component.quartz.QuartzComponent">
    <property name="startDelayedSeconds" value="5"/>
</bean>
```

クラスタリング

Camel 2.4 以降で利用可能

クラスターモードで Quartz を使用する場合、JobStore はクラスター化されます。その後、Quartz コンポーネントはノードの停止/シャットダウン時にトリガーを一時停止/削除し ません。これにより、トリガーはクラスター内の他のノードでも実行を継続できます。



注記

クラスター化されたノードで実行されている場合、エンドポイントに対して一意のジョブ名/グループを保証するチェックは行われません。

メッセージヘッダー

Apache Camel は Quartz Execution Context からの getter をヘッダー値として追加します。以下のヘッダーが 追加されます：calendar、fireTime、jobDetail、jobInstance、jobRuntTime、mergedJobDataMap、nextFireTime、previousFireTime、refireCount、Result、scheduledFireTime、scheduler、trigger Name、triggerGroup。

fireTime ヘッダーには、エクスチェンジがいつ実行されたかの java.util.Date が含まれます。

CRON トリガーの使用

Apache Camel 2.0 Quartz の時点では、便利な形式でタイマーを指定する Cron のような式 がサポートされます。これらの式は cron URI パラメーターで使用できますが、有効な URI エンコーディングを保持するには、スペースの代わりに + を使用できます。Quartz では、cron 式の使用方法に関する チュートリアルを少し 紹介します。

たとえば、以下は、毎週 12pm (noon)から 6pm までのメッセージを 5 分ごとに実行します。

```
from("quartz://myGroup/myTimerName?cron=0+0/5+12-18+?+*+MON-
FRI").to("activemq:Totally.Rocks");
```

これは cron 式の使用と同等です。

```
0 0/5 12-18 ? * MON-FRI
```

以下の表は、有効な URI 構文を保持するために使用する URI 文字エンコーディングを示しています。

URI 文字	Cron 文字
\+	スペース

タイムゾーンの指定

Camel 2.8.1 から利用可能です。Quartz Scheduler を使用すると、トリガーごとにタイムゾーンを設定できます。たとえば、国のタイムゾーンを使用するには、以下のように実行できます。

```
quartz://groupName/timerName?cron=0+0/5+12-18+?+*+MON-
FRI&trigger.timeZone=Europe/Stockholm
```

timeZone 値は、java.util.TimeZone で許可される値です。

Camel 2.8.0 以前のバージョンでは、エンドポイント uri から設定できるようにするには、カスタム文字列 を java.util.TimeZone Type Converter に提供する必要があります。Camel 2.8.1 以降では、このような Type Converter が camel-core に含まれています。

- *Quartz2*
- *Timer*

第130章 QUARTZ2

QUARTZ2 COMPONENT

Camel 2.12.0 から利用可能

quartz2: コンポーネントは、Quartz スケジューラー 2.x を使用してスケジュールされたメッセージの配信を提供します。各エンドポイントは、異なるタイマー(Quartz 用語、Trigger および JobDetail)を表します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-quartz2</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

注記： Quartz 2.x API は Quartz 1.x と互換性がありません。古い Quartz 1.x をそのまま使用する必要がある場合は、代わりに古い Quartz コンポーネントを使用してください。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
quartz2://timerName?options
quartz2://groupName/timerName?options
quartz2://groupName/timerName?cron=expression
quartz2://timerName?cron=expression
```

コンポーネントは CronTrigger または SimpleTrigger のいずれかを使用します。cron 式が指定され

ていない場合、コンポーネントは単純なトリガーを使用します。groupName が指定されていない場合、quartz コンポーネントは Camel グループ名を使用します。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

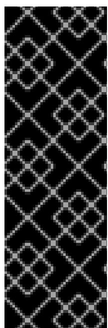
パラメーター	デフォルト	説明
cron	なし	**cron** 式を指定します(**trigger.*** または **job.*** オプションと互換性がありません)。
trigger.repeatCount	**0**	SimpleTrigger: タイマーを繰り返す回数はどれくらいですか？
trigger.repeatInterval	**1000**	SimpleTrigger: 繰り返されるトリガーの間隔（ミリ秒単位）。この間隔を使用して簡単なトリガーを使用するには、**trigger.repeatCount** を有効にする必要があります。
job.name	**null**	ジョブ名を設定します。
ジョブ。*XXX*	**null**	ジョブオプションを ***XXX*** セッター名で設定します。
trigger.*XXX*	**null**	***XXX*** セッター名で trigger オプションを設定します。
stateful	**false**	デフォルトのジョブの代わりに Quartz **@PersistJobDataAfterExecution** および **@DisallowConcurrentExecution** を使用します。

fireNow	**false**	true の場合、SimpleTrigger の使用時にルートが起動するとトリガーが実行されます。
deleteJob	**true**	true に設定すると、ルートが停止したときにトリガーが自動的に削除されます。false に設定すると、スケジューラーに残ります。false に設定すると、ユーザーは camel Uri で事前設定されたトリガーを再利用できます。名前が一致するだけです。deleteJob と pauseJob の両方を true に設定することはできないことに注意してください。
pauseJob	**false**	true に設定すると、ルート停止時にトリガーが自動的に一時停止します。false に設定すると、スケジューラーに残ります。false に設定すると、ユーザーは camel Uri で事前設定されたトリガーを再利用できます。名前が一致するだけです。deleteJob と pauseJob の両方を true に設定することはできないことに注意してください。
durableJob	**false**	**Camel 2.12.4/2.13**: 孤立した後にジョブが保存したままになるかどうか（トリガーを参照しない）。
recoverableJob	**false**	**Camel 2.12.4/2.13**: 'recovery' または 'fail-over' が発生した場合にジョブを再実行する必要があるかどうかをスケジューラーに指示します。
usingFixedCamelContextName	**false**	**Camel 2.15.0: true** の場合、**JobDataMap** は **CamelContext** 名を直接使用して Camel コンテキストを参照します。**false** の場合は、deployed がデプロイ中に変更される可能性のある管理名を使用します。**JobDataMap CamelContext**
customCalendar	なし	**Camel 2.17.0:** スケジューラーおよびトリガーにカスタムカレンダーを追加して、特定の日付範囲を回避します（例：Holidays）。**customCalendar** タイプは **org.quartz.Calendar** です。

たとえば、以下のルーティングルールは、mock:results エンドポイントに 2 つのタイマーイベントを実行します。

```
from("quartz2://myGroup/myTimerName?trigger.repeatInterval=2&trigger.repeatCount=1")
    .routeId("myRoute")
    .to("mock:result");
```

stateful=true を使用する場合、JobDataMap はジョブが実行されるたびに再永続化されるため、次の実行の状態を維持します。



OSGI での実行と QUARTZ ルートを持つ複数のバンドルがある

Apache ServiceMix や Apache Karaf などの OSGi で を実行し、Quartz2 エンドポイントから開始する Camel ルートを持つ複数のバンドルがある場合は、OSGi コンテナーの QuartzScheduler で必要となるため、この ID が一意である <camelContext> に ID を割り当てるようにしてください。<camelContext> に id を設定しないと、一意の ID が自動的に割り当てられ、問題はありません。

QUARTZ.PROPERTIES ファイルの設定

デフォルトでは、Quartz はクラスパスの org/quartz ディレクトリーで quartz.properties ファイルを検索します。WAR デプロイメントを使用している場合は、WEB-INF/classes/org/quartz の quartz.properties をドロップするだけです。

ただし、Camel Quartz2 コンポーネントでは、プロパティーを設定することもできます。

パラメーター	デフォルト	タイプ	説明
properties	null	プロパティー	java.util.Properties インスタンスを設定できます。
propertiesFile	null	文字列	クラスパスから読み込むプロパティーのファイル名

これには、Spring XML で以下のように設定します。

```
<bean id="quartz" class="org.apache.camel.component.quartz2.QuartzComponent">
   <property name="propertiesFile" value="com/mycompany/myquartz.properties"/>
</bean>
```

JMX での QUARTZ スケジューラーの有効化

JMX を有効にするには、quartz スケジューラープロパティーを設定する必要があります。これは通常、org.quartz.scheduler.jmx.export オプションを設定ファイルの true 値に設定します。

Camel 2.13 以降では、明示的に無効にしない限り、Camel はこのオプションを true に自動的に設定します。

QUARTZ スケジューラーの起動

Quartz2 コンポーネントは、Quartz スケジューラーの遅延、または自動起動を行わないオプションを提供します。

パラメーター	デフォルト	タイプ	説明
startDelayedSeconds	**0**	**int**	quartz スケジューラーを起動するまで待機する秒数。
autoStartScheduler	**true**	**boolean**	スケジューラーを自動起動するかどうか。

これには、Spring XML で以下のように設定します。

```
<bean id="quartz2" class="org.apache.camel.component.quartz2.QuartzComponent">
   <property name="startDelayedSeconds" value="5"/>
</bean>
```

クラスタリング

クラスターモードで Quartz を使用する場合、JobStore はクラスター化されます。次に、Quartz2 コンポーネントは、ノードの停止/シャットダウン時にトリガーを一時停止/削除し ません。これにより、トリガーはクラスター内の他のノードでも実行を継続できます。

注記: クラスターノードで実行している場合は、エンドポイントに対して一意のジョブ名/グループを確認するためのチェックが行われません。

メッセージヘッダー

Camel は Quartz Execution Context からの getter をヘッダー値として追加します。以下のヘッダーが 追加されます：
calendar、fireTime、jobDetail、jobInstance、jobRuntTime、mergedJobDataMap、nextFireTime、previousFireTime、refireCount、Result、scheduledFireTime、scheduler、trigger Name、triggerGroup。

fireTime ヘッダーには、エクスチェンジがいつ実行されたかの java.util.Date が含まれます。

CRON トリガーの使用

Quartz は、便利な形式でタイマーを指定する Cron のような式 をサポートしています。これらの式は cron URI パラメーターで使用できますが、有効な URI エンコーディングを保持するには、スペースの代わりに + を使用できます。

たとえば、以下は、毎週 12pm (noon)から 6pm までの 5 分ごとにメッセージを表示します。

```
from("quartz2://myGroup/myTimerName?cron=0+0/5+12-18+?+*+MON-FRI").to("activemq:Totally.Rocks");
```

これは cron 式の使用と同等です。

```
0 0/5 12-18 ? * MON-FRI
```

以下の表は、有効な URI 構文を保持するために使用する URI 文字エンコーディングを示しています。

URI 文字	Cron 文字
\+	スペース

タイムゾーンの指定

Quartz スケジューラーを使用すると、トリガーごとにタイムゾーンを設定できます。たとえば、国のタイムゾーンを使用するには、以下のように実行できます。

```
quartz2://groupName/timerName?cron=0+0/5+12-18+?+*+MON-FRI&trigger.timeZone=Europe/Stockholm
```

timeZone 値は、java.util.TimeZone で許可される値です。

QUARTZSCHEDULEDPOLLCONSUMERSCHEDULER の使用

Quartz2 コンポーネントは Polling Consumer スケジューラーを提供します。これにより、File や FTP コンシューマーなどの Polling Consumer に cron ベースのスケジューリングを使用できます。

たとえば、cron ベースの式を使用して 2 秒間隔でファイルをポーリングするには、Camel ルートを以下のように定義できます。

```
from("file:inbox?scheduler=quartz2&scheduler.cron=0/2+*+*+*+*+?")
  .to("bean:process");
```

scheduler=quartz2 を定義して、Quartz2 ベースのスケジューラーを使用するように Camel に指示することに注意してください。次に、scheduler.xxx オプションを使用してスケジューラーを設定します。Quartz2 スケジューラーでは、cron オプションを設定する必要があります。

以下のオプションがサポートされます。

パラメーター	デフォルト	タイプ	説明
quartzScheduler	null	org.quartz.Scheduler	カスタム Quartz スケジューラーを使用するには、以下を行います。設定しない場合は、Quartz2 コンポーネントから共有スケジューラーが使用されます。
cron	null	文字列	**必須**: ポーリングをトリガーするために cron 式を定義します。

triggerId	**null**	**文字列**	トリガー ID を指定します。指定しない場合は、UUID が生成され、使用されます。クラスター環境では triggerId を指定できず、クラスターの各ノードには一意の instanceId が必要です。
triggerGroup	**QuartzScheduledPollConsumerScheduler**	**文字列**	トリガーグループを指定します。
timeZone	デフォルト	**TimeZone**	CRON トリガーに使用するタイムゾーン。

重要： エンドポイント URI からこれらのオプションを設定することを忘れないでください。には、scheduler の接頭辞を指定する必要があります。たとえば、トリガー ID およびグループを設定するには、以下を実行します。

```
from("file:inbox?scheduler=quartz2&scheduler.cron=0/2+*+*+*+*+?&scheduler.triggerId=myId&scheduler.triggerGroup=myGroup")
    .to("bean:process");
```

Spring には CRON スケジューラーもあるため、以下を使用することもできます。

```
from("file:inbox?scheduler=spring&scheduler.cron=0/2+*+*+*+*+?")
    .to("bean:process");
```

-

  Quartz

-

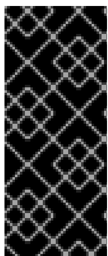
  Timer

第131章 QUICKFIX

QUICKFIX/J コンポーネント

Camel 2.0 で利用可能

クイックフィックス コンポーネントは、Camel で を使用するために QuickFIX/J FIX エンジンを調整します。このコンポーネントは、メッセージトランスポートに標準の Financial Interchange (FIX)プロトコル を使用します。



以前のバージョン

クイックフィックス コンポーネントは Camel 2.5 用に書き換えられました。2.5 よりも前の クイックフィックス コンポーネントの使用方法は、以下のドキュメントを参照してください。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-quickfix</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
quickfix:configFile[?sessionID=sessionID&lazyCreateEngine=true|false]
```

configFile は、FIX エンジン（クラスパスにあるリソースとして配置）に使用する QuickFIX/J 設定の名前です。オプションの sessionID は、特定の FIX セッションを識別します。sessionID の形式は次のとおりです。

```
(BeginString):(SenderCompID)[/(SenderSubID)[/(SenderLocationID)]]->(TargetCompID)
[/(TargetSubID)[/(TargetLocationID)]]
```

オプションの lazyCreateEngine (Camel 2.12.3+)パラメーターを使用すると、QuickFIX/J エンジンをオンデマンドで作成できます。値 true は、最初のメッセージが送信されるか、ルート定義にコンシューマーが設定されている場合にエンジンが開始されることを意味します。false の場合、エンジン

はエンドポイントの作成時に開始されます。このパラメーターがない場合、コンポーネントのプロパティー **lazyCreateEngines** の値が使用されます。

URI の例：

> quickfix:config.cfg
>
> quickfix:config.cfg?sessionID=FIX.4.2:MyTradingCompany->SomeExchange
>
> quickfix:config.cfg?sessionID=FIX.4.2:MyTradingCompany->SomeExchange&lazyCreateEngine=true

エンドポイント

FIX セッションは、クイックフィックス コンポーネントのエンドポイントです。エンドポイント URI は、特定の QuickFIX/J エンジンによって管理される単一のセッションまたはすべてのセッションを指定できます。一般的なアプリケーションは FIX エンジンを 1 つだけ使用しますが、上級ユーザーは クイック修正 コンポーネントのエンドポイント URI で異なる設定ファイルを参照することで、複数の FIX エンジンを作成できます。

コンシューマーがエンドポイント URI にセッション ID が含まれていない場合、URI で指定された設定ファイルに関連付けられた FIX エンジンによって管理されるすべてのセッションに対するエクスチェンジを受信します。プロデューサーがエンドポイント URI でセッションを指定しない場合、送信される FIX メッセージにセッション関連のフィールドを含める必要があります。URI でセッションが指定されている場合、コンポーネントはセッション関連のフィールドを FIX メッセージに自動的に挿入します。

エクスチェンジの形式

エクスチェンジヘッダーには、エクスチェンジのフィルターリング、ルーティング、およびその他の処理に役立つ情報が含まれます。以下のヘッダーを使用できます。

ヘッダー名	説明
EventCategory	**AppMessageReceived**、**AppMessageSent**、**AdminMessageReceived**、**AdminMessageSent**、**SessionCreated**、**SessionLogon**、**SessionLogoff** のいずれか。**QuickfixjEventCategory** enum を参照してください。
SessionID	FIX Message SessionID
MessageType	FIX MsgType タグの値

DataDictionary	受信メッセージの解析に使用するデータディクショナリーを指定します。データディクショナリーのインスタンス、または QuickFIX/J データディクショナリーファイルのリソースパスを指定できます。

DataDictionary ヘッダーは、文字列メッセージが受信され、ルートで解析する必要がある場合に役立ちます。QuickFix/J では、（たとえば、グループを繰り返して）特定のタイプのメッセージを解析するためのデータディクショナリーが必要です。メッセージ文字列の受信後にルートに DataDictionary ヘッダーを挿入することで、FIX エンジンはデータを適切に解析できます。

QUICKFIX/J CONFIGURATION EXTENSIONS

QuickFIX/J を直接使用する場合、1 つは通常、ロギングアダプター、メッセージストア、および通信コネクターのインスタンスを作成するためのコードを書き込みます。クイックフィックス コンポーネントは、設定ファイルの情報に基づいてこれらのクラスのインスタンスを自動的に作成します。また、一般的な必要な設定の多くにデフォルトを提供し、追加機能を追加します(JMX サポートをアクティブにする機能など)。

以下のセクションでは、クイックフィックス コンポーネントが QuickFIX/J 設定を処理する方法を説明します。QuickFIX/J 設定に関する包括的な情報は、QFJ ユーザーマニュアル を参照してください。

通信コネクター

コンポーネントによって、QuickFIX/J 設定ファイルでイニシエーターまたはアクセプターセッション設定を検出すると、対応するイニシエーターやアクセプターコネクターが自動的に作成されます。これらの設定は、デフォルトの または 設定ファイルの特定の session セクションに指定できます。

セッションの設定	コンポーネントの動作
ConnectionType=initiator	イニシエーターコネクターの作成
ConnectionType=acceptor	アクセプターコネクターの作成

QuickFIX/J セッションコネクターのスレッドモデルを指定することもできます。これらの設定は設定ファイルのすべてのセッションに影響し、設定の default セクションに配置する必要があります。

デフォルト/グローバル設定	コンポーネントの動作

ThreadModel=ThreadPerConnector	**SocketInitiator** または **SocketAcceptor** （デフォルト）の使用
ThreadModel=ThreadPerSession	**ThreadedSocketInitiator** または **ThreadedSocketAcceptor**の使用

ロギング

QuickFIX/J ロガー実装を指定するには、設定ファイルの default セクションに以下の設定を追加します。設定に以下の設定がない場合は、ScreenLog がデフォルトになります。複数のログ実装を意味する設定を含めるのはエラーです。

デフォルト/グローバル設定	コンポーネントの動作
ScreenLogShowEvents	**ScreenLog**を使用する
ScreenLogShowIncoming	**ScreenLog**を使用する
ScreenLogShowOutgoing	**ScreenLog**を使用する
SLF4J*	Camel 2.6+**SLF4JLog** を使用します。SLF4J 設定のいずれかにより、このログが使用されます。
FileLogPath	**FileLog**を使用する
JdbcDriver	**JdbcLog**の使用

メッセージストア

QuickFIX/J メッセージストアの実装を指定するには、設定ファイルの default セクションに以下の設定を追加します。設定に以下の設定がない場合は、MemoryStore がデフォルトになります。複数のメッセージストアの実装を意味する設定を含めるのはエラーです。

デフォルト/グローバル設定	コンポーネントの動作
JdbcDriver	**JdbcStore**の使用
FileStorePath	**FileStore**の使用
SleepycatDatabaseDir	**SleepcatStore**の使用

メッセージファクトリー

メッセージファクトリーは、未加工の FIX メッセージからドメインオブジェクトを構築するために使用されます。デフォルトのメッセージファクトリーは DefaultMessageFactory です。ただし、高度なアプリケーションではカスタムメッセージファクトリーが必要になる場合があります。これは、QuickFIX/J コンポーネントに設定できます。

JMX

デフォルト/グローバル設定	コンポーネントの動作
UseJmx	**Y** の場合は、QuickFIX/J JMX を有効にします。

その他のデフォルト

コンポーネントは、QuickFIX/J 設定ファイルで通常必要な設定についてデフォルト設定を提供します。SessionStartTime および SessionEndTime はデフォルトで 00:00:00 で、セッションは自動的に開始および停止されません。HeartBtInt (heartbeat 間隔)はデフォルトで 30 秒です。

最小限のイニシエーター設定の例

> *[SESSION]*
> *ConnectionType=initiator*
> *BeginString=FIX.4.4*
> *SenderCompID=YOUR_SENDER*
> *TargetCompID=YOUR_TARGET*

INOUT メッセージ交換パターンの使用

Camel 2.8+

FIX プロトコルはイベント駆動型で非同期ですが、リクエストリプライメッセージ交換を表す特定のメッセージのペアがあります。InOut 交換パターンを使用するには、リクエストメッセージと単一のリプライメッセージをリクエストに 1 つ設定する必要があります。例には、OrderStatusRequest メッセージおよび UserRequest が含まれます。

コンシューマーの INOUT エクスチェンジの実装

QuickFIX/J enpoint URI にexchangePattern=InOut を追加します。以下の例の MessageOrderStatusService は、同期サービスメソッドを持つ Bean です。メソッドはリクエストへの応答を返します（この場合は ExecutionReport）。その後、要求元セッションに送信されます。

```
    from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:MARKET->TRADER&exchangePattern=InOut")

.filter(header(QuickfixjEndpoint.MESSAGE_TYPE_KEY).isEqualTo(MsgType.ORDER_STATUS_REQUEST))
        .bean(new MarketOrderStatusService());
```

*プロデューサーの **INOUT** エクスチェンジの実装*

*プロデューサーの場合、メッセージの送信は応答を受信するか、タイムアウトが発生するまでブロックされます。**FIX** でリプライメッセージを関連付ける標準的な方法はありません。そのため、**InOut** エクスチェンジのタイプごとに相関基準を定義する必要があります。相関基準とタイムアウトは、**Exchange** プロパティーを使用して指定できます。*

説明	キー文字列	キー定数	デフォルト
相関基準	"CorrelationCriteria"	QuickfixjProducer.CORRELATION_CRITERIA_KEY	なし
相関タイムアウト (Milliseconds)	"CorrelationTimeout"	QuickfixjProducer.CORRELATION_TIMEOUT_KEY	1000

*相関基準は **MessagePredicate** オブジェクトで定義されます。以下の例では、トランザクションタイプが **STATUS** で、**Order ID** が要求に一致する指定のセッションの **FIX ExecutionReport** を処理します。セッション **ID** は 要求 側の、送信者とターゲットの **CompID** フィールドは、応答を検索する際に逆になります。*

```
exchange.setProperty(QuickfixjProducer.CORRELATION_CRITERIA_KEY,
    new MessagePredicate(new SessionID(sessionID), MsgType.EXECUTION_REPORT)
        .withField(ExecTransType.FIELD, Integer.toString(ExecTransType.STATUS))
        .withField(OrderID.FIELD, request.getString(OrderID.FIELD)));
```

例

*ソースコードには、コンシューマーおよびプロデューサーの **InOut** エクスチェンジを示す **RequestReplyExample** という例が含まれています。この例では、注文ステータスリクエストを受け入れる単純な **HTTP** サーバーエンドポイントを作成します。**HTTP** リクエストは **FIX***

OrderStatusRequestMessage に変換され、相関基準で拡張され、クイックフィックスエンドポイントにルーティングされます。その後、応答は JSON 形式の文字列に変換され、Web 応答として提供される HTTP サーバーエンドポイントに送信されます。

Spring 設定が Camel 2.9 以降に変更されました。以下に例を示します。

SPRING の設定

Camel 2.6 - 2.8.x

QuickFIX/J コンポーネントには、Spring コンテキスト内でセッション設定を設定するための Spring FactoryBean が含まれています。QuickFIX/J セッション ID 文字列の型コンバーターも含まれています。以下の例は、両方のセッションのデフォルト設定を持つアクセプターおよびイニシエーターセッションの簡単な設定を示しています。

```
<!-- camel route -->
    <camelContext id="quickfixjContext" xmlns="http://camel.apache.org/schema/spring">
    <route>
    <from uri="quickfix:example"/>
    <filter>
    <simple>${in.header.EventCategory} == 'AppMessageReceived'</simple>
    <to uri="log:test"/>
    </filter>
    </route>
    </camelContext>

    <!-- quickfix component -->
    <bean id="quickfix" class="org.apache.camel.component.quickfixj.QuickfixjComponent">
    <property name="engineSettings">
    <util:map>
    <entry key="quickfix:example" value-ref="quickfixjSettings"/>
    </util:map>
    </property>
    <property name="messageFactory">
    <bean
class="org.apache.camel.component.quickfixj.QuickfixjSpringTest.CustomMessageFactory"/>
    </property>
    </bean>

    <!-- quickfix settings -->
    <bean id="quickfixjSettings"
    class="org.apache.camel.component.quickfixj.QuickfixjSettingsFactory">
    <property name="defaultSettings">
    <util:map>
    <entry key="SocketConnectProtocol" value="VM_PIPE"/>
    <entry key="SocketAcceptProtocol" value="VM_PIPE"/>
    <entry key="UseDataDictionary" value="N"/>
    </util:map>
```

```
</property>
<property name="sessionSettings">
<util:map>
<entry key="FIX.4.2:INITIATOR->ACCEPTOR">
<util:map>
<entry key="ConnectionType" value="initiator"/>
<entry key="SocketConnectHost" value="localhost"/>
<entry key="SocketConnectPort" value="5000"/>
</util:map>
</entry>
<entry key="FIX.4.2:ACCEPTOR->INITIATOR">
<util:map>
<entry key="ConnectionType" value="acceptor"/>
<entry key="SocketAcceptPort" value="5000"/>
</util:map>
</entry>
</util:map>
</property>
</bean>
```

Camel 2.9 以降

QuickFIX/J コンポーネントには、セッション設定を設定するための QuickfixjConfiguration クラスが含まれています。QuickFIX/J セッション ID 文字列の型コンバーターも含まれています。以下の例は、両方のセッションのデフォルト設定を持つアクセプターおよびイニシエーターセッションの簡単な設定を示しています。

```
<!-- camel route -->
<camelContext id="quickfixjContext" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="quickfix:example"/>
    <filter>
      <simple>${in.header.EventCategory} == 'AppMessageReceived'</simple>
      <to uri="log:test"/>
    </filter>
  </route>
</camelContext>

<!-- quickfix component -->
<bean id="quickfix" class="org.apache.camel.component.quickfixj.QuickfixjComponent">
  <property name="configurations">
    <util:map>
      <entry key="example" value-ref="quickfixjConfiguration"/>
    </util:map>
  </property>
  <property name="messageFactory">
    <bean
class="org.apache.camel.component.quickfixj.QuickfixjSpringTest.CustomMessageFactory"/>
  </property>
</bean>

<!-- quickfix settings -->
```

```
<bean id="quickfixjConfiguration"
class="org.apache.camel.component.quickfixj.QuickfixjConfiguration">
  <property name="defaultSettings">
    <util:map>
      <entry key="SocketConnectProtocol" value="VM_PIPE"/>
      <entry key="SocketAcceptProtocol" value="VM_PIPE"/>
      <entry key="UseDataDictionary" value="N"/>
    </util:map>
  </property>
  <property name="sessionSettings">
    <util:map>
      <entry key="FIX.4.2:INITIATOR->ACCEPTOR">
        <util:map>
          <entry key="ConnectionType" value="initiator"/>
          <entry key="SocketConnectHost" value="localhost"/>
          <entry key="SocketConnectPort" value="5000"/>
        </util:map>
      </entry>
      <entry key="FIX.4.2:ACCEPTOR->INITIATOR">
        <util:map>
          <entry key="ConnectionType" value="acceptor"/>
          <entry key="SocketAcceptPort" value="5000"/>
        </util:map>
      </entry>
    </util:map>
  </property>
</bean>
```

例外処理

メッセージの処理中に特定の例外が出力された場合、QuickFix/J の動作を変更できます。受信ログオン管理メッセージの処理中に RejectLogon 例外が出力された場合、ログオンは拒否されます。

通常、QuickFIX/J はログオンプロセスを自動的に処理します。ただし、送信ログオンメッセージを変更して、FIX に必要な認証情報を含める必要がある場合があります。ログオンメッセージの送信時に FIX ログオンメッセージのボディーが変更された場合(EventCategory={{AdminMessageSent}})、変更されたメッセージは対応するメッセージに送信されます。送信ログオンメッセージが 同期的 *に処理されることが重要です。非同期的に処理されている場合（別のスレッド上で）、FIX エンジンはコールバックメソッドが返されると、変更されていない送信メッセージを即座に送信します。*

修正シーケンス番号管理

同期 *エクスチェンジの処理中にアプリケーション例外が出力された場合、QuickFIX/J は受信 FIX メッセージシーケンス番号をインクリメントせず、対応するメッセージを再送信します。この FIX プロトコルの動作は、主にアプリケーションエラーではなく* トランスポート *エラーを処理することを目的としています。このメカニズムを使用してアプリケーションエラーを処理することに関連するリスクがあります。主なリスクは、メッセージが受信されるたびにアプリケーションエラーを繰り返し発生することです。より良い解決策は、処理の直前に受信メッセージ（データベース、JMS キュー）を保持す*

ることです。また、これにより、アプリケーションはエラーの発生時にメッセージを失うことなくメッセージを非同期に処理することもできます。

ログ先の FIX セッションにメッセージを送信することは可能ですが（メッセージはログオン時に送信されます）、通常はセッションがログに記録されるまで待つことが推奨されます。これにより、ログオンに必要なシーケンス番号の再同期手順がなくなります。セッションログオンを待つには、SessionLogon イベントカテゴリーを処理し、メッセージの送信を開始するようにアプリケーションに通知するルートを設定します。

FIX シーケンス番号管理の詳細については、FIX プロトコルの仕様と QuickFIX/J ドキュメントを参照してください。

ルートの例

QuickFIX/J コンポーネントのソースコード（テストサブディレクトリー）にはいくつかの例が含まれています。これらの例の 1 つは、ミリュレートの超過シミュレーションを実装します。この例では、URI スキームの "trade-executor" を使用するアプリケーションコンポーネントを定義します。

以下のルートは、トレードオフエグゼキューターセッションのメッセージを受信し、アプリケーションメッセージを trade executor コンポーネントに渡します。

```
from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:MARKET->TRADER").

filter(header(QuickfixjEndpoint.EVENT_CATEGORY_KEY).isEqualTo(QuickfixjEventCategory.
AppMessageReceived)).
    to("trade-executor:market");
```

トレードオフエクゼキューターコンポーネントは、トレードオフセッションにルーティングされるメッセージを生成します。エンドポイント URI にはセッション ID が指定されていないため、セッション ID は FIX メッセージ自体で設定する必要があります。

```
from("trade-executor:market").to("quickfix:examples/inprocess.cfg");
```

トレーターセッションは、市場から実行レポートメッセージを消費し、それらを処理します。

```
from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:TRADER->MARKET").

filter(header(QuickfixjEndpoint.MESSAGE_TYPE_KEY).isEqualTo(MsgType.EXECUTION_REP
ORT)).
    bean(new MyTradeExecutionProcessor());
```

QUICKFIX/J COMPONENT PRIOR TO CAMEL 2.5

Camel 2.0 以降で利用可能

クイックフィックス コンポーネントは、Java 用の QuickFIX/J エンジンの実装です。このエンジンでは、FIX プロトコル標準に従って銀行メッセージを交換するために使用される FIX サーバーに接続できます。

注記： コンポーネントを使用して、FIX サーバーにメッセージを送受信できます。

URI 形式

```
quickfix-server:config file
quickfix-client:config file
```

設定ファイル は、起動時にエンジンを設定するために使用されるクイックフィックス設定ファイルの場所（クラスパス内）です。

注記： クイック修正に使用できるパラメーターに関する情報は、QuickFIX/J Web サイトを参照してください。

FIX ゲートウェイにメッセージを送信する場合は、fixed server FIX メッセージと quickfix-client エンドポイントから受信するために quickfix-server エンドポイントを使用する必要があります。

エクスチェンジデータ形式

QuickFIX/J エンジンは、MINA をプロトコルレイヤーとして使用し、FIX エンジンゲートウェイでソケット接続を作成する CXF コンポーネントと似ています。

QuickFIX/J エンジンがメッセージを受信すると、camel エンドポイントによって次に受信される QuickFix.Message インスタンスを作成します。このオブジェクトは、最初にキー/値のペアデータのコレクションとしてフォーマットされた FIX メッセージから作成されるマッピングオブジェクトです。このオブジェクトを使用するか、またはメソッド toString を使用して元の FIX メッセージを取得できます。

注記： または、camel bindy dataformat を使用して FIX メッセージを独自の Java POJO に変換することもできます。

メッセージを QuickFix に送信する必要がある場合は、QuickFix.Message インスタンスを作成する必要があります。

LAZY CREATING ENGINES

Camel 2.12.3 以降では、QuickFix コンポーネントをレイジーにエンジンを作成および起動するように設定できます。その後、これらのオンマンドのみを起動します。たとえば、マスター/スレーブのクラスターに複数の Camel アプリケーションがある場合に使用できます。また、スレーブをスタンバイ状態にします。

サンプル

方向 : FIX ゲートウェイ

```
<route>
  <from uri="activemq:queue:fix"/>
  <bean ref="fixService" method="createFixMessage" /> // bean method in charge to transform message into a QuickFix.Message
  <to uri="quickfix-client:META-INF/quickfix/client.cfg" /> // Quickfix engine who will send the FIX messages to the gateway
</route>
```

方向 : FIX ゲートウェイから

```
<route>
  <from uri="quickfix-server:META-INF/quickfix/server.cfg"/> // QuickFix engine who will receive the message from FIX gateway
  <bean ref="fixService" method="parseFixMessage" /> // bean method parsing the QuickFix.Message
  <to uri="uri="activemq:queue:fix"/>" />
</route>
```

第132章 RABBITMQ

RABBITMQ コンポーネント

Camel 2.12 以降で利用可能

rabbitmq: コンポーネントを使用すると、RabbitMQ インスタンスからメッセージを生成および消費できます。RabbitMQ AMQP クライアントを使用して、このコンポーネントは汎用 AMQP コンポーネントに純粋な RabbitMQ アプローチを提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-rabbitmq</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
rabbitmq://hostname[:port]/exchangeName?[options]
```

hostname は、実行中の rabbitmq インスタンスまたはクラスターのホスト名です。port はオプションであり、指定されていない場合、デフォルトでは RabbitMQ クライアントのデフォルト(5672)に設定されます。交換名は、生成されたメッセージの送信先となるエクスチェンジを決定します。コンシューマーの場合、エクスチェンジ名はキューがバインドするエクスチェンジを決定します。

オプション

プロパティー	デフォルト	説明
autoAck	**true**	メッセージを自動承認するかどうか。
autoDelete	**true**	true の場合、エクスチェンジは使用されなくなったときに削除されます。

durable	**true**	永続エクスチェンジを宣言する場合（エクスチェンジはサーバーの再起動後も維持されます）。
queue	**ランダムな uuid**	メッセージを受信するキュー。
routingKey	**null**	コンシューマーキューをエクスチェンジにバインドするときに使用するルーティングキー。プロデューサールーティングキーの場合は、ヘッダーを設定します（ヘッダーセクションを参照）。
threadPoolSize	**10**	コンシューマーは、一定数のスレッドを持つスレッドプールエグゼキューターを使用します。この設定により、そのスレッド数を設定できます。
username	**null**	アクセスが認証された場合のユーザー名。
password	**null**	認証されたアクセスのパスワード。
vhost	**/**	チャネルの vhost。
exchangeType	**direct**	**Camel 2.12.2:** direct や topic などのエクスチェンジタイプ。
bridgeEndpoint	**false**	**Camel 2.12.3:** bridgeEndpoint が true の場合、プロデューサーは rabbitmq.EXCHANGE_NAME および "rabbitmq.ROUTING_KEY" のメッセージヘッダーを無視します。
addresses	**null**	**Camel 2.12.3:** このオプションを設定すると、camel-rabbitmq はオプションアドレスの設定に基づいて接続の作成を試みます。addresses 値は、server1:12345、server2:12345 などの文字列です。
connectionTimeout	**0**	**Camel 2.14:** 接続のタイムアウト。
requestedChannelMax	**0**	**Camel 2.14:** 接続要求されたチャネルの最大数（提供されたチャネルの最大数）。

requestedFrameMax	**0**	**Camel 2.14:** Connection requested frame max (max size of frame)
requestedHeartbeat	**0**	**Camel 2.14:** 接続が要求されたハートビート（大ツは秒単位）。
sslProtocol	**null**	**Camel 2.14:** 接続時に SSL を有効にし、許可される値は 'true'、'TLS'、および 'SSLv3' です。
trustManager	**null**	**Camel 2.14:** SSL トラストマネージャーを設定します。このオプションを有効にするには、SSL を有効にする必要があります。
clientProperties	**null**	**Camel 2.14:** 接続クライアントプロパティー（サーバーとネゴシエートするために使用されるクライアント情報）。
connectionFactory	**null**	**Camel 2.14:** カスタムの RabbitMQ 接続ファクトリー。このオプションを設定すると、URI に設定されたすべての接続オプション (connectionTimeout、requestedChannelMax….)は使用されません。
automaticRecoveryEnabled	**false**	**Camel 2.14:** 接続の自動リカバリーを有効にします（接続シャットダウンがアプリケーションによって開始されていない場合に自動リカバリーを実行する接続実装を使用）。
networkRecoveryInterval	**5000**	**Camel 2.14:** ネットワークの回復間隔（ネットワーク障害からの復旧時に使用される間隔）。
topologyRecoveryEnabled	**true**	**Camel 2.14:** 接続トポロジーリカバリーを有効にします（トポロジーリカバリーは実行されませんか ?）。
prefetchEnabled	**false**	**Camel 2.14:** RabbitMQConsumer 側で QoS を有効にします。**prefetchSize**、**prefetchCount**、**prefetchGlobal** のオプションを同時に指定する必要があります。
prefetchSize	**0**	**Camel 2.14:** サーバーが配信するコンテンツの最大量（オクテットで測定）、無制限の場合は 0。

prefetchCount	**0**	Camel 2.14: サーバーが配信するメッセージの最大数。無制限の場合は 0。
prefetchGlobal	**false**	Camel 2.14: 設定を各コンシューマーではなくチャネル全体に適用する必要がある場合。
declare	**true**	Camel 2.14: オプションが true の場合、Camel はエクスチェンジとキューの名前を宣言し、それらをバインドします。オプションが false の場合、Camel はサーバー上でエクスチェンジおよびキュー名を宣言しません。
concurrentConsumers	**1**	Camel 2.14: ブローカーから消費する同時コンシューマーの数(JMS コンポーネントの同じオプションと同様)。
deadLetterRoutingKey		Camel 2.14: デッドレターエクスチェンジのルーティングキー。
deadLetterExchange		Camel 2.14: デッドレターエクスチェンジの名前。
deadLetterExchangeType	**direct**	Camel 2.14: デッドレターエクスチェンジのタイプ。
channelPoolMaxSize	**10**	Camel 2.14.1 (Producer のみ): メッセージの送信に使用されるチャネルの最大数。
channelPoolMaxWait	**1000**	Camel 2.14.1 (Producer のみ): チャネルを待つ最大時間（ミリ秒単位）。
queueArgsConfigurer	**null**	Camel 2.15.1: キューを宣言するときに **Args** マップを設定するために使用できるカスタム **ArgsConfigurer** インスタンス。
exchangeArgsConfigurer	**null**	Camel 2.15.1: エクスチェンジの宣言時に **Args** マップを設定するために使用できるカスタム **ArgsConfigurer** インスタンス。

requestTimeout	**20000**	**Camel 2.16:** プロデューサーのみ。InOut Exchange Pattern （ミリ秒単位）の使用時に応答を待機するタイムアウト。デフォルトは 40 秒です。**requestTimeoutCheckerInterval** オプションも参照してください。
requestTimeoutCheckerInterval	**1000**	**Camel 2.16:** RabbitMQ 経由でリクエスト/リプライを行うときに Camel がタイムアウトしたエクスチェンジをチェックする頻度を設定します。デフォルトでは、Camel は 1 秒あたり 1 回チェックします。ただし、タイムアウトの発生時に迅速な反応が必要な場合は、この間隔を下げてより頻繁にチェックできます。タイムアウトは **requestTimeout** オプションによって決定されます。
transferException	**false**	**Camel 2.16:** 有効で、リクエスト応答メッセージング(InOut)を使用し、コンシューマー側でエクスチェンジが失敗した場合、原因となった例外は応答で **byte[]** として送り返されます。クライアントが Camel の場合、返される例外が再出力されます。これにより、Camel RabbitMQ をルーティングのブリッジとして使用できます。たとえば、永続キューを使用して堅牢なルーティングを有効にすることができます。キャッチされた例外はシリアライズ可能である必要があります。コンシューマー側の元の例外は、プロデューサーに返される場合に **org.apache.camel.RuntimeCamelException** などの外部例外でラップできます。
skipQueueDeclare	**false**	**Camel 2.16.1:** **true** の場合、プロデューサーはキューを宣言してバインドしません。これは、既存のルーティングキーを介してメッセージを転送するために使用できます。

publisherAcknowledgements	**false**	**Camel 2.17: true** の場合、メッセージはパブリッシャーの確認が有効になっている状態で公開されます。
publisherAcknowledgementsTimeout	**0**	**Camel 2.17:** RabbitMQ サーバーからの **basic.ack** 応答を待つ時間（ミリ秒単位）。

カスタム接続ファクトリー

```
<bean id="customConnectionFactory" class="com.rabbitmq.client.ConnectionFactory">
  <property name="host" value="localhost"/>
  <property name="port" value="5672"/>
  <property name="username" value="camel"/>
  <property name="password" value="bugsbunny"/>
</bean>
<camelContext>
  <route>
    <from uri="direct:rabbitMQEx2"/>
    <to uri="rabbitmq://localhost:5672/ex2?connectionFactory=#customConnectionFactory"/>
  </route>
</camelContext>
```

HEADERS

以下のヘッダーは、メッセージ消費時にエクスチェンジに設定されます。

プロパティー	値
rabbitmq.ROUTING_KEY	メッセージの受信に使用されたルーティングキー、またはメッセージの生成時に使用されるルーティングキー
rabbitmq.EXCHANGE_NAME	メッセージが受信されたエクスチェンジ
rabbitmq.DELIVERY_TAG	受信したメッセージの rabbitmq 配信タグ
rabbitmq.REQUEUE	**Camel 2.14.2:** これは、コンシューマーがメッセージの拒否を制御するために使用されます。コンシューマーがエクスチェンジの処理を完了し、エクスチェンジが失敗した場合、コンシューマーは RabbitMQ ブローカーからのメッセージを拒否します。このヘッダーの値はこの動作を制御します。**false** の場合、メッセージは破棄/停止文字されます。**true** の場合、メッセージは再度キューに置かれます。デフォルトは **false** です。

*以下のヘッダーはプロデューサーによって使用されます。**camel** エクスチェンジにこれらが設定されている場合、**RabbitMQ** メッセージに設定されます。*

プロパティー	値
rabbitmq.ROUTING_KEY	メッセージの送信時に使用されるルーティングキー
rabbitmq.EXCHANGE_NAME	メッセージが受信された、または送信されたエクスチェンジ
rabbitmq.CONTENT_TYPE	RabbitMQ メッセージに設定する contentType
rabbitmq.PRIORITY	RabbitMQ メッセージに設定する優先度ヘッダー
rabbitmq.CORRELATIONID	RabbitMQ メッセージに設定する correlationId
rabbitmq.MESSAGE_ID	RabbitMQ メッセージに設定するメッセージ ID
rabbitmq.DELIVERY_MODE	メッセージを永続化すべきかどうか
rabbitmq.USERID	RabbitMQ メッセージに設定する userId
rabbitmq.CLUSTERID	RabbitMQ メッセージに設定する clusterId
rabbitmq.REPLY_TO	RabbitMQ メッセージに設定する replyTo
rabbitmq.CONTENT_ENCODING	RabbitMQ メッセージに設定する contentEncoding
rabbitmq.TYPE	RabbitMQ メッセージに設定するタイプ
rabbitmq.EXPIRATION	RabbitMQ メッセージに設定する有効期限
rabbitmq.TIMESTAMP	RabbitMQ メッセージに設定するタイムスタンプ
rabbitmq.APP_ID	RabbitMQ メッセージに設定する appId

メッセージが受信されると、ヘッダーはコンシューマーによって設定されます。プロデューサーは、エクスチェンジが実行されると、ダウンストリームプロセッサーのヘッダーも設定します。実稼働前に設定されたヘッダーは、プロデューサーセットがオーバーライドされます。

メッセージボディー

*コンポーネントは、ボディーの **Camel** エクスチェンジを **rabbit mq** メッセージのボディーとして使*

用します。オブジェクトの Camel エクスチェンジはバイトアレイに変換できる必要があります。そうしないと、プロデューサーはサポートされていないボディータイプの例外を出力します。

サンプル

ルーティングキー B を使用してエクスチェンジ A にバインドされているキューからメッセージを受信するには、以下を実行します。

```
from("rabbitmq://localhost/A?routingKey=B")
```

自動承認が無効になっている単一のスレッドを持つキューからメッセージを受信するには、以下を行います。

```
from("rabbitmq://localhost/A?routingKey=B&threadPoolSize=1&autoAck=false")
```

C というエクスチェンジにメッセージを送信するには、以下を行います。

```
...to("rabbitmq://localhost/B")
```

第133章 REF

REF コンポーネント

ref: コンポーネントは、レジストリーにバインドされる既存のエンドポイントの検索に使用されます。

URI 形式

```
ref:someName
```

someName はレジストリー内のエンドポイントの名前です（通常は、常に Spring レジストリーではありません）。Spring レジストリーを使用している場合、someName は Spring レジストリー内のエンドポイントの Bean ID になります。

ランタイムルックアップ

このコンポーネントは、実行時に URI を計算できるレジストリーのエンドポイントの動的検出が必要な場合に使用できます。次に、以下のコードを使用してエンドポイントを検索できます。

```
// lookup the endpoint
String myEndpointRef = "bigspenderOrder";
Endpoint endpoint = context.getEndpoint("ref:" + myEndpointRef);

Producer producer = endpoint.createProducer();
Exchange exchange = producer.createExchange();
exchange.getIn().setBody(payloadToSend);
// send the exchange
producer.process(exchange);
...
```

また、以下のようなレジストリーでエンドポイントの一覧を定義することも可能です。

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
   <endpoint id="normalOrder" uri="activemq:order.slow"/>
   <endpoint id="bigspenderOrder" uri="activemq:order.high"/>
   ...
</camelContext>
```

例

以下の例では、URI の ref: を使用して、Spring ID endpoint2 でエンドポイントを参照します。

```
<bean id="mybean" class="org.apache.camel.spring.example.DummyBean">
  <property name="endpoint" ref="endpoint1"/>
</bean>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" disabled="true"/>
  <endpoint id="endpoint1" uri="direct:start"/>
  <endpoint id="endpoint2" uri="mock:end"/>

  <route>
    <from ref="endpoint1"/>
    <to uri="ref:endpoint2"/>
  </route>
</camelContext>
```

当然ながら、代わりに ref 属性を使用できます。

```
<to ref="endpoint2"/>
```

これは、記述のより一般的な方法です。

第134章 REST

REST コンポーネント

Camel 2.14 から利用可能

REST コンポーネントを使用すると、Apache Camel Development Guide のDefining Services with REST DSL セクションを使用して REST エンドポイントを定義し、他の Camel コンポーネントを REST トランスポートとして接続すること ができます。

URI 形式

```
rest://method:path[:uriTemplate]?[options]
```

URI オプション

名前	デフォルト値	説明
method		get、post、put、patch、delete、head、trace、connect、または のいずれかのオプションである HTTP メソッド。
path		REST 構文をサポートするベースパス。例については、以下を参照してください。
uriTemplate		REST 構文をサポートする URI テンプレート。例については、以下を参照してください。
consumes		'text/xml' や application/json などのメディアタイプが、この REST サービスを受け付けます。デフォルトでは、すべての種類のタイプを受け入れます。
produces		'text/xml' または 'application/json' などのメディアタイプが、この REST サービスが返されます。

PATH および URITEMPLATE 構文

path および uriTemplate オプションは、パラメーターのサポートを使用して REST コンテキストパスを定義する REST 構文を使用して定義されます。

ヒント

uriTemplate が設定されていない場合、path オプションも同じように動作します。path のみを設定する場合や、両方のオプションを設定した場合は問題ありません。REST では、path と uriTemplate の両方を設定するのが一般的なプラクティスです。

以下は、パスのみを使用した Camel ルートです。

```
from("rest:get:hello")
  .transform().constant("Bye World");
```

以下のルートは、キーme を持つ Camel ヘッダーにマップされる パラメーターを使用します。

```
from("rest:get:hello/{me}")
  .transform().simple("Bye ${header.me}");
```

以下の例では、ベースパスを hello に設定し、uriTemplates を使用して 2 つの REST サービスを設定しています。

```
from("rest:get:hello:/{me}")
  .transform().simple("Hi ${header.me}");

from("rest:get:hello:/french/{me}")
  .transform().simple("Bonjour ${header.me}");
```

その他の例

Apache Camel Development Guide のDefining Services with REST DSLを参照してください。このセクションで は、Rest DSL を使用してこれらの設定を適切な RESTful 方法で定義する方法を説明します。

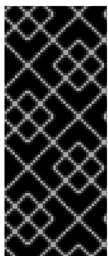
Apache Camel ディストリビューションには camel-example-servlet-rest-tomcat の例があり、Apache Camel Development Guide のDefining Services with REST DSL セクションと Apache Tomcat にデプロイできるトランスポートとして、または同様の Web コンテナーを使用する方法を説明します。 146章SERVLET

第135章 RESTLET

RESTLET コンポーネント

Restlet コンポーネントは、RESTful リソースを消費および生成するための Restletベースの エンドポイント を提供します。

Transport Layer Security (TLS)を使用するように Restlet コンポーネントを設定するには、Security Guide の Configuring Transport Security for Camel Components の章を参照してください。



重要

Restlet コンポーネントはデフォルトで非同期モードを有効にしますが、この設定はパフォーマンスに影響が出るように見えます。これが問題である場合は、エンドポイント URI にオプション synchronous=true を設定してパフォーマンスを向上できます。

URI 形式

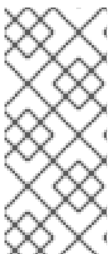
```
restlet:restletUrl[?options]
```

restletUrl の形式：

```
protocol://hostname[:port][/resourcePattern]
```

Restlet はプロトコルとアプリケーションの懸念の切り離されます。Restlet Engine の参照実装は、多くのプロトコルをサポートしています。ただし、HTTP プロトコルのみをテストしました。デフォルトのポートはポート 80 です。まだプロトコルに基づいてデフォルトのポートを自動的に切り替えません。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。



注記

Restlet はヘッダーを理解する際に大文字と小文字が区別されるようです。たとえば、content-type を使用するには、Content-Type を使用します。location には Location などを使用します。

オプション

名前	デフォルト値	説明
headerFilterStrategy=#refName	**RestletHeaderFilterStrategy** のインスタンス	Camel Registry のヘッダーフィルターストラテジーを参照するには、**#** 表記 (**headerFilterStrategy=#refName**)を使用します。ストラテジーは、**HeaderFilterStrategyAware** の場合、restlet バインディングにプラグインされます。
restletBinding=# refName	**DefaultRestletBinding**のインスタンス	Camel Registry の **RestletBinding** オブジェクトの Bean ID。
restletMethod	**GET**	プロデューサーエンドポイントでは、使用するリクエストメソッドを指定します。コンシューマーエンドポイントで、エンドポイントが **restletMethod** 要求のみを消費することを指定します。文字列値は **Method.valueOf (String)** メソッドによって **org.restlet.data.Method** に変換されます。
restletMethods	なし	**コンシューマーは、**restlet **コンシューマーエンドポイントによって提供されるコンマ(** **restletMethods=post,put**)で区**切られた 1 つ以上のメソッドのみ**を指定します。**restletMethod** および **restletMethods** オプションの両方を指定すると、**restletMethod** 設定は無視されます。
restletRealm	**null**	# 表記 (**restletRealm=#***refName*)を使用して、Camel レジストリーでレルムマップの Bean ID を指定します。

restletUriPatterns=#refName	なし	**コンシューマーは** restlet **コンシューマーエンドポイントによって処理される 1 つ以上の URI テンプレートのみ** を指定し、**#** 表記を使用して Camel レジストリーの **List<String** > を参照します。URI パターンがエンドポイント URI に定義されている場合、エンドポイントに定義された URI パターンと **restletUriPatterns** オプションの両方が受け入れられます。
throwExceptionOnFailure (2.6 以降)	**true**	プロデューサーは、プロデューサーの失敗時に例外 **のみ** を出力します。
connectionTimeout	300000	**Camel 2.12.3 Producer のみ 以降**、接続がタイムアウトした場合、クライアントは接続をタイムアウトし、無制限の待機時間は 0 になります。
socketTimeout	300000	**Camel 2.12.3 Producer はクライアントソケット受信タイムアウトのみ であるため**、無制限の待機時間は 0 になります。
disableStreamCache	**false**	**Camel 2.14:** Jetty からの raw 入力ストリームがキャッシュされているかどうかを判断します(Camel はストリームをファイル、ストリームキャッシュ)キャッシュにストリームします。http://camel.apache.org/stream-caching.htmlデフォルトでは、Camel は Jetty 入力ストリームをキャッシュして複数回読み取りし、Camel がストリームからすべてのデータを取得できるようにします。ただし、このオプションを **true** に設定することができます。たとえば、ファイルや他の永続ストアに直接ストリーミングするなど、raw ストリームにアクセスする必要がある場合などです。DefaultRestletBinding は、リクエスト入力ストリームをストリームキャッシュにコピーし、このオプションが **false** の場合、ストリームを複数回読み取るようにメッセージボディーに配置します。

コンポーネントオプション

***Restlet** コンポーネントは以下のオプションで設定できます。これらは コンポーネント オプションであり、エンドポイントでは設定できないことに注意してください。詳細は、以下を参照してください。*

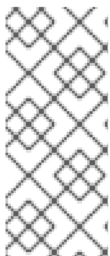
名前	デフォルト値	説明
controllerDaemon	**true**	Camel 2.10: **コントローラースレッドがデーモンであるべきかどうかを示します** (JVM の終了をブロックしません)。
controllerSleepTimeMs	**100**	Camel 2.10: コントローラースレッドが各制御間でスリープ状態になる時間。
inboundBufferSize	**8192**	Camel 2.10: メッセージの読み取り時のバッファーのサイズ。
minThreads	**1**	Camel 2.10: サービス要求を待機する最小スレッド。
maxThreads	**10**	Camel 2.10: リクエストを処理するスレッドの最大数。
lowThreads	**8**	Camel 2.13: コネクターがオーバーロードされているとみなされるタイミングを決定するワーカースレッドの数。
maxQueued	**0**	Camel 2.13: サービスに利用可能なワーカースレッドがなかった場合にキューに入れることができる最大呼び出し数。値が 0 の場合、キューは使用されず、ワーカースレッドが即座に利用できない場合は呼び出しは拒否されます。値が -1 の場合、バインドされていないキューが使用され、呼び出しは拒否されます。
maxConnectionsPerHost	**-1**	Camel 2.10: ホストごとの同時接続の最大数(IP アドレス)。
maxTotalConnections	**-1**	Camel 2.10: 合計同時接続の最大数。
outboundBufferSize	**8192**	Camel 2.10: メッセージの書き込み時のバッファーのサイズ。
persistingConnections	**true**	Camel 2.10: 呼び出しの後に接続を維持する必要があるかどうかを示します。

pipeliningConnections	**false**	**Camel 2.10:** パイプライン接続をサポートされるかどうかを示します。
threadMaxIdleTimeMs	**60000**	**Camel 2.10:** 収集される前にアイドル状態のスレッドが操作を待機する時間。
useForwardedForHeader	**false**	**Camel 2.10:** 一般的なプロキシーおよびキャッシュでサポートされる X-Forwarded-For ヘッダーを検索し、それを使用して Request.getClientAddresses （）メソッドの結果を設定します。この情報は、ローカルネットワーク内の中間コンポーネントに対してのみ安全です。他のアドレスは偽のヘッダーを設定することで簡単に変更でき、深刻なセキュリティーチェックでは信頼できません。
reuseAddress	**true**	**Camel 2.10.5/2.11.1:** SO_REUSEADDR ソケットオプションを有効/無効にします。詳細は、java.io.ServerSocket#reuseAddress プロパティーを参照してください。
disableStreamCache	**false**	**Camel 2. 14: Jetty からの raw 入力ストリームがキャッシュされているかどうかを判断します (Camel はストリームをファイル、ストリームキャッシュ) キャッシュにストリームを読み取ります。デフォルトでは、Camel は Jetty 入力ストリームをキャッシュして複数回読み取りし、Camel がストリームからすべてのデータを取得できるようにします。ただし、このオプションをファイルや他の永続ストアに直接ストリーミングするなど、raw ストリームにアクセスする必要がある場合などに設定することができます。true**DefaultRestletBinding は、要求入力ストリームをストリームキャッシュにコピーし、このオプションが 複数回ストリームの読み取りをサポートする場合はメッセージボディーに配置します。 **false**

メッセージヘッダー

名前	タイプ	説明
CamelContentType	**文字列**	アプリケーション/プロセッサーによって OUT メッセージに設定できるコンテンツタイプを指定します。値は、応答メッセージの **content-type** です。このヘッダーが設定されていない場合、**content-type** は OUT メッセージボディーのオブジェクトタイプに基づきます。Camel 2.3 以降では、Camel IN メッセージに Content-Type ヘッダーが指定されている場合、ヘッダーの値は Restlet リクエストメッセージのコンテンツタイプを決定します。nbsp;それ以外の場合は、デフォルトで application/x-www-form-urlencoded' になります。リリース 2.3 よりも前のバージョンでは、要求コンテンツタイプのデフォルトを変更することはできません。
CamelAcceptContentType	**文字列**	**Camel 2.9.3 以降：** 2.10.0: HTTP Accept リクエストヘッダー。
CamelHttpMethod	**文字列**	HTTP リクエストメソッド。これは IN メッセージヘッダーで設定されます。
CamelHttpQuery	**文字列**	リクエスト URI のクエリー文字列。restlet コンポーネントがリクエストを受信すると、**DefaultRestletBinding** によって IN メッセージに設定されます。
CamelHttpResponseCode	**文字列** または **整数**	応答コードは、アプリケーション/プロセッサーによって OUT メッセージに設定できます。値は、応答メッセージの応答コードです。このヘッダーが設定されていない場合、応答コードは restlet ランタイムエンジンによって設定されます。
CamelHttpUri	**文字列**	HTTP 要求 URI。これは IN メッセージヘッダーで設定されます。

CamelRestletLogin	**文字列**	Basic 認証のログイン名。これは、アプリケーションによって IN メッセージで設定され、Apache Camel による restlet リクエストヘッダーの前にフィルターリングされます。
CamelRestletPassword	**文字列**	Basic 認証のパスワード名。これは、アプリケーションによって IN メッセージで設定され、Apache Camel による restlet リクエストヘッダーの前にフィルターリングされます。
CamelRestletRequest	**Request**	**Camel 2.8:** すべてのリクエスト詳細を保持する **org.restlet.Request** オブジェクト。
CamelRestletResponse	**応答**	**Camel 2.8:** **org.restlet.Response** オブジェクト。これを使用して、Restlet から API を使用して応答を作成できます。以下の例を参照してください。
org.restlet.*		Apache Camel IN ヘッダーに伝播される Restlet メッセージの属性。
cache-control	**文字列** または **リスト <CacheDirective>**	**Camel 2.11:** ユーザーは、camel メッセージヘッダーから、String 値または List of CacheDirective of Restlet で設定できます。



注記

*基礎となる **Restlet** 実装は、ヘッダー名の解析時に大文字と小文字を区別します。たとえば、**content-type** ヘッダーを設定するには、**Content-Type** を指定し、**location** には **Location** を指定します。*

メッセージボディー

__Apache Camel__ は、__OUT__ ボディーの外部サーバーからの __restlet__ 応答を保存します。__IN__ メッセージからのヘッダーはすべて __OUT__ メッセージにコピーされ、ルーティング中にヘッダーが保持されます。

*認証のある **RESTLET** エンドポイント*

以下のルートは、 http://localhost:8080 で POST リクエストをリッスンする restlet コンシューマーエンドポイントを開始します。プロセッサーは、リクエストボディーと id ヘッダーの値をエコーする応答を作成します。

```
from("restlet:http://localhost:9080/securedOrders?
restletMethod=post&restletRealm=#realm").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        exchange.getOut().setBody(
                "received [" + exchange.getIn().getBody()
                + "] as an order id = "
                + exchange.getIn().getHeader("id"));
    }
});
```

URI クエリーの restletRealm 設定は、レジストリーで Realm Map を検索するために使用されます。このオプションを指定すると、restlet コンシューマーは情報を使用してユーザーログインを認証します。認証された 要求のみがリソースにアクセスできます。この例では、レジストリーとして機能する Spring アプリケーションコンテキストを作成します。レルムマップの Bean ID は restletRealmRef と一致する必要があります。

```
<util:map id="realm">
 <entry key="admin" value="foo" />
 <entry key="bar" value="foo" />
</util:map>
```

以下の例は、 http://localhost:8080 のサーバーにリクエストを送信する direct エンドポイントを開始します（つまり、restlet コンシューマーエンドポイント）。

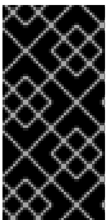
```
// Note: restletMethod and restletRealmRef are stripped
// from the query before a request is sent as they are
// only processed by Camel.
from("direct:start-auth").to("restlet:http://localhost:9080/securedOrders?
restletMethod=post");
```

必要なのはこれだけです。リクエストを送信し、restlet コンポーネントを試す準備ができました。

```
final String id = "89531";

Map<String, Object> headers = new HashMap<String, Object>();
headers.put(RestletConstants.RESTLET_LOGIN, "admin");
headers.put(RestletConstants.RESTLET_PASSWORD, "foo");
headers.put("id", id);

String response = (String) template.requestBodyAndHeaders("direct:start-auth",
        "<order foo='1'/>", headers);
```

サンプルクライアントは、以下のヘッダーを使用して direct:start-auth エンドポイントにリクエストを送信します。

- CamelRestletLogin (Apache Camel によって内部で使用される)
- CamelRestletPassword (Apache Camel によって内部で使用される)
- id（アプリケーションヘッダー）



注記

org.apache.camel.restlet.auth.login および org.apache.camel.restlet.auth.password は Restlet ヘッダーとして伝播されません。

サンプルクライアントは以下のような応答を取得します。

```
received [<order foo='1'/>] as an order id = 89531
```

単一の RESTLET エンドポイントから複数のメソッドおよび URI テンプレート(2.0 以降)を提供

restletMethods オプションを使用して、複数の HTTP メソッドを提供する単一のルートを作成できます。このスニペットには、ヘッダーからリクエストメソッドを取得する方法も示されています。

```
from("restlet:http://localhost:9080/users/{username}?restletMethods=post,get,put")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // echo the method
            exchange.getOut().setBody(exchange.getIn().getHeader(Exchange.HTTP_METHOD,
                                                     String.class));

        }
    });
```

次のスニペットは、複数のメソッドを提供する他に、restletUriPatterns オプションを使用して複数の URI テンプレートをサポートするエンドポイントを作成する方法を示しています。リクエスト URI は IN メッセージのヘッダーでも利用できます。URI パターンがエンドポイント URI で定義されている場合（このサンプルでは）、エンドポイントに定義された URI パターンと restletUriPatterns オプションの両方が尊重されます。

```
from("restlet:http://localhost:9080?
restletMethods=post,get&restletUriPatterns=#uriTemplates")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // echo the method
            String uri = exchange.getIn().getHeader(Exchange.HTTP_URI, String.class);
            String out = exchange.getIn().getHeader(Exchange.HTTP_METHOD, String.class);
            if ("http://localhost:9080/users/homer".equals(uri)) {
                exchange.getOut().setBody(out + " " + exchange.getIn().getHeader("username",
String.class));
            } else if ("http://localhost:9080/atom/collection/foo/component/bar".equals(uri)) {
                exchange.getOut().setBody(out + " " + exchange.getIn().getHeader("id", String.class)
                                + " " + exchange.getIn().getHeader("cid", String.class));

            }

        }
    });
```

restletUriPatterns=#uriTemplates オプションは、Spring XML 設定で定義され た List<String > Bean を参照します。

```
<util:list id="uriTemplates">
    <value>/users/{username}</value>
    <value>/atom/collection/{id}/component/{cid}</value>
</util:list>
```

RESTLET API を使用した応答の設定

Camel 2.8 から利用可能

org.restlet.Response API を使用して応答を設定することができます。これにより、Restlet API に完全にアクセスでき、応答を詳細に制御できます。インライン化された Camel プロセッサー からの応答を生成する以下のルートスニペットを参照してください。

```
from("restlet:http://localhost:" + portNum + "/users/{id}/like/{beer}")
     .process(new Processor() {
         public void process(Exchange exchange) throws Exception {
             // the Restlet request should be available if neeeded
             Request request = exchange.getIn().getHeader(RestletConstants.RESTLET_REQUEST,
Request.class);
             assertNotNull("Restlet Request", request);

             // use Restlet API to create the response
```

```
            Response response =
exchange.getIn().getHeader(RestletConstants.RESTLET_RESPONSE, Response.class);
            assertNotNull("Restlet Response", response);
            response.setStatus(Status.SUCCESS_OK);
            response.setEntity("<response>Beer is Good</response>", MediaType.TEXT_XML);
            exchange.getOut().setBody(response);
        }
    });
```

コンポーネントの最大スレッドの設定

最大スレッドオプションを設定するには、以下のようにコンポーネントでこれを実行する必要があります。

```
<bean id="restlet" class="org.apache.camel.component.restlet.RestletComponent">
  <property name="maxThreads" value="100"/>
</bean>
```

WEBAPP 内の RESTLET サーブレットの使用

Camel 2.8 では、サーブレットコンテナー内で Restlet アプリケーションを設定し、サブクラス化された SpringServerServlet を使用すると、Restlet コンポーネントを注入する ことで Camel 内の設定を有効にする 3 つの方法 があります。

サーブレットコンテナー内で Restlet サーブレットを使用すると、URI の相対パス（ハードコーディングされた絶対 URI の制限を変更する）や、（新しいポートで別のサーバープロセスを生成する必要のない）ホスティングサーブレットコンテナーが受信要求を処理するよう設定できます。

を設定するには、以下を camel-context.xml; に追加します。

```
<camelContext>
  <route id="RS_RestletDemo">
    <from uri="restlet:/demo/{id}" />
    <transform>
      <simple>Request type : ${header.CamelHttpMethod} and ID : ${header.id}</simple>
    </transform>
  </route>
</camelContext>

<bean id="RestletComponent" class="org.restlet.Component" />

<bean id="RestletComponentService"
class="org.apache.camel.component.restlet.RestletComponent">
  <constructor-arg index="0">
```

```
    <ref bean="RestletComponent" />
  </constructor-arg>
</bean>
```

web.xml に追加します。

```
<!-- Restlet Servlet -->
<servlet>
  <servlet-name>RestletServlet</servlet-name>
  <servlet-class>org.restlet.ext.spring.SpringServerServlet</servlet-class>
  <init-param>
    <param-name>org.restlet.component</param-name>
    <param-value>RestletComponent</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>RestletServlet</servlet-name>
  <url-pattern>/rs/*</url-pattern>
</servlet-mapping>
```

これにより、http://localhost:8080/mywebapp/rs/demo/1234 でデプロイされたルートにアクセスできます。

localhost:8080 はサーブレットコンテナー mywebapp のサーバーおよびポートで、デプロイされた webapp Your ブラウザーの名前で、以下の内容が表示されます。

```
"Request type : GET and ID : 1234"
```

Maven pom.xml ファイルで実行できる restlet に、Spring エクステンションの依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.restlet.jee</groupId>
  <artifactId>org.restlet.ext.spring</artifactId>
  <version>${restlet-version}</version>
</dependency>
```

さらに、restlet maven リポジトリーに依存関係も追加する必要があります。

```
<repository>
   <id>maven-restlet</id>
   <name>Public online Restlet repository</name>
   <url>http://maven.restlet.org</url>
</repository>
```

-

第136章 RMI

RMI コンポーネント

rmi: コンポーネントは Exchangeを RMI プロトコル(JRMP)にバインドします。

このバインディングは RMI のみを使用しているため、通常の RMI ルールは引き続き呼び出すことのできるメソッドに適用されます。このコンポーネントは、Remote インターフェイスを拡張するインターフェイスからのメソッド呼び出しを持つエクスチェンジのみをサポートします。メソッドのすべてのパラメーターは、Serializable または Remote オブジェクトのいずれかである必要があります。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-rmi</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
rmi://rmi-regisitry-host:rmi-registry-port/registry-path[?options]
```

以下に例を示します。

```
rmi://localhost:1099/path/to/service
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
メソッド	null	Apache Camel 1.3 では、呼び出すメソッドの名前を設定できます。

remoteInterfaces	**null**	XML DSL では、**Camel 2.7 からこのオプションを使用できるようになり** ました。コンマで区切られたインターフェイス名の一覧を指定できます。

使用

RMI レジストリーに登録されている既存の RMI サービスを呼び出すには、以下のようなルートを作成します。

```
from("pojo:foo").to("rmi://localhost:1099/foo");
```

RMI レジストリーで既存の Camel プロセッサーまたはサービスをバインドするには、以下のように RMI エンドポイントを定義します。

```
RmiEndpoint endpoint= (RmiEndpoint) endpoint("rmi://localhost:1099/bar");
endpoint.setRemoteInterfaces(ISay.class);
from(endpoint).to("pojo:bar");
```

RMI コンシューマーエンドポイントをバインドする場合は、公開されている リモート インターフェイスを指定する必要があります。

XML DSL では、以下のように Camel 2.7 以降実行 できます。

```
<camel:route>
    <from uri="rmi://localhost:37541/helloServiceBean?
remoteInterfaces=org.apache.camel.example.osgi.HelloService"/>
    <to uri="bean:helloServiceBean"/>
</camel:route>
```

第137章 ROUTEBOX

ROUTEBOX コンポーネント

Camel 2.6 以降で利用可能

ROUTEBOX が変更可能

Routebox コンポーネントは今後のリリースで再検討され、さらに簡素化され、より直感的でユーザーフレンドリーなものになります。関連する Context コンポーネントは、より簡単なコンポーネントとして考慮される可能性があります。このコンポーネントは、コンテキスト が優先されるため非推奨になる可能性が あり ます。

routebox コンポーネントを使用すると、自動作成またはユーザーが挿入された Camel コンテキストでホストされる Camel ルートのコレクションにカプセル化およびストラテジーベースの間接サービスを提供する特殊なエンドポイントを作成できます。

Routebox エンドポイントは、Camel ルートで直接呼び出すことができる Camel エンドポイントです。routebox エンドポイントは、以下の主要な機能 * カプセル化を実行します。ブラックボックスとして機能し、内部 Camel コンテキストに保存された Camel ルートのコレクションをホストします。内部コンテキストは routebox コンポーネントの制御下にあり、JVM バインド です。* ストラテジーベースの間接化 - ユーザー定義の内部ルーティングストラテジーまたはディスパッチマップに基づいて、camel ルートとともに routebox エンドポイントに送信される直接ペイロード。* Exchange propagation - routebox エンドポイントによって変更されたエクスチェンジを camel ルートの次のセグメントに転送します。

routebox コンポーネントは、コンシューマーエンドポイントとプロデューサーエンドポイントの両方をサポートします。

プロデューサーエンドポイントは 2 つのフレーバーです。* 受信したリクエストを外部の routebox コンシューマーエンドポイントに送信またはディスパッチするプロデューサーです。内部埋め込み Camel コンテキストでルートを直接呼び出すプロデューサーにより、外部コンシューマーにリクエストを送信しません。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-routebox</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ROUTEBOX エンドポイントの要件

routebox コンポーネントは、必要な複雑な環境での統合を容易にするように設計されています。

- ***ルート と の大規模なコレクション***
- ***さまざまな方法で統合を必要とするエンドポイントテクノロジーのセットが関係します。***

このような環境では、Camel ルート間での階層化を効果的に整理することで、統合ソリューションを作成する必要があることがよくあります。

- ***粒度の細かいルートや、統合フォーカスエリアを表す Routebox エンドポイントとして公開される内部または下位レベルのルートの集約コレクション。以下に例を示します。***

フォーカスエリア	粒度の細かいルートの例
部門中心	HR ルート、営業ルートなど
サプライチェーンと B2B Focus	ルート、フィルフィルメントルート、サードパーティーサービスなどの配送
テクノロジーの焦点	データベースルート、JMS ルート、スケジュール済みバッチルートなど

- ***粒度の細かいルート：単数で特定のビジネスや統合パターンを実行するルート。***

粒度の細かいルートで Routebox エンドポイントに送信される要求は、要求を内部の詳細なルートに委譲して特定の統合目的を達成し、最終的な内部結果を収集して、粒度の細かいルートで次のステップに進むことができます。

URI 形式

> ***routebox:routeboxname[?options]***

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
dispatchStrategy	**null**	インターフェイス **org.apache.camel.component.routebox.strategy.RouteboxDispatchStrategy**を実装するオブジェクト値と一致する Camel レジストリーのキーを表す文字列
dispatchMap	**null**	HashMap<String, String> タイプのオブジェクト値と一致する Camel レジストリーのキーを表す文字列。HashMap キーには、エクスチェンジヘッダー **ROUTE_DISPATCH_KEY** に設定された値に対して一致する文字列が含まれている必要があります。HashMap 値には、リクエストを転送する必要のある内部ルートコンシューマー URI が含まれる必要があります。

innerContext	**自動作成**	**org.apache.camel.CamelContext** 型のオブジェクト値と一致する Camel レジストリーのキーを表す文字列。ユーザーによって CamelContext が提供されていない場合、内部ルートのデプロイのために CamelContext が自動的に作成されます。
innerRegistry	**null**	Camel レジストリーのキーを表す文字列は、インターフェイス **org.apache.camel.spi.Registry** を実装するオブジェクト値と一致します。エンドポイントを作成するために内部ルートで使用されるレジストリー値を使用する場合は、innerRegistry パラメーターを指定する必要があります。
routeBuilders	**空のリスト**	**List<org.apache.camel.builder.RouteBuilder> タイプのオブジェクト値に一致する Camel レジストリーのキーを表す文字列。**ユーザーが内部ルートに先行する innerContexts を提供しない場合、routeBuilders オプションは内部ルートを含む RouteBuilder の空でないリストとして指定する必要があります。
innerProtocol	**Direct**	Routebox コンポーネントによって内部で使用される Protocol。Direct または SEDA にすることができます。**Routebox コンポーネントは、現在 JVM にバインドされるプロトコルを提供します。**
sendToConsumer	**true**	Producer エンドポイントがリクエストを外部 routebox コンシューマーに送信するかどうかを指定します。設定が false の場合、Producer は埋め込みの内部コンテキストを作成し、内部で要求を処理します。
forkContext	**true**	Routebox コンポーネントによって内部で使用される Protocol。Direct または SEDA にすることができます。**Routebox コンポーネントは、現在 JVM にバインドされるプロトコルを提供します。**

threads	20	リクエストを受信するために routebox が使用するスレッドの数。innerProtocol SEDA のみに適用可能な設定。
queueSize	無制限	要求を受信する固定サイズキューを作成します。innerProtocol SEDA のみに適用可能な設定。

ROUTEBOX へのメッセージの送受信

リクエストを送信する前に、以下に示すように、必要な URI パラメーターをレジストリーにロードして routebox を適切に設定する必要があります。Spring の場合、必要な Bean が正しく宣言されると、レジストリーは自動的に Camel によって入力されます。

ステップ 1: レジストリーへの内部ルート詳細の読み込み

```
@Override
protected JndiRegistry createRegistry() throws Exception {
    JndiRegistry registry = new JndiRegistry(createJndiContext());

    // Wire the routeDefinitions & dispatchStrategy to the outer camelContext where the
routebox is declared
    List<RouteBuilder> routes = new ArrayList<RouteBuilder>();
    routes.add(new SimpleRouteBuilder());
    registry.bind("registry", createInnerRegistry());
    registry.bind("routes", routes);

    // Wire a dispatch map to registry
    HashMap<String, String> map = new HashMap<String, String>();
    map.put("addToCatalog", "seda:addToCatalog");
    map.put("findBook", "seda:findBook");
    registry.bind("map", map);

    // Alternatively wiring a dispatch strategy to the registry
    registry.bind("strategy", new SimpleRouteDispatchStrategy());

    return registry;
}
```

```
private JndiRegistry createInnerRegistry() throws Exception {
    JndiRegistry innerRegistry = new JndiRegistry(createJndiContext());
    BookCatalog catalogBean = new BookCatalog();
    innerRegistry.bind("library", catalogBean);

    return innerRegistry;
}
...
CamelContext context = new DefaultCamelContext(createRegistry());
```

ステップ 2: ディスパッチマップの代わりにディスパッチストラテジーを使用するオプション

ディスパッチストラテジーを使用するには、以下の例のように **org.apache.camel.component.routebox.strategy.RouteboxDispatchStrategy** インターフェイスを実装する必要があります。

```
public class SimpleRouteDispatchStrategy implements RouteboxDispatchStrategy {

    /* (non-Javadoc)
     * @see
org.apache.camel.component.routebox.strategy.RouteboxDispatchStrategy#selectDestination
Uri(java.util.List, org.apache.camel.Exchange)
     */
    public URI selectDestinationUri(List<URI> activeDestinations,
            Exchange exchange) {
        URI dispatchDestination = null;

        String operation = exchange.getIn().getHeader("ROUTE_DISPATCH_KEY", String.class);
        for (URI destination : activeDestinations) {
            if (destination.toASCIIString().equalsIgnoreCase("seda:" + operation)) {
                dispatchDestination = destination;
                break;
            }
        }

        return dispatchDestination;
    }
}
```

ステップ 2: ROUTEBOX コンシューマーの起動

ルートコンシューマーの作成時に、routeboxUri の # エントリーが CamelContext レジストリーの作成された内部レジストリー、routebuilder リスト、および dispatchStrategy/dispatchMap に一致することに注意してください。すべての routebuilders および関連するルートが内部コンテキストで作成された routebox で起動することに注意してください。

```
private String routeboxUri = "routebox:multipleRoutes?
innerRegistry=#registry&routeBuilders=#routes&dispatchMap=#map";

public void testRouteboxRequests() throws Exception {
    CamelContext context = createCamelContext();
    template = new DefaultProducerTemplate(context);
    template.start();

    context.addRoutes(new RouteBuilder() {
        public void configure() {
            from(routeboxUri)
                .to("log:Routes operation performed?showAll=true");
        }
    });
    context.start();

    // Now use the ProducerTemplate to send the request to the routebox
    template.requestBodyAndHeader(routeboxUri, book, "ROUTE_DISPATCH_KEY",
"addToCatalog");
}
```

ステップ 3: ROUTEBOX プロデューサーの使用

routebox にリクエストを送信する場合、プロデューサーは内部ルートエンドポイント URI を認識する必要はありません。以下のように、ディスパッチストラテジーまたは dispatchMap で Routebox URI エンドポイントを呼び出すだけです。

要求を正しい内部ルートに送信できるように、ディスパッチマップのキーに一致するキー ROUTE_DISPATCH_KEY (Dispatch Strategy の場合オプション)と呼ばれる特別なエクスチェンジヘッダーを設定する必要があります。

```
from("direct:sendToStrategyBasedRoutebox")
    .to("routebox:multipleRoutes?
innerRegistry=#registry&routeBuilders=#routes&dispatchStrategy=#strategy")
    .to("log:Routes operation performed?showAll=true");

from ("direct:sendToMapBasedRoutebox")
    .setHeader("ROUTE_DISPATCH_KEY", constant("addToCatalog"))
    .to("routebox:multipleRoutes?
```

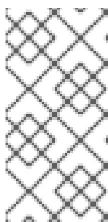
```
innerRegistry=#registry&routeBuilders=#routes&dispatchMap=#map")
    .to("log:Routes operation performed?showAll=true");
```

第138章 RSS

RSS コンポーネント

rss: コンポーネントは RSS フィードのポーリングに使用されます。Apache Camel はデフォルトでフィードを 60 秒ごとにポーリングします。

注記： コンポーネントは現在、ポーリング（かかる）フィードのみをサポートします。



注記

camel-rss は内部的に、ServiceMix でホストされるパッチが 適用されたバージョン の ROME を使用して、OSGi クラ出力ディングの問題 を解決します。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
rss:rssUri
```

rssUri は、ポーリングする RSS フィードへの URI です。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

プロパティー	デフォルト	説明

splitEntries	**true**	**true** の場合、Apache Camel はフィードを個別のエントリーに分割し、ポーリングによって各エントリーを返します。たとえば、フィードに 7 つのエントリーが含まれる場合、Apache Camel は最初のポーリングの最初のエントリー、2 番目のポーリングの 2 番目のエントリーなどを返します。フィードにエントリーが残っていない場合、Apache Camel はリモート RSS URI に接続して新しいフィードを取得します。**false** の場合、Apache Camel はポーリングごとに新しいフィードを取得し、フィードのエントリーをすべて返します。
filter	**true**	返されたエントリーをフィルターするために **splitEntries** オプションと組み合わせて使用します。デフォルトでは、Apache Camel は、フィードから新しいエントリーのみを返す **UpdateDateFilter** フィルターを適用し、コンシューマーエンドポイントがエントリーを複数回受信しないようにします。フィルターはエントリーを時系列に並べ、最後に返した順に並べ替えます。
throttleEntries	**true**	camel **2.5:** 単一のフィードポーリングで特定されたすべてのエントリーを即座に配信するかどうかを設定します。true の場合、consumer.delay ごとに 1 つのエントリーのみが処理されます。splitEntries が true に設定されている場合にのみ適用されます。
lastUpdate	**null**	**フィルター** オプションと組み合わせて を使用して、特定の日付/時間より前のエントリーをブロックします(**entry.updated** タイムスタンプを使用します)。形式は **yyyy-MM-ddTHH:MM:ss** です。例： **2007-12-24T17:45:59**
feedHeader	**true**	ROME **SyndFeed** オブジェクトをヘッダーとして追加するかどうかを指定します。

sortEntries	**false**	**splitEntries** が **true** の場合、エントリーを更新された日付でソートするかどうかを指定します。
consumer.delay	**60000**	各ポーリングの遅延（ミリ秒単位）。
consumer.initialDelay	**1000**	ポーリングが開始するまでの時間（ミリ秒単位）。
consumer.userFixedDelay	**false**	プール間の固定遅延を使用するには、**true** に設定します。それ以外の場合、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。
username		**Camel 2.16:** HTTP フィードからポーリングする場合の基本認証用。
password		**Camel 2.16:** HTTP フィードからポーリングする場合の基本認証用。

データ型の交換

Apache Camel は、ROME SyndFeed を使用してエクスチェンジの In ボディーを初期化します。splitEntries フラグの値に応じて、Apache Camel は SyndEntry が 1 つある SyndFeed または SyndEntry オブジェクトの java.util.List のいずれかを返します。

オプション	値	動作
splitEntries	**true**	現在のフィードからの 1 つのエントリーがエクスチェンジに設定されます。
splitEntries	**false**	現在のフィードのエントリー一覧全体がエクスチェンジに設定されます。

メッセージヘッダー

ヘッダー	説明
CamelRssFeed	Apache Camel 2.0: **SyndFeed** オブジェクト全体。

RSS DATAFORMAT

RSS コンポーネントには、String (XML として)と ROME RSS モデルオブジェクト間の変換に使用できる RSS データフォーマットが同梱されています。

- *marshal = ROME SyndFeed から XML Stringへ*
- *unmarshal = XML String から ROME SyndFeed*

これを使用するルートは以下のようになります。

```
from("rss:file:src/test/data/rss20.xml?
splitEntries=false&consumer.delay=1000").marshal().rss().to("mock:marshal");
```

この機能の目的は、RSS メッセージを操作するために Apache Camel の非常に組み込み式を使用できるようにすることです。以下に示すように、XPath 式を使用して RSS メッセージを絞り込むことができます。

```
// only entries with Camel in the title will get through the filter
from("rss:file:src/test/data/rss20.xml?splitEntries=true&consumer.delay=100")
    .marshal().rss().filter().xpath("//item/title[contains(.,'Camel')]").to("mock:result");
```

クエリーパラメーター

RSS フィードの URL がクエリーパラメーターを使用する場合、このコンポーネントはそれも理解します。たとえば、フィードが alt=rss を使用する場合など、do ("rss:http://someserver.com/feeds/posts/default?alt=rss&splitEntries=false&consumer.delay=1000").to ("bean:rss"); などに do ("rss:http://someserver.com/feeds/posts/default?alt=rss&splitEntries=false&consumer.delay=1000").to ("bean:rss");を実行できます。

エントリーのフィルターリング

上記のデータ形式セクションに示されるように、XPath を使用してエントリーを簡単にフィルターできます。Apache Camel の Bean インテグレーションを利用して、独自の条件を実装することもできます。たとえば、上記の XPath の例と同等のフィルターは以下のようになります。

```
// only entries with Camel in the title will get through the filter
from("rss:file:src/test/data/rss20.xml?splitEntries=true&consumer.delay=100").
    filter().method("myFilterBean", "titleContainsCamel").to("mock:result");
```

■

これに対するカスタム Bean は以下のようになります。

```
public static class FilterBean {
    public boolean titleContainsCamel(@Body SyndFeed feed) {
        SyndEntry firstEntry = (SyndEntry) feed.getEntries().get(0);
        return firstEntry.getTitle().contains("Camel");
    }
}
```

その他の参考資料

- *Atom*

第139章 SALESFORCE

SALESFORCE コンポーネント

Camel 2.12 以降で利用可能

このコンポーネントにより、プロデューサーおよびコンシューマーエンドポイントは Java DTO を使用して Salesforce と通信できます。これらの DTO を生成するコンパニオン Maven プラグインである Camel Salesforce プラグインがあります（以下を参照）。

TLS (Transport Layer Security)を使用するように camel-salesforce コンポーネントを設定するには、Security Guide の Configuring Transport Security for Camel Components の章を参照してください。

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-salesforce</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

salesforce コンポーネントの URI スキームは以下のとおりです。

```
salesforce:topic?options
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

サポートされる SALESFORCE API

コンポーネントは以下の Salesforce API をサポートします。

プロデューサーエンドポイントは、次の API を使用できます。ほとんどの API プロセスは一度に 1 レコードで、Query API は複数のレコードを取得できます。

REST API

- *getVersions - サポートされる Salesforce REST API バージョンの取得*
- *getResources - 利用可能な Salesforce REST リソースエンドポイントを取得します。*
- *getGlobalObjects : 利用可能なすべての SObject タイプのメタデータを取得します。*
- *getBasicInfo : 特定の SObject タイプの基本メタデータを取得します。*
- *getDescription : 特定の SObject タイプの包括的なメタデータを取得します。*
- *getSObject - Salesforce ID を使用して SObject を取得します。*
- *createSObject : SObject を作成します*
- *updateSObject : Id を使用して SObject を更新します。*
- *deleteSObject : Id を使用して SObject を削除します。*
- *getSObjectWithId : 外部（ユーザー定義）id フィールドを使用して SObject を取得します。*

- upsertSObject : 外部 ID を使用して SObject を更新または挿入します。

- deleteSObjectWithId : 外部 ID を使用して SObject を削除します。

- query - Salesforce SOQL クエリーを実行します。

- queryMore : クエリー API から返された結果リンクを使用して、より多くの結果（大量の結果の場合）を取得します。

- search - Salesforce SOSL クエリーを実行します。

たとえば、以下のプロデューサーエンドポイントは upsertSObject API を使用し、sObjectIdName パラメーターが外部 id フィールドとして指定されています。リクエストメッセージのボディーは、maven プラグインを使用して生成された SObject DTO である必要があります。応答メッセージは、既存のレコードが更新された場合、または新規レコードの ID を持つ CreateSObjectResult、または新規オブジェクトの作成時のエラーの一覧のいずれかです。

```
...to("salesforce:upsertSObject?sObjectIdName=Name")...
```

REST BULK API

プロデューサーエンドポイントは、次の API を使用できます。すべてのジョブデータ形式(xml、csv、zip/xml、および zip/csv)がサポートされます。要求と応答は、ルートによってマーシャリング/アンマーシャリングされる必要があります。通常、要求は CSV ファイルのようなストリームソースであり、応答もファイルに保存され、要求と関連付けられます。

- createJob - Salesforce 一括ジョブを作成します。

- getJob - Salesforce ID を使用してジョブを取得します。

- closeJob : ジョブを閉じる

- *abortJob : ジョブを中止します。*
- *createBatch - 一括ジョブ内でバッチを送信します。*
- *getBatch - Id を使用してバッチを取得します。*
- *getAllBatches - 一括ジョブ ID のすべてのバッチを取得します。*
- *getRequest - バッチのリクエストデータ(XML/CSV)を取得します。*
- *getResults : バッチの完了時に結果を取得します。*
- *createBatchQuery : SOQL クエリーから Batch を作成します。*
- *getQueryResultIds - バッチクエリーの結果 ID の一覧を取得します。*
- *getQueryResult : 結果 ID の結果を取得します。*

たとえば、以下のプロデューサーエンドポイントは createBatch API を使用して Job Batch を作成します。in メッセージには、XML、CSV、ZIP_XML または ZIP_CSV の InputStream （通常は UTF-8 CSV または XML コンテンツ）および Job および contentType のヘッダーフィールド jobId に変換できるボディーが含まれている必要があります。put メッセージのボディーには、成功時に BatchInfo が含まれたり、SalesforceException をエラーに出力したりします。

```
...to("salesforce:createBatchJob")..
```

REST STREAMING API

コンシューマーエンドポイントは、以下の sytax を使用して、エンドポイントをストリーミングし、作成/更新時に Salesforce の通知を受信することができます。

トピックを作成してサブスクライブするには、以下を行います。

```
from("salesforce:CamelTestTopic?
notifyForFields=ALL&notifyForOperations=ALL&sObjectName=Merchandise__c&updateTopic
=true&sObjectQuery=SELECT Id, Name FROM Merchandise__c")...
```

既存のトピックにサブスクライブするには、以下を実行します。

```
from("salesforce:CamelTestTopic&sObjectName=Merchandise__c")...
```

CONTENTWORKSPACE へのドキュメントのアップロード

Processor インスタンスを使用して、Java で ContentVersion を作成します。

```
public class ContentProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        Message message = exchange.getIn();

        ContentVersion cv = new ContentVersion();
        ContentWorkspace cw = getWorkspace(exchange);
        cv.setFirstPublishLocationId(cw.getId());
        cv.setTitle("test document");
        cv.setPathOnClient("test_doc.html");
        byte[] document = message.getBody(byte[].class);
        ObjectMapper mapper = new ObjectMapper();
        String enc = mapper.convertValue(document, String.class);
        cv.setVersionDataUrl(enc);
        message.setBody(cv);
    }

    protected ContentWorkspace getWorkSpace(Exchange exchange) {
        // Look up the content workspace somehow, maybe use enrich() to add it to a
        // header that can be extracted here
        ....
    }
}
```

プロセッサーから Salesforce コンポーネントに出力を付与します。

```
from("file:///home/camel/library")
    .to(new ContentProcessor())     // convert bytes from the file into a ContentVersion SObject
                                    // for the salesforce component
    .to("salesforce:createSObject");
```

CAMEL SALESFORCE MAVEN プラグイン

この Maven プラグインは、Camel Salesforce の DTO を生成します。

使用方法

プラグイン設定には、以下のプロパティーがあります。

オプション	説明
clientId	Salesforce client Id for Remote API access.
clientSecret	リモート API アクセス用の Salesforce クライアントシークレット。
userName	Salesforce アカウントのユーザー名。
password	Salesforce アカウントパスワード（シークレットトークンを含む）
version	Salesforce Rest API バージョン。デフォルトは 34.0 です。
outputDirectory	生成された DTO を配置するディレクトリー。デフォルトは **${project.build.directory}/generated-sources/camel-salesforce** です。
includes	追加する SObject タイプのリスト。
excludes	除外する SObject タイプのリスト。
includePattern	含める SObject タイプの Java RegEx。
excludePattern	除外する SObject タイプの Java RegEx。
packageName	生成された DTO の Java パッケージ名。デフォルトは **org.apache.camel.salesforce.dto** です。

> ***mvn camel-salesforce:generate -DclientId=<clientid> -DclientSecret=<clientsecret> -DuserName=<username> -Dpassword=<password>***

生成された DTO は Jackson および XStream アノテーションを使用します。すべての Salesforce フィールドタイプがサポートされます。日時フィールドは Joda DateTime にマッピングされ、リストフィールドは生成された Java Enumerations にマッピングされます。

第140章 SAP コンポーネント

概要

SAP コンポーネントは、10 つの異なる SAP コンポーネントのスイートで設定されるパッケージです。sRFC、tRFC、qRFC プロトコルをサポートするリモート関数呼び出し(RFC)コンポーネントがあり、IDoc 形式のメッセージを使用した通信を容易にする IDoc コンポーネントがあります。コンポーネントは SAP Java Connector (SAP JCo)ライブラリーを使用して SAP および SAP IDoc ライブラリーとの双方向通信を容易にし、Intermediate Document (IDoc)形式のドキュメントの送信を容易にします。

140.1. 概要

Dependencies

このコンポーネントを使用するには、Maven ユーザーは pom.xml ファイルに以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.fusesource</groupId>
  <artifactId>camel-sap</artifactId>
  <version>x.x.x</version>
<dependency>
```

SAP コンポーネントの追加プラットフォームの制限

SAP コンポーネントはサードパーティーの JCo 3 および IDoc 3 ライブラリーに依存するため、これらのライブラリーがサポートするプラットフォームにのみインストールできます。プラットフォームの制限に関する詳細は、Red Hat Fuse でサポートされる設定 を参照してください。

SAP JCo および SAP IDoc ライブラリー

SAP コンポーネントを使用するための前提条件は、SAP Java Connector (SAP JCo)ライブラリーおよび SAP IDoc ライブラリーが Java ランタイムの lib/ ディレクトリーにインストールされていることです。ターゲットのオペレーティングシステムに適した SAP ライブラリーのセットを SAP Service Marketplace からダウンロードするようにしてください。

ライブラリーファイルの名前は、表140.1「必要な SAP ライブラリー」 に示すように、ターゲットのオペレーティングシステムによって異なります。

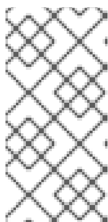
表140.1 必要な SAP ライブラリー

SAP コンポーネント	Linux および UNIX	Windows
SAP JCo 3	**sapjco3.jar** **libsapjco3.so**	**sapjco3.jar** **sapjco3.dll**
SAP IDoc	**sapidoc3.jar**	**sapidoc3.jar**

Fuse OSGi コンテナーへのデプロイ(Fabric 以外)

SAP JCo ライブラリーおよび SAP IDoc ライブラリーは、以下のように JBoss Fuse OSGi コンテナー(Fabric 以外)にインストールできます。

1. ***SAP JCo ライブラリーと SAP IDoc ライブラリーを SAP Service Marketplace からダウンロードし、ご使用のオペレーティングシステムのライブラリーの適切なバージョンを選択してください。http://service.sap.com/public/connectors***



注記

これらのライブラリーをダウンロードして使用するには、SAP Service Marketplace アカウント が必要です。

2. ***sapjco3.jar、libsapjco3.so (Windows では sapjco3.dll)、および sapidoc3.jar ライブラリーファイルを JBoss Fuse インストールのディレクトリーにコピーします。lib/***

3. ***テキストエディターで、設定プロパティーファイル etc/config.properties とカスタムプロパティーファイル etc/custom.properties の両方を開きます。etc/config.properties ファイルで、org.osgi.framework.system.packages.extra プロパティーを検索し、完全なプロパティー設定をコピーします（この設定は複数の行に拡張され、バックスラッシュ文字 \ を使用して行継続行を示します）。ここで、この設定を etc/custom.properties ファイルに貼り付けます。***

SAP ライブラリーをサポートするために必要なパッケージを追加できるようになりました。etc/custom.properties ファイルで、以下のように org.osgi.framework.system.packages.extra 設定に必要なパッケージを追加します。

```
org.osgi.framework.system.packages.extra = \
 ... , \
 com.sap.conn.idoc, \
 com.sap.conn.idoc.jco, \
 com.sap.conn.jco, \
```

```
com.sap.conn.jco.ext, \
com.sap.conn.jco.monitor, \
com.sap.conn.jco.rt, \
com.sap.conn.jco.server
```

ヒント

新しいエントリーの前の各行の最後にコンマとバックスラッシュ , \ を含めることを忘れないでください。これにより、リストが適切に継続されます。

4. これらの変更を有効にするには、コンテナーを再起動する必要があります。

5. camel-sap 機能をコンテナーにインストールする必要があります。Karaf コンソールで、以下のコマンドを入力します。

```
JBossFuse:karaf@root> features:install camel-sap
```

Fuse Fabric でのデプロイ

SAP コンポーネントを使用するための前提条件として、SAP Java Connector (SAP JCo)ライブラリーと SAP IDoc ライブラリーが Java ランタイムの lib/ ディレクトリーにインストールされている必要があります。つまり、sapjco3.jar、libsapjco3.so (Windows では sapjco3.dll)、および NORMAL です。sapidoc3.jar

Fuse Fabric デプロイメントの場合、これには特別な設定が必要です。Fabric コンテナーはネットワーク上のどこにでもデプロイ可能な必要があるため、単一のマシンの Java lib ディレクトリーに SAP ライブラリーをインストールすることはポイントはありません。正しいアプローチは、SAP JCo ライブラリーおよび SAP IDoc ライブラリーを実行する任意のホストにダウンロードおよびインストールできる特別なプロファイルを定義することです。

SAP JCo ライブラリーおよび SAP IDoc ライブラリーのプロファイルを以下のように定義できます。

1. JCo ライブラリーと IDoc ライブラリー-sapjco3.jar、libsapjco3.so (Windows では sapjco3.dll)、および sapidoc3.jar- をネットワークにアクセスできる場所にデプロイします。たとえば、Web サーバーにライブラリーをインストールして、JCo ライブラリーおよび IDoc ライブラリーを HTTP URL http://mywebserver/sapjco3.jar、http://mywebserver/libsapjco3.so、および http://mywebserver/sapidoc3.jar からダウンロードできます。

2. 以下のコンソールコマンドを入力して、新しいプロファイル camel-sap-profile を作成します。

```
JBossFuse:karaf@root> profile-create camel-sap-profile
```

3. 以下のコンソールコマンドを入力して、camel-sap-profile プロファイルのエージェントプロパティーを編集します。

```
JBossFuse:karaf@root> profile-edit camel-sap-profile
```

4. 組み込みプロファイルエディターが起動します。この組み込みテキストエディターを使用して、以下の内容をエージェントプロパティーに追加します。

```
# Profile:my-camel-sap-profile
attribute.parents = feature-camel

# Deploy JCo3 Libs to Container
lib.sapjco3.jar = http://mywebserver/sapjco3.jar
lib.sapjco3.so = http://mywebserver/libsapjco3.so
lib.sapidoc3.jar = http://mywebserver/sapidoc3.jar

# Append JCo3 Packages and IDoc packages to OSGi system property
# in order to expose JCo3 and IDoc classes to OSGi environment
config.org.osgi.framework.system.packages.extra= \
   ... Packages from etc/config.properties file ...\
   com.sap.conn.jco, \
   com.sap.conn.jco.ext, \
   com.sap.conn.jco.monitor, \
   com.sap.conn.jco.rt, \
   com.sap.conn.jco.server, \
   com.sap.conn.idoc, \
   com.sap.conn.idoc.jco
```

以下のようにプロパティー設定をカスタマイズします。

lib.sapjco3.jar

HTTP URL を Web サーバー上の sapjco3.jar ファイルの実際の場所にカスタマイズします。

lib.sapjco3.so

HTTP URL を Web サーバーの libsapjco3.so ファイル（または sapjco3.dll）の実際の場所にカスタマイズします。

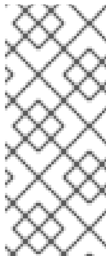
lib.sapidoc3.jar

HTTP URL を Web サーバー上の sapidoc3.jar ファイルの実際の場所にカスタマイズします。

config.org.osgi.framework.system.packages.extra

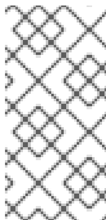
JBoss Fuse インストールのコンテナー設定プロパティーファイル etc/config.properties を開き、org.osgi.framework.system.packages.extra プロパティーの設定を見つけます。その設定のパッケージ一覧をコピーし、行を置き換えてプロファイルのエージェントプロパティーに貼り付けます。

```
... Packages from etc/config.properties file ...\
```



注記

config.org.osgi.framework.system.packages.extra の config. 接頭辞は、プロファイルにコンテナー設定プロパティーを設定する Fabric を示します。*



注記

バックスラッシュ \ は行継続文字(UNIX 規則)で、直後に改行文字を追加する必要があります。

完了したら、Ctrl-S と入力してプロパティーを保存します。

5. *camel-sap-profile プロファイルを、SAP コンポーネントを実行するすべての Fabric コンテナーにデプロイできます。たとえば、camel-sap-profileprofile を sap-instance コンテナーにデプロイするには、以下を実行します。*

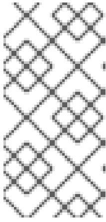
```
JBossFuse:karaf@root> container-add-profile sap-instance came-sap-profile
```

JBoss EAP コンテナーへのデプロイ

JBoss EAP コンテナーに SAP コンポーネントをデプロイするには、以下の手順を実行します。

1. *SAP JCo ライブラリーと SAP IDoc ライブラリーを SAP Service Marketplace からダウ*

ンロードし、ご使用のオペレーティングシステムのライブラリーの適切なバージョンを選択してください。http://service.sap.com/public/connectors

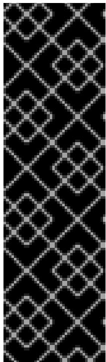


注記

これらのライブラリーをダウンロードして使用するには、SAP Service Marketplace アカウント が必要です。

2. JCo ライブラリーファイルと IDoc ライブラリーファイルを JBoss EAP インストールの適切なサブディレクトリーにコピーします。たとえば、ホストプラットフォームが 64 ビットの Linux (linux-x86_64)の場合は、以下のようにライブラリーファイルをインストールします。

```
cp sapjco3.jar sapidoc3.jar
$JBOSS_HOME/modules/system/layers/fuse/com/sap/conn/jco/main/
mkdir -p $JBOSS_HOME/modules/system/layers/fuse/com/sap/conn/jco/main/lib/linux-
x86_64
cp libsapjco3.so
$JBOSS_HOME/modules/system/layers/fuse/com/sap/conn/jco/main/lib/linux-x86_64/
```



重要

ネイティブライブラリー（例： libsapjco3.so）を JBoss EAP インストールにインストールする場合は、library サブディレクトリーに名前を付けるための標準化された規則があります。これに従う必要があります。64 ビットの Linux の場合、サブディレクトリーは linux-x86_64 になります。他のプラットフォームについては、を参照してください https://docs.jboss.org/author/display/MODULES/Native+Libraries。

3. SwitchYard サブシステム設定の org.switchyard.component.camel.sap モジュールのコメントを解除します。たとえば、JBoss EAP スタンドアロンモードで SAP コンポーネントを有効にするには、$JBOSS_HOME/standalone/configuration/standalone.xml ファイルを編集し、以下の行を検索してコメント解除します。

```
<!-- Uncomment this module to enable camel-sap binding
<module identifier="org.switchyard.component.camel.sap"
implClass="org.switchyard.component.camel.sap.deploy.CamelSapComponent"/>
-->
```

URI 形式

SAP コンポーネントが提供するエンドポイントには、Remote Function Call (RFC)エンドポイントと Intermediate Document (IDoc)エンドポイントの 2 種類があります。

RFC エンドポイントの URI 形式は以下のとおりです。

```
sap-srfc-destination:destinationName:rfcName
sap-trfc-destination:destinationName:rfcName
sap-qrfc-destination:destinationName:queueName:rfcName
sap-srfc-server:serverName:rfcName[?options]
sap-trfc-server:serverName:rfcName[?options]
```

IDoc エンドポイントの URI 形式は以下のとおりです。

```
sap-idoc-
destination:destinationName:idocType[:idocTypeExtension[:systemRelease[:applicationRelea
se]]]
sap-idoclist-
destination:destinationName:idocType[:idocTypeExtension[:systemRelease[:applicationRelea
se]]]
sap-qidoc-
destination:destinationName:queueName:idocType[:idocTypeExtension[:systemRelease[:appl
icationRelease]]]
sap-qidoclist-
destination:destinationName:queueName:idocType[:idocTypeExtension[:systemRelease[:appl
icationRelease]]]
sap-idoclist-
server:serverName:idocType[:idocTypeExtension[:systemRelease[:applicationRelease]]]
[?options]
```

sap-endpointKind-destination で始まる URI 形式は、宛先エンドポイント（つまり Camel プロデューサーエンドポイント）を定義するために使用され、destinationName は SAP インスタンスへの特定のアウトバウンド接続の名前です。アウトバウンド接続には、「宛先設定」 で説明されているように、コンポーネントレベルで名前が付けられ、設定されます。

sap-endpointKind-server で始まる URI 形式は、サーバーエンドポイント（つまり Camel コンシューマーエンドポイント）を定義するために使用され、serverName は SAP インスタンスからの特定の受信接続の名前です。「サーバー設定」 で説明されているように、受信接続にはコンポーネントレベルで名前が付けられ、設定されます。

RFC エンドポイント URI の他のコンポーネントは以下のとおりです。

rfcName

（必須） 宛先エンドポイント URI では、は接続された SAP インスタンスのエンドポイントによって呼び出される RFC の名前です。サーバーエンドポイント URI では、は接続された SAP インスタンスから呼び出されるときにエンドポイントによって処理される RFC の名前です。

queueName

このエンドポイントが SAP 要求を送信するキューを指定します。

IDoc エンドポイント URI の他のコンポーネントは以下のとおりです。

idocType

（必須） このエンドポイントによって生成された IDoc の Basic IDoc タイプを指定します。

idocTypeExtension

このエンドポイントによって生成された IDoc の IDoc の IDoc がある場合は、それを指定します。

systemRelease

このエンドポイントによって生成された IDoc の関連する SAP Basis Release を指定します（ある場合）。

applicationRelease

関連するアプリケーションリリースがある場合は、このエンドポイントによって生成された IDoc のように指定します。

queueName

このエンドポイントが SAP 要求を送信するキューを指定します。

RFC 宛先エンドポイントのオプション

RFC 宛先エンドポイント(sap-srfc-destination、sap-trfc-destination、および sap-qrfc-destination)は、以下の URI オプションをサポートします。

名前	デフォルト	説明
stateful	**false**	**true** の場合、このエンドポイントが SAP ステートフルセッションを開始することを指定します。

名前	デフォルト	説明
transacted	**false**	**true** の場合、このエンドポイントが SAP トランザクションを開始することを指定します。

RFC サーバーエンドポイントのオプション

SAP RFC サーバーエンドポイント(sap-srfc-server および sap-trfc-server)は以下の URI オプションをサポートします。

名前	デフォルト	説明
stateful	**false**	**true** の場合、このエンドポイントが SAP ステートフルセッションを開始することを指定します。
propagateExceptions	**false**	(SAP-trfc-server **エンドポイントのみ**) **true** の場合、このエンドポイントがエクスチェンジの例外ハンドラーではなく、SAP の呼び出し元に再度伝播することを指定します。

IDoc List Server エンドポイントのオプション

SAP IDoc List Server エンドポイント(sap-idoclist-server)は、以下の URI オプションをサポートします。

名前	デフォルト	説明
stateful	**false**	**true** の場合、このエンドポイントが SAP ステートフルセッションを開始することを指定します。
propagateExceptions	**false**	**true** の場合、このエンドポイントがエクスチェンジの例外ハンドラーではなく、SAP の呼び出し元に例外を伝播することを指定します。

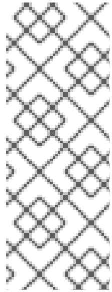
RFC および IDoc エンドポイントの概要

SAP コンポーネントパッケージは、以下の RFC および IDoc エンドポイントを提供します。

sap-srfc-destination

JBoss Fuse SAP Synchronous Remote Function Call Destination Camel コンポーネント。このエンドポイントは、Camel ルートが SAP システムに対するリクエストと応答の同期配信を必

要とする場合に使用する必要があります。

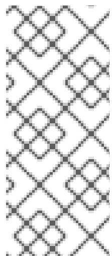


注記

このコンポーネントによって使用される sRFC プロトコルは、ベストエフォート で SAP システムとの間で要求および制限を提供します。要求の送信中に通信エラーが発生した場合、受信側の SAP システムのリモート関数呼び出しの完了ステータスは未確定のまま になります。

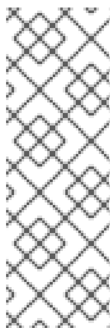
sap-trfc-destination

JBoss Fuse SAP Transactional Remote Function Call Destination Camel コンポーネント。このエンドポイントは、リクエストを受信 SAP システムに 最大 1 回配信する必要がある 場合に使用する必要があります。そのために、コンポーネントはトランザクション ID tid を生成します。このトランザクション ID は、ルートの交換でコンポーネントを介して送信されるすべての要求を処理します。受信側の SAP システムは、tid に要求を配信する前に記録します。SAP システムが同じ tid で要求を再度受け取った場合は、リクエストを配信しません。そのため、このコンポーネントのエンドポイント経由で要求を送信する際にルートが通信エラーに遭遇すると、リクエストが配信されて 1 回のみ実行されるのと同じエクスチェンジ内で要求の送信を再試行できます。



注記

このコンポーネントによって使用される tRFC プロトコルは非同期であり、応答を返しません。そのため、このコンポーネントのエンドポイントは応答メッセージを返しません。



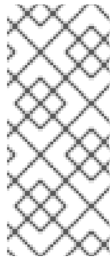
注記

このコンポーネントは、エンドポイント経由の一連の要求の順序を保証しません。これらの要求の配信および実行順序は、通信エラーおよび要求の再送信により受信側の SAP システムで異なる場合があります。Guaranteed Delivery order については、JBoss Fuse SAP Queued Remote Function Call Destination Camel component を参照してください。

sap-qrfc-destination

JBoss Fuse SAP Queued Remote Function Call Destination Camel コンポーネント。このコンポーネントは、エンドポイント経由でリクエストの配信を 順番に保証することで、JBoss Fuse Transactional Remote Function Call Destination camel コンポーネントの機能を拡張します。このエンドポイントは、一連の要求が相互に依存し、受信 SAP システムに 最大 1 回、かつ 順番に配信される必要がある 場合に使用する必要があり ます。コンポーネントは、JBoss Fuse SAP Transactional Remote Function Call Destination Camel コンポーネントと同じメカニズムを使用して配信の保証を 最大 1 度 実行します。順序の保証は、SAP システムによって受信された順序で受信された キュー に 要求をシリアライズすることで実現されます。インバウンドキューは、SAP

内の QIN スケジューラー によって処理されます。インバウンドキューがアクティブになる と、QIN スケジューラーはキュー要求を順番に実行します。



注記

このコンポーネントによって使用される qRFC プロトコルは非同期であり、応答を返しません。そのため、このコンポーネントのエンドポイントは応答メッセージを返しません。

sap-srfc-server

JBoss Fuse SAP Synchronous Remote Function Call Server Camel コンポーネント。このコンポーネントとそのエンドポイントは、Camel ルートが SAP システムとの間でリクエストと応答を同期的に処理する必要がある場合は使用する必要があります。

sap-trfc-server

JBoss Fuse SAP Transactional Remote Function Call Server Camel コンポーネント。このエンドポイントは、送信している SAP システムが要求を Camel ルートに 最大 1 度 配信する必要がある場合に使用する必要があります。これを実現するため、送信した SAP システムはトランザクション ID tid を生成します。これは、コンポーネントのエンドポイントに送信するすべての要求に対応します。送信元の SAP システムは、tid に関連する一連のリクエストを送信する前に、tid が指定の を受信しているかどうかを最初にチェックします。コンポーネントは、受信した tidの一覧を確認し、そのリストにない場合は送信済み tid を記録し、tid がすでに記録されたかどうかを示します。その後、tid が記録されていない場合、送信している SAP システムは一連のリクエストのみを送信します。これにより、送信した SAP システムは一連のリクエストを Camel ルートに確実に送信できます。

sap-idoc-destination

JBoss Fuse SAP IDoc Destination Camel コンポーネント。このエンドポイントは、Camel ルートが Intermediate Documents (IDoc)のリストを SAP システムに送信するために必要な場合に使用する必要があります。

sap-idoclist-destination

JBoss Fuse SAP IDoc List Destination Camel コンポーネント。このエンドポイントは、Camel ルートが Intermediate ドキュメント(IDoc)リストのリストを SAP システムに送信するために必要な場合に使用する必要があります。

sap-qidoc-destination

JBoss Fuse SAP Queued IDoc Destination Camel コンポーネント。このコンポーネントとそのエンドポイントは、Camel ルートが Intermediate ドキュメント(IDoc)のリストを SAP システム

に順番に送信する必要がある場合に使用する必要があります。

sap-qidoclist-destination

JBoss Fuse SAP Queued IDoc List Destination Camel component.このコンポーネントとそのエンドポイントは、Camel ルートが Intermediate ドキュメント(IDoc)リストのリストを SAP システムに順番に送信するために必要です。

sap-idoclist-server

JBoss Fuse SAP IDoc List Server Camel コンポーネント。このエンドポイントは、送信している SAP システムで中間ドキュメントリストの Camel ルートへの配信が必要な場合に使用してください。このコンポーネントは、'sap-trfc-server-standalone' クイックスタートで説明されているように、tRFC プロトコルを使用して SAP と通信します。

SAP RFC 宛先エンドポイント

RFC 宛先エンドポイントは SAP へのアウトバウンド通信をサポートします。これにより、これらのエンドポイントは SAP の ABAP 関数モジュールへの RFC 呼び出しを実行できます。RFC 宛先エンドポイントは、SAP インスタンスへの特定の接続を介して特定の ABAP 関数への RFC 呼び出しを行うように設定されます。RFC 宛先はアウトバウンド接続の論理指定であり、一意の名前を持ちます。RFC 宛先は、宛先データ と呼ばれる接続パラメーターのセットによって指定されます。

RFC 宛先エンドポイントは、受信する IN-OUT エクスチェンジの入力メッセージから RFC 要求を抽出し、その要求を SAP への関数呼び出しでディスパッチします。関数呼び出しからの応答は、エクスチェンジの出力メッセージで返されます。SAP RFC 宛先エンドポイントはアウトバウンド通信のみをサポートするため、RFC 宛先エンドポイントはプロデューサーの作成のみをサポートします。

SAP RFC サーバーエンドポイント

RFC サーバーエンドポイントは SAP からのインバウンド通信をサポートします。これにより、SAP の ABAP アプリケーションがサーバーエンドポイントへの RFC 呼び出しを実行できます。ABAP アプリケーションは、リモート関数モジュールであるかのように RFC サーバーエンドポイントと対話します。RFC サーバーエンドポイントは、SAP インスタンスから特定の接続を介して特定の RFC 関数への RFC 呼び出しを受信するように設定されます。RFC サーバーは受信接続の論理指定であり、一意の名前を持ちます。RFC サーバーは、サーバーデータ と呼ばれる接続パラメーターのセットによって指定されます。

RFC サーバーエンドポイントは受信 RFC 要求を処理し、それを IN-OUT エクスチェンジの入力メッセージとしてディスパッチします。エクスチェンジの出力メッセージは、RFC 呼び出しの応答として返されます。SAP RFC サーバーエンドポイントはインバウンド通信のみをサポートするため、RFC サーバーエンドポイントはコンシューマーの作成のみをサポートします。

SAP IDoc および IDoc リスト宛先エンドポイント

IDoc 宛先エンドポイントは、SAP へのアウトバウンド通信をサポートします。これは、IDoc メッセージでさらに処理できます。IDoc ドキュメントは、SAP 以外のシステムと簡単に交換できるビジネストランザクションを表します。IDoc 宛先は、宛先データ と呼ばれる接続パラメーターのセットによって指定されます。

IDoc list 宛先エンドポイントは IDoc 宛先エンドポイントと似ていますが、処理するメッセージが IDoc ドキュメントの リスト で設定される点が異なります。

SAP IDoc list server endpoint

IDoc リストサーバーのエンドポイントは SAP からのインバウンド通信をサポートし、Camel ルートが SAP システムから IDoc ドキュメントのリストを受信できるようにします。IDoc リストサーバーは、サーバーデータ と呼ばれる接続パラメーターのセットによって指定されます。

メタデータリポジトリー

メタデータリポジトリーは、以下のようなメタデータを保存するために使用されます。

関数モジュールに関するインターフェイスの説明

> *このメタデータは JCo および ABAP ランタイムによって使用され、RFC 呼び出しをチェックして、これらの呼び出しをディスパッチする前に通信パートナー間のデータタイプセーフ転送を確認します。リポジトリーにはリポジトリーデータが入力されます。リポジトリーデータは名前付き関数テンプレートのマップです。関数テンプレートには、すべてのパラメーターと関数モジュールとの間で渡される入力情報を記述するメタデータが含まれ、記述する関数モジュールの一意の名前があります。*

IDoc タイプの説明

> *このメタデータは IDoc ランタイムにより使用され、IDoc ドキュメントが通信パートナーに送信される前に正しくフォーマットされていることを確認します。基本的な IDoc タイプは、名前、許可されたセグメントの一覧、セグメント間の階層関係の説明で設定されます。セグメントに追加の制約を課すことができます。セグメントは必須または任意にすることができます。また、各セグメントの最小/最大範囲を指定できます（そのセグメントの許可される繰り返し回数を定義します）。*

そのため、SAP 宛先エンドポイントおよびサーバーエンドポイントでは、IDoc ドキュメントの送受信に RFC 呼び出しを送受信するためにリポジトリーへのアクセスが必要になります。RFC 呼び出しでは、エンドポイントによって呼び出しおよび処理されるすべての関数モジュールのメタデータはリポジトリー内に存在する必要があります。IDoc エンドポイントの場合は、すべての IDoc タイプおよび IDoc タイプの拡張機能のメタデータはリポジトリー内に存在する必要があります。宛先およびサー

バーエンドポイントによって使用されるリポジトリーの場所は、宛先データおよびそれぞれの接続のサーバーデータで指定されます。

SAP 宛先エンドポイントの場合、使用するリポジトリーは通常 SAP システムに存在し、デフォルトで接続されている SAP システムになります。このデフォルトでは、宛先データに明示的な設定は必要ありません。さらに、宛先エンドポイントが実行するリモート関数呼び出しのメタデータは、呼び出す既存の関数モジュールのリポジトリーにすでに存在します。そのため、宛先エンドポイントによる呼び出しのメタデータには SAP コンポーネントでの設定は必要ありません。

一方、サーバーエンドポイントによって処理される関数呼び出しのメタデータは通常 SAP システムのリポジトリーに存在しず、代わりに SAP コンポーネントに存在するリポジトリーが提供する必要があります。SAP コンポーネントは、名前付きメタデータリポジトリーのマップを維持します。リポジトリーの名前は、メタデータを提供するサーバー名に対応します。

140.2. 設定

概要

SAP コンポーネントは、宛先データ、サーバーデータ、リポジトリーデータを保存する 3 つのマップを維持します。宛先データストア と サーバーデータストア は、特別な設定オブジェクト SapConnectionConfiguration で設定され、自動的に SAP コンポーネントに注入されます(Blueprint XML 設定または Spring XML 設定ファイルのコンテキスト内)。リポジトリーデータストア は、関連する SAP コンポーネントに直接設定する必要があります。

140.2.1. 設定の概要

概要

SAP コンポーネントは、宛先データ、サーバーデータ、リポジトリーデータを保存する 3 つのマップを維持します。コンポーネントのプロパティー destinationDataStore は、宛先名によってキーされる宛先データである serverDataStore を、サーバー名が鍵するサーバーデータを保存し、プロパティー repositoryDataStore はリポジトリー名で鍵されたリポジトリーデータを保存します。これらの設定は初期化中にコンポーネントに渡す必要があります。

例

以下の例は、Blueprint XML ファイルにサンプル宛先データストアとサンプルサーバーデータストアを設定する方法を示しています。sap-configuration Bean（タイプ SapConnectionConfigurationの）は、この XML ファイルで使用されるすべての SAP コンポーネントに自動的に挿入されます。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ... >
    ...
    <!-- Configures the Inbound and Outbound SAP Connections -->
    <bean id="sap-configuration"
        class="org.fusesource.camel.component.sap.SapConnectionConfiguration">
        <property name="destinationDataStore">
            <map>
                <entry key="quickstartDest" value-ref="quickstartDestinationData" />
            </map>
        </property>
        <property name="serverDataStore">
            <map>
                <entry key="quickstartServer" value-ref="quickstartServerData" />
            </map>
        </property>
    </bean>

    <!-- Configures an Outbound SAP Connection -->
    <!-- *** Please enter the connection property values for your environment *** -->
    <bean id="quickstartDestinationData"
        class="org.fusesource.camel.component.sap.model.rfc.impl.DestinationDataImpl">
        <property name="ashost" value="example.com" />
        <property name="sysnr" value="00" />
        <property name="client" value="000" />
        <property name="user" value="username" />
        <property name="passwd" value="passowrd" />
        <property name="lang" value="en" />
    </bean>

    <!-- Configures an Inbound SAP Connection -->
    <!-- *** Please enter the connection property values for your environment ** -->
    <bean id="quickstartServerData"
        class="org.fusesource.camel.component.sap.model.rfc.impl.ServerDataImpl">
        <property name="gwhost" value="example.com" />
        <property name="gwserv" value="3300" />
        <!-- The following property values should not be changed -->
        <property name="progid" value="QUICKSTART" />
        <property name="repositoryDestination" value="quickstartDest" />
        <property name="connectionCount" value="2" />
    </bean>
</blueprint>
```

140.2.2. 宛先設定

概要

宛先の設定は、SAP コンポーネントの **destinationDataStore** プロパティーで維持されます。このマップの各エントリーは、SAP インスタンスへの個別のアウトバウンド接続を設定します。各エントリーのキーはアウトバウンド接続の名前であり、URI 形式セクション で説明されているように、宛先エンドポイント URI の **destinationName** コンポーネントで使用されます。

各エントリーの値は、アウトバウンド SAP 接続の設定を指定する宛先データ設定オブジェクト org.fusesource.camel.component.sap.model.rfc.impl.DestinationDataImpl です。

宛先設定のサンプル

以下の Blueprint XML コードは、quickstartDest の名前でサンプル宛先を設定する方法を示しています。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ... >
    ...
    <!-- Create interceptor to support tRFC processing -->
    <bean id="currentProcessorDefinitionInterceptor"
        class="org.fusesource.camel.component.sap.CurrentProcessorDefinitionInterceptStrategy" />

    <!-- Configures the Inbound and Outbound SAP Connections -->
    <bean id="sap-configuration"
       class="org.fusesource.camel.component.sap.SapConnectionConfiguration">
       <property name="destinationDataStore">
          <map>
             <entry key="quickstartDest" value-ref="quickstartDestinationData" />
          </map>
       </property>
    </bean>

    <!-- Configures an Outbound SAP Connection -->
    <!-- *** Please enter the connection property values for your environment *** -->
    <bean id="quickstartDestinationData"
       class="org.fusesource.camel.component.sap.model.rfc.impl.DestinationDataImpl">
       <property name="ashost" value="example.com" />
       <property name="sysnr" value="00" />
       <property name="client" value="000" />
       <property name="user" value="username" />
       <property name="passwd" value="password" />
       <property name="lang" value="en" />
    </bean>

</blueprint>
```

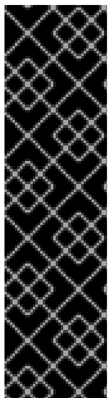
たとえば、前述の Blueprint XML ファイルのように宛先を設定した後、以下の URI を使用して quickstartDest 宛先で BAPI_FLCUST_GETLIST リモート関数呼び出しを呼び出すことができます。

```
sap-srfc-destination:quickstartDest:BAPI_FLCUST_GETLIST
```

tRFC および qRFC 宛先のインターセプター

前述の宛先設定のサンプルは、CurrentProcessorDefinitionInterceptStrategy オブジェクトのイン

スタンス化を示しています。このオブジェクトは、Camel ランタイムにインターセプターをインストールします。これにより、Camel SAP コンポーネントは RFC トランザクションの処理中に Camel ルート内の位置を追跡できます。詳細は、「トランザクション RFC 宛先エンドポイント」 を参照してください。



重要

このインターセプターは、トランザクション RFC 宛先エンドポイント(sap-trfc-destination や sap-qrfc-destinationなど)で重要であり、アウトバウンドトランザクション RFC 通信を適切に管理するために Camel ランタイムにインストールする必要があります。Destination RFC Transaction Handlers は、実行時にストラテジーが見つからない場合に Camel ログに警告を発行します。この場合、送信トランザクション RFC 通信を適切に管理するために Camel ランタイムを再プロビジョニングし、再起動する必要があります。

Logon および認証オプション

以下の表は、SAP 宛先データストアで宛先を設定するための ログオンおよび認証 *オプションを示しています。*

名前	デフォルト値	
client		SAP クライアント、必須の logon パラメ
user		Logon user、パスワードベースの認証用
aliasUser		ログオンユーザーエイリアス（ログオン
userId		ABAP AS へのログインに使用されるユーケット、証明書、現在のユーザー、またによって使用されます。ユーザー ID もコは、ユーザー ID が必須です。この ID はイムによってローカルで使用されます。
passwd		Logon password、パスワードベース認証
lang		Logon 言語（定義されていない場合は、
mysapsso2		指定された SAP Cookie Version 2 を SSC します。
x509cert		証明書ベースの認証には、指定された X5
lcheck		最初の呼び出し（有効化）まで認証を延す。

useSapGui		表示可能、非表示、または SAP GUI は使
codePage		logon パラメーターを変換するために使用 ラメーター。特別なケースでのみ使用
getsso2		ログオン後に SSO チケットの順序。取得 す。
denyInitialPassword		**1** に設定すると、初期パスワードを使用

接続オプション

以下の表は、SAP 宛先データストアで宛先を設定するための* 接続 *オプションを示しています。

名前	デフォルト値	
saprouter		SAP Router の背後にあるシステムに接続 文字列には、SAP ルーターのチェーンと ます。 **(/H/<host>[/S/<port>])+**
sysnr		SAP ABAP アプリケーションサーバーの
ashost		SAP ABAP アプリケーションサーバー（
mshost		SAP メッセージサーバー（ロードバラン
msserv		SAP メッセージサーバーポート、ロード サービス名 sapmsXXX を解決するには、 ングシステムのネットワーク層で実行し ポート番号を使用する場合は、ルックア ありません。
gwhost		アプリケーションサーバーへの接続を確 を指定できます。指定しない場合は、ア 用されます。
gwserv		gwhost を使用する場合は、設定する必要 ポートを指定できます。指定されていな トウェイのポートが使用されます。サー etc/services のルックアップをオペレー す。シンボリックサービス名の代わりに は実行されず、追加のエントリーは必要
r3name		SAP システムのシステム ID。負荷分散接
group		SAP アプリケーションサーバーのグルー ティー）

接続プールのオプション

以下の表は、SAP 宛先データストアで宛先を設定するための* 接続プール *オプションを示しています。

名前	デフォルト値	
peakLimit	**0**	宛先に同時に作成できるアクティブなア ティブな接続を無制限に許可します。そ 低い場合は、この値に自動的に増えます。 す。**poolCapacity** を指定しないと、デ
poolCapacity	**1**	宛先によって開いた状態で保持されるア は、接続プールがないという効果があり
expirationTime		宛先によって内部で保持される空き接続
expirationPeriod		宛先がリリースされた接続の有効期限を
maxGetTime		アプリケーションによって許可される接 合、接続を待つ最大時間（ミリ秒単位）

セキュアな接続オプション

以下の表は、SAP 宛先データストアで宛先を設定するための* セキュアなネットワーク *オプションを示しています。

名前	デフォルト値	
sncMode		セキュアな接続(SNC)モード、**0**（オフ）
sncPartnername		SNC パートナー（例：） **p:CN=R3, O=**
sncQop		SNC レベルのセキュリティー： **1 9**
sncMyname		独自の SNC 名。環境設定の上書き
sncLibrary		SNC サービスを提供するライブラリーへ

リポジトリーのオプション

以下の表は、SAP 宛先データストアで宛先を設定するための* リポジトリー *オプションを示しています。

名前	デフォルト値	
repositoryDest		リポジトリーとして使用する宛先を指定
repositoryUser		リポジトリーの宛先が設定されておらず これはリポジトリー呼び出しのユーザー リーの検索に別のユーザーを使用できま
repositoryPasswd		リポジトリーユーザーのパスワード。リ 合は必須です。
repositorySnc		**(オプション)** この宛先に SNC を使用 されている場合は、リポジトリー接続で **jco.client.snc**_mode の値です。特別な
repositoryRoundtripOptimization		1 つのラウンドトリップでリポジトリーデ API を有効にします。 **1** ABAP システム で **RFC_METADAT** **0** ABAP システム で **RFC_METADAT** プロパティーが設定されていない場合、 い、**RFC_METADATA_GET** が利用可 宛先はそれを使用します。 **注記：** リポジトリーがすでに初期化され るため）、このプロパティーは機能しま テムに関連し、同じ ABAP システムを参 ります。バックエンドの前提条件につい

トレース設定オプション

以下の表は、SAP 宛先データストアで宛先を 設定するためのトレース設定 *オプションを示しています。*

名前	デフォルト値	
trace		RFC トレースを有効/無効にします(**0** ま
cpicTrace		CPIC トレースの有効化/無効化 **[0..3]**

140.2.3. サーバー設定

概要

サーバーの設定は、SAP コンポーネントの serverDataStore プロパティーで維持されます。このマップの各エントリーは、SAP インスタンスからの個別の受信接続を設定します。各エントリーのキーはアウトバウンド接続の名前であり、URI 形式セクションで説明されているようにサーバーエンドポイント URI の serverName コンポーネントで使用されます。

各エントリーの値は、インバウンド SAP 接続の 設定を定義するサーバーデータ設定オブジェクトであるorg.fusesource.camel.component.sap.model.rfc.impl.ServerDataImpl です。

サーバーの設定例

以下の Blueprint XML コードは、quickstartServer という名前のサーバー設定例を作成する方法を示しています。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ... >
  ...
  <!-- Configures the Inbound and Outbound SAP Connections -->
  <bean id="sap-configuration"
    class="org.fusesource.camel.component.sap.SapConnectionConfiguration">
    <property name="destinationDataStore">
      <map>
        <entry key="quickstartDest" value-ref="quickstartDestinationData" />
      </map>
    </property>
    <property name="serverDataStore">
      <map>
        <entry key="quickstartServer" value-ref="quickstartServerData" />
      </map>
    </property>
  </bean>

  <!-- Configures an Outbound SAP Connection -->
  <!-- *** Please enter the connection property values for your environment *** -->
  <bean id="quickstartDestinationData"
    class="org.fusesource.camel.component.sap.model.rfc.impl.DestinationDataImpl">
    <property name="ashost" value="example.com" />
    <property name="sysnr" value="00" />
    <property name="client" value="000" />
    <property name="user" value="username" />
    <property name="passwd" value="passowrd" />
    <property name="lang" value="en" />
  </bean>

  <!-- Configures an Inbound SAP Connection -->
  <!-- *** Please enter the connection property values for your environment ** -->
  <bean id="quickstartServerData"
    class="org.fusesource.camel.component.sap.model.rfc.impl.ServerDataImpl">
    <property name="gwhost" value="example.com" />
    <property name="gwserv" value="3300" />
    <!-- The following property values should not be changed -->
```

```
        <property name="progid" value="QUICKSTART" />
        <property name="repositoryDestination" value="quickstartDest" />
        <property name="connectionCount" value="2" />
    </bean>
</blueprint>
```

この例では、サーバーがリモートの SAP インスタンスからメタデータを取得するのに使用する宛先接続 quickstartDest も設定します。この宛先は、repositoryDestination オプションを使用してサーバーデータで設定されます。このオプションを設定しない場合は、代わりにローカルのメタデータリポジトリーを作成する必要があります(「リポジトリーの設定」を参照してください)。

たとえば、前述の Blueprint XML ファイルのように宛先を設定した後、以下の URI を使用して、呼び出し元クライアントからの BAPI_FLCUST_GETLIST リモート関数呼び出しを処理できます。

```
sap-srfc-server:quickstartServer:BAPI_FLCUST_GETLIST
```

必須オプション

サーバーデータ設定オブジェクトに必要なオプションは以下のとおりです。

名前	デフォルト値	
gwhost		サーバー接続を登録する必要があるゲー
gwserv		ゲートウェイサービス：登録を実行でき 決するために、**etc/services** の検索はオ 実行されます。シンボリックサービス名 ルックアップは実行されず、追加のエン
progid		登録を行うプログラム ID。ゲートウェイ て機能します。
repositoryDestination		リモートの SAP サーバーでホストされる 得するためにサーバーが使用できる宛先
connectionCount		ゲートウェイで登録する必要のある接続

セキュアな接続オプション

サーバーデータ設定オブジェクトのセキュアな接続オプションは以下のとおりです。

名前	デフォルト値

sncMode		セキュアな接続(SNC)モード、**0**（オフ）
sncQop		SNC レベルのセキュリティー, **1 9**
sncMyname		サーバーの SNC 名。デフォルトの SNC **p:CN=JCoServer, O=ACompany, C**
sncLib		SNC サービスを提供するライブラリーへ い場合は、**jco.middleware.snc_lib** プ

その他のオプション

サーバーデータ設定オブジェクトの他のオプションは以下のとおりです。

名前	デフォルト値	
saprouter		ファイアウォールで保護されたシステム その ABAP システムのゲートウェイでサ みアクセスできます。通常のルーター文
maxStartupDelay		失敗時の 2 つの起動試行間の最大時間（ 後、最大値に達するか、サーバーを正常 す。
trace		RFC トレースを有効/無効にします(**0** ま
workerThreadCount		サーバー接続によって使用されるスレッ は、**connectionCount** の値が **worker** ドの最大数は 99 を超えることができま
workerThreadMinCount		サーバー接続によって使用されるスレッ は、**connectionCount** の値が **worker**

140.2.4. リポジトリーの設定

概要

リポジトリーの設定は、SAP コンポーネントの repositoryDataStore プロパティーで維持されます。このマップの各エントリーは、個別のリポジトリーを設定します。各エントリーのキーはリポジトリーの名前で、このキーはこのリポジトリーが割り当てられているサーバー名にも対応します。

各エントリーの値は、メタデータリポジトリーの内容を定義するリポジトリーデータ設定オブジェクト org.fusesource.camel.component.sap.model.rfc.impl.RepositoryDataImpl です。リポジトリーデータオブジェクトは、関数テンプレート設定オブジェクト

org.fuesource.camel.component.sap.model.rfc.impl.FunctionTemplateImpl のマップです。このマップの各エントリーは、関数モジュールのインターフェイスを指定し、各エントリーのキーは指定された関数モジュールの名前です。

リポジトリーデータの例

以下のコードは、メタデータリポジトリーを設定する簡単な例を示しています。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ... >
    ...
    <!-- Configures the sap-srfc-server component -->
    <bean id="sap-configuration"
        class="org.fusesource.camel.component.sap.SapConnectionConfiguration">
        <property name="repositoryDataStore">
            <map>
                <entry key="nplServer" value-ref="nplRepositoryData" />
            </map>
        </property>
    </bean>

    <!-- Configures a Meta-Data Repository -->
    <bean id="nplRepositoryData"
        class="org.fusesource.camel.component.sap.model.rfc.impl.RepositoryDataImpl">
        <property name="functionTemplates">
            <map>
                <entry key="BOOK_FLIGHT" value-ref="bookFlightFunctionTemplate" />
            </map>
        </property>
    </bean>
    ...
</blueprint>
```

関数テンプレートのプロパティー

関数モジュールのインターフェイスは、データが RFC 呼び出しの function モジュールに転送される 4 つのパラメーター一覧で設定されます。各パラメーター一覧は 1 つ以上のフィールドで設定され、各フィールドは RFC 呼び出しで転送される名前付きパラメーターです。以下のパラメーター一覧および例外リストがサポートされます。

- import パラメーター一覧 には、RFC 呼び出しの関数モジュールに送信されるパラメーター値が含まれます。

- export パラメーター一覧 には、RFC 呼び出しの function モジュールによって返されるパラメーター値が含まれます。

- *変更パラメーター一覧 には、RFC 呼び出しの関数モジュールとの間で送受信されるパラメーター値が含まれます。*

- *table パラメーター一覧 には、RFC 呼び出しの関数モジュールに送信され、返される内部テーブル値が含まれます。*

- *関数モジュールのインターフェイスは、RFC 呼び出しでモジュールが呼び出されたときに発生する可能性がある ABAP 例外の例外 リスト で設定されます。*

関数テンプレートは、関数インターフェイスの各パラメーター一覧のパラメーターの名前とタイプ、関数によって出力される ABAP 例外を記述します。関数テンプレートオブジェクトは、以下の表で説明されているように、メタデータオブジェクトの 5 つのプロパティーリストを維持します。

プロパティー	説明
importParameterList	list フィールドメタデータオブジェクト **org.fusesource.camel.component.sap.mod** 覧。関数モジュールへの RFC 呼び出しで送信され
changingParameterList	list フィールドメタデータオブジェクト **org.fusesource.camel.component.sap.mod** 覧。関数モジュールとの間で RFC 呼び出しで送受
exportParameterList	list フィールドメタデータオブジェクト **org.fusesource.camel.component.sap.mod** 覧。関数モジュールから RFC 呼び出しで返される
tableParameterList	list フィールドメタデータオブジェクト **org.fusesource.camel.component.sap.mod** 覧。関数モジュールとの間の RFC 呼び出しで送受
exceptionList	ABAP 例外メタデータオブジェクトの一覧 **org.fusesource.camel.component.sap.mod** ジュールの RFC 呼び出しで発生する可能性のある

関数テンプレートの例

以下の例は、関数テンプレートの設定方法の概要を示しています。

```
<bean id="bookFlightFunctionTemplate"
   class="org.fusesource.camel.component.sap.model.rfc.impl.FunctionTemplateImpl">
   <property name="importParameterList">
      <list>
```

```
            ...
          </list>
        </property>
        <property name="changingParameterList">
          <list>
            ...
          </list>
        </property>
        <property name="exportParameterList">
          <list>
            ...
          </list>
        </property>
        <property name="tableParameterList">
          <list>
            ...
          </list>
        </property>
        <property name="exceptionList">
          <list>
            ...
          </list>
        </property>
      </bean>
```

フィールドメタデータプロパティーを一覧表示します。

list フィールド meta-data object org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMeataDataImpl は、パラメーター一覧のフィールドの名前とタイプを指定します。要素的なパラメーターフィールド（CHAR, DATE, BCD, TIME, ----------|-----, splunk, splunk, splunk, splunk, splunk, splunk, ----------|-----, ----------|-----, ----------|-----, list field meta-data object: 通表は、リストフィールドの meta-data オブジェクトに設定できる設定プロパティーを示しています。BYTE NUM FLOAT INT INT1 INT2 DECF16 DECF34 STRING XSTRING

名前	デフォルト値	
name	-	パラメーターフィールドの名前。
type	-	フィールドのパラメータータイプ。
byteLength	-	Unicode 以外のレイアウトのフィールド(ターのタイプによって異なります。「RF い。
unicodeByteLength	-	Unicode レイアウトのフィールドの長さ タイプによって異なります。「RFC のメ
decimals	**0**	フィールド値の 10 進数数。パラメーター す。「RFC のメッセージボディー」を参

optional	false	true の場合、フィールドはオプションで せん。

すべての elementary パラメーターフィールドでは、name、type、byteLength、および unicodeByteLength プロパティーを meta-data オブジェクトで指定する必要があります。さらに、BCD、FLOAT、DECF16、および DECF34 フィールドには、meta-data オブジェクトで decimal プロパティーを指定する必要があります。

TABLE または STRUCTURE タイプの複雑なパラメーターフィールドの場合、以下の表には、list フィールド meta-data オブジェクトに設定できる設定プロパティーが記載されています。

名前	デフォルト値	
name	-	パラメーターフィールドの名前
type	-	フィールドのパラメータータイプ
recordMetaData	-	構造またはテーブルのメタデータ。レコ **org.fusesource.camel.component.s** は、構造行またはテーブル行にフィール
optional	**false**	**true** の場合、フィールドはオプションで せん。

すべての複雑なパラメーターフィールドでは、name、type および recordMetaData プロパティーをフィールドの meta-data オブジェクトに指定する必要があることに注意してください。recordMetaData プロパティーの値はレコードフィールドのメタデータオブジェクト org.fusesource.camel.component.sap.model.rfc.impl.RecordMetaDataImpl で、ネストされた構造またはテーブル行の構造を指定します。

elementary list field meta-data example

以下のメタデータ設定は、オプションの 24 桁のパックされた BCD 番号パラメーターと、TICKET_PRICE という名前の小数点を指定します。

```
<bean    class="org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMetaDataImpl">
   <property name="name" value="TICKET_PRICE" />
   <property name="type" value="BCD" />
   <property name="byteLength" value="12" />
   <property name="unicodeByteLength" value="24" />
   <property name="decimals" value="2" />
   <property name="optional" value="true" />
</bean>
```

複雑なリストフィールドのメタデータの例

以下のメタデータ設定は、TABLE レコードメタデータオブジェクトで指定された行構造を持つ CONNINFO という名前の必須パラメーターを指定します。connectionInfo

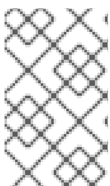
```
<bean   class="org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMetaDataImpl">
   <property name="name" value="CONNINFO" />
   <property name="type" value="TABLE" />
   <property name="recordMetaData" ref="connectionInfo" />
</bean>
```

レコードのメタデータ属性

レコードのメタデータオブジェクト org.fusesource.camel.component.sap.model.rfc.impl.RecordMetaDataImpl は、ネストされた STRUCTURE または TABLE パラメーターの名前および内容を指定します。レコードの meta-data オブジェクトは、レコードフィールドのメタデータオブジェクト org.fusesource.camel.component.sap.model.rfc.impl.FieldMetaDataImpl の一覧を維持します。これは、ネストされた構造またはテーブル行にあるパラメーターを指定します。

以下の表は、レコードメタデータオブジェクトに設定できる設定プロパティーの一覧です。

名前	デフォルト値	
name	-	レコードの名前。
recordFieldMetaData	-	レコードフィールドのメタデータオブジ **org.fusesource.camel.component.s** 造内に含まれるフィールドを指定します。



注記

レコードメタデータオブジェクトのすべてのプロパティーが必要です。

レコードのメタデータの例

以下の例は、レコード meta-data オブジェクトを設定する方法を示しています。

```
<bean   id="connectionInfo"
      class="org.fusesource.camel.component.sap.model.rfc.impl.RecordMetaDataImpl">
   <property name="name" value="CONNECTION_INFO" />
   <property name="recordFieldMetaData">
```

```
        <list>
          ...
        </list>
    </property>
</bean>
```

レコードフィールドのメタデータプロパティー

レコードフィールド meta-data オブジェクト org.fusesource.camel.component.sap.model.rfc.impl.FieldMetaDataImpl は、構造を持つパラメーターフィールドの名前およびタイプを指定します。

レコードフィールド meta-data オブジェクトはパラメーターフィールド meta-data オブジェクトと似ていますが、ネストされた構造またはテーブル行内の個別のフィールドのオフセットを追加で指定する必要があります。個々のフィールドの Unicode および Unicode オフセットは、構造または行に前述のフィールドの Unicode および Unicode バイトの長さ以外のものの合計から計算して指定する必要があります。ネストされた構造やテーブル行のフィールドのオフセットを適切に指定できないと、基礎となる JCo および ABAP ランタイムのパラメーターのフィールドストレージが重複し、RFC 呼び出しの値が適切に転送されないことに注意してください。

要素的なパラメーターフィールド（CHAR, DATE, BCD, TIME, ----------|-----, splunk, splunk, splunk, splunk, splunk, NORMAL, ----------|-----, ----------|-----, ----------|-----, ..." には、レコードフィールドの meta-data オブジェクトに設定できる設定プロパティーが記載されています。BYTE NUM FLOAT INT INT1 INT2 DECF16 DECF34 STRING XSTRING

名前	デフォルト値	
name	-	パラメーターフィールドの名前
type	-	フィールドのパラメータータイプ
byteLength	-	Unicode 以外のレイアウトのフィールド ターのタイプによって異なります。「RF い。
unicodeByteLength	-	Unicode レイアウトのフィールドの長さ タイプによって異なります。「RFC のメ
byteOffset	-	Unicode 以外のレイアウトのフィールド は、エンクロージング構造内のフィール
unicodeByteOffset	-	Unicode レイアウトのフィールドオフセ ンクロージング構造内のフィールドのバ
decimals	**0**	フィールド値の 10 進数数。パラメーター す。「RFC のメッセージボディー」を参

TABLE または STRUCTURE タイプの複雑なパラメーターフィールドの場合、以下の表には、レコードフィールドのメタデータオブジェクトに設定できる設定プロパティーが記載されています。

名前	デフォルト値	
name	-	パラメーターフィールドの名前
type	-	フィールドのパラメータータイプ
byteOffset	-	Unicode 以外のレイアウトのフィールド: は、エンクロージング構造内のフィール
unicodeByteOffset	-	Unicode レイアウトのフィールドオフセ ンクロージング構造内のフィールドのバ
recordMetaData	-	構造またはテーブルのメタデータ。レコ **org.fusesource.camel.component.s** は、構造行またはテーブル行にフィール

要素のレコードフィールドのメタデータの例

以下のメタデータ設定は、Unicode 以外のレイアウトの場合で、Unicode レイアウトの場合で 170 バイトをエンクロージング構造(Unicode レイアウトの場合)に 85 バイトに配置された DATE フィールドパラメーターを指定します。ARRDATE

```
<bean    class="org.fusesource.camel.component.sap.model.rfc.impl.FieldMetaDataImpl">
    <property name="name" value="ARRDATE" />
    <property name="type" value="DATE" />
    <property name="byteLength" value="8" />
    <property name="unicodeByteLength" value="16" />
    <property name="byteOffset" value="85" />
    <property name="unicodeByteOffset" value="170" />
</bean>
```

複雑なレコードフィールドのメタデータの例

以下のメタデータ設定は、flightInfo レコードメタデータオブジェクトで指定された構造を持つ FLTINFO という名前の STRUCTURE フィールドパラメーターを指定します。このパラメーターは、Unicode レイアウトと Unicode レイアウトの両方の場合、エンクロージング構造の先頭にあります。

```
<bean    class="org.fusesource.camel.component.sap.model.rfc.impl.FieldMetaDataImpl">
    <property name="name" value="FLTINFO" />
    <property name="type" value="STRUCTURE" />
    <property name="byteOffset" value="0" />
```

```
    <property name="unicodeByteOffset" value="0" />
    <property name="recordMetaData" ref="flightInfo" />
</bean>
```

140.3. RFC のメッセージボディー

要求および応答オブジェクト

SAP エンドポイントは、SAP リクエストオブジェクトが含まれるメッセージボディーを持つメッセージを受信することを想定し、SAP 応答オブジェクトが含まれるメッセージボディーを持つメッセージを返します。SAP 要求および応答は、事前定義データタイプを持つ名前付きフィールドが含まれる固定マップデータ構造です。

SAP 要求および応答の名前付きフィールドは SAP エンドポイントに固有のものであり、各エンドポイントが SAP 要求でパラメーターを定義し、応答することに注意してください。SAP エンドポイントは、要求オブジェクトおよびそれに固有の応答オブジェクトを作成するためのファクトリーメソッドを提供します。

```
public class SAPEndpoint ... {
    ...
    public Structure getRequest() throws Exception;

    public Structure getResponse() throws Exception;
    ...
}
```

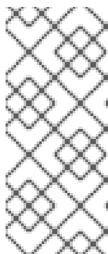
構造オブジェクト

SAP 要求および応答オブジェクトはいずれも、org.fusesource.camel.component.sap.model.rfc.Structure インターフェイスをサポートする構造オブジェクトとして Java で表されます。このインターフェイスは、java.util.Map インターフェイスと org.eclipse.emf.ecore.EObject インターフェイスの両方を拡張します。

```
public interface Structure extends org.eclipse.emf.ecore.EObject,
                                   java.util.Map<String, Object> {

    <T> T get(Object key, Class<T> type);

}
```

構造オブジェクトのフィールド値には、マップインターフェイスのフィールドの getter メソッドを介してアクセスします。さらに、構造インターフェイスは、フィールド値を取得するための type-restricted メソッドを提供します。

構造オブジェクトは、Eclipse Modeling Framework (EMF)を使用してコンポーネントランタイムに実装され、そのフレームワークの EObject インターフェイスをサポートします。構造オブジェクトのインスタンスには、提供するフィールドのマッピングの構造および内容を定義し、制限するメタデータが割り当てられます。このメタデータは、EMF が提供する標準の方法を使用してアクセスおよびイントロスペクションできます。詳細は、EMF のドキュメント を参照してください。



注記

構造オブジェクトに定義されていないパラメーターの取得を試みると、null を返します。構造に定義されていないパラメーターの設定を試みると、例外が発生し、誤ったタイプのパラメーターの値の設定を試みます。

以下のセクションで説明されているように、構造化オブジェクトには、複雑なフィールドタイプ(STRUCTURE および TABLE)の値が含まれるフィールドを含めることができます。これらのタイプのインスタンスを作成し、そのインスタンスを 構造に追加する必要はありません。これらのフィールド値のインスタンスは、エンクロージング構造でアクセスする際に必要な場合にオンデマンドで作成されます。

フィールドタイプ

SAP リクエストまたは応答の構造オブジェクト内にあるフィールドは、要素的 または 複雑な のいずれかになります。elementary フィールドには単一のスカラー値が含まれ、複雑なフィールドには要素型または複合型の 1 つ以上のフィールドが含まれます。

要素のフィールドタイプ

要素のフィールドは、文字、数値、16 進数、または文字列フィールドタイプのいずれかになります。以下の表では、構造オブジェクトに存在する可能性のある要素フィールドのタイプをまとめています。

フィールドタイプ	対応する Java タイプ	Byte Length	Unicode バイトの長さ	Number Decimals Digits	
CHAR	java.lang.String	1 から 65535	1 から 65535	-	ABAP Type
DATE	java.util.Date	8	16	-	ABAP Type
BCD	java.math.BigDecimal	1 から 16	1 から 16	0 から 14	ABAP Type バイトごと
TIME	java.util.Date	6	12	-	ABAP Type

BYTE	**byte[]**	1 から 65535	1 から 65535	-	ABAP Type
NUM	**java.lang.String**	1 から 65535	1 から 65535	-	ABAP Type
FLOAT	**java.lang.Double**	8	8	0 から 15	ABAP Type
INT	**java.lang.Integer**	4	4	-	ABAP Type
INT2	**java.lang.Integer**	2	2	-	ABAP Type
INT1	**java.lang.Integer**	1	1	-	ABAP Type
DECF16	**java.match.BigDecimal**	8	8	16	ABAP Type Point Numb
DECF34	**java.math.BigDecimal**	16	16	34	ABAP Type Point Numb
STRING	**java.lang.String**	8	8	-	ABAP Type
XSTRING	**byte[]**	8	8	-	ABAP Type

文字フィールドタイプ

文字フィールドには、基礎となる JCo ランタイムおよび ABAP ランタイムで、Unicode 文字または Unicode 文字エンコーディングのいずれかを使用できる固定サイズの文字列が含まれています。Unicode 以外の文字文字列は、バイトごとに 1 文字をエンコードします。Unicode 文字文字列は、UTF-16 エンコーディングを使用して 2 バイトでエンコードされます。文字フィールドの値は Java で java.lang.String オブジェクトとして表され、基礎となる JCo ランタイムは ABAP 表現への変換を行います。

文字フィールドは、関連付けられた byteLength および unicodeByteLength プロパティーでフィールドの長さを宣言します。これは、各エンコーディングシステムでフィールドの文字文字列の長さを決定します。

CHAR

CHAR 文字フィールドは英数字が含まれるテキストフィールドで、ABAP タイプ C に対応します。

NUM

NUM 文字フィールドは数値文字のみを含む数値テキストフィールドで、ABAP タイプ N に対応

します。

DATE

DATE 文字フィールドは、年、月、および日が YYYYMMDD としてフォーマットされた 8 文字の日付フィールドで、ABAP タイプ D に対応します。

TIME

TIME 文字フィールドは、時間、分、秒が HHMMSS としてフォーマットされた 6 文字の時間フィールドで、ABAP タイプ T に対応します。

数値フィールドタイプ

数値フィールドには数字が含まれます。以下の数字フィールドタイプがサポートされます。

INT

INT numeric フィールドは、基礎となる JCo および ABAP ランタイムの 4 バイトの整数値として保存される整数フィールドで、ABAP タイプ I に対応します。INT フィールドの値は、Java で java.lang.Integer オブジェクトとして表されます。

INT2

INT2 numeric フィールドは、基礎となる JCo および ABAP ランタイムの 2 バイトの整数値として保存される整数フィールドで、ABAP タイプ S に対応します。INT2 フィールドの値は、Java で java.lang.Integer オブジェクトとして表されます。

INT1

INT1 フィールドは、基礎となる JCo および ABAP ランタイム値の 1 バイトの整数値として保存される整数フィールドで、ABAP タイプ B に対応します。INT1 フィールドの値は、Java で java.lang.Integer オブジェクトとして表されます。

FLOAT

FLOAT フィールドは、基礎となる JCo および ABAP ランタイムの 8 バイトの 2 つの値として保存されるバイナリー浮動小数点数フィールドで、ABAP タイプ F に対応します。FLOAT フィールドは、フィールドの値が関連する 10 進数のプロパティーに含まれる 10 進数の数を宣言します。FLOAT フィールドでは、この 10 進数のプロパティーの値は 1 から 15 桁の値になります。FLOAT フィールド値は Java で java.lang.Double オブジェクトとして表されます。

BCD

BCD フィールドは、基礎となる JCo および ABAP ランタイムの 1 から 16 バイトパックの数字として保存されたバイナリーコード 10 進数フィールドで、ABAP タイプ P に対応します。パックされた数字は、1 バイトに 2 桁の数字を保存します。BCD フィールドは、関連付けられた byteLength および unicodeByteLength プロパティーでフィールドの長さを宣言します。BCD フィールドの場合、これらのプロパティーの値は 1 ~ 16 バイトになり、いずれのプロパティーも同じ値になります。BCD フィールドは、関連する decimal プロパティーでフィールドの値に含まれる 10 進数の数を宣言します。BCD フィールドでは、この 10 進数のプロパティーの値は 1 から 14 桁の値になります。BCD フィールド値は Java で java.math.BigDecimal として表されます。

DECF16

DECF16 フィールドは、基礎となる JCo ランタイムおよび ABAP ランタイムの浮動小数点値 8 バイトの IEEE 754 decimal64 浮動小数点値として保存され、ABAP タイプ decfloat16 に対応します。DECF16 フィールドの値には、16 桁の数字が含まれます。DECF16 フィールドの値は Java で java.math.BigDecimal として表されます。

DECF34

DECF34 フィールドは、基礎となる JCo ランタイムおよび ABAP ランタイムの浮動小数点値 16 バイト IEEE 754 decimal128 として保存され、ABAP タイプ decfloat34 に対応します。DECF34 フィールドの値には、34 桁の数字が含まれます。DECF34 フィールドの値は Java で java.math.BigDecimal として表されます。

16 進数フィールドタイプ

16 進数フィールドには raw バイナリーデータが含まれます。以下の 16 進数フィールドタイプがサポートされます。

BYTE

BYTE フィールドは、基礎となる JCo および ABAP ランタイムのバイト配列として保存される固定サイズのバイト文字列で、ABAP タイプ X に対応します。BYTE フィールドは関連する byteLength および unicodeByteLength プロパティーでフィールドの長さを宣言します。BYTE フィールドの場合、これらのプロパティーの値は 1 ~ 65535 バイトになり、両方のプロパティーの値が同じになります。BYTE フィールドの値は、Java で byte[] オブジェクトとして表されます。

文字列フィールドタイプ

文字列フィールドは、変数長の文字列値を参照します。この文字列値の長さは、ランタイムまで固定されません。文字列値のストレージは、基礎となる JCo および ABAP ランタイムに動的に作成されます。文字列フィールド自体のストレージが修正され、文字列ヘッダーのみが含まれます。

STRING

STRING フィールドは文字文字列を参照し、基礎となる JCo および ABAP ランタイムを 8 バイトの値として格納されます。ABAP タイプ G に対応します。STRING フィールドの値は Java で java.lang.String オブジェクトとして表されます。

XSTRING

XSTRING フィールドはバイト文字列を参照し、基礎となる JCo および ABAP ランタイムを 8 バイトの値として格納されます。ABAP タイプ Y に対応します。STRING フィールドの値は Java で byte[] オブジェクトとして表されます。

複雑なフィールドタイプ

複雑なフィールドは、構造フィールドまたはテーブルフィールドタイプのいずれかになります。以下の表では、これらの複雑なフィールドタイプをまとめています。

フィールドタイプ	対応する Java タイプ	Byte Length	Unicode バイトの長さ	Number Decimals Digits	
STRUCTURE	**org.fusesource.camel.component.sap.model.rfc.Structure**	個々のフィールドバイト長の合計	個々のフィールド Unicode バイト長の合計	-	ABAP Type
TABLE	**org.fusesource.camel.component.sap.model.rfc.Table**	行構造のバイト長	Unicode バイト長の行構造	-	ABAP タイ

構造化フィールドタイプ

STRUCTURE フィールドには構造オブジェクトが含まれ、下層の JCo および ABAP ランタイムに ABAP 構造レコードとして保存されます。ABAP タイプ u または v に対応します。STRUCTURE フィールドの値は、Java でインターフェイス org.fusesource.camel.component.sap.model.rfc.Structure を持つ構造オブジェクトとして表されます。

テーブルフィールドタイプ

TABLE フィールドにはテーブルオブジェクトが含まれ、下層の JCo および ABAP ランタイムに ABAP 内部テーブルとして保存されます。ABAP タイプ h に対応します。フィールドの値は、イン

ターフェイス org.fusesource.camel.component.sap.model.rfc.Table のあるテーブルオブジェクトによって Java で表されます。

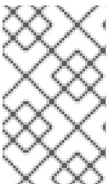
テーブルオブジェクト

テーブルオブジェクトは、同じ構造を持つ構造オブジェクトの行を含む同種のデータ構造です。このインターフェイスは、java.util.List インターフェイスと org.eclipse.emf.ecore.EObject インターフェイスの両方を拡張します。

```
public interface Table<S extends Structure>
    extends org.eclipse.emf.ecore.EObject,
    java.util.List<S> {

    /**
     * Creates and adds table row at end of row list
     */
    S add();

    /**
     * Creates and adds table row at index in row list
     */
    S add(int index);

}
```

テーブルオブジェクトの行一覧は、リストインターフェイスで定義された標準の方法を使用してアクセスおよび管理されます。また、テーブルインターフェイスは、構造オブジェクトを作成および行リストに追加する 2 つのファクトリーメソッドを提供します。

テーブルオブジェクトは、Eclipse Modeling Framework (EMF)を使用してコンポーネントランタイムに実装され、そのフレームワークの EObject インターフェイスをサポートします。テーブルオブジェクトのインスタンスには、提供する行の構造と内容を定義し、制限するメタデータがアタッチされます。このメタデータは、EMF が提供する標準の方法を使用してアクセスおよびイントロスペクションできます。詳細は、EMF のドキュメント を参照してください。



注記

間違ったタイプの行構造値を追加または設定しようとすると、例外が発生します。

140.4. IDOC のメッセージボディー

IDoc メッセージタイプ

IDoc Camel SAP エンドポイントのいずれかを使用する場合、メッセージボディーのタイプは、使用している特定のエンドポイントによって異なります。

sap-idoc-destination エンドポイントまたは sap-qidoc-destination エンドポイントの場合、メッセージボディーは Document タイプになります。

```
org.fusesource.camel.component.sap.model.idoc.Document
```

sap-idoclist-destination エンドポイント、sap-qidoclist-destination エンドポイント、または sap-idoclist-server エンドポイントの場合、メッセージボディーは DocumentList タイプになります。

```
org.fusesource.camel.component.sap.model.idoc.DocumentList
```

IDoc ドキュメントモデル

Camel SAP コンポーネントの場合、IDoc ドキュメントは Eclipse Modelling Framework (EMF)を使用してモデル化され、基礎となる SAP IDoc API に関するラッパー API を提供します。このモデルで最も重要なタイプは以下のとおりです。

```
org.fusesource.camel.component.sap.model.idoc.Document
org.fusesource.camel.component.sap.model.idoc.Segment
```

Document タイプは IDoc ドキュメントインスタンスを表します。概要として、Document インターフェイスは以下のメソッドを公開します。

```
// Java
package org.fusesource.camel.component.sap.model.idoc;
...
public interface Document extends EObject {
    // Access the field values from the IDoc control record
    String getArchiveKey();
    void setArchiveKey(String value);
    String getClient();
    void setClient(String value);
    ...

    // Access the IDoc document contents
    Segment getRootSegment();
}
```

以下の種類のメソッドは、Document インターフェイスによって公開されます。

コントロールレコードにアクセスする方法

メソッドのほとんどは、IDoc 制御レコードのフィールド値にアクセスまたは変更するためのものです。これらのメソッドは getAttributeName、setAttributeName の形式を取ります。AttributeName はフィールド値の名前です(表140.2「IDoc ドキュメントの属性」を参照)。

ドキュメントの内容にアクセスする方法

getRootSegment メソッドは、ドキュメントコンテンツ(IDoc データレコード)へのアクセスを提供し、コンテンツを Segment オブジェクトとして返します。各 Segment オブジェクトには任意の数の子セグメントを含めることができ、セグメントを任意のレベルでネストできます。

ただし、セグメント階層の正確なレイアウトは、ドキュメントの特定の IDoc タイプ で定義されることに注意してください。したがって、セグメント階層を作成（または読み取り）する場合、IDoc タイプで定義されている正確な構造に従うようにしてください。

Segment タイプは、IDoc ドキュメントのデータレコードにアクセスするために使用されます。セグメントは、ドキュメントの IDoc タイプで定義された構造に従って配置されます。概要として、Segment インターフェイスは以下のメソッドを公開します。

```
// Java
package org.fusesource.camel.component.sap.model.idoc;
...
public interface Segment extends EObject, java.util.Map<String, Object> {
    // Returns the value of the '<em><b>Parent</b></em>' reference.
    Segment getParent();

    // Return a immutable list of all child segments
    <S extends Segment> EList<S> getChildren();

    // Returns a list of child segments of the specified segment type.
    <S extends Segment> SegmentList<S> getChildren(String segmentType);

    EList<String> getTypes();

    Document getDocument();

    String getDescription();

    String getType();

    String getDefinition();

    int getHierarchyLevel();

    String getIdocType();

    String getIdocTypeExtension();
```

```
    String getSystemRelease();

    String getApplicationRelease();

    int getNumFields();

    long getMaxOccurrence();

    long getMinOccurrence();

    boolean isMandatory();

    boolean isQualified();

    int getRecordLength();

    <T> T get(Object key, Class<T> type);
}
```

getChildren(String segmentType) メソッドは、新しい（ネストされた）子をセグメントに追加する場合に特に便利です。以下のように定義されるタイプ SegmentList のオブジェクトを返します。

```
// Java
package org.fusesource.camel.component.sap.model.idoc;
...
public interface SegmentList<S extends Segment> extends EObject, EList<S> {
    S add();

    S add(int index);
}
```

したがって、E1SCU_CRE タイプのデータレコードを作成するには、以下のような Java コードを使用します。

```
Segment rootSegment = document.getRootSegment();

Segment E1SCU_CRE_Segment = rootSegment.getChildren("E1SCU_CRE").add();
```

IDoc が Document オブジェクトにどのように関連しているか

SAP ドキュメントによると、IDoc ドキュメントは以下の主要部分で設定されています。

制御レコード

制御レコード(IDoc ドキュメントのメタデータを含む)は、Document オブジェクトの属性で表されます。詳細は、表140.2「IDoc ドキュメントの属性」 を参照してください。

データレコード

データレコードは、Segment オブジェクトによって表され、セグメントのネストされた階層として構築されます。Document.getRootSegment メソッドを使用してルートセグメントにアクセスできます。

ステータスレコード

Camel SAP コンポーネントでは、ステータスレコードはドキュメントモデルによって表されません。ただし、コントロールレコードの status 属性を使用して、最新のステータス値にアクセスできます。

ドキュメントインスタンスの作成例

たとえば、例140.1「Java での IDoc ドキュメントの作成」 は、Java で IDoc モデル API を使用して、IDoc タイプ FLCUSTOMER_CREATEFROMDATA01 で IDoc ドキュメントを作成する方法を示しています。

例140.1 Java での IDoc ドキュメントの作成

```
// Java
import org.fusesource.camel.component.sap.model.idoc.Document;
import org.fusesource.camel.component.sap.model.idoc.Segment;
import org.fusesource.camel.component.sap.util.IDocUtil;

import org.fusesource.camel.component.sap.model.idoc.Document;
import org.fusesource.camel.component.sap.model.idoc.DocumentList;
import org.fusesource.camel.component.sap.model.idoc.IdocFactory;
import org.fusesource.camel.component.sap.model.idoc.IdocPackage;
import org.fusesource.camel.component.sap.model.idoc.Segment;
import org.fusesource.camel.component.sap.model.idoc.SegmentChildren;
...
//
// Create a new IDoc instance using the modelling classes
//


// Get the SAP Endpoint bean from the Camel context.
// In this example, it's a 'sap-idoc-destination' endpoint.
SapTransactionalIDocDestinationEndpoint endpoint =
    exchange.getContext().getEndpoint(
        "bean:SapEndpointBeanID",
        SapTransactionalIDocDestinationEndpoint.class
    );

// The endpoint automatically populates some required control record attributes
Document document = endpoint.createDocument()

// Initialize additional control record attributes
```

```
document.setMessageType("FLCUSTOMER_CREATEFROMDATA");
document.setRecipientPartnerNumber("QUICKCLNT");
document.setRecipientPartnerType("LS");
document.setSenderPartnerNumber("QUICKSTART");
document.setSenderPartnerType("LS");

Segment rootSegment = document.getRootSegment();

Segment E1SCU_CRE_Segment = rootSegment.getChildren("E1SCU_CRE").add();

Segment E1BPSCUNEW_Segment =
E1SCU_CRE_Segment.getChildren("E1BPSCUNEW").add();
E1BPSCUNEW_Segment.put("CUSTNAME", "Fred Flintstone");
E1BPSCUNEW_Segment.put("FORM", "Mr.");
E1BPSCUNEW_Segment.put("STREET", "123 Rubble Lane");
E1BPSCUNEW_Segment.put("POSTCODE", "01234");
E1BPSCUNEW_Segment.put("CITY", "Bedrock");
E1BPSCUNEW_Segment.put("COUNTR", "US");
E1BPSCUNEW_Segment.put("PHONE", "800-555-1212");
E1BPSCUNEW_Segment.put("EMAIL", "fred@bedrock.com");
E1BPSCUNEW_Segment.put("CUSTTYPE", "P");
E1BPSCUNEW_Segment.put("DISCOUNT", "005");
E1BPSCUNEW_Segment.put("LANGU", "E");
```

ドキュメント属性

表140.2「IDoc ドキュメントの属性」 は、Document オブジェクトに設定できる制御レコード属性を示しています。

表140.2 IDoc ドキュメントの属性

属性	長さ	SAP フィールド	説明
archiveKey	70	**ARCKEY**	EDI アーカイブキー
client	3	**MANDT**	クライアント
creationDate	8	**CREDAT**	日付 IDoc が作成されました

属性	長さ	SAP フィールド	説明
creationTime	6	**CRETIM**	時間 IDoc が作成されました
direction	1	**DIRECT**	方向
eDIMessage	14	**REFMES**	メッセージへの参照
eDIMessageGroup	14	**REFGRP**	メッセージグループへの参照
eDIMessageType	6	**STDMES**	EDI メッセージタイプ
eDIStandardFlag	1	**STD**	EDI 標準
eDIStandardVersion	6	**STDVRS**	EDI 標準のバージョン
eDITransmissionFile	14	**REFINT**	ファイル間の参照
iDocCompoundType	8	**DOCTYP**	IDoc タイプ
iDocNumber	16	**DOCNUM**	IDoc number
iDocSAPRelease	4	**DOCREL**	IDoc の SAP リリース

属性	長さ	SAP フィールド	説明
iDocType	30	**IDOCTP**	基本 IDoc タイプの名前
iDocTypeExtension	30	**CIMTYP**	エクステンションタイプの名前
messageCode	3	**MESCOD**	論理メッセージコード
messageFunction	3	**MESFCT**	論理メッセージ関数
messageType	30	**MESTYP**	論理メッセージタイプ
outputMode	1	**OUTMOD**	出力モード
recipientAddress	10	**RCVSAD**	受信者アドレス(SADR)
recipientLogicalAddress	70	**RCVLAD**	レシーバーの論理アドレス
recipientPartnerFunction	2	**RCVPFC**	レシーバーのパートナー機能
recipientPartnerNumber	10	**RCVPRN**	受信者のパートナー数
recipientPartnerType	2	**RCVPRT**	パートナータイプの受信者

属性	長さ	SAP フィールド	説明
recipientPort	10	**RCVPOR**	レシーバーポート(SAP System、EDI サブシステム)
senderAddress		**SNDSAD**	送信者アドレス(SADR)
senderLogicalAddress	70	**SNDLAD**	送信者の論理アドレス
senderPartnerFunction	2	**SNDPFC**	送信者のパートナーの機能
senderPartnerNumber	10	**SNDPRN**	パートナーの送信者数
senderPartnerType	2	**SNDPRT**	パートナータイプの送信者
senderPort	10	**SNDPOR**	送信者ポート(SAP System、EDI サブシステム)
serialization	20	**SERIAL**	EDI/ALE: シリアル化フィールド
status	2	**STATUS**	IDoc のステータス
testFlag	1	**TEST**	test フラグ

Java でのドキュメント属性の設定

Java でコントロールレコード属性を設定する場合(表140.2「IDoc ドキュメントの属性」から)、

Java Bean プロパティーの通常の規則に従います。つまり、name 属性は、属性値を取得および設定するには、getName および setName メソッドを使用してアクセスできます。たとえば、iDocType、iDocTypeExtension、および messageType 属性は、Document オブジェクトで以下のように設定できます。

```
// Java
document.setIDocType("FLCUSTOMER_CREATEFROMDATA01");
document.setIDocTypeExtension("");
document.setMessageType("FLCUSTOMER_CREATEFROMDATA");
```

XML でのドキュメント属性の設定

XML でコントロールレコード属性を設定する場合は、idoc:Document 要素に属性を設定する必要があります。たとえば、iDocType、iDocTypeExtension、および messageType 属性は以下のように設定できます。

```
<?xml version="1.0" encoding="ASCII"?>
<idoc:Document ...
            iDocType="FLCUSTOMER_CREATEFROMDATA01"
            iDocTypeExtension=""
            messageType="FLCUSTOMER_CREATEFROMDATA" ... >
   ...
</idoc:Document>
```

140.5. トランザクションサポート

BAPI トランザクションモデル

SAP コンポーネントは、SAP とのアウトバウンド通信用の BAPI トランザクションモデルをサポートします。transacted オプションが true に設定された URL を持つ宛先エンドポイントは、必要な場合は、エンドポイントのアウトバウンド接続でステートフルセッションを開始し、Camel Synchronization オブジェクトをエクスチェンジに登録します。この同期オブジェクトは BAPI サービスメソッド BAPI_TRANSACTION_COMMIT を呼び出して、メッセージ交換の処理が完了するとステートフルセッションを終了します。メッセージ交換の処理に失敗すると、同期オブジェクトは BAPI サーバーメソッド BAPI_TRANSACTION_ROLLBACK を呼び出してステートフルセッションを終了します。

RFC トランザクションモデル

tRFC プロトコルは、AT-MOST-ONCE 配信を実行し、各トランザクション識別子(TID)でトランザクション要求を識別することで保証を行います。TID には、プロトコルで送信される各リクエストが含まれます。tRFC プロトコルを使用する送信アプリケーションは、要求の送信時に一意の TID を持つリクエストの各インスタンスを特定する必要があります。アプリケーションは指定の TID でリクエストを複数回送信できますが、プロトコルにより、要求が最大 1 度受信システムで配信および処理されます。アプリケーションは、要求の送信時に通信またはシステムエラーが発生した場合に特定の TID でリクエストを再送信することを選択する可能性があるため、その要求が受信システムで配信および処理されたか

どうかが疑われます。通信エラーの発生時にリクエストを再送信することで、tRFC プロトコルを使用するクライアントアプリケーションは、EXACTLY-ONCE 配信を保証し、その要求の保証を処理することができます。

使用するトランザクションモデル

BAPI トランザクションはアプリケーションレベルのトランザクションです。これは、SAP データベースの BAPI メソッドまたは RFC 関数によって実行される永続データ変更に対して ACID 保証を課すという意味です。RFC トランザクションは通信トランザクションであり、BAPI メソッドおよび RFC 関数への要求に対する配信保証(AT-MOST-ONCE、EXACTLY-ONCE、EXACTLY-ONCE-IN-ORDER)を課します。

トランザクション RFC 宛先エンドポイント

以下の宛先エンドポイントは RFC トランザクションをサポートします。

- sap-trfc-destination
- sap-qrfc-destination

単一の Camel ルートには、複数のトランザクション RFC 宛先エンドポイントを含めることができ、複数の RFC 宛先にメッセージを送信したり、同じ RFC 宛先にメッセージを送信したりすることもできます。これは、Camel SAP コンポーネントが、ルートに渡される Exchange オブジェクトごとに 多数のトランザクション ID (TID)を追跡する必要があることを意味します。ルート処理に失敗し、再試行する必要がある場合、状況はかなり複雑になります。RFC トランザクションセマンティクスでは、ルートを介した各 RFC 宛先が、最初に使用されたのと 同じ TID を使用して呼び出す必要があります（および各宛先の TID が相互に区別されます）。つまり、Camel SAP コンポーネントは、ルートにどの TID が使用されたかを追跡し、TID が正しい順序で再生できるようにこの情報を覚えておく必要があります。

デフォルトでは、Camel は Exchange がルート内の場所を認識できるようにするメカニズムを提供しません。このようなメカニズムを提供するには、CurrentProcessorDefinitionInterceptStrategy インターセプターを Camel ランタイムにインストールする必要があります。Camel SAP コンポーネントがルート内の TID を追跡するには、このインターセプターを Camel ランタイムにインストールする必要があります。インターセプターの設定方法の詳細は、「tRFC および qRFC 宛先のインターセプター」 を参照してください。

トランザクション RFC サーバーエンドポイント

以下のサーバーエンドポイントは RFC トランザクションをサポートします。

- *sap-trfc-server*

トランザクションリクエストを処理する Camel エクスチェンジで処理エラーが発生すると、Camel は標準のエラー処理メカニズムを介して処理エラーを処理します。Camel ルートの処理が、エラーを呼び出し元に伝播するように設定されている場合、エクスチェンジを開始した SAP サーバーエンドポイントは失敗を認識し、送信している SAP システムにエラーが送信されます。送信元の SAP システムは、同じ TID で別のトランザクションリクエストを送信して再度要求を処理することで応答できます。

140.6. RFC の XML シリアライゼーション

概要

SAP 要求および応答オブジェクトは XML シリアライゼーション形式をサポートします。これにより、これらのオブジェクトを XML ドキュメントとの間でシリアライズすることができます。

XML namespace

リポジトリーの各 RFC は、Request および Response オブジェクトのシリアル化形式を設定する要素の特定の XML ネームスペースを定義します。この名前空間 URL の形式は以下のようになります。

```
http://sap.fusesource.org/rfc/<Repository Name>/<RFC Name>
```

RFC 名前空間 URL には共通の http://sap.fusesource.org/rfc 接頭辞があり、その後に RFC のメタデータが定義されているリポジトリーの名前が続きます。URL の最後のコンポーネントは RFC 自体の名前です。

リクエストおよび応答 XML ドキュメント

SAP リクエストオブジェクトは、Request という名前のドキュメントのルート要素を使用して XML ドキュメントにシリアライズされ、リクエストの RFC の namespace によってスコープが設定されます。

```
<?xml version="1.0" encoding="ASCII"?>
<BOOK_FLIGHT:Request
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
   ...
</BOOK_FLIGHT:Request>
```

SAP 応答オブジェクトは、Response という名前のドキュメントのルート要素を使用して XML ドキュメントにシリアライズされ、応答の RFC の namespace によってスコープが設定されます。

```
<?xml version="1.0" encoding="ASCII"?>
<BOOK_FLIGHT:Response
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
   ...
</BOOK_FLIGHT:Response>
```

構造フィールド

パラメーター一覧またはネストされた構造の構造フィールドは要素としてシリアライズされます。シリアライズされた構造の要素名は、エンクロージングパラメーターリスト、構造、またはテーブル行エントリー内の構造のフィールド名に対応します。

```
<BOOK_FLIGHT:FLTINFO
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
   ...
</BOOK_FLIGHT:FLTINFO>
```

RFC namespace の構造要素のタイプ名は、以下の例のように、構造を定義するレコードメタデータの名前に対応していることに注意してください。

```
<xs:schema
    targetNamespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
   ...
   <xs:complexType name="FLTINFO_STRUCTURE”>
   ...
   </xs:complexType>
   ...
</xs:schema>
```

この区別は、「例 3: SAP からのリクエストの処理」 で確認されるようにマーシャリングおよびアンマーシャリングする JAXB Bean を指定する場合に重要です。

テーブルフィールド

パラメーター一覧またはネストされた構造のテーブルフィールドは、要素としてシリアライズされます。シリアル化された構造の要素名は、存在するエンクロージングパラメーターリスト、構造、またはテーブル行エントリー内のテーブルのフィールド名に対応します。table 要素には、テーブルの行エントリーのシリアル化された値を保持する一連の row 要素が含まれます。

```
<BOOK_FLIGHT:CONNINFO
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
   <row ... > ... </row>
```

```
    ...
    <row ... > ... </row>
</BOOK_FLIGHT:CONNINFO>
```

RFC namespace の table 要素の type 名は、_TABLE のテーブルの行構造を定義するレコードメタデータオブジェクトの名前に対応していることに注意してください。RFC 名の table row 要素の type 名は、以下の例のようにテーブルの行構造を定義するレコードメタデータの名前に対応します。

```
<xs:schema
    targetNamespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    ...
    <xs:complextType name="CONNECTION_INFO_STRUCTURE_TABLE”>
        <xs:sequence>
            <xs:element
                name="row”
                minOccures="0”
                maxOccurs="unbounded”
                type="CONNECTION_INFO_STRUCTURE”/>
            ...
            <xs:sequence>
        </xs:sequence>
    </xs:complexType>

    <xs:complextType name="CONNECTION_INFO_STRUCTURE”>
            ...
    </xs:complexType>
    ...
</xs:schema>
```

この区別は、「例 3: SAP からのリクエストの処理」 で確認されるようにマーシャリングおよびアンマーシャリングする JAXB Bean を指定する場合に重要です。

要素フィールド

パラメーターリストの要素一覧または入れ子構造のフィールドは、エンクロージングパラメーターリストまたは構造の要素で属性としてシリアライズされます。以下の例のように、シリアル化されたフィールドの属性名は、エンクロージングパラメーターリスト、構造、またはテーブル行エントリー内のフィールド名に対応します。

```
<?xml version="1.0" encoding="ASCII"?>
<BOOK_FLIGHT:Request
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT"
    CUSTNAME="James Legrand"
   PASSFORM="Mr"
   PASSNAME="Travelin Joe"
   PASSBIRTH="1990-03-17T00:00:00.000-0500"
   FLIGHTDATE="2014-03-19T00:00:00.000-0400"
```

```
TRAVELAGENCYNUMBER="00000110"
DESTINATION_FROM="SFO"
DESTINATION_TO="FRA"/>
```

日付と時刻の形式

date および Time フィールドは、以下の形式で属性値にシリアライズされます。

```
yyyy-MM-dd'T'HH:mm:ss.SSSZ
```

日付フィールドは、設定された年、月、日、およびタイムゾーンコンポーネントのみでシリアライズされます。

```
DEPDATE="2014-03-19T00:00:00.000-0400"
```

時間フィールドは、hour、minute、second、millisecond、および timezone コンポーネントセットでのみシリアライズされます。

```
DEPTIME="1970-01-01T16:00:00.000-0500"
```

140.7. IDOC の XML シリアライゼーション

概要

IDoc メッセージボディーは、組み込み型コンバーターを使用して XML 文字列形式にシリアライズできます。

XML namespace

シリアル化された各 IDoc は、以下の一般的な形式を持つ XML 名前空間に関連付けられます。

```
http://sap.fusesource.org/idoc/repositoryName/idocType/idocTypeExtension/systemRelease/applicationRelease
```

repositoryName （リモート SAP メタデータリポジトリーの名前）と idocType (IDoc ドキュメントタイプ)の両方は必須ですが、名前空間の他のコンポーネントは空白のままにすることができます。たとえば、以下のような XML 名前空間を使用できます。

```
http://sap.fusesource.org/idoc/MY_REPO/FLCUSTOMER_CREATEFROMDATA01///
```

組み込み型コンバーター

Camel SAP コンポーネントには、Document 型との間で DocumentList オブジェクトまたは String オブジェクトを変換できる組み込み型コンバーターがあります。

たとえば、Document オブジェクトを XML 文字列にシリアライズするには、以下の行を XML DSL のルートに追加するだけです。

```
<convertBodyTo type="java.lang.String"/>
```

このアプローチを使用して、シリアライズされた XML メッセージを Document オブジェクトにすることもできます。たとえば、現在のメッセージボディーがシリアライズされた XML 文字列である場合、以下の行を XML DSL のルートに追加して、Document オブジェクトに戻すことができます。

```
<convertBodyTo type="org.fusesource.camel.component.sap.model.idoc.Document"/>
```

XML 形式の IDoc メッセージのボディーの例

IDoc メッセージを String に変換すると、XML ドキュメントにシリアライズされます。ルート要素は idoc:Document （単一ドキュメント用）または idoc:DocumentList （ドキュメントのリスト用）です。例140.2「XML の IDoc メッセージボディー」 は、idoc:Document 要素にシリアライズされた単一の IDoc ドキュメントを示しています。

例140.2 XML の IDoc メッセージボディー

```
<?xml version="1.0" encoding="ASCII"?>
<idoc:Document
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:FLCUSTOMER_CREATEFROMDATA01---
="http://sap.fusesource.org/idoc/XXX/FLCUSTOMER_CREATEFROMDATA01///"
  xmlns:idoc="http://sap.fusesource.org/idoc"
  creationDate="2015-01-28T12:39:13.980-0500"
  creationTime="2015-01-28T12:39:13.980-0500"
  iDocType="FLCUSTOMER_CREATEFROMDATA01"
  iDocTypeExtension=""
  messageType="FLCUSTOMER_CREATEFROMDATA"
  recipientPartnerNumber="QUICKCLNT"
  recipientPartnerType="LS"
  senderPartnerNumber="QUICKSTART"
  senderPartnerType="LS">
 <rootSegment xsi:type="FLCUSTOMER_CREATEFROMDATA01---:ROOT" document="/">
  <segmentChildren parent="//@rootSegment">
   <E1SCU_CRE parent="//@rootSegment" document="/">
    <segmentChildren parent="//@rootSegment/@segmentChildren/@E1SCU_CRE.0">
     <E1BPSCUNEW parent="//@rootSegment/@segmentChildren/@E1SCU_CRE.0"
       document="/"
```

```
                CUSTNAME="Fred Flintstone" FORM="Mr."
                STREET="123 Rubble Lane"
                POSTCODE="01234"
                CITY="Bedrock"
                COUNTR="US"
                PHONE="800-555-1212"
                EMAIL="fred@bedrock.com"
                CUSTTYPE="P"
                DISCOUNT="005"
                LANGU="E"/>
            </segmentChildren>
          </E1SCU_CRE>
        </segmentChildren>
      </rootSegment>
    </idoc:Document>
```

140.8. 例 1: SAP からのデータの読み取り

概要

この例は、SAP から FlightCustomer ビジネスオブジェクトデータを読み取るルートを示しています。ルートは、SAP 同期 RFC 宛先エンドポイントを使用して FlightCustomer BAPI メソッドを呼び出し、データを取得します。BAPI_FLCUST_GETLIST

ルートの Java DSL

サンプルルートの Java DSL は以下のとおりです。

```
from("direct:getFlightCustomerInfo")
    .to("bean:createFlightCustomerGetListRequest")
    .to("sap-srfc-destination:nplDest:BAPI_FLCUST_GETLIST")
    .to("bean:returnFlightCustomerInfo");
```

ルートの XML DSL

同じルートの Spring DSL は以下のようになります。

```
<route>
    <from uri="direct:getFlightCustomerInfo"/>
    <to uri="bean:createFlightCustomerGetListRequest"/>
    <to uri="sap-srfc-destination:nplDest:BAPI_FLCUST_GETLIST"/>
    <to uri="bean:returnFlightCustomerInfo"/>
</route>
```

createFlightCustomerGetListRequest bean

createFlightCustomerGetListRequest Bean は、後続の SAP エンドポイント の RFC 呼び出しで使用されるエクスチェンジメソッドで SAP 要求オブジェクトを構築します。以下のコードスニペットは、要求オブジェクトを構築するための操作シーケンスを示しています。

```
public void create(Exchange exchange) throws Exception {

    // Get SAP Endpoint to be called from context.
    SAPEndpoint endpoint =
        exchange.getContext().getEndpoint("bean:returnFlightCustomerInfo",
                                          SAPEndpoint.class);

    // Retrieve bean from message containing Flight Customer name to
    // look up.
    BookFlightRequest bookFlightRequest =
        exchange.getIn().getBody(BookFlightRequest.class);

    // Create SAP Request object from target endpoint.
    Structure request = endpoint.getRequest();

    // Add Customer Name to request if set
    if (bookFlightRequest.getCustomerName() != null &&
        bookFlightRequest.getCustomerName().length() > 0) {
            request.put("CUSTOMER_NAME",
                        bookFlightRequest.getCustomerName());
        }
    } else {
        throw new Exception("No Customer Name");
    }

    // Put request object into body of exchange message.
    exchange.getIn().setBody(request);
}
```

returnFlightCustomerInfo bean

returnFlightCustomerInfo Bean は、以前の SAP エンドポイント から受信するエクスチェンジメソッドで SAP 応答オブジェクトからデータを抽出します。以下のコードスニペットは、応答オブジェクトからデータを抽出する操作シーケンスを示しています。

```
public void createFlightCustomerInfo(Exchange exchange) throws Exception {

    // Retrieve SAP response object from body of exchange message.
    Structure flightCustomerGetListResponse =
        exchange.getIn().getBody(Structure.class);

    if (flightCustomerGetListResponse == null) {
        throw new Exception("No Flight Customer Get List Response");
    }
```

```
    // Check BAPI return parameter for errors
    @SuppressWarnings("unchecked")
    Table<Structure> bapiReturn =
       flightCustomerGetListResponse.get("RETURN", Table.class);
    Structure bapiReturnEntry = bapiReturn.get(0);
    if (bapiReturnEntry.get("TYPE", String.class) != "S") {
       String message = bapiReturnEntry.get("MESSAGE", String.class);
       throw new Exception("BAPI call failed: " + message);
    }

    // Get customer list table from response object.
    @SuppressWarnings("unchecked")
    Table<? extends Structure> customerList =
       flightCustomerGetListResponse.get("CUSTOMER_LIST", Table.class);

    if (customerList == null || customerList.size() == 0) {
       throw new Exception("No Customer Info.");
    }

    // Get Flight Customer data from first row of table.
    Structure customer = customerList.get(0);

    // Create bean to hold Flight Customer data.
    FlightCustomerInfo flightCustomerInfo = new FlightCustomerInfo();

    // Get customer id from Flight Customer data and add to bean.
    String customerId = customer.get("CUSTOMERID", String.class);
    if (customerId != null) {
       flightCustomerInfo.setCustomerNumber(customerId);
    }

    ...

    // Put bean into body of exchange message.
    exchange.getIn().setHeader("flightCustomerInfo", flightCustomerInfo);

 }
```

140.9. 例 2: SAP へのデータの書き込み

概要

この例は、SAP で FlightTrip ビジネスオブジェクトインスタンスを作成するルートを示しています。ルートは、FlightTrip BAPI メソッド BAPI_FLTRIP_CREATE を呼び出し、宛先エンドポイントを使用してオブジェクトを作成します。

ルートの Java DSL

サンプルルートの Java DSL は以下のとおりです。

```
from("direct:createFlightTrip")
    .to("bean:createFlightTripRequest")
    .to("sap-srfc-destination:nplDest:BAPI_FLTRIP_GETLIST?transacted=true")
    .to("bean:returnFlightTripResponse");
```

ルートの XML DSL

同じルートの Spring DSL は以下のようになります。

```
<route>
    <from uri="direct:createFlightTrip"/>
    <to uri="bean:createFlightTripRequest"/>
    <to uri="sap-srfc-destination:nplDest:BAPI_FLTRIP_GETLIST?transacted=true"/>
    <to uri="bean:returnFlightTripResponse"/>
</route>
```

トランザクションサポート

SAP エンドポイントの URL では、transacted オプションが true に設定されていることに注意してください。「トランザクションサポート」で説明されているように、このオプションを有効にすると、RFC 呼び出しを呼び出す前に SAP トランザクションセッションが開始されるようにします。このエンドポイントの RFC は SAP で新しいデータを作成するため、このオプションを使用してルートの変更を SAP で永続化する必要があります。

要求パラメーターの設定

createFlightTripRequest および returnFlightTripResponse Bean は、前述の例と同じ操作シーケンスに従って、SAP リクエストにリクエストパラメーターを設定し、SAP 応答から応答パラメーターを抽出するロールを果たします。

140.10. 例 3: SAP からのリクエストの処理

概要

この例は、SAP から BOOK_FLIGHT RFC への要求を処理するルートを示しています。これはルートによって実装されます。さらに、JAXB を使用して SAP リクエストオブジェクトおよび応答オブジェクトをカスタム Bean にアンマーシャリングおよびマーシャリングし、コンポーネントの XML シリアライゼーションサポートも示しています。

このルートは、移動エージェント FlightCustomer の代わりに FlightTrip ビジネスオブジェクトを作成します。ルートが最初に SAP サーバーエンドポイントによって受信された SAP リクエストオブジェクトをカスタム JAXB Bean にアンマーシャリングします。このカスタム Bean は、3 つのサブルートへの交換でマルチキャストされ、フライトのトリップの作成に必要な移動エージェント、フライト接

続、および乗客情報を収集します。最後のサブルートは、前の例に示すように、SAP でフライトトリップオブジェクトを作成します。最後のサブルートは、SAP 応答オブジェクトにマーシャリングされ、サーバーエンドポイントによって返されるカスタム JAXB Bean も作成および返します。

ルートの Java DSL

サンプルルートの Java DSL は以下のとおりです。

```
DataFormat jaxb = new JaxbDataFormat("org.fusesource.sap.example.jaxb");

from("sap-srfc-server:nplserver:BOOK_FLIGHT")
    .unmarshal(jaxb)
    .multicast()
    .to("direct:getFlightConnectionInfo",
        "direct:getFlightCustomerInfo",
        "direct:getPassengerInfo")
    .end()
    .to("direct:createFlightTrip")
    .marshal(jaxb);
```

ルートの XML DSL

同じルートの XML DSL は以下のようになります。

```
<route>
    <from uri="sap-srfc-server:nplserver:BOOK_FLIGHT"/>
    <unmarshal>
        <jaxb contextPath="org.fusesource.sap.example.jaxb"/>
    </unmarshal>
    <multicast>
        <to uri="direct:getFlightConnectionInfo"/>
        <to uri="direct:getFlightCustomerInfo"/>
        <to uri="direct:getPassengerInfo"/>
    </multicast>
    <to uri="direct:createFlightTrip"/>
    <marshal>
        <jaxb contextPath="org.fusesource.sap.example.jaxb"/>
    </marshal>
</route>
```

BookFlightRequest bean

以下のリストは、SAP BOOK_FLIGHT リクエストオブジェクトのシリアル化形式からアンマーシャリングする JAXB Bean を示しています。

```
@XmlRootElement(name="Request",
```

```
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class BookFlightRequest {

    @XmlAttribute(name="CUSTNAME")
    private String customerName;

    @XmlAttribute(name="FLIGHTDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date flightDate;

    @XmlAttribute(name="TRAVELAGENCYNUMBER")
    private String travelAgencyNumber;

    @XmlAttribute(name="DESTINATION_FROM")
    private String startAirportCode;

    @XmlAttribute(name="DESTINATION_TO")
    private String endAirportCode;

    @XmlAttribute(name="PASSFORM")
    private String passengerFormOfAddress;

    @XmlAttribute(name="PASSNAME")
    private String passengerName;

    @XmlAttribute(name="PASSBIRTH")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date passengerDateOfBirth;

    @XmlAttribute(name="CLASS")
    private String flightClass;

    ...
}
```

BookFlightResponse Bean

以下のリストは、SAP BOOK_FLIGHT 応答オブジェクトのシリアル化形式にマーシャリングする JAXB Bean を示しています。

```
@XmlRootElement(name="Response",
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class BookFlightResponse {

    @XmlAttribute(name="TRIPNUMBER")
    private String tripNumber;

    @XmlAttribute(name="TICKET_PRICE")
    private BigDecimal ticketPrice;

    @XmlAttribute(name="TICKET_TAX")
```

```
    private BigDecimal ticketTax;

    @XmlAttribute(name="CURRENCY")
    private String currency;

    @XmlAttribute(name="PASSFORM")
    private String passengerFormOfAddress;

    @XmlAttribute(name="PASSNAME")
    private String passengerName;

    @XmlAttribute(name="PASSBIRTH")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date passengerDateOfBirth;

    @XmlElement(name="FLTINFO")
    private FlightInfo flightInfo;

    @XmlElement(name="CONNINFO")
    private ConnectionInfoTable connectionInfo;


    ...
}
```

注記

応答オブジェクトの複雑なパラメーターフィールドは、応答の子要素としてシリアライズされます。

FlightInfo ビーン

以下のリストは、複雑な構造パラメーターのシリアル化形式にマーシャリングする JAXB Bean を示しています FLTINFO。

```
@XmlRootElement(name="FLTINFO",
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class FlightInfo {

    @XmlAttribute(name="FLIGHTTIME")
    private String flightTime;

    @XmlAttribute(name="CITYFROM")
    private String cityFrom;

    @XmlAttribute(name="DEPDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date departureDate;

    @XmlAttribute(name="DEPTIME")
```

```
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date departureTime;

    @XmlAttribute(name="CITYTO")
    private String cityTo;

    @XmlAttribute(name="ARRDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date arrivalDate;

    @XmlAttribute(name="ARRTIME")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date arrivalTime;

    ...
}
```

ConnectionInfoTable Bean

以下のリストは、複雑なテーブルパラメーターのシリアル化形式(CONNINFO)にマーシャリングする JAXB Bean を示しています。JAXB Bean のルート要素タイプの名前は、_TABLE で接尾辞が付けられた行構造タイプの名前に対応し、Bean には行要素のリストが含まれることに注意してください。

```
@XmlRootElement(name="CONNINFO_TABLE",
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class ConnectionInfoTable {

    @XmlElement(name="row")
    List<ConnectionInfo> rows;

    ...
}
```

ConnectionInfo bean

以下のリストは、上記のテーブル行要素のシリアル化された形式にマーシャリングする JAXB Bean を示しています。

```
@XmlRootElement(name="CONNINFO",
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class ConnectionInfo {

    @XmlAttribute(name="CONNID")
    String connectionId;

    @XmlAttribute(name="AIRLINE")
    String airline;
```

```
    @XmlAttribute(name="PLANETYPE")
    String planeType;

    @XmlAttribute(name="CITYFROM")
    String cityFrom;

    @XmlAttribute(name="DEPDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    Date departureDate;

    @XmlAttribute(name="DEPTIME")
    @XmlJavaTypeAdapter(DateAdapter.class)
    Date departureTime;

    @XmlAttribute(name="CITYTO")
    String cityTo;

    @XmlAttribute(name="ARRDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    Date arrivalDate;

    @XmlAttribute(name="ARRTIME")
    @XmlJavaTypeAdapter(DateAdapter.class)
    Date arrivalTime;

    ...
}
```

第141章 SAP NETWEAVER

SAP NETWEAVER GATEWAY コンポーネント

Camel 2.12 以降で利用可能

sap-netweaver は、HTTP トランスポートを使用して SAP NetWeaver Gateway と統合します。

この Camel コンポーネントはプロデューサーエンドポイントのみをサポートします。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-sap-netweaver</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

sap netweaver ゲートウェイコンポーネントの URI スキームは以下のとおりです。

```
sap-netweaver:https://host:8080/path?username=foo&password=secret
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

前提条件

このコンポーネントを活用するには、SAP NetWeaver システムへのアカウントが必要です。SAP は、アカウントに必要な デモ設定 を提供します。

このコンポーネントは、SAP NetWeaver にログインするために Basic 認証スキームを使用します。

コンポーネントおよびエンドポイントオプション

名前	デフォルト値	説明
username		アカウントのユーザー名。これは必須です。
password		アカウントのパスワードこれは必須です。
json	**true**	JSON 形式でデータを返すかどうか。このオプションが false の場合、XML は Atom 形式で返されます。
jsonAsMap	**true**	JSON を文字列からメッセージボディーの Map に変換するには、以下を行います。
flatternMap	**true**	JSON Map に単一のエントリーのみが含まれる場合、その単一のエントリー値をメッセージボディーとして保存し、フラット化します。

メッセージヘッダー

以下のヘッダーはプロデューサーで使用できます。

名前	タイプ	説明
CamelNetWeaverCommand	**文字列**	**必須**: MS ADO.Net Data Service 形式で実行するコマンド。

例

この例は、SAP のフライトデモの例を使用しています。これは、インターネット上でオンライン で利用できます。

以下のルートでは、以下の URL を使用して SAP NetWeaver デモサーバーを要求します。

```
https://sapes1.sapdevcenter.com/sap/opu/odata/IWBEP/RMTSAMPLEFLIGHT_2/
```

そして、次のコマンドを実行します。

```
FlightCollection(AirLineID='AA',FlightConnectionID='0017',FlightDate=datetime'2012-08-29T00%3A00%3A00')
```

*指定のフライトの詳細を取得するには、以下を実行します。コマンド構文は **MS ADO.Net Data Service** 形式です。*

以下の Camel ルートがあります。

```
from("direct:start")
    .toF("sap-netweaver:%s?username=%s&password=%s", url, username, password)
    .to("log:response")
    .to("velocity:flight-info.vm")
```

ここで、url、username、および password は以下のように定義されます。

```
    private String username = "P1909969254";
```

```
    private String password = "TODO";
    private String url =
"https://sapes1.sapdevcenter.com/sap/opu/odata/IWBEP/RMTSAMPLEFLIGHT_2/";
    private String command =
"FlightCollection(AirLineID='AA',FlightConnectionID='0017',FlightDate=datetime'2012-08-29T00%3A00%3A00')";
```

パスワードが無効です。デモを実行するには、最初に SAP にアカウントを作成する必要があります。

velocity テンプレートは、基本的な HTML ページへの応答のフォーマットに使用されます。

```
<html>
  <body>
  Flight information:

  <p/>
  <br/>Airline ID: $body["AirLineID"]
  <br/>Aircraft Type: $body["AirCraftType"]
  <br/>Departure city: $body["FlightDetails"]["DepartureCity"]
  <br/>Departure airport: $body["FlightDetails"]["DepartureAirPort"]
  <br/>Destination city: $body["FlightDetails"]["DestinationCity"]
  <br/>Destination airport: $body["FlightDetails"]["DestinationAirPort"]

  </body>
</html>
```

アプリケーションの実行時に、sampel の出力を取得します。

```
Flight information:
Airline ID: AA
Aircraft Type: 747-400
Departure city: new york
Departure airport: JFK
Destination city: SAN FRANCISCO
Destination airport: SFO
```

-

HTTP

第142章 スケジューラー

スケジューラーコンポーネント

Camel 2.15 以降で利用可能

scheduler: コンポーネントは、スケジューラーの実行時にメッセージ交換を生成するために使用されます。このコンポーネントは Timer コンポーネントと似ていますが、スケジューリングの面でより多くの機能を提供します。また、このコンポーネントは JDK ScheduledExecutorService を使用します。タイマーは JDK Timer を使用します。

このエンドポイントからのイベントのみを使用できます。

URI 形式

```
scheduler:name[?options]
```

name はスケジューラーの名前で、エンドポイント間で作成および共有されます。したがって、すべてのタイマーエンドポイントに同じ名前を使用する場合は、1 つのスケジューラースレッドプールとスレッドのみが使用されます。ただし、スレッドプールは、より多くの同時スレッドを許可するように設定できます。

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

注記： 生成されたエクスチェンジの IN ボディーは null です。exchange.getIn().getBody() は null を返します。

オプション

名前	デフォルト値	説明
initialDelay	1000	最初のポーリングが開始するまでの時間（ミリ秒単位）
delay	500	次のポーリングまでの時間（ミリ秒単位）

timeUnit	**MILLISECONDS**	**initialDelay** および **delay** オプションの時間単位。
useFixedDelay	**true**	固定遅延または固定レートを使用するかどうかを制御します。詳細は、JDK の ScheduledExecutorService を参照してください。
pollStrategy		プラグ可能な **org.apache.camel.PollingConsumerPollingStrategy** により、エクスチェンジ が作成され、Camel でルーティングされる **前** に、通常 **poll** 操作中に発生するエラー処理を制御するカスタム実装を提供できます。つまり、ポーリングにより情報を収集しているときにエラーが発生し (例えばファイルネットワークへのアクセスに失敗した)、Camel はそれにアクセスしてファイルをスキャンできません。デフォルトの実装は、**WARN** レベルで原因となった例外をログに記録し、無視します。
runLoggingLevel	**TRACE**	コンシューマーはポーリング時に開始/完了のログ行を記録します。このオプションを使用すると、ログレベルを設定できます。
sendEmptyMessageWhenIdle	**false**	ポーリングコンシューマーがファイルをポーリングしなかった場合、このオプションを有効にして、代わりに空のメッセージ (ボディーなし) を送信できます。
greedy	**false**	greedy が有効で、以前の実行が 1 つ以上のメッセージをポーリングした場合、ScheduledPollConsumer は即座に再度実行されます。

scheduler		ポーリングコンシューマーの実行時に実行するスケジューラーとして使用するカスタム **org.apache.camel.spi.ScheduledPollConsumerScheduler** をプラグインできるようにします。デフォルトの実装は**ScheduledExecutorService** を使用し、CRON 式をサポートする Quartz2 および Spring ベースがあります。**注記：** カスタムスケジューラーを使用する場合は、**initialDelay**、**useFixedDelay**、**timeUnit**、および**scheduledExecutorService** のオプションが使用されていない可能性があります。テキスト **quartz2** を使用して Quartz2 スケジューラーを使用し、テキスト **spring** を使用して Spring ベースを使用し、テキスト **#myScheduler** を使用してレジストリー の ID でカスタムスケジューラーを参照し ます。例については、Quartz2 ページを参照してください。
scheduler.xxx		カスタム **scheduler** または Quartz2、Spring ベースのスケジューラーのいずれかを使用する場合に追加のプロパティーを設定します。たとえば、Spring ベースのスケジューラーに cron 値を指定するには、**scheduler.cron** を使用します。
backoffMultiplier	**0**	後続のアイドル状態/エラーが連続して発生した場合に、スケジュールされたポーリングコンシューマーのバックオフを許可します。乗数は、実際に次の試行が行われる前にスキップされるポーリングの数です。このオプションを使用する場合は、**backoffIdleThreshold** や **backoffErrorThreshold** も設定する必要があります。
backoffIdleThreshold	**0**	**backoffMultipler** が開始する前に発生する必要のある後続のアイドルポーリングの数

backoffErrorThreshold	0	backoffMultipler が開始する前に発生すべき後続のエラーポーリングの数（エラーにより失敗する）。

補足情報

このコンポーネントはスケジューラー ポーリングコンシューマー で、上記のオプションに関する詳細情報と、Polling Consumer ページの例を確認できます。

エクスチェンジプロパティー

タイマーが実行されると、以下の情報をプロパティーとして Exchange に追加します。

名前	タイプ	説明
Exchange.TIMER_NAME	String	name オプションの値。
Exchange.TIMER_FIRED_TIME	Date	コンシューマーが実行した時間。

例

60 秒ごとにイベントを生成するルートを設定するには、以下を実行します。

```
from("scheduler://foo?period=60s").to("bean:myBean?method=someMethodName");
```

上記のルートはイベントを生成し、JNDI や Spring などの レジストリー の myBean という Bean で someMethodName メソッドを呼び出します。

Spring DSL のルートは以下のようになります。

```
<route>
  <from uri="scheduler://foo?period=60s"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

完了するとスケジューラーが即座にトリガーされるように強制します。

*以前のタスクの完了直後にスケジューラーをトリガーできるようにするには、オプション **greedy=true** を設定します。ただし、スケジューラーは常に実行を続けます。したがって、これは注意して使用してください。*

スケジューラーがアイドル状態になる

*スケジューラーをトリガーして増やすユースケースがあります。ただし、ポーリングするタスクがないためスケジューラーを **"tell the scheduler"** する必要があるかもしれません。そのため、スケジューラーはバックオフオプションを使用して **idle** モードに変更できます。これを行うには、鍵 **Exchange.SCHEDULER_POLLED_MESSAGES** を持つエクスチェンジでプロパティーをブール値 **false** に設定する必要があります。これにより、コンシューマーはポーリングされたメッセージがないことを示します。*

コンシューマーは、コンシューマーがエクスチェンジの処理を完了するたびに、スケジューラーにポーリングされた 1 メッセージを返します。

- *Timer*
- *Quartz*

第143章 SCHEMATRON

SCHEMATRON コンポーネント

Camel 2.14 から利用可能

Schematron は XML インスタンスドキュメントを検証する XML ベースの言語です。これは、XML ドキュメントのデータに関するアサーションを作成するために使用され、運用ルールとビジネスルールの表現にも使用されます。Schematron は ISO Standard です。schematron コンポーネントは、ISO schematron の主要な 実装 を使用します。これは XSLT ベースの実装です。schematron ルールは 4 つの XSLT パイプライン で実行されます。これは、XML ドキュメントに対してアサーションを実行するためのベースとして使用される最終的な XSLT を生成します。コンポーネントは、エンドポイントの開始時に Schematron ルールが読み込まれるように記述されます（一度のみ）。これは、ルールを表す Java テンプレートオブジェクトをインスタンス化するオーバーヘッドを最小限に抑えることです。

URI 形式

```
schematron://path?[options]
```

URI オプション

名前	デフォルト値	説明
path	必須	schematron ルールファイルへのパス。ファイルシステムのクラスパスまたは場所のいずれかになります。
強制終了	false	ルートを中止し、スキーマ検証例外を出力するフラグ。

HEADERS

名前	説明	タイプ	In/Out
CamelSchematronValidationStatus	schematron 検証ステータス：SUCCESS / FAILED	文字列	IN
CamelSchematronValidationReport	XML 形式のスキーマレポートボディー。以下の例を参照してください。	文字列	IN

URI およびパス構文

以下の例は、Java DSL で schematron プロセッサーを呼び出す方法を示しています。schematron ルールファイルはクラスパスから取得されます。

```
from("direct:start").to("schematron://sch/schematron.sch").to("mock:result")
```

以下の例は、XML DSL で schematron プロセッサーを呼び出す方法を示しています。スキーマルールファイルは、ファイルシステムから取得されます。

```
<route>
  <from uri="direct:start" />
  <to uri="schematron:///usr/local/sch/schematron.sch" />
  <log message="Schematron validation status: ${in.header.CamelSchematronValidationStatus}" />
  <choice>
    <when>
      <simple>${in.header.CamelSchematronValidationStatus} == 'SUCCESS'</simple>
      <to uri="mock:success" />
    </when>
    <otherwise>
      <log message="Failed schematron validation" />
      <setBody>
        <header>CamelSchematronValidationReport</header>
      </setBody>
      <to uri="mock:failure" />
    </otherwise>
  </choice>
</route>
```

SCHEMATRON ルールの保存先

Schematron ルールはビジネス要件で変更できるため、これらのルールをファイルシステムのどこかに保存することが推奨されます。schematron コンポーネントのエンドポイントが起動すると、ルールは Java Templates Object として XSLT にコンパイルされます。これは、Java テンプレートオブジェクトのインスタンス化のオーバーヘッドを最小限にするために 1 回のみ実行されます。これは、大規模なルールセットのコストのかかる操作であり、プロセスが XSLT 変換 の 4 つのパイプラインを通過することです。したがって、更新をファイルシステムに保存する場合は、ルートまたはコンポーネントを再起動する必要があります。ただし、これらのルールをクラスパスに保存するには害はありませんが、変更を取得するにはコンポーネントをビルドし、デプロイする必要があります。

SCHEMATRON ルールおよびレポートサンプル

schematron ルールの例を以下に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <title>Check Sections 12/07</title>
```

```
  <pattern id="section-check">
    <rule context="section">
      <assert test="title">This section has no title</assert>
      <assert test="para">This section has no paragraphs</assert>
    </rule>
  </pattern>
</schema>
```

以下は、schematron レポートの例です。

```
<?xml version="1.0" encoding="UTF-8"?>
<svrl:schematron-output xmlns:svrl="http://purl.oclc.org/dsdl/svrl"
 xmlns:iso="http://purl.oclc.org/dsdl/schematron"
 xmlns:saxon="http://saxon.sf.net/"
 xmlns:schold="http://www.ascc.net/xml/schematron"
 xmlns:xhtml="http://www.w3.org/1999/xhtml"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" schemaVersion="" title="">

  <svrl:active-pattern document="" />
  <svrl:fired-rule context="chapter" />
  <svrl:failed-assert test="title" location="/doc[1]/chapter[1]">
    <svrl:text>A chapter should have a title</svrl:text>
  </svrl:failed-assert>
  <svrl:fired-rule context="chapter" />
  <svrl:failed-assert test="title" location="/doc[1]/chapter[2]">
    <svrl:text>A chapter should have a title</svrl:text>
  </svrl:failed-assert>
  <svrl:fired-rule context="chapter" />
</svrl:schematron-output>
```

関連リンクおよびリソース

- Mulleberry テクノロジー による Schematron の概要Schematron を使い始めるための優れたドキュメントです。

- Schematron の公式サイト これには他のリソースへのリンクが含まれます。

第144章 SEDA

SEDA コンポーネント

seda: コンポーネントは非同期 SEDA 動作を提供するため、メッセージは BlockingQueue で交換され、コンシューマーはプロデューサーとは別のスレッドで呼び出されます。

キューは 単一 の CamelContext 内でのみ表示されることに注意してください。CamelContext インスタンス(Web アプリケーション間の通信など)間で通信する場合は、仮想マシン コンポーネントを参照してください。

このコンポーネントは、メッセージの処理中に仮想マシンが終了した場合に、永続性やリカバリーを実装しません。永続性、信頼性、または分散 SEDA が必要な場合は、JMS または ActiveMQ のいずれかを使用してください。

同期

Direct コンポーネントは、プロデューサーがメッセージエクスチェンジを送信するときに、コンシューマーの同期呼び出しを提供します。

URI 形式

```
seda:queueName[?options]
```

queueName には、現在の CamelContext 内のエンドポイントを一意に識別する任意の文字列を指定できます。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

> **注記**
>
> コンシューマーをプロデューサーエンドポイントに一致させると、queueName のみが考慮され、オプション設定は無視されます。つまり、コンシューマーエンドポイントのアイデンティティーは queueName にのみ依存します。複数のコンシューマーを同じキューに割り当てる場合は、「multipleConsumers の使用」 で説明されている方法を使用します。

オプション

名前	デフォルト	説明
size	unbounded	SEDA キューの最大容量（つまり、保持できるメッセージの数）。**注：** このオプションを使用する場合、キュー名で最初のエンドポイントが作成され、サイズを決定するのは最小限です。すべてのエンドポイントが同じサイズを使用するようにするには、すべてのエンドポイントで size オプションを設定するか、作成される最初のエンドポイントを設定します。**Camel 2.11** 以降では、同じキュー名に混合キューサイズを使用すると、Camel はこれを検出し、エンドポイントの作成に失敗するように検証が行われます。
concurrentConsumers	**1**	**Apache Camel 1.6.1/2.0:** 同時スレッド処理エクスチェンジの数。
waitForTaskToComplete	**IfReplyExpected**	非同期タスクが完了するまで呼び出し元が待機するかどうかを指定するオプション。**Always**、**Never**、または **IfReplyExpected** の 3 つのオプションがサポートされます。最初の 2 つの値は自己説明です。最後の値 **IfReplyExpected** は、メッセージが Request Reply-based である場合にのみ待機します。デフォルトのオプションは **IfReplyExpected** です。非同期メッセージングの詳細は、非同期メッセージング を参照してください。
timeout	**30000**	**Apache Camel 2.0:** Timeout in millis a seda producer will at most waiting for an async task to complete.詳細は、**waitForTaskToComplete** および Async を参照してください。**Camel 2.2** では、0 または負の値を使用してタイムアウトを無効にできるようになりました。

multipleConsumers	**false**	**Camel 2.2:** 複数のコンシューマーを許可するかどうかを指定します。有効にすると、メッセージングの公開/サブスクライブスタイルに SEDA を使用できます。SEDA キューにメッセージを送信し、複数のコンシューマーがメッセージのコピーを受信します。
limitConcurrentConsumers	**true**	**Camel 2.3:** concurrentConsumers を最大 500 に制限するかどうか。数値が大きい で設定された場合は、例外が発生します。このオプションをオフにすると、このチェックを無効にできます。
blockWhenFull	**false**	メッセージを完全な SEDA キューに送信するスレッドが、キューの容量が枯渇しなくなるまでブロックされるかどうか。デフォルトでは、キューがいっぱいであることを示す例外が出力されます。このオプションを有効にすると、呼び出しスレッドは代わりにブロックされ、メッセージが受け入れられるまで待機します。
queueSize		**コンポーネントのみ：** SEDA キューのデフォルトサイズ（保持可能なメッセージ数の容量）。このオプションは、**サイズ** が使用されていない場合に使用されます。
pollTimeout	**1000**	**コンシューマーのみ**: ポーリング時に使用されるタイムアウト。タイムアウトが発生すると、コンシューマーは実行を継続できるかどうかを確認できます。値を低く設定すると、シャットダウン時にコンシューマーがより迅速に対応できるようになります。
purgeWhenStopping	**false**	コンシューマー/ルートを停止するときにタスクキューをパージするかどうか。キューの保留中のメッセージは破棄されるため、はより迅速に停止できます。
queue	**null**	seda エンドポイントで使用されるキューインスタンスを定義します。

queueFactory	**null**	seda エンドポイントのキューを作成できる QueueFactory を定義します。
failIfNoConsumers	**false**	アクティブなコンシューマーのない SEDA キューに送信する際に、プロデューサーが例外を出力して失敗するかどうか。オプション **discardIfNoConsumers** と **failIfNoConsumers** を同時に有効にできるのは 1 つだけです。
discardIfNoConsumers	**false**	アクティブなコンシューマーのない SEDA キューに送信する際に、プロデューサーがメッセージを破棄するかどうか（メッセージをキューに追加しないでください）。オプション **discardIfNoConsumers** と **failIfNoConsumers** を同時に有効にできるのは 1 つだけです。

BLOCKINGQUEUE 実装の選択

Camel 2.12 以降で利用可能

デフォルトでは、SEDA コンポーネントは常に LinkedBlockingQueue になりますが、異なる実装を使用できます。この場合、size オプションは使用されません。

```
<bean id="arrayQueue" class="java.util.ArrayBlockingQueue">
<constructor-arg index="0" value="10" ><!-- size -->
<constructor-arg index="1" value="true" ><!-- fairness -->
</bean>
<!-- ... and later -->
<from>seda:array?queue=#arrayQueue</from>
```

または、BlockingQueueFactory 実装を参照することもできます。3 つの実装は LinkedBlockingQueueFactory、ArrayBlockingQueueFactory、および PriorityBlockingQueueFactory で提供されます。

```
<bean id="priorityQueueFactory"
class="org.apache.camel.component.seda.PriorityBlockingQueueFactory">
<property name="comparator">
```

```
<bean class="org.apache.camel.demo.MyExchangeComparator" />
</property>
</bean>
<!-- ... and later -->
<from>seda:priority?queueFactory=#priorityQueueFactory&size=100</from>
```

リクエスト応答の使用

SEDA コンポーネントは、リクエスト応答の使用 をサポート します。ここでは、呼び出し元は Async ルートが完了するまで待機します。たとえば、以下のようになります。

```
from("mina:tcp://0.0.0.0:9876?textline=true&sync=true").to("seda:input");

from("seda:input").to("bean:processInput").to("bean:createResponse");
```

上記のルートでは、受信リクエストを受け入れるポート 9876 の TCP リスナーがあります。リクエストは seda:input キューにルーティングされます。Request Reply メッセージであるため、応答を待ちます。seda:input キューのコンシューマーが完了すると、応答が元のメッセージの応答にコピーされます。

2.2 まで：2 つのエンドポイントでのみ機能する

SEDA または VM 上で リクエスト応答 を使用すると、2 つのエンドポイントでのみ機能します。A -> B -> C などに送信してエンドポイントを連鎖させることは できません。A -> B の間のみ。理由は、実装ロジックは非常に簡単です。3+ エンドポイントをサポートするために、待機中のスレッド間の順序付けと通知を適切に処理するためにロジックが複雑になります。

これは Camel 2.3 以降では改善され、必要な数だけエンドポイントを連鎖できるようになりました。

同時コンシューマー

デフォルトでは、SEDA エンドポイントは単一のコンシューマースレッドを使用しますが、同時コンシューマースレッドを使用するように設定できます。そのため、スレッドプールの代わりに以下を使用できます。

```
from("seda:stageName?concurrentConsumers=5").process(...)
```

スレッドプールと同時コンシューマーの違い

スレッドプール は、負荷に応じてランタイム時に動的に拡大/縮小できるプールですが、同時コンシューマーは常に固定されます。

スレッドプール

以下のようにしてスレッドプールを SEDA エンドポイントに追加することに注意してください。

```
from("seda:stageName").thread(5).process(...)
```

は、SEDA エンドポイントからの 2 つの BlockQueues と、スレッドプールの workqueue からの 1 つで優先され、望ましいものではない可能性があります。代わりに、スレッドプールで Direct エンドポイントを設定することを検討してください。これは、同期的かつ非同期的にメッセージを処理できます。以下に例を示します。

```
from("direct:stageName").thread(5).process(...)
```

concurrentConsumers オプションを使用して SEDA エンドポイントでメッセージを処理するスレッド数を直接設定することもできます。

例

以下のルートでは、SEDA キューを使用してこの非同期キューにリクエストを送信し、別のスレッドでさらに処理するために fire-and-forget メッセージを送信し、このスレッドの定数応答を元の呼び出し元に返します。

```
public void configure() throws Exception {
    from("direct:start")
        // send it to the seda queue that is async
        .to("seda:next")
        // return a constant response
        .transform(constant("OK"));

    from("seda:next").to("mock:result");
}
```

ここでは、Hello World メッセージを送信し、応答が OK であることが想定されます。

```
Object out = template.requestBody("direct:start", "Hello World");
assertEquals("OK", out);
```

Hello World メッセージは、さらに処理するために別のスレッドから SEDA キューから消費されます。これはユニットテストからのものであるため、ユニットテストでアサーションを実行できる モックエンドポイントに送信されます。

MULTIPLECONSUMERS の使用

Camel 2.2 で利用可能

この例では、2 つのコンシューマーを定義し、Spring Bean として登録しました。

```
<!-- define the consumers as spring beans -->
<bean id="consumer1" class="org.apache.camel.spring.example.FooEventConsumer"/>

<bean id="consumer2"
class="org.apache.camel.spring.example.AnotherFooEventConsumer"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
   <!-- define a shared endpoint which the consumers can refer to instead of using url -->
   <endpoint id="foo" uri="seda:foo?multipleConsumers=true"/>
</camelContext>
```

seda foo エンドポイントに multipleConsumers=true を指定しているため、これら 2 つのコンシューマーは、種類の pub-sub スタイルのメッセージングとしてメッセージの独自のコピーを受け取ることができます。

Bean はユニットテストの一部であるため、単にモックエンドポイントにメッセージを送信しますが、@Consume を使用して seda キューから消費する方法に注目してください。

```
public class FooEventConsumer {

   @EndpointInject(uri = "mock:result")
   private ProducerTemplate destination;

   @Consume(ref = "foo")
   public void doSomething(String body) {
      destination.sendBody("foo" + body);
   }

}
```

キュー情報の抽出。

必要な場合、以下のように JMX を使用せずにキューサイズなどの情報を取得することもできます。

```
SedaEndpoint seda = context.getEndpoint("seda:xxxx");
int size = seda.getExchanges().size()
```

- *Disruptor*
- *VM*
- *Direct*

第145章 SERVICENOW

SERVICENOW コンポーネント

ServiceNow コンポーネントは、すべての ServiceNow REST API へのアクセスを提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-servicenow</artifactId>
    <version>${camel-version}</version>
</dependency>
```

URI 形式

```
servicenow://InstanceName[?Options]
```

オプション

名前	デフォルト値	説明
userName	**null**	認証に使用するユーザー名
password	**null**	認証に使用するパスワード
oauthClientId	**null**	OAuth2 クライアント ID
oauthClientSecret	**null**	OAuth2 クライアントシークレット
oauthTokenUrl	**https://incenceName.service-now.com/oauth_token.do**	OAuth2 トークン URL
apiUrl	**https://incenceName.service-now.com/api/now**	ServiceNow API URL
table	**null**	デフォルトのテーブルはヘッダー CamelServiceNowTable で上書きできます。
excludeReferenceLink	**false**	true: 参照フィールドのテーブル API リンクを除外する

suppressAutoSysField	**false**	True: システムフィールドの自動生成を抑制します。
displayValue	**false**	参照フィールドの表示値(true)、実際の値(false)、またはその両方（すべて）を返します（デフォルト：false）。
inputDisplayValue	**false**	true: 入力フィールドに raw 値を設定します。
models	**null**	テーブルに使用するデフォルトのモデルを定義します (model.incident = my.company.model.Incident)。
mapper		ServiceNow コンポーネントは Jackson Databind を使用してリクエスト/応答を Json との間で変換し、カスタム ObjectMapper を指定してその実行方法をカスタマイズできます。

HEADERS

名前	タイプ	説明
CamelServiceNowResource	**String**	TABLE、AGGREGATE、IMPORT にアクセスするためのリソース
CamelServiceNowTable	**String**	アクセスするテーブル
CamelServiceNowAction	**String**	RETRIEVE、CREATE、MODIFY、DELETE、UPDATE を実行するアクション
CamelServiceNowModel	**Class**	データモデル
CamelServiceNowSysId	**String**	ServiceNow sysy_id
CamelServiceNowQuery	**String**	エンコードされたクエリー
CamelServiceNowDisplayValue	**String**	参照フィールドの表示値(true)、実際の値(false)、またはその両方（すべて）を返します（デフォルト：false）。
CamelServiceNowInputDisplayValue	**Boolean**	true: 入力フィールドに raw 値を設定します。

CamelServiceNowExcludeReferenceLink	**Boolean**	true: 参照フィールドのテーブル API リンクを除外する
CamelServiceNowFields	**String**	応答で返すコンマ区切りのフィールド名
CamelServiceNowMinFields	**String**	最小値を計算するフィールドのコンマ区切りリスト
CamelServiceNowMaxFields	**String**	最大値を計算するフィールドのコンマ区切りリスト
CamelServiceNowSumFields	**String**	値の合計を計算するフィールドのコンマ区切りリスト
CamelServiceNowLimit	**Integer**	ページネーションに適用される制限
CamelServiceNowView	**String**	UI ビュー。応答で返されるフィールドを決定します。
CamelServiceNowSuppressAutoSysField	**Boolean**	True: システムフィールドの自動生成を抑制します。
CamelServiceNowAvgFields	**String**	平均値を計算するフィールドのコンマ区切りリスト
CamelServiceNowCount	**Boolean**	ブール値フラグ。クエリーによって返されるレコードの数については、このパラメーターを **true** に設定します。
CamelServiceGroupBy	**String**	返されたデータをグループ化するフィールド
CamelServiceOrderBy	**String**	グループ化された結果を順序付ける値の一覧
CamelServiceHaving	**String**	集約操作に基づいてデータをフィルターリングできる追加のクエリー

使用例

```
context.addRoutes(new RouteBuilder() {
   public void configure() {
     from("direct:servicenow")
        .to("servicenow:{{env:SERVICENOW_INSTANCE}}"
           + "?userName={{env:SERVICENOW_USERNAME}}"
```

```
            + "&password={{env:SERVICENOW_PASSWORD}}"
            + "&oauthClientId={{env:SERVICENOW_OAUTH2_CLIENT_ID}}"
            + "&oauthClientSecret={{env:SERVICENOW_OAUTH2_CLIENT_SECRET}}"
            + "&model.incident=org.apache.camel.component.servicenow.model.Incident")
        .to("mock:servicenow");
    }
}); Map<String, Object> headers = new HashMap<>
();headers.put(ServiceNowConstants.RESOURCE, "table");
headers.put(ServiceNowConstants.ACTION, ServiceNowConstants.ACTION_RETRIEVE);
headers.put(ServiceNowConstants.SYSPARM_LIMIT, "10");
headers.put(ServiceNowConstants.TABLE,
"incident");template.sendBodyAndHeaders("direct:servicenow", null, headers);
```

第146章 SERVLET

SERVLET コンポーネント

servlet: コンポーネントは、HTTP エンドポイント に到達した HTTP リクエストを消費するために HTTP ベースのエンドポイントを提供し、このエンドポイントは公開されたサーブレットにバインドされます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-servlet</artifactId>
  <version>x.x.x</version>
  <\!-\- use the same version as your Camel core version \-->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
servlet://relative_path[?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
httpBindingRef	null	レジストリーの **org.apache.camel.component.http.HttpBinding** への参照。**HttpBinding** 実装を使用して、応答の作成方法をカスタマイズできます。

httpBinding	**null**	Camel 2.16: レジストリー の **org.apache.camel.component.http.HttpBinding** への 参照。**HttpBinding** 実装を使用して、応答の作成方法をカスタマイズできます。
matchOnUriPrefix	**false**	完全に一致するものが見つからない場合に、**CamelServlet** が URI 接頭辞と一致することでターゲットコンシューマーの検索を試みるかどうか。
servletName	**CamelServlet**	サーブレットエンドポイントがバインドするサーブレット名を指定します。サーブレット名が指定されていない場合、サーブレットエンドポイントは最初に公開されたサーブレットにバインドされます。
httpMethodRestrict	**null**	Camel 2.11:（コンシューマーのみ） **HttpMethod** が一致する場合にのみ消費できるようにするために使用されます（例：**GET**/**POST**/**PUT** など）。Camel 2.15 以降では、複数のメソッドをコンマで区切って指定できます。

メッセージヘッダー

Apache Camel は、HTTP コンポーネントと同じメッセージヘッダーを適用します。

Apache Camel は すべてのrequest.parameter および request.headers も設定します。たとえば、クライアント要求に URL http://myserver/myserver?orderid=123 がある場合、エクスチェンジには orderid という名前のヘッダー（値が 123）が含まれます。

使用方法

Servlet コンポーネントによって生成されたエンドポイントからのみ消費できます。そのため、Apache Camel ルートへの入力としてのみ使用する必要があります。他の HTTP エンドポイントに対して HTTP 要求を発行するには、HTTP コンポーネントを使用します。

CAMEL JAR のアプリケーションサーバーブートクラスパスへの配置

アプリケーションサーバーのブートクラスパス（通常は lib ディレクトリー内）に camel-core、camel-servlet などの Camel JAR を配置する場合、サーブレットマッピングリストがアプリケーションサーバーの複数のデプロイ済み Camel アプリケーション間で共有されることに注意してください。

通常、アプリケーションサーバーのブートクラスパスに Camel JAR を配置することはベストプラクティスではありません。

したがって、このような状況では、各 Camel アプリケーションでカスタムおよび一意のサーブレット名を定義する 必要があり ます。たとえば、web.xml では以下を定義します。

```
<servlet>
  <servlet-name>MyServlet</servlet-name>
  <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Camel エンドポイントにもサーブレット名が含まれる

```
<route>
  <from uri="servlet://foo?servletName=MyServlet"/>
  ...
</route>
```

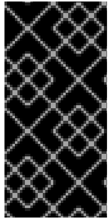
Camel 2.11 以降、Camel はこの重複を検出し、アプリケーションの起動に失敗します。以下のようにサーブレット init-parameter ignoreDuplicateServletName を true に設定すると、この重複を無視するように制御できます。

```
  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <display-name>Camel Http Transport Servlet</display-name>
    <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class>
    <init-param>
      <param-name>ignoreDuplicateServletName</param-name>
```

```
    <param-value>true</param-value>
  </init-param>
</servlet>
```

しかし、この重複の競合や副次 的な副次的な影響を回避するために、各 Camel アプリケーションに一意の servlet-name を使用することを強く推奨します。

例



重要

Camel 2.7 以降、Spring Web アプリケーションで サーブレット を簡単に使用できます。詳細は、サーブレット Tomcat の例 を参照してください。

この例では、http://localhost:8080/camel/services/hello で HTTP サービスを公開するルートを定義します。まず、通常の Web コンテナーまたは OSGi サービスを介して CamelHttpTransportServlet を公開する必要があります。Web.xml ファイルを使用して、以下のように CamelHttpTransportServlet を公開します。

```
<web-app>

  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <display-name>Camel Http Transport Servlet</display-name>
    <servlet-class>
        org.apache.camel.component.servlet.CamelHttpTransportServlet
    </servlet-class>

  </servlet>

  <servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>

</web-app>
```

次に、以下のようにルートを定義できます。

```
from("servlet:///hello?matchOnUriPrefix=true").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        String contentType = exchange.getIn().getHeader(Exchange.CONTENT_TYPE,
String.class);
        String path = exchange.getIn().getHeader(Exchange.HTTP_URI, String.class);
        path = path.substring(path.lastIndexOf("/"));
```

```
        assertEquals("Get a wrong content type", CONTENT_TYPE, contentType);
        // assert camel http header
        String charsetEncoding =
exchange.getIn().getHeader(Exchange.HTTP_CHARACTER_ENCODING, String.class);
        assertEquals("Get a wrong charset name from the message heaer", "UTF-8",
charsetEncoding);
        // assert exchange charset
        assertEquals("Get a wrong charset naem from the exchange property", "UTF-8",
exchange.getProperty(Exchange.CHARSET_NAME));
        exchange.getOut().setHeader(Exchange.CONTENT_TYPE, contentType + "; charset=UTF-
8");
        exchange.getOut().setHeader("PATH", path);
        exchange.getOut().setBody("<b>Hello World</b>");
    }
});
```

CAMEL-SERVLET エンドポイントの相対パスの指定

Http トランスポートをパブリッシュされたサーブレットとバインディングし、サーブレットのアプリケーションコンテキストパスを認識しないため、camel-servlet エンドポイントは相対パスを使用してエンドポイントの URL を指定します。クライアントはサーブレットパブリッシュアドレス ("http://localhost:8080/camel/services")+ RELATIVE_PATH ("/hello") を介して camel-servlet エンドポイントにアクセスできます。

SPRING 3.X を使用する場合の例

スタンドアロン Apache Camel パッケージには、Tomcat Web コンテナーに Servlet コンポーネントをデプロイする方法を示すデモが含まれています。デモンストレーションは examples/camel-example-servlet-tomcat ディレクトリーにあります。Web コンテナーに Servlet コンポーネントをデプロイする場合は、WEB-INF/web.xml ファイルに Spring ContextLoaderListener インスタンスを作成して、Spring アプリケーションコンテキストを明示的に作成する必要があります。

たとえば、camel-config.xml ファイルから Spring 定義(camelContext およびルート定義を含む)を読み込む Spring アプリケーションコンテキストを作成するには、以下のように web.xml ファイルを定義します。

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>My Web Application</display-name>

    <!-- location of spring xml files -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:camel-config.xml</param-value>
```

```
    </context-param>

    <!-- the listener that kick-starts Spring -->
    <listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- Camel servlet -->
    <servlet>
      <servlet-name>CamelServlet</servlet-name>
      <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-
class>
      <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Camel servlet mapping -->
    <servlet-mapping>
      <servlet-name>CamelServlet</servlet-name>
      <url-pattern>/camel/*</url-pattern>
    </servlet-mapping>

</web-app>
```

SPRING 2.X を使用する場合の例

Camel/Spring アプリケーションで Servlet コンポーネントを使用する場合は、Servlet コンポーネントの起動後 に Spring ApplicationContext をロードする必要があります。これは、ContextLoaderListener の代わりに Spring の ContextLoaderServlet を使用して実行できます。その場合、以下のように CamelHttpTransportServlet の後に ContextLoaderServlet を起動する必要があります。

```
<web-app>
  <servlet>
      <servlet-name>CamelServlet</servlet-name>
      <servlet-class>
        org.apache.camel.component.servlet.CamelHttpTransportServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet>
      <servlet-name>SpringApplicationContext</servlet-name>
      <servlet-class>
        org.springframework.web.context.ContextLoaderServlet
      </servlet-class>
      <load-on-startup>2</load-on-startup>
  </servlet>
<web-app>
```

OSGI を使用する場合の例

Camel 2.6.0 以降では、CamelHttpTransportServlet を OSGi サービスとして公開し、以下のような

SpringDM を利用して OSGi サービスとして公開できます。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <bean id="camelServlet"
class="org.apache.camel.component.servlet.CamelHttpTransportServlet"/>

    <!--
        Enlist it in OSGi service registry
        This will cause two things:
        1) As the pax web whiteboard extender is running the CamelServlet will
           be registered with the OSGi HTTP Service
        2) It will trigger the HttpRegistry in other bundles so the servlet is
           made known there too
    -->
    <service ref="camelServlet">
        <interfaces>
            <value>javax.servlet.Servlet</value>
            <value>org.apache.camel.component.http.CamelServlet</value>
        </interfaces>
        <service-properties>
            <entry key="alias" value="/camel/services" />
            <entry key="matchOnUriPrefix" value="true" />
            <entry key="servlet-name" value="CamelServlet"/>
        </service-properties>
    </service>

</blueprint>
```

次に、以下のように Camel ルートでこのサービスを使用します。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:camel="http://camel.apache.org/schema/blueprint">

    <reference id="servletref" interface="org.apache.camel.component.http.CamelServlet">
        <reference-listener bind-method="register" unbind-method="unregister">
            <ref component-id="httpRegistry"/>
        </reference-listener>
    </reference>

    <bean id="httpRegistry"
class="org.apache.camel.component.servlet.DefaultHttpRegistry"/>

    <bean id="servlet" class="org.apache.camel.component.servlet.ServletComponent">
        <property name="httpRegistry" ref="httpRegistry" />
    </bean>

    <bean id="servletProcessor" class="org.apache.camel.itest.osgi.servlet.ServletProcessor"
```

```
/>

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <route>
            <!-- notice how we can use the servlet scheme which is that osgi:reference above -->
            <from uri="servlet:///hello"/>
            <process ref="servletProcessor"/>
        </route>
    </camelContext>

</blueprint>
```

または、Camel 2.6 より前 - Activator を使用して、OSGi プラットフォームで CamelHttpTransportServlet を公開できます。

```
import java.util.Dictionary;
import java.util.Hashtable;

import org.apache.camel.component.servlet.CamelHttpTransportServlet;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.osgi.service.http.HttpContext;
import org.osgi.service.http.HttpService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.osgi.context.BundleContextAware;

public final class ServletActivator implements BundleActivator, BundleContextAware {
    private static final transient Logger LOG =
LoggerFactory.getLogger(ServletActivator.class);
    private static boolean registerService;

    /**
     * HttpService reference.
     */
    private ServiceReference httpServiceRef;

    /**
     * Called when the OSGi framework starts our bundle
     */
    public void start(BundleContext bc) throws Exception {
        registerServlet(bc);
    }

    /**
     * Called when the OSGi framework stops our bundle
     */
    public void stop(BundleContext bc) throws Exception {
        if (httpServiceRef != null) {
            bc.ungetService(httpServiceRef);
            httpServiceRef = null;
        }
    }
```

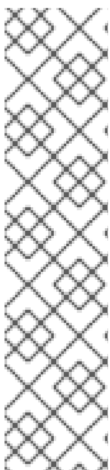
```
    protected void registerServlet(BundleContext bundleContext) throws Exception {
        httpServiceRef = bundleContext.getServiceReference(HttpService.class.getName());

        if (httpServiceRef != null && !registerService) {
            LOG.info("Register the servlet service");
            final HttpService httpService = (HttpService)bundleContext.getService(httpServiceRef);
            if (httpService != null) {
                // create a default context to share between registrations
                final HttpContext httpContext = httpService.createDefaultHttpContext();
                // register the hello world servlet
                final Dictionary<String, String> initParams = new Hashtable<String, String>();
                initParams.put("matchOnUriPrefix", "false");
                initParams.put("servlet-name", "CamelServlet");
                httpService.registerServlet("/camel/services", // alias
                    new CamelHttpTransportServlet(), // register servlet
                    initParams, // init params
                    httpContext // http context
                );
                registerService = true;
            }
        }
    }

    public void setBundleContext(BundleContext bc) {
        try {
            registerServlet(bc);
        } catch (Exception e) {
            LOG.error("Cannot register the servlet, the reason is " + e);
        }
    }

}
```

第147章 SERVLETLISTENER COMPONENT

SERVLETLISTENER COMPONENT

Camel 2.11 から利用可能

このコンポーネントは、Web アプリケーションで Camel アプリケーションのブートストラップに使用されます。たとえば、前者は Camel のブートストラップ方法を見つけるか、Spring などのサードパーティーフレームワークに依存して実行する必要があります。



サイドバー

このコンポーネントは Servlet 2.x 以降をサポートします。つまり、古い Web コンテナーでも動作します。これは、このコンポーネントの目的です。ただし、Servlet 2.x では web.xml ファイルを設定として使用する必要があります。

Servlet 3.x コンテナーでは、@WebListener を使用して Camel を改良し、Camel をブーストする独自のクラスを実装できます。これにより、エンドユーザーが古い school web.xml ファイルで無料で取得できる Camel を簡単に設定する方法が課題となります。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-servletlistener</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

使用

抽象クラス org.apache.camel.component.servletlistener.CamelServletContextListener の以下の実装のいずれかを選択する必要があります。

- *JndiRegistry を使用してレジストリーに JNDI を利用する JndiCamelServletContextListener。*

- *SimpleRegistry を使用して java.util.Map をレジストリーとして使用する SimpleCamelServletContextListener。*

これを使用するには、以下のように WEB-INF/web.xml ファイルで org.apache.camel.component.servletlistener.CamelServletContextListener を設定する必要があります。

```
<web-app>

  <!-- the test parameter is only to be used for unit testing -->
  <!-- you should not use this option for production usage -->
  <context-param>
    <param-name>test</param-name>
    <param-value>true</param-value>
  </context-param>

  <!-- you can configure any of the properties on CamelContext, eg setName will be configured
as below -->
  <context-param>
    <param-name>name</param-name>
    <param-value>MyCamel</param-value>
  </context-param>

  <!-- configure a route builder to use -->
  <!-- Camel will pickup any parameter names that start with routeBuilder (case ignored) -->
  <context-param>
    <param-name>routeBuilder-MyRoute</param-name>
    <param-value>org.apache.camel.component.servletlistener.MyRoute</param-value>
  </context-param>

  <!-- register Camel as a listener so we can bootstrap Camel when the web application starts -
->
  <listener>
    <listener-
class>org.apache.camel.component.servletlistener.SimpleCamelServletContextListener</liste
ner-class>
  </listener>

</web-app>
```

オプション

***org.apache.camel.component.servletlistener.CamelServletContextListener** は、**web.xml** ファイルで **context-param** として設定できる以下のオプションをサポートします。*

オプション	タイプ	説明
propertyPlaceholder.XXX		Camel で プロパティープレースホルダー を設定します。オプションの前に "propertyPlaceholder." を付ける必要があります。たとえば、場所を設定するには、propertyPlaceholder.location を名前として使用します。Properties コンポーネントからすべてのオプションを設定できます。
jmx.XXX		JMX を設定します。JMX を無効にするには、オプションの前に "jmx." を付ける必要があります。たとえば、jmx.disabled を名前として使用します。**org.apache.camel.spi.ManagementAgent** からすべてのオプションを設定できます。JMX ページに記載されているオプションも併せて参照してください。
name	**文字列**	CamelContext の名前を設定するには、以下を行います。
messageHistory	**ブール値**	**Camel 2.12.2:** メッセージ 履歴を有効または無効に するかどうか（デフォルトで有効）。
streamCache	**ブール値**	Stream Caching を有効にするかどうか。
trace	**ブール値**	トレーサー を有効にするかどうか。

delayer	**Long**	Delay Interceptor の遅延値を設定するには、以下を実行します。
handleFault	**ブール値**	障害の処理を有効にするかどうか。
errorHandlerRef	**文字列**	使用するコンテキストスコープの エラーハンドラー を参照します。
autoStartup	**ブール値**	Camel の起動時にすべてのルートを起動するかどうか。
useMDCLogging	**ブール値**	MDC ロギング を使用するかどうか。
useBreadcrumb	**ブール値**	ブレッドクラム を使用するかどうか。
managementNamePattern	**文字列**	JMX MBean のカスタム命名パターンを設定します。
threadNamePattern	**文字列**	スレッドにカスタムの命名パターンを設定します。
properties.XXX		**CamelContext.getProperties** にカスタムプロパティーを設定するには、以下を行います。これはほとんど使用されません。
routebuilder.XXX		使用するルートを設定します。詳細は、こちらを参照してください。
CamelContextLifecycle		**org.apache.camel.component.servletlistener.CamelContextLifecycle** の実装の FQN クラス名を参照します。これにより、CamelContext が起動または停止された前後にカスタムコードを実行できます。詳細は、以下を参照してください。
XXX		CamelContext にオプションを設定するには、以下を行います。

例

Servlet Tomcat No Spring Example を参照してください。

ルートの設定

web.xml ファイルで使用するルートを設定する必要があります。これはさまざまな方法で実行できますが、すべてのパラメーターの前に "routeBuilder" を付ける必要があります。

ROUTEBUILDER クラスの使用

デフォルトでは、Camel は以下のように param-value が Camel RouteBuilder クラスの FQN クラス名であると想定します。

```
<context-param>
  <param-name>routeBuilder-MyRoute</param-name>
  <param-value>org.apache.camel.component.servletlistener.MyRoute</param-value>
</context-param>
```

以下のように、同じ param-value で複数のクラスを指定できます。

```
<context-param>
  <param-name>routeBuilder-routes</param-name>
  <!-- we can define multiple values separated by comma -->
  <param-value>
    org.apache.camel.component.servletlistener.MyRoute,
    org.apache.camel.component.servletlistener.routes.BarRouteBuilder
  </param-value>
</context-param>
```

パラメーターの名前には、ランタイム時に意味がありません。これは一意で、routeBuilder で始まる必要があります。上記の例では、routeBuilder-routes があります。ただし、routeBuilder.foo という名前を付けただけでも構いません。

パッケージスキャンの使用

また、パッケージスキャンを使用するように Camel に指示することもできます。つまり、指定のパッケージで RouteBuilder タイプのすべてのクラスを検索し、Camel ルートとして自動的に追加します。これを行うには、以下のように packagescan: の接頭辞を付ける必要があります。

```
<context-param>
  <param-name>routeBuilder-MyRoute</param-name>
  <!-- define the routes using package scanning by prefixing with packagescan: -->
  <param-value>packagescan:org.apache.camel.component.servletlistener.routes</param-value>
</context-param>
```

XML ファイルの使用

XML DSL を使用して Camel ルートを定義することもできますが、Spring や Blueprint を使用しないため、XML ファイルには Camel ルートのみを含めることができます。web.xml では、以下のように "classpath"、"file"、または "http" url から指定できる XML ファイルを参照します。

```
<context-param>
  <param-name>routeBuilder-MyRoute</param-name>
  <param-value>classpath:routes/myRoutes.xml</param-value>
</context-param>
```

XML ファイルは以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- the xmlns="http://camel.apache.org/schema/spring" is needed -->
<routes xmlns="http://camel.apache.org/schema/spring">

  <route id="foo">
    <from uri="direct:foo"/>
    <to uri="mock:foo"/>
  </route>

  <route id="bar">
    <from uri="direct:bar"/>
    <to uri="mock:bar"/>
  </route>

</routes>
```

XML ファイルでは、root タグは <routes> であり、名前空間 http://camel.apache.org/schema/spring を使用する必要があります。この名前空間は、名前に spring を持ちますが、これは Spring が最初で XML DSL のみであったため、過去の理由が原因です。ランタイム時に Spring JAR は必要ありません。Camel 3.0 では、名前空間の名前を汎用名に変更できます。

適切なプレースホルダーの設定

以下は、クラスパスから myproperties.properties を読み込むプロパティープレースホルダーを設定するための web.xml 設定のスニペットです。

```
<!-- setup property placeholder to load properties from classpath -->
<!-- we do this by setting the param-name with propertyPlaceholder. as prefix and then any options such as location, cache etc -->
<context-param>
  <param-name>propertyPlaceholder.location</param-name>
  <param-value>classpath:myproperties.properties</param-value>
</context-param>
<!-- for example to disable cache on properties component, you do -->
<context-param>
  <param-name>propertyPlaceholder.cache</param-name>
  <param-value>false</param-value>
</context-param>
```

JMX の設定

以下は、JMX の無効化など、JMX を設定するための web.xml 設定のスニペットです。

```
<!-- configure JMX by using names that is prefixed with jmx. -->
<!-- in this example we disable JMX -->
<context-param>
  <param-name>jmx.disabled</param-name>
  <param-value>true</param-value>
</context-param>
```

JNDI または CAMEL レジストリーとしての SIMPLE

このコンポーネントは、JNDI または Simple をレジストリーとして使用します。これにより、JNDI の Beanおよびその他のサービスを検索し、独自の Beanのバインドおよびバインド解除を行うことができます。

これは、org.apache.camel.component.servletlistener.CamelContextLifecycle を実装して Java コードから実行されます。

カスタム CAMELCONTEXTLIFECYCLE の使用

以下のコードでは、beforeStart および afterStop コールバックを使用して Simple レジストリーでカスタム Bean を登録し、停止時にクリーンアップします。

```
/**
 * Our custom {@link CamelContextLifecycle} which allows us to enlist beans in the {@link
JndiContext}
 * so the Camel application can lookup the beans in the {@link
org.apache.camel.spi.Registry}.
 * <p/>
 * We can of course also do other kind of custom logic as well.
 */
public class MyLifecycle implements CamelContextLifecycle<SimpleRegistry> {

    @Override
    public void beforeStart(ServletCamelContext camelContext, SimpleRegistry registry) throws
Exception {
        // enlist our bean(s) in the registry
        registry.put("myBean", new HelloBean());
    }

    @Override
    public void afterStart(ServletCamelContext camelContext, SimpleRegistry registry) throws
Exception {
        // noop
    }

    @Override
    public void beforeStop(ServletCamelContext camelContext, SimpleRegistry registry) throws
Exception {
        // noop
    }

    @Override
    public void afterStop(ServletCamelContext camelContext, SimpleRegistry registry) throws
Exception {
        // unbind our bean when Camel has been stopped
```

```
        registry.remove("myBean");
    }
}
```

その後、パラメーター名 "CamelContextLifecycle" を使用して、このクラスを以下のように web.xml ファイルに登録する必要があります。この値は、org.apache.camel.component.servletlistener.CamelContextLifecycle インターフェイスを実装するクラスを参照する FQN である必要があります。

```
<context-param>
  <param-name>CamelContextLifecycle</param-name>
  <param-value>org.apache.camel.component.servletlistener.MyLifecycle</param-value>
</context-param>
```

名前 myBean を使用して HelloBean Bean を登録すると、以下のように Camel ルートでこの Bean を参照できます。

```
public class MyBeanRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("seda:foo").routeId("foo")
            .to("bean:myBean")
            .to("mock:foo");
    }
}
```

重要： org.apache.camel.component.servletlistener.JndiCamelServletContextListener を使用する場合は、CamelContextLifecycle も JndiRegistry を使用する必要があります。サーブレットが org.apache.camel.component.servletlistener.SimpleCamelServletContextListener の場合と同様に、CamelContextLifecycle は SimpleRegistryを使用する必要があります。

第148章 SHIRO セキュリティー

SHIRO セキュリティーコンポーネント

Camel 2.5 で利用可能

Camel の shiro-security コンポーネントは、Apache Shiro セキュリティープロジェクトをベースとしたセキュリティーに焦点を当てたコンポーネントです。

Apache Shiro は、認証、承認、エンタープライズセッション管理、および暗号化を適切に処理する強力かつ柔軟なオープンソースセキュリティーフレームワークです。Apache Shiro プロジェクトの目的は、利用可能な最も堅牢で包括的なアプリケーションセキュリティーフレームワークを提供しながら、非常に簡単に理解でき、非常に簡単です。

この camel の shiro-security コンポーネントにより、認証および承認サポートを Camel ルートの異なるセグメントに適用できます。

Shiro セキュリティーは、Camel ポリシーを使用してルートに適用されます。Camel の Policy は、Camel プロセッサーにインターセプターを適用するストラテジーパターンを使用します。camel ルートのセクション/セグメントに、セキュリティー、トランザクションなどの相互の懸念（セキュリティー、トランザクションなど）を適用する機能を提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-shiro</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

SHIRO セキュリティーの基本

camel ルートで Shiro セキュリティーを使用するには、セキュリティー設定の詳細（ユーザー、パスワード、ロールなど）を使用して ShiroSecurityPolicy オブジェクトをインスタンス化する必要があります。その後、このオブジェクトは Camel ルートに適用する必要があります。この ShiroSecurityPolicy オブジェクトは、Camel レジストリー(JNDI または ApplicationContextRegistry)に登録してから、Camel Context の他のルートでも使用することができます。

設定の詳細は、Ini ファイル（プロパティーファイル）または Ini オブジェクトを使用して ShiroSecurityPolicy に提供されます。Ini ファイルは、以下のようにユーザー/ロールの詳細を含む標準の Shiro 設定ファイルです。

```
[users]
# user 'ringo' with password 'starr' and the 'sec-level1' role
ringo = starr, sec-level1
george = harrison, sec-level2
john = lennon, sec-level3
paul = mccartney, sec-level3

[roles]
# 'sec-level3' role has all permissions, indicated by the
# wildcard '*'
sec-level3 = *

# The 'sec-level2' role can do anything with access of permission
# readonly (*) to help
sec-level2 = zone1:*

# The 'sec-level1' role can do anything with access of permission
# readonly
sec-level1 = zone1:readonly:*
```

SHIROSECURITYPOLICY オブジェクトのインスタンス化

ShiroSecurityPolicy オブジェクトは以下のようにインスタンス化されます。

```
        private final String iniResourcePath = "classpath:shiro.ini";
        private final byte[] passPhrase = {
            (byte) 0x08, (byte) 0x09, (byte) 0x0A, (byte) 0x0B,
            (byte) 0x0C, (byte) 0x0D, (byte) 0x0E, (byte) 0x0F,
            (byte) 0x10, (byte) 0x11, (byte) 0x12, (byte) 0x13,
            (byte) 0x14, (byte) 0x15, (byte) 0x16, (byte) 0x17};
        List<permission> permissionsList = new ArrayList<permission>();
        Permission permission = new WildcardPermission("zone1:readwrite:*");
        permissionsList.add(permission);

        final ShiroSecurityPolicy securityPolicy =
            new ShiroSecurityPolicy(iniResourcePath, passPhrase, true, permissionsList);
```

SHIROSECURITYPOLICY OPTIONS

名前	デフォルト値	型	説明

iniResourcePath または ini	**none**	リソース文字列または Ini オブジェクト	iniResourcePath または Ini オブジェクトのインスタンスの必須の Resource String をセキュリティーポリシーに渡す必要があります。"file:, classpath:, url:" で始まる場合は、ファイルシステム、クラスパス、または URL からそれぞれリソースを取得できます。例："classpath:shiro.ini"
passPhrase	**AES 128 ベースのキー**	byte[]	メッセージエクスチェンジとともに送信される ShiroSecurityToken (s) を復号化する passPhrase
alwaysReauthenticate	**true**	boolean	を設定して、個々のリクエストで再認証できるようにします。false に設定すると、ユーザーは認証され、今後実行する同じユーザーからの要求のみではなく認証およびロックされます。
permissionsList	**none**	List<Permission>	認証されたユーザーが追加のアクションを実行できるようにするために必要なパーミッションのリスト。つまり、ルートでさらに続行されます。ShiroSecurityPolicy オブジェクトに Permissions リストが提供されていない場合、承認は不要であると判断されます。
cipherService	**AES**	org.apache.shiro.crypto.CipherService	Shiro には AES および Blowfish ベースの CipherServices が同梱されています。これらは 1 つ使用することも、独自の暗号実装で渡すこともできます。

base64	false	boolean	Camel 2.12: セキュリティートークンヘッダーに base64 エンコーディングを使用します。これにより、JMS でのヘッダーの転送が可能になります。このオプションは **ShiroSecurityTokenInjector** でも設定する必要があります。

CAMEL ルートでの SHIRO 認証の適用

ShiroSecurityPolicy は、メッセージヘッダーに暗号化された SecurityToken を含む受信メッセージエクスチェンジをテストおよび許可し、さらに適切な認証を続行します。SecurityToken オブジェクトには、ユーザーが有効なユーザーである場所を決定するために使用される Username/Password の詳細が含まれます。

```
protected RouteBuilder createRouteBuilder() throws Exception {
    final ShiroSecurityPolicy securityPolicy =
        new ShiroSecurityPolicy("classpath:shiro.ini", passPhrase);

    return new RouteBuilder() {
        public void configure() {
            onException(UnknownAccountException.class).
                to("mock:authenticationException");
            onException(IncorrectCredentialsException.class).
                to("mock:authenticationException");
            onException(LockedAccountException.class).
                to("mock:authenticationException");
            onException(AuthenticationException.class).
                to("mock:authenticationException");

            from("direct:secureEndpoint").
                to("log:incoming payload").
                policy(securityPolicy).
                to("mock:success");
        }
    };
}
```

CAMEL ルートでの SHIRO 承認の適用

パーミッションリストを ShiroSecurityPolicy に関連付けることで、承認を Camel ルートに適用できます。Permissions List は、ユーザーがルートセグメントの実行に進むために必要なパーミッションを指定します。ユーザーに適切なパーミッションが設定されていない場合、リクエストはこれ以上続行できません。

```
protected RouteBuilder createRouteBuilder() throws Exception {
    final ShiroSecurityPolicy securityPolicy =
        new ShiroSecurityPolicy("./src/test/resources/securityconfig.ini", passPhrase);

    return new RouteBuilder() {
        public void configure() {
            onException(UnknownAccountException.class).
                to("mock:authenticationException");
            onException(IncorrectCredentialsException.class).
                to("mock:authenticationException");
            onException(LockedAccountException.class).
                to("mock:authenticationException");
            onException(AuthenticationException.class).
                to("mock:authenticationException");

            from("direct:secureEndpoint").
                to("log:incoming payload").
                policy(securityPolicy).
                to("mock:success");
        }
    };
}
```

SHIROSECURITYTOKEN を作成し、メッセージエクスチェンジに注入する

ShiroSecurityToken オブジェクトは、ShiroSecurityTokenInjector と呼ばれる Shiro Processor を使用してメッセージエクスチェンジを作成し、挿入することができます。クライアントで ShiroSecurityTokenInjector を使用して ShiroSecurityToken を注入する例を以下に示します。

```
ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "starr");
ShiroSecurityTokenInjector shiroSecurityTokenInjector =
    new ShiroSecurityTokenInjector(shiroSecurityToken, passPhrase);

from("direct:client").
    process(shiroSecurityTokenInjector).
    to("direct:secureEndpoint");
```

SHIROSECURITYPOLICY でセキュリティーが保護されたルートへのメッセージ送信

セキュリティーポリシーが適用される Camel ルートとともに送信されるメッセージおよびメッセージエクスチェンジは、Exchange ヘッダーの SecurityToken によって付随する必要があります。SecurityToken は、Username と Password を保持する暗号化されたオブジェクトです。SecurityToken はデフォルトで AES 128 ビットセキュリティーを使用して暗号化され、任意の暗号に変更できます。

以下は、Camel の ProducerTemplate と SecurityToken を使用してリクエストを送信する方法の例になります。

_

```
@Test
public void testSuccessfulShiroAuthenticationWithNoAuthorization() throws Exception {
    //Incorrect password
    ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "stirr");

    // TestShiroSecurityTokenInjector extends ShiroSecurityTokenInjector
    TestShiroSecurityTokenInjector shiroSecurityTokenInjector =
        new TestShiroSecurityTokenInjector(shiroSecurityToken, passPhrase);

    successEndpoint.expectedMessageCount(1);
    failureEndpoint.expectedMessageCount(0);

    template.send("direct:secureEndpoint", shiroSecurityTokenInjector);

    successEndpoint.assertIsSatisfied();
    failureEndpoint.assertIsSatisfied();
}
```

SHIROSECURITYPOLICY によってセキュリティーが保護されたルートへのメッセージ送信(CAMEL 2.12 以降により簡単)

Camel 2.12 以降では、サブジェクトを 2 つの方法で提供できるため、さらに簡単になりました。

SHIROSECURITYTOKEN の使用

ユーザー名とパスワードが含まれる org.apache.camel.component.shiro.security.Shiro.security.ShiroSecurityToken タイプのキー ShiroSecurityConstants. SHIRO_SECURITY_TOKEN のヘッダーを持つ Camel ルートにメッセージを送信することができます。以下に例を示します。

```
    ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "starr");

    template.sendBodyAndHeader("direct:secureEndpoint", "Beatle Mania",
ShiroSecurityConstants.SHIRO_SECURITY_TOKEN, shiroSecurityToken);
```

また、以下のように、2 つの異なるヘッダーでユーザー名とパスワードを指定することもできます。

```
    Map<String, Object> headers = new HashMap<String, Object>();
    headers.put(ShiroSecurityConstants.SHIRO_SECURITY_USERNAME, "ringo");
```

```
headers.put(ShiroSecurityConstants.SHIRO_SECURITY_PASSWORD, "starr");
template.sendBodyAndHeaders("direct:secureEndpoint", "Beatle Mania", headers);
```

ユーザー名とパスワードヘッダーを使用すると、Camel ルートの ShiroSecurityPolicy がキー ShiroSecurityConstants.SHIRO_SECURITY_TOKEN を持つ単一のヘッダーに自動的に変換されます。次に、トークンは base64 表現の ShiroSecurityToken インスタンスです（後者は base64=true を設定する場合です）。

第149章 SIP

SIP コンポーネント

Camel 2.5 で利用可能

Camel の sip コンポーネントは Jain SIP 実装をベースとする通信コンポーネントです(JCP ライセンスで利用可能)。

Session Initiation Protocol (SIP)は IETF 定義のシグナリングプロトコルで、音声や IP （インターネットプロトコル）によるビデオコール(IP)などのマルチメディア通信セッションを制御するために広く使用されています。SIP プロトコルは、基盤となるトランスポート層から独立するように設計されたアプリケーションレイヤープロトコルで、Transmission Control Protocol (TCP)、User Datagram Protocol (UDP)、または Stream Control Transmission Protocol (SCTP)で実行できます。

Jain SIP 実装は TCP および UDP のみをサポートしています。

Camel SIP コンポーネントは、RFC3903 - Session Initiation Protocol (SIP) Extension for Eventで説明されているように、SIP Publish および Subscribe 機能 のみ をサポートします。

この Camel コンポーネントは、プロデューサーとコンシューマーエンドポイントの両方をサポートします。

Camel SIP Producers (Event Publishers)および SIP Consumers (Event Subscribers)は、SIP Presence Agent （ステートフルブローカー処理エンティティー）と呼ばれる中間エンティティーを使用してイベントと状態情報を相互に通信します。

SIP ベースの通信では、リスナーのある SIP スタックを SIP プロデューサーとコンシューマーの両方でインスタンス化する 必要 があります(localhost を使用する場合は別のポートを使用します)。これは、通信中に SIP スタック間で交換されるハンドシェイクと確認応答をサポートするために必要です。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sip</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

sip エンドポイントの URI スキームは以下のとおりです。

```
sip://johndoe@localhost:99999[?options]
sips://johndoe@localhost:99999/[?options]
```

このコンポーネントは、TCP と UDP の両方のプロデューサーおよびコンシューマーエンドポイントをサポートします。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

SIP コンポーネントは、SIP プロトコルを介して状態を伝播するために必要なカスタムステートフルヘッダーを作成するための幅広い設定オプションと機能を提供します。

名前	デフォルト値	説明
stackName	**NAME_NOT_SET**	SIP エンドポイントに関連付けられた SIP Stack インスタンスの名前。

transport	**tcp**	Configuring for choice of transport potocol.有効な選択肢は tcp またはudp です。
fromUser		メッセージの送信元のユーザー名。**レジストリーベースのカスタム FromHeader が指定されていない限り、必須設定。**
fromHost		メッセージの送信元のホスト名。**レジストリーベースの FromHeader が指定されていない限りの設定**
fromPort		メッセージの送信元のポート。**レジストリーベースの FromHeader が指定されていない限りの設定**
toUser		メッセージレシーバーのユーザー名。**レジストリーベースのカスタム ToHeader が指定されていない限りの設定。**
toHost		メッセージレシーバーのホスト名。**レジストリーベースの ToHeader が指定されていない限りの設定**
toPort		メッセージレシーバーの portName。**レジストリーベースの ToHeader が指定されていない限りの設定**
maxforwards	0	メッセージをメッセージの受信者に転送することができる中間者の数。**オプションの設定。または、をレジストリーベースの MaxForwardsHeader として使用して設定することもできます。**
eventId		文字列ベースのイベント ID の設定。**レジストリーベースの FromHeader が指定されていない限りの設定**
eventHeaderName		文字列ベースのイベント ID の設定。**レジストリーベースの FromHeader が指定されていない限りの設定**

maxMessageSize	**1048576**	最大許容メッセージサイズ（バイト単位）の設定。
cacheConnections	**false**	接続作成のコストを削減するには、SipStack が接続をキャッシュする必要があります。これは、接続が長時間実行される会話に使用される場合に便利です。
consumer	**false**	この設定は、このエンドポイントに作成する必要があるヘッダーの種類(FromHeader、ToHeader など)を決定するために使用されます。
automaticDialogSupport	**off**	を設定し、すべての通信をダイアログに関連付けるかどうかを指定します。
contentType	**text**	contentType の設定は、有効な MimeType に設定できます。
contentSubType	**xml**	contentSubType の設定は、有効な MimeSubType に設定できます。
receiveTimeoutMillis	**10000**	応答や確認応答を待機する時間を指定するための設定は、別の SIP スタックから受信できます。
useRouterForAllUris	**false**	この設定は、要求がプロキシー経由で Presence Agent に送信される場合に使用されます。
msgExpiration	**3600**	エンドポイントで受信されるメッセージが有効とみなされる時間
presenceAgent	**false**	この設定は、Presence Agent とコンシューマー間の非表示に使用されます。これは、SIP Camel コンポーネントに基本的な Presence Agent（テスト目的のみ）が同梱されるという事実によるものです。コンシューマーはこのフラグを true に設定する必要があります。

レジストリーベースのオプション

SIP では、要求の一部として送信/受信される多数のヘッダーが必要です。これらの SIP ヘッダーは、Spring XML ファイルなどのレジストリーに登録できます。

渡すことができる値は次のとおりです。

名前	説明
fromHeader	メッセージ送信元設定を含むカスタム Header オブジェクト。タイプ javax.sip.header.FromHeader を実装する必要があります。
toHeader	メッセージのレシーバー設定を含むカスタム Header オブジェクト。タイプ javax.sip.header.ToHeader を実装する必要があります。
viaHeaders	javax.sip.header.ViaHeader タイプのカスタム Header オブジェクトのリスト。各 ViaHeader には要求転送のプロキシーアドレスが含まれます。（リクエストがリスナーに到達すると、このヘッダーは各プロキシーによって自動的に更新されます）
contentTypeHeader	メッセージコンテンツの詳細を含むカスタム Header オブジェクト。タイプ javax.sip.header.ContentTypeHeader を実装する必要があります。
callIdHeader	呼び出しの詳細を含むカスタム Header オブジェクト。タイプ javax.sip.header.CallIdHeader を実装する必要があります。
maxForwardsHeader	最大プロキシー転送の詳細を含むカスタム Header オブジェクト。このヘッダーは viaHeaders に上限を設定します。タイプ javax.sip.header.MaxForwardsHeader を実装する必要があります。

eventHeader	イベントの詳細を含むカスタム Header オブジェクト。タイプ javax.sip.header.EventHeader を実装する必要があります。
contactHeader	詳細な連絡先情報が含まれるオプションのカスタム Header オブジェクト（メール、電話番号など）。タイプ javax.sip.header.ContactHeader を実装する必要があります。
expiresHeader	メッセージの有効期限情報が含まれるカスタム Header オブジェクト。タイプ javax.sip.header.ExpiresHeader を実装する必要があります。
extensionHeader	ユーザー/アプリケーション固有の詳細を含むカスタム Header オブジェクト。タイプ javax.sip.header.ExtensionHeader を実装する必要があります。

SIP エンドポイントとの間でメッセージを送信

CAMEL SIP パブリッシャーの作成

以下の例では、SIP Publisher が作成され、SIP イベントパブリケーションをユーザー agent@localhost:5152 に送信します。これは、SIP Publisher と Subscriber 間のブローカーとして機能する SIP Presence Agent のアドレスです。

- ***client という名前の SIP スタックの使用***
- ***evtHdrName というレジストリーベースの eventHeader の使用***
- ***evtId というレジストリーベースの eventId の使用***

- リスナーが user2@localhost:3534 として設定された SIP スタックから

- 公開されるイベントは EVENT_A です。

- REQUEST_METHOD と呼ばれる必須ヘッダーは Request.Publish に設定され、エンドポイントを Event パブリッシャーとして設定します。

```
producerTemplate.sendBodyAndHeader(
    "sip://agent@localhost:5152?
stackName=client&eventHeaderName=evtHdrName&eventId=evtid&fromUser=user2&fromHos
t=localhost&fromPort=3534",
    "EVENT_A",
    "REQUEST_METHOD",
    Request.PUBLISH);
```

CAMEL SIP SUBSCRIBER の作成

以下の例では、ユーザー johndoe@localhost:5154 に送信された SIP イベントパブリケーションを受信するために SIP Subscriber が作成されます。

- Subscriber という名前の SIP スタックの使用

- agent@localhost:5152 という Presence Agent ユーザーへの登録

- evtHdrName というレジストリーベースの eventHeader の使用。evtHdrName には、Event_A に属するイベントが含まれます。

- evtId というレジストリーベースの eventId の使用

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
```

```
            // Create PresenceAgent
            from("sip://agent@localhost:5152?
stackName=PresenceAgent&presenceAgent=true&eventHeaderName=evtHdrName&eventId=e
vtid")
                .to("mock:neverland");

            // Create Sip Consumer(Event Subscriber)
            from("sip://johndoe@localhost:5154?
stackName=Subscriber&toUser=agent&toHost=localhost&toPort=5152&eventHeaderName=ev
tHdrName&eventId=evtid")
                .to("log:ReceivedEvent?level=DEBUG")
                .to("mock:notification");

        }
    };
}
```

Camel SIP コンポーネントは、テストおよびデモ目的にのみ使用される Presence Agent も含まれます。優先順位エージェントをインスタンス化する例は、上記を参照してください。

Presence Agent はユーザー agent@localhost:5152 として設定されており、Publisher と Subscriber の両方と通信できることに注意してください。Publisher や Subscriber とは異なる個別の SIP stackName があります。Camel コンシューマーとして設定されますが、実際にはルートとメッセージをエンドポイント "mock:neverland" に送信しません。

第150章 SJMS

SJMS コンポーネント

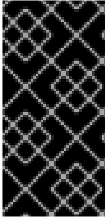
Camel 2.11 から利用可能

Simple JMS Component (SJMS)は、JMS クライアントの作成と設定に関してよく知られているベストプラクティスを使用する Camel で使用するための JMS クライアントです。SJMS には、Camel 向けに明示的に作成された新しい JMS クライアント API が含まれており、軽量で回復力のあるサードパーティーのメッセージング実装を排除します。これに含まれる機能を以下に示します。

- *標準キューとトピックサポート(Durable & Non-Durable)*
- *InOnly & InOut MEPのサポート*
- *非同期プロデューサーおよびコンシューマー処理*
- *内部 JMS トランザクションサポート*

その他の主な機能は以下のとおりです。

- *プラグイン可能な接続リソース管理*
- *セッション、コンシューマー、プロデューサープーリングおよびキャッシング管理*
- *バッチコンシューマーおよびプロデューサー*
- *トランザクションバッチコンシューマーおよびプロデューサー*
- *カスタマイズ可能なトランザクションのコミットストラテジーのサポート（ローカル JMS トランザクションのみ）*



SJMS の S 理由

s は Simple および Standard および Springless の略です。また、camel-jms はすでに取得されています。 : -)

警告

これは、複雑な JMS メッセージングにおける非常に新しいコンポーネントです。そのため、このコンポーネントは継続的に開発および強化されています。Spring JMS をベースとする従来の JMS コンポーネントは、強化され、広範囲にテストされました。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-sjms</artifactId>
<version>x.x.x</version>
<!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
sjms:[queue:|topic:]destinationName[?options]
```

destinationName は JMS キューまたはトピック名です。デフォルトでは、destinationName はキュー名として解釈されます。たとえば、キューに接続するには、FOO.BAR を次のように使用します。

```
sjms:FOO.BAR
```

必要に応じて、オプションの queue: 接頭辞を含めることができます。

```
sjms:queue:FOO.BAR
```

トピックに接続するには、topic: 接頭辞を含める 必要があり ます。たとえば、Stocks.Prices トピックに接続するには、以下を使用します。

```
sjms:topic:Stocks.Prices
```

?option=value&option=value&.. という形式を使用して、URI にクエリーオプションを追加します。

コンポーネントのオプションおよび設定

SJMS コンポーネントは以下の設定オプションをサポートします。

オプション	必須	デフォルト値	説明
connectionCount		**1**	このコンポーネントで起動したエンドポイントが使用できる接続の最大数
connectionFactory	(/)	**null**	SjmsComponent を有効にするには ConnectionFactory が必要です。これは直接設定するか、ConnectionResource の一部として設定できます。
connectionResource		**null**	ConnectionResource は、ConnectionFactory のカスタマイズおよびコンテナー制御を可能にするインターフェイスです。詳細は、プラグイン可能な接続リソース管理を参照してください。
headerFilterStrategy		**DefaultJmsKeyFormatStrategy**	

jmsKeyFormatStrategy		**DefaultJmsKeyFormatStrategy**	Camel 2.16: JMS キーをエンコードおよびデコードするための Pluggable ストラテジー。JMS 仕様に準拠できるようにします。Camel は、追加設定なしで **default** と **passthrough** の 2 つの実装を提供します。**default** ストラテジーは、ドットとハイフン(**.** および **-**)を安全にマーシャリングします。**passthrough** ストラテジーはキーをそのまま残します。JMS ヘッダーキーに不正な文字が含まれているかどうかは問題にならない JMS ブローカーに使用できます。**org.apache.camel.component.jms.JmsKeyFormatStrategy** の独自の実装を提供し、**#** 表記を使用して参照できます。
transactionCommitStrategy		**null**	
DestinationCreationStrategy		**DefaultDestinationCreationStrategy**	Camel 2.15.0: SJMS コンポーネントでカスタム **DestinationCreationStrategy** を設定できるようにします。
messageCreatedStrategy			Camel 2.16: Camel が JMS メッセージを送信するときに Camel が **javax.jms.Message** オブジェクトの新規インスタンスを作成する際に呼び出される指定の **MessageCreatedStrategy** を使用します。

以下は、必要な ConnectionFactory プロバイダーを使用して SjmsComponent を設定する方法の例になります。デフォルトで単一の接続を作成し、component 内部プーリング API を使用してこれを保存し、スレッドセーフな方法でセッション作成要求に対応できるようにします。

```
SjmsComponent component = new SjmsComponent();
component.setConnectionFactory(new ActiveMQConnectionFactory("tcp://localhost:61616"));
getContext().addComponent("sjms", component);
```

永続サブスクリプションのサポートに必要な SjmsComponent の場合、デフォルトの ConnectionFactoryResource インスタンスを上書きし、clientId プロパティーを設定します。

```
ConnectionFactoryResource connectionResource = new ConnectionFactoryResource();
connectionResource.setConnectionFactory(new
ActiveMQConnectionFactory("tcp://localhost:61616"));
connectionResource.setClientId("myclient-id");

SjmsComponent component = new SjmsComponent();
component.setConnectionResource(connectionResource);
component.setMaxConnections(1);
```

プロデューサー設定オプション

SjmsProducer エンドポイントは以下のプロパティーをサポートします。

オプション	デフォルト値	説明
acknowledgementMode	**AUTO_ACKNOWLEDGE**	**SESSION_TRANSACTED**、**AUTO_ACKNOWLEDGE** または **DUPS_OK_ACKNOWLEDGE** のいずれかである JMS の確認名。現時点では、**CLIENT_ACKNOWLEDGE** はサポートされません。
consumerCount	**1**	**InOut のみ。**応答コンシューマー用の MessageListener インスタンスの数を定義します。
exchangePattern	**InOnly**	Producers メッセージ交換パターンを設定します。
namedReplyTo	**null**	**InOut のみ。**応答の宛先への名前付き応答を指定します。
永続	**true**	メッセージを永続化を有効にして配信するかどうか。

producerCount	**1**	MessageProducer インスタンスの数を定義します。
responseTimeOut	**5000**	**InOut のみ。**InOut Producer が応答を待つ時間を指定します。
同期	**true**	Endpoint が同期または非同期処理を使用するかどうかを設定します。
transacted	**false**	エンドポイントが JMS セッショントランザクションを使用する場合。
ttl	**-1**	デフォルトでは無効にされています。Message time を live ヘッダーに設定します。
prefillPool	**true**	**Camel 2.14:** 起動時にプロデューサー接続プールを事前に入力するか、または必要に応じてコネクションの遅延を作成するかどうか。
allowNullBody	**true**	**Camel 2.15.1:** ボディーのないメッセージの送信を許可するかどうか。**false** とメッセージボディーが **null** の場合は、**JMSException** が出力されます。
mapJmsMessage	**true**	**Camel 2.16:** Camel が受信した JMS メッセージを **javax.jms.TextMessage** や **String** などの適切なペイロードタイプに自動的にマップするかどうかを指定します。
messageCreatedStrategy		**Camel 2.16:** Camel が JMS メッセージを送信するときに Camel が **javax.jms.Message** オブジェクトの新規インスタンスを作成する際に呼び出される指定の **MessageCreatedStrategy** を使用します。

jmsKeyFormatStrategy		**Camel 2.16:** JMS キーをエンコードおよびデコードするための Pluggable ストラテジー。JMS 仕様に準拠できるようにします。Camel は、追加設定なしで **default** と **passthrough** の 2 つの実装を提供します。**default** ストラテジーは、ドットとハイフン (**.** および **-**)を安全にマーシャリングします。**passthrough** ストラテジーはキーをそのまま残します。JMS ヘッダーキーに不正な文字が含まれているかどうかは問題にならない JMS ブローカーに使用できます。**org.apache.camel.component.jms.JmsKeyFormatStrategy** の独自の実装を提供し、**#** 表記を使用して参照できます。
includeAllJMSXProperties		**Camel 2.16:** JMS から Camel Message にマッピングするときに、すべての **JMSX*xxx*** プロパティーを含めるかどうか。これを **true** に設定すると、**JMSXAppID** や **JMSXUserID** などのプロパティーが含まれます。注記：カスタムの **headerFilterStrategy** を使用している場合、このオプションは適用されません。

プロデューサーの使用

INONLY PRODUCER -（デフォルト）

InOnly Producer は SJMS Producer Endpoint のデフォルトの動作です。

```
from("direct:start")
.to("sjms:queue:bar");
```

INOUT プロデューサー

InOut 動作を有効にするには、URI に exchangePattern 属性を追加します。デフォルトでは、コンシューマーごとに専用の TemporaryQueue を使用します。

```
from("direct:start")
.to("sjms:queue:bar?exchangePattern=InOut");
```

namedReplyTo を指定できますが、これによりモニターポイントが改善されます。

```
from("direct:start")
.to("sjms:queue:bar?exchangePattern=InOut&namedReplyTo=my.reply.to.queue");
```

コンシューマー設定オプション

SjmsConsumer エンドポイントは以下のプロパティーをサポートします。

オプション	デフォルト値	説明
acknowledgementMode	**AUTO_ACKNOWLEDGE**	**TRANSACTED**、**AUTO_ACKNOWLEDGE** または **DUPS_OK_ACKNOWLEDGE** のいずれかである JMS の確認名。現時点では、**CLIENT_ACKNOWLEDGE** はサポートされません。
consumerCount	**1**	MessageListener インスタンスの数を定義します。
durableSubscriptionId	**null**	永続サブスクリプションに必要です。
exchangePattern	**InOnly**	Consumers メッセージ交換パターンを設定します。
messageSelector	**null**	メッセージセレクターを設定します。

同期	true	Endpoint が同期または非同期処理を使用するかどうかを設定します。
transacted	false	エンドポイントが JMS セッショントランザクションを使用する場合。
transactionBatchCount	1	ローカル JMS トランザクションをコミットする前に処理するエクスチェンジの数。**トランザクション** プロパティーも true に設定する必要があります。そうしないと、このプロパティーは無視されます。
transactionBatchTimeout	5000	消費済み内容をコミットする前にトランザクションがメッセージ間で開いたままになる時間。最小値は 1000ms です。
ttl	\-1	デフォルトでは無効にされています。Message time を live ヘッダーに設定します。

コンシューマーの使用

INONLY CONSUMER: （デフォルト）

InOnly Consumer は、SJMS コンシューマーエンドポイントのデフォルトのエクスチェンジ動作です。

```
from("sjms:queue:bar")
.to("mock:result");
```

INOUT コンシューマー

InOut 動作を有効にするには、URI に exchangePattern 属性を追加します。

```
from("sjms:queue:in.out.test?exchangePattern=InOut")
.transform(constant("Bye Camel"));
```

高度な使用方法に関する注意点

プラグイン可能な接続リソース管理

SJMS は、ビルトイン 接続 プールを介して JMS 接続リソース管理を提供します。これにより、サードパーティーの API プーリングロジックに依存する必要がなくなります。ただし、J2EE や OSGi コンテナーなど、外部 Connection リソースマネージャーの使用が必要になる場合があります。この SJMS では、内部 SJMS 接続プール機能を上書きするために使用できるインターフェイスを提供します。これは ConnectionResource インターフェイスを介して実行されます。

ConnectionResource は、Connection pool を SJMS コンポーネントに提供するために使用されるコントラクトとして、必要に応じて接続を借用して返すメソッドを提供します。ユーザーは、SJMS を外部接続プールマネージャーと統合する必要がある場合に使用する必要があります。

標準の ConnectionFactory プロバイダーの場合は、このコンポーネントに対して最適化されているように SJMS で提供される ConnectionFactoryResource 実装を使用するか、または拡張することが推奨されます。

以下は、ActiveMQ PooledConnectionFactory でプラグ可能な ConnectionResource を使用する例です。

```
public class AMQConnectionResource implements ConnectionResource {
private PooledConnectionFactory pcf;

public AMQConnectionResource(String connectString, int maxConnections) {
super();
pcf = new PooledConnectionFactory(connectString);
pcf.setMaxConnections(maxConnections);
pcf.start();
}

public void stop() {
pcf.stop();
}

@Override
public Connection borrowConnection() throws Exception {
Connection answer = pcf.createConnection();
answer.start();
return answer;
}

@Override
```

```
public Connection borrowConnection(long timeout) throws Exception {
// SNIPPED...
}

@Override
public void returnConnection(Connection connection) throws Exception {
// Do nothing since there isn't a way to return a Connection
// to the instance of PooledConnectionFactory
log.info("Connection returned");
}
}
```

次に、ConnectionResource を SjmsComponent に渡します。

```
CamelContext camelContext = new DefaultCamelContext();
AMQConnectionResource pool = new AMQConnectionResource("tcp://localhost:33333", 1);
SjmsComponent component = new SjmsComponent();
component.setConnectionResource(pool);
camelContext.addComponent("sjms", component);
```

その使用例をすべて確認するには、ConnectionResourceIT を参照してください。

セッション、コンシューマー、プロデューサープーリングおよびキャッシング管理

近日中にお待ちください ...

バッチメッセージのサポート

SjmsProducer は、List をカプセル化するエクスチェンジを作成して、メッセージのコレクションを公開します。この SjmsProducer は List の内容を繰り返し処理し、各メッセージを個別に公開します。

メッセージのバッチを生成する場合、各メッセージに固有のヘッダーを設定する必要があります。SJMS BatchMessage クラスを使用できます。SjmsProducer が BatchMessage List に遭遇すると、各 BatchMessage を繰り返し処理し、含まれるペイロードとヘッダーを公開します。

以下は BatchMessage クラスの使用例です。まず、BatchMessages の一覧を作成します。

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
String body = "Hello World " + i;
BatchMessage<String> message = new BatchMessage<String>(body, null);
messages.add(message);
}
```

次に、一覧を公開します。

```
template.sendBody("sjms:queue:batch.queue", messages);
```

カスタマイズ可能なトランザクションのコミットストラテジー（ローカル JMS トランザクションのみ）

SJMS は、開発者に TransactionCommitStrategy インターフェイスを使用してカスタムおよびプラグイン可能なトランザクションストラテジーを作成する手段を提供します。これにより、セッションをコミットするタイミングを決定するために SessionTransactionSynchronization が使用する一意の一連の状況を定義できます。その使用例は BatchTransactionCommitStrategy です。これについては次のセクションで説明します。

トランザクションバッチコンシューマーおよびプロデューサー

SjmsComponent は、Producer エンドポイントと Consumer エンドポイントの両方でローカル JMS トランザクションのバッチ処理をサポートするように設計されています。ただし、それぞれの処理方法は非常に異なります。

SjmsConsumer エンドポイントは、関連付けられた Session でコミットする前に X メッセージを処理する straitforward 実装です。コンシューマーでバッチ処理されたトランザクションを有効にするには、最初に transacted パラメーターを true に設定し、transactionBatchCount を追加し、これを 0 より大きい値に設定します。たとえば、以下の設定は 10 メッセージごとに Session をコミットします。

```
sjms:queue:transacted.batch.consumer?transacted=true&transactionBatchCount=10
```

コンシューマーエンドポイントでバッチの処理中に例外が発生すると、Session ロールバックが呼び出され、メッセージが次に利用可能なコンシューマーに再配信されます。カウンターは、関連付けられた Session の BatchTransactionCommitStrategy に対しても 0 にリセットされます。ユーザーがフックをバッチメッセージのプロセッサーに配置し、JMSRedelivered ヘッダーが true に設定されたメッセージを監視するのはユーザーの責任です。これは、メッセージが特定の時点でロールバックされ、成功した処理の検証が発生することを示しています。

トランザクションバッチコンシューマーは、セッションで開かれたトランザクションをコミットする

前に、メッセージ間のデフォルトの時間(5000ms)を待機する内部タイマーのインスタンスも伝送します。デフォルト値の 5000ms （最低 1000ms）はほとんどのユースケースに適していますが、さらなるチューニングが必要な場合は transactionBatchTimeout パラメーターを設定します。

```
sjms:queue:transacted.batch.consumer?
transacted=true&transactionBatchCount=10&transactionBatchTimeout=2000
```

許可される最小値は 1000ms です。コンテキストの切り替え量は、利点を得られることなく不要なパフォーマンスへの影響を引き起こす可能性があるためです。

ただし、プロデューサーエンドポイントははるかに異なる方法で処理されます。各メッセージが宛先に配信された後、プロデューサーでは Exchange が閉じられ、そのメッセージへの参照はなくなります。すべてのメッセージを再配信で利用可能にするには、BatchMessages を公開する Producer Endpoint でトランザクションを有効にするだけです。トランザクションは、バッチリストのすべてのメッセージが含まれるエクスチェンジの最後にコミットします。追加設定は必要ありません。以下に例を示します。

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
String body = "Hello World " + i;
BatchMessage<String> message = new BatchMessage<String>(body, null);
messages.add(message);
}
```

トランザクションを有効にして List を公開します。

```
template.sendBody("sjms:queue:batch.queue?transacted=true", messages);
```

追記

メッセージヘッダーの形式

SJMS コンポーネントは、Camel JMS Component で使用されるのと同じヘッダーフォーマットストラテジーを使用します。このプラグイン可能なストラテジーにより、ネットワーク経由で送信されるメッセージが JMS Message 仕様に準拠するようになります。

exchange.in.header の場合、ヘッダーキーに以下のルールが適用されます。

JMS または JMSX で始まるキーは予約されます。exchange.in.headers キーはリテラルで、すべて有効な Java 識別子である必要があります（キー名のドットは使用しないでください）。Camel は、JMS メッセージを使用するときにドットとハイフンを後継に置き換えます。

- は、Camel がメッセージを消費するときに DOT と逆の置換に置き換えられます。
- は、Camel がメッセージを消費するときに HYPHEN に置き換えられ、逆の置換です。鍵のフォーマットに独自のカスタムストラテジーを使用できるオプション jmsKeyFormatStrategy も参照してください。

exchange.in.header の場合、以下のルールがヘッダー値に適用されます。

メッセージの内容

ネットワーク経由でコンテンツを提供するには、配信されるメッセージのボディーが JMS Message Specification に準拠することを確認する必要があります。そのため、生成されるすべてはプリミティブまたはカウンターオブジェクト(Integer、Long、Charracter など)である必要があります。型、String、CharSequence、Date、BigDecimal、BigInteger はすべて、その toString （）表現に変換されます。その他のタイプはすべて破棄されます。

クラスタリング

クラスター環境で SJMS で InOut を使用する場合は、TemporaryQueue 宛先を使用するか、InOut プロデューサーエンドポイントごとに宛先への一意の名前付き応答を使用する必要があります。メッセージ相関は、ブローカーのメッセージセレクターではなく、エンドポイントによって処理されます。InOut Producer Endpoint は、Message JMSCorrelationID によってキャッシュされた Java Concurrency Exchangers を使用します。これにより、対象のコンシューマーによって生成される順序ですべてのメッセージが宛先から消費されるため、ブローカーのオーバーヘッドを削減しつつ優れたパフォーマンスが向上します。

現在、相関ストラテジーは JMSCorrelationId を使用することです。InOut Consumer はこのストラテジーを使用します。また、含まれる JMSReplyTo 宛先へのすべての応答メッセージにも、リクエストから JMSCorrelationId がコピーされるようにします。

トランザクションサポート

現在、SJMS は内部 JMS トランザクションの使用のみをサポートします。Camel Transaction Processor または Java Transaction API (JTA)はサポートされません。

SPRINGLESS MEAN I CAN'T USE SPRING?

まったくありません。以下は、Spring DSL を使用した SJMS コンポーネントの例になります。

```
<route id="inout.named.reply.to.producer.route">
  <from uri="direct:invoke.named.reply.to.queue" />
  <to uri="sjms:queue:named.reply.to.queue?
namedReplyTo=my.response.queue&xchangePattern=InOut" />
</route>
```

Springless は、Spring JMS API の依存関係から離れることを指します。新しい JMS クライアント API は、SJMS をゼロから開発しています。

第151章 SJMS BATCH

SJMS バッチコンポーネント

Camel 2.16 以降で利用可能

SJMS Batch は、JMS キューからの非常に高性能なトランザクションバッチ消費のための特殊なコンポーネントです。これは、コンシューマーのみのコンポーネントとアグリゲーターのハイブリッドと考えることができます。

Camel の一般的なユースケースは、集約された状態を別のエンドポイントに送信する前にキューからメッセージを消費し、それらを集約することです。処理を実行しているシステムが失敗した場合にデータが失われるのを防ぐために、通常はキューからのトランザクション内で消費され、JDBC コンポーネントにあるものなど、永続 AggregationRepository に保存されたデータを集約します。

Aggregator パターンの動作では、受信メッセージが集約される前に AggregationRepository からデータを取得し、後で結果を書き込みます。性質的に、集計されたアーティファクトの数が増えると、読み取りおよび書き込みは徐々に長くなります。これの影響を示す任意の時間単位を使用したこの例は次のとおりです。

項目	読み取り時間	書き込み時間	合計時間
0	0	1	1
1	1	2	4
2	2	3	9
3	3	4	16
4	4	5	25
5	5	6	36
6	6	7	49
7	7	8	64
8	8	9	81
9	9	10	100

一方、SJMS Batch コンポーネントを使用した消費パフォーマンスは linear です。各メッセージは消費され、次のメッセージがフェッチされる前に AggregationStrategy を使用して集約されます。すべての消費と集約が単一の JMS トランザクションで実行されるため、中間状態を永続化するために外部ストレージは必要ありません。これにより、上記の読み取りおよび書き込みコストを回避します。実際には、これにより複数の順番でより高いスループットが得られます。

最初のメッセージ以降のサイズまたは期間のいずれかで完了条件が満たされると、集約された Exchange がルートに渡されます。この Exchange の処理中に例外が出力された場合、またはシステムがシャットダウンすると、元のメッセージがすべてキューに戻されます（または、ブローカーの設定に応じてデッドレターキューに配置されます）。

通常のアグリゲーターとは異なり、集約条件には機能がありません。つまり、複数のメッセージのグループに対して同時に消費をバッチ処理することはできません。消費されたメッセージはすべて 1 つのバッチに集約されます。これを回避するには、一般的な設計アプローチとして、最初にメッセージの異なるグループを異なるキューに集約し、バッチを 1 つずつ個別にバッチ処理する方法が挙げられます。

複数の JMS コンシューマーサポートを利用できます。これにより、1 つのルートを使用して並行して消費でき、同時に JMS メッセージグループなどの機能を使用して関連メッセージをグループ化できます。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sjms</artifactId>
  <version>2.17.0.redhat-630xxx</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

SJMS Batch は SJMS のサブコンポーネントで、同じライブラリーにあります。

URI 形式

```
sjms:[queue:]destinationName[?options]
```

destinationName は JMS キューです。デフォルトでは、destinationName はキュー名として解釈されます。

```
sjms:FOO.BAR
```

必要に応じて、**queue:** 接頭辞を含めることができます。

```
sjms:queue:FOO.BAR
```

このコンテキスト内でバッチ消費を使用する利点がないため、トピックの使用はサポートされていません。トピックメッセージは通常永続的ではなく、損失は許容されます。障害が発生したトランザクション内で消費されると、トピックメッセージはブローカーによって再配信されない可能性があります。このシナリオでは、プレーン SJMS コンシューマーエンドポイントは、通常の非永続性でサポートされるアグリゲーターと組み合わせて使用できます。

コンポーネントのオプションおよび設定

SJMS Batch コンポーネントは以下の設定オプションをサポートします。

オプション	必須	デフォルト値	説明
aggregationStrategy	Yes	**null**	Camel レジストリーでの **AggregationStrategy** への参照（例：**#myAggregationStrategy**）
completionSize		**200**	集約するバッチのサイズ。 これが JMS コンシューマーの事前フェッチバッファー、またはブローカーのキューの最大ページサイズよりも大きくならないように注意する必要があります。タイムアウトを使用しない場合は、コンシューマーがハングする可能性があります。 0 または less の値は、**completionTimeout** のみを使用する必要があることを示します。

completionTimeout		**500**	エクスチェンジを生成する前に最初のメッセージの受信から待機する最大時間。 0 または less の値は、**completionSize** のみを使用する必要があることを示します。 完了タイムアウトと完了間隔を同時に使用することはできません。設定できるのは 1 つのみです。
completionTimeout			**Camel 2.17:** 完了間隔（ミリ秒単位）。これにより、バッチが間隔ごとにスケジュールされる固定レートで完了されます。タイムアウトがトリガーされ、バッチにメッセージがない場合、バッチは空になる可能性があります。完了タイムアウトと完了間隔を同時に使用することはできません。設定できるのは 1 つのみです。
sendEmptyMessageWhenIdle		**false**	**Camel 2.17:** 完了タイムアウトまたは間隔を使用する場合、タイムアウトがトリガーされ、バッチにメッセージがない場合にバッチが空になることがあります。このオプションが true であり、バッチが空の場合、空のメッセージがバッチに追加され、空のメッセージがルーティングされます。
pollDuration		**1000**	**MessageConsumer.receive()** への呼び出しの最大長。タイムアウトの残りはバッチ内で優先されます。 この値は、実質的にバッチ間のポーリング時間です。

timeoutCheckerExecutorService			**Camel 2.17:** completionInterval オプションを使用すると、バックグラウンドスレッドが作成され、完了間隔がトリガーされます。すべてのコンシューマーに新しいスレッドを作成するのではなく、カスタムスレッドプールを提供するには、このオプションを設定します。

completionSize endpoint 属性は completionTimeout とともに使用され、最初に満たされる条件により、集約された Exchange がルートに出力されます。

第152章 SLACK

SLACK コンポーネント

Camel 2.16 以降で利用可能

Slack コンポーネントを使用すると、Slack のインスタンスに接続し、事前確立済みの Slack 受信 Webhook を介してメッセージボディーに含まれるメッセージを配信できます。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-slack</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

メッセージをチャネルに送信します。

```
slack:#channel[?options]
```

slackuser への直接メッセージを送信するには、次のコマンドを実行します。

```
slack:@username[?options]
```

オプション

名前	説明	例
username	これは、チャネルまたはユーザーにメッセージを送信するときにボットが必要とするユーザー名です。	username=CamelUser

iconUrl	メッセージをチャネルまたはユーザーに送信する際にコンポーネントが使用するアバター。	iconUrl=http://somehost.com/avatar.gif
iconEmoji	Slack 絵文字をアバターとして使用する	iconEmoji=:camel:

SLACKCOMPONENT

XML を含む SlackComponent は、パラメーターとしてインテグレーションの受信 Webhook URL が含まれる Spring または Blueprint Bean として設定する必要があります。

```
<bean id="slack" class="org.apache.camel.component.slack.SlackComponent">
    <property name="webhookUrl"
value="https://hooks.slack.com/services/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"/>
</bean>
```

Java の場合は、Java コードを使用して設定できます。

例

Blueprint を使用する CamelContext は、以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" default-activation="lazy">

    <bean id="slack" class="org.apache.camel.component.slack.SlackComponent">
        <property name="webhookUrl"
value="https://hooks.slack.com/services/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"/>
    </bean>

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <route>
            <from uri="direct:test"/>
            <to uri="slack:#channel?iconEmoji=:camel:&amp;username=CamelTest"/>
        </route>
    </camelContext>

</blueprint>
```

第153章 SMPP

SMPP コンポーネント

このコンポーネントは、SMPP プロトコルを使用して SMSC (Short Message Service Center)にアクセスし、SMMS を送受信します。JSMPP ライブラリーは、プロトコルの実装に使用されます。

現在、Camel コンポーネントは ESME（外部ショートカットメッセージングエンティティー）として動作し、SMSC 自体としては動作しません。

Camel 2.9 以降では、ReplaceSm、QuerySm、SubmitMulti、および CancelSm を実行することもできます。DataSm

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-smpp</artifactId>
  <version>2.17.0.redhat-630xxx</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

SMS の制限

SMS は信頼性も安全ではありません。信頼できる安全な配信を必要とするユーザーは、選択したプロトコルをサポートするスマートフォンアプリと組み合わせて、XMPP または SIP コンポーネントの使用を検討してください。

- 信頼性： SMPP 標準はエラーを示すさまざまなフィードバックメカニズムを提供していますが、再配信や配信の確認は、モバイルネットワークがこれらの応答を非表示またはシミュレートする一般的ではありません。たとえば、一部のネットワークは、宛先番号が無効であるか、切り替えされていない場合でも、すべてのメッセージの配信確認を自動的に送信します。一部のネットワークは、スパムだと判断すると、メッセージを警告なしでドロップします。ネットワークでのスパム検出ルールは非常に困難である可能性があり、1 つの送信者の 1 日あたり 100 個以上のメッセージがスパムとみなされる場合があります。

- セキュリティー： ラジオ Tower から受信者ハンドセットまで最後のホップに基本的な暗号化があります。SMS メッセージは、ネットワークの他の部分で暗号化されず、認証されません。一部のオペレーターでは、スタッフのアウトレットや電話センターによる SMS メッセージ履歴の閲覧を許可するものもあります。メッセージ送信者のアイデンティティーは簡単に偽

装できます。レギュレーターやモバイル電話業界でも、二要素認証スキームやセキュリティーが重要なその他の目的で SMS の使用には注意が必要です。

Camel コンポーネントは SMS ネットワークにメッセージを送信できる限り簡単ですが、これらの問題に対する簡単な解決策を提供することはできません。

データコーディング、アルファベット、および国際文字セット

データコーディングとアルファベットは、メッセージごとに指定できます。エンドポイントにはデフォルト値を指定できます。これらのオプションの関係と、複数の値が設定されている場合にコンポーネントが動作する方法を理解することが重要です。

データコーディングは、SMPP 有線形式の 8 ビットフィールドです。alphabet は、データコーディングフィールドのビット 0-3 に対応します。メッセージクラスが使用される一部のタイプのメッセージでは、（データコーディングフィールドのビット 5 を設定して）、データコーディングフィールドの 2 ビット未満がアルファベットとして解釈されず、ビット 2 と 3 のみがアルファベットに影響します。

さらに、現在のバージョンの JSMPP ライブラリーは、ビット 2 および 3 のみをサポートするようです。ビット 0 と 1 がメッセージクラスに使用されると仮定しています。このため、JSMPP の Alphabet クラスは ISO-8859-1 を示す値 3 (binary 0011)をサポートしません。

JSMPP はメッセージクラスパラメーターの表現を提供しますが、現在、Camel コンポーネントはデータコーディングフィールドに対応するビットを手動で設定する方法以外の方法を提供しません。

送信メッセージにデータコーディングフィールドを設定する場合、Camel コンポーネントは以下の値を考慮し、最初に見つけられるものを使用します。

- ヘッダーに指定されたデータコーディング
- ヘッダーに指定されたアルファベット
- エンドポイント設定(URI パラメーター)に指定されたデータコーディング。

SMSC にデータコーディング値を送信する他に、Camel コンポーネントはメッセージボディーの分析を試み、それを Java String (Unicode)に変換し、それを対応するアルファベットのバイトアレイに変換します。バイトアレイで使用するアルファベットを決定すると、Camel SMPP コンポーネントは

データコーディング値（ヘッダーまたは設定）を考慮しません。ヘッダーまたはエンドポイントパラメーターから、指定されたアルファーのみを考慮します。

String の一部の文字を選択したアルファベットで表示できない場合は、疑問符 ?, 記号に置き換えることができます。API のユーザーは、メッセージボディーをコンポーネントに渡す前に ISO-8859-1 に変換できるかどうかを確認したい場合があり、そうでない場合は、alphabet ヘッダーを UCS-2 エンコーディングを要求するように設定します。アルファベットおよびデータコーディングオプションがまったく指定されていない場合、コンポーネントは必要なエンコーディングを検出し、データコーディングを設定することがあります。

アルファベットコードの一覧は、SMPP 仕様 v3.4 セクション 5.2.19 に記載されています。SMPP 仕様の主な制限の 1 つは、GSM 3.38 (7 ビット)文字セットを明示的に使用するアルファベットコードがないことです。アルファベットの値 0 を選択すると、SMSC のデフォルトのアルファベットが選択されます。これは通常、GSM 3.38 を意味しますが、保証されていません。SMPP ゲートウェイ Nexmo を使用すると、実際には control panel オプションを使用して、デフォルトを別の文字セットにマッピングできます。ユーザーは、SMSC オペレーターでチェックインして、デフォルトとして使用されている文字セットを正確に確認することが推奨されます。

メッセージの分割とスロットリング

メッセージボディーを String から byte 配列に変換すると、Camel コンポーネントは JSMPP に渡す前にメッセージを分割します(140 バイトの SMS サイズ制限内)。これは自動的に完了します。

GSM 3.38 alphabet を使用すると、コンポーネントは最大 160 文字を 140 バイトメッセージのボディーにパックします。8 ビット文字セット（例：western Europe の場合は ISO-8859-1）を使用すると、140 文字が 140 バイトメッセージのボディー内で許可されます。16 ビット UCS-2 エンコーディングを使用する場合は、140 バイトメッセージごとに 70 文字のみになります。

一部の SMSC プロバイダーはスロットリングルールを実装します。分割されたメッセージの各部分は、プロバイダーのスロットリングメカニズムによって個別にカウントされる可能性があります。Camel Throttler コンポーネントは、SMSC に渡す前に SMPP ルート内のメッセージをスロットリングするのに便利です。

URI 形式

```
smpp://[username@]hostname[:port][?options]
smpps://[username@]hostname[:port][?options]
```

ユーザー 名が指定されていない場合、Apache Camel はデフォルト値の smppclient を提供します。ポート 番号が指定されていない場合、Apache Camel はデフォルト値の 2775 を提供しま

*す。**Camel 2.3:** プロトコル名が **smpps** の場合、**SSLSocket** を使用してサーバーへの接続を開始します。*

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	説明
password	**password**	SMSC へのログインに使用するパスワードを指定します。
systemType	**cp**	このパラメーターは、SMSC (max. 13 文字)にバインドする ESME （外部ショートカットメッセージエンティティー）のタイプを分類するために使用されます。
dataCoding	**0**	**Camel 2.11** は、SMPP 3.4 仕様のセクション 5.2.19 に従って、データのコーディングを定義します。(**Camel 2.9** より前は、このオプションもサポートされます。) データエンコーディングの例は次のとおりです。 • **0**: SMSC デフォルトアルファベット • **3**: ラテン 1 (ISO-8859-1) • **4**: オクテットが未指定 (8-bit バイナリー) • **8**: UCS2 (ISO/IEC-10646) • **13**: Extended Kanji JIS (X 0212-1990)

alphabet	**0**	**Camel 2.5** は、SMPP 3.4 仕様、セクション 5.2.19 に従ってデータのエンコーディングを定義します。このオプションは **Alphabet.java** にマッピングされ、SMSC に送信される **byte[]** を作成するために使用されます。例： **0**: SMSC Default Alphabet**4**: 8 ビットアルファベット **8**: UCS2 Alphabet
encoding	**ISO-8859-1**	**SubmitSm のみ**、**ReplaceSm および SubmitMulti** は、短いメッセージユーザーデータのエンコーディングスキームを定義します。
enquireLinkTimer	**5000**	自信チェックの間隔をミリ秒単位で定義します。自信チェックは、ESME と SMSC の間の通信パスをテストするために使用されます。
transactionTimer	**10000**	トランザクションの後に許容される最大非アクティブ期間を定義します。その後、SMPP エンティティーはセッションがアクティブでなくなったと仮定する可能性があります。このタイマーは、SMPP エンティティーの通信 (SMSC または ESME)のいずれかでアクティブになります。
initialReconnectDelay	**5000**	接続が失われた場合にコンシューマー/プロデューサーが SMSC への再接続を試みた後の最初の遅延をミリ秒単位で定義します。
reconnectDelay	**5000**	SMSC への接続が失われ、以前の成功していない場合に、再接続試行の間隔をミリ秒単位で定義します。
registeredDelivery	**1**	**SubmitSm**、**replaceSm**、**SubmitMulti**、**および DataSm** Is は、SMSC 配信受信および SME による確認応答の要求に使用されます。**0**: No SMSC delivery receipt requested の値を定義します。**1**: SMSC 配信受信が要求され、最終的な配信結果が成功または失敗します。**2**: SMSC 配信応答が要求され、最終的な配信結果が配信に失敗しました。

serviceType	**CMT**	サービスタイプパラメーターを使用して、メッセージに関連付けられた SMS アプリケーションサービスを指定できます。以下の汎用 service_types が定義されます。 ● **CMT**: Cellular Messaging ● **CPT**: Cellular Paging ● **VMN**: Voice Mail Notification ● **VMA**: Voice Mail Alerting ● **WAP**: ワイヤレスアプリケーションプロトコル ● **USSD**: 非構造化 Supplementary サービスデータ
sourceAddr	**1616**	このメッセージを発信した SME (Short Message Entity)のアドレスを定義します。
destAddr	**1717**	**SubmitSm、SubmitMulti、CancelSm、および DataSm のみ**が宛先 SME アドレスを定義します。モバイル終端メッセージの場合、これは受信者 MS のディレクトリー番号です。
sourceAddrTon	**0**	SME 送信元アドレスパラメーターで使用される番号(TON)のタイプを定義します。以下の TON 値を定義します。 ● **0**: unknown ● **1**: 国際 ● **2**: 国語 ● **3**: ネットワーク固有 ● **4**: サブスクライバー番号 ● **5**: 英数字 ● **6**: 省略形

destAddrTon	**0**	**SubmitSm、SubmitMulti、CancelSm、および DataSm にのみ**、SME 宛先アドレスパラメーターで使用される番号のタイプ(TON)を定義します。以下の TON 値を定義します。 • **0**: unknown • **1**: 国際 • **2**: 国語 • **3**: ネットワーク固有 • **4**: サブスクライバー番号 • **5**: 英数字 • **6**: 省略形
sourceAddrNpi	**0**	SME 送信元アドレスパラメーターで使用される数値計画インジケーター(NPI)を定義します。以下の NPI 値を定義します。 • **0**: unknown • **1**: ISDN (E163/E164) • **2**: データ(X.121) • **3**: テレックス(F.69) • **6**: land Mobile (E.212) • **8**: National • **9**: プライベート • **10**: ERMES • **13**: インターネット(IP) • **18**: WAP クライアント ID (WAP Forum で定義)

destAddrNpi	**0**	**SubmitSm、SubmitMulti、CancelSm、および DataSm にのみ**、SME 宛先アドレスパラメーターで使用される数値計画インジケーター(NPI)を定義します。以下の NPI 値を定義します。 • **0**: unknown • **1**: ISDN (E163/E164) • **2**: データ(X.121) • **3**: テレックス(F.69) • **6**: land Mobile (E.212) • **8**: National • **9**: プライベート • **10**: ERMES • **13**: インターネット(IP) • **18**: WAP クライアント ID (WAP Forum で定義)
priorityFlag	**1**	**SubmitSm および SubmitMulti にのみ**、送信元の SME が短いメッセージに優先度レベルを割り当てることができます。4 つの優先度レベルがサポートされます。 • **0**: レベル 0 （最低）の優先度 • **1**: レベル 1 の優先度 • **2**: レベル 2 の優先度 • **3**: レベル 3 （最高）の優先度
replaceIfPresentFlag	**0**	**SubmitSm および SubmitMulti Used が SMSC を要求する場合にのみ**、以前に送信されたメッセージを置き換えます。これは、配信が保留されています。SMSC は、ソースアドレス、宛先アドレス、およびサービスタイプが新しいメッセージの同じフィールドと一致する場合に、既存のメッセージを置き換えます。フラグ値が定義されている場合は、以下の置換を行います。 • **0**: 置き換えません。 • **1**: replace

dataCoding	**0**	**Camel 2.5 onwarts** は、SMPP 3.4 仕様セクション 5.2.19 に従ってデータのエンコーディングを定義します。データエンコーディングの例： **0**: SMSC Default Alphabet **4**: 8 ビットアルファベット **8**: UCS2 Alphabet
typeOfNumber	**0**	SME で使用される番号(TON)のタイプを定義します。以下の TON 値を定義します。 • **0**: unknown • **1**: 国際 • **2**: 国語 • **3**: ネットワーク固有 • **4**: サブスクライバー番号 • **5**: 英数字 • **6**: 省略形
numberingPlanIndicator	**0**	SME で使用される数値計画インジケーター(NPI)を定義します。以下の NPI 値を定義します。 • **0**: unknown • **1**: ISDN (E163/E164) • **2**: データ(X.121) • **3**: テレックス(F.69) • **6**: land Mobile (E.212) • **8**: National • **9**: プライベート • **10**: ERMES • **13**: インターネット(IP) • **18**: WAP クライアント ID (WAP Forum で定義)

lazySessionCreation	**false**	**Camel 2.8 以降** の Sessions は、Camel プロデューサーの開始時に SMSC が利用できない場合に例外を回避するために遅延して作成できます。**Camel 2.11 以降** の Camel は、最初のエクスチェンジのメッセージヘッダー 'CamelSmppSystemId' および 'CamelSmppPassword' で を確認します。存在する場合は、Camel はこれらのデータを使用して SMSC に接続します。
httpProxyHost	**null**	**Camel 2.9.1:** HTTP プロキシー経由で SMPP をトンネルする必要がある場合は、この属性を HTTP プロキシーのホスト名または IP アドレスに設定します。
httpProxyPort	**3128**	**Camel 2.9.1:** HTTP プロキシー経由で SMPP をトンネルする必要がある場合は、この属性を HTTP プロキシーのポートに設定します。
httpProxyUsername	**null**	**Camel 2.9.1:** HTTP プロキシーに Basic 認証が必要な場合は、この属性を HTTP プロキシーに必要なユーザー名に設定します。
httpProxyPassword	**null**	**Camel 2.9.1:** HTTP プロキシーに Basic 認証が必要な場合は、この属性を HTTP プロキシーに必要なパスワードに設定します。
sessionStateListener	**null**	Camel 2.9.3: **Registry** の **org.jsmpp.session.SessionStateListener** を参照して、セッション状態が変更されたときにコールバックを受け取ることができます。
addressRange	**""**	**Camel 2.11:** SMPP 3.4 仕様のセクション 5.2.7 で定義されているように **SmppConsumer** のアドレス範囲を指定できます。**SmppConsumer** は、この範囲内のアドレス(MSISDN または IP アドレス)をターゲットにする SMSC からのみメッセージを受信します。

splittingPolicy	**ALLOW**	Camel 2.14.1 および 2.15.0: 以下のように、長いメッセージを処理するポリシーを指定できます。 ● **ALLOW**: (デフォルト) 長いメッセージはメッセージごとに 140 バイトに分割されます。 ● **TRUNCATE**: 長いメッセージは分割され、最初のフラグメントのみが SMSC に送信されます。一部の受け入れ者は後続のフラグメントを削除するため、配信されないメッセージの一部を送信する SMPP 接続の負荷が軽減されます。 ● **REJECT**: メッセージを分割する必要がある場合は、SMPP **NegativeResponseException** で拒否され、メッセージを示す理由コードは長すぎます。
proxyHeaders	**null**	Camel 2.17: 接続の確立中にこれらのヘッダーがプロキシーサーバーに渡されます。

これらのオプションはいくつでも使用できます。

> ***smpp://smppclient@localhost:2775?password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consumer***

プロデューサーメッセージヘッダー

以下のメッセージヘッダーを使用して、SMPP プロデューサーの動作に影響を与えることができます。

ヘッダー	タイプ	説明

CamelSmppDestAddr	**List/String**	**SubmitSm、SubmitMulti、CancelSm、および DataSm のみ**が宛先 SME アドレスを定義します。モバイル終端メッセージの場合、これは受信者 MS のディレクトリー番号です。SubmitMulti の場合は **List<String>** であり、それ以外の場合は **String** である必要があります。
CamelSmppDestAddrTon	**Byte**	**SubmitSm、SubmitMulti、CancelSm、および DataSm にのみ**、SME 宛先アドレスパラメーターで使用される番号のタイプ (TON)を定義します。以下の TON 値を定義します。 ● **0**: unknown ● **1**: 国際 ● **2**: 国語 ● **3**: ネットワーク固有 ● **4**: サブスクライバー番号 ● **5**: 英数字 ● **6**: 省略形
CamelSmppDestAddrNpi	**Byte**	**SubmitSm、SubmitMulti、CancelSm、および DataSm にのみ**、SME 宛先アドレスパラメーターで使用される数値計画インジケーター(NPI)を定義します。以下の NPI 値を定義します。 ● **0**: unknown ● **1**: ISDN (E163/E164) ● **2**: データ(X.121) ● **3**: テレックス(F.69) ● **6**: land Mobile (E.212) ● **8**: National ● **9**: プライベート ● **10**: ERMES ● **13**: インターネット(IP) ● **18**: WAP クライアント ID (WAP Forum で定義)

CamelSmppSourceAddr	**文字列**	このメッセージを発信した SME (Short Message Entity)のアドレスを定義します。
CamelSmppSourceAddrTon	**Byte**	SME 送信元アドレスパラメーターで使用される番号(TON)のタイプを定義します。以下の TON 値を定義します。 • **0**: unknown • **1**: 国際 • **2**: 国語 • **3**: ネットワーク固有 • **4**: サブスクライバー番号 • **5**: 英数字 • **6**: 省略形
CamelSmppSourceAddrNpi	**Byte**	SME 送信元アドレスパラメーターで使用される数値計画インジケーター(NPI)を定義します。以下の NPI 値を定義します。 • **0**: unknown • **1**: ISDN (E163/E164) • **2**: データ(X.121) • **3**: テレックス(F.69) • **6**: land Mobile (E.212) • **8**: National • **9**: プライベート • **10**: ERMES • **13**: インターネット(IP) • **18**: WAP クライアント ID (WAP Forum で定義)

CamelSmppServiceType	**文字列**	サービスタイプパラメーターを使用して、メッセージに関連付けられた SMS アプリケーションサービスを指定できます。以下の汎用 service_types が定義されます。 • **CMT**: Cellular Messaging • **CPT**: Cellular Paging • **VMN**: Voice Mail Notification • **VMA**: Voice Mail Alerting • **WAP**: ワイヤレスアプリケーションプロトコル • **USSD**: 非構造化 Supplementary サービスデータ
CamelSmppRegisteredDelivery	**Byte**	**SubmitSm、replaceSm、SubmitMulti、および DataSm** Is は、SMSC 配信受信および SME による確認応答の要求に使用されます。以下の値が定義されます。 • **0**: no SMSC delivery receipt requested. • **1**: SMSC 配信受信が要求され、最終的な配信結果が成功または失敗します。 • **2**: SMSC 配信応答が要求され、最終的な配信結果が配信に失敗しました。
CamelSmppPriorityFlag	**Byte**	**SubmitSm および SubmitMulti にのみ**、送信元の SME が短いメッセージに優先度レベルを割り当てることができます。4 つの優先度レベルがサポートされます。 • **0**: レベル 0 （最低）の優先度 • **1**: レベル 1 の優先度 • **2**: レベル 2 の優先度 • **3**: レベル 3 （最高）の優先度

CamelSmppValidityPeriod	**string**/{{Date}}	**SubmitSm、SubmitMulti、および ReplaceSm のみ。**validity period パラメーターは SMSC の有効期限を示します。その後、宛先に配信されない場合はメッセージを破棄する必要があります。**日付** として提供されている場合は、絶対時間として解釈されます。**Camel 2.9.1 以降：** smpp 仕様 v3.4 の章 7.1.1 で指定されている **文字列** として指定する場合は、絶対時間形式または相対時間形式で定義できます。
CamelSmppReplaceIfPresentFlag	**Byte**	**SubmitSm および SubmitMulti にのみ**、replaces フラグパラメーターを使用して、以前に送信されたメッセージを置き換えるように SMSC を要求します。これは、配信が保留されています。SMSC は、ソースアドレス、宛先アドレス、およびサービスタイプが新しいメッセージの同じフィールドと一致する場合に、既存のメッセージを置き換えます。以下の値が定義されます。 ● **0**: 置き換えません。 ● **1**: replace
CamelSmppAlphabet / **CamelSmppDataCoding**	**Byte**	**Camel 2.5For SubmitSm, SubmitMulti and ReplaceSm** (**Camel 2.9** の場合は、**CamelSmppAlphabet** の代わりに **CamelSmppDataCoding** を使用します。) SMPP 3.4 仕様、セクション 5.2.19 に準拠したデータのコーディング。上記の URI オプション **アルファベット** 設定を使用してください。
CamelSmppOptionalParameters	**Map<String, String>**	**非推奨および は Camel 2.13.0/3.0.0 Camel 2.10.5 および 2.11.1 以降で削除され、SubmitSm、SubmitMulti、および DataSm に対してのみ**削除されます。オプションのパラメーターは SMSC によって返信されます。

CamelSmppOptionalParameter	**Map<Short, Object>**	Camel 2.10.7 および 2.11.2 以降、SubmitSm、SubmitMulti、および DataSm に対してのみ、SMSC に送信されるオプションのパラメーターです。値は以下の方法で変換されます。 **string** -> **org.jsmpp.bean.OptionalParameter.COctetStringbyte[]** -> **org.jsmpp.bean.OptionalParameter.OctetStringByte** -> **org.jsmpp.bean.OptionalParameter.ByteInteger -> org.jsmpp.bean .OptionalParameter.IntShort** -> **org.jsmpp.bean.OptionalParameter.Shortnull** -> **org.jsmpp.bean.OptionalParameter.Null**
CamelSmppEncoding	**文字列**	Camel 2.14.1 および Camel 2.15.0: および **SubmitSm**、**SubmitMulti** および **DataSm** のみ。メッセージボディーのバイトのエンコーディング（文字セット名）を指定します。メッセージボディーが文字列である場合、Java 文字列は常に Unicode であるため、これは関係ありません。ボディーがバイト配列である場合、このヘッダーを使用して ISO-8859-1 またはその他の値であることを示します。デフォルト値は、エンドポイント設定パラメーターのエンコーディングによって指定されます。
CamelSmppSplittingPolicy	**文字列**	Camel 2.14.1 および Camel 2.15.0: および **SubmitSm**、**SubmitMulti** および **DataSm** のみ。このエクスチェンジのメッセージ分割のポリシーを指定します。使用できる値は、エンドポイント設定パラメーター **splittingPolicy** で説明されています。

以下のメッセージヘッダーは SMPP プロデューサーによって使用され、メッセージヘッダーの SMSC からの応答を設定します。

ヘッダー	タイプ	説明

CamelSmppId	**List<String>**/{{String}}	後で使用するために送信された短いメッセージを識別する ID。**Camel 2.9.0**: ReplaceSm、QuerySm、CancelSm、および DataSm のヘッダー vaule は **文字列** です。SubmitSm または SubmitMultiSm の場合、このヘッダー vaule は **List<String> です**。
CamelSmppSentMessageCount	**整数**	**Camel 2.9 以降、SubmitSm および SubmitMultiSm に対してのみ**、送信されたメッセージの合計数。
CamelSmppError	**Map<String, List<Map<String, Object>>>**	**Camel 2.9 以降、SubmitMultiSm でのみ** 短いメッセージを送信し、Map<String, **List<Map<String, Object>>> (messageID : (destAddr : address,** error : errorCode))形式で送信して発生したエラーです。
CamelSmppOptionalParameters	**Map<String, String>**	**非推奨および は Camel 2.13.0/3.0.0 のCamel 2.11.1 からのみ削除さ れます。その後、DataSm に対して**のみ削除されます。任意のパラメーターは、メッセージを送信して SMSC から返されます。
CamelSmppOptionalParameter	**Map<Short, Object>**	**Camel 2.10.7 以降では、2.11.2 以降の DataSm のみ**。メッセージを送信して SMSC から返されるオプションのパラメーターです。キーは、オプションのパラメーターの **Short** コードです。値は以下の方法で変換されます。**org.jsmpp.bean.OptionalParameter.COctetString** -> **Stringorg.jsmpp.bean.OptionalParameter.OctetString** -> **byte[]org.jsmpp.bean.OptionalParameter.Byte** -> **Byteorg.jsmpp.bean.OptionalParameter.Int ->integerorg.jsmpp.bean.OptionalParameter.Short -> Shortorg.jsmpp.bean.OptionalParameter.Null -> null**

コンシューマーメッセージヘッダー

*以下のメッセージヘッダーは **SMPP** コンシューマーによって使用され、メッセージヘッダーの **SMSC** からのリクエストデータを設定します。*

ヘッダー	タイプ	説
CamelSmppSequenceNumber	**整数**	**Al** ト 必
CamelSmppCommandId	**整数**	**Al** の 照
CamelSmppSourceAddr	**文字列**	**Al** M
CamelSmppSourceAddrNpi	**Byte**	**Al** 数
CamelSmppSourceAddrTon	**Byte**	**Al** 番

CamelSmppEsmeAddr	**文字列**	Al れ
CamelSmppEsmeAddrNpi	**Byte**	ア (N
CamelSmppEsmeAddrTon	**Byte**	ア ま
CamelSmppId	**文字列**	sn ジ
CamelSmppDelivered	**整数**	配 デ 頭
CamelSmppDoneDate	**日付**	sn st

CamelSmppStatus	**DeliveryReceiptState**	sn de し り M
CamelSmppCommandStatus	**整数**	**Da**
CamelSmppError	**文字列**	sn er せ
CamelSmppSubmitDate	**日付**	短 合 お
CamelSmppSubmitted	**整数**	最 ジ て
CamelSmppDestAddr	**文字列**	**D** 場
CamelSmppScheduleDeliveryTime	**文字列**	**D** 指 の き
CamelSmppValidityPeriod	**文字列**	**D** さ で 指
CamelSmppServiceType	**文字列**	**D** メ se

CamelSmppRegisteredDelivery	**Byte**	Da 以
CamelSmppDestAddrNpi	**Byte**	Da 義
CamelSmppDestAddrTon	**Byte**	Da 以
CamelSmppMessageType	文字列	Ca 通 **D**
CamelSmppOptionalParameters	**Map<String, Object>**	非 除

CamelSmppOptionalParameter	Map<Short, Object>	Ca は 以 St by By ;ir SI

JSMPP ライブラリー

基礎となるライブラリーの詳細は、JSMPP ライブラリーのドキュメントを参照してください。

例外処理

このコンポーネントは、一般的な Camel 例外処理機能をサポートします。

SubmitSm （デフォルトアクション）でメッセージの送信にエラーが発生した場合、org.apache.camel.component.smpp.SmppException はネストされた例外 org.jsmpp.extra.NegativeResponseException で出力されます。NegativeResponseException.getCommandStatus() を呼び出して、正確な SMPP 負の応答コードを取得します。値は、SMPP 仕様 3.4 のセクション 5.1.3 で説明されています。

Camel 2.8 onwarts: SMPP コンシューマーが DeliverSm または DataSm 短いメッセージを受信し、これらのメッセージの処理に失敗した場合は、失敗を処理する代わりに ProcessRequestException を出力することもできます。この場合、この例外は基礎となる JSMPP ライブラリー に転送され、含まれるエラーコードを SMSC に戻します。この機能は、たとえば、後で短いメッセージを再送信するように SMSC に指示するのに役立ちます。これは、以下のコード行を使用して実行できます。

```
from("smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consum
er")
  .doTry()
    .to("bean:dao?method=updateSmsState")
  .doCatch(Exception.class)
    .throwException(new ProcessRequestException("update of sms state failed", 100))
  .end();
```

エラーコードとその意味の完全なリストについては、SMPP 仕様 を参照してください。

サンプル

Java DSL を使用して SMS を送信するルート：

```
from("direct:start")
  .to("smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=producer
");
```

Spring XML DSL を使用して SMS を送信するルート：

```
<route>
  <from uri="direct:start"/>
  <to uri="smpp://smppclient@localhost:2775?
password=password&nquireLinkTimer=3000&ransactionTimer=5000&ystemType=producer"/>
</route>
```

Java DSL を使用して SMS を受信するルート：

```
from("smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consum
er")
  .to("bean:foo");
```

Spring XML DSL を使用して SMS を受信するルート：

```
  <route>
     <from uri="smpp://smppclient@localhost:2775?
password=password&nquireLinkTimer=3000&ransactionTimer=5000&ystemType=consumer"/
>
     <to uri="bean:foo"/>
  </route>
```

SMSC シミュレーター

テストに SMSC シミュレーターが必要な場合は、Logica が提供するシミュレーターを使用できます。

デバッグロギング

このコンポーネントにはログレベル DEBUG があり、問題のデバッグに役立ちます。log4j を使用する場合は、以下の行を設定に追加できます。

```
log4j.logger.org.apache.camel.component.smpp=DEBUG
```

第154章 SNMP

SNMP コンポーネント

snmp: コンポーネントを使用すると、SNMP 対応デバイスをポーリングしたり、トラップを受信したりできます。

URI 形式

```
snmp://hostname[:port][?Options]
```

コンポーネントは、SNMP 対応のデバイスからの OID 値のポーリングとトラップの受信をサポートします。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	デフォルト値	説明
type	none	実行するアクションのタイプ。**POLL** または **TRAP** に実際に入力できます。値 **POLL** は、提供された OID キーに対して指定のホストをポーリングするようエンドポイントに指示します。**TRAP** に を追加すると、SNMP トレイトイベントのリスナーを設定します。
address	none	これは、ポーリングするホストの IP アドレスとポート、または Trap Receiver を設定する場所です。例： **127.0.0.1:162**
protocol	**udp**	ここでは、使用するプロトコルを選択できます。**udp** または **tcp** のいずれかを使用できます。
retries	**2**	要求をキャンセルする前に再試行が実行される頻度を定義します。
timeout	**1500**	リクエストのタイムアウト値をミリ秒単位で設定します。

snmpVersion	**0** (SNMPv1)	リクエストの SNMP バージョンを設定します。
snmpCommunity	**public**	snmp リクエストのコミュニティーオクテット文字列を設定します。
delay	**60000**	2 つのポーリングサイクル間の遅延をミリ秒単位で定義します。以前のリリースでは、このオプションは秒単位で指定されていました。
oids	none	対象の値を定義します。理解を深めるには、Wikipedia をご参照ください。OID を 1 つまたは区切ったリストを指定できます。例: **oids="1.3.6.1.2.1.1.3.0,1.3.6.1.2.1.25.3.2.1.5.1,1.3.6.1.2.1.25.3.5.1.1.1,1.3.6.1.2.1.43.5.1.1.11.1"**

ポーリングの結果

以下の OID をポーリングする状況を想定します。

```
1.3.6.1.2.1.1.3.0
1.3.6.1.2.1.25.3.2.1.5.1
1.3.6.1.2.1.25.3.5.1.1.1
1.3.6.1.2.1.43.5.1.1.11.1
```

結果は以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<snmp>
  <entry>
    <oid>1.3.6.1.2.1.1.3.0</oid>
    <value>6 days, 21:14:28.00</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.25.3.2.1.5.1</oid>
    <value>2</value>
  </entry>
  <entry>
```

```
    <oid>1.3.6.1.2.1.25.3.5.1.1.1</oid>
    <value>3</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.43.5.1.1.11.1</oid>
    <value>6</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.1.1.0</oid>
    <value>My Very Special Printer Of Brand Unknown</value>
  </entry>
</snmp>
```

認識されているように、requested...1.3.6.1.2.1.1.1.0 よりも多くの結果がある可能性があります。この特別なケースでは、デバイスが自動的に入力されます。そのため、絶対的に発生する可能性があり、要求よりも多くの要求を受け取る可能性があります。...be の準備は完了しています。

例

リモートデバイスのポーリング

```
snmp:192.168.178.23:161?protocol=udp&type=POLL&oids=1.3.6.1.2.1.1.5.0
```

トラップレシーバーの設定(ここでは OID 情報は必要ありません)。

```
snmp:127.0.0.1:162?protocol=udp&type=TRAP
```

Camel 2.10.0 から、メッセージヘッダー 'peerAddress' を持つ SNMP TRAP のメッセージヘッダー securityName' のピアアドレスを持つ SNMP TRAP のコミュニティーを取得できます。

Java のルーティングの例(SNMP PDU を XML 文字列に変換します)。

```
from("snmp:192.168.178.23:161?protocol=udp&type=POLL&oids=1.3.6.1.2.1.1.5.0").
convertBodyTo(String.class).
to("activemq:snmp.states");
```

第155章 SOLR

SOLR コンポーネント

Camel 2.9 以降で利用可能

Solr コンポーネントを使用すると、[Apache Lucene Solr](#) サーバー(SolrJ 3.5.0)とのインターフェイスが可能になります。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-solr</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

URI 形式は以下のとおりです。

```
solr://host[:port]/solr?[options]
solrs://host[:port]/solr ?[options]
solrCloud://host[:port]/solr?[options]
```

エンドポイントオプション

以下の [SolrServer](#) オプションは、Solr エンドポイントで設定できます。

name	デフォルト値	description
maxRetries	**0**	一時的なエラーが発生した場合に試行する最大再試行回数

soTimeout	**1000**	基礎となる HttpConnectionManager で読み取りタイムアウト。これはクエリーに適していますが、インデックス作成には適していません。
connectionTimeout	**100**	基礎となる HttpConnectionManager の connectionTimeout
defaultMaxConnectionsPerHost	**2**	基礎となる HttpConnectionManager の maxConnectionsPerHost
maxTotalConnections	**20**	基礎となる HttpConnectionManager の maxTotalConnection
followRedirects	**false**	Solr サーバーへリダイレクトが使用されるかどうかを示します。
allowCompression	**false**	これを有効にするには、サーバー側が gzip または deflate をサポートする必要があります。
requestHandler	/update (xml)	使用するリクエストハンドラーの設定
streamingThreadCount	**2**	**Camel 2.9.2** で StreamingUpdateSolrServerのスレッド数を設定する
streamingQueueSize	**10**	**Camel 2.9.2** で StreamingUpdateSolrServerのキューサイズを設定する
zkhost	**null**	**Camel 2.14.0** は、zkhost=localhost:8123など、solrCloud が使用可能な zoo keeper ホスト情報を設定します。
collection	**null**	**Camel 2.14.0** は、solrCloud サーバーが使用できるコレクション名を設定します

メッセージ操作

*以下の Solr 操作は現在サポートされています。キー **SolrOperation** でエクスチェンジヘッダーを設定し、値は以下のいずれかに設定します。一部の操作では、メッセージボディーも設定する必要があり*

ます。

- *INSERT 操作は CommonsHttpSolrServerを使用します。*
- *INSERT_STREAMING 操作は StreamingUpdateSolrServer (Camel 2.9.2)を使用します。*

operation	メッセージボディー	descript
INSERT/INSERT_STREAMING	該当なし	メッセー 必要です
INSERT/INSERT_STREAMING	File	指定され (Conter
INSERT/INSERT_STREAMING	SolrInputDocument	**Camel 2** ます。
INSERT/INSERT_STREAMING	String XML	**Camel 2** (SolrInp
ADD_BEAN	Bean インスタンス	アノテー
ADD_BEANS	collection<bean>	**Camel 2** インデッ
DELETE_BY_ID	削除するインデックス ID	ID 別に
DELETE_BY_QUERY	クエリー文字列	クエリー
COMMIT	該当なし	保留中の
ROLLBACK	該当なし	保留中の
OPTIMIZE	該当なし	保留中の す。

例

以下は、簡単な INSERT、DELETE および COMMIT の例です。

```
from("direct:insert")
    .setHeader(SolrConstants.OPERATION, constant(SolrConstants.OPERATION_INSERT))
    .setHeader(SolrConstants.FIELD + "id", body())
```

```
    .to("solr://localhost:8983/solr");

from("direct:delete")
    .setHeader(SolrConstants.OPERATION,
constant(SolrConstants.OPERATION_DELETE_BY_ID))
    .to("solr://localhost:8983/solr");

from("direct:commit")
    .setHeader(SolrConstants.OPERATION, constant(SolrConstants.OPERATION_COMMIT))
    .to("solr://localhost:8983/solr");
```

```
<route>
    <from uri="direct:insert"/>
    <setHeader headerName="SolrOperation">
        <constant>INSERT</constant>
    </setHeader>
    <setHeader headerName="SolrField.id">
        <simple>${body}</simple>
    </setHeader>
    <to uri="solr://localhost:8983/solr"/>
</route>
<route>
    <from uri="direct:delete"/>
    <setHeader headerName="SolrOperation">
        <constant>DELETE_BY_ID</constant>
    </setHeader>
    <to uri="solr://localhost:8983/solr"/>
</route>
<route>
    <from uri="direct:commit"/>
    <setHeader headerName="SolrOperation">
        <constant>COMMIT</constant>
    </setHeader>
    <to uri="solr://localhost:8983/solr"/>
</route>
```

クライアントは単にメッセージボディーを挿入ルートまたは削除ルートに渡してから、コミットルートを呼び出す必要があります。

```
    template.sendBody("direct:insert", "1234");
    template.sendBody("direct:commit", null);
    template.sendBody("direct:delete", "1234");
    template.sendBody("direct:commit", null);
```

SOLR のクエリー

現在、このコンポーネントはデータのクエリーをネイティブにサポートしません（後で追加される可能性があります）。ここでは、以下のように HTTP を使用して Solr をクエリーできます。

```
//define the route to perform a basic query
```

```
from("direct:query")
   .recipientList(simple("http://localhost:8983/solr/select/?q=${body}"))
   .convertBodyTo(String.class);
...
//query for an id of '1234' (url encoded)
String responseXml = (String) template.requestBody("direct:query", "id%3A1234");
```

詳細は、これらのリソースを参照してください。

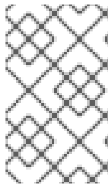
Solr Query Tutorial

Solr クエリー構文

第156章 APACHE SPARK

APACHE SPARK コンポーネント

このドキュメントページでは、Apache Camel の Apache Spark コンポーネントについて説明します。Camel との Spark インテグレーションの主な目的は、Camel コネクターと Spark タスクの間にブリッジを提供することです。特に Camel コネクターは、さまざまなトランスポートからメッセージをルーティングする方法を提供し、実行するタスクを動的に選択し、受信メッセージをそのタスクの入力データとして使用し、最後に実行の結果を Camel パイプラインに戻します。

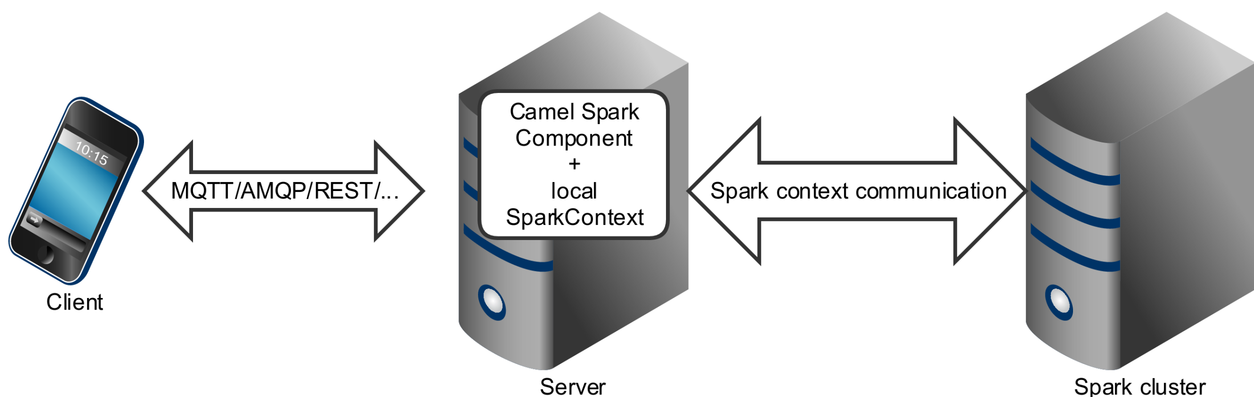


注記

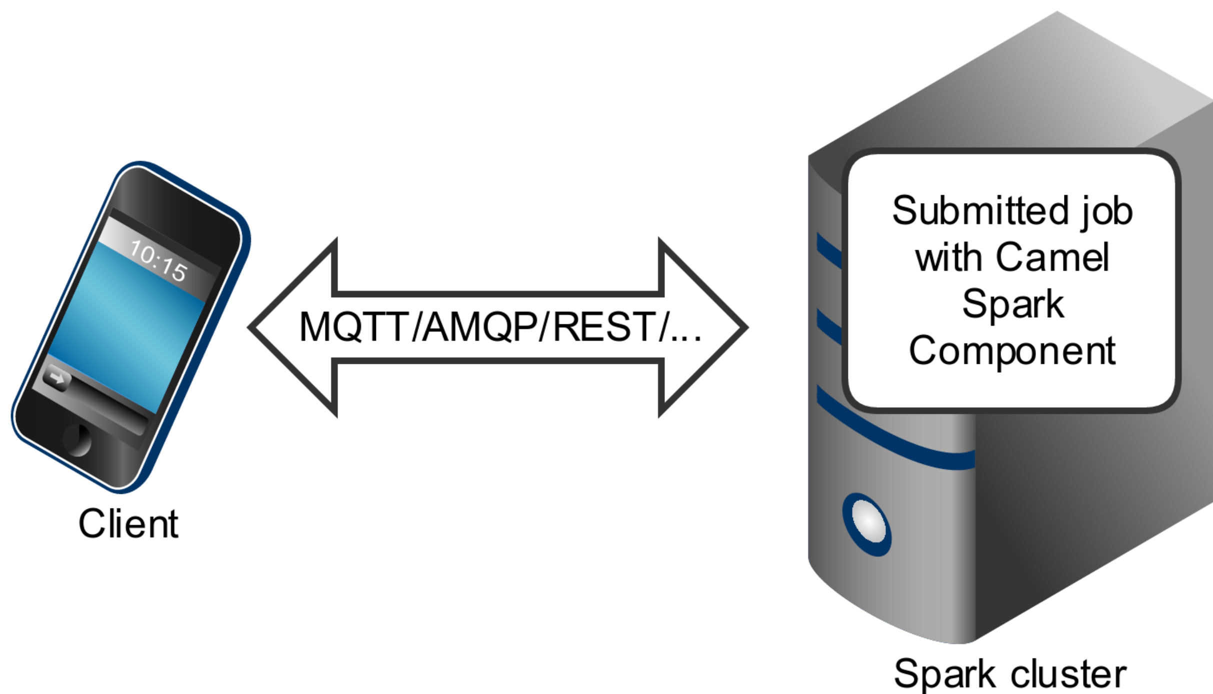
Apache Spark は Apache Karaf ではサポートされていません。

サポート対象のアーキテクチャースタイル

Spark コンポーネントは、アプリケーションサーバーにデプロイされるドライバーアプリケーションとして使用できます（または fat jar として実行）。



Spark コンポーネントは、Spark クラスターに直接ジョブとして送信することもできます。



Spark コンポーネントは、Spark クラスターと他のエンドポイント間のブリッジとして機能する 長時間実行されるジョブ *として機能するように設計されていますが、これは* 1 回 *限りの短いジョブとして使用することもできます。*

OSGI サーバーでの SPARK の実行

現在、Spark コンポーネントは OSGi コンテナーの実行をサポートしていません。Spark は、fat jar として実行されるように設計されています。通常はジョブとしてクラスターに送信されます。OSGi サーバーで Spark を実行する理由は、少なくとも困難であり、Camel ではサポートされません。

URI 形式

現在、Spark コンポーネントはプロデューサーのみをサポートします。これは、Spark ジョブを呼び出して結果を返すことを目的としています。RDD、データフレーム、または Hive SQL ジョブを呼び出すことができます。

```
spark:{rdd|dataframe|hive}
```

RDD ジョブ

RDD ジョブを呼び出すには、以下の URI を使用します。

```
spark:rdd?rdd=#testFileRdd&rddCallback=#transformation
```

rdd オプションは Camel レジストリーから RDD インスタンスの名前(org.apache.spark.api.java.JavaRDDLikeのサブクラス)を指し、rddCallback は org.apache.camel.component.spark.RddCallback インターフェイス（別名レジストリー）の実装を指します。RDD コールバックは、指定の RDD に対して受信メッセージを適用するために使用される単一のメソッドを提供します。コールバック計算の結果は、エクスチェンジへのボディーとして保存されます。

```
public interface RddCallback<T> {
    T onRdd(JavaRDDLike rdd, Object... payloads);
}
```

以下のスニペットは、メッセージを入力としてジョブに送信し、結果を返す方法を示しています。

```
String pattern = "job input";
long linesCount = producerTemplate.requestBody("spark:rdd?
rdd=#myRdd&rddCallback=#countLinesContaining", pattern, long.class);
```

上記の Spring Bean として登録されたスニペットの RDD コールバックは、以下のようになります。

```
@Bean
RddCallback<Long> countLinesContaining() {
    return new RddCallback<Long>() {
        Long onRdd(JavaRDDLike rdd, Object... payloads) {
            String pattern = (String) payloads[0];
            return rdd.filter({line -> line.contains(pattern)}).count();
        }
    }
}
```

Spring の RDD 定義は以下のようになります。

```
@Bean
JavaRDDLike myRdd(JavaSparkContext sparkContext) {
  return sparkContext.textFile("testrdd.txt");
}
```

RDD ジョブオプション

オプション	説明	デフォルト値
rdd	RDD インスタンス（サブクラス：**org.apache.spark.api.java.JavaRDDLike**）	**null**

rddCallback	org.apache.camel.component.spark.RddCallback インターフェイスのインスタンス。	null

VOID RDD コールバック

RDD コールバックが Camel パイプラインに値を返さない場合は、null 値を返すか、VoidRddCallback ベースクラスを使用できます。

```
@Bean
RddCallback<Void> rddCallback() {
  return new VoidRddCallback() {
        @Override
        public void doOnRdd(JavaRDDLike rdd, Object... payloads) {
            rdd.saveAsTextFile(output.getAbsolutePath());
        }
    };
}
```

RDD コールバックの変換

RDD コールバックに送信される入力データのタイプが分かっている場合は、ConvertingRddCallback を使用し、Camel がそれらをコールバックに挿入する前に受信メッセージを自動的に変換させることができます。

```
@Bean
RddCallback<Long> rddCallback(CamelContext context) {
  return new ConvertingRddCallback<Long>(context, int.class, int.class) {
            @Override
            public Long doOnRdd(JavaRDDLike rdd, Object... payloads) {
                return rdd.count() * (int) payloads[0] * (int) payloads[1];
            }
        };
    };
}
```

アノテーション付きの RDD コールバック

RDD コールバックを使用する最も簡単な方法は、@RddCallback アノテーションが付けられたメソッドをクラスに提供することです。

```
import static
org.apache.camel.component.spark.annotations.AnnotatedRddCallback.annotatedRddCallbac
k;
```

```
@Bean
RddCallback<Long> rddCallback() {
    return annotatedRddCallback(new MyTransformation());
}

...

import org.apache.camel.component.spark.annotation.RddCallback;

public class MyTransformation {

    @RddCallback
    long countLines(JavaRDD<String> textFile, int first, int second) {
        return textFile.count() * first * second;
    }

}
```

CamelContext をアノテーション付きの RDD コールバックファクトリーメソッドに渡すと、作成されたコールバックは受信ペイロードをアノテーション付きメソッドのパラメーターに一致するように変換できます。

```
import static
org.apache.camel.component.spark.annotations.AnnotatedRddCallback.annotatedRddCallbac
k;

@Bean
RddCallback<Long> rddCallback(CamelContext camelContext) {
    return annotatedRddCallback(new MyTransformation(), camelContext);
}

...


import org.apache.camel.component.spark.annotation.RddCallback;

public class MyTransformation {

    @RddCallback
    long countLines(JavaRDD<String> textFile, int first, int second) {
        return textFile.count() * first * second;
    }

}

...

// Convert String "10" to integer
long result = producerTemplate.requestBody("spark:rdd?
rdd=#rdd&rddCallback=#rddCallback" Arrays.asList(10, "10"), long.class);
```

DATAFRAME ジョブ

RDDs Spark コンポーネントを操作する代わりに、DataFrames でも機能します。

DataFrame ジョブを呼び出すには、以下の URI を使用します。

```
spark:dataframe?dataFrame=#testDataFrame&dataFrameCallback=#transformation
```

dataFrame オプションは Camel レジストリーからのデータフレームインスタンスの名前(org.apache.spark.sql.DataFrameのインスタンス)を指し、dataFrameCallback は org.apache.camel.component.spark.DataFrameCallback インターフェイスの実装（またはレジストリーから）を指します。DataFrame コールバックは、指定の DataFrame に対して受信メッセージを適用するために使用される単一のメソッドを提供します。コールバック計算の結果は、エクスチェンジへのボディーとして保存されます。

```
public interface DataFrameCallback<T> {
    T onDataFrame(DataFrame dataFrame, Object... payloads);
}
```

以下のスニペットは、メッセージを入力としてジョブに送信し、結果を返す方法を示しています。

```
String model = "Micra";
long linesCount = producerTemplate.requestBody("spark:dataFrame?
dataFrame=#cars&dataFrameCallback=#findCarWithModel", model, long.class);
```

上記のスニペットの DataFrame コールバック(Spring Bean として登録されたもの)は、以下のようになります。

```
@Bean
RddCallback<Long> findCarWithModel() {
    return new DataFrameCallback<Long>() {
     @Override
     public Long onDataFrame(DataFrame dataFrame, Object... payloads) {
         String model = (String) payloads[0];
         return dataFrame.where(dataFrame.col("model").eqNullSafe(model)).count();
     }
 };
}
```

Spring の DataFrame 定義は以下のようになります。

```
@Bean
DataFrame cars(HiveContext hiveContext) {
   DataFrame jsonCars = hiveContext.read().json("/var/data/cars.json");
  jsonCars.registerTempTable("cars");
 return jsonCars;
}
```

DATAFRAME ジョブオプション

オプション	説明	デフォルト値
dataFrame	DataFrame インスタンス（サブクラス：**org.apache.spark.sql.DataFrame**）	**null**
dataFrameCallback	**org.apache.camel.component.spark.DataFrameCallback** インターフェイスのインスタンス。	**null**

HIVE ジョブ

RDD または DataFrame Spark コンポーネントを使用する代わりに、Hive SQL クエリーをペイロードとして受信することもできます。Hive クエリーを Spark コンポーネントに送信するには、以下の URI を使用します。

```
spark:hive
```

以下のスニペットは、メッセージを入力としてジョブに送信し、結果を返す方法を示しています。

```
long carsCount = template.requestBody("spark:hive?collect=false", "SELECT * FROM cars",
Long.class);
List<Row> cars = template.requestBody("spark:hive", "SELECT * FROM cars", List.class);
```

クエリーを実行するテーブルは、クエリーをクエリーする前に HiveContext に登録する必要があります。たとえば、Spring ではこのような登録は以下のようになります。

```
@Bean
DataFrame cars(HiveContext hiveContext) {
   DataFrame jsonCars = hiveContext.read().json("/var/data/cars.json");
```

```
 jsonCars.registerTempTable("cars");
 return jsonCars;
}
```

HIVE ジョブオプション

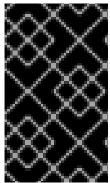
オプション	説明	デフォルト値
collect	結果を(**org.apache.spark.sql.Row** インスタンスのリストとして)収集するか、または **count()** をそれらに対して呼び出す必要があるかどうかを示します。	**true**

第157章 SPARK REST

SPARK REST コンポーネント

Camel 2.14 から利用可能

Spark-rest コンポーネントを使用すると、Rest DSL を使用して Spark REST Java ライブラリーを使用して REST エンドポイントを定義できます。



重要

Spark Java には Java 8 ランタイムが必要です。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-spark-rest</artifactId>
   <version>${camel-version}</version>
</dependency>
```

URI 形式

```
spark-rest://verb:path?[options]
```

URI オプション

名前	デフォルト値	説明
verb		get、post、put、patch、delete、head、trace、connect、または options。
path		Spark 構文をサポートするコンテンツパス。例については、以下を参照してください。
accept		'text/xml' または 'application/json' などの Accept タイプ。デフォルトでは、すべての種類のタイプを受け入れます。

SPARK 構文を使用したパス

path オプションは、パラメーターおよびスプラットのサポートを使用して REST コンテキストパスを定義する Spark REST 構文を使用して定義されます。詳細は、Spark Java Route ドキュメントを参照してください。

以下は、固定パスを使用した Camel ルートです。

```
from("spark-rest:get:hello")
  .transform().constant("Bye World");
```

以下のルートは、キーme を持つ Camel ヘッダーにマップされる パラメーターを使用します。

```
from("spark-rest:get:hello/:me")
  .transform().simple("Bye ${header.me}");
```

CAMEL メッセージへのマッピング

Spark Request オブジェクトは、getRequest メソッドを使用して生の Spark 要求にアクセスできる org.apache.camel.component.sparkrest.SparkMessage として Camel Message にマッピングされます。デフォルトでは、Spark ボディーは Camel メッセージボディーにマッピングされ、HTTP ヘッダー/Spark パラメーターは Camel メッセージヘッダーにマッピングされます。Spark スプラット構文には特別なサポートがあり、これはキースプラットを使用して Camel メッセージヘッダーにマッピングされます。

たとえば、以下のルートは、コンテキストパスで Spark splat （アスタリスク記号）を使用します。このコンテキストは、Simple 言語の形式のヘッダーとしてアクセスして応答メッセージを構築できます。

```
from("spark-rest:get:/hello/*/to/*")
  .transform().simple("Bye big ${header.splat[1]} from ${header.splat[0]}");
```

REST DSL

Apache Camel は、REST サービスを nice REST スタイルで定義できるようにする新しい Rest DSL を提供します。たとえば、以下のように REST hello サービスを定義できます。

```
return new RouteBuilder() {
    @Override
    public void configure() throws Exception {
          rest("/hello/{me}").get()
```

```
            .route().transform().simple("Bye ${header.me}");
        }
    };
```

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <rest uri="/hello/{me}">
    <get>
      <route>
        <transform>
          <simple>Bye ${header.me}</simple>
        </transform>
      </route>
    </get>
  </rest>
</camelContext>
```

詳細は Rest DSL を参照してください。

その他の例

Apache Camel ディストリビューションには camel-example-spark-rest-tomcat の例があり、Apache Tomcat または同様の Web コンテナー上にデプロイできる Web アプリケーションで camel-spark-rest を使用する方法を実証します。

第158章 SPLUNK

SPLUNK コンポーネント

Camel 2.13 で利用可能

Splunk コンポーネントは、Splunk が提供する クライアント api を使用して Splunk にアクセスし、Splunk でイベントを公開して検索できます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-splunk</artifactId>
   <version>${camel-version}</version>
</dependency>
```

URI 形式

```
splunk://[endpoint]?[options]
```

プロデューサーエンドポイント：

エンドポイント	説明
stream	データを名前付きインデックスにストリーミングするか、指定されていない場合はデフォルトにストリーミングします。ストリームモードを使用する場合は、Splunk にはイベントがインデックスに到達する前に一部の内部バッファー(about 1MB など)があることを認識します。リアルタイムが必要な場合は、**submit** または **tcp** モードを使用することが推奨されます。
submit	送信モード。Splunk REST API を使用して、イベントを名前付きインデックスまたはデフォルト（指定されていない場合はデフォルト）に公開します。
tcp	TCP モード。データを TCP ポートにストリーミングし、Splunk で開いているレシーバーポートを必要とします。

イベントを公開する場合、メッセージボディーには SplunkEvent が含まれている必要があります。

例

```
from("direct:start").convertBodyTo(SplunkEvent.class)
   .to("splunk://submit?
username=user&password=123&index=myindex&sourceType=someSourceType&source=my
Source")...
```

この例では、SplunkEvent クラスに変換するコンバーターが必要です。

コンシューマーエンドポイント:

エンドポイント	説明
normal	通常の検索を実行し、検索オプションで検索クエリーを必要とします。
savedsearch	splunk に保存されている検索クエリーに基づいて検索を実行し、savedSearch オプションにクエリーの名前が必要です。

例

```
from("splunk://normal?delay=5s&username=user&password=123&initEarliestTime=-
10s&search=search index=myindex sourcetype=someSourcetype")
   .to("direct:search-result");
```

camel-splunk は、ボディーに SplunkEvent を使用して検索結果ごとにルートエクスチェンジを作成します。

URI オプション

名前	デフォルト値	コンテキスト	説明
host	**localhost**	両方	Splunk ホスト。
port	**8089**	両方	Splunk ポート
scheme	**https**	両方	**http** または **https** のいずれかとして使用するスキーム。

username	**null**	両方	Splunk のユーザー名
password	**null**	両方	Splunk のパスワード
connectionTimeout	**5000**	両方	Splunk サーバーへの接続時の MS のタイムアウト
useSunHttpsHandler	**false**	両方	sun.net.www.protocol.https.Handler Https hanlder を使用して Splunk 接続を確立します。は、アプリケーションサーバーで実行している場合に、アプリケーション(server https の処理)を回避するのに役立ちます。
sslProtocol	**TLSv1.2**	両方	Camel 2.16: 使用する SSL プロトコル。**TLSv1.2**、**TLSv1.1**、**TLSv1**、および **SSLv3** のいずれかです。scheme が **https** の場合にのみ使用されます。
index	**null**	プロデューサー	書き込む Splunk インデックス
sourceType	**null**	プロデューサー	Splunk sourcetype arguement
source	**null**	プロデューサー	Splunk ソース引数
tcpReceiverPort	**0**	プロデューサー	tcp プロデューサーエンドポイントを使用する場合の Splunk tcp receiver ポート。
raw	**false**	プロデューサー	Camel 2.16.0: ボディーを挿入する必要あり。**true** の場合、ボディーは Splunk に送信される前に **String** に変換されます。
initEarliestTime	**null**	コンシューマー	最初の検索の最初の開始オフセット。必須
earliestTime	**null**	コンシューマー	検索時間枠の初期時間。
latestTime	**null**	コンシューマー	検索時間枠の最新の時間。

count	0	コンシューマー	返すエンティティーの最大数を示す数字。これは、現在サポートされていない maxMessagesPerPoll とは異なることに注意してください。
search	null	コンシューマー	実行する Splunk クエリー
savedSearch	null	コンシューマー	実行する Splunk に保存されているクエリーの名前
streaming	false	コンシューマー	Camel 2.14.0 : すべてのエクスチェンジを 1 つのバッチで返すのではなく、Splunk から受信されるエクスチェンジをストリーミングします。これは、結果をより迅速に受信できるだけでなく、交換がコンポーネントでバッファーリングされないため、必要なメモリーが少なくなるという利点があります。
eventHost	null	プロデューサー	Camel 2.17: デフォルトの Splunk イベントホストフィールドを上書きします。

メッセージボディー

Splunk は、キー/値のペアでデータを操作します。SplunkEvent クラスはこのようなデータのプレースホルダーであり、プロデューサーのメッセージボディーにある必要があります。同様に、コンシューマーの検索結果ごとにボディーに返されます。

Camel 2.16.0 以降では、プロデューサーエンドポイントに raw オプションを設定して、生データを Splunk に送信できます。これは、Splunk がサポートされる JSon/XML およびその他のペイロードに役立ちます。

ユースケース

Twitter でミュージックのあるツイートを検索し、Splunk にイベントを公開

```
from("twitter://search?
```

```
type=polling&keywords=music&delay=10&consumerKey=abc&consumerSecret=def&accessT
oken=hij&accessTokenSecret=xxx")
        .convertBodyTo(SplunkEvent.class)
        .to("splunk://submit?username=foo&password=bar&index=camel-
tweets&sourceType=twitter&source=music-tweets");
```

Tweet を SplunkEvent に変換するには、以下のようなコンバーターを使用できます。

```
@Converter
public class Tweet2SplunkEvent {
   @Converter
   public static SplunkEvent convertTweet(Status status) {
      SplunkEvent data = new SplunkEvent("twitter-message", null);
      //data.addPair("source", status.getSource());
      data.addPair("from_user", status.getUser().getScreenName());
      data.addPair("in_reply_to", status.getInReplyToScreenName());
      data.addPair(SplunkEvent.COMMON_START_TIME, status.getCreatedAt());
      data.addPair(SplunkEvent.COMMON_EVENT_ID, status.getId());
      data.addPair("text", status.getText());
      data.addPair("retweet_count", status.getRetweetCount());
      if (status.getPlace() != null) {
         data.addPair("place_country", status.getPlace().getCountry());
         data.addPair("place_name", status.getPlace().getName());
         data.addPair("place_street", status.getPlace().getStreetAddress());
      }
      if (status.getGeoLocation() != null) {
         data.addPair("geo_latitude", status.getGeoLocation().getLatitude());
         data.addPair("geo_longitude", status.getGeoLocation().getLongitude());
      }
      return data;
   }
}
```

Splunk でツイートを検索する

```
    from("splunk://normal?username=foo&password=bar&initEarliestTime=-
2m&search=search index=camel-tweets sourcetype=twitter")
        .log("${body}");
```

その他のコメント

Splunk には、これを分析および表示するために事前にビルドされたアプリケーションでマシンが生成したデータを活用するためのさまざまなオプションがあります。たとえば、jmx app. を使用して jmx 属性（例：route および jvm メトリクス）を Splunk に公開し、これをダッシュボードで表示できます。

関連項目

- *Configuring Camel (Camel の設定)*
- *Component (コンポーネント)*
- *Endpoint (エンドポイント)*
- *スタートガイド*

第159章 SPRINGBATCH

SPRING BATCH コンポーネント

spring-batch: コンポーネントとサポートクラスは、Camel と Spring Batch インフラストラクチャー間の統合ブリッジを提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-spring-batch</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
spring-batch:jobName[?options]
```

jobName は、Camel レジストリーにある Spring Batch ジョブの名前を表します。

警告

このコンポーネントはプロデューサーエンドポイントの定義にのみ使用できます。つまり、from () ステートメントで Spring Batch コンポーネントを使用することはできません。

オプション

名前	デフォルト値	説明
jobLauncherRef	**null**	**非推奨で、Camel 3.0!Camel 2.10:**代わりに jobLauncher=#theName を使用してください。
jobLauncher	**null**	**Camel 2.11.1:** Camel レジストリーから使用する **JobLauncher** を明示的に指定します。

使用方法

Spring Batch コンポーネントがメッセージを受信すると、ジョブ実行がトリガーされます。ジョブは、以下のアルゴリズムに従って解決された org.springframework.batch.core.launch.JobLaucher インスタンスを使用して実行されます。

- ***JobLauncher がコンポーネントに手動で設定されている場合は、これを使用します。***
- ***jobLauncherRef オプションがコンポーネントに設定されている場合、Camel Registry で指定の名前で JobLauncher を検索します。非推奨で、Camel 3.0 で削除されます。***
- ***jobLauncher 名で Camel レジストリーに登録されている JobLauncher がある場合は、これを使用します。***
- ***上記の手順で JobLauncher を解決できず、Camel レジストリーに JobLauncher インスタンスが 1 つだけある場合は、これを使用します。***

メッセージで見つかったすべてのヘッダーは、ジョブパラメーターとして JobLauncher に渡されます。文字列、Long、Double、および java.util.Date の値は org.springframework.batch.core.JobParametersBuilder にコピーされます。他のデータ型は String に変換されます。

例

Spring Batch ジョブ実行をトリガーします。

```
from("direct:startBatch").to("spring-batch:myJob");
```

JobLauncher セットで Spring Batch ジョブの実行を明示的にトリガーします。

```
from("direct:startBatch").to("spring-batch:myJob?jobLauncherRef=myJobLauncher");
```

JobLauncher によって返される Camel 2.11.1JobExecution インスタンス以降、SpringBatchProducer によって出力メッセージに転送されます。JobExecution インスタンスを使用して、Spring Batch API を直接使用して一部の操作を実行できます。

```
from("direct:startBatch").to("spring-batch:myJob").to("mock:JobExecutions");
...
MockEndpoint mockEndpoint = ...;
JobExecution jobExecution =
mockEndpoint.getExchanges().get(0).getIn().getBody(JobExecution.class);
BatchStatus currentJobStatus = jobExecution.getStatus();
```

サポートクラス

Component とは別に、Camel Spring Batch は Spring Batch インフラストラクチャーへのフックに使用できるサポートクラスも提供します。

CAMELITEMREADER

CamelItemReader を使用すると、Camel インフラストラクチャーから直接バッチデータを読み取ることができます。

たとえば、以下のスニペットは、JMS キューからデータを読み取るように Spring Batch を設定します。

```
<bean id="camelReader"
class="org.apache.camel.component.spring.batch.support.CamelItemReader">
  <constructor-arg ref="consumerTemplate"/>
  <constructor-arg value="jms:dataQueue"/>
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="camelReader" writer="someWriter" commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

CAMELITEMWRITER

CamelItemWriter は CamelItemReader と同様に目的がありますが、処理されたデータのチャンクの書き込み専用です。

たとえば、以下のスニペットは、データを JMS キューに書き込むように Spring Batch を設定します。

```
<bean id="camelwriter"
class="org.apache.camel.component.spring.batch.support.CamelItemWriter">
  <constructor-arg ref="producerTemplate"/>
  <constructor-arg value="jms:dataQueue"/>
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="someReader" writer="camelwriter" commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

CAMELITEMPROCESSOR

CamelItemProcessor は Spring Batch org.springframework.batch.item.ItemProcessor インターフェイスの実装です。後者の実装は Request Reply パターン をリレーし、バッチ項目の処理を Camel インフラストラクチャーに委譲します。処理する項目は、メッセージのボディーとして Camel エンドポイントに送信されます。

たとえば、以下のスニペットは、Direct エンドポイント と Simple 式言語 を使用してバッチアイテムの簡単な処理を実行します。

```
<camel:camelContext>
  <camel:route>
    <camel:from uri="direct:processor"/>
    <camel:setExchangePattern pattern="InOut"/>
    <camel:setBody>
      <camel:simple>Processed ${body}</camel:simple>
    </camel:setBody>
  </camel:route>
</camel:camelContext>

<bean id="camelProcessor"
class="org.apache.camel.component.spring.batch.support.CamelItemProcessor">
  <constructor-arg ref="producerTemplate"/>
  <constructor-arg value="direct:processor"/>
```

```
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="someReader" writer="someWriter" processor="camelProcessor"
commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

CAMELJOBEXECUTIONLISTENER

CamelJobExecutionListener は、ジョブ実行イベントを Camel エンドポイントに送信する org.springframework.batch.core.JobExecutionListener インターフェイスの実装です。

Spring Batch によって生成された org.springframework.batch.core.JobExecution インスタンスは、メッセージのボディーとして送信されます。before- と after-callbacks SPRING_BATCH_JOB_EVENT_TYPE ヘッダーを区別するには、BEFORE または AFTER 値に設定されます。

以下のスニペット例は、Spring Batch ジョブ実行イベントを JMS キューに送信します。

```
<bean id="camelJobExecutionListener"
class="org.apache.camel.component.spring.batch.support.CamelJobExecutionListener">
  <constructor-arg ref="producerTemplate"/>
  <constructor-arg value="jms:batchEventsBus"/>
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="someReader" writer="someWriter" commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
  <batch:listeners>
    <batch:listener ref="camelJobExecutionListener"/>
  </batch:listeners>
</batch:job>
```

第160章 SPRINGINTEGRATION

SPRING INTEGRATION コンポーネント

***spring-integration:* コンポーネントは、Apache Camel コンポーネントが Spring *インテグレーションエンドポイント* と通信するためのブリッジを提供します。**

URI 形式

> ***spring-integration:defaultChannelName[?options]***

***defaultChannelName* は、Spring Integration Spring コンテキストによって使用されるデフォルトのチャネル名を表します。Spring Integration コンシューマーの *inputChannel* 名と Spring Integration プロバイダーの *outputChannel* 名と同じになります。**

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	説明	例	必須	デフォルト値
inputChannel	指定されたチャネル名が Spring コンテキストで定義される、このエンドポイントが使用する Spring インテグレーション入力チャネル名。	**inputChannel=requestChannel**	いいえ	
outputChannel	メッセージを Spring インテグレーションコンテキストに送信するために使用される Spring インテグレーション出力チャネル名。	**outputChannel=replyChannel**	いいえ	

inOut	Spring インテグレーションエンドポイントが使用する必要のある交換パターン。	**inOut=true**	いいえ	Spring インテグレーションコンシューマーの場合は InOnly および Spring インテグレーションプロバイダーの **outOnly**
consumer.delay	各ポーリングの遅延（ミリ秒単位）。	**consumer.delay=60000**	いいえ	**500**
consumer.initialDelay	ポーリングが開始するまでの時間（ミリ秒単位）。	**consumer.initialDelay=10000**	いいえ	**1000**
consumer.userFixedDelay	**true** を指定してポーリング間の固定遅延を使用します。使用しない場合は固定レートが使用されます。詳細は、ScheduledExecutorService クラスを参照してください。	**consumer.userFixedDelay=false**	いいえ	**false**

使用方法

Spring インテグレーションコンポーネントは、Spring インテグレーションの入力チャネルおよび出力チャネルを介して Apache Camel エンドポイントを Spring インテグレーションエンドポイントに接続するブリッジです。このコンポーネントを使用すると、Camel メッセージを Spring Integration エンドポイントに送信したり、Camel ルーティングコンテキストの Spring インテグレーションエンドポイントからメッセージを受信したりできます。

SPRING インテグレーションエンドポイントの使用

以下のように、URI を使用して Spring インテグレーションエンドポイントを設定できます。

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns:beans="http://www.springframework.org/schema/beans"
xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
  http://www.springframework.org/schema/integration/spring-integration.xsd
  http://camel.apache.org/schema/spring
  http://camel.apache.org/schema/spring/camel-spring.xsd">

<channel id="inputChannel"/>
   <channel id="outputChannel"/>
   <channel id="onewayChannel"/>

<service-activator input-channel="inputChannel"
          ref="helloService"
          method="sayHello"/>

<service-activator input-channel="onewayChannel"
          ref="helloService"
          method="greet"/>

<beans:bean id="helloService"
class="org.apache.camel.component.spring.integration.HelloWorldService"/>

   <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
     <route>
       <from uri="direct:twowayMessage"/>
       <!-- Using the &as the separator of & -->
       <to uri="spring-integration:inputChannel?inOut=true&nputChannel=outputChannel"/>
     </route>
     <route>
       <from uri="direct:onewayMessage"/>
       <to uri="spring-integration:onewayChannel?inOut=false"/>
     </route>
   </camelContext>
```

```
<channel id="requestChannel"/>
<channel id="responseChannel"/>

<beans:bean id="myProcessor"
class="org.apache.camel.component.spring.integration.MyProcessor"/>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <!-- Using the &as the separator of & -->
    <from uri="spring-integration://requestChannel?
outputChannel=responseChannel&nOut=true"/>
    <process ref="myProcessor"/>
  </route>
</camelContext>
```

または、Spring インテグレーションチャネル名を直接使用 します。

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:beans="http://www.springframework.org/schema/beans"
xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
  http://www.springframework.org/schema/integration/spring-integration.xsd
  http://camel.apache.org/schema/spring
  http://camel.apache.org/schema/spring/camel-spring.xsd">
<channel id="outputChannel"/>

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
      <!-- camel will create a spring integration endpoint automatically -->
      <from uri="outputChannel"/>
      <to uri="mock:result"/>
    </route>
  </camelContext>
```

SOURCE アダプターおよび TARGET アダプター

Spring インテグレーションは、Spring インテグレーションのソースおよびターゲットアダプターも提供します。これは、Spring インテグレーションチャネルから Apache Camel エンドポイント、または Apache Camel エンドポイントから Spring インテグレーションチャネルにメッセージをルーティングできます。

この例では、以下の namespace を使用します。

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
       xmlns:beans="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel-si="http://camel.apache.org/schema/spring/integration"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.xsd
       http://camel.apache.org/schema/spring/integration
       http://camel.apache.org/schema/spring/integration/camel-spring-integration.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd
    ">
```

以下のように、ソースまたはターゲットを Apache Camel エンドポイントにバインドできます。

```
<!-- Create the camel context here -->
<camelContext id="camelTargetContext" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:EndpointA" />
```

```
        <to uri="mock:result" />
    </route>
    <route>
        <from uri="direct:EndpointC"/>
        <process ref="myProcessor"/>
      </route>
</camelContext>

<!-- We can bind the camelTarget to the camel context's endpoint by specifying the
camelEndpointUri attribute -->
<camel-si:camelTarget id="camelTargetA" camelEndpointUri="direct:EndpointA"
expectReply="false">
    <camel-si:camelContextRef>camelTargetContext</camel-si:camelContextRef>
</camel-si:camelTarget>

<camel-si:camelTarget id="camelTargetB" camelEndpointUri="direct:EndpointC"
replyChannel="channelC" expectReply="true">
    <camel-si:camelContextRef>camelTargetContext</camel-si:camelContextRef>
</camel-si:camelTarget>

<camel-si:camelTarget id="camelTargetD" camelEndpointUri="direct:EndpointC"
expectReply="true">
    <camel-si:camelContextRef>camelTargetContext</camel-si:camelContextRef>
</camel-si:camelTarget>

<beans:bean id="myProcessor"
class="org.apache.camel.component.spring.integration.MyProcessor"/>
```

```
<!-- spring integration channels -->
<channel id="channelA"/>
<channel id="channelB"/>
<channel id="channelC"/>

<!-- spring integration service activator -->
<service-activator input-channel="channelB" output-channel="channelC" ref="helloService"
method="sayHello"/>

<!-- custom bean -->
<beans:bean id="helloService"
class="org.apache.camel.component.spring.integration.HelloWorldService"/>

<camelContext id="camelSourceContext" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:OneWay"/>
        <to uri="direct:EndpointB" />
    </route>
    <route>
        <from uri="direct:TwoWay"/>
        <to uri="direct:EndpointC" />
    </route>
</camelContext>

<!-- camelSource will redirect the message coming for direct:EndpointB to the spring
requestChannel channelA -->

<camel-si:camelSource id="camelSourceA" camelEndpointUri="direct:EndpointB"
```

```
                          requestChannel="channelA" expectReply="false">
    <camel-si:camelContextRef>camelSourceContext</camel-si:camelContextRef>
</camel-si:camelSource>

<!-- camelSource will redirect the message coming for direct:EndpointC to the spring
requestChannel channelB
 then it will pull the response from channelC and put the response message back to
direct:EndpointC -->

<camel-si:camelSource id="camelSourceB" camelEndpointUri="direct:EndpointC"
    requestChannel="channelB" replyChannel="channelC" expectReply="true">
    <camel-si:camelContextRef>camelSourceContext</camel-si:camelContextRef>
</camel-si:camelSource>
```

第161章 SPRING イベント

SPRING イベントコンポーネント

spring-event: コンポーネントは Spring ApplicationEvent オブジェクトへのアクセスを提供します。これにより、ApplicationEvent オブジェクトを Spring ApplicationContext に公開したり、消費したりできます。その後、エンタープライズ統合パターンを 使用して メッセージフィルターなどの処理を行うことができます。

URI 形式

```
spring-event://default[?options]
```

現時点では、このコンポーネントにはオプションがない点に注意してください。今後のリリースで簡単に変更できるため、再度確認してください。

第162章 SPRING LDAP

SPRING LDAP コンポーネント

Camel 2.11 以降で利用可能

spring-ldap: コンポーネントは、Spring LDAP の Camel ラッパーを提供します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-ldap</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
spring-ldap:springLdapTemplate[?options]
```

springLdapTemplate は Spring LDAP テンプレート Bean の名前です。この Bean では、LDAP アクセスの URL および認証情報を設定します。

オプション

名前	タイプ	説明

operation	文字列	実行する LDAP 操作。**検索、バインド、またはバインド 解除** のいずれかでなければなりません。
scope	文字列	検索操作の範囲。**オブジェクト**、1 次、または **サブツリー** の**いずれか** でなければなりません。http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#Search_and_Compare も参照してください。

使用方法

*コンポーネントはプロデューサーエンドポイントのみをサポートします。コンシューマーエンドポイントを作成しようとすると、**UnsupportedOperationException** が発生します。メッセージのボディーはマップ(**java.util.Map**のインスタンス)である必要があります。このマップには、**LDAP** 操作を実行するルートノードを指定するキー **dn** を持つエントリーが含まれる必要があります。マップの他のエントリーは操作固有です（以下を参照）。*

*メッセージのボディーには、バインドおよびバインド 解除 操作は変更されません。検索 操作では、ボディーは検索結果に設定されます。**http://static.springsource.org/spring-ldap/site/apidocs/org/springframework/ldap/core/LdapTemplate.html#search%28java.lang.String,%20java.lang.String,%20int,%20org.springframework.ldap.core.AttributesMapper%29** を参照してください。*

検索

*メッセージ本文には、キー フィルター を持つエントリーが必要です。値は、有効な **LDAP** フィルターを表す **String** である必要があります。**http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#Search_and_Compare** を参照してください。*

バインド

メッセージ本文には、キー 属性 を持つエントリーが必要です。値は javax.naming.directory.Attributes のインスタンスである必要があります。このエントリーは、作成される LDAP ノードを指定します。

UNBIND

追加のエントリーは必要ありません。指定された dn を持つノードが削除されます。

キー定義

エラーのスペルを避けるために、以下の定数が org.apache.camel.springldap.SpringLdapProducer で定義されます。

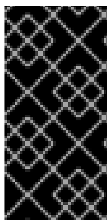
- *public static final String DN = "dn"*
- *public static final String FILTER = "filter"*
- *public static final String ATTRIBUTES = "attributes"*

第163章 SPRING REDIS

SPRING REDIS コンポーネント

Camel 2.11 から利用可能

このコンポーネントにより、Redis からのメッセージの送受信が可能になります。Redis は高度なキーと値のストアで、キーには文字列、ハッシュ、リスト、セット、およびソートされたセットを含めることができます。さらに、アプリケーション間通信に pub/sub 機能を提供します。Camel はコマンドを実行するためのプロデューサーを提供します。pub/sub メッセージをサブスクライブするコンシューマーは、重複メッセージをフィルターリングするためのべき等リポジトリーになります。



前提条件

このコンポーネントを使用するには、Redis サーバーが実行中である必要があります。

URI 形式

```
spring-redis://host:port[?options]
```

URI にクエリーオプションは ?options=value&option2=value&.. の形式で追加できます。

URI オプション

名前	デフォルト値	コンテキスト	説明
host	**null**	両方	Redis サーバーが実行されているホスト。
port	**null**	両方	Redis ポート番号。

command	**SET**	両方	デフォルトコマンド。メッセージヘッダーで上書きできます。
channels	**SET**	コンシューマー	サブスクライブするトピック名または名前パターンのリスト。
redisTemplate	**null**	プロデューサー	レジストリーに事前設定された **org.springframework.data.redis.core.RedisTemplate** インスタンスへの参照。
connectionFactory	**null**	両方	レジストリーの **org.springframework.data.redis.connection.RedisConnectionFactory** インスタンスへの参照。
listenerContainer	**null**	コンシューマー	レジストリーのレジストリーインスタンスで **org.springframework.data.redis.listener.RedisMessageListenerContainer** インスタンスへの参照。
serializer	**null**	コンシューマー	レジストリーの **org.springframework.data.redis.serializer.RedisSerializer** インスタンスへの参照。

使用方法

***REDIS** プロデューサーによって評価されるメッセージヘッダー*

プロデューサーはサーバーにコマンドを発行し、各コマンドには特定タイプの異なるパラメーターのセットがあります。コマンドの実行の結果はメッセージボディーで返されます。

ハッシュコマンド	説明	パラメーター	結果

HSET	ハッシュフィールドの文字列値を設定します。	CamelRedis.Key (文字列), CamelRedis.Field (文字列), CamelRedis.Value (オブジェクト型)	void
HGET	ハッシュフィールドの値を取得します。	CamelRedis.Key (文字列), CamelRedis.Field (文字列)	文字列
HSETNX	フィールドが存在しない場合にのみ、ハッシュフィールドの値を設定します。	CamelRedis.Key (文字列), CamelRedis.Field (文字列), CamelRedis.Value (オブジェクト型)	void
HMSET	複数のハッシュフィールドを複数の値に設定します。	CamelRedis.Key (String), CamelRedis.Values(Map <String, Object>)	void
HMGET	指定されたハッシュフィールドの値を取得します。	CamelRedis.Key (文字列), CamelRedis.Fields (Collection<String>)	コレクション<Object>
HINCRBY	ハッシュフィールドの整数値を指定の数だけ増やします。	CamelRedis.Key (文字列), CamelRedis.Field (文字列), CamelRedis.Value(Long 型)	Long
HEXISTS	ハッシュフィールドが存在するかどうかを判断する	CamelRedis.Key (文字列), CamelRedis.Field (文字列)	ブール値
HDEL	1 つ以上のハッシュフィールドを削除します。	CamelRedis.Key (文字列), CamelRedis.Field (文字列)	void
HLEN	ハッシュ内のフィールド数を取得します。	CamelRedis.Key (文字列)	Long
HKEYS	ハッシュ内のすべてのフィールドを取得します。	CamelRedis.Key (文字列)	Set<String>
HVALS	ハッシュのすべての値を取得します。	CamelRedis.Key (文字列)	コレクション<Object>

HGETALL	ハッシュのすべてのフィールドおよび値を取得します。	CamelRedis.Key (文字列)	Map<String, Object>

list commands	説明	パラメーター	結果
RPUSH	1 つまたは複数の値をリストに追加します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	Long
RPUSHX	リストが存在する場合に限り、リストに値を追加します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	Long
LPUSH	1 つまたは複数の値をリストに追加します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	Long
LLEN	リストの長さを取得します。	CamelRedis.Key (文字列)	Long
LRANGE	リストからの要素の範囲を取得します。	CamelRedis.Key (文字列), CamelRedis.Start(Long 型), CamelRedis.End(Long 型)	List<Object>
LTRIM	リストを指定の範囲にトリミングする	CamelRedis.Key (文字列), CamelRedis.Start(Long 型), CamelRedis.End(Long 型)	void
LINDEX	インデックスでリストから要素を取得します。	CamelRedis.Key (文字列), CamelRedis.Index(Long 型)	文字列
LINSERT	リスト内の別の要素の前後に要素を挿入	CamelRedis.Key （文字列）、CamelRedis.Value (Object)、CamelRedis.Pivot (String)、CamelRedis.Position (String)	Long

LSET	リスト内の要素の値をインデックスで設定します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型), CamelRedis.Index(Long 型)	void
LREM	リストからの要素の削除	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型), CamelRedis.Count(Long 型)	Long
LPOP	リストの最初の要素を削除して取得します。	CamelRedis.Key (文字列)	オブジェクト
RPOP	リストの最後の要素を削除して取得します。	CamelRedis.Key (文字列)	文字列
RPOPLPUSH	リストの最後の要素を削除し、別のリストに追記して返します。	CamelRedis.Key (文字列), CamelRedis.Destination (文字列)	オブジェクト
BRPOPLPUSH	リストから値をポップし、その値を別のリストにプッシュして返します。または 1 つが利用可能になるまでブロックします。	CamelRedis.Key (文字列), CamelRedis.Destination (文字列), CamelRedis.Timeout(Long 型)	オブジェクト
BLPOP	リストの最初の要素を削除して取得するか、以下が利用可能になるまでブロックします。	CamelRedis.Key (文字列), CamelRedis.Timeout(Long 型)	オブジェクト
BRPOP	リストの最後の要素を削除して取得するか、以下が利用可能になるまでブロックします。	CamelRedis.Key (文字列), CamelRedis.Timeout(Long 型)	文字列

コマンドの設定	説明	パラメーター	結果
SADD	セットに 1 つ以上のメンバーを追加します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	ブール値
SMEMBERS	セット内のすべてのメンバーを取得します。	CamelRedis.Key (文字列)	Set<Object>

SREM	セットから1つ以上のメンバーを削除します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	ブール値
SPOP	セットからランダムなメンバーを削除して返します。	CamelRedis.Key (文字列)	文字列
SMOVE	あるセットから別のセットにメンバーを移動します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型), CamelRedis.Destination (文字列)	ブール値
SCARD	セット内のメンバー数を取得します。	CamelRedis.Key (文字列)	Long
SISMEMBER	指定された値がセットのメンバーであるかどうかを決定する	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	ブール値
SINTER	複数セットの補間	CamelRedis.Key (文字列), CamelRedis.Keys (文字列)	Set<Object>
SINTERSTORE	複数のセットに干渉し、作成されるセットをキーに保存します。	CamelRedis.Key (文字列), CamelRedis.Keys (文字列), CamelRedis.Destination (文字列)	void
SUNION	複数のセットの追加	CamelRedis.Key (文字列), CamelRedis.Keys (文字列)	Set<Object>
SUNIONSTORE	複数のセットを追加し、作成される をキーに保存します。	CamelRedis.Key (文字列), CamelRedis.Keys (文字列), CamelRedis.Destination (文字列)	void
SDIFF	複数のセットを減算	CamelRedis.Key (文字列), CamelRedis.Keys (文字列)	Set<Object>
SDIFFSTORE	複数のセットを引き、作成される をキーに保存します。	CamelRedis.Key (文字列), CamelRedis.Keys (文字列), CamelRedis.Destination (文字列)	void

SRANDMEMBER	セットから1つまたは複数のランダムメンバーを取得します。	CamelRedis.Key (文字列)	文字列

順序付きの設定コマンド	説明	パラメーター	結果
ZADD	ソートされたセットにメンバーを1つ以上追加するか、スコアが存在する場合は更新します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型), CamelRedis.Score (Double 型)	ブール値
ZRANGE	インデックスごとにソートされたセット内のメンバーの範囲を返します。	CamelRedis.Key （文字列）、CamelRedis.Start (Long)、CamelRedis.End (Long)、CamelRedis.WithScore (Boolean)	オブジェクト
ZREM	ソートされたセットから1つ以上のメンバーを削除します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	ブール値
ZINCRBY	ソートされたセット内のメンバーのスコアを増やす	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型), CamelRedis.Increment (Double 型)	double
ZRANK	ソートされたセットでメンバーのインデックスを決定します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	Long
ZREVRANK	ソートされたセットでメンバーのインデックスを決定します。スコアは high から low に付けられます。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	Long
ZREVRANGE	ソートされたセット内のメンバーの範囲を、インデックス別に返します。スコアは high から low に付けられます。	CamelRedis.Key （文字列）、CamelRedis.Start (Long)、CamelRedis.End (Long)、CamelRedis.WithScore (Boolean)	オブジェクト

ZCARD	ソートされたセット内のメンバー数を取得します。	CamelRedis.Key (文字列),	Long
ZCOUNT	指定の値内のスコアでソートされたセット内のメンバーをカウントします。	CamelRedis.Key (文字列), CamelRedis.Min (Double 型), CamelRedis.Max (Double 型)	Long
ZRANGEBYSCORE	スコアでソートされたセット内のメンバーの範囲を返します。	CamelRedis.Key (文字列), CamelRedis.Min (Double 型), CamelRedis.Max (Double 型)	Set<Object>
ZREVRANGEBYSCORE	スコアが high から low のソートされたセット内のメンバーの範囲を返します。	CamelRedis.Key (文字列), CamelRedis.Min (Double 型), CamelRedis.Max (Double 型)	Set<Object>
ZREMRANGEBYRANK	指定されたインデックス内でソートされたセットのすべてのメンバーを削除します。	CamelRedis.Key (文字列), CamelRedis.Start(Long 型), CamelRedis.End(Long 型)	void
ZREMRANGEBYSCORE	指定のスコア内でソートされたセット内のすべてのメンバーを削除します。	CamelRedis.Key (文字列), CamelRedis.Start(Long 型), CamelRedis.End(Long 型)	void
ZUNIONSTORE	複数のソートされたセットを追加し、作成されるソートされたセットを新しいキーに保存します。	CamelRedis.Key (文字列), CamelRedis.Keys (文字列), CamelRedis.Destination (文字列)	void
ZINTERSTORE	複数のソートされたセットを補間し、作成されるソートされたセットを新しいキーに保存します。	CamelRedis.Key (文字列), CamelRedis.Keys (文字列), CamelRedis.Destination (文字列)	void

string commands	説明	パラメーター	結果

SET	キーの文字列値を設定します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	void
GET	キーの値を取得します。	CamelRedis.Key (文字列)	オブジェクト
STRLEN	キーに保存されている値の長さを取得します。	CamelRedis.Key (文字列)	Long
APPEND	キーに値を追加します。	CamelRedis.Key (文字列), CamelRedis.Value (文字列)	整数
SETBIT	キーに保存される文字列値のオフセットでビットを設定または消去します。	CamelRedis.Key (文字列), CamelRedis.Offset(Long 型), CamelRedis.Value (ブール型)	void
GETBIT	キーに保存される文字列値のオフセットでビット値を返します。	CamelRedis.Key (文字列), CamelRedis.Offset(Long 型)	ブール値
SETRANGE	指定されたオフセットから開始するキーの文字列の一部を上書きします。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型), CamelRedis.Offset(Long 型)	void
GETRANGE	キーに保存されている文字列のサブ文字列を取得します。	CamelRedis.Key (文字列), CamelRedis.Start(Long 型), CamelRedis.End(Long 型)	文字列
SETNX	キーが存在しない場合にのみ、キーの値を設定します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	ブール値
SETEX	キーの値および有効期限を設定します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型), CamelRedis.Timeout(Long 型), SECONDS	void

DECRBY	指定された数でキーの整数値をデクリメントする	CamelRedis.Key (文字列), CamelRedis.Value(Long 型)	Long
DECR	キーの整数値を 1 つずつ減らす	CamelRedis.Key (文字列),	Long
INCRBY	指定された量でキーの整数値を増やします。	CamelRedis.Key (文字列), CamelRedis.Value(Long 型)	Long
INCR	キーの整数値を 1 つずつ増やします。	CamelRedis.Key (文字列)	Long
MGET	指定したすべてのキーの値を取得します。	CamelRedis.Fields (Collection<String>)	List<Object>
MSET	複数のキーに複数の値を設定する	CamelRedis.Values(Map <String, Object>)	void
MSETNX	複数のキーを複数値に設定します（キーが存在しない場合のみ）。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	void
GETSET	キーの文字列値を設定し、その古い値を返します。	CamelRedis.Key (文字列), CamelRedis.Value (オブジェクト型)	オブジェクト

主要なコマンド	説明	パラメーター	結果
EXISTS	キーが存在するかどうかを判断する	CamelRedis.Key (文字列)	ボルア語
DEL	キーを削除します。	CamelRedis.Keys (String)	void
TYPE	キーに保存されているタイプを特定します。	CamelRedis.Key (文字列)	DataType
KEYS	指定したパターンに一致するすべてのキーを検索します。	CamelRedis.Pattern (String)	Collection<String>
RANDOMKEY	キースペースからランダムなキーを返します。	CamelRedis.Pattern（文字列）、CamelRedis.Value （文字列）	文字列

RENAME	キーの名前変更	CamelRedis.Key (文字列)	void
RENAMENX	キーの名前変更（新しいキーが存在しない場合のみ）	CamelRedis.Key (文字列), CamelRedis.Value (文字列)	ブール値
EXPIRE	キーの時間を秒単位で設定する	CamelRedis.Key (文字列), CamelRedis.Timeout(Long 型)	ブール値
SORT	リスト、設定、またはソートされたセット内の要素を並べ替えます。	CamelRedis.Key (文字列)	List<Object>
PERSIST	キーから有効期限を削除します。	CamelRedis.Key (文字列)	ブール値
EXPIREAT	キーの有効期限を UNIX タイムスタンプとして設定します。	CamelRedis.Key (文字列), CamelRedis.Timestamp(Long 型)	ブール値
PEXPIRE	キーの時間をミリ秒単位で設定します。	CamelRedis.Key (文字列), CamelRedis.Timeout(Long 型)	ブール値
PEXPIREAT	キーの有効期限をミリ秒単位で指定された UNIX タイムスタンプとして設定します。	CamelRedis.Key (文字列), CamelRedis.Timestamp(Long 型)	ブール値
TTL	キーの有効期間を取得します。	CamelRedis.Key (文字列)	Long
MOVE	キーを別のデータベースに移動する	CamelRedis.Key (文字列), CamelRedis.Db (整数)	ブール値

その他のコマンド	説明	パラメーター	結果
MULTI	トランザクションブロックの開始をマークします。	none	void

DISCARD	MULTI の後に発行されたすべてのコマンドを破棄します。	none	void
EXEC	MULTI の後に発行されたすべてのコマンドを実行します。	none	void
WATCH	MULTI/EXEC ブロックの実行を決定するために指定のキーを監視します。	CamelRedis.Keys (String)	void
UNWATCH	監視されるすべてのキーについて忘れる	none	void
ECHO	指定の文字列を echo します。	CamelRedis.Value (String)	文字列
PING	サーバーに ping します。	none	文字列
QUIT	接続を閉じる	none	void
PUBLISH	メッセージをチャネルに投稿	CamelRedis.Channel (文字列), CamelRedis.Message (オブジェクト型)	void

REDIS コンシューマー

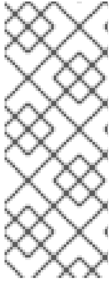
コンシューマーは、SUBSCRIBE コマンドを使用してチャネル名または P SUBSCRIBE コマンドを使用して文字列パターンでチャネルにサブスクライブします。PUBLISH コマンドを使用してチャネルにメッセージを送信すると、メッセージが消費され、メッセージは Camel メッセージボディーとして利用できます。メッセージは、設定されたシリアライザーまたはデフォルトの JdkSerializationRedisSerializer を使用してシリアライズされます。

コンシューマーによって設定されたメッセージヘッダー

ヘッダー	タイプ	説明
CamelRedis.Channel	**文字列**	メッセージが受信されたチャネル名。
CamelRedis.Pattern	**文字列**	メッセージが受信されたチャネルに一致するパターン。

DEPENDENCIES

Maven ユーザーは、以下の依存関係を pom.xml に追加する必要があります。



POM.XML

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-redis</artifactId>
  <version>${camel-version}</version>
</dependency>
```

${camel-version} は、実際のバージョンの Camel (2.11 以降)に置き換える必要があります。

第164章 SPRING WEB SERVICES

SPRING WEB SERVICES コンポーネント

spring-ws: コンポーネントを使用すると、Spring Web Services と統合できます。Web サービスへのアクセス、および独自のコントラクトファースト Web サービスを作成するためのサーバー側のサポートの両方を提供します。

> **DEPENDENCIES**
>
> Camel 2.8 以降、このコンポーネントは Spring-WS 2.0.x に同梱されており（残りの Camel のように）、Spring 3.0.x が必要です。
>
> 以前のバージョンの Camel には Spring 2.5.x および 3.0.x との互換性がある Spring-WS 1.5.9 が同梱されていました。Spring 2.5.x で以前のバージョンの camel-spring-ws を実行するには、Spring 2.5.x から spring-webmvc モジュールを追加する必要があります。Spring 3.0.x で Spring-WS 1.5.9 を実行するには、このモジュールが Spring-WS 1.5.9 にも含まれているため、OXM モジュールを Spring 3.0.x から除外する必要があります（を参照）。http://stackoverflow.com/questions/3313314/can-spring-ws-1-5-be-used-with-spring-3

URI 形式

このコンポーネントの URI スキームは以下のとおりです。

```
spring-ws:[mapping-type:]address[?options]
```

Web サービスを公開するには、mapping-type を次のいずれかの値に設定する必要があります。

マッピングタイプ	説明
rootqname	メッセージに含まれるルート要素の修飾名に基づいて Web サービスリクエストをマップするオプションを提供します。
soapaction	メッセージのヘッダーに指定された SOAP アクションに基づいて Web サービスリクエストをマップするために使用されます。
uri	特定の URI をターゲットとする Web サービスリクエストをマップします。

xpathresult	XPath **式** の評価に基づいて Web サービスリクエストを受信メッセージに対してマッピングするために使用されます。評価の結果は、エンドポイント URI に指定された XPath 結果と一致する必要があります。
beanname	**PayloadRootQNameEndpointMapping、SonapActionEndpointMapping などの既存の（レガシー） エンドポイントマッピング と統合するために、org.apache.camel.component.spring.ws.bean.CamelEndpoint Dispatcher** を参照できます。

コンシューマーの場合、address には、指定された mapping-type (SOAP アクション、XPath 式など)に関連する値が含まれている必要があります。プロデューサーとして、アドレスを呼び出している Web サービスの URI に設定する必要があります。

URI にクエリーオプションを追加するには、?option=value&option=value&.. の形式を使用します。

オプション

名前	必須？	説明
soapAction	いいえ	リモート Web サービスにアクセスするときに SOAP リクエスト内に含める SOAP アクション
wsAddressingAction	いいえ	Web サービスへのアクセス時に含む WS-Addressing 1.0 アクションヘッダー。**To** ヘッダーは、エンドポイント URI で指定される Web サービスの **アドレス** に設定されます（デフォルトでは Spring-WS の動作）。
outputAction	いいえ	メソッドによって提供される応答 WS-Addressing Action ヘッダーの値を示します。

faultAction	いいえ	メソッドによって提供される **faultAction** 応答 WS-Addressing Fault Action ヘッダーの値を示します。
faultTo	いいえ	メソッドによって提供される **faultAction** 応答 WS-Addressing FaultTo ヘッダーの値を示します。
replyTo	いいえ	メソッドによって提供される **replyTo** 応答 WS-Addressing ReplyTo ヘッダーの値を示します。
expression	mapping-type が **xpathresult**の場合のみ	Web サービス要求のマッピングプロセスで使用する XPath 式。**xpathresult**で指定された結果と一致する必要があります。
timeout	いいえ	**Camel 2.10:** プロデューサーを使用して Web サービスを呼び出す際にソケットの読み取りタイムアウト（ミリ秒単位）を設定します。 URLConnection.setReadTimeout （）および CommonsHttpMessageSender.setReadTimeout （）を参照してください。組み込みメッセージの送信者実装を使用する場合は、オプションが機能する： **CommonsHttpMessageSender** nd **HttpUrlConnectionMessageSender**。コンポーネントに指定された Spring WS 設定オプションをカスタマイズしない限り、これらの実装はデフォルトで HTTP ベースのサービスに使用されます。標準以外の送信者を使用していること。 独自のタイムアウト設定を処理することを前提とします。**Camel 2.12:** We built-in message sender **HttpComponentsMessageSender** s considered **instead of CommonsHttpMessageSender** hich has been deprecated, see HttpComponentsMessageSender.setReadTimeout （）を参照してください。

sslContextParameters	いいえ	**Camel 2.10:** Registry の **org.apache.camel.util.jsse.SSLContextParameters** オブジェクトへの参照。 Using the JSSE Configuration Utility を参照してください。 このオプションは、組み込みメッセージ送信者の実装 (**CommonsHttpMessageSender** および **HttpUrlConnectionMessageSender**)を使用する場合に機能します。 これらの実装の 1 つは、コンポーネントに指定された Spring WS 設定オプションをカスタマイズしない限り、デフォルトで HTTP ベースのサービスに使用されます。 標準以外の送信者を使用している場合は、独自の TLS 設定を処理することを前提とします。 Security Guide の Configuring Transport Security for Camel Components の章を参照してください。 **Camel 2.12:** 組み込みメッセージの送信者 **HttpComponentsMessageSender** は、非推奨となった**CommonsHttpMessageSender の代わり** に考慮されます。
webServiceTemplate	いいえ	カスタム WebServiceTemplate を提供するオプション。これにより、カスタムインターセプターの追加やフォールトリゾルバー、メッセージ送信者、メッセージファクトリーの指定など、クライアント側の Web サービス処理を完全に制御できます。
messageSender	いいえ	カスタム WebServiceMessageSender を提供するオプション。たとえば、認証を実行したり、代替トランスポートを使用したりするには、以下を実行します。
messageFactory	いいえ	カスタム WebServiceMessageFactory を提供するオプション。たとえば、Apache Axiom が SAAJ ではなく Web サービスメッセージを処理する場合などです。

endpointMappingKey	いいえ	**org.apache.camel.component.spring.ws.type.EndpointMappingKey** のインスタンスへの参照。
endpointMapping	mapping-type が **rootqname**、**soapaction**、**uri**、または **xpathresult**の場合のみ	Registry/ApplicationContext の **org.apache.camel.component.spring.ws.bean.CamelEndpointMapping** への参照。すべての Camel/Spring-WS エンドポイントを提供するには、レジストリーに 1 つの Bean のみが必要です。この Bean は MessageDispatcher によって自動検出され、エンドポイントに指定された特性（ルート QName、SOAP アクションなど）に基づいてリクエストを Camel エンドポイントにマッピングするために使用されます。
endpointDispatcher	いいえ	Spring **org.springframework.ws.server.endpoint.MessageEndpoint**: Spring-WS によって受信されたメッセージを Camel エンドポイントにディスパッチし、**PayloadRootQNameEndpointMapping**、**SoapActionEndpointMapping** などの既存の（レガシー）エンドポイントマッピングと統合します。
messageFilter	いいえ	2.10.3 以降カスタム MessageFilter を提供するオプション。たとえば、ヘッダーまたは添付ファイルを独自に処理する場合などです。
messageIdStrategy	いいえ	一意のメッセージ ID の生成を制御するカスタム **MessageIdStrategy**。
webServiceEndpointUri	いいえ	プロデューサーに使用するデフォルトの Web サービスエンドポイント URI。

メッセージヘッダー

名前	タイプ	説明
CamelSpringWebserviceEndpointUri	**String**	クライアントとしてアクセスする Web サービスの URI。エンドポイント URI の **アドレス** 部分を上書きします。
CamelSpringWebserviceSoapAction	**String**	メッセージの SOAP アクションを指定するヘッダー。**soapAction** オプションが存在する場合はそれを上書きします。
CamelSpringWebserviceSoapHeader	Source	**Camel 2.11.1:** このヘッダーを使用してメッセージの SOAP ヘッダーを指定/アクセスします。
CamelSpringWebserviceAddressingAction	**URI**	このヘッダーを使用してメッセージの WS-Addressing アクションを指定します。**wsAddressingAction** オプションが存在する場合はオーバーライドします。
CamelSpringWebserviceAddressingFaultTo	**URI**	このヘッダーを使用して WS-Addressing FaultTo を指定します。存在する場合、**faultTo** オプションを上書きします。
CamelSpringWebserviceAddressingReplyTo	**URI**	このヘッダーを使用して WS-Addressing ReplyTo を指定します。存在する場合、**replyTo** オプションを上書きします。
CamelSpringWebserviceAddressingOutputAction	**URI**	このヘッダーを使用して WS-Addressing Action を指定し、**outputAction** オプションを上書きします（存在する場合）。
CamelSpringWebserviceAddressingFaultAction	**URI**	このヘッダーを使用して WS-Addressing Fault Action を指定します。存在する場合、**faultAction** オプションを上書きします。

WEB サービスへのアクセス

Web サービス を呼び出すには、単にルートを定義します。

```
from("direct:example").to("spring-ws:http://foo.com/bar")
```

メッセージを送信し、以下を実行します。

```
template.requestBody("direct:example", "<foobar xmlns=\"http://foo.com\"><msg>test
message</msg></foobar>");
```

呼び出す SOAP サービスの場合は、SOAP タグを含める必要はありません。Spring-WS は XML から SOAP マーシャリングを実行します。

SOAP および WS-ADDRESSING アクションヘッダーの送信

リモート Web サービスで SOAP アクションが必要な場合、または WS-Addressing 標準を使用する場合は、以下のようにルートを定義します。

```
from("direct:example")
.to("spring-ws:http://foo.com/bar?
soapAction=http://foo.com&wsAddressingAction=http://bar.com")
```

オプションで、エンドポイントオプションをヘッダー値で上書きできます。

```
template.requestBodyAndHeader("direct:example",
"<foobar xmlns=\"http://foo.com\"><msg>test message</msg></foobar>",
SpringWebserviceConstants.SPRING_WS_SOAP_ACTION, "http://baz.com");
```

SOAP ヘッダーの使用

Camel 2.11.1 以降で利用可能

String の以下の SOAP ヘッダーにメッセージを送信するときに、SOAP ヘッダーを Camel Message ヘッダーとして提供できます。

```
String body = ...
String soapHeader = "<h:Header xmlns:h=\"http://www.webserviceX.NET/\">
<h:MessageID>1234567890</h:MessageID><h:Nested><h:NestedID>1111</h:NestedID>
</h:Nested></h:Header>";
```

以下のように、Camel Message にボディーとヘッダーを設定できます。

```
exchange.getIn().setBody(body);
exchange.getIn().setHeader(SpringWebserviceConstants.SPRING_WS_SOAP_HEADER,
soapHeader);
```

その後、エクスチェンジを spring-ws エンドポイントに送信し、Web サービスを呼び出します。

同様に、spring-ws コンシューマーは SOAP ヘッダーで Camel Message も補完します。

この ユニットテスト の例は、を参照してください。

ヘッダーおよび添付の伝播

Spring WS Camel は、バージョン 2.10.3 以降の Spring-WS WebServiceMessage 応答へのヘッダーおよび添付の伝搬をサポートします。エンドポイントは、"hook" と呼ばれるものを使用します。MessageFilter （デフォルトは BasicMessageFilter）を使用して、交換ヘッダーと添付ファイルを WebServiceMessage 応答に伝播します。これで

```
exchange.getOut().getHeaders().put("myCustom","myHeaderValue")
exchange.getIn().addAttachment("myAttachment", new DataHandler(...))
```

注記：パイプラインのエクスチェンジヘッダーにテキストが含まれる場合、soap ヘッダーに Qname (key)=value 属性が生成されます。QName クラスを直接作成し、任意のキーをヘッダーに配置することが推奨されます。

MTOM 添付ファイルの使用方法

BasicMessageFilter は、MTOM メッセージを生成するために Apache Axiom に必要なすべての情報を提供します。Apache Axiom 内で Apache Camel Spring WS を使用する場合は、1 の例を示します。messageFactory を bellow として定義し、Spring-WS は MTOM ストラテジーを使用して、最適化された添付ファイルで SOAP メッセージを設定します。

```
<bean id="axiomMessageFactory"
class="org.springframework.ws.soap.axiom.AxiomSoapMessageFactory">
<property name="payloadCaching" value="false" />
<property name="attachmentCaching" value="true" />
<property name="attachmentCacheThreshold" value="1024" />
</bean>
```

2.以下の依存関係を pom.xml に追加します。

```
<dependency>
<groupId>org.apache.ws.commons.axiom</groupId>
<artifactId>axiom-api</artifactId>
<version>1.2.13</version>
</dependency>
<dependency>
<groupId>org.apache.ws.commons.axiom</groupId>
<artifactId>axiom-impl</artifactId>
<version>1.2.13</version>
<scope>runtime</scope>
</dependency>
```

3.Processor 実装などを使用して、添付をパイプラインに追加します。

```
private class Attachement implements Processor {
public void process(Exchange exchange) throws Exception
{ exchange.getOut().copyFrom(exchange.getIn()); File file = new File("testAttachment.txt");
exchange.getOut().addAttachment("test", new DataHandler(new FileDataSource(file)));  }
}
```

4.以下のように、エンドポイント(producer)を ussual として定義します。

```
from("direct:send")
.process(new Attachement())
.to("spring-ws:http://localhost:8089/mySoapService?
soapAction=mySoap&messageFactory=axiomMessageFactory");
```

5.これで、プロデューサーは otpmized attachments で MTOM メッセージを生成します。

カスタムヘッダーおよび添付フィルター

ヘッダーまたは添付のカスタム処理を提供する必要がある場合は、既存の BasicMessageFilter を拡

張し、適切なメソッドを上書きするか、MessageFilter インターフェイスの完全に新しい実装を作成します。カスタムフィルターを使用するには、これを Spring コンテキストに追加します。以下のようにグローバル a またはローカルメッセージフィルターのいずれかを指定できます。a）すべての Spring-WS エンドポイントのグローバル設定を提供するグローバルカスタムフィルターを指定します。

```
<bean id="messageFilter" class="your.domain.myMessageFiler" scope="singleton" />
```

または b）ローカルの messageFilter をエンドポイント上で以下のように直接実行します。

```
to("spring-ws:http://yourdomain.com?messageFilter=#myEndpointSpecificMessageFilter");
```

詳細は CAMEL-5724を参照してください。

独自の MessageFilter を作成する場合は、BasicMessageFilter クラスの MessageFilter のデフォルト実装で以下のメソッドを上書きすることを検討してください。

```
protected void doProcessSoapHeader(Message inOrOut, SoapMessage soapMessage)
{your code /*no need to call super*/ }
protected void doProcessSoapAttachements(Message inOrOut, SoapMessage response)
{ your code /*no need to call super*/ }
```

カスタム MESSAGESENDER および MESSAGEFACTORY の使用

レジストリー内のカスタムメッセージの送信者またはファクトリーは、以下のように参照できます。

```
from("direct:example")
.to("spring-ws:http://foo.com/bar?
messageFactory=#messageFactory&messageSender=#messageSender")
```

Spring の設定：

```
<!-- authenticate using HTTP Basic Authentication -->
<bean id="messageSender"
class="org.springframework.ws.transport.http.HttpComponentsMessageSender">
 <property name="credentials">
  <bean class="org.apache.commons.httpclient.UsernamePasswordCredentials">
   <constructor-arg index="0" value="admin"/>
   <constructor-arg index="1" value="secret"/>
  </bean>
 </property>
</bean>
```

```
<!-- force use of Sun SAAJ implementation, http://static.springsource.org/spring-
ws/sites/1.5/faq.html#saaj-jboss -->
<bean id="messageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">
 <property name="messageFactory">
  <bean class="com.sun.xml.messaging.saaj.soap.ver1_1.SOAPMessageFactory1_1Impl">
</bean>
 </property>
</bean>
```

WEB サービスの公開

このコンポーネントを使用して Web サービスを公開するには、最初に MessageDispatcher を設定して Spring XML ファイルでエンドポイントマッピングを検索する必要があります。サーブレットコンテナー内で実行を計画する場合は、web.xml で設定した MessageDispatcherServlet を使用する必要があります。

デフォルトでは、MessageDispatcherServlet は /WEB-INF/spring-ws-servlet.xml という名前の Spring XML を検索します。Spring-WS で Camel を使用するには、その XML ファイルの唯一の必須 Bean は CamelEndpointMapping です。この Bean により、MessageDispatcher は Web サービスリクエストをルートにディスパッチできます。

web.xml

```
<web-app>
    <servlet>
        <servlet-name>spring-ws</servlet-name>
        <servlet-
class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

spring-ws-servlet.xml

```
<bean id="endpointMapping"
class="org.apache.camel.component.spring.ws.bean.CamelEndpointMapping" />

<bean id="wsdl" class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition">
   <property name="schema">
      <bean class="org.springframework.xml.xsd.SimpleXsdSchema">
         <property name="xsd" value="/WEB-INF/foobar.xsd"/>
```

```
    </bean>
  </property>
  <property name="portTypeName" value="FooBar"/>
  <property name="locationUri" value="/"/>
  <property name="targetNamespace" value="http://example.com/"/>
</bean>
```

Spring-WS の設定に関する詳細は Writing Contract-First Web Services を参照してください。

ルートでのエンドポイントマッピング

XML 設定のインプレースでは、Camel の DSL を使用して、エンドポイントによって処理される Web サービスリクエストを定義できるようになりました。以下のルートは、 http://example.com/ namespace 内に GetFoo という名前のルート要素を持つすべての Web サービス要求を受信します。

```
from("spring-ws:rootqname:{http://example.com/}GetFoo?
endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

以下のルートは、http://example.com/GetFoo SOAP アクションを含む Web サービスリクエストを受信します。

```
from("spring-ws:soapaction:http://example.com/GetFoo?
endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

以下のルートは、http://example.com/foobar に送信されたすべての要求を受信します。

```
from("spring-ws:uri:http://example.com/foobar?endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

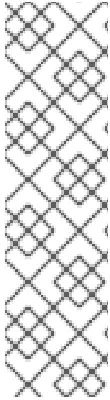
以下のルートは、メッセージ内の任意の場所（およびデフォルトの namespace）の < foobar>abc </foobar> 要素が含まれる要求を受信します。

```
from("spring-ws:xpathresult:abc?
expression=//foobar&endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

既存のエンドポイントマッピングを使用した代替設定

mapping-type beanname を持つすべてのエンドポイントに対して、対応する名前を持つ CamelEndpointDispatcher タイプの 1 つの Bean が Registry/ApplicationContext に必要です。この

Bean は、Camel エンドポイントと PayloadRootQNameEndpointMapping などの既存の エンドポイントマッピング との間のブリッジとして機能します。



注記

beanname マッピングタイプの使用は、主に Spring-WS を使用し、Spring XML ファイルで定義されたエンドポイントマッピングを持つ（レガシー）状況を対象としています。beanname mapping-type を使用すると、Camel ルートを既存のエンドポイントマッピングに接続できます。ゼロから始めている場合は、エンドポイントマッピングを Camel URI の（上記のように endpointMapping で説明）として定義することが推奨されます。これは設定が少なく、より表現的です。または、アノテーションを使用して vanilla Spring-WS を使用することもできます。

beanname を使用したルートの例：

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
 <route>
  <from uri="spring-ws:beanname:QuoteEndpointDispatcher" />
  <to uri="mock:example" />
 </route>
</camelContext>

<bean id="legacyEndpointMapping"
class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMappi
ng">
    <property name="mappings">
        <props>
            <prop key="{http://example.com/}GetFuture">FutureEndpointDispatcher</prop>
            <prop key="{http://example.com/}GetQuote">QuoteEndpointDispatcher</prop>
        </props>
    </property>
</bean>

<bean id="QuoteEndpointDispatcher"
class="org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher" />
<bean id="FutureEndpointDispatcher"
class="org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher" />
```

POJO (UN) MARSHALLING

Camel の プラグ可能なデータ形式は、JAXB、XStream、JibX、Castor、XMLBean などのライブラリーを使用した POJO/XML マーシャリングをサポートします。ルートでこれらのデータ形式を使用して、Web サービスとの間で POJO (Plain Old Java Objects)を送受信できます。

Web サービスにアクセス する場合は、リクエストをマーシャリングし、応答メッセージをアンマーシャリングできます。

```
JaxbDataFormat jaxb = new JaxbDataFormat(false);
jaxb.setContextPath("com.example.model");

from("direct:example").marshal(jaxb).to("spring-ws:http://foo.com/bar").unmarshal(jaxb);
```

同様に、Web サービスを 提供 する場合と同様に、POJO への XML リクエストをアンマーシャリングし、応答メッセージを XML にマーシャリングすることができます。

```
from("spring-ws:rootqname:{http://example.com/}GetFoo?
endpointMapping=#endpointMapping").unmarshal(jaxb)
.to("mock:example").marshal(jaxb);
```

第165章 SQL COMPONENT

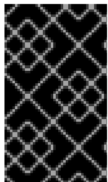
SQL COMPONENT

sql: コンポーネントを使用すると、JDBC クエリーを使用してデータベースを操作することができます。このコンポーネントと JDBC コンポーネントの違いは、SQL の場合、クエリーがエンドポイントのプロパティーであり、クエリーに渡されるパラメーターとしてメッセージペイロードを使用する点です。

このコンポーネントは、実際の SQL 処理の背後で spring-jdbc を使用します。

SQL コンポーネントは以下もサポートします。

- Idempotent Consumer EIP パターンの JDBC ベースのリポジトリー。詳細は以下を参照してください。
- Aggregator EIP パターンの JDBC ベースのリポジトリー。詳細は以下を参照してください。



重要

このコンポーネントは、Transactional Client として使用できます。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

camel-cdi コンポーネントと組み合わせて使用すると、Java EE アノテーションは Camel でデータソースを利用できるようにします。以下の例は @Named アノテーションを使用して、Camel が必要なデータソースを特定できるようにします。

```
public class DatasourceProducer {
    @Resource(name = "java:jboss/datasources/ExampleDS")
    DataSource dataSource;
```

```
    @Produces
    @Named("wildFlyExampleDS")
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

データソースは、以下のように camel-sql エンドポイント設定の dataSource パラメーターから参照できるようになりました。

```
@ApplicationScoped
@ContextName("camel-sql-cdi-context")
@Startup
public class CdiRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("sql:select name from information_schema.users?dataSource=wildFlyExampleDS")
        .to("direct:end");
    }
}
```

URI 形式

警告

Camel 2.11 以降では、このコンポーネントはコンシューマー(from()など)とプロデューサーエンドポイント(g など)の両方を作成できます。to()).以前のバージョンでは、プロデューサーとしてのみ動作できました。

SQL コンポーネントは、以下のエンドポイント URI 表記を使用します。

```
sql:select * from table where id=# order by name[?options]
```

Camel 2.11 以降では、以下のように #name_of_the_parameter スタイルを使用して名前付きパラメーターを使用できます。

```
sql:select * from table where id=:#myId order by name[?options]
```

名前付きパラメーターを使用する場合、Camel は、メッセージヘッダーから java.util.Map 2. の場合

にメッセージボディーから名前を検索します。

名前付きパラメーターを解決できない場合は、例外が発生します。

Camel 2.14 以降では、以下のように Simple 式をパラメーターとして使用できます。

```
sql:select * from table where id=:#${property.myId} order by name[?options]
```

SQL クエリーへのパラメーターを示す標準の ? 記号は、エンドポイントのオプションの指定に ? 記号が使用されるため、# 記号に置き換えられることに注意してください。? 記号の置換は、エンドポイントベースで設定できます。

Camel 2.17 以降では、以下のように SQL クエリーをクラスパスまたはファイルシステムのファイルに外部化できます。

```
sql:classpath:sql/myquery.sql[?options]
```

myquery.sql ファイルはクラスパスにあり、プレーンテキストです。

```
-- this is a comment
select *
from table
where
  id = :#${property.myId}
order by
  name
```

このファイルでは、複数の行を使用し、SQL を必要に応じてフォーマットできます。また、– ダッシュ行などのコメントも使用します。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	型	デフォルト	説明

batch	**boolean**	**false**	Camel 2.7.5、2.8、および 2.9: SQL バッチ更新ステートメントを実行します。**true** に設定されている場合は、インバウンドメッセージボディーのハンドルがどのように変化するかについて以下の注記を参照してください。
dataSourceRef	**文字列**	**null**	**非推奨および は** Camel 3.0: **レジストリーで検索するための DataSource への参照で削除され** ます。代わりに **dataSource=#theName** を使用してください。
dataSource	**文字列**	**null**	Camel 2.11: レジストリーで検索するための **DataSource** への参照。
placeholder	**文字列**	**#**	Camel 2.4: SQL クエリーで **?** に置き換えられる文字を指定します。単純な **String.replaceAll** （）操作であり、SQL 解析が関与していないことに注意してください（引用符で囲まれた文字列も変更されます）。
usePlaceholder	**boolean**	**true**	Camel 2.17: SQL クエリーでプレースホルダーを使用し、すべてのプレースホルダー文字を ? 記号に置き換えるかどうかを設定します。

template.<xxx>		**null**	クエリーを実行するために背後で使用される Spring **JdbcTemplate** に追加のオプションを設定します。例：**template.maxRows=10**詳細なドキュメントは、JdbcTemplate javadoc のドキュメントを参照してください。
allowNamedParameters	**boolean**	**true**	Camel 2.11: クエリーで名前付きパラメーターの使用を許可するかどうか。
processingStrategy			Camel 2.11:SQL consumer only: コンシューマーが行/バッチを処理したときに、プラグインがカスタム **org.apache.camel.component.sql.SqlProcessingStrategy** を使用してクエリーを実行できます。
prepareStatementStrategy			Camel 2.11: プラグインがカスタムの **org.apache.camel.component.sql.SqlPrepareStatementStrategy** を使用して、クエリーおよび準備済みステートメントの準備を制御できます。
consumer.delay	**long**	**500**	Camel 2.11:SQL consumer only: 各ポーリング間の遅延（ミリ秒単位）。
consumer.initialDelay	**long**	**1000**	Camel 2.11:SQL **コンシューマーのみ：** ポーリングの開始前の Milliseconds。

consumer.useFixedDelay	**boolean**	**false**	Camel 2.11:SQL **コンシューマーのみ：** ポーリング間の固定遅延を使用するには **true** に設定します。それ以外の場合は、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。
maxMessagesPerPoll	**int**	**0**	Camel 2.11:SQL **コンシューマーのみ：** ポーリングごとに収集するメッセージの最大数を定義する整数値。デフォルトでは最大値は設定されていません。
useIterator	**boolean**	**true**	Camel 2.11:SQL consumer only: ポーリングが個別に処理されるときに返された各行が **true** の場合。**false** の場合、データの **java.util.List** 全体が IN ボディーとして設定されます。Camel 2.15.x 以前では、このオプションの前に **consumer.** 接頭辞を付ける必要があります（例：**consumer.useIterator=true**
routeEmptyResultSet	**boolean**	**false**	Camel 2.11:SQL **コンシューマーのみ：** ポーリングするデータがない場合に、単一の空の エクスチェンジ をルーティングするかどうか。Camel 2.15.x 以前では、このオプションの前に **consumer.** 接頭辞を付ける必要があります（例：**consumer.useIterator=true**

onConsume	**文字列**	**null**	Camel 2.11:SQL consumer only: 各行を処理した後、エクスチェンジ が正常に処理された場合、このクエリーを実行できます。たとえば、行を processed とマークします。クエリーには パラメーターを指定できます。Camel 2.15.x 以前では、このオプションの前に **consumer.** 接頭辞を付ける必要があります（例： **consumer.useIterator=true**
onConsumeFailed	**文字列**	**null**	Camel 2.11:SQL consumer only: 各行の処理後、エクスチェンジ が失敗した場合にこのクエリーを実行できます。たとえば、行が失敗したとマークされます。クエリーには パラメーターを指定できます。Camel 2.15.x 以前では、このオプションの前に **consumer.** 接頭辞を付ける必要があります（例： **consumer.useIterator=true**
onConsumeBatchComplete	**文字列**	**null**	Camel 2.11:SQL コン**シューマーのみ：** バッチ全体の処理後、このクエリーを実行して行を一括更新することができます。クエリーにはパラメーターを指定できません。Camel 2.15.x 以前では、このオプションの前に **consumer.** 接頭辞を付ける必要があります（例： **consumer.useIterator=true**

expectedUpdateCount	**int**	**\-1**	Camel 2.11:SQL コンシューマーのみ：**consumer.onConsume** を使用する場合、このオプションを使用して、更新される予想される行数を設定できます。通常、これを 1 に設定すると **1** 行が更新されることが予想されます。Camel 2.15.x 以前では、このオプションの前に **consumer.** 接頭辞を付ける必要があります（例：**consumer.useIterator=true**
breakBatchOnConsumeFail	**boolean**	**false**	Camel 2.11:SQL コンシューマーのみ：**consumer.onConsume** を使用し、失敗すると、このオプションはバッチを分割するか、バッチからの次の行の処理を継続するかどうかを制御します。Camel 2.15.x 以前では、このオプションの前に **consumer.** 接頭辞を付ける必要があります（例：**consumer.useIterator=true**

alwaysPopulateStatement	**boolean**	**false**	Camel 2.11:SQL producer only: 有効にすると、**org.apache.camel.component.sql.SqlPrepareStatementStrategy** の **populateStatement** メソッドが常に呼び出されます（準備が想定されるパラメーターがない場合も同様です）。これが **false** の場合、1 つ以上の想定されるパラメーターが設定されている場合にのみ **populateStatement** が呼び出されます。たとえば、パラメーターなしの SQL クエリーのメッセージボディー/ヘッダーの読み取りを回避します。
separator	**char**	**,**	Camel 2.11.1: パラメーター値がメッセージボディー（ボディーが String 型である場合）から取得される場合に使用する区切り文字。# プレースホルダーに挿入されます。名前付きパラメーターを使用する場合は、代わりに **Map** タイプが使用されることに注意してください。

outputType	**文字列**	**SelectList**	Camel 2.12.0: コンシューマーまたはプロデューサーの出力を Map の List に、**SelectOne** を単一の Java オブジェクトとして作成する(a)クエリーに列が 1 つしかない場合は、その JDBC Column オブジェクトが返されます。**SELECT COUNT (*) FROM PROJECT** はLong オブジェクトを返します。b（クエリーに複数の列がある場合） **その結果の Map を返します。c）outputClass が設定されている場合には、列名に一致するすべてのセッターを呼び出すことで、クエリー結果を Java Bean オブジェクトに変換します。クラスにインスタンスを作成するデフォルトのコンストラクターがあるとします。d）クエリーによって複数の行が発生した場合は、一意でない結果例外が出力されます。** **Camel 2.14.1 以降では、SelectList は、SelectOne として各行の Java オブジェクトへのマッピングもサポートしています(step c のみ)。**
outputClass	**文字列**	**null**	Camel 2.12.0: **outputType=SelectOne** の場合に変換として使用する完全なパッケージおよびクラス名を指定します。

outputHeader	**文字列**	**null**	**Camel 2.15:** メッセージのボディーではなくヘッダーとして結果を保存します。これにより、既存のメッセージボディーをそのまま保存できます。
parametersCount	**int**	**0**	**Camel 2.11.2/2**.12.0 がゼロより大きい場合、Camel はこの数のパラメーターを使用でき、JDBC メタデータ API 経由でクエリーするのではなく、置き換えられます。これは、JDBC ベンダーが正しいパラメーター数を返すことができない場合に、代わりに上書きできる場合に便利です。
noop	**boolean**	**false**	**Camel 2.12.0** が設定されている場合、は SQL クエリーの結果を無視し、既存の IN メッセージを処理継続の OUT メッセージとして使用します。
useMessageBodyForSql	**boolean**	**false**	**Camel 2.16:** メッセージボディーを SQL として使用し、ヘッダーをパラメーターに使用するかどうか。このオプションが有効な場合には、uri の SQL は使用されません。その後、SQL パラメーターは、キー CamelSqlParameters を含むヘッダーに提供する必要があります。このオプションはプロデューサー専用です。

transacted	boolean	false	**Camel 2.16.2:** SQL consumer only: トランザクションを有効または無効にします。有効にすると、エクスチェンジの処理が失敗した場合、コンシューマーは追加のエクスチェンジを処理しなくなり、ロールバックの Eager が発生します。

メッセージボディーの処理

*SQL コンポーネントはメッセージボディーを **java.util.Iterator** タイプのオブジェクトに変換しようとし、この **iterator** を使用してクエリーパラメーターを埋めます（各クエリーパラメーターはエンドポイント URI の # 記号または他の設定されたプレースホルダーで表されます）。メッセージボディーが配列またはコレクションではない場合、変換により、ボディー自体の 1 つのオブジェクトのみに対して反復する **iterator** が生成されます。*

*たとえば、メッセージボディーが **java.util.List** のインスタンスである場合、リストの最初の項目は SQL クエリーの最初の **#** に置き換えられます。*

***batch** が **true** に設定されている場合、インバウンドメッセージボディーの解釈はパラメーターのイテレーターではなく、若干変更されます。コンポーネントは、パラメーター **iterators** が含まれるイテレーターを想定し、外部イテレーターのサイズによってバッチサイズが決まります。*

***Camel 2.16** 以降では、**useMessageBodyForSql** オプションを使用してメッセージボディーを SQL ステートメントとして使用し、SQL パラメーターをキー **SqlConstants.SQL_PARAMETERS** のヘッダーに指定する必要があります。SQL クエリーはメッセージボディーからのものであるため、SQL コンポーネントはより動的に機能します。*

クエリーの結果

*選択 操作の場合、結果は **JdbcTemplate.queryFor List （）メソッドによって返される List <Map<String, Object** >> タイプのインスタンスになります。更新 操作の場合、結果は更新された行数で、整数 として返され ます。*

*デフォルトでは、結果はメッセージのボディーに配置されます。**outputHeader** パラメーターを設定すると、結果はヘッダーに配置されます。これは、完全な Message Enrichment パターンを使用してヘッダーを追加する代わりに、シーケンスやその他の小さい値をヘッダーにクエリーするための簡潔な構文を提供します。**outputHeader** と **outputType** を一緒に使用すると便利です。以下に例を示します。*

```
from("jms:order.inbox")
 .to("sql:select order_seq.nextval from dual?outputHeader=OrderId&outputType=SelectOne")
 .to("jms:order.booking");
```

ヘッダーの値

更新 操作を実行すると、SQL コンポーネントは以下のメッセージヘッダーに更新数を保存します。

ヘッダー	説明
CamelSqlUpdateCount	Apache Camel 2.0: **更新操作用に更新** された行数。**Integer** オブジェクトとして返されます。
CamelSqlRowCount	Apache Camel 2.0: **Integer** オブジェクトとして返される **選択** 操作に対して返される行数。
CamelSqlQuery	**Camel 2.8:** 実行するクエリー。このクエリーは、エンドポイント URI で指定されたクエリーよりも優先されます。ヘッダーのクエリーパラメーターは、**#** 記号で **はなく ? で表示さ** れることに注意してください。

insert 操作の実行時に、SQL コンポーネントは生成されたキーとこれらの行の数を以下のメッセージヘッダーに格納します(Camel 2.12.4、2.13.1 で利用可能)。

ヘッダー	説明
CamelSqlGeneratedKeysRowCount	生成されたキーが含まれるヘッダーの行数。
CamelSqlGeneratedKeyRows	生成されたキー（キーのマップの一覧）を含む行。

生成されるキー

Camel 2.12.4、2.13.1、および 2.14 から利用可能

SQL INSERT を使用してデータを挿入すると、----------|----- は自動生成されたキーをサポートする可能性があります。SQL プロデューサーに、ヘッダーで生成されたキーを返すように指示できます。これには、ヘッダー CamelSqlRetrieveGeneratedKeys=true を設定します。次に、生成された鍵が上記の表に記載されているキーが含まれるヘッダーとして提供されます。

詳細は、この ユニットテスト を参照してください。

設定

URI の DataSource への参照を直接設定できるようになりました。

```
sql:select * from table where id=# order by name?dataSourceRef=myDS
```

例

以下の例では、クエリーを実行し、結果を List of rows として取得します。各行は Map<String, Object で、キーは列名になります。

まず、サンプルに使用するテーブルを設定します。これはユニットテストに基づいているため、java コードを実行します。

```
// this is the database we create with some initial data for our unit test
db = new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.DERBY).addScript("sql/createAndPopulateDatabase.sql").b
uild();
```

SQL スクリプト createAndPopulateDatabase.sql は、以下のように実行します。

```
create table projects (id integer primary key, project varchar(10), license varchar(5));
insert into projects values (1, 'Camel', 'ASF');
insert into projects values (2, 'AMQ', 'ASF');
insert into projects values (3, 'Linux', 'XXX');
```

次に、ルートと sql コンポーネントを設定します。sql エンドポイントの前に ダイレクト エンドポイントを使用することに注意してください。これにより、URI direct:simple を使用して direct エンドポイントにエクスチェンジを送信できます。これは、クライアントが長い sql: URI よりも簡単に使用できます。DataSource はレジストリーで検索されるため、標準の Spring XML を使用して DataSource を設定できます。

```
from("direct:simple")
   .to("sql:select * from projects where license = # order by id?
dataSourceRef=jdbc/myDataSource")
   .to("mock:result");
```

次に、メッセージをデータベースをクエリーする sql コンポーネントにルーティングする direct エンドポイントに実行します。

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedMessageCount(1);

// send the query to direct that will route it to the sql where we will execute the query
// and bind the parameters with the data from the body. The body only contains one value
// in this case (XXX) but if we should use multi values then the body will be iterated
// so we could supply a List<String> instead containing each binding value.
template.sendBody("direct:simple", "XXX");

mock.assertIsSatisfied();

// the result is a List
List<?> received = assertIsInstanceOf(List.class,
mock.getReceivedExchanges().get(0).getIn().getBody());

// and each row in the list is a Map
Map<?, ?> row = assertIsInstanceOf(Map.class, received.get(0));

// and we should be able the get the project from the map that should be Linux
assertEquals("Linux", row.get("PROJECT"));
```

Spring XML の DataSource を以下のように設定できます。

```
<jee:jndi-lookup id="myDS" jndi-name="jdbc/myDataSource"/>
```

名前付きパラメーターの使用

Camel 2.11 から利用可能

以下のルートでは、Projects テーブルからすべてのプロジェクトを取得します。SQL クエリーには、 : #lic と :#min の 2 つの名前付きパラメーターがあることに注意してください。その後、Camel はメッセージボディーまたはメッセージヘッダーからこれらのパラメーターを検索します。上記の例では、名前付きパラメーターに定数値で 2 つのヘッダーを設定することに注意してください。

```
from("direct:projects")
  .setHeader("lic", constant("ASF"))
  .setHeader("min", constant(123))
  .to("sql:select * from projects where license = :#lic and id > :#min order by id")
```

メッセージボディーが java.util.Map の場合、名前付きパラメーターはボディーから取得されます。

```
from("direct:projects")
  .to("sql:select * from projects where license = :#lic and id > :#min order by id")
```

式パラメーターの使用

Camel 2.14 から利用可能

以下のルートでは、データベースからすべてのプロジェクトを取得します。ライセンスを定義するためにエクスチェンジのボディーを使用し、2 番目のパラメーターとしてプロパティーの値を使用します。

```
from("direct:projects")
  .setBody(constant("ASF"))
  .setProperty("min", constant(123))
  .to("sql:select * from projects where license = :#${body} and id > :#${property.min} order by
id")
```

動的値による IN クエリーの使用

Camel 2.17 以降では、SQL プロデューサーは、IN ステートメントで SQL クエリーを使用できます。ここで、IN 値は動的に計算されます。たとえば、メッセージボディーやヘッダーなどから。

IN を使用するには、以下を行う必要があります。

- パラメーター名の前に in: および を付けます。
- パラメーターの前後に ( ) を追加します。

たとえば、以下のクエリーが使用されるとします。

```
-- this is a comment
select *
from projects
where project in (:#in:names)
order by id
```

以下のルートでは、以下のようになります。

```
from("direct:query")
    .to("sql:classpath:sql/selectProjectsIn.sql")
    .to("log:query")
    .to("mock:query");
```

次に、IN クエリーは、以下のような動的な値を持つキー名にヘッダーを使用できます。

```
// use an array
template.requestBodyAndHeader("direct:query", "Hi there!", "names", new String[]{"Camel",
"AMQ"});


// use a list
List<String> names = new ArrayList<String>();
names.add("Camel");
names.add("AMQ");

template.requestBodyAndHeader("direct:query", "Hi there!", "names", names);


// use a string separated values with comma
template.requestBodyAndHeader("direct:query", "Hi there!", "names", "Camel,AMQ");
```

クエリーは、外部化されるのではなく、エンドポイントで指定することもできます（外部化により SQL クエリーのメンテナーンスが容易になることに注意してください）。

```
from("direct:query")
    .to("sql:select * from projects where project in (:#in:names) order by id")
    .to("log:query")
    .to("mock:query");
```

JDBC ベースのべき等リポジトリーの使用

このセクションでは、JDBC ベースのべき等リポジトリーを使用します。

抽象クラス

Camel 2.9 以降では、org.apache.camel.processor.idempotent.jdbc.AbstractJdbcMessageIdRepository の抽象クラスがあり、カスタム JDBC べき等リポジトリーをビルドできます。

まず、べき等リポジトリーで使用されるデータベーステーブルを作成する必要があります。

Camel 2.8 では、createdAt 列を追加しました。

```
CREATE TABLE CAMEL_MESSAGEPROCESSED (
  processorName VARCHAR(255),
  messageId VARCHAR(100),
  createdAt TIMESTAMP
)
```

警告

SQL Server TIMESTAMP タイプは、固定長の binary-string タイプです。JDBC 時間タイプ(DATE、TIME、または TIMESTAMP)にはマップされません。

processorName および messageId 列に対して、一意の制約を使用することが推奨されます。この制約の構文はデータベースとデータベースごとに異なるため、ここには表示されません。

次に、spring XML ファイルで javax.sql.DataSource を設定する必要があります。

```
<jdbc:embedded-database id="dataSource" type="DERBY" />
```

最後に、spring XML ファイルで JDBC べき等リポジトリーを作成することもできます。

```
<bean id="messageIdRepository"
class="org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository">
 <constructor-arg ref="dataSource" />
 <constructor-arg value="myProcessorName" />
</bean>

<camel:camelContext>
 <camel:errorHandler id="deadLetterChannel" type="DeadLetterChannel"
deadLetterUri="mock:error">
  <camel:redeliveryPolicy maximumRedeliveries="0" maximumRedeliveryDelay="0"
logStackTrace="false" />
 </camel:errorHandler>

 <camel:route id="JdbcMessageIdRepositoryTest" errorHandlerRef="deadLetterChannel">
```

```
    <camel:from uri="direct:start" />
    <camel:idempotentConsumer messageIdRepositoryRef="messageIdRepository">
     <camel:header>messageId</camel:header>
     <camel:to uri="mock:result" />
    </camel:idempotentConsumer>
  </camel:route>
</camel:camelContext>
```

JDBCMESSAGEIDREPOSITORY のカスタマイズ

Camel 2.9.1 以降では、ニーズに合わせて org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository を調整するオプションがいくつかあります。

パラメーター	デフォルト値	説明
createTableIfNotExists	**true**	Camel が存在しない場合に Camel がテーブルの作成を試行するかどうかを定義します。
tableExistsString	SELECT 1 FROM CAMEL_MESSAGEPROCESSED WHERE 1 = 0	このクエリーは、テーブルがすでに存在しているかどうかを確認するために使用されます。テーブルが存在しないことを示す例外を出力する必要があります。
createString	CREATE TABLE CAMEL_MESSAGEPROCESSED (processorName VARCHAR (255), messageId VARCHAR (100), createdAt TIMESTAMP)	テーブルの作成に使用されるステートメント。
queryString	SELECT COUNT(*) FROM CAMEL_MESSAGEPROCESSED WHERE processorName = ?AND messageId = ?	メッセージがすでにリポジトリーに存在するかどうかを確認するために使用されるクエリー（結果は 0 と等しくありません）。2 つのパラメーターを取ります。最初の 1 つはプロセッサー名（）で、2 つ目はメッセージ ID (**文字列**)です。

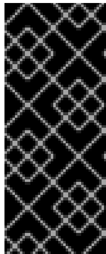
insertString	INSERT INTO CAMEL_MESSAGEPROCESSED (processorName, messageId, createdAt) VALUES (?, ?, ?)	テーブルにエントリーを追加するために使用される ステートメント。3 つのパラメーターを取ります。1 つ目はプロセッサー名(**文字列**)で、2 つ目はメッセージ ID (**文字列**)で、3 つ目は、このエントリーがリポジトリーに追加されたときのタイムスタンプ (**java.sql.Timestamp**)です。
deleteString	DELETE FROM CAMEL_MESSAGEPROCESSED WHERE processorName = ?AND messageId = ?	データベースからエントリーを削除するために使用されるステートメント。2 つのパラメーターを取ります。最初の 1 つはプロセッサー名（）で、2 つ目はメッセージ ID (**文字列**)です。

カスタマイズした org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository は以下のようになります。

```
<bean id="messageIdRepository"
class="org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository">
 <constructor-arg ref="dataSource" />
 <constructor-arg value="myProcessorName" />
 <property name="tableExistsString" value="SELECT 1 FROM
CUSTOMIZED_MESSAGE_REPOSITORY WHERE 1 = 0" />
 <property name="createString" value="CREATE TABLE
CUSTOMIZED_MESSAGE_REPOSITORY (processorName VARCHAR(255), messageId
VARCHAR(100), createdAt TIMESTAMP)" />
 <property name="queryString" value="SELECT COUNT(*) FROM
CUSTOMIZED_MESSAGE_REPOSITORY WHERE processorName = ? AND messageId = ?" />
 <property name="insertString" value="INSERT INTO CUSTOMIZED_MESSAGE_REPOSITORY
(processorName, messageId, createdAt) VALUES (?, ?, ?)" />
 <property name="deleteString" value="DELETE FROM
CUSTOMIZED_MESSAGE_REPOSITORY WHERE processorName = ? AND messageId = ?" />
</bean>
```

JDBC ベースの集約リポジトリーの使用

Camel 2.6 以降で利用可能



CAMEL 2.6 での JDBCAGGREGATIONREPOSITORY の使用

Camel 2.6 では、JdbcAggregationRepository は camel-jdbc-aggregator コンポーネントで提供されます。Camel 2.7 以降、JdbcAggregationRepository は camel-sql コンポーネントで提供されます。

JdbcAggregationRepository は AggregationRepository で、オンザフライで集約されたメッセージを永続化します。これにより、デフォルトのアグリゲーターはメモリーの AggregationRepository のみを使用するので、メッセージを失わないようにします。JdbcAggregationRepository を使用すると、Camel とともに Aggregator の永続的なサポートを提供できます。

これには、以下のオプションがあります。

オプション	タイプ	説明
dataSource	**DataSource**	**必須：** データベースへのアクセスに使用する **javax.sql.DataSource**。
repositoryName	**文字列**	**必須：** リポジトリーの名前。
transactionManager	**TransactionManager**	**必須：** データベースのトランザクションを管理する **org.springframework.transaction.PlatformTransactionManager**。TransactionManager はデータベースをサポートできる必要があります。
lobHandler	**LobHandler**	データベースの Lob タイプを処理する **org.springframework.jdbc.support.lob.LobHandler**。Oracle を使用する場合など、ベンダー固有の LobHandler を使用するには、このオプションを使用します。
returnOldExchange	boolean	get 操作が存在する場合は、get 操作によって古い既存のエクスチェンジが返されるかどうか。デフォルトでは、集計時に古いエクスチェンジを必要としないため、このオプションは **false** で最適化されます。

useRecovery	boolean	リカバリーが有効になっているかどうか。このオプションは、デフォルトで **true** です。有効にすると、Camel Aggregator は自動的に集約されたエクスチェンジをリカバリーし、再提出します。
recoveryInterval	long	recovery が有効になっている場合、バックグラウンドタスクは x 度ごとに実行され、失敗したエクスチェンジをスキャンしてリカバリーし、再送信します。デフォルトでは、この間隔は 5000 ミリ秒です。
maximumRedeliveries	int	リカバリーされたエクスチェンジの再配信試行の最大数を制限できます。有効にすると、すべての再配信試行に失敗すると、エクスチェンジはデッドレターチャネルに移動します。デフォルトでは、このオプションは無効です。このオプションを使用する場合は、**deadLetterUri** オプションも指定する必要があります。
deadLetterUri	文字列	使い切られるリカバリーされたエクスチェンジが移動される Dead Letter Channel のエンドポイント URI。このオプションを使用する場合は、**maximumRedeliveries** オプションも指定する必要があります。
storeBodyAsText	boolean	**Camel 2.11:** メッセージボディーを文字列として格納するかどうか（人間が読める）。デフォルトでは、このオプション **は** ボディーをバイナリー形式で格納します。
headersToStoreAsText	**List<String>**	**Camel 2.11:** ヘッダーを文字列として保存し、人間が判読できる文字列として保存できます。デフォルトでは、このオプションは無効になっており、ヘッダーをバイナリー形式で保存します。
optimisticLocking	**false**	**Camel 2.12:** 複数の Camel アプリケーションが同じ JDBC ベースの集約リポジトリーを共有するクラスター化された環境で必要となる楽観的ロックを有効にします。

jdbcOptimisticLockingExceptionMapper		Camel 2.12: カスタムの **org.apache.camel.processor.aggregate.jdbc.JdbcOptimisticLockingExceptionMapper** をプラグインして、Camel が再試行できるようにベンダー固有のエラーコードを optimistick ロックエラーにマップできるようにします。これには、**optimisticLocking** を有効にする必要があります。

永続化時に保持される内容

JdbcAggregationRepository は、Serializable と互換性のあるデータタイプのみを保持します。データ型がそのようなタイプの場合はドロップされ、WARN がログに記録されます。また、メッセージ本文とメッセージ ヘッダーのみを保持 し ます。Exchange プロパティーは永続化され ません。

Camel 2.11 以降では、メッセージボディーを保存し、ヘッダーを String として個別の列に選択します。

復元

JdbcAggregationRepository はデフォルトで失敗したエクスチェンジ を復元し ます。これは、永続ストアで失敗した エクスチェンジをスキャンするバックグラウンドタスクを持つことで行われます。checkInterval オプションを使用して、このタスクの実行頻度を設定できます。リカバリーはトランザクションとして機能し、Camel が失敗したエクスチェンジのリカバリーおよび再配信を試み ます。リカバリー さ れたエクスチェンジは永続ストアから復元され、再送信されて再度送信されます。

エクスチェンジ のリカバリー/再配信時に、以下のヘッダーが設定されます。

ヘッダー	タイプ	説明
Exchange.REDELIVERED	ブール値	エクスチェンジ が再配信されていることを示すために が true に設定されます。
Exchange.REDELIVERY_COUNTER	整数	1 から始まる再配信の試行。

エクスチェンジ が正常に処理された場合のみ、完了とマークされます。これは、AggregationRepository で confirm メソッドが呼び出されたときに発生します。つまり、同じ エ

クスチェンジ が再び失敗すると、成功するまで再試行されます。

maximumRedeliveries オプションを使用して、特定のリカバリーエクスチェンジの再配信試行の最大数を制限でき ます。maximumRedeliveries に達したときに エクスチェンジ を送信する場所を Camel が認識できるように deadLetterUri オプションも設定する必要があります。

camel-sql のユニットテストの例の一部を確認できます（例： このテスト ）。

データベース

各アグリゲーターは集約と完了した 2 つのテーブルを使用します。慣例により、完了により、接尾辞が _COMPLETED の集約と同じ名前が付けられます。名前は Spring Bean で RepositoryName プロパティーで設定する必要があります。以下の例では、集約が使用されます。

両方のテーブルのテーブル構造定義は同一です。いずれの場合も、String 値がキー（）として使用されますが、Blob にはバイトアレイでシリアライズされたエクスチェンジが含まれます。ただし、1 つの違いを覚えておくべきです。id フィールドはテーブルに応じて同じコンテンツを持つことができません。集約テーブル ID は、メッセージ を 集約するためにコンポーネントによって使用される相関 ID を保持します。完了したテーブルでは、id は、対応する Blob フィールドに保存されるエクスチェンジの ID を保持します。

以下は、テーブルの作成に使用する SQL クエリーです。aggregation はアグリゲーターリポジトリー名に置き換えます。

```
CREATE TABLE aggregation (
    id varchar(255) NOT NULL,
    exchange blob NOT NULL,
    constraint aggregation_pk PRIMARY KEY (id)
);
CREATE TABLE aggregation_completed (
    id varchar(255) NOT NULL,
    exchange blob NOT NULL,
    constraint aggregation_completed_pk PRIMARY KEY (id)
);
```

ボディーとヘッダーをテキストとして保存

Camel 2.11 から利用可能

JdbcAggregationRepository を設定してメッセージボディーを保存し、ヘッダーを別の列に String

として選択できます。たとえば、ボディーと以下の 2 つのヘッダー companyName および accountName を保存するには、以下の SQL を使用します。

```
CREATE TABLE aggregationRepo3 (
   id varchar(255) NOT NULL,
   exchange blob NOT NULL,
   body varchar(1000),
   companyName varchar(1000),
   accountName varchar(1000),
   constraint aggregationRepo3_pk PRIMARY KEY (id)
);
CREATE TABLE aggregationRepo3_completed (
   id varchar(255) NOT NULL,
   exchange blob NOT NULL,
   body varchar(1000),
   companyName varchar(1000),
   accountName varchar(1000),
   constraint aggregationRepo3_completed_pk PRIMARY KEY (id)
);
```

次に、以下のようにこの動作を有効にするようにリポジトリーを設定します。

```
   <bean id="repo3"
class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">
      <property name="repositoryName" value="aggregationRepo3"/>
      <property name="transactionManager" ref="txManager3"/>
      <property name="dataSource" ref="dataSource3"/>
      <!-- configure to store the message body and following headers as text in the repo -->
      <property name="storeBodyAsText" value="true"/>
      <property name="headersToStoreAsText">
        <list>
         <value>companyName</value>
       <value>accountName</value>
        </list>
      </property>
   </bean>
```

CODEC (SERIALIZATION)

任意のタイプのペイロードを含めることができるため、エクスチェンジは設計によってシリアライズできません。データベース BLOB フィールドに保存されるバイトアレイに変換されます。これらの変換はすべて JdbcCodec クラスによって処理されます。コードの詳細の 1 つである ClassLoadingAwareObjectInputStream に注意してください。

ClassLoadingAwareObjectInputStream が Apache ActiveMQ プロジェクトから再利用されました。ObjectInputStream をラップし、currentThread ではなく ContextClassLoader で使用します。

この利点は、他のバンドルによって公開されるクラスをロードできることです。これにより、エクスチェンジボディーとヘッダーにカスタム型オブジェクトの参照を設定できます。

TRANSACTION

トランザクションのオーケストレーションには、Spring PlatformTransactionManager が必要です。

サービス（開始/停止）

start メソッドは、データベースのコネクションと必要なテーブルが存在することを確認します。何らかの誤りがある場合は、起動時に失敗します。

AGGREGATOR の設定

対象の環境によっては、Aggregator の設定が必要になる場合があります。ご存知のように、各アグリゲーターには独自のリポジトリー（データベースに作成された対応するテーブルのペアを含む）とデータソースが必要です。デフォルトの lobHandler がデータベースシステムに適応しない場合は、lobHandler プロパティーでインジェクトできます。

以下は Oracle の宣言です。

```
<bean id="lobHandler" class="org.springframework.jdbc.support.lob.OracleLobHandler">
    <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>
</bean>

<bean id="nativeJdbcExtractor"
class="org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor"/>

<bean id="repo"
class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="repositoryName" value="aggregation"/>
    <property name="dataSource" ref="dataSource"/>
    <!-- Only with Oracle, else use default -->
    <property name="lobHandler" ref="lobHandler"/>
</bean>
```

OPTIMISTIC LOCKING

Camel 2.12 以降では、optimisticLocking をオンにし、複数の Camel アプリケーションが集約リポジトリーに対して同じデータベースを共有するクラスター環境でこの JDBC ベースの集約リポジトリーを使用できます。競合状態がある場合、JDBC ドライバーは JdbcAggregationRepository が応答でき

るベンダー固有の例外を出力します。JDBC ドライバーからの例外が optimistick ロックエラーと見なされるかを知るには、マッパーが必要です。そのため、org.apache.camel.processor.aggregate.jdbc.JdbcOptimisticLockingExceptionMapper があり、必要に応じてカスタムロジックを実装できます。デフォルトの実装 org.apache.camel.processor.aggregate.jdbc.DefaultJdbcOptimisticLockingExceptionMapper は以下のように機能します。

以下のチェックが行われます。

- 原因となった例外が SQLException の場合、SQLState は 23 で始まる場合にチェックされます。
- 発生した例外が DataIntegrityViolationExceptionの場合
- 発生した例外クラス名の名前が ConstraintViolation である場合。
- クラス名が設定されている場合、FQN クラス名が一致するためのオプションチェック

さらに FQN クラス名を追加できます。発生した例外（またはネストされた）のいずれかが FQN クラス名と一致する場合は、optimistick locking エラーになります。

以下の例は、JDBC ベンダーから 2 つの追加の FQN クラス名を定義します。

```
<bean id="repo"
class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="repositoryName" value="aggregation"/>
    <property name="dataSource" ref="dataSource"/>
    <property name"jdbcOptimisticLockingExceptionMapper" ref="myExceptionMapper"/>
</bean>

<!-- use the default mapper with extra FQN class names from our JDBC driver -->
<bean id="myExceptionMapper"
class="org.apache.camel.processor.aggregate.jdbc.DefaultJdbcOptimisticLockingExceptionM
apper">
  <property name="classNames">
    <util:set>
      <value>com.foo.sql.MyViolationExceptoion</value>
      <value>com.foo.sql.MyOtherViolationExceptoion</value>
```

```
    </util:set>
  </property>
</bean>
```

以下も参照してください。

- JDBC

第166章 SQL ストアドプロシージャ

SQL ストアドプロシージャコンポーネント

Camel 2.17 以降で利用可能

sql-stored: コンポーネントを使用すると、JDBC Stored Procedure クエリーを使用してデータベースを操作することができます。このコンポーネントは SQL コンポーネントの拡張ですが、ストアドプロシージャーを呼び出すために特化されています。

このコンポーネントは、実際の SQL 処理のために spring-jdbc を背後で使用します。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-sql</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

SQL コンポーネントは、以下のエンドポイント URI 表記を使用します。

```
sql-stored:template[?options]
```

ここでの template は、ストアドプロシージャーの名前および IN および OUT 引数を宣言するテンプレートです。

また、ファイルシステム上の外部ファイルのテンプレートや、以下のようなクラスパスを参照することもできます。

```
sql-stored:classpath:sql/myprocedure.sql[?options]
```

sql/myprocedure.sql は、以下のようにテンプレートを含むクラスパスのプレーンテキストファイルです。

```
SUBNUMBERS(
  INTEGER ${headers.num1},
  INTEGER ${headers.num2},
  OUT INTEGER resultofsub
)
```

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

オプション

オプション	型	デフォルト	説明
batch	**boolean**	**false**	バッチモードを有効または無効にします。
dataSource	**string**		レジストリーで検索するための **DataSource** への参照。
noop	**boolean**	**false**	が設定されている場合、テンプレートの結果を無視し、処理を継続するために OUT メッセージとして既存の IN メッセージを使用します。
outputHeader	**string**		テンプレート結果をメッセージのボディーではなくヘッダーに保存します。デフォルトでは outputHeader == null とテンプレートの結果はメッセージボディーに保存されます。メッセージボディーの既存のコンテンツは破棄されます。outputHeader が設定されている場合は、テンプレートの結果を格納するヘッダーの名前として値が使用され、元のメッセージボディーは保持されます。

useMessageBodyForTemplate	**boolean**	**false**	メッセージボディーをテンプレートとして使用し、ヘッダーをパラメーターに使用するかどうか。このオプションが有効な場合には、uri のテンプレートは使用されません。

ストアドプロシージャーテンプレートの宣言

*テンプレートは、**Java** メソッド署名と同様の構文を使用して宣言されます。ストアドプロシージャーの名前、そして括弧で囲まれた引数。この例では、以下も説明しています。*

```
<to uri="sql-stored:SUBNUMBERS(INTEGER ${headers.num1},INTEGER
${headers.num2},OUT INTEGER resultofsub)"/>
```

*引数は 型によって宣言され、**Simple** 式を使用した **Camel** メッセージへのマッピングです。この例では、最初の **2** つのパラメーターは **INTEGER** タイプの **IN** 値で、メッセージヘッダーにマッピングされます。最後のパラメーターは **OUT** 値で、**INTEGER** タイプでもあります。*

***SQL** 用語では、ストアドプロシージャーは次のように宣言できます。*

```
CREATE PROCEDURE SUBNUMBERS(VALUE1 INTEGER, VALUE2 INTEGER,OUT RESULT
INTEGER)
```

第167章 SSH

SSH

Camel 2.10 以降で利用可能

SSH コンポーネントを使用すると、SSH コマンドを送信して応答を処理できるように SSH サーバーにアクセスできます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-ssh</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
ssh:[username[:password]@]host[:port][?options]
```

オプション

名前	デフォルト値	説明
host		SSH サーバーのホスト名
port	22	SSH サーバーのポート
username		SSH サーバーでの認証に使用されるユーザー名。

password		SSH サーバーによる認証に使用されるパスワード。**keyPairProvider** が null の場合に使用されます。
keyPairProvider		認証用のキーの読み込みに使用する **org.apache.sshd.common.KeyPairProvider** を参照します。このオプションを使用する場合は、**パスワード** は使用されません。
keyType	**ssh-rsa**	**keyPairProvider** からロードするキータイプを参照します。キータイプは、たとえば ssh-rsa または ssh-dss です。
certResource	**null**	Camel 2.11: 公開鍵証明書へのパス参照。クラスパスで接頭辞 **path:**、**file:**、または **http:** を付けます。
certFilename	**null**	@**非推奨：** 代わりに **certResource** を使用してください。ファイルベースの **keyPairProvider** 内で使用するファイル名を参照します。
timeout	**30000**	SSH サーバーへの接続がタイムアウトするまで待機する時間（ミリ秒単位）。

コンシューマーのみのオプション

名前	デフォルト値	説明
initialDelay	**1000**	SSH サーバーが起動するポーリングをポーリングするまでの時間（ミリ秒単位）。

delay	500	SSH サーバーの次のポーリングまでの時間（ミリ秒単位）。
useFixedDelay	true	固定遅延または固定レートを使用するかどうかを制御します。詳細は、JDK の **ScheduledExecutorService** を参照してください。
pollCommand	null	各ポーリングサイクル中に SSH サーバーに送信するコマンド。コマンドを改行で終了する必要があり、URL でエンコードされた **%0A** である必要があります。

プロデューサーエンドポイントとしての使用

SSH コンポーネントがプロデューサー(.to ("ssh://...")として使用される場合)は、メッセージボディーをリモート SSH サーバーで実行するコマンドとして送信します。

***以下は、XML DSL 内の例になります。コマンドには、XML でエンコードされた改行(
)があることに注意してください。***

```
<route id="camel-example-ssh-producer">
  <from uri="direct:exampleSshProducer"/>
  <setBody>
    <constant>features:list&#10;</constant>
  </setBody>
  <to uri="ssh://karaf:karaf@localhost:8101"/>
  <log message="${body}"/>
</route>
```

認証

SSH コンポーネントは、公開鍵証明書またはユーザー名/パスワードの 2 つのメカニズムを使用して、リモート SSH サーバーに対して認証できます。SSH コンポーネントがどのように認証を行うかの設定は、どのオプションがどのように設定されているかに基づいています。

1. まず、certResource オプションが設定されているかどうかを確認し、そのオプションを使用している場合は、これを使用して参照される公開鍵証明書を見つけ、それを認証に使用することができます。

2. certResource が設定されていない場合、keyPairProvider が設定されているかどうかを確認し、設定されている場合は、そのものを証明書ベースの認証に使用します。

3. certResource も keyPairProvider も設定されていない場合は、認証に username オプションおよび password オプションが使用されます。

以下のルートのフラグメントは、クラスパスからの証明書を使用して SSH ポーリングコンシューマーを示しています。

XML DSL では、以下のようになります。

```
<route>
  <from uri="ssh://scott@localhost:8101?
certResource=classpath:test_rsa&seFixedDelay=true&elay=5000&ollCommand=features:list%
0A"/>
  <log message="${body}"/>
</route>
```

Java DSL では、以下のようになります。

```
from("ssh://scott@localhost:8101?
certResource=classpath:test_rsa&useFixedDelay=true&delay=5000&pollCommand=features:li
st%0A")
    .log("${body}");
```

公開鍵認証の使用例は、examples/camel-example-ssh-security にあります。

証明書の依存関係

証明書ベースの認証を使用する場合は、追加のランタイム依存関係を追加する必要があります。表示されている依存関係バージョンは Camel 2.11 で、使用している Camel のバージョンに応じて、より新しいバージョンを使用する必要がある場合があります。

```
<dependency>
  <groupId>org.apache.sshd</groupId>
  <artifactId>sshd-core</artifactId>
  <version>0.8.0</version>
</dependency>
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcpg-jdk15on</artifactId>
  <version>1.47</version>
</dependency>
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcpkix-jdk15on</artifactId>
  <version>1.47</version>
</dependency>
```

例

Camel ディストリビューションの examples/camel-example-ssh および examples/camel-example-ssh-security を参照してください。

第168章 STAX

STAX COMPONENT

Camel 2.9 以降で利用可能

StAX コンポーネントを使用すると、SAX ContentHandler を介してメッセージを処理できます。このコンポーネントのもう 1 つの機能は、Splitter EIP を使用する JAXB レコードの反復を可能にすることです。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-stax</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
stax:content-handler-class
```

例：

```
stax:org.superbiz.FooContentHandler
```

Camel 2.11.1 以降では、以下のように # 構文を使用して、レジストリーから org.xml.sax.ContentHandler Bean を検索できます。

```
stax:#myHandler
```

コンテンツハンドラーを STAX パーサーとして使用する

処理後のメッセージボディーはハンドラー自体です。

以下に例を示します。

```
from("file:target/in")
  .to("stax:org.superbiz.handler.CountingHandler")
  // CountingHandler implements org.xml.sax.ContentHandler or extends
org.xml.sax.helpers.DefaultHandler
  .process(new Processor() {
    @Override
    public void process(Exchange exchange) throws Exception {
        CountingHandler handler = exchange.getIn().getBody(CountingHandler.class);
        // do some great work with the handler
    }
  });
```

JAXB および STAX を使用してコレクションを繰り返し処理します。

最初に JAXB オブジェクトがあるとします。

たとえば、ラッパーオブジェクトのレコード一覧は以下のようになります。

```
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "records")
public class Records {
    @XmlElement(required = true)
    protected List<Record> record;

    public List<Record> getRecord() {
        if (record == null) {
            record = new ArrayList<Record>();
        }
        return record;
    }
}
```

および

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
```

```
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "record", propOrder = { "key", "value" })
public class Record {
    @XmlAttribute(required = true)
    protected String key;

    @XmlAttribute(required = true)
    protected String value;

    public String getKey() {
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

次に、処理する XML ファイルを取得します。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<records>
  <record value="v0" key="0"/>
  <record value="v1" key="1"/>
  <record value="v2" key="2"/>
  <record value="v3" key="3"/>
  <record value="v4" key="4"/>
  <record value="v5" key="5"/>
</record>
```

StAX コンポーネントは、Camel Splitter で XML 要素を反復処理する際に使用できる StAXBuilder を提供します。

```
from("file:target/in")
    .split(stax(Record.class)).streaming()
        .to("mock:records");
```

stax は、Java コードで静的インポートできる org.apache.camel.component.stax.StAXBuilder の静的メソッドです。stax ビルダーは、デフォルトで、それが使用する XMLReader を認識しま

す。Camel 2.11.1 以降では、以下のようにブール値パラメーターを false に設定するとこの機能をオフにできます。

```
from("file:target/in")
   .split(stax(Record.class, false)).streaming()
      .to("mock:records");
```

XML DSL を使用した前述の例

上記の例は、XML DSL で以下のように実装できます。

```
<!-- use STaXBuilder to create the expression we want to use in the route below for splitting
the XML file -->
<!-- notice we use the factory-method to define the stax method, and to pass in the parameter
as a constructor-arg -->
<bean id="staxRecord" class="org.apache.camel.component.stax.StAXBuilder" factory-
method="stax">
  <!-- FQN class name of the POJO with the JAXB annotations -->
  <constructor-arg index="0" value="org.apache.camel.component.stax.model.Record"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <!-- pickup XML files -->
    <from uri="file:target/in"/>
    <split streaming="true">
      <!-- split the file using StAX (ref to bean above) -->
      <!-- and use streaming mode in the splitter -->
      <ref>staxRecord</ref>
      <!-- and send each splitted to a mock endpoint, which will be a Record POJO instance -->
      <to uri="mock:records"/>
    </split>
  </route>
</camelContext>
```

第169章 STOMP

STOMP コンポーネント

Camel 2.12 以降で利用可能

stomp: コンポーネントは、Apache ActiveMQ や ActiveMQ Apolloなどの Stomp 準拠のメッセージブローカーとの通信に使用されます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-stomp</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
stomp:queue:destination[?options]
```

destination はキューの名前です。

オプション

プロパティー	デフォルト	説明

brokerURL	**tcp://localhost:61613**	接続する Stomp ブローカーの URI
login		ユーザー名
passcode		パスワード
host		Camel 2.15.3/2.16: 仮想ホスト。
sslContextParameters	**null**	Camel 2.17: レジストリーの **org.apache.camel.util.jsse.SSLContextParameters** への参照。ブローカー URL は **ssl** をプロトコルとして使用する必要があります。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

サンプル

メッセージの送信：

```
from("direct:foo").to("stomp:queue:test");
```

メッセージの使用：

```
from("stomp:queue:test").transform(body().convertToString()).to("mock:result")
```

第170章 ストリーム

ストリームコンポーネント

stream: コンポーネントは、System.in、System.out および System.err ストリームへのアクセスを提供し、ファイルと URL のストリーミングを許可します。

URI 形式

```
stream:in[?options]
stream:out[?options]
stream:err[?options]
stream:header[?options]
```

さらに、ファイル および URL エンドポイント URI は Apache Camel 2.0 でサポートされています。

```
stream:file?fileName=/foo/bar.txt
stream:url[?options]
```

stream:header URI が指定されている場合、ストリーム ヘッダーを使用して、書き込むストリームを検索します。このオプションは、ストリームプロデューサー（つまり、from （）には表示されません）でのみ利用できます。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

オプション

名前	デフォルト値	説明
delay	0	ストリームの消費または生成前の初期遅延（ミリ秒単位）。

encoding	JVM のデフォルト	1.4 の時点では、テキストベースのストリーム（例：メッセージボディーは **String** オブジェクト）を使用するようにエンコーディング（文字 セット名）を設定できます。指定のない場合、Apache Camel は JVM のデフォルト Charset を 使用します。
promptMessage	**null**	Apache Camel 2.0: **stream:in** から読み取るときに使用するメッセージプロンプト。たとえば、これを **Enter a command に設定できます。**
promptDelay	**0**	Apache Camel 2.0: メッセージプロンプトを表示する前のオプションの遅延（ミリ秒単位）。
initialPromptDelay	**2000**	Apache Camel 2.0: メッセージプロンプトを表示する前の初期遅延（ミリ秒単位）。この遅延は 1 回だけ発生します。システムの起動時に使用され、システムへの他のロギングが行われている間にメッセージのプロンプトが書き込まれないようにすることができます。
fileName	**null**	Apache Camel 2.0: **stream:file** URI 形式を使用する場合、このオプションはストリーミング元/元のファイル名を指定します。
url	**null**	**stream:url** URI 形式を使用する場合、このオプションはストリーム間の URL を指定します。入力ストリームは、JDK URLConnection 機能を使用して開きます。
scanStream	**false**	Apache Camel 2.0: unix **tail** コマンドなどのストリームを継続的に読み取るために使用されます。**Camel 2.4 to Camel 2.6:** 上書きされるとファイルのオープンを再試行します。**tail --retry** のようになります。
retry	**false**	**Camel 2.7:** ファイルが上書きされた場合にファイルを開く再試行します。**tail --retry** のようになります。

scanStreamDelay	**0**	Apache Camel 2.0: **scanStream** を使用する場合の読み取り試行間の遅延（ミリ秒単位）。
groupLines	**0**	**Camel 2.5:** コンシューマーの X 数の行をグループ化します。たとえば、10 行をグループ化し、行ごとに 1 Exchange ではなく、10 行だけのエクスチェンジをスプアウトします。
autoCloseCount	**0**	**Camel 2.10.0:** (2.9.3 および 2.8.6) プロデューサーでストリームを閉じる前に処理するメッセージの数。デフォルトではストリームを閉じません(Producer が停止した場合のみ)。より多くのメッセージが送信されると、ストリームは別の **autoCloseCount** バッチに対して再度開きます。
closeOnDone	**false**	**Camel 2.11.0:** このオプションは、Splitter と streaming を同じファイルに組み合わせて使用します。パフォーマンスを向上させるために、ストリームを開いた状態にし、スプリッターが完了したときのみ閉じることです。ここでは、2 つ以上のファイルではなく、同じファイルにのみストリームを行う必要があり、最後の分割されたメッセージがストリームエンドポイントにルーティングされ、閉じられるシグナルを取得する必要があることに注意してください。

メッセージの内容

stream: コンポーネントは、ストリームに書き込むために String または byte[] のいずれかをサポートします。String または byte[] コンテンツのいずれかを message.in.body に追加します。ストリームに送信されたメッセージ： バイナリーモードのプロデューサーは、(String メッセージではなく)改行文字が続くことはありません。null ボディーを持つメッセージは出力ストリームに追加されません。

サンプル

以下の例では、direct:in エンドポイントから System.out ストリームにメッセージをルーティングします。

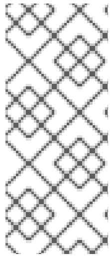
```
// Route messages to the standard output.
from("direct:in").to("stream:out");

// Send String payload to the standard output.
// Message will be followed by the newline.
template.sendBody("direct:in", "Hello Text World");

// Send byte[] payload to the standard output.
// No newline will be added after the message.
template.sendBody("direct:in", "Hello Bytes World".getBytes());
```

以下の例は、ヘッダータイプを使用して、使用するストリームを判断する方法を示しています。この例では、独自の出力ストリーム MyOutputStream を使用します。

```
private OutputStream mystream = new MyOutputStream();
private StringBuffer sb = new StringBuffer();

@Test
public void testStringContent() {
    template.sendBody("direct:in", "Hello");
    // StreamProducer appends \n in text mode
    assertEquals("Hello\n", sb.toString());
}

@Test
public void testBinaryContent() {
    template.sendBody("direct:in", "Hello".getBytes());
    // StreamProducer is in binary mode so no \n is appended
    assertEquals("Hello", sb.toString());
}

protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
            from("direct:in").setHeader("stream", constant(mystream)).
                to("stream:header");
        }
    };
}

private class MyOutputStream extends OutputStream {

    public void write(int b) throws IOException {
        sb.append((char)b);
    }
}
```

以下の例は、ファイルストリームを継続的に読み取る方法を示しています(UNIX tail コマンドに似ています)。

```
from("stream:file?
fileName=/server/logs/server.log&scanStream=true&scanStreamDelay=1000").to("bean:logSer
vice?method=parseLogLine");
```



注記

scanStream と retry の組み合わせを使用する場合の 1 つは、ファイルが再開かれ、scanStreamDelay の反復ごとにスキャンされることです。NIO2 が利用可能になるまで、ファイルがいつ削除または再作成されるかを確実に検出することはできません。

第171章 STRINGTEMPLATE

STRING TEMPLATE

string-template: コンポーネントを使用すると、String Template を使用してメッセージを処理できます。これは、Templating を使用してリクエストの応答を生成する場合に理想的です。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-stringtemplate</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
string-template:templateName[?options]
```

templateName は、呼び出すテンプレートのクラスパスローカル URI、またはリモートテンプレートの完全な URL です。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	デフォルト	説明
allowContextMapAll (producer)	**false**	コンテキストマップが前詳細へのアクセスを許可するかどうかを設定します。デフォルトでは、メッセージの本文とヘッダーにのみアクセスできます。このオプションは、現在の Exchange および CamelContext へのフルアクセスに対して有効にできます。これを行うと、CamelContext API の全機能へのアクセスが開かれるため、潜在的なセキュリティーリスクが発生します。

allowTemplateFromHeader (producer)	**false**	ヘッダーのリソーステンプレートの使用を許可するかどうか (デフォルトは false)。**このオプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼されていないコンテンツやユーザー派生コンテンツが含まれている場合、これは最終的にエンドアプリケーションの信頼性および完全性に影響を与える可能性があるため、このオプションは注意して使用してください。**
contentCache	**false**	読み込み時のリソースコンテンツのキャッシュ。注記： Camel 2.9 でキャッシュされたリソースコンテンツは、エンドポイントの **clearContentCache** 操作を使用して JMX 経由でクリアできます。
delimiterStart	**null**	Camel 2.11.1 以降、変数開始区切り文字の設定
delimiterStop	**null**	Camel 2.11.1 以降、変数終了区切り文字の設定

HEADERS

Apache Camel は、キー org.apache.camel.stringtemplate.resource を持つメッセージヘッダーのリソースへの参照を保存します。リソースは org.springframework.core.io.Resource オブジェクトです。

ホットリロード

文字列テンプレートリソースは、デフォルトではファイルとクラスパスリソース（展開形式の jar）の両方に対してホットリロードが可能です。contentCache=true を設定すると、Apache Camel はリソースを 1 度だけ読み込み、ホットリロードができません。このシナリオは、リソースが変更されない場合に実稼働で使用できます。

STRINGTEMPLATE 属性

Apache Camel は、エクスチェンジ情報を属性(java.util.Mapのみ)として文字列テンプレートに提供します。Exchange は以下のように転送されます。

key	value
exchange	エクスチェンジ自体。
ヘッダー	In メッセージのヘッダー。
camelContext	Camel コンテキスト。
request	In メッセージ。
in	In メッセージ。
ボディー	In メッセージのボディー。
out	Out メッセージ(InOut メッセージ交換パターンにのみ有効)。
response	Out メッセージ(InOut メッセージ交換パターンにのみ有効)。

Camel 2.14 以降、以下のコードと同様に、メッセージヘッダーCamelStringTemplateVariableMapを設定してカスタムコンテキストマップを定義できます。

```
Map<String, Object> variableMap = new HashMap<String, Object>();
Map<String, Object> headersMap = new HashMap<String, Object>();
headersMap.put("name", "Willem");
variableMap.put("headers", headersMap);
variableMap.put("body", "Monday");
variableMap.put("exchange", exchange);
exchange.getIn().setHeader("CamelStringTemplateVariableMap", variableMap);
```

サンプル

たとえば、メッセージへの応答を形成するために、以下のように文字列テンプレートを使用できます。

```
from("activemq:My.Queue").
  to("string-template:com/acme/MyResponse.tm");
```

電子メールのサンプル

この例では、文字列テンプレートを使用して注文確認メールを送信します。メールテンプレートは StringTemplate にそのまま配置されます。この例では camel 2.11.0 で機能します。Camel バージョン

が 2.11.0 未満の場合は、変数を起動して $ で終了する必要があります。

```
Dear <headers.lastName>, <headers.firstName>

Thanks for the order of <headers.item>.

Regards Camel Riders Bookstore
<body>
```

Java コードは以下のようになります。

```
private Exchange createLetter() {
    Exchange exchange = context.getEndpoint("direct:a").createExchange();
    Message msg = exchange.getIn();
    msg.setHeader("firstName", "Claus");
    msg.setHeader("lastName", "Ibsen");
    msg.setHeader("item", "Camel in Action");
    msg.setBody("PS: Next beer is on me, James");
    return exchange;
}

@Test
public void testVelocityLetter() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedBodiesReceived("Dear Ibsen, Claus! Thanks for the order of Camel in
Action. Regards Camel Riders Bookstore PS: Next beer is on me, James");

    template.send("direct:a", createLetter());

    mock.assertIsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:a").to("string-
template:org/apache/camel/component/stringtemplate/letter.tm").to("mock:result");
        }
    };
}
```

第172章 STUB

スタブコンポーネント

Camel 2.10 以降で利用可能

スタブ： コンポーネントは、開発またはテスト中に物理エンドポイントをスタブアウトする簡単な方法を提供し、たとえば特定の SMTP または HTTP エンドポイントに接続しなくてもルートを実行できます。スタブ を追加するだけです。エンドポイント URI の前にあるエンドポイント URI をスタブアウトします。

Stub コンポーネントが内部的に VM エンドポイントを作成します。Stub と VM の主な違いは、仮想マシンが指定する URI とパラメーターを検証することです。したがって、vm: をクエリー引数を持つ一般的な URI の前に配置すると、通常は失敗します。スタブは、基本的にすべてのクエリーパラメーターを無視して、ルート内の 1 つ以上のエンドポイントを一時的にスタブにするためです。

URI 形式

```
stub:someUri
```

someUri は、任意のクエリーパラメーターを持つ任意の URI にすることができます。

例

以下はいくつかのサンプルです。

-

stub:smtp://somehost.foo.com?user=whatnot&something=else

-

stub:http://somehost.bar.com/something

第173章 SWAGGER

概要

Swagger コンポーネントを使用すると、CamelContext 内の Rest 定義されたルートおよびエンドポイントの API ドキュメントを作成できます。Swagger コンポーネントは、CamelContext と統合したサーブレットを作成します。このサーブレットは、各 Rest エンドポイントから情報を取得して API ドキュメント(JSON ファイル)を生成します。

173.1. 概要

Camel 2.14 から利用可能

Rest DSL は、camel-swagger コンポーネントと統合できます。このコンポーネントは、Swagger を使用して REST サービスとその API を公開するために使用されます。

Dependencies

このコンポーネントを使用するには、Maven ユーザーは pom.xml ファイルに以下の依存関係を追加する必要があります。

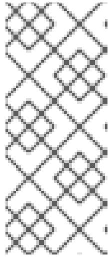
```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-swagger</artifactId>
   <version>2.17.0.redhat-630xxx</version>
   <!-- Use the same version as your Camel core version -->
<dependency>
```

Swagger サーブレットの選択

使用するサーブレットは、使用する Camel バージョンによって異なります。

- Camel 2.15.x

  org.apache.camel.component.swagger.DefaultCamelSwaggerServlet



注記

このデフォルトのサーブレットは任意の環境と統合し、JMX を使用して使用する CamelContext を検出します。これは、Camel 2.15 以降で非推奨となった Camel 2.14.x サーブレットの両方を置き換えます。

- *Camel 2.14.x*

  Swagger サーブレットは Spring または servletListener コンポーネントと統合されます。

  - *Spring*

    org.apache.camel.component.swagger.spring.SpringRestSwaggerApiDeclarationServlet

  - *servletListener component*

    org.apache.camel.component.swagger.servletlistener.ServletListenerRestSwaggerApiDelarationServlet

サーブレット設定パラメーター

すべてのサーブレットは、以下のオプションをサポートします。

パラメーター	タイプ	説明
api.contact	文字列	API 関連の対応に使用する電子メールを指定します。
api.description	文字列	[**必須**] アプリケーションの簡単な説明を指定します。
api.license	文字列	API に使用されるライセンスの名前を指定します。
api.licenseUrl	文字列	API に使用されるライセンスの URL を指定します。

パラメーター	タイプ	説明
api.path	文字列	[**必須**] API が利用可能である場所を指定します。 ● Camel 2.15 +– 相対パスのみを入力します : **host:port/context-path/api.path** の実行時に camel-swagger は絶対 api パスを計算します。 ● Camel 2.14.x- 絶対 api パス(**protocol://host:port/context-path/api.path**) を入力します。
api.termsOfServiceUrl	文字列	API 利用規約への URL を指定します。
api.title	文字列	[**必須**] アプリケーションのタイトルを指定します。
api.version	文字列	API のバージョンを指定します。デフォルトは **0.0.0** です。
base.path	文字列	[**必須**] REST サービスが利用できる場所を指定します。 ● Camel 2.15 +– 相対パスのみを入力します(**host:port/context-path/base.path**)。ランタイム時に、camel-swagger は絶対ベースパスを計算します。 ● Camel 2.14.x- 絶対ベースパス(**protocol://host:port/context-path/base.path**)を入力します。
cors	ブール値	CORS を有効にするかどうかを指定します。このパラメーターは、API ブラウザーの CORS のみを有効にします。REST サービスへのアクセスは有効化されません。デフォルトは **false** です。 このパラメーターを使用する代わりに、CorsFilter (x を参照)を使用することが推奨されます。
swagger.version	文字列	Swagger 仕様のバージョンを指定します。デフォルトは **1.2** です。

CorsFilter の使用

Swagger UI を使用して REST API を表示する場合は、CORS のサポートを有効にする必要がある場合があります。Swagger UI が REST API とは異なるホスト名/ポートでホストされ、実行されている場合は、オリジン(CORS)全体で REST リソースにアクセスする必要があります。CorsFilter は、CORS を有効にするために必要な HTTP ヘッダーを追加します。

すべてのリクエストに対して、CorsFilter は以下のヘッダーを設定します。

- *Access-Control-Allow-Origin = **

- *access-Control-Allow-Methods = GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH*

- *Access-Control-Max-Age = 3600*

- *Access-Control-Allow-Headers = Origin, Accept, X-Requested-With, Content-Type, Access-Control-Request-Method, Access-Control-Request-Headers*

これを使用するには、WAR デプロイメントで org.apache.camel.component.swagger.RestSwaggerCorsFilter を WEB-INF/web.xml ファイルに追加します。以下に例を示します。

```
<!-- Use CORs filter so people can use Swagger UI to browse and test the APIs -->

<filter>
    <filter-name>RestSwaggerCorsFilter</filter-name
    <filter-class>org.apache.camel.component.swagger.RestSwaggerCorsFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>RestSwaggerCorsFilter</filter-name>
    <url-pattern>/api-docs/*</url-pattern>
    <url-pattern>/rest/*</url-pattern>
</filter-mapping>
```

注記

この例は、非常に簡単な CORS フィルターを示しています。特定のクライアントに異なるヘッダー値を設定したり、特定のクライアントをブロックする場合などに、より高度なフィルターを使用してヘッダー値を設定する必要がある場合があります。

173.2. WAR デプロイメントの設定

WAR 実装の場合は、WEB-INF/web.xml ファイルでサーブレットオプションを設定する必要があります。

Camel 2.15.x

base.path および api.path パラメーターの両方に相対パスを使用します。

たとえば、あらゆる環境に Camel Swagger API サーブレットを設定するには、以下を実行します。

```
...
<servlet>

  <servlet-name>ApiDeclarationServlet</servlet-name>
  <servlet-class>org.apache.camel.component.swagger.DefaultCamelSwaggerServlet</servlet-
class>

<!-- Specify the base.path and the api.path values using relative notation
because the actual paths will be calculated at runtime as
http://server:port/contextpath/rest and http://server:port/contextpath/api-docs,
respectively -->
  <init-param>
    <param-name>base.path</param-name>
    <param-value>rest</param-value>
  </init-param>
   <init-param>
    <param-name>api.path</param-name>
    <param-value>api-docs</param-value>
  </init-param>

  <init-param>
    <param-name>api.version</param-name>
    <param-value>1.2.3</param-value>
  </init-param>
  <init-param>
    <param-name>api.title</param-name>
    <param-value>User Services</param-value>
  </init-param>
   <init-param>
    <param-name>api.description</param-name>
    <param-value>Camel Rest Example with Swagger that provides a User Rest
    service</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>

</servlet>

<!-- swagger api declaration -->
<servlet-mapping>
   <servlet-name>ApiDeclarationServlet</servlet-name>
   <url-pattern>/api-docs/*</url-pattern>
</servlet-mapping>
```

Camel 2.14.x

両方のサーブレット org.apache.camel.component.swagger.spring.SpringRestSwaggerApiDeclarationServlet および org.apache.camel.component.swagger.servletlistener.ServletListenerRestSwaggerApiDelaration Servlet が同じオプションをサポートします。

base.path および api.path パラメーターの両方に絶対パスを使用します。

たとえば、Spring の Camel Swagger API サーブレットを設定するには、以下を実行します。

```
...
<servlet>

  <servlet-name>ApiDeclarationServlet</servlet-name>
  <servlet-
class>org.apache.camel.component.swagger.spring.SpringRestSwaggerApiDeclarationServlet
</servlet-class>

  <init-param>
    <param-name>base.path</param-name>
    <param-value>http://localhost:8080/rest</param-value>
  </init-param>
   <init-param>
    <param-name>api.path</param-name>
    <param-value>http://localhost:8080/api-docs</param-value>
  </init-param>

  <init-param>
    <param-name>api.version</param-name>
    <param-value>1.2.3</param-value>
  </init-param>
  <init-param>
    <param-name>api.title</param-name>
    <param-value>User Services</param-value>
  </init-param>
   <init-param>
    <param-name>api.description</param-name>
    <param-value>Camel Rest Example with Swagger that provides a User Rest
    service</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>

</servlet>

<!-- swagger api declaration -->
<servlet-mapping>
   <servlet-name>ApiDeclarationServlet</servlet-name>
   <url-pattern>/api-docs/*</url-pattern>
</servlet-mapping>
```

173.3. OSGI デプロイメントの設定

org.apache.camel.component.swagger.DefaultCamelSwaggerServlet は、「サーブレット設定パラメーター」 で説明されているオプションをサポートします。

OSGi デプロイメントの場合、**blueprint.xml** ファイルでサーブレットオプションと **REST** 設定を設定する必要があります。以下に例を示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
         http://www.osgi.org/xmlns/blueprint/v1.0.0
         https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
         http://camel.apache.org/schema/blueprint
         http://camel.apache.org/schema/blueprint/camel-blueprint.xsd">


   <service interface="javax.servlet.http.HttpServlet">
      <service-properties>
         <entry key="alias" value="/api-docs/*"/>
         <entry key="init-prefix" value="init."/>
         <entry key="init.base.path" value="//localhost:8080/"/>
         <entry key="init.api.path" value="//localhost:8181/api-docs"/>
         <entry key="init.api.title" value="Camel Rest Example API"/>
         <entry key="init.api.version" value="1.2"/>
         <entry key="init.api.description"
            value="Camel Rest Example with Swagger that provides an User REST service"/>
      </service-properties>
      <bean class="org.apache.camel.component.swagger.DefaultCamelSwaggerServlet" />
   </service>

   <!--
    The namespace for the camelContext element in Blueprint
    is 'http://camel.apache.org/schema/blueprint'.

    While it is not required to assign id's to the <camelContext/> and <route/> elements,
    it is a good idea to set those for runtime management purposes (logging, JMX MBeans, ...)
   -->

   <camelContext id="log-example-context"
      xmlns="http://camel.apache.org/schema/blueprint">

      <restConfiguration component="jetty" port="8080"/>
      <rest path="/say">
         <get uri="/hello">
            <to uri="direct:hello"/>
         </get>
         <get uri="/bye" consumes="application/json">
            <to uri="direct:bye"/>
         </get>
         <post uri="/bye">
```

```
            <to uri="mock:update"/>
          </post>
        </rest>
        <route id="rte1-log-example">
          <from uri="direct:hello"/>
          <transform>
            <constant>Hello World</constant>
          </transform>
        </route>
        <route id="rte2-log-example">
          <from uri="direct:bye"/>
          <transform>
            <constant>Bye World</constant>
          </transform>
        </route>

      </camelContext>

    </blueprint>
```

service

service 要素は camel swagger サーブレット(<bean class="org.apache.camel.component.swagger.DefaultCamelSwaggerServlet"/>)を公開し、複数のサーブレットプロパティーを初期化します。

alias

alias プロパティーは、camel swagger サーブレットを /api-docs/ にバインドします。*

init-prefix

init-prefix プロパティーは、すべての camel swagger サーブレットプロパティーの接頭辞を init に設定します。これは、WAR 実装で web.xml 設定で init-param 要素を使用するのと似ています。

restConfiguration

camelContext では、restConfiguration 要素はポート 8080 の Web サーブレットとして Jetty を指定します。

rest

camelContext では、rest 要素は 2 つの REST エンドポイントを設定し、それらを以下の 2 つの route 要素で定義された camel エンドポイントにルーティングします。

第174章 SWAGGER JAVA

SWAGGER JAVA コンポーネント

Camel 2.16 以降で利用可能

Rest DSL は、Swagger を使用して REST サービスとその API を公開するために使用される camel-swagger-java モジュールと統合できます。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

Camel 2.16 以降、swagger コンポーネントは純粋に Java ベースであり、そのコンポーネントになります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-swagger-java</artifactId>
   <version>2.17.0.redhat-630xxx</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

camel-swagger-java モジュールはサーブレットとして、または REST コンポーネントから直接使用することもできます（サーブレットは必要ありません）。

Apache Camel ディストリビューションの例ディレクトリー(camel-example-swagger-cdi)を参照してください。

SWAGGER をサーブレットとして使用

以下のように web.xml で swagger を設定します。

```
 <servlet>

   <servlet-name>SwaggerServlet</servlet-name>
   <servlet-class>org.apache.camel.swagger.servlet.RestSwaggerServlet</servlet-class>

   <init-param>
     <!-- we specify the base.path using relative notation, that means the actual path will be
calculated at runtime as
```

```
        http://server:port/contextpath/rest -->
      <param-name>base.path</param-name>
      <param-value>rest</param-value>
    </init-param>
    <init-param>
      <!-- we specify the api.path using relative notation, that means the actual path will be
calculated at runtime as
        http://server:port/contextpath/api-docs -->
      <param-name>api.path</param-name>
      <param-value>api-docs</param-value>
    </init-param>

    <init-param>
      <param-name>api.version</param-name>
      <param-value>1.2.3</param-value>
    </init-param>
    <init-param>
      <param-name>api.title</param-name>
      <param-value>User Services</param-value>
    </init-param>
    <init-param>
      <param-name>api.description</param-name>
      <param-value>Camel Rest Example with Swagger that provides an User REST
service</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- swagger api -->
  <servlet-mapping>
    <servlet-name>SwaggerServlet</servlet-name>
    <url-pattern>/api-docs/*</url-pattern>
  </servlet-mapping>
```

REST-DSL での SWAGGER の使用

rest-dsl から swagger api を有効にするには、apiContextPath dsl を以下のように設定します。

```
public class UserRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        // configure we want to use servlet as the component for the rest DSL
        // and we enable json binding mode
        restConfiguration().component("netty4-http").bindingMode(RestBindingMode.json)
            // and output using pretty print
            .dataFormatProperty("prettyPrint", "true")
            // setup context path and port number that netty will use
            .contextPath("/").port(8080)
            // add swagger api-doc out of the box
            .apiContextPath("/api-doc")
                .apiProperty("api.title", "User API").apiProperty("api.version", "1.2.3")
                // and enable CORS
                .apiProperty("cors", "true");
```

```
        // this user REST service is json only
        rest("/user").description("User rest service")
            .consumes("application/json").produces("application/json")
            .get("/{id}").description("Find user by id").outType(User.class)
                .param().name("id").type(path).description("The id of the user to
get").dataType("int").endParam()
                .to("bean:userService?method=getUser(${header.id})")
            .put().description("Updates or create a user").type(User.class)
                .param().name("body").type(body).description("The user to update or
create").endParam()
                .to("bean:userService?method=updateUser")
            .get("/findAll").description("Find all users").outTypeList(User.class)
                .to("bean:userService?method=listUsers");
    }
}
```

オプション

***swagger** モジュールは、以下のオプションを使用して設定できます。サーブレットを使用して設定するには、上記のように **init-param** を使用します。**rest-dsl** で直接設定する場合は、**enableCORS, host,contextPath, dsl** などの適切な方法を使用します。**api.xxx** のオプションは、**apiProperty dsl** を使用して設定されます。*

オプション	タイプ	説明
cors	ブール値	CORS を有効にするかどうか。これにより、api ブラウザーの CORS のみが有効になり、REST サービスへの実際のアクセスは有効ではないことに注意してください。デフォルトは false です。このオプションを使用する代わりに、CorsFilte を使用することが推奨されます。詳細は、以下を参照してください。
swagger.version	文字列	Swagger 仕様バージョン。デフォルトは 2.0 です。
host	文字列	ホスト名を設定します。camel-swagger-java が設定されていない場合、名前は localhost ベースとして計算されます。

スキーマ	文字列	使用するプロトコルスキーム。"http,https" のように、複数の値をコンマで区切ることができます。デフォルト値は「http」です。このオプションは schemes という名前であるため、Camel 2.17 以降では **非推奨** となりました。
schemes	文字列	**Camel 2.17:** 使用するプロトコルスキーム"http,https" のように、複数の値をコンマで区切ることができます。デフォルト値は「http」です。
base.path	文字列	**必須**: REST サービスが利用できるベースパスを設定します。パスは相対(http/https で開始されない)であり、camel-swagger-java は実行時に絶対ベースパスを計算します。 `protocol://host:port/context-path/base.path`
api.path	文字列	API が利用できるパスを設定します（例：/api-docs）。パスは相対(http/https で開始されない)であり、camel-swagger-java は実行時に絶対ベースパスを計算します。 `protocol://host:port/context-path/api.path` そのため、相対パスの使用ははるかに簡単です。例については、上記を参照してください。
api.version	文字列	API のバージョン。デフォルトは 0.0.0 です。
api.title	文字列	アプリケーションのタイトル。
api.description	文字列	アプリケーションの簡単な説明。
api.termsOfService	文字列	API の利用規約への URL。
api.contact.name	文字列	連絡先に使用する個人または組織の名前
api.contact.email	文字列	API 関連の対応に使用するメール。

api.contact.url	文字列	API 関連の問い合わせ先の Web サイトへの URL。
api.license.name	文字列	API に使用されるライセンス名。
api.license.url	文字列	API に使用されるライセンスへの URL。
apiContextIdListing	boolean	REST サービスを持つ JVM のすべての CamelContext 名の一覧表示を許可するかどうか。有効にすると、api-doc のルートパスにより、すべてのコンテキストが一覧表示されます。無効にするとコンテキスト ID が表示されず、api-doc のルートパスは現在の CamelContext を一覧表示します。デフォルトは false です。
apiContextIdPattern	文字列	コンテキストリストに示されている CamelContext 名をフィルターできるようにするパターン。このパターンは正規表現と * をワイルドカードとして使用します。Intercep で使用されるのと同じパターンの一致

CORSFILTER

swagger ui を使用して REST API を表示する場合は、CORS のサポートを有効にする必要がある場合があります。これは、swagger ui がホストされ、実際の RESTapis 以外のホスト名/ポートで実行されている場合に必要になります。これを行う場合は、swagger ui が元の CORS (CORS)全体で REST リソースにアクセスできるようにする必要があります。CorsFilter は、CORS を有効にするために必要な HTTP ヘッダーを追加します。

CORS を使用するには、以下のフィルター org.apache.camel.swagger.servlet.RestSwaggerCorsFilter を web.xml に追加します。

```
<!-- enable CORS filter so people can use swagger ui to browse and test the apis -->
<filter>
  <filter-name>RestSwaggerCorsFilter</filter-name>
  <filter-class>org.apache.camel.swagger.rest.RestSwaggerCorsFilter</filter-class>
</filter>


<filter-mapping>
  <filter-name>RestSwaggerCorsFilter</filter-name>
```

```
  <url-pattern>/api-docs/*</url-pattern>
  <url-pattern>/rest/*</url-pattern>
</filter-mapping>
```

CorsFilter は、すべてのリクエストに以下のヘッダーを設定します。

- Access-Control-Allow-Origin = *
- access-Control-Allow-Methods = GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH
- Access-Control-Max-Age = 3600
- Access-Control-Allow-Headers = Origin, Accept, X-Requested-With, Content-Type, Access-Control-Request-Method, Access-Control-Request-Headers

これは非常に単純な CORS フィルターであることに注意してください。特定のクライアントに異なるヘッダー値を設定するには、より高度なフィルターを使用してヘッダー値を設定する必要がある場合があります。または、特定のクライアントなどをブロックします。

CONTEXTIDLISTING ENABLED

contextIdListing が有効になっている場合、そのタスクは、同じ JVM で実行されている CamelContext をすべて検出します。これらのコンテキストはルートパスに一覧表示されます（例：'/api-docs'）は、名前の簡単なリストとして json 形式になります。swagger のドキュメントにアクセスするには、context-path に 'api-docs/myCamel' などの Camel コンテキスト ID を追加する必要があります。apiContextIdPattern オプションは、このリストの名前をフィルターするために使用できます。

JSON または YAML

Camel 2.17 以降で利用可能

camel-swagger-java モジュールは、設定設定なしで JSon と Yaml の両方をサポートします。リクエスト URL に /swagger.json または /swagger.yaml のいずれかを使用して、返すものを指定できます。指定がない場合は、json または yaml を許可するかどうかを検出するために HTTP Accept ヘッ

ダーが使用されます。両方が受け入れられるか、または何も受け入れられないと、json がデフォルトの形式として返されます。

例

Apache Camel ディストリビューションでは、この Swagger コンポーネントの使用を示す camel-example-swagger-cdi および camel-example-swagger-java が含まれています。

第175章 TEST

テストコンポーネント

テスト コンポーネントは Mock コンポーネントを拡張し、起動時に別のエンドポイントからメッセージをプルするようにサポートし、基礎となる Mock エンドポイントに想定されるメッセージ本文を設定します。つまり、ルートでテストエンドポイントを使用し、それに到達するメッセージは、他の場所から抽出される予想されるメッセージと暗黙的に比較されます。

そのため、たとえば、予想されるメッセージボディーのセットをファイルとして使用できます。これにより、適切に設定された Mock エンドポイントが設定されます。これは、受信したメッセージが予想されるメッセージの数に一致し、メッセージペイロードが等しい場合にのみ有効です。

Camel 2.8 以前を使用している場合、Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-spring</artifactId>
   <version>2.17.0.redhat-630xxx</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

Camel 2.9 以降では、Test コンポーネントは camel-core に直接提供されます。

URI 形式

```
test:expectedMessagesEndpointUri
```

expectedMessagesEndpointUri は、テストを開始する前に想定されるメッセージボディーがプルされる他の Component URI を参照します。

URI オプション

名前	デフォルト値	説明
timeout	**2000**	**Camel 2.12:** URI からメッセージボディーをポーリングする際に使用するタイムアウト。
anyOrder	**false**	**Camel 2.17:** 予想されるメッセージが同じ順序で到達するか、または任意の順序で到達できるかどうか。
split	**false**	**Camel 2.17:** 有効にすると、テストエンドポイントからロードされたメッセージは **\n**, **\r** delimiters（新しい行）を使用して分割されるため、各行が想定されるメッセージになります。たとえば、file エンドポイントを使用して、各行が想定されるメッセージである ファイルを読み込むには、以下を実行します。
delimiter	\n|\r	**Camel 2.17:** 分割が有効な場合に使用するスプリット区切り文字。デフォルトでは、区切り文字は新しい行ベースです。区切り文字は正規表現にすることができます。

例

たとえば、以下のようにテストケースを作成できます。

```
from("seda:someEndpoint").
  to("test:file://data/expectedOutput?noop=true");
```

テストによって MockEndpoint.assertIsSatisfied (camelContext)メソッド が呼び出されると、テストケースが必要なアサーションを実行します。

以下は、Mock と Spring を使用したテストケース例と Spring XML の例です。

テストエンドポイントに他の期待を設定する方法は、Mock コンポーネント を参照してください。

第176章 TIMER

タイマーコンポーネント

timer: コンポーネントは、タイマーが実行されるとメッセージエクスチェンジを生成するために使用されます。このエンドポイントからイベントのみを消費できます。

URI 形式

```
timer:name[?options]
```

ここでの name は Timer オブジェクトの名前で、これはエンドポイント間で作成および共有されます。したがって、すべてのタイマーエンドポイントに同じ名前を使用する場合は、1 つの Timer オブジェクトとスレッドのみが使用されます。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

注記： 生成されたエクスチェンジの IN ボディーは null です。したがって、exchange.getIn().getBody() は null を返します。

高度なスケジューラー

より詳細スケジューリングをサポートする Quartz コンポーネントも参照してください。

人間が分かりやすい形式で時間を指定

Camel 2.3 以降では、人間が分かりやすい構文 で時間を指定できます。

オプション

名前	デフォルト値	説明
time	**null**	**java.util.Date** は **最初** のイベントを生成する必要があります。URI を使用する場合、パターンは **yyyy-MM-dd HH:mm:ss** または **yyyy-MM-dd'T'HH:mm:ss** です。

pattern	**null**	URI 構文を使用して time オプションの設定に使用するカスタムの **Date** パターンを指定できます。
period	**1000**	0 を超える場合は、**期間**（ミリ秒単位）ごとに定期的なイベントを生成します。
delay	**1000**	最初のイベントが生成されるまで待機する時間（ミリ秒単位）。**time** オプションと併用しないでください。**Camel 2.17 以降**では、負の遅延を指定できます。このシナリオでは、タイマーがイベントをできるだけ早く生成し、実行します。
fixedRate	**false**	イベントは、指定された期間で区切られた、約一定間隔で行われます。
daemon	**true**	タイマーエンドポイントに関連付けられたスレッドがデーモンとして実行されるかどうかを指定します。
repeatCount	**0**	**camel 2.8:** 実行の最大数を指定します。そのため、これを 1 に設定するとタイマーは一度だけ実行されます。これを 5 に設定した場合、5 回だけ実行されます。0 または負の値を設定すると、無制限に実行されます。

エクスチェンジプロパティー

*タイマーが実行されると、以下の情報をプロパティーとして **Exchange** に追加します。*

名前	タイプ	説明
Exchange.TIMER_NAME	文字列	**name** オプションの値。
Exchange.TIMER_TIME	日付	**time** オプションの値。
Exchange.TIMER_PERIOD	**long**	**period** オプションの値。
Exchange.TIMER_FIRED_TIME	日付	コンシューマーが実行した時間。

Exchange.TIMER_COUNTER	**Long**	Camel 2.8: 現在の fire カウンター。1 から始まります。

メッセージヘッダー

タイマーが実行されると、以下の情報をヘッダーとして IN メッセージに追加します。

名前	タイプ	説明
Exchange.TIMER_FIRED_TIME	**java.util.Date**	コンシューマーが実行される時間

例

60 秒ごとにイベントを生成するルートを設定するには、以下を実行します。

```
from("timer://foo?fixedRate=true&period=60000").to("bean:myBean?method=someMethodName");
```

ヒント

60000 の代わりに、より読みやすい(period=60s)を指定できます。

上記のルートはイベントを生成し、JNDI や Spring などの レジストリー の myBean という Bean で someMethodName メソッドを呼び出します。

Spring DSL のルートは以下のようになります。

```
<route>
  <from uri="timer://foo?fixedRate=true&amp;period=60000"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

できるだけ早く実行する

Camel 2.17: できるだけ早く Camel ルートでメッセージを生成することができます。負の遅延を使用できます。

```
<route>
  <from uri="timer://foo?delay=-1"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

*このようにして、タイマーは即座にメッセージを実行します。**repeatCount** パラメーターを負の遅延と共に指定して、固定数に達した後にメッセージの実行を停止することもできます。**repeatCount** を指定しないと、ルートが停止するまでタイマーがメッセージを実行します。*

1 度だけ発行

Camel 2.8 から利用可能

*ルートの起動時になど、**Apache Camel** ルートでメッセージを **1** 度だけ実行することをお勧めします。これを行うには、以下のように **repeatCount** オプションを使用します。*

```
<route>
  <from uri="timer://foo?repeatCount=1"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

以下も参照してください。

- *Quartz*

第177章 TWITTER

TWITTER

Camel 2.10 以降で利用可能

Twitter コンポーネントは、Twitter4J をカプセル化することで、Twitter API で最も便利な機能を有効にします。これにより、タイムライン、ユーザー、傾向、およびダイレクトメッセージの直接、ポーリング、またはイベント駆動型の消費が可能になります。また、ステータスの更新またはダイレクトメッセージとしてのメッセージの生成にも対応しています。

Twitter では、すべてのクライアントアプリケーション認証に OAuth を使用する必要があります。アカウントで camel-twitter を使用するには、https://dev.twitter.com/apps/new で Twitter 内に新しいアプリケーションを作成し、アプリケーションにアカウントへのアクセスを許可する必要があります。最後に、アクセストークンおよびシークレットを生成します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-twitter</artifactId>
    <version>${camel-version}</version>
</dependency>
```

URI 形式

```
twitter://endpoint[?options]
```

TWITTERCOMPONENT:

twitter コンポーネントは、使用する前に設定する必要がある Twitter アカウント設定で設定できます。これらのオプションは、エンドポイントで直接設定することもできます。

オプション	説明
consumerKey	コンシューマーキー

consumerSecret	コンシューマーシークレット
accessToken	アクセストークン
accessTokenSecret	アクセストークンのシークレット

コンシューマーエンドポイント：

1 つのルートエクスチェンジで List を返すエンドポイントではなく、camel-twitter は返されたオブジェクトごとにルートエクスチェンジを 1 つ作成します。たとえば、timeline/home がステータス 5 回作成されると、ルートは 5 回実行されます(Statusごとに 1 回)。

エンドポイント	コンテキスト	ボディタイプ	注意
directmessage	直接、ポーリング	**twitter4j.DirectMessage**	
search	直接、ポーリング	**twitter4j.Status**	
streaming/filter	イベント、ポーリング	**twitter4j.Status**	
streaming/sample	イベント、ポーリング	**twitter4j.Status**	
streaming/user	イベント、ポーリング	**twitter4j.Status** **twitter4j.DirectMessage** **twitter4j.UserList**	Camel 2.16: 保護されたユーザーおよびアカウントからツイートを受信するため。 Camel 2.17: DirectMessages, Favorites, Lists, Following events がサポートされるようになりました。
timeline/home	直接、ポーリング	**twitter4j.Status**	
timeline/mentions	直接、ポーリング	**twitter4j.Status**	
timeline/retweetsofme	直接、ポーリング	**twitter4j.Status**	
timeline/user	直接、ポーリング	**twitter4j.Status**	

プロデューサーエンドポイント：

エンドポイント	ボディタイプ
directmessage	**String**
search	**List<twitter4j.Status>**
timeline/user	**String**

URI オプション

名前	デフォルト値	説明
type	**直接的な**	直接、イベント、またはポーリング
delay		ポーリング間の遅延。デフォルト値は 60 秒です。**Camel 2.16** 以前の値は秒単位です。**Camel 2.17** 以降のミリ秒単位。
consumerKey	**null**	Consumer Key。代わりに **TwitterComponent** レベルで設定することもできます。
consumerSecret	**null**	コンシューマーシークレット。代わりに **TwitterComponent** レベルで設定することもできます。
accessToken	**null**	アクセストークン。代わりに **TwitterComponent** レベルで設定することもできます。
accessTokenSecret	**null**	アクセストークンシークレット。代わりに **TwitterComponent** レベルで設定することもできます。
user	**null**	ユーザー名。ユーザーのタイムライン消費、ダイレクトメッセージの実稼働などに使用されます。

locations	**null**	'lat,lon;lat,lon;...' lat/lons のペアによって作成されたバインドされたボックス。ストリーミング/フィルターに使用できます。
keywords	**null**	'foo1,foo2,foo3...' 検索およびストリーミング/フィルターに使用できます。OR などで 検索するためのキーワード構文については、高度 な検索 を参照してください。
userIds	**null**	'username,username...' ストリーミング/フィルターに使用できます。
filterOld	**true**	以前にポーリングされた古いツイートを除外します。この状態はメモリーにのみ保存され、最後のツイート ID に基づいています。**Camel 2.11.0 以降**、検索プロデューサーはこのオプションをサポートします
sinceId	**1**	**Camel 2.11.0:** ツイートをプルするために使用される最後のツイート ID。これは、長時間実行した後に camel ルートが再起動する場合に便利です。
lang	**null**	**Camel 2.11.0:** 検索に使用される lang 文字列 ISO_639-1
count	**null**	**Camel 2.11.0:** ページごとの結果の制限数。
numberOfPages	**1**	**Camel 2.11.0:** camel-twitter が消費するページ結果の数。
httpProxyHost	**null**	**Camel 2.12.3:** camel-twitter に使用できる http プロキシーホスト。
httpProxyPort	**null**	**Camel 2.12.3:** camel-twitter に使用できる http プロキシーポート。
httpProxyUser	**null**	**Camel 2.12.3:** camel-twitter に使用できる http プロキシーユーザー。
httpProxyPassword	**null**	**Camel 2.12.3:** camel-twitter に使用できる http プロキシーパスワード。

latitude		**Camel 2.16:** latitude による検索に、ストリーム以外の地理的な検索で使用されます。**longitude**、**latitude**、**radius**、および **distanceMetric** のすべてのオプションを設定する必要があります。
longitude		**Camel 2.16:** ストリーム以外の地理的な検索により、長期間検索に使用されます。**longitude**、**latitude**、**radius**、および **distanceMetric** のすべてのオプションを設定する必要があります。
radius		**Camel 2.16:** radius で検索するために、ストリーム以外の地理的な検索で使用されます。**longitude**、**latitude**、**radius**、および **distanceMetric** のすべてのオプションを設定する必要があります。
distanceMetric	**km**	**Camel 2.16:** ストリーム以外の地理的な検索で使用され、設定されたメトリクスを使用して radius で検索します。この単位は、ミリ秒の場合は **mi**、kilometer の場合は **km** となります。**longitude**、**latitude**、**radius**、および **distanceMetric** のすべてのオプションを設定する必要があります。

メッセージヘッダー

名前	説明
CamelTwitterKeywords	このヘッダーは、検索キー単語を動的に変更するために検索プロデューサーによって使用されます。
CamelTwitterSearchLanguage	**Camel 2.11.0:** このヘッダーは、検索エンドポイントの検索言語を動的に設定する **lang** のオプションを上書きできます。

CamelTwitterCount	Camel 2.11.0 このヘッダーは、返される最大 twitters を設定する **count** のオプションを上書きできます。
CamelTwitterNumberOfPages	Camel 2.11.0 このヘッダーは、twitter を返すページ数を設定する **numberOfPages** のオプションを収束できます。
CamelTwitterNumberOfPages	Camel 2.17.0: 受信したイベントのタイプ(**org.apache.camel.component.twitter.consumer.TwitterEventType**を参照してください)。
CamelTwitterNumberOfPages	Camel 2.17.0: 当事者を特定します。
CamelTwitterNumberOfPages	Camel 2.17.0: 当事者ロールを識別します。

メッセージボディー

すべてのメッセージ本文は、Twitter4J API によって提供されるオブジェクトを使用します。

API 流量制御

Twitter4J によってカプセル化された Twitter REST API は、API Rate Limiting の対象となります。メソッドごとの制限は、API Rate Limits に関するドキュメントを参照してください。そのページに一覧表示されていないエンドポイント/リソースは、ウィンドウごとに割り当てられたユーザーごとに 15 要求にデフォルト設定されることに注意してください。

TWITTER プロファイル内でステータス更新を作成するには、このプロデューサーを STRING ボディーを送信します。

```
from("direct:foo")
  .to("twitter://timeline/user?consumerKey=[s]&consumerSecret=[s]&accessToken=
[s]&accessTokenSecret=[s]);
```

60 秒ごとに、自宅のタイムラインのすべてのステータスをポーリングするには、以下を実行します。

```
from("twitter://timeline/home?type=polling&delay=60&consumerKey=[s]&consumerSecret=
[s]&accessToken=[s]&accessTokenSecret=[s]")
  .to("bean:blah");
```

キーワード 'CAMEL' ですべてのステータスを検索するには、以下を実行します。

```
from("twitter://search?type=direct&keywords=camel&consumerKey=[s]&consumerSecret=
[s]&accessToken=[s]&accessTokenSecret=[s]")
  .to("bean:blah");
```

静的キーワードでプロデューサーを使用した検索

```
from("direct:foo")
  .to("twitter://search?keywords=camel&consumerKey=[s]&consumerSecret=
[s]&accessToken=[s]&accessTokenSecret=[s]");
```

ヘッダーから動的キーワードを持つプロデューサーを使用した検索

バーヘッダーには検索するキーワードがあるため、この値を CamelTwitterKeywords ヘッダーに割り当てることができます。

```
from("direct:foo")
  .setHeader("CamelTwitterKeywords", header("bar"))
  .to("twitter://search?consumerKey=[s]&consumerSecret=[s]&accessToken=
[s]&accessTokenSecret=[s]");
```

例

Twitter Websocket の例 も参照してください。

- Twitter Websocket の例

第178章 UNDERTOW

UNDERTOW コンポーネント

Camel 2.16 以降で利用可能

undertow コンポーネントは、HTTP リクエストを使用し、生成するための HTTP ベースのエンドポイントを提供します。つまり、Undertow コンポーネントは単純な Web サーバーとして動作します。Undertow は HTTP クライアントとしても使用でき、Camel でプロデューサーとして使用することもできます。

Maven ユーザーは、このコンポーネントの以下の依存関係を pom.xml に追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-undertow</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI 形式

```
undertow:http://hostname[:port][/resourceUri][?options]
```

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

オプション

名前	デフォルト値	説明
httpMethodRestrict		GET/POST/PUT などの HttpMethod が一致する場合にのみ消費できるようにするために使用されます。複数のメソッドはコンマで区切って指定できます。
matchOnUriPrefix		完全に一致するものが見つからない場合に、コンシューマーが URI 接頭辞と一致することでターゲットコンシューマーの検索を試みるかどうか。

headerFilterStrategy		カスタムの HeaderFilterStrategy を使用して、Camel メッセージとの間でヘッダーをフィルタリングします。
sslContextParameters		**SSLContextParameters** オブジェクトを使用してセキュリティーを設定するには、以下を行います。Security Guide の Configuring Transport Security for Camel Components の章を参照してください。
throwExceptionOnFailure		オプションが true の場合、HttpProducer は Exchange.HTTP_URI ヘッダーを無視し、リクエストにエンドポイントの URI を使用します。また、オプション throwExceptionOnFailure を false に設定して、プロデューサーがすべての障害応答を返信するようにすることもできます。
transferException		リモートサーバーからの応答が失敗した場合に HttpOperationFailedException を出力することを無効にするオプション。これにより、HTTP ステータスコードに関係なくすべての応答を取得できます。
undertowHttpBinding		カスタムの UndertowHttpBinding を使用して Camel メッセージと undertow 間のマッピングを制御します。
keepAlive	true	**Camel 2.16.1: Producer のみ：** 非アクティブのためにソケットが閉じられないように設定
tcpNoDelay	true	**Camel 2.16.1: Producer only:**Setting to improve TCP protocol performance (Camel 2.16.1: プロデューサーのみ：TCP プロトコルのパフォーマンスを改善するための設定)
reuseAddresses	true	**Camel 2.16.1: プロデューサーのみ：**ソケットの多重化を容易にするための設定

options.XXX		**Camel 2.16.1: Producer only:**追加のチャネルオプションを設定します。使用できるオプションは、**org.xnio.Options** で定義されています。エンドポイント URI から設定するには、"option.close-abort=true&option.send-buffer=8192" などの各オプションの前に options. を付けます。
enableOptions	false	**Camel 2.17:** この Undertow コンシューマーに対して HTTP OPTIONS を有効にするかどうかを指定します。デフォルトでは、OPTIONS はオフになっています。

メッセージヘッダー

Camel は HTTP コンポーネントと同じメッセージヘッダーを使用します。Camel 2.2 では、Exchange.HTTP_CHUNKED,CamelHttpChunked ヘッダーを使用して camel-undertow コンシューマーでチェンドエンコーディングをオンまたはオフにします。

Camel は すべて の request.parameter および request.headers も設定します。たとえば、URL http://myserver/myserver?orderid=123 を持つクライアントリクエストの場合、エクスチェンジには値が 123 の orderid という名前のヘッダーが含まれます。

コンポーネントオプション

UndertowComponent では、以下のオプションを提供します。

名前	デフォルト値	説明
undertowHttpBinding		カスタムの UndertowHttpBinding を使用して Camel メッセージと undertow 間のマッピングを制御します。
httpConfiguration		共有 HttpConfiguration を基本設定として使用します。

プロデューサーの例

以下は、HTTP リクエストを既存の HTTP エンドポイントに送信する方法の基本的な例です。

Java DSL で

```
from("direct:start").to("undertow:http://www.google.com");
```

Spring XML で または を使用します。

```
<route>
   <from uri="direct:start"/>
   <to uri="undertow:http://www.google.com"/>
<route>
```

コンシューマーの例

この例では、http://localhost:8080/myapp/myservice で HTTP サービスを公開するルートを定義します。

```
<route>
  <from uri="undertow:http://localhost:8080/myapp/myservice"/>
  <to uri="bean:myBean"/>
</route>
```

URL で localhost を指定すると、Camel はローカルの TCP/IP ネットワークインターフェイスでのみエンドポイントを公開するため、操作するマシンからアクセスすることはできません。

特定のネットワークインターフェイスで Jetty エンドポイントを公開する必要がある場合は、このインターフェイスの数値の IP アドレスをホストとして使用する必要があります。すべてのネットワークインターフェイスで Jetty エンドポイントを公開する必要がある場合は、0.0.0.0 アドレスを使用する必要があります。

第179章 検証

検証コンポーネント

Validation コンポーネントは JAXP Validation API を使用してメッセージボディーの XML 検証を実行し、サポートされる任意の XML スキーマ言語（デフォルトは XML スキーマ）に基づきます。

Jing コンポーネントは、以下の便利なスキーマ言語もサポートしていることに注意してください。

- RelaxNG Compact 構文
- RelaxNG XML 構文

MSV コンポーネントは、RelaxNG XML 構文もサポートします。

URI 形式

```
validator:someLocalOrRemoteResource
```

someLocalOrRemoteResource は、クラスパス上のローカルリソースへの URL、または検証する XSD を含むファイルシステムのリモートリソースまたはリソースへの完全な URL になります。以下に例を示します。

- msv:org/foo/bar.xsd
- msv:file:../foo/bar.xsd
- msv:http://acme.com/cheese.xsd
- validator:com/mypackage/myschema.xsd

Camel 2.8 以前を使用している場合、Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring</artifactId>
  <version>2.17.0.redhat-630xxx</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

Camel 2.9 以降では、Validation コンポーネントは camel-core に直接提供されます。

オプション

オプション	デフォルト	説明
resourceResolverFactory	**DefaultValidatorResourceResolverFactory**	Camel 2.17: エンドポイントごとにリソースリゾルバーを作成する **org.apache.camel.component.validator.ValidatorResourceResolverFactory** を参照します。デフォルトの実装では、デフォルトのリソースリゾルバー **org.apache.camel.component.validator.DefaultLSResourceResolver** を作成するエンドポイントごとに **org.apache.camel.component.validator.DefaultLSResourceResolver** のインスタンスを作成します。デフォルトのリソースリゾルバーは、クラスパスとファイルシステムからスキーマファイルを読み取ります。このオプションは、**resourceResolver** オプションではなく、リソースリゾルバーがエンドポイントで指定されたルートスキーマドキュメントのリソース URI に依存する場合に使用されます。たとえば、デフォルトのリソースリゾルバーを拡張する場合などです。このオプションは Validator コンポーネントでも利用できるため、リソースリゾルバーファクトリーはすべてのエンドポイントに対して 1 回のみ設定できます。
resourceResolver	**null**	Camel 2.9: レジストリーの **org.w3c.dom.ls.LSResourceResolver** への参照。

useDom	**false**	Apache Camel 2.0: **DOMSource**/{{DOMResult}} または **SaxSource**/{{SaxResult}} をバリデーターが使用するかどうか。
useSharedSchema	**true**	Camel 2.3: **スキーマ** インスタンスを共有すべきかどうか。このオプションは、JDK 1.6.x バグ を回避するために導入されています。Xerces にはこの問題はありません。
failOnNullBody	**true**	Camel 2.9.5/2.10.3: ボディーが存在しない場合に失敗するかどうか。
headerName	**null**	Camel 2.11: メッセージボディーではなくヘッダーに対して検証するには、以下を行います。
failOnNullHeader	**true**	Camel 2.11: ヘッダーに対して検証する際にヘッダーが存在しない場合に失敗するかどうか。

例

以下の 例 は、エンドポイント direct:start からのルートを設定する方法を示しています。これは、XML が（クラスパスで提供されている）指定されたスキーマと一致するかどうかに基づいて、2 つのエンドポイント(mock:valid または mock:invalid)のいずれかになります。

```
<route>
   <from uri="direct:start"/>
   <doTry>
      <to uri="validator:org/apache/camel/component/validator/schema.xsd"/>
      <to uri="mock:valid"/>
      <doCatch>
         <exception>org.apache.camel.ValidationException</exception>
         <to uri="mock:invalid"/>
      </doCatch>
      <doFinally>
         <to uri="mock:finally"/>
      </doFinally>
   </doTry>
</route>
```

ADVANCED: JMX METHOD CLEARCACHEDSCHEMA

Camel 2.17 以降、バリデーターエンドポイント内のキャッシュされたスキーマが消去され、JMX 操作 clearCachedSchema を使用して次のプロセス呼び出しで再読み取りされるように強制できます。この方法を使用して、キャッシュをプログラム的にクリアすることもできます。このメソッドは、ValidatorEndpoint クラスで利用できます。

第180章 VELOCITY

VELOCITY

velocity: コンポーネントを使用すると、Apache Velocity テンプレートを使用してメッセージを処理できます。これは、Templating を使用してリクエストの応答を生成する場合に理想的です。

URI 形式

```
velocity:templateName[?options]
```

templateName は、呼び出すテンプレートのクラスパスローカル URI、またはリモートテンプレートの完全な URL（例： file://folder/myfile.vm）に置き換えます。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

オプション	デフォルト	説明
allowContextMapAll (producer)	**false**	コンテキストマップが前詳細へのアクセスを許可するかどうかを設定します。デフォルトでは、メッセージの本文とヘッダーにのみアクセスできます。このオプションは、現在の Exchange および CamelContext へのフルアクセスに対して有効にできます。これを行うと、CamelContext API の全機能へのアクセスが開かれるため、潜在的なセキュリティーリスクが発生します。
allowTemplateFromHeader (producer)	**false**	ヘッダーのリソーステンプレートの使用を許可するかどうか (デフォルトは false)。**このオプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼されていないコンテンツやユーザー派生コンテンツが含まれている場合、これは最終的にエンドアプリケーションの信頼性および完全性に影響を与える可能性があるため、このオプションは注意して使用してください。**

loaderCache	**true**	Velocity ベースのファイルローダーキャッシュ。
contentCache	**true**	ロード時のリソースコンテンツのキャッシュ。注記：Camel 2.9 でキャッシュされたリソースコンテンツは、エンドポイントの **clearContentCache** 操作を使用して JMX 経由でクリアできます。
encoding	**null**	リソースコンテンツの文字エンコーディング。
propertiesFile	**null**	Camel 2.1 の新機能：VelocityEngine の初期化に使用されるプロパティーファイルの URI。

メッセージヘッダー

***velocity** コンポーネントは、メッセージに **2** つのヘッダーを設定します（これらは独自に設定できず、**Camel 2.1 velocity** コンポーネントからもこれらのヘッダーを設定しないため、動的テンプレートのサポートに副作用が生じます）。*

ヘッダー	説明
CamelVelocityResourceUri	**String** オブジェクトとしての **templateName**。
CamelVelocitySupplementalContext	Camel 2.16: 使用された **VelocityContext** に追加情報を追加するには、以下を行います。このヘッダーの値は、追加するキー/値を持つ **Map** である必要があります（同じ名前の既存のキーを上書きします）。これは、velocity エンドポイントで再利用する一般的なキー/値の一部を設定するために使用できます。

***Velocity** 評価中に設定されたヘッダーはメッセージに返され、ヘッダーとして追加されます。その後、事実上 **Velocity** から **Message** に値を返すことができます。たとえば、**Velocity** テンプレート **template.tm** でヘッダー値 **fruit** を設定するには、次のコマンドを実行します。*

```
$in.setHeader("fruit", "Apple")
```

***fruit** ヘッダーが **message.out.headers** からアクセス可能になりました。*

VELOCITY コンテキスト

Apache Camel は Velocity コンテキストで交換情報を提供します（マップのみ）。Exchange は以下のように転送されます。

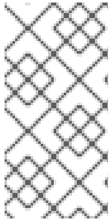
key	value
exchange	**Exchange** 自体。
exchange.properties	**Exchange** プロパティー。
ヘッダー	In メッセージのヘッダー。
camelContext	Camel Context インスタンス。
request	In メッセージ。
in	In メッセージ。
ボディー	In メッセージのボディー。
out	Out メッセージ(InOut メッセージ交換パターンにのみ有効)。
response	Out メッセージ(InOut メッセージ交換パターンにのみ有効)。

Camel-2.14 以降、以下のようにメッセージヘッダー CamelVelocityContext を設定して、カスタムの Velocity Context を独自に設定できます。

```
VelocityContext velocityContext = new VelocityContext(variableMap);
exchange.getIn().setHeader("CamelVelocityContext", velocityContext);
```

ホットリロード

Velocity テンプレートリソースは、デフォルトでは、ファイルとクラスパスリソース（展開された jar）の両方に対してホットリロードが可能です。contentCache=true を設定すると、Apache Camel はリソースを 1 度だけロードするため、ホットリロードはできません。このシナリオは、リソースが変更されない場合に実稼働環境で使用することができます。



注記

velocity は、ref: を接頭辞として使用して、レジストリーからリソースファイルをロードできます。

動的テンプレート

Camel 2 .1 Camel では利用可能な 2 つの ヘッダーで、テンプレートまたはテンプレートコンテンツ自体の異なるリソースの場所を定義できます。これらのヘッダーのいずれかが設定されている場合、Camel は設定されたエンドポイントでこれを使用します。これにより、ランタイム時に動的テンプレートを指定できます。

ヘッダー	タイプ	説明
CamelVelocityResourceUri	文字列	**Camel 2.1:** 設定されたエンドポイントの代わりに使用するテンプレートリソースの URI。
CamelVelocityTemplate	文字列	**Camel 2.1:** 設定されたエンドポイントの代わりに使用するテンプレート。

サンプル

たとえば、以下のようなものを使用できます。

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

Velocity テンプレートを使用して InOut メッセージエクスチェンジ(JMSReplyTo ヘッダーがある)のメッセージへの応答を形成するには、以下を実行します。

InOnly を使用してメッセージを消費し、別の宛先に送信する場合は、以下のルートを使用できます。

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

また、コンテンツキャッシュを使用するには、たとえば .vm テンプレートは変更されません。

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm?contentCache=true").
  to("activemq:Another.Queue");
```

また、ファイルベースのリソースは以下のようになります。

```
from("activemq:My.Queue").
  to("velocity:file://myfolder/MyResponse.vm?contentCache=true").
  to("activemq:Another.Queue");
```

Camel 2.1 では、以下のように、コンポーネントがヘッダーを介して動的に使用するテンプレートを指定できます。

```
from("direct:in").
  setHeader("CamelVelocityResourceUri").constant("path/to/my/template.vm").
  to("velocity:dummy?allowTemplateFromHeader=true");
```

Camel 2.1 では、以下のように、コンポーネントはヘッダーを介して動的に使用する必要があるヘッダーとしてテンプレートを直接指定できます。

```
from("direct:in").
  setHeader("CamelVelocityTemplate").constant("Hi this is a velocity template that can do
templating ${body}").
  to("velocity:dummy?allowTemplateFromHeader=true");
```

警告

allowTemplateFromHeader オプションを有効にすると、セキュリティーの問題があります。たとえば、ヘッダーに信頼できないコンテンツまたはユーザー派生コンテンツが含まれる場合、これは最終的に、エンドアプリケーションの確実性と整合性に及ぼす可能性があるため、このオプションを使用してください。

電子メールのサンプル

この例では、注文確認メールに Velocity テンプレートを使用します。メールテンプレートは、以下のように Velocity に配置されます。

```
Dear ${headers.lastName}, ${headers.firstName}
```

```
Thanks for the order of ${headers.item}.

Regards Camel Riders Bookstore
${body}
```

Java コード：

```
private Exchange createLetter() {
    Exchange exchange = context.getEndpoint("direct:a").createExchange();
    Message msg = exchange.getIn();
    msg.setHeader("firstName", "Claus");
    msg.setHeader("lastName", "Ibsen");
    msg.setHeader("item", "Camel in Action");
    msg.setBody("PS: Next beer is on me, James");
    return exchange;
}

@Test
public void testVelocityLetter() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedBodiesReceived("Dear Ibsen, Claus\n\nThanks for the order of Camel in
Action.\n\nRegards Camel Riders Bookstore\nPS: Next beer is on me, James");

    template.send("direct:a", createLetter());

    mock.assertIsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {

from("direct:a").to("velocity:org/apache/camel/component/velocity/letter.vm").to("mock:result
");
        }
    };
}
```

第181章 VERTX

VERTX コンポーネント

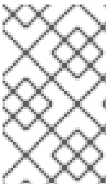
Camel 2.12 以降で利用可能

vertx コンポーネントは、Verx EventBus の使用用です。http://vertx.io/

vertx EventBus は JSON イベントを送信し、受信します。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-vertx</artifactId>
   <version>2.17.0.redhat-630xxx</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```



注記

Camel 2.16 以降 では、は実行時に Java 8 を必要とする Vertx 3 を使用します。

URI 形式

```
vertx:channelName[?options]
```

以下の形式で URI にクエリーオプションを追加できます。 ?option=value&option=value&...

オプション

名前	デフォルト値	説明
pubSub	**false**	Camel 2.12.3: vertx エンドポイントに送信するときに、ポイントの代わりにパブリッシュ/サブスクライブを使用するかどうか。

既存の VERT.X インスタンスへの接続

JVM にすでに存在する Vert.x インスタンスに接続する場合は、コンポーネントレベルでインスタンスを設定できます。

```
Vertx vertx = ...;
VertxComponent vertxComponent = new VertxComponent();
vertxComponent.setVertx(vertx);
camelContext.addComponent("vertx", vertxComponent);
```

第182章 VM

仮想マシンコンポーネント

vm: コンポーネントは非同期 SEDA 動作を提供し、BlockingQueue でメッセージを交換し、別のスレッドプールでコンシューマーを呼び出します。

このコンポーネントは、VM が CamelContext インスタンス間の通信をサポートする点で SEDA コンポーネントとは異なります。そのため、このメカニズムを使用して Web アプリケーション全体で通信できます(camel-core.jar が システム/ブート クラスパスにあることを想定)。

仮想マシンは SEDA コンポーネントの拡張です。

URI 形式

```
vm:queueName[?options]
```

queueName には、JVM 内のエンドポイントを一意に識別する任意の文字列（または少なくとも camel-core.jar をロードしたクラ出力ダー内）を指定できます。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

CAMEL 2.3 より前のバージョン - プロデューサーとコンシューマーの両方に同じ URI を使用する必要があります。

完全に同一の 仮想マシン エンドポイント URI は、プロデューサーとコンシューマーエンドポイントの両方に使用 する必要があり ます。そうしないと、URI の queueName 部分が同じであるにもかかわらず、Camel は 2 つ目の 仮想マシン エンドポイントを作成します。以下に例を示します。

```
from("direct:foo").to("vm:bar?concurrentConsumers=5");

from("vm:bar?concurrentConsumers=5").to("file://output");
```

プロデューサーとコンシューマーの両方にオプションを含む、完全な URI を使用する必要があります。

Camel 2.4 では、キュー名のみが一致するように修正されました。キュー名 バー を使用して、以下のように以前の exmple を書き換えることができます。

```
from("direct:foo").to("vm:bar");

from("vm:bar?concurrentConsumers=5").to("file://output");
```

オプション

同じルールが 仮想マシン コンポーネントに適用されるオプションおよびその他の重要な使用方法は、SEDA コンポーネントを参照してください。

サンプル

以下のルートでは、CamelContext インスタンス全体のエクスチェンジを order.email という名前の仮想マシンキューに送信します。

```
from("direct:in").bean(MyOrderBean.class).to("vm:order.email");
```

その後、他の Camel コンテキストでエクスチェンジを受信します（別の .war アプリケーションにデプロイされるなど）。

```
from("vm:order.email").bean(MyOrderEmailSender.class);
```

- *SEDA*

第183章 VIDEO

VIDEO コンポーネント

Camel 2.12 以降で利用可能

video : コンポーネントは、Open Weather Map （フリーのグローバルな情報と予測情報を提供するサイト）からの照会に使用されます。この情報は json String オブジェクトとして返されます。

Camel はデフォルトで 1 時間ごとに現在の更新をポーリングし、予測します。また、プロデューサーとして使用されるエンドポイントで定義されるパラメーターに基づいて、この API にクエリーすることもできます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-weather</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

重要

2015 年 10 月 9 日以降、OpenWeather サービスにアクセスするには API キーが必要です。このキーは、appid パラメーターを使用して、パラメーターとして video エンドポイントの URI 定義に渡されます。

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
weather://<unused name>[?options]
```

オプション

プロパティー	デフォルト	説明
appid	**null**	**Camel 2.16.1/2.15.5:** API サーバーに接続されたユーザーを認証するために使用される APPID ID。このキーは必須です。
location	**null**	null Camel が IP アドレスの地理位置位置を使用して現在の場所を判断しようとする場合は、都市（都市）を指定します。よく知られている都市名については、Open Weather Map が最適であると判断しますが、複数の結果が返される場合があります。したがって、と country も指定すると、より正確なデータが返されます。"current" を場所として指定すると、コンポーネントは現在の流れと長いものを取得しようとし、それを使用して詳細を取得します。location の代わりに lat オプションおよび lon オプションを使用できます。
lat	**null**	お気に入りの場所。location の代わりに lat オプションおよび lon オプションを使用できます。
lon	**null**	Longitude of location.location の代わりに lat オプションおよび lon オプションを使用できます。
period	**null**	null の場合、現在のものが返されます。それ以外の場合は、5、7、14 日の値を使用します。予測期間の数値のみが実際に解析されるため、期間のスペル化、大文字（無視）が行われます。
headerName	**null**	メッセージボディーではなくこのヘッダーに結果を保存するには、以下を行います。現在のメッセージボディーをそのままそのまま使用する場合に使用できます。

mode	**JSON**	splunk データの出力形式。使用できる値は **HTML**、**JSON**、または **XML**です。
units	**METRIC**	温度測定の単位。使用できる値は **IMPERIAL** または **METRIC**です。
consumer.delay	**3600000**	各ポーリングの遅延（デフォルトは 1 時間）
consumer.initialDelay	**1000**	ポーリングを開始する前にミリス。
consumer.userFixedDelay	**false**	**true** の場合、ポーリング間の固定遅延を使用します。そうでない場合は、固定レートが使用されます。詳細は、JDK の ScheduledExecutorService を参照してください。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

エクスチェンジデータ形式

Camel はボディーを json 形式の java.lang.String として配信します（上記の mode オプションを参照）。

メッセージヘッダー

ヘッダー	説明
CamelWeatherQuery	Open Weather Map サイトに送信された元のクエリー URL
CamelWeatherLocation	エンドポイントの場所を上書きし、代わりにこのヘッダーからの場所を使用するようにプロデューサーによって使用されます。

サンプル

この例では、スペインの Madrid の 7 日間のキャストがあります。

```
from("weather:foo?location=Madrid,Spain&period=7
days&appid=APIKEY").to("jms:queue:weather");
```

現在の場所の現在の場所を見つけるには、以下を使用できます。

```
from("weather:foo&appid=APIKEY").to("jms:queue:weather");
```

また、プロデューサーを使用して以下を見つけるには、以下を実行します。

```
from("direct:start")
  .to("weather:foo?location=Madrid,Spain&appid=APIKEY");
```

また、以下のように、ヘッダーのあるメッセージで送信して、任意の場所を取得できます。

```
String json = template.requestBodyAndHeader("direct:start", "", "CamelWeatherLocation",
"Paris,France&appid=APIKEY", String.class);
```

また、現在の場所で以下を取得するには、以下を実行します。

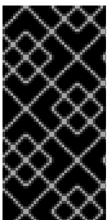
```
String json = template.requestBodyAndHeader("direct:start", "", "CamelWeatherLocation",
"current&appid=APIKEY", String.class);
```

第184章 WEBSOCKET

WEBSOCKET コンポーネント

Camel 2.10 以降で利用可能

Websocket コンポーネントは、Websocket を使用してクライアントと通信するための WebSocket エンドポイント を提供します。コンポーネントは Eclipse Jetty Server を使用します。これは、IETF 仕様（ドラフトおよび RFC 6455）を実装します。ws:// および wss:// のプロトコルをサポートします。wss:// プロトコルを使用するには、SSLContextParameters オブジェクトを定義する必要があります。



現在サポートされているバージョン

Camel 2.10 は Jetty 7.5.4.v20111024 を使用するため、D00 から D13 IETF 実装のみを利用できます。Camel 2.11 は Jetty 7.6.7 を使用します。

URI 形式

```
websocket://hostname[:port][/resourceUri][?options]
```

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

コンポーネントオプション

ホストを WebSocket サーバーとして機能するように設定するには、使用する前に websocket コンポーネントを設定できます。

オプション	デフォルト	説明
host	0.0.0.0	ホスト名。
port	9292	ポート番号。

staticResources	**null**	index.html ファイルなどの静的リソースのパス。このオプションが設定された場合、サーバーは指定されたホスト名とポートで起動し、index.html ファイルなどの静的リソースを処理します。このオプションが設定されていない場合、サーバーは起動されません。
sslContextParameters		レジストリー の **org.apache.camel.util.jsse.SSLContextParameters** オブジェクトへの 参照。 この参照は、コンポーネントレベルで設定された **SSLContextParameters** を上書きします。Security Guide および Using the JSSE Configuration Utility の Configuring Transport Security for Camel Components の章 を参照してください。
enableJmx	**false**	このオプションが true の場合、このエンドポイントに対して Jetty JMX サポートが有効になります。
sslKeyPassword	**null**	**コンシューマーのみ**: SSL 使用時のキーストアのパスワード。
sslPassword	**null**	**コンシューマーのみ**: SSL を使用する場合のパスワード。
sslKeystore	**null**	**コンシューマーのみ**: キーストアへのパス。
minThreads	**null**	**コンシューマーのみ**: サーバースレッドプールでスレッドの最小数の値を設定するには、以下を実行します。
maxThreads	**null**	**コンシューマーのみ**: サーバースレッドプールの最大スレッド数に値を設定します。
threadPool	**null**	**コンシューマーのみ**: サーバーにカスタムスレッドプールを使用します。

エンドポイントオプション

WebsocketEndpoint は、使用前に設定できます。

オプション	デフォルト	説明
sslContextParametersRef		**非推奨および は** Camel 3.0: Registry **の org.apache.camel.util.jsse.SSLContextParameters への参照で削除され** ます。 この参照は、コンポーネントレベルで設定済みの SSLContextParameters を上書きします。Using the JSSE Configuration Utility を参照してください。代わりに **sslContextParameters** オプションを使用してください。
sslContextParameters		Camel 2.11.1: レジストリー の **org.apache.camel.util.jsse.SSLContextParameters** への 参照。この参照は、コンポーネントレベルで設定済みの SSLContextParameters を上書きします。Security Guide および Using the JSSE Configuration Utility の Configuring Transport Security for Camel Components の章 を参照してください。
sendToAll	**null**	**プロデューサーのみ：** すべての WebSocket サブスクライバーに送信する。メッセージで **WebsocketConstants.SEND_TO_ALL** ヘッダーを使用する代わりに、エンドポイントレベルでを設定できます。
staticResources	**null**	Web リソースまたはクラスパスのルートディレクトリー。コンポーネントがファイルシステムまたはクラスパスからリソースを読み込むかどうかに応じて、プロトコル file: または classpath: を使用します。

sslContextParameters		レジストリー の **org.apache.camel.util.jsse.SSLContextParameters** オブジェクトへの 参照。この参照は、コンポーネントレベルで設定済みの SSLContextParameters を上書きします。Security Guide および Using the JSSE Configuration Utility の Configuring Transport Security for Camel Components の章 を参照してください。
enableJmx	**false**	このオプションが true の場合、このエンドポイントに対して Jetty JMX サポートが有効になります。詳細は Jetty JMX support を参照してください。
sslKeyPassword	**null**	**コンシューマーのみ**: SSL 使用時のキーストアのパスワード。
sslPassword	**null**	**コンシューマーのみ**: SSL を使用する場合のパスワード。
sslKeystore	**null**	**コンシューマーのみ**: キーストアへのパス。
minThreads	**null**	**コンシューマーのみ**: サーバースレッドプールでスレッドの最小数の値を設定するには、以下を実行します。
maxThreads	**null**	**コンシューマーのみ**: サーバースレッドプールの最大スレッド数に値を設定します。
threadPool	**null**	**コンシューマーのみ**: サーバーにカスタムスレッドプールを使用します。

メッセージヘッダー

***WebSocket** コンポーネントは 2 つのヘッダーを使用して、メッセージを単一/現在のクライアント、またはすべてのクライアントに送信することを示します。*

キー	説明

WebsocketConstants.SEND_TO_ALL	現在接続しているすべてのクライアントにメッセージを送信します。このヘッダーを使用する代わりに、エンドポイントで **sendToAll** オプションを使用できます。
WebsocketConstants.CONNECTION_KEY	指定の接続キーを使用してクライアントにメッセージを送信します。

使用方法

この例では、Camel はクライアントが通信できる WebSocket サーバーを公開します。Websocket サーバーは、0.0.0.0:9292 というデフォルトのホストおよびポートを使用します。この例では、入力の echo を送り返します。メッセージを送信するには、変換されたメッセージを同じエンドポイント "websocket://echo" に送信する必要があります。メッセージングのデフォルトは InOnly であるため、この設定が必要です。

```
// expose a echo websocket client, that sends back an echo
from("websocket://echo")
    .log(">>> Message received from WebSocket Client : ${body}")
    .transform().simple("${body}${body}")
    // send back to the client, by sending the message to the same endpoint
    // this is needed as by default messages is InOnly
    // and we will by default send back to the current client using the provided connection key
    .to("websocket://echo");
```

この例はユニットテストの一部で、ここでは を参照してください。クライアントとして、Web ソケットのサポートを提供する AHC ライブラリーを使用します。

さらに、webapp リソースの場所が定義され、Jetty Application Server が WebSocket サーブレットを登録するだけでなく、ブラウザーの Web リソースも公開できるように定義されています。リソースは webapp ディレクトリーの下に定義する必要があります。

```
from("activemq:topic:newsTopic")
   .routeId("fromJMStoWebSocket")
   .to("websocket://localhost:8443/newsTopic?
sendToAll=true&staticResources=classpath:webapp");
```

WEBOCKET コンポーネントの SSL の設定

JSSE 設定ユーティリティーの使用

Camel 2.10 の時点で、Websocket コンポーネントは Using the JSSE Configuration Utility を通じて SSL/TLS 設定をサポートします。このユーティリティーは、作成する必要のあるコンポーネント固有のコードの量を大幅に減らし、エンドポイントおよびコンポーネントレベルで設定可能です。 以下の例は、エンドポイントの設定方法を示しています。> Security Guide の Configuring Transport Security for Camel Components の章を参照してください。

エンドポイントの SPRING DSL ベースの設定

```
...
  <camel:sslContextParameters
      id="sslContextParameters">
    <camel:keyManagers
        keyPassword="keyPassword">
      <camel:keyStore
          resource="/users/home/server/keystore.jks"
          password="keystorePassword"/>
    </camel:keyManagers>
    <camel:trustManagers>
      <camel:keyStore
          resource="/users/home/server/keystore.jks"
          password="keystorePassword"/>
    </camel:trustManagers>
  </camel:sslContextParameters>...
...
  <to uri="websocket://127.0.0.1:8443/test?sslContextParameters=#sslContextParameters"/>...
```

エンドポイントの JAVA DSL ベースの設定

```
...
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() {

                String uri = "websocket://127.0.0.1:8443/test?
sslContextParameters=#sslContextParameters";

                from(uri)
                     .log(">>> Message received from WebSocket Client : ${body}")
                     .to("mock:client")
                     .loop(10)
                         .setBody().constant(">> Welcome on board!")
                         .to(uri);
...
```

- *AHC*

- *Jetty*

- *Twitter Websocket Example では、twitter 検索の定数フィードをポーリングし、Web ソケットを使用して結果を Web ページにリアルタイムで公開する方法を実証します。*

第185章 XMLRPC

XMLRPC COMPONENT

Camel 2.11 から利用可能

このコンポーネントは、xml の dataformat を提供します。これにより、Apache XmlRpc のバインドデータ形式を使用した要求メッセージおよび応答メッセージのシリアライズおよびデシリアライズが可能になります。camel-xmlrpc プロデューサーを介して XMLRPC サービスを呼び出すこともできます。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-xmlrpc</artifactId>
   <version>x.x.x</version>
   <!-- use the same version as your Camel core version -->
</dependency>
```

XMLRPC OVERVIEW

これは、異なるオペレーティングシステム上でソフトウェアを実行できる 仕様 および実装セットで、インターネットを介してプロシージャーコールを行う異なる環境で実行されます。

HTTP をトランスポートとして使用し、XML をエンコーディングとして使用してリモートプロシージャーを呼び出します。XML-RPC は可能な限りシンプルになるように設計されていますが、複雑なデータ構造を送信、処理、および返すことができます。

一般的な XML-RPC 要求の例を以下に示します。

```
<?xml version="1.0"?>
<methodCall>
```

```
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
        <value><i4>40</i4></value>
    </param>
  </params>
</methodCall>
```

一般的な XML-RPC 応答の例を以下に示します。

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
        <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

一般的な XML-RPC 障害は次のとおりです。

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

URI 形式

```
xmlrpc://serverUri[?options]
```

オプション

プロパティー	デフォルト	説明
basicEncoding	**null**	Basic 認証のエンコーディングを設定します。null は UTF-8 が選択されることを意味します。
basicUserName	**null**	Basic 認証のユーザー名。
basicPassword	**null**	Basic 認証のパスワード。
clientConfigurer	**null**	XmlRpcClientConfigurer のインターフェイスを実装して XmlRpcClientConfigurer をユーザー希望として設定する XmlRpcClient アドバイザーの参照 ID。値は #myConfigurer のように # で始まる必要があります。
connectionTimeout	**0**	接続タイムアウトをミリ秒単位で設定します。0 は無効にすることです。
contentLengthOptional	**false**	Content-Length ヘッダーを省略するかどうか。XML-RPC 仕様では、このようなヘッダーが存在することが要求されます。
enabledForExceptions	**false**	エラーの発生時に応答に faultCause 要素が含まれるべきかどうか。faultCause は例外で、サーバーがシリアライズ可能なオブジェクトとしてバイトストリームにトラップおよび書き込んだものです。
enabledForExtensions	**false**	エクステンションが有効であるかどうか。デフォルトでは、クライアントまたはサーバーは XML-RPC 仕様に厳密に準拠し、拡張機能は無効になっています。
encoding	**null**	リクエストエンコーディングを設定します。null は UTF-8 が選択されることを意味します。

gzipCompressing	**false**	要求の送信に gzip 圧縮が使用されているかどうか。
gzipRequesting	**false**	要求の送信に gzip 圧縮が使用されているかどうか。
replyTimeout	**0**	応答タイムアウトをミリ秒単位で設定します。0 は無効にします。
userAgent	**null**	xmlrpc 要求の実行時に設定する http ユーザーエージェントヘッダー

メッセージヘッダー

Camel XmlRpc はこれらのヘッダーを使用します。

ヘッダー	説明
CamelXmlRpcMethodName	XmlRpc サーバーの呼び出しに使用する XmlRpc メソッド名。

XMLRPC データフォーマットの使用

XmlRpc メッセージはリクエストまたは応答になる可能性があるため、XmlRpcDataFormat を使用する場合は request に dataformat is を指定する必要があります。

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">

  <!-- we define the xml rpc data formats to be used -->
  <dataFormats>
    <xmlrpc id="xmlrpcRequest" request="true"/>
    <xmlrpc id="xmlrpcResponse" request="false"/>
  </dataFormats>
```

```
    <route>
        <from uri="direct:request"/>
        <marshal ref="xmlrpcRequest"/>
        <unmarshal>
          <xmlrpc request="true"/>
        </unmarshal>
        <to uri="mock:request" />
    </route>

    <route>
        <from uri="direct:response"/>
        <marshal>
           <xmlrpc request="false"/>
        </marshal>
        <unmarshal ref="xmlrpcResponse"/>
        <to uri="mock:response" />
    </route>
</camelContext>
```

クライアントからの XMLRPC サービスの呼び出し

XmlRpc サービスを呼び出すには、メッセージヘッダーに methodName を指定し、パラメーターを以下のコードのようにメッセージボディーに配置する必要があります。障害メッセージが返された場合は、XmlRpcException が発生する例外が発生するはずです。

```
String response = template.requestBodyAndHeader(xmlRpcServiceAddress, new Object[]
{"me"}, XmlRpcConstants.METHOD_NAME, "hello", String.class);
```

JAVA コードで XMLRPCCLIENT を設定する方法

camel-xmlrpc は、コンポーネントによって使用される XmlRpcClientClient を設定するためのプラグ可能なストラテジーを提供し、ユーザーは XmlRpcClientConfigurer インターフェイスを実装するだけで、XmlRpcClient を必要に応じて設定できます。clientConfigure インスタンスの参照は、uri オプション clientConfigure を使用して設定できます。

```
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

public class MyClientConfigurer implements XmlRpcClientConfigurer {

    @Override
```

```
    public void configureXmlRpcClient(XmlRpcClient client) {
        // get the configure first
        XmlRpcClientConfigImpl clientConfig = (XmlRpcClientConfigImpl)client.getClientConfig();
        // change the value of clientConfig
        clientConfig.setEnabledForExtensions(true);
        // set the option on the XmlRpcClient
        client.setMaxThreads(10);
    }

}
```

第186章 XML セキュリティーコンポーネント

XML セキュリティーコンポーネント

Camel 2.12.0 から利用可能

この Apache Camel コンポーネントを使用すると、W3C 標準 XML 署名 構文および処理で説明されているように、または後継 バージョン 1.1 で説明されているように、XML 署名を生成および 検証できます。XML 暗号化サポートについては、XML Security Data Format を参照してください。

XML 署名の概要は、を参照して ください。コンポーネントの実装は JSR 105 (W3C 標準に対応する Java API)をベースとしており、JSR 105 の Apache Santuario および JDK プロバイダーをサポートします。この実装は、最初に Apache Santuario プロバイダーの使用を試みます。Santuario プロバイダーが見つからない場合は、JDK プロバイダーを使用します。さらに、実装は DOM ベースです。

Camel 2.15.0 以降、署名側のエンドポイントに対して XAdES-BES/EPES もサポートします。「Signer エンドポイントの XAdES-BES/EPES」 を参照してください。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xmlsecurity</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

XML 署名ラッピングモード

XML 署名は、エンベロープ、エンベロープ、および切り離された XML 署名によって異なります。エンベロープされた XML 署名の場合、XML 署名は署名済み XML ドキュメントによってラップされます。つまり、XML 署名要素は親要素の子要素で、署名された XML ドキュメントに属します。エンベロープ XML 署名の場合、XML 署名に署名されたコンテンツが含まれます。その他のケースはすべて、デタッチされた XML 署名と呼ばれます。Camel 2.14.0 以降、特定の形式の XML 署名がサポートされます。

エンベロープされた XML 署名の場合、サポートされる生成された XML 署名の構造は次のとおりです (変数は [] で囲まれます) 。

```
<[parent element]>
   ... <!-- Signature element is added as last child of the parent element-->
   <Signature Id="generated_unique_signature_id">
     <SignedInfo>
         <Reference URI="">
             <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature"/>
             (<Transform>)* <!-- By default "http://www.w3.org/2006/12/xml-c14n11" is added
to the transforms -->
             <DigestMethod>
             <DigestValue>
         </Reference>
         (<Reference URI="#[keyinfo_Id]">
             <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
             <DigestMethod>
             <DigestValue>
         </Reference>)?
         <!-- further references possible, see option 'properties' below -->
     </SignedInfo>
     <SignatureValue>
     (<KeyInfo Id="[keyinfo_id]">)?
     <!-- Object elements possible, see option 'properties' below -->
   </Signature>
</[parent element]>
```

エンベロープ XML 署名の場合、サポートされている生成された XML 署名の構造は次のとおりです。

```
<Signature Id="generated_unique_signature_id">
  <SignedInfo>
      <Reference URI="#generated_unique_object_id" type="[optional_type_value]">
          (<Transform>)* <!-- By default "http://www.w3.org/2006/12/xml-c14n11" is added to
the transforms -->
          <DigestMethod>
          <DigestValue>
      </Reference>
      (<Reference URI="#[keyinfo_id]">
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
          <DigestMethod>
          <DigestValue>
      </Reference>)?
       <!-- further references possible, see option 'properties' below  -->
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo Id="[keyinfo_id]">)?
  <Object Id="generated_unique_object_id"/> <!-- The Object element contains the in-
message body -->
  <!-- The object ID can either be generated or set by the option parameter
"contentObjectId" -->
  <!-- Further Object elements possible, see option 'properties' below -->
</Signature>
```

Camel 2.14.0 では、以下の構造を持つ分離された XML 署名がサポートされます(「署名要素のシブ

リングとしての分離された XML 署名」も参照してください)。

```
(<[signed element] Id="[id_value]">
<!-- signed element must have an attribute of type ID -->
    ...

</[signed element]>
<other sibling/>*
<!-- between the signed element and the corresponding signature element, there can be other siblings.
 Signature element is added as last sibling. -->
<Signature Id="generated_unique_ID">
        <SignedInfo>
            <CanonicalizationMethod>
            <SignatureMethod>
            <Reference URI="#[id_value]" type="[optional_type_value]">
            <!-- reference URI contains the ID attribute value of the signed element -->
                (<Transform>)* <!-- By default "http://www.w3.org/2006/12/xml-c14n11" is added to the transforms -->
                <DigestMethod>
                <DigestValue>
            </Reference>
            (<Reference URI="#[generated_keyinfo_Id]">
                <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
                <DigestMethod>
                <DigestValue>
            </Reference>)?
        </SignedInfo>
        <SignatureValue>
        (<KeyInfo Id="[generated_keyinfo_id]">)?
</Signature>)+
```

URI 形式

camel コンポーネントは、以下の URI 形式の 2 つのエンドポイントで設定されます。

```
xmlsecurity:sign:name[?options]
xmlsecurity:verify:name[?options]
```

- 署名側のエンドポイントでは、インメッセージの本文の XML 署名を生成できます。これは、XML ドキュメントまたはプレーンテキストのいずれかになります。エンベロープされたエンベロープ(Camel 12.14より)の XML 署名は、out-message のボディーに設定されます。

- ベリファイアのエンドポイントでは、エンベロープまたはエンベロープ XML 署名、またはインメッセージの本文に含まれるエンベロープまたはエンベロープ(Camel 2.14.0の) XML 署名を複数検証できます。検証に成功すると、元のコンテンツが XML 署名から抽出され、アウトメッセージの本文に設定されます。

- URI の name 部分をユーザーが選択して、Camel コンテキスト内の異なる署名/検証エンドポイントを区別できます。

基本的な例

以下の例は、コンポーネントの基本的な使用方法を示しています。

```
from("direct:enveloping").to("xmlsecurity:sign://enveloping?keyAccessor=#accessor",
                    "xmlsecurity:verify://enveloping?keySelector=#selector","mock:result")
```

Spring XML の場合：

```
<from uri="direct:enveloping" />
  <to uri="xmlsecurity:sign://enveloping?keyAccessor=#accessor" />
  <to uri="xmlsecurity:verify://enveloping?keySelector=#selector" />
<to uri="mock:result" />
```

署名プロセスには、秘密鍵が必要です。この秘密鍵を提供するキーアクセサー Bean を指定します。検証には、対応する公開鍵が必要です。この公開鍵を提供するキーセレクター Bean を指定します。

キーアクセサー Bean は KeyAccessor インターフェイスを実装する必要があります。パッケージ org.apache.camel.component.xmlsecurity.api には、Java キーストアから秘密鍵を読み取るデフォルトの実装クラス DefaultKeyAccessor が含まれます。

キーセレクター Bean は javax.xml.crypto.KeySelector インターフェイスを実装する必要があります。パッケージ org.apache.camel.component.xmlsecurity.api には、キーストアから公開鍵を読み取るデフォルトの実装クラス DefaultKeySelector が含まれます。

この例では、デフォルトの署名アルゴリズム http://www.w3.org/2000/09/xmldsig#rsa-sha1 が使用されます。選択した署名アルゴリズムは、オプション signatureAlgorithm（以下を参照）で設定できます。署名側のエンドポイントは、エンベロープ XML 署名を作成します。エンベロープされた XML 署名を作成する場合は、Signature 要素の親要素を指定する必要があります。詳細は、options parentLocalName を参照してください。

デタッチされた XML 署名の作成については、「署名要素のシブリングとしての分離された XML 署名」を参照してください。

一般的な署名およびオプションの検証

エンドポイント(signer および verifier)の両方に使用できるオプションもあります。

名前	タイプ	デフォルト	説
uriDereferencer	javax.xml.crypto.URIDereferencer	**null**	U し
baseUri	**String**	**null**	U
cryptoContextProperties	**Map<String, ? extends Object>**	**null**	暗 フ 検 は す フ **o** は
disallowDoctypeDecl	**Boolean**	**Boolean.TRUE**	受
omitXmlDeclaration	**Boolean**	**Boolean.FALSE**	X
clearHeaders	**Boolean**	**Boolean.TRUE**	X へ
schemaResourceUri	**String**	**null**	C マ す フ 合
outputXmlEncoding	**String**	**null**	C

署名オプション

署名側のエンドポイントには以下のオプションがあります。

名前	タイプ	デフォルト	説明

keyAccessor	KeyAccessor	**null**	署名キーと KeyInfo インスタンスを提供します。キーストアを使用する実装の例は、DefaultKeyAccessorを参照してください。
addKeyInfoReference	**Boolean**	**Boolean.TRUE**	キーアクセサーが提供する **KeyInfo** 要素を参照する **Reference** 要素を XML 署名に追加するかどうかを示します。
signatureAlgorithm	**String**	**http://www.w3.org/2000/09/xmldsig#rsa-sha1**	ダイジェストおよび暗号化アルゴリズムで設定される署名アルゴリズム。ダイジェストアルゴリズムは SignedInfo 要素のダイジェストを計算するために使用され、暗号化アルゴリズムはこのダイジェストに署名するために使用されます。設定可能な値：**http://www.w3.org/2000/09/xmldsig#dsa-sha1**, **http://www.w3.org/2000/09/xmldsig#rsa-sha1**, **http://www.w3.org/2001/04/xmldsig-more#rsa-sha256**, **http://www.w3.org/2001/04/xmldsig-more#rsa-sha384**, **http://www.w3.org/2001/04/xmldsig-more#rsa-sha512**

digestAlgorithm	**String**	説明 を参照してください。	メッセージ内ボディーのダイジェストを計算するダイジェストアルゴリズム。指定のない場合は、署名アルゴリズムのダイジェストアルゴリズムが使用されます。使用できる値は **http://www.w3.org/2000/09/xmldsig#sha1**、**http://www.w3.org/2001/04/xmlenc#sha256**、**http://www.w3.org/2001/04/xmldsig-more#sha384**、および **http://www.w3.org/2001/04/xmlenc#sha512** です。
parentLocalName	**String**	null	**Signature** 要素の親のローカル名。**Signature** 要素は、親の子の最後に追加されます。エンベロープされた XML 署名に必要です。このオプションと **parentXpath** オプションが **null** の場合は、エンベロープ XML 署名が作成されます。**parentNamespace** も参照してください。または、**parentXpath** オプションを使用して親を指定することもできます。
parentNamespace	**String**	null	Signature 要素の親の名前空間。オプション **parentLocalName**を参照してください。

parentXpath	**XPathFilterParameterSpec**	null	Camel 2.15.0 以降XPath を **Signature** 要素の親に追加します。**Signature** 要素は、親の子の最後に追加されます。エンベロープされた XML 署名に必要です。このオプションと **parentLocalName** オプションが **null** の場合は、エンベロープ XML 署名が作成されます。または、**parentLocalName** オプションを使用して親を指定することもできます。例：**/p1:root/SecurityItem[last()]** この例では、**SecurityItem** という名前の最後のシブリングを選択します。このような選択は、**parentLocalName** オプションではできません。
canonicalizationMethod	javax.xml.crypto.AlgorithmMethod	C14n	ダイジェストの計算前に SignedInfo 要素の正規化に使用される正規化メソッド。ヘルパーメソッド XmlSignatureHelper.getCanonicalizationMethod(String algorithm)または getCanonicalizationMethod (String algorithm, List<String> inclusiveNamespacePrefixes)を使用して正規化メソッドを作成できます。

transformMethods	**List<javax.xml.crypto.AlgorithmMethod>**	説明 を参照してください。	ダイジェストが計算される前にメッセージボディーで実行される変換。デフォルトでは、C14n が追加され、エンベロープ署名の場合は(**parentLocalName**を参照)、http://www.w3.org/2000/09/xmldsig#enveloped-signature もリストの位置 0 に追加されます。XmlSignatureHelper のメソッドを使用して変換メソッドを作成します。 Camel ヘッダー **CamelXmlSignatureTransformMethods** はこのオプションを上書きします(**Camel 2.17.0 以降**)。ヘッダーの値は **String** タイプである必要があります。変換アルゴリズムをコンマ区切りリストで指定します（例：**http://www.w3.org/2000/09/xmldsig#enveloped-signature,http://www.w3.org/TR/2001/REC-xml-c14n-20010315**）。ヘッダーでは、**http://www.w3.org/TR/1999/REC-xslt-19991116**、**http://www.w3.org/2002/06/xmldsig-filter2**、**http://www.w3.org/TR/1999/REC-xpath-19991116** などのパラメーターが必要な変換アルゴリズムを指定できません。
prefixForXmlSignatureNamespace	**String**	**ds**	XML 署名名前空間の接頭辞。**null** が指定されているか、空の文字列の場合は、署名名前空間に接頭辞は使用されません。

contentReferenceUri	**String**	説明 を参照してください。	署名されたコンテンツ（メッセージ本文）への参照の URI。**null** でエンベロープされた XML 署名の場合、URI は空の文字列に設定されます。**null** でエンベロープ XML 署名の場合、URI は **generated_object_id** に設定されます。これは、参照が in-message ボディーが含まれる **Object** 要素を参照することを意味します。メッセージ内ボディーに署名する必要がない場合は、このオプションを使用して in-message ボディーの特定部分を参照できます。この値はヘッダー **CamelXmlSignature ContentReferenceUri** で上書きできます。XML ID 属性の値（例：**#ID_value**）を使用する場合は、入力 XML ドキュメントに含まれる **doctype** 定義を使用するか、オプション **schemaResourceUri** で指定できる XML スキーマドキュメントを介して ID 属性に関する情報を提供する必要があります。**schemaResourceURi** オプションを使用した XML スキーマによる ID 属性の定義は、エンベロープ署名ケースでのみ機能します。このオプションが **xpathsToIdAttributes** に設定されている場合、このオプションはデタッチされた署名の場合は無視されます。
contentReferenceType	**String**	null	コンテンツ参照の type 属性の値。この値はヘッダー CamelXmlSignatureContentReferenceType で上書きできます。

plainText	**Boolean**	Boolean.FALSE	メッセージ内の本文にプレーンテキストが含まれているかどうかを示します。通常、署名ジェネレーターは受信メッセージのボディーを XML として扱います。メッセージ本文がプレーンテキストの場合は、このオプションを **true** に設定する必要があります。値はヘッダー **CamelXmlSignatureMessageIsPlainText** で上書きできます。
plainTextEncoding	**String**	null	**plainText** オプションが **true** に設定されている場合にのみ使用されます。次に、プレーンテキストのエンコーディングを指定できます。**null** の場合、UTF-8 が使用されます。値はヘッダー CamelXmlSignatureMessageIsPlainTextEncoding で上書きできます。
properties	XmlSignatureProperties	null	追加のプロパティーが含まれる XML 署名に参照およびオブジェクトを追加するには、XmlSignatureProperties インターフェイスを実装する Bean を指定できます。
contentObjectId	**String**	null	**Object** 要素の **Id** 属性の値。エンベロープ XML 署名ケースでのみ使用されます。**null** の場合、一意の値が生成されます。2.12.2 から利用可能

xpathsToIdAttributes	**List<XPathFilterParameterSpec>**	空のリスト	**2.14.0 以降**署名する要素の ID 属性に対する XPATH 式のリスト。デタッチされた XML 署名に使用されます。は、オプションと組み合わせてのみ使用できます。**schemaResourceUri**値はヘッダー**CamelXmlSignatureXpathsToIdAttributes**で上書きできます。オプション **parentLocalNam**が同時に 設定されている場合、例外が発生します。クラス **XPathFilterParameterSpec** にはパッケージ **javax.xml.crypto.dsig.spec** があります。詳細は、サブ章のデタッチされた XML 署名（署名要素 のシブリングとしての XML 署名）を参照してください。
signatureId	**String**	null	**2.14.0 以降**Signature 要素の Id 属性の値。**null** の場合は、一意の ID が生成されます。値が空の文字列("")の場合、Signature 要素に Id 属性は追加されません。

オプションの検証

verifier エンドポイントには以下のオプションがあります。

名前	タイプ	デフォルト	説明
keySelector	javax.xml.crypto.KeySelector	**null**	XML 署名を検証するためのキーを提供します。キーストアを使用する実装の例は、DefaultKeySelector を参照してください。

xmlSignatureChecker	XmlSignatureChecker	**null**	このインターフェイスを使用すると、検証の実行前にアプリケーションが XML 署名を確認できます。この手順は、http://www.w3.org/TR/xmldsig-bestpractices/#check-what-is-signedで推奨されます。
validationFailedHandler	ValidationFailedHandler	DefaultValidationFailedHandler	さまざまな検証に失敗した状況を処理します。デフォルトの実装は、さまざまな状況で特定の例外を出力します（すべての例外には **org.apache.camel.component.xmlsecurity.api** のパッケージ名があり、**XmlSignatureInvalidException** のサブクラスです）。署名値の検証に失敗すると、**XmlSignatureInvalidValueException** が出力されます。参照検証に失敗すると、**XmlSignatureInvalidContentHashException** が出力されます。詳細は、JavaDoc を参照してください。
xmlSignature2Message	XmlSignature2Message	DefaultXmlSignature2Message	検証後に XML 署名を ouput-message にマップする Bean。このマッピングは、**outputNodeSearchType** オプション、**outputNodeSearch** オプション、および **removeSignatureElements** オプションを使用して設定できます。デフォルトの実装では、3 つの出力ノード検索タイプ Default、ElementName、および XPath に関連する 3 つの可能性が提供されます。デフォルトの実装はシリアライズされ、ouput メッセージのボディーに

			設定されるノードを決定します。検索タイプが "ElementName" の場合、ouput ノード（この場合は要素でなければならない）は、検索値で定義されたローカル名と名前空間によって決定されます（オプション **outputNodeSearch**を参照）。検索タイプが XPath の場合、出力ノードは検索値で指定された XPath で決定されます（この場合、ouput ノードは "Element"、"TextNode"、または "Document"）のタイプになります）。出力ノード検索タイプが Default の場合、以下のルールが適用されます。エンベロープされた XML 署名の場合(URI="" の参照で、"http://www.w3.org/2000/09/xmldsig#enveloped-signature")は、Signature 要素のない受信 XML ドキュメントが出力メッセージボディーに設定されます。非承認の XML 署名の場合、メッセージボディーは参照されるオブジェクトから決定されます。詳細は、Enveloping XML Signature Case の Output Node Determination の章を参照してください。
outputNodeSearchType	**String**	**Default**	出力ノードの検索タイプを決定します。オプション **xmlSignature2Message** を参照してください。デフォルトの実装 **DefaultXmlSignature2Message** は、Default、ElementName、および XPath の 3 つの検索タイプをサポートします。

outputNodeSearch	**Object**	**null**	出力ノード検索の値を検索します。タイプは検索タイプによって異なります。デフォルトの検索実装 **DefaultXmlSignature2Message** の場合、以下の値を指定できます。検索タイプが Default の場合、検索値は使用されません。検索タイプが "ElementName" の場合、検索値には出力要素の名前空間とローカル名が含まれます。名前空間は括弧内に置く必要があります。検索タイプが XPath の場合、検索値には XPath を表す **javax.xml.crypto.dsig.spec.XPathFilterParameterSpec** のインスタンスが含まれます。このようなインスタンスは、**XmlSignatureHelper メソッド .getXpathFilter (String xpath, Map<String, String> namespaceMap)を使用し** て作成できます。XPath は、タイプ Element、TextNode、または Document で使用できる出力ノードを決定します。
removeSignatureElements	**Boolean**	**Boolean.FALSE**	エンベロープされた XML 署名ケースの出力メッセージの署名要素を削除するインジケーター。**XmlSignature2Message** インスタンスで使用されます。デフォルトの実装では、このインジケーターを使用して 2 つの検索タイプ ElementName と XPath を使用します。

secureValidation	Boolean	Boolean.TRUE	セキュアな検証を有効にします。true の場合、セキュアな検証が有効になります。詳細は こちら を参照してください。

ENVELOPING XML 署名ケースの出力ノードの決定

検証後、ノードは XML 署名ドキュメントから抽出され、最終的に output-message ボディーに返されます。XML 署名の場合、デフォルトの実装 Default XmlSignature2Message は、次の方法でノード検索タイプ Default に対してこれを行います（オプション xmlSignature2Messageを参照）。

まず、オブジェクト参照が決定されます。

- *同じドキュメント参照のみが考慮されます(URI は # で始まる必要があります)。*
- *また、マニフェストを使用したオブジェクトへの間接的なドキュメント参照も考慮されます。*
- *結果のオブジェクト参照数は 1 である必要があります。*

次に、オブジェクトは逆参照され、オブジェクトには XML 要素が 1 つだけ含まれる必要があります。この要素は、出力ノードとして返されます。

これは、エンベロープ XML 署名の構造が必要であることを意味します。

```
<Signature>
    <SignedInfo>
      <Reference URI="#object"/>
      <!-- further references possible but they must not point to an Object or Manifest containing an object reference -->
      ...
    </SignedInfo>

    <Object Id="object">
        <!-- contains one XML element which is extracted to the message body -->
    <Object>
    <!-- further object elements possible which are not referenced-->
```

```
        ...
        (<KeyInfo>)?
    </Signature>
```

または構造です。

```
    <Signature>
        <SignedInfo>
          <Reference URI="#manifest"/>
          <!-- further references  are possible but they must not point to an Object or other
manifest containing an object reference -->
          ...
        </SignedInfo>

        <Object >
          <Manifest Id="manifest">
             <Reference URI=#object/>
          </Manifest>
        </Objet>
        <Object Id="object">
           <!-- contains the DOM node which is extracted to the message body -->
        </Object>
         <!-- further object elements possible which are not referenced -->
        ...
        (<KeyInfo>)?
    </Signature>
```

署名要素のシブリングとしての分離された XML 署名

2.14.0 以降

署名が署名された要素のシブリングであるデタッチされた署名を作成できます。以下の例には、2 つの分離署名が含まれます。最初の署名は要素 C 用で、2 つ目の署名は要素 A 用です。署名は ネスト化され ます。2 つ目の署名は、最初の署名が含まれる要素 A の です。

例186.1 デタッチされた XML 署名の例

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
   <A ID="IDforA">
      <B>
         <C ID="IDforC">
            <D>dvalue</D>
         </C>
         <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
            Id="_6bf13099-0568-4d76-8649-faf5dcb313c0">
            <ds:SignedInfo>
               <ds:CanonicalizationMethod
```

```
                Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
            <ds:SignatureMethod
                Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
            <ds:Reference URI="#IDforC">
                ...
            </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>aUDFmiG71</ds:SignatureValue>
      </ds:Signature>
    </B>
  </A>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"Id="_6b02fb8a-30df-42c6-ba25-76eba02c8214">
    <ds:SignedInfo>
      <ds:CanonicalizationMethod
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      <ds:SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <ds:Reference URI="#IDforA">
        ...
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>q3tvRoGgc8cMUqUSzP6C21zb7tt04riPnDuk=</ds:SignatureValue>
  </ds:Signature>
<root>
```

*この例では、複数の要素に署名でき、その要素ごとに署名がシブリングとして作成されます。署名する要素には、タイプ ID の属性が必要です。属性の ID タイプは XML スキーマで定義する必要があります（オプション **schemaResourceUri**を参照）。タイプ ID の属性を参照する XPATH 式の一覧を指定します（オプション **xpathsToIdAttributes**を参照してください）。これらの属性は、署名する要素を決定します。要素は、keyAccessor Bean によって指定された同じキーで署名されます。より高い (=deeper)階層レベルの Ements が最初に署名されます。この例では、要素 "C" は要素 A の前に署名されています。*

例186.2 Java DSL の例

```
from("direct:detached")
  .to("xmlsecurity:sign://detached?keyAccessor=#keyAccessorBeant&xpathsToIdAttributes=#xpathsToIdAttributesBean&schemaResourceUri=Test.xsd")
  .to("xmlsecurity:verify://detached?keySelector=#keySelectorBean&schemaResourceUri=org/apache/camel/component/xmlsecurity/Test.xsd")
  .to("mock:result");
```

例186.3 Spring の例

```
<bean id="xpathsToIdAttributesBean" class="java.util.ArrayList">
    <constructor-arg type="java.util.Collection">
```

```
            <list>
                <bean
                    class="org.apache.camel.component.xmlsecurity.api.XmlSignatureHelper"
                    factory-method="getXpathFilter">
                    <constructor-arg type="java.lang.String"
                        value="/ns:root/a/@ID" />
                    <constructor-arg>
                        <map key-type="java.lang.String" value-type="java.lang.String">
                            <entry key="ns" value="http://test" />
                        </map>
                    </constructor-arg>
                </bean>
            </list>
        </constructor-arg>
    </bean>
...
        <from uri="direct:detached" />
            <to
                uri="xmlsecurity:sign://detached?
keyAccessor=#keyAccessorBean&amp;xpathsToIdAttributes=#xpathsToIdAttributesBean&amp;sche
maResourceUri=Test.xsd" />
            <to
                uri="xmlsecurity:verify://detached?
keySelector=#keySelectorBean&amp;schemaResourceUri=Test.xsd" />
            <to uri="mock:result" />
```

SIGNER エンドポイントの XADES-BES/EPES

***Camel 2.15.0 NORMAL** XML Advanced Electronic Signatures (XAdES) では、**XML Signature** への 拡張機能を定義します。この標準は ヨーロッパの Telecomunication Standards Institute によって定義され、電子署名のコミュニティーフレームワークで European Union Directive (1999/93/EC)に準拠する署名を 作成できます。**XAdES** は、署名フォームと呼ばれるさまざまな署名プロパティーのセットを定義します。**Signer Endpoint** の署名フォーム* **Basic Electronic Signature** *(XAdES-BES)および* **Explicit Policy Based Electronic Signature (XAdES-EPES)をサポートします。Validation Data** ***XAdES-T** および **XAdES-C** を使用した **Electronic Signature** 形式はサポートされません。**XAdES-EPES** フォームの以下のプロパティーをサポートします("?" はゼロまたは 1 回を意味します)。*

```
<QualifyingProperties Target>
    <SignedProperties>
        <SignedSignatureProperties>
            (SigningTime)?
            (SigningCertificate)?
            (SignaturePolicyIdentifier)
            (SignatureProductionPlace)?
            (SignerRole)?
        </SignedSignatureProperties>
        <SignedDataObjectProperties>
            (DataObjectFormat)?
            (CommitmentTypeIndication)?
        </SignedDataObjectProperties>
    </SignedProperties>
</QualifyingProperties>
```

XAdES-BES フォームのプロパティーは、SignaturePolicyIdentifier プロパティーが XAdES-BES の一部ではない点以外は同じです。

XAdES-BES/EPES プロパティーは Bean org.apache.camel.component.xmlsecurity.api.XAdESSignatureProperties で設定でき、org.apache.camel.component.xmlsecurity.api.DefaultXAdESSignatureProperties. XAdESSignatureProperties は、SigningCertificate プロパティーを除くすべてのプロパティーをサポートします。SigningCertificate プロパティーを取得するには、XAdESSignatureProperties.getSigningCertificate() or XAdESSignatureProperties.getSigningCertificateChain(). The class DefaultXAdESSignatureProperties メソッドを上書きする必要があります。また、キーストアとエイリアスを介して署名証明書を指定できるようにする必要があります。getSigningCertificate()以下の例は、指定できるすべてのパラメーターを示しています。特定のパラメーターが必要ない場合は、それらを省略できます。

```
Keystore keystore = ... // load a keystore
DefaultKeyAccessor accessor = new DefaultKeyAccessor();
accessor.setKeyStore(keystore);
accessor.setPassword("password");
accessor.setAlias("cert_alias"); // signer key alias

DefaultXAdESSignatureProperties props = new DefaultXAdESSignatureProperties();
props.setNamespace("http://uri.etsi.org/01903/v1.3.2#"); // sets the namespace for the
XAdES elements; the namspace is related to the XAdES version, default value is
"http://uri.etsi.org/01903/v1.3.2#", other possible values are "http://uri.etsi.org/01903/v1.1.1#" and
"http://uri.etsi.org/01903/v1.2.2#"
props.setPrefix("etsi"); // sets the prefix for the XAdES elements, default value is "etsi"

// signing certificate
```

```
        props.setKeystore(keystore));
        props.setAlias("cert_alias"); // specify the alias of the signing certificate in the keystore =
signer key alias
        props.setDigestAlgorithmForSigningCertificate(DigestMethod.SHA256); // possible values for
the algorithm are "http://www.w3.org/2000/09/xmldsig#sha1",
"http://www.w3.org/2001/04/xmlenc#sha256", "http://www.w3.org/2001/04/xmldsig-
more#sha384", "http://www.w3.org/2001/04/xmlenc#sha512", default value is
"http://www.w3.org/2001/04/xmlenc#sha256"
        props.setSigningCertificateURIs(Collections.singletonList("http://certuri"));

        // signing time
        props.setAddSigningTime(true);

        // policy
        props.setSignaturePolicy(XAdESSignatureProperties.SIG_POLICY_EXPLICIT_ID);
        // also the values XAdESSignatureProperties.SIG_POLICY_NONE ("None"), and
XAdESSignatureProperties.SIG_POLICY_IMPLIED ("Implied")are possible, default value is
XAdESSignatureProperties.SIG_POLICY_EXPLICIT_ID ("ExplicitId")
        // For "None" and "Implied" you must not specify any further policy parameters
        props.setSigPolicyId("urn:oid:1.2.840.113549.1.9.16.6.1");
        props.setSigPolicyIdQualifier("OIDAsURN"); //allowed values are empty string, "OIDAsURI",
"OIDAsURN"; default value is empty string
        props.setSigPolicyIdDescription("invoice version 3.1");
        props.setSignaturePolicyDigestAlgorithm(DigestMethod.SHA256);// possible values for the
algorithm are "http://www.w3.org/2000/09/xmldsig#sha1",
http://www.w3.org/2001/04/xmlenc#sha256", "http://www.w3.org/2001/04/xmldsig-more#sha384",
"http://www.w3.org/2001/04/xmlenc#sha512", default value is
http://www.w3.org/2001/04/xmlenc#sha256"
        props.setSignaturePolicyDigestValue("Ohixl6upD6av8N7pEvDABhEL6hM=");
        // you can add  qualifiers for the signature policy either by specifying text or an XML fragment
with the root element "SigPolicyQualifier"
        props.setSigPolicyQualifiers(Arrays
            .asList(new String[] {
                "<SigPolicyQualifier xmlns=\"http://uri.etsi.org/01903/v1.3.2#\">
<SPURI>http://test.com/sig.policy.pdf</SPURI><SPUserNotice><ExplicitText>display
text</ExplicitText>"
                    + "</SPUserNotice></SigPolicyQualifier>", "category B" }));
        props.setSigPolicyIdDocumentationReferences(Arrays.asList(new String[]
{"http://test.com/policy.doc.ref1.txt",
            "http://test.com/policy.doc.ref2.txt" }));

        // production place
        props.setSignatureProductionPlaceCity("Munich");
        props.setSignatureProductionPlaceCountryName("Germany");
        props.setSignatureProductionPlacePostalCode("80331");
        props.setSignatureProductionPlaceStateOrProvince("Bavaria");

        //role
        // you can add claimed roles either by specifying text or an XML fragment with the root
element "ClaimedRole"
        props.setSignerClaimedRoles(Arrays.asList(new String[] {"test",
            "<a:ClaimedRole xmlns:a=\"http://uri.etsi.org/01903/v1.3.2#\">
<TestRole>TestRole</TestRole></a:ClaimedRole>" }));
        props.setSignerCertifiedRoles(Collections.singletonList(new
XAdESEncapsulatedPKIData("Ahixl6upD6av8N7pEvDABhEL6hM=",
            "http://uri.etsi.org/01903/v1.2.2#DER", "IdCertifiedRole")));
```

```
        // data object format
        props.setDataObjectFormatDescription("invoice");
        props.setDataObjectFormatMimeType("text/xml");
        props.setDataObjectFormatIdentifier("urn:oid:1.2.840.113549.1.9.16.6.2");
        props.setDataObjectFormatIdentifierQualifier("OIDAsURN"); //allowed values are empty
string, "OIDAsURI", "OIDAsURN"; default value is empty string
        props.setDataObjectFormatIdentifierDescription("identifier desc");
        props.setDataObjectFormatIdentifierDocumentationReferences(Arrays.asList(new String[] {
            "http://test.com/dataobject.format.doc.ref1.txt",
"http://test.com/dataobject.format.doc.ref2.txt" }));

        //commitment
        props.setCommitmentTypeId("urn:oid:1.2.840.113549.1.9.16.6.4");
        props.setCommitmentTypeIdQualifier("OIDAsURN"); //allowed values are empty string,
"OIDAsURI", "OIDAsURN"; default value is empty string
        props.setCommitmentTypeIdDescription("description for commitment type ID");
        props.setCommitmentTypeIdDocumentationReferences(Arrays.asList(new String[]
{"http://test.com/commitment.ref1.txt",
            "http://test.com/commitment.ref2.txt" }));
        // you can specify a commitment type qualifier either by simple text or an XML fragment with
root element "CommitmentTypeQualifier"
        props.setCommitmentTypeQualifiers(Arrays.asList(new String[] {"commitment qualifier",
            "<c:CommitmentTypeQualifier xmlns:c=\"http://uri.etsi.org/01903/v1.3.2#\"><C>c</C>
</c:CommitmentTypeQualifier>" }));

 beanRegistry.bind("xmlSignatureProperties",props);
 beanRegistry.bind("keyAccessorDefault",keyAccessor);

 // you must reference the properties bean in the "xmlsecurity" URI
 from("direct:xades").to("xmlsecurity:sign://xades?
keyAccessor=#keyAccessorDefault&properties=#xmlSignatureProperties")
                    .to("mock:result");
```

```
    ...
    <from uri="direct:xades" />
            <to
                uri="xmlsecurity:sign://xades?
keyAccessor=#accessorRsa&amp;properties=#xadesProperties" />
            <to uri="mock:result" />
    ...
    <bean id="xadesProperties"
        class="org.apache.camel.component.xmlsecurity.api.XAdESSignatureProperties">
        <!-- For more properties see the the previous Java DSL example.
             If you want to have a signing certificate then use the bean class
DefaultXAdESSignatureProperties (see the previous Java DSL example). -->
        <property name="signaturePolicy" value="ExplicitId" />
        <property name="sigPolicyId" value="http://www.test.com/policy.pdf" />
        <property name="sigPolicyIdDescription" value="factura" />
        <property name="signaturePolicyDigestAlgorithm"
value="http://www.w3.org/2000/09/xmldsig#sha1" />
        <property name="signaturePolicyDigestValue" value="Ohixl6upD6av8N7pEvDABhEL1hM="
/>
        <property name="signerClaimedRoles" ref="signerClaimedRoles_XMLSigner" />
        <property name="dataObjectFormatDescription" value="Factura electrónica" />
        <property name="dataObjectFormatMimeType" value="text/xml" />
    </bean>
    <bean class="java.util.ArrayList" id="signerClaimedRoles_XMLSigner">
        <constructor-arg>
            <list>
                <value>Emisor</value>
                <value>&lt;ClaimedRole
                    xmlns=&quot;http://uri.etsi.org/01903/v1.3.2#&quot;&gt;&lt;test
                    xmlns=&quot;http://test.com/&quot;&gt;test&lt;/test&gt;&lt;/ClaimedRole&gt;</value>
            </list>
        </constructor-arg>
    </bean>
```

HEADERS

ヘッダー	タイプ	説明
CamelXmlSignatureXAdESQualifyingPropertiesId	文字列	**QualifyingProperties** 要素の 'Id' 属性値の場合
CamelXmlSignatureXAdESSignedDataObjectPropertiesId	文字列	**SignedDataObjectProperties** 要素の 'Id' 属性値の場合
CamelXmlSignatureXAdESSignedSignaturePropertiesId	文字列	**SignedSignatureProperties** 要素の 'Id' 属性値の場合
CamelXmlSignatureXAdESDataObjectFormatEncoding	文字列	**DataObjectFormat** 要素の **Encoding** 要素の値

CamelXmlSignatureXAdESNamespace	文字列	XAdES 名前空間パラメーターの値を上書きします。
CamelXmlSignatureXAdESPrefix	文字列	XAdES 接頭辞パラメーターの値を上書きします。

XADES バージョン 1.4.2 に関する制限

- ***XAdES-T および XAdES-C の署名形式はサポートされません。***
- ***署名側の部分のみが実装されます。検証用の部分は現在利用できません。***
- ***'QualifyingPropertiesReference' 要素はサポートされません（仕様のセクション 6.3.2 を参照）。***
- ***SignaturePolicyId 要素に含まれる Transforms 要素のサポートはありません：SignaturePolicyIdentifier element***
- ***CounterSignature 要素のサポートなし --> UnsignedProperties 要素のサポートなし***
- ***最大 1 つの DataObjectFormat 要素。署名されるデータオブジェクトが 1 つしかないため、複数の DataObjectFormat 要素は意味を持ちません（これは XML 署名エンドポイントへの受信メッセージボディーです）。***
- ***最大 1 つの CommitmentTypeIndication 要素。署名されるデータオブジェクトが 1 つしかないため、複数の CommitmentTypeIndication要素は意味を持ちません（これは XML 署名エンドポイントへの受信メッセージボディーです）。***
- ***CommitmentTypeIndication 要素には、常に AllSignedDataObjects 要素が含まれます。CommitmentTypeIndication 要素内の ObjectReference 要素はサポートされません。***
- ***AllDataObjectsTimeStamp 要素はサポートされません。***
- ***IndividualDataObjectsTimeStamp 要素はサポートされません。***

関連項目

- *ベストプラクティス*

第187章 XMPP

XMPP コンポーネント

xmpp: コンポーネントは XMPP (Jabber)トランスポートを実装します。

URI 形式

> *xmpp://[login@]hostname[:port][/participant][?Options]*

コンポーネントは、部屋ベースと非個人の会話の両方をサポートします。コンポーネントはプロデューサーとコンシューマーの両方をサポートします(XMPP からメッセージを取得したり、XMPP にメッセージを送信したりできます)。コンシューマーモードは、部屋をサポートします。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

オプション

名前	説明
room	このオプションを指定すると、コンポーネントは MUC (Multi User Chat)に接続します。通常、MUC のドメイン名はログインドメインとは異なります。たとえば、**superman@jabber.org** で、**krypton** 部屋に加入する場合、部屋の URL は **krypton@conference.jabber.org** となります。**会議** の部分に注意します。
user	ユーザー名（サーバー名なし）。指定しない場合、匿名ログインが試行されます。
password	Password。
resource	XMPP リソース。デフォルトは **Camel** です。
createAccount	**true** の場合、アカウントの作成の試行が行われます。デフォルトは **false** です。
参加者	メッセージを受信するユーザーの JID (Jabber ID)。**room** パラメーターは、**参加者** よりも優先されます。

ニックネーム	部屋への参加時にニックネームを使用します。room が指定され、ニックネームが指定されていない場合、**ユーザー** はニックネームに使用されます。
serviceName	接続しているサービスの名前。Google Talk の場合、これは **gmail.com** になります。
testConnectionOnStartup	*Camel 2.11* 起動時に接続をテストするかどうかを指定します。これは、ルートの開始時に XMPP クライアントに XMPP サーバーへの有効な接続を確保するために使用されます。接続を確立できない場合に Camel は起動時に例外を出力します。このオプションが false に設定されている場合、Camel はプロデューサーが必要とするときにレイジー接続を確立しようとし、接続が確立されるまでコンシューマー接続をポーリングします。デフォルトは **true** です。
connectionPollDelay	*Camel 2.11* XMPP 接続の正常性を検証するポーリングの間隔（秒単位）、または最初のコンシューマー接続を確立しようとする間隔（秒単位）。非アクティブになると、Camel は接続の再確立を試みます。デフォルトは **10 秒** です。
pubsub	**Camel 2.15:** 入力で pubsub パケットを許可します。デフォルトは **false** です。
doc	**Camel 2.15:** 着信パケットの Document フォームを含む **In** メッセージに doc ヘッダーを設定します。デフォルトは **true** で、存在または pubsub は true、それ以外の場合は **false** です。

ヘッダーおよび SUBJECT または LANGUAGE の設定

Apache Camel は、メッセージ IN ヘッダーを XMPP メッセージのプロパティーとして設定します。ヘッダーのカスタムフィルターが必要な場合は、HeaderFilterStategy を設定できます。XMPP メッセージの Subject および Language も IN ヘッダーとして提供されている場合に設定されます。

例

ユーザー superman は、パスワード secret を使用して jabber サーバーで部屋 krypton に参加するためのスーパーマンです。

```
xmpp://superman@jabber.org/?room=krypton@conference.jabber.org&password=secret
```

ユーザー superman は、joker にメッセージを送信します。

```
xmpp://superman@jabber.org/joker@jabber.org?password=secret
```

Java でのルーティングの例：

```
from("timer://kickoff?period=10000").
setBody(constant("I will win!\n Your Superman.")).
to("xmpp://superman@jabber.org/joker@jabber.org?password=secret");
```

joker からキューにあるすべてのメッセージを evil.talk に書き込むコンシューマー設定。

```
from("xmpp://superman@jabber.org/joker@jabber.org?password=secret").
to("activemq:evil.talk");
```

部屋メッセージをリッスンするコンシューマーの設定：

```
from("xmpp://superman@jabber.org/?
password=secret&room=krypton@conference.jabber.org").
to("activemq:krypton.talk");
```

短い表記の部屋（ドメイン部分なし）：

```
from("xmpp://superman@jabber.org/?password=secret&room=krypton").
to("activemq:krypton.talk");
```

Google Chat サービスに接続する場合は、serviceName と認証情報を指定する必要があります。

```
// send a message from fromuser@gmail.com to touser@gmail.com
from("direct:start").
        to("xmpp://talk.google.com:5222/touser@gmail.com?
serviceName=gmail.com&user=fromuser&password=secret").
        to("mock:result");
```

第188章 XQUERY エンドポイント

XQUERY

xquery: コンポーネントを使用すると、XQuery テンプレートを使用してメッセージを処理できます。これは、Templating を使用して要求の応答を生成する場合に理想的です。

URI 形式

```
xquery:templateName
```

templateName は、呼び出すテンプレートのクラスパスローカル URI、またはリモートテンプレートの完全な URL です。

たとえば、以下のように使用できます。

```
from("activemq:My.Queue").
  to("xquery:com/acme/mytransform.xquery");
```

XQuery テンプレートを使用して InOut メッセージエクスチェンジ(JMSReplyTo ヘッダーがある)のメッセージへの応答を形成。

InOnly を使用し、メッセージを消費して別の宛先に送信する場合は、以下のルートを使用できます。

```
from("activemq:My.Queue").
  to("xquery:com/acme/mytransform.xquery").
  to("activemq:Another.Queue");
```

第189章 XSLT

XSLT

xslt: コンポーネントを使用すると、XSLT テンプレートを使用してメッセージを処理できます。これは、Templating を使用してリクエストの応答を生成する場合に理想的です。

URI 形式

```
xslt:templateName[?options]
```

templateName は、呼び出すテンプレートのクラスパスローカル URI、またはリモートテンプレートの完全な URL です。URI 構文の詳細は、Spring ドキュメント を参照してください。

URI にクエリーオプションは ?option=value&option=value&.. の形式で追加できます。

以下は URI の例です。

URI	説明
xslt:com/acme/mytransform.xsl	クラスパスのファイル **com/acme/mytransform.xsl** を参照します。
xslt:file:///foo/bar.xsl	ファイル(**/foo/bar.xsl**)を参照します。
xslt:http://acme.com/cheese/foo.xsl	リモート HTTP リソースを参照します。

Camel 2.9 以降では、XSLT コンポーネントは camel-core に直接提供されます。

オプション

名前	デフォルト値	説明

converter	**null**	デフォルトの XmlConverter を上書きするオプション。レジストリーでコンバーターを検索します。提供される変換のタイプは org.apache.camel.converter.jaxp.XmlConverter である必要があります。
transformerFactory	**null**	デフォルトの TransformerFactory を上書きするオプション。レジストリーで transformerFactory を検索します。提供されるトランスフォーマーファクトリーのタイプは **javax.xml.transform.TransformerFactory** である必要があります。
transformerFactoryClass	**null**	デフォルトの TransformerFactory を上書きするオプション。**TransformerFactoryClass** インスタンスを作成し、コンバーターに設定します。
uriResolverFactory	**DefaultXsltUriResolverFactory**	Camel 2.17: エンドポイントごとに URI リゾルバーを作成する **org.apache.camel.component.xslt.XsltUriResolverFactory** を参照します。デフォルトの実装は **org.apache.camel.component.xslt.DefaultXsltUriResolverFactory** のインスタンスを返します。これにより、エンドポイントごとにデフォルトの URI リゾルバー org.apache.camel.builder.xml.XsltUriResolver が作成されます。デフォルトの URI リゾルバーは、クラスパスとファイルシステムから XSLT ドキュメントを読み取ります。URI リゾルバーがエンドポイントで指定されたルート XSLT ドキュメントのリソース URI に依存する場合、このオプション **uriResolver** ではなくこのオプションが使用されます。たとえば、デフォルトの URI リゾルバーを拡張する場合などです。このオプションは XSLT コンポーネントでも利用できるため、リソースリゾルバーファクトリーはすべてのエンドポイントに 1 回のみ設定できます。

uriResolver	**null**	Camel 2.3: カスタムの **javax.xml.transformation.URIResolver** を使用できます。Camel はデフォルトで、クラスパスからロードできる独自の実装 **org.apache.camel.builder.xml.XsltUriResolver** を使用します。
resultHandlerFactory	**null**	Camel 2.3: カスタムの **org.apache.camel.builder.xml.ResultHandler** タイプを使用できるカスタム **org.apache.camel.builder.xml.ResultHandlerFactory** を使用できます。
failOnNullBody	**true**	Camel 2.3: 入力ボディーが null の場合に例外を出力するかどうか。
deleteOutputFile	**false**	Camel 2.6: **output=file** がある場合、このオプションは エクスチェンジ の処理が完了したときに出力ファイルを削除するかどうかを決定します。たとえば、出力ファイルが一時ファイルである場合は、使用後に削除することが推奨されます。
output	**string**	Camel 2.3: 使用する出力タイプを指定するオプション。使用できる値は 、**string**、**byte**、**DOM**、**file** です。最初の 3 つのオプションは、すべてメモリーベースで、**ファイル** は直接 **java.io.File** にストリーミングされます。**ファイル** の場合、キー **Exchange.XSLT_FILE_NAME** で IN ヘッダーのファイル名を指定する **必要** があります。これは **CamelXsltFileName** です。また、ファイル名につながるパスも事前に作成する必要があります。作成しない場合は、実行時に例外が発生します。
contentCache	**true**	Camel 2.6: 読み込み時のリソースコンテンツ（スタイルシートファイル）のキャッシュ。**false** に設定すると、Camel は各メッセージ処理のスタイルシートファイルを再読み込みします。これは開発に適しています。

allowStAX	**true**	Camel 2.8.3/2.9: StAX を **javax.xml.transform.Source** として使用することを許可するかどうか。
transformerCacheSize	**0**	Camel 2.9.3/2.10.1: **Template.newTransformer（）への呼び出しを回避するために再利用するためにキャッシュされる javax.xml.transform.Transformer** オブジェクトの数。
saxon	**false**	Camel 2.11: Saxon を **transformerFactoryClass** として使用するかどうか。有効にすると、クラス **net.sf.saxon.TransformerFactoryImpl** が使用されます。Saxon をクラスパスに追加する必要があります。
saxonExtensionFunctions	**null**	Camel 2.17: 1 つ以上のカスタム **net.sf.saxon.lib.ExtensionFunctionDefinition** を設定できます。Saxon をクラスパスに追加する必要があります。このオプションを設定すると、**saxon** オプションが自動的に有効になります。
errorListener		Camel 2.14: カスタム **javax.xml.transform.ErrorListener** を使用するように設定できます。これを行う場合は、プロパティーが使用されていないため、エラーや致命的なエラーを取得し、エクスチェンジに関する情報を保存するデフォルトのエラーリスナーに注意してください。そのため、このオプションは特別なユースケースにのみ使用してください。

XSLT エンドポイントの使用

たとえば、以下のようなものを使用できます。

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl");
```

XSLT テンプレートを使用して InOut メッセージエクスチェンジ(JMSReplyTo ヘッダーがある)のメッセージの応答を形成するには、以下を行います。

InOnly を使用してメッセージを消費し、別の宛先に送信する場合は、以下のルートを使用できます。

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl").
  to("activemq:Another.Queue");
```

連携する XSLT へのパラメーターの取得

デフォルトでは、すべてのヘッダーが XSLT で利用可能なパラメーターとして追加されます。これを実行するには、パラメーターを宣言して を使用できるようにする必要があり ます。

```
<setHeader headerName="myParam"><constant>42</constant></setHeader>
<to uri="xslt:MyTransform.xsl"/>
```

また、XSLT を使用できるようにするには、最上位レベルで宣言する必要があります。

```
<xsl: ...... >

   <xsl:param name="myParam"/>

    <xsl:template ...>
```

SPRING XML バージョン

Spring XML で上記の例を使用するには、たとえば を使用します。

```
  <camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
    <route>
      <from uri="activemq:My.Queue"/>
      <to uri="xslt:org/apache/camel/spring/processor/example.xsl"/>
      <to uri="activemq:Another.Queue"/>
    </route>
  </camelContext>
```

具体的な例が必要な場合は、テストケース と その Spring XML があります。

XSL:INCLUDE の使用

Camel は独自の URIResolver の実装を提供します。これにより、Camel は含まれるファイルをクラスパスからロードでき、以前よりもインテリジェントなファイルを読み込むことができます。

たとえば、以下のようになります。

```
<xsl:include href="staff_template.xsl"/>
```

これで開始エンドポイントからの相対的な位置が置かれます。以下に例を示します。

```
.to("xslt:org/apache/camel/component/xslt/staff_include_relative.xsl")
```

つまり、Camel は クラス パス内のファイルを org/apache/camel/component/xslt/staff_template.xsl として特定します。これにより、org/apache/camel/component/xslt の例と同じフォルダーに xsl include を使用し、xsl ファイルを配置できます。

以下の 2 つの接頭辞 classpath: または file: を使用して、クラスパスまたはファイルシステムのいずれかを検索するよう Camel に指示することができます。接頭辞を省略すると、Camel はエンドポイント設定から接頭辞を使用します。の両方に がない場合、クラスパスが想定されます。

などのパスで再参照することもできます。

```
<xsl:include href="../staff_other_template.xsl"/>
```

次に、org/apache/camel/component の下にある xsl ファイルを解決します。

XSL:INCLUDE およびデフォルト接頭辞の使用

xsl:include を使用する場合：

```
<xsl:include href="staff_template.xsl"/>
```

その後、Camel 2.10.3 以前は、Camel は classpath: をデフォルトの接頭辞として使用し、クラスパスからリソースをロードします。これはほとんどの場合機能しますが、開始リソースをファイルからロードするように設定すると、

```
.to("xslt:file:etc/xslt/staff_include_relative.xsl")
```

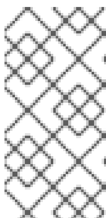
.. 次に、すべてのインクルードの前に "file:" も指定する必要があります。

```
<xsl:include href="file:staff_template.xsl"/>
```

Camel 2.10.4 以降では、Camel はエンドポイント設定の接頭辞をデフォルトの接頭辞として使用するため、これはより簡単になりました。そのため、Camel 2.10.4 以降では以下を実行できます。

```
<xsl:include href="staff_template.xsl"/>
```

エンドポイントが接頭辞として file: で設定されているため、ファイルシステムから staff_template.xsl リソースをロードします。接頭辞を明示的に設定し、一致させることもできます。と には、ファイルとクラスパスの両方の読み込みがあります。ただし、ほとんどのユーザーはファイルまたはクラスパスベースのリソースを使用するため、これは一般的ではありません。



注記

XSLT は、ref: を接頭辞として使用して、レジストリーからリソースファイルを読み込むことができます。

SAXON 拡張機能の使用

Saxon 9.2 では、拡張関数の記述が 統合エクステンション関数 と呼ばれる新しいメカニズムで補完されているため、camel を簡単に使用できるようになりました。

Java の例：

```
SimpleRegistry registry = new SimpleRegistry();
registry.put("function1", new MyExtensionFunction1());
registry.put("function2", new MyExtensionFunction2());

CamelContext context = new DefaultCamelContext(registry);
context.addRoutes(new RouteBuilder() {
   @Override
   public void configure() throws Exception {
       from("direct:start")
           .to("xslt:org/apache/camel/component/xslt/extensions/extensions.xslt?
saxonExtensionFunctions=#function1,#function2");
   }
});
```

Spring XML の例:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:extensions"/>
    <to uri="xslt:org/apache/camel/component/xslt/extensions/extensions.xslt?
saxonExtensionFunctions=#function1,#function2"/>
  </route>
</camelContext>


<bean id="function1" class="org.apache.camel.component.xslt.extensions.MyExtensionFunction1"/>
<bean id="function2" class="org.apache.camel.component.xslt.extensions.MyExtensionFunction2"/>
```

動的スタイルシート

ランタイム時に動的スタイルシートを指定するには、動的 URI を定義します。たとえば、Java DSL の .recipientList コマンドを使用して呼び出される Recipient List Enterprise Integration Pattern (EIP)を使用してこれを実行できます。

XSLT ERRORLISTENER からの警告、エラー、および致命的なエラーへのアクセス

Camel 2.14 から利用可能

Camel 2.14 以降では、警告/エラーまたは fatalError は、キー Exchange.XSLT_ERROR、Exchange.XSLT_FATAL_ERROR、または Exchange.XSLT_WARNING を持つプロパティーとして現在のエクスチェンジに保存されます。これにより、エンドユーザーは変換中に発生するエラーを保持できます。

たとえば、以下のスタイルシートでは、スタッフに空の dob フィールドがある場合は終了します。xsl:message を使用したカスタムエラーメッセージを含めるには、以下を行います。

```
<xsl:template match="/">
  <html>
    <body>
      <xsl:for-each select="staff/programmer">
        <p>Name: <xsl:value-of select="name"/><br />
          <xsl:if test="dob=''">
            <xsl:message terminate="yes">Error: DOB is an empty string!</xsl:message>
          </xsl:if>
        </p>
      </xsl:for-each>
```

```
    </body>
  </html>
</xsl:template>
```

この情報は、例外の getMessage() メソッドにメッセージが含まれる例外として保存されたエクスチェンジでは使用できません。例外が鍵の警告としてエクスチェンジに保存されます。Exchange.XSLT_WARNING.

第190章 XSTREAM

XSTREAM コンポーネント

XStream コンポーネントは、XStream ライブラリーを使用して Java オブジェクトを XML との間でマーシャリングおよびアンマーシャリングする XStream データフォーマットを提供します。たとえば、以下のように xstream() DSL コマンドを使用してメッセージボディーを XML に変換できます。

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .marshal().xstream();
    }
});

camelctx.start();
try {
    ProducerTemplate producer = camelctx.createProducerTemplate();
    String customer = producer.requestBody("direct:start", new Customer("John", "Doe"),
String.class);
} finally {
    camelctx.stop();
}
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

第191章 YAMMER

YAMMER

Yammer コンポーネントを使用すると、Yammer 企業のソーシャルネットワークと対話できます。メッセージ、ユーザー、およびユーザー関係、および新規メッセージの作成がサポートされます。

Yammer は、すべてのクライアントアプリケーション認証に OAuth 2 を使用します。アカウントで camel-yammer を使用するには、Yammer 内に新しいアプリケーションを作成し、アプリケーションにアカウントへのアクセスを許可する必要があります。最後に、アクセストークンを生成します。詳細は、https://developer.yammer.com/v1.0/docs/authentication を参照してください。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-yammer</artifactId>
   <version>${camel-version}</version>
</dependency>
```

URI 形式

```
yammer:[function]?[options]
```

YAMMERCOMPONENT

yammer コンポーネントは、Yammer アカウント設定で設定可能です。これは、使用前に設定する必要があります。これらのオプションは、エンドポイントで直接設定することもできます。

オプション	説明
consumerKey	コンシューマーキー
consumerSecret	コンシューマーシークレット
accessToken	アクセストークン

メッセージの消費

***camel-yammer** コンポーネントは、メッセージを消費するための複数のエンドポイントを提供します。*

URI	説明
Yammer:messages?オプション	ユーザーの(API 呼び出しを行うのにアクセストークンが使用されている)ユーザーのすべての公開メッセージ）Yammer ネットワーク。Yammer Web インターフェイスの "All" 会話に対応します。
Yammer:my_feed?オプション	"Follow" と "Top" 会話の間の選択に基づいて、ユーザーのフィード。
Yammer:algo?options	Yammer Web インターフェイスで大部分のユーザーに表示される Top 会話に対応するユーザーのアルゴリズムフィード。
Yammer:following?オプション	ユーザーが従うユーザー、グループ、トピックに関する会話となる "Follow" フィード。
Yammer:sent?オプション	ユーザーが送信するすべてのメッセージ。
Yammer:private?オプション	ユーザーが受信したプライベートメッセージ。
Yammer:received?オプション	**Camel 2.12.1:** ユーザーが受信したすべてのメッセージ

メッセージの消費用の URI オプション

名前	デフォルト値	説明
useJson	**false**	POJO に変換せずに raw JSON を使用する場合は true に設定します。
delay	**5000**	ミリ秒単位
consumerKey	**null**	Consumer Key。代わりに、**YammerComponent** レベルでも設定できます。

consumerSecret	**null**	コンシューマーシークレット。代わりに、**YammerComponent** レベルでも設定できます。
accessToken	**null**	アクセストークン。代わりに、**YammerComponent** レベルでも設定できます。
limit	**-1**	指定されたメッセージ数のみを返します。threaded=true および threaded=extended で機能します。
threaded	**null**	threaded=true は、各スレッドの最初のメッセージのみを返します。このパラメーターは、メッセージスレッドが折りたたまれたアプリケーション向けのものです。threading=extended は、最後にアクティブなメッセージと、Yammer Web インターフェイスのデフォルトビューで表示される 2 つの最新のメッセージ順にスレッドスターターメッセージを返します。
olderThan	**-1**	数値の文字列として指定されたメッセージ ID よりも古いメッセージを返します。これは、メッセージのページネーションに役立ちます。たとえば、現在 20 個のメッセージが表示されており、最も古いメッセージが 2912 である場合は、リクエストに "?olderThan=2912? を追加して、表示しているメッセージの前に 20 個のメッセージを取得できます。
newerThan	**-1**	数値の文字列として指定されたメッセージ ID よりも新しいメッセージを返します。これは、新しいメッセージをポーリングする際に使用する必要があります。メッセージを確認し、返された最新のメッセージが 3516 の場合は、パラメーター "?newerThan=3516? で要求を行い、ページにすでにあるメッセージの複製コピーを取得しないようにします。

メッセージの形式

デフォルトでは、すべてのメッセージは org.apache.camel.component.yammer.model パッケージで提供される POJO モデルに変換されます。yammer からの元のメッセージは JSON にあります。エンドポイントの使用および生成を行うすべてのメッセージでは、Messages オブジェクトが返されます。以下のようなルートの例を以下に示します。

```
from("yammer:messages?
consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAcce
ssToken").to("mock:result");
```

yammer サーバーが以下を返すとします。


```
{
 "messages":[
  {
   "replied_to_id":null,
   "network_id":7654,
   "url":"https://www.yammer.com/api/v1/messages/305298242",
   "thread_id":305298242,
   "id":305298242,
   "message_type":"update",
   "chat_client_sequence":null,
   "body":{
    "parsed":"Testing yammer API...",
    "plain":"Testing yammer API...",
    "rich":"Testing yammer API..."
   },
   "client_url":"https://www.yammer.com/",
   "content_excerpt":"Testing yammer API...",
   "created_at":"2013/06/25 18:14:45 +0000",
   "client_type":"Web",
   "privacy":"public",
   "sender_type":"user",
   "liked_by":{
    "count":1,
    "names":[
     {
      "permalink":"janstey",
      "full_name":"Jonathan Anstey",
      "user_id":1499642294
     }

    ]

   },
   "sender_id":1499642294,
   "language":null,
   "system_message":false,
   "attachments":[

   ],
   "direct_message":false,
```

```
    "web_url":"https://www.yammer.com/redhat.com/messages/305298242"
   },
   {
    "replied_to_id":null,
    "network_id":7654,
    "url":"https://www.yammer.com/api/v1/messages/294326302",
    "thread_id":294326302,
    "id":294326302,
    "message_type":"system",
    "chat_client_sequence":null,
    "body":{
     "parsed":"(Principal Software Engineer) has [[tag:14658]] the redhat.com network. Take a
moment to welcome Jonathan.",
     "plain":"(Principal Software Engineer) has #joined the redhat.com network. Take a moment
to welcome Jonathan.",
     "rich":"(Principal Software Engineer) has #joined the redhat.com network. Take a moment
to welcome Jonathan."
    },
    "client_url":"https://www.yammer.com/",
    "content_excerpt":"(Principal Software Engineer) has #joined the redhat.com network. Take
a moment to welcome Jonathan.",
    "created_at":"2013/05/10 19:08:29 +0000",
    "client_type":"Web",
    "sender_type":"user",
    "privacy":"public",
    "liked_by":{
     "count":0,
     "names":[

     ]

    }
   }
  ]

 }
```


Camel は、これを 2 つの Message オブジェクトが含まれる Messages オブジェクトにマーシャリングします。以下に示すように、必要な情報を簡単に取得できる豊富なオブジェクトモデルがあります。

```
        Exchange exchange = mock.getExchanges().get(0);
        Messages messages = exchange.getIn().getBody(Messages.class);

        assertEquals(2, messages.getMessages().size());
        assertEquals("Testing yammer API...",
messages.getMessages().get(0).getBody().getPlain());
        assertEquals("(Principal Software Engineer) has #joined the redhat.com network. Take a
moment to welcome Jonathan.", messages.getMessages().get(1).getBody().getPlain());
```

そのため、このデータを POJO にマーシャリングすることは無料ではないので、useJson=false オプションを URI に追加して、純粋な JSON を使用するように切り替える必要がある場合は、URI に useJson=false オプションを追加します。

メッセージの作成

現在のユーザーの アカウントに新しいメッセージを作成するには、以下の URI を使用できます。

```
yammer:messages?[options]
```

現在の Camel メッセージボディーは、Yammer メッセージのテキストを設定するために使用されるものです。応答本文には、メッセージ（デフォルトでは Messages オブジェクトとして）を消費する場合と同じようにフォーマットされた新しいメッセージが含まれます。

たとえば、以下のルートを作成します。

```
from("direct:start").to("yammer:messages?
consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAcce
ssToken").to("mock:result");
```

direct:start エンドポイントに "Hi from Camel!" メッセージボディーを送信すると、以下を行います。

```
template.sendBody("direct:start", "Hi from Camel!");
```

新しいメッセージがサーバーの現在のユーザーのアカウントに作成され、この新しいメッセージは Camel に返され、Messages オブジェクトに変換されます。メッセージを使用するときと同様に、Messages オブジェクトを以下のように対話できます。

```
Exchange exchange = mock.getExchanges().get(0);
Messages messages = exchange.getIn().getBody(Messages.class);

assertEquals(1, messages.getMessages().size());
assertEquals("Hi from Camel!", messages.getMessages().get(0).getBody().getPlain());
```

ユーザー関係の取得

camel-yammer コンポーネントは、ユーザー関係を取得できます。

```
yammer:relationships?[options]
```

関係を取得するための URI オプション

名前	デフォルト値	説明
useJson	**false**	POJO に変換せずに raw JSON を使用する場合は true に設定します。
delay	**5000**	ミリ秒単位
consumerKey	**null**	Consumer Key。代わりに、**YammerComponent** レベルでも設定できます。
consumerSecret	**null**	コンシューマーシークレット。代わりに、**YammerComponent** レベルでも設定できます。
accessToken	**null**	アクセストークン。代わりに、**YammerComponent** レベルでも設定できます。
userId	**現在のユーザー**	現在のユーザー以外のユーザーの関係を表示します。

ユーザーの取得

camel-yammer コンポーネントは、ユーザーを取得するための複数のエンドポイントを提供します。

URI	説明
yammer:users?[options]	現在のユーザーの Yammer ネットワークでユーザーを取得します。
yammer:current?[options]	現在のユーザーのデータを表示します。

ユーザーを取得するための URI オプション

名前	デフォルト値	説明

useJson	**false**	POJO に変換せずに raw JSON を使用する場合は true に設定します。
delay	**5000**	ミリ秒単位
consumerKey	**null**	Consumer Key。代わりに、**YammerComponent** レベルでも設定できます。
consumerSecret	**null**	コンシューマーシークレット。代わりに、**YammerComponent** レベルでも設定できます。
accessToken	**null**	アクセストークン。代わりに、**YammerComponent** レベルでも設定できます。

ENRICHER の使用

camel-yammer のポーリングコンシューマーの 1 つで始まるルートではなく、Enricher パターンを使用すると便利です（または、常にユーザーや関係コンシューマーの場合）。これは、コンシューマーが繰り返し実行されますが、多くの場合、遅延を設定します。ユーザーのデータを検索するか、一度にメッセージを取得する場合は、そのコンシューマーを一度呼び出してルートで取得することが推奨されます。

ある時点でルートを取得して、現在のユーザーのユーザーデータをフェッチする必要があるとします。このユーザーのポーリングを再度ポーリングするのではなく、pollEnrich DSL メソッドを使用します。

```
from("direct:start").pollEnrich("yammer:current?consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAccessToken").to("mock:result");
```

これにより、現在のユーザーの User オブジェクトを取得して、Camel メッセージボディーとして設定します。

第192章 ZOOKEEPER

ZOOKEEPER

Camel 2.9 以降で利用可能

ZooKeeper コンポーネントは、ZooKeeper クラスターとの対話を許可し、以下の機能を Camel に公開します。

1. ZooKeeper 作成モードのいずれかのノードの作成。

2. 任意のクラスターノードのデータコンテンツを取得および設定します（設定されるデータは byte[]に変換する必要があります）。

3. 特定のノードに割り当てられている子ノードの一覧を作成して取得します。

4. ZooKeeper によって調整された Leader 選択を利用して、エクスチェンジを処理する必要があるかどうかを判断する Distributed RoutePolicy。

Maven ユーザーは、このコンポーネントの pom.xml に以下の依存関係を追加する必要があります。

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-zookeeper</artifactId>
    <version>2.17.0.redhat-630xxx</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

CAMEL ON EAP デプロイメント

このコンポーネントは、Red Hat JBoss Enterprise Application Platform (JBoss EAP) コンテナー上で簡素化されたデプロイメントモデルを提供する Camel on EAP (Wildfly Camel) フレームワークによってサポートされます。このモデルの詳細は、Deploying into a Web Server の Apache Camel on JBoss EAP の章を参照してください。

URI 形式

```
zookeeper://zookeeper-server[:port][/path][?options]
```

uri のパスは、エンドポイントのターゲットとなる ZooKeeper サーバー（別名 znode）のノードを指定します。

オプション

名前	デフォルト値	説明
path		ZooKeeper サーバー（別名 znode）のノード
listChildren	**false**	ノードの子をリストすべきかどうか。
repeat	**false**	znode への変更は監視され、繰り返し処理される必要があります。
backoff	**5000**	再試行する前にエラーをバックオフする間隔。
timeout	**5000**	タイムアウトする前に接続を待機する時間間隔。
create	**false**	まだ存在していない場合は、エンドポイントがノードを作成するはずです。
createMode	**EPHEMERAL**	新規に作成されたノードに使用する create モード（以下を参照）。
sendEmptyMessageOnDelete	**true**	**Camel 2.10:** znode の削除で、空のメッセージがコンシューマーに送信される必要がある

ユースケース

ZNODE からの読み取り。

以下のスニペットは、すでに存在している場合に znode '/somepath/somenode/' からデータを読み取ります。取得したデータはエクスチェンジに置かれ、残りのルートに渡されます。

```
from("zookeeper://localhost:39913/somepath/somenode").to("mock:result");
```

ノードが存在しない場合は、エンドポイントの作成を待たせるフラグを指定できます。

```
from("zookeeper://localhost:39913/somepath/somenode?
awaitCreation=true").to("mock:result");
```

ZNODE からの読み取り - (追加の CAMEL 2.10 以降)

ZooKeeper アンサンブルから受信された WatchedEvent によりデータが読み取られると、CamelZookeeperEventType ヘッダーはその WatchedEvent からの ZooKeeper の EventType 値を保持します。データが最初に読み取られる場合(WatchedEvent によってトリガーされない)、CamelZookeeperEventType ヘッダーは設定されません。

ZNODE への書き込み。

以下のスニペットは、エクスチェンジのペイロードを '/somepath/somenode/' にすでに存在している場合に znode に書き込みます。

```
from("direct:write-to-znode").to("zookeeper://localhost:39913/somepath/somenode");
```

エンドポイントを使用すると、ターゲットの znode をメッセージヘッダーとして動的に指定することができます。文字列 CamelZooKeeperNode でキーのヘッダーが存在する場合、ヘッダーの値はサーバー上の znode へのパスとして使用されます。たとえば、上記の同じルート定義を使用する場合、以下のコードスニペットは /somepath/somenode ではなく、ヘッダー /somepath/someothernode からパスに書き込みます。ZooKeeper に保存されているデータはバイトベースであるため、testPayload は byte[] に変換する必要があります。

```
Object testPayload = ...
template.sendBodyAndHeader("direct:write-to-znode", testPayload, "CamelZooKeeperNode",
"/somepath/someothernode");
```

ノードが存在しない場合にも作成するには、'create' オプションを使用する必要があります。

```
from("direct:create-and-write-to-
znode").to("zookeeper://localhost:39913/somepath/somenode?create=true");
```

バージョン 2.11 以降では、'DELETE' に設定してヘッダー 'CamelZookeeperOperation' を使用してノードを 削除 することもできます。

```
from("direct:delete-znode").setHeader(ZooKeeperMessage.ZOOKEEPER_OPERATION,
constant("DELETE")).to("zookeeper://localhost:39913/somepath/somenode");
```

または同等に

```
<route>
  <from uri="direct:delete-znode" />
  <setHeader headerName="CamelZookeeperOperation">
    <constant>DELETE</constant>
  </setHeader>
  <to uri="zookeeper://localhost:39913/somepath/somenode" />
</route>
```

ZooKeeper ノードは異なるタイプを持つことができます。これらは 'Ephemeral' または 'Persistent' および 'Sequenced' または 'Unsequenced' にすることができます。各タイプの詳細は、を参照してください。デフォルトでは、エンドポイントは、配列されていない一時ノードを作成しますが、タイプは uri 設定パラメーターまたは特別なメッセージヘッダーを介して簡単に操作できます。create モードに想定される値は、CreateMode 列挙から簡単に名前です。

- PERSISTENT
- PERSISTENT_SEQUENTIAL
- EPHEMERAL
- EPHEMERAL_SEQUENTIAL

たとえば、URI 設定を介して永続的な znode を作成するには、以下を実行します。

```
from("direct:create-and-write-to-persistent-
znode").to("zookeeper://localhost:39913/somepath/somenode?
create=true&createMode=PERSISTENT");
```

または、ヘッダー CamelZookeeperCreateMode を使用します。ZooKeeper に保存されているデータはバイトベースであるため、testPayload は byte[] に変換する必要があります。

```
Object testPayload = ...
template.sendBodyAndHeader("direct:create-and-write-to-persistent-znode", testPayload,
"CamelZooKeeperCreateMode", "PERSISTENT");
```

ZOOKEEPER が有効な ROUTE ポリシー。

ZooKeeper では、追加設定なしで非常にシンプルで効果的なリーダーの選択が可能になります。このコンポーネントは、RoutePolicy でこの選択機能を悪用し、ルートが有効であるタイミングと方法を制御します。このポリシーは通常、フェイルオーバーのシナリオで使用され、Camel ベースのサーバーのクラスター全体でルートの同一インスタンスを制御します。非常に一般的なシナリオは、クラスター全体で分散されるルートのインスタンスが複数あり、そのうちの 1 つのインスタンスのみが一度に実行される必要がある単純な Master-Slave 設定です。マスターが失敗した場合は、利用可能なスレーブから新規マスターを選択し、この新規マスターのルートを起動する必要があります。

このポリシーは、選出に関与する RoutePolicy のすべてのインスタンスで共通の znode パスを使用します。各ポリシーはこの ID をこのノードに書き込みます。zookeeper は書き込みを受信順に順序付けます。次に、ポリシーはノードの一覧を読み取り、その ID の投稿を確認します。この投稿は、ルートを開始する必要があるかどうかを判断するために使用されます。ポリシーは、クラスター全体で起動する必要があるルートインスタンスの数で起動時に設定され、リスト内の位置がこの値よりも小さい場合、そのルートが起動します。マスター/スレーブのシナリオでは、ルートはルートインスタンス 1 つで設定され、一覧の最初のエントリーのみがルートを起動します。すべてのポリシーは、リストへの更新を監視し、リストによってルートが開始される必要がある場合に再度計算されるかどうかを監視します。Zookeeper の Leader 選択機能の詳細は、このページ を参照してください。

以下の例では、選択にノード /someapplication/somepolicy を使用し、ノード一覧で上位 1 エントリーのみを起動するように設定されます。つまり、マスターを選択します。

```
ZooKeeperRoutePolicy policy = new
ZooKeeperRoutePolicy("zookeeper:localhost:39913/someapp/somepolicy", 1);
from("direct:policy-controlled").routePolicy(policy).to("mock:controlled");
```