



Red Hat Integration 2022.Q2

Camel K を使用したインテグレーションの開発および管理

Camel K の開発者ガイド

Red Hat Integration 2022.Q2 Camel K を使用したインテグレーションの開発および管理

Camel K の開発者ガイド

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/Developing_and_Managing_Integrations_Using_Camel_K.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

Camel K アプリケーションの開発、設定、および管理の基本

目次

前書き	5
多様性を受け入れるオープンソースの強化	5
第1章 CAMEL K インテグレーションの管理	6
1.1. CAMEL K インテグレーションの管理	6
1.2. CAMEL K インテグレーションのロギングレベルの管理	8
1.3. CAMELK 統合のスケーリング	10
第2章 CAMEL K インテグレーションのモニタリング	11
2.1. OPENSIFT でのユーザーワークロードモニタリングの有効化	11
2.2. CAMEL K インテグレーションメトリクスの設定	12
2.3. カスタム CAMEL K インテグレーションメトリクスの追加	12
第3章 CAMEL K OPERATOR のモニタリング	16
3.1. CAMEL K OPERATOR メトリクス	16
3.2. CAMEL K OPERATOR モニタリングの有効化	16
3.3. CAMEL K OPERATOR のアラート	17
第4章 CAMEL K インテグレーションの設定	22
4.1. ビルド時の設定プロパティの指定	22
4.2. ランタイム設定オプションの指定	23
4.2.1. ランタイムプロパティの指定	24
4.2.1.1. コマンドラインでのランタイムプロパティの指定	24
4.2.1.2. プロパティファイルでのランタイムプロパティの指定	25
4.2.2. 設定値の指定	26
4.2.2.1. テキストファイルの指定	27
4.2.2.2. ConfigMap の指定	28
4.2.2.3. シークレットの指定	28
4.2.2.4. ConfigMap またはシークレットに含まれるプロパティの参照	29
4.2.2.5. ConfigMap またはシークレットから取得した設定値の絞り込み	30
4.2.3. 実行中のインテグレーションへのリソースの提供	31
4.2.3.1. リソースとしてのテキストまたはバイナリーファイルの指定	31
4.2.3.2. リソースとしての ConfigMap の指定	32
4.2.3.3. リソースとしてのシークレットの指定	33
4.2.3.4. リソースの宛先パスの指定	34
4.2.3.5. ConfigMap またはシークレットデータの絞り込み	35
4.3. CAMEL インテグレーションコンポーネントの設定	36
4.4. CAMEL K インテグレーション依存関係の設定	36
第5章 KAFKA に対する CAMEL K の認証	38
5.1. KAFKA の設定	38
5.1.1. AMQ Streams を使用した Kafka の設定	38
5.1.1.1. AMQ Streams の OpenShift クラスターの準備	38
5.1.1.2. AMQ Streams を使用した Kafka トピックの設定	39
5.1.2. OpenShift Streams を使用した Kafka の設定	40
5.1.2.1. OpenShift Streams の OpenShift クラスターの準備	40
5.1.2.2. RHOAS を使用した Kafka トピックの設定	41
5.1.2.3. Kafka クレデンシャルの取得	42
5.1.2.4. SASL/Plain 認証を使用したシークレットの作成	44
5.1.2.5. SASL/OAUTHBearer 認証を使用したシークレットの作成	44
5.2. KAFKA インテグレーションの実行	45
第6章 CAMEL K トレイト設定の参考情報	47

Camel K 機能トレイト	47
Camel K コアプラットフォームトレイト	47
6.1. CAMEL K トレイトおよびプロファイルの設定	48
6.2. CAMEL K 機能トレイト	49
6.2.1. Knative トレイト	49
6.2.1.1. 設定	49
6.2.2. Knative Service トレイト	50
6.2.2.1. 設定	50
6.2.3. Prometheus トレイト	51
6.2.3.1. 設定	52
6.2.4. Pdb トレイト	52
6.2.4.1. 設定	52
6.2.5. Pull Secret トレイト	53
6.2.5.1. 設定	53
6.2.6. Route トレイト	54
6.2.6.1. 設定	54
6.2.6.2. 例	56
6.2.6.2.1. 自己署名証明書を生成し、シークレットを作成します	56
6.2.6.2.2. ルートへの HTTP リクエストの作成	56
6.2.7. Service トレイト	58
6.2.7.1. 設定	58
6.3. CAMEL K プラットフォームトレイト	58
6.3.1. Builder トレイト	58
6.3.1.1. 設定	58
6.3.2. Container トレイト	59
6.3.2.1. 設定	59
6.3.3. Camel トレイト	61
6.3.3.1. 設定	61
6.3.4. Dependencies トレイト	61
6.3.4.1. 設定	62
6.3.5. Deployer トレイト	62
6.3.5.1. 設定	62
6.3.6. Deployment トレイト	63
6.3.6.1. 設定	63
6.3.7. Environment トレイト	63
6.3.7.1. 設定	64
6.3.8. Error Handler トレイト	64
6.3.8.1. 設定	64
6.3.9. JVM トレイト	65
6.3.9.1. 設定	65
6.3.9.2. 例	66
6.3.10. Kamelets トレイト	66
6.3.10.1. 設定	66
6.3.11. Openapi トレイト	66
6.3.11.1. 設定	67
6.3.12. Owner トレイト	67
6.3.12.1. 設定	67
6.3.13. プラットフォームトレイト	68
6.3.13.1. 設定	68
6.3.14. Quarkus トレイト	69
6.3.14.1. 設定	69
6.3.14.2. サポートされる Camel コンポーネント	69

第7章 CAMEL K コマンドリファレンス	70
7.1. CAMEL K コマンドライン	70
7.1.1. サポートされるコマンド	70
7.2. CAMEL K モードラインオプション	72

前書き

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#)をご覧ください。

第1章 CAMEL K インテグレーションの管理

Camel K コマンドラインまたは開発ツールを使用して、Red Hat Integration - Camel K インテグレーションを管理できます。本章では、コマンドラインで Camel K インテグレーションを管理する方法を説明し、VS Code 開発ツールの使用方法を説明する追加のリソースへのリンクを提供します。

- [「Camel K インテグレーションの管理」](#)
- [「Camel K インテグレーションのロギングレベルの管理」](#)
- [「CamelK 統合のスケーリング」](#)

1.1. CAMEL K インテグレーションの管理

コマンドラインで OpenShift クラスターの Camel K インテグレーションを管理するためのさまざまなオプションがあります。ここでは、以下のコマンドを使用する簡単な例を紹介します。

- **kamel get**
- **kamel describe**
- **kamel ログ**
- **kamel delete**

前提条件

- [Setting up your Camel K development environment](#)
- Java または YAML DSL で記述された Camel インテグレーションが作成済みである必要があります。

手順

1. 以下の例のように、Camel K Operator が OpenShift クラスターで稼働していることを確認します。

```
oc get pod
```

NAME	READY	STATUS	RESTARTS	AGE
camel-k-operator-86b8d94b4-pk7d6	1/1	Running	0	6m28s

2. **kamel run** コマンドを入力し、OpenShift のクラウドでインテグレーションを実行します。以下に例を示します。

```
kamel run hello.camelk.yaml
```

```
integration "hello" created
```

3. **kamel get** コマンドを入力し、インテグレーションの状態を確認します。

```
kamel get
```

```
NAME PHASE KIT
hello Building Kit kit-bqatqib5t4kse5vukt40
```

4. **kamel describe** コマンドを入力し、インテグレーションに関する詳細情報を出力します。

```
kamel describe integration hello
```

```
Name:          hello
Namespace:      myproject
Creation Timestamp: Fri, 13 Aug 2021 16:23:21 +0200
Phase:          Building Kit
Runtime Version: 1.7.1.fuse-800025-redhat-00001
Kit:            myproject/kit-c4ci6mbe9hl5ph5c9sjg
Image:
Version:        1.6.0
Dependencies:
  camel:core
  camel:log
  camel:timer
  mvn:org.apache.camel.k:camel-k-runtime
  mvn:org.apache.camel.quarkus:camel-quarkus-yaml-dsl
Sources:
  Name          Language Compression Ref Ref Key
  camel-k-embedded-flow.yaml yaml false
Conditions:
  Type          Status Reason Message
  IntegrationPlatformAvailable True IntegrationPlatformAvailable myproject/camel-k
  IntegrationKitAvailable True IntegrationKitAvailable kit-c4ci6mbe9hl5ph5c9sjg
  CronJobAvailable False CronJobNotAvailableReason different controller
strategy used (deployment)
  DeploymentAvailable True DeploymentAvailable deployment name is hello
  KnativeServiceAvailable False KnativeServiceNotAvailable different controller
strategy used (deployment)
  Ready          True ReplicaSetReady
```

5. **kamel log** コマンドを入力して、ログを **stdout** に出力します。

```
kamel log hello
```

```
...
[1] 2021-08-13 14:37:15,860 INFO [info] (Camel (camel-1) thread #0 - timer://yaml)
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from yaml]
...
```

6. **Ctrl-C** キーを押して、ターミナルでログインを終了します。
7. **kamel delete** を入力して、OpenShift にデプロイされたインテグレーションを削除します。

```
kamel delete hello
```

```
Integration hello deleted
```

- ロギングの詳細は、「[Managing Camel K integration logging levels](#)」を参照してください。
- デプロイメントのターンアラウンド時間を短縮するには、「[Running Camel K integrations in development mode](#)」を参照してください。
- インテグレーションを管理するための開発ツールの詳細は、[Red Hat による Apache Camel K の VS Code ツール](#) を参照してください。

1.2. CAMEL K インテグレーションのロギングレベルの管理

Camel K は Quarkus Logging メカニズムをインテグレーションのロギングフレームワークとして使用します。実行時にコマンドラインでさまざまなロガーのロギングレベルを設定するには、**quarkus.log.category** プレフィックスをインテグレーションプロパティとして指定します。以下に例を示します。

例

```
--property 'quarkus.log.category."org".level'=DEBUG
```



注記

プロパティを一重引用符でエスケープすることが重要です。

前提条件

- [Setting up your Camel K development environment](#)

手順

1. **kamel run** コマンドを入力し、**--property** オプションを使用してログレベルを指定します。以下に例を示します。

```
kamel run --dev --property 'quarkus.log.category."org.apache.camel.support".level'=DEBUG
Basic.java

...
integration "basic" created
  Progress: integration "basic" in phase Initialization
  Progress: integration "basic" in phase Building Kit
  Progress: integration "basic" in phase Deploying
  Condition "IntegrationPlatformAvailable" is "True" for Integration basic: myproject/camel-k
  Integration basic in phase "Initialization"
  Integration basic in phase "Building Kit"
  Integration basic in phase "Deploying"
  Condition "IntegrationKitAvailable" is "True" for Integration basic: kit-
c4dn5l62v9g3aopkocag
  Condition "DeploymentAvailable" is "True" for Integration basic: deployment name is basic
  Condition "CronJobAvailable" is "False" for Integration basic: different controller strategy
used (deployment)
  Progress: integration "basic" in phase Running
  Condition "KnativeServiceAvailable" is "False" for Integration basic: different controller
strategy used (deployment)
  Integration basic in phase "Running"
```

Condition "Ready" is "False" for Integration basic
 Condition "Ready" is "True" for Integration basic
 [1] Monitoring pod basic-575b97f64b-7l5rl
 [1] 2021-08-17 08:35:22,906 DEBUG [org.apa.cam.sup.LRUCacheFactory] (main)
 Creating DefaultLRUCacheFactory
 [1] 2021-08-17 08:35:23,132 INFO [org.apa.cam.k.Runtime] (main) Apache Camel K
 Runtime 1.7.1.fuse-800025-redhat-00001
 [1] 2021-08-17 08:35:23,134 INFO [org.apa.cam.qua.cor.CamelBootstrapRecorder] (main)
 bootstrap runtime: org.apache.camel.quarkus.main.CamelMainRuntime
 [1] 2021-08-17 08:35:23,224 INFO [org.apa.cam.k.lis.SourcesConfigurer] (main) Loading
 routes from: SourceDefinition{name='Basic', language='java',
 location='file:/etc/camel/sources/Basic.java', }
 [1] 2021-08-17 08:35:23,232 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
 RoutesBuilderLoader: org.apache.camel.dsl.java.joor.JavaRoutesBuilderLoader via: META-
 INF/services/org/apache/camel/java
 [1] 2021-08-17 08:35:23,232 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
 and using RoutesBuilderLoader:
 org.apache.camel.dsl.java.joor.JavaRoutesBuilderLoader@68dc098b
 [1] 2021-08-17 08:35:23,236 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
 ResourceResolver: org.apache.camel.impl.engine.DefaultResourceResolvers\$FileResolver
 via: META-INF/services/org/apache/camel/file
 [1] 2021-08-17 08:35:23,237 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
 and using ResourceResolver:
 org.apache.camel.impl.engine.DefaultResourceResolvers\$FileResolver@5b67bb7e
 [1] 2021-08-17 08:35:24,320 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Lookup
 Language with name simple in registry. Found:
 org.apache.camel.language.simple.SimpleLanguage@74d7184a
 [1] 2021-08-17 08:35:24,328 DEBUG [org.apa.cam.sup.EventHelper] (main) Ignoring
 notifying event Initializing CamelContext: camel-1. The EventNotifier has not been started
 yet: org.apache.camel.quarkus.core.CamelManagementEventBridge@3301500b
 [1] 2021-08-17 08:35:24,336 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Lookup
 Component with name timer in registry. Found:
 org.apache.camel.component.timer.TimerComponent@3ef41c66
 [1] 2021-08-17 08:35:24,342 DEBUG [org.apa.cam.sup.DefaultComponent] (main)
 Creating endpoint uri=[timer://java?period=1000], path=[java]
 [1] 2021-08-17 08:35:24,350 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
 ProcessorFactory: org.apache.camel.processor.DefaultProcessorFactory via: META-
 INF/services/org/apache/camel/processor-factory
 [1] 2021-08-17 08:35:24,351 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
 and using ProcessorFactory:
 org.apache.camel.processor.DefaultProcessorFactory@704b2127
 [1] 2021-08-17 08:35:24,369 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
 InternalProcessorFactory: org.apache.camel.processor.DefaultInternalProcessorFactory via:
 META-INF/services/org/apache/camel/internal-processor-factory
 [1] 2021-08-17 08:35:24,369 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
 and using InternalProcessorFactory:
 org.apache.camel.processor.DefaultInternalProcessorFactory@4f8caaf3
 [1] 2021-08-17 08:35:24,442 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Lookup
 Component with name log in registry. Found:
 org.apache.camel.component.log.LogComponent@46b695ec
 [1] 2021-08-17 08:35:24,444 DEBUG [org.apa.cam.sup.DefaultComponent] (main)
 Creating endpoint uri=[log://info], path=[info]
 [1] 2021-08-17 08:35:24,461 DEBUG [org.apa.cam.sup.EventHelper] (main) Ignoring
 notifying event Initialized CamelContext: camel-1. The EventNotifier has not been started yet:
 org.apache.camel.quarkus.core.CamelManagementEventBridge@3301500b
 [1] 2021-08-17 08:35:24,467 DEBUG [org.apa.cam.sup.DefaultProducer] (main) Starting

```

producer: Producer[log://info]
[1] 2021-08-17 08:35:24,469 DEBUG [org.apache.camel.sup.DefaultConsumer] (main) Build
consumer: Consumer[timer://java?period=1000]
[1] 2021-08-17 08:35:24,475 DEBUG [org.apache.camel.sup.DefaultConsumer] (main) Starting
consumer: Consumer[timer://java?period=1000]
[1] 2021-08-17 08:35:24,481 INFO [org.apache.camel.imp.eng.AbstractCamelContext] (main)
Routes startup summary (total:1 started:1)
[1] 2021-08-17 08:35:24,481 INFO [org.apache.camel.imp.eng.AbstractCamelContext] (main)
Started java (timer://java)
[1] 2021-08-17 08:35:24,482 INFO [org.apache.camel.imp.eng.AbstractCamelContext] (main)
Apache Camel 3.10.0.fuse-800010-redhat-00001 (camel-1) started in 170ms (build:0ms
init:150ms start:20ms)
[1] 2021-08-17 08:35:24,487 INFO [io.quarkus] (main) camel-k-integration 1.6.0 on JVM
(powered by Quarkus 1.11.7.Final-redhat-00009) started in 2.192s.
[1] 2021-08-17 08:35:24,488 INFO [io.quarkus] (main) Profile prod activated.
[1] 2021-08-17 08:35:24,488 INFO [io.quarkus] (main) Installed features: [camel-bean,
camel-core, camel-java-joor-dsl, camel-k-core, camel-k-runtime, camel-log, camel-support-
common, camel-timer, cdi]
[1] 2021-08-17 08:35:25,493 INFO [info] (Camel (camel-1) thread #0 - timer://java)
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from java]
[1] 2021-08-17 08:35:26,479 INFO [info] (Camel (camel-1) thread #0 - timer://java)
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from java]
...

```

2. **Ctrl-C** キーを押して、ターミナルでログインを終了します。

関連情報

- ロギングフレームワークの詳細は、「[Configuring logging format](#)」を参照してください。
- ログを表示する開発ツールの詳細は、[Red Hat による Apache Camel K の VS Code ツール](#) を参照してください。

1.3. CAMELK 統合のスケーリング

ocscale コマンドを使用して統合をスケーリングできます。

手順

- Camel K 統合をスケーリングするには、次のコマンドを実行します。

```
oc scale it <integration_name> --replicas <number_of_replicas>
```

- 統合リソースを直接編集して、統合をスケーリングすることもできます。

```
oc patch it <integration_name> --type merge -p '{"spec":{"replicas":<number_of_replicas>}}'
```

統合のレプリカの数を表示するには、次のコマンドを使用します。

```
oc get it <integration_name> -o jsonpath='{.status.replicas}'
```

第2章 CAMEL K インテグレーションのモニタリング

Red Hat Integration - Camel K モニタリングは [OpenShift モニタリングシステム](#) に基づいています。本章では、実行時に Red Hat Integration - Camel K インテグレーションを監視するためにオプションを使用する方法について説明します。OpenShift Monitoring の一部としてすでにデプロイされている Prometheus Operator を使用して、独自のアプリケーションを監視することができます。

- [「OpenShift でのユーザーワークロードモニタリングの有効化」](#)
- [「Camel K インテグレーションメトリクスの設定」](#)
- [「カスタム Camel K インテグレーションメトリクスの追加」](#)

2.1. OPENSIFT でのユーザーワークロードモニタリングの有効化

OpenShift 4.3 以降には、OpenShift Monitoring の一部としてすでにデプロイされている組み込みの Prometheus Operator が含まれています。ここでは、OpenShift Monitoring で独自のアプリケーションサービスのモニタリングを有効にする方法について説明します。このオプションを使用すると、個別の Prometheus インスタンスをインストールおよび管理するための追加のオーバーヘッドが発生しません。

前提条件

- Camel K Operator がインストールされている OpenShift クラスターにクラスター管理者としてアクセスする必要があります。「[Installing Camel K](#)」を参照してください。

手順

1. 以下のコマンドを実行して、**cluster-monitoring-config** ConfigMap オブジェクトが **openshift-monitoring project** に存在するかを確認します。

```
$ oc -n openshift-monitoring get configmap cluster-monitoring-config
```

2. **cluster-monitoring-config** ConfigMap がない場合は作成します。

```
$ oc -n openshift-monitoring create configmap cluster-monitoring-config
```

3. **cluster-monitoring-config** ConfigMap を編集します。

```
$ oc -n openshift-monitoring edit configmap cluster-monitoring-config
```

4. **data:config.yaml:** で、**enableUserWorkload** を **true** に設定します。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |
    enableUserWorkload: true
```

関連情報

- [ユーザー定義プロジェクトのモニタリングの有効化](#)

2.2. CAMEL K インテグレーションメトリクスの設定

実行時に Camel K Prometheus トレイトを使用すると、Camel K インテグレーションのモニタリングを自動的に設定できます。これにより、依存関係およびインテグレーション Pod の設定が自動化され、Prometheus によって検出および表示されるメトリクスエンドポイントが公開されます。[Camel Quarkus MicroProfile Metrics エクステンション](#) は、[OpenMetrics](#) 形式でデフォルトの Camel K メトリクスを自動的に収集および公開します。

前提条件

- OpenShift で、独自のサービスのモニタリングが有効になっている必要があります。「[Enabling user workload monitoring in OpenShift](#)」を参照してください。

手順

1. 以下のコマンドを入力して、Prometheus トレイトを有効にして Camel K インテグレーションを実行します。

```
kamel run myIntegration.java -t prometheus.enabled=true
```

または、以下のようにインテグレーションプラットフォームを更新すると、Prometheus トレイトを1度グローバルに有効にすることができます。

```
$ oc patch ip camel-k --type=merge -p '{"spec":{"traits":{"prometheus":{"configuration":{"enabled":true}}}}}'
```

2. Prometheus で Camel K インテグレーションメトリクスのモニタリングを確認します。たとえば、埋め込み Prometheus の場合は、OpenShift 管理者または開発者 Web コンソールで **Monitoring > Metrics** と選択します。
3. 表示する Camel K メトリクスを入力します。たとえば、Administrator コンソールの **Insert Metric at Cursor** で **application_camel_context_uptime_seconds** を入力し、**Run Queries** をクリックします。
4. **Add Query** をクリックして追加のメトリクスを表示します。

関連情報

- [Prometheus トレイト](#)
- [Camel Quarkus MicroProfile Metrics](#)

2.3. カスタム CAMEL K インテグレーションメトリクスの追加

Java コードで Camel MicroProfile Metrics コンポーネントおよびアノテーションを使用すると、カスタムメトリクスを Camel K インテグレーションに追加できます。その後、これらのカスタムメトリクスは Prometheus によって自動的に検出され、表示されます。

ここでは、Camel MicroProfile Metrics アノテーションを Camel K インテグレーションおよびサービス実装コードに追加する例を紹介します。

前提条件

- OpenShift で、独自のサービスのモニタリングが有効になっている必要があります。「[Enabling user workload monitoring in OpenShift](#)」を参照してください。

手順

1. Camel MicroProfile Metrics コンポーネントアノテーションを使用して、カスタムメトリクスを Camel インテグレーションコードに登録します。以下の例は、**Metrics.java** インテグレーションを示しています。

```
// camel-k: language=java trait=prometheus.enabled=true dependency=mvn:org.my/app:1.0
```

1

```
import org.apache.camel.Exchange;
import org.apache.camel.LoggingLevel;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.microprofile.metrics.MicroProfileMetricsConstants;
```

```
import javax.enterprise.context.ApplicationScoped;
```

```
@ApplicationScoped
```

```
public class Metrics extends RouteBuilder {
```

```
    @Override
```

```
    public void configure() {
```

```
        onException()
```

```
            .handled(true)
```

```
            .maximumRedeliveries(2)
```

```
            .logStackTrace(false)
```

```
            .logExhausted(false)
```

```
            .log(LoggingLevel.ERROR, "Failed processing ${body}")
```

```
            // Register the 'redelivery' meter
```

```
            .to("microprofile-metrics:meter:redelivery?mark=2")
```

```
            // Register the 'error' meter
```

```
            .to("microprofile-metrics:meter:error"); 2
```

```
        from("timer:stream?period=1000")
```

```
            .routeId("unreliable-service")
```

```
            .setBody(header(Exchange.TIMER_COUNTER).prepend("event #"))
```

```
            .log("Processing ${body}...")
```

```
            // Register the 'generated' meter
```

```
            .to("microprofile-metrics:meter:generated") 3
```

```
            // Register the 'attempt' meter via @Metered in Service.java
```

```
            .bean("service") 4
```

```
            .filter(header(Exchange.REDELIVERED))
```

```
                .log(LoggingLevel.WARN, "Processed ${body} after
```

```
                ${header.CamelRedeliveryCounter} retries")
```

```
                .setHeader(MicroProfileMetricsConstants.HEADER_METER_MARK,
```

```
                header(Exchange.REDELIVERY_COUNTER))
```

```
                // Register the 'redelivery' meter
```

```
                .to("microprofile-metrics:meter:redelivery") 5
```

```
            .end()
```

```
            .log("Successfully processed ${body}")
```

```
            // Register the 'success' meter
```

```

        .to("microprofile-metrics:meter:success"); ❸
    }
}

```

- ❶ Camel K モードラインを使用して、Prometheus トレイトと Maven 依存関係を自動的に設定します。
- ❷ **error**: 処理されていないイベントの数に対応するエラー数のメトリック。
- ❸ **generated**: 処理されるイベント数のメトリック。
- ❹ **attempt**: 受信イベントを処理するためにサービス Bean に対して行われる呼び出し数のメトリック。
- ❺ **redelivery**: イベントを処理するために行われた再試行回数のメトリック。
- ❻ **success**: 正常に処理されたイベント数のメトリクス。

2. 必要に応じて Camel MicroProfile Metrics アノテーションを実装ファイルに追加します。以下の例は、ランダムに失敗を生成する Camel K インテグレーションによって呼び出される **service** Bean を示しています。

```

package com.redhat.integration;

import java.util.Random;

import org.apache.camel.Exchange;
import org.apache.camel.RuntimeExchangeException;

import org.eclipse.microprofile.metrics.Meter;
import org.eclipse.microprofile.metrics.annotation.Metered;
import org.eclipse.microprofile.metrics.annotation.Metric;

import javax.inject.Named;
import javax.enterprise.context.ApplicationScoped;

@Named("service")
@ApplicationScoped
@io.quarkus.arc.Unremovable

public class Service {

    //Register the attempt meter
    @Metered(absolute = true)
    public void attempt(Exchange exchange) { ❶
        Random rand = new Random();
        if (rand.nextDouble() < 0.5) {
            throw new RuntimeExchangeException("Random failure", exchange); ❷
        }
    }
}

```

- ❶ **@Metered** MicroProfile Metrics アノテーションは meter を宣言し、名前はメトリクスメソッド名 (この場合は **attempt**) に基づいて自動的に生成されます。

- 2 この例は、メトリクスのエラーを生成するために無作為に失敗します。
3. 「[Configuring Camel K integration metrics](#)」の手順に従い、インテグレーションを実行し、Prometheus でカスタム Camel K メトリクスを確認します。
この例では、**Metrics.java** ですでに Camel K モードラインが使用され、Prometheus と **Service.java** に必要な Maven 依存関係が自動的に設定されます。

関連情報

- [Camel MicroProfile Metrics component](#)
- [Camel Quarkus MicroProfile Metrics Extension](#)

第3章 CAMEL K OPERATOR のモニタリング

Red Hat Integration - Camel K モニタリングは [OpenShift モニタリングシステム](#) に基づいています。本章では、実行時に Red Hat Integration - Camel K Operator を監視するためにオプションを使用する方法について説明します。OpenShift Monitoring の一部としてすでにデプロイされている Prometheus Operator を使用して、独自のアプリケーションを監視することができます。

- [「Camel K Operator メトリクス」](#)
- [「Camel K Operator モニタリングの有効化」](#)
- [「Camel K Operator のアラート」](#)

3.1. CAMEL K OPERATOR メトリクス

Camel K Operator モニタリングエンドポイントは、以下のメトリクスを公開します。

表3.1 Camel K Operator メトリクス

名前	タイプ	説明	バケット	ラベル
<code>camel_k_reconciliation_duration_seconds</code>	HistogramVec	調整要求の期間	0.25 s、0.5 s、1 s、5 s	<code>namespace,</code> <code>group, version,</code> <code>kind, result:</code> <code>Reconciled Errored Requeued,</code> <code>tag:</code> <code>"" PlatformError UserError</code>
<code>camel_k_build_duration_seconds</code>	HistogramVec	ビルド期間	30 s、1 m、1.5 m、2 m、5 m、10 m	<code>result:</code> <code>Succeeded Error</code>
<code>camel_k_build_recovery_attempts</code>	ヒストグラム	ビルドリカバリーの試行	0、1、2、3、4、5	<code>result:</code> <code>Succeeded Error</code>
<code>camel_k_build_queue_duration_seconds</code>	ヒストグラム	ビルドキューの期間	5 s、15 s、30 s、1 m、5 m、	該当なし
<code>camel_k_integration_first_readiness_seconds</code>	ヒストグラム	最初のインテグレーションの準備ができるまでの時間	5 s、10 s、30 s、1 m、2 m	該当なし

3.2. CAMEL K OPERATOR モニタリングの有効化

OpenShift 4.3 以降には、OpenShift Monitoring の一部としてすでにデプロイされている組み込みの Prometheus Operator が含まれています。ここでは、OpenShift Monitoring で独自のアプリケーションサービスのモニタリングを有効にする方法について説明します。

前提条件

- Camel K Operator がインストールされている OpenShift クラスターにクラスター管理者としてアクセスする必要があります。「[Installing Camel K](#)」を参照してください。
- OpenShift で、独自のサービスのモニタリングが有効になっている必要があります。「[Enabling user workload monitoring in OpenShift](#)」を参照してください。

手順

1. Operator メトリクスエンドポイントをターゲットにする **PodMonitor** リソースを作成し、Prometheus サーバーが Operator によって公開されるメトリクスを収集できるようにします。

operator-pod-monitor.yaml

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: camel-k-operator
  labels:
    app: "camel-k"
    camel.apache.org/component: operator
spec:
  selector:
    matchLabels:
      app: "camel-k"
      camel.apache.org/component: operator
  podMetricsEndpoints:
    - port: metrics
```

2. **PodMonitor** リソースを作成します。

```
oc apply -f operator-pod-monitor.yaml
```

関連情報

- 検索メカニズムおよび Operator リソース間の関係についての詳細は、「[Prometheus Operator getting started guide](#)」を参照してください。
- Operator メトリクスが検出されない場合は、「[Troubleshooting ServiceMonitor changes](#)」を参照してください。これは、**PodMonitor** リソースのトラブルシューティングにも適用されます。

3.3. CAMEL K OPERATOR のアラート

OpenShift モニタリングスタックからの AlertManager インスタンスが Camel K Operator によって公開されるメトリクスに基づいてアラートをトリガーできるように、**PrometheusRule** リソースを作成することができます。

例

以下のように、公開されたメトリクスに基づいてアラートルールで **PrometheusRule** リソースを作成できます。

```
apiVersion: monitoring.coreos.com/v1
```

```

kind: PrometheusRule
metadata:
  name: camel-k-operator
spec:
  groups:
    - name: camel-k-operator
      rules:
        - alert: CamelKReconciliationDuration
          expr: |
            (
              1 - sum(rate(camel_k_reconciliation_duration_seconds_bucket{le="0.5"}[5m])) by (job)
            /
              sum(rate(camel_k_reconciliation_duration_seconds_count[5m])) by (job)
            )
            * 100
            > 10
          for: 1m
          labels:
            severity: warning
          annotations:
            message: |
              {{ printf "%0.0f" $value }}% of the reconciliation requests
              for {{ $labels.job }} have their duration above 0.5s.
        - alert: CamelKReconciliationFailure
          expr: |
            sum(rate(camel_k_reconciliation_duration_seconds_count{result="Errored"}[5m])) by (job)
            /
            sum(rate(camel_k_reconciliation_duration_seconds_count[5m])) by (job)
            * 100
            > 1
          for: 10m
          labels:
            severity: warning
          annotations:
            message: |
              {{ printf "%0.0f" $value }}% of the reconciliation requests
              for {{ $labels.job }} have failed.
        - alert: CamelKSuccessBuildDuration2m
          expr: |
            (
              1 - sum(rate(camel_k_build_duration_seconds_bucket{le="120",result="Succeeded"}[5m])) by
(job)
            /
              sum(rate(camel_k_build_duration_seconds_count{result="Succeeded"}[5m])) by (job)
            )
            * 100
            > 10
          for: 1m
          labels:
            severity: warning
          annotations:
            message: |
              {{ printf "%0.0f" $value }}% of the successful builds
              for {{ $labels.job }} have their duration above 2m.
        - alert: CamelKSuccessBuildDuration5m
          expr: |

```

```

(
  1 - sum(rate(camel_k_build_duration_seconds_bucket{le="300",result="Succeeded"}[5m])) by
(job)
/
sum(rate(camel_k_build_duration_seconds_count{result="Succeeded"}[5m])) by (job)
)
* 100
> 1
for: 1m
labels:
  severity: critical
annotations:
  message: |
    {{ printf "%0.0f" $value }}% of the successful builds
    for {{ $labels.job }} have their duration above 5m.
- alert: CamelKBuildFailure
expr: |
  sum(rate(camel_k_build_duration_seconds_count{result="Failed"}[5m])) by (job)
/
sum(rate(camel_k_build_duration_seconds_count[5m])) by (job)
* 100
> 1
for: 10m
labels:
  severity: warning
annotations:
  message: |
    {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }} have failed.
- alert: CamelKBuildError
expr: |
  sum(rate(camel_k_build_duration_seconds_count{result="Error"}[5m])) by (job)
/
sum(rate(camel_k_build_duration_seconds_count[5m])) by (job)
* 100
> 1
for: 10m
labels:
  severity: critical
annotations:
  message: |
    {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }} have errored.
- alert: CamelKBuildQueueDuration1m
expr: |
(
  1 - sum(rate(camel_k_build_queue_duration_seconds_bucket{le="60"}[5m])) by (job)
/
sum(rate(camel_k_build_queue_duration_seconds_count[5m])) by (job)
)
* 100
> 1
for: 1m
labels:
  severity: warning
annotations:
  message: |
    {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }}

```

```

        have been queued for more than 1m.
- alert: CamelKBuildQueueDuration5m
  expr: |
    (
      1 - sum(rate(camel_k_build_queue_duration_seconds_bucket{le="300"}[5m])) by (job)
    /
      sum(rate(camel_k_build_queue_duration_seconds_count[5m])) by (job)
    )
    * 100
    > 1
  for: 1m
  labels:
    severity: critical
  annotations:
    message: |
      {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }}
      have been queued for more than 5m.

```

Camel K Operator のアラート

以下の表は、**PrometheusRule** リソースで定義されるアラートルールを示しています。

名前	重大度	詳細
CamelKReconciliationDuration	warning	全体の 10% を超える調整要求で、最低でも 1 分の間に期間が 0.5 秒を超える。
CamelKReconciliationFailure	warning	全体の 1% を超える調整要求が、最低でも 10 分間失敗している。
CamelKSuccessBuildDuration2m	warning	全体の 10% を超える正常なビルドで、最低でも 1 分の間に期間が 2 分を超える。
CamelKSuccessBuildDuration5m	critical	全体の 1% を超える正常なビルドで、最低でも 1 分の間に期間が 5 分を超える。
CamelKBuildError	critical	全体の 1% を超えるビルドで、最低でも 10 分間エラーが発生している。
CamelKBuildQueueDuration1m	warning	全体の 1% を超えるビルドが、最低でも 1 分間 1 分を超えてキューに入れられている。
CamelKBuildQueueDuration5m	critical	全体の 1% を超えるビルドが、最低でも 1 分間 5 分を超えてキューに入れられている。

アラートについての詳細は、OpenShift ドキュメントの「[アラートルールの作成](#)」を参照してください。

第4章 CAMEL K インテグレーションの設定

Camel K インテグレーションのライフサイクルには、以下の 2 つの設定フェーズがあります。

- **ビルド時**: Camel Quarkus が Camel K インテグレーションをビルドする場合、ビルド時プロパティが使用されます。
- **ランタイム**: Camel K インテグレーションが実行されると、インテグレーションはローカルファイル、OpenShift ConfigMap、または Secret からのランタイムプロパティまたは設定情報を使用します。

kamel run コマンドで以下のオプションを使用して設定情報を指定します。

- ビルド時の設定の場合は、「[Specifying build-time configuration properties](#)」で説明されているように **--build-property** オプションを使用します。
- ランタイム設定の場合は、「[Specifying runtime configuration options](#)」で説明されているように、**--property**、**--config**、または **--resource** オプションを使用します。

たとえば、ビルド時とランタイムオプションを使用して、[Connect Camel K with databases](#) のサンプル設定のように、Camel K のデータソースを迅速に設定できます。

- [「ビルド時の設定プロパティの指定」](#)
- [「ランタイム設定オプションの指定」](#)
- [「Camel インテグレーションコンポーネントの設定」](#)
- [「Camel K インテグレーション依存関係の設定」](#)

4.1. ビルド時の設定プロパティの指定

Camel K インテグレーションをビルドできるように、プロパティ値を Camel Quarkus ランタイムに提供する必要がある場合があります。ビルド時に有効な Quarkus 設定の詳細は、[Quarkus Build Time 設定についてのドキュメント](#)を参照してください。ビルド時のプロパティはコマンドラインで直接指定するか、プロパティファイルを参照して指定できます。プロパティが両方の場所に定義されている場合は、コマンドラインで直接指定された値は、プロパティファイルの値よりも優先されます。

前提条件

- Camel K Operator および OpenShift Serverless Operator がインストールされている OpenShift クラスターにアクセスできる必要があります。
- [Camel K のインストール](#)
- [OperatorHub からの OpenShift Serverless のインストール](#)
- Camel K インテグレーションに適用する Camel Quarkus 設定オプションを把握している必要があります。

手順

- Camel K **kamel run** コマンドで **--build property** オプションを指定するには、以下を実行します。

```
kamel run --build-property <quarkus-property>=<property-value> <camel-k-integration>
```

たとえば、以下の Camel K インテグレーション (**my-simple-timer.yaml** という名前の) は **quarkus.application.name** 設定オプションを使用します。

```
- from:
  uri: "timer:tick"
  steps:
    - set-body:
      constant: "{{quarkus.application.name}}"
    - to: "log:info"
```

デフォルトのアプリケーション名を上書きするには、インテグレーションの実行時に **quarkus.application.name** プロパティの値を指定します。

たとえば、名前を **my-simple-timer** から **my-favorite-app** に変更するには、次のコマンドを実行します。

```
kamel run --build-property quarkus.application.name=my-favorite-app my-simple-timer.yaml
```

- 複数の build-time プロパティを指定するには **--build-property** オプションを **kamel run** コマンドに追加します。

```
kamel run --build-property <quarkus-property1>=<property-value1> -build-property=
<quarkus-property2>=<property-value12> <camel-k-integration>
```

また、複数のプロパティを指定する必要がある場合は、プロパティファイルを作成して **--build-property file** オプションでプロパティファイルを指定することもできます。

```
kamel run --build-property file:<property-filename> <camel-k-integration>
```

たとえば、以下のプロパティファイル (名前: **quarkus.properties**) は 2 つの Quarkus プロパティを定義します。

```
quarkus.application.name = my-favorite-app
quarkus.banner.enabled = true
```

quarkus.banner.enabled プロパティは、インテグレーションの起動時に Quarkus バナーを表示するように指定します。

Camel K **kamel run** コマンドで **quarkus.properties** ファイルを指定するには、以下を実行します。

```
kamel run --build-property file:quarkus.properties my-simple-timer.yaml
```

Quarkus はプロパティファイルを解析し、プロパティ値を使用して Camel K インテグレーションを設定します。

関連情報

Camel K インテグレーションのランタイムとしての Camel Quarkus の詳細は、「[Quarkus Trait](#)」を参照してください。

4.2. ランタイム設定オプションの指定

稼働時に使用する Camel K インテグレーションの以下のランタイム設定情報を指定できます。

- コマンドラインまたは `.properties` ファイルで指定するランタイムプロパティ
- インテグレーションの開始時に Camel K Operator が処理し、ランタイムプロパティとして解析する設定値。設定値は、ローカルのテキストファイル、OpenShift ConfigMap、または OpenShift シークレットで指定できます。
- インテグレーションの起動時にプロパティファイルとして解析されないリソース情報。ローカルのテキストファイル、バイナリーファイル、OpenShift ConfigMap、または OpenShift シークレットでリソース情報を指定できます。

以下の **kamel run** オプションを使用します。

- **--property**
--property オプションを使用して、コマンドラインでランタイムプロパティを直接指定するか、Java `*.properties` ファイルを参照します。Camel K Operator は、稼働中のインテグレーションの **user.properties** ファイルにプロパティファイルの内容を追加します。
- **--config**
--config オプションを使用して、インテグレーションの開始時に Camel K Operator が処理し、ランタイムプロパティとして解析する設定値を指定します。

ローカルのテキストファイル (1 MiB の最大ファイルサイズ)、ConfigMap (3 MB) またはシークレット (3 MB) を指定できます。ファイルは UTF-8 リソースである必要があります。マテリアル化されたファイル (提供するファイルからインテグレーションを起動する時に生成される) は、クラスパスレベルで利用可能になります。これにより、正確な場所を指定しなくてもインテグレーションコードで参照できるようになります。

注記: 非 UTF-8 リソース (バイナリーファイルなど) を提供する場合がある場合は、**--resource** オプションを使用します。

- **--resource**
--resource オプションを使用して、インテグレーションの稼働時にアクセスするリソースを提供します。ローカルのテキストファイルまたはバイナリーファイル (1 MiB の最大ファイルサイズ)、ConfigMap (最大 3 MB) またはシークレット (最大 3 MB) を指定できます。オプションで、リソース用にマテリアル化したファイルの宛先を指定できます。たとえば、HTTPS 接続を設定する場合は、**--resource** オプションを使用して、指定した場所にあると予想される SSL 証明書 (バイナリーファイル) を指定します。

Camel K Operator はプロパティのリソースを解析したり、リソースをクラスパスに追加したりしません。(クラスパスにリソースを追加する場合は、インテグレーションで **JVM トレイト** を使用できます)。

4.2.1. ランタイムプロパティの指定

kamel run コマンドで **--property** オプションを使用して、コマンドラインで直接または Java `*.properties` ファイルを参照してランタイムプロパティを指定できます。

--property オプションを使用してインテグレーションを実行する場合、Camel K Operator は稼働中のインテグレーションの **user.properties** ファイルにプロパティを追加します。

4.2.1.1. コマンドラインでのランタイムプロパティの指定

実行時にコマンドラインで Camel K インテグレーションのプロパティを設定できます。プロパティのプレースホルダーを使用してインテグレーションのプロパティを定義する場合 (例:

`{{my.message}}`）、`--property my.message=Hello` のようにコマンドラインでプロパティ値を指定できます。1つのコマンドで複数のプロパティを指定できます。

前提条件

- [Setting up your Camel K development environment](#)

手順

1. プロパティを使用する Camel インテグレーションを開発します。以下の簡単な例には、`{{my.message}}` プロパティプレースホルダーが含まれています。

```
...
- from:
  uri: "timer:tick"
  steps:
    - set-body:
      constant: "{{my.message}}"
    - to: "log:info"
...
```

2. 以下の構文を使用してインテグレーションを実行し、実行時にプロパティ値を設定します。

```
kamel run --property <property>=<value> <integration>
```

または、(`--property` の代わりに) `--p` 短縮表記を使用することもできます。

```
kamel run --property <property>=<value> <integration>
```

以下に例を示します。

```
kamel run --property my.message="Hola Mundo" HelloCamelK.java --dev
```

または

```
kamel run --p my.message="Hola Mundo" HelloCamelK.java --dev
```

以下は、結果の例になります。

```
...
[1] 2020-04-13 15:39:59.213 INFO [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.RoutesDumper@6e0dec4a executed in phase Started
[1] 2020-04-13 15:40:00.237 INFO [Camel (camel-k) thread #1 - timer://java] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hola Mundo from java]
...
```

関連項目

- [プロパティファイルでのランタイムプロパティの指定](#)

4.2.1.2. プロパティファイルでのランタイムプロパティの指定

実行時にコマンドラインでプロパティファイル `*.properties` を指定すると、Camel K インテグレーション

ションに複数のプロパティを設定できます。たとえば、**--p file my-integration.properties** プロパティプレースホルダーを使用してインテグレーションでプロパティを定義する場合、**{{my.items}}** のようにプロパティファイルを使用してコマンドラインでプロパティ値を指定できます。

前提条件

- [Setting up your Camel K development environment](#)

手順

1. インテグレーションのプロパティファイルを作成します。以下の例は、**my.properties** という名前のファイルからのものです。

```
my.key.1=hello
my.key.2=world
```

2. プロパティファイルに定義されたプロパティを使用する Camel インテグレーションを開発します。以下の **Routing.java** 統合例では **{{my.key.1}}** および **{{my.key.2=world}}** プロパティプレースホルダーを使用します。

```
import org.apache.camel.builder.RouteBuilder;

public class Routing extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("timer:property-file")
            .routeId("property-file")
            .log("property file content is: {{my.key.1}} {{my.key.2}}");

    }
}
```

3. 以下の構文を使用してインテグレーションを実行し、プロパティファイルを参照します。

```
kamel run --property file:<my-file.properties> <integration>
```

または、(**--property** の代わりに) **--p** 短縮表記を使用することもできます。

```
kamel run --p file:<my-file.properties> <integration>
```

以下に例を示します。

```
kamel run Routing.java --property:file=my.properties --dev
```

関連情報

- [基本的な Camel K Java インテグレーションのデプロイ](#)
- [コマンドラインでのランタイムプロパティの指定](#)

4.2.2. 設定値の指定

kamel run コマンドの **--config** オプションを使用して、Camel K Operator がランタイムプロパティーとして処理し、解析する設定値を指定できます。設定値は、ローカルのテキスト (UTF-8) ファイル、OpenShift ConfigMap、または OpenShift シークレットで指定できます。

インテグレーションの実行時に、Camel K Operator は提供されたファイルをマレリアル化し、これをクラスパスに追加します。これにより、正確な場所を指定しなくてもインテグレーションコードの設定値を参照できます。

4.2.2.1. テキストファイルの指定

設定値が含まれる UTF-8 テキストファイルがある場合は、**--config file:/path/to/file** オプションを使用して、実行中のインテグレーションのクラスパスでファイルを (同じファイル名で) 利用可能にすることができます。

前提条件

- [Setting up your Camel K development environment](#)
- 設定値が含まれる 1 つ以上の (バイナリー以外の) テキストファイルがある。
たとえば、以下のテキスト行を含む **resources-data.txt** という名前のファイルを作成します。

```
the file body
```

手順

1. 設定値が含まれるテキストファイルを参照する Camel K インテグレーションを作成します。
たとえば、以下のインテグレーション (**ConfigFileRoute.java**) は、実行時にクラスパスで **resources-data.txt** ファイルが利用可能でなければなりません。

```
import org.apache.camel.builder.RouteBuilder;

public class ConfigFileRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("timer:config-file")
            .setBody()
            .simple("resource:classpath:resources-data.txt")
            .log("resource file content is: ${body}");

    }
}
```

2. インテグレーションを実行し、実行中のインテグレーションで使えるように、**--config** オプションを使用してテキストファイルを指定します。以下に例を示します。

```
kamel run --config file:resources-data.txt ConfigFileRoute.java --dev
```

必要に応じて、**--config** オプションを繰り返し追加して、複数のファイルを指定できます。以下に例を示します。

```
kamel run --config file:resources-data1.txt --config file:resources-data2.txt
ConfigFileRoute.java --dev
```

4.2.2.2. ConfigMap の指定

設定値が含まれる OpenShift ConfigMap があり、Camel K インテグレーションで使えるように ConfigMap をマテリアル化する必要がある場合には、`--config configmap:<configmap-name>` 構文を使用します。

前提条件

- [Setting up your Camel K development environment](#)
- OpenShift クラスター上に1つ以上の ConfigMap ファイルがある。
たとえば、以下のコマンドを使用して ConfigMap を作成できます。

```
oc create configmap my-cm --from-literal=my-configmap-key="configmap content"
```

手順

1. ConfigMap を参照する Camel K インテグレーションを作成します。
たとえば、以下のインテグレーション (名前: **ConfigConfigmapRoute.java**) は、**my-cm** という名前の ConfigMap の **my-configmap-key** という名前の設定値を参照します。

```
import org.apache.camel.builder.RouteBuilder;

public class ConfigConfigmapRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("timer:configmap")
            .setBody()
                .simple("resource:classpath:my-configmap-key")
            .log("configmap content is: ${body}");

    }
}
```

2. インテグレーションを実行し、**--config** オプションを使用して ConfigMap ファイルをマテリアル化し、実行中のインテグレーションで使えるようにします。以下に例を示します。

```
kamel run --config configmap:my-cm ConfigConfigmapRoute.java --dev
```

インテグレーションが起動すると、Camel K Operator は ConfigMap の内容で OpenShift ボリュームをマウントします。

注記: クラスターでまだ利用できない ConfigMap を指定した場合、Integration は待機し ConfigMap が利用可能になって初めて起動します。

4.2.2.3. シークレットの指定

OpenShift シークレットを使用して、設定情報を安全に含めることができます。**--config secret** 構文を使用して、Camel K インテグレーションが利用できるようシークレットをマテリアル化することができます。

前提条件

- [Setting up your Camel K development environment](#)
- OpenShift クラスター上に1つ以上のシークレットがある。
たとえば、以下のコマンドを使用してシークレットを作成できます。

```
oc create secret generic my-sec --from-literal=my-secret-key="very top secret"
```

手順

1. ConfigMap を参照する Camel K インテグレーションを作成します。
たとえば、以下のインテグレーション (名前: **ConfigSecretRoute.java**) は、**my-sec** という名前のシークレットの **my-secret** プロパティを参照します。

```
import org.apache.camel.builder.RouteBuilder;

public class ConfigSecretRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("timer:secret")
            .setBody()
            .simple("resource:classpath:my-secret")
            .log("secret content is: ${body}");

    }
}
```

2. インテグレーションを実行し、**--config** オプションを使用してシークレットをマテリアル化し、実行中のインテグレーションで使用できるようにします。以下に例を示します。

```
kamel run --config secret:my-sec ConfigSecretRoute.java --dev
```

インテグレーションが起動すると、Camel K Operator はシークレットの内容で OpenShift ボリュームをマウントします。

4.2.2.4. ConfigMap またはシークレットに含まれるプロパティの参照

インテグレーションを実行し、**--config** オプションで ConfigMap またはシークレットを指定する場合、Camel K Operator は ConfigMap またはシークレットをランタイムプロパティファイルとして解析します。インテグレーション内で、他のランタイムプロパティを参照する際にプロパティを参照できます。

前提条件

- [Setting up your Camel K development environment](#)

手順

1. プロパティが含まれるテキストファイルを作成します。
たとえば、以下のプロパティを含む **my.properties** という名前のファイルを作成します。

```
my.key.1=hello
my.key.2=world
```

2. プロパティファイルに基づいて ConfigMap またはシークレットを作成します。
たとえば、以下のコマンドを使用して my.properties ファイルからシークレットを作成します。

```
oc create secret generic my-sec --from-file my.properties
```

3. インテグレーションで、シークレットに定義されたプロパティを参照します。
たとえば、以下のインテグレーション (名前: **ConfigSecretRoute.java**) は、**my.key.1** と **my.key.2** プロパティを参照します。

```
import org.apache.camel.builder.RouteBuilder;

public class ConfigSecretPropertyRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("timer:secret")
            .routeId("secret")
            .log("{}my.key.1{} {}my.key.2{}");

    }
}
```

4. インテグレーションを実行し、**--config** オプションを使用して、**my.key.1** および **my.key.2** プロパティが含まれるシークレットを指定します。
以下に例を示します。

```
kamel run --config secret:my-sec ConfigSecretPropertyRoute.java --dev
```

4.2.2.5. ConfigMap またはシークレットから取得した設定値の絞り込み

ConfigMap およびシークレットは複数のソースを保持できます。たとえば、以下のコマンドは2つのソースからシークレット (**my-sec-multi**) を作成します。

```
oc create secret generic my-sec-multi --from-literal=my-secret-key="very top secret" --from-literal=my-secret-key-2="even more secret"
```

--config configmap または **--config secret** オプションの後に **/key** 表記を使用して、インテグレーションが取得する情報量を1つのソースに制限することができます。

前提条件

- [Setting up your Camel K development environment](#)
- 複数のソースを保持する ConfigMap またはシークレットがある。

手順

1. ConfigMap またはシークレットのいずれかのソースだけからの設定値を使用するインテグレーションを作成します。
たとえば、以下のインテグレーション (**ConfigSecretKeyRoute.java**) は、**my-sec-multi** シークレット内のいずれかのソースだけからのプロパティを使用します。

```
import org.apache.camel.builder.RouteBuilder;
```

```
public class ConfigSecretKeyRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("timer:secret")
            .setBody()
                .simple("resource:classpath:my-secret-key-2")
            .log("secret content is: ${body}");
    }
}
```

2. **--config secret** オプションと **/key** 表記を使用して、インテグレーションを実行します。以下に例を示します。

```
kamel run --config secret:my-sec-multi/my-secret-key-2 ConfigSecretKeyRoute.java --dev
```

3. インテグレーション Pod を確認し、指定されたソース（例: **my-secret-key-2**）のみがマウントされていることを確認します。
たとえば、以下のコマンドを実行して Pod のすべてのボリュームを一覧表示します。

```
oc set volume pod/<pod-name> --all
```

4.2.3. 実行中のインテグレーションへのリソースの提供

kamel run コマンドの **--resource** オプションを使用すると、実行しているときに使用するインテグレーションのリソースを指定できます。ローカルのテキストファイル (1 MiB の最大ファイルサイズ)、ConfigMap (3 MB) またはシークレット (3 MB) を指定できます。オプションで、リソース用にマテリアル化したファイルの宛先を指定できます。たとえば、HTTPS 接続を設定する場合は、**--resource** オプションを使用します。これは、既知の場所にあると予想されるバイナリーファイルである SSL 証明書を提供する必要があるためです。

--resource オプションを使用する場合、Camel K Operator はランタイムプロパティを検索するリソースを解析せず、リソースをクラスパスに追加しません。（クラスパスにリソースを追加する場合は、[JVMトレイト](#)を使用できます。

4.2.3.1. リソースとしてのテキストまたはバイナリーファイルの指定

設定値が含まれるテキストファイルまたはバイナリーファイルがある場合は、**--resource file:/path/to/file** オプションを使用してファイルをマテリアル化できます。デフォルトでは、Camel K Operator はマテリアル化されたファイルを **/etc/camel/resources/** ディレクトリーにコピーします。オプションで、「[リソースの宛先パスの指定](#)」で説明されているように、別の宛先ディレクトリーを指定できます。

前提条件

- [Setting up your Camel K development environment](#)
- 設定プロパティが含まれる1つ以上のテキストまたはバイナリーファイルがある。

手順

1. 指定するファイルの内容を読み取る Camel K インテグレーションを作成します。

たとえば、以下の統合(**ResourceFileBinaryRoute.java**)を展開して **resources-data.zip** ファイルを読み込みます。

```
import org.apache.camel.builder.RouteBuilder;

public class ResourceFileBinaryRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("file:/etc/camel/resources/?fileName=resources-
data.zip&noop=true&idempotent=false")
            .unmarshal().zipFile()
            .log("resource file unzipped content is: ${body}");

    }
}
```

2. インテグレーションを実行し、**--resource** オプションを使用してファイルをデフォルトの宛先ディレクトリー (**/etc/camel/resources/**) にコピーします。以下に例を示します。

```
kamel run --resource file:resources-data.zip ResourceFileBinaryRoute.java -d camel-zipfile --dev
```

注記: バイナリーファイルを指定すると、ファイルの内容のバイナリー表現が作成され、インテグレーションで透過的にデコードされます。

必要に応じて、**--resource** オプションを繰り返し追加して、複数のリソースを指定できます。以下に例を示します。

```
kamel run --resource file:resources-data1.txt --resource file:resources-data2.txt
ResourceFileBinaryRoute.java -d camel-zipfile --dev
```

4.2.3.2. リソースとしての ConfigMap の指定

設定値が含まれる OpenShift ConfigMap があり、ConfigMap をインテグレーションのリソースとしてマテリアル化する必要がある場合は、**--resource <configmap-file>** オプションを使用します。

前提条件

- [Setting up your Camel K development environment](#)
- OpenShift クラスター上に1つ以上の ConfigMap ファイルがある。たとえば、以下のコマンドを使用して ConfigMap を作成できます。

```
oc create configmap my-cm --from-literal=my-configmap-key="configmap content"
```

手順

1. OpenShift クラスターに保存されている ConfigMap を参照する Camel K インテグレーションを作成します。
たとえば、以下のインテグレーション (**Resource ConfigmapRoute.java**) は **my-configmap-key** が含まれる **my-cm** という名前の ConfigMap を参照します。

```
import org.apache.camel.builder.RouteBuilder;

public class ResourceConfigmapRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("file:/etc/camel/resources/my-cm/?fileName=my-configmap-
key&noop=true&idempotent=false")
            .log("resource file content is: ${body}");

    }
}
```

2. インテグレーションを実行し、**--resource** オプションを使用してデフォルトの **/etc/camel/resources/** ディレクトリーの ConfigMap ファイルをマテリアル化し、実行中のインテグレーションで使用できるようにします。
以下に例を示します。

```
kamel run --resource configmap:my-cm ResourceConfigmapRoute.java --dev
```

インテグレーションが起動すると、Camel K Operator は ConfigMap の内容でボリュームをマウントします (例: **my-configmap-key**)。

注記: クラスターでまだ利用できない ConfigMap を指定した場合、Integration は待機し ConfigMap が利用可能になって初めて起動します。

4.2.3.3. リソースとしてのシークレットの指定

設定情報が含まれる OpenShift シークレットがあり、1つ以上のインテグレーションが利用できるリソースとしてマテリアル化する必要がある場合は、**--resource <secret>** 構文を使用します。

前提条件

- [Setting up your Camel K development environment](#)
- OpenShift クラスター上に1つ以上のシークレットファイルがある。たとえば、以下のコマンドを使用してシークレットを作成できます。

```
oc create secret generic my-sec --from-literal=my-secret-key="very top secret"
```

手順

1. OpenShift クラスターに保存されているシークレットを参照する Camel K インテグレーションを作成します。
たとえば、以下のインテグレーション (名前: **ResourceSecretRoute.java**) は **my-sec** シークレットを参照します。

```
import org.apache.camel.builder.RouteBuilder;

public class ResourceSecretRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("file:/etc/camel/resources/my-sec/?fileName=my-secret-
```

```
key&noop=true&idempotent=false")
    .log("resource file content is: ${body}");

}
}
```

2. インテグレーションを実行し、**--resource** オプションを使用してデフォルトの **/etc/camel/resources/** ディレクトリーの Secret ファイルをマテリアル化し、実行中のインテグレーションで使用できるようにします。
以下に例を示します。

```
kamel run --resource secret:my-sec ResourceSecretRoute.java --dev
```

インテグレーションが起動すると、Camel K Operator はシークレットの内容でボリュームをマウントします (例: **my-sec**)。

注記: クラスターでまだ利用できないシークレットを指定した場合、Integration は待機しシークレットが利用可能になって初めて起動します。

4.2.3.4. リソースの宛先パスの指定

/etc/camel/resources/ ディレクトリーは、**--resource** オプションで指定するリソースをマウントするためのデフォルトの場所です。リソースをマウントする別のディレクトリーを指定する必要がある場合は、**--resource @path** 構文を使用します。

前提条件

- [Setting up your Camel K development environment](#)
- 1つ以上の設定プロパティーが含まれるファイル、ConfigMap、またはシークレットがある。

手順

1. 設定プロパティーが含まれるファイル、ConfigMap、またはシークレットを参照する Camel K インテグレーションを作成します。たとえば、以下のインテグレーション (名前: **ResourceFileLocationRoute.java**) は **myprops** ファイルを参照します。

```
import org.apache.camel.builder.RouteBuilder;

public class ResourceFileLocationRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("file:/tmp/?fileName=input.txt&noop=true&idempotent=false")
            .log("resource file content is: ${body}");

    }
}
```

2. インテグレーションを実行し、**@path** 構文で **--resource** オプションを使用して、リソースの内容 (ファイル、ConfigMap、または Secret のいずれか) をマウントする場所を指定します。たとえば、以下のコマンドは、**/tmp** ディレクトリーを使用して **input.txt** ファイルをマウントするように指定します。

```
kamel run --resource file:resources-data.txt@/tmp/input.txt ResourceFileLocationRoute.java -dev
```

3. インテグレーションの Pod で、ファイル (例: **input.txt**) が正しい場所 (例: **tmp** ディレクトリー内) にマウントされていることを確認します。たとえば、以下のコマンドを実行します。

```
oc exec <pod-name> -- cat /tmp/input.txt
```

4.2.3.5. ConfigMap またはシークレットデータの絞り込み

ConfigMap またはシークレットの作成時に、複数の情報ソースを指定できます。たとえば、以下のコマンドは、2つのソースから ConfigMap (**my-cm-multi** という名前) を作成します。

```
oc create configmap my-cm-multi --from-literal=my-configmap-key="configmap content" --from-literal=my-configmap-key-2="another content"
```

--resource オプションでインテグレーションを実行すると、ConfigMap またはシークレットは複数のソースで作成され、デフォルトでは両方のソースがマテリアル化されます。

ConfigMap またはシークレットからリカバリーする情報量を制限する場合は、ConfigMap またはシークレット名の後に **--resource** オプションの **/key** 表記を指定できます。例: **--resource configmap:my-cm/my-key** または **--resource secret:my-secret/my-key**

--resource configmap または **--resource secret** オプションの後に **/key** 表記を使用して、インテグレーションが取得する情報量を1つのソースに制限することができます。

前提条件

- [Setting up your Camel K development environment](#)
- 複数のソースからの値を保持する ConfigMap またはシークレットがある。

手順

1. ConfigMap またはシークレットのいずれかのリソースだけからの設定値を使用するインテグレーションを作成します。たとえば、以下のインテグレーション (**ResourceConfigmapKeyLocationRoute.java** という名称) は **my-cm-multi** ConfigMap を参照します。

```
import org.apache.camel.builder.RouteBuilder;

public class ResourceConfigmapKeyLocationRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {

        from("file:/tmp/app/data/?fileName=my-configmap-key-2&noop=true&idempotent=false")
            .log("resource file content is: ${body} consumed from ${header.CamelFileName}");

    }
}
```

2. インテグレーションを実行し、**@path** 構文で **--resource** オプションを使用して、リソースの内容 (ファイル、ConfigMap、または Secret のいずれか) をマウントする場所を指定します。

たとえば、以下のコマンドは ConfigMap 内に含まれるソースのいずれか (**my-configmap-key-2@**) のみを使用し、**/tmp/app/data** ディレクトリーを使用してマウントすることを指定します。

```
kamel run --resource configmap:my-cm-multi/my-configmap-key-2@/tmp/app/data
ResourceConfigmapKeyLocationRoute.java --dev
```

3. インテグレーションの Pod で、1つのファイル (例: **my-configmap-key-2**) だけが正しい場所 (例: **/tmp/app/data**) にマウントされていることを確認します。たとえば、以下のコマンドを実行します。

```
oc exec <pod-name> -- cat /tmp/app/data/my-configmap-key-2
```

4.3. CAMEL インテグレーションコンポーネントの設定

Camel コンポーネントは、インテグレーションコードにプログラミングを使用して設定でき、実行時にコマンドラインで設定プロパティを使用して設定することもできます。以下の構文を使用して Camel コンポーネントを設定できます。

```
camel.component.${scheme}.${property}=${value}
```

たとえば、段階的なイベント駆動型アーキテクチャーの Camel **seda** コンポーネントのキューサイズを変更するには、コマンドラインで以下のプロパティを設定します。

```
camel.component.seda.queueSize=10
```

前提条件

- [Setting up your Camel K development environment](#)

手順

- **kamel run** コマンドを入力し、**--property** オプションを使用して Camel コンポーネント設定を指定します。以下に例を示します。

```
kamel run --property camel.component.seda.queueSize=10 examples/Integration.java
```

関連情報

- [コマンドラインでのランタイムプロパティの指定](#)
- [Apache Camel SEDA component](#)

4.4. CAMEL K インテグレーション依存関係の設定

Camel K は、インテグレーションコードの実行に必要なさまざまな依存関係を自動的に解決します。ただし、**kamel run --dependency** オプションを使用すると、実行時にコマンドラインに依存関係を明示的に追加できます。

以下のインテグレーションの例では Camel K の依存関係の自動解決が使用されます。

```
...
```



```
from("imap://admin@myserver.com")  
  .to("seda:output")  
...
```

このインテグレーションには **imap:** プレフィックスで始まるエンドポイントがあるため、Camel K は自動的に **camel-mail** コンポーネントに必要な依存関係のリストに追加できます。**seda:** エンドポイントは、すべてのインテグレーションに自動的に追加される **camel-core** に属しているため、Camel K はこのコンポーネントにその他の依存関係を追加しません。

Camel K 依存関係の自動解決は、実行時にユーザーに対して透過的です。これは、開発ループを終了せずに必要なすべてのコンポーネントを素早く追加できるため、開発モードで非常に便利です。

kamel run --dependency または **-d** オプションを使用して、依存関係を明示的に追加できます。これを使用して、Camel カタログに含まれていない依存関係を指定する必要がある場合があります。コマンドラインで複数の依存関係を指定できます。

前提条件

- [Setting up your Camel K development environment](#)

手順

- **kamel run** コマンドを入力し、**-d** オプションを使用して依存関係を指定します。以下に例を示します。

```
kamel run -d mvn:com.google.guava:guava:26.0-jre -d camel-mina2 Integration.java
```



注記

-trait dependencies.enabled=false のように、依存関係トレイトを無効にすると、依存関係の自動解決を無効することができます。ただし、これはほとんどの場合で推奨されません。

関連情報

- [開発モードでの Camel K インテグレーションの実行](#)
- [Camel K トレイトおよびプロファイルの設定](#)
- [Apache Camel Mail component](#)
- [Apache Camel SEDA component](#)

第5章 KAFKA に対する CAMEL K の認証

Apache Kafka に対して Camel K を認証できます。

以下の例では、Red Hat OpenShift Streams for Apache Kafka を使用して Kafka トピックを設定し、それを単純なプロデューサー/コンシューマーパターンのインテグレーションで使用方法を実証しています。

5.1. KAFKA の設定

Kafka を設定するには、必要な OpenShift Operator のインストール、Kafka インスタンスの作成、Kafka トピックの作成が必要になります。

以下の Red Hat 製品のいずれかを使用して Kafka を設定します。

- **Red Hat Advanced Message Queuing (AMQ) ストリーム**: 自己管理の Apache Kafka オファリング。AMQ Streams はオープンソースの [Strimzi](#) をベースとしており、[Red Hat Integration](#) の一部として組み込まれています。AMQ Streams は、パブリッシュ/サブスクライブメッセージングブローカーが含まれる Apache Kafka をベースとした分散型でスケーラブルなストリーミングプラットフォームです。Kafka Connect は、Kafka ベースのシステムを外部システムと統合するフレームワークを提供します。Kafka Connect を使用すると、外部システムと Kafka ブローカーとの間で双方向にデータをストリーミングするように ソースおよびシンクコネクタを設定できます。
- **Red Hat OpenShift Streams for Apache Kafka**: Apache Kafka の実行プロセスを簡素化するマネージドクラウドサービスです。これにより、新しいクラウドネイティブアプリケーションを構築、デプロイ、およびスケーリングする際、または既存システムを現代化する際に、効率的な開発者エクスペリエンスが提供されます。

5.1.1. AMQ Streams を使用した Kafka の設定

AMQ Streams は、OpenShift クラスターで Apache Kafka を実行するプロセスを簡素化します。

5.1.1.1. AMQ Streams の OpenShift クラスターの準備

Camel K または Kamelets および Red Hat AMQ Streams を使用するには、以下の Operator およびツールをインストールする必要があります。

- **Red Hat Integration - AMQ Streams Operator**: OpenShift Cluster と AMQ Streams for Apache Kafka インスタンスの間の通信を管理します。
- **Red Hat Integration - Camel K Operator**: Camel K (OpenShift のクラウドでネイティブに実行される軽量なインテグレーションフレームワーク) をインストールし、管理します。
- **Camel K CLI ツール**: すべての Camel K 機能にアクセスできます。

前提条件

- Apache Kafka の概念を理解している。
- 適切なアクセスレベルで OpenShift 4.6 (またはそれ以降の) クラスターにアクセスできること。この場合、プロジェクトの作成および Operator のインストールができること。また、OpenShift および Camel K CLI をローカルシステムにインストールできること。

- コマンドラインで OpenShift クラスターと対話できるように OpenShift CLI ツール (**oc**) をインストールしていること。

手順

AMQ Streams を使用して Kafka を設定するには、以下を行います。

1. OpenShift クラスターの Web コンソールにログインします。
2. インテグレーションを作成する予定のプロジェクト (例: **my-camel-k-kafka**) を作成または開きます。
3. 「[Installing Camel K](#)」の説明に従って、Camel K Operator および Camel K CLI をインストールします。
4. AMQ Streams Operator をインストールします。
 - a. 任意のプロジェクトから **Operators > OperatorHub** を選択します。
 - b. **Filter by Keyword** フィールドに **AMQ Streams** を入力します。
 - c. **Red Hat Integration - AMQ Streams**カードをクリックしてから **Install** をクリックします。
Install Operator ページが開きます。
 - d. デフォルトを受け入れ、**Install** をクリックします。
5. **Operators > Installed Operators** を選択し、Camel K および AMQ Streams Operator がインストールされていることを確認します。

次のステップ

[AMQ Streams を使用した Kafka トピックの設定](#)

5.1.1.2. AMQ Streams を使用した Kafka トピックの設定

Kafka トピックは、Kafka インスタンスのデータの保存先を提供します。データを送信する前に、Kafka トピックを設定する必要があります。

前提条件

- OpenShift クラスターにアクセスできる。
- 「[Preparing your OpenShift cluster](#)」の手順に従って、**Red Hat Integration - Camel K**および**Red Hat Integration - AMQ Streams**Operator をインストールしている。
- OpenShift CLI (**oc**) および Camel K CLI (**kamel**) をインストールしている。

手順

AMQ Streams を使用して Kafka トピックを設定するには、以下を行います。

1. OpenShift クラスターの Web コンソールにログインします。
2. **Projects** を選択してから、**Red Hat Integration - AMQ Streams**Operator をインストールしたプロジェクトをクリックします。たとえば、**my-camel-k-kafka** プロジェクトをクリックします。

3. **Operators > Installed Operators** の順に選択し、**Red Hat Integration - AMQ Streams**をクリックします。
4. Kafka クラスターを作成します。
 - a. **Kafka** で、**Create instance** をクリックします。
 - b. **kafka-test** などクラスターの名前を入力します。
 - c. その他のデフォルトを受け入れ、**Create** をクリックします。
Kafka インスタンスを作成するプロセスの完了に数分かかる場合があります。

ステータスが **ready** になったら、次のステップに進みます。
5. Kafka トピックを作成します。
 - a. **Operators > Installed Operators** の順に選択し、**Red Hat Integration - AMQ Streams**をクリックします。
 - b. **Kafka Topic** で **Create Kafka Topic** をクリックします。
 - c. トピックの名前を入力します (例: **test-topic**)。
 - d. その他のデフォルトを受け入れ、**Create** をクリックします。

5.1.2. OpenShift Streams を使用した Kafka の設定

Red Hat OpenShift Streams for Apache Kafka は、Apache Kafka の実行プロセスを簡素化する管理クラウドサービスです。

OpenShift Streams for Apache Kafka を使用するには、Red Hat アカウントにログインする必要があります。

関連項目

- [Product documentation for Red Hat OpenShift Streams for Apache Kafka](#)

5.1.2.1. OpenShift Streams の OpenShift クラスターの準備

Red Hat OpenShift Streams for Apache Kafka 管理クラウドサービスを使用するには、以下の Operator およびツールをインストールする必要があります。

- **OpenShift Application Services (RHOAS) CLI**: ターミナルからアプリケーションサービスを管理できます。
- **Red Hat Integration - Camel K Operator** は、Camel K (OpenShift のクラウドでネイティブに実行される軽量なインテグレーションフレームワーク) をインストールし、管理します。
- **Camel K CLI ツール**: すべての Camel K 機能にアクセスできます。

前提条件

- Apache Kafka の概念を理解している。

- 適切なアクセスレベルで OpenShift 4.6 (またはそれ以降の) クラスターにアクセスできること。この場合、プロジェクトの作成および Operator のインストールができること。また、OpenShift および Apache Camel K CLI をローカルシステムにインストールできること。
- コマンドラインで OpenShift クラスターと対話できるように OpenShift CLI ツール (**oc**) をインストールしていること。

手順

1. クラスター管理者アカウントで OpenShift Web コンソールにログインします。
2. Camel K または Kamelets アプリケーションの OpenShift プロジェクトを作成します。
 - a. **Home > Projects** を選択します。
 - b. **Create Project** をクリックします。
 - c. プロジェクトの名前 (例: **my-camel-k-kafka**) を入力し、続いて **Create** をクリックします。
3. 『[Getting started with the rhoas CLI](#)』の説明に従って、RHOAS CLI をダウンロードし、インストールします。
4. 「[Installing Camel K](#)」の説明に従って、Camel K Operator および Camel K CLI をインストールします。
5. **Red Hat Integration - Camel K Operator** がインストールされていることを確認するには、**Operators > Installed Operators** の順にクリックします。

次のステップ

RHOAS を使用した Kafka トピックの設定

5.1.2.2. RHOAS を使用した Kafka トピックの設定

Kafka は **トピック** に関するメッセージを整理します。各トピックには名前があります。アプリケーションは、トピックにメッセージを送信し、トピックからメッセージを取得します。Kafka トピックは、Kafka インスタンスのデータの保存先を提供します。データを送信する前に、Kafka トピックを設定する必要があります。

前提条件

- 適切なアクセスレベルで OpenShift クラスターにアクセスできること。この場合、プロジェクトの作成および Operator のインストールができること。また、OpenShift および Camel K CLI をローカルシステムにインストールできること。
- 「[Preparing your OpenShift cluster](#)」の手順に従って、OpenShift CLI(**oc**)、Camel K CLI(**kamel**)、および RHOAS CLI(**rhoas**)ツールをインストールしている。
- 「[Preparing your OpenShift cluster](#)」の手順に従って、**Red Hat Integration - Camel K Operator** をインストールしている。
- [Red Hat Cloud サイト](#) にログインしている。

手順

Red Hat OpenShift Streams for Apache Kafka を使用して Kafka トピックを設定するには、以下を行います。

1. コマンドラインから OpenShift クラスターにログインします。
2. プロジェクトを開きます。以下に例を示します。
oc project my-camel-k-kafka
3. Camel K Operator がプロジェクトにインストールされていることを確認します。

oc get csv

結果には、Red Hat Camel K Operator が表示され、それが **Succeeded** フェーズにあることを示します。

4. Kafka インスタンスを準備し、RHOAS に接続します。
 - a. 以下のコマンドを使用して RHOAS CLI にログインします。

rhoas login

- b. **kafka-test** などの kafka インスタンスを作成します。

rhoas kafka create kafka-test

Kafka インスタンスを作成するプロセスの完了に数分かかる場合があります。

5. Kafka インスタンスのステータスを確認するには、以下を実行します。

rhoas status

Web コンソールでステータスを表示することもできます。

<https://cloud.redhat.com/application-services/streams/kafkas/>

ステータスが **ready** になったら、次のステップに進みます。

6. 新しい Kafka トピックを作成します。
rhoas kafka topic create --name test-topic
7. Kafka インスタンス (クラスター) を Openshift Application Services インスタンスに接続します。
rhoas cluster connect
8. クレデンシャルトークンを取得するスクリプトの手順に従います。
以下のような出力が表示されるはずです。

```
Token Secret "rh-cloud-services-accesstoken-cli" created successfully
Service Account Secret "rh-cloud-services-service-account" created successfully
KafkaConnection resource "kafka-test" has been created
KafkaConnection successfully installed on your cluster.
```

次のステップ

- [Kafka クレデンシャルの取得](#)

5.1.2.3. Kafka クレデンシャルの取得

アプリケーションまたはサービスを Kafka インスタンスに接続するには、まず以下の Kafka クレデンシャルを取得する必要があります。

- ブートストラップ URL を取得します。
- クレデンシャル (ユーザー名とパスワード) を使用してサービスアカウントを作成します。

OpenShift Streams では、認証プロトコルは SASL_SSL です。

前提条件

- Kafka インスタンスを作成し、ステータスが ready である。
- Kafka トピックを作成している。

手順

1. Kafka ブローカーの URL (ブートストラップ URL) を取得します。

rhoas status

コマンドは、以下のような出力を返します。

```
Kafka
-----
ID:          1ptdfZRHmLKwqW6A3YKM2MawgDh
Name:        my-kafka
Status:      ready
Bootstrap URL: my-kafka--ptdfzrhmlkwqw-a-ykm-mawgdh.kafka.devshift.org:443
```

2. ユーザー名とパスワードを取得するには、以下の構文を使用してサービスアカウントを作成します。

rhoas service-account create --name "<account-name>" --file-format json



注記

サービスアカウントの作成時に、ファイル形式および場所を選択して認証情報を保存できます。詳細は、**rhoas service-account create --help** コマンドを参照してください。

以下に例を示します。

rhoas service-account create --name "my-service-acct" --file-format json

サービスアカウントが作成され、JSON ファイルに保存されます。

3. サービスアカウントの認証情報を確認するには、**credentials.json** ファイルを表示します。

cat credentials.json

コマンドは、以下のような出力を返します。

```
{"clientID":"srvc-acct-eb575691-b94a-41f1-ab97-50ade0cd1094", "password":"facf3df1-3c8d-4253-aa87-8c95ca5e1225"}
```

4. Kafka トピックとの間でメッセージを送受信する権限を付与します。以下のコマンドを使用します。ここで、**clientID** は (ステップ 3 からの) **credentials.json** ファイルで指定される値に置き換えます。

```
rhoas kafka acl grant-access --producer --consumer --service-account $CLIENT_ID --topic
test-topic --group all
```

以下に例を示します。

```
rhoas kafka acl grant-access --producer --consumer --service-account srvc-acct-eb575691-
b94a-41f1-ab97-50ade0cd1094 --topic test-topic --group all
```

5.1.2.4. SASL/Plain 認証を使用したシークレットの作成

取得したクレデンシャル（Kafka ブートストラップ URL、サービスアカウント ID およびサービスアカウントのシークレット）を使用してシークレットを作成できます。

手順

1. **application.properties** ファイルを編集し、Kafka 認証情報を追加します。

application.properties ファイル

```
camel.component.kafka.brokers = <YOUR-KAFKA-BOOTSTRAP-URL-HERE>
camel.component.kafka.security-protocol = SASL_SSL
camel.component.kafka.sasl-mechanism = PLAIN
camel.component.kafka.sasl-jaas-
config=org.apache.kafka.common.security.plain.PlainLoginModule required
username='<YOUR-SERVICE-ACCOUNT-ID-HERE>' password='<YOUR-SERVICE-
ACCOUNT-SECRET-HERE>';
consumer.topic=<TOPIC-NAME>
producer.topic=<TOPIC-NAME>
```

2. 以下のコマンドを実行して、**application.properties** ファイルに機密プロパティが含まれるシークレットを作成します。

```
oc create secret generic kafka-props --from-file application.properties
```

Camel K インテグレーションの実行時に、このシークレットを使用します。

関連項目

[Camel K Kafka ベーシッククイックスタート](#)

5.1.2.5. SASL/OAUTHBearer 認証を使用したシークレットの作成

取得したクレデンシャル（Kafka ブートストラップ URL、サービスアカウント ID およびサービスアカウントのシークレット）を使用してシークレットを作成できます。

手順

1. **application-oauth.properties** ファイルを編集し、Kafka 認証情報を追加します。

application-oauth.properties ファイル

```
camel.component.kafka.brokers = <YOUR-KAFKA-BOOTSTRAP-URL-HERE>
camel.component.kafka.security-protocol = SASL_SSL
```



```
camel.component.kafka.sasl-mechanism = OAUTHBEARER
camel.component.kafka.sasl-jaas-config =
org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
oauth.client.id='<YOUR-SERVICE-ACCOUNT-ID-HERE>' \
oauth.client.secret='<YOUR-SERVICE-ACCOUNT-SECRET-HERE>' \
oauth.token.endpoint.uri="https://identity.api.openshift.com/auth/realms/rhoas/protocol/openid-
connect/token" ;
camel.component.kafka.additional-
properties[sasl.login.callback.handler.class]=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginC
allbackHandler

consumer.topic=<TOPIC-NAME>
producer.topic=<TOPIC-NAME>
```

2. 以下のコマンドを実行して、**application.properties** ファイルに機密プロパティが含まれるシークレットを作成します。

```
oc create secret generic kafka-props --from-file application-oauth.properties
```

Camel K インテグレーションの実行時に、このシークレットを使用します。

関連項目

[Camel K Kafka ベシッククイックスタート](#)

5.2. KAFKA インテグレーションの実行

プロデューサーインテグレーションの実行

1. サンプルプロデューサーインテグレーションを作成します。これにより、トピックには 10 秒ごとにメッセージが表示されます。

Sample SaslSSLKafkaProducer.java

```
// kamel run --secret kafka-props SaslSSLKafkaProducer.java --dev
// camel-k: language=java dependency=mvn:org.apache.camel.quarkus:camel-quarkus-
kafka dependency=mvn:io.strimzi:kafka-oauth-client:0.7.1.redhat-00003

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.kafka.KafkaConstants;

public class SaslSSLKafkaProducer extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        log.info("About to start route: Timer -> Kafka ");
        from("timer:foo")
            .routeId("FromTimer2Kafka")
            .setBody()
            .simple("Message #${exchangeProperty.CamelTimerCounter}")
            .to("kafka:{{producer.topic}}")
            .log("Message correctly sent to the topic!");
    }
}
```

- その後、プロデューサーインテグレーションを実行します。

```
kamel run --secret kafka-props SaslSSLKafkaProducer.java --dev
```

プロデューサーは新しいメッセージを作成し、トピックにプッシュし、一部の情報をログに記録します。

```
[2] 2021-05-06 08:48:11,854 INFO [FromTimer2Kafka] (Camel (camel-1) thread #1 -
KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:11,854 INFO [FromTimer2Kafka] (Camel (camel-1) thread #3 -
KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:11,973 INFO [FromTimer2Kafka] (Camel (camel-1) thread #5 -
KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:12,970 INFO [FromTimer2Kafka] (Camel (camel-1) thread #7 -
KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:13,970 INFO [FromTimer2Kafka] (Camel (camel-1) thread #9 -
KafkaProducer[test]) Message correctly sent to the topic!
```

コンシューマーインテグレーションの実行

- コンシューマーインテグレーションを作成します。

Sample SaslSSLKafkaConsumer.java

```
// kamel run --secret kafka-props SaslSSLKafkaConsumer.java --dev
// camel-k: language=java dependency=mvn:org.apache.camel.quarkus:camel-quarkus-
kafka dependency=mvn:io.strimzi:kafka-oauth-client:0.7.1.redhat-00003

import org.apache.camel.builder.RouteBuilder;

public class SaslSSLKafkaConsumer extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        log.info("About to start route: Kafka -> Log ");
        from("kafka:{{consumer.topic}}")
            .routeId("FromKafka2Log")
            .log("${body}");
    }
}
```

- 別のシェルを開き、コマンドを使用してコンシューマーインテグレーションを実行します。

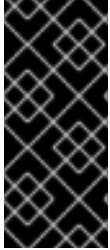
```
kamel run --secret kafka-props SaslSSLKafkaConsumer.java --dev
```

コンシューマーは、トピックにあるイベントのロギングを開始します。

```
[1] 2021-05-06 08:51:08,991 INFO [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #8
[1] 2021-05-06 08:51:10,065 INFO [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #9
[1] 2021-05-06 08:51:10,991 INFO [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #10
[1] 2021-05-06 08:51:11,991 INFO [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #11
```

第6章 CAMEL K トレイト設定の参考情報

本章では、**トレイト**を使用して実行時にコマンドラインで設定できる高度な機能とコア機能に関する参考情報を紹介します。Camel K は、特定の機能および技術を設定する **機能トレイト** (feature trait) を提供します。Camel K は、内部の Camel K コア機能を設定する **プラットフォームトレイト** を提供します。



重要

Red Hat Integration - Camel K 1.6 には、**OpenShift** および **Knative** プロファイルが含まれています。**Kubernetes** プロファイルのサポートはコミュニティのみに限定されます。これには、インテグレーション用の Java および YAML DSL のサポートも含まれます。XML、Groovy、JavaScript、Kourtlín などのその他の言語のサポートはコミュニティのみに限定されます。

本章には、以下が含まれます。

Camel K 機能トレイト

- [「Knative トレイト」](#) : テクノロジープレビュー
- [「Knative Service トレイト」](#) : テクノロジープレビュー
- [「Prometheus トレイト」](#)
- [「Pdb トレイト」](#)
- [「Pull Secret トレイト」](#)
- [「Route トレイト」](#)
- [「Service トレイト」](#)

Camel K コアプラットフォームトレイト

- [「Builder トレイト」](#)
- [「Camel トレイト」](#)
- [「Container トレイト」](#)
- [「Dependencies トレイト」](#)
- [「Deployer トレイト」](#)
- [「Deployment トレイト」](#)
- [「Environment トレイト」](#)
- [「Error Handler トレイト」](#)
- [「JVM トレイト」](#)
- [「Kamelets トレイト」](#)
- [「Openapi トレイト」](#) : テクノロジープレビュー

- 「Owner トレイト」
- 「プラットフォームトレイト」
- 「Quarkus トレイト」

6.1. CAMEL K トレイトおよびプロファイルの設定

ここでは、実行時に 高度な Camel K 機能を設定するために使用される **トレイト** および **プロファイル** の重要な Camel K の概念について説明します。

Camel K トレイト

Camel K トレイトは高度な機能およびコア機能で、Camel K インテグレーションをカスタマイズするためにコマンドラインで設定できます。たとえば、これには、3scale API Management、Quarkus、Knative、Prometheus などのテクノロジーとの対話を設定する **機能トレイト** (feature trait) が含まれます。Camel K は、Camel サポート、コンテナー、依存関係の解決、JVM サポートなどの重要なコアプラットフォーム機能を設定する、内部の**プラットフォームトレイト** も提供します。

Camel K プロファイル

Camel K プロファイルは、Camel K インテグレーションが実行されるターゲットクラウドプラットフォームを定義します。サポートされるプロファイルは **OpenShift** および **Knative** プロファイルです。



注記

OpenShift でインテグレーションを実行するときに、OpenShift Serverless がクラスターにインストールされている場合、Camel K は **Knative** プロファイルを使用します。OpenShift Serverless がインストールされていない場合、Camel K は **OpenShift** プロファイルを使用します。

kamel run --profile オプションを使用して、実行時にプロファイルを指定することもできます。

Camel K は、インテグレーションが実行されるターゲットプロファイルを考慮し、便利なデフォルトをすべてのトレイトに提供します。ただし、上級ユーザーはカスタム動作の Camel K トレイトを設定できます。一部のトレイトは、**OpenShift** や **Knative** などの特定のプロファイルにのみ適用されます。詳細は、各トレイトの説明にある利用可能なプロファイルを参照してください。

Camel K トレイトの設定

各 Camel トレイトには、コマンドラインでトレイトを設定するために使用する一意の ID があります。たとえば、以下のコマンドは、インテグレーションの OpenShift Service の作成を無効にします。

```
kamel run --trait service.enabled=false my-integration.yaml
```

-t オプションを使用してトレイトを指定することもできます。

Camel K トレイトプロパティ

enabled プロパティを使用して、各トレイトを有効または無効にすることができます。すべてのトレイトには、ユーザーが明示的にアクティブ化しない場合に有効にする必要があるかどうかを判断する独自の内部ロジックがあります。



警告

プラットフォームトレイトを無効にすると、プラットフォームの機能が低下する可能性があります。

一部のトレイトには、環境に応じてトレイトの自動設定を有効または無効にするために使用する **auto** プロパティがあります。たとえば、3scale、Cron、Knative などのトレイトが含まれます。この自動設定では、**enabled** プロパティが明示的に設定されていない場合にトレイトを有効または無効にすることができ、トレイトの設定を変更することができます。

ほとんどのトレイトには、コマンドラインで設定できる追加のプロパティがあります。詳細は、これ以降のセクションで各トレイトの説明を参照してください。

6.2. CAMEL K 機能トレイト

6.2.1. Knative トレイト

Knative トレイトは Knative リソースのアドレスを自動検出し、実行中のインテグレーションに注入します。

完全な Knative 設定は、JSON 形式の CAMEL_KNATIVE_CONFIGURATION に注入されます。その後、Camel Knative コンポーネントは完全な設定を使用してルートを設定します。

このトレイトは、Knative プロファイルがアクティブであるとデフォルトで有効になります。

このトレイトは、Knative プロファイルで利用できます。

6.2.1.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait knative.[key]=[value] --trait knative.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
knative.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
knative.configuration	string	Knative の完全な設定を JSON 形式で注入するために使用されます。
knative.channel-sources	[]string	インテグレーションルートのソースとして使用されるチャンネルの一覧。簡単なチャンネル名または完全な Camel URI を含めることができます。

プロパティ	型	詳細
knative.channel-sinks	[]string	インテグレーションルートの宛先として使用されるチャネルの一覧。簡単なチャネル名または完全な Camel URI を含めることができます。
knative.endpoint-sources	[]string	インテグレーションルートのソースとして使用されるチャネルの一覧。
knative.endpoint-sinks	[]string	インテグレーションルートの宛先として使用されるエンドポイントの一覧。簡単なエンドポイント名または完全な Camel URI を含めることができます。
knative.event-sources	[]string	インテグレーションがサブスクライブされるイベントタイプのリスト。簡単なイベントタイプまたは完全な Camel URI を含めることができます (デフォルト以外の特定のブローカーを使用するため)。
knative.event-sinks	[]string	インテグレーションが生成するイベントタイプのリスト。簡単なイベントタイプまたは完全な Camel URI を含めることができます (特定のブローカーを使用するため)。
knative.filter-source-channels	bool	ヘッダー「ce-knativehistory」を基にしてイベントのフィルターを有効にします。このヘッダーは新しいバージョンの Knative で削除されたため、フィルターはデフォルトで無効になっています。
knative.sink-binding	bool	Knative SinkBinding リソース経由のインテグレーションからシンクへのバインドを許可します。これは、インテグレーションが単一のシンクをターゲットにする場合に使用できます。インテグレーションが単一のシンクをターゲットにする場合、デフォルトで有効になります (インテグレーションが Knative ソースによって所有されている場合を除く)。
knative.auto	bool	すべてのトレイトプロパティの自動検出を有効にします。

6.2.2. Knative Service トレイト

Knative Service トレイトは、標準の Kubernetes デプロイメントではなく、Knative サービスとしてインテグレーションを実行する場合に、オプションの設定を可能にします。

Knative Services としてインテグレーションを実行すると、自動スケーリング (およびゼロへのスケーリング) 機能が追加されますが、この機能はルートが HTTP エンドポイントコンシューマーを使用する場合にのみ有効です。

このトレイトは、Knative プロファイルで利用できます。

6.2.2.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

-

```
$ kamel run --trait knative-service.[key]=[value] --trait knative-service.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
knative-service.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
knative-service.autoscaling-class	string	<p>Knative 自動スケーリングクラスプロパティを設定します (hpa.autoscaling.knative.dev または kpa.autoscaling.knative.dev の自動スケーリングを設定します)。</p> <p>詳細は、Knative ドキュメントを参照してください。</p>
knative-service.autoscaling-metric	string	<p>Knative 自動スケーリングメトリクスプロパティを設定します (例: concurrency または cpu ベースの自動スケーリングを設定します)。</p> <p>詳細は、Knative ドキュメントを参照してください。</p>
knative-service.autoscaling-target	int	<p>各 Pod に許可される同時実行レベルまたは CPU の割合 (自動スケーリングメトリクスによる) を設定します。</p> <p>詳細は、Knative ドキュメントを参照してください。</p>
knative-service.min-scale	int	<p>インテグレーションに対して稼働している必要がある Pod の最小数。デフォルトは ゼロ であるため、設定された期間に使用されなければインテグレーションはゼロにスケールダウンされます。</p> <p>詳細は、Knative ドキュメントを参照してください。</p>
knative-service.max-scale	int	<p>インテグレーションで並行して実行できる Pod 数の上限。Knative には、インストールによって異なる独自の上限値があります。</p> <p>詳細は、Knative ドキュメントを参照してください。</p>
knative-service.auto	bool	<p>以下のすべての条件が保持されると、インテグレーションを Knative サービスとして自動的にデプロイします。</p> <ul style="list-style-type: none"> ● インテグレーションは Knative プロファイルを使用する。 ● すべてのルートは、HTTP ベースのコンシューマーまたはパッシブコンシューマーから開始される (例: direct はパッシブコンシューマー)。

6.2.3. Prometheus トレイト

Prometheus トレイトは、Prometheus と互換性のあるエンドポイントを設定します。Prometheus Operator の使用時にエンドポイントを自動的にスクレイプできるように、**PodMonitor** リソースも作成します。

メトリクスは MicroProfile Metrics を使用して公開されます。



警告

PodMonitor リソースを作成するには、[Prometheus Operator](#) のカスタムリソース定義をインストールする必要があります。Prometheus トレイトが Prometheus Operator なしで機能するように、**pod-monitor** を **false** に設定できます。

Prometheus トレイトはデフォルトで無効になっています。

このトレイトは **Kubernetes**、**Knative**、および **OpenShift** プロファイルで利用できます。

6.2.3.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait prometheus.[key]=[value] --trait prometheus.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
prometheus.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
prometheus.pod-monitor	bool	PodMonitor リソースが作成されるかどうか (デフォルトは true)。
prometheus.pod-monitor-labels	[]string	PodMonitor リソースラベル。 pod-monitor が true の場合に適用されます。

6.2.4. Pdb トレイト

PDB トレイトを使用すると、インテグレーション Pod の PodDisruptionBudget リソースを設定することができます。

このトレイトは **Kubernetes**、**Knative**、および **OpenShift** プロファイルで利用できます。

6.2.4.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait pdb.[key]=[value] --trait pdb.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
pdb.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
pdb.min-available	string	エビクション後も引き続き利用可能である必要があるインテグレーションの Pod 数。絶対数またはパーセンテージのいずれかで指定できます。 min-available および max-unavailable のいずれかのみを指定できます。
pdb.max-unavailable	string	エビクション後に、利用不可能であっても許容されるインテグレーションの Pod 数。絶対数またはパーセンテージのいずれかで指定できます (min-available も設定されない場合、デフォルトは 1 です)。 max-unavailable および min-available のいずれかのみを指定できます。

6.2.5. Pull Secret トレイト

Pull Secret トレイトは Pod にプルシークレットを設定し、Kubernetes が外部レジストリーからコンテナイメージを取得できるようにします。

プルシークレットは手動で指定するか、**IntegrationPlatform** で外部コンテナレジストリーに対する認証を設定している場合には、イメージをプルするのに同じシークレットを使用します。

デフォルトでは、外部コンテナレジストリーの認証を設定する際に常に有効になっているので、外部レジストリーがプライベートであることを前提としています。

イメージをプルするためにレジストリーで認証が必要ない場合は、この特性を無効にすることができます。

このトレイトは Kubernetes、Knative、および OpenShift プロファイルで利用できます。

6.2.5.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait pull-secret.[key]=[value] --trait pull-secret.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
pull-secret.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
pull-secret.secret-name	string	Pod に設定されるプルシークレット名。空のままにすると、自動的に IntegrationPlatform レジストリー設定から取得されます。
pull-secret.image-puller-delegation	bool	共有プラットフォームでグローバル Operator を使用する場合は、Operator namespace 上の system:image-puller クラスターロールの統合サービスアカウントへの委譲が可能になります。

プロパティ	型	詳細
pull-secret.auto	bool	タイプが kubernetes.io/dockerconfigjson の場合、Pod にプラットフォームレジストリーシークレットを自動的に設定します。

6.2.6. Route トレイト

Route トレイトを使用すると、インテグレーションの OpenShift ルートの作成を設定できます。

証明書とキーの内容は、ローカルファイルシステムまたは Openshift **secret** オブジェクトのいずれかから取得できます。ユーザーは、**-secret**で終わるパラメーター (例: **tls-certificate-secret**) を使用して、**secret**に格納されている証明書を参照できます。**-secret**で終わるパラメーターの優先順位は高く、同じルートパラメーターが設定されている場合 (例: **tls-key-secret**と**tls-key**)、**tls-key-secret**が使用されます。キーと証明書を設定するための推奨されるアプローチは、**secrets**を使用してコンテンツを格納し、**tls-certificate-secret**、**tls-key-secret**、**tls-ca-certificate-secret**、**tls-destination-ca-certificate-secret** パラメーターを使用してそれらを参照することです。設定オプションについては、このページの最後にある例のセクションを参照してください。

このトレイトは、OpenShift のプロファイルで利用できます。

6.2.6.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait route.[key]=[value] --trait route.[key2]=[value2] integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
route.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
route.host	string	ルートによって公開されるホストを設定します。
route.tls-termination	string	edge 、 passthrough 、または reencrypt などの TLS 終端タイプ。 追加情報については、OpenShift ルートのドキュメントを参照してください。
route.tls-certificate	string	TLS 証明書の内容。 追加情報については、OpenShift ルートのドキュメントを参照してください。

プロパティ	型	詳細
route.tls-certificate-secret	string	<p>TLS 証明書へのシークレット名とキー参照。形式は <code>secret-name[/key-name]</code> です。値はシークレット名を表します。シークレットにキーが1つしかない場合は読み取られます。それ以外の場合は、/ で区切ってキー名を設定できます。</p> <p>追加情報については、OpenShift ルートのドキュメントを参照してください。</p>
route.tls-key	string	<p>TLS 証明書キーの内容。</p> <p>追加情報については、OpenShift ルートのドキュメントを参照してください。</p>
route.tls-key-secret	string	<p>TLS 証明書キーへのシークレット名とキー参照。形式は <code>secret-name[/key-name]</code> です。値はシークレット名を表します。シークレットにキーが1つしかない場合は読み取られます。それ以外の場合は、/ で区切ってキー名を設定できます。</p> <p>追加情報については、OpenShift ルートのドキュメントを参照してください。</p>
route.tls-ca-certificate	string	<p>TLS CA 証明書の内容。</p> <p>追加情報については、OpenShift ルートのドキュメントを参照してください。</p>
route.tls-ca-certificate-secret	string	<p>TLS CA 証明書へのシークレット名とキー参照。形式は <code>secret-name[/key-name]</code> です。値はシークレット名を表します。シークレットにキーが1つしかない場合は読み取られます。それ以外の場合は、/ で区切ってキー名を設定できます。</p> <p>追加情報については、OpenShift ルートのドキュメントを参照してください。</p>
route.tls-destination-ca-certificate	string	<p>宛先 CA 証明書は、最終宛先の CA 証明書の内容を提供します。reencrypt の停止を使用する場合、ルーターがセキュアな接続のヘルスチェックに使用するためにこのファイルを提供する必要があります。このフィールドが指定されていない場合、ルーターは独自の宛先 CA を提供し、短いサービス名 (<code>service.namespace.svc</code>) を使用してホスト名の検証を実行する可能性があります。これにより、インフラストラクチャーが生成した証明書を自動的に検証できます。</p> <p>追加情報については、OpenShift ルートのドキュメントを参照してください。</p>

プロパティ	型	詳細
route.tls-destination-ca-certificate-secret	string	宛先 CA 証明書へのシークレット名とキー参照。形式は <code>secret-name[/key-name]</code> です。値はシークレット名を表します。シークレットにキーが1つしかない場合は読み取られます。それ以外の場合は、/ で区切ってキー名を設定できます。 追加情報については、OpenShift ルートのドキュメントを参照してください。
route.tls-insecure-edge-termination-policy	string	セキュアでないトラフィック (Allow 、 Disable 、または Redirect トラフィックなど) に対応する方法を設定します。 追加情報については、OpenShift ルートのドキュメントを参照してください。

6.2.6.2. 例

これらの例では、**secrets** を使用して、統合で参照される証明書とキーを格納します。ルートの詳細については、OpenShift ルートのドキュメントをお読みください。 [PlatformHttpServer.java](#) は統合の例です。

これらの例を実行するための要件として、キーと証明書を含む **secret** が必要です。

6.2.6.2.1. 自己署名証明書を生成し、シークレットを作成します

```
openssl genrsa -out tls.key
openssl req -new -key tls.key -out csr.csr -subj "/CN=my-server.com"
openssl x509 -req -in csr.csr -signkey tls.key -out tls.crt
oc create secret tls my-combined-certs --key=tls.key --cert=tls.crt
```

6.2.6.2.2. ルートへの HTTP リクエストの作成

すべての例で、次の `curl` コマンドを使用して HTTP リクエストを作成できます。インラインスクリプトを使用して、`openshift` 名前空間とクラスターベースドメインを取得します。これらのインラインスクリプトをサポートしないシェルを使用している場合は、インラインスクリプトを実際の名前空間とベースドメインの値に置き換える必要があります。

```
curl -k https://platform-http-server-`oc config view --minify -o 'jsonpath={..namespace}`.`oc get dnses/cluster -ojsonpath='{.spec.baseDomain}`/hello?name=Camel-K
```

- シークレットを使用してエッジルートを追加するには、**-secret**で終わるパラメーターを使用して、証明書を含むシークレット名を設定します。このルート例の特性は、**tls.key**と**tls.crt**という名前の2つのキーを含む**my-combined-certs**という名前のシークレットを参照しています。

```
kamel run --dev PlatformHttpServer.java -t route.tls-termination=edge -t route.tls-certificate-secret=my-combined-certs/tls.crt -t route.tls-key-secret=my-combined-certs/tls.key
```

- シークレットを使用してパススルールートを追加するには、統合 Pod で TLS を設定し、実行中の統合 Pod でキーと証明書を表示する必要があります。これを実現するには、**--resource** `kamel` パラメーターを使用して統合 Pod にシークレットをマウントします。次に、いくつかの `camel quarkus` パラメーターを使用して、実行中の Pod でこれらの証明書ファイルを参照しま

す。これらのファイルは、**-p quarkus.http.ssl.certificate** で始まります。このルート例の特性は、**tls.key**と**tls.crt**という名前の2つのキーを含む**my-combined-certs**という名前のシークレットを参照しています。

```
kamel run --dev PlatformHttpServer.java --resource secret:my-combined-certs@/etc/ssl/my-combined-certs -p quarkus.http.ssl.certificate.file=/etc/ssl/my-combined-certs/tls.crt -p quarkus.http.ssl.certificate.key-file=/etc/ssl/my-combined-certs/tls.key -t route.tls-termination=passthrough -t container.port=8443
```

- シークレットを使用して**再暗号化**ルートを追加するには、統合 Pod で TLS を設定し、実行中の統合 Pod でキーと証明書を表示する必要があります。これを実現するには、**--resource** **kamel** パラメーターを使用して統合 Pod にシークレットをマウントします。次に、いくつかの **camel quarkus** パラメーターを使用して、実行中の Pod でこれらの証明書ファイルを参照します。これらのファイルは、**-p quarkus.http.ssl.certificate** で始まります。このルート例の特性は、**tls.key**と**tls.crt**という名前の2つのキーを含む**my-combined-certs**という名前のシークレットを参照しています。

```
kamel run --dev PlatformHttpServer.java --resource secret:my-combined-certs@/etc/ssl/my-combined-certs -p quarkus.http.ssl.certificate.file=/etc/ssl/my-combined-certs/tls.crt -p quarkus.http.ssl.certificate.key-file=/etc/ssl/my-combined-certs/tls.key -t route.tls-termination=reencrypt -t route.tls-destination-ca-certificate-secret=my-combined-certs/tls.crt -t route.tls-certificate-secret=my-combined-certs/tls.crt -t route.tls-key-secret=my-combined-certs/tls.key -t container.port=8443
```

- ルートのシークレットからの特定の証明書と統合エンドポイントの**証明書を提供**する **Openshift サービス**を使用して、**再暗号化**ルートを追加します。このように、証明書を提供する Openshift サービスは統合 Pod でのみセットアップされます。キーと証明書は実行中の統合 Pod に表示される必要があります。これを実現するには、**--resource** **kamel** パラメーターを使用して統合 Pod にシークレットをマウントし、次にいくつかの **camel quarkus** パラメーターを使用して実行中の Pod でこれらの証明書ファイルを参照します。**-p quarkus.http.ssl.certificate** で始まります。このルート例の特性は、**tls.key**と**tls.crt**という名前の2つのキーを含む**my-combined-certs**という名前のシークレットを参照しています。

```
kamel run --dev PlatformHttpServer.java --resource secret:cert-from-openshift@/etc/ssl/cert-from-openshift -p quarkus.http.ssl.certificate.file=/etc/ssl/cert-from-openshift/tls.crt -p quarkus.http.ssl.certificate.key-file=/etc/ssl/cert-from-openshift/tls.key -t route.tls-termination=reencrypt -t route.tls-certificate-secret=my-combined-certs/tls.crt -t route.tls-key-secret=my-combined-certs/tls.key -t container.port=8443
```

次に、統合サービスにアノテーションを付けて、証明書を提供する Openshift サービスを注入する必要があります

```
oc annotate service platform-http-server service.beta.openshift.io/serving-cert-secret-name=cert-from-openshift
```

- ローカルファイルシステムから提供された証明書と秘密鍵を使用して**エッジ**ルートを追加します。この例では、インラインスクリプトを使用して証明書と秘密鍵ファイルの内容を読み取り、すべての改行文字を削除します(これは、証明書をパラメーターの値として設定するために必要です)。したがって、値は1行になります。

```
kamel run PlatformHttpServer.java --dev -t route.tls-termination=edge -t route.tls-certificate="$(cat tls.crt|awk 'NF {sub(/\r/, ""); printf "%s\n",$0;}')" -t route.tls-key="$(cat tls.key|awk 'NF {sub(/\r/, ""); printf "%s\n",$0;}")"
```

6.2.7. Service トレイト

サービストレイトは、Service リソースとのインテグレーションを公開し、同じ namespace の他のアプリケーション (またはインテグレーション) からアクセスできるようにします。

インテグレーションが HTTP エンドポイントを公開できる Camel コンポーネントに依存する場合は、デフォルトで有効になっています。

このトレイトは **Kubernetes および OpenShift** プロファイルで利用できます。

6.2.7.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait service.[key]=[value] --trait service.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
service.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
service.auto	bool	サービスを作成する必要がある場合にコードから自動検出されます。
service.node-port	bool	Service が NodePort として公開できるようにします。

6.3. CAMEL K プラットフォームトレイト

6.3.1. Builder トレイト

Builder トレイトは、IntegrationKits を構築および設定するために最適なストラテジーを決定するために内部で使用されます。

このトレイトは **Kubernetes、Knative、および OpenShift** プロファイルで利用できます。



警告

Builder トレイトは**プラットフォームトレイト**です。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.1.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait builder.[key]=[value] --trait builder.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
builder.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
builder.verbose	bool	サポートするビルドコンポーネント (OpenShift ビルド Pod など) で詳細なロギングを有効にします。Kaniko および Buildah はサポートされません。
builder.properties	[]string	ビルドタスクに提供されるプロパティ一覧

6.3.2. Container トレイト

Container トレイトを使用すると、インテグレーションが実行されるコンテナのプロパティを設定できます。

また、コンテナに関連付けられたサービスの設定も提供します。

このトレイトは Kubernetes、Knative、および OpenShift プロファイルで利用できます。



警告

Container トレイトはプラットフォームトレイトです。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.2.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait container.[key]=[value] --trait container.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
container.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
container.auto	bool	
container.request-cpu	string	必要な CPU の最小量。

プロパティ	型	詳細
container.request-memory	string	必要なメモリーの最小容量。
container.limit-cpu	string	必要な CPU の最大量。
container.limit-memory	string	必要なメモリーの最大容量。
container.expose	bool	kubernetes サービス経由の公開を有効または無効にするために使用できます。
container.port	int	コンテナによって公開される別のポートを設定します (デフォルトは 8080)。
container.port-name	string	コンテナによって公開されるポートに異なるポート名を設定します (デフォルトは http)。
container.service-port	int	コンテナポートを公開するサービスポートを設定します (デフォルト 80)。
container.service-port-name	string	コンテナポートを公開するサービスポート名を設定します (デフォルト http)。
container.name	string	メインのコンテナ名。デフォルトでは、名前付き integration になります。
container.image	string	主なコンテナイメージ
container.probes-enabled	bool	コンテナのプロブで ProbesEnabled を有効/無効にします (デフォルトは false)。
container.liveness-initial-delay	int32	コンテナが起動してから liveness プロブが開始されるまでの秒数。
container.liveness-timeout	int32	プロブがタイムアウトするまでの秒数。liveness プロブに適用されます。
container.liveness-period	int32	プロブを実行する頻度。liveness プロブに適用されます。
container.liveness-success-threshold	int32	失敗後に、プロブが正常とみなされるための最小の連続成功回数。liveness プロブに適用されます。
container.liveness-failure-threshold	int32	正常に実行された後に失敗とみなされるプロブの連続失敗回数の最小値。liveness プロブに適用されます。

プロパティ	型	詳細
container.readiness-initial-delay	int32	コンテナが起動してから readiness プローブが開始されるまでの秒数。
container.readiness-timeout	int32	プローブがタイムアウトするまでの秒数。readiness プローブに適用されます。
container.readiness-period	int32	プローブを実行する頻度。readiness プローブに適用されます。
container.readiness-success-threshold	int32	失敗後に、プローブが正常とみなされるための最小の連続成功回数。readiness プローブに適用されます。
container.readiness-failure-threshold	int32	正常に実行された後に失敗とみなされるプローブの連続失敗回数の最小値。readiness プローブに適用されます。

6.3.3. Camel トレイト

Camel トレイトを使用すると Apache Camel K ランタイムおよび関連ライブラリーのバージョンを設定できますが、無効にすることはできません。

このトレイトは Kubernetes、Knative、および OpenShift プロファイルで利用できます。



警告

Camel トレイトはプラットフォームトレイトです。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.3.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait camel.[key]=[value] --trait camel.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
camel.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。

6.3.4. Dependencies トレイト

Dependencies トレイトは、ユーザーが実行するインテグレーションに基づいてランタイムの依存関係を自動的に追加するために内部で使用されます。

このトレイトは **Kubernetes**、**Knative**、および **OpenShift** プロファイルで利用できます。



警告

Dependencies トレイトは**プラットフォームトレイト**です。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.4.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait dependencies.[key]=[value] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
dependencies.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。

6.3.5. Deployer トレイト

Deployer トレイトを使用すると、インテグレーションをデプロイする高レベルのリソースの種類を明示的に選択できます。

このトレイトは **Kubernetes**、**Knative**、および **OpenShift** プロファイルで利用できます。



警告

Deployer トレイトは**プラットフォームトレイト**です。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.5.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait deployer.[key]=[value] --trait deployer.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
deployer.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
deployer.kind	string	インテグレーションを実行するリソースを作成する際に、 deployment 、 cron-job 、または knative-service との間で必要なデプロイメントの種類を明示的に選択できます。

6.3.6. Deployment トレイト

Deployment トレイトは、インテグレーションがクラスターで実行されるようにする Kubernetes デプロイメントを生成します。

このトレイトは Kubernetes、Knative、および OpenShift プロファイルで利用できます。



警告

Deployment トレイトはプラットフォームトレイトです。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.6.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait deployment.[key]=[value] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
deployment.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。

6.3.7. Environment トレイト

Environment トレイトは、**NAMESPACE**、**POD_NAME** などのインテグレーションコンテナに標準の環境変数を注入するために内部で使用されます。

このトレイトは Kubernetes、Knative、および OpenShift プロファイルで利用できます。

**警告**

Environment トレイトはプラットフォームトレイトです。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.7.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait environment.[key]=[value] --trait environment.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
environment.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
environment.contains-meta	bool	NAMESPACE および POD_NAME 環境変数の挿入を有効にします（デフォルトは true ）。

6.3.8. Error Handler トレイト

error-handler は、Error Handler ソースをインテグレーションランタイムに注入するのに使用されるプラットフォームトレイトです。

このトレイトは Kubernetes、Knative、および OpenShift プロファイルで利用できます。

**警告**

error-handler トレイトはプラットフォームトレイトです。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.8.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait error-handler.[key]=[value] --trait error-handler.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
error-handler.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
error-handler.ref	string	指定された、またはアプリケーションプロパティで見つかったエラーハンドラー参照名

6.3.9. JVM トレイト

JVM トレイトは、インテグレーションを実行する JVM の設定に使用されます。

このトレイトは Kubernetes、Knative、および OpenShift プロファイルで利用できます。



警告

JVM トレイトは **プラットフォームトレイト** です。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.9.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait jvm.[key]=[value] --trait jvm.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
jvm.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
jvm.debug	bool	リモートデバッグをアクティベートし、たとえばポート転送を使用して、デバッガーを JVM に接続できるようにします。
jvm.debug-suspend	bool	メインクラスがロードされる直前にターゲット JVM を一時停止します。
jvm.print-command	bool	コンテナログに JVM の開始に使用されるコマンドを出力します（デフォルトは true ）。
jvm.debug-address	string	新たに起動された JVM をリッスンするトランスポートアドレス（デフォルトは *:5005 ）
jvm.options	[]string	JVM オプションの一覧

プロパティ	型	詳細
<code>jvm.classpath</code>	<code>string</code>	追加の JVM クラスパス (Linux クラスパスセパレーターを使用)

6.3.9.2. 例

- **Integration** に追加のクラスパスを含めます。

```
$ kamel run -t jvm.classpath=/path/to/my-dependency.jar:/path/to/another-dependency.jar ...
```

6.3.10. Kamelets トレイト

kamelets トレイトは、Kamelets をインテグレーションランタイムに注入するのに使用されるプラットフォームトレイトです。

このトレイトは Kubernetes、Knative、および OpenShift プロファイルで利用できます。



警告

kamelets トレイトは **プラットフォームトレイト** です。無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.10.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait kamelets.[key]=[value] --trait kamelets.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
<code>kamelets.enabled</code>	<code>bool</code>	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
<code>kamelets.auto</code>	<code>bool</code>	参照される Kamelets とそのデフォルト設定を自動的に注入します (デフォルトで有効です)。
<code>kamelets.list</code>	<code>string</code>	現在のインテグレーションにロードする Kamelet 名のコンマ区切りリスト

6.3.11. Openapi トレイト

OpenAPI DSL トレイトは、OpenAPI 仕様からインテグレーションを作成できるように内部で使用されます。

このトレイトは **Kubernetes**、**Knative**、および **OpenShift** プロファイルで利用できます。



警告

openapi トレイトは **プラットフォームトレイト** です。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.11.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait openapi.[key]=[value] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
openapi.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。

6.3.12. Owner トレイト

Owner トレイトは、作成されたすべてのリソースが作成されたインテグレーションに属するようにし、インテグレーションのアノテーションおよびラベルをこれらの所有されたリソースに転送します。

このトレイトは **Kubernetes**、**Knative**、および **OpenShift** プロファイルで利用できます。



警告

Owner トレイトは **プラットフォームトレイト** です。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.12.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait owner.[key]=[value] --trait owner.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
owner.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
owner.target-annotations	[]string	転送するアノテーションのセット
owner.target-labels	[]string	転送するラベルのセット

6.3.13. プラットフォームトレイト

プラットフォームトレイトは、インテグレーションプラットフォームをインテグレーションに割り当てるために使用されるベーストレイトです。

プラットフォームが見つからない場合、トレイトはデフォルトのプラットフォームを作成できます。この機能は、プラットフォームにカスタム設定が必要ない場合に便利です (例: 組み込みのコンテナイメージレジストリーがあるため OpenShift ではデフォルトの設定が動作します)。

このトレイトは **Kubernetes**、**Knative**、および **OpenShift** プロファイルで利用できます。



警告

プラットフォームトレイト を無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.13.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait platform.[key]=[value] --trait platform.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
platform.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。
platform.create-default	bool	プラットフォームがない場合にデフォルトのプラットフォーム (空のプラットフォーム) を作成します。
platform.global	bool	グローバル Operator の場合、プラットフォームがグローバルに作成されるかどうかを示します (デフォルトは true です)。

プロパティ	型	詳細
platform.auto	bool	デフォルトのプラットフォームを作成できる場合に環境から自動検出します (OpenShift のみで作成)。

6.3.14. Quarkus トレイト

Quarkus トレイトにより、Quarkus ランタイムがアクティベートされます。

これはデフォルトで有効になっています。

このトレイトは Kubernetes、Knative、および OpenShift プロファイルで利用できます。



警告

quarkus トレイトは **プラットフォームトレイト** です。よって、これを無効にすると、プラットフォームの機能が低下する可能性があります。

6.3.14.1. 設定

CLI でインテグレーションを実行する際にトレイトプロパティを指定できます。

```
$ kamel run --trait quarkus.[key]=[value] --trait quarkus.[key2]=[value2] Integration.java
```

以下の設定オプションが利用できます。

プロパティ	型	詳細
quarkus.enabled	bool	トレイトを有効または無効にするのに使用できます。すべてのトレイトがこの共通プロパティを共有します。

6.3.14.2. サポートされる Camel コンポーネント

Quarkus が有効な状態で実行する場合、Camel K は Camel Quarkus エクステンションとして利用できる Camel コンポーネントのみをサポートします。エクステンションの一覧は、[Camel Quarkus ドキュメント](#) を参照してください。

第7章 CAMEL K コマンドリファレンス

本章では、Camel K コマンドラインインターフェース (CLI) の参考情報と、**kamel** コマンドの使用例を紹介します。本章では、ランタイムで実行される Camel K インテグレーションソースファイルで指定できる Camel K モードラインオプションの参考情報も提供します。

本章には、以下が含まれます。

- [「Camel K コマンドライン」](#)
- [「Camel K モードラインオプション」](#)

7.1. CAMEL K コマンドライン

Camel K CLI は、OpenShift で Camel K インテグレーションを実行するためのメインエントリーポイントとして **kamel** コマンドを提供します。

7.1.1. サポートされるコマンド

以下のキーに注意してください。

Symbol	詳細
✓	サポート対象
■	サポートされないか、まだサポート対象ではない

表7.1 kamel コマンド

名前	サポート対象	説明	例
bind	✓	インテグレーションフローの Kamelets などの Kubernetes リソースを、Knative チャネル、Kafka トピック、またはその他のエンドポイントにバインドします。	kamel bind telegram-source -p "source.authorizationToken=The Token" channel:mychannel
completion	■	補完スクリプトを生成します。	kamel completion bash
debug	■	ローカルデバッガーを使用してリモートインテグレーションをデバッグします。	kamel debug my-integration
削除	✓	OpenShift にデプロイされたインテグレーションを削除します。	kamel delete my-integration

名前	サポート対象	説明	例
describe	✓	Camel K リソースの詳細情報を取得します。これには、 integration 、 kit 、または platform が含まれます。	kamel describe integration my-integration
get	✓	OpenShift にデプロイされたインテグレーションのステータスを取得します。	kamel get
help	✓	利用可能なコマンドの完全リストを取得します。詳細は、各コマンドのパラメーターとして --help を入力します。	<ul style="list-style-type: none"> • kamel help • kamel run --help
init	✓	Java または YAML に実装された空の Camel K ファイルを初期化します。	kamel init MyIntegration.java
install	■	OpenShift クラスターに Camel K をインストールします。 注意: Camel K のインストールおよびアンインストールには、OpenShift Camel K を使用することが推奨されます。	kamel install
kit	■	Integration Kit を設定します。	kamel kit create my-integration --secret
local	■	一連の入力インテグレーションファイルのセットを指定して、インテグレーションアクションをローカルで実行します。	kamel local run
log	✓	実行中のインテグレーションのログを出力します。	kamel log my-integration
rebuild	✓	1つ以上のインテグレーションの状態を消去すると再ビルドされます。	kamel rebuild my-integration

名前	サポート対象	説明	例
reset	■	現在の Camel K インストールをリセットします。	kamel reset
run	✓	OpenShift でインテグレーションを実行します。	kamel run MyIntegration.java
uninstall	■	OpenShift クラスターから Camel K をアンインストールします。 注意: Camel K のインストールおよびアンインストールには、OpenShift Camel K を使用することが推奨されます。	kamel uninstall
version	✓	Camel-K クライアントバージョンを表示します。	kamel version

関連情報

- 「[Installing Camel K](#)」を参照してください。

7.2. CAMEL K モードラインオプション

Camel K モードラインを使用すると、たとえば **kamel run MyIntegration.java** を使用して、起動時に実行される Camel K インテグレーションソースファイルに設定オプションを入力できます。詳細は、「[Running Camel K integrations using modeline](#)」を参照してください。

kamel run コマンドで利用可能なすべてのオプションは、モードコマンドラインオプションとして指定できます。

以下の表は、最も一般的に使用されるモードラインオプションの一部を表しています。

表7.2 Camel K モードラインオプション

オプション	詳細
build-property	ビルド時のプロパティファイルを追加します。 Syntax: [my-key=my-value file:/path/to/my-conf.properties]

オプション	詳細
config	<p>Configmap、シークレット、またはファイルからのランタイム設定を追加します。</p> <p>構文: [configmap secret file]:name[/key]</p> <ul style="list-style-type: none"> - name は、ローカルファイルパスまたは ConfigMap/シークレット名を表します。 - key は、フィルターされる ConfigMap/Secret キーを表します。
dependency	<p>外部ライブラリー (Maven 依存関係など) が含まれます。</p> <p>例: dependency=mvn:org.my:app:1.0</p>
env	<p>インテグレーションコンテナに環境変数を設定します。例: env=MY_ENV_VAR=my-value</p>
ラベル	<p>インテグレーションのラベルを追加します。例: label=my.company=hello</p>
name	<p>インテグレーション名を追加します。例: name=my-integration</p>
open-api	<p>OpenAPI v2 仕様を追加します。たとえば、open-api=path/to/my-hello-api.json です。</p>
profile	<p>デプロイメントに使用する Camel K トレイトプロファイルを設定します。例: openshift</p>
プロパティ	<p>ランタイムプロパティファイルを追加します。</p> <p>構文: [my-key=my-value file:/path/to/my-conf.properties])</p>
resource	<p>ConfigMap、シークレット、またはファイルからのランタイムリソースを追加します。</p> <p>構文: [configmap secret file]:name[/key][@path]</p> <ul style="list-style-type: none"> - name はローカルファイルパスまたは ConfigMap/Secret 名を表します。 - key (オプション) は、フィルターされる ConfigMap またはシークレットキーを表します。s - path (オプション) は宛先パスを表します。
trait	<p>トレイトで Camel K 機能またはコア機能を設定します。例: trait=service.enabled=false.</p>

