Red Hat Process Automation Manager 7.12

Red Hat Process Automation Manager でのプロセスサービスの開発

Last Updated: 2023-04-07

Red Hat Process Automation Manager 7.12 Red Hat Process Automation Manager でのプロセスサービスの開発

法律上の通知

Copyright © 2023 Red Hat, Inc.

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

本ガイドでは、Business Process Model and Notation (BPMN) 2.0 モデルを使用して、Red Hat Process Automation Manager でプロセスサービスおよびケース定義を開発する方法を説明します。本ガイドでは、プロセスおよびケース管理の概念およびオプションについても説明します。

目次

はじめに 8

多様性を受け入れるオープンソースの強化 9

パート I. BPMN モデルを使用したビジネスプロセスの作成 10

第1章 ビジネスプロセス 11

第2章 RED HAT PROCESS AUTOMATION MANAGER の BPMN モデラーおよび DMN モデラー 12
2.1. RED HAT PROCESS AUTOMATION MANAGER VS CODE 拡張機能バンドルのインストール 12
2.2. RED HAT PROCESS AUTOMATION MANAGER スタンドアロンのエディターの設定 13

第3章 MAVEN を使用した DMN モデルおよび BPMN モデルの作成および実行 17

第4章 BUSINESS PROCESS MODELING AND NOTATION バージョン 2.0 19
4.1. BPMN2 向けの RED HAT PROCESS AUTOMATION MANAGER サポート 20
4.2. プロセスデザイナーでの BPMN2 イベント 25
4.3. プロセスデザイナーでの BPMN2 タスク 34
4.4. プロセスデザイナーでの BPMN2 カスタムタスク 37
4.5. プロセスデザイナー内の BPMN2 サブプロセス 45
4.6. プロセスデザイナーでの BPMN2 ゲートウェイ 49
4.7. プロセスデザイナーでの BPMN2 接続オブジェクト 52
4.8. プロセスデザイナーでの BPMN2 スイムレーン 52
4.9. プロセスデザイナーでの BPMN2 アーティファクト 54

第5章 BUSINESS CENTRAL でのビジネスプロセスの作成 56
5.1. ビジネスルールタスクの作成 59
5.2. スクリプトタスクの作成 60
5.3. サービスタスクの作成 61
5.4. ユーザータスクの作成 66
5.5. プロセスデザイナーでの BPMN2 ユーザータスクのライフサイクル 69
5.6. プロセスデザイナーでの BPMN2 タスクパーミッションのマトリックス 69
5.7. ビジネスプロセスのコピーの作成 70
5.8. 要素のサイズを変更し、ズーム機能を使用したビジネスプロセスの表示 71
5.9. BUSINESS CENTRAL でのプロセスドキュメントの生成 72

第6章 VARIABLES 74
6.1. 変数タグ 74
6.2. グローバル変数の定義 76
6.3. プロセス変数の定義 77
6.4. ローカル変数の定義 78
6.5. プロセス変数値の編集 79

第7章 アクションスクリプト 80

第8章 タイマー 81
8.1. RED HAT PROCESS AUTOMATION MANAGER でサポート対象のタイマー 81
8.2. 遅延と期間を使用したタイマーの設定 81
8.3. ISO-8601 の日付形式を使用したタイマーの設定 81
8.4. プロセス変数でのタイマーの設定 82
8.5. 実行中のプロセスインスタンス内でのタイマー更新 82

第9章 制約 84

第10章 BUSINESS CENTRAL でのビジネスプロセスのデプロイ 85

第11章 BUSINESS CENTRAL でのビジネスプロセスの実行 **86**

第12章 ビジネスプロセスのテスト **88**
12.1. 外部サービスとの統合テスト 92

第13章 BUSINESS CENTRAL でのプロセス定義とプロセスインスタンス **94**
13.1. プロセス定義ページからのプロセスインスタンスの開始 95
13.2. プロセスインスタンスページからプロセスインスタンスの開始 95
13.3. XML でのプロセス定義 95

第14章 BUSINESS CENTRAL のフォーム **98**
14.1. フォームモデラー 98
14.2. BUSINESS CENTRAL でのプロセスフォームおよびタスクフォームの生成 99
14.3. BUSINESS CENTRAL での手動によるフォームの作成 100
14.4. フォームまたはプロセスでのドキュメントの添付 100

第15章 詳細にわたるプロセスの概念およびタスク **110**
15.1. ビジネスプロセスで DECISION MODEL AND NOTATION (DMN) サービスを呼び出す 110

第16章 関連情報 **116**

パート II. プロセスおよびタスクとの対話 **117**

第17章 BUSINESS CENTRAL のビジネスプロセス **118**
17.1. ナレッジワーカーのユーザー 118

第18章 BUSINESS CENTRAL でのナレッジワーカーのタスク **119**
18.1. タスクの開始 119
18.2. タスクの停止 119
18.3. タスクの委譲 119
18.4. タスクの要求 120
18.5. タスクのリリース 120
18.6. タスクの一括アクション 121

第19章 BUSINESS CENTRAL でのタスクのフィルターリング **124**
19.1. タスクリストの列の管理 124
19.2. 基本的なフィルターを使用したタスクのフィルターリング 124
19.3. 詳細フィルターを使用したタスクのフィルターリング 125
19.4. デフォルトのフィルターを使用したタスクの管理 125
19.5. 基本的なフィルターを使用したタスク変数の表示 126
19.6. 詳細フィルターを使用したタスク変数の表示 126

第20章 BUSINESS CENTRAL でのプロセスインスタンスのフィルターリング **128**
20.1. 基本フィルターを使用したプロセスインスタンスのフィルターリング 128
20.2. 詳細フィルターを使用したプロセスインスタンスのフィルターリング 129
20.3. デフォルトのフィルターを使用したプロセスインスタンスの管理 129
20.4. 基本フィルターを使用したプロセスインスタンス変数の表示 130
20.5. 詳細フィルターを使用したプロセスインスタンス変数の表示 130

第21章 タスク通知でのメールの設定 **132**

第22章 タスクの期日と優先順位の設定 **133**

第23章 タスクに対するコメントの表示および追加 **134**

第24章 タスクの履歴ログの表示 **135**

第25章 プロセスインスタンスの履歴ログの表示 **136**

パート III. BUSINESS CENTRAL でのビジネスプロセスの管理および監視 **137**

第26章 プロセスの監視 **138**

第27章 BUSINESS CENTRAL でのプロセス定義とプロセスインスタンス **139**
27.1. プロセス定義ページからのプロセスインスタンスの開始 140
27.2. プロセスインスタンスページからプロセスインスタンスの開始 140
27.3. BUSINESS CENTRAL でのプロセスドキュメントの生成 140

第28章 プロセスインスタンス管理 **142**
28.1. プロセスインスタンスのフィルターリング 143
28.2. カスタムのプロセスインスタンス一覧の作成 144
28.3. デフォルトのフィルターを使用したプロセスインスタンスの管理 144
28.4. 基本フィルターを使用したプロセスインスタンス変数の表示 145
28.5. 詳細フィルターを使用したプロセスインスタンス変数の表示 145
28.6. BUSINESS CENTRAL を使用したプロセスインスタンスの中止 145
28.7. BUSINESS CENTRAL からプロセスインスタンスのシグナル送信 146
28.8. 非同期シグナルイベント 146
28.9. プロセスインスタンスの操作 148

第29章 タスク管理 **150**
29.1. タスクのフィルターリング 151
29.2. カスタムタスクフィルターの作成 154
29.3. デフォルトのフィルターを使用したタスクの管理 156
29.4. 基本的なフィルターを使用したタスク変数の表示 157
29.5. 詳細フィルターを使用したタスク変数の表示 157
29.6. BUSINESS CENTRAL でのカスタムタスクの管理 158
29.7. ユーザータスク管理 161
29.8. タスクの一括アクション 162

第30章 ログデータの管理 **165**
30.1. 自動クリーンアップジョブの設定 165
30.2. 手動クリーンアップ 166
30.3. データベースからのログの削除 166
30.4. RED HAT PROCESS AUTOMATION MANAGER データベースでのカスタムクエリーの実行 168

第31章 実行エラー管理 **170**
31.1. BUSINESS CENTRAL でのプロセスの実行エラー表示 170
31.2. 実行エラーの管理 170
31.3. エラーのフィルターリング 171
31.4. 実行エラーの自動承認 174
31.5. エラー一覧のクリーンアップ 176

第32章 プロセスインスタンスの移行 **178**
32.1. プロセスインスタンスの移行サービスのインストール 178
32.2. 移行プランの作成 180
32.3. 移行プランの編集 182
32.4. 移行プランのエクスポート 182
32.5. 移行プランの実行 183
32.6. 移行プランの削除 184

パート IV. ケース管理の設計およびビルド **185**

第33章 ケース管理 **186**

第34章 CASE MANAGEMENT MODEL AND NOTATION (CMMN) **187**

第35章 ケースファイル 188
35.1. ケース ID 接頭辞の設定 188
35.2. ケース ID 式の設定 189

第36章 サブケース 192

第37章 アドホックおよび動的タスク 195

第38章 KIE SERVER REST API を使用してケースへの動的タスクおよびプロセスの追加 196
38.1. KIE SERVER REST API を使用した動的ユーザータスクの作成 197
38.2. KIE SERVER REST API を使用した動的サービスタスクの作成 198
38.3. KIE SERVER REST API を使用した動的サブプロセスの作成 200

第39章 コメント 202

第40章 CASE ROLES 203
40.1. ケー出力ルの作成 204
40.2. ロールの認証 205
40.3. ロールへのタスクの割り当て 206
40.4. SHOWCASE を使用してランタイム時にケースのロール割り当ての修正 208
40.5. REST API を使用してランタイム時にケースのロール割り当ての修正 209

第41章 ステージ 213
41.1. ステージの定義 213
41.2. ステージのアクティベーションおよび完了条件の設定 214
41.3. ステージへの動的タスクの追加 215

第42章 マイルストーン 217
42.1. マイルストーンの設定およびトリガー 217

第43章 変数タグ 220

第44章 ケースイベントリスナー 223

第45章 ケース管理のルール 225
45.1. ルールを使用したケースの前進 225

第46章 ケース管理のセキュリティー 230
46.1. ケース管理のセキュリティーの設定 230

第47章 ケースの終了 232
47.1. KIE SERVER REST API を使用したケースの終了 232
47.2. SHOWCASE アプリケーションを使用したケースの終了 233

第48章 ケースのキャンセルまたは破棄 234
48.1. データベースからケースログの削除 234

第49章 関連情報 237

パート V. ケース管理への SHOWCASE アプリケーションの使用 238

第50章 ケース管理 239

第51章 ケース管理の SHOWCASE アプリケーション 240
Showcase サポート 240

第52章 SHOWCASE アプリケーションのインストールおよびログイン 241

第53章 CASE ROLES 243

第54章 動的タスクおよびプロセスの開始 245

第55章 SHOWCASE アプリケーションで IT 発注ケースの開始 249

第56章 SHOWCASE と BUSINESS CENTRAL を使用した IT_ORDERSケースの完了 252

第57章 関連情報 258

パート VI. カスタムタスクとワークアイテムハンドラー 259

第58章 BUSINESS CENTRAL でのカスタムタスクの管理 260

第59章 ワークアイテムハンドラーのプロジェクト作成 264

第60章 ワークアイテムハンドラープロジェクトのカスタマイズ 267

第61章 ワークアイテム定義 269
61.1. @WID アノテーション 269
61.2. テキストファイル 272

第62章 カスタムタスクのデプロイ 275
62.1. BUSINESS CENTRAL のカスタムタスクリポジトリーの使用 275
62.2. JAR アーティファクトの BUSINESS CENTRAL へのアップロード 275
62.3. BUSINESS CENTRAL MAVEN リポジトリーへのワークアイテム定義の手動コピー 275

第63章 カスタムタスクの登録 277
63.1. BUSINESS CENTRAL でデプロイメント記述子を使用したカスタムタスクの登録 277
63.2. BUSINESS CENTRAL 外でのデプロイメント記述子を使用したカスタムタスクの登録 278

第64章 カスタムタスクの配置 280

パート VII. RED HAT PROCESS AUTOMATION MANAGER のプロセスエンジン 281

第65章 RED HAT PROCESS AUTOMATION MANAGER のプロセスエンジン 282

第66章 プロセスエンジンのコアエンジン API 285
66.1. KIE ベースおよび KIE セッション 285
66.2. ランタイムマネージャー 290
66.3. プロセスエンジンのサービス 305
66.4. プロセスエンジンのスレッド 343
66.5. プロセスエンジンの実行エラー 344
66.6. プロセスエンジンのイベントリスナー 346
66.7. プロセスエンジンの設定 353

第67章 プロセスエンジンの永続性およびトランザクション 357
67.1. プロセスランタイム状態の永続性 357
67.2. 永続的な監査ログ 358
67.3. プロセスエンジンのトランザクション 369
67.4. プロセスエンジンでの永続性の設定 373
67.5. RED HAT PROCESS AUTOMATION MANAGER の個別のデータベーススキーマにおけるプロセス変数の永続化 378

第68章 JAVA フレームワークとの統合 383
68.1. APACHE MAVEN との統合 383
68.2. CDI との統合 388
68.3. SPRING との統合 396
68.4. EJB との統合 404
68.5. OSGI との統合 410

付録A バージョン情報 411

付録B お問い合わせ先 412

はじめに

プロセスの開発者は、Red Hat Process Automation Manager を使用して、Business Process Model and Notation (BPMN) 2.0 モデルを使用してプロセスサービスおよびケース定義を開発することができます。BPMN プロセスモデルは、ビジネスの目標を達成するために必要なステップをグラフィックで表示します。BPMN の詳細は、Object Management Group (OMG) の Business Process Model and Notation 2.0 specification を参照してください。

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 プロパティー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。この取り組みにより、これらの変更は今後の複数のリリースに対して段階的に実施されます。詳細は、**弊社の CTO である Chris Wright のメッセージ**を参照してください。

パート I. BPMN モデルを使用したビジネスプロセスの作成

ビジネスプロセス開発者は、Red Hat Process Automation Manager の Business Central または VS Code の Red Hat Process Automation Manager BPMN モデラーを使用して、ビジネス要件に合ったビジネスプロセスを作成できます。本書では、ビジネスプロセスと、Red Hat Process Automation Manager のプロセスデザイナーを使用してビジネスプロセスを作成するための概念とオプションを説明します。また、Red Hat Process Automation Manager の BPMN2 要素についても説明しています。BPMN2 に関する詳細は、Business Process Model and Notation Version 2.0 仕様を参照してください。

前提条件

- Red Hat JBoss Enterprise Application Platform 7.4 がインストールされている。詳細情報は、**Red Hat JBoss Enterprise Application Platform 7.4 Installation Guide**を参照してください。
- Red Hat Process Automation Manager がインストールされ、KIE Server で設定されている。詳細は **Red Hat JBoss EAP 7.4 への Red Hat Process Automation Manager のインストールおよび設定** を参照してください。
- Red Hat Process Automation Manager が稼働し、**developer** ロールで Business Central にログインできる。詳細は、**Red Hat Process Automation Manager インストールの計画**を参照してください。

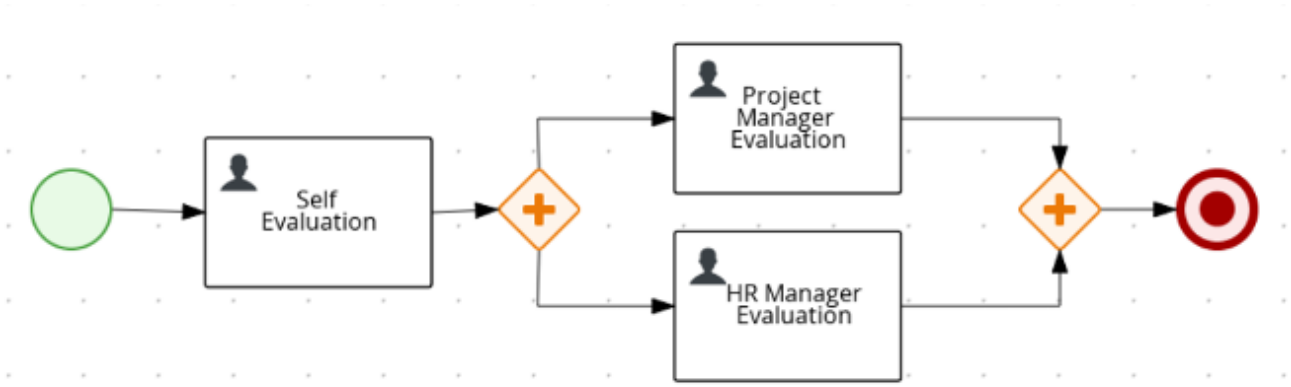
第1章 ビジネスプロセス

ビジネスプロセスとは、一連の手順の実行すべき順番を説明し、事前定義済みのノードや接続で設定されるダイアグラムのことです。各ノードは、プロセス内の手順 1 つを表し、接続はノード間の移動方法を指定します。

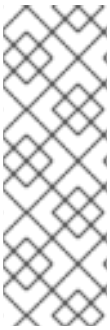
典型的なビジネスプロセスは、以下のコンポーネントで設定されています。

- プロセスの名前、インポート、変数などのグローバル要素で設定されるヘッダーセクション
- プロセスの一部であるすべての異なるノードを含むノードセクション
- これらのノードを相互にリンクしてフローチャートを作成する接続セクション

図1.1 ビジネスプロセス



Red Hat Process Automation Manager には、レガシーのプロセスデザイナーと、ビジネスプロセスダイアグラムを作成するための新しいプロセスデザイナーが含まれています。新しいプロセスデザイナーのレイアウトと機能セットは改善されており、開発が続けられています。レガシーのプロセスデザイナーのすべての機能が新しいプロセスデザイナーに完全に実装されるまで、両方のデザイナーを Business Central で使用できます。



注記

Business Central のレガシーのプロセスデザイナーは、Red Hat Process Automation Manager 7.12.0 で非推奨になりました。このツールは、今後の Red Hat Process Automation Manager リリースで削除予定です。そのため、レガシーのプロセスデザイナーには新しい機能拡張や機能は追加されません。新しいプロセスデザイナーを使用する場合は、お使いのプロセスを新しいデザイナーに移行し始めます。新しいプロセスデザイナーですべての新規プロセスを作成します。新規デザイナーへの移行に関する詳細は、**Business Central におけるプロジェクトの管理** を参照してください。

第2章 RED HAT PROCESS AUTOMATION MANAGER の BPMN モデラーおよび DMN モデラー

Red Hat Process Automation Manager は、グラフィカルモデラーを使用して Business Process Model and Notation (BPMN) プロセスモデルと、Decision Model and Notation (DMN) デシジョンモデルを設計するのに使用できる次の拡張機能またはアプリケーションを提供します。

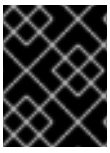
- **Business Central**: 関連する埋め込みデザイナーで、BPMN モデル、DMN モデル、およびテストシナリオファイルを表示および設計できます。
  Business Central を使用するには、Business Central を含む開発環境を設定してビジネスルールおよびプロセスを作成し、KIE Server を作成して、作成したビジネスルールとプロセスを実行およびテストします。

- **Red Hat Process Automation Manager VS Code 拡張** Visual Studio Code (VS Code) で BPMN モデル、DMN モデル、およびテストシナリオファイルを表示して、作成できるようにします。VS Code 拡張機能には VS Code 1.46.0 以降が必要です。
  Red Hat Process Automation Manager VS Code 拡張機能をインストールするには、VS Code で **Extensions** メニューオプションを選択して、**Red Hat Business Automation Bundle**拡張を検索し、インストールします。

- **スタンドアロン BPMN および DMN エディター**: Web アプリケーションに組み込まれた BPMN モデルおよび DMN モデルを表示して、作成できます。必要なファイルをダウンロードするには、NPM レジストリー から NPM アーティファクトを使用するか、**https://<YOUR_PAGE>/dmn/index.js** (DMN スタンドアロンのエディターライブラリーの場合)、または https://**<YOUR_PAGE>/bpmn/index.js** (BPMN スタンドアロンエディターライブラリーの場合) で JavaScript ファイルを直接ダウンロードします。

2.1. RED HAT PROCESS AUTOMATION MANAGER VS CODE 拡張機能バンドルのインストール

Red Hat Process Automation Manager は、**Red Hat Business Automation Bundle**VS Code 拡張機能を提供します。これにより、Decision Model and Notation (DMN) デシジョンモデル、Business Process Model and Notation (BPMN) 2.0 ビジネスプロセス、およびテストシナリオを VS Code で直接作成できます。VS Code は、新しいビジネスアプリケーションを開発するために推奨される統合開発環境 (IDE) です。Red Hat Process Automation Manager は、必要に応じて DMN サポートまたは BPMN サポートの **DMN Editor** および **BPMN Editor** VS Code 拡張機能をそれぞれ提供します。



重要

VS Code のエディターは、Business Central のエディターと部分的に互換性があり、VS Code では複数の Business Central 機能がサポートされていません。

前提条件

- VS Code の最新の安定版がインストールされている。

手順

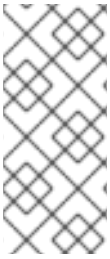
1. VS Code IDE で **Extensions** メニューオプションを選択し、DMN、BPMN、およびテストシナリオファイルのサポートに対して **Red Hat Business Automation Bundle**を検索します。
   DMN ファイルまたは BPMN ファイルだけをサポートする場合は、**DMN Editor** または **BPMN Editor** 拡張機能をそれぞれ検索することもできます。

2. **Red Hat Business Automation Bundle**拡張機能が VS Code に表示される際に、これを選択し、**Install** をクリックします。

3. VS Code エディターの動作を最適化するには、拡張機能のインストールが完了した後に、VS Code のインスタンスを再度読み込み、閉じるか、または再起動します。

VS Code 拡張バンドルをインストールした後、VS Code で開くか作成するすべての **.dmn** ファイル、**.bpmn** ファイル、または **.bpmn2** ファイルがグラフィカルモデルとして自動的に表示されます。さらに、開くまたは作成する **.scesim** ファイルは、ビジネスデシジョンの機能をテストするテーブルテストシナリオモデルとして自動的に表示されます。

DMN、BPMN、またはテストシナリオモデラーが DMN、BPMN、またはテストシナリオファイルの XML ソースのみを開き、エラーメッセージが表示される場合は、報告されたエラーおよびモデルファイルを確認して、すべての要素が正しく定義されていることを確認します。



注記

新しい DMN モデルまたは BPMN モデルの場合は、Web ブラウザーで **dmn.new** または **bpmn.new** を入力して、オンラインモデラーで DMN モデルまたは BPMN モデルを設計することもできます。モデルの作成が終了したら、オンラインモデラーページで **Download** をクリックして、DMN ファイルまたは BPMN ファイルを VS Code の Red Hat Process Automation Manager プロジェクトにインポートできます。

2.2. RED HAT PROCESS AUTOMATION MANAGER スタンドアロンのエディターの設定

Red Hat Process Automation Manager は、自己完結型のライブラリーに分散されたスタンドアロンのエディターを提供し、エディターごとにオールインワンの JavaScript ファイルを提供します。JavaScript ファイルは、包括的な API を使用してエディターを設定および制御します。

以下の方法を使用して、スタンドアロンのエディターをインストールします。

- 各 JavaScript ファイルを手動でダウンロード
- NPM パッケージの使用

手順

1. 以下の方法のいずれかを使用して、スタンドアロンのエディターをインストールします。
各 JavaScript ファイルを手動でダウンロード: この方法の場合は、以下の手順に従います。

   a. JavaScript ファイルをダウンロードします。

   b. ダウンロードした Javascript ファイルをホスト型アプリケーションに追加します。

   c. 以下の **<script>** タグを HTML ページに追加します。

   DMN エディターの HTML ページのスクリプトタグ

   ```
   <script src="https://<YOUR_PAGE>/dmn/index.js"></script>
   ```

   BPMN エディターの HTML ページのスクリプトタグ

   ```
   <script src="https://<YOUR_PAGE>/bpmn/index.js"></script>
   ```

NPM パッケージの使用: この方法の場合は、以下の手順に従います。

a. NPM パッケージを **package.json** ファイルに追加します。

NPM パッケージの追加

```
npm install @kogito-tooling/kie-editors-standalone
```

b. 各エディターライブラリーを **TypeScript** ファイルにインポートします。

各エディターのインポート

```
import * as DmnEditor from "@kogito-tooling/kie-editors-standalone/dist/dmn"
import * as BpmnEditor from "@kogito-tooling/kie-editors-standalone/dist/bpmn"
```

2. スタンドアロンのエディターをインストールしたら、以下の例のように提供されたエディター API を使用して必要なエディターを開き、DMN エディターを開きます。API はエディターごとに同じです。

DMN スタンドアロンのエディターを開く

```
const editor = DmnEditor.open({
  container: document.getElementById("dmn-editor-container"),
  initialContent: Promise.resolve(""),
  readOnly: false,
  origin: "",
  resources: new Map([
    [
      "MyIncludedModel.dmn",
      {
        contentType: "text",
        content: Promise.resolve("")
      }
    ]
  ])
});
```

エディター API で以下のパラメーターを使用します。

表2.1 パラメーターの例

パラメーター	説明
container	エディターが追加される HTML 要素。

パラメーター	説明
initialContent	DMN モデルのコンテンツへの Promise。以下の例のように、このパラメーターは空にすることができます。 ● **Promise.resolve("")** ● **Promise.resolve("<DIAGRAM_CONTENT_DIRECTLY_HERE>")** ● **fetch("MyDmnModel.dmn").then(content ⇒ content.text())**
readonly (任意)	エディターでの変更を許可します。コンテンツの編集を許可する場合は **false** (デフォルト)、エディターで読み取り専用モードの場合は **true** に設定します。
origin (任意)	リポジトリーの起点。デフォルト値は **window.location.origin** です。
resources (任意)	エディターのリソースのマッピング。たとえば、このパラメーターを使用して、BPMN エディターの DMN エディターまたは作業アイテム定義に含まれるモデルを提供します。マップの各エントリーには、リソース名と、**content-type** (**text** または **binary**) および **content** (**initialContent** パラメーターと同様) で設定されるオブジェクトが含まれています。

返されるオブジェクトには、エディターの操作に必要なメソッドが含まれます。

表2.2 返されたオブジェクトメソッド

メソッド	説明
getContent(): Promise<string>	エディターのコンテンツを含む promise を返します。
setContent(path: string, content: string): void	エディターの内容を設定します。
getPreview(): Promise<string>	現在のダイアグラムの SVG 文字列が含まれる promise を返します。
subscribeToContentChanges(callback: (isDirty: boolean) ⇒ void): (isDirty: boolean) ⇒ void	エディターでコンテンツを変更し、サブスクライブ解除に使用されるのと同じコールバックを返す際に呼び出されるコールバックを設定します。
unsubscribeToContentChanges(callback: (isDirty: boolean) ⇒ void): void	エディターでコンテンツが変更される際に渡されたコールバックのサブスクライブを解除します。

メソッド	説明
markAsSaved(): void	エディターの内容が保存されることを示すエディターの状態をリセットします。また、コンテンツの変更に関連するサブスクライブされたコールバックをアクティベートします。
undo(): void	エディターの最後の変更を元に戻します。また、コンテンツの変更に関連するサブスクライブされたコールバックをアクティベートします。
redo(): void	エディターで、最後に元に戻した変更をやり直します。また、コンテンツの変更に関連するサブスクライブされたコールバックをアクティベートします。
close(): void	エディターを終了します。
getElementPosition(selector: string): Promise<Rect>	要素をキャンバスまたはビデオコンポーネント内に置いた場合に、標準のクエリーセレクターを拡張する方法を提供します。**selector** パラメーターは、**Canvas:::MySquare**、**Video:::PresenterHand** などの **<PROVIDER>:::<SELECT>** 形式に従う必要があります。このメソッドは、要素の位置を表す **Rect** を返します。
envelopeApi: MessageBusClientApi<KogitoEditorEnvelopeApi>	これは高度なエディター API です。高度なエディター API の詳細は、MessageBusClientApi および KogitoEditorEnvelopeApi を参照してください。

第3章 MAVEN を使用した DMN モデルおよび BPMN モデルの作成および実行

Maven アーキタイプを使用して、Business Central ではなく Red Hat Process Automation Manager VS Code 拡張機能を使用して、VS Code で DMN モデルおよび BPMN モデルを開発できます。その後、必要に応じて、Business Central で、アーキタイプを Red Hat Process Automation Manager のデシジョンサービスおよびプロセスサービスに統合できます。DMN モデルおよび BPMN モデルを開発する方法は、Red Hat Process Automation Manager VS Code 拡張機能を使用して新規ビジネスアプリケーションを構築する場合に便利です。

手順

1. コマンドターミナルで、新しい Red Hat Process Automation Manager プロジェクトを保存するローカルディレクトリーに移動します。
2. Maven アーキタイプを使用して、定義されたフォルダー内のプロジェクトを生成するには、次のコマンドを入力します。

   Maven アーキタイプを使用したプロジェクトの生成

   ```
   mvn archetype:generate \
       -DarchetypeGroupId=org.kie \
       -DarchetypeArtifactId=kie-kjar-archetype \
       -DarchetypeVersion=7.59.0.Final-redhat-00006
   ```

   このコマンドにより、必要な依存関係で Maven プロジェクトが生成され、ビジネスアプリケーションを構築するのに必要なディレクトリーとファイルが生成されます。プロジェクトの開発には、バージョン管理システム Git) 推奨を使用することができます。

   同じディレクトリーに複数のプロジェクトを生成する場合は、直前のコマンドに **-DgroupId=<groupid> -DartifactId=<artifactId>** を追加して、生成されたビジネスアプリケーションの **artifactId** および **groupId** を指定できます。
3. VS Code IDE で **File** をクリックし、**Open Folder** を選択し、直前のコマンドを使用して生成されたディレクトリーに移動します。
4. 最初のアセットを作成する前に、ビジネスアプリケーションのパッケージ (例: **org.kie.businessapp**) を設定し、以下のパスにそれぞれのディレクトリーを作成します。
   - **PROJECT_HOME/src/main/java**
   - **PROJECT_HOME/src/main/resources**
   - **PROJECT_HOME/src/test/resources**

   たとえば、**org.kie.businessapp** パッケージの **PROJECT_HOME/src/main/java/org/kie/businessapp** を作成できます。
5. VS Code を使用して、ビジネスアプリケーションにアセットを作成します。以下の方法で、Red Hat Process Automation Manager VS Code 拡張機能がサポートするアセットを作成できます。
   - ビジネスプロセスを作成するには、**PROJECT_HOME/src/main/java/org/kie/businessapp** ディレクトリーに、**.bpmn** または **.bpmn2** の新規ファイルを作成します (例: **Process.bpmn**)。

- DMN モデルを作成するには、**PROJECT_HOME/src/main/java/org/kie/businessapp** ディレクトリーに、**.dmn** の新規ファイルを作成します (例: **AgeDecision.dmn**)。
- テストシナリオシミュレーションモデルを作成するには、**PROJECT_HOME/src/main/java/org/kie/businessapp** ディレクトリーに、**.scesim** の新規ファイルを作成します (例: **TestAgeScenario.scesim**)。

6. Maven アーキタイプでアセットを作成したら、コマンドラインで (**pom.xml** がある) プロジェクトのルートディレクトリーに移動し、以下のコマンドを実行してプロジェクトのナレッジ JAR (KJAR) を構築します。

```
mvn clean install
```

ビルドに失敗したら、コマンドラインのエラーメッセージに記載されている問題に対応し、ビルドに成功するまでプロジェクトの妥当性確認を行います。ただし、ビルドに成功すると、**PROJECT_HOME/target** ディレクトリーでビジネスアプリケーションのアーティファクトを確認できます。

注記

mvn clean install コマンドを使用して、開発中の主要な変更ごとにプロジェクトを検証します。

REST API を使用して実行中の KIE Server に、ビジネスアプリケーションの生成されたナレッジ JAR (KJAR) をデプロイできます。プロセスの REST API の使用方法は、KIE API を使用した Red Hat Process Automation Manager の操作 を参照してください。

第4章 BUSINESS PROCESS MODELING AND NOTATION バージョン 2.0

Business Process Modeling and Notation バージョン 2.0 (BPMN2) 仕様は、ビジネスプロセスを描画表現するための標準や要素の実行セマンティクスを定義し、XML 形式でのプロセス定義を指定する Object Management Group (OMG) 仕様です。

プロセスは、定義するか、プロセス定義を使用して判断されます。また、プロセスはナレッジベースに存在しており、ID で識別されます。

表4.1 一般的なプロセスプロパティー

ラベル	説明
名前	プロセスの名前を入力します。
ドキュメント	プロセスを記述します。このフィールドのテキストはプロセスドキュメントに含まれます (該当する場合)。
ID	このプロセスの識別子 (**orderItems** など) を入力します。
Package	Red Hat Process Automation Manager プロジェクトにおけるこのプロセスのパッケージの場所を入力します (例: **org.acme**)。
ProcessType	プロセスがパブリックまたはプライベートであるかを指定します。ただし、現時点ではサポートされていません。
バージョン	プロセスのアーティファクトバージョンを入力します。
アドホック	このプロセスがアドホックサブプロセスである場合は、このオプションを選択します。
Process Instance Description	プロセスの目的の説明を入力します。
Imports	クリックして **Imports** ウィンドウを開き、プロセスに必要なデータタイプクラスを追加します。あるいは、すでに定義済みのデータタイプを選択することができます。
実行可能	このオプションを選択して、プロセスを Red Hat Process Automation Manager プロジェクトの実行可能な部分にします。
SLA 期日	サービスレベルアグリーメント (SLA) の有効期限の日付を入力します。
プロセス変数	プロセスのプロセス変数を追加します。プロセス変数は、特定のプロセスインスタンス内で表示されます。プロセス変数はプロセスの作成時に初期化され、プロセスの完了時に破棄されます。変数タグを使用すると、変数の動作をより細かく制御できます。たとえば、変数に **required** タグを付けたり、**readonly** タグを付けたりできます。変数タグの詳細は、6章*Variables* を参照してください。

ラベル	説明
Metadata Attributes	メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を追加します。
グローバル変数	プロセスにグローバル変数を追加します。グローバル変数は、プロジェクトのすべてのプロセスインスタンスとアセットに表示されます。グローバル変数は通常、ビジネスルールおよび制約によって使用され、ルールまたは制約によって動的に作成されます。

プロセスは、一連のモデリング要素のコンテナーです。これには、フローオブジェクトとフローを使用してビジネスプロセスまたはそのパーツの実行ワークフローを指定する要素が含まれています。各プロセスには、独自の BPMN2 ダイアグラムがあります。Red Hat Process Automation Manager には、BPMN2 ダイアグラムを作成するための新規プロセスデザイナーのほか、**.bpmn2** 拡張を使用して以前の BPMN2 ダイアグラムを開くレガシープロセスデザイナーが含まれます。新規プロセスデザイナーでは、レイアウトと機能セットが向上し、今後も開発が続けられる予定です。デフォルトでは、新規ダイアグラムは新規プロセスデザイナーで作成されます。

4.1. BPMN2 向けの RED HAT PROCESS AUTOMATION MANAGER サポート

Red Hat Process Automation Manager では、BPMN 2.0 標準を使用して、ビジネスプロセスのモデル化が可能です。Red Hat Process Automation Manager を使用してこれらのビジネスプロセスを実行、管理、監視することができます。包括的な BPMN 2.0 仕様には、コレオグラフィーやコラボレーションなどの項目を表現する方法が含まれます。ただし、Red Hat Process Automation Manager は、実行可能なプロセスの指定に使用可能な仕様の部分のみを使用します。これには、BPMN2 仕様の共通の実行可能なサブクラスに定義されている全要素および属性だけでなく、追加の要素や属性も含まれます。

以下の表は、BPMN2 要素がレガシーのプロセスデザイナーでサポートされているか、レガシーおよび新規プロセスデザイナーでサポートされているか、またはサポートされていないかを示すアイコン一覧を示しています。

表4.2 サポートの状態を示すアイコン

キー	説明
	レガシーおよび新規プロセスデザイナーでのサポート
	レガシーのプロセスデザイナーでのみサポート
	サポートなし

アイコンのない要素は、BPMN2 仕様には存在しません。

表4.3 BPMN2 の Catch イベント

要素名	開始	中間
なし		
メッセージ		
Timer		
エラー		
エスカレーション		
キャンセル		
補正		
条件付き		
リンク		
シグナル		
複数		

要素名	開始	中間
並列多重	✗	✗

表4.4 BPMN2 送出および中断なしイベント

要素名	送出		中断なし	
	終了	中間	開始	中間
なし	✓			
メッセージ	✓	✓	✓	✓
Timer			✓	✓
エラー	✓			
エスカレーション	✓	✓	✓	✓
キャンセル	✗	✗		✗
補正	✓	✓		
条件付き			✓	✓
リンク		✓		

要素名	送出		中断なし	
シグナル	✓	✓	✓	✓
終了	✓			
複数	✗	✗	✗	✗
並列多重			✗	✗

表4.5 BPMN2 要素

要素タイプ	要素	対応
タスク	ビジネスルール	✓
	スクリプト	✓
	ユーザータスク	✓
	サービスタスク	✓
複数のインスタンスサブプロセスを含むサブプロセス	組み込み	✓
	アドホック	✓

要素タイプ	要素	対応
	再利用可能	
	イベント	
ゲートウェイ	含む	
	排他的	
	並行	
	イベントベース	
	複雑	
オブジェクトの接続	シーケンスフロー	
	関連フロー	
スイムレーン	スイムレーン	
アーティファクト	グループ	

要素タイプ	要素	対応
	テキストのアノテーション	✓
	データオブジェクト	✓

BPMN2 の背景およびアプリケーションに関する詳細は、OMG Business Process Model and Notation (BPMN) Version 2.0 を参照してください。

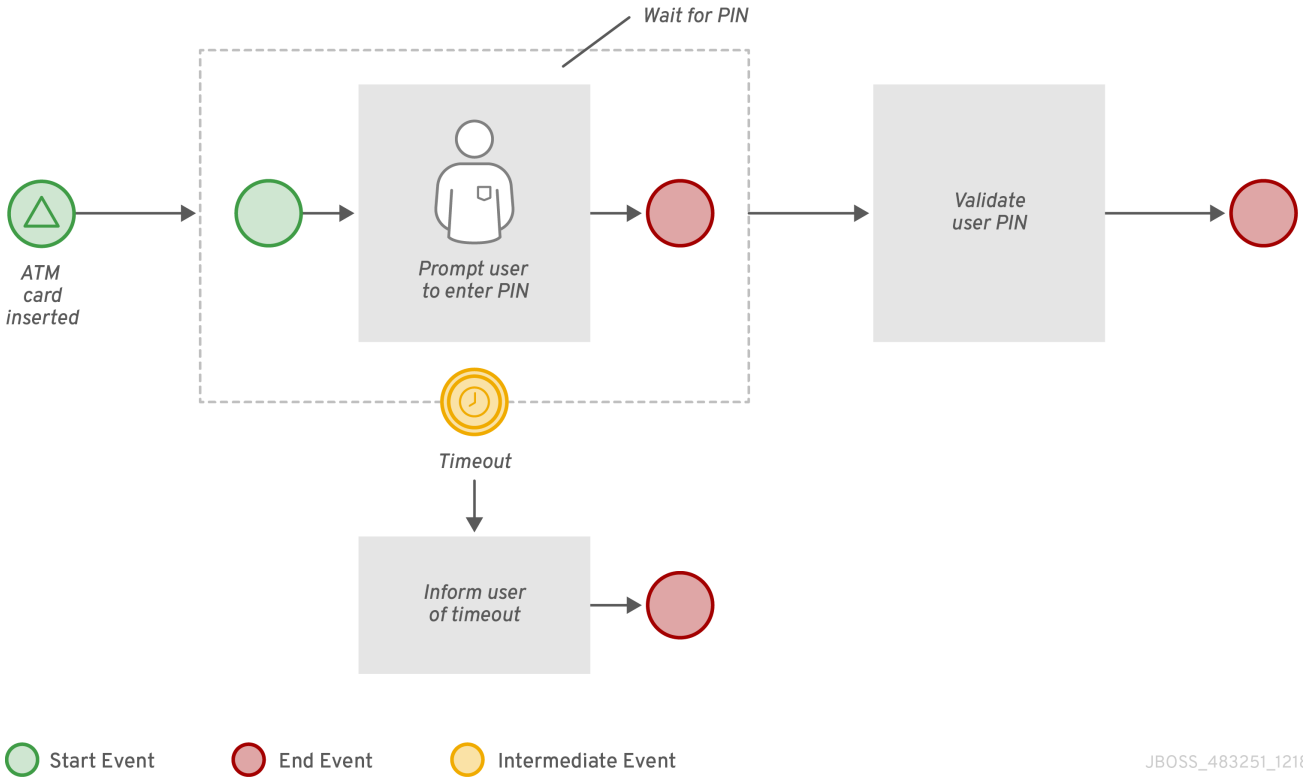
4.2. プロセスデザイナーでの BPMN2 イベント

イベントとは、ビジネスプロセスに発生する内容のことです。BPMN2 では、3 つのイベントカテゴリーをサポートします。

- 開始
- 終了
- 中間

開始イベントはイベントトリガーをキャッチし、終了イベントはイベントトリガーを出力します。中間イベントは、イベントトリガーをキャッチおよび出力できます。

以下のビジネスプロセスダイアグラムは、イベントの例を示しています。



この例では、以下のイベントが発生します。

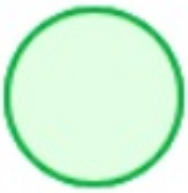
- シグナルを受信すると、ATM カード挿入のシグナル開始イベントがトリガーされます。
- タイムアウトの中間イベントは、タイマートリガーをもとにした割り込みイベントです。つまり、タイマーイベントがトリガーされると、暗証番号待ちのサブプロセスがキャンセルされるという意味です。
- プロセスへの入力に応じて、Validate User Pin タスクに関連付けられた終了イベント、または Inform User of Timeout タスクに関連付けられた終了イベントが、プロセスを終了します。

4.2.1. 開始イベント

開始イベントを使用して、ビジネスプロセスの始端を示します。開始イベントには、受信シーケンスフローを割り当てることができず、外向きシーケンスフローだけを割り当てる必要があります。なし (none) 開始イベントは、トップレベルプロセス、埋め込みサブプロセス、呼び出し可能サブプロセス、イベントサブプロセスで使用できます。

なし (none) 開始イベントの例外を除けば、開始イベントはすべてキャッチイベントです。たとえば、シグナルの開始イベントは、参照のシグナル (イベントトリガー) を受け取った場合にのみプロセスを開始します。イベントサブプロセスの開始イベントを割り込みまたは割り込みなしイベントに設定できます。イベントサブプロセスに対する割り込みありの開始イベントでは、包含プロセスまたは親プロセスの実行を停止または中断します。割り込みなしの開始イベントは、包含プロセスまたは親プロセスの実行を停止したり、中断したりしません。

表4.6 開始イベント

開始イベントタイプ	トップレベル	サブプロセス	
		割り込み	割り込みなし
なし			
条件付き			
補正			

開始イベントタイプ	トップレベル	サブプロセス	
エラー			
エスカレーション			
メッセージ			
シグナル			
タイマー			

なし

なしの開始イベントは、トリガー条件のない開始イベントです。プロセスまたはサブプロセスには、最大1つのなし開始イベントを含めることができます。このイベントは、デフォルトでプロセスまたはサブプロセスの開始によりトリガーされ、外向きフローがすぐに実行されます。

サブプロセスでなし開始イベントを使用すると、親プロセスからサブプロセスに、プロセスフローの実行が移動し、なし開始イベントがトリガーされます。これは、トークン (プロセスフローの内の現在の場所) が親プロセスからサブプロセスのアクティビティーに渡され、サブプロセスのなし開始イベントが独自のトークンを生成します。

条件付き

条件付きの開始イベントは、ブール型の条件定義を含む開始イベントです。条件が最初に **false** と評価され、次に **true** に評価された場合に実行がトリガーされます。プロセスの実行は、開始イベントがインスタンス化された後に条件が **true** と評価された場合にのみ開始されます。

プロセスには、複数の条件開始イベントを含めることができます。

補正

補正開始イベントは、補正中間イベントのターゲットアクティビティーとしてサブプロセスを使用した場合に、補正イベントのサブプロセスを開始するのに使用します。

エラー

プロセスまたはサブプロセスには、複数のエラー開始イベントを含めることができます。特定の **ErrorRef** プロパティーを含むエラーオブジェクトを受け取った場合に、この開始イベントが開始します。エラーオブジェクトは、エラーの終了イベントで生成可能です。これは、プロセスの終端が不正であることを示します。エラーの開始イベントが含まれるプロセスインスタンスは、該当するエラーオブジェクトを受け取ると実行が開始します。エラー開始イベントは、エラーオブジェクトの受け取り直後に実行し、外向きフローが実行します。

エスカレーション

エスカレーション開始イベントは、特定のエスカレーションコードを含むエスカレーションによりトリガーされる開始イベントです。プロセスには、複数のエスカレーション開始イベントを含めることができます。エスカレーション開始イベントが含まれるプロセスインスタンスは、定義されているエスカレーションオブジェクトを受け取ると、実行を開始します。プロセスがインスタンス化され、直後にエスカレーション開始イベントが実行し、外向きフローが実行します。

メッセージ

プロセスまたはイベントのサブプロセスには、複数のメッセージ開始イベントを含めることができます。これらのイベントは、通常特定のメッセージにより開始します。メッセージ開始イベントが含まれるプロセスインスタンスの実行は、該当のメッセージを受け取った後に、このイベントからのみ開始します。メッセージの受け取り後、プロセスはインスタンス化され、メッセージ開始イベントが即座に実行します (外向きフローが実行します)。

メッセージは、要素なしなど、任意の数のプロセスおよびプロセス要素により消費可能であるため、1 つのメッセージで複数のメッセージ開始イベントをトリガーできるため、複数のプロセスがインスタンス化されます。

シグナル

シグナル開始イベントは、特定のシグナルコードを含むシグナルによりトリガーされます。プロセスには、複数のシグナル開始イベントを含めることができます。シグナル開始イベントは、インスタンスが該当のシグナルを受け取った後にのみ、プロセスインスタンス内で実行します。その後に、シグナル開始イベントが実行され、外向きフローが実行されます。

タイマー

タイマー開始イベントは、タイミングのメカニズムを含む開始イベントです。プロセスには、複数のタイマー開始イベントを含めることができます。タイマー開始イベントは、タイミングのメカニズムが適用された後に、プロセスの開始時に発生します。

サブプロセスでタイマー開始イベントを使用すると、プロセスフローの実行が親プロセスからサブプロセスに移動し、タイマー開始イベントが発生します。親サブプロセスアクティビティーからトークンを取得し、サブプロセスのタイマー開始イベントが開始し、タイマーが発生するまで待機します。タイミングの定義で指定した時間が経過したら、外向きフローが実行されます。

4.2.2. 中間イベント

中間イベントは、ビジネスプロセスのフローを駆動します。中間イベントは、ビジネスプロセスの実行中にイベントをキャッチまたは出力するときに使用します。中間イベントは、開始イベントと終了イベントの間に配置され、サブプロセス、人間のタスクなどのアクティビティーの境界にキャッチイベントとして使用することもできます。BPMN モデラーでは、プロセスインスタンスの詳細にアクセスするために追加のプロセスで使用される境界イベントの **Data Output and Assignments** フィールドにデータの出力を設定できます。補正イベントは、データ出力変数の設定機能をサポートしないことに注意してください。

たとえば、境界イベントに以下のデータ出力変数を設定できます。

- **nodeInstance**: 境界イベントがトリガーされたときに追加のプロセスで使用するノードインスタンスの詳細を取得します。
- **signal**: シグナルの名前を取得します。
- **event**: イベントの詳細を取得します。
- **WorkItem**: ワークアイテムの詳細を取得します。この変数は、ワークアイテムまたはユーザータスクに設定できます。

境界キャッチイベントは、割り込みまたは割り込みなしとして設定できます。割り込みありの境界キャッチイベントは、バインドされているアクティビティーを取り消しますが、割り込みなしのイベントは取り消しません。

中間イベントは、プロセス実行時に発生する特定の状況を処理します。このような状況が中間イベントのトリガーになります。プロセスには、外向きフローを 1 つ含む中間イベントを、アクティビティーの境界に配置できます。

アクティビティーの実行時にイベントが発生した場合には、このイベントにより、外向きフローへの実行が発生します。1 つのアクティビティーに、複数の境界中間イベントが含まれる可能性があります。境界中間イベントで、アクティビティーからの必要な動作によって、以下のいずれかの中間イベントタイプを使用できる点に注意してください。

- 割り込みあり: アクティビティーの実行は中断され、中間イベントの実行が発生します。
- 割り込みなし: 中間イベントがトリガーされ、アクティビティーの実行が続行します。

表4.7 中間イベント

中間イベントタイプ	キャッチ	境界		送出
		割り込み	割り込みなし	
メッセージ				

中間イベントタイプ	キャッチ	境界		送出
タイマー				
エラー				
シグナル				
条件付き				
補正				
エスカレーション				

中間イベントタイプ	キャッチ	境界	送出
リンク			

メッセージ

メッセージの中間イベントは、メッセージオブジェクトを管理可能にする中間イベントです。以下のイベントのいずれかを使用します。

- 出力メッセージの中間イベントでは、定義したプロパティーをもとにメッセージオブジェクトを作成します。
- キャッチメッセージの中間イベントは、定義したプロパティーを使用してメッセージオブジェクトがないかリッスンします。

タイマー

タイマー中間イベントでは、ワークフローの実行を遅延させたり、定期的に発生させたりできます。このイベントは、指定した期間が経過したら1回または複数回、トリガーできるタイマーを表します。タイマー中間イベントが開始したら、タイマー条件 (定義した時間) がチェックされ、外向きフローが実行されます。タイマー中間イベントがプロセスワークフローに配置されている場合には、内向きフローが1つと、外向きフローが1つ含まれます。内向きフローがイベントに移動すると、これが実行されます。タイマー中間イベントがアクティビティー境界に配置されている場合は、アクティビティーの実行と同時に、この実行がトリガーされます。

包含のプロセスインスタンスを完了するか、中断するなど、タイマー要素がキャンセルされると、タイマーがキャンセルされます。

条件付き

条件の中間イベントは、ブール型の条件がトリガーとして含まれる中間イベントです。このイベントは、条件で **true** と判断され、外向きフローが実行された場合に、さらにワークフロー実行をトリガーします。

このイベントは、**Expression** プロパティーを定義する必要があります。条件の中間イベントがプロセスワークフローに配置されている場合は、内向きフロー1つ、外向きフロー1つ含まれ、内向きフローがイベントに移動したときに、実行が開始されます。条件の中間イベントがアクティビティー境界に配置されている場合は、アクティビティーの実行時に、この実行が発生します。イベントが割り込みなしの場合は、条件が **true** の場合は継続して、このイベントが発生します。

シグナル

シグナルの中間イベントでは、シグナルオブジェクトを生成または消費できます。以下のオプションのいずれかを使用してください。

- 出力シグナルの中間イベントは、定義したプロパティーをもとにシグナルオブジェクトを生成します。
- キャッチシグナルの中間イベントは、定義したプロパティーを使用してシグナルオブジェクトがないかリッスンします。

エラー

エラーの中間イベントは、アクティビティー境界でのみ使用可能な中間イベントです。このイベントでは、プロセスが、該当するアクティビティー内のエラー終了イベントに反応できるようになります。このアクティビティーは、アトミックにしないでください。アクティビティーが、エラー終了イベントで完了し、対応の **ErrorCode** プロパティーでエラーオブジェクトを生成した場合は、エラーの中間イベントがこのエラーオブジェクトをキャッチして、実行が外向きフローに進みます。

補正

補正中間イベントは、トランザクションサブプロセスのアクティビティーに接続している境界イベントです。補正終了イベントまたはキャンセル終了イベントで、このイベントを終了できます。補正中間イベントは、補正アクティビティーに接続しているフローと関連付ける必要があります。

境界補正の仲介イベントに関連付けられているアクティビティーは、トランザクションサブプロセスが補正終了イベントで終了した場合に実行します。この実行は、対応のフローで続行します。

エスカレーション

エスカレーション中間イベントは、エスカレーションオブジェクトを生成または消費できる中間イベントです。イベント要素が実行すべきアクションに合わせて、以下のオプションのいずれかを使用する必要があります。

- 出力エスカレーションの中間イベントは、定義したプロパティーをもとにエスカレーションオブジェクトを生成します。
- キャッチエスカレーションの中間イベントは、定義したプロパティーを使用してエスカレーションオブジェクトがないかリッスンします。

リンク

リンク中間イベントは、プロセスに追加のロジックを追加しなくても、プロセスのダイアグラムを簡単に理解できるようにする中間イベントです。リンク中間イベントは単一のプロセスレベルに制限されます。たとえば、リンク中間イベントは、サブプロセスと親プロセスを接続できません。

以下のオプションのいずれかを使用してください。

- 発生するリンク中間イベントは、定義されたプロパティーに基づいてリンクオブジェクトを生成します。
- 取得するリンク中間イベントは、定義されたプロパティーでリンクオブジェクトをリッスンします。

4.2.3. 終了イベント

終了イベントは、ビジネスプロセスを終了するために使用され、発信シーケンスフローがない場合があります。ビジネスプロセスには複数の終了イベントが存在する場合があります。なし、および中断終了イベント以外の終了イベントはすべて出力イベントです。

終了イベントは、ビジネスプロセスの完了を示します。終了イベントは、特定のワークフローを終了するノードです。このイベントには、1 つまたは複数の内向きシーケンスフローがあり、外向きフローはありません。

プロセスには最低でも 1 つの終了イベントが含まれている必要があります。

ランタイム中は、終了イベントでプロセスワークフローを終了します。終了イベントは、そのイベントに到達したワークフローのみ終了できます。終了イベントタイプによってはプロセスインスタンス内の全ワークフローを終了できます。

表4.8 終了イベント

終了イベント	アイコン
なし	
メッセージ	
シグナル	
エラー	
補正	
エスカレーション	
終了	

なし

なし終了イベントは、他に特別な動作がプロセスの終端に関連付けられていないことを示します。

メッセージ

フローがメッセージの終了イベントに入ると、このフローは終了し、終了イベントがプロパティーに定義されているようにメッセージを生成します。

シグナル

出力シグナルの終了イベントは、プロセスまたはサブプロセスフローの終了に使用します。実行フローがこの要素に入ると、実行フローが終了し、**SignalRef** プロパティーで特定されたシグナルを生成します。

エラー

出力エラーの終了イベントは、内向きワークフローを完了します。つまり、内向きのトークンを消費し、エラーオブジェクトを生成します。プロセスまたはサブプロセスで他に実行されているワークフローは、影響を受けません。

補正

補正終了イベントは、トランザクションのサブプロセスを終了し、サブプロセスアクティビティーの境界に接続されている補正中間イベントで定義した補正を発生させるのに使用します。

エスカレーション

エスカレーション終了イベントは、内向きワークフローを終了します。これは、内向きのトークンを消費して、プロパティーに定義されているようにエスカレーションシグナルを生成し、エスカレーションプロセスを発生させます。

終了

中断終了イベントは、指定したプロセスインスタンス内の全実行フローを終了します。実行中のアクティビティーはキャンセルされます。サブプロセスインスタンスは、中断終了イベントに到達した場合は中断されます。

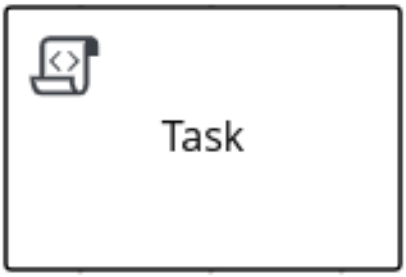
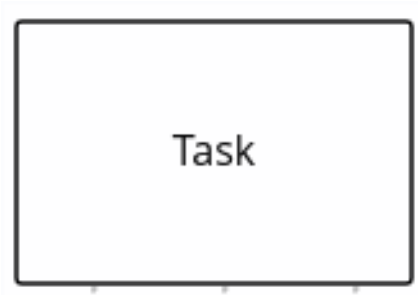
4.3. プロセスデザイナーでの BPMN2 タスク

タスクは、プロセスモデルに定義されている自動アクティビティーで、プロセスフロー内で最小の作業単位です。BPMN 2 仕様に定義されているタスクタイプで、Red Hat Process Automation Manager のプロセスデザイナーパレットで利用できるのは以下のとおりです。

- ビジネスルールタスク
- スクリプトタスク
- ユーザータスク
- サービスタスク
- タスクなし

表4.9 タスク

ビジネスルールタスク	

スクリプトタスク	
ユーザータスク	
サービスタスク	
タスクなし	

さらに、BPMN2 仕様では、カスタムタスクの作成が可能になります。カスタムタスクの詳細は、「プロセスデザイナーでの BPMN2 カスタムタスク」を参照してください。

ビジネスルールタスク

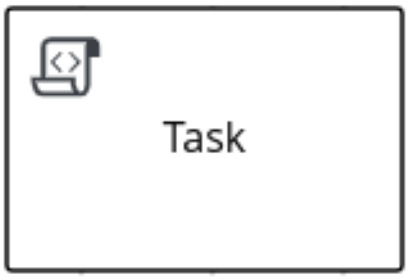
ビジネスルールタスクは、DMN モデルまたはルールフローグループを使用して、意思決定を行う方法を定義します。

プロセスが DMN モデルで定義したビジネスルールタスクに到達したら、プロセスエンジンが、入力された内容を使用して DMN モデルを実行します。

プロセスがルールフローグループで定義したビジネスルールタスクに到達したら、プロセスエンジンは、定義済みのルールフローグループでルールの実行を開始します。ルールフローグループにアクティブなルールがない場合、実行は次の要素に移動します。ルールフローグループの実行中は、アクティブなルールフローグループに所属するアクティベーションは、他のルールで変更されるため、新たにアジェンダに追加できます。

スクリプトタスク

スクリプトタスクは、プロセス実行中に実行されるスクリプトを表します。



関連付けられたスクリプトは、プロセス変数やグローバル変数にアクセスできます。スクリプトタスクを使用する前に以下の一覧をレビューしてください。

- プロセスでは詳細にわたる実装の内容は回避してください。スクリプトタスクは、変数の操作に使用できますが、より複雑な操作をモデル化する場合にサービスタスクまたはカスタムタスクの使用を検討してください。
- スクリプトがすぐに実行されることを確認してください。すぐに実行しない場合は、非同期サービスタスクを使用してください。
- スクリプトタスクを使用して外部のサービスの問い合わせを回避してください。サービスタスクを使用して、外部サービスとの通信をモデル化します。
- スクリプトで例外が出力されないようにしてください。ランタイムの例外はスクリプト内などで、キャッチし、管理するか、プロセス内で処理できるシグナルまたはエラーに変換する必要があります。

実行中にスクリプトタスクに到達したら、スクリプトが実行され、外向きフローに移動します。

ユーザータスク

ユーザータスクは、システムで自動的に実行できないプロセスワークフローに含まれるタスクであるため、ユーザー (人間)、つまりアクターの介入が必要です。

実行時に、ユーザータスク要素は、1 つ以上のアクターのタスク一覧に表示されるタスクとしてインスタンス化されます。ユーザータスク要素で **Groups** 属性が定義されている場合に、このユーザータスク要素は、グループに所属する全ユーザー一覧に表示されます。このグループに所属するメンバーは誰でもタスクを要求できます。

タスクがクレームされると、他のユーザーのタスク一覧からこのタスクは消失します。

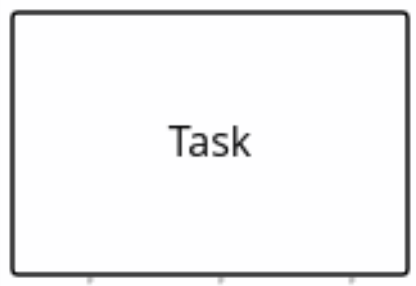
ユーザータスクは、ドメイン固有のタスクとして実装され、カスタムタスクのベースとして機能します。

サービスタスク

サービスタスクは、人間の介入を必要としないタスクです。これらは、外部のソフトウェアサービスによって自動的に完了します。

タスクなし

アクティベーション時に完了するタスクはありません。これは概念モデルのみです。タスクなしは、IT システムによって実際に実行されることはありません。

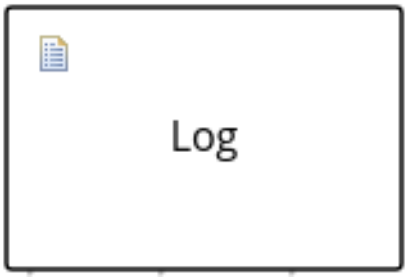


4.4. プロセスデザイナーでの BPMN2 カスタムタスク

BPMN2 仕様は、**bpmn2:task** 要素を拡張してソフトウェア実装でカスタムタスクを作成する機能をサポートします。標準の BPMN タスクと同様に、カスタムタスクは、ビジネスプロセスモデルで完了するアクションを特定しますが、これには、特定のタイプ (REST、電子メール、または Web サービス) の外部サービスとの互換性や、プロセス (マイルストーン) 内のチェックポイント動作などの特化した機能も含まれます。

Red Hat Process Automation Manager は、BPMN モデラーパレットの **Custom Tasks** の下に、以下の事前定義済みのカスタムタスクを提供します。

表4.10 サポートされるカスタムタスク

カスタムタスクのタイプ	カスタムタスクのノード
Rest	
電子メール	
ログ	
WebService	
マイルストーン	

カスタムタスクのタイプ	カスタムタスクのノード
DecisionTask	
BusinessRuleTask	
KafkaPublishMessages	

Business Central でのカスタムタスクを有効または無効にする方法は、58章*Business Central でのカスタムタスクの管理* を参照してください。

BPMN モデラーでは、一部のカスタムタスクに対して以下の一般的なプロパティーを設定できます。

表4.11 一般的なカスタムタスクプロパティー

ラベル	説明
名前	タスクの名前を識別します。タスクノードをダブルクリックして名前を編集することもできます。
ドキュメント	タスクについて記述します。このフィールドのテキストはプロセスドキュメントに含まれます (該当する場合)。
非同期です	このタスクが非同期で呼び出されるかどうかを決定します。

ラベル	説明
アドホックの自動開始	これが自動的に開始されるアドホックタスクであるかどうかを決定します。このオプションを使用すると、タスクは、シグナルイベントにより開始するのではなく、プロセスが作成されたときに自動的に開始します。
開始時アクション	タスクの開始時のアクションを指定する Java、JavaScript、または MVEL スクリプトを定義します。
終了時アクション	タスクの終了時にアクションを指定する Java、JavaScript、または MVEL スクリプトを定義します。
SLA 期日	サービスレベルアグリーメント (SLA) の有効期限が切れるまでの期間 (文字列タイプ) を指定します。期間は、日数、分、秒、およびミリ秒で指定できます。たとえば、**SLA due date** フィールドの **1m** 値は 1 分を示します。
割当	タスクのデータの入力と出力を定義します。
Metadata Attributes	メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を定義します。 **メタデータ属性**は、BPMN 図に新しい **metaData** 拡張を可能にし、タスク全体の動作を変更します。

Rest

Rest カスタムタスクは、リモートの RESTful サービスを呼び出すか、プロセスから HTTP 要求を実行するために使用されます。

Rest カスタムタスクを使用するには、プロセスモデラーに URL、HTTP メソッド、および認証情報を設定します。プロセスが Rest カスタムタスクに到達したら、HTTP 要求を生成し、応答を文字列として返します。

Properties パネルで **Assignments** をクリックし、**REST Data I/O** ウィンドウを開きます。**REST Data I/O** ウィンドウで、必要に応じてデータの入力と出力を設定できます。たとえば、Rest カスタムタスクを実行するには、**Data Inputs and Assignments** フィールドで以下のデータ入力を実行します。

- **URL**: REST サービスのエンドポイント URL。この属性は必須です。
- **Method**: 呼び出されたエンドポイントのメソッド (例: **GET** および **POST** など)。デフォルト値は **GET** です。

- **ContentType**: データ送信時のデータタイプ。この属性は、**POST** および **PUT** 要求では必須です。
- **ContentTypeCharset**: **ContentType** に設定された文字セット。
- **Content**: 送信するデータ。この属性は後方互換性に対応し、代わりに **ContentData** 属性を使用します。
- **ContentData**: 送信するデータ。この属性は、**POST** および **PUT** 要求では必須です。
- **ConnectTimeout**: 接続タイムアウト (秒単位)。デフォルト値は 60000 ミリ秒です。入力値はミリ秒単位で指定する必要があります。
- **ReadTimeout**: 応答のタイムアウト (秒単位)。デフォルト値は 60000 ミリ秒です。入力値はミリ秒単位で指定する必要があります。
- **Username**: 認証用のユーザー名。
- **Password**: 認証用のパスワード。
- **AuthUrl**: 認証を処理する URL。
- **AuthType**: 認証を処理する URL のタイプ。
- **HandleResponseErrors** (オプション): 応答コードが失敗した場合に、エラーを発生させるようにハンドラーに指示 (2XX は除く)。
- **ResultClass**: 応答がアンマーシャリングされるクラスの有効な名前。指定されない場合は、未加工の応答が文字列形式で返されます。
- **AcceptHeader**: Accept ヘッダーの値。
- **AcceptCharset**: Accept ヘッダーの文字セット。
- **Headers**: REST 呼び出しへ渡すヘッダー (例: **content-type=text/html** など)。

以下のデータの出力を **Data Outputs and Assignments** に追加し、タスク実行の出力を保存できます。

- **Result**: 残りのカスタムタスクの出力変数 (オブジェクトタイプ)。

電子メール

プロセスからの電子メールの送信には、電子メールのカスタムタスクが使用されます。これには、関連する電子メールボディーが含まれます。

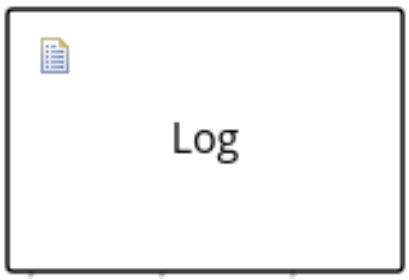
電子メールのカスタムタスクがアクティブになると、電子メールデータがタスクのデータ入力プロパティーに割り当てられます。関連する電子メールが送信されると、電子メールのカスタムタスクを完了します。

Properties パネルで Assignments をクリックし、Email Data I/O ウィンドウを開きます。Email Data I/O ウィンドウで、必要に応じてデータ入力を設定できます。たとえば、電子メールカスタムタスクを実行するには、Data Inputs and Assignments フィールドで以下のデータ入力を実行します。

- Body: 電子メールのボディー
- From: 送信者の電子メールアドレス。
- Subject: 電子メールの件名。
- To: 受信者の電子メールアドレス。セミコロン (;) で区切られた複数の電子メールアドレスを指定できます。
- Template (任意): 電子メールのボディーを生成するテンプレート。**Template** 属性は、入力した場合には **Body** パラメーターをオーバーライドします。
- Reply-To: 返信メッセージの送信先となる電子メールアドレス。
- Cc: カーボンコピーの受信者の電子メールアドレス。セミコロン (;) で区切られた複数の電子メールアドレスを指定できます。
- Bcc: ブラインドカーボンコピーの受信者の電子メールアドレス。セミコロン (;) で区切られた複数の電子メールアドレスを指定できます。
- Attachments: 電子メールと共に送信する添付ファイル。
- Debug: デバッグロギングを有効にするフラグ。

ログ

ログカスタムタスクは、プロセスからメッセージをログに記録する際に使用されます。ビジネスプロセスがログカスタムタスクに到達すると、メッセージデータがデータ入力プロパティーに割り当てられます。



関連付けられたメッセージがログに記録されると、ログカスタムタスクを完了します。Properties パネルで Assignments をクリックし、Log Data I/O ウィンドウを開きます。Log Data I/O ウィンドウで、必要に応じてデータ入力を設定できます。たとえば、ログカスタムタスクを実行するには、Data Inputs and Assignments フィールドで以下のデータ入力を実行します。

- Message: プロセスからのログメッセージ。

WebService

プロセスから Web サービスを呼び出すために使用される Web サービスのカスタムタスク。このカスタムタスクは、文字列として保存された Web サービス応答を使用して Web サービスクライアントとして機能します。

プロセスから Web サービスを呼び出すには、正しいタスクタイプを使用する必要があります。**Properties** パネルで **Assignments** をクリックし、**WS Data I/O** ウィンドウを開きます。**WS Data I/O** ウィンドウで、必要に応じてデータ入力と出力を設定できます。たとえば、Web サービスタスクを実行するには、**Data Inputs and Assignments** フィールドで以下のデータ入力を実行します。

- **Endpoint**: 呼び出す Web サービスのエンドポイントの場所。
- **Interface**: **Weather** などのサービスの名前。
- **Mode**: **SYNC**、**ASYNC**、または **ONEWAY** などのサービスのモード。
- **Namespace**: Web サービスの名前空間 (例: **http://ws.cdyne.com/WeatherWS/**)。
- **Operation**: 呼び出し用のメソッド名。
- **Parameter**: 操作に送信するオブジェクトまたは配列。
- **Url**: Web サービスの URL (**http://wsf.cdyne.com/WeatherWS/Weather.asmx?WSDL** など)。

以下のデータの出力を **Data Outputs and Assignments** に追加し、タスク実行の出力を保存できます。

- **Result**: Web サービスタスクの出力変数 (オブジェクトタイプ)。

マイルストーン

マイルストーンは、プロセスインスタンス内の 1 つの達成地点を表します。マイルストーンを使用して、特定のイベントにフラグを付けて他のタスクをトリガーするか、プロセスの進捗を追跡できます。

マイルストーンは、重要業績評価指標 (KPI) の追跡や、完了前のタスクの特定に役立ちます。マイルストーンは、プロセスのステージの最後に発生したり、他のマイルストーンを達成した結果として発生したりする場合もあります。

マイルストーンは、プロセスの実行中に以下の状態に到達できます。

- **Active**: マイルストーンの条件がマイルストーンノードに対して定義されているが、条件がまだ満たされていない。
- **Completed**: マイルストーンの条件が満たされ (該当する場合)、マイルストーンが達成されたため、このプロセスは次のタスクに進むか、または終了することができる。

Properties パネルで **Assignments** をクリックし、**Milestone Data I/O** ウィンドウを開きます。**Milestone Data I/O** ウィンドウで、必要に応じてデータ入力を設定できます。たとえば、マイルストーンを実行するには、**Data Inputs and Assignments** フィールドで以下のデータ入力を実行します。

- **Condition**: マイルストーンが満たす条件。たとえば、プロセス変数を使用する Java 式 (文字列データタイプ) を入力します。

DecisionTask

デシジョンタスクを使用して、DMN ダイアグラムを実行し、プロセスからデシジョンエンジンサービスを呼び出します。デフォルトでは、デシジョンタスクは DMN デシジョンにマッピングします。

デシジョンタスクを使用して、プロセスで運用上の意思決定を行うことができます。デシジョンタスクは、プロセスにおいて下す必要のある主要なデシジョンを特定する際に役立ちます。

Properties パネルで **Assignments** をクリックし、**Decision Task Data I/O** ウィンドウを開きます。**Decision Task Data I/O** ウィンドウで、必要に応じてデータ入力を設定できます。たとえば、デシジョンタスクを実行するには、**Data Inputs and Assignments** フィールドで以下のデータ入力を実行します。

- **Decision**: プロセスで下すデシジョン。
- **Language**: デシジョンタスクの言語。デフォルトは DMN です。
- **Model**: DMN モデル名。
- **Namespace**: DMN モデルの名前空間。

BusinessRuleTask

ビジネスルールタスクを使用して、DRL ルールを評価し、プロセスからデシジョンエンジンサービスを呼び出します。デフォルトでは、ビジネスルールタスクは DRL ルールにマッピングします。

ビジネスルールタスクを使用して、ビジネスプロセスで主要なビジネスルールを評価できます。**Properties** パネルで **Assignments** をクリックし、**Business Rule Task Data I/O** ウィンドウを開きます。**Business Rule Task Data I/O** ウィンドウで、必要に応じてデータ入力を設定できます。たとえば、ビジネスルールタスクを実行するには、**Data Inputs and Assignments** フィールドで以下のデータ入力を実行します。

- **KieSessionName**: KIE セッションの名前。

- **KieSessionType**: KIE セッションのタイプ。
- **Language**: ビジネスルールタスクの言語。デフォルトは DRL です。

KafkaPublishMessages

Kafka ワークアイテムは、イベントを Kafka トピックに送信するために使用されます。このカスタムタスクには、Kafka プロデューサーを使用して特定の Kafka サーバートピックにメッセージを送信するワークアイテムハンドラーが含まれます。たとえば、**KafkaPublishMessages** タスクは、プロセスから Kafka トピックにメッセージを公開します。

Properties パネルで **Assignments** をクリックし、**KafkaPublishMessages Data I/O** ウィンドウを開きます。**KafkaPublishMessages Data I/O** ウィンドウで、必要に応じてデータ入力と出力を設定できます。たとえば、Kafka のワークアイテムを実行するには、**Data Inputs and Assignments** フィールドで以下のデータ入力を実行します。

- **Key**: 送信される Kafka メッセージのキー。
- **Topic**: Kafka トピックの名前。
- **Value**: 送信される Kafka メッセージの値

以下のデータの出力を **Data Outputs and Assignments** に追加し、ワークアイテム実行の出力を保存できます。

- **Result**: ワークアイテムの出力変数 (文字列タイプ)。

ビジネスプロセスでの **KafkaPublishMessages** の詳細は Red Hat Process Automation Manager と Red Hat AMQ Streams の統合 を参照してください。

4.5. プロセスデザイナー内の BPMN2 サブプロセス

サブプロセスは、複数のノードが含まれるアクティビティーです。サブプロセスにメインのプロセスの一部を埋め込むことができます。また、サブプロセスには変数定義を追加できます。これらの変数は、サブプロセス内の全ノードにアクセスできます。

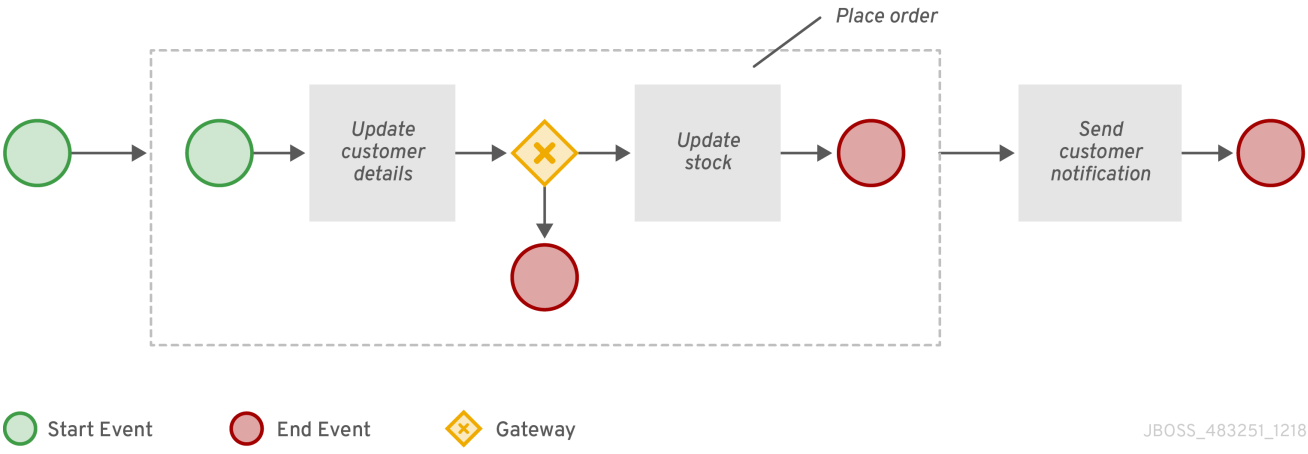
サブプロセスには、内向きの接続と外向きの接続を少なくとも 1 つずつ含める必要があります。サブプロセス内の中断終了イベントは、サブプロセスインスタンスを終了しますが、親プロセスインスタンスを自動的に終了することはありません。サブプロセスからアクティブな要素がなくなると、サブプロセスが終了します。

Red Hat Process Automation Manager では、以下のサブプロセスのタイプがサポートされます。

- 埋め込みサブプロセス: 親プロセス実行の一部で、親プロセスデータを共有し、独自のローカルサブプロセス変数を宣言します。
- アドホックサブプロセス: 厳密な要素実行の順番がないサブプロセス。

- 再利用可能なサブプロセス: 親プロセスから独立しているサブプロセス。
- イベントサブプロセス: 開始イベントまたはタイマーでのみトリガーされるサブプロセス。
- マルチインスタンスサブプロセス: 複数回インスタンス化されるサブプロセス。

以下の例では、発注のサブプロセスは、その注文を受けるのに十分な在庫があるかを確認し、注文できた場合に在庫情報を更新します。注文の可否により、メインのプロセス経由で、顧客に通知が行きます。



組み込みサブプロセス

埋め込みサブプロセスは、プロセスの一部をカプセル化します。このサブプロセスには、開始イベントと、最低でも 1 つの終了イベントが含まれている必要があります。この要素を使用して、このコンテナー内の全要素にアクセスできるローカルのサブプロセス変数を定義できます。

アドホックサブプロセス。

アドホックサブプロセスまたはプロセスには、埋め込みの内部アクティビティーが複数含まれ、通常のプロセスルーティングに比べて、より柔軟な順番で実行することを目的としています。通常のプロセスとは違い、アドホックサブプロセスには、開始イベントから終了イベントまでといった、完全な体系化された BPMN2 ダイアグラムの説明は含まれません。代わりに、アクティビティー、シーケンスフロー、ゲートウェイ、中間イベントのみが含まれます。また、アドホックサブプロセスには、データオブジェクトやデータの関連付けも含めることができます。アドホックサブプロセス内のアクティビティーでは、内向きおよび外向きのシーケンスフローを含める必要はありません。ただし、その中に含まれているアクティビティー間のシーケンスフローを指定できます。使用する場合、シーケンスフローは通常のプロセスと同じ順序の制約を提供します。意味を持つには、中間イベントに送信シーケンスフローを含める必要があり、アドホックサブプロセスがアクティブなときに複数回トリガーされる可能性があります。

再利用可能なサブプロセス

再利用可能なサブプロセスは、親プロセス内で縮小表示されます。再利用可能なサブプロセスを設定するには、再利用可能なサブプロセスを選択し、 をクリックして **Implementation/Execution** を展開します。以下のプロパティーを設定します。

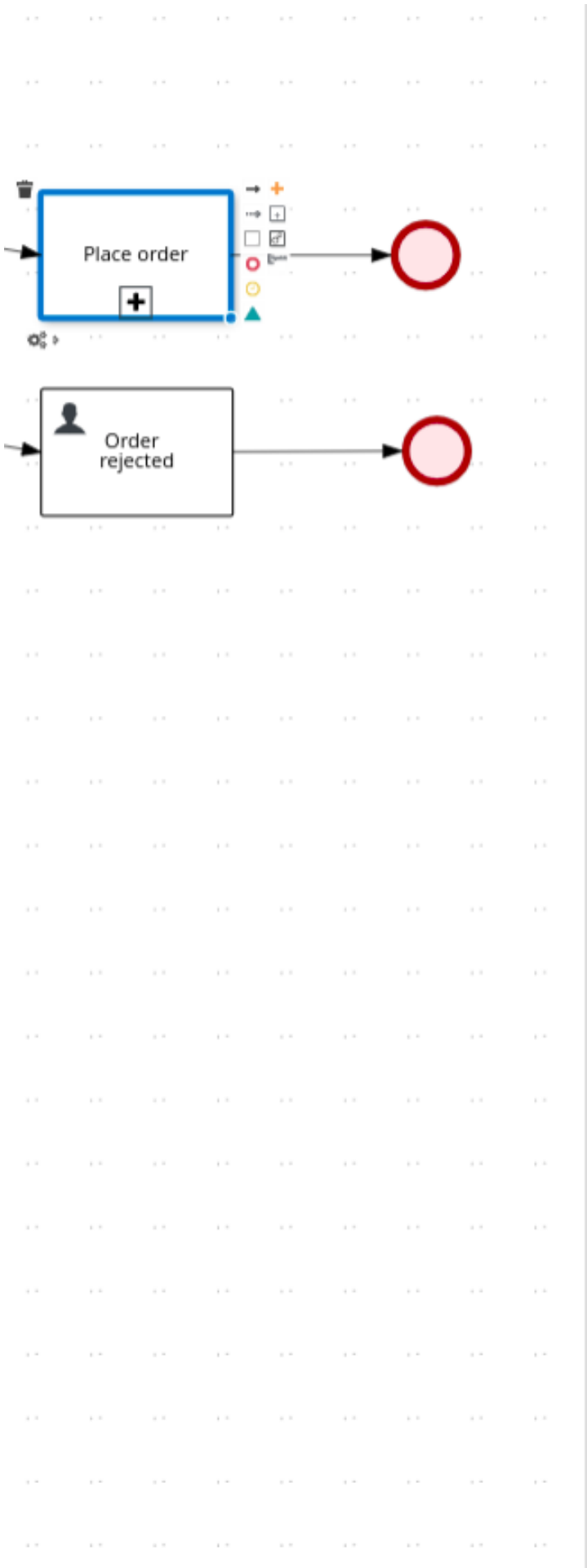
- **呼び出された要素**: アクティビティーにより呼び出してインスタンス化するサブプロセスの ID。
- **独立**: 選択されている場合は、サブプロセスが独立プロセスとして開始します。選択されていない場合は、親プロセスが中断されると、アクティブなサブプロセスが取り消されます。
- **親の強制終了**: この項目が選択されていて、呼び出したプロセスインスタンスの実行中にエラー

が発生した場合は、再利用可能な従属サブプロセスで、親プロセスを中断できます。たとえば、サブプロセスを呼び出そうとしてエラーが発生した時や、サブプロセスインスタンスを中断する時などです。このプロパティーは、**Independent** プロパティーが選択されている時しか、表示されません。以下のルールが適用されます。

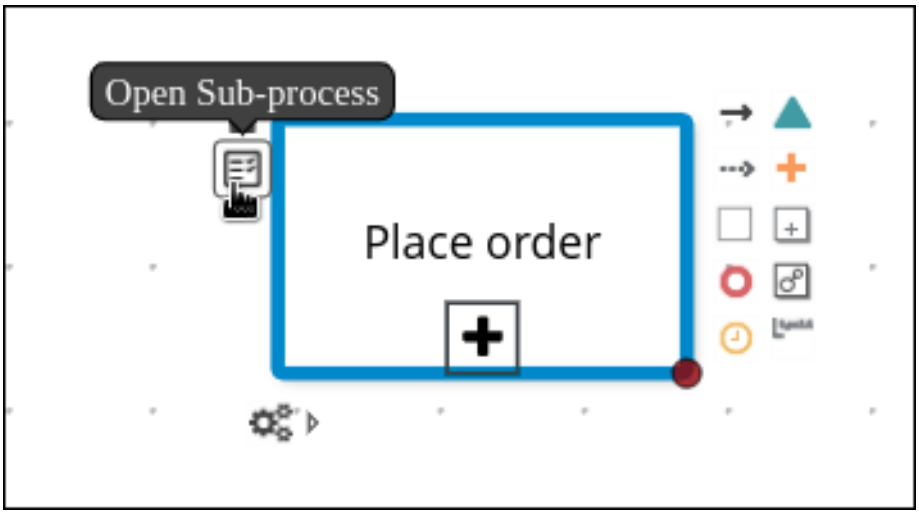
  - 再利用可能なサブプロセスが独立している場合に、**親の強制終了** は使用できません。
  - 再利用可能なサブプロセスが独立していない場合 (従属してい場合) は、**親の強制終了** を使用できます。

- **完了するまで待機**: 選択されている場合には、呼び出されたサブプロセスインスタンスが終了するまで、指定の **終了時アクション** は実行しません。親プロセスの実行は、**終了時アクション** が完了するまで継続されます。このプロパティーはデフォルトで選択されています (**true** に設定されています)。
- **非同期**: タスクを非同期で呼び出して、すぐに実行できないようにする場合に選択します。
- **複数インスタンス**: サブプロセス要素を指定の回数実行する場合に選択します。選択されている場合には、以下のオプションを使用できます。
  - **MI 実行モード**: 複数インスタンスを並行して実行するか、順次実行するかを指定します。**Sequential** に設定されている場合には、以前のインスタンスが完了するまで新規インスタンスは作成されません。
  - **MI コレクション入力**: 新規インスタンスを作成する要素コレクションを表現する変数を選択します。サブプロセスは、コレクションのサイズと同じ回数だけ、インスタンス化されます。
  - **MI データ入力**: コレクションで、選択された要素が含まれる変数名を指定します。この変数は、コレクション内の要素にアクセスする時に使用します。
  - **MI コレクション出力**: マルチインスタンスノードの出力を収集する要素コレクションを表現する任意の変数。
  - **MI データ出力**: **MI コレクション出力** プロパティーで選択した出力コレクションに追加する変数名を指定します。
  - **MI 完了条件 (mvel)**: 指定した複数のインスタンスノードを完了できるかどうかを確認するために、完了済みのインスタンスを評価する MVEL 式。**true** と評価された場合には、残りのインスタンスはすべて取り消されます。
- **開始時アクション**: タスクの開始時のアクションを指定する Java または MVEL スクリプト。
- **終了時アクション**: タスクの終了時のアクションを指定する Java または MVEL スクリプト。
- **SLA 期日**: サービスレベルアグリーメント (SLA) の有効期限の日付。期間は、日数、分、秒、およびミリ秒で指定できます。たとえば、SLA due date フィールドの **1m** 値は 1 分を示します。
- **メタデータ属性**: メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を追加します。

図4.1 再利用可能なサブプロセスのプロパティー



Business Central の新規エディターでサブプロセスを開くには、メインプロセスの **Place order** タスク、**Open Sub-process** タスクアイコンの順にクリックします。



イベントサブプロセス

イベントサブプロセスは、開始イベントがトリガーされるとアクティブになります。親プロセスのコンテキストを中断するか、並行して実行できます。

外向きまたは内向きの接続では、イベントまたはタイマーがサブプロセスをトリガーできます。サブプロセスは、通常のコントロールフローの一部ではありません。自己完結型ではありますが、バインドされているプロセスのコンテキストで実行されます。

プロセスフロー内のイベントサブプロセスを使用して、主なプロセスフロー外で発生するイベントを処理します。たとえば、飛行機の予約時には、以下の 2 つのイベントが発生する可能性があります。

- 予約の取り消し (割り込み)
- 予約ステータスの確認 (割り込みなし)

イベントのサブプロセスを使用して、これらのイベントの両方をモデル化します。

複数インスタンスサブプロセス

マルチインスタンスサブプロセスは、実行がトリガーされると、複数回インスタンス化されます。インスタンスは順次作成されるか、並行して作成されます。順次モードを設定すると、前のインスタンスが完了した後のみ、新しいサブプロセスインスタンスが作成されます。ただし、並列モードを設定すると、全サブプロセスインスタンスが一度に作成されます。

マルチインスタンスサブプロセスには、内向きの接続 1 つと、外向きの接続 1 つが含まれます。

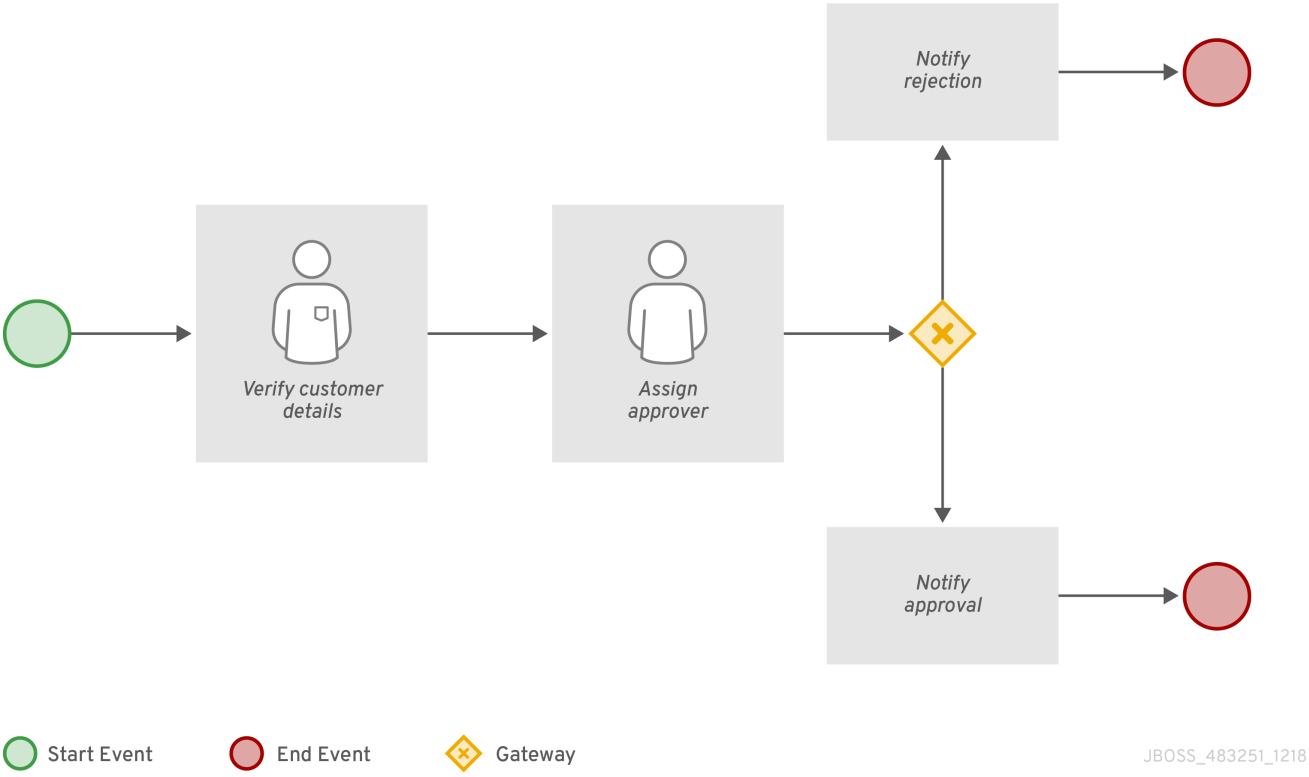
4.6. プロセスデザイナーでの BPMN2 ゲートウェイ

ゲートウェイは、ゲートメカニズムと呼ばれる条件セットを使用して、ワークフロー内にブランチを作成するか、ワークフロー内のブランチを同期するのに使用します。BPMN2 は、2 種類のゲートウェイをサポートします。

- 収束ゲートウェイ。複数のフローを 1 つにマージします。
- 分岐ゲートウェイ。1 つのフローを複数のフローに分割します。

1 つのゲートウェイに、複数の内向きと外向きフローを割り当てることはできません。

以下のビジネスプロセスダイアグラムでは、XOR ゲートウェイは、条件が true と評価された内向きフローのみを評価します。



この例では、顧客の詳細がユーザーにより検証され、ユーザーが承認できるようにプロセスが割り当てられます。承認されると、承認通知がユーザーに送信されます。要求イベントが却下された場合は、却下通知がユーザーに送信されます。

表4.12 ゲートウェイ要素

要素タイプ	アイコン
排他的論理和 (XOR)	
含む	
並行	

要素タイプ	アイコン
イベント	

排他的

排他的な分岐ゲートウェイでは、最初の内向きフローで条件が True と評価されたもののみが選択されます。収束ゲートウェイでは、トリガーされた内向きのフローごとに、次のノードがトリガーされます。

このゲートウェイは、1つだけ外向きフローをトリガーします。フローの条件が True と評価され、優先順位が **最も低い** 数字が選択されます。

重要

実行時に、最低でも1つの外向きフローが True と評価されるようにしてください。そうでないと、プロセスインスタンスは、ランタイムの例外で中断されます。

収束ゲートウェイでは、ワークフローブランチが、ゲートウェイに到達すると同時に外向きフローに進むことができます。内向きフローの1つがゲートウェイをトリガーすると、ワークフローはゲートウェイの外向きフローに進みます。複数の内向きフローからトリガーされた場合は、トリガーごとに次のノードをトリガーします。

含む

包含的な分岐ゲートウェイでは、内向きフローが選択され、さらに True と評価された外向きフローすべてが選択されます。優先順位の数値が低い接続は、高い接続よりも先にトリガーされます。優先順位は評価されますが、BPMN2 仕様では優先順位の順番は保証されません。ワークフローで **priority** 属性に依存しないようにしてください。

重要

実行時に、最低でも1つの外向きフローが True と評価されるようにしてください。そうでないと、プロセスインスタンスは、ランタイムの例外で中断されます。

包含的な収束ゲートウェイでは、包含的な分岐ゲートウェイでこれまでに作成された内向きフローすべてがマージされます。これは、包含ゲートウェイブランチの同期エントリーポイントとして機能します。

並行

並列ゲートウェイを使用して、並列フローを同期し、作成します。並列の分岐ゲートウェイでは、内向きフローが選択されると同時に、外向きフローもすべて選択されます。収束並列ゲートウェイでは、ゲートウェイは、内向きのフローがすべて到達するまで待機してからでないと、外向きフローをトリガーしません。

イベント

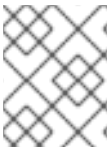
イベントベースのゲートウェイは分岐のみで、データをもとにした排他的ゲートウェイ (プロセスデータに反応) とは対照的に、発生する可能性のあるイベントに反応できます。発生したイベントをもとに、外向きフローに移動します。一度に実行できる外向きフローは 1 つとなっています。ゲートウェイは、イベントベースのゲートウェイに接続されている中間イベントが発生した場合にのみ、プロセスがインスタンス化される、開始イベントとして機能する可能性があります。

4.7. プロセスデザイナーでの BPMN2 接続オブジェクト

接続オブジェクトは、BPMN2 要素 2 つの間の関連性を作成します。接続オブジェクトが転送された場合、関連付けは順番に行われ、プロセスのインスタンス内で、要素の 1 つが他の要素よりも先に即座に実行されることを指定します。接続オブジェクトは、関連付けられているプロセス要素の上部、下部、右側、左側で開始、終了できます。OMG BPMN2 仕様では、プロセスの動作を簡単に理解して従うことができるように、接続オブジェクトを独断で配置できます。

BPMN2 は主に、2 種類の接続オブジェクトをサポートします。

- シーケンスフロー: プロセスの要素を接続し、インスタンス内でこれらの要素を実行する順番を定義します。
- 関連付けフロー: 実行セマンティクスなしでプロセスの要素を接続します。関連付けフローは転送できません。できる場合は、一方向となっています。



注記

新しいプロセスデザイナーは、転送されない関連付けフローのみをサポートします。レガシーのデザイナーは、一方向と単方向のフローをサポートしています。

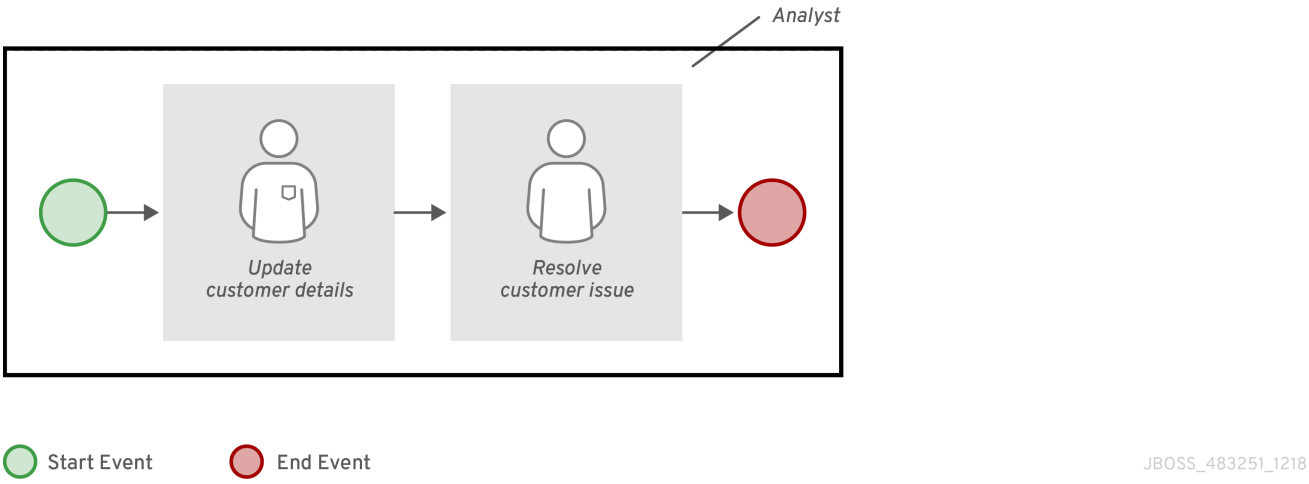
4.8. プロセスデザイナーでの BPMN2 スイムレーン

スイムレーンは、1 つのグループまたはユーザーに関連のあるタスクを視覚的にグループ化するプロセス要素です。スイムレーンとユーザータスクを組み合わせて使用し、スイムレーンの **Autoclaim** プロパティーにより、複数のユーザータスクを同じアクターに割り当てることができます。グループの所有者がスイムレーンの最初のタスクを要求すると、他のタスクが同じ所有者に直接割り当てられます。したがって、他のタスクの要求は、グループの残りの所有者が不要です。**Autoclaim** プロパティーは、スイムレーンに関連するタスクの自動割り当てを有効にします。

注記

スイムレーンの残りのユーザータスクに複数の事前定義された **ActorIds** が含まれる場合は、ユーザータスクが自動的に割り当てられません。

以下の例では、アナリストレーンは 2 つのユーザータスクで設定されます。



Update Customer Details タスクと **Resolve Customer Issue** タスクの **Group** フィールドには **analyst** 値が割り当てられています。プロセスが開始し、**Update Customer Details** タスクがアナリストにより要求されるか、開始するか、完了すると、**Resolve Customer Issue** タスクが要求されて、最初のタスクを完了したユーザーに割り当てられます。ただし、**Update Customer Details** タスクにアナリストグループが割り当てられていて、2 番目のタスクにユーザーやグループが割り当てられていない場合、プロセスは最初のタスク完了後に停止します。

スイムレーンの **Autoclaim** プロパティーを無効にすることができます。**Autoclaim** プロパティーが無効になっていると、スイムレーンに関連するタスクは自動的に割り当てられません。デフォルトでは、**Autoclaim** プロパティーの値は **true** に設定されます。必要に応じて、**Autoclaim** プロパティーのデフォルト値を Business Central のプロジェクト設定から、またはデプロイメント記述子ファイルを使用して変更することもできます。

Business Central のスイムレーンの **Autoclaim** プロパティーのデフォルト値を変更するには、以下を実行します。

1. プロジェクトの **Settings** に移動します。
2. **Deployment → Environment entries** を開きます。
3. 指定のフィールドに以下の値を入力します。
   - **名前**: **Autoclaim**
   - **値**: **"false"**

XML デプロイメント記述子の環境エントリーを設定するには、以下のコードを **kie-deployment-descriptor.xml** ファイルに追加します。

```
<environment-entries>
  ..
   <environment-entry>
      <resolver>mvel</resolver>
      <identifier>new String ("false")</identifier>
      <parameters/>
      <name>Autoclaim</name>
   </environment-entry>
  ..
</environment-entries>
```

4.9. プロセスデザイナーでの BPMN2 アーティファクト

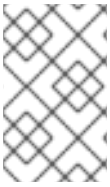
アーティファクトは、プロセスに関する追加情報を渡すのに使用します。アーティファクトは、BPMN2 ダイアグラムに描写されているオブジェクトで、プロセスワークフローの一部出ないものを指します。アーティファクトには、内向きフローオブジェクトも、外向きフローオブジェクトもありません。アーティファクトの目的は、ダイアグラムを理解するために必要な追加情報を提供することです。アーティファクトの表には、レガシーのプロセスデザイナーでサポートされているアーティファクトが一覧で表示されています。

表4.13 アーティファクト

アーティファクトのタイプ	説明
グループ	全体的なプロセスに大きな影響のあるタスクまたはプロセスを整理します。新しいプロセスデザイナーでは、グループアーティファクトはサポートされません。
テキストのアノテーション	BPMN2 ダイアグラムの文字情報を追加で提供します。
データオブジェクト	BPMN2 のダイアグラムにプロセスを通過するデータフローを表示します。

4.9.1. データオブジェクトの作成

データオブジェクトは、たとえば、物理およびデジタルの形式のプロセスで使用されるドキュメントを表します。データオブジェクトは、右上隅に折りたたまれたページとして表示されます。次の手順は、データオブジェクトの作成の一般的な概要です。



注記

Red Hat Process Automation Manager 7.12.0 では、データ入力、データ出力、および関連付けのサポートを除く、限定されたサポートをデータオブジェクトに提供しています。

手順

1. ビジネスプロセスを作成します。
2. プロセスデザイナーで、ツールパレットから **Artifacts → Data Object** を選択します。
3. データオブジェクトをプロセスデザイナーキャンバスにドラッグアンドドロップするか、キャンバスの空白エリアをクリックします。
4. 必要に応じて、画面の右上隅で、**Properties** アイコンをクリックします。
5. 必要に応じて、以下の表に一覧表示されているデータオブジェクト情報を追加または定義します。

表4.14 データオブジェクトのパラメーター

ラベル	説明

ラベル	説明
名前	データオブジェクトの名前。データオブジェクトシェイプをダブルクリックして名前を編集することもできます。
Metadata Attributes	メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を追加します。 **メタデータ属性**は、BPMN 図に新しい **metaData** 拡張を可能にし、データオブジェクト全体の動作を変更します。
タイプ	データオブジェクトのタイプを選択します。 **重要** データオブジェクトのデータ型を定義すると、**Imports** および **Data Assignments** など、**Properties** の他のデータ型フィールドに同じデータ型を使用することができます。

6. **Save** をクリックします。

第5章 BUSINESS CENTRAL でのビジネスプロセスの作成

プロセスデザイナーは、Red Hat Process Automation Manager のプロセスモデラーです。モデラーの出力は、BPMN 2.0 プロセス定義ファイルです。この定義は、定義に基づいてプロセスインスタンスを作成する Red Hat Process Automation Manager プロセスエンジンの入力として使用されます。

このセクションの手順では、簡単なビジネスプロセスを作成する方法の概要を説明します。より詳細なビジネスプロセスの例は、**プロセスサービスの使用ガイド** を参照してください。

前提条件

- Red Hat Process Automation Manager プロジェクトを作成済みまたはインポート済みである。プロジェクトの作成に関する詳細は、**Business Central におけるプロジェクトの管理** を参照してください。
- 必要なユーザーを作成済みである。ユーザーの権限および設定は、ユーザーに割り当てたロールと、ユーザーが属するグループで制御されます。ユーザーの作成に関する詳細は、**Red Hat JBoss EAP 7.4 への Red Hat Process Automation Manager のインストールおよび設定** を参照してください。

手順

1. Business Central で、**Menu → Design → Projects** に移動します。
2. プロジェクト名をクリックして、プロジェクトのアセットリストを開きます。
3. **Add Asset → Business Process**の順にクリックします。
4. **Create new Business Process**ウィザードで、以下の値を入力します。
   - **Business Process**: 新しいビジネスプロセス名
   - **Package**: 新しいビジネスプロセスのパッケージの場所 (例: **com.myspace.myProject**)
5. **OK** をクリックしてプロセスデザイナーを開きます。
6. 右上隅の **Properties** アイコンをクリックし、プロセスデータや変数などのビジネスプロセスプロパティー情報を追加します。
   a. スクロールダウンして、**Process Data** を展開します。
   b. **Process Variables** の横にある をクリックし、ビジネスプロセスで使用するプロセス変数を定義します。

表5.1 一般的なプロセスプロパティー

ラベル	説明
名前	プロセスの名前を入力します。
ドキュメント	プロセスを記述します。このフィールドのテキストはプロセスドキュメントに含まれます (該当する場合)。

ラベル	説明
ID	このプロセスの識別子 (**orderItems** など) を入力します。
Package	Red Hat Process Automation Manager プロジェクトでのこのプロセスのパッケージの場所を入力します (例: **org.acme**)。
ProcessType	プロセスがパブリックであるかプライベートであるかを指定します (該当しない場合は null を指定)。
バージョン	プロセスのアーティファクトバージョンを入力します。
アドホック	このプロセスがアドホックサブプロセスである場合は、このオプションを選択します。
Process Instance Description	プロセスの目的の説明を入力します。
インポート	クリックして **Imports** ウィンドウを開き、プロセスに必要なデータオブジェクトクラスを追加します。
実行可能	このオプションを選択して、プロセスを Red Hat Process Automation Manager プロジェクトの実行可能な部分にします。
SLA 期日	サービスレベルアグリーメント (SLA) の有効期限の日付を入力します。
プロセス変数	プロセスのプロセス変数を追加します。プロセス変数は、特定のプロセスインスタンス内で表示されます。プロセス変数はプロセスの作成時に初期化され、プロセスの完了時に破棄されます。変数 **タグ** を使用すると、変数の動作をより細かく制御できます。たとえば、変数は **required** または **readonly** になります。変数タグの詳細は、6 章*Variables* を参照してください。
Metadata Attributes	メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を追加します。
グローバル変数	プロセスにグローバル変数を追加します。グローバル変数は、プロジェクトのすべてのプロセスインスタンスとアセットに表示されます。グローバル変数は通常、ビジネスルールおよび制約によって使用され、ルールまたは制約によって動的に作成されます。

Metadata Attributes エントリーは、新しい **metaData** 拡張を BPMN ダイアグラムに有効にするという点で **Process Variables** タグに似ています。ただし、プロセス変数タグは、特定の変数が **required** または **readonly** かどうかなど、特定のプロセス変数の動作を変更しますが、メタデータ属性はプロセス全体の動作を変更する key-value 定義になります。

たとえば、BPMN プロセスの以下のカスタムメタデータ属性 **riskLevel** および値 **low** は、プロセスを開始するためのカスタムイベントリスナーに対応します。

図5.1 BPMN モデラーのメタデータ属性と値の例

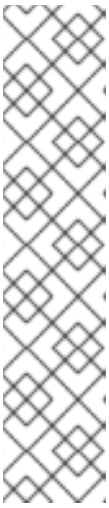
BPMN ファイルのメタデータ属性と値の例

```
<bpmn2:process id="approvals" name="approvals" isExecutable="true"
processType="Public">
  <bpmn2:extensionElements>
    <tns:metaData name="riskLevel">
      <tns:metaValue><![CDATA[low]]></tns:metaValue>
    </tns:metaData>
  </bpmn2:extensionElements>
```

メタデータ値を持つイベントリスナーの例

```
public class MyListener implements ProcessEventListener {
    ...
    @Override
    public void beforeProcessStarted(ProcessStartedEvent event) {
        Map < String, Object > metadata =
event.getProcessInstance().getProcess().getMetaData();
        if (metadata.containsKey("low")) {
            // Implement some action for that metadata attribute
        }
    }
}
```

7. プロセスデザイナーキャンバスで、左側のツールバーを使用して BPMN コンポーネントをドラッグアンドドロップし、ビジネスプロセスロジック、接続、イベント、タスク、またはその他の要素を定義します。



注記

Red Hat Process Automation Manager のタスクおよびイベントには、1 つの受信フローと 1 つの送信フローが必要です。複数の内向きフローおよび複数の外向きフローでビジネスプロセスを設計する場合は、ゲートウェイを使用してビジネスプロセスを再設計することを検討してください。ゲートウェイを使用すると、シーケンスフローが実行しているロジックが明確になります。そのため、ゲートウェイは複数の接続に対するベストプラクティスとみなされます。

ただし、タスクまたはイベントに複数の接続を使用する必要がある場合は、JVM (Java 仮想マシン) システムプロパティー **jbpm.enable.multi.con** を **true** に設定する必要があります。Business Central および KIE Server が異なるサーバーで実行する場合は、いずれのサーバーにも **jbpm.enable.multi.con** システムプロパティーの両方が有効になっていないと、プロセスエンジンが例外を出力します。

8. ビジネスプロセスのすべてのコンポーネントを追加して定義した後に、**Save** をクリックし、完了したビジネスプロセスを保存します。

5.1. ビジネスルールタスクの作成

ビジネスルールタスクは、Decision Model and Notation (DMN) モデルまたはルールフローグループを使用して意思決定を行うために使用されます。

手順

1. ビジネスプロセスを作成します。
2. プロセスデザイナーで、ツールパレットから **Activities** ツールを選択します。
3. **Business Rule** を選択します。
4. プロセスデザイナーキャンバスの空白エリアをクリックします。
5. 必要に応じて、画面の右上隅で、**Properties** アイコンをクリックします。
6. 必要に応じて、以下の表に一覧表示されているタスク情報を追加または定義します。

表5.2 ビジネスルールタスクのパラメーター

ラベル	説明
名前	ビジネスルールタスクの名前。また、ビジネスルールタスクシェイプをダブルクリックして名前を編集することもできます。
ルール言語	タスクの出力言語。Decision Model and Notation (DMN) または Drools (DRL) を選択します。
ルールフローグループ	このビジネスタスクに関連付けられたルールフローグループ。一覧からルールフローグループを選択するか、新しいルールフローグループを指定します。
開始時アクション	タスクの開始時のアクションを指定する Java、JavaScript、または MVEL スクリプト。
終了時アクション	タスクの終了時のアクションを指定する Java、JavaScript、または MVEL スクリプト。
非同期です	このタスクを非同期で呼び出す必要がある場合に選択します。外部サービスによって実行されるタスクなど、タスクを瞬時に実行できない場合は、タスクを非同期にします。
アドホックの自動開始	これが自動的に開始される必要があるアドホックタスクである場合に選択します。**AdHoc Autostart** を使用すると、タスクは、開始タスクにより開始するのではなく、プロセスまたはケースインスタンスが作成されたときに自動的に開始します。多くの場合、ケース管理で使用されます。

ラベル	説明
SLA 期日	サービスレベルアグリーメント (SLA) の有効期限の日付。
割当	クリックしてローカル変数を追加します。
Metadata Attributes	メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を追加します。 **メタデータ属性**は、BPMN 図に新しい **metaData** 拡張を可能にし、タスク全体の動作を変更します。

7. **Save** をクリックします。

5.2. スクリプトタスクの作成

スクリプトタスクは、Java、JavaScript、または MVEL で記述されたコードを実行するために使用されます。これらには、スクリプトタスクのアクションを指定するコードスニペットが含まれています。スクリプトにグローバル変数とプロセス変数を含めることができます。

MVEL は有効な Java コードをすべて受け入れ、さらにパラメーターのネストされたアクセスをサポートすることに留意してください。たとえば、Java 呼び出し **person.getName()** に相当する MVEL は、**person.name** です。MVEL は Java に対して他の改善も提供し、MVEL 式は一般にビジネスユーザーにとってより使いやすいものになります。

手順

1. ビジネスプロセスを作成します。
2. プロセスデザイナーで、ツールパレットから **Activities** ツールを選択します。
3. **Script** を選択します。
4. プロセスデザイナーキャンバスの空白エリアをクリックします。
5. 必要に応じて、画面の右上隅で、**Properties** アイコンをクリックします。
6. 必要に応じて、以下の表に一覧表示されているタスク情報を追加または定義します。

表5.3 スクリプトタスクのパラメーター

ラベル	説明
名前	スクリプトタスクの名前。また、スクリプトタスクシェイプをダブルクリックして名前を編集することもできます。
ドキュメント	タスクの説明を入力します。このフィールドのテキストは、プロセスのドキュメントに含まれています。プロセスデザイナーキャンバスの左上にある **Documentation** タブをクリックして、プロセスドキュメントを表示します。

ラベル	説明
スクリプト	タスクによって実行されるスクリプトを Java、JavaScript、または MVEL で入力し、スクリプトタイプを選択します。
非同期です	このタスクを非同期で呼び出す必要がある場合に選択します。外部サービスによって実行されるタスクなど、タスクを瞬時に実行できない場合は、タスクを非同期にします。
アドホックの自動開始	これが自動的に開始される必要があるアドホックタスクである場合に選択します。**AdHoc Autostart** を使用すると、タスクは、開始タスクにより開始するのではなく、プロセスまたはケースインスタンスが作成されたときに自動的に開始します。多くの場合、ケース管理で使用されます。
Metadata Attributes	メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を追加します。 **メタデータ属性**は、BPMN 図に新しい **metaData** 拡張を可能にし、タスク全体の動作を変更します。

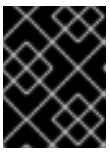
7. **Save** をクリックします。

5.3. サービスタスクの作成

サービスタスクは、Web サービス呼び出しまたは Java クラスメソッドを基にアクションを実行するタスクです。サービスタスクの例としては、これらのタスクを実行する際の電子メールの送信やメッセージのロギングなどがあります。サービスタスクに関連付けられているパラメーター (入力) と結果 (出力) を定義できます。1 つのオブジェクトに対する全入力を含めて、ラップパラメーターを定義することも可能です。ラップパラメーターを定義するには、データ割り当てで **Wrapped` : `True** を使用して、新しいワークアイテムハンドラーを作成します。サービスタスクには、内向きの接続 1 つと外向きの接続 1 つが必要です。

手順

1. Business Central で、画面の右上隅にある **Admin** アイコンを選択し、**Artifacts** を選択します。
2. **Upload** をクリックして、**Artifact upload** ウィンドウを開きます。
3. **.jar** ファイルを選択し、 をクリックします。



重要

.jar ファイルには、Web サービスのデータタイプ (データオブジェクト)、および Java サービスタスクの Java クラスが含まれます。

4. 使用するプロジェクトを作成します。

5. プロジェクトの **Settings** → **Dependencies** に移動します。
6. **Add from repository** をクリックしてアップロードした **.jar** ファイルを見つけ、**Select** をクリックします。
7. プロジェクトの **Settings** → **Work Item Handler** を開きます。
8. 指定のフィールドに以下の値を入力します。
   - Name: **Service Task**
   - Value - **new org.jbpm.process.workitem.bpmn2.ServiceTaskHandler(ksession, classLoader)**
9. プロジェクトを保存します。

Web サービスタスクの作成例

BPMN2 仕様のサービスタスクのデフォルトの実装は Web サービスです。Web サービスのサポートは Apache CXF 動的クライアントをベースとしています。これは、**WorkItemHandler** インターフェイスを実装する専用のサービスタスクハンドラーを提供します。

org.jbpm.process.workitem.bpmn2.ServiceTaskHandler

Web サービスを使用してサービスタスクを作成するには、Web サービスを設定する必要があります。

a. ビジネスプロセスを作成します。

b. 必要に応じて、画面の右上隅で、**Properties** アイコンをクリックします。

c. **Imports** プロパティーで をクリックし、**Imports** ウィンドウを開きます。

d. **WSDL Imports** の横にある **+Add** をクリックして、必要な WSDL (Web Services Description Language) の値をインポートします。以下に例を示します。
   - **場所**: **http://localhost:8080/sample-ws-1/SimpleService?wsdl**
     この場所は、サービスの WSDL ファイルを参照します。
   - **名前空間**: **http://bpmn2.workitem.process.jbpm.org/**
     名前空間は、WSDL ファイルの **targetNamespace** と一致している必要があります。

e. プロセスデザイナーで、ツールパレットから **Activities** ツールを選択します。

f. **Service Task** を選択します。

g. プロセスデザイナーキャンバスの空白エリアをクリックします。

h. 必要に応じて、以下の表に一覧表示されているタスク情報を追加または定義します。

表5.4 Web サービスタスクパラメーター

ラベル	説明
名前	サービスタスクの名前。サービスタスクシェイプをダブルクリックして名前を編集することもできます。

ラベル	説明
ドキュメント	タスクの説明を入力します。このフィールドのテキストは、プロセスのドキュメントに含まれています。プロセスデザイナーキャンバスの左上にある **Documentation** タブをクリックして、プロセスドキュメントを表示します。
Implementation	Web サービスを指定します。
Interface	**CountriesPortService** などのスクリプトの実装に使用されるサービス。
操作	**getCountry** などのインターフェイスによって呼び出される操作。
割当	クリックしてローカル変数を追加します。
アドホックの自動開始	これが自動的に開始される必要があるアドホックタスクである場合に選択します。**AdHoc Autostart** を使用すると、タスクは、開始タスクにより開始するのではなく、プロセスまたはケースインスタンスが作成されたときに自動的に開始します。多くの場合、ケース管理で使用されます。
非同期です	このタスクを非同期で呼び出す必要がある場合に選択します。外部サービスによって実行されるタスクなど、タスクを瞬時に実行できない場合は、タスクを非同期にします。
Is Multiple Instance	このタスクに複数のインスタンスがある場合に選択します。
MI 実行モード	複数のインスタンスが並列または順次実行されるかどうかを選択します。
MI コレクション入力	**inputCountryNames** などの、新規インスタンスが作成される要素のコレクションを表す変数を指定します。
MI データ入力	**Parameter** などの Web サービスに転送される入力データ割り当てを指定します。
MI コレクション出力	**outputCountries** などの Web サービスタスクから返された値を保存する配列の一覧。
MI データ出力	**Result** などのサーバーでのクラス実行の結果を保存する Web サービスタスクの出力データ割り当てを指定します。
MI 完了条件 (mvel)	指定した複数のインスタンスノードを完了できるかどうかを確認するために、完了済みの各インスタンスを評価する MVEL 式を指定します。

ラベル	説明
開始時アクション	タスクの開始時のアクションを指定する Java、JavaScript、または MVEL スクリプト。
終了時アクション	タスクの終了時のアクションを指定する Java、JavaScript、または MVEL スクリプト。
SLA 期日	サービスレベルアグリーメント (SLA) の有効期限の日付。
Metadata Attributes	メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を追加します。 **メタデータ属性**は、BPMN 図に新しい **metaData** 拡張を可能にし、タスク全体の動作を変更します。

Java サービスタスクの作成例

Java メソッドを使用してサービスタスクを作成する場合、メソッドにはパラメーターを 1 つだけ含み、単一の値を返すことができます。Java メソッドを使用してサービスタスクを作成するには、Java クラスをプロジェクトの依存関係に追加する必要があります。

a. ビジネスプロセスを作成します。

b. プロセスデザイナーで、ツールパレットから **Activities** ツールを選択します。

c. **Service Task** を選択します。

d. プロセスデザイナーキャンバスの空白エリアをクリックします。

e. 必要に応じて、画面の右上隅で、**Properties** アイコンをクリックします。

f. 必要に応じて、以下の表に一覧表示されているタスク情報を追加または定義します。

表5.5 Java サービスタスクパラメーター

ラベル	説明
名前	サービスタスクの名前。サービスタスクシェイプをダブルクリックして名前を編集することもできます。
ドキュメント	タスクの説明を入力します。このフィールドのテキストは、プロセスのドキュメントに含まれています。プロセスデザイナーキャンバスの左上にある **Documentation** タブをクリックして、プロセスドキュメントを表示します。
Implementation	タスクが Java に実装されるように指定します。
Interface	**org.xyz.HelloWorld** などのスクリプトの実装に使用されるクラス。

ラベル	説明
操作	**sayHello** などのインターフェイスによって呼び出されるメソッド。
割当	クリックしてローカル変数を追加します。
アドホックの自動開始	これが自動的に開始される必要があるアドホックタスクである場合に選択します。**AdHoc Autostart** を使用すると、タスクは、開始タスクにより開始するのではなく、プロセスまたはケースインスタンスが作成されたときに自動的に開始します。多くの場合、ケース管理で使用されます。
非同期です	このタスクを非同期で呼び出す必要がある場合に選択します。外部サービスによって実行されるタスクなど、タスクを瞬時に実行できない場合は、タスクを非同期にします。
Is Multiple Instance	このタスクに複数のインスタンスがある場合に選択します。
MI 実行モード	複数のインスタンスが並列または順次実行されるかどうかを選択します。
MI コレクション入力	**InputCollection** などの、新規インスタンスが作成される要素のコレクションを表す変数を指定します。
MI データ入力	Java クラスに転送される入力データ割り当てを指定します。たとえば、入力データの割り当てを **Parameter** および **ParameterType** に設定できます。**ParameterType** は、**Parameter** のタイプを表し、Java メソッドの実行に引数を送信します。
MI コレクション出力	**OutputCollection** など、Java クラスから返される値を保存する配列リスト。
MI データ出力	**Result** など、サーバーでクラス実行の結果を保存する Java サービスタスクへの出力データ割り当てを指定します。
MI 完了条件 (mvel)	指定した複数のインスタンスノードを完了できるかどうかを確認するために、完了済みの各インスタンスを評価する MVEL 式を指定します。たとえば、**OutputCollection.size() <= 3** は、3 人以上のユーザーがアドレス指定されていないことを示します。
開始時アクション	タスクの開始時のアクションを指定する Java、JavaScript、または MVEL スクリプト。
終了時アクション	タスクの終了時のアクションを指定する Java、JavaScript、または MVEL スクリプト。

ラベル	説明
SLA 期日	サービスレベルアグリーメント (SLA) の有効期限の日付。
Metadata Attributes	メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を追加します。 **メタデータ属性**は、BPMN 図に新しい **metaData** 拡張を可能にし、タスク全体の動作を変更します。

10. **Save** をクリックします。

5.4. ユーザータスクの作成

ユーザータスクは、ビジネスプロセスに人間が行うアクションをインプットとして追加するために使用します。

手順

1. ビジネスプロセスを作成します。
2. プロセスデザイナーで、ツールパレットから **Activities** ツールを選択します。
3. **User** を選択します。
4. ユーザータスクをプロセスデザイナーキャンバスにドラッグアンドドロップします。
5. 必要に応じて、画面の右上隅で、**Properties** アイコンをクリックします。
6. 必要に応じて、以下の表に一覧表示されているタスク情報を追加または定義します。

表5.6 ユーザータスクパラメーター

ラベル	説明
名前	ユーザータスクの名前。ユーザータスクシェイプをダブルクリックして名前を編集することもできます。
ドキュメント	タスクの説明を入力します。このフィールドのテキストは、プロセスのドキュメントに含まれています。プロセスデザイナーキャンバスの左上にある **Documentation** タブをクリックして、プロセスドキュメントを表示します。
タスク名	ヒューマンタスクの名前。
件名	タスクの件名を入力します。

ラベル	説明
アクター	ヒューマンタスクの実行を担当するアクター。**Add** をクリックして行を追加し、一覧からアクターを選択するか、**New** をクリックして新しいアクターを追加します。
Groups	ヒューマンタスクの実行を担当するグループ。**Add** をクリックして行を追加し、一覧からグループを選択するか、**New** をクリックして新しいグループを追加します。
割当	このタスクのローカル変数。**Task Data I/O** ウィンドウをクリックして開き、必要に応じてデータの入力と出力を追加します。MVEL 式をデータ入力および出力の割り当てとして追加することもできます。MVEL 言語の詳細は、Language Guide for 2.0 を参照してください。
再割り当て	別のアクターを指定して、このタスクを完了します。
通知	クリックして、タスクに関連付けられた通知を指定します。
非同期です	このタスクを非同期で呼び出す必要がある場合に選択します。外部サービスによって実行されるタスクなど、タスクを瞬時に実行できない場合は、タスクを非同期にします。
省略可能	このタスクが必須ではない場合に選択します。
優先順位	タスクの優先度を指定します。
説明	ヒューマンタスクの説明を入力します。
作成者	このタスクを作成したユーザー。
アドホックの自動開始	これが自動的に開始される必要があるアドホックタスクである場合に選択します。**AdHoc Autostart** を使用すると、タスクは、開始タスクにより開始するのではなく、プロセスまたはケースインスタンスが作成されたときに自動的に開始します。多くの場合、ケース管理で使用されます。
複数のインスタンス	このタスクに複数のインスタンスがある場合に選択します。
開始時アクション	タスクの開始時のアクションを指定する Java、JavaScript、または MVEL スクリプト。
終了時アクション	タスクの終了時のアクションを指定する Java、JavaScript、または MVEL スクリプト。
コンテンツ	スクリプトのコンテンツ。
SLA 期日	サービスレベルアグリーメント (SLA) の有効期限の日付。

ラベル	説明
Metadata Attributes	メタデータ属性が存在する場合に何らかのアクションを実装するリスナーなど、カスタムイベントリスナーに使用するカスタムメタデータ属性の名前と値を追加します。 **メタデータ属性**は、BPMN 図に新しい **metaData** 拡張を可能にし、タスク全体の動作を変更します。

7. **Save** をクリックします。

5.4.1. ユーザータスク割り当てストラテジーの設定

ユーザータスク割り当てストラテジーは、タスクを自動的に適切なユーザーに割り当てるために使用します。割り当てストラテジーにより、見込みオーナー、タスクの優先順位、タスクデータなどの関連するプロパティーに基づいて、より効率的なタスクの割り当てが可能になります。**org.jbpm.task.assignment.strategy** は、Red Hat Process Automation Manager のユーザータスク割り当てストラテジーのシステムプロパティーです。Business Central でユーザータスクの割り当てストラテジーを明示的に定義することもできます。

前提条件

- Business Central でプロジェクトを作成している。
- **org.jbpm.task.assignment.enabled** システムプロパティーを **true** に設定する必要があります。

手順

1. ビジネスプロセスを作成します。
   Business Central でのビジネスプロセスの作成に関する詳細は、*5章Business Central でのビジネスプロセスの作成* を参照してください。
2. ユーザータスクを作成します。
   Business Central でユーザータスクを作成する方法は、「ユーザータスクの作成」を参照してください。
3. 画面の右上隅で、**Properties** アイコンをクリックします。
4. **Implementation/Execution** を展開し、**Assignments** の下にある をクリックして、**Data I/O** ウィンドウを開きます。
5. 名前に **AssignmentStrategy**、タイプに **String**、コンスタントソースにストラテジー名などを指定してデータの入力を追加します。

   注記

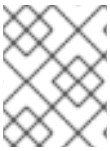
   AssignmentStrategy が null に設定されている場合には、そのタスクには割り当てストラテジーは使用されません。

6. **OK** をクリックします。
   AssignmentStrategy 変数は、ユーザータスクへのデータ入力として追加されます。

5.5. プロセスデザイナーでの BPMN2 ユーザータスクのライフサイクル

プロセスインスタンスの実行時にユーザータスク要素をトリガーしてユーザータスクを作成できます。タスク実行エンジンのユーザータスクサービスは、ユーザータスクインスタンスを実行します。プロセスインスタンスは、関連するユーザータスクが完了したか、または中止された場合にのみ実行を継続します。ユーザータスクのライフサイクルは以下のとおりです。

- プロセスインスタンスがユーザータスク要素に入ると、ユーザータスクは **Created** ステージに入ります。
- **Created** は一時的なステージで、ユーザータスクはすぐに **Ready** ステージに入ります。タスクを実行可能な全アクターのタスク一覧に、タスクは表示されます。
- アクターがユーザータスクをクレームすると、タスクは **Reserved** になります。



注記

ユーザータスクに、利用可能なアクターが 1 つある場合、タスクは作成されるとそのアクターに割り当てられます。

- ユーザータスクをクレームしたアクターが実行を開始すると、ユーザータスクのステータスは **InProgress** に変更します。
- アクターがユーザータスクを完了したら、実行の結果に応じて、ステータスが **Completed** または **Failed** に変わります。

以下のように、他にもライフサイクルメソッドが複数あります。

- ユーザータスクが別のアクターに割り当てられるように、ユーザータスクを委任または転送する。
- ユーザータスクを取り消すと、ユーザータスクは単一のアクターにより要求されなくなりますが、それを実行することを許可されているすべてのアクターが使用できます。
- ユーザータスクを一時停止して再開する。
- 進行中のユーザータスクを停止する。
- タスクの実行が一時停止されたユーザータスクをスキップする。

ユーザータスクのライフサイクルに関する詳細は、Web Services Human Task の仕様 を参照してください。

5.6. プロセスデザイナーでの BPMN2 タスクパーミッションのマトリックス

ユーザータスクのパーミッションマトリックスは、特定のユーザーロールで可能なアクションをまとめています。ユーザーロールは以下のとおりです。

- 潜在的な所有者: タスク (以前にクレームされた後に解放、転送されたタスク) をクレーム可能なユーザー。ステータスが **Ready** のタスクはクレーム可能で、潜在的な所有者はタスクの実際の所有者になります。
- 実際の所有者: タスクを要求し、タスクが完了するか、失敗するまで進めるユーザー。
- ビジネス管理者: ステータスを変更して、タスクのライフサイクルのどの時点にでもタスクを進めることができるスーパーユーザー。

以下のパーミッションマトリックスは、タスクの全変更操作の承認を表します。

- **+** は、ユーザーロールが指定の操作を実行できることを表します。
- **-** は、ユーザーロールが指定の操作を実行できないか、操作がユーザーのロールと一致しないことを表します。

表5.7 主な操作のパーミッションマトリックス

操作	Potential_Owner	実際の所有者	ビジネス管理者
アクティベート	-	-	+
クレーム	+	-	+
完了	-	+	+
権限委譲	+	+	+
失敗	-	+	+
進む	+	+	+
ノミネート	-	-	+
リリース	-	+	+
削除	-	-	+
再開	+	+	+
スキップ	+	+	+
開始	+	+	+
停止	-	+	+
一時停止	+	+	+

5.7. ビジネスプロセスのコピーの作成

Business Central でビジネスプロセスのコピーを作成し、必要に応じてコピーしたプロセスを変更できます。

手順

1. ビジネスプロセスデザイナーで、右上のツールバーの **Copy** をクリックします。
2. **Make a Copy** ウィンドウで、コピーしたビジネスプロセスの新しい名前を入力し、ターゲットパッケージを選択して、必要に応じて、コメントを追加します。

3. **Make a Copy** をクリックします。

4. 必要に応じてコピーしたビジネスプロセスを変更し、**Save** をクリックして、更新されたビジネスプロセスを保存します。

5.8. 要素のサイズを変更し、ズーム機能を使用したビジネスプロセスの表示

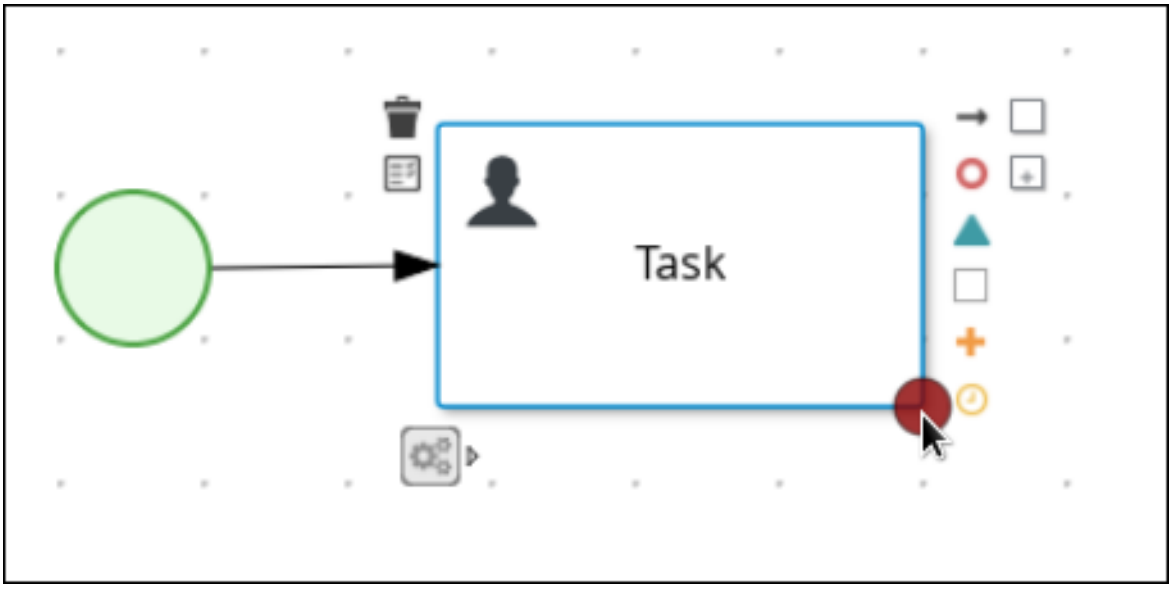
ビジネスプロセスの個々の要素のサイズを変更し、ズームインまたはズームアウトして、ビジネスプロセスの表示を変更できます。

手順

1. ビジネスプロセスデザイナーで、要素を選択し、要素の右下隅にある赤い点をクリックします。

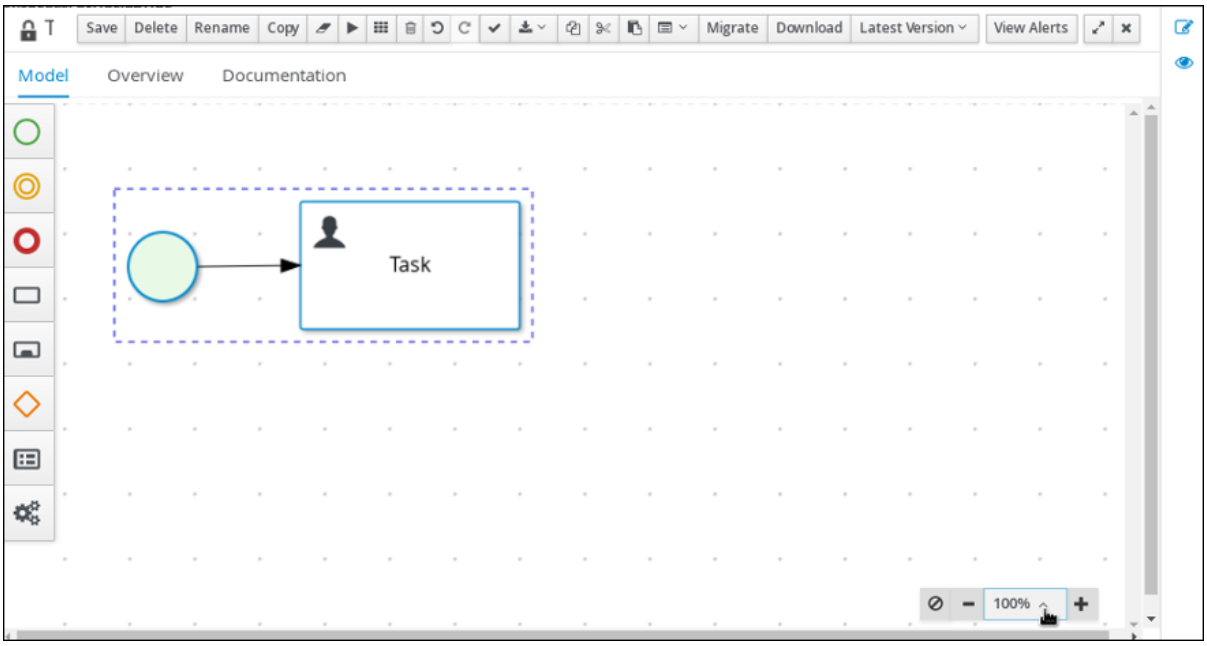
2. 赤い点をドラッグして、要素のサイズを変更します。

図5.2 要素のサイズ変更



3. ズームインまたはズームアウトしてダイアグラム全体を表示するには、キャンバスの右下にあるプラス記号またはマイナス記号をクリックします。

図5.3 ビジネスプロセスの拡大または縮小



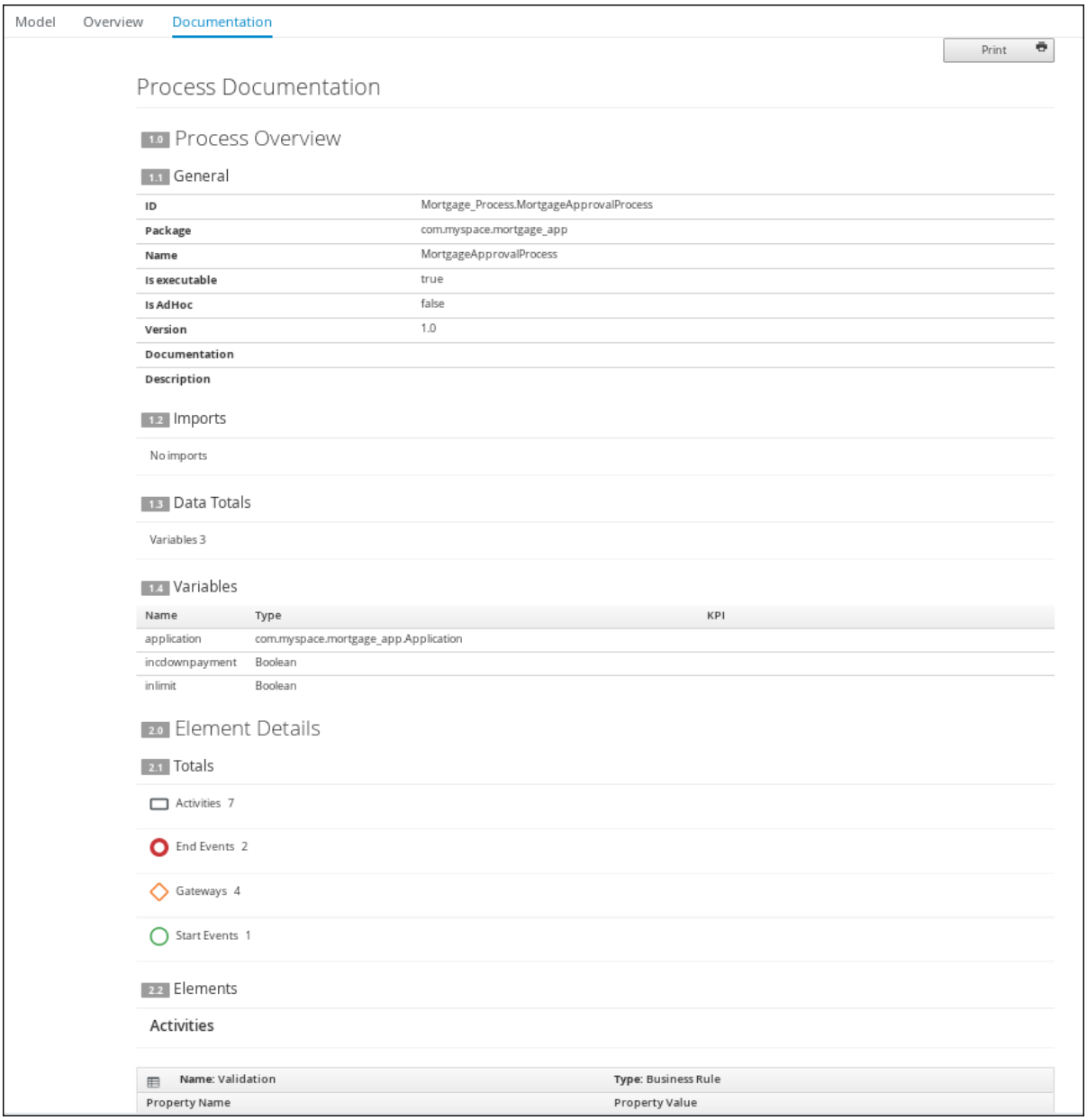
5.9. BUSINESS CENTRAL でのプロセスドキュメントの生成

Business Central のプロセスデザイナーでは、プロセス定義のレポートを表示し、出力できます。プロセスドキュメントには、コンポーネント、データ、プロセスの視覚的なフローが簡単に出力して共有できるように PDF 形式でまとめられています。

手順

1. Business Central で、ビジネスプロセスが含まれるプロジェクトに移動し、プロセスを選択します。
2. プロセスデザイナーの **Documentation** タブでプロセスファイルの概要を表示し、画面の右上隅の **Print** をクリックして PDF レポートを出力します。

図5.4 プロセスドキュメントの生成



第6章 VARIABLES

ランタイム時に使用するデータを格納する変数。プロセスデザイナーは、3 種類の変数を使用します。

グローバル変数

グローバル変数は、特定のセッションのすべてのプロセスインスタンスとアセットに表示されます。これらは、主にビジネスルールおよび制約によって使用されることを目的としており、ルールまたは制約によって動的に作成されます。

プロセス変数

プロセス変数は、BPMN2 定義ファイルのプロパティーとして定義され、プロセスインスタンス内で表示されます。プロセスの作成時に初期化され、プロセスの完了時に破棄されます。

ローカル変数

ローカル変数は、アクティビティーなどの特定のプロセス要素に関連付けられ、その中で使用できます。要素コンテキストが初期化されると、ローカル変数は初期化されます。つまり、実行ワークフローがノードに入り、**onEntry** アクションの実行が終了した場合に初期化されます (該当する場合)。要素コンテキストが破棄されると、ローカル変数は破棄されます。つまり、実行ワークフローが要素を離れる場合に破棄されます。

プロセス、サブプロセス、またはタスクなどの要素は、独自のコンテキストと親コンテキストの変数にのみアクセスできます。要素は、要素の子要素で定義された変数にアクセスできません。したがって、実行時に要素が変数へのアクセスを必要とする場合は、独自のコンテキストが最初に検索されます。

変数が要素のコンテキストで直接見つからない場合は、直近の親コンテキストが検索されます。検索は、プロセスコンテキストに到達するまで続行されます。グローバル変数の場合、検索はセッションコンテナーで直接実行されます。

変数が見つからない場合、読み取りアクセス要求は **null** を返し、書き込みアクセスはエラーメッセージを生成して、プロセスは実行を継続します。変数は ID に基づいて検索されます。

6.1. 変数タグ

変数の動作をより細かく制御するには、BPMN プロセスファイルでプロセス変数とローカル変数にタグ付けすることができます。タグは、特定の変数にメタデータとして追加する単純な文字列値です。

Red Hat Process Automation Manager は、プロセス変数とローカル変数の以下のタグをサポートします。

- **required**: プロセスインスタンスを開始するための要件として変数を設定します。必要な変数なしでプロセスインスタンスが起動すると、Red Hat Process Automation Manager は **VariableViolationException** エラーを生成します。
- **readonly**: 変数が情報提供のみを目的としており、設定できるのはプロセスインスタンスの実行中に 1 回のみであることを示します。readonly 変数の値がいずれかの時点で変更されると、Red Hat Process Automation Manager は **VariableViolationException** エラーを生成します。
- **restricted**: **VariableGuardProcessEventListener** で使用される特別なタグで、必要かつ既存のロールに基づいて変数を変更する権限が付与されていることを示します。
  VariableGuardProcessEventListener は、**DefaultProcessEventListener** から拡張されたもので、2 つの異なるコンストラクターをサポートします。
  - **VariableGuardProcessEventListener**

```
public VariableGuardProcessEventListener(String requiredRole, IdentityProvider
identityProvider) {
    this("restricted", requiredRole, identityProvider);
}
```

- **VariableGuardProcessEventListener**

```
public VariableGuardProcessEventListener(String tag, String requiredRole,
IdentityProvider identityProvider) {
    this.tag = tag;
    this.requiredRole = requiredRole;
    this.identityProvider = identityProvider;
}
```

したがって、以下の例に示すように、許可されたロール名とユーザーロールを返す ID プロバイダーを使用して、イベントリスナーをセッションに追加する必要があります。

```
ksession.addEventListener(new VariableGuardProcessEventListener("AdminRole",
myIdentityProvider));
```

上記の例では、**VariableGuardProcessEventListener** メソッドで、変数にセキュリティー制約タグ (**restricted**) が付いているかどうかを確認します。ユーザーに必要なロールがない場合、Red Hat Process Automation Manager は **VariableViolationException** エラーを生成します。

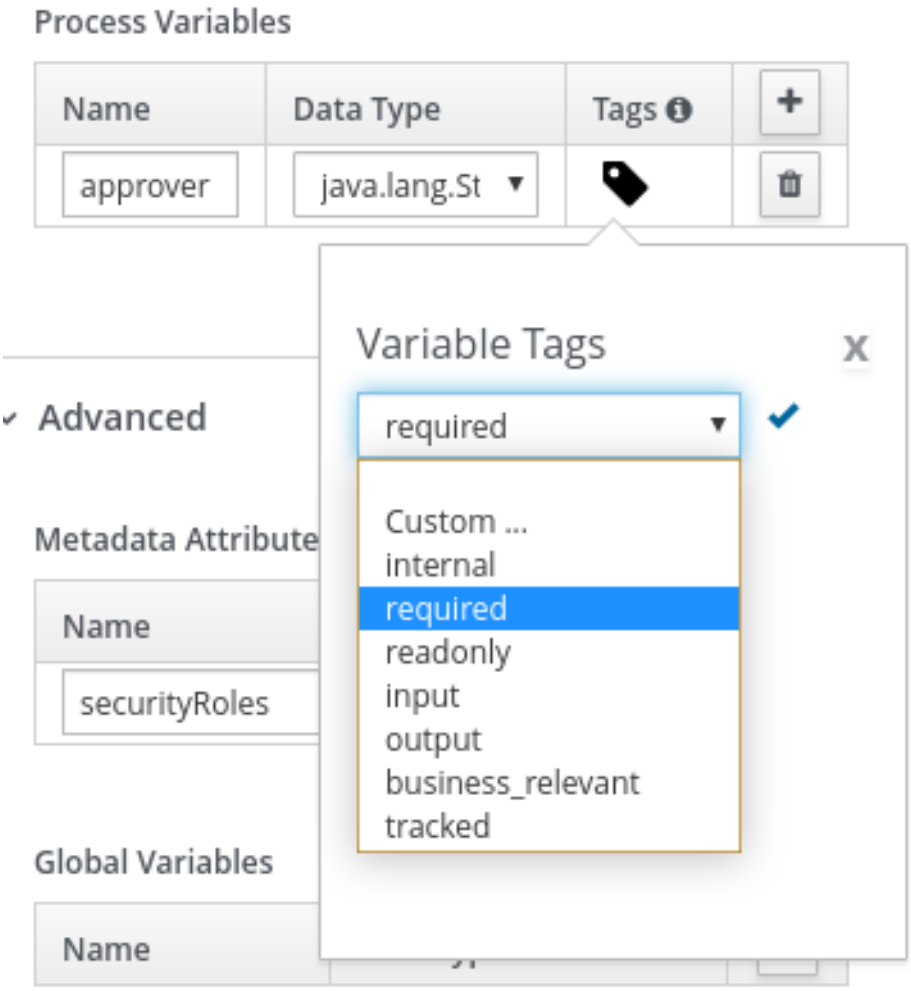
注記

Business Central UI に表示される変数タグ (**internal**、**input**、**output**、**business-relevant**、**tracked** など) は、Red Hat Process Automation Manager ではサポートされません。

タグは、**![CDATA[TAG_NAME]]** 形式で定義されたタグ値を使用し、**customTags** メタデータプロパティーとして BPMN プロセスソースファイルに直接追加できます。

たとえば、以下の BPMN プロセスは、**required** タグを **approver** プロセス変数に適用します。

図6.1 BPMN モデラーでタグ付けされた変数の例



BPMN ファイルでタグ付けされた変数の例

```
<bpmn2:property id="approver" itemSubjectRef="ItemDefinition_9" name="approver">
  <bpmn2:extensionElements>
    <tns:metaData name="customTags">
      <tns:metaValue><![CDATA[required]]></tns:metaValue>
    </tns:metaData>
  </bpmn2:extensionElements>
</bpmn2:property>
```

必要に応じて、変数に複数のタグを使用できます。BPMN ファイルでカスタム変数タグを定義して、Red Hat Process Automation Manager プロセスイベントリスナーが変数データを利用できるようにすることも可能です。カスタムタグは、標準の変数タグのように Red Hat Process Automation Manager のランタイムに影響を与えることはせず、情報提供のみを目的としています。カスタム変数タグは、標準の Red Hat Process Automation Manager 変数タグに使用する場合と同じ **customTags** メタデータプロパティー形式で定義します。

6.2. グローバル変数の定義

グローバル変数はナレッジセッションに存在し、アクセスが可能で、そのセッションのすべてのアセットで共有されます。これらはナレッジベースの特定のセッションに属し、エンジンに情報を渡すために使用されます。すべてのグローバル変数は、その ID とアイテムサブジェクト参照を定義します。ID は変数名として機能し、プロセス定義内で一意である必要があります。アイテムサブジェクト参照は、変数が保存するデータタイプを定義します。

重要

ルールは、ファクトが挿入されたときに評価されます。したがって、ファクトパターンを制約するグローバル変数を使用していて、グローバルが設定されていない場合、システムは **NullPointerException** を返します。

グローバル変数は、変数定義を持つプロセスがセッションに追加されるとき、またはセッションがパラメーターとしてグローバルで初期化されるときに初期化されます。

グローバル変数の値は通常、割り当て中に変更できます。これは、プロセス変数とアクティビティー変数間のマッピングになります。続いてグローバル変数は、ローカルアクティビティーコンテキスト、ローカルアクティビティー変数、または子コンテキストから変数への直接呼び出しによって関連付けられます。

前提条件

- Business Central でプロジェクトを作成済みである。これにはビジネスプロセスアセットが 1 つ以上含まれます。

手順

1. ビジネスプロセスアセットを開きます。
2. プロセスデザイナーキャンバスの空白エリアをクリックします。
3. 画面の右上にある **Properties** アイコンをクリックして、**Properties** パネルを開きます。
4. 必要に応じて、**Process** セクションを展開します。
5. **Global Variables** のサブセクションで、プラスアイコンをクリックします。
6. **Name** ボックスに変数の名前を入力します。
7. **Data Type** メニューからデータタイプを選択します。

6.3. プロセス変数の定義

プロセス変数は、BPMN2 定義ファイルのプロパティーとして定義され、プロセスインスタンス内で表示されます。プロセスの作成時に初期化され、プロセスの完了時に破棄されます。

プロセス変数は、プロセスコンテキストに存在する変数であり、そのプロセスまたはその子要素からアクセスできます。プロセス変数は特定のプロセスインスタンスに属し、他のプロセスインスタンスからアクセスすることはできません。すべてのプロセス変数は、その ID とアイテムサブジェクト参照を定義します。ID は変数名として機能し、プロセス定義内で一意である必要があります。アイテムサブジェクト参照は、変数が保存するデータタイプを定義します。

プロセス変数は、プロセスインスタンスの作成時に初期化されます。それらの値は、グローバル変数がローカルアクティビティーコンテキスト、ローカルアクティビティー変数に関連付けられている場合に、割り当てを使用するプロセスアクティビティーによって変更できます。または、子コンテキストから変数への直接呼び出しにより変更できます。

プロセス変数は、ローカル変数にマッピングする必要があることに注意してください。

前提条件

- Business Central でプロジェクトを作成済みである。これにはビジネスプロセスアセットが 1 つ以上含まれます。

手順

1. ビジネスプロセスアセットを開きます。
2. プロセスデザイナーキャンバスの空白エリアをクリックします。
3. 画面の右上にある **Properties** アイコンをクリックして、**Properties** パネルを開きます。
4. 必要に応じて、**Process Data** セクションを展開します。
5. **Process Variables** サブセクションで、プラスアイコンをクリックします。
6. **Name** ボックスに変数の名前を入力します。
7. **Data Type** メニューからデータタイプを選択します。

6.4. ローカル変数の定義

ローカル変数は、アクティビティーなどのプロセス要素内で使用できます。要素コンテキストが初期化されると、ローカル変数は初期化されます。つまり、実行ワークフローがノードに入り、**onEntry** アクションの実行が終了した場合に初期化されます (該当する場合)。要素コンテキストが破棄されると、ローカル変数は破棄されます。つまり、実行ワークフローが要素を離れる場合に破棄されます。

ローカル変数の値は、グローバル変数またはプロセス変数にマッピングできます。これにより、ローカル変数を収容する親要素の相対的な独立性を維持できます。このような分離は、技術的な例外の防止に役立つ場合があります。

ローカル変数は、プロセスの子要素コンテキストに存在する変数であり、このコンテキスト内からのみアクセスできます。ローカル変数は、プロセスの特定の要素に属します。

スクリプトタスクを除くタスクの場合は、**Assignments** プロパティーの **Data Input Assignments** および **Data Output Assignments** を定義できます。Data Input Assignments は、Task に入る変数を定義し、タスクの実行に必要な入力データを提供します。Data Output Assignments は、実行後の Task のコンテキストを参照して、出力データを取得できます。

ユーザータスクは、ユーザータスクを実行しているアクターに関連するデータを提示します。さらに、ユーザータスクは、実行に関連する結果データを提供するようにアクターに要求します。

データを要求および提供するには、タスクフォームを使用し、Data Input Assignment パラメーターのデータを変数にマッピングします。データを出力として保持する場合は、Data Output Assignment パラメーターでユーザーが提供したデータをマッピングします。

前提条件

- Business Central でプロジェクトを作成済みである。プロジェクトには、スクリプトタスクではないタスクが 1 つ以上あるビジネスプロセスアセットが 1 つ以上含まれます。

手順

1. ビジネスプロセスアセットを開きます。
2. スクリプトタスクではないタスクを選択します。

3. 画面の右上にある **Properties** アイコンをクリックして、**Properties** パネルを開きます。
4. **Assignments** サブセクションの下のボックスをクリックします。**Task Data I/O** ダイアログボックスが開きます。
5. **Data Inputs and Assignments** または **Data Outputs and Assignments** の横にある **Add** をクリックします。
6. **Name** ボックスにローカル変数の名前を入力します。
7. **Data Type** メニューからデータタイプを選択します。
8. ソースまたはターゲットを選択してから、**Save** をクリックします。

6.5. プロセス変数値の編集

プロセスインスタンスを開始したら、Business Central でプロセス変数の値を編集できます。サポート対象の変数タイプは、Boolean、Float、Integer、Enums です。

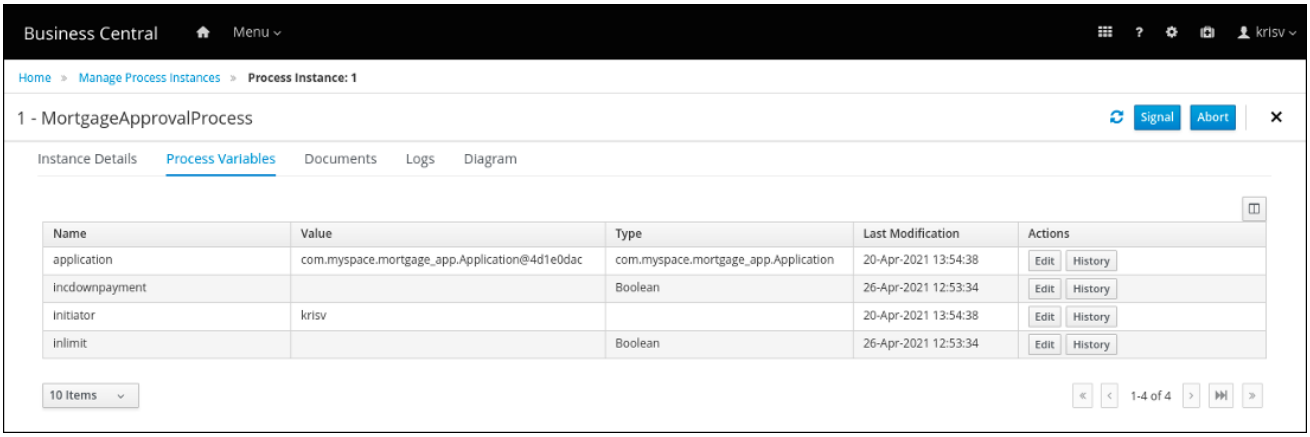
前提条件

- Business Central でプロジェクトを作成しており、プロセスインスタンスを開始している。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Process Variables** タブを選択し、編集する変数名の **Edit** をクリックします。
3. **変数値** を追加または変更して、**Save** をクリックします。

図6.2 Business Central の変数値の編集ボタン



第7章 アクションスクリプト

アクションスクリプトは、**Script** プロパティーまたは要素のインターセプターアクションを定義するコードの一部です。アクションスクリプトは、グローバル変数、プロセス変数、および事前定義変数 **kcontext** にアクセスできます。kcontext は、**ProcessContext** インターフェイスのインスタンスです。**kcontext** 変数の詳細は、**ProcessContext** Javadoc を参照してください。

Java および MVEL は、アクションスクリプト定義の方言としてサポートされます。MVEL は有効な Java コードを受け入れ、さらにパラメーターへのネストされたアクセスをサポートします。たとえば、MVEL 呼び出し **person.name** は Java 呼び出し **person.getName()** と同等です。

Java 方言および MVEL 方言でのアクションスクリプトの例

```
// Java dialect
System.out.println(person.getName());

// MVEL dialect
System.out.println(person.name);
```

アクションスクリプトを使用して、プロセスインスタンスに関する情報を表示することもできます。たとえば、以下のコマンドを使用して以下を使用します。

- プロセスインスタンスの ID を返します。

  ```
  System.out.println(kcontext.getProcessInstance().getId());
  ```

- プロセスインスタンスに親がある場合は、親プロセスインスタンス ID を返します。

  ```
  System.out.println(kcontext.getProcessInstance().getParentProcessInstanceId());
  ```

- プロセスインスタンスに関連するプロセス定義の ID を返します。

  ```
  System.out.println(kcontext.getProcessInstance().getProcessId());
  ```

- プロセスインスタンスに関連するプロセス定義の名前を返します。

  ```
  System.out.println(kcontext.getProcessInstance().getProcessName());
  ```

- プロセスインスタンスの状態を返します。

  ```
  System.out.println(kcontext.getProcessInstance().getState());
  ```

アクションスクリプトにプロセス変数を設定するには、**kcontext.setVariable("VARIABLE_NAME", "VALUE")** を使用します。

第8章 タイマー

タイマーを使用して、特定の期間後にロジックをトリガーするか、または一定の間隔で特定のアクションを繰り返すことができます。タイマーは、1回または繰り返しトリガーするまで、事前定義の時間を待ちます。

8.1. RED HAT PROCESS AUTOMATION MANAGER でサポート対象のタイマー

Red Hat Process Automation Manager は、2 種類のタイマーをサポートします。

- Quartz: Spring Boot と Tomcat での使用を推奨しています。
- EJB: オンプレミスおよび Red Hat OpenShift Container Platform 両方で Red Hat JBoss EAP と使用する場合に推奨しています。

注記

以下のビジネスストラテジーにはタイマーを使用しないでください。

- プロセスポーリングストラテジーとしてタイマーを使用しないでください。たとえば、外部サービスを直接呼び出して1秒タイマーを追加する代わりに、Fuse を使用して非同期ルートを登録します。Fuse コールバックを使用して、想定していた応答を受け取ると、プロセスが次に移動するイベントを実行します。Business Process Model and Notation (BPMN) プロセスモデル以外のビジネスストラテジーの一部としてワークアイテムハンドラーを作成し、タイマーをワークアイテムハンドラーに移動します。
- プロセスの実行中にセーフポイントやコミットを強制的に実行するタイマーを使用しないでください。タイマーは、ビジネスプロセスで期間 (duration) を表すように設計されていますが、エンジン固有の動作を強制的に実行するためのものではありません。

8.2. 遅延と期間を使用したタイマーの設定

遅延と特定の期間を使用してタイマーを設定できます。遅延は、ノードのアクティブ化後の待機時間を指定し、期間は後続のトリガーのアクティベーションの間隔を定義します。期間の値が **0** の場合は、1回限りのタイマーが作成されます。遅延および期間式は、**[#d][#h][#m][#s][#[ms]]** 形式で指定できます。これは、日数、時間、分、秒、およびミリ秒 (デフォルト) を示します。たとえば、**1h** の式は、タイマーを再度トリガーするまで1時間待機する時間を示します。

8.3. ISO-8601 の日付形式を使用したタイマーの設定

1回限りのタイマーと繰り返し可能なタイマーの両方をサポートする ISO-8601 日付形式でタイマーを設定できます。タイマーは、日時表現、期間、または間隔として定義できます。以下に例を示します。

- 日付 **2020-12-24T20:00:00.000+02:00** 記号は、タイマーが 8:00 p.m を完全にトリガーすることを表します。
- 期間 **PT1S** 記号で、1秒後にタイマーが1回トリガーされます。
- 繰り返し間隔 **R/PT1S** は、タイマーが制限なしで1秒ごとにトリガーされることを示します。タイマー **R5/PT1S** は、1秒ごとに5回トリガーします。

8.4. プロセス変数でのタイマーの設定

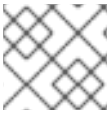
また、遅延と期間の文字列表現、または ISO8601 日付形式で設定されるプロセス変数を使用してタイマーを指定することもできます。**#{variable}** を指定する場合、エンジンは式を解析し、式の値を変数に置き換えます。プロセスでは、以下の方法でタイマーを使用できます。

- タイマーイベントをプロセスフローに追加します。プロセスのアクティブ化によりタイマーが開始し、タイマーがトリガーされると (1 回または繰り返し)、タイマーノードの後続ノードがアクティブ化されます。その後、正の期間値を持つタイマーの送信接続が複数回トリガーされます。タイマーノードがキャンセルされると、関連付けられたタイマーもキャンセルされ、トリガーはこれ以上発生しません。
- タイマーを境界イベントとしてサブプロセスまたはタスクに関連付けます。

8.5. 実行中のプロセスインスタンス内でのタイマー更新

場合によっては、遅延、期間、または制限の繰り返しなど、新しい要件を満たすようにスケジュールされたタイマーを再スケジュールする必要があります。タイマーの更新には多くの低レベルの操作が含まれるため、Red Hat Process Automation Manager は以下のコマンドを実行して、タイマーの更新に関連する低レベルの操作をアトミック操作として実行します。以下のコマンドは、すべての操作が同じトランザクション内で実行されるようにします。

org.jbpm.process.instance.command.UpdateTimerCommand



注記

更新には、境界タイマーイベントと中間タイマーイベントのみがサポートされます。

タイマーを再スケジューリングするには、2 つの必須パラメーターと **UpdateTimerCommand** クラスの 3 つの任意のパラメーターセットを指定します。

表8.1 UpdateTimerCommand クラスのパラメーターおよびパラメーターセット

パラメーターまたはパラメーターセット	タイプ
プロセスインスタンス ID (必須)	**long**
タイマーノード名 (必須)	**String**
遅延 (任意)	**long**
期間 (任意)	**long**
繰り返しの制限 (任意)	**init**

再スケジュール時間イベントの例

```
// Start the process instance and record its ID:
long id = kieSession.startProcess(BOUNDARY_PROCESS_NAME).getId();
```

```
// Set the timer delay to 3 seconds:
kieSession.execute(new UpdateTimerCommand(id, BOUNDARY_TIMER_ATTACHED_TO_NAME,
3));
```

第9章 制約

制約は、制約が含まれる要素が実行される場合に評価されるブール式です。分岐ゲートウェイなど、プロセスのさまざまな場所で制約を使用できます。

Red Hat Process Automation Manager は、以下を含む 2 種類の制約をサポートします。

- **コード制約**: Java、Javascript、Drools、または MVEL で定義される制約。コード制約は、グローバル変数やプロセス変数など、作業メモリー内のデータにアクセスできます。以下のコード制約の例には、プロセスの **person** を変数として含めます。

  Java コードの制約の例

  ```
  return person.getAge() > 20;
  ```

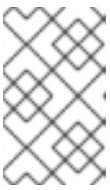
  MVEL コードの制約の例

  ```
  return person.age > 20;
  ```

  Javascript コードの制約の例

  ```
  person.age > 20
  ```

- **ルール制約**: DRL ルール条件の形式で定義される制約。ルール制約は、グローバル変数など、作業メモリー内のデータにアクセスできます。ただし、ルール制約はプロセス内で直接変数にアクセスできず、プロセスインスタンスを使用します。親プロセスインスタンスの参照を取得するには、**WorkflowProcessInstance** タイプの **processInstance** 変数を使用します。

  注記

  必要に応じてプロセスインスタンスをセッションに挿入して更新できます (たとえば、プロセスで Java コードや on-entry、on-exit、または明示的なアクションを使用)。

  以下の例は、プロセス内の **name** 変数の値と同じ名前を持つユーザーを検索するルール制約を示しています。

  プロセス変数の割り当てを伴うルール制約の例

  ```
  processInstance : WorkflowProcessInstance()
  Person( name == ( processInstance.getVariable("name") ) )
  # add more constraints here ...
  ```

第10章 BUSINESS CENTRAL でのビジネスプロセスのデプロイ

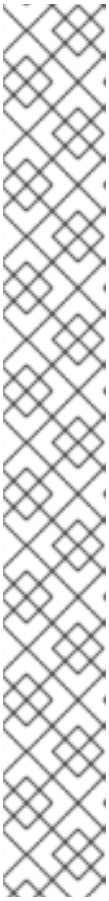
Business Central でビジネスプロセスを設計したら、Business Central でプロジェクトをビルドおよびデプロイして、KIE Server でプロセスを使用できるようにします。

前提条件

- KIE Server がデプロイされて Business Central に接続されている。KIE Server の設定に関する詳細は、**Red Hat JBoss EAP 7.4 への Red Hat Process Automation Manager のインストールおよび設定** を参照してください。

手順

1. Business Central で、**Menu → Design → Projects** に移動します。
2. デプロイするプロジェクトをクリックします。
3. **Deploy** をクリックします。



注記

Build & Install オプションを選択してプロジェクトをビルドし、KJAR ファイルを KIE Server にデプロイせずに設定済みの Maven リポジトリーに公開することもできます。開発環境では、**Deploy** をクリックすると、ビルドされた KJAR ファイルを KIE Server に、実行中のインスタンス (がある場合はそれ) を停止せずにデプロイできます。または **Redeploy** をクリックして、ビルドされた KJAR ファイルをデプロイしてすべてのインスタンスを置き換えることもできます。次回、ビルドされた KJAR ファイルをデプロイまたは再デプロイすると、以前のデプロイメントユニット (KIE コンテナー) が同じターゲット KIE Server で自動的に更新されます。実稼働環境では **Redeploy** オプションは無効になっており、**Deploy** をクリックして、ビルドされた KJAR ファイルを KIE Server 上の新規デプロイメントユニット (KIE コンテナー) にデプロイすることのみが可能です。

KIE Server の環境モードを設定するには、**org.kie.server.mode** システムプロパティーを **org.kie.server.mode=development** または **org.kie.server.mode=production** に設定します。Business Central の対応するプロジェクトでのデプロイメント動作を設定するには、プロジェクトの **Settings → General Settings → Version** に移動し、**Development Mode** オプションを選択します。デフォルトでは、KIE Server および Business Central のすべての新規プロジェクトは開発モードになっています。**Development Mode** をオンにしたプロジェクトをデプロイしたり、実稼働モードになっている KIE Server に手動で **SNAPSHOT** バージョンの接尾辞を追加したプロジェクトをデプロイしたりすることはできません。

プロジェクトのデプロイメントに関する詳細を確認するには、画面の上部にあるデプロイメントバナーの **View deployment details** か、**Deploy** のドロップダウンメニューをクリックします。このオプションを使用すると、**Menu → Deploy → Execution Servers** ページに移動します。

第11章 BUSINESS CENTRAL でのビジネスプロセスの実行

ビジネスプロセスを含むプロジェクトをビルドおよびデプロイした後、ビジネスプロセスに定義された機能を実行できます。

例として、この手順では Business Central の **Mortgage_Process** のサンプル例を使用します。このシナリオでは、住宅ローンブローカーとして、住宅ローン申請書にデータを入力します。**MortgageApprovalProcess** ビジネスプロセスが実行し、プロジェクトで定義しておいたデシジョンルールに基づいて、申請者が条件に合った頭金を提示したかどうかを判断します。このビジネスプロセスは、ルールのテストを終了するか、続行するために頭金の増額を依頼します。申請書がビジネスルールのテストをパスしたら、銀行の承認者が申請書を見直し、ローンを承認または却下します。

前提条件

- KIE Server がデプロイされて Business Central に接続されている。KIE Server の設定に関する詳細は、**Red Hat JBoss EAP 7.4 への Red Hat Process Automation Manager のインストールおよび設定** を参照してください。

手順

1. Business Central で、**Menu → Projects** に移動して、スペースを選択します。デフォルトのスペースは MySpace です。
2. ウィンドウの右上隅にある **Add Project** の横の矢印をクリックし、**Try Samples** を選択します。
3. **Mortgage_Process** サンプルを選択し、**OK** をクリックします。
4. プロジェクトページで、**Mortgage_Process** を選択します。
5. **Mortgage_Process** ページで、**Build** をクリックします。
6. プロジェクトがビルドされたら、**Deploy** をクリックします。
7. **Menu → Manage → Process Definitions** の順にクリックします。
8. **MortgageApprovalProcess** 行の任意の場所をクリックし、プロセスの詳細を表示します。
9. **Diagram** タブをクリックし、エディターでビジネスプロセスダイアグラムを表示します。
10. **New Process Instance** をクリックすると **Application** フォームが開き、以下の値をフォームフィールドに入力します。
    - Down Payment: **30000**
    - Years of amortization: **10**
    - Name: **Ivo**
    - Annual Income: **60000**
    - SSN: **123456789**
    - Age of property: **8**
    - Address of property: **Brno**

- Locale: **Rural**
- Property Sale Price: **50000**

11. **Submit** をクリックして、新しいプロセスインスタンスを開始します。プロセスインスタンスを開始すると、**Instance Details** ビューが開きます。
12. **Diagram** タブをクリックして、プロセスダイアグラムのプロセスフローを表示します。各タスクを通過した時のプロセスの状態が強調表示されます。
13. **Menu → Manage → Tasks** をクリックします。
この例では、対応するタスクで作業しているユーザーは、以下のグループのメンバーです。
    - approver: **Qualify** タスクの場合
    - broker: **Correct Data** タスクおよび **Increase Down Payment**タスクの場合
    - manager: **Final Approval** タスクの場合
14. 承認者として、**Qualify** タスク情報を確認し、**Claim** をクリックしてから **Start** をクリックしてタスクを開始します。続いて、**Is mortgage application in limit?**を選択し、**Complete** をクリックしてタスクフローを完了します。
15. **Tasks** ページで、**Final Approval** 行の任意の場所をクリックし、**Final Approval** タスクを開きます。
16. **Claim** をクリックして、タスクの担当を要求し、**Complete** をクリックして、ローンの承認プロセスを終了します。

注記

Save ボタンおよび **Release** ボタンは、承認プロセスを中断したり、(フィールド値を待っている場合は) インスタンスを保存したり、別のユーザーが修正するタスクを解除したりするために使用します。

第12章 ビジネスプロセスのテスト

ビジネスプロセスは動的に更新でき、エラーを発生させる可能性があるため、プロセスビジネスのテストは、他の開発アーティファクトと同様のビジネスプロセスライフサイクルの一部でもあります。

ビジネスプロセスのユニットテストにより、特定のユースケースでプロセスが想定通りに動作するようになります。たとえば、特定の入力に基づいて出力をテストできます。単体テストを簡素化するために、Red Hat Process Automation Manager には **org.jbpm.test.JbpmJUnitBaseTestCase** クラスが含まれています。

JbpmJUnitBaseTestCase は、Red Hat Process Automation Manager 関連のテストに使用するベーステストケースクラスとして実行されます。**JbpmJUnitBaseTestCase** は、以下の使用領域を提供します。

- JUnit ライフサイクルメソッド

  表12.1 JUnit ライフサイクルメソッド

メソッド	説明
setUp	このメソッドは **@Before** のアノテーションが付けられます。データソースおよび **EntityManagerFactory** を設定し、シングルトンのセッション ID を削除します。
tearDown	このメソッドは **@After** としてアノテーションが付けられます。履歴を削除し、**EntityManagerFactory** およびデータソースを閉じ、**RuntimeManager** および **RuntimeEngines** を破棄します。

- ナレッジベースおよびナレッジセッション管理メソッド: セッションを作成するには、**RuntimeManager** および **RuntimeEngine** を作成します。以下のメソッドを使用して **RuntimeManager** を作成および破棄します。

  表12.2 RuntimeManager および RuntimeEngine の管理方法

メソッド	説明
createRuntimeManager	特定のアセットセットと選択したストラテジーに対して **RuntimeManager** を作成します。
disposeRuntimeManager	テストの範囲でアクティブな **RuntimeManager** を破棄します。
getRuntimeEngine	指定されたコンテキスト用に新しい **RuntimeEngine** を作成します。

- アサーション: アセットの状態をテストするには、以下のメソッドを使用します。

  表12.3 管理メソッド RuntimeManager および RuntimeEngine

アサーション	説明
assertProcessInstanceActive(long processInstanceId, KieSession ksession)	指定した **processInstanceId** を持つプロセスインスタンスがアクティブかどうかを確認します。
assertProcessInstanceCompleted(long processInstanceId)	指定した **processInstanceId** を持つプロセスインスタンスが完了しているかどうかを確認します。セッション永続性が有効な場合にこのメソッドを使用できます。それ以外の場合は、**assertProcessInstanceNotActive(long processInstanceId, KieSession ksession)** を使用します。
assertProcessInstanceAborted(long processInstanceId)	指定された **processInstanceId** を持つプロセスインスタンスが中断されているかどうかを確認します。セッション永続性が有効な場合にこのメソッドを使用できます。それ以外の場合は、**assertProcessInstanceNotActive(long processInstanceId, KieSession ksession)** を使用します。
assertNodeExists(ProcessInstance process, String… nodeNames)	指定したプロセスに指定されたノードが含まれているかどうかを検証します。
assertNodeActive(long processInstanceId, KieSession ksession, String… name)	指定した **processInstanceId** を持つプロセスインスタンスに、指定されたノード名を持つアクティブなノードが 1 つ以上含まれるかどうかを確認します。
assertNodeTriggered(long processInstanceId, String… nodeNames)	指定したプロセスインスタンスの実行中に、指定した各ノードに対してノードインスタンスがトリガーされているかどうかを確認します。
assertProcessVarExists(ProcessInstance process, String… processVarNames)	指定したプロセスに指定されたプロセス変数が含まれているかどうかを検証します。
assertProcessNameEquals(ProcessInstance process, String name)	指定した名前が指定されたプロセス名と一致するかどうかを確認します。
assertVersionEquals(ProcessInstance process, String version)	指定したプロセスバージョンが指定されたプロセスバージョンと一致するかどうかを確認します。

- ヘルパーメソッド: 以下のメソッドを使用して、永続性を使用または使用しない特定のプロセスセットに対して新しい **RuntimeManager** および **RuntimeEngine** を作成します。永続性の詳細は、Red Hat Process Automation Manager のプロセスエンジン を参照してください。

表12.4 管理メソッド RuntimeManager および RuntimeEngine

メソッド	説明
setupPoolingDataSource	データソースを設定します。
getDs	設定したデータソースを返します。
getEmf	設定済みの **EntityManagerFactory** を返します。
getTestWorkItemHandler	デフォルトのワークアイテムハンドラーに加えて登録できるテストワークアイテムハンドラーを返します。
clearHistory	履歴ログを消去します。

以下の例は、開始イベント、スクリプトタスク、および終了イベントが含まれます。JUnit テスト例では、新規セッションを作成し、**hello.bpmn** プロセスを開始し、プロセスインスタンスが完了し、**StartProcess** ノード、**Hello** ノード、および **EndProcess** ノードが実行されているかどうかを確認します。

図12.1hello.bpmn プロセスの JUnit テストの例

```
public class ProcessPersistenceTest extends JbpmJUnitBaseTestCase {

    public ProcessPersistenceTest() {
        super(true, true);
    }

    @Test
    public void testProcess() {

        createRuntimeManager("hello.bpmn");

        RuntimeEngine runtimeEngine = getRuntimeEngine();

        KieSession ksession = runtimeEngine.getKieSession();

        ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello");

        assertProcessInstanceNotActive(processInstance.getId(), ksession);
```

```
        assertNodeTriggered(processInstance.getId(), "StartProcess", "Hello", "EndProcess");
    }
}
```

JbpmJUnitBaseTestCase は、ユニットテストの一環として事前定義された **RuntimeManager** ストラテジーをすべてサポートします。そのため、1 つのテストの一部として **RuntimeManager** を作成するときに使用されるストラテジーを指定するだけで十分です。以下の例は、ユーザータスクを管理するタスクサービスでの PerProcessInstance ストラテジーの使用を示しています。

```
public class ProcessHumanTaskTest extends JbpmJUnitBaseTestCase {

    private static final Logger logger = LoggerFactory.getLogger(ProcessHumanTaskTest.class);

    public ProcessHumanTaskTest() {
        super(true, false);
    }

    @Test
    public void testProcessProcessInstanceStrategy() {
        RuntimeManager manager = createRuntimeManager(Strategy.PROCESS_INSTANCE,
"manager", "humantask.bpmn");
        RuntimeEngine runtimeEngine = getRuntimeEngine(ProcessInstanceIdContext.get());
        KieSession ksession = runtimeEngine.getKieSession();
        TaskService taskService = runtimeEngine.getTaskService();

        int ksessionID = ksession.getId();
        ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello");

        assertProcessInstanceActive(processInstance.getId(), ksession);
        assertNodeTriggered(processInstance.getId(), "Start", "Task 1");

        manager.disposeRuntimeEngine(runtimeEngine);
        runtimeEngine = getRuntimeEngine(ProcessInstanceIdContext.get(processInstance.getId()));

        ksession = runtimeEngine.getKieSession();
        taskService = runtimeEngine.getTaskService();

        assertEquals(ksessionID, ksession.getId());

        // let John execute Task 1
        List<TaskSummary> list = taskService.getTasksAssignedAsPotentialOwner("john", "en-UK");
        TaskSummary task = list.get(0);
        logger.info("John is executing task {}", task.getName());
        taskService.start(task.getId(), "john");
        taskService.complete(task.getId(), "john", null);

        assertNodeTriggered(processInstance.getId(), "Task 2");

        // let Mary execute Task 2
        list = taskService.getTasksAssignedAsPotentialOwner("mary", "en-UK");
        task = list.get(0);
        logger.info("Mary is executing task {}", task.getName());
        taskService.start(task.getId(), "mary");
        taskService.complete(task.getId(), "mary", null);

        assertNodeTriggered(processInstance.getId(), "End");
```

```
        assertProcessInstanceNotActive(processInstance.getId(), ksession);
    }
}
```

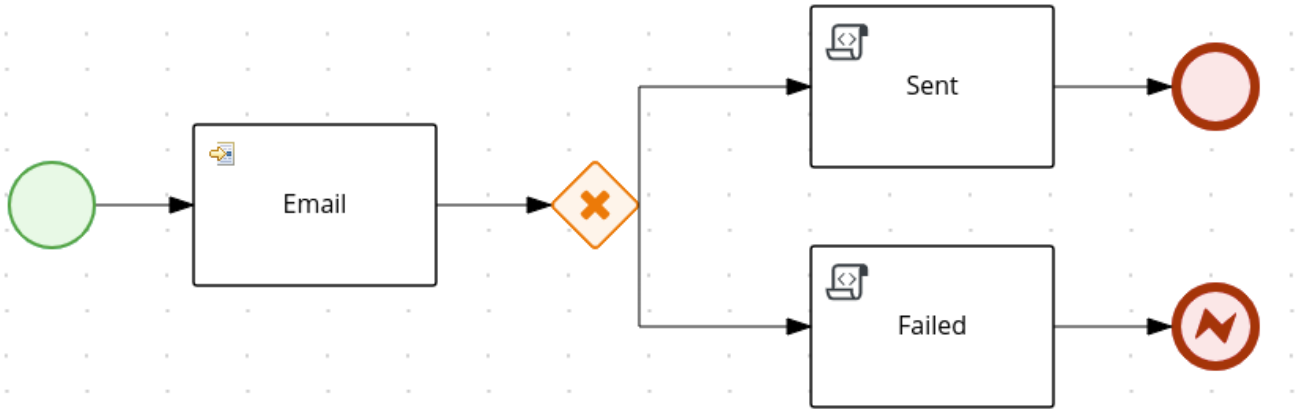
12.1. 外部サービスとの統合テスト

ビジネスプロセスには、多くの場合は、外部サービスの呼び出しが含まれます。ビジネスプロセスのユニットテストでは、特定のサービスが正しく要求されているかどうかを検証するテストハンドラーを登録し、要求されたサービスのテスト応答を提供できます。

外部サービスとの対話をテストするには、デフォルトの **TestWorkItemHandler** ハンドラーを使用します。**TestWorkItemHandler** を登録して、特定タイプのワークアイテムをすべて収集できます。また、**TestWorkItemHandler** にはタスクに関連するデータも含まれます。ワークアイテムは、特定の電子メールを送信したり特定のサービスを呼び出すなど、作業単位 1 つを表します。**TestWorkItemHandler** は、プロセスの実行中に特定のワークアイテムが要求されているかどうかを確認し、関連付けられたデータが正しいことを確認します。

以下の例は、メールタスクを検証する方法と、電子メールが送信されていない場合に例外が発生するかどうかを示しています。ユニットテストは、メールの要求時に実行されるテストハンドラーを使用し、送信者や受信者などの電子メールに関連するデータをテストできます。**abortWorkItem()** メソッドがメール配信の失敗についてエンジンに通知すると、ユニットテストではエラーを生成してアクションをログに記録することにより、プロセスがこのようなケースを処理することを検証します。この場合、プロセスインスタンスは最終的に中止されます。

図12.2 メールプロセスの例




```
public void testProcess2() {

    createRuntimeManager("sample-process.bpmn");

    RuntimeEngine runtimeEngine = getRuntimeEngine();

    KieSession ksession = runtimeEngine.getKieSession();

    TestWorkItemHandler testHandler = getTestWorkItemHandler();

    ksession.getWorkItemManager().registerWorkItemHandler("Email", testHandler);

    ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello2");

    assertProcessInstanceActive(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "StartProcess", "Email");
```

```
    WorkItem workItem = testHandler.getWorkItem();
    assertNotNull(workItem);
    assertEquals("Email", workItem.getName());
    assertEquals("me@mail.com", workItem.getParameter("From"));
    assertEquals("you@mail.com", workItem.getParameter("To"));

    ksession.getWorkItemManager().abortWorkItem(workItem.getId());
    assertProcessInstanceNotActive(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "Gateway", "Failed", "Error");

}
```

第13章 BUSINESS CENTRAL でのプロセス定義とプロセスインスタンス

プロセス定義は、Business Process Model and Notation (BPMN) 2.0 ファイルであり、プロセスとその BPMN ダイアグラムのコンテナーとして機能します。プロセス定義には、関連するサブプロセスや、選択した定義に参加しているユーザーとグループの数など、ビジネスプロセスに関する利用可能な情報がすべて表示されます。

プロセス定義は、プロセス定義が使用するインポートされたプロセスの **import** エントリー、および **relationship** エントリーも定義します。

プロセス定義の BPMN2 ソース

```
<definitions id="Definition"
             targetNamespace="http://www.jboss.org/drools"
             typeLanguage="http://www.java.com/javaTypes"
             expressionLanguage="http://www.mvel.org/2.0"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"Rule Task
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL BPMN20.xsd"
             xmlns:g="http://www.jboss.org/drools/flow/gpd"
             xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
             xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
             xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
             xmlns:tns="http://www.jboss.org/drools">

    <process>
      PROCESS
    </process>

    <bpmndi:BPMNDiagram>
     BPMN DIAGRAM DEFINITION
    </bpmndi:BPMNDiagram>

    </definitions>
```

ビジネスプロセスを含むプロジェクトを作成して設定し、デプロイした後に、Business Central の **Menu → Manage → Process Definitions** ですべてのプロセス定義の一覧を確認できます。右上の更新ボタンをクリックすれば、デプロイしたプロセス定義の一覧をいつでも更新できます。

プロセス定義の一覧には、プラットフォームにデプロイした利用可能なすべてのプロセス定義が表示されます。各プロセス定義をクリックすると、対応するプロセス定義の詳細が表示されます。そのプロセスに関連するサブプロセスが存在するかどうか、プロセス定義にユーザーおよびグループがいくつ存在するかなど、プロセス定義の情報が表示されます。プロセス定義の詳細ページの **ダイアグラム** タブには、プロセス定義の BPMN2 ベースのダイアグラムが含まれます。

選択したプロセス定義内からそれぞれ、右上隅の **New Process Instance** ボタンをクリックして、プロセス定義用の新規プロセスインスタンスを起動できます。利用可能なプロセスから起動したプロセスインスタンスは、**Menu → Manage → Process Instances** に一覧表示されます。

Manage ドロップダウンメニュー (**Process Definition**、**Process Instances**、**Tasks**、**Jobs** および **Execution Errors**) と **Menu → Track → Task Inbox** で、全ユーザーのデフォルトページネーションオプションを定義することも可能です。

Business Central でのプロセスおよびタスクの管理に関する詳細は、**Business Central でのビジネスプロセスの管理とモニタリング** を参照してください。

13.1. プロセス定義ページからのプロセスインスタンスの開始

Menu → Manage → Process Definitions からプロセスインスタンスを起動できます。これは、同時に複数のプロジェクトまたはプロセス定義を使用する環境では有用です。

前提条件

- Business Central に、プロセス定義が設定されたプロジェクトがデプロイされている。

手順

1. Business Central で **Menu → Manage → Process Definitions** に移動します。
2. 一覧から、新しいプロセスインスタンスを開始するプロセス定義を選択します。定義の詳細ページが開きます。
3. 右上隅の **New Process Instance** をクリックし、新しいプロセスインスタンスを開始します。
4. プロセスインスタンスに必要な情報を提供します。
5. **Submit** をクリックして、プロセスインスタンスを作成します。
6. **Menu → Manage → Process Instances** で新規プロセスインスタンスを表示します。

13.2. プロセスインスタンスページからプロセスインスタンスの開始

Menu → Manage → Process Instances で、新規プロセスインスタンスを作成するか、実行中のプロセスインスタンスの全リストを表示することができます。

前提条件

- Business Central に、プロセス定義が設定されたプロジェクトがデプロイされている。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. 右上隅の **New Process Instance** をクリックし、ドロップダウンリストから新しいプロセスインスタンスを開始するプロセス定義を選択します。
3. 新しいプロセスインスタンスを開始するために必要な情報を入力します。
4. **Start** をクリックして、プロセスインスタンスを作成します。
   Manage Process Instances 一覧に新しいプロセスインスタンスが表示されます。

13.3. XML でのプロセス定義

BPMN 2.0 仕様を使用して、XML 形式でプロセスを直接作成できます。これらの XML プロセスの構文は、BPMN 2.0 XML スキーマ定義を使用して定義されます。

プロセス XML ファイルは、以下のコアセクションで設定されます。

- **process**: これは、さまざまなノードとそのプロパティーの定義を含むプロセス XML の最上部になります。プロセス XML ファイルは、1 つの **<process>** 要素で設定されます。この要素には、プロセスに関連するパラメーター (そのタイプ、名前、ID、およびパッケージ名) が含まれ、3 つのサブセクションで設定されます。つまり、変数、グローバル、インポート、レーンなどのプロセスレベル情報が定義されるヘッダーセクション、プロセス内の各ノードを定義するノードセクション、およびプロセス内のすべてのノード間の接続を含む接続セクションの 3 つです。

- **BPMNDiagram**: これは、ノードの場所など、すべての描画情報を含むプロセス XML ファイルの下の部分になります。ノードセクションには、各ノードの特定の要素が含まれ、そのノードタイプのさまざまなパラメーターとサブ要素を定義します。

以下のプロセス XML ファイルのフラグメントは、開始イベント、**"Hello World"** をコンソールに出力するスクリプトタスク、および終了イベントを含むシンプルなプロセスを示しています。

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions
  id="Definition"
  targetNamespace="http://www.jboss.org/drools"
  typeLanguage="http://www.java.com/javaTypes"
  expressionLanguage="http://www.mvel.org/2.0"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL BPMN20.xsd"
  xmlns:g="http://www.jboss.org/drools/flow/gpd"
  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
  xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
  xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
  xmlns:tns="http://www.jboss.org/drools">

  <process processType="Private" isExecutable="true" id="com.sample.hello" name="Hello Process">
    <!-- nodes -->
    <startEvent id="_1" name="Start" />

    <scriptTask id="_2" name="Hello">
      <script>System.out.println("Hello World");</script>
    </scriptTask>

    <endEvent id="_3" name="End" >
      <terminateEventDefinition/>
    </endEvent>

    <!-- connections -->

    <sequenceFlow id="_1-_2" sourceRef="_1" targetRef="_2" />
    <sequenceFlow id="_2-_3" sourceRef="_2" targetRef="_3" />
  </process>

  <bpmndi:BPMNDiagram>
    <bpmndi:BPMNPlane bpmnElement="com.sample.hello" >

      <bpmndi:BPMNShape bpmnElement="_1" >
        <dc:Bounds x="16" y="16" width="48" height="48" />
      </bpmndi:BPMNShape>
```

```
    <bpmndi:BPMNShape bpmnElement="_2" >
      <dc:Bounds x="96" y="16" width="80" height="48" />
    </bpmndi:BPMNShape>

    <bpmndi:BPMNShape bpmnElement="_3" >
      <dc:Bounds x="208" y="16" width="48" height="48" />
    </bpmndi:BPMNShape>

    <bpmndi:BPMNEdge bpmnElement="_1-_2" >
      <di:waypoint x="40" y="40" />
      <di:waypoint x="136" y="40" />
    </bpmndi:BPMNEdge>

    <bpmndi:BPMNEdge bpmnElement="_2-_3" >
      <di:waypoint x="136" y="40" />
      <di:waypoint x="232" y="40" />
    </bpmndi:BPMNEdge>

  </bpmndi:BPMNPlane>
 </bpmndi:BPMNDiagram>

</definitions>
```

第14章 BUSINESS CENTRAL のフォーム

フォームは、HTML として定義されたページのレイアウト定義であり、プロセスおよびタスクのインスタンス化の間にユーザーにダイアログウィンドウとして表示されます。タスクフォームは、プロセスとタスクインスタンスの両方の実行のためにユーザーからデータを取得しますが、プロセスフォームはプロセス変数から入力と出力を受け取ります。

入力は、Data Input Assignment を使用してタスクにマッピングされ、タスク内で使用できます。タスクが完了すると、データは Data Output Assignment としてマッピングされ、データを親プロセスインスタンスに提供します。

14.1. フォームモデラー

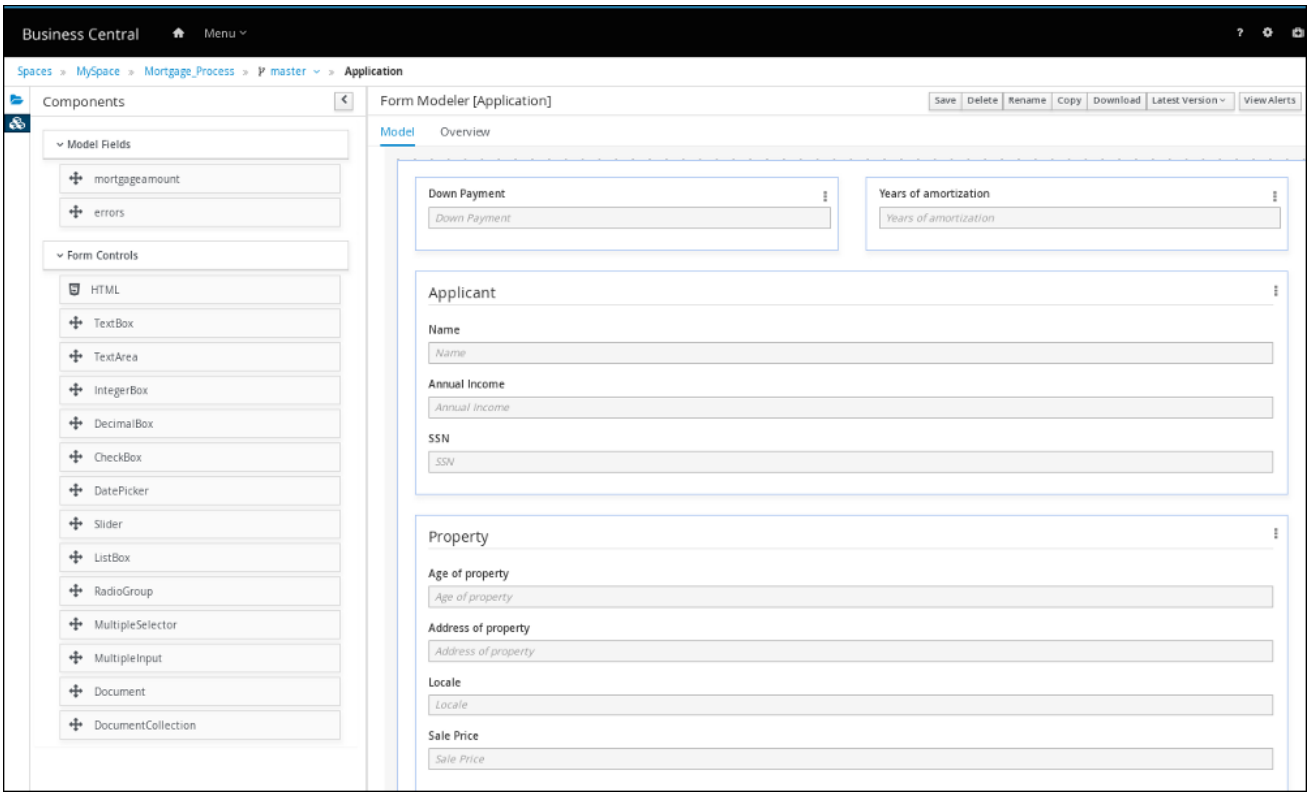
Red Hat Process Automation Manager は、Form Modeler と呼ばれるフォームを定義するためのカスタムエディターを提供します。Form Modeler を使用すると、コードを記述せずに、データオブジェクトのフォーム、タスクフォーム、およびプロセス開始フォームを生成できます。Form Modeler には、複数のデータタイプをバインドするためのウィジェットライブラリーと、フォームの値が変更されたときに通知を送信するコールバックメカニズムが含まれています。Form Modeler は、Bean ベースの検証を使用し、フォームフィールドの静的または動的モデルへのバインドをサポートします。

Form Modeler には以下の機能が含まれています。

- フォームのフォームモデリングユーザーインターフェイス
- データモデルまたは Java オブジェクトからのフォーム自動生成
- Java オブジェクトのデータバインディング
- 公式と式
- カスタマイズされたフォームレイアウト
- 埋め込みフォーム

Form Modeler には、フォームを作成するためにキャンバスに配置する定義済みのフィールドタイプが付属します。

図14.1 住宅ローンの申し込みフォームの例



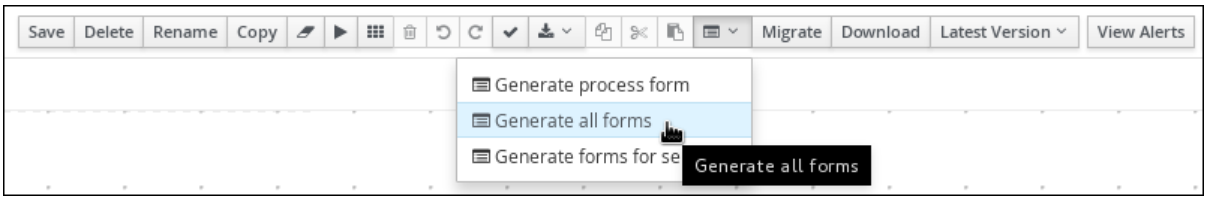
14.2. BUSINESS CENTRAL でのプロセスフォームおよびタスクフォームの生成

プロセスをインスタンス化したときに、プロセスをインスタンス化したユーザーに表示されるビジネスプロセスからプロセスフォームを生成できます。また、ビジネスプロセスからタスクフォームを生成することもできます。このタスクフォームは、実行フローがタスクに到達すると、ユーザータスクのインスタンス化時にユーザータスクのアクターに表示されます。

手順

1. Business Central で、**Menu → Design → Projects** に移動します。
2. プロジェクト名をクリックしてアセットビューを開いてから、ビジネスプロセス名をクリックします。
3. プロセスデザイナーで、フォームを作成するプロセスタスクをクリックします (該当する場合)。
4. 右上のツールバーで、**Form Generation** アイコンをクリックして、生成するフォームを選択します。
   - **Generate process form**: プロセス全体のフォームを生成します。これは、プロセスインスタンスの起動時にユーザーが入力する必要がある初期フォームです。
   - **Generate all forms**: プロセス全体およびすべてのユーザータスクのフォームを生成します。
   - **Generate forms for selection**: 選択したユーザータスクノードのフォームを生成します。

図14.2 フォーム生成メニュー



フォームは、プロジェクトのルートディレクトリーに作成されます。

5. Business Central で、プロジェクトのルートディレクトリーに移動して新しいフォーム名をクリックし、Form Modeler を使用して要件に合わせてフォームをカスタマイズします。

14.3. BUSINESS CENTRAL での手動によるフォームの作成

プロジェクトアセットビューから、タスクおよびプロセスフォームを手動で作成できます。これは、ビジネスプロセスからフォームを生成することを選択せずにフォームを生成する別の方法です。たとえば、Form Modeler は外部データオブジェクトからのフォームの作成をサポートするようになりました。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクト名をクリックします。
2. **Add Asset → Form** をクリックします。
3. **Create new Form** ウィンドウで、以下の情報を入力します。
   - フォーム名 (一意である必要があります)
   - パッケージ名
   - モデルタイプ: **Business Process** または **Data Object** のいずれかを選択します。
     - **Business Process** モデルタイプの場合、**Select Process** ドロップダウンメニューからビジネスプロセスを選択し、**Select Form** ドロップダウンメニューから作成するフォームを選択します。
     - **Data Object** モデルタイプの場合は、**Select Data Object from Project** のドロップダウンメニューから、プロジェクトデータオブジェクトの 1 つを選択します。
4. **OK** をクリックして、Form Modeler を開きます。
5. Form Modeler の左側の **Components** ビューで、**Model Fields** および **Form Controls** メニューを展開し、必要なフィールドとフォームコントロールをキャンバスにドラッグして新しいフォームを作成します。
6. **Save** をクリックして変更を保存します。

14.4. フォームまたはプロセスでのドキュメントの添付

Red Hat Process Automation Manager は、**Document** フォームフィールドを使用したフォームでのドキュメントの添付をサポートしています。**Document** フォームフィールドを使用すると、フォームまたはプロセスの一部として必要なドキュメントをアップロードできます。

フォームおよびプロセスでドキュメントの添付を有効にするには、以下の手順を実行します。

1. ドキュメントマーシャリング戦略を設定します。
2. ビジネスプロセスでドキュメント変数を作成します。
3. タスクの入力と出力をドキュメント変数にマッピングします。

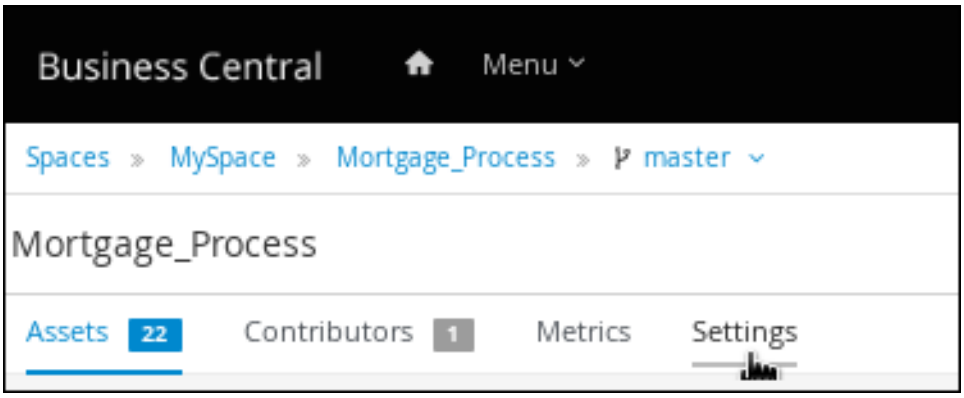
14.4.1. ドキュメントマーシャリングストラテジーの設定

プロジェクトのドキュメントマーシャリングストラテジーにより、フォームおよびプロセスで使用するドキュメントの保存場所が決まります。Red Hat Process Automation Manager のデフォルトのドキュメントマーシャリングストラテジーは **org.jbpm.document.marshalling.DocumentMarshallingStrategy** です。このストラテジーでは、**PROJECT_HOME/.docs** ディレクトリーにドキュメントをローカルに保存する **DocumentStorageServiceImpl** クラスを使用します。Business Central または **kie-deployment-descriptor.xml** ファイルのプロジェクトに、このドキュメントマーシャリングストラテジーまたはカスタムドキュメントマーシャリングストラテジーを設定できます。

手順

1. Business Central で、**Menu → Design → Projects** に移動します。
2. プロジェクトを選択します。プロジェクトの **Assets** ウィンドウが開きます。
3. **Settings** タブをクリックします。

図14.3 Settings タブ



4. **Deployments → Marshalling Strategies→ Add Marshalling Strategy** をクリックします。
5. **Name** フィールドにドキュメントマーシャリングのストラテジーを入力し、**Resolver** のドロップダウンメニューで、対応するリゾルバータイプを選択します。
   - 1 つのドキュメントの場合: ドキュメントマーシャリングストラテジーとして **org.jbpm.document.marshalling.DocumentMarshallingStrategy** を入力し、リゾルバータイプを **Reflection** に設定します。
   - 複数のドキュメントの場合: ドキュメントマーシャリングストラテジーとして **new org.jbpm.document.marshalling.DocumentCollectionImplMarshallingStrategy(new org.jbpm.document.marshalling.DocumentMarshallingStrategy())** を入力し、リゾルバータイプを **MVEL** に設定します。
   - カスタムドキュメントサポートの場合: カスタムドキュメントマーシャリングストラテジーの識別子を入力し、関連するリゾルバータイプを選択します。

6. **Test** をクリックして、デプロイメント記述子ファイルを検証します。

7. **Deploy** をクリックして、更新されたプロジェクトをビルドおよびデプロイします。
   または、Business Central を使用していない場合は、**PROJECT_HOME/src/main/resources/META_INF/kie-deployment-descriptor.xml** (該当する場合) に移動し、必要な **<marshalling-strategies>** 要素を使用してデプロイメント記述子ファイルを編集します。

8. **Save** をクリックします。

複数のドキュメントのドキュメントマーシャリングストラテジーを使用したデプロイメント記述子ファイルの例

```
<deployment-descriptor
    xsi:schemaLocation="http://www.jboss.org/jbpm deployment-descriptor.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <persistence-unit>org.jbpm.domain</persistence-unit>
  <audit-persistence-unit>org.jbpm.domain</audit-persistence-unit>
  <audit-mode>JPA</audit-mode>
  <persistence-mode>JPA</persistence-mode>
  <runtime-strategy>SINGLETON</runtime-strategy>
  <marshalling-strategies>
    <marshalling-strategy>
      <resolver>mvel</resolver>
      <identifier>new org.jbpm.document.marshalling.DocumentCollectionImplMarshallingStrategy(new
org.jbpm.document.marshalling.DocumentMarshallingStrategy());</identifier>
    </marshalling-strategy>
  </marshalling-strategies>
```

14.4.1.1. コンテンツ管理システム (CMS) でのカスタムドキュメントマーシャリングストラテジーの使用

プロジェクトのドキュメントマーシャリングストラテジーにより、フォームおよびプロセスで使用するドキュメントの保存場所が決まります。Red Hat Process Automation Manager のデフォルトのドキュメントマーシャリングストラテジーは **org.jbpm.document.marshalling.DocumentMarshallingStrategy** です。このストラテジーでは、**PROJECT_HOME/.docs** ディレクトリーにドキュメントをローカルに保存する **DocumentStorageServiceImpl** クラスを使用します。集中型のコンテンツ管理システム (CMS) などのカスタムの場所にフォームおよびプロセスドキュメントを保存する場合は、カスタムドキュメントマーシャリングストラテジーをプロジェクトに追加します。このドキュメントマーシャリングストラテジーは、Business Central または **kie-deployment-descriptor.xml** ファイルで直接設定できます。

手順

1. **org.kie.api.marshalling.ObjectMarshallingStrategy** インターフェイスの実装を含むカスタムマーシャリングストラテジー **.java** ファイルを作成します。このインターフェイスを使用すると、カスタムドキュメントマーシャリングストラテジーに必要な変数の永続性を実装できます。
   このインターフェイスの以下のメソッドは、戦略の作成に役立ちます。

   - **boolean accept(Object object)**: 指定されたオブジェクトをストラテジーでマーシャリングできるかどうかを決定します。

- **byte[] marshal(Context context, ObjectOutputStream os, Object object)**: 指定されたオブジェクトをマーシャリングし、マーシャリングされたオブジェクトを **byte[]** として返します。
- **Object unmarshal(Context context, ObjectInputStream is, byte[] object, ClassLoader classloader)**: 受け取ったオブジェクトを **byte[]** として読み取り、マーシャリングされていないオブジェクトを返します。
- **void write(ObjectOutputStream os, Object object)**: 下位互換性のために提供されている **marshal** メソッドと同じです。
- **Object read(ObjectInputStream os)**: 下位互換性のために提供されている **unmarshal** メソッドと同じです。

以下のコードサンプルは、コンテンツ管理相互運用サービス (CMIS) システムからデータを保存および取得するための **ObjectMarshallingStrategy** 実装例です。

CMIS システムからデータを保存および取得するための実装例

```
package org.jbpm.integration.cmis.impl;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.HashMap;

import org.apache.chemistry.opencmis.client.api.Folder;
import org.apache.chemistry.opencmis.client.api.Session;
import org.apache.chemistry.opencmis.commons.data.ContentStream;
import org.apache.commons.io.IOUtils;
import org.drools.core.common.DroolsObjectInputStream;
import org.jbpm.document.Document;
import org.jbpm.integration.cmis.UpdateMode;

import org.kie.api.marshalling.ObjectMarshallingStrategy;

public class OpenCMISPlaceholderResolverStrategy extends OpenCMISSupport implements
ObjectMarshallingStrategy {

 private String user;
 private String password;
 private String url;
 private String repository;
 private String contentUrl;
 private UpdateMode mode = UpdateMode.OVERRIDE;

 public OpenCMISPlaceholderResolverStrategy(String user, String password, String url,
String repository) {
  this.user = user;
  this.password = password;
  this.url = url;
  this.repository = repository;
 }
```

```
 public OpenCMISPlaceholderResolverStrategy(String user, String password, String url,
String repository, UpdateMode mode) {
  this.user = user;
  this.password = password;
  this.url = url;
  this.repository = repository;
  this.mode = mode;
 }

    public OpenCMISPlaceholderResolverStrategy(String user, String password, String url,
String repository, String contentUrl) {
        this.user = user;
        this.password = password;
        this.url = url;
        this.repository = repository;
        this.contentUrl = contentUrl;
    }

    public OpenCMISPlaceholderResolverStrategy(String user, String password, String url,
String repository, String contentUrl, UpdateMode mode) {
        this.user = user;
        this.password = password;
        this.url = url;
        this.repository = repository;
        this.contentUrl = contentUrl;
        this.mode = mode;
    }

 public boolean accept(Object object) {
  if (object instanceof Document) {
   return true;
  }
  return false;
 }

 public byte[] marshal(Context context, ObjectOutputStream os, Object object) throws
IOException {
  Document document = (Document) object;
  Session session = getRepositorySession(user, password, url, repository);
  try {
   if (document.getContent() != null) {
    String type = getType(document);
    if (document.getIdentifier() == null || document.getIdentifier().isEmpty()) {
     String location = getLocation(document);

     Folder parent = findFolderForPath(session, location);
     if (parent == null) {
      parent = createFolder(session, null, location);
     }
     org.apache.chemistry.opencmis.client.api.Document doc = createDocument(session,
parent, document.getName(), type, document.getContent());
     document.setIdentifier(doc.getId());
     document.addAttribute("updated", "true");
    } else {
     if (document.getContent() != null && "true".equals(document.getAttribute("updated"))) {
      org.apache.chemistry.opencmis.client.api.Document doc = updateDocument(session,
```

```
document.getIdentifier(), type, document.getContent(), mode);

      document.setIdentifier(doc.getId());
      document.addAttribute("updated", "false");
     }
    }
   }
   ByteArrayOutputStream buff = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream( buff );
        oos.writeUTF(document.getIdentifier());
        oos.writeUTF(object.getClass().getCanonicalName());
        oos.close();
        return buff.toByteArray();
  } finally {
   session.clear();
  }
 }

 public Object unmarshal(Context context, ObjectInputStream ois, byte[] object, ClassLoader
classloader) throws IOException, ClassNotFoundException {
  DroolsObjectInputStream is = new DroolsObjectInputStream( new ByteArrayInputStream(
object ), classloader );
  String objectId = is.readUTF();
  String canonicalName = is.readUTF();
  Session session = getRepositorySession(user, password, url, repository);
  try {
   org.apache.chemistry.opencmis.client.api.Document doc =
(org.apache.chemistry.opencmis.client.api.Document) findObjectForId(session, objectId);
   Document document = (Document) Class.forName(canonicalName).newInstance();
   document.setAttributes(new HashMap<String, String>());

   document.setIdentifier(objectId);
   document.setName(doc.getName());
   document.setLastModified(doc.getLastModificationDate().getTime());
   document.setSize(doc.getContentStreamLength());
   document.addAttribute("location", getFolderName(doc.getParents()) +
getPathAsString(doc.getPaths()));
   if (doc.getContentStream() != null && contentUrl == null) {
    ContentStream stream = doc.getContentStream();
    document.setContent(IOUtils.toByteArray(stream.getStream()));
    document.addAttribute("updated", "false");
    document.addAttribute("type", stream.getMimeType());
   } else {
      document.setLink(contentUrl + document.getIdentifier());
   }
   return document;
  } catch(Exception e) {
   throw new RuntimeException("Cannot read document from CMIS", e);
  } finally {
   is.close();
   session.clear();
  }
 }

 public Context createContext() {
  return null;
```

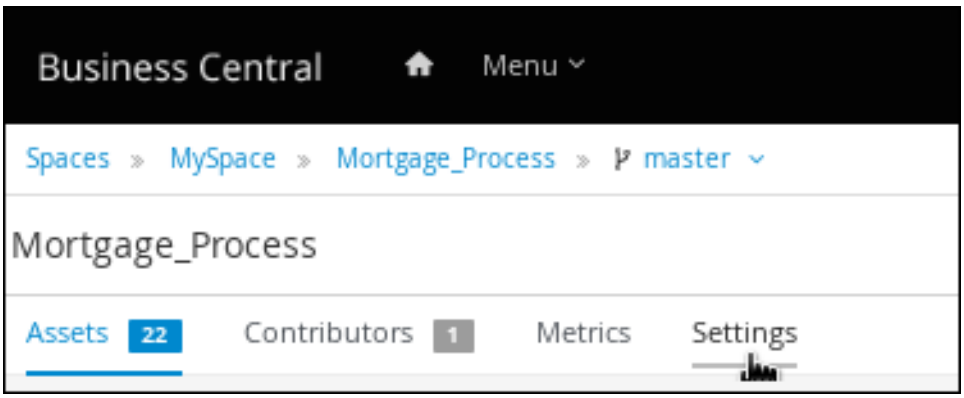
```
 }

 // For backward compatibility with previous serialization mechanism
 public void write(ObjectOutputStream os, Object object) throws IOException {
  Document document = (Document) object;
  Session session = getRepositorySession(user, password, url, repository);
  try {
   if (document.getContent() != null) {
    String type = document.getAttribute("type");
    if (document.getIdentifier() == null) {
     String location = document.getAttribute("location");

     Folder parent = findFolderForPath(session, location);
     if (parent == null) {
      parent = createFolder(session, null, location);
     }
     org.apache.chemistry.opencmis.client.api.Document doc = createDocument(session,
parent, document.getName(), type, document.getContent());
     document.setIdentifier(doc.getId());
     document.addAttribute("updated", "false");
    } else {
     if (document.getContent() != null && "true".equals(document.getAttribute("updated"))) {
      org.apache.chemistry.opencmis.client.api.Document doc = updateDocument(session,
document.getIdentifier(), type, document.getContent(), mode);

      document.setIdentifier(doc.getId());
      document.addAttribute("updated", "false");
     }
    }
   }
   ByteArrayOutputStream buff = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream( buff );
        oos.writeUTF(document.getIdentifier());
        oos.writeUTF(object.getClass().getCanonicalName());
        oos.close();
  } finally {
   session.clear();
  }
 }

 public Object read(ObjectInputStream os) throws IOException, ClassNotFoundException {
  String objectId = os.readUTF();
  String canonicalName = os.readUTF();
  Session session = getRepositorySession(user, password, url, repository);
  try {
   org.apache.chemistry.opencmis.client.api.Document doc =
(org.apache.chemistry.opencmis.client.api.Document) findObjectForId(session, objectId);
   Document document = (Document) Class.forName(canonicalName).newInstance();

   document.setIdentifier(objectId);
   document.setName(doc.getName());
   document.addAttribute("location", getFolderName(doc.getParents()) +
getPathAsString(doc.getPaths()));
   if (doc.getContentStream() != null) {
    ContentStream stream = doc.getContentStream();
    document.setContent(IOUtils.toByteArray(stream.getStream()));
```

```
    document.addAttribute("updated", "false");
    document.addAttribute("type", stream.getMimeType());
   }
   return document;
  } catch(Exception e) {
   throw new RuntimeException("Cannot read document from CMIS", e);
  } finally {
   session.clear();
  }
 }

}
```

2. Business Central で、**Menu → Design → Projects** に移動します。

3. プロジェクトの名前をクリックしてから、**Settings** をクリックします。

図14.4 Settings タブ



4. **Deployments → Marshalling Strategies→ Add Marshalling Strategy**をクリックします。

5. **Name** フィールドに、この例の **org.jbpm.integration.cmis.impl.OpenCMISPlaceholderResolverStrategy** のように、カスタムドキュメントマーシャリングストラテジーの識別子を入力します。

6. この例の **Reflection** のように、**Resolver** ドロップダウンメニューから関連オプションを選択します。

7. **Test** をクリックして、デプロイメント記述子ファイルを検証します。

8. **Deploy** をクリックして、更新されたプロジェクトをビルドおよびデプロイします。
または、Business Central を使用していない場合は、**PROJECT_HOME/src/main/resources/META_INF/kie-deployment-descriptor.xml** (該当する場合) に移動し、必要な **<marshalling-strategies>** 要素を使用してデプロイメント記述子ファイルを編集します。

カスタムドキュメントマーシャリング戦略を使用したデプロイメント記述子ファイルの例

```
<deployment-descriptor
   xsi:schemaLocation="http://www.jboss.org/jbpm deployment-descriptor.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <persistence-unit>org.jbpm.domain</persistence-unit>
  <audit-persistence-unit>org.jbpm.domain</audit-persistence-unit>
  <audit-mode>JPA</audit-mode>
```

```
<persistence-mode>JPA</persistence-mode>
<runtime-strategy>SINGLETON</runtime-strategy>
<marshalling-strategies>
  <marshalling-strategy>
    <resolver>reflection</resolver>
    <identifier>
      org.jbpm.integration.cmis.impl.OpenCMISPlaceholderResolverStrategy
    </identifier>
  </marshalling-strategy>
</marshalling-strategies>
```

9. カスタムの場所に保存されたドキュメントをフォームおよびプロセスにアタッチできるようにするには、関連するプロセスでドキュメント変数を作成し、Business Central でそのドキュメント変数にタスクの入力と出力をマッピングします。

14.4.2. ビジネスプロセスでのドキュメント変数の作成

ドキュメントマーシャリングストラテジーを設定したら、関連プロセスでドキュメント変数を作成して、ドキュメントをヒューマンタスクにアップロードし、Business Central の **Process Instances** ビューにドキュメントを表示できるようにします。

前提条件

- 「ドキュメントマーシャリングストラテジーの設定」の説明に従って、ドキュメントマーシャリング戦略を設定している。

手順

1. Business Central で、**Menu → Design → Projects** に移動します。
2. プロジェクト名をクリックしてアセットビューを開き、ビジネスプロセス名をクリックします。
3. キャンバスをクリックし、ウィンドウの右側にある をクリックして、**Properties** パネルを開きます。
4. **Process Data** を展開し、 をクリックし、以下の値を入力します。
   - Name: **document**
   - Custom Type: 1 つのドキュメントの場合は **org.jbpm.document.Document**、複数のドキュメントの場合は **org.jbpm.document.DocumentCollection**。

14.4.3. ドキュメント変数へのタスクの入力と出力のマッピング

タスクフォーム内の添付ファイルを表示または変更する場合は、タスクの入力および出力の中に割り当てを作成します。

前提条件

- 1 つ以上のユーザータスクを持つビジネスプロセスアセットを含むプロジェクトがある。

手順

1. Business Central で、**Menu → Design → Projects** に移動します。
2. プロジェクト名をクリックしてアセットビューを開き、ビジネスプロセス名をクリックします。
3. ユーザータスクをクリックし、ウィンドウの右側にある をクリックして、**Properties** パネルを開きます。
4. **Implementation/Execution** を展開し、**Assignments** の横にある をクリックして、**Data I/O** ウィンドウを開きます。
5. **Data Inputs and Assignments**の横の **Add** をクリックして、以下の値を入力します。
   - Name: **taskdoc_in**
   - Data Type: 1 つのドキュメントの場合は **org.jbpm.document.Document** で、複数のドキュメントの場合は **org.jbpm.document.DocumentCollection**。
   - Source: **document**
6. **Data Outputs and Assignments** の横の **Add** をクリックし、以下の値を入力します。
   - Name: **taskdoc_out**
   - Data Type: 1 つのドキュメントの場合は **org.jbpm.document.Document** で、複数のドキュメントの場合は **org.jbpm.document.DocumentCollection**。
   - Target: **document**

   Source フィールドおよび **Target** フィールドには、前に作成したプロセス変数の名前が含まれています。
7. **Save** をクリックします。

第15章 詳細にわたるプロセスの概念およびタスク

15.1. ビジネスプロセスで DECISION MODEL AND NOTATION (DMN) サービスを呼び出す

Decision Model and Notation (DMN) を使用して、Business Central の意思決定要件ダイアグラム (DRD) でデシジョンサービスを描画を使ってモデル化し、Business Central のビジネスプロセスの一環としてその DMN サービスを呼び出すことができます。ビジネスプロセスは、DMN サービスを識別し、DMN 入力とビジネスプロセスプロパティー間でビジネスデータをマッピングすることにより、DMN サービスと対話します。

例として、この手順では、列車の経路ロジックを定義する **TrainStation** プロジェクトの例を使用します。このサンプルプロジェクトには、経路決定ロジック用に Business Central で設計された以下のデータオブジェクトと DMN コンポーネントが含まれています。

Train オブジェクトの例

```
public class Train {

    private String departureStation;

    private String destinationStation;

    private BigDecimal railNumber;

    // Getters and setters
}
```

図15.1 Compute Rail DMN モデルの例

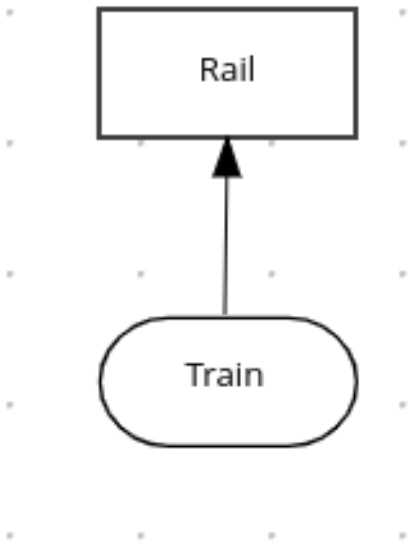
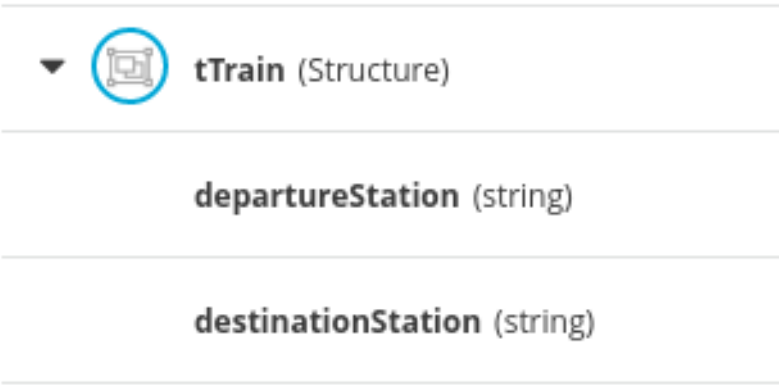


図15.2 Rail DMN デシジョンテーブルの例

Rail *(Decision Table)*

F	Train.departureStation *(string)*	Train.destinationStation *(string)*	Rail *(number)*	Description
1	"Prague"	"Hamburg"	5	
2	"Prague"	"Krakow"	2	
3	-	"Belgrade"	1	Just one option to Belgrade
4	"Zagreb"	-	3	Just one option from Zagreb
5	"Graz"	"Vienna"	1	

図15.3 tTrain DMN データタイプの例



Business Central での DMN モデルの作成方法に関する詳細は、**DMN モデルを使用したデシジョンサービスの作成** を参照してください。

前提条件

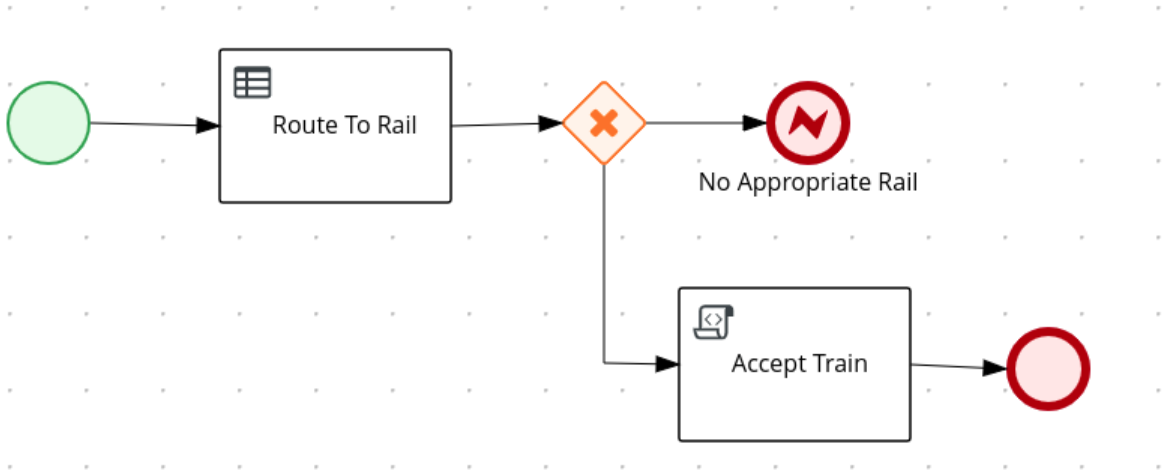
- 必要なすべてのデータオブジェクトと DMN モデルコンポーネントは、プロジェクトで定義済みである。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクト名をクリックします。
2. DMN サービスを呼び出すビジネスプロセスアセットを選択または作成します。
3. プロセスデザイナーで、左側のツールバーを使用して通常どおりに BPMN コンポーネントをドラッグアンドドロップし、ビジネスプロセスロジック、接続、イベント、タスク、またはその他の要素全体を定義します。

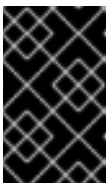
4. ビジネスプロセスに DMN サービスを組み込むには、左側のツールバーまたは開始ノードオプションから **Business Rule** タスクを追加し、プロセスフローの関連する場所にタスクを挿入します。
この例では、以下の **Accept Train** ビジネスプロセスは、DMN サービスを **Route To Rail** ノードに組み込みます。

図15.4 DMN サービスを使用した Accept Train ビジネスプロセスの例



5. DMN サービスに使用するビジネスルールタスクノードを選択し、プロセスデザイナーの右上隅にある **Properties** をクリックしてから、**Implementation/Execution** で以下のフィールドを定義します。

- **Rule Language**: **DMN** を選択します。
- **Namespace**: DMN モデルファイルから一意の名前空間を入力します。例: **https://www.drools.org/kie-dmn**
- **Decision Name**: 選択したプロセスノードで呼び出す DMN デシジョンノードの名前を入力します。例: **Rail**
- **DMN Model Name**: DMN モデル名を入力します。例: **Compute Rail**

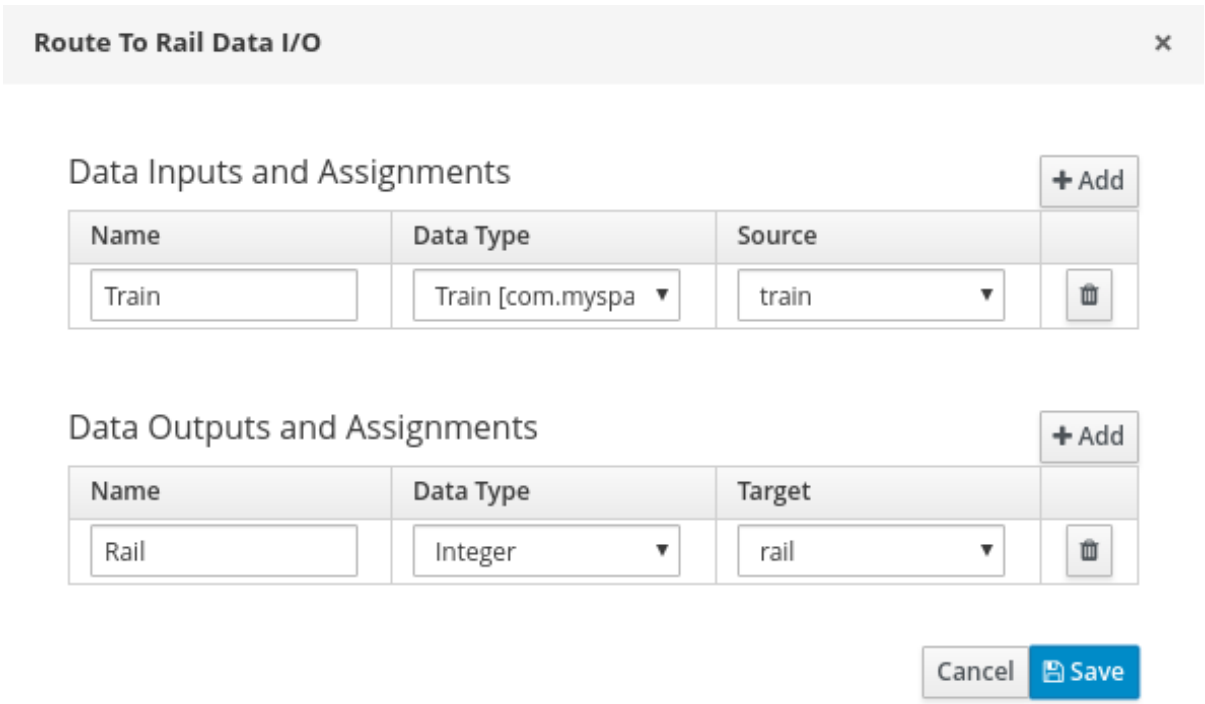


重要

root ノードを使用する場合は、**Namespace** フィールドと **DMN Model Name** フィールドに DMN ダイアグラムと同じ BPMN の値で設定されるようにします。

6. **Data Assignments → Assignments** 配下で **Edit** アイコンをクリックし、DMN の入力および出力データを追加して、DMN サービスとプロセスデータ間のマッピングを定義します。
この例の **Route To Rail** DMN サービスノードの場合は、DMN モデルの入力ノードに対応する **Train** の入力割り当てを追加し、DMN モデルのデシジョンノードに対応する **Rail** の出力割り当てを追加します。**Data Type** は、DMN モデルでそのノードに設定したタイプに一致する必要があり、**Source** および **Target** の定義は、指定されたオブジェクトに関連する変数またはフィールドです。

図15.5 Route To Rail DMN サービスノードの入力および出力のマッピングの例



7. **Save** をクリックして、入力データおよび出力データを保存します。

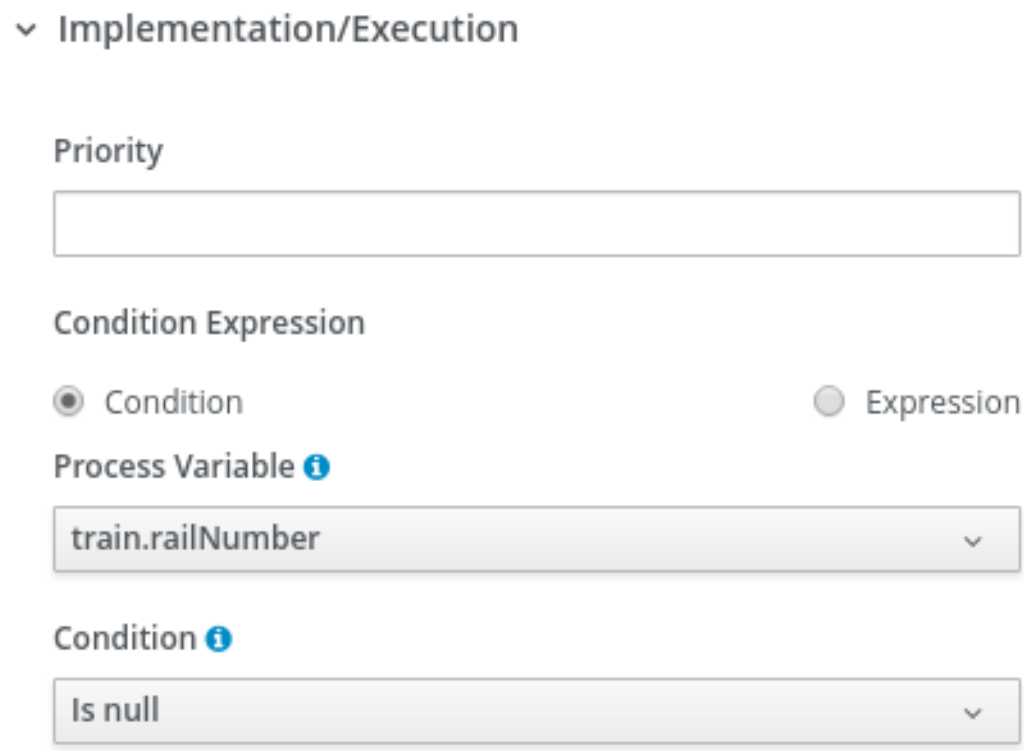
8. 完了した DMN サービスをどのように処理するかに応じて、ビジネスプロセスの残りの部分を定義します。
   この例では、**Properties → Implementation/Execution → On Exit Action** の値が以下のコードに設定され、**Route To Rail** DMN サービスの完了後にレール番号を保存します。

   On Exit Action のサンプルコード

   ```
   train.setRailNumber(rail);
   ```

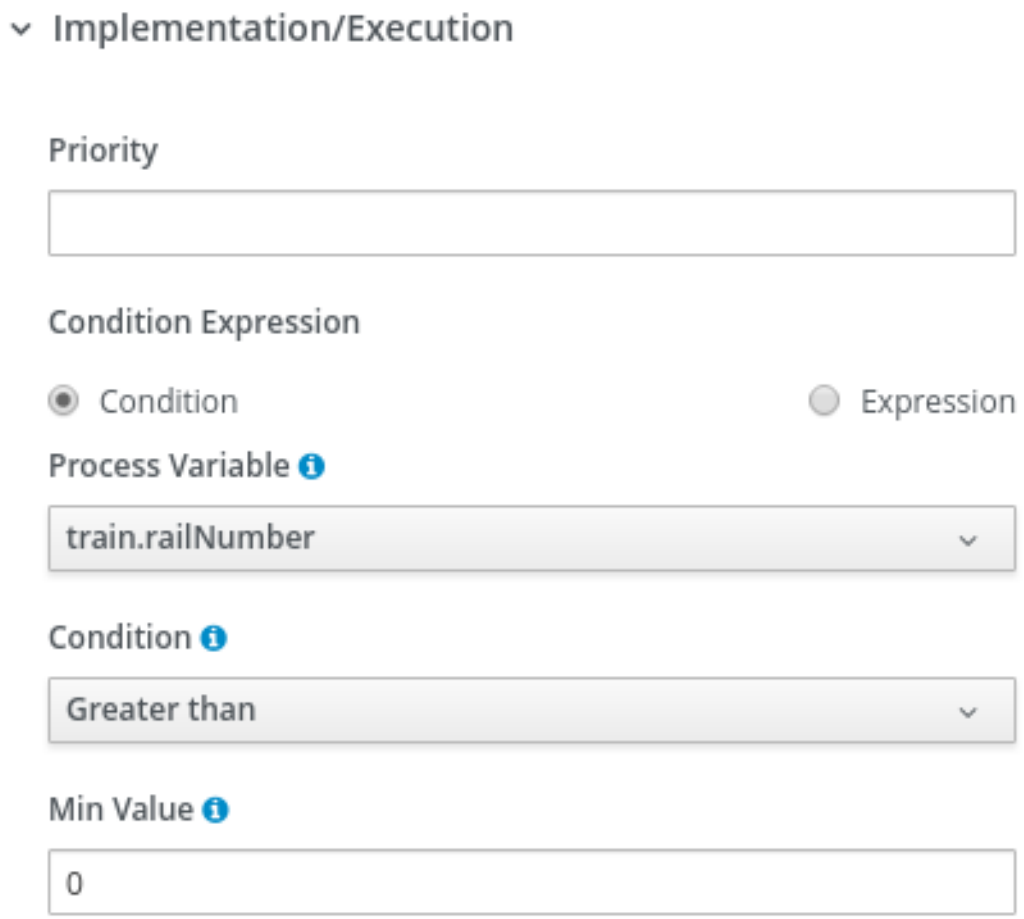
   レール番号が計算されない場合、プロセスは、以下の条件式で定義された **No Appropriate Rail** 終了エラーノードに到達します。

図15.6 No Appropriate Rail 終了エラーノードの条件の例



レール番号が計算されると、プロセスは、以下の条件式で定義された **Accept Train** スクリプトタスクに到達します。

図15.7 Accept Train スクリプトタスクノードの条件の例

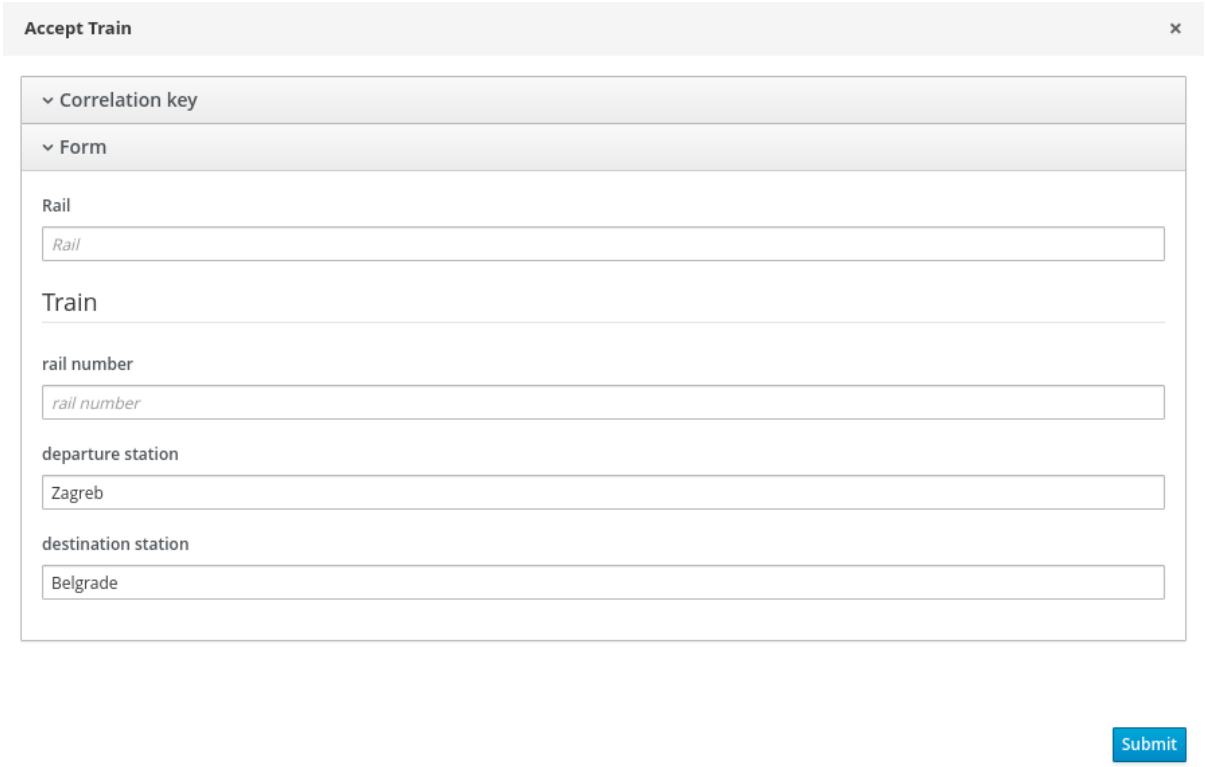


Accept Train スクリプトタスクは、**Properties → Implementation/Execution → Script** で、以下のスクリプトも使用して、列車のルートと現在のレールに関するメッセージを出力します。

```
com.myspace.trainstation.Train t =
    (com.myspace.trainstation.Train) kcontext.getVariable("train");
System.out.println("Train from: " + t.getDepartureStation() +
                   ", to: " + t.getDestinationStation() +
                   ",  is on rail: " + t.getRailNumber());
```

9. 組み込まれた DMN サービスでビジネスプロセスを定義した後、プロセスデザイナーでプロセスを保存してプロジェクトをデプロイし、対応するプロセス定義を実行して DMN サービスを呼び出します。
この例では、**TrainStation** プロジェクトをデプロイし、対応するプロセス定義を実行する際に、**Accept Train** プロセス定義のプロセスインスタンスフォームを開いて **departure station** フィールドおよび **destination station** フィールドを設定し、実行をテストします。

図15.8 Accept Train プロセス定義のプロセスインスタンスフォームの例



プロセスが実行されると、サーバーログに指定した列車のルートを示すメッセージが表示されます。

Accept Train プロセスのサーバーログ出力の例

```
Train from: Zagreb, to: Belgrade,  is on rail: 1
```

第16章 関連情報

- プロセスサービスのスタートガイド
- Business Central でのビジネスプロセスの管理とモニターリング
- プロセスおよびタスクとの対話

パート II. プロセスおよびタスクとの対話

ナレッジワーカーは、Red Hat Process Automation Manager で Business Central を使用して、シチズンデベロッパーが開発したビジネスプロセスアプリケーションのプロセスやタスクを実行します。ビジネスプロセスとは、プロセスフローで定義されているとおりに、実行する一連の手順のことです。プロセスとタスクを効率的に対話するには、ビジネスプロセスを明確に理解し、プロセスまたはタスクの現在の手順が何かを判断できなければなりません。タスクを開始および停止できます。タスクとプロセスインスタンスを検索およびフィルターリングします。タスクの委任、要求、および解放を行います。タスクの期日と優先順位を設定します。タスクを表示してコメントを追加します。タスク履歴ログを表示します。

前提条件

- Red Hat Process Automation Manager をインストールしている。インストールオプションは、**Red Hat Process Automation Manager インストールの計画**を参照してください。

第17章 BUSINESS CENTRAL のビジネスプロセス

Business Central では、シチズンデベロッパーが作成したビジネスプロセスアプリケーションは、ビジネスプロセスのフローを一連のステップとして表します。各ステップは、プロセスのフローチャートに従って実行されます。プロセスは、小さい個別タスク 1 つまたは複数で設定されます。ナレッジワーカーが、ビジネスプロセスの実行時に発生するプロセスおよびタスクに取り組みます。

たとえば、金融機関の住宅ローン部門が、Red Hat Process Automation Manager を使用して、住宅ローンのビジネスプロセスをすべて自動化できます。新しい住宅ローンの依頼が入ると、住宅ローンアプリケーションに新しいプロセスインスタンスが作成されます。要求はすべて、同じルールセットに従い処理されるため、全ステップで一貫性が確保されます。これにより、処理時間と労力を削減する効率的なプロセスが実現されます。

17.1. ナレッジワーカーのユーザー

金融機関で住宅ローンの依頼を処理する顧客アカウント担当者の例を考えてみましょう。顧客アカウント担当者は、次のタスクを実行できます。

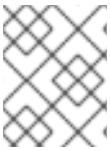
- 住宅ローンの依頼を承認および拒否する
- 依頼を検索およびフィルターリングする
- 依頼を委任、要求、およびリリースする
- 依頼の期日と優先順位を設定する
- 依頼を表示してコメントを追加する
- 依頼の履歴ログを表示する

第18章 BUSINESS CENTRAL でのナレッジワーカーのタスク

タスクは、特定のユーザーが一要求して実行できるビジネスプロセスフローの一部を指します。タスクの処理は、Business Central の **Menu → Track → Task Inbox** から行います。タスク受信箱には、ログイン中のユーザーのタスクリストが表示されます。タスクは、特定のユーザー、複数のユーザー、ユーザーのグループに割り当てることができます。タスクが複数のユーザーまたはユーザーグループに割り当てられる場合は、全ユーザーのタスクリストに表示され、誰でもそのタスクを要求できます。タスクがユーザーから要求された場合は、他のユーザーのタスクリストからそのタスクが削除されます。

18.1. タスクの開始

Business Central の **Menu → Manage → Tasks** および **Menu → Track → Task Inbox** でユーザータスクを開始できます。



注記

ログインされていること、タスクの開始/停止の適切な権限が割り当てられていることを確認してください。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページで、タスクをクリックして開きます。
3. タスクページの **Work** タブで、**Start** をクリックします。タスクを開始したら、タスクのステータスが **InProgress** に変わります。
   タスクのステータスは、**Task Inbox** と **Manage Tasks** ページで、確認できます。

注記

process-admin ロールが割り当てられたユーザーのみが、**Manage Tasks** ページのタスク一覧を表示できます。**admin** ロールを持つユーザーは、**Manage Tasks** ページにアクセスできますが、空のタスクリストのみが表示されます。

18.2. タスクの停止

Tasks および **Task Inbox** ページからユーザータスクを停止できます。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページで、タスクをクリックして開きます。
3. タスクページの **Work** タブで、**Complete** をクリックします。

18.3. タスクの委譲

Business Central でタスクを作成したら、他のユーザーにタスクを委譲できます。

注記

ロールを割り当てられたユーザーは、そのユーザーに表示されているタスクを委譲、要求、またはリリースできます。**Task Inbox** ページの **Actual Owner** 列には、対象のタスクの現在の所有者名が表示されます。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページで、タスクをクリックして開きます。
3. タスクページで、**Assignments** タブをクリックします。
4. **User** フィールドに、タスクを委譲するユーザーまたはグループの名前を入力します。
5. **Delegate** をクリックします。タスクが委譲されたら、タスクの所有者が変更されます。

18.4. タスクの要求

Business Central でタスクを作成した後には、リリースされたタスクを要求できます。ユーザーは、自分が所属するグループに、タスクが割り当てられている場合にのみ、**Task Inbox** ページからタスクを要求できます。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページで、タスクをクリックして開きます。
3. タスクページの **Work** タブで、**Claim** をクリックします。
4. リリースされたタスクを **Task Inbox** ページから要求するには、次のタスクのいずれかを実行します。
   - **Actions** 列の 3 つの点をクリックし、**Claim** を選択します。
   - **Actions** 列の 3 つの点をクリックし、**Claim and Work** を選択してタスクの詳細を表示、確認、変更します。

タスクを要求したユーザーがタスクの所有者になります。

18.5. タスクのリリース

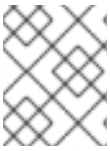
Business Central でタスクを作成したら、他のユーザーが要求できるように、タスクをリリースできます。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページで、タスクをクリックして開きます。
3. タスクページで、**Release** をクリックします。リリースされたタスクには所有者が割り当てられていません。

18.6. タスクの一括アクション

Business Central の **Tasks** および **Task Inbox** ページでは、1回の操作で複数のタスクに対して一括アクションを実行できます。



注記

指定された一括アクションがタスクステータスに基づいて許可されていない場合は、通知が表示され、そのタスクでは操作は実行されません。

18.6.1. タスクを一括で要求する

Business Central でタスクを作成した後、使用可能なタスクを一括して要求できます。

手順

1. Business Central で、以下のいずれかの手順を実行します。
   - **Menu → Track → Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
   - **Menu → Manage → Tasks** の順に選択すると、**Tasks** ページが表示されます。
2. タスクを一括して請求するには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** ドロップダウンリストから、**Bulk Claim** を選択します。
4. 確認するには、**Claim selected tasks** ウィンドウで、**Claim** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

18.6.2. タスクを一括でリリースする

所有しているタスクを一括してリリースし、他のユーザーが要求できるようにすることができます。

手順

1. Business Central で、以下のいずれかの手順を実行します。
   - **Menu → Track → Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
   - **Menu → Manage → Tasks** の順に選択すると、**Tasks** ページが表示されます。
2. タスクを一括してリリースするには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** のドロップダウンリストから、**Bulk Release**を選択します。
4. 確認するには、**Release selected tasks** ウィンドウで、**Release** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

18.6.3. タスクを一括で再開する

Business Central に一時停止されたタスクがある場合は、それらを一括して再開できます。

手順

1. Business Central で、以下のいずれかの手順を実行します。
   - **Menu → Track → Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
   - **Menu → Manage → Tasks** の順に選択すると、**Tasks** ページが表示されます。
2. タスクを一括して再開するには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** のドロップダウンリストから、**Bulk Resume** を選択します。
4. 確認するには、**Resume selected tasks** ウィンドウで **Resume** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

18.6.4. タスクを一括で一時停止する

Business Central でタスクを作成した後、タスクを一括して一時停止できます。

手順

1. Business Central で、以下のいずれかの手順を実行します。
   - **Menu → Track → Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
   - **Menu → Manage → Tasks** の順に選択すると、**Tasks** ページが表示されます。
2. タスクを一括して一時停止するには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** のドロップダウンリストから、**Bulk Suspend** を選択します。
4. 確認するには、**Suspend selected tasks** ウィンドウで **Suspend** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

18.6.5. タスクを一括で再割り当てする

Business Central でタスクを作成した後、タスクを一括して再割り当てし、他のユーザーに委任できます。

手順

1. Business Central で、以下のいずれかの手順を実行します。
   - **Menu → Track → Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
   - **Menu → Manage → Tasks** の順に選択すると、**Tasks** ページが表示されます。
2. タスクを一括して再割り当てするには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** のドロップダウンリストから、**Bulk Reassign** を選択します。

4. **Tasks reassignment** ウィンドウで、タスクを再割り当てするユーザーのユーザー ID を入力します。

5. **Delegate** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

第19章 BUSINESS CENTRAL でのタスクのフィルターリング

Business Central には、タスクの検索ができるように、フィルター機能が含まれています。**Status**、**Filter By**、**Process Definition Id**、**Created On** などの属性でタスクをフィルターリングできます。**Advanced Filters** オプションを使用して、カスタムのタスクフィルターを作成することもできます。新規作成されたカスタムのフィルターが、**Saved Filters** ペインに追加されます。このペインには、**Task Inbox** ページの左側にある星のアイコンをクリックしてアクセスできます。

19.1. タスクリストの列の管理

Task Inbox ウィンドウおよび **Manage Tasks** ウィンドウのタスクリストで、表示する列を指定し、列の順序を変更してタスク情報をより適切に管理することができます。

注記

process-admin ロールが割り当てられたユーザーのみが、**Manage Tasks** ページのタスク一覧を表示できます。**admin** ロールを持つユーザーは、**Manage Tasks** ページにアクセスできますが、空のタスクリストのみが表示されます。

手順

1. Business Central で **Menu → Manage → Tasks** に移動するか、**Menu → Track → Task Inbox** に移動します。
2. **Manage Task** ページまたは **Task Inbox** ページで、**Bulk Actions** の右側にある **Show/hide columns** アイコンをクリックします。
3. 表示する列を選択または選択解除します。リストに変更を加えると、タスクリストの列が表示されたり、消えたりします。
4. 列を再配置するには、列のヘディングを新しい位置にドラッグします。列をドラッグする前に、ポインターを以下の図に示すアイコンに変更する必要があることに注意してください。

5. 変更をフィルターとして保存するには、**Save Filters** をクリックして名前を入力し、**Save** をクリックします。
6. 新しいフィルターを使用するには、画面左側にある **Saved Filters** アイコン (スター) をクリックし、リストからフィルターを選択します。

19.2. 基本的なフィルターを使用したタスクのフィルターリング

Business Central には、**Status**、**Filter By**、**Process Definition Id**、および **Created On** などの属性をもとに、タスクのフィルターリングや検索を行う基本的なフィルターリング機能が含まれています。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページの左側にあるフィルターアイコンをクリックして、**Filters** ペインを展開し、使用するフィルターを選択します。

- **Status**: ステータスをもとにタスクをフィルターリングします。
- **Filter By**: **Id**、**Task**、**Correlation Key**、**Actual Owner**、または **Process Instance Description** 属性をもとにタスクをフィルターリングします。
- **Process Definition Id**: プロセス定義 ID をもとにタスクをフィルターリングします。
- **Created On**: 作成日をもとにタスクをフィルターリングします。

Advanced Filters オプションを使用して、Business Central でカスタムフィルターを作成できます。

19.3. 詳細フィルターを使用したタスクのフィルターリング

Business Central で **Advanced Filters** オプションを使用して、カスタムタスクフィルターを作成できます。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページの左側にある詳細フィルターアイコンをクリックして、**Advanced Filters** ペインを展開します。
3. **Advanced Filters** パネルで、フィルター名と説明を入力して、**Add New** をクリックします。
4. **Select column** ドロップダウンリストから **Name** などの属性を選択します。ドロップダウンの内容が **Name != value1**に変わります。
5. もう一度ドロップダウンをクリックして、必要な論理クエリーを選択します。**Name** 属性については、**equals to** を選択してください。
6. フィルターリングするタスク名の横にあるテキストフィールドの値を変更します。

注記

名前は、プロジェクトのビジネスプロセスで定義した値と一致させる必要があります。

7. **Save** をクリックして、フィルター定義に従い、タスクをフィルターリングします。
8. 星のアイコンをクリックして、**Saved Filters** ペインを開きます。
Saved Filters ペインで、保存した詳細フィルターを表示できます。

19.4. デフォルトのフィルターを使用したタスクの管理

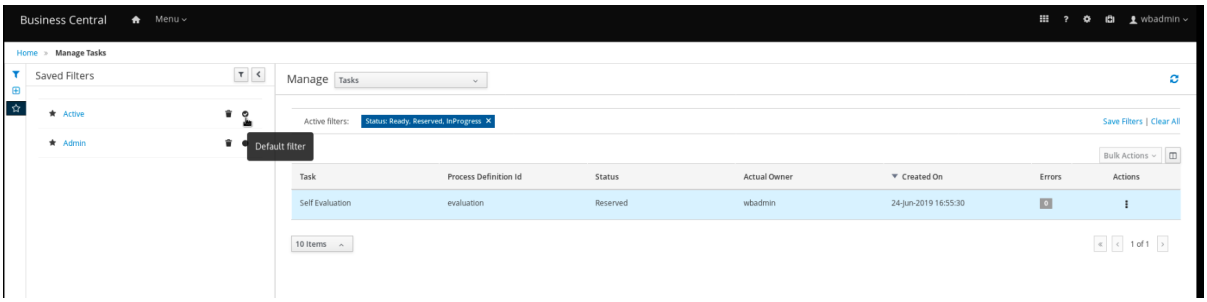
Business Central の **Saved Filter** オプションを使用して、タスクフィルターをデフォルトフィルターとして設定できます。ユーザーがページを開くたびにデフォルトのフィルターリングが実行されます。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動するか、**Menu → Manage → Tasks** に移動します。

2. **Task Inbox** ページまたは、**Manage Tasks** ページの左側にある星アイコンをクリックして、**Saved Filters** パネルを展開します。
Saved Filters パネルで、保存した詳細フィルターを表示できます。

Tasks または Task Inbox のデフォルトフィルターのオプション



3. **Saved Filters** パネルで、保存したタスクフィルターを、デフォルトのフィルターとして設定します。

19.5. 基本的なフィルターを使用したタスク変数の表示

Business Central には基本フィルターがあり、**Manage Tasks** および **Task Inbox** のタスク変数を表示できます。タスクのタスク変数は、**Show/hide columns** を使用し列として表示できます。

手順

1. Business Central で **Menu** → **Manage** → **Tasks** に移動するか、**Menu** → **Track** → **Task Inbox** に移動します。

2. **Task Inbox** ページの左側にあるフィルターアイコンをクリックして、**Filters** パネルを展開します。

3. **Filters** パネルを選択し、**Task Name** をクリックします。
フィルターは現在のタスクリストに適用されます。

4. タスクリストの右上にある **Show/hide columns** をクリックすると、指定のタスク ID のタスク変数が表示されます。

5. 星のアイコンをクリックして、**Saved Filters** パネルを開きます。
Saved Filters パネルで、保存した詳細フィルターをすべて表示できます。

19.6. 詳細フィルターを使用したタスク変数の表示

Business Central の **Advanced Filters** オプションを使用して **Manage Tasks** と **Task Inbox** のタスク変数を表示できます。タスクを定義したフィルターを作成すると、**Show/hide columns** でタスクのタスク変数を列として表示できます。

手順

1. Business Central で **Menu** → **Manage** → **Tasks** に移動するか、**Menu** → **Track** → **Task Inbox** に移動します。

2. **Manage Tasks** ページ、または **Task Inbox** ページの詳細フィルターアイコンをクリックして、**Advanced Filters** パネルを展開します。

3. **Advanced Filters** パネルで、フィルターの名前と説明を入力して、**Add New** をクリックします。

4. **Select column** リストから **name** 属性を選択します。この値は、 **name != value1** に変わります。

5. **Select column** リストから論理クエリーに **equals to** を選択します。

6. テキストフィールドに、タスク名を入力します。

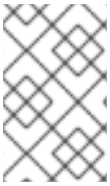
7. **Save** をクリックして、フィルターを現在のタスクリストに適用します。

8. タスクリストの右上にある **Show/hide columns** をクリックすると、指定のタスク ID のタスク変数が表示されます。

9. 星のアイコンをクリックして、**Saved Filters** パネルを開きます。
Saved Filters パネルで、保存した詳細フィルターをすべて表示できます。

第20章 BUSINESS CENTRAL でのプロセスインスタンスのフィルターリング

Business Central には、プロセスインスタンスの検索とフィルターができるように、基本フィルターと詳細フィルターが追加されました。**State**、**Errors**、**Filter By**、**Name**、**Start Date**、および **Last update** などの属性でプロセスをフィルターリングできます。**Advanced Filters** オプションを使用してカスタムのフィルターを作成することも可能です。新規作成したカスタムフィルターは **Saved Filters** ペインに追加されます。このペインには、**Manage Process Instances** ページの左側にある星のアイコンをクリックしてアクセスできます。



注記

manager ロールまたは **rest-all** ロールが割り当てられたユーザー以外はすべて、Business Central でプロセスインスタンスにアクセスしてフィルターリング機能を使用できます。

20.1. 基本フィルターを使用したプロセスインスタンスのフィルターリング

Business Central には、**State**、**Errors**、**Filter By**、**Name**、**Start Date**、**Last update** などの属性をもとにプロセスインスタンスをフィルターリングして検索する基本フィルター機能が含まれます。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Manage Process Instances** ページで、ページの左側にあるフィルターアイコンをクリックして、**Filters** ペインを展開して、使用するフィルターを選択します。
   - **State**: 状態をもとにプロセスインスタンスをフィルターリングします (**Active**、**Aborted**、**Completed**、**Pending**、および **Suspended**)。
   - **Errors**: エラーが最低 1 つ含まれるプロセスインスタンス、またはエラーが含まれないプロセスインスタンスをフィルターリングします。
   - **Filter By**: **Id**、**Initiator**、**Correlation Key**、または **Description** 属性をもとにプロセスインスタンスをフィルターリングします。
   - **Name**: プロセス定義名をもとにプロセスインスタンスをフィルターリングします。
   - **Definition ID**: インスタンス定義の ID。
   - **Deployment ID**: インスタンスデプロイメントの ID。
   - **SLA Compliance**: SLA コンプライアンスのステータス (**Aborted**、**Met**、**N/A**、**Pending**、および **Violated**)。
   - **Parent Process ID**: 親プロセスインスタンスの ID です。
   - **Start Date**: 作成日をもとにプロセスインスタンスをフィルターリングします。
   - **Last update**: 最終変更日をもとにプロセスインスタンスをフィルターリングします。

Advanced Filters オプションを使用して、Business Central でカスタムフィルターを作成することもできます。

20.2. 詳細フィルターを使用したプロセスインスタンスのフィルターリング

Business Central で **Advanced Filters** オプションを使用して、カスタムプロセスのインスタンスフィルターを作成できます。

手順

1. Business Central で、**Menu → Manage → Process Instances** の順にクリックします。
2. **Manage Process Instances** ページで、**Advanced Filters** をクリックします。
3. **Advanced Filters** ペインで、フィルターの名前と説明を入力して、**Add New** をクリックします。
4. **Select column** ドロップダウンリストから **processName** などの属性を選択します。ドロップダウンの内容が **processName != value1** に変わります。
5. もう一度ドロップダウンをクリックして、必要な論理クエリーを選択します。**processName** 属性については、**equals to** を選択してください。
6. フィルターリングするプロセス名の横にあるテキストフィールドの値を変更します。

注記

processName は、プロジェクトのビジネスプロセスで定義した値と一致させる必要があります。

7. **Save** をクリックして、フィルター定義に従い、プロセスをフィルターリングします。
8. 星のアイコンをクリックして、**Saved Filters** ペインを開きます。
Saved Filters ペインで、保存した詳細フィルターをすべて表示できます。

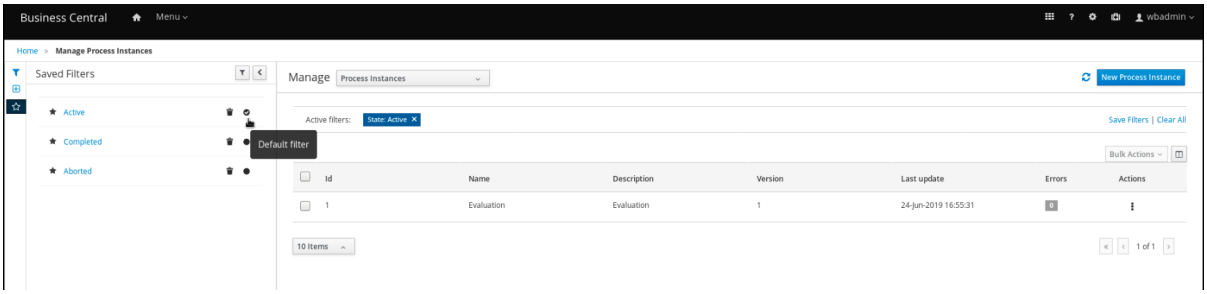
20.3. デフォルトのフィルターを使用したプロセスインスタンスの管理

Business Central の **Saved Filter** オプションを使用して、プロセスインスタンスフィルターをデフォルトフィルターとして設定できます。ユーザーがページを開くたびにデフォルトのフィルターリングが実行されます。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Manage Process Instances** ページの左側にある星アイコンをクリックして、**Saved Filters** ペインを展開します。
Saved Filters パネルで、保存した詳細フィルターを表示できます。

プロセスインスタンスのデフォルトのフィルターオプション



3. **Saved Filters** パネルで、保存したプロセスインスタンスフィルターを、デフォルトのフィルターとして設定します。

20.4. 基本フィルターを使用したプロセスインスタンス変数の表示

Business Central には、プロセスインスタンス変数を表示する基本フィルターが含まれます。プロセスのプロセスインスタンス変数は、**Show/hide columns** を使用して列として表示できます。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Manage Process Instances** ページの左側にあるフィルターアイコンをクリックして、**Filters** パネルを展開します。
3. **Filters** パネルを選択し、**Definition Id** をクリックします。
   フィルターは、現在のプロセスインスタンスリストに適用されます。
4. プロセスインスタンスリストの右上にある **Show/hide columns** をクリックすると、指定のプロセスインスタンス ID のプロセスインスタンス変数が表示されます。
5. 星のアイコンをクリックして、**Saved Filters** パネルを開きます。
   Saved Filters パネルで、保存した詳細フィルターをすべて表示できます。

20.5. 詳細フィルターを使用したプロセスインスタンス変数の表示

Business Central の **Advanced Filters** オプションを使用して、プロセスインスタンス変数を表示できます。**processId** の列のフィルターを作成した場合には、**Show/hide columns** を使用してプロセスのプロセスインスタンス変数を列として表示できます。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Manage Process Instances** ページで、詳細フィルターのアイコンをクリックして **Advanced Filters** パネルを展開します。
3. **Advanced Filters** パネルで、フィルターの名前と説明を入力して、**Add New** をクリックします。
4. **Select column** リストから **processId** 属性を選択します。この値は、**processId != value1** に変わります。
5. **Select column** リストから論理クエリーに **equals to** を選択します。
6. テキストフィールドに、プロセス ID 名を入力します。

7. **Save** をクリックして、フィルターを現在のプロセスインスタンス一覧に適用します。

8. プロセスインスタンスリストの右上にある **Show/hide columns** をクリックすると、指定のプロセスインスタンス ID のプロセスインスタンス変数が表示されます。

9. 星のアイコンをクリックして、**Saved Filters** パネルを開きます。
Saved Filters パネルで、保存した詳細フィルターをすべて表示できます。

第21章 タスク通知でのメールの設定

以前のリリースでは、Business Central のユーザーまたはグループにのみ通知を送信できました。これで、メールアドレスも直接追加できるようになりました。

前提条件

Business Central でプロジェクトを作成している。

手順

1. ビジネスプロセスを作成します。
   Business Central でのビジネスプロセスの作成に関する詳細は、*5章Business Central でのビジネスプロセスの作成* を参照してください。
2. ユーザータスクを作成します。
   Business Central でユーザータスクを作成する方法は、「ユーザータスクの作成」を参照してください。
3. 画面の右上隅で、**Properties** アイコンをクリックします。
4. **Implementation/Execution** を展開し、**Notifications** の横にある をクリックして **Notifications** ウィンドウを開きます。
5. **Add** をクリックします。
6. **Notifications** ウィンドウで **To: email(s)** フィールドにメールアドレスを入力し、タスクの通知メールの受信者を設定します。
   コンマで区切られた複数のメールアドレスを追加できます。
7. メールの件名と本文を入力します。
8. **OK** をクリックします。
   追加されたメールアドレスは、**Notifications** ウィンドウで **To: email(s)** 列に表示されます。
9. **OK** をクリックします。

第22章 タスクの期日と優先順位の設定

Task Inbox ページから、Business Central のタスクの優先順位、期日、および時間を設定できます。タスクの優先順位と期日の設定権限は、全ユーザーに割り当てられるわけではない点に注意してください。

手順

1. Business Central で **Menu** → **Track** → **Task Inbox** に移動します。
2. **Task Inbox** ページで、タスクをクリックして開きます。
3. タスクページで、**Details** タブをクリックします。
4. **Due Date** フィールドで、カレンダーから必要な日付を選択し、ドロップダウンリストから期日を選択します。
5. **Priority** フィールドで、必要な優先順位を選択します。
6. **Update** をクリックします。

第23章 タスクに対するコメントの表示および追加

Business Central では、タスクにコメントを追加したり、タスクの既存のコメントを表示したりできます。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページで、タスクをクリックして開きます。
3. タスクページの **Work** タブまたは、**Comments** タブをクリックします。
4. **Comment** フィールドに、タスク関連のコメントを入力し、**Add Comment** アイコンをクリックします。
   タスク関連のコメントはすべて、**Work** と **Comments** タブにテーブル形式で表示されます。

注記

Show task comments at work tabチェックボックスを選択または選択解除するには、Business Central ホームページに移動し、**Settings** アイコンをクリックして、**Process Administration** オプションを選択します。**admin** ロールが割り当てられたユーザーのみが、この機能を有効または無効にするアクセス権があります。

第24章 タスクの履歴ログの表示

タスクの **Logs** タブから、Business Central のタスクの履歴ログを表示できます。履歴ログでは、イベントが 日付: タスクイベントの形式で表示されます。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページで、タスクをクリックして開きます。
3. タスクページで、**Logs** タブをクリックします。
   タスクのライフサイクル中に発生するイベントはすべて、**Logs** タブに表示されます。

第25章 プロセスインスタンスの履歴ログの表示

Logs タブで、Business Central のプロセスインスタンスの履歴ログを表示できます。ログでは、全イベントが **Date Time: Event Node Type: Event Type** 形式で表示されます。

Event Node Type および **Event Type** をもとに、ログをフィルターリングできます。また、ログでヒューマンノードの詳細を表示することもできます。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Process Instances** ページで、表示するログのプロセスインスタンスをクリックします。
3. インスタンスページで、**Logs** タブをクリックします。
4. 必要に応じて、**Event Node Type** ペインおよび **Event Type** ペインから必要なチェックボックスを選択して、ログをフィルターリングします。
5. ヒューマンノードに関する追加情報を表示するには、**Details** を展開します。
6. **Reset** をクリックして、フィルターの選択肢をデフォルトの設定に戻します。
プロセスインスタンスのライフサイクル中に発生するイベントはすべて、**Logs** タブに表示されます。

パート III. BUSINESS CENTRAL でのビジネスプロセスの管理および監視

プロセス管理者は、Red Hat Process Automation Manager の Business Central を使用して、多くのプロジェクトで実行しているプロセスインスタンスとタスクを管理および監視できます。Business Central から、新しいプロセスインスタンスを開始し、全プロセスインスタンスのステータスを確認し、プロセスを中止できます。プロセスに関連付けられているジョブとタスクの一覧を表示したり、プロセスエラーを理解して対応できます。

前提条件

- Red Hat JBoss Enterprise Application Platform 7.4 がインストールされている。インストールの詳細は、Red Hat JBoss Enterprise Application Platform 7.4 Installation Guideを参照してください。
- Red Hat Process Automation Manager をインストールしている。詳細は、Red Hat Process Automation Manager インストールの計画 を参照してください。
- Red Hat Process Automation Manager が稼働し、**process-admin** ロールで Business Central にログインできる。詳細は、Red Hat Process Automation Manager インストールの計画を参照してください。

第26章 プロセスの監視

Red Hat Process Automation Manager は、ビジネスプロセスのリアルタイム監視を提供します。以下のことができるようになります。

- ビジネスマネージャーは、リアルタイムでプロセスを監視できる。
- 顧客は、要求の現在のステータスを監視できる。
- 管理者は、プロセス実行に関するエラーを簡単に監視できる。

第27章 BUSINESS CENTRAL でのプロセス定義とプロセスインスタンス

プロセス定義は、Business Process Model and Notation (BPMN) 2.0 ファイルであり、プロセスとその BPMN ダイアグラムのコンテナーとして機能します。プロセス定義には、関連するサブプロセスや、選択した定義に参加しているユーザーとグループの数など、ビジネスプロセスに関する利用可能な情報がすべて表示されます。

プロセス定義は、プロセス定義が使用するインポートされたプロセスの **import** エントリー、および **relationship** エントリーも定義します。

プロセス定義の BPMN2 ソース

```
<definitions id="Definition"
             targetNamespace="http://www.jboss.org/drools"
             typeLanguage="http://www.java.com/javaTypes"
             expressionLanguage="http://www.mvel.org/2.0"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"Rule Task
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL BPMN20.xsd"
             xmlns:g="http://www.jboss.org/drools/flow/gpd"
             xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
             xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
             xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
             xmlns:tns="http://www.jboss.org/drools">

  <process>
    PROCESS
  </process>

  <bpmndi:BPMNDiagram>
   BPMN DIAGRAM DEFINITION
  </bpmndi:BPMNDiagram>

  </definitions>
```

ビジネスプロセスを含むプロジェクトを作成して設定し、デプロイした後に、Business Central の **Menu → Manage → Process Definitions** ですべてのプロセス定義の一覧を確認できます。右上の更新ボタンをクリックすれば、デプロイしたプロセス定義の一覧をいつでも更新できます。

プロセス定義の一覧には、プラットフォームにデプロイした利用可能なすべてのプロセス定義が表示されます。各プロセス定義をクリックすると、対応するプロセス定義の詳細が表示されます。そのプロセスに関連するサブプロセスが存在するかどうか、プロセス定義にユーザーおよびグループがいくつ存在するかなど、プロセス定義の情報が表示されます。プロセス定義の詳細ページの **ダイアグラム** タブには、プロセス定義の BPMN2 ベースのダイアグラムが含まれます。

選択したプロセス定義内からそれぞれ、右上隅の **New Process Instance** ボタンをクリックして、プロセス定義用の新規プロセスインスタンスを起動できます。利用可能なプロセスから起動したプロセスインスタンスは、**Menu → Manage → Process Instances** に一覧表示されます。

Manage ドロップダウンメニュー (**Process Definition**、**Process Instances**、**Tasks**、**Jobs** および **Execution Errors**) と **Menu → Track → Task Inbox** で、全ユーザーのデフォルトページネーションオプションを定義することも可能です。

27.1. プロセス定義ページからのプロセスインスタンスの開始

Menu → Manage → Process Definitions からプロセスインスタンスを起動できます。これは、同時に複数のプロジェクトまたはプロセス定義を使用する環境では有用です。

前提条件

- Business Central に、プロセス定義が設定されたプロジェクトがデプロイされている。

手順

1. Business Central で **Menu → Manage → Process Definitions** に移動します。
2. 一覧から、新しいプロセスインスタンスを開始するプロセス定義を選択します。定義の詳細ページが開きます。
3. 右上隅の **New Process Instance** をクリックし、新しいプロセスインスタンスを開始します。
4. プロセスインスタンスに必要な情報を提供します。
5. **Submit** をクリックして、プロセスインスタンスを作成します。
6. **Menu → Manage → Process Instances** で新規プロセスインスタンスを表示します。

27.2. プロセスインスタンスページからプロセスインスタンスの開始

Menu → Manage → Process Instances で、新規プロセスインスタンスを作成するか、実行中のプロセスインスタンスの全リストを表示することができます。

前提条件

- Business Central に、プロセス定義が設定されたプロジェクトがデプロイされている。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. 右上隅の **New Process Instance** をクリックし、ドロップダウンリストから新しいプロセスインスタンスを開始するプロセス定義を選択します。
3. 新しいプロセスインスタンスを開始するために必要な情報を入力します。
4. **Start** をクリックして、プロセスインスタンスを作成します。
   Manage Process Instances 一覧に新しいプロセスインスタンスが表示されます。

27.3. BUSINESS CENTRAL でのプロセスドキュメントの生成

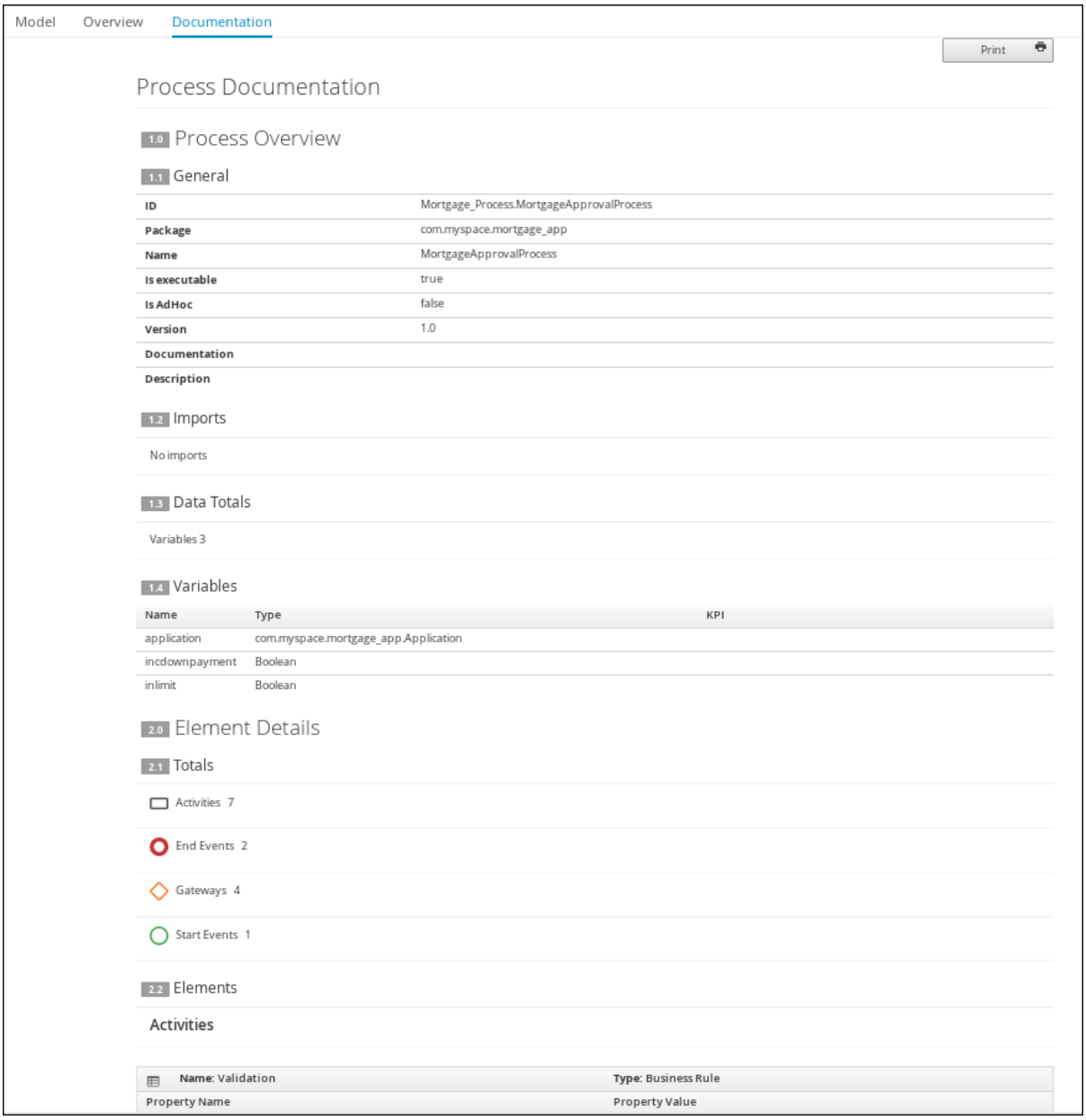
Business Central のプロセスデザイナーでは、プロセス定義のレポートを表示し、出力できます。プロセスドキュメントには、コンポーネント、データ、プロセスの視覚的なフローが簡単に出力して共有できるように PDF 形式でまとめられています。

手順

1. Business Central で、ビジネスプロセスが含まれるプロジェクトに移動し、プロセスを選択します。

2. プロセスデザイナーの **Documentation** タブでプロセスファイルの概要を表示し、画面の右上隅の **Print** をクリックして PDF レポートを出力します。

図27.1 プロセスドキュメントの生成

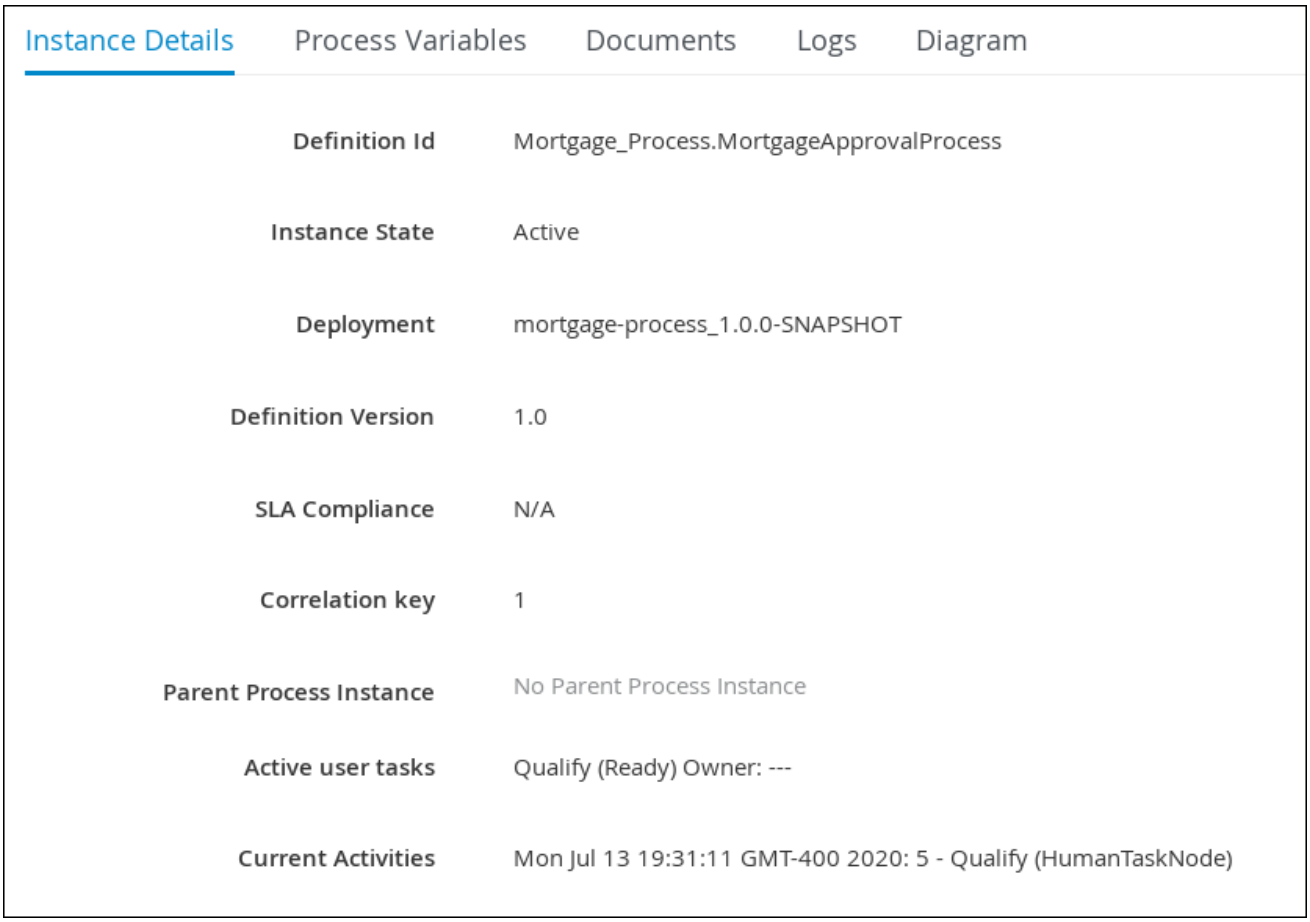


第28章 プロセスインスタンス管理

プロセスインスタンスを表示するには、Business Central で **Menu** → **Manage** → **Process Instances** の順にクリックします。

+ 注記: **Manage Process Instances** 一覧の各行には、特定のプロセス定義のプロセスインスタンスが表示されます。各インスタンスには、プロセスが操作している情報の内部状態があります。プロセスインスタンスをクリックして、プロセスに関連するランタイム情報のある対応するタブを表示します。

図28.1 プロセスインスタンスのタブの表示



- **Instance Details**: プロセス内で何が起きているかについて簡単な概要を表示します。ここには、インスタンスの現在のステータスや、実行中のアクティビティーが表示されます。
- **Process Variables**: インスタンスが操作するすべてのプロセス変数 (ドキュメントを含む変数は除く) を表示します。プロセス変数の値を編集し、その履歴を表示することができます。
- **Documents**: プロセスに **org.jbpm.Document** タイプの変数が含まれる場合は、プロセスのドキュメントを表示します。これにより、添付されるドキュメントへのアクセス、ダウンロード、操作が可能になります。
- **Logs**: エンドユーザーのプロセスインスタンスログを表示します。詳細は、**プロセスおよびタスクとの対話** を参照してください。
- **Diagram**: BPMN2 ダイアグラムを通じてプロセスインスタンスの進行状況を追跡します。進行中のプロセスフローのノードは赤く強調表示されます。再利用可能なサブプロセスは、親プロセス内で縮小表示されます。再利用可能なサブプロセスノードをダブルクリックして、親プロセスダイアグラムからそのダイアグラムを開きます。

KIE Server ランタイムデータへのアクセスに必要なユーザー認証情報および条件の詳細は、Red Hat Process Automation Manager インストールの計画 を参照してください。

28.1. プロセスインスタンスのフィルターリング

Menu → Manage → Process Instances のプロセスインスタンスについては、必要に応じて **Filters** パネルと **Advanced Filters** パネルを使用してプロセスインスタンスを分類してください。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Manage Process Instances** ページで、ページの左側にある **フィルター** アイコンをクリックして、使用するフィルターを選択します。
   - **State**: 状態をもとにプロセスインスタンスをフィルターリングします (**Active**、**Aborted**、**Completed**、**Pending**、および **Suspended**)。
   - **Errors**: エラーが最低 1 つ含まれるプロセスインスタンス、またはエラーが含まれないプロセスインスタンスをフィルターリングします。
   - **Filter By**: 以下の属性に基づいてプロセスインスタンスをフィルターリングします。
     - **Id**: プロセスインスタンス ID 別にフィルターリングします。
       入力: **Numeric**
     - **Initiator**: プロセスインスタンスイニシエーターのユーザー ID でフィルターリングします。
       ユーザー ID は一意の値で、ID 管理システムにより異なります。

       入力: **String**
     - **Correlation key**: 相関キー別にフィルターリングします。
       入力: **String**
     - **Description**: プロセスインスタンスを説明別にフィルターリングします。
       入力: **String**
   - **Name**: プロセス定義名をもとにプロセスインスタンスをフィルターリングします。
   - **Definition ID**: インスタンス定義の ID。
   - **Deployment ID**: インスタンスデプロイメントの ID。
   - **SLA Compliance**: SLA コンプライアンスのステータス (**Aborted**、**Met**、**N/A**、**Pending**、および **Violated**)。
   - **Parent Process ID**: 親プロセスの ID。
   - **Start Date**: 作成日をもとにプロセスインスタンスをフィルターリングします。
   - **Last update**: 最終変更日をもとにプロセスインスタンスをフィルターリングします。

Advanced Filters オプションを使用して、Business Central でカスタムフィルターを作成することもできます。

28.2. カスタムのプロセスインスタンス一覧の作成

Business Central の **Menu → Manage → Process Instances** で、実行中のプロセスインスタンスの一覧を表示できます。このページから、実行中のインスタンスを管理し、実行を監視できます。表示する列や、1 ページに表示する行数をカスタマイズして、結果をフィルターリングできます。カスタムプロセスインスタンス一覧を作成することもできます。

前提条件

- Business Central に、プロセス定義が設定されたプロジェクトがデプロイされている。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Manage Process Instances** ページで、左側の詳細フィルターアイコンをクリックして、プロセスインスタンスの **Advanced Filters** オプション一覧を開きます。
3. **Advanced Filters** パネルで、カスタムのプロセスインスタンスリストに使用するフィルターの名前と説明を入力して、**Add New** をクリックします。
4. フィルターの値のリストから、パラメーターと値を選択し、カスタムのプロセスインスタンスリストを設定して、**Save** をクリックします。
   新しいフィルターが作成され、即座にプロセスインスタンス一覧に適用されます。このフィルターは、**Saved Filters** リストにも保存されます。保存したフィルターには、**Manage Process Instances** ページの左側にある星アイコンをクリックして、アクセスできます。

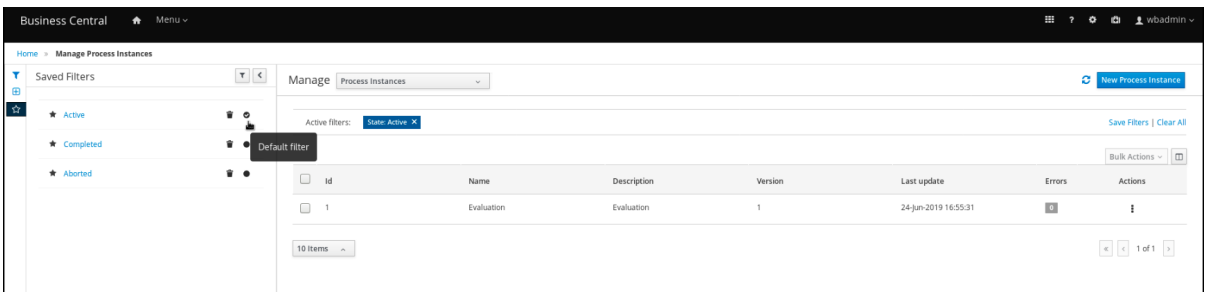
28.3. デフォルトのフィルターを使用したプロセスインスタンスの管理

Business Central の **Saved Filter** オプションを使用して、プロセスインスタンスフィルターをデフォルトフィルターとして設定できます。ユーザーがページを開くたびにデフォルトのフィルターリングが実行されます。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Manage Process Instances** ページの左側にある星アイコンをクリックして、**Saved Filters** ペインを展開します。
   Saved Filters パネルで、保存した詳細フィルターを表示できます。

   プロセスインスタンスのデフォルトのフィルターオプション



3. **Saved Filters** パネルで、保存したプロセスインスタンスフィルターを、デフォルトのフィルターとして設定します。

28.4. 基本フィルターを使用したプロセスインスタンス変数の表示

Business Central には、プロセスインスタンス変数を表示する基本フィルターが含まれます。プロセスのプロセスインスタンス変数は、**Show/hide columns** を使用して列として表示できます。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Manage Process Instances** ページの左側にあるフィルターアイコンをクリックして、**Filters** パネルを展開します。
3. **Filters** パネルを選択し、**Definition Id** をクリックして一覧から定義 ID を選択します。
   フィルターは現在のプロセスインスタンス一覧に適用されます。
4. 画面の右上にあるコラムアイコン (**Bulk Actions** の右横) をクリックして、プロセスインスタンスのテーブルのコラムを表示または非表示します。
5. 星のアイコンをクリックして、**Saved Filters** パネルを開きます。
   Saved Filters パネルで、保存した詳細フィルターをすべて表示できます。

28.5. 詳細フィルターを使用したプロセスインスタンス変数の表示

Business Central の **Advanced Filters** オプションを使用して、プロセスインスタンス変数を表示できます。**processId** の列のフィルターを作成した場合には、**Show/hide columns** を使用してプロセスのプロセスインスタンス変数を列として表示できます。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. **Manage Process Instances** ページで、詳細フィルターのアイコンをクリックして **Advanced Filters** パネルを展開します。
3. **Advanced Filters** パネルで、フィルターの名前と説明を入力して、**Add New** をクリックします。
4. **Select column** リストから **processId** 属性を選択します。この値は、**processId != value1** に変わります。
5. **Select column** リストからクエリーに **equals to** を選択します。
6. テキストフィールドに、プロセス ID 名を入力します。
7. **Save** をクリックして、フィルターを現在のプロセスインスタンス一覧に適用します。
8. プロセスインスタンスリストの右上にあるコラムアイコン (**Bulk Actions** の右側) をクリックすると、指定のプロセスインスタンス ID のプロセスインスタンス変数が表示されます。
9. 星のアイコンをクリックして、**Saved Filters** パネルを開きます。
   Saved Filters パネルで、保存した詳細フィルターをすべて表示できます。

28.6. BUSINESS CENTRAL を使用したプロセスインスタンスの中止

プロセスインスタンスが必要なくなった場合は、Business Central でプロセスインスタンスを中止できます。

手順

1. Business Cnetral で、**Menu → Manage → Process Instances** の順にクリックして、利用可能なプロセスインスタンスの一覧を表示します。
2. 一覧から、中止するプロセスインスタンスを選択します。
3. プロセスの詳細ページの右上にある **Abort** ボタンをクリックします。

28.7. BUSINESS CENTRAL からプロセスインスタンスのシグナル送信

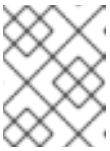
Business Central からプロセスインスタンスのシグナルを送信できます。

前提条件

- Business Central に、プロセス定義が設定されたプロジェクトがデプロイされている。

手順

1. Business Central で、**Menu → Manage → Process Instances** に移動します。
2. 必要なプロセスインスタンスを探し、 ボタンをクリックして、ドロップダウンメニューから **Signal** を選択します。
3. 以下のフィールドに入力します。
   - **Signal Name**: シグナルの **SignalRef** 属性または **MessageRef** 属性に相当します。このフィールドは必須です。



注記

Message イベントをプロセスに送信することもできます。そのためには、**MessageRef** 値に **Message-** 接頭辞を追加します。

   - **Signal Data**: シグナルに添付されるデータに相当します。このフィールドは任意です。

注記

Business Central ユーザーインターフェイスを使用している場合は、シグナル中間キャッチイベントのシグナルだけを送信できます。

28.8. 非同期シグナルイベント

異なるプロセス定義の複数のプロセスインスタンスが同じシグナルを待機している場合に、これらのプロセスインスタンスは同じスレッドで順次実行されます。ただし、このプロセスインスタンスの 1 つがランタイム例外を出力すると、他のすべてのプロセスインスタンスが影響を受け、通常はトランザクションがロールバックされます。この状況を回避するために、Red Hat Process Automation Manager は非同期シグナルイベントを使用します。

- 中間シグナルイベントの出力

- 終了イベント

28.8.1. 中間イベントの非同期シグナルの設定

中間イベントは、ビジネスプロセスのフローを駆動します。中間イベントは、ビジネスプロセスの実行中にイベントをキャッチまたは出力するときに使用します。中間イベントは、プロセス実行時に発生する特定の状況を処理します。出力シグナルの中間イベントは、定義したプロパティーをもとにシグナルオブジェクトを生成します。

Business Central で中間イベントの非同期シグナルを設定できます。

前提条件

- Business Central でプロジェクトを作成済みである。これにはビジネスプロセスアセットが 1 つ以上含まれます。
- Business Central に、プロセス定義が設定されたプロジェクトがデプロイされている。

手順

1. ビジネスプロセスアセットを開きます。
2. プロセスデザイナーキャンバスで、左のツールバーから **Intermediate Signal** をドラッグアンドドロップします。
3. 右上隅の をクリックし、**Properties** パネルを開きます。
4. **Data Assignments** を展開します。
5. **Assignments** サブセクションの下のボックスをクリックします。**Task Data I/O** ダイアログボックスが開きます。
6. **Data Inputs and Assignments**の横にある **Add** をクリックします。
7. **Name** フィールドに、出力されたイベント名を **async** と入力します。
8. **Data Type** と **Source** のフィールドは空白のままにします。
9. **OK** をクリックします。

各セッションに executor サービスが自動的に設定されます。これにより、各プロセスインスタンスが異なるトランザクションで通知されるようになります。

28.8.2. 終了イベントの非同期シグナルの設定

終了イベントは、ビジネスプロセスの完了を示します。なし、および中断終了イベント以外の終了イベントはすべて出力イベントです。出力シグナルの終了イベントは、プロセスまたはサブプロセスフローの終了に使用します。実行フローがこの要素に入ると、実行フローが終了し、**SignalRef** プロパティーで特定されたシグナルを生成します。

Business Central で終了イベントの非同期シグナルを設定できます。

前提条件

- Business Central でプロジェクトを作成済みである。これにはビジネスプロセスアセットが 1 つ以上含まれます。
- Business Central に、プロセス定義が設定されたプロジェクトがデプロイされている。

手順

1. ビジネスプロセスアセットを開きます。
2. プロセスデザイナーキャンバスで、左のツールバーから **End Signal** をドラッグアンドドロップします。
3. 右上隅の をクリックし、**Properties** パネルを開きます。
4. **Data Assignments** を展開します。
5. **Assignments** サブセクションの下のボックスをクリックします。**Task Data I/O** ダイアログボックスが開きます。
6. **Data Inputs and Assignments** の横にある **Add** をクリックします。
7. **Name** フィールドに、出力されたイベント名を **async** と入力します。
8. **Data Type** と **Source** のフィールドは空白のままにします。
9. **OK** をクリックします。

各セッションに executor サービスが自動的に設定されます。これにより、各プロセスインスタンスが異なるトランザクションで通知されるようになります。

28.9. プロセスインスタンスの操作

プロセスインスタンス管理 API は、プロセスエンジンおよび個々のプロセスインスタンスの以下の操作を公開します。

- **get process nodes - by process instance id** プロセスインスタンスに存在する、組み込みの全サブプロセスなど、すべてのノードを返します。指定のプロセスインスタンスからノードを取得して、他の管理操作で使用できるように、ノードが存在し、有効な ID が割り当てられていることを確認する必要があります。
- **cancel node instance - by process instance id and node instance id** プロセスおよびノードインスタンス ID を使用してプロセスインスタンス内のノードインスタンスを取り消します。
- **retrigger node instance - by process instance id and node instance id** アクティブなノードインスタンスを取り消してノードインスタンスを再度発生させ、プロセスとノードインスタンス ID を使用して同じタイプのノードインスタンスを新規作成します。
- **update timer - by process instance id and timer id** タイマーのスケジュール時間からの経過時間をもとに、アクティブなタイマーの有効期限を更新します。たとえば、タイマーが最初に遅延が 1 時間として作成されており、30 分後に遅延を 2 時間に更新設定した場合に、更新してから 1 時間半後に有効期限が切れます。
  - **delay**: タイマーの有効期限が切れてからの期間
  - **period**: 次のサイクルタイマーの有効期限までの間隔

    - **repeat limit** サイクルタイマーの失効回数を指定の数に制限。

- **update timer relative to current time - by process instance id and timer id**現在の時間をもとにアクティブなタイマーの有効期限を更新します。たとえば、タイマーの遅延が 1 時間として最初に作成されており、30 分後に遅延を 2 時間に更新設定した場合に、更新した時間から 2 時間で有効期限が切れます。

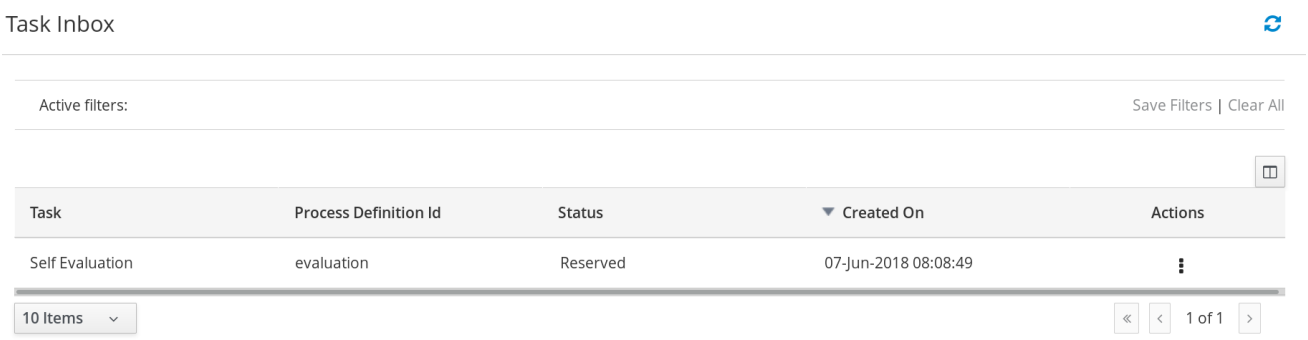
- **list timer instances - by process instance id** 指定のプロセスインスタンスでアクティブなタイマーをすべて返します。

- **trigger node - by process instance id and node id**任意のタイミングでプロセスインスタナス内のノードをトリガーします。

第29章 タスク管理

現在のユーザーに割り当てられたタスクは、Business Cnetral の **Menu → Track → Task Inbox** に表示されます。タスクをクリックして、タスクを開き、作業を開始できます。

特定のユーザー、複数のユーザー、またはグループにユーザータスクを割り当てることができます。複数のユーザー、またはグループに割り当てた場合は、割り当てた全ユーザーのタスク一覧に表示され、タスクの要求が可能なユーザーであれば誰でも要求できます。別のユーザーにタスクが割り当てられると、そのタスクは **Task Inbox** に表示されなくなります。



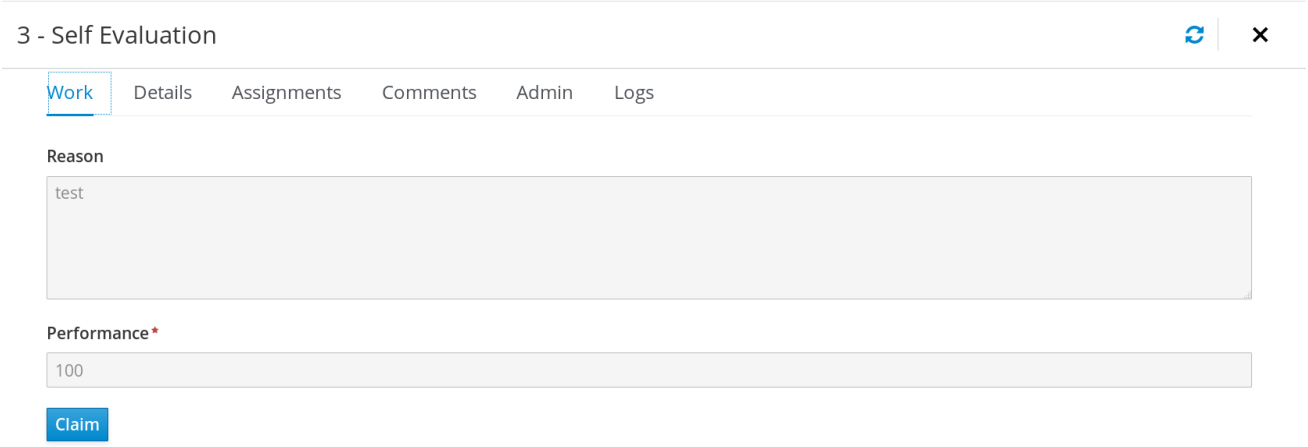
ビジネス管理者は、Business Central の **Tasks** ページから、すべてのユーザータスクを表示および管理できます。そのページは、**Menu → Manage → Tasks** で表示できます。**admin** ロールまたは **process-admin** ロールが割り当てられているユーザーは **Tasks** ページにアクセスできますが、デフォルトではタスクを表示して管理するアクセス権限がありません。

全タスクを管理するには、以下の条件を定義してプロセス管理者としてユーザーを指定する必要があります。

- ユーザーを **task admin user** として指定する。デフォルトの値は **Administrator** です。
- ユーザーをタスク管理者グループに所属させる。デフォルト値は **Administrators** です。

ユーザーとユーザーグループの割当は、**org.jbpm.ht.admin.user** と **org.jbpm.ht.admin.group** のシステムプロパティーで設定できます。

一覧にあるタスクをクリックしてビューを開き、期日、優先順位、タスクの説明など、タスクの詳細を修正できます。タスクページでは以下のタブが利用できます。



- **Work**: タスクおよびタスクの所有者に関する基本的な詳細を表示します。**Claim** ボタンをクリックして、タスクを要求できます。要求プロセスを取り消すには、**Release** ボタンをクリックします。
- **Details**: タスクの説明、ステータス、期日などの情報を表示します。
- **Assignments**: タスクの現在の所有者を表示し、別のユーザーまたはグループにタスクを委譲できます。
- **Comments**: タスクユーザーが追加したコメントを表示します。既存のコメントを削除して、新しいコメントを追加できます。
- **Admin**: タスクの所有者候補が表示されるため、タスクを別のユーザーやグループに転送できます。また、タスクの実際の所有者も表示し、タスクの実際の所有者にリマインダーを送信できます。
- **Logs**: タスクのライフサイクルイベント (タスクの開始、要求、終了など) を含むタスクログと、タスクフィールド (タスクの期日および優先順位など) に行った更新を表示します。

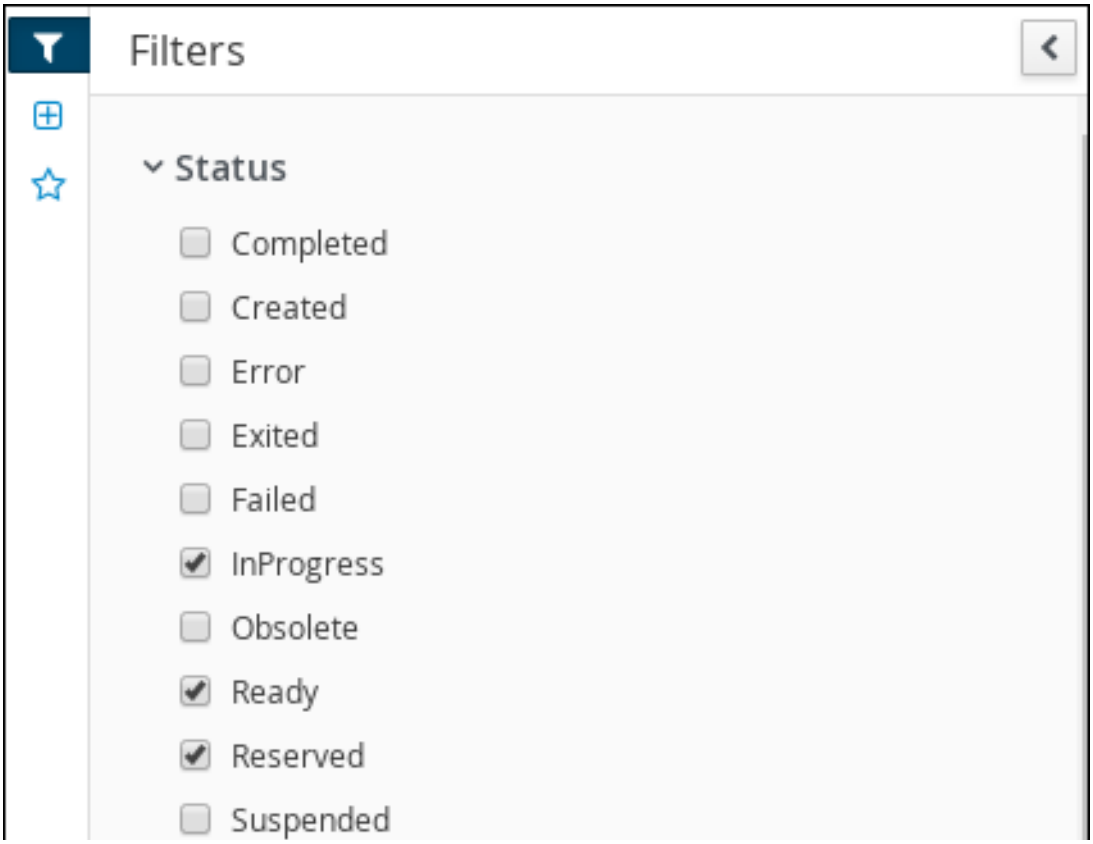
ページの左側にある **フィルター** アイコンをクリックして、利用可能なフィルターパラメーターをもとに、タスクをフィルターリングできます。フィルターリングの詳細は、「タスクのフィルターリング」を参照してください。

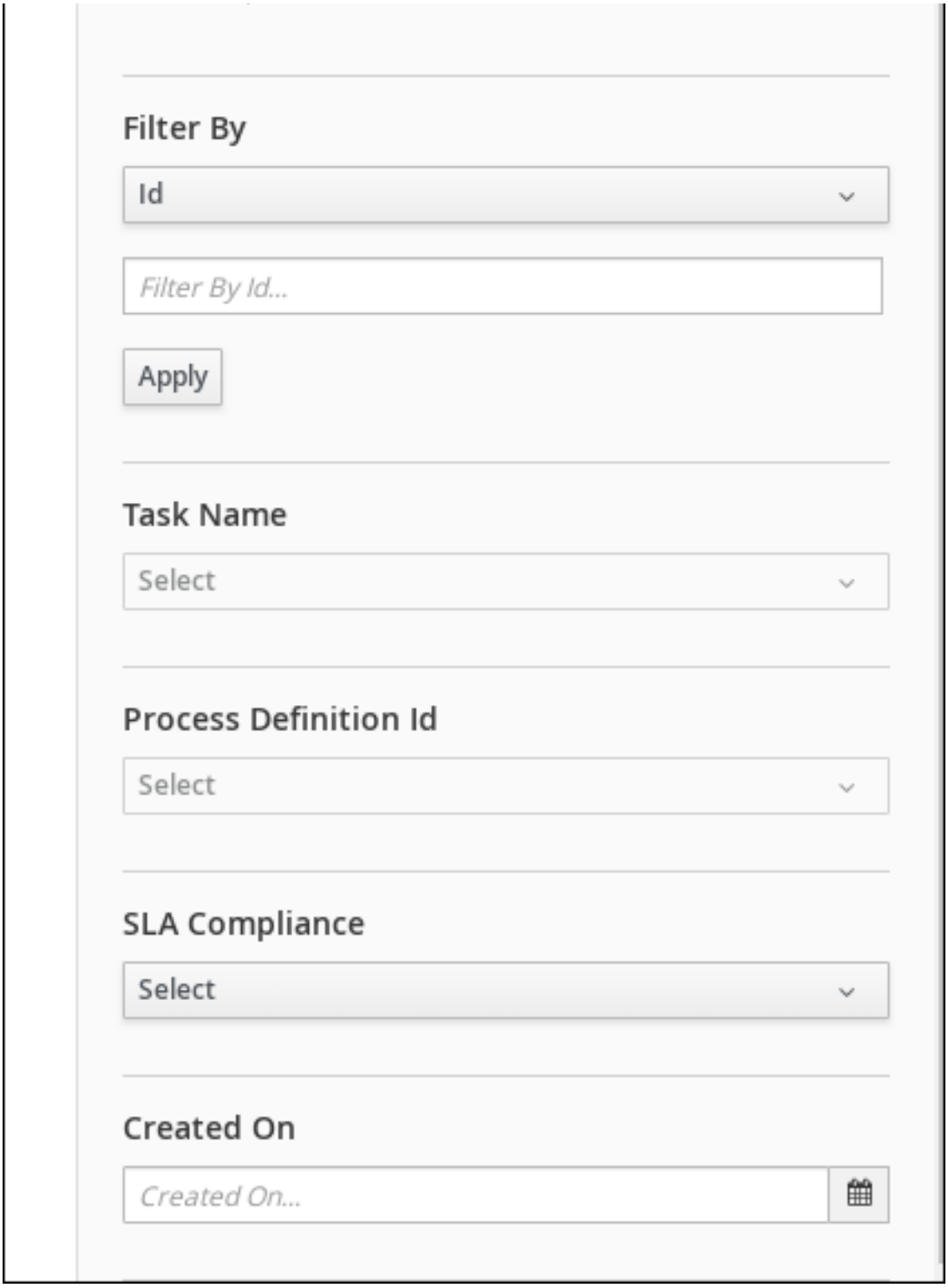
その他にも、定義するクエリーパラメーターに基づいてタスクをフィルターリングできるカスタムのフィルターを作成できます。カスタムタスクフィルターの詳細は、「カスタムタスクフィルターの作成」 を参照してください。

29.1. タスクのフィルターリング

Menu → Manage → Tasks および **Menu → Track → Task Inbox** のタスクについては、必要に応じて **Filters** と **Advanced Filters** パネルを使用してタスクを分類できます。

図29.1 タスクのフィルターリング: デフォルト表示





Manage Tasks ページは、管理者、およびプロセスの管理者だけが利用できます。

Filters パネルの以下の属性別に、タスクをフィルターリングできます。

ステータス

タスクステータスでフィルターリングします。ステータスを 1 つ以上選択して、選択したステータスのすべてを満たす結果を表示します。ステータスフィルターを削除すると、ステータスにかかわらずすべてのプロセスが表示されます。
以下のフィルターステータスを利用できます。

- 完了
- Created

- エラー
- 終了
- Failed
- 実行中
- 廃止
- 使用可能
- 予約済み
- 一時停止

ID

プロセスインスタンス ID でフィルターリングします。
入力: **Numeric**

タスク

タスク名でフィルターリングします。
入力: **String**

相関キー

Correlation key でフィルターリングします。
入力: **String**

実際の所有者

タスクの所有者でフィルターリングします。
実際の所有者は、タスクを実行するユーザーを指します。検索はユーザー ID に基づいていますが、これは一意の値で、ID 管理システムによって異なります。

入力: **String**

Process Instance Description

プロセスインスタンスの説明でフィルターリングします。
入力: **String**

タスク名

タスク名でフィルターリングします。

プロセス定義 ID

プロセス定義 ID でフィルターリングします。

SLA コンプライアンス

SLA コンプライアンス状態でフィルターリングします。
以下のフィルターステータスを利用できます。

- 強制終了
- 適合

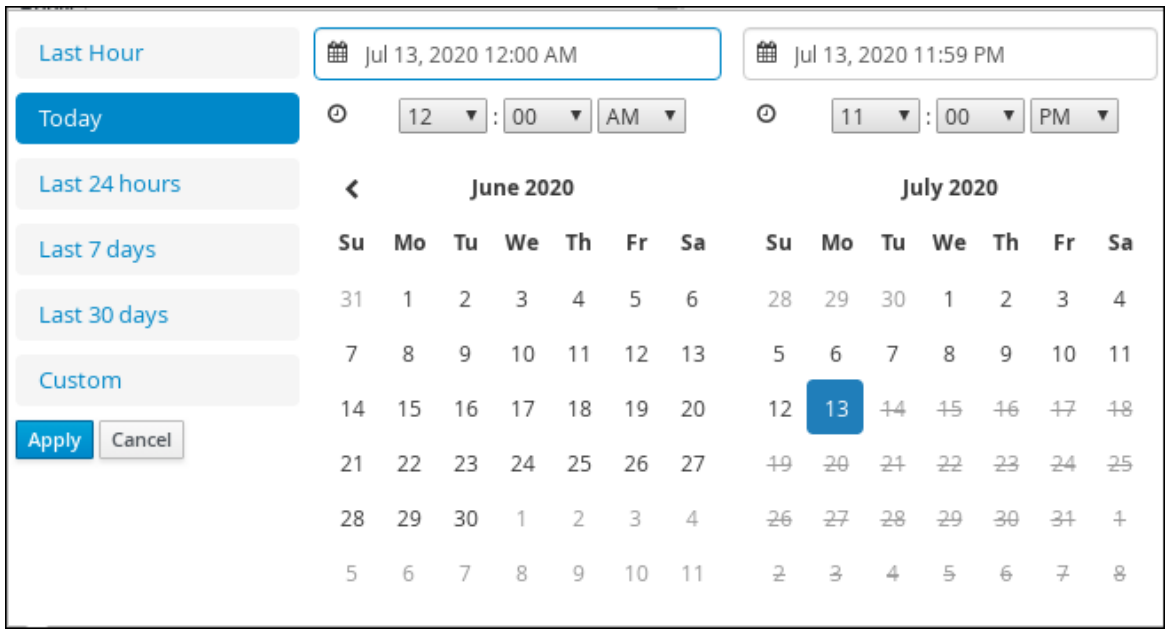
- 該当なし
- 保留中
- 違反

作成日

日付または時間でフィルターリングします。
このフィルターには、以下のような、フィルターのクイックオプションがあります。

- 過去 1 時間
- 本日
- 過去 24 時間
- 過去 7 日間
- 過去 30 日間
- カスタム
  Custom の日時フィルターリングを選択すると、カレンダーツールが開いて日時の範囲を選択できます。

図29.2 日付で検索



29.2. カスタムタスクフィルターの作成

Menu → Manage → Tasks または、**Menu → Track → Task Inbox** (現在のユーザーに割り当てられているタスク) で指定されたクエリーをもとに、カスタムのタスクフィルターを作成できます。

手順

1. Business Central で **Menu → Manage → Tasks** に移動します。

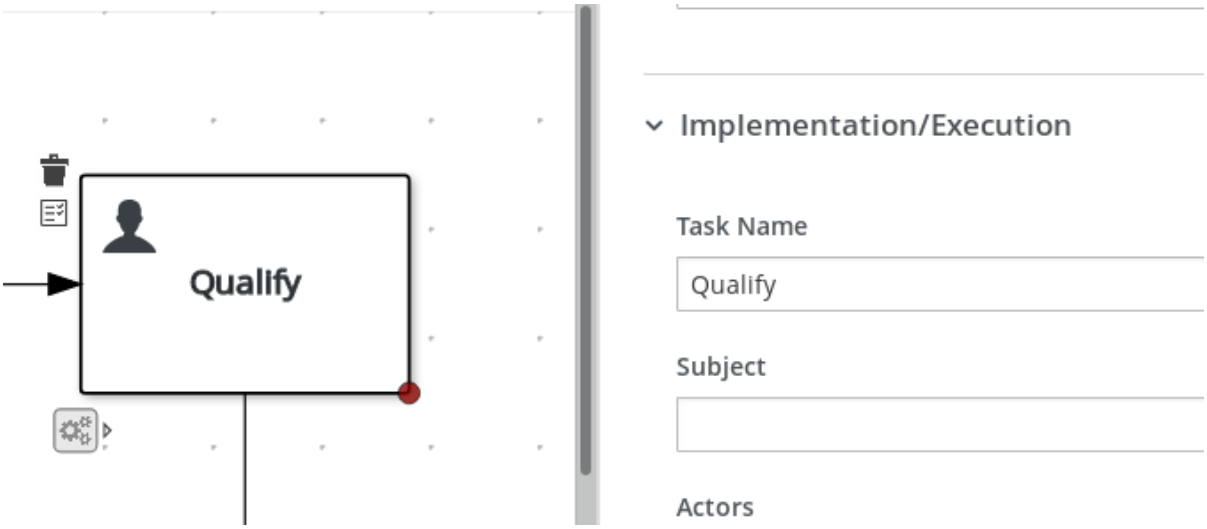
2. Manage Tasks ページで左側の詳細フィルターアイコンをクリックして、Advanced Filters オプションの一覧を開きます。

3. Advanced Filters パネルで、フィルターの名前と説明を入力して、Add New をクリックします。

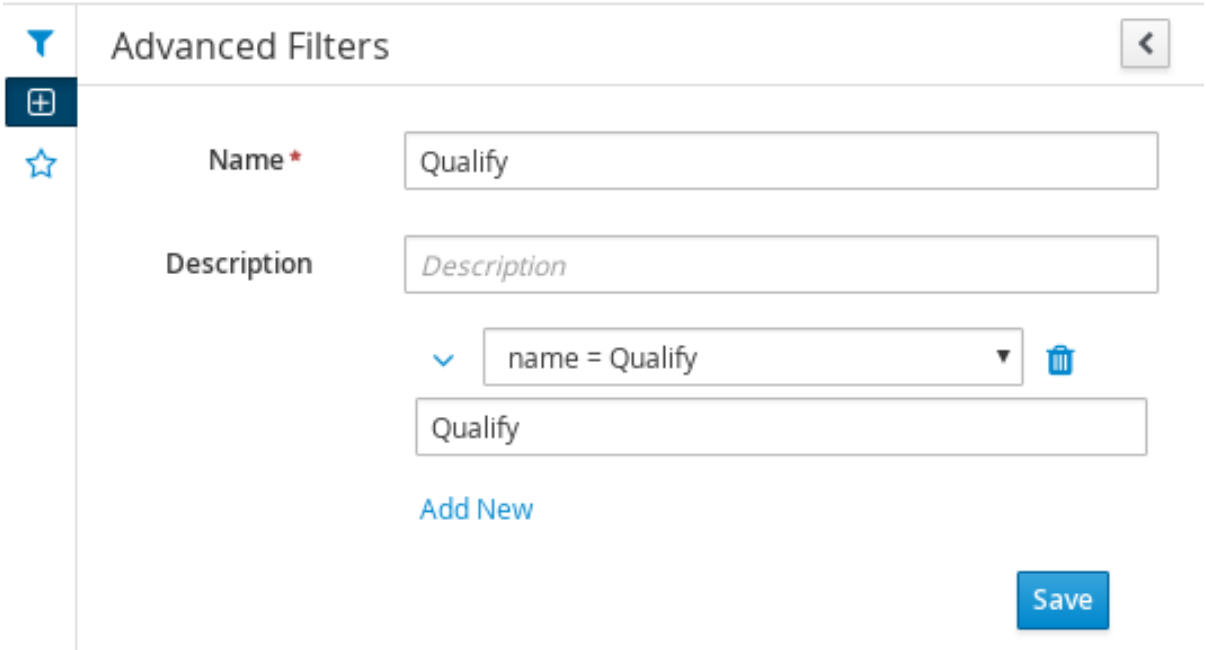
4. Select column ドロップダウンメニューで、name を選択します。
ドロップダウンメニューのコンテンツを **name != value1** に変更します。

5. ドロップダウンメニューを再度クリックして、equals to を選択します。

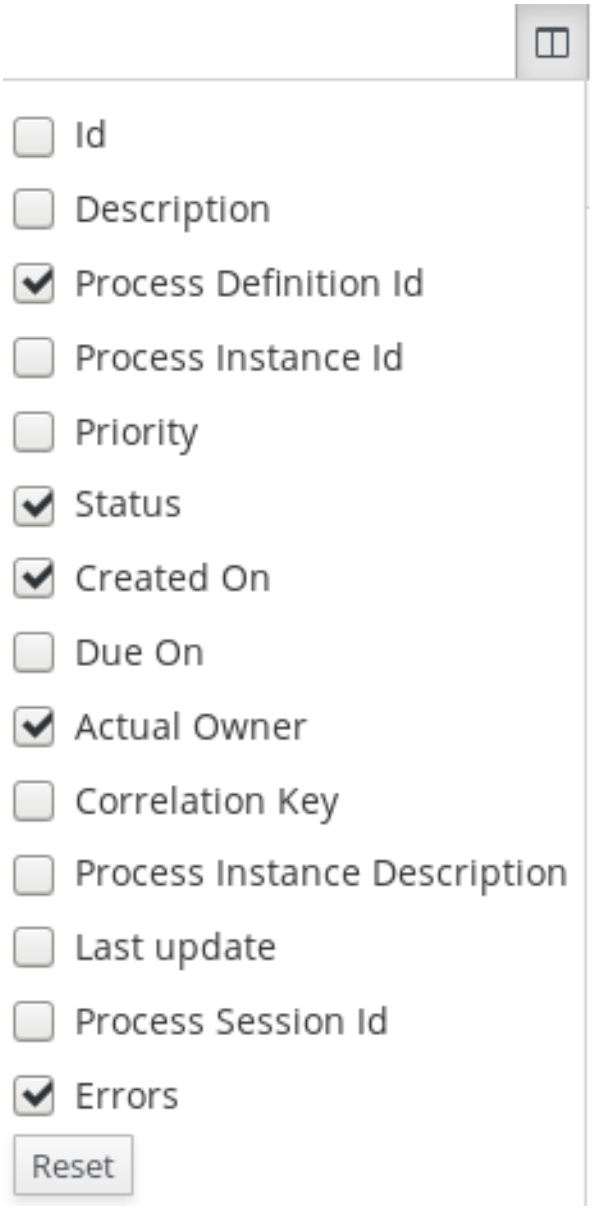
6. テキストフィールドの値を、フィルターリングするタスクの名前に書き替えます。この名前は、関連のビジネスプロセスで定義した値に一致させる必要があります。



7. OK をクリックして、カスタムタスクフィルターを保存します。



指定した制限が設定されたフィルターを適用すると、設定可能な列のセットは特定のカスタムタスクフィルターに基づいており、以下の列オプションが含まれます。



29.3. デフォルトのフィルターを使用したタスクの管理

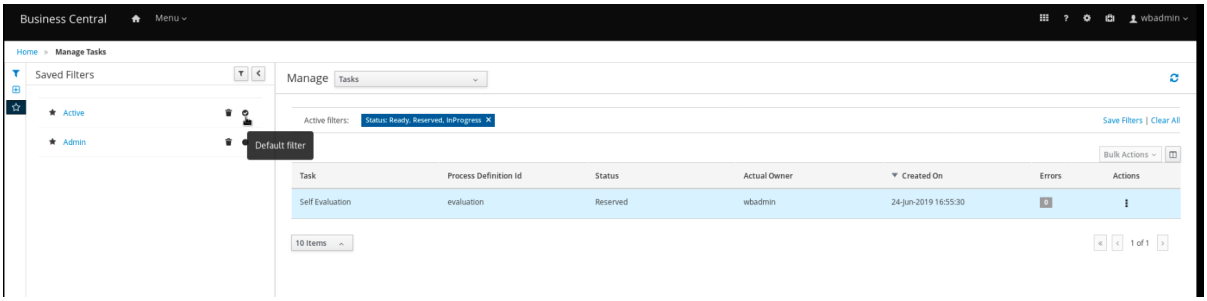
Business Central の **Saved Filter** オプションを使用して、タスクフィルターをデフォルトフィルターとして設定できます。ユーザーがページを開くたびにデフォルトのフィルターリングが実行されます。

手順

1. Business Central で **Menu → Track → Task Inbox** に移動するか、**Menu → Manage → Tasks** に移動します。

2. **Task Inbox** ページまたは、**Manage Tasks** ページの左側にある星アイコンをクリックして、**Saved Filters** パネルを展開します。
   Saved Filters パネルで、保存した詳細フィルターを表示できます。

   Tasks または Task Inbox のデフォルトフィルターのオプション



3. **Saved Filters** パネルで、保存したタスクフィルターを、デフォルトのフィルターとして設定します。

29.4. 基本的なフィルターを使用したタスク変数の表示

Business Central には基本フィルターがあり、**Manage Tasks** および **Task Inbox** のタスク変数を表示できます。タスクのタスク変数は、**Show/hide columns** を使用し列として表示できます。

手順

1. Business Central で **Menu → Manage → Tasks** に移動するか、**Menu → Track → Task Inbox** に移動します。
2. **Task Inbox** ページの左側にあるフィルターアイコンをクリックして、**Filters** パネルを展開します。
3. **Filters** パネルを選択し、**Task Name** をクリックします。
   フィルターは現在のタスクリストに適用されます。
4. タスクリストの右上にある **Show/hide columns** をクリックすると、指定のタスク ID のタスク変数が表示されます。
5. 星のアイコンをクリックして、**Saved Filters** パネルを開きます。
   Saved Filters パネルで、保存した詳細フィルターをすべて表示できます。

29.5. 詳細フィルターを使用したタスク変数の表示

Business Central の **Advanced Filters** オプションを使用して **Manage Tasks** と **Task Inbox** のタスク変数を表示できます。タスクを定義したフィルターを作成すると、**Show/hide columns** でタスクのタスク変数を列として表示できます。

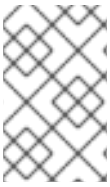
手順

1. Business Central で **Menu → Manage → Tasks** に移動するか、**Menu → Track → Task Inbox** に移動します。
2. **Manage Tasks** ページ、または **Task Inbox** ページの詳細フィルターアイコンをクリックして、**Advanced Filters** パネルを展開します。
3. **Advanced Filters** パネルで、フィルターの名前と説明を入力して、**Add New** をクリックします。
4. **Select column** リストから **name** 属性を選択します。この値は、**name != value1** に変わります。
5. **Select column** リストから論理クエリーに **equals to** を選択します。

6. テキストフィールドに、タスク名を入力します。
7. **Save** をクリックして、フィルターを現在のタスクリストに適用します。
8. タスクリストの右上にある **Show/hide columns** をクリックすると、指定のタスク ID のタスク変数が表示されます。
9. 星のアイコンをクリックして、**Saved Filters** パネルを開きます。
   Saved Filters パネルで、保存した詳細フィルターをすべて表示できます。

29.6. BUSINESS CENTRAL でのカスタムタスクの管理

カスタムタスク (作業アイテム) は、カスタムロジックを実行できるタスクです。複数のビジネスプロセスまたは Business Central の全プロジェクトでタスクをカスタマイズして再利用できます。また、名前、アイコン、サブカテゴリー、入出力パラメーター、ドキュメントなど、デザイナーパレットにカスタム要素を追加することもできます。Red Hat Process Automation Manager は、Business Central のカスタムタスクリポジトリー内でカスタムタスクセットを提供します。デフォルトのカスタムタスクを有効化または無効化して、カスタムのタスクを Business Central にアップロードし、適切なプロセスにこのタスクを実装できます。



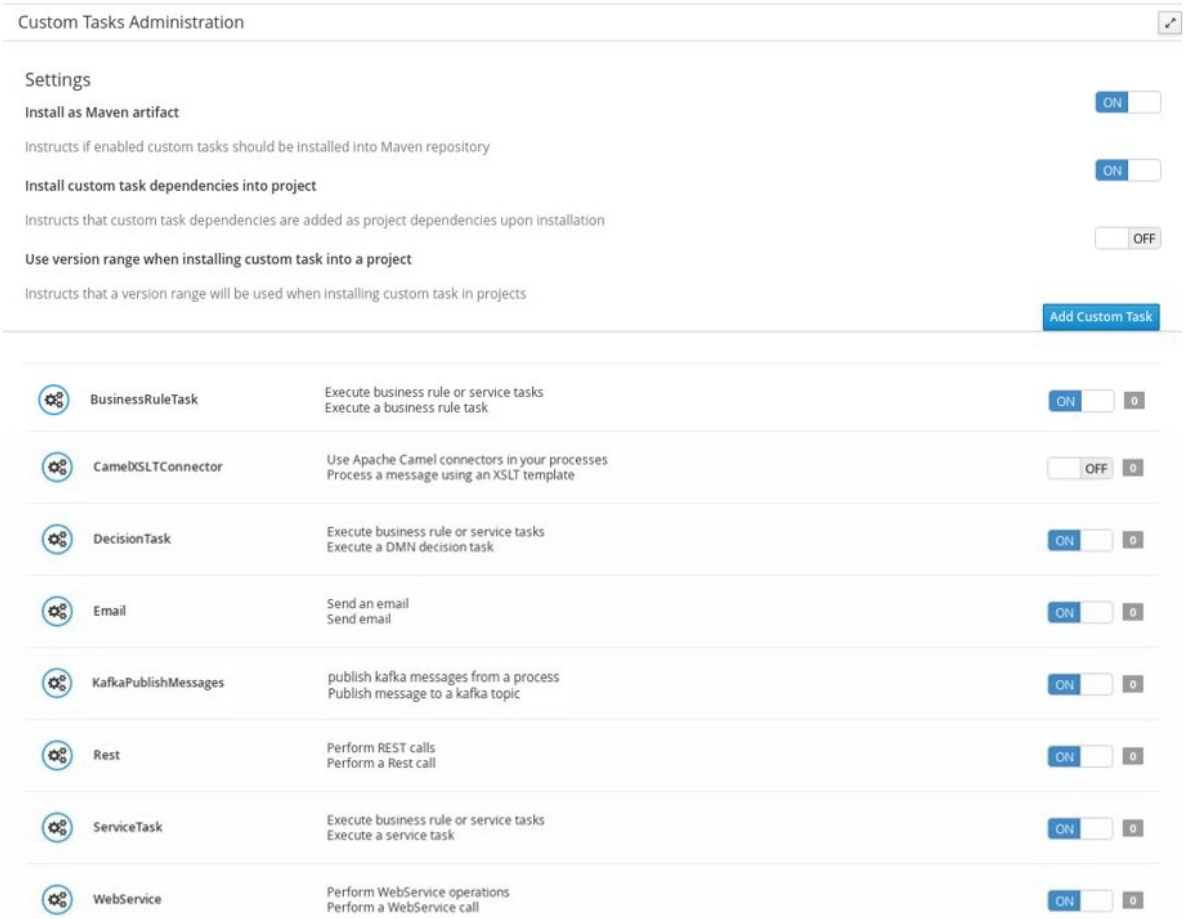
注記

Red Hat Process Automation Manager には、サポートされるカスタムタスクの限定セットが含まれています。Red Hat Process Automation Manager に含まれていないカスタムタスクはサポートされません。

手順

1. Business Central で、右上隅の をクリックし、**Custom Tasks Administration** を選択します。
   このページは、カスタムタスクのインストール設定や、Business Central 全体にあるプロジェクトのプロセスで利用可能なカスタムタスクを表示します。このページで有効にしたカスタムタスクは、プロジェクトレベルの設定で利用できます。プロジェクトレベルの設定で、プロセスで使用する各カスタムタスクをインストールできます。カスタムタスクをプロジェクトにインストールする方法は、**Custom Tasks Administration** ページの **Settings** で有効または無効にしたグローバル設定により決まります。
2. **Settings** で、各設定を有効または無効にして、ユーザーがプロジェクトレベルでインストールするときに、利用可能なカスタムタスクを実装する方法を決定します。
   以下のカスタムタスクの設定が利用できます。
   - **Install as Maven artifact** ファイルがない場合は、カスタムタスクの JAR ファイルを Maven リポジトリーにアップロードし、Business Central で設定します。
   - **Install custom task dependencies into project** カスタムタスクの依存関係をプロジェクトの **pom.xml** ファイルに追加します。このファイルでタスクがインストールされます。
   - **Use version range when installing custom task into project** プロジェクトの依存関係として追加されるカスタムタスクの固定バージョンではなく、バージョン範囲を使用します。たとえば、**7.16.0.Final** ではなく **[7.16,)** です。
3. 必要に応じて利用可能なカスタムタスクを有効または無効にします (**ON** または **OFF** に設定)。有効化したカスタムタスクは、Business Central の全プロジェクトのプロジェクトレベル設定に表示されます。

図29.3 カスタムタスクとカスタムタスク設定の有効化



4. カスタムタスクを追加するには、**Add Custom Task**をクリックし、関連する JAR ファイルを参照し、**Upload** アイコンをクリックします。クラスが **WorkItemHandler** を実装する場合は、ファイルを Business Central に個別に追加して、アノテーションを **.wid** ファイルに置き換えることができます。

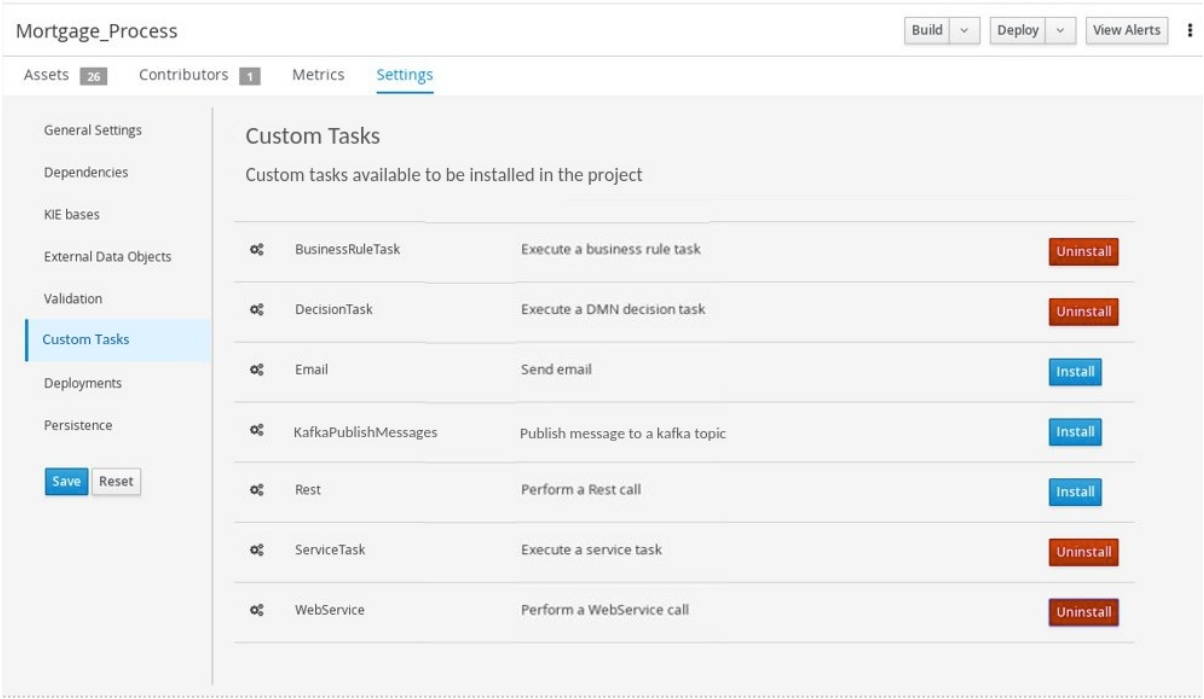
5. 必要に応じてカスタムタスクを削除するには、削除するカスタムタスクの行にある **remove** をクリックし、**OK** をクリックして削除を確定します。

6. Business Central ですべての必須カスタムタスクを設定した後に、プロジェクトの **Settings** → **Custom Tasks** ページに移動すると、有効化したカスタムタスクで利用可能なものが表示されます。

7. カスタムタスクごとに、**Install** をクリックして、対象のプロジェクトのプロセスでタスクを利用できるようにするか、**Uninstall** をクリックして、プロジェクトのプロセスからタスクを除外します。

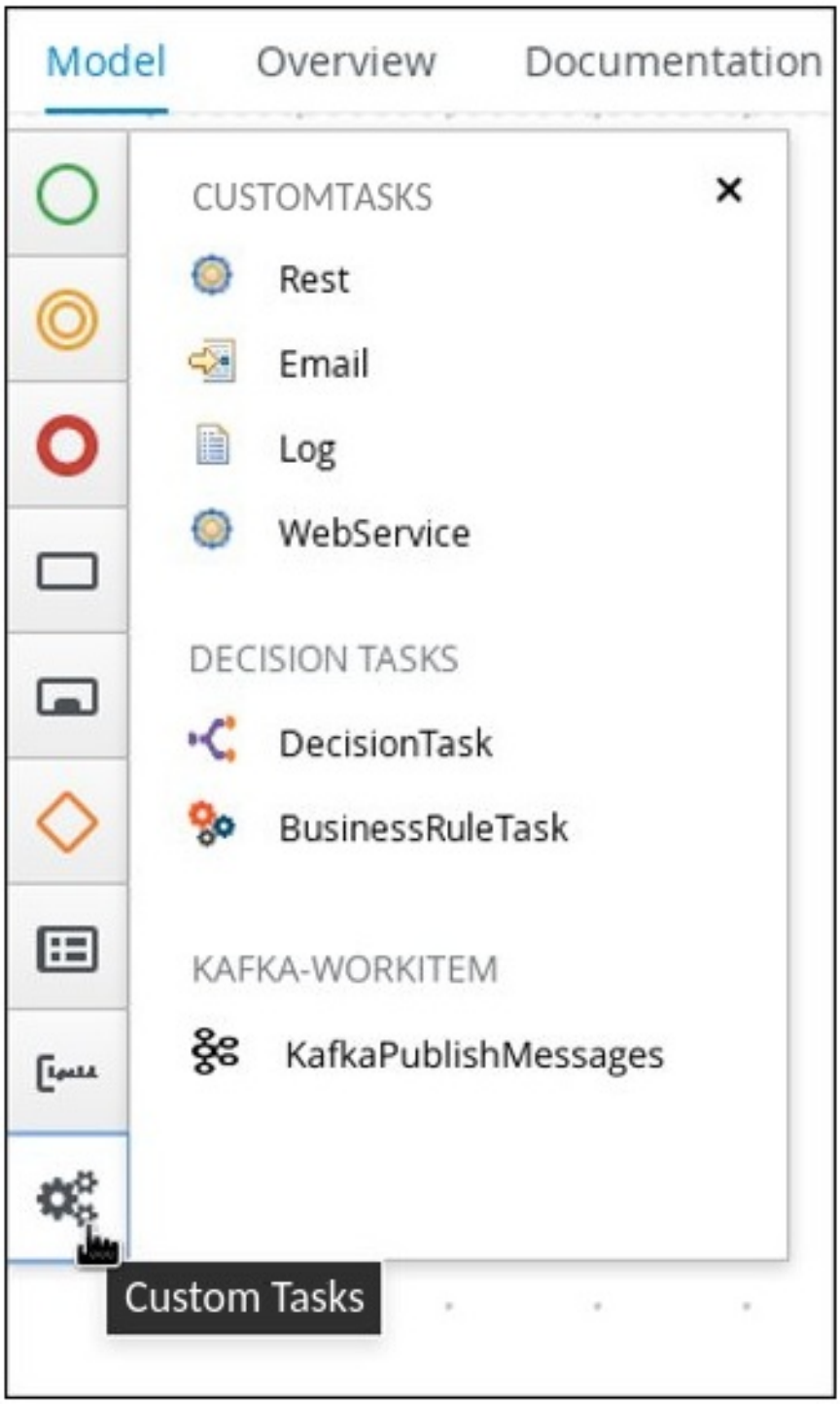
8. カスタムタスクのインストール時に追加情報を求められた場合は、必要な情報を入力して、もう一度 **Install** をクリックします。
カスタムタスクの必須パラメーターは、タスクのタイプにより異なります。たとえば、ルールとデシジョンタスクにはアーティファクトの GAV 情報 (グループ ID、アーティファクト ID、およびバージョン) が、メールタスクにはホストとポートアクセスの情報が、REST タスクには API の認証情報が必要です。他のカスタムタスクでは、追加のパラメーターが必要でない場合もあります。

図29.4 プロセスで使用するカスタムタスクのインストール



9. **Save** をクリックします。
10. プロジェクトページに戻り、プロジェクトのビジネスプロセスを選択または追加します。プロセスデザイナーパレットで **Custom Tasks** オプションを選択すると、有効にしてインストールした、利用可能なカスタムタスクが表示されます。

図29.5 プロセスデザイナーでのインストール済みカスタムタスクへのアクセス



29.7. ユーザータスク管理

ユーザータスクを使用すると、作成したビジネスプロセスに対する入力として、人間のアクションを追加できます。ユーザータスク管理では、ユーザータスクやグループタスクの割り当て、データ処理、時間ベースの自動通知、再割り当てを操作する手法が提供されます。

以下のユーザータスク操作は、Business Central で利用できます。

- **add/remove potential owners - by task id** タスク ID を使用してユーザーとグループを追加または削除します。
- **add/remove excluded owners - by task id** タスク ID を使用して除外する所有者を追加または削除します。

- **add/remove business administrators - by task id** タスク ID を使用してビジネス管理者を追加または削除します。
- **add task inputs - by task id** タスク ID を使用してタスクの作成後に、タスクの入力コンテンツを変更する手段を提供します。
- **remove task inputs - by task id** タスク ID を使用してタスクの入力変数を削除します。
- **remove task output - by task id** タスク ID を使用してタスクの出力変数を削除します。
- **schedules new reassignment to given users/groups after given time elapses - by task id**時間の式およびタスクの状態をもとに、自動的な再割当てをスケジュールします。
  - **reassign if not started**: タスクが **InProgress** の状態に移動されなかった場合に使用します。
  - **reassign if not completed**: タスクが **Completed** の状態に移動されなかった場合に使用します。
- **schedules new email notification to given users/groups after given time elapses - by task id** 時間の式およびタスクの状態をもとにメールの自動通知をスケジュールします。
  - **notify if not started**: タスクが **InProgress** の状態に移動されなかった場合に使用します。
  - **notify if not completed**: タスクが **Completed** の状態に移動されなかった場合に使用します。
- **list scheduled task notifications - by task id** タスク ID を使用してアクテイブなタスク通知をすべて返します。
- **list scheduled task reassignments - by task id** タスク ID を使用してアクテイブなタスクの再割り当てすべてを返します。
- **cancel task notification - by task id and notification id** タスク ID を使用してタスクの通知を取り消して、スケジュールをキャンセルします。
- **cancel task reassignment - by task id and reassignment id**タスク ID を使用してタスクの再割り当てを取り消して、スケジュールをキャンセルします。

29.8. タスクの一括アクション

Business Central の **Tasks** および **Task Inbox** ページでは、1 回の操作で複数のタスクに対して一括アクションを実行できます。

注記

指定された一括アクションがタスクステータスに基づいて許可されていない場合は、通知が表示され、そのタスクでは操作は実行されません。

29.8.1. タスクを一括で要求する

Business Central でタスクを作成した後、使用可能なタスクを一括して要求できます。

手順

1. Business Central で、以下のいずれかの手順を実行します。

- **Menu** → **Track** → **Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
- **Menu** → **Manage** → **Tasks** の順に選択すると、**Tasks** ページが表示されます。

2. タスクを一括して請求するには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** ドロップダウンリストから、**Bulk Claim** を選択します。
4. 確認するには、**Claim selected tasks** ウィンドウで、**Claim** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

29.8.2. タスクを一括でリリースする

所有しているタスクを一括してリリースし、他のユーザーが要求できるようにすることができます。

手順

1. Business Central で、以下のいずれかの手順を実行します。
   - **Menu** → **Track** → **Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
   - **Menu** → **Manage** → **Tasks** の順に選択すると、**Tasks** ページが表示されます。
2. タスクを一括してリリースするには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** のドロップダウンリストから、**Bulk Release**を選択します。
4. 確認するには、**Release selected tasks** ウィンドウで、**Release** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

29.8.3. タスクを一括で再開する

Business Central に一時停止されたタスクがある場合は、それらを一括して再開できます。

手順

1. Business Central で、以下のいずれかの手順を実行します。
   - **Menu** → **Track** → **Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
   - **Menu** → **Manage** → **Tasks** の順に選択すると、**Tasks** ページが表示されます。
2. タスクを一括して再開するには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** のドロップダウンリストから、**Bulk Resume** を選択します。
4. 確認するには、**Resume selected tasks** ウィンドウで **Resume** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

29.8.4. タスクを一括で一時停止する

Business Central でタスクを作成した後、タスクを一括して一時停止できます。

手順

1. Business Central で、以下のいずれかの手順を実行します。
   - **Menu → Track → Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
   - **Menu → Manage → Tasks** の順に選択すると、**Tasks** ページが表示されます。
2. タスクを一括して一時停止するには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** のドロップダウンリストから、**Bulk Suspend** を選択します。
4. 確認するには、**Suspend selected tasks** ウィンドウで **Suspend** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

29.8.5. タスクを一括で再割り当てする

Business Central でタスクを作成した後、タスクを一括して再割り当てし、他のユーザーに委任できます。

手順

1. Business Central で、以下のいずれかの手順を実行します。
   - **Menu → Track → Task Inbox** の順に選択すると、**Task Inbox** ページが表示されます。
   - **Menu → Manage → Tasks** の順に選択すると、**Tasks** ページが表示されます。
2. タスクを一括して再割り当てするには、**Task Inbox** ページまたは **Manage Tasks** ページで、**Task** テーブルから 2 つ以上のタスクを選択します。
3. **Bulk Actions** のドロップダウンリストから、**Bulk Reassign** を選択します。
4. **Tasks reassignment** ウィンドウで、タスクを再割り当てするユーザーのユーザー ID を入力します。
5. **Delegate** をクリックします。

選択したタスクごとに、結果を示す通知が表示されます。

第30章 ログデータの管理

Red Hat Process Automation Manager は、必要なメンテナンスランタイムデータを管理します。以下のデータタイプを含む、一部のデータが自動的に削除されます。

- **プロセスインタンスデータ**。プロセスインスタンスの完了時に削除されます。
- **ワークアイテムデータ**。ワークアイテムの完了時に削除されます。
- **タスクインスタンスデータ**。指定のタスクが所属するプロセスの完了時に削除されます。

自動的に消去されないランタイムデータには、選択したランタイムストラテジーを基にしたセッション情報データが含まれます。

- **シングルトンストラテジー** を使用して、セッション情報のランタイムデータが自動的に削除されないようにします。
- **リクエスト別のストラテジー** を使用することで、要求が終了した時点で自動的に削除できます。
- **プロセス別のインスタンス** は、プロセスインスタンスが完了または中断されたセッションにマッピングされると、プロセスインスタンスデータを自動的に削除します。

Red Hat Process Automation Manager は、監査ログデータテーブルも提供します。これらのテーブルは、現在および過去のプロセスインスタンスを追跡するために使用することができます。デフォルトでは、Red Hat Process Automation Manager は監査ログテーブルからデータを削除しません。

監査データテーブルを管理維持する方法は 3 つあります。

- 「自動クリーンアップジョブの設定」で説明されているように、Business Central を使用してこれらのテーブルの自動クリーンアップを設定することができます。
- 「手動クリーンアップ」で説明されているように、Java API を使用してテーブルから情報を手動で削除することができます。
- 「Red Hat Process Automation Manager データベースでのカスタムクエリーの実行」で説明するように、監査ログテーブルを含む Red Hat Process Automation Manager データベースに対してカスタムクエリーを実行することができます。

30.1. 自動クリーンアップジョブの設定

Business Central では自動クリーンアップジョブを設定できます。

手順

1. Business Central で、**Manage** > **Jobs** に移動します。
2. **New Job** をクリックします。
3. **Business Key** フィールド、**Due On** フィールド、および **Retries** フィールドに値を入力します。
4. **Type** フィールドに以下のコマンドを入力します。

```
org.jbpm.executor.commands.LogCleanupCommand
```

5. パラメーターを設定するには、次の手順を実行します。
   a. **Advanced** タブをクリックします。
   b. **パラメーターの追加** をクリックします。
   c. **Key** 欄にパラメーターを入力します。
   d. **Value** 欄にパラメーターを入力します。

   コマンドのパラメーター一覧は、「データベースからのログの削除」を参照してください。
6. **Create** をクリックします。

Business Central は自動クリーンアップジョブを作成します。

30.2. 手動クリーンアップ

手動クリーンアップを実行するには、監査 Java API を使用できます。監査 API は、以下のエリアで設定されます。

表30.1 監査 API エリア

名前	説明
プロセス監査	これは、**jbpm-audit** モジュールでアクセス可能なプロセス、ノード、変数ログをクリーンアップするのに使用します。 たとえば、**org.jbpm.process.audit.JPAAuditLogService** のようにモジュールにアクセスできます。
タスク監査	これは、**jbpm-human-task-audit** モジュールでアクセス可能なタスクやイベントをクリーンアップするのに使用します。 たとえば、**org.jbpm.services.task.audit.service.TaskJPAAuditService** のように、モジュールにアクセスできます。
エグゼキュータージョブ	これは、**jbpm-executor** モジュールでアクセス可能なエグゼキュータージョブおよびエラーのクリーンアップに使用します。 たとえば、**org.jbpm.executor.impl.jpa.ExecutorJPAAuditService** のようにモジュールにアクセスできます。

30.3. データベースからのログの削除

LogCleanupCommand エグゼキューターコマンドを使用して、データベースの領域を使用するデータをクリーンアップします。**LogCleanupCommand** は、自動ですべてのデータをクリーンアップするロジックと、選択したデータをクリーンアップするロジックで設定されます。

LogCleanupCommand と合わせて使用可能な設定オプションが複数あります。

表30.2 LogCleanupCommand パラメーターテーブル

名前	説明	排他的
SkipProcessLog	コマンドの実行時に、プロセスおよびノードインスタンス、プロセス変数ログのクリーンアップをスキップするかどうかを指定します。デフォルト値は **false** です。	いいえ、他のパラメーターと併用します。
SkipTaskLog	タスク監査およびイベントログのクリーンアップをスキップするかどうかを指定します。デフォルト値は **false** です。	いいえ、他のパラメーターと併用します。
SkipExecutorLog	Red Hat Process Automation Manager エグゼキューターのエントリー消去をスキップするかどうかを指定します。デフォルト値は **false** です。	いいえ、他のパラメーターと併用します。
SingleRun	ジョブルーチンを 1 回だけ実行するかどうかを指定します。デフォルト値は **false** です。	いいえ、他のパラメーターと併用します。
NextRun	次のジョブ実行をスケジュールします。デフォルト値は、**24h** です。 たとえば、12 時間ごとにジョブを実行するには、**12h** と指定します。**SingleRun** が **true** にされており、**SingleRun** と **NextRun** の両方が設定されていないと、このスケジュールは無視されます。両方が設定されている場合には、**NextRun** のスケジュールが優先されます。正確な日付を設定するには、ISO 形式を使用できます。	いいえ、他のパラメーターと併用します。
OlderThan	指定の日付より古いログを削除します。日付の形式は、**YYYY-MM-DD** です。通常、このパラメーターは単一のジョブ実行に使用します。	はい。**OlderThanPeriod** パラメーターと併用しません。
OlderThanPeriod	指定のタイマー式より古いログを削除します。たとえば、30 日が経過したログを削除するには、**30d** を設定します。	はい。**OlderThan** パラメーターと併用しません。
ForProcess	削除するログのプロセス定義 ID を指定します。	いいえ、他のパラメーターと併用します。
RecordsPerTransaction	削除するトランザクションのレコード数を指定します。デフォルト値は **0** で、すべてのレコードを示します。	いいえ、他のパラメーターと併用します。
ForDeployment	削除するログのデプロイメント ID を指定します。	いいえ、他のパラメーターと併用します。
EmfName	削除操作の実行に使用する永続性ユニット名。	該当なし

注記

LogCleanupCommand では、プロセスインスタンス、タスクインスタンス、またはエグゼキュータジョブの実行など、アクティブなインスタンスは削除されません。

30.4. RED HAT PROCESS AUTOMATION MANAGER データベースでのカスタムクエリーの実行

ExecuteSQLQueryCommand エグゼキューターコマンドを使用して、監査ログデータテーブルを含む Red Hat Process Automation Manager データベースに対してカスタムクエリーを実行することができます。このコマンドを実行するジョブを Business Central で設定することができます。

手順

1. Business Central で、**Manage** > **Jobs** を選択します。
2. **New Job** をクリックします。
3. **Business Key** フィールド、**Due On** フィールド、および **Retries** フィールドに値を入力します。
4. **Type** フィールドに以下のコマンドを入力します。

```
org.jbpm.executor.commands.ExecuteSQLQueryCommand
```

5. パラメーターを設定するには、次の手順を実行します。
   a. **Advanced** タブをクリックします。
   b. **パラメーターの追加** をクリックします。
   c. **Key** 欄にパラメーター値を入力します。
   d. **Value** 欄にパラメーター値を入力します。

   コマンドのパラメーター一覧は、「**ExecuteSQLQueryCommand** コマンドのパラメーター」を参照してください。
6. **Create** をクリックします。Business Central はカスタムクエリージョブを作成します。
7. オプション: クエリーの結果を取得する場合は、次の手順を実行します。
   a. Business Central が表示するジョブの一覧から、開始したジョブを探します。ジョブがリストに存在しない場合は、**Active filters** リストからいずれかのフィルターを削除します。
   b. ジョブの **id** 値を記録します。
   c. Web ブラウザーを使用して、**<kie_server_address>/docs** から KIE Server の Swagger ドキュメントにアクセスします (例: **http://localhost:8080/kie-server/docs/**)。
   d. **GET /server/jobs/{jobId}** リクエストをクリックします。
   e. **jobId** フィールドに、記録した **id** 値を入力します。
   f. **withErrors** リストから、**true** を選択します。

g. **withData** リストから、**true** を選択します。

h. **Execute** をクリックします。

i. **Server response** フィールドを確認します。SQL クエリーが成功した場合、**"response-data"** キーの下に結果が格納されます。

30.4.1. ExecuteSQLQueryCommand コマンドのパラメーター

ExecuteSQLQueryCommand エグゼキューターコマンドを使用して、監査ログテーブルを含む Red Hat Process Automation Manager データベースに対してカスタムクエリーを実行します。監査ログテーブルのスキーマについては、Red Hat Process Automation Manager のプロセスエンジンを参照してください。

ExecuteSQLQueryCommand コマンドには、以下のパラメーターを設定することができます。

表30.3 ExecuteSQLQueryCommand パラメーターテーブル

名前	説明
SingleRun	クエリーを一度だけ起動できる場合は **true**。クエリーを複数回起動できる場合は **false**。
EmfName	クエリーを実行するために使用する永続ユニットの名前
businessKey	クエリーで使用するビジネスキー。Business Central でコマンドを設定する場合、ジョブに設定したビジネスキーを使用します
SQL	実行するネイティブ SQL クエリー。パラメーターの前に **:** 記号を付けます
parametersList	SQL クエリーの全パラメーターのリスト。パラメーターは **,** 記号で区切ってください
SQL パラメーター名	SQL パラメーターの値。SQL パラメーターごとに個別のコマンドパラメーターを作成します

例えば、2 つのパラメーターを持つクエリーを使用することができます。

```
SELECT * FROM RequestInfo WHERE id = :paramId AND businessKey = :paramKey
```

ExecuteSQLQueryCommand コマンドには、以下のパラメーターを設定します。

- **SQL**: **SELECT * FROM RequestInfo WHERE id = :paramId AND businessKey = :paramKey** ;
- **parametersList**: **paramId,paramKey**
- **paramId**: **id** の値
- **paramKey**: **businessKey** の値

第31章 実行エラー管理

ビジネスプロセスの実行エラーが発生すると、プロセスが停止し、直近の安定した状態 (直近の安全なポイント) に戻り、実行を継続します。プロセスがエラーを処理していない場合は、トランザクション全体がロールバックされ、プロセスインスタンスを 1 つ前の待ち状態のままにします。実行エラーは、プロセスエンジンにリクエストを送った呼び出し元が見ることができます。

プロセスの管理者 (**process-admin**)、または管理者 (**admin**) のロールが割り当てられているユーザーは、Business Central の実行エラーメッセージにアクセスできます。実行エラーメッセージ機能では、主に以下の利点があります。

- より優れたトレーサビリティー
- 重大なプロセスの表示
- エラー状態に基づいたレポートおよび解析
- 外部システムエラー処理および補正

31.1. BUSINESS CENTRAL でのプロセスの実行エラー表示

Business Central では、2 つの場所でプロセスエラーを表示できます。

- **Menu → Manage → Process Instances**
- **Menu → Manage → Execution Errors**

現在のプロセスインスタンスにエラーが存在する場合は、**Manage Process Instances** ページの **Errors** 列に、エラーの数が表示されます。

前提条件

- Business Central でのプロセスの実行時にエラーが発生している。

手順

1. Business Cnetral で **Menu → Manage → Process Instances** に移動し、**Errors** で表示されている数の上にマウスをかざします。
2. **Errors** 列に表示されるエラーの数をクリックして、**Manage Execution Errors** ページに移動します。
   Manage Execution Errors ページには、すべてのプロセスインスタンスのエラー一覧が表示されます。

31.2. 実行エラーの管理

定義上、検出および保存されたプロセスエラーは何も確認されておらず、何らかの形 (エラーからの自動回復の場合) で処理する必要があります。承認された、または承認されなかったエラーのフィルターリングされたリストを表示することができます。エラーを承認すると、追跡のために、ユーザー情報およびタイムスタンプが保存されます。

手順

1. Business Central で、**Menu → Manage → Execution Errors** の順に選択します。

2. 一覧からエラーを選択し、**Details** タブを開きます。**Details** タブに、エラーに関する情報が表示されます。

3. **Acknowledge** ボタンをクリックして、エラーを承認します。**Manage Execution Errors** ページの **Acknowledged** フィルターで **Yes** を選択すれば、後から承認されたエラーを表示できます。
エラーがタスクに関連する場合は、**Go to Task**ボタンが表示されます。

4. オプション: 該当する場合は、**Go to Task**ボタンをクリックして、**Manage Tasks** ページに、関連するジョブ情報を表示します。
Manage Tasks ページでは、対応するタスクの再起動、再スケジュール、または再試行を行うことができます。

31.3. エラーのフィルターリング

Manage Execution Errors 画面の実行エラーについては、**Filters** パネルを使用して、選択した条件に合致するエラーのみを表示することができます。

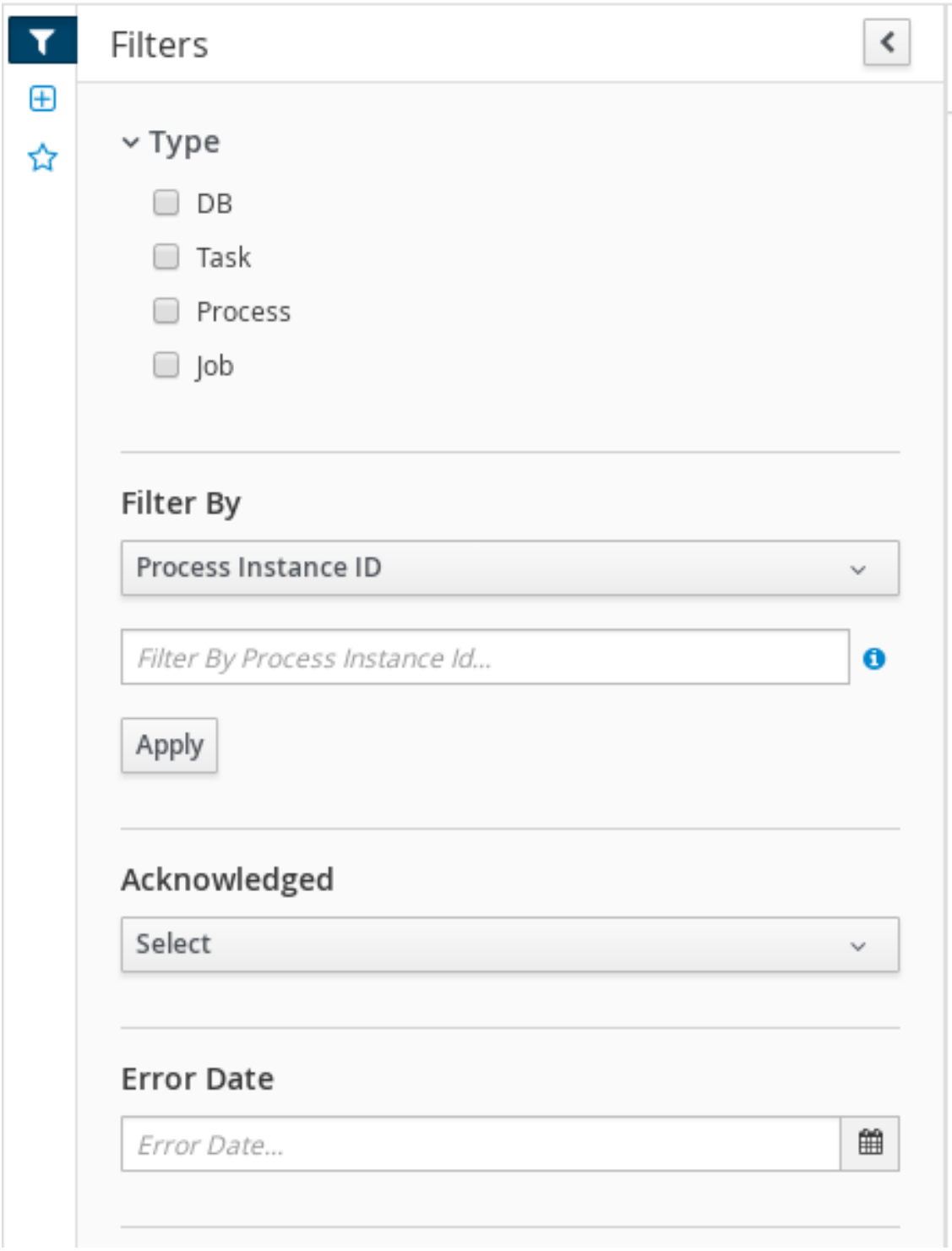
前提条件

- **Manage Execution Errors** 画面が表示されている。

手順

画面左側の **Filters** パネルで、必要に応じて変更を加えます。

図31.1 エラーのフィルターリング: デフォルト表示



タイプ

実行エラーを種類でフィルターリングします。フィルターリングの種類は複数選択できます。すべての種類の選択を解除すると、種類に関係なく、すべてのエラーが表示されます。
以下の実行エラーのタイプを使用できます。

- DB
- タスク
- Process
- Job

プロセスインスタンス ID

プロセスインスタンス ID でフィルターリングします。
入力: **Numeric**

ジョブ ID

ジョブ ID でフィルターリングします。ジョブ ID は、ジョブの作成時に自動的に作成されます。
入力: **Numeric**

ID

プロセスインスタンス ID でフィルターリングします。
入力: **Numeric**

確認済み

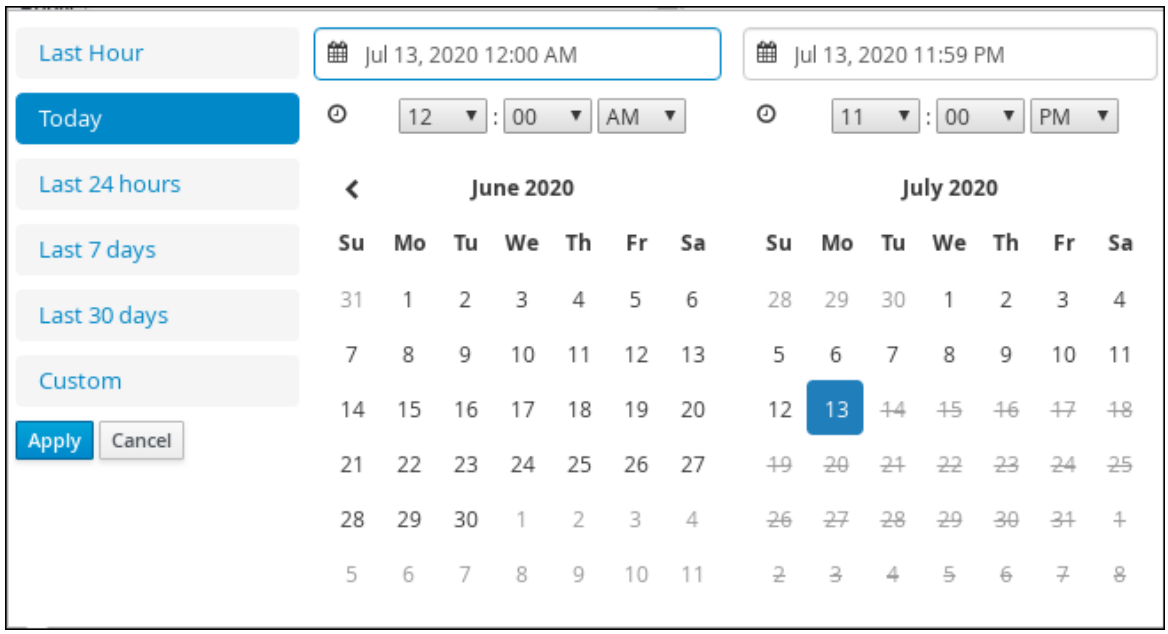
エラーを承認したかどうかでフィルターリングします。

エラー日付

イベントが発生した日付または時間でフィルターリングします。
このフィルターには、以下のような、フィルターのクイックオプションがあります。

- 過去 1 時間
- 本日
- 過去 24 時間
- 過去 7 日間
- 過去 30 日間
- カスタム
  カスタム のオプションを選択すると、日付と時間の範囲を選択するためのカレンダーツールが表示されます。

図31.2 日付で検索



31.4. 実行エラーの自動承認

デフォルトでは、実行エラーは発生しても承認されません。すべての実行エラーを手動で承認する必要がないように、一部またはすべての実行エラーを自動で承認するようにジョブを設定することができます。

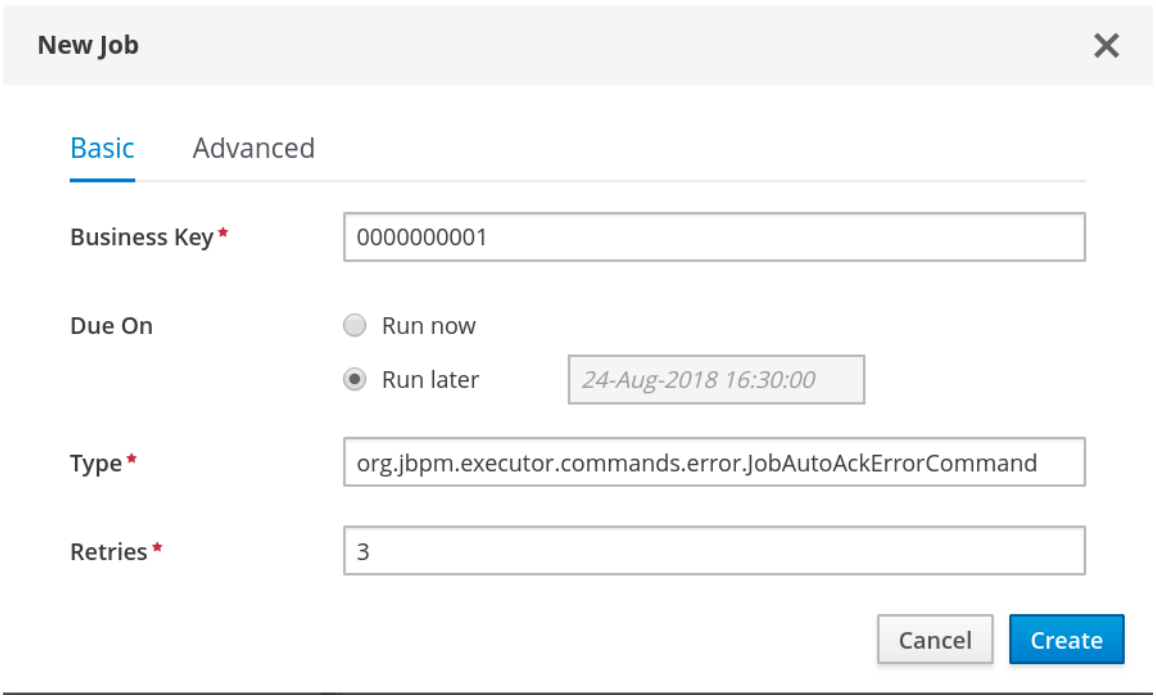
注記

自動承認ジョブを設定した場合、デフォルトで毎日ジョブが実行されます。実行エラーを一度だけ自動承認させるには、**SingleRun** パラメーターを **true** に設定します。

手順

1. Business Central で、**Menu → Manage → Jobs** を選択します。
2. 画面右上の **New Job** をクリックします。
3. **Business Key** フィールドにジョブの何らかの識別子を入力します。
4. **Type** フィールドに、自動承認ジョブの種類を入力します。
   - **org.jbpm.executor.commands.error.JobAutoAckErrorCommand**: タイプが **Job** のすべての実行エラーを承認します。この場合、エラーが関連するジョブは現在キャンセル、完了、または別の実行のために再スケジュールされています。
   - **org.jbpm.executor.commands.error.TaskAutoAckErrorCommand**: タイプが **Task** のすべての実行エラーを承認します。この場合、エラーが関連するタスクは終了状態 (完了、失敗、終了、廃止) にあります。
   - **org.jbpm.executor.commands.error.ProcessAutoAckErrorCommand**: あらゆるタイプのすべての実行エラーを承認します。この場合、エラーの発生元であるプロセスインスタンスは既に終了している (完了または中止) か、またはエラーの発生元であるタスクが既に終了しています。
5. **Due On** でジョブの完了時間を選択します。
   - ジョブをすぐに実行する場合は、**Run now** オプションを選択します。
   - 特定の時間にジョブを実行する場合は、**Run later** を選択します。**Run later** オプションの横に日時フィールドが表示されます。フィールドをクリックしてカレンダーを開き、ジョブの特定の日時をスケジュールします。

図31.3 自動承認ジョブのスケジューリング例



6. デフォルトでは、最初の実行後、ジョブは毎日1回実行されます。この設定を変更するには、次の手順を実行します。

   a. **Advanced** タブをクリックします。

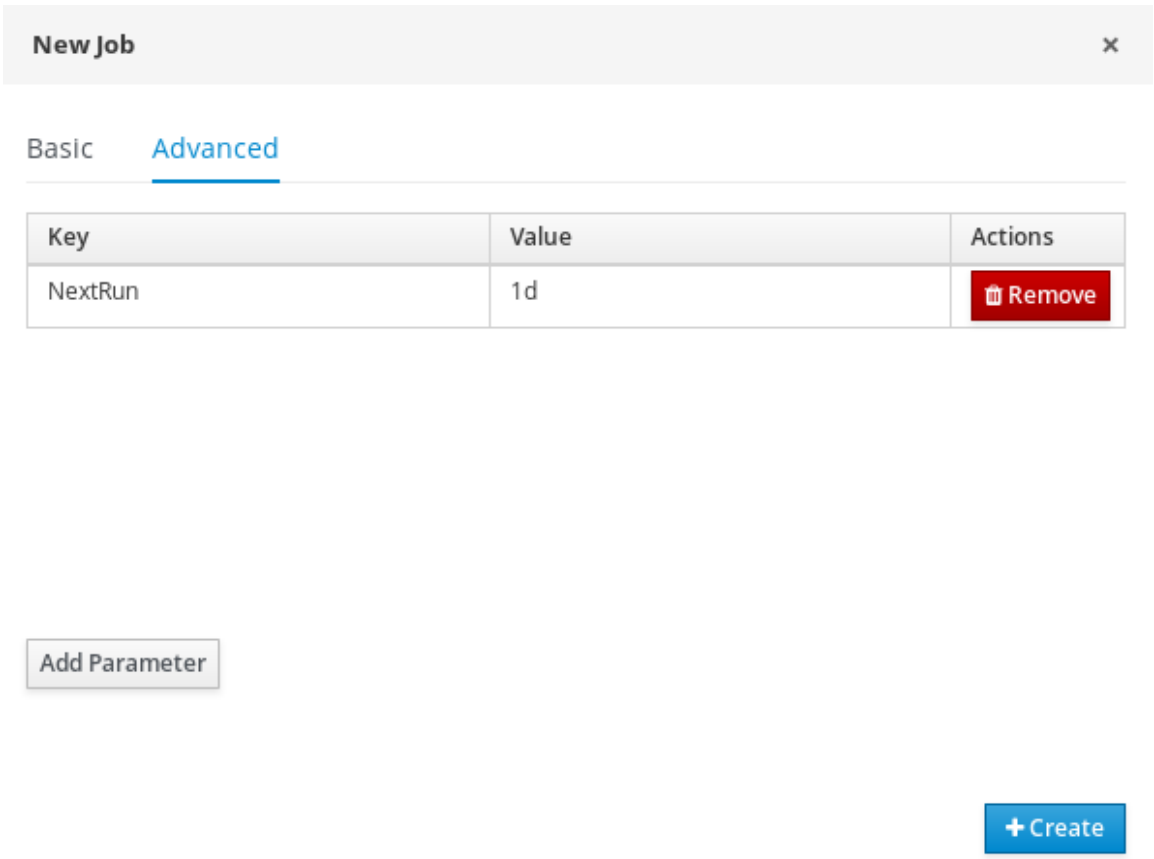
   b. **Add Parameter** ボタンをクリックします。

   c. ジョブに適用する設定パラメーターを入力します。

      - ジョブを一度だけ実行させたい場合は、**SingleRun** パラメーターを追加して、値を **true** にします。
      - ジョブを定期的に実行させたい場合は、**NextRun** パラメーターを追加して、値を有効な時間表現 (**2h**、**5d**、**1m** など) にします。

   d. オプション: カスタムエンティティーマネージャーのファクトリー名を設定するには、**EmfName** パラメーターを入力します。

図31.4 自動承認ジョブのパラメーター設定の例



7. **Create** をクリックしてジョブを作成し、**Manage Jobs** ページに戻ります。

31.5. エラー一覧のクリーンアップ

プロセスエンジンは、実行エラーを **ExecutionErrorInfo** データベーステーブルに格納します。テーブルをクリーンアップしてエラーを永久に除去したい場合は、**org.jbpm.executor.commands.ExecutionErrorCleanupCommand** コマンドでジョブをスケジューリングすることができます。

このコマンドは、完了または中止されたプロセスインスタンスに関連する実行エラーを削除します。

手順

1. Business Central で、**Menu → Manage → Jobs** を選択します。
2. 画面右上の **New Job** をクリックします。
3. **Business Key** フィールドにジョブの何らかの識別子を入力します。
4. **Type** フィールドに **org.jbpm.executor.commands.ExecutionErrorCleanupCommand** と入力します。
5. **Due On** でジョブの完了時間を選択します。
   - ジョブをすぐに実行する場合は、**Run now** オプションを選択します。
   - 特定の時間にジョブを実行する場合は、**Run later** を選択します。**Run later** オプションの横に日時フィールドが表示されます。フィールドをクリックしてカレンダーを開き、ジョブの特定の日時をスケジュールします。

6. Advanced タブをクリックします。

7. 必要に応じて、以下のいずれかのパラメーターを追加します。

    - **DateFormat**: パラメーター内の日付の書式を指定します。設定されていない場合は、**SimpleDateFormat** クラスのパターンと同様に **yyyy-MM-dd** が使用される。

    - **EmfName**: クエリーに使用するカスタムエンティティーマネージャーファクトリーの名前。

    - **SingleRun**: ジョブを 1 回実行するようにスケジューリングします。**true** に設定すると、ジョブは 1 回実行され、繰り返し実行されるようにスケジュールされません。

    - **NextRun**: ある期間内に繰り返し実行されるようにジョブをスケジューリングします。値は、**1d**、**5h**、**10m** などの有効な時間表現でなければなりません。

    - **OlderThan**: 設定された日付より古いエラーのみを削除します。値は日付でなければなりません。

    - **OlderThanPeriod**: 現在の時刻と比較して、指定された期間より古いエラーだけを削除します。値は、**1d**、**5h**、**10m** などの有効な時間表現でなければなりません。

    - **ForProcess**: プロセス定義に関連するエラーのみを削除します。値はプロセス定義の識別子でなければなりません。

    - **ForProcessInstance**: プロセスインスタンスに関連するエラーのみを削除します。値はプロセスインスタンスの識別子でなければなりません。

    - **ForDeployment**: デプロイメント識別子に関連するエラーのみを削除します。値はデプロイメントの識別子でなければなりません。

第32章 プロセスインスタンスの移行

プロセスインスタンス移行 (PIM) は、ユーザーインターフェイスとバックエンドを含むスタンドアロンのサービスです。Quarkus のミュータブル JAR ファイルとしてパッケージ化されています。プロセスインスタンスの移行サービスを使用して、2 つの異なるプロセス定義間の移行を定義できます。これは移行プランと呼ばれます。特定の KIE Server で実行中のプロセスインスタンスに対して、この移行プランを適用できます。

PIM サービスの詳細は、**KIE (Drools, OptaPlanner and jBPM)**の Process Instance Migration Service を参照してください。

32.1. プロセスインスタンスの移行サービスのインストール

プロセスインスタンス移行 (PIM) サービスを使用して、移行プランを作成、エクスポート、実行します。PIM サービスは、GitHub リポジトリー経由で提供されます。PIM サービスをインストールするには、GitHub リポジトリーのクローンを作成してから、サービスを実行して、Web ブラウザーにアクセスします。

前提条件

- バックアップを作成した Red Hat Process Automation Manager 開発環境でプロセスを定義している。
- Java Runtime Environment (JRE) バージョン 11 以降がインストールされている。

手順

1. Red Hat Process Automation Manager 7.12 の Software Downloads ページから **rhpam-7.12.0-add-ons.zip** ファイルをダウンロードします。
2. **rhpam-7.12.0-add-ons.zip** ファイルを展開します。
3. **rhpam-7.12.0-process-migration-service.zip** ファイルを抽出します。
4. 次のコマンドを入力して、データベーステーブルを作成します。**<user>** をユーザー名に、**<host>** をローカルホストの名前に置き換えます。

```
$ psql -U <user> -h <host> -d rhpam7 -f ~/process-migration/ddl-scripts/postgres/postgresql-quartz-schema.sql
$ psql -U <user> -h <host> -d rhpam7 -f ~/process-migration/ddl-scripts/postgres/postgresql-pim-schema.sql
```

5. ディレクトリーを **process-migration** ディレクトリーに変更します。
6. テキストエディターを使用して、次の内容で **servers.yaml** 設定ファイルを作成し、**process-migration** ディレクトリーに保存します。この例では、**<user_name>** と **<password>** を、KieServer にログインするための認証情報に置き換えます。

```
kieservers:
  - host: http://localhost:8080/kie-server/services/rest/server
    username: <user_name>
    password: <password>
```

7. テキストエディターを使用して、次の内容の **datasource.yaml** で **datasource.yaml** 設定ファイルを作成し、**process-migration** ディレクトリーに保存します。この例では、**<user_name>** と **<password>** を、データベースにログインするための認証情報に置き換えます。

```
quarkus:
  datasource:
    db-kind: postgresql
    jdbc:
      url: jdbc:postgresql://localhost:5432/rhpam7
    username: <user_name>
    password: <password>
```

8. **quarkus-run.jar** ファイルを再構築して、PostgreSQL ドライバーを含めます。

```
$ java -jar -Dquarkus.launch.rebuild=true -Dquarkus.datasource.db-kind=postgresql quarkus-app/quarkus-run.jar
```

このコマンドの出力は、以下のようになります。

```
INFO [io.qua.dep.QuarkusAugmentor] (main) Quarkus augmentation completed in 2657ms
```

9. **quarkus-app** JAR ファイルを実行します。

```
$ java -jar -Dquarkus.http.port=8090 -Dquarkus.config.locations=servers.yaml,datasource.yaml quarkus-app/quarkus-run.jar
```

このコマンドは、以下の例のような出力を返します。

```
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2022-03-11 13:04:18,261 INFO  [org.fly.cor.int.lic.VersionPrinter] (main) Flyway Community Edition 7.14.0 by Redgate
2022-03-11 13:04:18,262 INFO  [org.fly.cor.int.dat.bas.BaseDatabaseType] (main) Database: jdbc:postgresql://localhost:5432/rhpam7 (PostgreSQL 13.4)
2022-03-11 13:04:18,280 INFO  [org.fly.cor.int.com.DbMigrate] (main) Current version of schema "public": 1.0
2022-03-11 13:04:18,281 INFO  [org.fly.cor.int.com.DbMigrate] (main) Schema "public" is up to date. No migration necessary.
2022-03-11 13:04:18,601 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Freed 0 triggers from 'acquired' / 'blocked' state.
2022-03-11 13:04:18,603 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Recovering 0 jobs that were in-progress at the time of the last shut-down.
2022-03-11 13:04:18,603 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Recovery complete.
2022-03-11 13:04:18,603 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Removed 0 'complete' triggers.
2022-03-11 13:04:18,603 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Removed 0 stale fired job entries.
2022-03-11 13:04:18,624 INFO  [org.kie.ser.api.mar.MarshallerFactory] (main) Marshaller extensions init
2022-03-11 13:04:18,710 INFO  [org.kie.pro.ser.imp.KieServiceImpl] (main) Loaded kie server configuration for: org.kie.processmigration.model.config.KieServers$KieServer9579928Impl@4b6b5352
2022-03-11 13:04:18,715 INFO  [org.kie.pro.ser.RecoveryService] (main) Resuming ongoing
```

```
migrations ...
2022-03-11 13:04:18,856 INFO  [io.quarkus] (main) process-migration-service 7.59.0.Final-redhat-00006 on JVM (powered by Quarkus 2.2.3.Final-redhat-00013) started in 1.443s. Listening on: http://0.0.0.0:8090
2022-03-11 13:04:18,857 INFO  [io.quarkus] (main) Profile prod activated.
2022-03-11 13:04:18,857 INFO  [io.quarkus] (main) Installed features: [agroal, cdi, config-yaml, flyway, hibernate-orm, hibernate-orm-panache, jdbc-db2, jdbc-h2, jdbc-mariadb, jdbc-mssql, jdbc-mysql, jdbc-oracle, jdbc-postgresql, narayana-jta, quartz, resteasy, resteasy-jackson, scheduler, security, security-jdbc, security-ldap, security-properties-file, smallrye-context-propagation, smallrye-health]
```

10. Process Instance Migration アプリケーションにアクセスするには、Web ブラウザーに **http://localhost:8090/** と入力します。
11. プロンプトが表示されたら、ユーザー名 **admin** とパスワード **admin1!** を入力します。プロセスインスタンス移行コンソールが表示されます。

32.2. 移行プランの作成

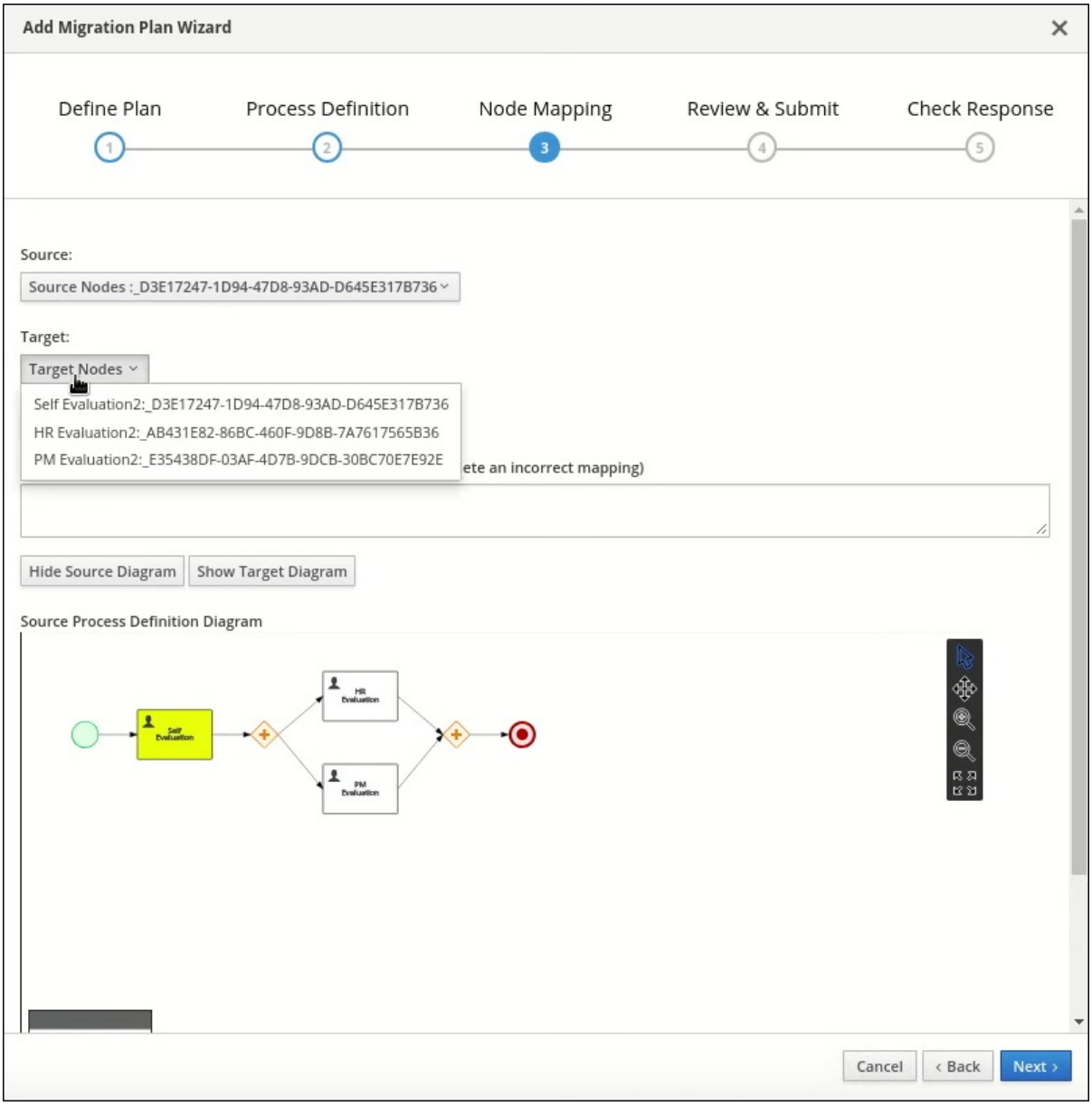
プロセスインスタンス移行 (PIM) サービスの Web UI で、移行プランと呼ばれる、2 つの異なるプロセス間の移行を定義できます。

前提条件

- バックアップを作成した Red Hat Process Automation Manager 開発環境でプロセスを定義している。
- プロセスインスタンス移行サービスが実行中である。

手順

1. Web ブラウザーで **http://localhost:8080** と入力します。
2. PIM サービスにログインします。
3. **Process Instance Migration** ページの右上隅の KIE サービスリストから、移行プランを追加する KIE サービスを選択します。
4. **Add Plan** をクリックします。**Add Migration Plan Wizard** ウィンドウが開きます。
5. **Name** フィールドで、移行プランの名前を入力します。
6. 必要に応じて、**Description** フィールドで、移行プランの説明を入力します。
7. **Next** をクリックします。
8. **Source ContainerID** フィールドで、ソースコンテナー ID を入力します。
9. **Source ProcessId** フィールドで、ソースプロセス ID を入力します。
10. **Copy Source To Target** をクリックします。
11. **Target ContainerID** フィールドで、ターゲットコンテナー ID を更新します。
12. **Retrieve Definition from backend** をクリックして **Next** をクリックします。



13. **Source Nodes** リストから、マッピングするソースノードを選択します。
14. **Target Nodes** リストから、マッピングするターゲットノードを選択します。
15. **Source Process Definition Diagram** ペインが表示されない場合は、**Show Source Diagram** をクリックします。
16. **Target Process Definition Diagram** ペインが表示されない場合は、**Show Target Diagram** をクリックします。
17. 必要に応じて、ダイアグラムペインでビューを変更するには、以下のタスクのいずれかを実行します。

    - テキストを選択するには、 アイコンを選択します。
    - パンするには、 アイコンを選択します。
    - ズームインするには、 アイコンを選択します。

- ズームアウトするには、 アイコンを選択します。
- ビューアーに適合するには、 アイコンを選択します。

18. **Map these two nodes** をクリックします。
19. **Next** をクリックします。
20. 必要に応じて、**JSON** ファイルとしてエクスポートするには、**Export** をクリックします。
21. **Review & Submit** タブで、プランをレビューして、**Submit Plan** をクリックします。
22. 必要に応じて、**JSON** ファイルとしてエクスポートするには、**Export** をクリックします。
23. 応答を確認して、**Close** をクリックします。

32.3. 移行プランの編集

プロセスインスタンス移行 (PIM) サービスの Web UI で移行プランを編集できます。変更できるのは、移行プランの名前、説明、指定されたノード、およびプロセスインスタンスです。

前提条件

- バックアップを作成した Red Hat Process Automation Manager 開発環境でプロセスを定義している。
- PIM サービスが実行している。

手順

1. Web ブラウザーで **http://localhost:8080** と入力します。
2. PIM サービスにログインします。
3. **Process Instance Migration** ページで、編集する移行プランの行にある **Edit Migration Plan** アイコンを選択します。**Edit Migration Plan** ウィンドウが開きます。
4. 各タブで、変更内容を加えます。
5. **Next** をクリックします。
6. 必要に応じて、**JSON** ファイルとしてエクスポートするには、**Export** をクリックします。
7. **Review & Submit** タブで、プランをレビューして、**Submit Plan** をクリックします。
8. 必要に応じて、**JSON** ファイルとしてエクスポートするには、**Export** をクリックします。
9. 応答を確認して、**Close** をクリックします。

32.4. 移行プランのエクスポート

プロセスインスタンス移行 (PIM) サービスの Web UI で、JSON ファイルとして移行プランをエクスポートできます。

前提条件

- バックアップを作成した Red Hat Process Automation Manager 開発環境でプロセスを定義している。
- PIM サービスが実行している。

手順

1. Web ブラウザーで **http://localhost:8080** と入力します。
2. PIM サービスにログインします。
3. **Process Instance Migration** ページで、実行する移行プランの行にある **Export Migration Plan** アイコンを選択します。**Export Migration Plan** ウィンドウが開きます。
4. 確認して、**Export** をクリックします。

32.5. 移行プランの実行

プロセスインスタンス移行 (PIM) サービスの Web UI で、移行プランを実行できます。

前提条件

- バックアップを作成した Red Hat Process Automation Manager 開発環境でプロセスを定義している。
- PIM サービスが実行している。

手順

1. Web ブラウザーで **http://localhost:8080** と入力します。
2. PIM サービスにログインします。
3. **Process Instance Migration** ページで、実行する移行プランの行にある **Execute Migration Plan** アイコンを選択します。**Execute Migration Plan Wizard** ウィンドウが開きます。
4. 移行プランテーブルから、移行する実行中のプロセスインスタンスの行にあるチェックボックスを選択して、**Next** をクリックします。
5. **Callback URL** フィールドで、コールバック URL を入力します。
6. **Run migration** の右側で、以下のタスクの 1 つを実行します。
   - 移行をすぐ実行するには、**Now** を選択します。

- 移行をスケジュールするには、**Schedule** を選択して、テキストフィールドで **06/20/2019 10:00 PM** など日時を入力します。

7. **Next** をクリックします。
8. 必要に応じて、**JSON** ファイルとしてエクスポートするには、**Export** をクリックします。
9. **Execute Plan** をクリックします。
10. 必要に応じて、**JSON** ファイルとしてエクスポートするには、**Export** をクリックします。
11. 応答を確認して、**Close** をクリックします。

32.6. 移行プランの削除

プロセスインスタンス移行 (PIM) サービスの Web UI で移行プランを削除できます。

前提条件

- バックアップを作成した Red Hat Process Automation Manager 開発環境でプロセスを定義している。
- PIM サービスが実行している。

手順

1. Web ブラウザーで **http://localhost:8080** と入力します。
2. PIM サービスにログインします。
3. **Process Instance Migration** ページで、削除する移行プランの行にある **Delete** アイコンをクリックします。**Delete Migration Plan** ウィンドウが開きます。
4. **Delete** をクリックして削除を確定します。

パート IV. ケース管理の設計およびビルド

開発者は、Business Central を使用して、ケース管理できるように、Red Hat Process Automation Manager アセットを設定できます。

ケース管理はビジネスプロセス管理 (BPM) とは異なります。目標を達成するために実行する一連の手順ではなく、ケース全体で処理される実際のデータに重点を置いています。ケースデータは、自動的なケース処理における情報の中で最も重要な要素ですが、ビジネスの状況や意思決定はケース作業担当者の管理下に置かれます。

Red Hat Process Automation Manager では、Business Central に **IT_Orders** サンプルプロジェクトが含まれます。本書では、ケース管理の概念を説明して例を提示する場合にこのサンプルを参照します。

ケース管理の使用ガイド のチュートリアルでは、Business Central で **IT_Orders** プロジェクトを作成してテストする方法を説明します。本ガイドでコンセプトを確認した後に、チュートリアルの説明に沿って、ご自身のケースプロジェクトを正常に作成してデプロイし、テストできるように進めてください。

前提条件

- Red Hat JBoss Enterprise Application Platform 7.4 がインストールされている。Red Hat JBoss Enterprise Application Platform 7.4 のインストールに関する情報は **Red Hat JBoss Enterprise Application Platform 7.4 Installation Guide** を参照してください。
- Red Hat Process Automation Manager をインストールしている。Red Hat Process Automation Manager のインストールに関する情報は、**Red Hat Process Automation Manager インストールの計画** を参照してください。
- Red Hat Process Automation Manager が稼働し、**user** ロールで Business Central にログインできる。ユーザーおよびパーミッションに関する情報は、**Red Hat Process Automation Manager インストールの計画** を参照してください。
- Showcase アプリケーションがデプロイされている。Showcase アプリケーションをインストールしてログインする方法は、**ケース管理への Showcase アプリケーションの使用** を参照してください。

第33章 ケース管理

ケース管理は、Business Process Management (BPM) の拡張機能で、適用可能なビジネスプロセスを管理します。

BPM は、反復可能で共通のパターンを持つタスクの自動化に使用する管理プラクティスで、プロセスを完全化して最適化を図ることに焦点を当てます。ビジネスプロセスは通常、ビジネスの目標へのパスを明確に定義し、モデル化されています。これにより、通常は、量産原理に基づいた、多くの予測可能性が必要となります。ただし、実在する多くのアプリケーションでは、(可能なパス、脱線、例外など) 開始と終了を完全に説明することができません。特定のケースでプロセス指向のアプローチを使用すると、複雑なソリューションとなり、管理が困難になります。

ケース管理は、ルーティンで、予測可能なタスクを対象する BPM の効率指向アプローチとは対照的に、繰り返さず、予測できないプロセスに対する問題解決を提供します。ここでは、プロセスが前もって予測できない、一回限りの状況が管理されます。ケース定義は、通常、特定のマイルストーン、そして最終的には企業目標となるように、直接または間接的につながる結合が弱いプロセスの断片で設定されます。一方、プロセスは、ランタイム時に発生する変更に応えて動的に管理されます。

Red Hat Process Automation Manager のケース管理には、以下のコアプロセスエンジン機能が含まれます。

- ケースファイルのインスタンス
- ケースごとのランタイム戦略
- ケースコメント
- マイルストーン
- ステージ
- アドホックフラグメント
- 動的タスクおよびプロセス
- ケース識別子 (相関キー)
- ケースのライフサイクル (閉じる、再開、キャンセル、破棄)

ケース定義は常にアドホックのプロセス定義で、明示的な開始ノードは必要ありません。ケース定義は、ビジネスユースケースの主なエントリーポイントです。

プロセス定義は、ケースでサポートされる設定概念として導入され、ケース定義に指定された通り、または必要に応じて追加処理に動的に取り込むために呼び出すことができます。ケース定義は、以下の新しいオブジェクトを定義します。

- アクティビティー (必須)
- ケースファイル (必須)
- マイルストーン
- ロール
- ステージ

第34章 CASE MANAGEMENT MODEL AND NOTATION (CMMN)

Business Central を使用して、Case Management Model and Notation (CMMN) ファイルのコンテンツをインポートして表示し、変更できます。プロジェクトを作成するには、ケース管理モデルをインポートしてから、そのケース管理モデルをアセットリストから選択して標準の XML エディターで表示または変更できます。

以下の CMMN コンストラクトが現在利用できます。

- Tasks (human task、process task、decision task、case task)
- Discretionary tasks (上記と同じ)
- ステージ
- マイルストーン
- Case file items
- Sentries (entry および exit)

次のタスクはサポート対象外です。

- 必須
- Repeat
- Manual activation

個別タスクの Sentry は、entry 基準に限定されていますが、entry と exit の基準は stages と milestones でサポートされています。Decision task はデフォルトで DMN decision にマッピングされます。イベントリスナーはサポートされていません。

Red Hat Process Automation Manager には CMMN のモデリング機能は含まれておらず、モデルの実行のみに焦点を当てています。

第35章 ケースファイル

ケースインスタンスは、ケース定義を含む1つのインスタンスで、ビジネスコンテキストをカプセル化します。ケースインスタンスデータはすべてケースファイルに保存され、特定のケースインスタンスに参加する可能性のあるすべてのプロセスインスタンスからアクセスできます。各ケースインスタンスと、ケースインスタンスのケースファイルは、他のケースから完全に分離されています。ケースファイルにアクセスできるのは、必要なケー出力ルに割り当てられているユーザーのみです。

ケースファイルは、ケースインスタンス全体のデータリポジトリーとして、ケースの管理に使用します。このファイルには、全ロール、データオブジェクト、データマップなどのデータが含まれます。ケースを完了してから、同じケースファイルを添付して、後日再度開くことができます。ケースインスタンスはいつでも終了でき、完了に特別な解決策を提示する必要はありません。

ケースファイルには、埋め込み型のドキュメント、参考資料、PDF 添付ファイル、Web リンクなどのオプションも含めることができます。

35.1. ケース ID 接頭辞の設定

caseId パラメーターは、ケースインスタンス ID の文字列値です。Red Hat Process Automation Manager デザイナーで **Case ID Prefix** を設定して、異なるケースタイプを識別できます。

以下の手順では **IT_Orders** サンプルプロジェクトを使用し、特定のビジネスニーズに合わせて、一意のケース ID 接頭辞を作成します。

前提条件

- Business Central で **IT_Orders** サンプルプロジェクトを開いている。

手順

1. Business Central で、**Menu → Design → Projects** に移動します。既存のプロジェクトがある場合は、**MySpace** のデフォルトのスペースをクリックして、**Add Project** ドロップダウンメニューから **Try Samples** を選択して、サンプルにアクセスできます。既存のプロジェクトがない場合には、**Try samples** をクリックします。
2. **IT_Orders** を選択し、**OK** をクリックします。
3. **Assets** ウィンドウで **orderhardware** ビジネスプロセスをクリックしてデザイナーを開きます。
4. キャンバスの空きスペースをクリックし、右上隅の **Properties** アイコンをクリックします。
5. 下方向にスクロールして、**Case Management** を展開します。
6. **Case ID Prefix** フィールドに ID 値を入力します。ID 形式は、内部で **ID-XXXXXXXXXX** として定義されています。ここで、**XXXXXXXXXX** は、ケースインスタンスに一意の ID を提供する生成された番号です。
   接頭辞が指定されていない場合、デフォルトの接頭辞には **CASE** と以下の識別子が使用されます。

   CASE-0000000001

   CASE-0000000002

CASE-0000000003

接頭辞は、任意で指定できます。たとえば、接頭辞 **IT** を指定する場合は、次の識別子が生成されます。

IT-0000000001

IT-0000000002

IT-0000000003

図35.1 ケース ID 接頭辞のフィールド

35.2. ケース ID 式の設定

以下の手順は、IT_Orders サンプルプロジェクトを使用して、設定したメタデータ属性キーで **caseId** を生成する式をカスタマイズする方法を説明します。

前提条件

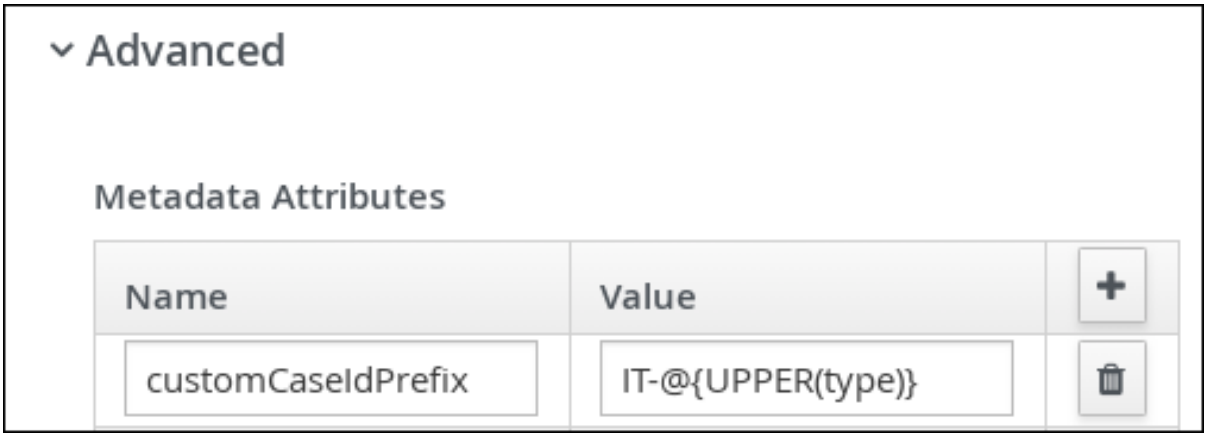
- Business Central で **IT_Orders** サンプルプロジェクトを開いている。

手順

1. Business Central で、**Menu → Design → Projects** に移動します。既存のプロジェクトがある場合は、**MySpace** のデフォルトのスペースをクリックして、**Add Project** ドロップダウンメニューから **Try Samples** を選択して、サンプルにアクセスできます。既存のプロジェクトがない場合には、**Try samples** をクリックします。
2. **IT_Orders** を選択し、**OK** をクリックします。
3. **Assets** ウィンドウで **orderhardware** ビジネスプロセスをクリックしてデザイナーを開きます。
4. キャンバスの空きスペースをクリックし、右上隅の **Properties** アイコンをクリックします。
5. **Advanced** メニューを展開して **Metadata Attributes** フィールドにアクセスします。
6. **customCaseIdPrefix** メタデータ属性に以下の関数を 1 つ指定します。
   - **LPAD**: 左パディング
   - **RPAD**: 右パディング

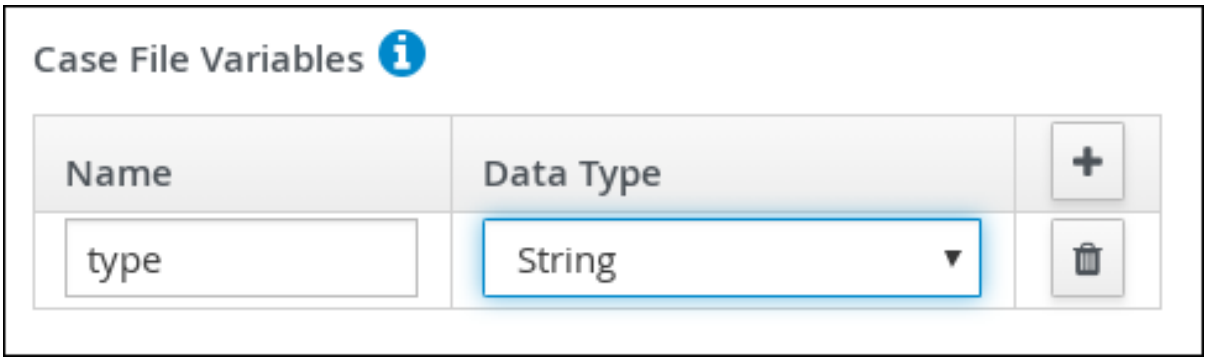
- TRUNCATE: 省略
- UPPER: 大文字

図35.2 customCaseIdPrefix メタデータ属性の UPPER 関数の設定



この例では、**type** は、Case File Variables フィールドに設定された変数です。この変数は、ランタイム時に **type1** の値に変更できます。**UPPER** は事前に組み込まれた関数で変数を大文字に変換します。ただし、**IT-** は静的な接頭辞となっています。そのため、動的なケース ID は **IT-TYPE1-0000000001**、**IT-TYPE1-0000000002**、および **IT-TYPE1-0000000003** のようになります。

図35.3 ケースファイル変数



customCaseIdPrefixIsSequence ケースメタデータ属性が **false** (デフォルト値 **true**) に設定されている場合は、ケースインスタンスでシーケンスが作成されず、**caseIdPrefix** 式がケース ID になります。たとえば、社会保障番号をもとにケース ID を生成した場合、特定のシーケンスまたはインスタンス ID は必要ありません。

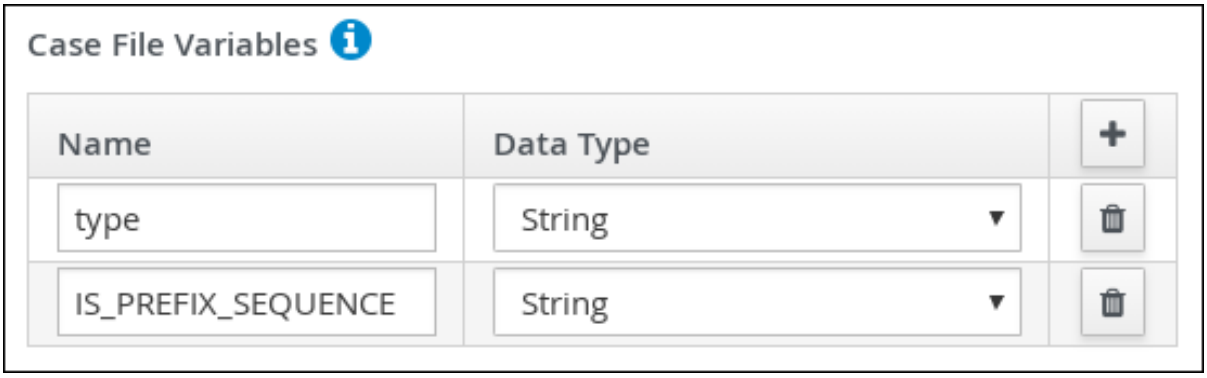
customCaseIdPrefixIsSequence メタデータ属性は必要に応じて追加し、**false** (デフォルト値 **true**) に設定してケース ID の番号順を無効にします。カスタムのケース ID に使用する式にケースファイル変数が含まれており、一般的なシーケンス値ではなく、一意のビジネス ID で表現されている場合に便利です。たとえば、社会保障番号をもとにケース ID を生成した場合、特定のシーケンスまたはインスタンス ID は必要ありません。以下の例で
は、**SOCIAL_SECURITY_NUMBER** は、ケースファイル変数として宣言された変数でもあります。

図35.4 customCaseIdPrefixIsSequence metadata attribute

IS_PREFIX_SEQUENCE ケースファイル変数は必要に応じてランタイム時にフラグとして追加され、ケース ID をシーケンスで生成するのを無効または有効にします。たとえば、個人医療保険の補償範囲にシーケンスの接頭辞を作成する必要はありません。複数家族の保険契約の場合、企業は **IS_PREFIX_SEQUENCE** のケース変数を **true** に設定し、家族の一人ずつ、順番に番号を付与します。

customCaseIdPrefixIsSequence メタデータ属性を静的に **false** として使用します。**IS_PREFIX_SEQUENCE** ケースファイル変数を使用して、ランタイム時に値を **false** に設定しても同じ結果が得られます。

図35.5 IS_PREFIX_SEQUENCE ケース変数



第36章 サブケース

サブケースは、他のケースを含めて複雑なケースを柔軟に設定できます。つまり、大規模で複雑なケースを複数の抽象階層、さらには複数のケースプロジェクトに分割できるようになります。これは、プロセスを複数のサブプロセスに分割するのと類似します。

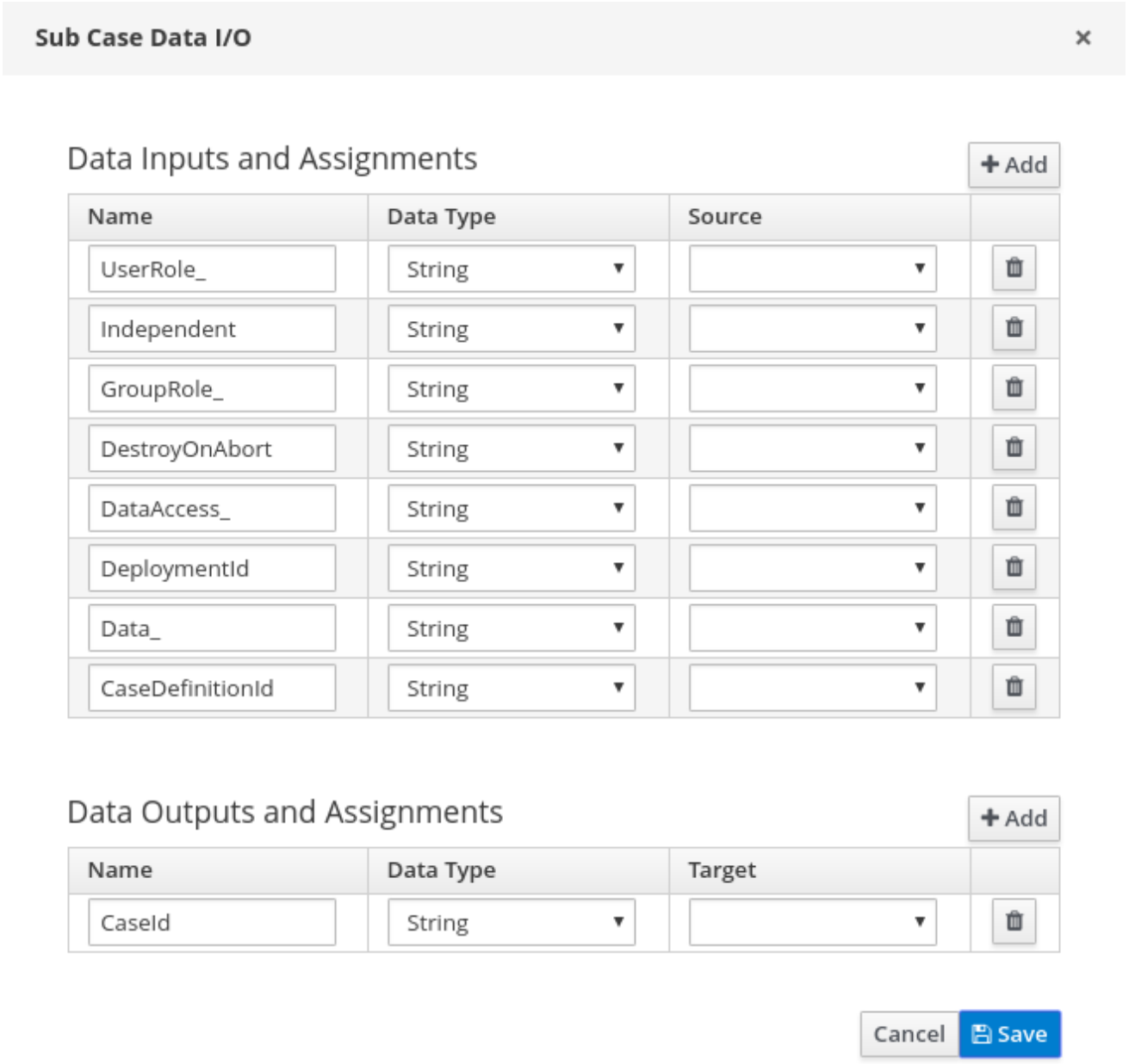
サブケースは、別のケースインスタンス、または通常のプロセスインスタンスから呼び出された別のケース定義です。これには、通常のケースインスタンスの機能がすべて含まれます。

- 専用のケースファイルがある。
- 別のケースインスタンスから分離している。
- ケー出力ルのセットを所有する。
- 独自のケース接頭辞がある。

ケースの定義にサブクラスを追加するには、プロセスデザイナーを使用できます。サブクラスは、ケースプロジェクト内のケースのことで、プロセスに含まれるサブプロセスに似ています。サブクラスは、通常のビジネスプロセスに追加することもできます。こうすることで、プロセスインスタンス内からケースを起動できます。

ケース定義へのサブケースの追加に関する詳細は、**ケース管理の使用ガイド** を参照してください。

Sub Case Data I/O ウィンドウでは、サブケースを設定して起動できるように、以下の入力パラメーターをサポートします。



独立

ケースインスタンスが独立しているかどうかを、プロセスデザイナーに通知する任意のインジケーター。独立している場合は、メインのケースインスタンスは、完了するまで待機しません。デフォルトでは、このプロパティーの値は **false** です。

GroupRole_XXX

ケー出力ルマッピングの任意のグループ。このケースインスタンスに所属するロール名はここで参照できるため、メインのケースの参加者を、サブケースの参加者にマッピングできます。つまり、メインケースに割り当てられたグループは、サブケースに自動的に割り当てられます。**XXX** はロール名に、プロパティーの値は、グループのロール割り当ての値に置き換えてください。

DataAccess_XXX

任意のデータアクセス制限。**XXX** は、データ項目の名前に、プロパティーの値はアクセス制限に置き換えてください。

DestroyOnAbort

サブケースのアクティビティーが中断された場合に、サブケースを取り消して、破棄するかどうかをプロセスエンジンに指示する任意のインジケーター。デフォルト値は、**true** です。

UserRole_XXX

ケー出力ルマッピングの任意のユーザー。ここで、ケースインスタンスのロール名を参照できるた

め、メインのケースの所有者を、サブケースの所有者にマッピングできます。つまり、メインケースに割り当てられたユーザーは、サブケースに自動的に割り当てられます。**XXX** はロール名に、プロパティーの値は、ユーザーのロール割り当ての値に置き換えてください。

Data_XXX

ケースインスタンスまたはビジネスプロセスからサブケースへの任意のデータマッピング。**XXX** は、対象のサブクラスに含まれるデータ名に置き換えます。このパラメーターは、必要に応じていくつでも指定できます。

DeploymentId

任意のデプロイメント ID (または KIE Server の場合はコンテキスト ID)。この ID で、対象のケース定義がどこにあるかを指定します。

CaseDefinitionId

開始するのに必要なケース定義 ID

CaseId

起動後のサブケースのケースインスタンス ID

第37章 アドホックおよび動的タスク

エンドツーエンドプロセスに厳密に従う代わりに、ケース管理を使用して、アドホックにタスクを実行できます。タスクは、ランタイム時にケースに動的に追加することもできます。

アドホックタスクが、ケースモデリングフェーズに定義されます。**AdHoc Autostart** として設定されていないアドホックタスクは任意であるため、ケース時に処理されない場合もあります。したがって、そのタスクは、1 つのイベントまたは 1 つの Java API からトリガーするる必要があります。

動的タスクはケース実行時に定義され、ケース定義モデルには表示されません。動的タスクは、ケース時に発生する特定の要求に対応します。Red Hat Process Automation Manager Showcase のデモにあるように、タスクはケースに追加され、ケースアプリケーションを使用していつでも作業できます。動的タスクは、Java および Remote API コールから追加することもできます。

動的タスクはユーザーまたはサービスアクティビティーとなりますが、アドホックタスクはどのタスクタイプにも設定できます。タスクタイプに関する詳細は、**BPMN モデルを使用したビジネスプロセスの作成** のプロセスデザイナーの BPMN2 タスクを参照してください。

動的プロセスは、ケースプロジェクトから再利用できるサブプロセスです。

内向き接続がないアドホックノードは、ノードの **AdHoc Autostart** プロパティーで設定でき、ケースインスタンスの起動時に自動的に開始します。

アドホックタスクは、ケース定義に設定される任意のタスクです。このタスクはアドホックであるため、通常はシグナルイベントまたは Java API コールによって発生します。

第38章 KIE SERVER REST API を使用してケースへの動的タスクおよびプロセスの追加

ランタイム時に動的タスクとプロセスをケースに追加して、ケースのライフサイクル時に発生する可能性がある予定外の変更を処理できます。動的アクティビティーはケース定義に定義されているわけでないため、定義したアドホックタスクまたはプロセスが可能な方法をシグナル化することはできません。

以下の動的アクティビティーをケースに追加できます。

- ユーザータスク
- サービスタスク (作業項目として実装されるすべてのタイプ)
- 再利用可能なサブプロセス

動的ユーザーおよびサービスタスクがケースインスタンスに追加され、直ちに実行されます。動的タスクの特性に従って、開始して完了を待つ (ユーザータスク) か、実行後に直接完了 (サービスタスク) します。動的サブプロセスの場合、プロセスエンジンは、この動的プロセスに対するプロセス定義を含む KJAR を要求して、ID でプロセスを探して実行します。このサブプロセスはケースに属し、ケースファイルのすべてのデータにアクセスします。

Swagger REST API アプリケーションを使用して、動的タスクおよびサブプロセスを作成します。

前提条件

- Business Central にログインしており、Showcase アプリケーションを使用してケースインスタンスを起動している。Showcase の使用方法の詳細は、**ケース管理のための Showcase アプリケーションの使用** を参照してください。

手順

1. Web ブラウザーで、以下の URL を開きます。
   http://localhost:8080/kie-server/docs
2. **Case instances :: Case Management** で利用可能なエンドポイントの一覧を開きます。
3. **POST** メソッドのエンドポイントを探し、動的アクティビティーを作成します。
   POST /server/containers/{id}/cases/instances/{caseId}/tasks

   ケースインスタンスに動的タスク (ペイロードに合わせてユーザーまたはサービスを選択) を追加します。

   POST /server/containers/{id}/cases/instances/{caseId}/stages/{caseStageId}/tasks

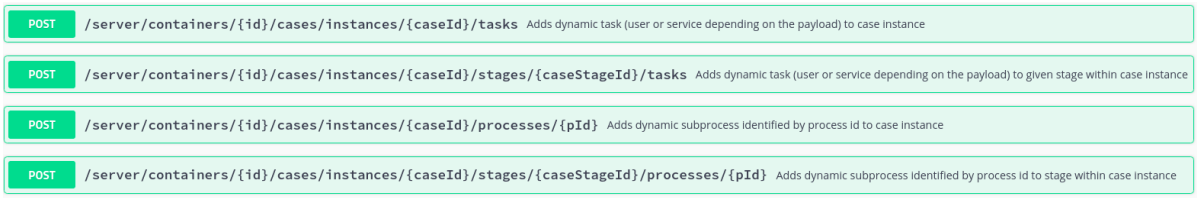
   ケースインスタンス内の特定のステージに動的タスク (ペイロードに合わせてユーザーまたはサービスを選択) を追加します。

   POST /server/containers/{id}/cases/instances/{caseId}/processes/{pId}

   プロセス ID で識別される動的サブプロセスをケースインスタンスに追加します。

   POST
   /server/containers/{id}/cases/instances/{caseId}/stages/{caseStageId}/processes/{pId}

   ケースインスタンス内のステージに、プロセス ID で識別される動的サブプロセスを追加します。



4. Swagger UI を開くには、動的タスクまたはプロセスの作成に必要な REST エンドポイントをクリックします。
5. **Try it out** をクリックして、動的アクティビティーの作成に必要なパラメーターとボディーを入力します。
6. **Execute** をクリックして、REST API を使用する動的タスクまたはサブプロセスを作成します。

38.1. KIE SERVER REST API を使用した動的ユーザータスクの作成

ケースの実行時には、REST API を使用して、動的ユーザータスクを作成できます。動的ユーザータスクを作成するには、次の情報を指定する必要があります。

- タスク名
- タスクの件名 (任意ですが、推奨されます)
- アクターまたはグループ (もしくはその両方)
- 入力データ

以下の手順に沿って、Swagger REST API ツールを使用して、Business Central で利用可能な **IT_Orders** サンプルプロジェクトの動的ユーザータスクを作成します。Swagger のない REST API でも、同じエンドポイントを利用できます。

前提条件

- Business Central にログインしており、Showcase アプリケーションを使用して IT Orders ケースインスタンスを起動している。Showcase の使用方法の詳細は、**ケース管理のための Showcase アプリケーションの使用** を参照してください。

手順

1. Web ブラウザーで、以下の URL を開きます。
http://localhost:8080/kie-server/docs.
2. **Case instances :: Case Management** で利用可能なエンドポイントの一覧を開きます。
3. 以下の **POST** メソッドのエンドポイントをクリックし、詳細を開きます。
/server/containers/{id}/cases/instances/{caseId}/tasks
4. **Try it out** をクリックしてから、以下のパラメーターを入力します。

表38.1 パラメーター

名前	説明
id	**itorders**
caseId	**IT-0000000001**

要求のボディー

```
{
 "name" : "RequestManagerApproval",
 "data" : {
   "reason" : "Fixed hardware spec",
   "caseFile_hwSpec" : "#{caseFile_hwSpec}"
  },
 "subject" : "Ask for manager approval again",
 "actors" : "manager",
 "groups" : ""
}
```

5. Swagger アプリケーションで、**Execute** をクリックして動的タスクを作成します。

この手順は、ケース **IT-000000001** に関連付けられている新しいユーザータスクを作成します。このタスクは、**manager** ケー出力ルに割り当てるユーザーに割り当てられます。このタスクには、2 つの入力変数があります。

- **reason**
- **caseFile_hwSpec**: プロセスまたはケースデータのランタイム取得を可能にする式として定義されます。

タスクによっては、タスク名で検索できる、ユーザーフレンドリーな UI を提供するフォームが提供されている場合があります。IT Orders のケースでは、**RequestManagerApproval** タスクには、KJAR に **RequestManagerApproval-taskform.form** フォームが含まれます。

タスクを作成すると、Business Central で、タスクが割り当てられたユーザーの **Task Inbox** にタスクが表示されます。

38.2. KIE SERVER REST API を使用した動的サービスタスクの作成

サービスタスクは通常、ユーザータスクより複雑ではありませんが、正常に実行するにはさらにデータが必要となる可能性があります。サービスタスクには以下の情報が必要です。

- **name**: アクティビティー名
- **nodeType**: ワークアイテムハンドラーの検索に使用するノードタイプ
- **data**: 正しく実行を処理するためのデータのマッピング

ケースの実行時に、ユーザータスクと同じエンドポイントを使用して動的サービスタスクを作成できますが、ボディーペイロードは異なります。

以下の手順に沿って、Swagger REST API ツールを使用して、Business Central で利用可能な **IT_Orders** サンプルプロジェクトの動的サービスタスクを作成します。Swagger のない REST API でも、同じエンドポイントを利用できます。

前提条件

- Business Central にログインしており、Showcase アプリケーションを使用して IT Orders ケースインスタンスを起動している。Showcase の使用方法の詳細は、**ケース管理のための Showcase アプリケーションの使用** を参照してください。

手順

1. Web ブラウザーで、以下の URL を開きます。
   http://localhost:8080/kie-server/docs
2. **Case instances :: Case Management** で利用可能なエンドポイントの一覧を開きます。
3. 以下の **POST** メソッドのエンドポイントをクリックし、詳細を開きます。
   /server/containers/{id}/cases/instances/{caseId}/stages/{caseStageId}/tasks
4. **Try it out** をクリックしてから、以下のパラメーターを入力します。

   表38.2 パラメーター

名前	説明
id	**itorders**
caseId	**IT-0000000001**

   要求のボディー

```
{
 "name" : "InvokeService",
 "data" : {
   "Parameter" : "Fixed hardware spec",
   "Interface" : "org.jbpm.demo.itorders.services.ITOrderService",
   "Operation" : "printMessage",
   "ParameterType" : "java.lang.String"
  },
 "nodeType" : "Service Task"
}
```

5. Swagger アプリケーションで、**Execute** をクリックして動的タスクを作成します。

この例では、Java ベースのサービスが実行されます。この例には、**org.jbpm.demo.itorders.services.ITOrderService** のパブリッククラスと、**String** 引数が 1 つ指定された **printMessage** パブリックメソッドとインターフェイスで設定されます。このサービスを実行すると、パラメーターの値がメソッドに渡されて実行されます。

サービスタスク作成に指定する数字、名前、他のタイプのデータは、サービスタスクのハンドラーの実装により異なります。提供されている例では、**org.jbpm.process.workitem.bpmn2.ServiceTaskHandler** ハンドラーが使用されています。

注記

カスタムサービスタスクの場合は、**Work Item Handlers** セクションのデプロイメント記述子にハンドラーが登録されていることを確認します。名前は、動的サービスタスクの作成に使用される **nodeType** と同じです。デプロイメント記述子に関する詳細は、Red Hat Process Automation Manager でのプロセスサービスの開発 を参照してください。

38.3. KIE SERVER REST API を使用した動的サブプロセスの作成

動的サブクラスを作成すると、任意のデータのみが提示されます。動的タスクの作成時には、特別なパラメーターはありません。

以下の手順では、Swagger REST API ツールを使用して、Business Central で利用可能な **IT_Orders** サンプルプロジェクトの動的なサブプロセスタスクを作成する方法を説明します。Swagger のない REST API でも、同じエンドポイントを利用できます。

前提条件

- Business Central にログインしており、Showcase アプリケーションを使用して IT Orders ケースインスタンスを起動している。Showcase の使用方法の詳細は、ケース管理のための Showcase アプリケーションの使用 を参照してください。

手順

1. Web ブラウザーで、以下の URL を開きます。
http://localhost:8080/kie-server/docs.
2. **Case instances :: Case Management** で利用可能なエンドポイントの一覧を開きます。
3. 以下の **POST** メソッドのエンドポイントをクリックし、詳細を開きます。
/server/containers/{id}/cases/instances/{caseId}/processes/{pId}
4. **Try it out** をクリックして以下のパラメーターを入力します。

表38.3 パラメーター

名前	説明
id	**itorders**
caseId	**IT-0000000001**
pId	**itorders-data.place-order**

pId は、作成するサブプロセスのプロセス ID です。

要求のボディー

```
{
 "placedOrder" : "Manually"
}
```

5. Swagger アプリケーションで、**Execute** をクリックして動的サブプロセスを開始します。

この例では、ケース ID **IT-0000000001** の **place-order** サブプロセスが IT 発注ケースで開始しています。Business Central の **Menu → Manage → Process Instances** の下で、このプロセスを確認できます。

説明に使用されている例を正しく実行したら、**place-order** プロセスがプロセスインスタンスの一覧に表示されます。プロセスの詳細を開き、プロセスの相関キーに IT 発注ケースインスタンス ID が含まれていることに注意してください。**Process Variables** 一覧には、REST API 本文に配信されているように、**Manually** 値を持つ **placedOrder** 変数が含まれます。

第39章 コメント

ケース管理では、コメントを使用すればケースインスタンス内での協業作業が容易になり、簡単にケース作業者が互いに情報を交換できるようになります。

コメントはケースインスタンスにバインドされます。ケースインスタンスはケースファイルの一部であるため、コメントを使用してインスタンスに対してアクションを実行できます。基本的なテキストベースのコメントには、CRUD (作成、読み取り、更新、削除) と同様の完全な操作セットを含めることができます。

第40章 CASE ROLES

ケー出力ルは、ユーザーがケース処理に参加する追加の抽象層を提供します。ロール、ユーザー、およびグループは、ケース管理の別の目的に使用されます。

ロール

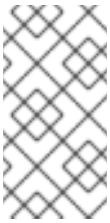
ロールは、ケースインスタンスの認証や、ユーザーアクティビティーの割り当てを可能にします。ユーザー、または1つ以上のグループを所有者ロールに割り当てることができます。所有者は、ケースを所有するユーザーになります。ケースの定義では、ロールはユーザーまたはグループ1つだけに制限されません。特定のユーザーまたはグループにタスクを割り当てる代わりに、ロールを使用してタスクの割り当てを指定することで、ケースを動的に保ちます。

Groups

グループとは、特定のタスクを実行できるユーザー、または指定の責任が割り当てられたユーザーの集合です。グループには何人でも割り当てることができ、ロールにはどのグループでも割り当てることができます。グループのメンバーをいつでも追加または変更できます。特定のタスクにグループをハードコーディングしないでください。

ユーザー

ユーザーとは、ロールに割り当てたり、グループに追加したりして、特定のタスクを割り当てることができる個人を指します。



注記

プロセスエンジンまたは KIE Server で **unknown** という名前のユーザーは作成しないでください。**unknown** ユーザーアカウントは、superuser のアクセス権限があるシステム名用に予約されています。**unknown** ユーザーアカウントでは、ログインしているユーザーがない場合に、SLA 違反リスナーに関連するタスクを実行します。

以下の例では、以下の情報で、前述のケース管理の概念をホテル予約にどのように適用するかを説明します。

- **ロール**: **Guest**
- **グループ**: **Receptionist**、**Maid**
- **ユーザー**: **Marilyn**

Guest のロールを割り当てると、関連ケースの特定の作業に影響があり、ケースインスタンスごとに固有です。すべてのケースインスタンスには、独自のロール割り当てがあります。ロールに割り当てることができるユーザーまたはグループの数はケースの **Cardinality** で制限されています。これは、プロセス設計者やケース定義でのロール作成時に設定されます。たとえば、ホテル予約ケースではゲストロールが1つ、**IT_Orders** サンプルプロジェクトでは IT ハードウェア業者ロールが 2 つです)。

ロールが定義されている場合は、ロールがケース定義の一部としてユーザー1人またはグループ1つにハードコードされておらず、ケースインスタンスごとに違うものを指定できるようにする必要があります。ケースのロール割り当てが重要なのは、このような理由からです。

ロールは、ケースの開始時や、ケースがアクテイブになった時点で割り当てまたは割り当ての解除ができます。ロールは任意ですが、ケース定義でロールを使用して、整理されたワークフローを維持します。

重要

タスク割り当てに実際のユーザーまたはグループ名を使用する代わりに、ロールを使用します。これにより、必要に応じて、ケースと、ユーザーまたはグループを割り当てるタイミングを遅らせることができます。

ロールはユーザーまたはグループに割り当てられ、ケースインスタンスの起動時にタスクを実行する権限があります。

40.1. ケー出力ルの作成

プロセスデザイナーでケースの設計時に、ケース定義でケースのロールを作成して、定義できます。ケースのロールは、ケースの定義レベルで設定して、ケースインスタンスを処理するアクターと分離させることができます。また、ロールは、ユーザータスクに割り当てるか、ケースのライフサイクル全体で問い合わせの参照として使用することができますが。固有のユーザーまたはユーザーのグループとして、ケースには定義されていません。

ケースインスタンスには、ケースの作業を実際に処理する個人が含まれます。新規ケースインスタンスを開始する場合にはロールを割り当ててください。ケースのランタイム中にケースのロール割り当てを変更して、ケースの柔軟性を保つことができますが、以前のロール割り当てをもとにすでに作成されているタスクには効果がありません。ロールに割り当てられたアクターには柔軟性がありますが、ロール自体はどのケースも同じままです。

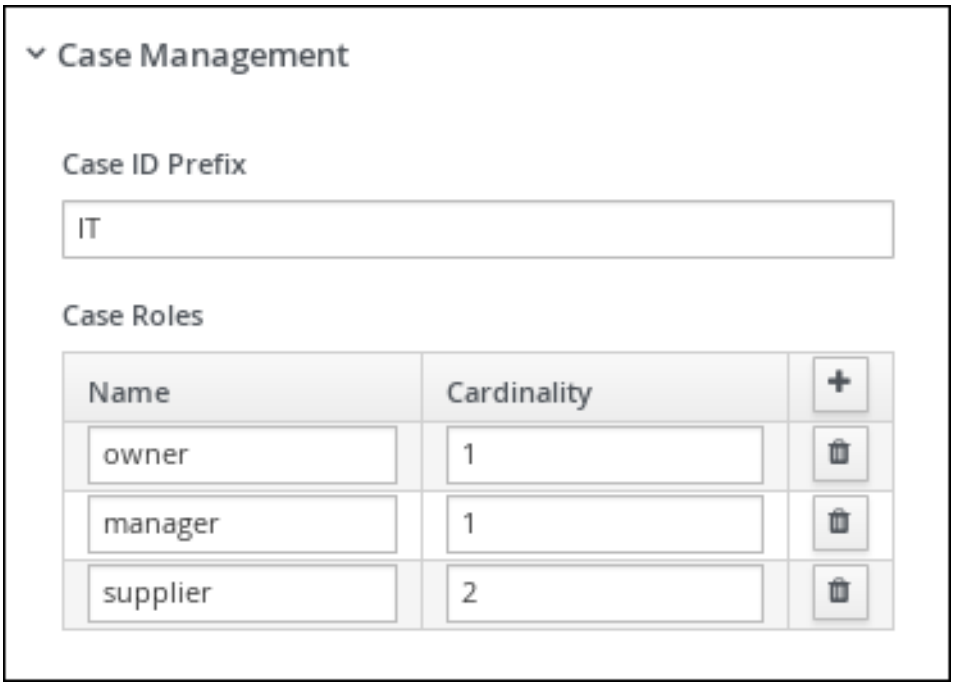
前提条件

- ケース定義が含まれるケースプロジェクトが Business Central に存在する。
- プロセスデザイナーでケース定義アセットが開いている。

手順

1. ケースに関連するロールを定義するには、エディターのキャンバスの空のスペースをクリックし、 をクリックして **Properties** メニューを開きます。
2. **Case Management** を展開してケー出力ルを追加します。
   ケースのロールには、ロールの名前とケースカーディナリティーが必要です。ケースカーディナリティーとは、ケースインスタンスでロールに割り当てられたアクターの数です。たとえば、**IT_Orders** サンプルのケース管理プロジェクトには、以下のロールが含まれます。

図40.1 ITOrders ケー出力ル



この例では、ケースの **owner** にアクター (ユーザーまたはグループ) 1 つのみを、**manager** ロールには 1 つのアクターのみを割り当てることができます。**supplier** ロールには、アクターを 2 つ割り当てることができます。ケースによっては、ロールに設定済みのケースカーディナリティーをもとに、特定のロールにいくつでもアクターを割り当てることができます。

40.2. ロールの認証

ロールには、Showcase アプリケーションまたは REST API を使用して新しいケースインスタンスを開始するときに、特定のケース管理タスクを実行する権限があります。

以下の手順では、REST API を使用して新しい IT 発注ケースを開始します。

前提条件

- **IT_Orders** サンプルプロジェクトが Business Central に実装されており、KIE Server にデプロイされている。

手順

1. 以下のエンドポイントで **POST** REST API コールを作成します。
   http://host:port/kie-server/services/rest/server/containers/itorders/cases/itorders.orderhardware/instances
   - **itorders**: KIE Server にデプロイしているコンテナーエイリアス。
   - **itorders.orderhardware**: ケース定義の名前。
2. 要求ボディーに以下のロール設定を追加します。

```
{
  "case-data" : {  },
  "case-user-assignments" : {
    "owner" : "cami",
    "manager" : "cami"
```

```
  },
  "case-group-assignments" : {
    "supplier" : "IT"
  }
}
```

これにより、定義されたロールを持つ新しいケースが開始するほか、開始済みで取り組む準備ができているアクティビティーを自動開始します。ロールのうち 2 つはユーザーへの割り当て (**owner** および **manager**) で、3 つ目はグループへの割り当て (**supplier**) です。

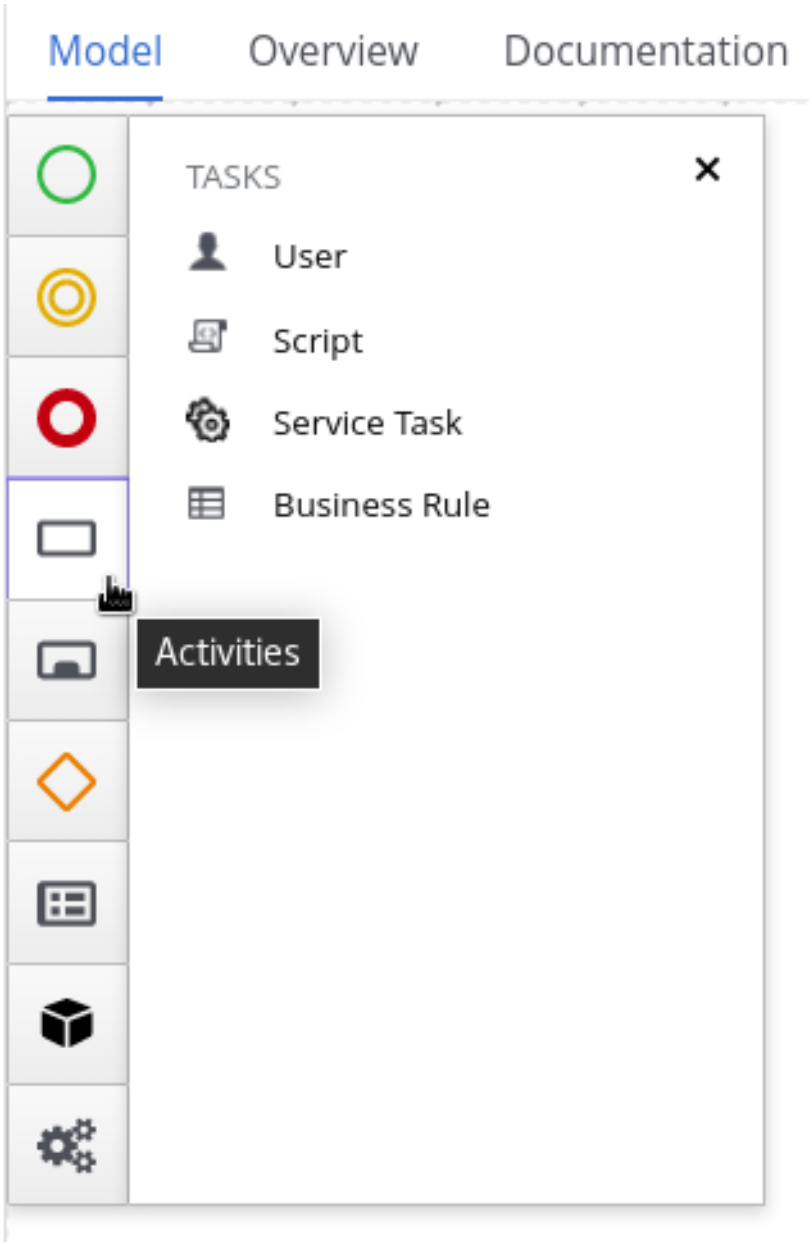
ケースインスタンスが正常に開始すると、ケース ID **IT-0000000001** を返します。

Showcase アプリケーションを使用して新規ケースインスタンスを開始する方法は、**ケース管理への Showcase アプリケーションの使用** を参照してください。

40.3. ロールへのタスクの割り当て

ケース管理プロセスは、ランタイム時に動的に発生する変更に対応するために、できるだけ柔軟である必要があります。たとえば、新しいケースインスタンスまたはアクティブなケースのユーザー割り当てを変更するなどです。このため、ケース定義ではロールを単一のユーザーまたはグループのセットにハードコードしないようにしてください。代わりに、ロールの割り当ては、ケース作成時にロールに割り当てられたユーザーまたはグループを使用して、ケース定義においてタスクノードで定義することができます。

Red Hat Process Automation Manager には、ビジネスプロセスの作成を簡略化する、事前定義済みのノードタイプが各種含まれます。事前定義済みのノードパネルは、ダイアグラムエディターの左側に置かれます。

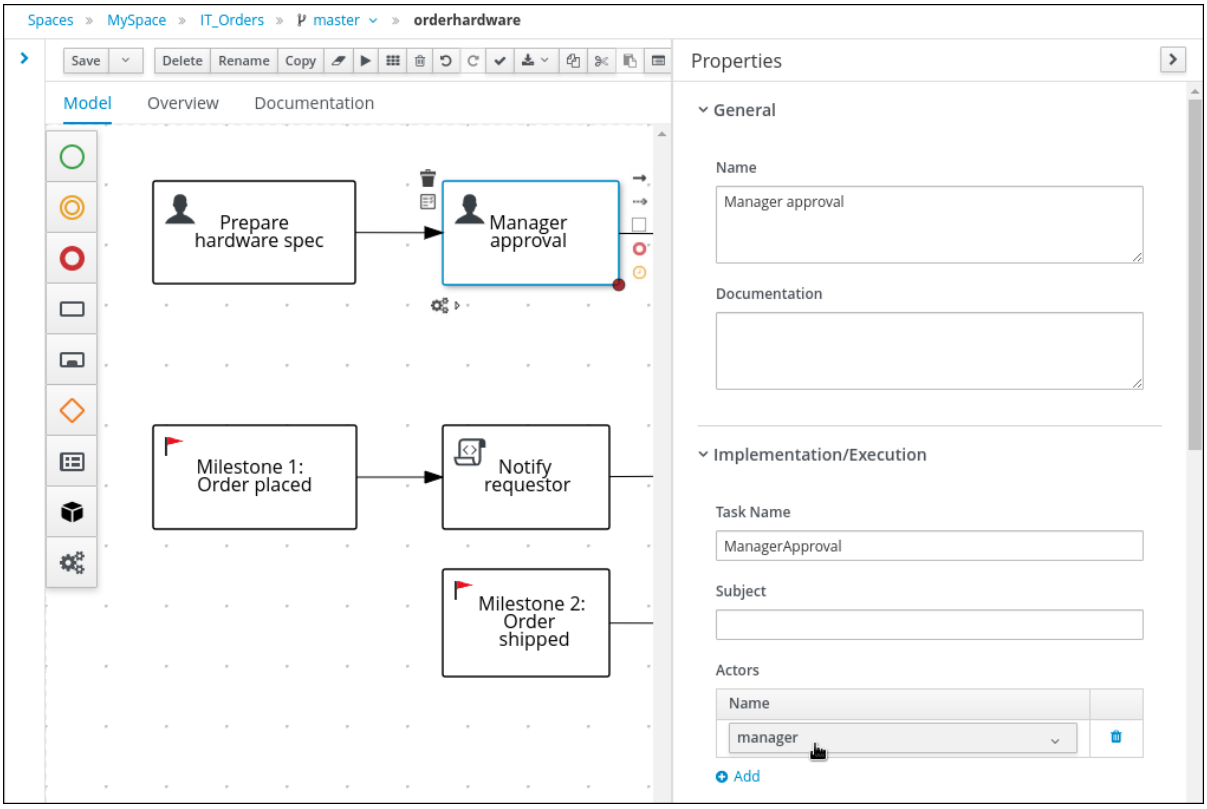


前提条件

- ケース定義が、ケース設定レベルに設定したケー出力ルで作成されている。ケー出力ルの作成方法は「ケー出力ルの作成」を参照してください。

手順

1. デザイナーパレットで **Activities** メニューを開き、ケース定義に追加するユーザーまたはサービスタスクをプロセスデザイナーキャンバスにドラッグします。
2. タスクノードを選択した状態で、 をクリックし、デザイナーの右側にある **Properties** パネルを開きます。
3. **Implementation/Execution** を展開し、**Actors** プロパティーの下にある **Add** をクリックして、タスクを割り当てるロール名を選択するか、入力します。グループの割当も同じように、**Groups** プロパティーを使用します。
   たとえば、**IT_Orders** のサンプルプロジェクトでは、**Manager approval** ユーザータスクが **manager** ロールに割り当てられています。



この例では、**Prepare hardware spec** ユーザータスクが完了すると、**manager** ロールに割り当てられているユーザーは、Business Central の **Task Inbox** で **Manager approval** を受け取ります。

ロールに割り当てられているユーザーはケースのランタイム時に変更できますが、タスクそのものには引き続き同じロールが割り当てられます。たとえば、**manager** ロールに最初に割り当てられたユーザーが (病気などで) 時間休をとる場合、または予定外に退職する場合などが考えられます。そのような状況でこの変更に応えるには、**manager** ロールの割り当てを編集して、そのロールに関連付けられているタスクに他のユーザーを割り当てることができます。

ケースのランタイム時にロールの割り当てを変更する方法については、「Showcase を使用してランタイム時にケースのロール割り当ての修正」または「REST API を使用してランタイム時にケースのロール割り当ての修正」を参照してください。

40.4. SHOWCASE を使用してランタイム時にケースのロール割り当ての修正

Showcase アプリケーションを使用して、ケースのランタイム時にケースインスタンスのロール割り当てを変更できます。ロールはケース定義に定義され、ケースのライフサイクルでタスクに割り当てます。ロールは事前に定義されるためランタイム時に変更できませんが、ロールに割り当てたアクターを、ケースタスクを実行するユーザーに変更できます。

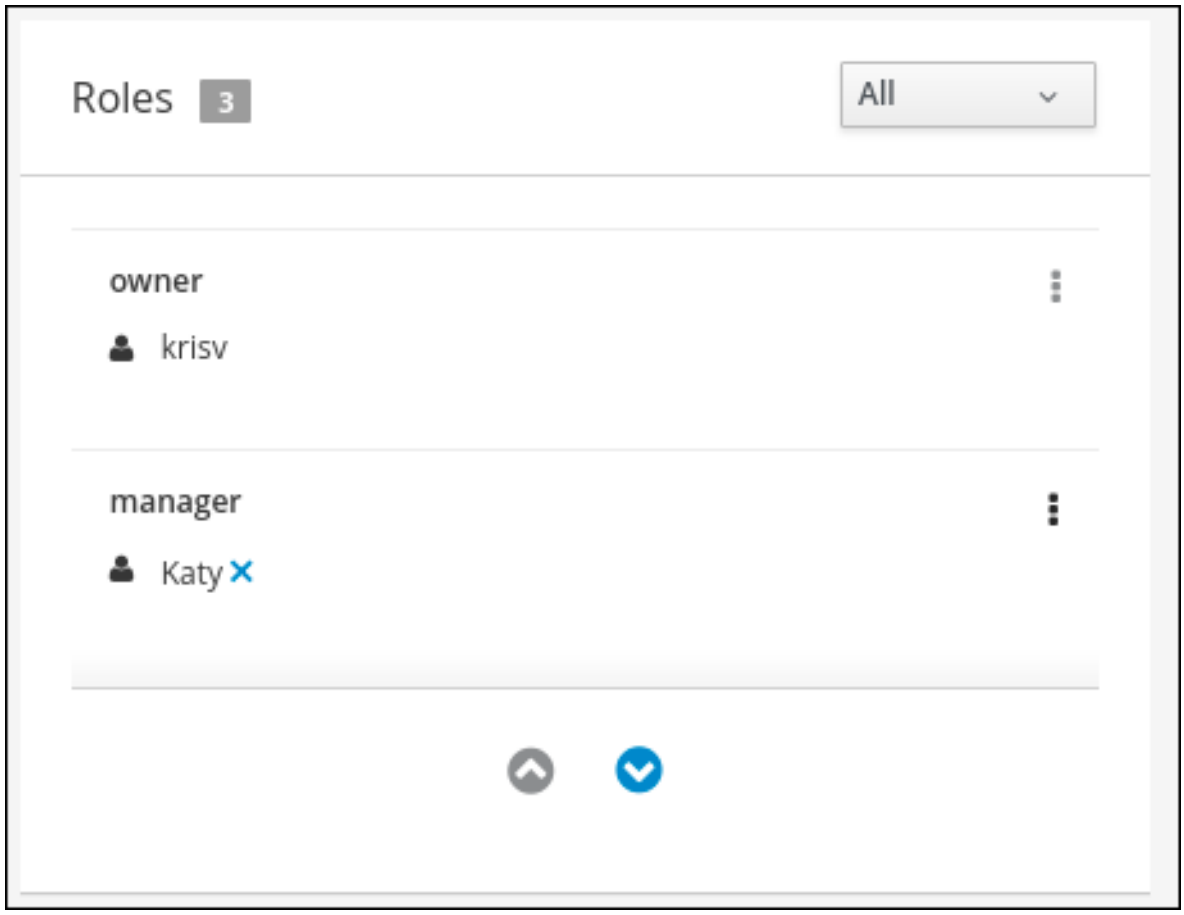
前提条件

- アクティブなケースインスタンスがあり、その中ですでにユーザーまたはグループが最低でも 1 つのケー出力ルに割り当てられている。

手順

1. Showcase アプリケーションで、**Case list** から作業するケースをクリックし、ケースの概要を開きます。

2. ページの右下の **Roles** ボックスで、変更するロール割り当てを探します。



3. ロール割り当てからユーザーまたはグループを削除するには、割り当ての横にある をクリックします。確認ウィンドウで、**Remove** をクリックして、ロールからユーザーまたはグループを削除します。

4. ロールからすべてのロール割り当てを削除するには、ロールの横にある をクリックし、**Remove all assignments** オプションを選択します。確認ウィンドウで、**Remove** をクリックして、ロールからユーザーとグループ割り当てすべてを削除します。

5. ロール割り当てを別のユーザーまたはグループに変更するには、ロールの横にある をクリックし、**Edit** オプションを選択します。

6. **Edit role assignment** ウィンドウで、ロール割り当てから削除する割り当て先の名前を削除します。ロールに割り当てるユーザーの名前を **User** フィールドに入力するか、割り当てるグループを **Group** フィールドに追加します。
ロール割り当ての編集時に、1 つ以上のユーザーまたはグループが割り当てられている必要があります。

7. **Assign** をクリックしてルールの割り当てを完了します。

40.5. REST API を使用してランタイム時にケースのロール割り当ての修正

REST API または Swagger アプリケーションを使用して、ケースのランタイム時にケースインスタンスのロール割り当てを変更できます。ロールはケース定義に定義され、ケースのライフサイクルでタスクに割り当てます。ロールは事前に定義されるためランタイム時に変更できませんが、ロールに割り当てたアクターを、ケースタスクを実行するユーザーに変更できます。

以下の手順には、**IT_Orders** サンプルプロジェクトをもとにした例が含まれます。Swagger アプリケーション、または他の REST API クライアントと同じ REST API エンドポイントを使用するか、Curl を使用します。

前提条件

- IT 発注ケースインスタンスを、**owner**、**manager**、および **supplier** のロールがアクターにすでに割り当てられている状態で開始している。

手順

1. 以下のエンドポイントで **GET** リクエストを使用して現在のロール割り当ての一覧を取得します。
   http://localhost:8080/kie-server/services/rest/server/containers/{id}/cases/instances/{caseId}/roles

   表40.1パラメーター

名前	説明
id	**itorders**
caseId	**IT-0000000001**

   これにより、以下の応答が返されます。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<case-role-assignment-list>
    <role-assignments>
        <name>owner</name>
        <users>Aimee</users>
    </role-assignments>
    <role-assignments>
        <name>manager</name>
        <users>Katy</users>
    </role-assignments>
    <role-assignments>
        <name>supplier</name>
        <groups>Lenovo</groups>
    </role-assignments>
</case-role-assignment-list>
```

2. **manager** ロールに割り当てられているユーザーを変更する場合は、最初に **DELETE** を使用して、ユーザー **Katy** からロール割り当てを削除する必要があります。
   /server/containers/{id}/cases/instances/{caseId}/roles/{caseRoleName}

   Swagger クライアントリクエストに以下の情報を追加します。

   表40.2 パラメーター

名前	説明
id	**itorders**
caseId	**IT-0000000001**
caseRoleName	**manager**
user	**Katy**

Execute をクリックします。

3. 最初の手順の **GET** リクエストを再実行し、**manager** ロールにユーザーが割り当てられなくなったことを確認します。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<case-role-assignment-list>
    <role-assignments>
        <name>owner</name>
        <users>Aimee</users>
    </role-assignments>
    <role-assignments>
        <name>manager</name>
    </role-assignments>
    <role-assignments>
        <name>supplier</name>
        <groups>Lenovo</groups>
    </role-assignments>
</case-role-assignment-list>
```

4. 以下のエンドポイントで **PUT** リクエストを使用して、**Cami** ユーザーを **manager** ロールに割り当てます。
/server/containers/{id}/cases/instances/{caseId}/roles/{caseRoleName}

Swagger クライアントリクエストに以下の情報を追加します。

表40.3 パラメーター

名前	説明
id	**itorders**
caseId	**IT-0000000001**
caseRoleName	**manager**
user	**Cami**

Execute をクリックします。

5. 最初の手順の **GET** リクエストを再実行し、**manager** ロールが **Cami** に割り当てられていることを確認します。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<case-role-assignment-list>
    <role-assignments>
        <name>owner</name>
        <users>Aimee</users>
    </role-assignments>
    <role-assignments>
        <name>manager</name>
        <users>Cami</users>
    </role-assignments>
    <role-assignments>
        <name>supplier</name>
        <groups>Lenovo</groups>
    </role-assignments>
</case-role-assignment-list>
```

第41章 ステージ

ケース管理ステージはタスクの集まりです。ステージは、プロセスデザイナーを使用して定義できるアドホックサブプロセスで、マイルストーンなどの、別のケース管理ノードに含まれる可能性もあります。マイルストーンは、1つのステージまたは複数のステージが完了した場合に完了するように設定されます。したがって、マイルストーンはステージの完了によりアクティベートまたは達成することができ、ステージにはマイルストーンを1つまたは複数追加できます。

たとえば、患者のトリアージケースでは、最初のステージでは、明らかな身体症状を観察して書き留めたり、その症状が何かを患者に説明してもらい、2番目のステージでテストを行い、3番目のステージで診断および治療を行います。

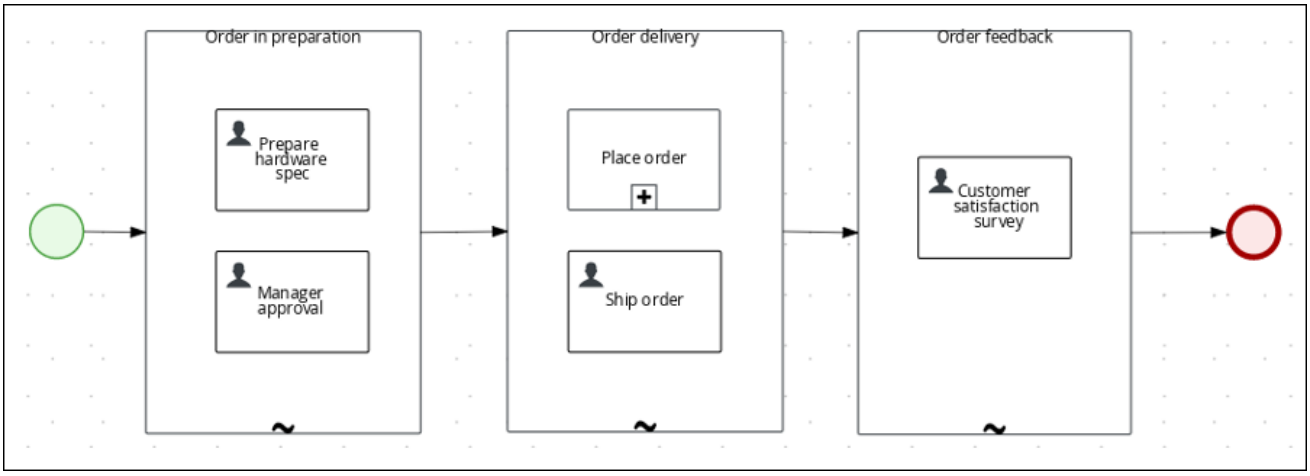
ステージを完了する方法が3つあります。

- 完了条件
- 終端の終了イベント
- **Completion Condition** を **autocomplete** に設定して、ステージにアクティブなタスクがなくなったら、自動的にそのステージを完了します。

41.1. ステージの定義

ステージは、プロセスデザイナーを使用して BPMN2 でモデル化できます。ステージは、関連するタスクをグループ化する方法であり、ステージがアクティブ化された場合、ケースの次のステージが始まる前に完了する必要があるアクティビティーを明確に定義します。たとえば、ステージを使用して、以下の方法で IT_Orders ケース定義を設定することもできます。

図41.1IT_Orders プロジェクトステージの例



手順

1. ダイアグラムエディターの左側にある事前定義済みのノードパネルから、**Adhoc** サブプロセスノードをデザインキャンバスにドラッグアンドドロップして、ステージノードの名前を指定します。
2. ステージをアクティベートする方法を定義します。
   - 受信ノードがステージをアクティベートした場合は、ステージを、受信ノードのシーケンスフローラインに接続します。

- シグナルイベントで、このステージが代わりにアクティベートされている場合は、シグナルノードに、最初の手順で設定したステージの名前で **SignalRef** を設定します。
- または、条件が満たされると、ステージをアクティベートするように **AdHocActivationCondition** プロパティーを設定します。

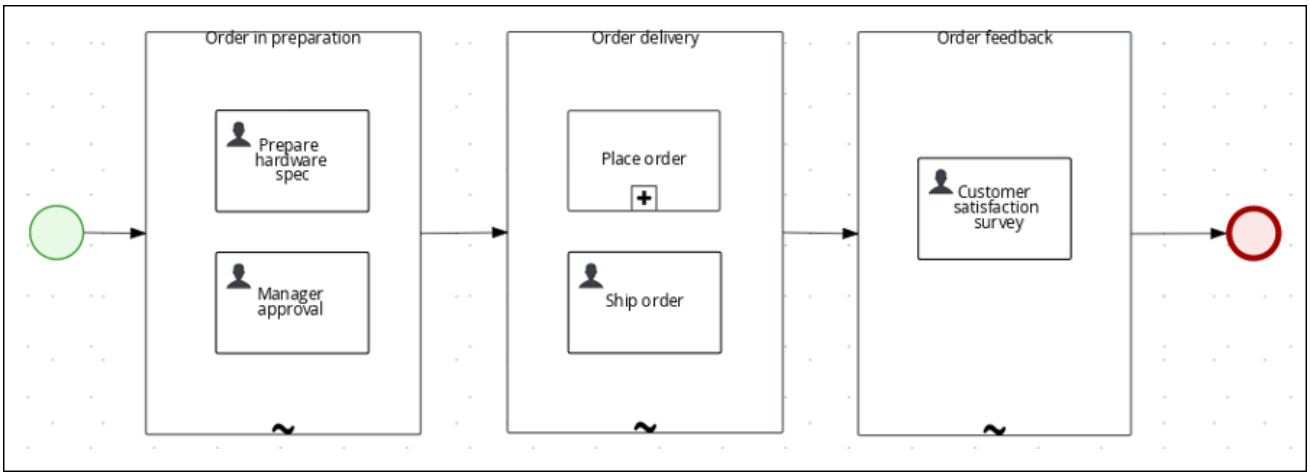
3. ステージにタスクノードを追加する余裕を持たせるために、必要に応じてノードのサイズを変更します。
4. 関連タスクをステージに追加して、必要に応じて設定します。
5. (任意) ステージの完了条件を設定します。アドホックサブプロセスとして、ステージはデフォルトで **autocomplete** として設定されます。これは、ステージが自動的に完了し、ステージ内のすべてのインスタンスがアクティブでなくなると、ケース定義の次のアクティビティーが発生することを示しています。
   完了条件を変更するには、ステージノードを選択して、右側の **Properties** パネルを選択し、**Implementation/Execution** を展開して、必要な完了条件になるように free-form Drools 式を使用して **AdHocCompletionCondition** プロパティーを変更します。ステージ完了条件に関する情報は、「ステージのアクティベーションおよび完了条件の設定」を参照してください。
6. ステージを設定したら、シーケンスフローラインを使用して、ケース定義内の次のアクティビティーに接続します。

41.2. ステージのアクティベーションおよび完了条件の設定

開始ノード、中間ノード、または手動の API コールを使用してステージを発生できます。

free-form Drools ルールを使用して、マイルストーンの完了条件を設定するのと同じ方法で、アクティベーションと完了条件の両方を含めてステージを設定できます。たとえば、**IT_Orders** サンプルプロジェクトでは、**Milestone 2: Order shipped** の完了条件 (**org.kie.api.runtime.process.CaseData(data.get("shipped") == true)**) を、ここで使用されている **Order delivery** の完了条件として使用することも可能です。

図41.2 IT_Orders プロジェクトステージの例



ステージをアクティベートする **AdHocActivationCondition** プロパティーを設定するアクティベーション条件は、Free Form Drools ルールを使用しても設定できます。

前提条件

- Business Central プロセスデザイナーでケース定義を作成している。

- アドホックサブプロセスを、ステージとして使用されるケース定義に追加している。

手順

1. ステージを選択した状態で、 をクリックし、デザイナーの右側にある **Properties** パネルを開きます。

2. **Implementation/Execution** を展開して、**AdHocActivationCondition** プロパティーエディターを開き、開始ノードのアクティベーション条件を定義します。たとえば、**autostart: true** を設定して、新規ケースインスタンスが開始されたら、ステージが自動的にアクティベートされるようにします。

3. **AdHocCompletionCondition** はデフォルトでは、**autocomplete** に設定されています。これを変更するには、free-form Drools 式を使用して完了条件を入力します。たとえば、**org.kie.api.runtime.process.CaseData(data.get("ordered") == true)** と設定して、以前の例の 2 つ目のステージをアクティベートします。

IT_Orders サンプルプロジェクトで使用する条件に関する例や情報は **ケース管理の使用ガイド** を参照してください。

41.3. ステージへの動的タスクの追加

動的タスクは、REST API 要求を使用してランタイム時にケースステージに追加できます。これは、ケースインスタンスに動的タスクを追加することに似ていますが、タスクが追加されるステージの **caseStageId** を定義する必要もあります。

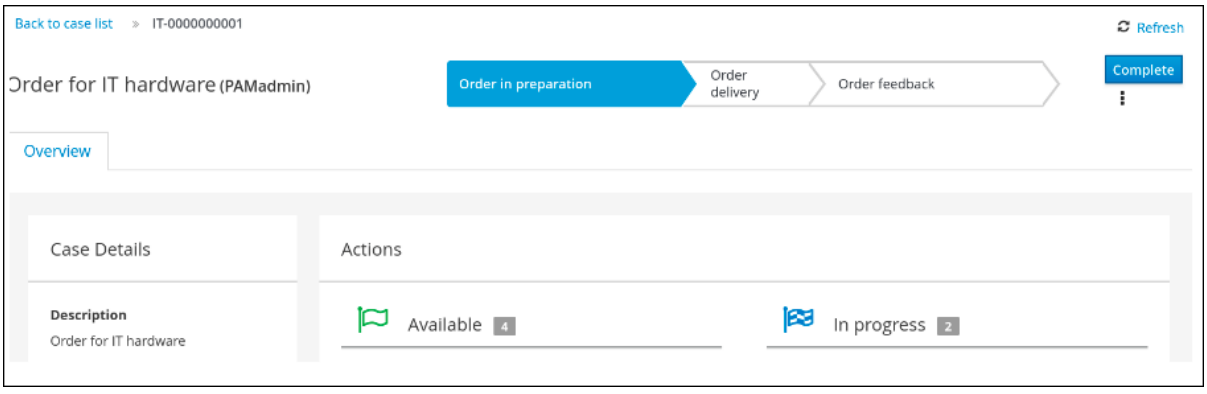
以下の手順に沿って、Swagger REST API ツールを使用して、Business Central で利用可能な IT_Orders サンプルプロジェクトの動的タスクをステージに追加します。Swagger のない REST API でも、同じエンドポイントを利用できます。

前提条件

- 以前の例に示されているように、**IT_Orders** サンプルプロジェクトの BPMN2 ケース定義は、マイルストーンではなくステージを使用して再設定できます。ケース管理向けにステージを設定する方法は、「ステージの定義」を参照してください。

手順

1. Showcase アプリケーションを使用して新規ケースインスタンスを開始します。Showcase の使用方法の詳細は、**ケース管理のための Showcase アプリケーションの使用** を参照してください。
このケースはステージを使用して作成されているため、ケース詳細ページにはステージの追跡が表示されます。



最初のステージは、ケースインスタンスの開始時に自動的に開始します。

2. **manager** ユーザーとして、Business Central の **Menu → Track → Task Inbox** の下で、ハードウェア明細書を承認し、ケースの進捗を確認します。

   a. Business Central で **Menu → Manage → Process Instances** の順にクリックし、アクティブケースインスタンス **IT-0000000001** を開きます。

   b. **Diagram** をクリックして、ケースの進捗を表示します。

3. Web ブラウザーで、以下の URL を開きます。
   http://localhost:8080/kie-server/docs.

4. **Case instances :: Case Management** で利用可能なエンドポイントの一覧を開きます。

5. 以下の **POST** メソッドのエンドポイントをクリックし、詳細を開きます。
   /server/containers/{id}/cases/instances/{caseId}/stages/{caseStageId}/tasks

6. **Try it out** をクリックして、以下のパラメーターを完了します。

   表41.1 パラメーター

名前	説明
id	**itorders**
caseId	**IT-0000000001**
caseStageId	**Order delivery**

   caseStageId は、ケース定義に含まれるステージ名です。このケース定義で、動的タスクが作成されます。これには、動的またはサービスタスクペイロードを指定できます。Red Hat Process Automation Manager でのプロセスサービスの開発 を参照してください。

動的タスクをステージに追加したら、ステージを完了して、ケースフローの次の項目にプロセスを進ませるために、その動的タスクを完了する必要があります。

第42章 マイルストーン

マイルストーンとは、プロセスデザイナーパレットにマイルストーンノードを追加して、ケース定義デザイナーで設定できる、特別なサービスタスクのことです。新規ケース定義の作成時に、**AdHoc Autostart** として設定されたマイルストーンは、デフォルトでデザインパレットに含まれます。新規作成したマイルストーンはデフォルトで **AdHoc Autostart** には設定されません。

ケース管理のマイルストーンは、ステージの最後に発生するのが通常ですが、他のマイルストーンを達成した結果として発生する場合もあります。マイルストーンには、進捗を追跡するために、条件を定義する必要があります。マイルストーンは、ケースにデータを追加すると、ケースファイルのデータに反応します。また、マイルストーンは、ケースインスタンス内の達成地点を表します。これは、重要業績評価指標 (KPI) の追跡や、完了前のタスクの特定に有用な場合があります。

マイルストーンには、ケース実行中の以下のいずれのか状態を指定できます。

- **Active**: 条件はマイルストーンで定義されているが、条件がまだ満たされていない。
- **Completed**: マイルストーンの条件が満たされ、達成されたため、このケースは次のタスクに進むことができる。
- **Terminated**: マイルストーンがケースプロセスから除外され、必要なくなっている。

マイルストーンが使用可能または完了している間は、シグナルによって手動でトリガーすることも、ケースインスタンスの開始時に **AdHoc Autostart** が設定されている場合は自動的にトリガーすることもできます。マイルストーンは何回でもトリガーできますが、条件が満たされている場合には、直接マイルストーンが達成されます。

42.1. マイルストーンの設定およびトリガー

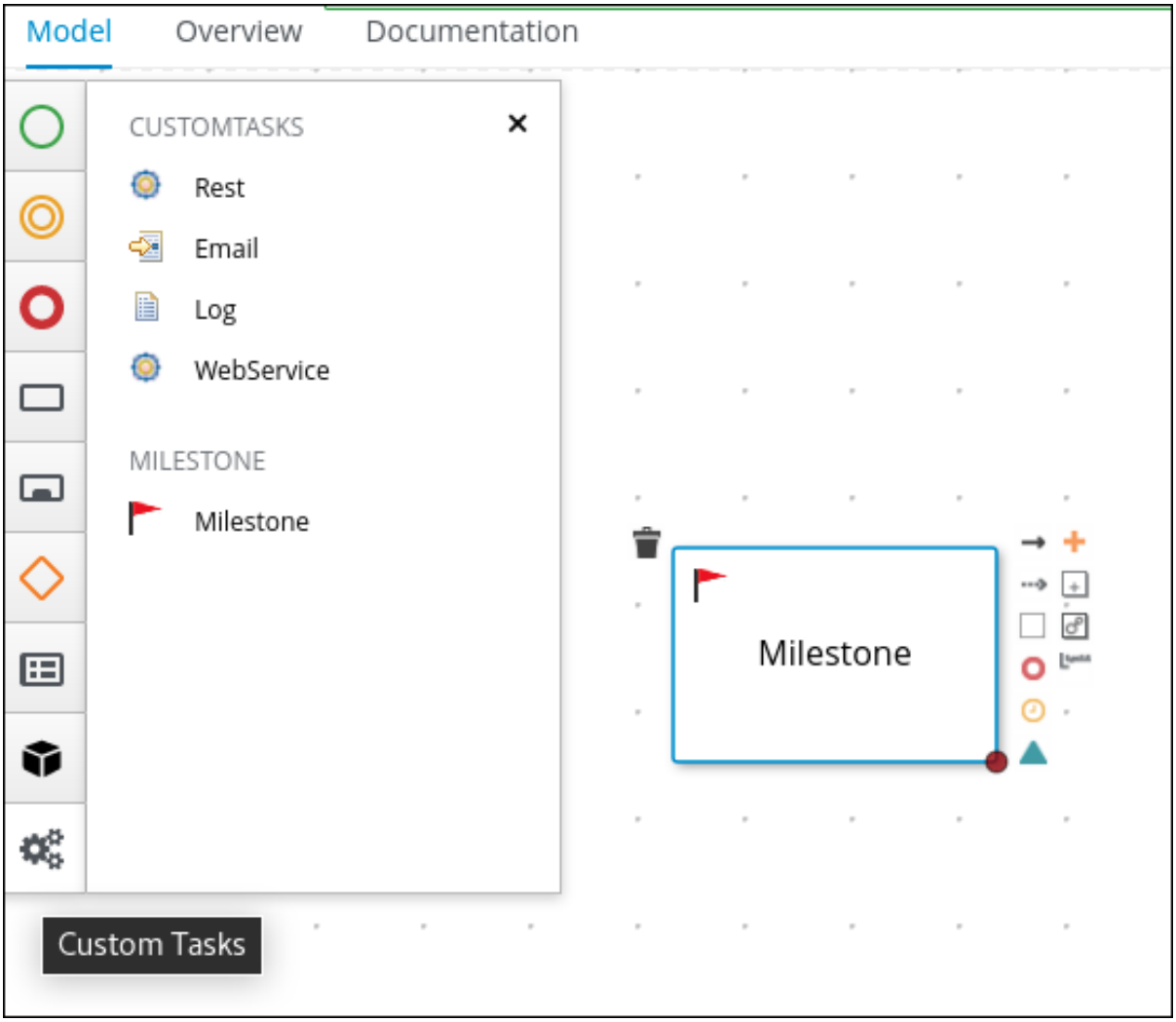
マイルストーンは、ケースインスタンスの開始時に自動的に開始するように設定できます。または、ケース設計時に手動で設定したシグナルを使用して発生させることもできます。

前提条件

- Business Central でケースプロジェクトが作成されている。
- ケース定義が作成されている。

手順

1. ダイアグラムエディターの左側にある事前定義済みのノードパネルから、**Milestone** オブジェクトをパレットにドラッグアンドドロップします。



2. マイルストーンを選択した状態で、 をクリックし、デザイナーの右側にある **Properties** パネルを開きます。

3. **Data Assignments** を展開して完了条件を追加します。マイルストーンには、デフォルトで **Condition** パラメーターが含まれます。

4. マイルストーンに完了条件を定義するには、**Source** 一覧から **Constant** を選択します。条件は Drools 構文で定義する必要があります。

5. **Implementation/Execution** を展開して、**AdHoc Autostart** プロパティーを設定します。

   - ケースインスタンスの開始時に自動的に開始する必要があるマイルストーンの場合は、チェックボックスを選択して、このプロパティーを **true** に設定します。

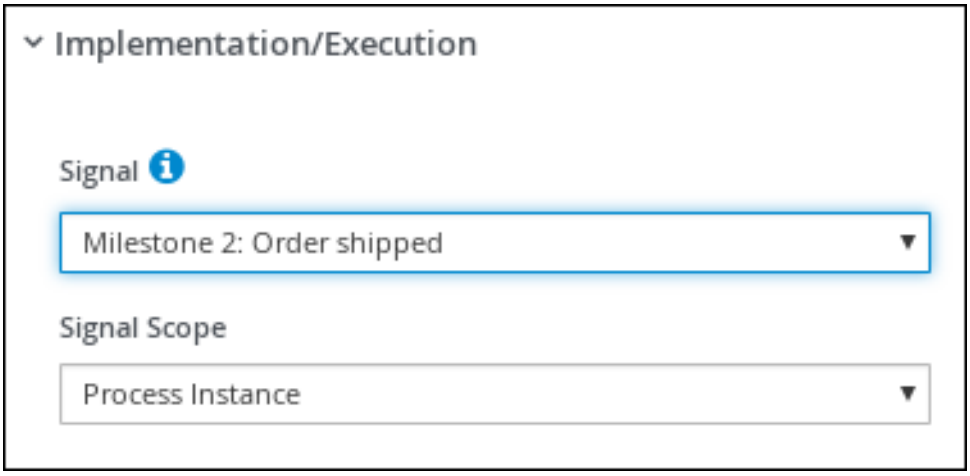
   - シグナルイベントで発生させるマイルストーンの場合は、チェックボックスを選択せずに、このプロパティーを **false** に設定します。

6. 必要に応じて、ケースゴールが達成した場合にマイルストーンを発生させるシグナルイベントを設定します。

   a. ケース設計パレットでシグナルイベントを選択した状態で、右側に **Properties** パネルを開きます。

   b. **Signal Scope** プロパティーを **Process Instance** に設定します。

   c. **SignalRef** 式エディターを開いて、発生させるマイルストーンの名前を入力します。



7. **Save** をクリックします。

第43章 変数タグ

ランタイム時に使用するデータを格納する変数。変数の動作をより細かく制御するには、BPMN ケースファイルでケース変数とローカル変数にタグ付けできます。タグは、特定の変数にメタデータとして追加する単純な文字列値です。

Red Hat Process Automation Manager は、ケースとローカル変数の以下のタグをサポートします。

- **required**: ケースを開始するための要件として変数を設定します。要件である変数なしでケースインスタンスを起動すると、Red Hat Process Automation Manager は **VariableViolationException** エラーを生成します。
- **readonly**: 変数が情報提供のみを目的としており、設定できるのはケースの実行中に 1 回のみであることを示します。readonly 変数の値がいずれかの時点で変更されると、Red Hat Process Automation Manager は **VariableViolationException** エラーを生成します。
- **restricted**: **VariableGuardProcessEventListener** で使用するタグで、既存のロールをもとに変数を変更できるパーミッションが付与されていることを示します。2 つ目のコンストラクターを使用して、新規タグ名を渡す場合には、**restricted** タグは他のタグ名に置き換えることができます。

VariableGuardProcessEventListener クラスは、**DefaultProcessEventListener** クラスから拡張されたもので、2 つの異なるコンストラクターをサポートします。

- **VariableGuardProcessEventListener**

```
public VariableGuardProcessEventListener(String requiredRole, IdentityProvider
identityProvider) {
    this("restricted", requiredRole, identityProvider);
}
```

- **VariableGuardProcessEventListener**

```
public VariableGuardProcessEventListener(String tag, String requiredRole, IdentityProvider
identityProvider) {
    this.tag = tag;
    this.requiredRole = requiredRole;
    this.identityProvider = identityProvider;
}
```

  したがって、以下の例に示すように、許可されたロール名とユーザーロールを返す ID プロバイダーを使用して、イベントリスナーをセッションに追加する必要があります。

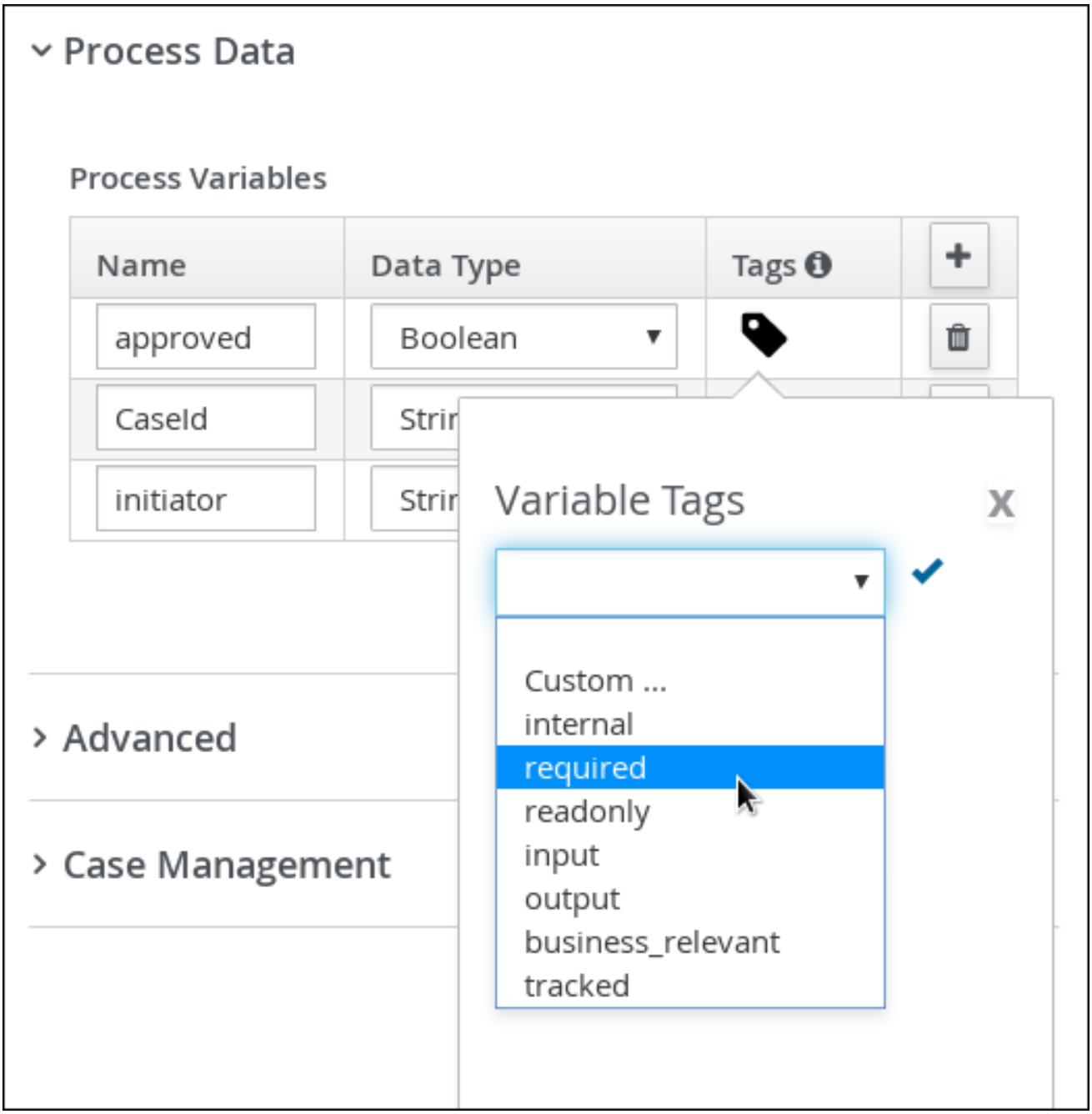
```
ksession.addEventListener(new VariableGuardProcessEventListener("AdminRole",
myIdentityProvider));
```

  上記の例では、**VariableGuardProcessEventListener** メソッドで、変数にセキュリティー制約タグ (**restricted**) が付いているかどうかを確認します。必要なロールがユーザーに割り当てられていない場合 (例: **AdminRole**) には、Red Hat Process Automation Manager は **VariableViolationException** エラーを生成します。注記: Business Central UI (例: **internal**、**input**、**output**、**business-relevant**、および **tracked**) に表示される変数タグは、Red Hat Process Automation Manager ではサポートされません。

タグは、**![CDATA[TAG_NAME]]** 形式で定義されたタグ値を使用し、**customTags** メタデータプロパティーとして BPMN プロセスソースファイルに直接追加できます。

たとえば、以下の BPMN プロセスは、**required** タグを **approved** プロセス変数に適用します。

図43.1 BPMN モデラーでタグ付けされた変数の例



BPMN ファイルでタグ付けされた変数の例

```
<bpmn2:property id="approved" itemSubjectRef="ItemDefinition_9" name="approved">
  <bpmn2:extensionElements>
    <tns:metaData name="customTags">
      <tns:metaValue><![CDATA[required]]></tns:metaValue>
    </tns:metaData>
  </bpmn2:extensionElements>
</bpmn2:property>
```

必要に応じて、変数に複数のタグを使用できます。BPMN ファイルでカスタム変数タグを定義して、Red Hat Process Automation Manager プロセスイベントリスナーが変数データを利用できるようにすることも可能です。カスタムタグは、標準の変数タグのように Red Hat Process Automation Manager

のランタイムに影響を与えることはせず、情報提供のみを目的としています。カスタム変数タグは、標準の Red Hat Process Automation Manager 変数タグに使用する場合と同じ **customTags** メタデータプロパティー形式で定義します。

第44章 ケースイベントリスナー

CaseEventListener は、ケースインスタンスで呼び出される、ケース関連のイベントやオペレーションの通知を開始するのに使用します。ケースイベントリスナーは、特定のユースケースに対して必要に応じてメソッドを上書きして、実装します。

Menu → Design → PROJECT_NAME → Settings → Deployments で、Business Central にあるデプロイメント記述子を使用して、リスナーを設定できます。

新規プロジェクトが作成されると、**kie-deployment-descriptor.xml** ファイルがデフォルト値で生成されます。

CaseEventListener メソッド

```
public interface CaseEventListener extends EventListener {

    default void beforeCaseStarted(CaseStartEvent event) {
    };

    default void afterCaseStarted(CaseStartEvent event) {
    };

    default void beforeCaseClosed(CaseCloseEvent event) {
    };

    default void afterCaseClosed(CaseCloseEvent event) {
    };

    default void beforeCaseCancelled(CaseCancelEvent event) {
    };

    default void afterCaseCancelled(CaseCancelEvent event) {
    };

    default void beforeCaseDestroyed(CaseDestroyEvent event) {
    };

    default void afterCaseDestroyed(CaseDestroyEvent event) {
    };

    default void beforeCaseReopen(CaseReopenEvent event) {
    };

    default void afterCaseReopen(CaseReopenEvent event) {
    };

    default void beforeCaseCommentAdded(CaseCommentEvent event) {
    };

    default void afterCaseCommentAdded(CaseCommentEvent event) {
    };

    default void beforeCaseCommentUpdated(CaseCommentEvent event) {
    };

    default void afterCaseCommentUpdated(CaseCommentEvent event) {
```

```
    };

    default void beforeCaseCommentRemoved(CaseCommentEvent event) {
    };

    default void afterCaseCommentRemoved(CaseCommentEvent event) {
    };

    default void beforeCaseRoleAssignmentAdded(CaseRoleAssignmentEvent event) {
    };

    default void afterCaseRoleAssignmentAdded(CaseRoleAssignmentEvent event) {
    };

    default void beforeCaseRoleAssignmentRemoved(CaseRoleAssignmentEvent event) {
    };

    default void afterCaseRoleAssignmentRemoved(CaseRoleAssignmentEvent event) {
    };

    default void beforeCaseDataAdded(CaseDataEvent event) {
    };

    default void afterCaseDataAdded(CaseDataEvent event) {
    };

    default void beforeCaseDataRemoved(CaseDataEvent event) {
    };

    default void afterCaseDataRemoved(CaseDataEvent event) {
    };

    default void beforeDynamicTaskAdded(CaseDynamicTaskEvent event) {
    };

    default void afterDynamicTaskAdded(CaseDynamicTaskEvent event) {
    };

    default void beforeDynamicProcessAdded(CaseDynamicSubprocessEvent event) {
    };

    default void afterDynamicProcessAdded(CaseDynamicSubprocessEvent event) {
    };
}
```

第45章 ケース管理のルール

ケースは、シーケンスフローには従わず、データ駆動型です。ケースを解決するために必要な手順は、ケースに関係する人によって提供されるデータに依存します。または、利用可能なデータに基づいてさらにアクションを発生させるようにシステムを設定できます。後者の場合、ビジネスルールを使用して、ケースを続行または解決するために必要な追加のアクションを決定できます。

データは、ケース内のどの時点でも、ケースファイルに挿入できます。デシジョンエンジンは常に、ケースファイルデータをモニターリングしているため、ケースファイルに含まれるデータに、ルールが反応します。ルールを使用してケースファイルデータの変更を監視および応答することで、ある程度自動的にケースが前に進むようにします。

45.1. ルールを使用したケースの前進

Business Central のケース管理に関する **IT_Orders** サンプルプロジェクトを参照します。

業者が入力した特定のハードウェア明細書に誤りがあるか無効だったとしましょう。ケースを続行させるためには、業者が、有効な注文を新たに行う必要があります。マネージャーが無効な明細書を却下して、業者に対して新しい要求を作成する代わりに、入力した明細書が無効であることをケースデータが示したらすぐに応答するビジネスルールを作成できます。このルールは、業者に新しいハードウェア明細書の要求を作成できます。

以下の手順は、このシナリオを実行するビジネスルールを作成して使用する方法を実演します。

前提条件

- **IT_Orders** サンプルプロジェクトが Business Central で開いているが、KIE Server にデプロイされていない。
- **ServiceRegistry** は **jbpm-services-api** モジュールの一部で、クラスパスで利用できるようにする必要があります。

注記

Business Central 外でプロジェクトをビルドする場合は、以下の依存関係をプロジェクトに追加する必要があります。

- **org.jbpm:jbpm-services-api**
- **org.jbpm:jbpm-case-mgmt-api**

手順

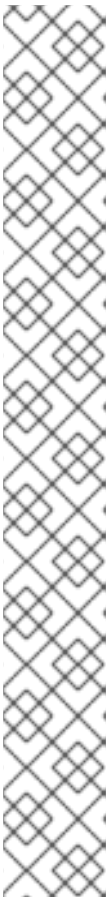
1. **validate-document.drl** という名前で、以下のビジネスルールファイルを作成します。

```
package defaultPackage;

import java.util.Map;
import java.util.HashMap;
import org.jbpm.casemgmt.api.CaseService;
import org.jbpm.casemgmt.api.model.instance.CaseFileInstance;
import org.jbpm.document.Document;
import org.jbpm.services.api.service.ServiceRegistry;

rule "Invalid document name - reupload"
```

```
when
   $caseData : CaseFileInstance()
   Document(name == "invalid.pdf") from $caseData.getData("hwSpec")

then

   System.out.println("Hardware specification is invalid");
   $caseData.remove("hwSpec");
   update($caseData);
   CaseService caseService = (CaseService)
ServiceRegistry.get().service(ServiceRegistry.CASE_SERVICE);
   caseService.triggerAdHocFragment($caseData.getCaseId(), "Prepare hardware spec",
null);
end
```

このビジネスルールは、**invalid.pdf** ファイルをケースファイルにアップロードしたタイミングを検出します。これは **invalid.pdf** ドキュメントを削除し、**Prepare hardware spec** ユーザータスクの新しいインスタンスを作成します。

2. **Deploy** をクリックして、**IT_Orders** プロジェクトをビルドし、KIE Server にデプロイします。



注記

Build & Install オプションを選択してプロジェクトをビルドし、KJAR ファイルを KIE Server にデプロイせずに設定済みの Maven リポジトリーに公開することもできます。開発環境では、**Deploy** をクリックすると、ビルドされた KJAR ファイルを KIE Server に、実行中のインスタンス (がある場合はそれ) を停止せずにデプロイできます。または **Redeploy** をクリックして、ビルドされた KJAR ファイルをデプロイしてすべてのインスタンスを置き換えることもできます。次回、ビルドされた KJAR ファイルをデプロイまたは再デプロイすると、以前のデプロイメントユニット (KIE コンテナー) が同じターゲット KIE Server で自動的に更新されます。実稼働環境では **Redeploy** オプションは無効になっており、**Deploy** をクリックして、ビルドされた KJAR ファイルを KIE Server 上の新規デプロイメントユニット (KIE コンテナー) にデプロイすることのみが可能です。

KIE Server の環境モードを設定するには、**org.kie.server.mode** システムプロパティーを **org.kie.server.mode=development** または **org.kie.server.mode=production** に設定します。Business Central の対応するプロジェクトでのデプロイメント動作を設定するには、プロジェクトの **Settings → General Settings → Version** に移動し、**Development Mode** オプションを選択します。デフォルトでは、KIE Server および Business Central のすべての新規プロジェクトは開発モードになっています。**Development Mode** をオンにしたプロジェクトをデプロイしたり、実稼働モードになっている KIE Server に手動で **SNAPSHOT** バージョンの接尾辞を追加したプロジェクトをデプロイしたりすることはできません。

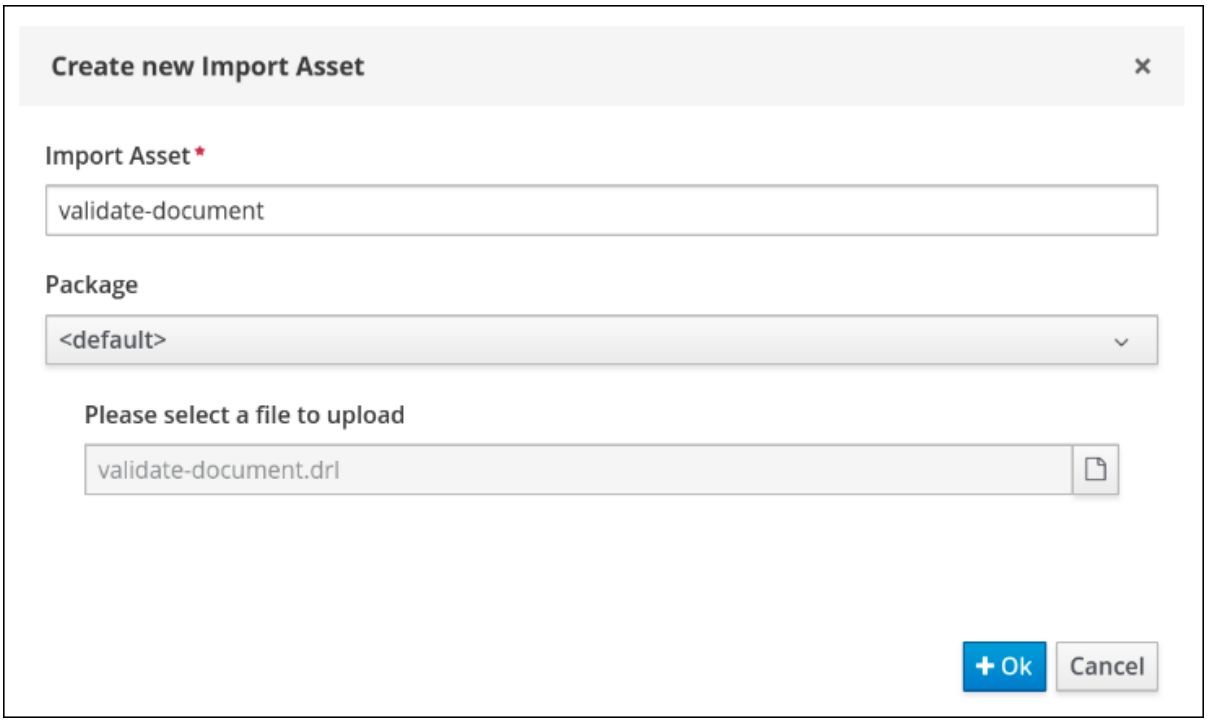
3. **invalid.pdf** ファイルを作成し、ローカルで保存します。

4. **valid-spec.pdf** ファイルを作成し、ローカルで保存します。

5. Business Central で、**Menu → Projects → IT_Orders** に移動して、**IT_Orders** プロジェクトを開きます。

6. ページの右上で **Import Asset** をクリックします。

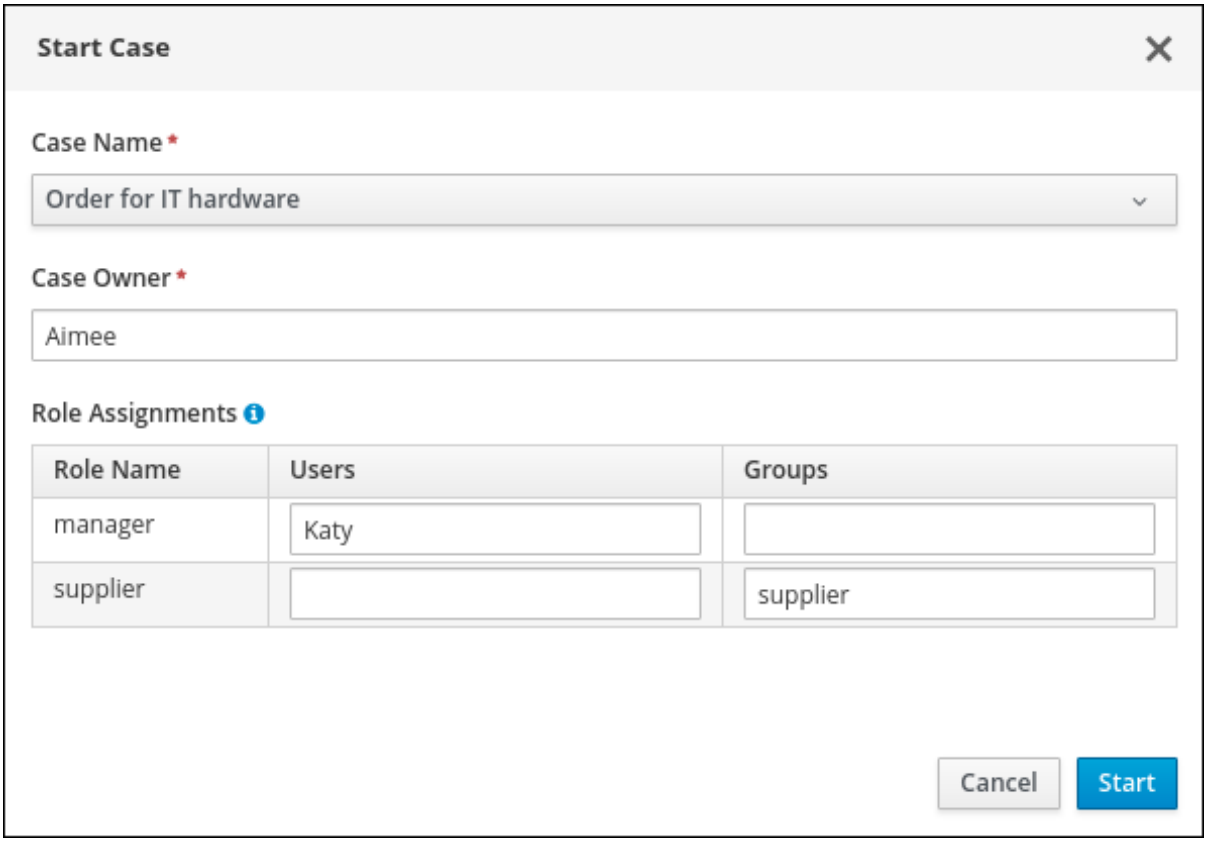
7. **validate-document.drl** ファイルを **default** パッケージ (**src/main/resources**) にアップロードし、**Ok** をクリックします。



validate-document.drl ルールがルールエディターに表示されます。**Save** をクリックするか、ルールエディターを閉じて終了します。

8. **Apps launcher** (インストールされている場合) をクリックするか、**http://localhost:8080/rhpam-case-mgmt-showcase/jbpm-cm.html** に移動して、Showcase アプリケーションを開きます。

9. **IT_Orders** の **Start Case** をクリックします。
この例では、Aimee がケースの **owner**、Katy が **manager**、および業者グループが **supplier** になります。

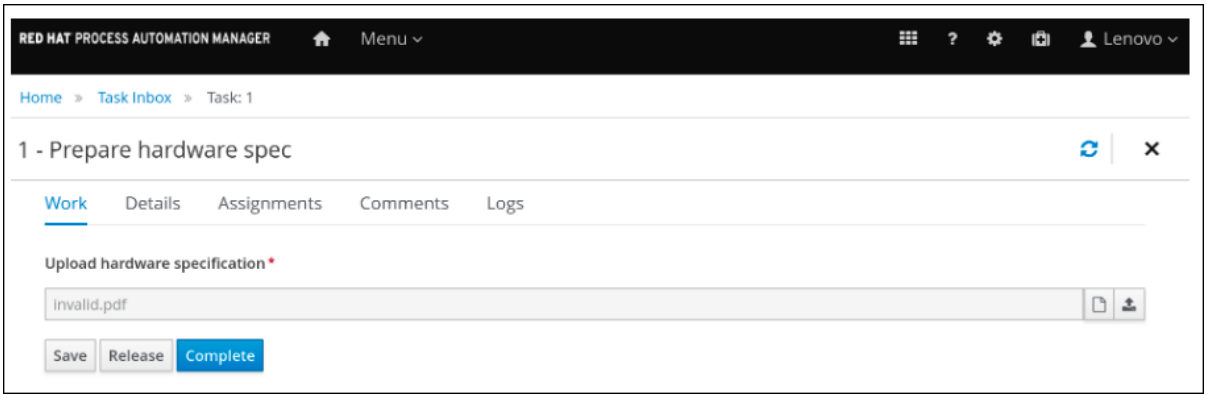


10. Business Central からログアウトし、**supplier** グループに属するユーザーでログインし直します。

11. **Menu** → **Track** → **Task Inbox** の順に移動します。

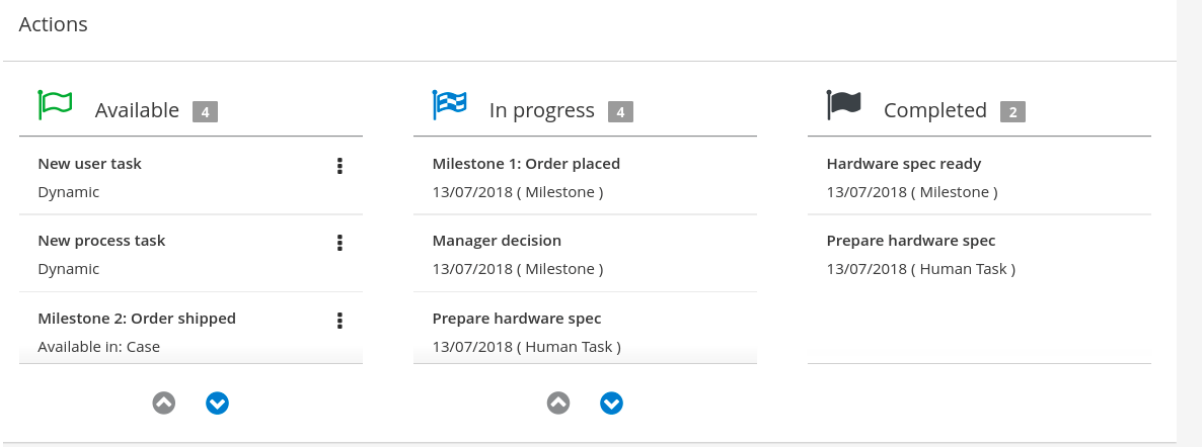
12. **Prepare hardware spec** タスクを開いて、**Claim** をクリックします。これにより、タスクがログインユーザーに割り当てられます。

13. **Start** をクリックし、 をクリックして **invalid.pdf** ハードウェア仕様ファイルの場所を特定します。 をクリックしてファイルをアップロードします。



14. **Complete** をクリックします。
Prepare hardware spec の Task Inbox の値が **Ready** になります。

15. Showcase の右上にある **Refresh** をクリックします。**Prepare hardware task** メッセージが **Completed** の列と、**In Progress** の列に表示されていることを確認します。



これは、最初の **Prepare hardware spec** タスクが明細書ファイル **invalid.pdf** を使用して完了されているためです。その結果、ビジネスルールによりタスクおよびファイルが破棄され、新しいユーザータスクが作成されます。

16. Business Central の **Task Inbox** で、前の手順を繰り返して、**invalid.pdf** の代わりに **valid-spec.pdf** ファイルをアップロードします。

第46章 ケース管理のセキュリティー

ケースは、ケー出力ルを使用して、ケース定義レベルで設定されます。これは、ケース処理に関与する一般的な参加者です。このようなロールはユーザータスクに割り当てられるか、連絡先参照として使用されます。ロールは、特定のユーザーまたはグループにはハードコードされず、ケース定義を、指定したケースインスタンスに関与する実際のアクターとは独立させます。ケースインスタンスがアクティブである限り、いつでもケー出力ル割り当てを修正できます。ただし、ロール割り当ては、以前のロール割り当てに基づいて作成されているタスクには影響を及ぼしません。

ケースインスタンスのセキュリティーはデフォルトで有効になります。ケース定義は、そのケースに属していないユーザーがケースデータにアクセスしないようにします。ユーザーにケー出力ル割り当て (ユーザーまたはグループメンバーとしての割り当て) がない場合は、ケースインスタンスにアクセスできません。

ケースのセキュリティーは、ケースインスタンスを開始する場合に、ケー出力ルを割り当てることが推奨される理由の 1 つです。これにより、ケースにアクセスできないユーザーにタスクが割り当てられないようにします。

46.1. ケース管理のセキュリティーの設定

以下のシステムプロパティーを **false** に設定すると、ケースインスタンスの認証を無効にできます。

org.jbpm.cases.auth.enabled

システムプロパティーは、ケースインスタンスに対するセキュリティーコンポーネントの 1 つでしかありません。さらに、**case-authorization.properties** ファイルを使用して実行サーバーレベルでケース操作を設定できます。これは、実行サーバーアプリケーションのクラスパスの root (**kie-server.war/WEB-INF/classes**) で利用できます。

可能なすべてのケース定義に簡易な設定ファイルを使用すると、ケース管理はドメイン固有のものとして見なされます。ケースセキュリティーの **AuthorizationManager** はプラグ可能で、特定のセキュリティー処理にカスタムコードを追加できます。

以下のケースインスタンス操作をケー出力ルに制限できます。

- **CANCEL_CASE**
- **DESTROY_CASE**
- **REOPEN_CASE**
- **ADD_TASK_TO_CASE**
- **ADD_PROCESS_TO_CASE**
- **ADD_DATA**
- **REMOVE_DATA**
- **MODIFY_ROLE_ASSIGNMENT**
- **MODIFY_COMMENT**

前提条件

- Red Hat Process Automation Manager の KIE Server が実行されていない。

手順

1. 任意のエディターで、**JBOSS_HOME/standalone/deployments/kie-server.war/WEB-INF/classes/case-authorization.properties** ファイルを開きます。
   デフォルトでは、ファイルには以下の操作制限が含まれます。

```
CLOSE_CASE=owner,admin
CANCEL_CASE=owner,admin
DESTROY_CASE=owner,admin
REOPEN_CASE=owner,admin
```

2. 必要に応じて、この操作に対するロールパーミッションを追加または削除します。

   a. ロールが操作を実行するためのパーミッションを削除するには、**case-authorization.properties** ファイルでその操作に対して認証されているロールの一覧から削除します。たとえば、**admin** ロールを **CLOSE_CASE** 操作から削除すると、すべてのケースで、そのケース所有者に対してケースを閉じるパーミッションに制限がかかります。

   b. ケース操作を実行するロールパーミッションを付与するには、**case-authorization.properties** ファイルでその操作に対して承認されているロールの一覧に追加します。たとえば、**manager** ロールがあれば **CLOSE_CASE** 操作を実行できるように許可する場合は、ロールの一覧に、コンマ区切りのロールを追加できます。
   CLOSE_CASE=owner,admin,manager

3. ファイルに挙げられているその他のケース操作にロール制限を追加するには、行から **#** を削除し、以下の形式でロール名を一覧表示します。
   OPERATION=role1,role2,roleN

   ファイル内で **#** で始まる操作の制限は無視され、ケースに関与するユーザーは誰でも実行できます。

4. ロールパーミッションの割り当てを終了したら、**case-authorization.properties** ファイルを保存して閉じます。

5. 実行サーバーを開始します。
   ケース認証設定は、実行サーバーのすべてのケースに適用します。

第47章 ケースの終了

ケースインスタンスは、実行するアクティビティーがなくなったかビジネスゴールに達成した場合に完了、または永続的に閉じることができます。通常は、すべての作業が完了してケースゴールを満たした場合に、ケースの所有者がケースを閉じます。ケースを閉じる際には、ケースインスタンスを閉じる理由についてコメントを追加することを検討してください。

必要に応じて、後から閉じたケースを同じケース ID を使用して再開することができます。ケースが再度開かれると、ケースが閉じられたときにアクティブだったステージが、ケースが再び開かれたときにアクティブになります。

KIE Server REST API 要求を使用してケースインスタンスをリモートで、または Showcase アプリケーションで直接閉じることができます。

47.1. KIE SERVER REST API を使用したケースの終了

REST API 要求を使用してケースインスタンスを閉じることができます。Red Hat Process Automation Manager には Swagger クライアントが含まれており、REST API 要求のエンドポイントやドキュメントが提供されます。もしくは、サンプルエンドポイントで、任意のクライアントや Curl を使用して API コールを作成できます。

前提条件

- Showcase を使用して、ケースインスタンスを開始している。
- **admin** ロールを持つユーザーとして、API 要求を認証できる。

手順

1. Web ブラウザーで Swagger REST API クライアントを開きます。
   http://localhost:8080/kie-server/docs
2. **Case Instances :: Case Management** の下で、以下のエンドポイントで **POST** リクエストを開きます。
   /server/containers/{id}/cases/instances/{caseId}
3. **Try it out** をクリックし、必要なパラメーターを入力します。

   表47.1 パラメーター

名前	説明
id	**itorders**
caseId	**IT-0000000001**

4. 必要に応じて、ケースファイルに含まれるコメントを含めます。コメントを残す場合は、**body** テキストフィールドに **文字列** で入力します。
5. **Execute** をクリックして、ケースを閉じます。
6. ケースが閉じたことを確定するには、Showcase アプリケーションを開いて、ケースリストのステータスを **Closed** に変更します。

47.2. SHOWCASE アプリケーションを使用したケースの終了

ケースインスタンスは、実行するアクティビティーがなくなり、ビジネス目標が達成されたら終了します。ケースが完了したら、ケースを閉じて、ケースが完了し、それ以上の作業が不要であることを示すことができます。ケースを終了したら、ケースの終了理由について具体的なコメントを追加することを検討してみてください。必要な場合には、同じケース ID を使用して、対象のケースをもう一度、開くことができます。

Showcase アプリケーションを使用していつでもケースを閉じることができます。Showcase から、簡単にケースの詳細を表示したり、閉じる前にコメントを残したりできます。

前提条件

- Showcase アプリケーションにログインしていること。また、終了するケースインスタンスの所有者か、管理者である。

手順

1. Showcase アプリケーションで、ケースインスタンスの一覧から閉じるケースインスタンスを見つけます。
2. 詳細を確認せずに先にケースを終了するには、**Close** をクリックします。
3. ケース詳細ページからケースを閉じるには、リストでケースをクリックして、開きます。
   ケースの概要ページで、ケースにコメントを追加して、ケース情報に基づいて正しいケースを閉じていることを確認します。
4. **Close** をクリックして、ケースを閉じます。
5. ページ右上で **Back to Case List**をクリックし、Showcase ケース一覧ビューに戻ります。
6. **Status** の横にあるドロップダウンリストをクリックし、**Canceled** をクリックすると、閉じたケースおよびキャンセルしたケースの一覧が表示されます。

第48章 ケースのキャンセルまたは破棄

ケースが必要なくなったり、ケース作業を実行する必要がなくなった場合は、ケースをキャンセルできます。キャンセルしたケースは、その後も同じケースインスタンス ID とケースファイルデータを使用して再開できます。ケースを再開できないようにケースを永続的に破棄したい場合もあります。

ケースのキャンセルまたは破棄は、API 要求からしかできません。Red Hat Process Automation Manager には Swagger クライアントが含まれており、REST API 要求のエンドポイントやドキュメントが提供されます。もしくは、サンプルエンドポイントで、任意のクライアントや Curl を使用して API コールを作成できます。

前提条件

- Showcase を使用して、ケースインスタンスを開始している。
- **admin** ロールを持つユーザーとして、API 要求を認証できる。

手順

1. Web ブラウザーで Swagger REST API クライアントを開きます。
   http://localhost:8080/kie-server/docs
2. **Case Instances :: Case Management** の下で、以下のエンドポイントで **DELETE** リクエストを開きます。
   /server/containers/{id}/cases/instances/{caseId}

   DELETE リクエストを使用してキャンセルできます。任意で、**destroy** パラメーターを使用してケースを破棄することもできます。
3. **Try it out** をクリックし、必要なパラメーターを入力します。

   表48.1 パラメーター

名前	説明
id	**itorders**
caseId	**IT-0000000001**
destroy	**true** (任意。永続的にケースを破壊します。このパラメーターはデフォルトで **false** です)。

4. **Execute** をクリックして、ケースをキャンセル (または破棄) します。
5. ケースがキャンセルしたことを確定するために、Showcase アプリケーションを開いて、ケースリストのステータスを **Canceled** にします。ケースが破棄されていると、ケースリストには表示されません。

48.1. データベースからケースログの削除

CaseLogCleanupCommand を使用して、データベースのスペースを占有しているキャンセル済みのケースなど、ケースを消去します。**CaseLogCleanupCommand** コマンドには、選択したケースまたは、全ケースを自動的に消去するロジックが含まれています。

CaseLogCleanupCommand コマンドでは、以下の設定オプションを使用できます。

表48.2 CaseLogCleanupCommand パラメーターテーブル

名前	説明	排他的
SkipProcessLog	コマンドの実行時に、プロセスやノードインスタンス、プロセス変数ログの消去を省略するかどうかを指定します。デフォルト値は **false** です。	いいえ、他のパラメーターと併用できる
SkipTaskLog	コマンドの実行時に、タスク監査、タスクイベント、タスク変数ログの消去を省略するかどうかを指定します。デフォルト値は **false** です。	いいえ、他のパラメーターと併用できる
SkipExecutorLog	コマンドの実行時に、Red Hat Process Automation Manager エグゼキューターのエントリー消去を省略するかどうかを指定します。デフォルト値は **false** です。	いいえ、他のパラメーターと併用できる
SingleRun	ジョブルーチンを 1 回だけ実行するかどうかを指定します。デフォルト値は **false** です。	いいえ、他のパラメーターと併用できる
NextRun	次のジョブ実行をスケジュールします。たとえば、12 時間ごとにジョブを実行するには、**12h** と指定します。**SingleRun** が **true** にされており、**SingleRun** と **NextRun** の両方が設定されていないと、このスケジュールは無視されます。両方が設定されている場合には、**NextRun** のスケジュールが優先されます。正確な日付を設定するには、ISO 形式を使用できます。デフォルト値: **24h**	いいえ、他のパラメーターと併用できる
OlderThan	指定の日付より古いログを削除します。日付の形式は、**YYYY-MM-DD** です。通常、このパラメーターは単一のジョブ実行に使用します。	はい。**OlderThanPeriod** パラメーターを使用する場合にはこのパラメーターは使用できません。

名前	説明	排他的
OlderThanPeriod	指定のタイマー式より古いログを削除します。たとえば、30 日が経過したログを削除するには、30d を設定します。	はい。**OlderThan** パラメーターを使用する場合は使用できません。
ForCaseDefId	削除するログのケース定義 ID を指定します。	いいえ、他のパラメーターと併用できる
ForDeployment	削除するログのデプロイメント ID を指定します。	いいえ、他のパラメーターと併用できる
EmfName	削除操作の実行に使用する永続ユニット名。デフォルト値: **org.jbpm.domain**	該当なし
DateFormat	時間関連のパラメーターの日付形式を指定します。デフォルト値: **yyyy-MM-dd**	いいえ、他のパラメーターと併用できる
ステータス	削除されたログのケースインスタンスのステータス	いいえ、他のパラメーターと併用できる

第49章 関連情報

- ケース管理の使用ガイド
- ケース管理への Showcase アプリケーションの使用

パート V. ケース管理への SHOWCASE アプリケーションの使用

ケース作業者またはプロセスの管理者は、Business Central でケース作業が行われている間、Showcase アプリケーションを使用してケース管理アプリケーションを管理および監視できます。

ケース管理はビジネスプロセス管理 (BPM) とは異なり、ケース時に処理する実際のデータに焦点をあて、目的を達成する一連の手順にはあまり重点を置きません。ケースデータは、ケース処理における情報の中で最も重要な要素ですが、ビジネスの状況や意思決定はケース作業担当者の管理下に置かれます。

本書を使用して Showcase アプリケーションをインストールし、Business Central の **IT_Orders** サンプルケース管理プロジェクトを使用してケースインスタンスを開始します。Business Central を使用して、IT 発注ケースを完了するのに必要なタスクを完了します。

前提条件

- Red Hat JBoss Enterprise Application Platform 7.4 がインストールされている。詳細は、**Red Hat JBoss EAP 7.4 インストールガイド** を参照してください。
- Red Hat JBoss EAP に Red Hat Process Automation Manager がインストールされ、KIE Server で設定されている。詳細は **Red Hat JBoss EAP 7.4 への Red Hat Process Automation Manager のインストールおよび設定** を参照してください。
- **KieLoginModule** を **standalone-full.xml** に設定している。これは、KIE Server への接続に必要です。KIE Server の設定方法は、**Red Hat Process Automation Manager インストールの計画** を参照してください。
- Red Hat Process Automation Manager が稼働し、**kie-server** および **user** の両ロールを持つユーザーで Business Central にログインしている。ロールの詳細は、**Red Hat Process Automation Manager インストールの計画** を参照してください。
- **IT_Orders** サンプルプロジェクトが Business Central に実装されており、KIE Server にデプロイされている。ケース管理の詳細は、**ケース管理の使用ガイド** を参照してください。

第50章 ケース管理

ケース管理は、Business Process Management (BPM) の拡張機能で、適用可能なビジネスプロセスを管理します。

BPM は、反復可能で共通のパターンを持つタスクの自動化に使用する管理プラクティスで、プロセスを完全化して最適化を図ることに焦点を当てます。ビジネスプロセスは通常、ビジネスの目標へのパスを明確に定義し、モデル化されています。これにより、通常は、量産原理に基づいた、多くの予測可能性が必要となります。ただし、実在する多くのアプリケーションでは、(可能なパス、脱線、例外など) 開始と終了を完全に説明することができません。特定のケースでプロセス指向のアプローチを使用すると、複雑なソリューションとなり、管理が困難になります。

ケース管理は、ルーティンで、予測可能なタスクを対象する BPM の効率指向アプローチとは対照的に、繰り返さず、予測できないプロセスに対する問題解決を提供します。ここでは、プロセスが前もって予測できない、一回限りの状況が管理されます。ケース定義は、通常、特定のマイルストーン、そして最終的には企業目標となるように、直接または間接的につながる結合が弱いプロセスの断片で設定されます。一方、プロセスは、ランタイム時に発生する変更に応えて動的に管理されます。

Red Hat Process Automation Manager のケース管理には、以下のコアプロセスエンジン機能が含まれます。

- ケースファイルのインスタンス
- ケースごとのランタイム戦略
- ケースコメント
- マイルストーン
- ステージ
- アドホックフラグメント
- 動的タスクおよびプロセス
- ケース識別子 (相関キー)
- ケースのライフサイクル (閉じる、再開、キャンセル、破棄)

ケース定義は常にアドホックのプロセス定義で、明示的な開始ノードは必要ありません。ケース定義は、ビジネスユースケースの主なエントリーポイントです。

プロセス定義は、ケースでサポートされる設定概念として導入され、ケース定義に指定された通り、または必要に応じて追加処理に動的に取り込むために呼び出すことができます。ケース定義は、以下の新しいオブジェクトを定義します。

- アクティビティー (必須)
- ケースファイル (必須)
- マイルストーン
- ロール
- ステージ

第51章 ケース管理の SHOWCASE アプリケーション

Showcase アプリケーションは Red Hat Process Automation Manager ディストリビューションに同梱され、アプリケーション環境でケース管理機能を実演します。Showcase は、ビジネスプロセス管理 (BPM) とケース管理の対話を示す概念実証として使用することを目的としています。このアプリケーションを使用して、ケースの開始、終了、監視、対話が行えます。

Showcase は、Business Central アプリケーションおよび KIE Server とともにインストールする必要があります。Showcase アプリケーションでは新しいケースインスタンスを開始するのに必要ですが、ケース作業は引き続き Business Central で行われます。

ケースインスタンスが作成され、作業を開始したら、**Case List** でケースをクリックし、ケースの **Overview** ページを開くことで、Showcase アプリケーションでケースを監視できます。

Showcase サポート

Showcase アプリケーションは Red Hat Process Automation Manager の根幹部分ではなく、ケース管理のデモ目的で提供されるものです。Showcase は、お客様が特定のニーズに合わせて機能するように選択および変更することを奨励するために提供されています。アプリケーションの内容そのものに、製品固有のサービスレベルアグリーメント (SLA) は適用されません。Showcase の更新時に考慮できるように、問題、機能拡張の要求、およびフィードバックがあれば是非お知らせください。

Red Hat サポートは、想定される用途に対し、商慣習上妥当と考えられる範囲でこのテンプレートの使用に関するガイダンスを提供します。ただし、テンプレートで提供される UI コードの例は除きます。

注記

製品サポート は、Red Hat Process Automation Manager ディストリビューションに限定されます。

第52章 SHOWCASE アプリケーションのインストールおよびログイン

Showcase アプリケーションは、アドオン Zip ファイルの Red Hat Process Automation Manager 7.12 ディストリビューションに含まれます。このアプリケーションの目的は、Red Hat Process Automation Manager のケース管理機能を実演し、Business Central に作成したケースを操作することです。Red Hat JBoss Enterprise Application Platform インスタンスまたは OpenShift に Showcase アプリケーションをインストールできます。以下の手順では、Red Hat JBoss EAP に Showcase アプリケーションをインストールする方法を説明します。

前提条件

- Business Central および KIE Server が Red Hat JBoss EAP インスタンスにインストールされている。
- **kie-server** ロールおよび **user** ロールを持つユーザーを作成している。**user** ロールは Showcase アプリケーションにログインできます。稼働中の KIE Server でユーザーがリモート操作を実行するには、**kie-server** ロールが必要です。
- Business Central を実行している。

手順

1. Red Hat カスタマーポータルの Software Downloads ページに移動し (ログインが必要)、ドロップダウンオプションから製品およびバージョンを選択します。
   - Product: Process Automation Manager
   - Version: 7.12
2. **Red Hat Process Automation Manager 7.12 Add Ons**(**rhpam-7.12.0-add-ons.zip**) をダウンロードします。
3. **rhpam-7.12.0-add-ons.zip** ファイルを展開します。**rhpam-7.12-case-mgmt-showcase-eap7-deployable.zip** ファイルは展開したディレクトリーにあります。
4. **rhpam-7.12-case-mgmt-showcase-eap7-deployable.zip** アーカイブを一時ディレクトリーに展開します。以下の例では、この名前を **TEMP_DIR** とします。
5. **_TEMP_DIR/rhpam-7.12-case-mgmt-showcase-eap7-deployable/jboss-eap-7.4** ディレクトリーのコンテンツを **EAP_HOME** にコピーします。
   ファイルの上書き、またはディレクトリーのマージを確認したら、**はい** を選択します。

警告

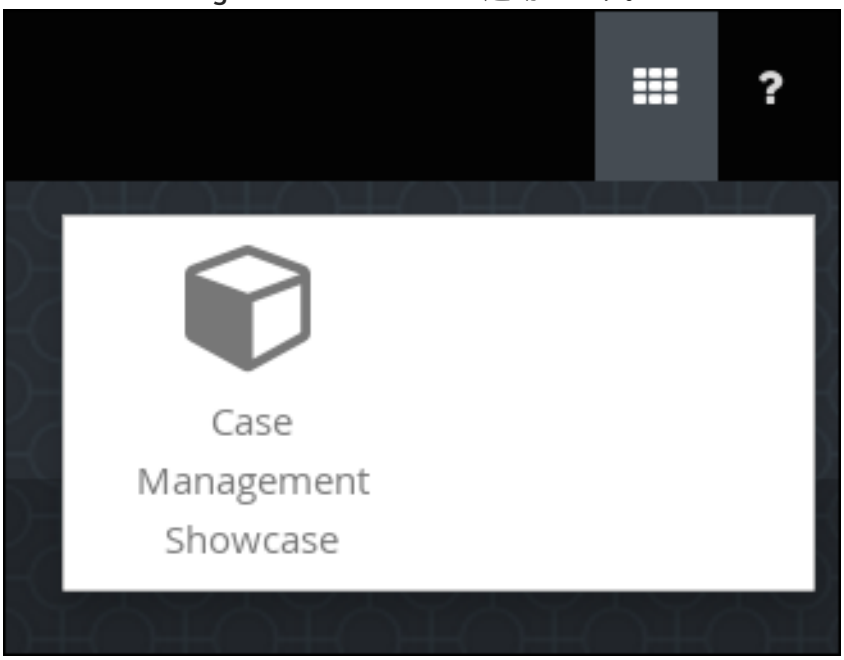
コピーする Red Hat Process Automation Manager デプロイメントの名前が、Red Hat JBoss EAP インスタンスの既存デプロイメントと競合しないことを確認します。

6. お使いのデプロイメントの **jboss-eap-7.4/standalone/configuration/standalone-full.xml** ファイルに移動して、以下のシステムプロパティーを追加します。
   <property name="org.jbpm.casemgmt.showcase.url" value="/rhpam-case-mgmt-showcase"/>

7. ターミナルアプリケーションで **EAP_HOME/bin** に移動して、スタンドアロンの設定を実行し、Business Central を起動します。
   ./standalone.sh -c standalone-full.xml

8. Web ブラウザーで **localhost:8080/business-central** を開きます。
   ドメイン名から実行するように Red Hat Process Automation Manager を設定した場合は、以下のように **localhost** をドメイン名に置き換えます。

   http://www.example.com:8080/business-central

9. Business Central の右上で、**Apps launcher** ボタンをクリックして、新しいブラウザーウィンドウで **Case Management Showcase** を起動します。
   

10. Business Central ユーザー認証情報を使用して Showcase アプリケーションにログインします。

第53章 CASE ROLES

ケー出力ルは、ユーザーがケース処理に参加する追加の抽象層を提供します。ロール、ユーザー、およびグループは、ケース管理の別の目的に使用されます。

ロール

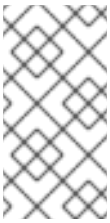
ロールは、ケースインスタンスの認証や、ユーザーアクティビティーの割り当てを可能にします。ユーザー、または1つ以上のグループを所有者ロールに割り当てることができます。所有者は、ケースを所有するユーザーになります。ケースの定義では、ロールはユーザーまたはグループ1つだけに制限されません。特定のユーザーまたはグループにタスクを割り当てる代わりに、ロールを使用してタスクの割り当てを指定することで、ケースを動的に保ちます。

Groups

グループとは、特定のタスクを実行できるユーザー、または指定の責任が割り当てられたユーザーの集合です。グループには何人でも割り当てることができ、ロールにはどのグループでも割り当てることができます。グループのメンバーをいつでも追加または変更できます。特定のタスクにグループをハードコーディングしないでください。

ユーザー

ユーザーとは、ロールに割り当てたり、グループに追加したりして、特定のタスクを割り当てることができる個人を指します。



注記

プロセスエンジンまたは KIE Server で **unknown** という名前のユーザーは作成しないでください。**unknown** ユーザーアカウントは、superuser のアクセス権限があるシステム名用に予約されています。**unknown** ユーザーアカウントでは、ログインしているユーザーがない場合に、SLA 違反リスナーに関連するタスクを実行します。

以下の例では、以下の情報で、前述のケース管理の概念をホテル予約にどのように適用するかを説明します。

- **ロール**: **Guest**
- **グループ**: **Receptionist**、**Maid**
- **ユーザー**: **Marilyn**

Guest のロールを割り当てると、関連ケースの特定の作業に影響があり、ケースインスタンスごとに固有です。すべてのケースインスタンスには、独自のロール割り当てがあります。ロールに割り当てることができるユーザーまたはグループの数はケースの **Cardinality** で制限されています。これは、プロセス設計者やケース定義でのロール作成時に設定されます。たとえば、ホテル予約ケースではゲストロールが1つ、**IT_Orders** サンプルプロジェクトでは IT ハードウェア業者ロールが 2 つです)。

ロールが定義されている場合は、ロールがケース定義の一部としてユーザー1人またはグループ1つにハードコードされておらず、ケースインスタンスごとに違うものを指定できるようにする必要があります。ケースのロール割り当てが重要なのは、このような理由からです。

ロールは、ケースの開始時や、ケースがアクテイブになった時点で割り当てまたは割り当ての解除ができます。ロールは任意ですが、ケース定義でロールを使用して、整理されたワークフローを維持します。

重要

タスク割り当てに実際のユーザーまたはグループ名を使用する代わりに、ロールを使用します。これにより、必要に応じて、ケースと、ユーザーまたはグループを割り当てるタイミングを遅らせることができます。

ロールはユーザーまたはグループに割り当てられ、ケースインスタンスの起動時にタスクを実行する権限があります。

第54章 動的タスクおよびプロセスの開始

ランタイム時に、ケースに動的タスクおよびプロセスを追加できます。動的アクションは、変化する状況に対応する方法で、ケース時に想定外の変更が発生した場合は、新しいタスクまたはプロセスをケースに組み込む必要があります。

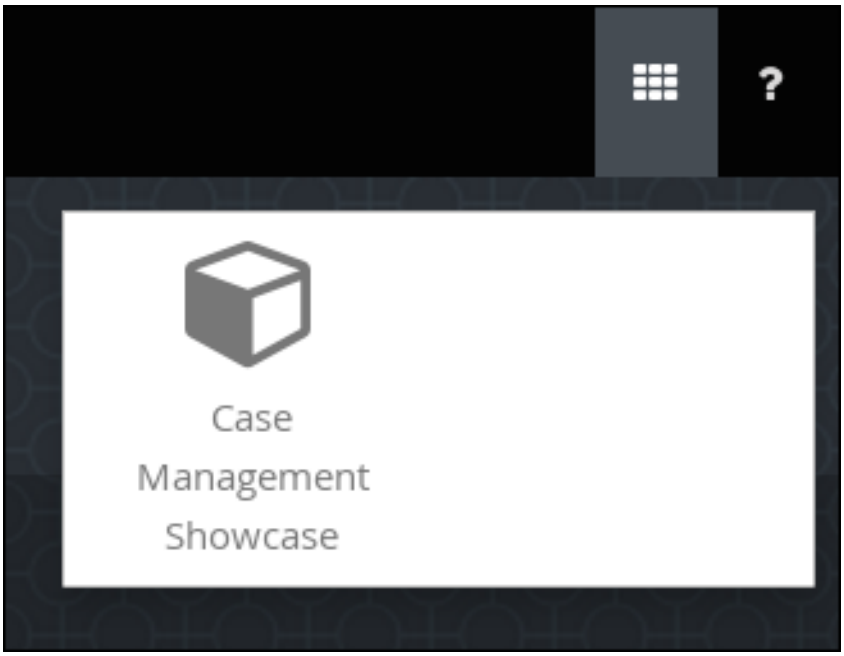
ケースアプリケーションを使用して、ランタイム時に動的タスクを追加します。デモの目的で、Business Central ディストリビューションには、IT 発注アプリケーションに対して新しい動的タスクまたはプロセスを開始できる Showcase アプリケーションが同梱されています。

前提条件

- KIE Server がデプロイされて Business Central に接続されている。
- IT Orders プロジェクトが KIE Server にデプロイされている。
- Business Central と、Showcase アプリケーションの **.war** ファイルがデプロイされている。

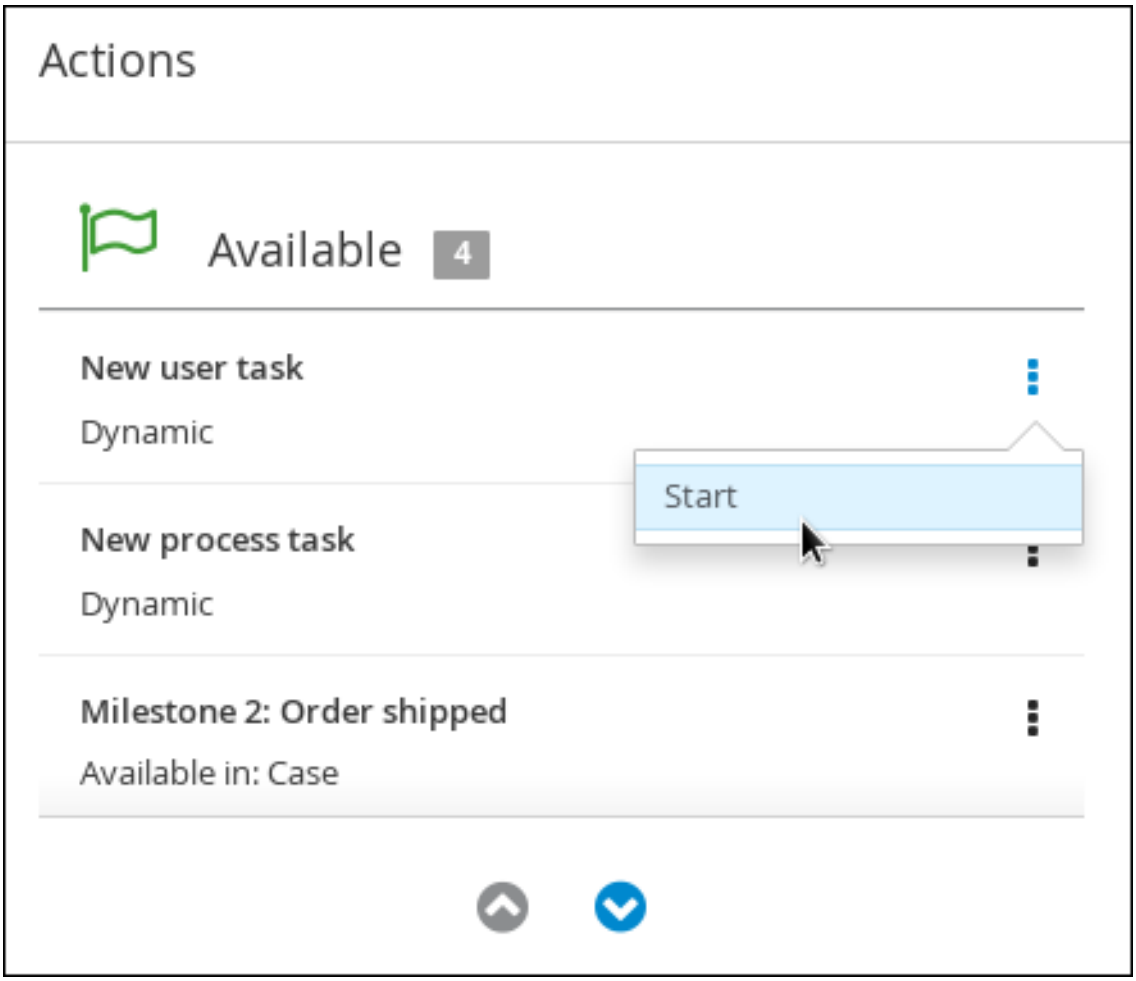
手順

1. Web ブラウザーの KIE Server に **IT_Orders_New** プロジェクトをデプロイして起動し、Showcase ログインページ **http://localhost:8080/rhpam-case-mgmt-showcase/** に移動します。
   Apps launcher ボタンを表示するように Business Central を設定している場合は、新しいブラウザーウィンドウで Showcase ログインページを開きます。

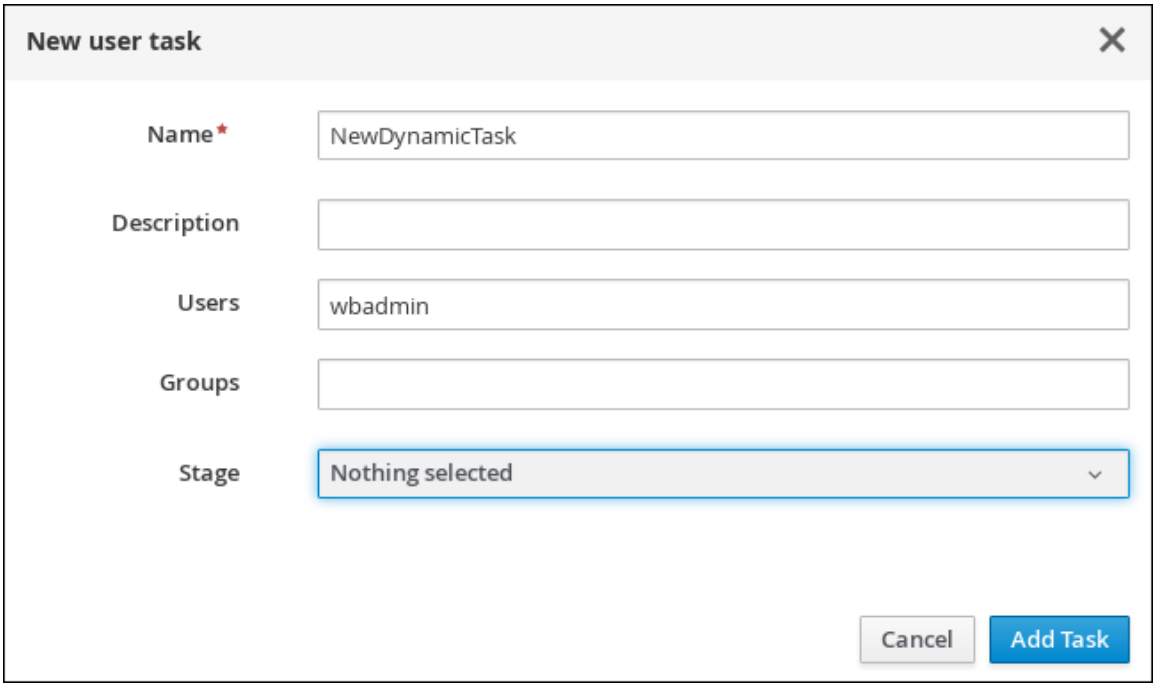


2. Business Central ログインの認証情報を使用して、Showcase アプリケーションにログインします。
3. 一覧からアクティブなケースインスタンスを選択して開きます。
4. **Overview → Actions → Available** で、**New user task** または **New process task** の横にあるボタンをクリックして、新しいタスクまたはプロセスタスクを追加します。

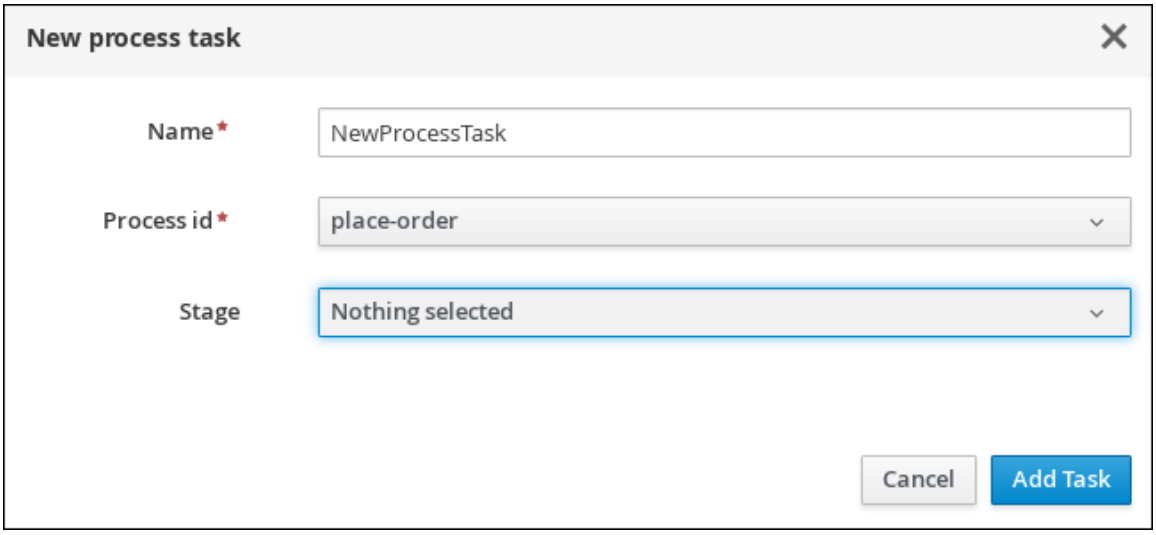
図54.1 Showcase 動的アクション



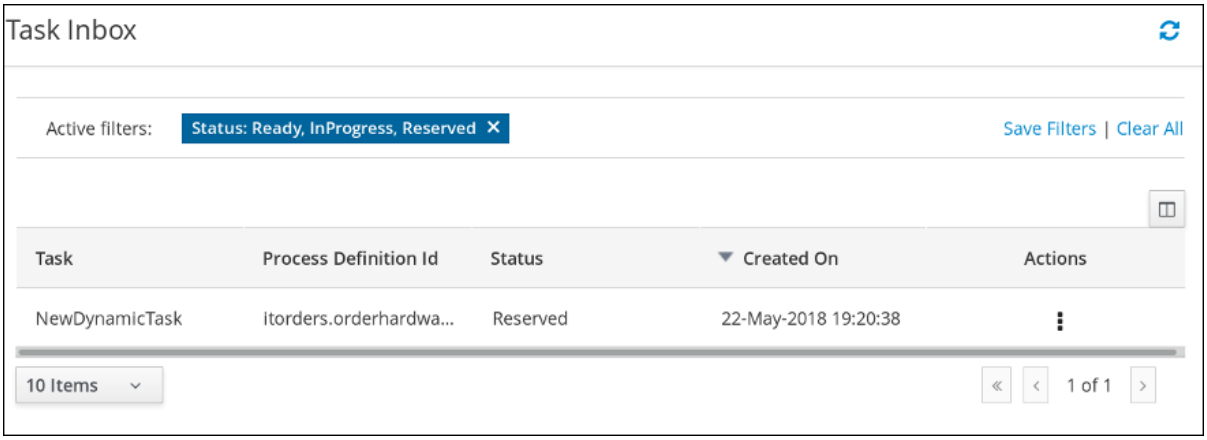
- 動的ユーザータスクを作成するには、**New user task** を開始して、必要な情報を入力します。



- 動的プロセスタスクを作成するには、新しいプロセスタスクを開始して必要な情報を入力します。



5. Business Central で動的ユーザータスクを表示するには、**Menu → Track → Task Inbox** の順にクリックします。Showcase アプリケーションを使用して動的に追加したユーザータスクが、タスク作成時にタスクに割り当てられるユーザーの **Task Inbox** に表示されます。



   a. **Task Inbox** で動的タスクをクリックしてタスクを開きます。このページで多くのアクションタブが利用できます。

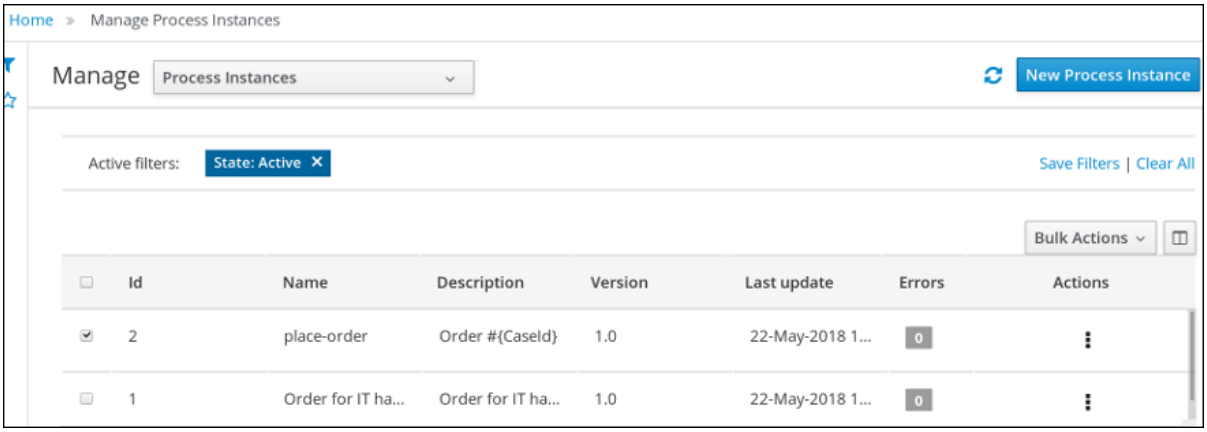
   b. タスクタブで利用可能なアクションを使用すると、タスクでの作業を開始できます。

   c. Showcase アプリケーションで、右上の再読み込みボタンをクリックします。進行中のケースタスクおよびプロセスは **Overview → Actions → In progress** に表示されます。

   d. タスクでの作業を完了したら、**Work** タブで **Complete** ボタンをクリックします。

   e. Showcase アプリケーションで、右上の再読み込みボタンをクリックします。**Overview → Actions → Completed** に完了したタスクが表示されます。

6. Business Central で動的プロセスタスクを表示する場合は、**Menu → Manage → Process Instances** の順にクリックします。



a. 利用可能なプロセスインスタンスの一覧で動的プロセスインスタンスをクリックして、プロセスインスタンスの情報を表示します。

b. Showcase アプリケーションで、右上の再読み込みボタンをクリックします。進行中のケースタスクおよびプロセスは **Overview → Actions → In progress** に表示されます。

第55章 SHOWCASE アプリケーションで IT 発注ケースの開始

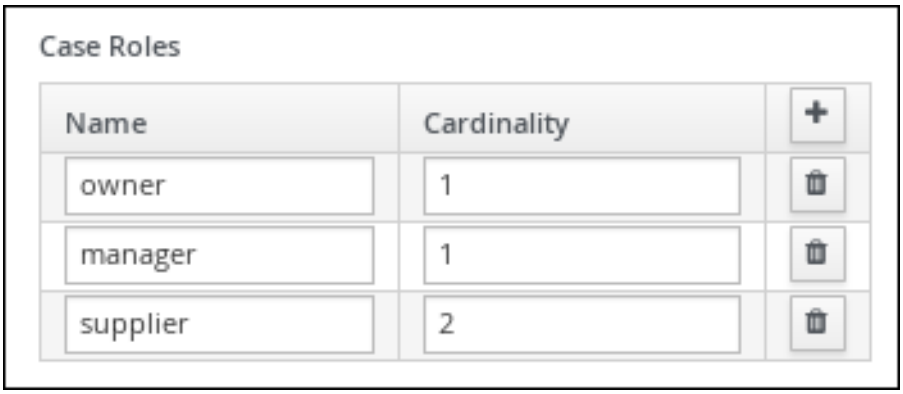
Showcase アプリケーションで、IT 発注サンプルケース管理プロジェクトの新しいケースインスタンスを開始します。

IT 発注サンプルケース管理プロジェクトには以下のロールが含まれます。

- **owner**: ハードウェア注文リクエストを行う従業員。このロールを持つユーザーは 1 人だけになる可能性もあります。
- **manager**: 従業員のマネージャー。要求されたハードウェアを承認または拒否する人。IT 発注プロジェクトにはマネージャーが 1 人だけ存在します。
- **supplier**: システム内の IT ハードウェアの利用可能なサプライヤー。通常は複数の業者が存在します。

このロールは、ケース定義レベルで設定されます。

図55.1 ITOrders ケー出力ル



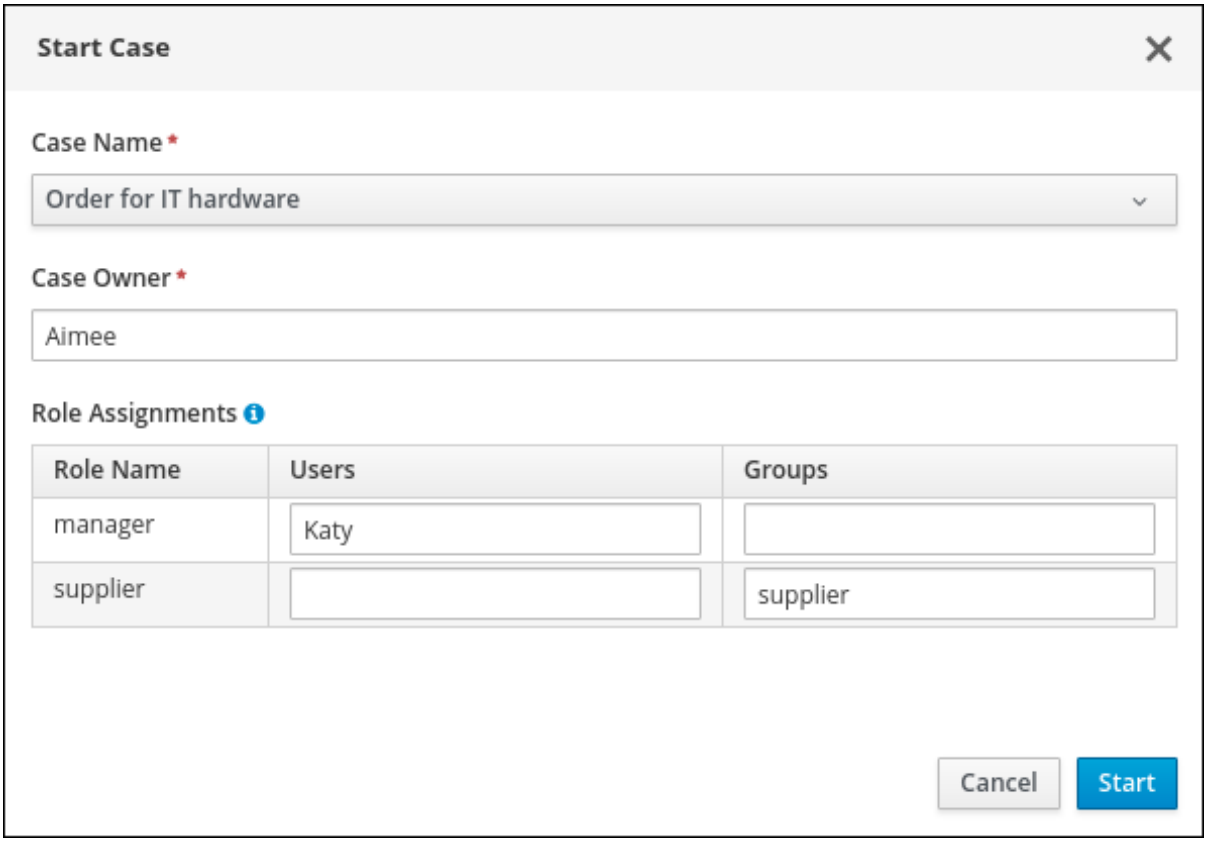
新しいケースファイルインスタンスの開始時に、このロールにユーザーまたはグループを割り当てます。

前提条件

- IT 発注サンプルプロジェクトを Business Central にインポートおよびデプロイしている。
- 52章*Showcase アプリケーションのインストールおよびログイン* の手順に従って、Showcase アプリケーションをインストールしてログインしている。

手順

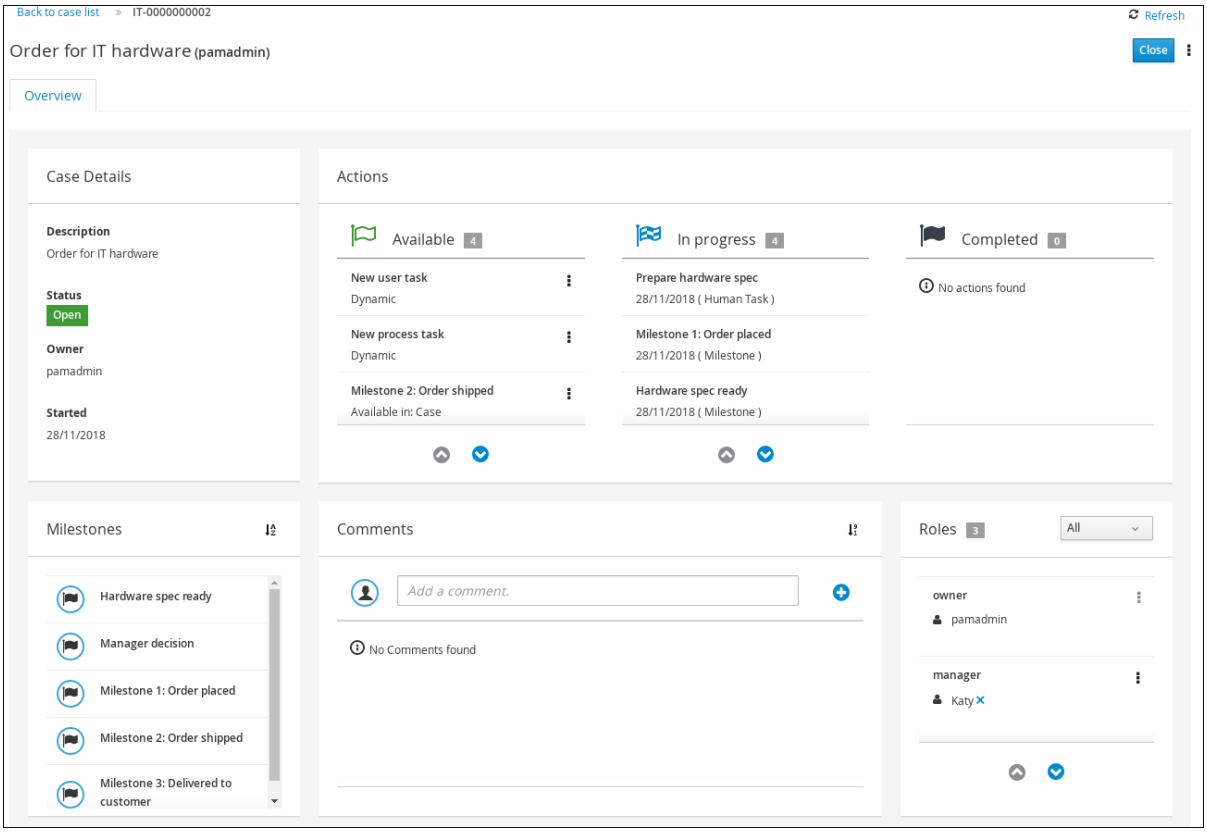
1. Showcase アプリケーションで、**Start Case** ボタンをクリックして新しいケースインスタンスを開始します。
2. 一覧から **Order for IT hardware** ケース名を選択し、以下のようにロール情報を完成させます。



この例では、Aimee がケースの **owner**、Katy が **manager**、および業者グループが **supplier** になります。

3. **Start** をクリックして、ケースインスタンスを開始します。

4. **Case List** からケースを選択します。**Overview** ページが開きます。
Overview ページから、ケースを進捗を監視、コメントを追加、新しい動的タスクおよびプロセスを開始、そしてケースを完了して閉じることができます。



注記

ケースは、Showcase アプリケーションで開始して閉じることができますが、このアプリケーションを使用して再開することはできません。ケースの再開には JMS または REST API コールを使用する必要があります。

第56章 SHOWCASE と BUSINESS CENTRAL を使用した IT_ORDERSケースの完了

Showcase アプリケーションでケースインスタンスを開始すると、ケース定義で **AdHoc Autostart** に設定したタスクが自動的に割り当てられ、各タスクに対するロール割り当てを持つユーザーが利用できるようになります。ケース作業者は、Business Central のタスクに取り組み、完了させ、ケースを前に進ませます。

IT_Orders ケースプロジェクトでは、以下のケース定義ノードが **AdHoc Autostart** プロパティーで設定されます。

- **Prepare hardware spec**
- **Hardware spec ready**
- **Manager decision**
- **Milestone 1: Order placed**

この中で、唯一のユーザータスクは **supplier** グループに割り当てられている **Prepare hardware spec** です。これは、IT Orders で完了する最初の人的タスクです。このタスクが完了すると、**manager** ロールが割り当てられたユーザーに **Manager approval** タスクが利用可能になり、残りのケース作業が終了すると、タスクを完了するために、**Customer satisfaction survey** タスクがケース所有者に割り当てられます。

前提条件

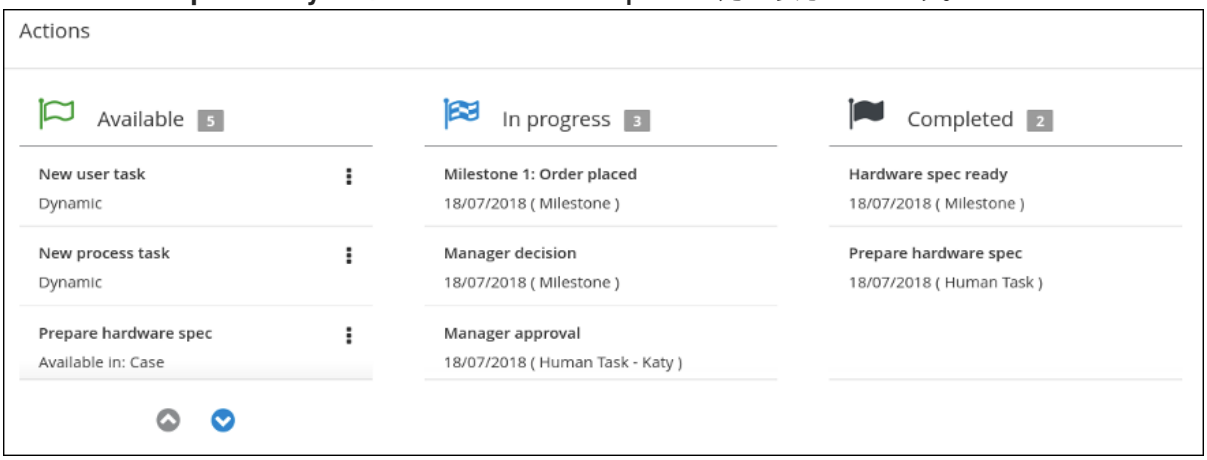
- **wbadmin** ユーザーとして、Showcase アプリケーションで **IT_Orders** ケースを開始している。

手順

1. Business Central からログアウトし、**supplier** グループに属するユーザーでログインし直します。
2. **Menu → Track → Task Inbox** の順に移動します。
3. **Prepare hardware spec** タスクを開いて、**Claim** をクリックします。これにより、タスクがログインユーザーに割り当てられます。
4. **Start** をクリックし、 をクリックし、ハードウェア仕様ファイルを見つけます。 をクリックしてファイルをアップロードします。

5. **Complete** をクリックします。

6. Showcase の右上にある **Refresh** をクリックします。**Prepare hardware task** ユーザータスクと **Hardware spec ready** マイルストーンが **Completed** 列に表示されます。



7. Business Central で **Menu → Track → Task Inbox** に移動します。**wbadmin** の **Manager approval** タスクを開きます。

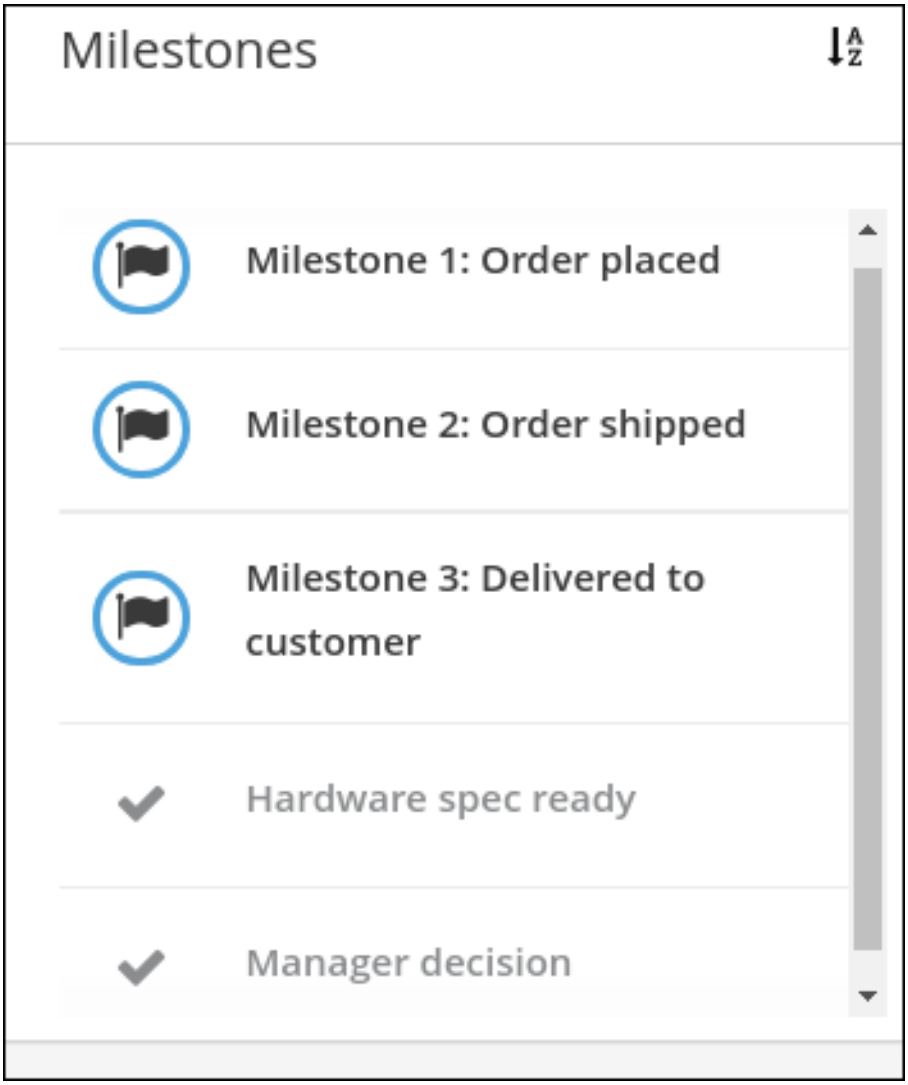
   a. **Claim** をクリックして、**Start** をクリックします。

   b. **valid-spec.pdf** ファイルを含むタスクの **approve** ボックスを確認して、**Complete** をクリックします。

8. **Menu → Manage → Process Instances** に移動し、**Order for IT hardware** プロセスインスタンスを開きます。

   a. **Diagram** タブを開きます。**Place order** タスクが完了しています。

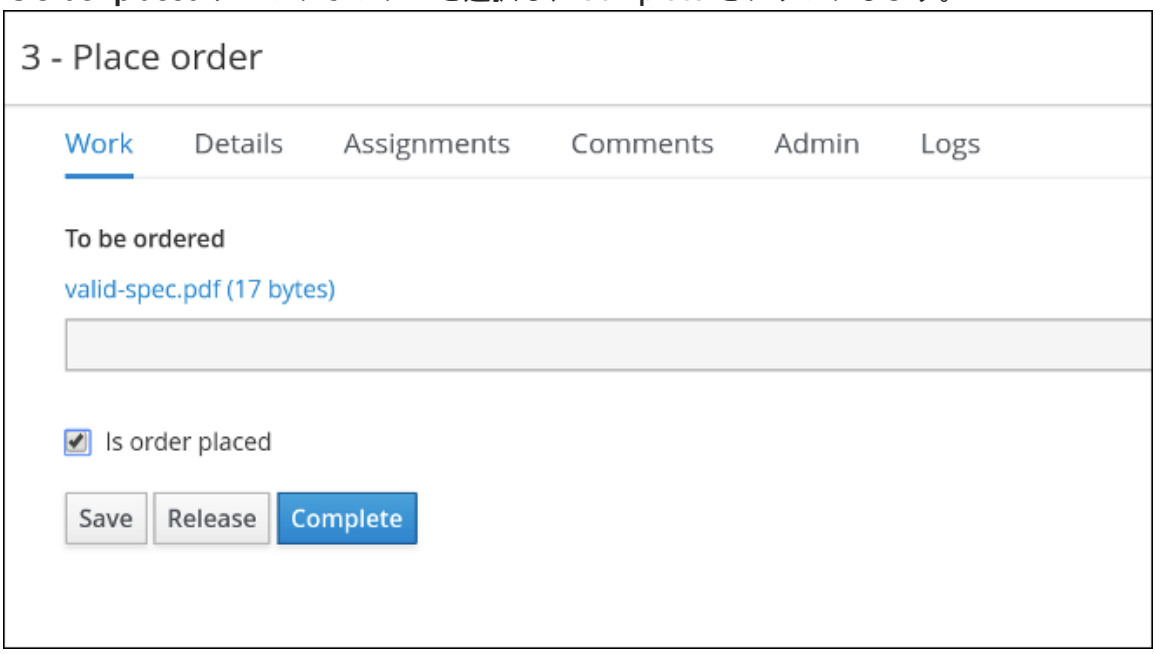
   b. Showcase ページを再読み込みし、**Manager approval** タスクと **Manager decision** マイルストーンが **Completed** 列にあることを確認します。Showcase 概要ページの左下にある **Milestones** ペインにも、完了または保留中のマイルストーンが表示されます。



9. Business Central で **Menu** → **Manage** → **Tasks** に移動します。**Place order** タスクをクリックして開きます。

   a. **Claim** をクリックして、**Start** をクリックします。

   b. **Is order placed** チェックボックスを選択し、**Complete** をクリックします。



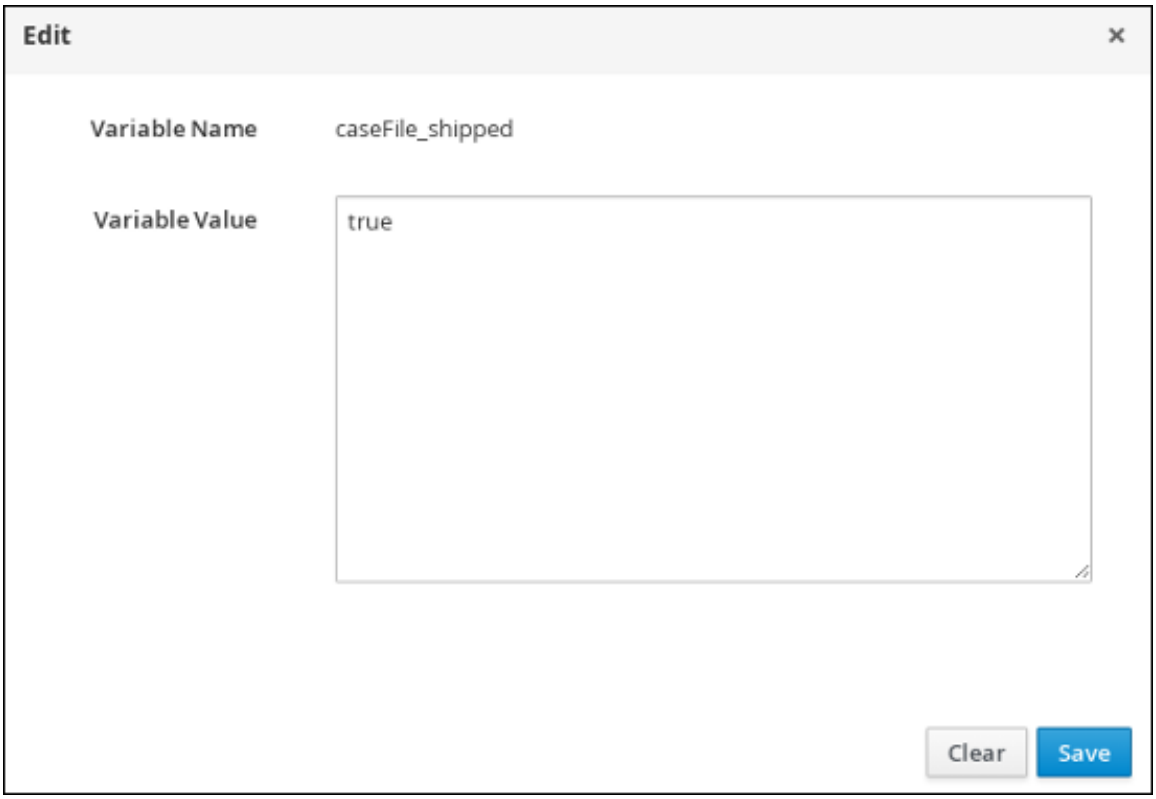
プロセスインスタンスの図に、**Milestones 2: Order shipped** ケースの進捗が表示されるようになりました。

c. Showcase ページを再読み込みして、ケースの進捗を表示します。

10. **Menu → Manage → Process Instances** に移動し、**Order for IT hardware** を開きます。

a. **Process Variables** タブを開きます。**caseFile_shipped** 変数を見つけ、**Edit** をクリックします。

b. **Edit** ウィンドウに **true** を入力し、**Save** をクリックします。



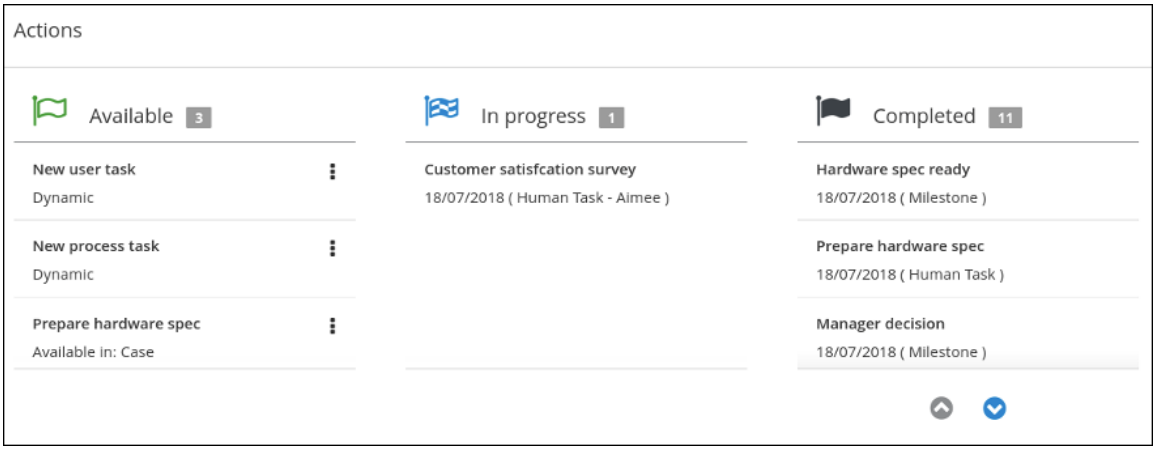
c. Showcase ページを再読み込みします。**Milestone 2: Order shipped** マイルストーンが **Completed** になっていることを確認します。
最後のマイルストーン **Milestone 3: Delivered to customer** は In progress になっています。

11. **Menu → Manage → Process Instances** に移動し、**Order for IT hardware** を開きます。

a. **Process Variables** タブを開きます。**caseFile_delivered** 変数を見つけて、**Edit** をクリックします。

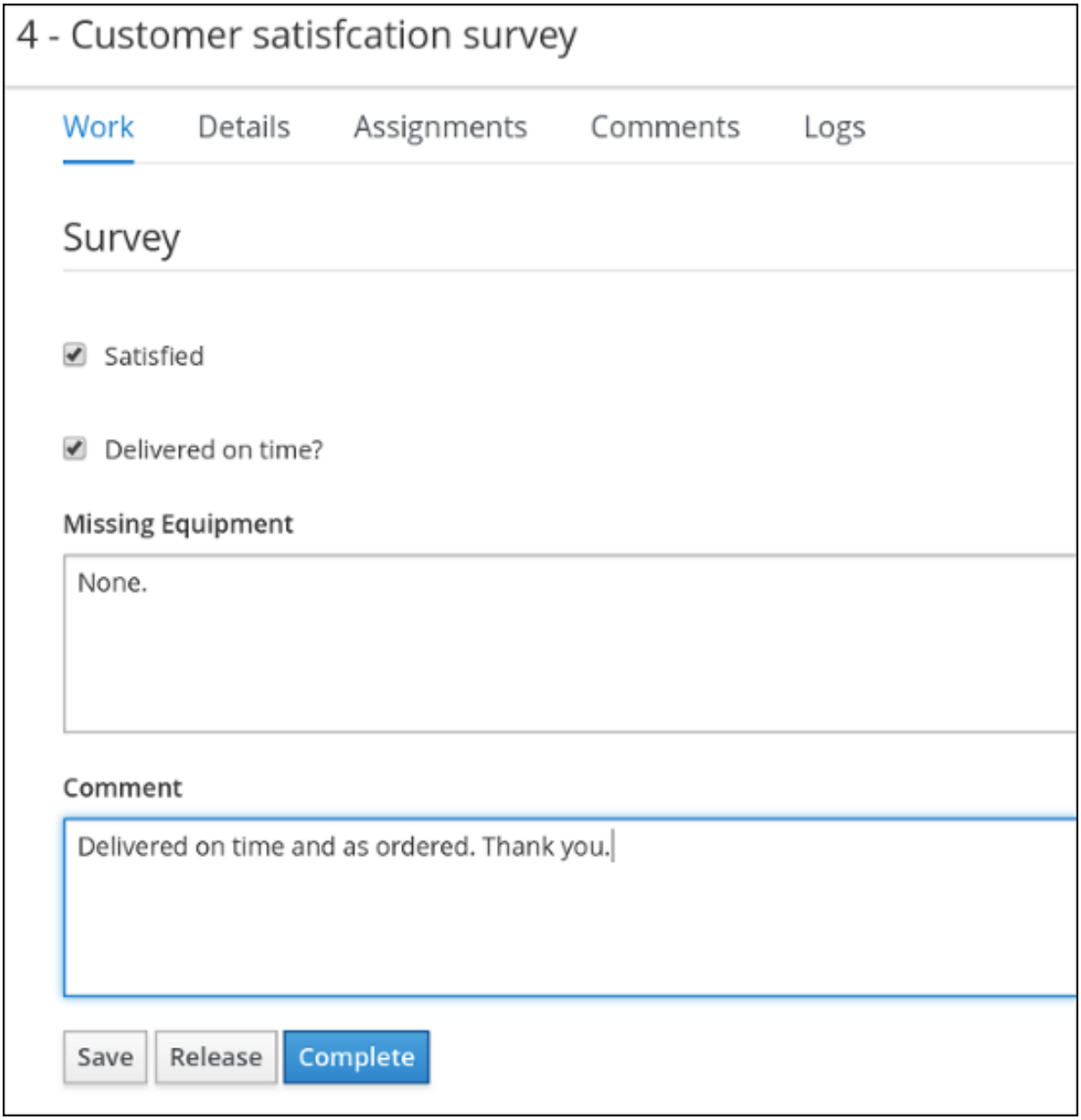
b. **Edit** ウィンドウに **true** を入力し、**Save** をクリックします。

c. Showcase ページを再読み込みします。**Milestone 3: Delivered to customer** マイルストーンが **Completed** になっていることを確認します。左下の **Milestones** ペインにあるすべてのマイルストーンは完了となります。
IT 発注ケースの最後のタスク **Customer satisfaction survey** が In progress の下に表示されます。



12. Business Central で **Menu → Track → Task Inbox** に移動します。**Customer satisfaction survey** タスクをクリックして開きます。
このタスクは、すでに **wbadmin** 向けに確保されています。

13. **Start** をクリックして、アンケートに答えます。



14. **Complete** をクリックします。

15. **Menu → Manage → Process Instances** に移動し、**Order for IT hardware** プロセスインスタンスを開きます。

    a. **Diagram** タブを開きます。これにより、必要なすべてのケースプロセスノードが完了し、このケースインスタンスで必要なアクションが残っていないことが確認できます。

    b. Showcase ページを再読み込みして、**In progress** にアクションが残っていないことを確認します。

16. Showcase で、**Comments** の下のフィールドにコメントを入力します。 をクリックして、ケースファイルにコメントを追加します。

17. Showcase ページの右上の **Close** をクリックして、ケースを完了して閉じます。

第57章 関連情報

- ケース管理の設計およびビルド
- ケース管理の使用ガイド

パート VI. カスタムタスクとワークアイテムハンドラー

ビジネスルール開発者は、カスタムタスクやワークアイテムハンドラーを作成して、プロセスフロー内でカスタムコードを実行し、Red Hat Process Automation Manager で使用できるように操作を拡張することができます。カスタムタスクを使用して、Red Hat Process Automation Manager に直接含まれていない操作を開発して、プロセスダイアグラムに追加できます。カスタムタスクとワークアイテムハンドラーは、Business Central、スタンドアロンエディター、および Visual Studio Code (VSCode) をサポートします。本章では、Business Central を使用したカスタムタスクとワークアイテムハンドラーの作成、管理、デプロイの方法を説明します。

Business Central では、プロセスダイアグラムの各タスクには Java クラス **WorkItemDefinition** と、関連付けられた Java クラス **WorkItemHandler** があります。**WorkItemDefinition** は、**.wid** ファイルの MVEL リストの一部として MVFLEX Expression Language (MVEL) マップを使用して定義できます。**.wid** ファイルはプロジェクトルート下の **グローバル** ディレクトリーか、BPMN ファイルのディレクトリーに配置します。ワークアイテムハンドラーには、Business Central に登録された Java コードが含まれており、**org.kie.api.runtime.process.WorkItemHandler** を実装します。

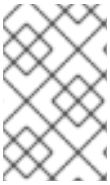
タスクがトリガーされると、ワークアイテムハンドラーの Java コードが実行されます。ワークアイテムハンドラーをカスタマイズして登録し、カスタムタスクで独自の Java コードを実行できます。

前提条件

- Business Central がデプロイされ、Web またはアプリケーションサーバーで実行されている。
- Business Central にログインしている。
- Maven がインストールされている。
- ホストからインターネットにアクセスできる。ビルドプロセスは、インターネットを使用して、外部のリポジトリーから Maven パッケージをダウンロードします。
- お使いのシステムから Red Hat の Maven リポジトリーにローカルまたはオンラインでアクセスできる。

第58章 BUSINESS CENTRAL でのカスタムタスクの管理

カスタムタスク (作業アイテム) は、カスタムロジックを実行できるタスクです。複数のビジネスプロセスまたは Business Central の全プロジェクトでタスクをカスタマイズして再利用できます。また、名前、アイコン、サブカテゴリー、入出力パラメーター、ドキュメントなど、デザイナーパレットにカスタム要素を追加することもできます。Red Hat Process Automation Manager は、Business Central のカスタムタスクリポジトリー内でカスタムタスクセットを提供します。デフォルトのカスタムタスクを有効化または無効化して、カスタムのタスクを Business Central にアップロードし、適切なプロセスにこのタスクを実装できます。



注記

Red Hat Process Automation Manager には、サポートされるカスタムタスクの限定セットが含まれています。Red Hat Process Automation Manager に含まれていないカスタムタスクはサポートされません。

手順

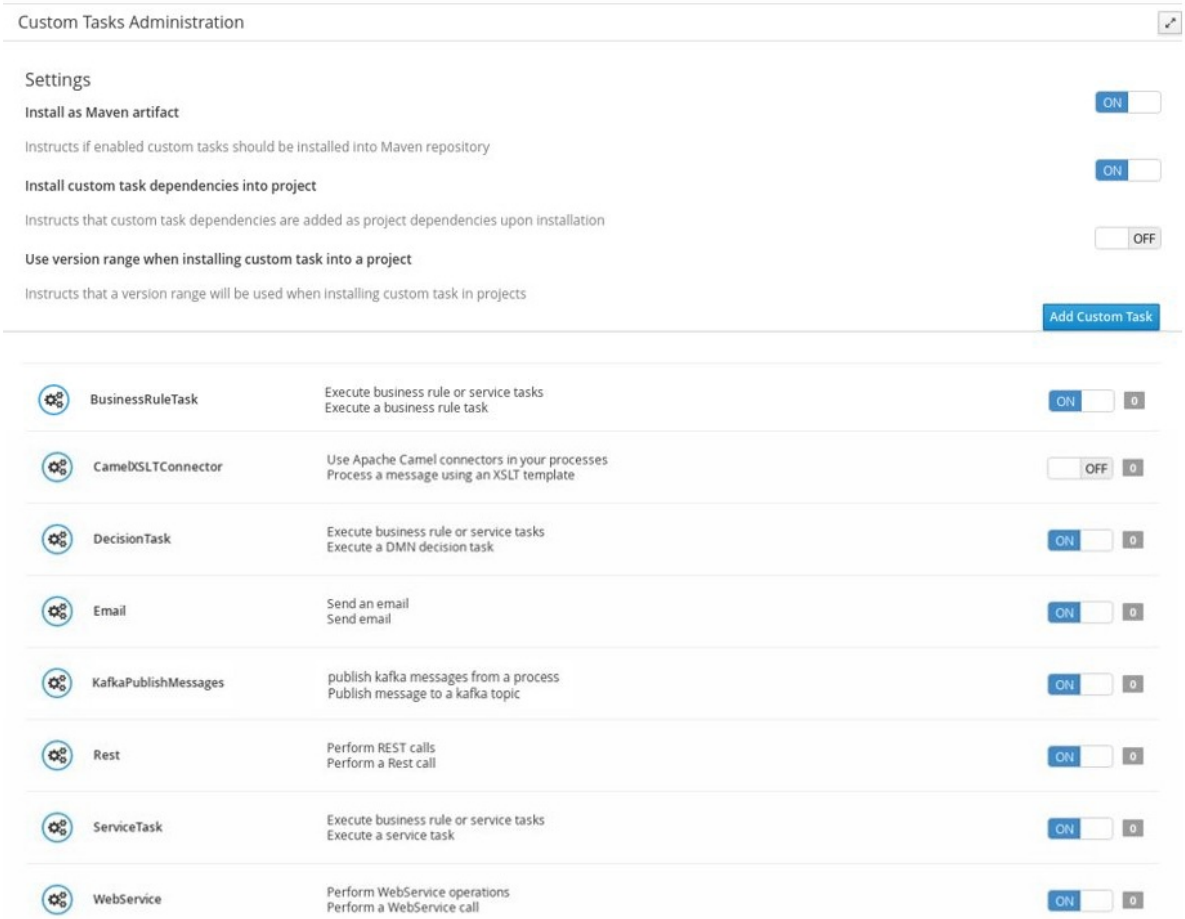
1. Business Central で右上隅の をクリックし、**Custom Task Administration** を選択します。
   このページは、カスタムタスクのインストール設定や、Business Central 全体にあるプロジェクトのプロセスで利用可能なカスタムタスクを表示します。このページで有効にしたカスタムタスクは、プロジェクトレベルの設定で利用できます。プロジェクトレベルの設定で、プロセスで使用する各カスタムタスクをインストールできます。カスタムタスクをプロジェクトにインストールする方法は、**Custom Tasks Administration** ページの **Settings** で有効または無効にしたグローバル設定により決まります。

2. **Settings** で、各設定を有効または無効にして、ユーザーがプロジェクトレベルでインストールするときに、利用可能なカスタムタスクを実装する方法を決定します。
   以下のカスタムタスクの設定が利用できます。

   - **Install as Maven artifact** ファイルがない場合は、カスタムタスクの JAR ファイルを Maven リポジトリーにアップロードし、Business Central で設定します。
   - **Install custom task dependencies into project** カスタムタスクの依存関係をプロジェクトの **pom.xml** ファイルに追加します。このファイルでタスクがインストールされます。
   - **Use version range when installing custom task into project** プロジェクトの依存関係として追加されるカスタムタスクの固定バージョンではなく、バージョン範囲を使用します。たとえば、**7.16.0.Final** ではなく **[7.16,)** です。

3. 必要に応じて利用可能なカスタムタスクを有効または無効にします (**ON** または **OFF** に設定)。有効化したカスタムタスクは、Business Central の全プロジェクトのプロジェクトレベル設定に表示されます。

図58.1 カスタムタスクとカスタムタスク設定の有効化



4. カスタムタスクを追加するには、**Add Custom Task**をクリックし、関連する JAR ファイルを参照し、**Upload** アイコンをクリックします。クラスが **WorkItemHandler** を実装する場合は、ファイルを Business Central に個別に追加して、アノテーションを **.wid** ファイルに置き換えることができます。

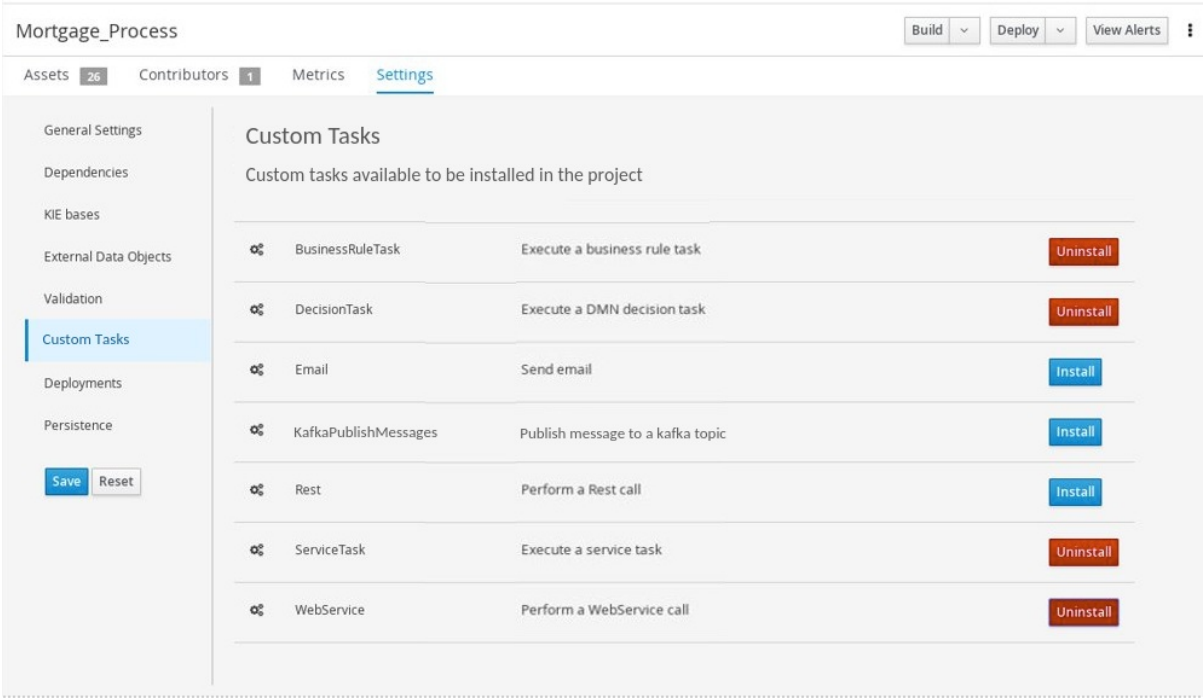
5. 必要に応じてカスタムタスクを削除するには、削除するカスタムタスクの行にある **remove** をクリックし、**OK** をクリックして削除を確定します。

6. Business Central ですべての必須カスタムタスクを設定した後に、プロジェクトの **Settings** → **Custom Tasks** ページに移動すると、有効化したカスタムタスクで利用可能なものが表示されます。

7. カスタムタスクごとに、**Install** をクリックして、対象のプロジェクトのプロセスでタスクを利用できるようにするか、**Uninstall** をクリックして、プロジェクトのプロセスからタスクを除外します。

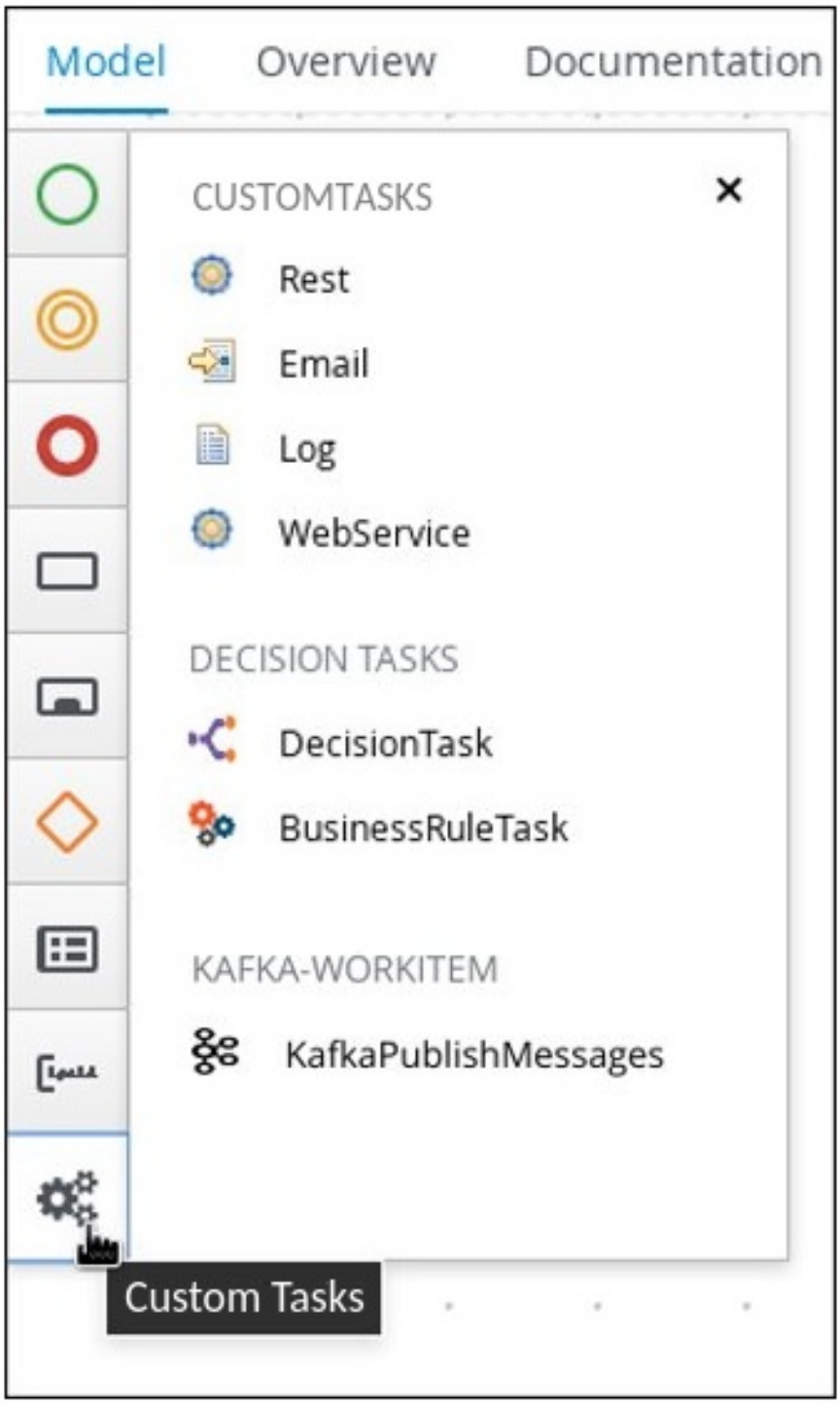
8. カスタムタスクのインストール時に追加情報を求められた場合は、必要な情報を入力して、もう一度 **Install** をクリックします。
カスタムタスクの必須パラメーターは、タスクのタイプにより異なります。たとえば、ルールとデシジョンタスクにはアーティファクトの GAV 情報 (グループ ID、アーティファクト ID、およびバージョン) が、メールタスクにはホストとポートアクセスの情報が、REST タスクには API の認証情報が必要です。他のカスタムタスクでは、追加のパラメーターが必要でない場合もあります。

図58.2 プロセスで使用するカスタムタスクのインストール



9. **Save** をクリックします。
10. プロジェクトページに戻り、プロジェクトのビジネスプロセスを選択または追加します。プロセスデザイナーパレットで **Custom Tasks** オプションを選択すると、有効にしてインストールした、利用可能なカスタムタスクが表示されます。

図58.3 プロセスデザイナーでのインストール済みカスタムタスクへのアクセス



第59章 ワークアイテムハンドラーのプロジェクト作成

カスタムタスクの設定、マッピング、実行可能なコードをすべて含むソフトウェアプロジェクトを作成します。

最初からワークアイテムハンドラーを作成するか、Maven アーキタイプを使用してサンプルプロジェクトを作成できます。Red Hat Process Automation Manager では、この目的のために、Red Hat Maven リポジトリーから **jbpm-workitems-archetype** を提供します。

手順

1. コマンドラインを開き、**workitem-home** などのワークアイテムハンドラーをビルドするディレクトリーを作成します。

```
$ mkdir workitem-home
```

2. Maven **settings.xml** ファイルを確認して、Red Hat Maven リポジトリーがリポジトリー一覧に含まれていることを確認します。

注記

Maven の設定は、本書の対象外となります。

たとえば、オンラインの Red Hat Maven リポジトリーを Maven **settings.xml** ファイルに追加します。

```
<settings>
  <profiles>
    <profile>
      <id>my-profile</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
        <repository>
          <id>redhat-ga</id>
          <url>http://maven.repository.redhat.com/ga/</url>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
          <releases>
            <enabled>true</enabled>
          </releases>
        </repository>
        ...
      </repositories>
    </profile>
  </profiles>
  ...
</settings>
```

3. Red Hat ライブラリーバージョンを検索して、以下のタスクの 1 つを実行します。

- オンラインでライブラリーのバージョンを検索する場合には、What is the mapping between Red Hat Process Automation Manager and the Maven library version? を参照してください。
- オフラインでライブラリーのバージョンを検索するには、**business-central.war/META-INF/MANIFEST.MF** の **Implementation-Version** か、**kie-server.war/META-INF/MANIFEST.MF** の **Implementation-Version** を確認してください。

4. **workitem-home** ディレクトリーで、以下のコマンドを実行します。

```
$ mvn archetype:generate \
-DarchetypeGroupId=org.jbpm \
-DarchetypeArtifactId=jbpm-workitems-archetype \
-DarchetypeVersion=<redhat-library-version> \
-Dversion=1.0.0-SNAPSHOT \
-DgroupId=com.redhat \
-DartifactId=myworkitem \
-DclassPrefix=MyWorkItem
```

表59.1 パラメーターの説明

パラメーター	説明
-DarchetypeGroupId	アーキタイプ固有となるため、変更しないようにしてください。
-DarchetypeArtifactId	アーキタイプ固有となるため、変更しないようにしてください。
-DarchetypeVersion	Maven が **jbpm-workitems-archetype** アーティファクトをダウンロードしようとすると、検索される Red Hat ライブラリーのバージョン
-Dversion	特定のプロジェクトのバージョン。たとえば、**1.0.0-SNAPSHOT** です。
-DgroupId	特定のプロジェクトの Maven グループ。たとえば、**com.redhat** です。
-DartifactId	特定のプロジェクトの Maven ID。たとえば、**myworkitem** です。
-DclassPrefix	簡単に特定できるように Maven がクラスを生成したときに、Java クラスの最初に追加される文字列。たとえば、**MyWorkItem** です。

myworkitem ディレクトリーは、**workitem-home** ディレクトリーで作成されます。以下に例を示します。

```
assembly/
  assembly.xml
src/
```

```
  main/
    java/
      com/
        redhat/
          MyWorkItemWorkItemHandler.java
    repository/
    resources/
  test/
    java/
      com/
        redhat/
          MyWorkItemWorkItemHandlerTest.java
          MyWorkItemWorkItemIntegrationTest.java
    resources/
      com/
        redhat/
pom.xml
```

5. **pom.xml** ファイルに対するワークアイテムハンドラーが必要とする Maven の依存関係を追加します。

6. このプロジェクトのデプロイ可能な JAR を作成するには、pom.xml ファイルが配置されている親プロジェクトのフォルダーで、以下のコマンドを実行します。

```
$ mvn clean package
```

複数のファイルが **target/** ディレクトリーに作成されます。このディレクトリーには主に以下の 2 つのファイルが含まれます。

表59.2 ファイルの説明

パラメーター	説明
myworkitems-<version>.jar	Red Hat Process Automation Manager に直接デプロイする時に使用します。
myworkitems-<version>.zip	サービスリポジトリーを使用するデプロイメントに使用します。

第60章 ワークアイテムハンドラープロジェクトのカスタマイズ

ワークアイテムハンドラープロジェクトのコードをカスタマイズできます。ワークアイテムハンドラーが必要とする Java メソッドは **executeWorkItem** と **abortWorkItem** の 2 つです。

表60.1 Java メソッドの説明

Java メソッド	説明
executeWorkItem(WorkItem workItem, WorkItemManager manager)	ワークアイテムハンドラーの実行時にデフォルトで実行されます。
abortWorkItem(WorkItem workItem, WorkItemManager manager)	ワークアイテムが中断すると実行します。

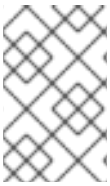
いずれのメソッドでも、**WorkItemDefinition** パラメーターには GUI または API 呼び出しでカスタムタスクに入力したパラメーターが含まれており、**WorkItem** パラメーターがカスタムタスクの状態を追跡します。

コード構造の例

```
public class MyWorkItemWorkItemHandler extends AbstractLogOrThrowWorkItemHandler {

  public void executeWorkItem(WorkItem workItem, WorkItemManager manager) {
    try {
      RequiredParameterValidator.validate(this.getClass(), workItem);

      // sample parameters
      String sampleParam = (String) workItem.getParameter("SampleParam");
      String sampleParamTwo = (String) workItem.getParameter("SampleParamTwo");

      // complete workitem impl...

      // return results
      String sampleResult = "sample result";
      Map<String, Object> results = new HashMap<String, Object>();
      results.put("SampleResult", sampleResult);
      manager.completeWorkItem(workItem.getId(), results);
    } catch(Throwable cause) {
      handleException(cause);
    }
  }

  @Override
  public void abortWorkItem(WorkItem workItem, WorkItemManager manager) {
    // similar
  }
}
```

表60.2 パラメーターの説明

パラメーター	説明
RequiredParameterValidator.validate(this.getClass(), workItem);	全パラメーターに required とマーク付けされていることを確認します。マーク付けされていない場合は、**IllegalArgumentException** が送出されます。
String sampleParam = (String) workItem.getParameter("SampleParam");	**WorkItem** クラスからパラメーターを取得する例。名前は、常に文字列です。たとえば、**WorkItem** です。この例では、**SampleParam** は常に文字列ですが、これに関連付けられているオブジェクトには多数の型を指定でき、エラーを回避するにはキャストが必要です。
// **complete workitem impl…**	パラメーターの受信時に、カスタムの Java コードが実行されます。
results.put("SampleResult", sampleResult);	カスタムタスクに結果を渡します。この結果は、カスタムタスクのデータ出力エリアに配置されます。
manager.completeWorkItem(workItem.getId(), results);	ワークアイテムハンドラーを完了とマークします。**WorkItemManager** はワークアイテムの状態を制御し、**WorkItem** ID を取得して、取得した結果と正しいカスタムタスクを関連付けます。
abortWorkItem()	カスタムの Java コードを中断します。ワークアイテムが中断されるように設計されていない場合は、空白のままにすることができます。



注記

Red Hat Process Automation Manager には、サポートされるカスタムタスクの限定セットが含まれています。Red Hat Process Automation Manager に含まれていないカスタムタスクはサポートされません。

第61章 ワークアイテム定義

Red Hat Process Automation Manager では、Business Central に表示するデータフィールドを特定して、API 呼び出しを受け入れるのにワークアイテム定義 (WID) ファイルが必要です。WID ファイルで、Red Hat Process Automation Manager のユーザーの操作と、ワークアイテムハンドラーに渡されるデータの間をマッピングします。WID ファイルでは、カスタムタスク名、Business Central のパレットに表示されるカテゴリー、カスタムタスクの指定に使用するアイコン、カスタムタスクがマッピングするワークアイテムハンドラーなど、UI の情報も処理します。

Red Hat Process Automation Manager は、次の 2 つの方法で WID ファイルを作成できます。

- ワークアイテムハンドラーをコード化する時に、**@Wid** アノテーションを使用する
- **.wid** テキストファイルを作成します。たとえば、**definitions-example.wid** です。

61.1. @WID アノテーション

Maven アーキタイプを使用してワークアイテムハンドラープロジェクトを生成するときに **@Wid** アノテーションは自動的に作成されます。このアノテーションは、手動でも追加できます。

@Wid の例

```
@Wid(widfile="MyWorkItemDefinitions.wid",
    name="MyWorkItemDefinitions",
    displayName="MyWorkItemDefinitions",
    icon="",
    defaultHandler="mvel: new com.redhat.MyWorkItemWorkItemHandler()",
    documentation = "myworkitem/index.html",
    parameters={
      @WidParameter(name="SampleParam", required = true),
      @WidParameter(name="SampleParamTwo", required = true)
    },
    results={
      @WidResult(name="SampleResult")
    },
    mavenDepends={
      @WidMavenDepends(group="com.redhat",
      artifact="myworkitem",
      version="7.52.0.Final-example-00007")
    },
    serviceInfo={
      @WidService(category = "myworkitem",
      description = "${description}",
      keywords = "",
      action = @WidAction(title = "Sample Title"),
      authinfo = @WidAuth(required = true,
      params = {"SampleParam", "SampleParamTwo"},
      paramsdescription = {"SampleParam", "SampleParamTwo"},
      referencesite = "referenceSiteURL"))
    }
)
```

表61.1 @Wid の説明

	説明
@Wid	WID ファイルを自動生成するトップレベルのアノテーション
widfile	Red Hat Process Automation Manager にデプロイするときに、カスタムタスク用に自動的に作成されるファイルの名前。
name	内部で使用されるカスタムタスク名。この名前は、Red Hat Process Automation Manager にデプロイするカスタムタスクで一意でなければなりません。
displayName	カスタムタスクの表示名。この名前は、Business Central のパレットに表示されます。
icon	**src/main/resources/** から、現在のプロジェクトに配置されているアイコンへのパス。このアイコンは、Business Central のパレットに表示されます。アイコンを指定する場合は、16x16 ピクセルの PNG または GIF ファイルでなければなりません。この値を空白のままにして、デフォルトの Service Task アイコンを使用することができます。
description	カスタムタスクの説明
defaultHandler	カスタムタスクにリンクされたワークアイテムハンドラーの Java クラス。このエントリーの形式は、**<language> : <class>** です。Red Hat Process Automation Manager は、この属性の言語の値として **mvel** を使用することを推奨しますが、**java** を使用することも可能です。mvel に関する詳細は、MVEL Documentation を参照してください。
ドキュメント	カスタムタスクの説明が含まれる現在のプロジェクトの HTML ファイルへのパス
@WidParameter	**@Wid** の子アノテーション。Business Central GUI で生成される値、またはカスタムタスクのデータ入力として API 呼び出しが必要とする値を指定します。複数のパラメーターを指定できます。 **name** - パラメーター名 **注記** この名前は、REST や SOAP などの転送メソッドの API 呼び出しで使用される可能性があるため、スペースや特殊文字を含めないでください。 **required** - パラメーターが、実行するカスタムタスクに必要かどうかを指定するブール値。

	説明
@WidResult	**@Wid** の子アノテーション。Business Central GUI で生成される値、またはカスタムタスクのデータ出力として API 呼び出しが必要とする値を指定します。複数の結果を指定できます。 **name** - 結果の名前 **注記** この名前は、REST や SOAP などの転送メソッドの API 呼び出しで使用される可能性があるため、スペースや特殊文字を含めないでください。
@WidMavenDepends	**@Wid** の子アノテーション。ワークアイテムハンドラーが正しく機能するのに必要な Maven の依存関係を指定します。複数の依存関係を指定できます。 **group** - 依存関係の Maven グループ ID **artifact** - 依存関係の Maven アーキタイプ ID **version** - 依存関係の Maven バージョン番号
@WidService	**@Wid** の子アノテーション。サービスリポジトリーで生成される値を指定します。 **category** - ハンドラーを配置する UI パレットカテゴリー。この値は、**@Wid** アノテーションの **category** フィールドと一致する必要があります。 **description** - サービスリポジトリーに表示されるハンドラーの説明 **keywords** - ハンドラーに適用するキーワードのコンマ区切りの一覧。注: 現在、Business Central サービスリポジトリーでは使用されていません。 **action** - **@WidAction** オブジェクト。 **authinfo** - **@WidAuth** オブジェクト。オプション:
@WidAction	**@WidService** のオブジェクトです。 **title** - ハンドラーアクションのタイトル **description** - ハンドラーアクションの説明。

	説明
@WidAuth	**@WidService** のオブジェクトです。 **required** - 認証が必要がどうかを判断するブール値 **params** - 必要な認証パラメーターが含まれるアレイ **paramsdescription** - 各認証パラメーターの説明が含まれるアレイ **referencesite** - ハンドラーのドキュメントの場所を指す URL。注: 現在、Business Central サービスリポジトリーでは使用されていません。

61.2. テキストファイル

グローバルな **WorkDefinitions** WID テキストファイルは、ビジネスプロセスが追加されると、新規プロジェクトで自動的に生成されます。WID テキストファイルは JSON 形式に似ていますが、完全に有効な JSON ファイルではありません。このファイルは、Business Central で開くことができます。既存のプロジェクトから **Add Asset** > **Work item definitions** の順に選択して、追加の WID ファイルを作成できます。

テキストファイルの例

```
[
  [
    "name" : "MyWorkItemDefinitions",
    "displayName" : "MyWorkItemDefinitions",
    "category" : "",
    "description" : "",
    "defaultHandler" : "mvel: new com.redhat.MyWorkItemWorkItemHandler()",
    "documentation" : "myworkitem/index.html",
    "parameters" : [
      "SampleParam" : new StringDataType(),
      "SampleParamTwo" : new StringDataType()
    ],
    "results" : [
      "SampleResult" : new StringDataType()
    ],
    "mavenDependencies" : [
      "com.redhat:myworkitem:7.52.0.Final-example-00007"
    ],
    "icon" : ""
  ]
]
```

ファイルは、JSON のような構造を使用してプレーンテキストファイルとして設定されます。ファイル名の拡張子は、**.wid** です。

表61.2 テキストファイルの説明

	説明

	説明
name	内部で使用されるカスタムタスク名。この名前は、Red Hat Process Automation Manager にデプロイするカスタムタスクで一意でなければなりません。
displayName	カスタムタスクの表示名。この名前は、Business Central のパレットに表示されます。
icon	**src/main/resources/** から、現在のプロジェクトに配置されているアイコンへのパス。このアイコンは、Business Central のパレットに表示されます。アイコンを指定する場合は、16x16 ピクセルの PNG または GIF ファイルでなければなりません。この値を空白のままにして、デフォルトの Service Task アイコンを使用することができます。
category	このカスタムタスクが表示される Business Central パレット内のカテゴリー名
description	カスタムタスクの説明
defaultHandler	カスタムタスクにリンクされたワークアイテムハンドラーの Java クラス。このエントリーの形式は、**<language> : <class>** です。Red Hat Process Automation Manager は、この属性の言語の値として **mvel** を使用することを推奨しますが、**java** を使用することも可能です。mvel に関する詳細は、MVEL Documentation を参照してください。
ドキュメント	カスタムタスクの説明が含まれる現在のプロジェクトの HTML ファイルへのパス
parameters	Business Central GUI で生成される値または、カスタムタスクのデータ入力として API 呼び出しが必要とする値を指定します。パラメーターは、**<key> : <DataType>** 形式を使用します。許可されるデータタイプは、**StringDataType()**、**IntegerDataType()**、および **ObjectDataType()** です。複数の値を指定できます。
results	Business Central GUI で生成される値、またはカスタムタスクのデータ出力として API 呼び出しが必要とする値を指定します。結果は、**<key> : <DataType>** 形式を使用します。許可されるデータタイプは、**StringDataType()**、**IntegerDataType()**、および **ObjectDataType()** です。複数の結果を指定できます。
mavenDependencies	任意: ワークアイテムハンドラーを正しく機能させるのに必要な Maven 依存関係を指定します。依存関係は、ワークアイテムハンドラーの **pom.xml** ファイルで指定することもできます。依存関係は、**<group>:<artifact>:<version>** 形式で指定します。複数の依存関係を指定できます。

Red Hat Process Automation Manager はデフォルトで、2 つの場所で **.wid** ファイルの場所を特定しようと試みます。

- Business Central 内にあるプロジェクトのトップレベルの **global/** ディレクトリー。これは、プロジェクトがビジネスプロセスアセットに初めて追加されると自動的に作成される、デフォルトの **WorkDefinitions.wid** ファイルです。

- Business Central 内にあるプロジェクトの **src/main/resources/** ディレクトリー。これは、Business Central で作成した WD ファイルの配置場所です。WID ファイルは、Java パッケージレベルで作成できるため、**<default>** のパッケージ場所で作成される WID ファイルは、直接 **src/main/resources/** 内に作成され、**com.redhat** のパッケージ場所で作成される WID ファイルは **src/main/resources/com/redhat/** に作成されます。

警告

Red Hat Process Automation Manager では、**defaultHandler** タグの値が実行可能化どうか、有効な Java クラス化どうかは検証されません。このタグに無効なクラスや間違ったクラスを指定するとエラーが返されます。

第62章 カスタムタスクのデプロイ

ワークアイテムハンドラーは、Red Hat Process Automation Manager 外にカスタムコードとして作成されます。カスタムタスクでこのコードを使用するには、このコードはサーバーにデプロイする必要があります。ワークアイテムハンドラープロジェクトは、Maven リポジトリーに配置可能な Java JAR ファイルでなければなりません。

Red Hat Process Automation Manager では、以下の 2 つの方法でカスタムタスクをデプロイできます。

- Business Central のカスタムタスクリポジトリー内。詳細は、**Business Central でのビジネスプロセスの管理および監視** を参照してください。
- Business Central を使用せずに、JAR ファイルを Maven リポジトリーに手動でコピーできます。

62.1. BUSINESS CENTRAL のカスタムタスクリポジトリーの使用

Business Central のカスタムタスクリポジトリー内でカスタムタスクを有効または無効にしたり、デプロイしたりできます。詳細は、**Business Central におけるカスタムタスクの管理** を参照してください。

62.2. JAR アーティファクトの BUSINESS CENTRAL へのアップロード

Business Central の管理者設定を使用して、ワークアイテムハンドラーの JAR を Business Central Maven リポジトリーにアップロードすることができます。

手順

1. Business Central で、画面の右上隅にある **Admin** アイコンを選択し、**Artifacts** を選択します。
2. **Upload** をクリックします。
3. **Artifact Upload** ウインドウで、**Choose File** アイコンをクリックします。
4. ワークアイテムハンドラーの JAR の場所に移動し、ファイルを選択して **Open** をクリックします。
5. ポップアップダイアログで **Upload** アイコンをクリックします。
   アーティファクトがアップロードされ、**Artifacts** ページで表示して参照できるようになります。

62.3. BUSINESS CENTRAL MAVEN リポジトリーへのワークアイテム定義の手動コピー

Business Central は、Maven リポジトリーディレクトリーを自動的に作成するか、再利用します。デフォルトでは、Red Hat JBoss EAP を起動したユーザーの場所をもとに、場所が決定されます。たとえば、デフォルトのパスは、**<startup location>/repositories/kie/global** です。このディレクトリー内の **<groupId>/<artifactId>/<versionId>/** の標準の Maven リポジトリーフォルダーレイアウトを複製し、ワークアイテムハンドラー JAR ファイルをこの場所にコピーすることができます。以下に例を示します。

```
<startup location>/repositories/kie/global/com/redhat/myworkitem/1.0.0-SNAPSHOT/myworkitems-1.0.0-SNAPSHOT.jar
```

この形式でコピーしたアーティファクトは、サーバーを再起動しなくても Red Hat Process Automation Manager で利用できます。Business Central の **Artifacts** ページでアーティファクトを表示するには、**Refresh** をクリックする必要があります。

第63章 カスタムタスクの登録

Red Hat Process Automation Manager は、カスタムタスクのワークアイテムと、ワークアイテムハンドラーが実行するコードと関連付ける方法を知っておく必要があります。作業項目定義ファイルは、名前と Java クラスによってカスタムタスクを作業項目ハンドラーにリンクします。ワークアイテムハンドラーの Java クラスは、Red Hat Process Automation Manager で利用できるように登録しておく必要があります。

注記

サービスリポジトリーには、各種システムとプロセスを統合できるように、ドメイン固有のサービスが含まれています。サービスリポジトリーを使用する場合は、インポートプロセスでカスタムタスクが登録されるため、カスタムタスクの登録は、必要ありません。

Red Hat Process Automation Manager では、ビジネスプロセスが最低 1 つ含まれるプロジェクトには、デフォルトで WID ファイルが作成されます。ワークアイテムハンドラーの登録時に WID ファイルを作成したり、デフォルトの WID ファイルを編集したりできます。WID ファイルの場所とフォーマットに関する詳細は、61章 *ワークアイテム定義*を参照してください。

サービスリポジトリーを使用しないデプロイメントの場合、ワークアイテムハンドラーは 2 種類の方法で登録できます。

- デプロイメント記述子を使用した登録
- Spring コンポーネント登録を使用した登録

63.1. BUSINESS CENTRAL でデプロイメント記述子を使用したカスタムタスクの登録

Business Central でデプロイメント記述子を使用してワークアイテムハンドラーで、カスタムタスクのワークアイテムを登録できます。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクト名を選択します。
2. プロジェクトペインで **Settings → Deployments → Work Item Handlers** の順に選択します。
3. **Add Work Item Handler** をクリックします。
4. **Name** フィールドで、カスタムタスクの表示名を入力します。
5. **Resolver** リストから **MVEL**、**Reflection** または **Spring** を選択します。
6. **Value** フィールドに、リゾルバー-タイプをもとに値を入力します。

注記

値フィールドは自動的に入力できます。

- MVEL の場合には、**new <full Java package>.<Java work item handler class name>()** の形式を使用します。
  例: **new com.redhat.MyWorkItemWorkItemHandler()**

- Reflection の場合には、**<full Java package>.<Java work item handler class name>** の形式を使用します。
  例: **com.redhat.MyWorkItemWorkItemHandler**

- Spring の場合には **<Spring bean identifier>** の形式を使用します。
  例: **workItemSpringBean**

7. **Save** をクリックして変更を保存します。

63.2. BUSINESS CENTRAL 外でのデプロイメント記述子を使用したカスタムタスクの登録

Business Central 外でデプロイメント記述子を使用して、ワークアイテムハンドラーでカスタムタスクのワークアイテムを登録できます。

手順

1. Git プロジェクトのディレクトリーで、**src/main/resources/META-INF/kie-deployment-descriptor.xml** を開きます。
2. **<work-item-handlers>** のリゾルバータイプをもとに、以下の内容を追加します。

   - MVEL の場合には、以下を追加します。

```
<work-item-handler>
  <resolver>mvel</resolver>
  <identifier>new com.redhat.MyWorkItemWorkItemHandler()</identifier>
  <parameters/>
  <name>MyWorkItem</name>
</work-item-handler>
```

   - Reflections の場合には、以下を追加します。

```
<work-item-handler>
  <resolver>reflection</resolver>
  <identifier>com.redhat.MyWorkItemWorkItemHandler</identifier>
  <parameters/>
  <name>MyWorkItem</name>
</work-item-handler>
```

   - Spring の場合には、以下を追加し、識別子が Spring Bean の識別子であることを確認します。

```
<work-item-handler>
  <resolver>spring</resolver>
  <identifier>beanIdentifier</identifier>
  <parameters/>
  <name>MyWorkItem</name>
</work-item-handler>
```

注記

Spring を使用して、Spring bean を検出して設定する場合には、**org.springframework.stereotype.Component** クラスのアノテーションを使用して、ワークアイテムハンドラーを自動的に登録できます。

ワークアイテムハンドラー内で、ワークアイテムハンドラークラスの宣言の前に **@Component("<Name>")** のアノテーションを追加します。例:
@Component("MyWorkItem") public class MyWorkItemWorkItemHandler extends AbstractLogOrThrowWorkItemHandler {

第64章 カスタムタスクの配置

Red Hat Process Automation Manager でカスタムタスクを登録すると、カスタムタスクはプロセスデザイナーのパレットに表示されます。カスタムタスクに名前がつけられ、対応する WID ファイルのエントリーに従って分類されます。

前提条件

- カスタムタスクが Red Hat Process Automation Manager に登録されます。詳細は、63章*カスタムタスクの登録* を参照してください。
- カスタムタスクに名前がつけられ、対応する WID ファイルに従って分類されます。WID ファイルの場所とフォーマットに関する詳細は、61章 *ワークアイテム定義*を参照してください。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクトをクリックします。
2. カスタムタスクを追加するビジネスプロセスを選択します。
3. パレットからカスタムタスクを選択し、BPMN2 ダイアグラムにドラッグします。
4. 必要に応じて、カスタムタスク属性を変更します。たとえば、該当する WID ファイルから、データの出入力を変更します。

注記

WID ファイルがプロジェクトに表示されず、プロジェクトの **Others** カテゴリーに **Work Item Definition** オブジェクトが表示されない場合は、カスタムタスクを登録する必要があります。カスタムタスクの登録に関する詳細は、63章 *カスタムタスクの登録* を参照してください。

パート VII. RED HAT PROCESS AUTOMATION MANAGER のプロセスエンジン

ビジネスプロセスまたは開発者は、Red Hat Process Automation Manager のプロセスエンジンを理解すると、より効率的なビジネスアセットとよりスケーラブルなプロセス管理アーキテクチャーを設計できるようになります。プロセスエンジンは、Red Hat Process Automation Manager で Business Process Management (BPM) パラダイムを実装し、プロセスを設定するビジネスアセットを管理し、実行します。本ガイドでは、お客様が Red Hat Process Automation Manager でビジネスプロセス管理システムおよびプロセスサービスを作成する際に検討すべき、プロセスエンジンに関する概念および機能を説明します。

第65章 RED HAT PROCESS AUTOMATION MANAGER のプロセスエンジン

プロセスエンジンは、Red Hat Process Automation Manager に Business Process Management (BPM) のパラダイムを実装します。BPM は、企業内でプロセスのモデル化、測定、最適化を可能にするビジネス方法論です。

BPM では、繰り返し可能なビジネスプロセスはワークフローダーイアグラムとして表示されます。BPMN (Business Process Model and Notation) 仕様は、このダイアグラムで利用可能な要素を定義します。プロセスエンジンは BPMN 2.0 仕様の大規模なサブセットを実装します。

プロセスエンジンを使用すると、ビジネスアナリストはダイアグラム自体を作成できます。開発者はコード内にフローのすべての要素のビジネスロジックを実装して、実行可能なビジネスプロセスを作成できます。ユーザーは、ビジネスプロセスを実行し、必要に応じて対話できます。プロセスの効率を反映するメトリックスを生成できます。

ワークフローダーイアグラムは、複数のノードで設定されます。BPMN 仕様では、以下のプリンシパルタイプを含む、さまざまな種類のノードが定義されていました。

- **イベント**: プロセスの内外で発生するプロセスを表すノード。通常、イベントはプロセスの開始点と終了点です。イベントは、他のプロセスにメッセージを発生させ、このようなメッセージを取得できます。ダイアグラムの丸印はイベントを表します。

- **アクティビティー**: (自動またはユーザーの関与を伴うかどうかに関係なく) 実行する必要があるアクションを表すノード。典型的なイベントは、プロセス内で行われるアクション、およびサブプロセスの呼び出しを表すタスクです。角の丸い長方形はアクティビティーを表しています。

- **ゲートウェイ**: 分岐ノードまたはマージノード。一般的なゲートウェイは式を評価し、結果によっては複数の実行パスの 1 つに続きます。ダイアグラム内のひし形はゲートウェイを表しています。

ユーザーがプロセスを開始すると、プロセスインスタンスが作成されます。プロセスインスタンスには、プロセス変数に保存されたデータまたは **コンテキスト** のセットが含まれます。プロセスインスタンスの **状態** には、すべてのコンテキストデータと現在のアクティブなノード (場合によっては複数のアクティブなノード) が含まれます。

これらの変数の一部は、ユーザーがプロセスの開始時に初期化できます。アクティビティーはプロセス変数から読み取り、変数への書き込みが可能です。ゲートウェイはプロセス変数を評価し、実行パスを判断できます。

たとえば、ショップでの購入プロセスはビジネスプロセスである可能性があります。ユーザーのカートの内容は、最初のプロセスコンテキストにすることができます。実行の最後に、プロセスコンテキストは、支払いの確認と出荷追跡の詳細を含めることができます。

必要に応じて、Business Central で BPMN データモデラ—を使用して、プロセス変数内のデータのモデルを設計できます。

ワークフローダーイアグラムは、XML の **ビジネスプロセス定義** によってコードで表されます。イベント、ゲートウェイ、およびサブプロセスの呼び出しのロジックは、ビジネスプロセス定義内に定義されます。

一部のタスクタイプ (スクリプトタスクや標準のデシジョンエンジンのルールタスクなど) がエンジンに実装されているものもあります。すべてのカスタムタスクを含む他のタスクタイプについては、タスクを実行する際に、プロセスエンジンが **Work Item Handler API** を使用して呼び出しを実行します。エン

ジンの外部にあるコードはこの API を実装し、さまざまなタスクを実装するための柔軟なメカニズムを提供します。

プロセスエンジンには、定義済みのタスクタイプが含まれます。これらのタイプには、ユーザー Java コードを実行するスクリプトタスク、Java メソッドまたは Web サービスを呼び出すサービスタスク、デシジョンエンジンサービスを呼び出すデシジョンタスク、およびその他のカスタムタスク (REST やデータベースコールなど) が含まれます。

もう 1 つの事前定義されたタスクタイプは **ユーザータスク** で、ユーザーとの対話が含まれます。プロセス内のユーザータスクは、ユーザーおよびグループに割り当てることができます。

プロセスエンジンは、**KIE API** を使用して他のソフトウェアコンポーネントと対話します。KIE Server のサービスとしてビジネスプロセスを実行し、KIE API の REST 実装を使用してそれらと対話できます。そのため、アプリケーションにビジネスプロセスを埋め込むことができ、KIE API Java 呼び出しを使用して対話できます。この場合は、任意の Java 環境でプロセスエンジンを実行できます。

Business Central には、人的タスクを実行するユーザー向けのユーザーインターフェイスと、人的タスクの Web フォームを作成するフォームモデラーが含まれています。ただし、KIE API を使用してプロセスエンジンと対話するカスタムユーザーインターフェイスを実装することもできます。

プロセスエンジンでは、以下の追加機能がサポートされます。

- JPA 標準を使用したプロセス情報の永続化のサポート。永続は、すべてのプロセスインスタンスの状態およびコンテキスト (プロセス変数内のデータ) を保持するため、コンポーネントが再起動またはオフラインになった場合に失われることはありません。SQL データベースエンジンを使用して永続情報を保存することができます。

- JTA 標準を使用したプロセス要素のトランザクション実行に対するプラグ可能なサポート。JTA トランザクションマネージャーを使用する場合は、ビジネスプロセスのすべての要素がトランザクションとして開始します。要素が完了しない場合、プロセスインスタンスのコンテキストは、要素の起動前の状態に復元されます。

- 新規ノード種別や他のプロセス言語を含むカスタム拡張コードのサポート。

- さまざまなイベントについて通知するカスタムリスナークラスのサポート。

- 実行中のプロセスインスタンスを、新しいバージョンのプロセス定義に移行するためのサポート

プロセスエンジンは、他の独立したコアサービスと統合することもできます。

- **人的タスクサービス** は、人間アクターがプロセスに参加する必要がある場合にユーザータスクを管理できます。これは完全にプラグ可能で、デフォルトの実装は WS-HumanTask 仕様をベースとしています。ヒューマンタスクサービスは、タスク、タスクリスト、タスクフォームのライフサイクルを管理します。また、エスカレーション、委任、ルールベースの割り当てなどのより高度な機能のライフサイクルも管理します。

- **履歴ログ** は、プロセスエンジンのすべてのプロセスの実行に関する情報をすべて保存できます。ランタイム永続化はすべてのアクティブなプロセスインスタンスの現在の状態を保存しますが、履歴ログで履歴情報にアクセスできるようにする必要があります。履歴ログには、アクティブかつ完了したすべてのプロセスインスタンスの現在および履歴状態がすべて含まれます。ログを使用して、監視および分析のプロセスインスタンスの実行に関連する情報をクエリーできます。

関連情報

- BPMN モデルを使用したビジネスプロセスの作成

- **KIE API を使った Red Hat Process Automation Manager の操作**
- 公開 KIE API の Java ドキュメント

第66章 プロセスエンジンのコアエンジン API

プロセスエンジンは、ビジネスプロセスを実行します。プロセスを定義するには、プロセス定義やカスタムタスクなどの **ビジネスアセット** を作成します。

Core Engine API を使用して、プロセスエンジンでプロセスを読み込み、実行、および管理できます。

複数の制御レベルが利用できます。

- 最小レベルでは、**KIE ベース** と **KIE セッション** を直接作成できます。KIE ベースは、ビジネスプロセスのすべてのアセットを表します。KIE セッションは、ビジネスプロセスのインスタンスを実行するプロセスエンジン内のエンティティーです。このレベルは粒度の細かい制御を提供しますが、コード内のプロセスインスタンス、タスクハンドラー、イベントハンドラー、およびその他のプロセスエンジンエンティティーの明示的な宣言および設定が必要になります。

- **RuntimeManager** クラスを使用してセッションおよびプロセスを管理できます。このクラスは設定可能なストラテジーを使用して、必要なプロセスインスタンスのセッションを提供します。KIE セッションとタスクサービスとの間の対話を自動的に設定します。不要になったプロセスエンジンエンティティーを破棄し、リソースを最適に使用できるようにします。Fluent API を使用して、必要なビジネスアセットで **RuntimeManager** をインスタンス化し、その環境を設定できます。

- **Services API** を使用してプロセスの実行を管理することができます。たとえば、デプロイメントサービスはビジネスアセットをエンジンにデプロイし、**デプロイメントユニット** を形成します。プロセスサービスは、このデプロイメントユニットからプロセスを実行します。
  プロセスエンジンをアプリケーションに埋め込む場合、サービス API はエンジンの設定および管理に関する詳細を非表示にするため、最も便利なオプションになります。

- 最後に、KJAR ファイルからビジネスアセットを読み込んでプロセスを実行する **KIE Server** をデプロイできます。KIE Server は、プロセスの読み込みおよび管理のための REST API を提供します。Business Central を使用して KIE Server を管理することもできます。
  KIE Server を使用する場合は、Core Engine API を使用する必要はありません。KIE Server へのプロセスのデプロイおよび管理に関する情報は、**Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ** を参照してください。

すべてのパブリックプロセスエンジン API 呼び出しに関する完全なリファレンス情報は、Java ドキュメント を参照してください。他の API クラスもコードに存在しますが、今後のバージョンで変更できる内部 API です。開発および維持するアプリケーションでパブリック API を使用します。

66.1. KIE ベースおよび KIE セッション

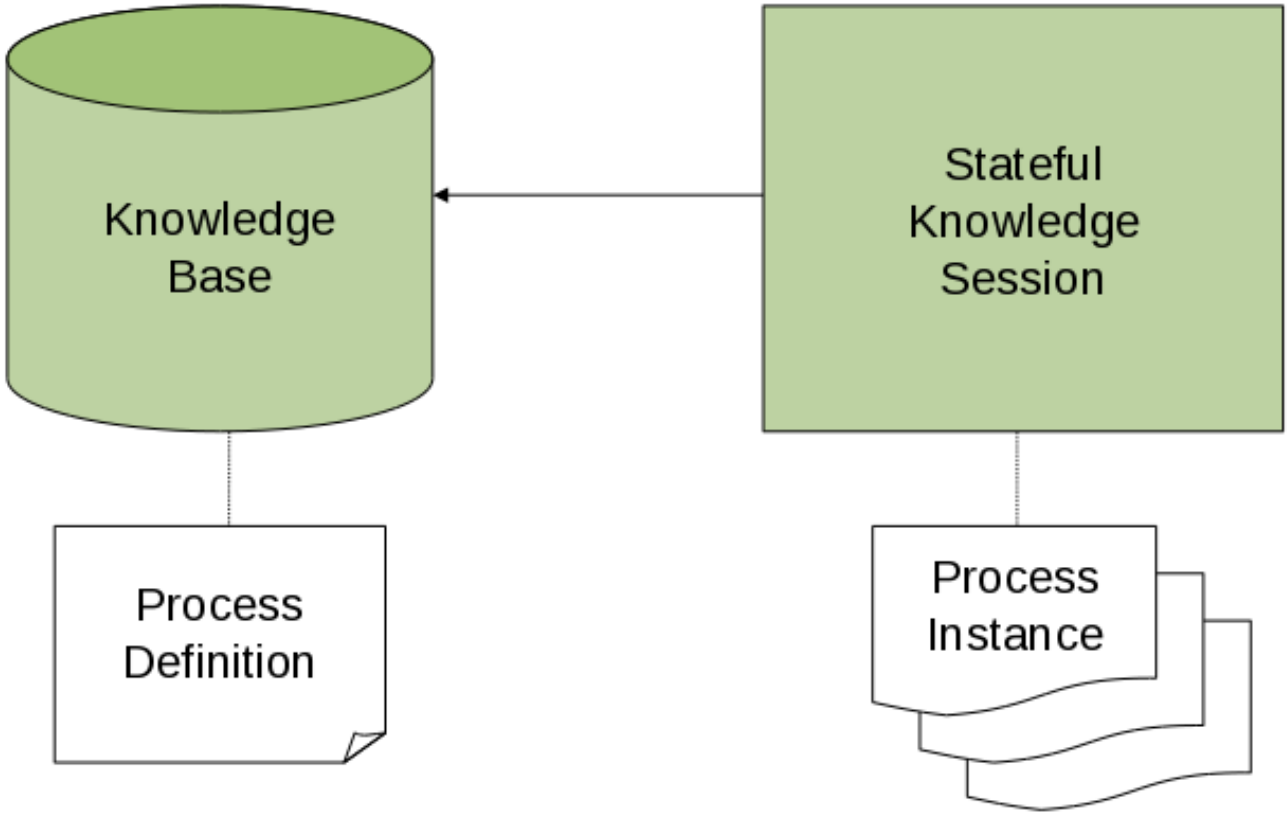
KIE base には、すべてのプロセス定義およびプロセスに関連するその他のアセットへの参照が含まれます。エンジンはこの KIE ベースを使用してプロセスの全情報を検索するか、必要に応じて複数のプロセスについて検索します。

クラスパス、ファイルシステム、プロセスリポジトリーなど、さまざまなソースからアセットを KIE ベースに読み込むことができます。KIE ベースの作成は、さまざまなソースからのアセットの読み込みおよび解析を行う必要があるため、リソース負荷の高い操作です。KIE ベースを動的に変更して、ランタイム時にプロセス定義やその他のアセットを追加または削除できます。

KIE ベースを作成したら、この KIE ベースに基づいて **KIE セッション** をインスタンス化できます。この KIE セッションを使用して、KIE ベースの定義に基づいてプロセスを実行します。

KIE セッションを使用してプロセスを開始すると、新しい **プロセスインスタンス** が作成されます。このインスタンスは、特定のプロセス状態を維持します。同じ KIE セッションで異なるインスタンスは同じプロセス定義を使用できますが、状態は異なります。

図66.1 プロセスエンジンの KIE ベースおよび KIE セッション



たとえば、アプリケーションを開発して売上注文を処理する場合は、注文の処理方法を決定する 1 つ以上のプロセス定義を作成できます。アプリケーションの起動時に、これらのプロセス定義を含む KIE ベースを作成する必要があります。次に、この KIE ベースに基づいてセッションを作成できます。新しい売上注文が終了したら、注文用の新しいプロセスインスタンスを開始します。このプロセスインスタンスには、特定の売上リクエストのプロセスの状態が含まれます。

同じ KIE ベースに多くの KIE セッションを作成でき、同じ KIE セッション内に、プロセスのインスタンスを多数作成できます。KIE セッションを作成して、KIE セッション内にプロセスインスタンスを作成し、KIE ベースを作成するよりもはるかに少ないリソースを使用します。KIE ベースを変更する場合は、これを使用する KIE セッションで変更を自動的に使用できます。

ほとんどの単純なユースケースでは、1 つの KIE セッションを使用してすべてのプロセスを実行できます。必要に応じて、複数のセッションを使用することもできます。たとえば、異なる顧客による注文処理を完全に独立させる場合は、顧客ごとに KIE セッションを作成できます。スケーラビリティーの理由から複数のセッションを使用することもできます。

一般的なアプリケーションでは、KIE ベースまたは KIE セッションを直接作成する必要はありません。ただし、プロセスエンジン API の他のレベルを使用する場合は、このレベルが定義する API の要素と対話することができます。

66.1.1. Kie ベース

KIE ベースには、アプリケーションによるビジネスプロセスの実行に必要となる可能性のあるプロセス定義およびその他のアセットがすべて含まれます。

KIE ベースを作成するには、**KieHelper** インスタンスを使用して、クラスパスやファイルシステムなどの各種リソースからプロセスを読み込み、新規の KIE ベースを作成します。

以下のコードスニペットは、クラスパスから読み込まれるプロセス定義が 1 つだけの KIE ベースを作成する方法を示しています。

1つのプロセス定義を含む KIE ベースの作成

```
KieHelper kieHelper = new KieHelper();
KieBase kieBase = kieHelper
  .addResource(ResourceFactory.newClassPathResource("MyProcess.bpmn"))
  .build();
```

ResourceFactory クラスには、ファイル、URL、InputStream、Reader、およびその他のソースからリソースを読み込む同様のメソッドがあります。

注記

KIE ベースを作成するこの手動プロセスは、他の方法よりも簡単ですが、アプリケーションが管理しにくくなります。**RuntimeManager** クラスやサービス API など、KIE ベースを作成する他の方法を使用して、開発および維持が長い期間にわたって予定されているアプリケーションに使用します。

66.1.2. KIE セッション

KIE ベースを作成して読み込んだら、KIE セッションを作成してプロセスエンジンと対話できます。このセッションを使用してプロセスを開始および管理し、イベントにシグナルを送信できます。

以下のコードスニペットは、以前に作成した KIE ベースに基づいてセッションを作成し、プロセスインスタンスを開始し、プロセス定義で ID を参照します。

KIE セッションの作成およびプロセスインスタンスの開始

```
KieSession ksession = kbase.newKieSession();
ProcessInstance processInstance = ksession.startProcess("com.sample.MyProcess");
```

66.1.3. ProcessRuntime インターフェイス

KieSession クラスは、以下の定義が示すように、プロセスと対話するためのすべてのセッションメソッドを定義する **ProcessRuntime** インターフェイスを公開します。

ProcessRuntime インターフェイスの定義

```
/**
* Start a new process instance.  Use the process (definition) that
* is referenced by the given process ID.
*
* @param processId  The ID of the process to start
* @return the ProcessInstance that represents the instance of the process that was started
*/
 ProcessInstance startProcess(String processId);

 /**
* Start a new process instance.  Use the process (definition) that
* is referenced by the given process ID.  You can pass parameters
* to the process instance as name-value pairs, and these parameters set
* variables of the process instance.
 *
* @param processId  the ID of the process to start
```

```
     * @param parameters  the process variables to set when starting the process instance
     * @return the ProcessInstance that represents the instance of the process that was started
     */
    ProcessInstance startProcess(String processId,
                                 Map<String, Object> parameters);

    /**
     * Signals the process engine that an event has occurred. The type parameter defines
     * the type of event and the event parameter can contain additional information
     * related to the event.  All process instances that are listening to this type
     * of (external) event will be notified.  For performance reasons, use this type of
     * event signaling only if one process instance must be able to notify
     * other process instances. For internal events within one process instance, use the
     * signalEvent method that also include the processInstanceId of the process instance
     * in question.
     *
     * @param type the type of event
     * @param event the data associated with this event
     */
    void signalEvent(String type,
                     Object event);

    /**
     * Signals the process instance that an event has occurred. The type parameter defines
     * the type of event and the event parameter can contain additional information
     * related to the event.  All node instances inside the given process instance that
     * are listening to this type of (internal) event will be notified.  Note that the event
     * will only be processed inside the given process instance.  All other process instances
     * waiting for this type of event will not be notified.
     *
     * @param type the type of event
     * @param event the data associated with this event
     * @param processInstanceId the id of the process instance that should be signaled
     */
    void signalEvent(String type,
                     Object event,
                     long processInstanceId);

    /**
     * Returns a collection of currently active process instances.  Note that only process
     * instances that are currently loaded and active inside the process engine are returned.
     * When using persistence, it is likely not all running process instances are loaded
     * as their state is stored persistently.  It is best practice not to use this
     * method to collect information about the state of your process instances but to use
     * a history log for that purpose.
     *
     * @return a collection of process instances currently active in the session
     */
    Collection<ProcessInstance> getProcessInstances();

    /**
     * Returns the process instance with the given ID.  Note that only active process instances
     * are returned. If a process instance has been completed already, this method returns
     * null.
     *
     * @param id the ID of the process instance
```

```
     * @return the process instance with the given ID, or null if it cannot be found
     */
    ProcessInstance getProcessInstance(long processInstanceId);

    /**
     * Aborts the process instance with the given ID. If the process instance has been completed
     * (or aborted), or if the process instance cannot be found, this method will throw an
     * IllegalArgumentException.
     *
     * @param id the ID of the process instance
     */
    void abortProcessInstance(long processInstanceId);

    /**
     * Returns the WorkItemManager related to this session. This object can be used to
     * register new WorkItemHandlers or to complete (or abort) WorkItems.
     *
     * @return the WorkItemManager related to this session
     */
    WorkItemManager getWorkItemManager();
```

66.1.4. 相関キー

プロセスを処理する場合には、ビジネス ID をプロセスインスタンスに割り当ててから、生成されたインスタンス ID を保存せずに、割り当てたビジネス ID を使用してインスタンスを参照しなければならない場合があります。

このような機能を提供するために、プロセスエンジンは **CorrelationKey** インターフェイスを使用して **CorrelationProperties** を定義できます。**CorrelationKey** を実装するクラスには、記述される単一のプロパティーまたはマルチプロパティーセットがあります。プロパティーの値または複数のプロパティーの値の組み合わせは一意のインスタンスを参照します。

KieSession クラスは、相関機能をサポートする **CorrelationAwareProcessRuntime** インターフェイスを実装します。このインターフェイスは以下のメソッドを公開します。

CorrelationAwareProcessRuntime インターフェイスのメソッド

```
    /**
     * Start a new process instance.  Use the process (definition) that
     * is referenced by the given process ID.  You can pass parameters
     * to the process instance (as name-value pairs), and these parameters set
     * variables of the process instance.
     *
     * @param processId  the ID of the process to start
     * @param correlationKey custom correlation key that can be used to identify the process instance
     * @param parameters  the process variables to set when starting the process instance
     * @return the ProcessInstance that represents the instance of the process that was started
     */
    ProcessInstance startProcess(String processId, CorrelationKey correlationKey, Map<String,
Object> parameters);

    /**
     * Create a new process instance (but do not yet start it).  Use the process
     * (definition) that is referenced by the given process ID.
     * You can pass to the process instance (as name-value pairs),
```

```
     * and these parameters set variables of the process instance.
     * Use this method if you need a reference to the process instance before actually
     * starting it.  Otherwise, use startProcess.
     *
     * @param processId  the ID of the process to start
     * @param correlationKey custom correlation key that can be used to identify the process instance
     * @param parameters  the process variables to set when creating the process instance
     * @return the ProcessInstance that represents the instance of the process that was created (but
not yet started)
     */
    ProcessInstance createProcessInstance(String processId, CorrelationKey correlationKey,
Map<String, Object> parameters);

    /**
     * Returns the process instance with the given correlationKey.  Note that only active process
instances
     * are returned.  If a process instance has been completed already, this method will return
     * null.
     *
     * @param correlationKey the custom correlation key assigned when the process instance was
created
     * @return the process instance identified by the key or null if it cannot be found
     */
    ProcessInstance getProcessInstance(CorrelationKey correlationKey);
```

相関は通常、長時間実行されるプロセスで使用されます。相関情報を永続的に保存する場合は、永続性を有効にする必要があります。

66.2. ランタイムマネージャー

RuntimeManager クラスは、プロセスエンジン API でレイヤーを提供し、その使用方法を単純化し、強化します。このクラスは、KIE ベースおよび KIE セッションと、プロセス内のすべてのタスクにハンドラーを提供するタスクサービスをカプセル化および管理します。ランタイムマネージャーの KIE セッションとタスクサービスが相互に機能するように設定されているため、このような設定を指定する必要はありません。たとえば、ヒューマンタスクハンドラーを登録し、必要なサービスに接続されていることを確認する必要はありません。

ランタイムマネージャーは、事前定義されたストラテジーに従って KIE セッションを管理します。以下のストラテジーを利用できます。

- **シングルトン**: ランタイムマネージャーは単一の **KieSession** を維持し、要求されたすべてのプロセスに使用します。
- **リクエストごと**: ランタイムマネージャーはリクエストごとに新しい **KieSession** を作成します。
- **プロセスインスタンスごと**: ランタイムマネージャーはプロセスインスタンスと **KieSession** の間のマッピングを維持し、指定のプロセスインスタンスを使用するたびに常に同じ **KieSession** を提供します。

ストラテジーに関係なく、**RuntimeManager** クラスはプロセスエンジンコンポーネントの初期化と設定で同じ機能を保証します。

- **KieSession** インスタンスは同じファクトリーで読み込まれます (メモリーまたは JPA ベース)。

- ワークアイテムハンドラーは、すべての **KieSession** インスタンスに登録されます (データベースから読み込むか、新規に作成されているかのいずれか)。
- イベントリスナー (**Process**、**Agenda**、**WorkingMemory**) は、データベースからセッションが読み込まれるか、新たに作成されるかに関係なく、すべての KIE セッションに登録されます。
- タスクサービスは、以下の必須コンポーネントで設定されます。
  - JTA トランザクションマネージャー
  - **KieSession** インスタンスに使用されるものと同じエンティティーマネージャーファクトリー
  - 環境で設定できる **UserGroupCallback** インスタンス

ランタイムマネージャーは、プロセスエンジンを正常に破棄することもできます。これは、必要がなくなった場合に **RuntimeEngine** インスタンスを破棄する専用のメソッドを提供し、取得した可能性のあるリソースを解放します。

以下のコードは、**RuntimeManager** インターフェイスの定義を示しています。

RuntimeManager インターフェイスの定義

```
public interface RuntimeManager {

 /**
  * Returns a <code>RuntimeEngine</code> instance that is fully initialized:
  * <ul>
  *  <li>KieSession is created or loaded depending on the strategy</li>
  *  <li>TaskService is initialized and attached to the KIE session (through a listener)</li>
  *  <li>WorkItemHandlers are initialized and registered on the KIE session</li>
  *  <li>EventListeners (process, agenda, working memory) are initialized and added to the KIE
session</li>
  * </ul>
  * @param context the concrete implementation of the context that is supported by given
<code>RuntimeManager</code>
  * @return instance of the <code>RuntimeEngine</code>
  */
   RuntimeEngine getRuntimeEngine(Context<?> context);

   /**
    * Unique identifier of the <code>RuntimeManager</code>
    * @return
    */
   String getIdentifier();

   /**
    * Disposes <code>RuntimeEngine</code> and notifies all listeners about that fact.
    * This method should always be used to dispose <code>RuntimeEngine</code> that is not
needed
    * anymore. <br/>
    * Do not use KieSession.dispose() used with RuntimeManager as it will break the internal
    * mechanisms of the manager responsible for clear and efficient disposal.<br/>
    * Disposing is not needed if <code>RuntimeEngine</code> was obtained within an active JTA
transaction,
    * if the getRuntimeEngine method was invoked during active JTA transaction, then disposing of
```

```
     * the runtime engine will happen automatically on transaction completion.
     * @param runtime
     */
    void disposeRuntimeEngine(RuntimeEngine runtime);

    /**
     * Closes <code>RuntimeManager</code> and releases its resources. Call this method when
     * a runtime manager is not needed anymore. Otherwise it will still be active and operational.
     */
    void close();

}
```

RuntimeManager クラスは、基礎となるプロセスエンジンコンポーネントへのアクセスを取得するメソッドが含まれる **RuntimeEngine** クラスも提供します。

RuntimeEngine インターフェイスの定義

```
public interface RuntimeEngine {

 /**
  * Returns the <code>KieSession</code> configured for this <code>RuntimeEngine</code>
  * @return
  */
   KieSession getKieSession();

   /**
  * Returns the <code>TaskService</code> configured for this <code>RuntimeEngine</code>
  * @return
  */
   TaskService getTaskService();
}
```

注記

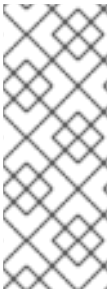
RuntimeManager クラスの識別子は、ランタイムの実行中に **deploymentId** として使用されます。たとえば、識別子は、**Task** が永続化される **Task** の **deploymentId** として永続化されています。**Task** の **deploymentID** は、**Task** が完了してプロセスインスタンスを再開する際に、これを **RuntimeManager** に関連付けます。

同じ **deploymentId** も履歴ログテーブルの **externalId** として永続化されます。

RuntimeManager インスタンスの作成時に識別子を指定しない場合は、ストラテジーに応じてデフォルト値が適用されます (例: **PerProcessInstanceRuntimeManager** の **default-per-pinstance**)。つまり、アプリケーションはライフサイクル全体で **RuntimeManager** クラスと同じデプロイメントを使用することを意味します。

アプリケーションで複数のランタイムマネージャーを維持する場合は、すべての **RuntimeManager** インスタンスに一意の ID を指定する必要があります。

たとえば、デプロイメントサービスは複数のランタイムマネージャーを維持し、KJAR ファイルの GAV 値を識別子として使用します。Business Central と KIE Server でも、デプロイメントサービスに依存するため、同じロジックが使用されます。



注記

ハンドラーまたはリスナー内からプロセスエンジンまたはタスクサービスと対話する必要がある場合は、**RuntimeManager** インターフェイスを使用して、指定のプロセスインスタンスの **RuntimeEngine** インスタンスを取得し、**RuntimeEngine** インスタンスを使用して **KieSession** インスタンスまたは **TaskService** インスタンスを取得します。このアプローチにより、選択したストラテジーに従って管理されるエンジンの正常な状態が維持されます。

66.2.1. ランタイムマネージャーストラテジー

RuntimeManager クラスは、KIE セッションを管理するための以下のストラテジーをサポートします。

シングルトンストラテジー

このストラテジーは、単一の **RuntimeEngine** インスタンスを維持するようにランタイムマネージャーに指示します (これにより、単一の **KieSession** インスタンスおよび **TaskService** インスタンス)。ランタイムエンジンへのアクセスは同期されるためスレッドセーフになりますが、同期によりパフォーマンスの低下が発生します。
このストラテジーは、簡単なユースケースに使用します。

このストラテジーには以下の特徴があります。

- ランタイムエンジンのインスタンスおよびタスクサービスが 1 つのインスタンスを持つメモリーフットプリントは小さくなります。
- 設計と使用はシンプルでコンパクトです。
- アクセスが同期されているため、プロセスエンジンの低から中程度の負荷に適しています。
- このストラテジーでは、単一の **KieSession** インスタンスが原因で、(ファクトなど) すべての状態オブジェクトがすべてのプロセスインスタンスに直接表示され、その逆も同様です。
- ストラテジーはコンテキストではありません。シングルトン **RuntimeManager** から **RuntimeEngine** のインスタンスを取得する場合は、**Context** インスタンスを考慮する必要はありません。通常は、**EmptyContext.get()** をコンテキストとして使用できますが、null 引数も受け入れ可能です。
- このストラテジーでは、ランタイムマネージャーは **KieSession** の ID を追跡するため、**RuntimeManager** の再起動後も同じセッションが使用されたままになります。ID はファイルシステムの一時的な場所にシリアライズファイルとして保存され、環境によっては以下のディレクトリーのいずれかになります。
  - **jbpm.data.dir** システムプロパティーの値
  - **jboss.server.data.dir** システムプロパティーの値
  - **java.io.tmpdir** システムプロパティーの値

警告

Singleton ストラテジーと EJB Timer スケジューラーの組み合わせにより、負荷がかかった状態で Hibernate の問題が発生する可能性があります。この組み合わせは、実稼働アプリケーションで使用しないでください。EJB Timer スケジューラーは、KIE Server のデフォルトスケジューラーです。

リクエスト別のストラテジー

このストラテジーは、ランタイムマネージャーに対し、リクエストごとに **RuntimeEngine** の新しいインスタンスを提供するように指示します。1 つのトランザクション内でのプロセスエンジンを 1 つ以上呼び出すことは、1 つのリクエストとみなされます。
状態が正確になるように、単一のトランザクション内で **RuntimeEngine** の同じインスタンスを使用する必要があります。それ以外の場合は、1 つの呼び出しで完了した操作が次の呼び出しで表示されません。

プロセスの状態は要求内でのみ保持されるため、このストラテジーはステートレスです。要求が完了すると、**RuntimeEngine** インスタンスは永続的に破棄されます。永続性を使用する場合は、KIE セッションに関連する情報は永続データベースから削除されます。

このストラテジーには以下の特徴があります。

- すべてのリクエストに対して完全に分離されたプロセスエンジンとタスクサービス操作を提供します。
- ファクトは要求期間中のみ保存されるため、完全にステートレスになります。
- これは、高負荷のステートレスプロセスに適しています。この場合、リクエスト間でファクトやタイマーを保持する必要はありません。
- このストラテジーでは、KIE セッションは要求のライフサイクル中にのみ利用でき、要求の最後に破棄されます。
- ストラテジーはコンテキストではありません。リクエストごとの **RuntimeManager** から **RuntimeEngine** のインスタンスを取得する場合は、**Context** インスタンスを考慮する必要はありません。通常は、**EmptyContext.get()** をコンテキストとして使用できますが、null 引数も受け入れ可能です。

プロセスインスタンスごとのストラテジー

このストラテジーは、KIE セッションとプロセスインスタンス間の厳密な関係を維持するように **RuntimeManager** に指示します。各 **KieSession** は、それが属する **ProcessInstance** がアクティブである限り利用できます。
このストラテジーは、ルール評価やプロセスインスタンス間の分離など、プロセスエンジンの高度な機能を使用する最も柔軟なアプローチを提供します。これにより、パフォーマンスを最大化し、同期によって生じる潜在的なボトルネックを減らします。同時に、要求ストラテジーとは異なり、KIE セッションの数は、要求の合計数ではなく、実際のプロセスインスタンス数に減らされます。

このストラテジーには以下の特徴があります。

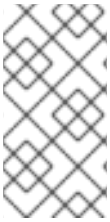
- すべてのプロセスインスタンスに分離を提供します。

- これは、**KieSession** と **ProcessInstance** の間の厳密な関係を維持し、指定の **ProcessInstance** に常に同じ **KieSession** を提供するようにします。
- これは、**KieSession** のライフサイクルを **ProcessInstance** にマージし、プロセスインスタンスの完了時または中止時に両方破棄されます。
- これにより、プロセスインスタンスの範囲内でファクトやタイマーなどのデータのメンテナーンスが可能になります。プロセスインスタンスのみがデータにアクセスできます。
- プロセスインスタンスの **KieSession** を検索して読み込む必要があるため、オーバーヘッドが発生します。
- **KieSession** の使用状況をすべて検証し、他のプロセスインスタンスには使用できません。別のプロセスインスタンスが同じ **KieSession** を使用する場合は、例外が発生します。
- ストラテジーはコンテキストであり、以下のコンテキストインスタンスを許可します。
  - **EmptyContext** または null: プロセスインスタンス ID が利用できないため、プロセスインスタンスを開始するときに使用
  - **ProcessInstanceIdContext**: プロセスインスタンスの作成後に使用します。
  - **CorrelationKeyContext**: **ProcessInstanceIdContext** の代わりに、プロセスインスタンス ID の代わりにカスタム (business) キーを使用します。

66.2.2. ランタイムマネージャーの典型的な使用シナリオ

ランタイムマネージャーの典型的な使用シナリオは、以下の段階で設定されます。

- アプリケーションの起動時に、以下の段階を完了します。
  - **RuntimeManager** インスタンスをビルドし、アプリケーションの有効期間全体を維持します。これはスレッドセーフで、同時にアクセスできます。
- 要求時に、以下のステージを完了します。
  - **RuntimeManager** クラスに設定したストラテジーによって決定される適切なコンテキストインスタンスを使用して、**RuntimeManager** から **RuntimeEngine** を取得します。
  - **RuntimeEngine** から **KieSession** オブジェクトおよび **TaskService** オブジェクトを取得します。
  - **startProcess**、**completeTask** などの操作には、**KieSession** オブジェクトおよび **TaskService** オブジェクトを使用します。
  - 処理が完了したら、**RuntimeManager.disposeRuntimeEngine** メソッドを使用して **RuntimeEngine** を破棄します。
- アプリケーションのシャットダウン時に、以下の段階を完了します。
  - **RuntimeManager** インスタンスを閉じます。



注記

アクティブな JTA トランザクションで **RuntimeManager** から **RuntimeEngine** を取得した場合は、トランザクションの完了時に、(完了ステータスが commit または rollback に関わらず) **RuntimeManager** が自動的に **RuntimeEngine** を破棄するため、最後に **RuntimeEngine** を破棄する必要はありません。

以下の例は、**RuntimeManager** インスタンスをビルドし、そこから (**KieSession** クラスおよび **TaskService** クラスをカプセル化する) **RuntimeEngine** インスタンスを取得する方法を示しています。

RuntimeManager インスタンスをビルドして、RuntimeEngine および KieSession を取得します。

```
    // First, configure the environment to be used by RuntimeManager
    RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
    .newDefaultInMemoryBuilder()
    .addAsset(ResourceFactory.newClassPathResource("BPMN2-ScriptTask.bpmn2"),
ResourceType.BPMN2)
    .get();

    // Next, create the RuntimeManager - in this case the singleton strategy is chosen
    RuntimeManager manager =
RuntimeManagerFactory.Factory.get().newSingletonRuntimeManager(environment);

    // Then get RuntimeEngine from the runtime manager, using an empty context because singleton
does not keep track
    // of runtime engine as there is only one
    RuntimeEngine runtime = manager.getRuntimeEngine(EmptyContext.get());

    // Get the KieSession from the RuntimeEngine - already initialized with all handlers, listeners, and
other requirements
    // configured on the environment
    KieSession ksession = runtimeEngine.getKieSession();

    // Add invocations of the process engine here,
    // for example, ksession.startProcess(processId);

    // Finally, dispose the runtime engine
    manager.disposeRuntimeEngine(runtimeEngine);
```

この例では、**RuntimeManager** クラスおよび **RuntimeEngine** クラスを使用する最も簡単な、または最小限の方法を提供します。これには、以下の特徴があります。

- **KieSession** インスタンスは、**newDefaultInMemoryBuilder** ビルダーを使用してメモリーに作成されます。
- アセットとして追加される 1 つのプロセスが実行に利用できます。
- **TaskService** クラスは **LocalHTWorkItemHandler** インターフェイスを介して設定されて **KieSession** インスタンスに割り当てられ、プロセス内でユーザータスク機能をサポートします。

66.2.3. ランタイム環境設定オブジェクト

RuntimeManager クラスは、ハンドラーの作成、破棄、登録など、内部プロセスエンジンの複雑さをカプセル化します。

また、プロセスエンジンの設定を詳細に制御することもできます。この設定を設定するには、**RuntimeEnvironment** オブジェクトを作成してから、**RuntimeManager** オブジェクトを作成する必要があります。

次の定義は、**RuntimeEnvironment** インターフェイスで利用可能なメソッドを示しています。

RuntimeEnvironment インターフェイスのメソッド

```
 public interface RuntimeEnvironment {

/**
 * Returns <code>KieBase</code> that is to be used by the manager
 * @return
 */
  KieBase getKieBase();

  /**
   * KieSession environment that is to be used to create instances of <code>KieSession</code>
   * @return
   */
  Environment getEnvironment();

  /**
   * KieSession configuration that is to be used to create instances of <code>KieSession</code>
   * @return
   */
  KieSessionConfiguration getConfiguration();

  /**
   * Indicates if persistence is to be used for the KieSession instances
   * @return
   */
  boolean usePersistence();

  /**
   * Delivers a concrete implementation of <code>RegisterableItemsFactory</code> to obtain
handlers and listeners
   * that is to be registered on instances of <code>KieSession</code>
   * @return
   */
  RegisterableItemsFactory getRegisterableItemsFactory();

  /**
   * Delivers a concrete implementation of <code>UserGroupCallback</code> that is to be registered
on instances
   * of <code>TaskService</code> for managing users and groups.
   * @return
   */
  UserGroupCallback getUserGroupCallback();

  /**
   * Delivers a custom class loader that is to be used by the process engine and task service
instances
```

```
     * @return
     */
    ClassLoader getClassLoader();

    /**
     * Closes the environment, permitting closing of all dependent components such as ksession
factories
     */
    void close();
```

66.2.4. ランタイム環境ビルダー

必要なデータが含まれる **RuntimeEnvironment** のインスタンスを作成するには、**RuntimeEnvironmentBuilder** クラスを使用します。このクラスは Fluent API を提供し、事前定義された設定で **RuntimeEnvironment** インスタンスを設定します。

次の定義は、**RuntimeEnvironmentBuilder** インターフェイスのメソッドを示しています。

RuntimeEnvironmentBuilder インターフェイスのメソッド

```
public interface RuntimeEnvironmentBuilder {

 public RuntimeEnvironmentBuilder persistence(boolean persistenceEnabled);

 public RuntimeEnvironmentBuilder entityManagerFactory(Object emf);

 public RuntimeEnvironmentBuilder addAsset(Resource asset, ResourceType type);

 public RuntimeEnvironmentBuilder addEnvironmentEntry(String name, Object value);

 public RuntimeEnvironmentBuilder addConfiguration(String name, String value);

 public RuntimeEnvironmentBuilder knowledgeBase(KieBase kbase);

 public RuntimeEnvironmentBuilder userGroupCallback(UserGroupCallback callback);

 public RuntimeEnvironmentBuilder registerableItemsFactory(RegisterableItemsFactory factory);

 public RuntimeEnvironment get();

 public RuntimeEnvironmentBuilder classLoader(ClassLoader cl);

 public RuntimeEnvironmentBuilder schedulerService(Object globalScheduler);
```

RuntimeEnvironmentBuilderFactory クラスを使用して **RuntimeEnvironmentBuilder** のインスタンスを取得します。設定のない空のインスタンスとともに、ランタイムマネージャーの設定オプションがいくつか事前設定されたビルダーを取得できます。

次の定義は、**RuntimeEnvironmentBuilderFactory** インターフェイスのメソッドを示しています。

RuntimeEnvironmentBuilderFactory インターフェイスのメソッド

```
public interface RuntimeEnvironmentBuilderFactory {

 /**
```

```
    * Provides a completely empty <code>RuntimeEnvironmentBuilder</code> instance to manually
    * set all required components instead of relying on any defaults.
    * @return new instance of <code>RuntimeEnvironmentBuilder</code>
    */
   public RuntimeEnvironmentBuilder newEmptyBuilder();

   /**
    * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is based on:
    * <ul>
    *  <li>DefaultRuntimeEnvironment</li>
    * </ul>
    * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
    *
    * @see DefaultRuntimeEnvironment
    */
   public RuntimeEnvironmentBuilder newDefaultBuilder();

   /**
    * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is based on:
    * <ul>
    *  <li>DefaultRuntimeEnvironment</li>
    * </ul>
    * but does not have persistence for the process engine configured so it will only store process
instances in memory
    * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
    *
    * @see DefaultRuntimeEnvironment
    */
   public RuntimeEnvironmentBuilder newDefaultInMemoryBuilder();

   /**
    * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is based on:
    * <ul>
    *  <li>DefaultRuntimeEnvironment</li>
    * </ul>
    * This method is tailored to work smoothly with KJAR files
    * @param groupId group id of kjar
    * @param artifactId artifact id of kjar
    * @param version version number of kjar
    * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
    *
    * @see DefaultRuntimeEnvironment
    */
   public RuntimeEnvironmentBuilder newDefaultBuilder(String groupId, String artifactId, String
version);

   /**
    * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is based on:
    * <ul>
    *  <li>DefaultRuntimeEnvironment</li>
    * </ul>
    * This method is tailored to work smoothly with KJAR files and use the kbase and ksession
settings in the KJAR
```

```
     * @param groupId group id of kjar
     * @param artifactId artifact id of kjar
     * @param version version number of kjar
     * @param kbaseName name of the kbase defined in kmodule.xml stored in kjar
     * @param ksessionName name of the ksession define in kmodule.xml stored in kjar
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(String groupId, String artifactId, String
version, String kbaseName, String ksessionName);

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * This method is tailored to work smoothly with KJAR files and use the release ID defined in the
KJAR
     * @param releaseId <code>ReleaseId</code> that described the kjar
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(ReleaseId releaseId);

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
    * This method is tailored to work smoothly with KJAR files and use the kbase, ksession, and release
ID settings in the KJAR
     * @param releaseId <code>ReleaseId</code> that described the kjar
     * @param kbaseName name of the kbase defined in kmodule.xml stored in kjar
     * @param ksessionName name of the ksession define in kmodule.xml stored in kjar
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(ReleaseId releaseId, String kbaseName,
String ksessionName);

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * It relies on KieClasspathContainer that requires the presence of kmodule.xml in the META-INF
folder which
     * defines the kjar itself.
     * Expects to use default kbase and ksession from kmodule.
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
```

```
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newClasspathKmoduleDefaultBuilder();

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
    * It relies on KieClasspathContainer that requires the presence of kmodule.xml in the META-INF
folder which
     * defines the kjar itself.
     * @param kbaseName name of the kbase defined in kmodule.xml
     * @param ksessionName name of the ksession define in kmodule.xml
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newClasspathKmoduleDefaultBuilder(String kbaseName,
String ksessionName);
```

ランタイムマネージャーは、KIE セッションと通信するように設定された **RuntimeEngine** オブジェクトの統合コンポーネントとして **TaskService** オブジェクトへのアクセスも提供します。デフォルトビルダーのいずれかを使用する場合は、タスクサービスの以下の設定が表示されます。

- 永続ユニット名は **org.jbpm.persistence.jpa** (プロセスエンジンとタスクサービスの両方) に設定されます。
- ヒューマンタスクハンドラーは KIE セッションに登録されます。
- JPA ベースの履歴ログイベントリスナーは KIE セッションに登録されます。
- ルールタスク評価をトリガーするイベントリスナー (**fireAllRules**) が KIE セッションに登録されます。

66.2.5. ランタイムエンジンのハンドラーおよびリスナーの登録

ランタイムマネージャー API を使用する場合、ランタイムエンジンオブジェクトはプロセスエンジンを表します。

独自のハンドラーまたはリスナーを使用してランタイムエンジンを拡張するには、**RegisterableItemsFactory** インターフェイスを実装し、**RuntimeEnvironmentBuilder.registerableItemsFactory()** メソッドを使用してランタイム環境に追加します。次に、ランタイムマネージャーは、ハンドラーまたはリスナーを、作成するすべてのランタイムエンジンに自動的に追加します。

以下の定義は、**RegisterableItemsFactory** インターフェイスのメソッドを示しています。

RegisterableItemsFactory インターフェイスのメソッド

```
 /**
  * Returns new instances of <code>WorkItemHandler</code> that will be registered on
<code>RuntimeEngine</code>
```

```
   * @param runtime provides <code>RuntimeEngine</code> in case handler need to make use of it
internally
   * @return map of handlers to be registered - in case of no handlers empty map shall be returned.
   */
    Map<String, WorkItemHandler> getWorkItemHandlers(RuntimeEngine runtime);

    /**
   * Returns new instances of <code>ProcessEventListener</code> that will be registered on
<code>RuntimeEngine</code>
   * @param runtime provides <code>RuntimeEngine</code> in case listeners need to make use of it
internally
   * @return list of listeners to be registered - in case of no listeners empty list shall be returned.
   */
    List<ProcessEventListener> getProcessEventListeners(RuntimeEngine runtime);

    /**
   * Returns new instances of <code>AgendaEventListener</code> that will be registered on
<code>RuntimeEngine</code>
   * @param runtime provides <code>RuntimeEngine</code> in case listeners need to make use of it
internally
   * @return list of listeners to be registered - in case of no listeners empty list shall be returned.
   */
    List<AgendaEventListener> getAgendaEventListeners(RuntimeEngine runtime);

    /**
   * Returns new instances of <code>WorkingMemoryEventListener</code> that will be registered on
<code>RuntimeEngine</code>
   * @param runtime provides <code>RuntimeEngine</code> in case listeners need to make use of it
internally
   * @return list of listeners to be registered - in case of no listeners empty list shall be returned.
   */
    List<WorkingMemoryEventListener> getWorkingMemoryEventListeners(RuntimeEngine runtime);
```

プロセスエンジンは、**RegisterableItemsFactory** のデフォルト実装を提供します。これらの実装を拡張して、カスタムハンドラーおよびリスナーを定義できます。

利用可能な実装は役に立ちます。

- **org.jbpm.runtime.manager.impl.SimpleRegisterableItemsFactory**: 最も単純な実装。事前定義されたコンテンツがなく、リフレクションを使用して指定のクラス名に基づいてハンドラーおよびリスナーのインスタンスを生成します。
- **org.jbpm.runtime.manager.impl.DefaultRegisterableItemsFactory**: Simple 実装の拡張。この拡張により、デフォルトのランタイム環境ビルダーと同じデフォルトが導入され、引き続き Simple 実装と同じ機能を提供します。
- **org.jbpm.runtime.manager.impl.cdi.InjectableRegisterableItemsFactory**: CDI 環境向けに調整され、プロデューサーを使用してハンドラーとリスナーを検索する CDI スタイルのアプローチを提供するデフォルト実装の拡張。

66.2.5.1. ファイルを使用したワークアイテムハンドラーの登録

CustomWorkItem.conf ファイルでファイルを定義し、クラスパスにファイルを配置して、ステートレスまたは **KieSession** の状態に依存する簡単なワークアイテムハンドラーを登録することができます。

手順

1. クラスパスのルートの **META-INF** サブディレクトリーに **drools.session.conf** という名前のファイルを作成します。Web アプリケーションの場合、ディレクトリーは **WEB-INF/classes/META-INF** になります。

2. 以下の行を **drools.session.conf** ファイルに追加します。

```
drools.workItemHandlers = CustomWorkItemHandlers.conf
```

3. 同じディレクトリーに **CustomWorkItemHandlers.conf** という名前のファイルを作成します。

4. **CustomWorkItemHandlers.conf** ファイルで、以下の例のように MVEL スタイルを使用してカスタムのワークアイテムハンドラーを定義します。

```
[
  "Log": new org.jbpm.process.instance.impl.demo.SystemOutWorkItemHandler(),
  "WebService": new
org.jbpm.process.workitem.webservice.WebServiceWorkItemHandler(ksession),
  "Rest": new org.jbpm.process.workitem.rest.RESTWorkItemHandler(),
  "Service Task" : new org.jbpm.process.workitem.bpmn2.ServiceTaskHandler(ksession)
]
```

結果

リストしたワークアイテムハンドラーは、アプリケーションがランタイムマネージャー API を使用しているかどうかに関係なく、アプリケーションによって作成されたすべての KIE セッションに登録されます。

66.2.5.2. CDI 環境におけるハンドラーおよびリスナーの登録

アプリケーションがランタイムマネージャー API を使用して、CDI 環境で実行される場合、クラスは専用のプロデューサーインターフェイスを実装し、カスタムワークアイテムハンドラーおよびイベントリスナーをすべてのランタイムエンジンに提供できます。

ワークアイテムハンドラーを作成するには、**WorkItemHandlerProducer** インターフェイスを実装する必要があります。

WorkItemHandlerProducer インターフェイスの定義

```
public interface WorkItemHandlerProducer {

    /**
     * Returns a map of work items (key = work item name, value=  work item handler instance)
     * to be registered on the KieSession
     * <br/>
     * The following parameters are accepted:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     *
     * @param identifier - identifier of the owner - usually RuntimeManager that allows the producer to
filter out
```

```
     * and provide valid instances for given owner
     * @param params - the owner might provide some parameters, usually KieSession, TaskService,
RuntimeManager instances
     * @return map of work item handler instances (recommendation is to always return new instances
when this method is invoked)
     */
    Map<String, WorkItemHandler> getWorkItemHandlers(String identifier, Map<String, Object>
params);
}
```

イベントリスナーを作成するには、**EventListenerProducer** インターフェイスを実装する必要があります。イベントリスナープロデューサーに適切な修飾子にアノテーションを付け、提供されるリスナーのタイプを示します。以下のアノテーションのいずれかを使用します。

- **ProcessEventListener** の **@Process**
- **AgendaEventListener** の **@Agenda**
- **WorkingMemoryEventListener** の場合は **@WorkingMemory**

EventListenerProducer インターフェイスの定義

```
public interface EventListenerProducer<T> {

    /**
     * Returns a list of instances for given (T) type of listeners
     * <br/>
     * The following parameters are accepted:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     * @param identifier - identifier of the owner - usually RuntimeManager that allows the producer to
filter out
     * and provide valid instances for given owner
     * @param params - the owner might provide some parameters, usually KieSession, TaskService,
RuntimeManager instances
     * @return list of listener instances (recommendation is to always return new instances when this
method is invoked)
     */
    List<T> getEventListeners(String identifier, Map<String, Object>  params);
}
```

META-INF サブディレクトリーに **beans.xml** を追加し、これらのインターフェイスの実装を Bean アーカイブとしてパッケージ化します。アプリケーションクラスパスに Bean アーカイブを配置します (例: Web アプリケーションの **WEB-INF/lib**)。CDI ベースのランタイムマネージャーはパッケージを検出し、データストアから作成または読み込みを行うすべての **KieSession** でワークアイテムハンドラーおよびイベントリスナーを登録します。

プロセスエンジンは、ステートフルで高度な操作を有効にするために、特定のパラメーターをプロデューサーに提供します。たとえば、ハンドラーまたはリスナーはパラメーターを使用して、エラーが発生した場合にプロセスエンジンまたはプロセスインスタンスに通知できます。プロセスエンジンは、以下のコンポーネントをパラメーターとして提供します。

- **KieSession**

- **TaskService**
- **RuntimeManager**

さらに、**RuntimeManager** クラスインスタンスの識別子はパラメーターとして提供されます。フィルターを識別子に適用して、この **RuntimeManager** インスタンスがハンドラーとリスナーを受け取るかどうかを決定することができます。

66.3. プロセスエンジンのサービス

プロセスエンジンは、ランタイムマネージャー API の上部で実行される高レベルのサービスを提供します。

サービスは、アプリケーションにプロセスエンジンを組み込む最も便利な方法を提供します。KIE Server は、これらのサービスを内部で使用します。

サービスを使用する場合は、ランタイムマネージャー、ランタイムエンジン、セッション、およびその他のプロセスエンジンエンティティーの独自の処理を実装する必要はありません。ただし、必要に応じてサービスを介して基礎となる **RuntimeManager** オブジェクトにアクセスできます。

注記

サービス API に EJB リモートクライアントを使用する場合、**RuntimeManager** オブジェクトはシリアル化後にクライアント側で正しく動作しないためです。

66.3.1. プロセスエンジンサービスのモジュール

プロセスエンジンサービスはモジュールのセットとして提供されます。これらのモジュールは、フレームワークの依存関係によってグループ化されます。他のモジュールが使用するフレームワークに依存しなくても、適切なモジュールを選択し、これらのモジュールのみを使用できます。

以下のモジュールが利用できます。

- **jbpm-services-api**: API クラスおよびインターフェイスのみ
- **jbpm-kie-services**: フレームワークの依存関係なしで純粋な Java でのサービス API のコード実装
- **jbpm-services-cdi**: コアサービス実装の上に CDI ラッパー
- **jbpm-services-ejb-api**: EJB 要件をサポートするサービス API の拡張機能
- **jbpm-services-ejb-impl**: コアサービス実装の上に EJB ラッパー
- **jbpm-services-ejb-timer**: タイマーイベントやデッドラインなどの時間ベースの操作をサポートする EJB タイマーサービスに基づくスケジューラーサービス
- **jbpm-services-ejb-client**: EJB リモートクライアント実装で現在 Red Hat JBoss EAP のみをサポートします。

66.3.2. デプロイメントサービス

デプロイメントサービスは、プロセスエンジンにユニットをデプロイし、デプロイを解除します。

デプロイメントユニット は、KJAR ファイルの内容を表します。デプロイメントユニットには、プロセ

ス定義、ルール、フォーム、データモデルなどのビジネスアセットが含まれます。ユニットのデプロイ後に、定義したプロセスを実行できます。利用可能なデプロイメントユニットをクエリーすることもできます。

すべてのデプロイメントユニットには、**deploymentUnitId** として知られる一意の ID 文字列 **deploymentId** があります。この識別子を使用して、サービスアクションをデプロイメントユニットに適用することができます。

このサービスの典型的なユースケースでは、複数の KJAR を同時に読み込み、アンロードでき、必要に応じてプロセスを同時に実行できます。

以下のコード例は、デプロイメントサービスの簡単な使用例を示しています。

デプロイメントサービスの使用

```
// Create deployment unit by providing the GAV of the KJAR
DeploymentUnit deploymentUnit = new KModuleDeploymentUnit(GROUP_ID, ARTIFACT_ID,
VERSION);
// Get the deploymentId for the deployed unit
String deploymentId = deploymentUnit.getIdentifier();
// Deploy the unit
deploymentService.deploy(deploymentUnit);
// Retrieve the deployed unit
DeployedUnit deployed = deploymentService.getDeployedUnit(deploymentId);
// Get the runtime manager
RuntimeManager manager = deployed.getRuntimeManager();
```

以下の定義は、**DeploymentService** の完全なインターフェイスを示しています。

DeploymentService インターフェイスの定義

```
public interface DeploymentService {

    void deploy(DeploymentUnit unit);

    void undeploy(DeploymentUnit unit);

    RuntimeManager getRuntimeManager(String deploymentUnitId);

    DeployedUnit getDeployedUnit(String deploymentUnitId);

    Collection<DeployedUnit> getDeployedUnits();

    void activate(String deploymentId);

    void deactivate(String deploymentId);

    boolean isDeployed(String deploymentUnitId);
}
```

66.3.3. 定義サービス

デプロイメントサービスを使用してプロセス定義をデプロイすると、定義サービスは自動的に定義をスキャンし、プロセスを解析して、プロセスエンジンが必要とする情報を抽出します。

定義サービス API を使用して、プロセス定義についての情報を取得できます。サービスは、この情報を BPMN2 プロセス定義から直接抽出します。以下の情報が表示されます。

- ID、名前、説明などの **プロセス定義**
- すべての変数の名前とタイプを含む **Process variables**
- プロセスで使用される **再利用可能なサブプロセス** (存在する場合)
- ドメイン固有のアクティビティーを表す **サービスタスク**
- 割り当て情報を含む **ユーザータスク**
- 入力情報および出力情報を含む **タスクデータ**

以下のコード例は、定義サービスの簡単な使用を示しています。**processID** は、デプロイメントサービスを使用してデプロイ済みの KJAR ファイルのプロセス定義の ID に対応している必要があります。

定義サービスの使用

```
String processId = "org.jbpm.writedocument";

Collection<UserTaskDefinition> processTasks =
bpmn2Service.getTasksDefinitions(deploymentUnit.getIdentifier(), processId);

Map<String, String> processData =
bpmn2Service.getProcessVariables(deploymentUnit.getIdentifier(), processId);

Map<String, String> taskInputMappings =
bpmn2Service.getTaskInputMappings(deploymentUnit.getIdentifier(), processId, "Write a Document"
);
```

定義サービスを使用して、KJAR ファイルを使用せずに BPMN2 準拠の XML コンテンツとして指定する定義をスキャンすることもできます。**buildProcessDefinition** メソッドは、この機能を提供します。

以下の定義は、すべての **DefinitionService** インターフェイスを示しています。

DefinitionService インターフェイスの定義

```
public interface DefinitionService {

    ProcessDefinition buildProcessDefinition(String deploymentId, String bpmn2Content, ClassLoader
classLoader, boolean cache) throws IllegalArgumentException;

    ProcessDefinition getProcessDefinition(String deploymentId, String processId);

    Collection<String> getReusableSubProcesses(String deploymentId, String processId);

    Map<String, String> getProcessVariables(String deploymentId, String processId);

    Map<String, String> getServiceTasks(String deploymentId, String processId);

    Map<String, Collection<String>> getAssociatedEntities(String deploymentId, String processId);

    Collection<UserTaskDefinition> getTasksDefinitions(String deploymentId, String processId);
```

```
    Map<String, String> getTaskInputMappings(String deploymentId, String processId, String
taskName);

    Map<String, String> getTaskOutputMappings(String deploymentId, String processId, String
taskName);

}
```

66.3.4. プロセスサービス

デプロイメントおよび定義のサービスは、プロセスエンジンのプロセスデータを準備します。このデータに基づいたプロセスを実行するには、プロセスサービスを使用します。プロセスサービスは、以下のアクションを含むプロセスエンジン実行環境との対話をサポートします。

- 新規プロセスインスタンスの開始
- 単一トランザクションとしてのプロセスの実行
- イベントのシグナル送信、情報の詳細の取得、変数の値の設定など、既存のプロセスインスタンスの使用
- ワークアイテムの使用

プロセスサービスはコマンドエグゼキューターでもあります。このコマンドを使用して、KIE セッションでコマンドを実行して、その機能を拡張できます。

重要

プロセスサービスはランタイム操作向けに最適化されています。シグナルイベントや変更変数など、プロセスの実行やプロセスインスタンスの変更が必要な場合に使用します。利用可能なプロセスインスタンスを表示するなどの読み取り操作の場合は、ランタイムデータサービスを使用します。

以下のコード例は、プロセスのデプロイおよび実行を示しています。

デプロイメントおよびプロセスサービスを使用したプロセスのデプロイと実行

```
KModuleDeploymentUnit deploymentUnit = new KModuleDeploymentUnit(GROUP_ID,
ARTIFACT_ID, VERSION);

deploymentService.deploy(deploymentUnit);

long processInstanceId = processService.startProcess(deploymentUnit.getIdentifier(), "customtask");

ProcessInstance pi = processService.getProcessInstance(processInstanceId);
```

startProcess メソッドでは、**deploymentId** が最初の引数として想定されます。この引数を使用すると、アプリケーションが複数のデプロイメントを持つ可能性がある場合に、特定のデプロイメントでプロセスを開始できます。

たとえば、異なる KJAR ファイルから同じプロセスの異なるバージョンをデプロイする場合などです。その後、正しい **deploymentId** を使用して必要なバージョンを起動できます。

以下の定義は、完全な **ProcessService** インターフェイスを示しています。

ProcessService インターフェイスの定義

```
public interface ProcessService {

 /**
  * Starts a process with no variables
  *
  * @param deploymentId deployment identifier
  * @param processId process identifier
  * @return process instance IDentifier
  * @throws RuntimeException in case of encountered errors
  * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
  * @throws DeploymentNotActiveException in case the deployment with the given deployment
identifier is not active
  */
 Long startProcess(String deploymentId, String processId);

 /**
  * Starts a process and sets variables
  *
  * @param deploymentId deployment identifier
  * @param processId process identifier
  * @param params process variables
  * @return process instance IDentifier
  * @throws RuntimeException in case of encountered errors
  * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
  * @throws DeploymentNotActiveException in case the deployment with the given deployment
identifier is not active
  */
   Long startProcess(String deploymentId, String processId, Map<String, Object> params);

 /**
  * Starts a process with no variables and assigns a correlation key
  *
  * @param deploymentId deployment identifier
  * @param processId process identifier
  * @param correlationKey correlation key to be assigned to the process instance - must be unique
  * @return process instance IDentifier
  * @throws RuntimeException in case of encountered errors
  * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
     * @throws DeploymentNotActiveException in case the deployment with the given deployment
identifier is not active
  */
 Long startProcess(String deploymentId, String processId, CorrelationKey correlationKey);

 /**
  * Starts a process, sets variables, and assigns a correlation key
  *
  * @param deploymentId deployment identifier
  * @param processId process identifier
  * @param correlationKey correlation key to be assigned to the process instance - must be unique
  * @param params process variables
  * @return process instance IDentifier
```

```
   * @throws RuntimeException in case of encountered errors
   * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
     * @throws DeploymentNotActiveException in case the deployment with the given deployment
identifier is not active
   */
    Long startProcess(String deploymentId, String processId, CorrelationKey correlationKey,
Map<String, Object> params);

   /**
    * Run a process that is designed to start and finish in a single transaction.
    * This method starts the process and returns when the process completes.
    * It returns the state of process variables at the outcome of the process
    *
    * @param deploymentId deployment identifier for the KJAR file of the process
    * @param processId process identifier
    * @param params process variables
    * @return the state of process variables at the end of the process
    */
     Map<String, Object> computeProcessOutcome(String deploymentId, String processId,
Map<String, Object> params);

    /**
     * Starts a process at the listed nodes, instead of the normal starting point.
     * This method can be used for restarting a process that was aborted. However,
     * it does not restore the context of a previous process instance. You must
     * supply all necessary variables when calling this method.
     * This method does not guarantee that the process is started in a valid state.
     *
     * @param deploymentId deployment identifier
     * @param processId process identifier
     * @param params process variables
     * @param nodeIds list of BPMN node identifiers where the process must start
     * @return process instance IDentifier
     * @throws RuntimeException in case of encountered errors
     * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
     * @throws DeploymentNotActiveException in case the deployment with the given deployment
identifier is not active
     */
    Long startProcessFromNodeIds(String deploymentId, String processId, Map<String, Object>
params, String... nodeIds);

    /**
     * Starts a process at the listed nodes, instead of the normal starting point,
     * and assigns a correlation key.
     * This method can be used for restarting a process that was aborted. However,
     * it does not restore the context of a previous process instance. You must
     * supply all necessary variables when calling this method.
     * This method does not guarantee that the process is started in a valid state.
     *
     * @param deploymentId deployment identifier
     * @param processId process identifier
     * @param key correlation key (must be unique)
     * @param params process variables
     * @param nodeIds list of BPMN node identifiers where the process must start.
```

```
     * @return process instance IDentifier
     * @throws RuntimeException in case of encountered errors
     * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
     * @throws DeploymentNotActiveException in case the deployment with the given deployment
identifier is not active
     */
    Long startProcessFromNodeIds(String deploymentId, String processId, CorrelationKey key,
Map<String, Object> params, String... nodeIds);

    /**
     * Aborts the specified process
     *
     * @param processInstanceId process instance unique identifier
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    void abortProcessInstance(Long processInstanceId);

    /**
     * Aborts the specified process
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId process instance unique identifier
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    void abortProcessInstance(String deploymentId, Long processInstanceId);

    /**
   * Aborts all specified processes
   *
   * @param processInstanceIds list of process instance unique identifiers
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
   */
    void abortProcessInstances(List<Long> processInstanceIds);

    /**
     * Aborts all specified processes
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceIds list of process instance unique identifiers
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    void abortProcessInstances(String deploymentId, List<Long> processInstanceIds);

    /**
   * Signals an event to a single process instance
   *
   * @param processInstanceId the process instance unique identifier
```

```
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
   */
    void signalProcessInstance(Long processInstanceId, String signalName, Object event);

    /**
     * Signals an event to a single process instance
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId the process instance unique identifier
     * @param signalName the ID of the signal in the process
     * @param event the event object to be passed with the event
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    void signalProcessInstance(String deploymentId, Long processInstanceId, String signalName,
Object event);

    /**
   * Signal an event to a list of process instances
   *
   * @param processInstanceIds list of process instance unique identifiers
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
   */
    void signalProcessInstances(List<Long> processInstanceIds, String signalName, Object event);

    /**
     * Signal an event to a list of process instances
     *
     * @param deploymentId deployment to which the process instances belong
     * @param processInstanceIds list of process instance unique identifiers
     * @param signalName the ID of the signal in the process
     * @param event the event object to be passed with the event
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    void signalProcessInstances(String deploymentId, List<Long> processInstanceIds, String
signalName, Object event);

  /**
     * Signal an event to a single process instance by correlation key
     *
     * @param correlationKey the unique correlation key of the process instance
     * @param signalName the ID of the signal in the process
     * @param event the event object to be passed in with the event
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given key was
```

```
not found
     */
    void signalProcessInstanceByCorrelationKey(CorrelationKey correlationKey, String signalName,
Object event);

  /**
   * Signal an event to a single process instance by correlation key
   *
   * @param deploymentId deployment to which the process instance belongs
   * @param correlationKey the unique correlation key of the process instance
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed in with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given key was
not found
   */
  void signalProcessInstanceByCorrelationKey(String deploymentId, CorrelationKey correlationKey,
String signalName, Object event);

  /**
   * Signal an event to given list of correlation keys
   *
   * @param correlationKeys list of unique correlation keys of process instances
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed in with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with one of the given
keys was not found
   */
  void signalProcessInstancesByCorrelationKeys(List<CorrelationKey> correlationKeys, String
signalName, Object event);

  /**
   * Signal an event to given list of correlation keys
   *
   * @param deploymentId deployment to which the process instances belong
   * @param correlationKeys list of unique correlation keys of process instances
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed in with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with one of the given
keys was not found
   */
  void signalProcessInstancesByCorrelationKeys(String deploymentId, List<CorrelationKey>
correlationKeys, String signalName, Object event);

    /**
     * Signal an event to a any process instance that listens to a given signal and belongs to a given
deployment
     *
     * @param deployment identifier of the deployment
     * @param signalName the ID of the signal in the process
     * @param event the event object to be passed with the event
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     */
    void signalEvent(String deployment, String signalName, Object event);
```

```
    /**
   * Returns process instance information. Will return null if no
   * active process with the ID is found
   *
   * @param processInstanceId The process instance unique identifier
   * @return Process instance information
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   */
    ProcessInstance getProcessInstance(Long processInstanceId);

    /**
     * Returns process instance information. Will return null if no
     * active process with the ID is found
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId The process instance unique identifier
     * @return Process instance information
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     */
    ProcessInstance getProcessInstance(String deploymentId, Long processInstanceId);

    /**
   * Returns process instance information. Will return null if no
   * active process with that correlation key is found
   *
   * @param correlationKey correlation key assigned to the process instance
   * @return Process instance information
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   */
    ProcessInstance getProcessInstance(CorrelationKey correlationKey);

    /**
     * Returns process instance information. Will return null if no
     * active process with that correlation key is found
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param correlationKey correlation key assigned to the process instance
     * @return Process instance information
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     */
    ProcessInstance getProcessInstance(String deploymentId, CorrelationKey correlationKey);

    /**
   * Sets a process variable.
   * @param processInstanceId The process instance unique identifier
   * @param variableId The variable ID to set
   * @param value The variable value
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
  not found
   */
    void setProcessVariable(Long processInstanceId, String variableId, Object value);

    /**
     * Sets a process variable.
```

```
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId The process instance unique identifier
     * @param variableId The variable id to set.
     * @param value The variable value.
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    void setProcessVariable(String deploymentId, Long processInstanceId, String variableId, Object
value);

    /**
   * Sets process variables.
   *
   * @param processInstanceId The process instance unique identifier
   * @param variables map of process variables (key = variable name, value = variable value)
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
   */
    void setProcessVariables(Long processInstanceId, Map<String, Object> variables);

    /**
     * Sets process variables.
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId The process instance unique identifier
     * @param variables map of process variables (key = variable name, value = variable value)
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    void setProcessVariables(String deploymentId, Long processInstanceId, Map<String, Object>
variables);

    /**
   * Gets a process instance variable.
   *
   * @param processInstanceId the process instance unique identifier
   * @param variableName the variable name to get from the process
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
   */
    Object getProcessInstanceVariable(Long processInstanceId, String variableName);

    /**
     * Gets a process instance variable.
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId the process instance unique identifier
     * @param variableName the variable name to get from the process
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
```

```
    */
    Object getProcessInstanceVariable(String deploymentId, Long processInstanceId, String
variableName);

  /**
   * Gets a process instance variable values.
   *
   * @param processInstanceId The process instance unique identifier
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
   */
  Map<String, Object> getProcessInstanceVariables(Long processInstanceId);

  /**
     * Gets a process instance variable values.
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId The process instance unique identifier
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    Map<String, Object> getProcessInstanceVariables(String deploymentId, Long processInstanceId);

  /**
   * Returns all signals available in current state of given process instance
   *
   * @param processInstanceId process instance ID
   * @return list of available signals or empty list if no signals are available
   */
    Collection<String> getAvailableSignals(Long processInstanceId);

    /**
     * Returns all signals available in current state of given process instance
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId process instance ID
     * @return list of available signals or empty list if no signals are available
     */
    Collection<String> getAvailableSignals(String deploymentId, Long processInstanceId);

  /**
   * Completes the specified WorkItem with the given results
   *
   * @param id workItem ID
   * @param results results of the workItem
   * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws WorkItemNotFoundException in case a work item with the given ID was not found
   */
    void completeWorkItem(Long id, Map<String, Object> results);

    /**
     * Completes the specified WorkItem with the given results
     *
     * @param deploymentId deployment to which the process instance belongs
```

```
     * @param processInstanceId process instance ID to which the work item belongs
     * @param id workItem ID
     * @param results results of the workItem
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws WorkItemNotFoundException in case a work item with the given ID was not found
     */
    void completeWorkItem(String deploymentId, Long processInstanceId, Long id, Map<String,
Object> results);

    /**
     * Abort the specified workItem
     *
     * @param id workItem ID
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws WorkItemNotFoundException in case a work item with the given ID was not found
     */
    void abortWorkItem(Long id);

    /**
     * Abort the specified workItem
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId process instance ID to which the work item belongs
     * @param id workItem ID
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws WorkItemNotFoundException in case a work item with the given ID was not found
     */
    void abortWorkItem(String deploymentId, Long processInstanceId, Long id);

    /**
     * Returns the specified workItem
     *
     * @param id workItem ID
     * @return The specified workItem
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws WorkItemNotFoundException in case a work item with the given ID was not found
     */
    WorkItem getWorkItem(Long id);

    /**
     * Returns the specified workItem
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId process instance ID to which the work item belongs
     * @param id workItem ID
     * @return The specified workItem
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws WorkItemNotFoundException in case a work item with the given ID was not found
     */
    WorkItem getWorkItem(String deploymentId, Long processInstanceId, Long id);

    /**
     * Returns active work items by process instance ID.
     *
     * @param processInstanceId process instance ID
     * @return The list of active workItems for the process instance
```

```
     * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    List<WorkItem> getWorkItemByProcessInstance(Long processInstanceId);

    /**
     * Returns active work items by process instance ID.
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId process instance ID
     * @return The list of active workItems for the process instance
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given ID was
not found
     */
    List<WorkItem> getWorkItemByProcessInstance(String deploymentId, Long processInstanceId);


    /**
     * Executes the provided command on the underlying command executor (usually KieSession)
     * @param deploymentId deployment identifier
     * @param command actual command for execution
     * @return results of the command execution
     * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
     * @throws DeploymentNotActiveException in case the deployment with the given deployment
identifier is not active for restricted commands (for example, start process)
     */
    public <T> T execute(String deploymentId, Command<T> command);

    /**
     * Executes the provided command on the underlying command executor (usually KieSession)
     * @param deploymentId deployment identifier
     * @param context context implementation to be used to get the runtime engine
     * @param command actual command for execution
     * @return results of the command execution
     * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
     * @throws DeploymentNotActiveException in case the deployment with the given deployment
identifier is not active for restricted commands (for example, start process)
     */
    public <T> T execute(String deploymentId, Context<?> context, Command<T> command);

}
```

66.3.5. ランタイムデータサービス

ランタイムデータサービスを使用して、開始したプロセスインスタンスや実行ノードインスタンスなど、プロセスに関するすべてのランタイム情報を取得できます。

たとえば、リストベースの UI をビルドして、ランタイムデータサービスによって提供される情報に基づいて、プロセス定義、プロセスインスタンス、特定ユーザーのタスクなどのデータを表示できます。

このサービスは、必要な情報をすべて提供しながら、できるだけ効率的になるように最適化されます。

以下の例は、このサービスのさまざまな使用方法を示しています。

全プロセス定義の取得

```
Collection definitions = runtimeDataService.getProcesses(new QueryContext());
```

アクティブなプロセスインスタンスの取得

```
Collection<processinstancedesc> instances = runtimeDataService.getProcessInstances(new
QueryContext());
```

特定のプロセスインスタンスでのアクティブなノードの取得

```
Collection<nodeinstancedesc> instances =
runtimeDataService.getProcessInstanceHistoryActive(processInstanceId, new QueryContext());
```

ユーザー john に割り当てられたタスクの取得

```
List<tasksummary> taskSummaries =
runtimeDataService.getTasksAssignedAsPotentialOwner("john", new QueryFilter(0, 10));
```

ランタイムデータサービスメソッドは、**QueryContext** および **QueryFilter** の 2 つの重要なパラメーターをサポートします。**QueryFilter** は **QueryContext** の拡張機能です。これらのパラメーターを使用して、結果セット、ページネーション、ソート、順序付けを管理します。ユーザータスクを検索する際に、それらを使用して追加のフィルターリングを適用することもできます。

以下の定義は、**RuntimeDataService** インターフェイスのメソッドを示しています。

RuntimeDataService インターフェイスの定義

```
public interface RuntimeDataService {
 /**
  * Represents type of node instance log entries
  *
  */
 enum EntryType {

        START(0),
        END(1),
        ABORTED(2),
        SKIPPED(3),
        OBSOLETE(4),
        ERROR(5);
 }

    // Process instance information

    /**
     * Returns a list of process instance descriptions
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the available process
instances
     */
```

```
    Collection<ProcessInstanceDesc> getProcessInstances(QueryContext queryContext);

    /**
     * Returns a list of all process instance descriptions with the given statuses and initiated by
<code>initiator</code>
     * @param states list of possible state (int) values that the {@link ProcessInstance} can have
     * @param initiator the initiator of the {@link ProcessInstance}
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
match
     *         the given criteria (states and initiator)
     */
    Collection<ProcessInstanceDesc> getProcessInstances(List<Integer> states, String initiator,
QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions found for the given process ID and statuses and
initiated by <code>initiator</code>
     * @param states list of possible state (int) values that the {@link ProcessInstance} can have
     * @param processId ID of the {@link Process} (definition) used when starting the process instance
     * @param initiator initiator of the {@link ProcessInstance}
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
match
     *         the given criteria (states, processId, and initiator)
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByProcessId(List<Integer> states, String
processId, String initiator, QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions found for the given process name and statuses
and initiated by <code>initiator</code>
     * @param states list of possible state (int) values that the {@link ProcessInstance} can have
     * @param processName name (not ID) of the {@link Process} (definition) used when starting the
process instance
     * @param initiator initiator of the {@link ProcessInstance}
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
match
     *         the given criteria (states, processName and initiator)
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByProcessName(List<Integer> states,
String processName, String initiator, QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions found for the given deployment ID and statuses
     * @param deploymentId deployment ID of the runtime
     * @param states list of possible state (int) values that the {@link ProcessInstance} can have
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
match
     *         the given criteria (deploymentId and states)
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByDeploymentId(String deploymentId,
List<Integer> states, QueryContext queryContext);
```

```
    /**
     * Returns process instance descriptions found for the given processInstanceId. If no descriptions
are found, null is returned. At the same time, the method
     * fetches all active tasks (in status: Ready, Reserved, InProgress) to provide the information about
what user task is keeping each instance
     * and who owns the task (if the task is already claimed by a user)
     * @param processInstanceId ID of the process instance to be fetched
     * @return process instance information, in the form of a {@link ProcessInstanceDesc} instance
     */
    ProcessInstanceDesc getProcessInstanceById(long processInstanceId);

    /**
     * Returns the active process instance description found for the given correlation key. If none is
found, returns null. At the same time it
     * fetches all active tasks (in status: Ready, Reserved, InProgress) to provide information about
which user task is keeping each instance
     * and who owns the task (if the task is already claimed by a user)
     * @param correlationKey correlation key assigned to the process instance
     * @return process instance information, in the form of a {@link ProcessInstanceDesc} instance
     */
    ProcessInstanceDesc getProcessInstanceByCorrelationKey(CorrelationKey correlationKey);

    /**
     * Returns process instances descriptions (regardless of their states) found for the given correlation
key. If no descriptions are found, an empty list is returned
     * This query uses 'LIKE' to match correlation keys so it accepts partial keys. Matching
     * is performed based on a 'starts with' criterion
     * @param correlationKey correlation key assigned to the process instance
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
match
     *         the given correlation key
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByCorrelationKey(CorrelationKey
correlationKey, QueryContext queryContext);

    /**
     * Returns process instance descriptions, filtered by their states, that were found for the given
correlation key. If none are found, returns an empty list
     * This query uses 'LIKE' to match correlation keys so it accepts partial keys. Matching
     * is performed based on a 'starts with' criterion
     * @param correlationKey correlation key assigned to process instance
     * @param states list of possible state (int) values that the {@link ProcessInstance} can have
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
match
     *         the given correlation key
     */
    Collection<ProcessInstanceDesc>
getProcessInstancesByCorrelationKeyAndStatus(CorrelationKey correlationKey, List<Integer> states,
QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions found for the given process definition ID
     * @param processDefId ID of the process definition
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
match
```

```
     *         the given criteria (deploymentId and states)
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByProcessDefinition(String processDefId,
QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions found for the given process definition ID, filtered
by state
     * @param processDefId ID of the process definition
     * @param states list of possible state (int) values that the {@link ProcessInstance} can have
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
match
     *         the given criteria (deploymentId and states)
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByProcessDefinition(String processDefId,
List<Integer> states, QueryContext queryContext);

    /**
     * Returns process instance descriptions that match process instances that have the given variable
defined, filtered by state
     * @param variableName name of the variable that process instance should have
     * @param states list of possible state (int) values that the {@link ProcessInstance} can have. If
null, returns only active instances
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
have the given variable defined
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByVariable(String variableName,
List<Integer> states, QueryContext queryContext);

    /**
     * Returns process instance descriptions that match process instances that have the given variable
defined and the value of the variable matches the given variableValue
     * @param variableName name of the variable that process instance should have
     * @param variableValue value of the variable to match
     * @param states list of possible state (int) values that the {@link ProcessInstance} can have. If
null, returns only active instances
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process instances that
have the given variable defined with the given value
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByVariableAndValue(String variableName,
String variableValue, List<Integer> states, QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions that have the specified parent
     * @param parentProcessInstanceId ID of the parent process instance
     * @param states list of possible state (int) values that the {@link ProcessInstance} can have. If
null, returns only active instances
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the available process
instances
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByParent(Long parentProcessInstanceId,
List<Integer> states, QueryContext queryContext);
```

```
    /**
     * Returns a list of process instance descriptions that are subprocesses of the specified process, or
subprocesses of those subprocesses, and so on. The list includes the full hierarchy of subprocesses
under the specified parent process
     * @param processInstanceId ID of the parent process instance
     * @return list of {@link ProcessInstanceDesc} instances representing the full hierarchy of this
process
     */
    Collection<ProcessInstanceDesc>
getProcessInstancesWithSubprocessByProcessInstanceId(Long processInstanceId, List<Integer>
states, QueryContext queryContext);

    // Node and Variable instance information

    /**
     * Returns the active node instance descriptor for the given work item ID, if the work item exists and
is active
     * @param workItemId identifier of the work item
     * @return NodeInstanceDesc for work item if it exists and is still active, otherwise null is returned
     */
    NodeInstanceDesc getNodeInstanceForWorkItem(Long workItemId);

    /**
     * Returns a trace of all active nodes for the given process instance ID
     * @param processInstanceId unique identifier of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return
     */
    Collection<NodeInstanceDesc> getProcessInstanceHistoryActive(long processInstanceId,
QueryContext queryContext);

    /**
     * Returns a trace of all executed (completed) nodes for the given process instance ID
     * @param processInstanceId unique identifier of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return
     */
    Collection<NodeInstanceDesc> getProcessInstanceHistoryCompleted(long processInstanceId,
QueryContext queryContext);

    /**
     * Returns a complete trace of all executed (completed) and active nodes for the given process
instance ID
     * @param processInstanceId unique identifier of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return {@link NodeInstance} information, in the form of a list of {@link NodeInstanceDesc}
instances,
     *         that come from a process instance that matches the given criteria (deploymentId, processId)
     */
    Collection<NodeInstanceDesc> getProcessInstanceFullHistory(long processInstanceId,
QueryContext queryContext);

    /**
     * Returns a complete trace of all events of the given type (START, END, ABORTED, SKIPPED,
OBSOLETE or ERROR) for the given process instance
```

```
     * @param processInstanceId unique identifier of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @param type type of events to be returned (START, END, ABORTED, SKIPPED, OBSOLETE or
ERROR). To return all events, use {@link #getProcessInstanceFullHistory(long, QueryContext)}
     * @return collection of node instance descriptions
     */
    Collection<NodeInstanceDesc> getProcessInstanceFullHistoryByType(long processInstanceId,
EntryType type, QueryContext queryContext);


    /**
     * Returns a trace of all nodes for the given node types and process instance ID
     * @param processInstanceId unique identifier of the process instance
     * @param nodeTypes list of node types to filter nodes of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return collection of node instance descriptions
     */
    Collection<NodeInstanceDesc> getNodeInstancesByNodeType(long processInstanceId,
List<String> nodeTypes, QueryContext queryContext);

    /**
     * Returns a trace of all nodes for the given node types and correlation key
     * @param correlationKey correlation key
     * @param states list of states
     * @param nodeTypes list of node types to filter nodes of process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return collection of node instance descriptions
     */
    Collection<NodeInstanceDesc> getNodeInstancesByCorrelationKeyNodeType(CorrelationKey
correlationKey,  List<Integer> states, List<String> nodeTypes, QueryContext queryContext);


    /**
     * Returns a collection of all process variables and their current values for the given process
instance
     * @param processInstanceId process instance ID
     * @return information about variables in the specified process instance,
     *         represented by a list of {@link VariableDesc} instances
     */
    Collection<VariableDesc> getVariablesCurrentState(long processInstanceId);

    /**
     * Returns a collection of changes to the given variable within the scope of a process instance
     * @param processInstanceId unique identifier of the process instance
     * @param variableId ID of the variable
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return information about the variable with the given ID in the specified process instance,
     *         represented by a list of {@link VariableDesc} instances
     */
    Collection<VariableDesc> getVariableHistory(long processInstanceId, String variableId,
QueryContext queryContext);


    // Process information
```

```
    /**
     * Returns a list of process definitions for the given deployment ID
     * @param deploymentId deployment ID of the runtime
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessDefinition} instances representing processes that match
     *         the given criteria (deploymentId)
     */
    Collection<ProcessDefinition> getProcessesByDeploymentId(String deploymentId, QueryContext
queryContext);

    /**
     * Returns a list of process definitions that match the given filter
     * @param filter regular expression
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessDefinition} instances with a name or ID that matches the given
regular expression
     */
    Collection<ProcessDefinition> getProcessesByFilter(String filter, QueryContext queryContext);

    /**
     * Returns all process definitions available
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of all available processes, in the form a of a list of {@link ProcessDefinition}
instances
     */
    Collection<ProcessDefinition> getProcesses(QueryContext queryContext);

    /**
     * Returns a list of process definition identifiers for the given deployment ID
     * @param deploymentId deployment ID of the runtime
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of all available process id's for a particular deployment/runtime
     */
    Collection<String> getProcessIds(String deploymentId, QueryContext queryContext);

    /**
     * Returns process definitions for the given process ID regardless of the deployment
     * @param processId ID of the process
     * @return collection of {@link ProcessDefinition} instances representing the {@link Process}
     *         with the specified process ID
     */
    Collection<ProcessDefinition> getProcessesById(String processId);

    /**
     * Returns the process definition for the given deployment and process identifiers
     * @param deploymentId ID of the deployment (runtime)
     * @param processId ID of the process
     * @return {@link ProcessDefinition} instance, representing the {@link Process}
     *         that is present in the specified deployment with the specified process ID
     */
    ProcessDefinition getProcessesByDeploymentIdProcessId(String deploymentId, String processId);

  // user task query operations

  /**
   * Return a task by its workItemId
```

```
     * @param workItemId
     * @return @{@link UserTaskInstanceDesc} task
     */
      UserTaskInstanceDesc getTaskByWorkItemId(Long workItemId);

    /**
     * Return a task by its taskId
     * @param taskId
     * @return @{@link UserTaskInstanceDesc} task
     */
    UserTaskInstanceDesc getTaskById(Long taskId);

    /**
     * Return a task by its taskId with SLA data if the withSLA param is true
     * @param taskId
     * @param withSLA
     * @return @{@link UserTaskInstanceDesc} task
     */
    UserTaskInstanceDesc getTaskById(Long taskId, boolean withSLA);

    /**
     * Return a list of assigned tasks for a Business Administrator user. Business
     * administrators play the same role as task stakeholders but at task type
     * level. Therefore, business administrators can perform the exact same
     * operations as task stakeholders. Business administrators can also observe
     * the progress of notifications
     *
     * @param userId identifier of the Business Administrator user
     * @param filter filter for the list of assigned tasks
     * @return list of @{@link TaskSummary} task summaries
     */
    List<TaskSummary> getTasksAssignedAsBusinessAdministrator(String userId, QueryFilter filter);

    /**
        * Return a list of assigned tasks for a Business Administrator user for with one of the listed
        * statuses
        * @param userId identifier of the Business Administrator user
        * @param statuses the statuses of the tasks to return
        * @param filter filter for the list of assigned tasks
        * @return list of @{@link TaskSummary} task summaries
        */
    List<TaskSummary> getTasksAssignedAsBusinessAdministratorByStatus(String userId,
    List<Status> statuses, QueryFilter filter);

    /**
     * Return a list of tasks that a user is eligible to own
     *
     * @param userId identifier of the user
     * @param filter filter for the list of tasks
     * @return list of @{@link TaskSummary} task summaries
     */
    List<TaskSummary> getTasksAssignedAsPotentialOwner(String userId, QueryFilter filter);

    /**
     * Return a list of tasks the user or user groups are eligible to own
     *
```

```
   * @param userId identifier of the user
   * @param groupIds a list of identifiers of the groups
   * @param filter filter for the list of tasks
   * @return list of @{@link TaskSummary} task summaries
   */
  List<TaskSummary> getTasksAssignedAsPotentialOwner(String userId, List<String> groupIds,
QueryFilter filter);

  /**
   * Return a list of tasks the user is eligible to own and that are in one of the listed
   * statuses
   *
   * @param userId identifier of the user
   * @param status filter for the task statuses
   * @param filter filter for the list of tasks
   * @return list of @{@link TaskSummary} task summaries
   */
  List<TaskSummary> getTasksAssignedAsPotentialOwnerByStatus(String userId, List<Status>
status, QueryFilter filter);

  /**
   * Return a list of tasks the user or groups are eligible to own and that are in one of the listed
   * statuses
   * @param userId identifier of the user
   * @param groupIds filter for the identifiers of the groups
   * @param status filter for the task statuses
   * @param filter filter for the list of tasks
   * @return list of @{@link TaskSummary} task summaries
   */
  List<TaskSummary> getTasksAssignedAsPotentialOwner(String userId, List<String> groupIds,
List<Status> status, QueryFilter filter);

  /**
   * Return a list of tasks the user is eligible to own, that are in one of the listed
   * statuses, and that have an expiration date starting at <code>from</code>. Tasks that do not have
expiration date set
   * will also be included in the result set
   *
   * @param userId identifier of the user
   * @param status filter for the task statuses
   * @param from earliest expiration date for the tasks
   * @param filter filter for the list of tasks
   * @return list of @{@link TaskSummary} task summaries
   */
  List<TaskSummary> getTasksAssignedAsPotentialOwnerByExpirationDateOptional(String userId,
List<Status> status, Date from, QueryFilter filter);

  /**
   * Return a list of tasks the user has claimed, that are in one of the listed
   * statuses, and that have an expiration date starting at <code>from</code>. Tasks that do not have
expiration date set
   * will also be included in the result set
   *
   * @param userId identifier of the user
   * @param strStatuses filter for the task statuses
   * @param from earliest expiration date for the tasks
```

```
    * @param filter filter for the list of tasks
    * @return list of @{@link TaskSummary} task summaries
    */
   List<TaskSummary> getTasksOwnedByExpirationDateOptional(String userId, List<Status>
  strStatuses, Date from, QueryFilter filter);

   /**
    * Return a list of tasks the user has claimed
    *
    * @param userId identifier of the user
    * @param filter filter for the list of tasks
    * @return list of @{@link TaskSummary} task summaries
    */
   List<TaskSummary> getTasksOwned(String userId, QueryFilter filter);

   /**
    * Return a list of tasks the user has claimed with one of the listed
    * statuses
    *
    * @param userId identifier of the user
    * @param status filter for the task statuses
    * @param filter filter for the list of tasks
    * @return list of @{@link TaskSummary} task summaries
    */
   List<TaskSummary> getTasksOwnedByStatus(String userId, List<Status> status, QueryFilter filter);

   /**
    * Get a list of tasks the Process Instance is waiting on
    *
    * @param processInstanceId identifier of the process instance
    * @return list of task identifiers
    */
   List<Long> getTasksByProcessInstanceId(Long processInstanceId);

   /**
    * Get filter for the tasks the Process Instance is waiting on that are in one of the
    * listed statuses
    *
    * @param processInstanceId identifier of the process instance
    * @param status filter for the task statuses
    * @param filter filter for the list of tasks
    * @return list of @{@link TaskSummary} task summaries
    */
   List<TaskSummary> getTasksByStatusByProcessInstanceId(Long processInstanceId, List<Status>
  status, QueryFilter filter);

     /**
    * Get a list of task audit logs for all tasks owned by the user, applying a query filter to the list of tasks
    *
    *
    * @param userId identifier of the user that owns the tasks
    * @param filter filter for the list of tasks
    * @return list of @{@link AuditTask} task audit logs
    */
     List<AuditTask> getAllAuditTask(String userId, QueryFilter filter);
```

```
    /**
   * Get a list of task audit logs for all tasks that are active and owned by the user, applying a query
filter to the list of tasks
   *
   * @param userId identifier of the user that owns the tasks
   * @param filter filter for the list of tasks
   * @return list of @{@link AuditTask} audit tasks
   */
    List<AuditTask> getAllAuditTaskByStatus(String userId, QueryFilter filter);

    /**
   * Get a list of task audit logs for group tasks (actualOwner == null) for the user, applying a query
filter to the list of tasks
   *
   * @param userId identifier of the user that is associated with the group tasks
   * @param filter filter for the list of tasks
   * @return list of @{@link AuditTask} audit tasks
   */
    List<AuditTask> getAllGroupAuditTask(String userId, QueryFilter filter);


    /**
   * Get a list of task audit logs for tasks that are assigned to a Business Administrator user, applying a
query filter to the list of tasks
   *
   * @param userId identifier of the Business Administrator user
   * @param filter filter for the list of tasks
   * @return list of @{@link AuditTask} audit tasks
   */
    List<AuditTask> getAllAdminAuditTask(String userId, QueryFilter filter);

    /**
     * Gets a list of task events for the given task
     * @param taskId identifier of the task
     * @param filter for the list of events
   * @return list of @{@link TaskEvent} task events
     */
    List<TaskEvent> getTaskEvents(long taskId, QueryFilter filter);

    /**
     * Query on {@link TaskSummary} instances
     * @param userId the user associated with the tasks queried
     * @return {@link TaskSummaryQueryBuilder} used to create the query
     */
    TaskSummaryQueryBuilder taskSummaryQuery(String userId);

    /**
     * Gets a list of {@link TaskSummary} instances for tasks that define a given variable
     * @param userId the ID of the user associated with the tasks
     * @param variableName the name of the task variable
     * @param statuses the list of statuses that the task can have
     * @param queryContext the query context
     * @return a {@link List} of {@link TaskSummary} instances
     */
    List<TaskSummary> getTasksByVariable(String userId, String variableName, List<Status>
statuses, QueryContext queryContext);
```

```
    /**
     * Gets a list of {@link TaskSummary} instances for tasks that define a given variable and the
variable is set to the given value
     * @param userId the ID of the user associated with the tasks
     * @param variableName the name of the task variable
     * @param variableValue the value of the task variable
     * @param statuses the list of statuses that the task can have
     * @param context the query context
     * @return a {@link List} of {@link TaskSummary} instances
     */
    List<TaskSummary> getTasksByVariableAndValue(String userId, String variableName, String
variableValue, List<Status> statuses, QueryContext context);

}
```

66.3.6. ユーザータスクサービス

ユーザータスクサービスは、個別のタスクの完全なライフサイクルに対応し、サービスを使用して開始から終了までユーザータスクを管理できます。

タスククエリーは、ユーザータスクサービスの一部ではありません。ランタイムデータサービスを使用して、タスクのクエリーを行います。以下を含む、1 つのタスクでスコープ設定された操作にユーザータスクサービスを使用します。

- 選択したプロパティーの変更
- タスク変数へのアクセス
- タスク割り当てへのアクセス
- タスクコメントへのアクセス

ユーザータスクサービスは、コマンドエグゼキューターでもあります。カスタムタスクコマンドを実行するのに使用できます。

以下の例は、プロセスを開始して、プロセスのタスクを操作する例を示しています。

プロセスの開始、およびこのプロセスのユーザータスクとの対話

```
long processInstanceId =
processService.startProcess(deployUnit.getIdentifier(), "org.jbpm.writedocument");

List<Long> taskIds =
runtimeDataService.getTasksByProcessInstanceId(processInstanceId);

Long taskId = taskIds.get(0);

userTaskService.start(taskId, "john");
UserTaskInstanceDesc task = runtimeDataService.getTaskById(taskId);

Map<String, Object> results = new HashMap<String, Object>();
results.put("Result", "some document data");
userTaskService.complete(taskId, "john", results);
```

66.3.7. quartz ベースのタイマーサービス

プロセスエンジンは、Quartz を使用してクラスター対応のタイマーサービスを提供します。サービスを使用すると、いつでも KIE セッションを破棄または読み込むことができます。サービスは、各タイマーを適切に実行するために KIE セッションがアクティブである期間を管理できます。

以下の例は、クラスター環境用の基本的な Quartz 設定ファイルを示しています。

クラスター環境の quartz 設定ファイル

```
#============================================================================

# Configure Main Scheduler Properties
#============================================================================


org.quartz.scheduler.instanceName = jBPMClusteredScheduler
org.quartz.scheduler.instanceId = AUTO

#============================================================================

# Configure ThreadPool
#============================================================================


org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 5
org.quartz.threadPool.threadPriority = 5

#============================================================================

# Configure JobStore
#============================================================================


org.quartz.jobStore.misfireThreshold = 60000

org.quartz.jobStore.class=org.quartz.impl.jdbcjobstore.JobStoreCMT
org.quartz.jobStore.driverDelegateClass=org.quartz.impl.jdbcjobstore.StdJDBCDelegate
org.quartz.jobStore.useProperties=false
org.quartz.jobStore.dataSource=managedDS
org.quartz.jobStore.nonManagedTXDataSource=nonManagedDS
org.quartz.jobStore.tablePrefix=QRTZ_
org.quartz.jobStore.isClustered=true
org.quartz.jobStore.clusterCheckinInterval = 20000

#=====================================================================
# Configure Datasources
#=====================================================================
org.quartz.dataSource.managedDS.jndiURL=jboss/datasources/psbpmsDS
org.quartz.dataSource.nonManagedDS.jndiURL=jboss/datasources/quartzNonManagedDS
```

お使いの環境に合わせて、以前の例を変更する必要があります。

66.3.8. クエリーサービス

クエリーサービスは、Dashbuilder データセットに基づく高度な検索機能を提供します。

この方法では、基礎となるデータストアからデータを取得する方法を制御できます。JPA エンティティーテーブルやカスタムシステムのデータベーステーブルなどの外部テーブルで複雑な **JOIN** ステートメントを使用できます。

クエリーサービスは、以下の 2 つの操作セットの前後に構築されます。

- 管理操作:
  - クエリー定義の登録
  - クエリー定義の置き換え
  - クエリー定義の登録解除 (削除)
  - クエリー定義の取得
  - 登録済みクエリー定義をすべて取得
- ランタイム操作:
  - **QueryParam** をフィルタープロバイダーとする単純なクエリー
  - **QueryParamBuilder** をフィルタープロバイダーとする高度なクエリー

Dashbuilder データセットは、CSV、SQL、Elastic Search などの複数のデータソースをサポートします。ただし、プロセスエンジンは RDBMS ベースのバックエンドを使用し、SQL ベースのデータセットにフォーカスします。

したがって、プロセスエンジンのクエリーサービスは、単純な API で効率的なクエリーを可能にする Dashbuilder データセット機能のサブセットです。

66.3.8.1. クエリーサービスのキークラス

クエリーサービスは、以下の主要なクラスに依存します。

- **QueryDefinition**: データセットの定義を表します。定義は、一意の名前、SQL 式 (クエリー) および **source** (クエリーの実行時に使用するデータソースの JNDI 名) で設定されます。
- **QueryParam**: 個別のクエリーパラメーターまたは条件を表す基本的な構造です。この構造は、列名、演算子、および予想される値で設定されます。
- **QueryResultMapper**: raw データセットデータ (行および列) をオブジェクト表現にマッピングするクラス。
- **QueryParamBuilder**: クエリー定義に適用されているクエリーフィルターをビルドしてクエリーを呼び出すクラス。

QueryResultMapper

QueryResultMapper は、データベース (データセット) からオブジェクト表現に取得したデータをマッピングします。これは、テーブルをエンティティーにマップする **hibernate** などの ORM プロバイダーと似ています。

データセット結果を表すのに使用できるオブジェクトタイプが多数あります。したがって、既存のマッパーは、常にニーズに適しているとは限りません。**QueryResultMapper** のマッパーはプラグ可能で、必要に応じてデータセットデータを必要なタイプに変換するために独自のマッパーを提供できます。

プロセスエンジンは以下のマッパーを提供します。

- **ProcessInstances** 名で登録した **org.jbpm.kie.services.impl.query.mapper.ProcessInstanceQueryMapper**
- **ProcessInstancesWithVariables** 名で登録した **org.jbpm.kie.services.impl.query.mapper.ProcessInstanceWithVarsQueryMapper**
- **ProcessInstancesWithCustomVariables** 名で登録した **org.jbpm.kie.services.impl.query.mapper.ProcessInstanceWithCustomVarsQueryMapper**
- **UserTasks** 名で登録した **org.jbpm.kie.services.impl.query.mapper.UserTaskInstanceQueryMapper**
- **UserTasksWithVariables** で登録した **org.jbpm.kie.services.impl.query.mapper.UserTaskInstanceWithVarsQueryMapper**
- **UserTasksWithCustomVariables** で登録した **org.jbpm.kie.services.impl.query.mapper.UserTaskInstanceWithCustomVarsQueryMapper**
- **TaskSummaries** で登録した **org.jbpm.kie.services.impl.query.mapper.TaskSummaryQueryMapper**。
- **RawList** で登録した **org.jbpm.kie.services.impl.query.mapper.RawListQueryMapper**

各 **QueryResultMapper** は、一意の文字列名で登録されます。完全なクラス名を参照する代わりに、この名前でマッパーを検索できます。この機能は、クライアント側での特定の実装に依存しないようにするため、サービスの EJB リモート呼び出しを使用する場合に特に重要です。

文字列名で **QueryResultMapper** を参照するには、**jbpm-services-api** モジュールの一部である **NamedQueryMapper** を使用します。このクラスは委譲 (遅延委譲) として機能し、クエリーの実行時に実際のマッパーを検索します。

NamedQueryMapper の使用

```
queryService.query("my query def", new NamedQueryMapper<Collection<ProcessInstanceDesc>>
("ProcessInstances"), new QueryContext());
```

QueryParamBuilder

QueryParamBuilder は、データセットにフィルターを構築する高度な方法を提供します。

デフォルトでは、ゼロ以上の **QueryParam** インスタンスを許可する **QueryService** のクエリーメソッドを使用する場合、このパラメーターはすべて **AND** 演算子と結合されるため、データエントリーはそれらすべてに一致する必要があります。

ただし、パラメーター間でより複雑な関係が必要になる場合があります。**QueryParamBuilder** を使用して、クエリーの発行時にフィルターを提供するカスタムビルダーを構築できます。

QueryParamBuilder の既存の実装の 1 つは、プロセスエンジンで使用できます。**コア関数** に基づくデフォルトの **QueryParams** を説明します。

これらのコア機能は、以下の条件を含む SQL ベースの条件です。

- **IS_NULL**

- **NOT_NULL**
- **EQUALS_TO**
- **NOT_EQUALS_TO**
- **LIKE_TO**
- **GREATER_THAN**
- **GREATER_OR_EQUALS_TO**
- **LOWER_THAN**
- **LOWER_OR_EQUALS_TO**
- **BETWEEN**
- **IN**
- **NOT_IN**

クエリーを呼び出す前に、メソッドが null 以外の値を返す間、プロセスエンジンは **QueryParamBuilder** インターフェイスのビルドメソッドを必要な回数呼び出します。このアプローチにより、**QueryParams** の単純なリストで表現できない複雑なフィルターオプションを構築できます。

以下の例は、**QueryParamBuilder** の基本実装を示しています。DashBuilder Dataset API に依存します。

QueryParamBuilder の基本実装

```
public class TestQueryParamBuilder implements QueryParamBuilder<ColumnFilter> {

    private Map<String, Object> parameters;
    private boolean built = false;
    public TestQueryParamBuilder(Map<String, Object> parameters) {
        this.parameters = parameters;
    }

    @Override
    public ColumnFilter build() {
        // return null if it was already invoked
        if (built) {
            return null;
        }

        String columnName = "processInstanceId";

        ColumnFilter filter = FilterFactory.OR(
                FilterFactory.greaterOrEqualsTo((Long)parameters.get("min")),
                FilterFactory.lowerOrEqualsTo((Long)parameters.get("max")));
        filter.setColumnId(columnName);

        built = true;
        return filter;
```

```
    }

}
```

ビルダーの実装後に、以下の例のように **QueryService** サービスでクエリーを実行する際にこのクラスのインスタンスを使用できます。

QueryService サービスを使用したクエリーの実行

```
queryService.query("my query def", ProcessInstanceQueryMapper.get(), new QueryContext(),
paramBuilder);
```

66.3.8.2. 典型的なシナリオでのクエリーサービスの使用

以下の手順では、コードがクエリーサービスを使用する一般的な方法を概説します。

手順

1. 使用するデータのビューであるデータセットを定義します。サービス API の **QueryDefinition** クラスを使用して、この操作を完了します。

   データセットの定義

   ```
   SqlQueryDefinition query = new SqlQueryDefinition("getAllProcessInstances",
   "java:jboss/datasources/ExampleDS");
   query.setExpression("select * from processinstancelog");
   ```

   この例では、最も簡単なクエリー定義を表しています。

   コンストラクターには以下のパラメーターが必要です。

   - ランタイム時にクエリーを識別する一意の名前
   - この定義でクエリーを実行するのに使用する JNDI データソース名
     setExpression() メソッドのパラメーターは、データセットビューをビルドする SQL ステートメントです。クエリーサービスのクエリーは、このビューのデータを使用し、必要に応じてこのデータをフィルターリングします。

2. クエリーを登録します。

   クエリーの登録

   ```
   queryService.registerQuery(query);
   ```

3. 必要な場合は、フィルターリングせずに、データセットからすべてのデータを収集します。

   データセットからすべてのデータを収集します。

   ```
   Collection<ProcessInstanceDesc> instances = queryService.query("getAllProcessInstances",
   ProcessInstanceQueryMapper.get(), new QueryContext());
   ```

   この簡単なクエリーは、ページングとソートに **QueryContext** のデフォルトを使用します。

4. 必要な場合は、ページングおよびソートのデフォルト値を変更する **QueryContext** オブジェクトを使用します。

   QueryContext オブジェクトを使用したデフォルトの変更

   ```
   QueryContext ctx = new QueryContext(0, 100, "start_date", true);

   Collection<ProcessInstanceDesc> instances = queryService.query("getAllProcessInstances",
   ProcessInstanceQueryMapper.get(), ctx);
   ```

5. 必要な場合は、クエリーを使用してデータをフィルターリングします。

   クエリーを使用したデータのフィルター

   ```
   // single filter param
   Collection<ProcessInstanceDesc> instances = queryService.query("getAllProcessInstances",
   ProcessInstanceQueryMapper.get(), new QueryContext(),
   QueryParam.likeTo(COLUMN_PROCESSID, true, "org.jbpm%"));

   // multiple filter params (AND)
   Collection<ProcessInstanceDesc> instances = queryService.query("getAllProcessInstances",
   ProcessInstanceQueryMapper.get(), new QueryContext(),
    QueryParam.likeTo(COLUMN_PROCESSID, true, "org.jbpm%"),
    QueryParam.in(COLUMN_STATUS, 1, 3));
   ```

クエリーサービスでは、取得するデータやフィルターリング方法を定義できます。JPA プロバイダーまたはその他の同様の制限は適用されません。データベースクエリーを環境に調整して、パフォーマンスを向上させることができます。

66.3.9. 高度なクエリーサービス

高度なクエリーサービスは、プロセスおよびタスクの属性、プロセス変数、ユーザータスクの内部変数に基づいて、プロセスおよびタスクを検索する機能を提供します。検索は、プロセスエンジン内の既存のプロセスをすべて自動的に対応します。

属性および変数の名前および必要な値は **QueryParam** オブジェクトで定義されます。

プロセス属性には、プロセスインスタンス ID、相関キー、プロセス定義 ID、およびデプロイメント ID が含まれます。タスク属性には、タスク名、所有者、ステータスが含まれます。

以下の検索方法を使用できます。

- **queryProcessByVariables**: プロセス属性とプロセス変数値のリストをもとにプロセスインスタンスを検索します。その結果に追加するには、プロセスインスタンスに一覧表示される属性と、プロセス変数に一覧表示される値が必要です。
- **queryProcessByVariablesAndTask**: プロセス属性、プロセス変数値、タスク変数値のリストをもとにプロセスインスタンスを検索します。その結果に追加するには、プロセスインスタンスに一覧表示される属性と、プロセス変数に一覧表示される値が必要です。また、タスク変数にリストされた値を持つタスクも含める必要があります。
- **queryUserTasksByVariables**: タスク属性、タスク変数値、プロセス変数値のリストを基にしてユーザータスクを検索します。結果を含めるには、タスクに、一覧表示される属性と、そのタスク変数に値の一覧が含まれている必要があります。また、プロセス変数に一覧表示された値が含まれるプロセスにも組み込む必要があります。

サービスは **AdvanceRuntimeDataService** クラスによって提供されます。このクラスのインターフェイスは、事前定義のタスクおよびプロセス属性名も定義します。

AdvanceRuntimeDataService インターフェイスの定義

```
public interface AdvanceRuntimeDataService {

    String TASK_ATTR_NAME = "TASK_NAME";
    String TASK_ATTR_OWNER = "TASK_OWNER";
    String TASK_ATTR_STATUS = "TASK_STATUS";
    String PROCESS_ATTR_INSTANCE_ID = "PROCESS_INSTANCE_ID";
    String PROCESS_ATTR_CORRELATION_KEY = "PROCESS_CORRELATION_KEY";
    String PROCESS_ATTR_DEFINITION_ID = "PROCESS_DEFINITION_ID";
    String PROCESS_ATTR_DEPLOYMENT_ID = "PROCESS_DEPLOYMENT_ID";
    String PROCESS_COLLECTION_VARIABLES = "ATTR_COLLECTION_VARIABLES";

    List<ProcessInstanceWithVarsDesc> queryProcessByVariables(List<QueryParam> attributes,
      List<QueryParam> processVariables, QueryContext queryContext);

    List<ProcessInstanceWithVarsDesc> queryProcessByVariablesAndTask(List<QueryParam>
attributes,
        List<QueryParam> processVariables, List<QueryParam> taskVariables,
        List<String> potentialOwners, QueryContext queryContext);

    List<UserTaskInstanceWithPotOwnerDesc> queryUserTasksByVariables(List<QueryParam>
attributes,
        List<QueryParam> taskVariables, List<QueryParam> processVariables,
        List<String> potentialOwners, QueryContext queryContext);
}
```

66.3.10. プロセスインスタンス移行サービス

プロセスインスタンスの移行サービスは、プロセスインスタンスをあるデプロイメントから別のデプロイメントに移行するユーティリティーです。プロセスまたはタスクの変数は移行の影響を受けません。ただし、新規デプロイメントでは異なるプロセス定義を使用することができます。

また、プロセスインスタンス移行サービスは、プロセスの移行時に、プロセスの全サブプロセス、サブプロセス、サブプロセスのサブプロセスなどを自動的に移行します。親プロセスを移行せずにサブプロセスを移行しようとすると、移行に失敗します。

プロセス移行の最も簡単な方法として、アクティブなプロセスインスタンスを終了して、新しいデプロイメントで新しいプロセスインスタンスを開始することができます。このアプローチがニーズに適していない場合は、プロセスインスタンスの移行を開始する前に、以下の問題を考慮してください。

- 後方互換性
- データ変更
- ノードのマッピングに必要

可能な場合は、プロセス定義を拡張して後方互換性のプロセスを作成します。たとえば、プロセス定義からノードを削除すると、互換性が失われます。このような変更を加える場合は、ノードマッピングを指定する必要があります。アクティブなプロセスインスタンスが削除されたノードにある場合、プロセスインスタンスの移行は、ノードのマッピングを使用します。

ノードマップには、新規プロセス定義内のターゲットノード ID にマップされた古いプロセス定義のソースノード ID が含まれます。ユーザータスクをユーザータスクへなど、同じタイプのノードのみをマップできます。

Red Hat Process Automation Manager は、移行サービスの実装を複数提供します。

移行サービスを実装する ProcessInstanceMigrationService インターフェイスのメソッド

```
public interface ProcessInstanceMigrationService {
 /**
 * Migrates a given process instance that belongs to the source deployment into the target process ID that belongs to the target deployment.
 * The following rules are enforced:
 * <ul>
 * <li>the source deployment ID must point to an existing deployment</li>
 * <li>the process instance ID must point to an existing and active process instance</li>
 * <li>the target deployment must exist</li>
 * <li>the target process ID must exist in the target deployment</li>
 * </ul>
 * Returns a migration report regardless of migration being successful or not; examine the report for the outcome of the migration.
 * @param sourceDeploymentId deployment to which the process instance to be migrated belongs
 * @param processInstanceId ID of the process instance to be migrated
 * @param targetDeploymentId ID of the deployment to which the target process belongs
 * @param targetProcessId ID of the process to which the process instance should be migrated
 * @return returns complete migration report
 */
 MigrationReport migrate(String sourceDeploymentId, Long processInstanceId, String targetDeploymentId, String targetProcessId);
 /**
 * Migrates a given process instance (with node mapping) that belongs to source deployment into the target process ID that belongs to the target deployment.
 * The following rules are enforced:
 * <ul>
 * <li>the source deployment ID must point to an existing deployment</li>
 * <li>the process instance ID must point to an existing and active process instance</li>
 * <li>the target deployment must exist</li>
 * <li>the target process ID must exist in the target deployment</li>
 * </ul>
 * Returns a migration report regardless of migration being successful or not; examine the report for the outcome of the migration.
 * @param sourceDeploymentId deployment to which the process instance to be migrated belongs
 * @param processInstanceId ID of the process instance to be migrated
 * @param targetDeploymentId ID of the deployment to which the target process belongs
 * @param targetProcessId ID of the process to which the process instance should be migrated
 * @param nodeMapping node mapping - source and target unique IDs of nodes to be mapped - from process instance active nodes to new process nodes
 * @return returns complete migration report
 */
 MigrationReport migrate(String sourceDeploymentId, Long processInstanceId, String targetDeploymentId, String targetProcessId, Map<String, String> nodeMapping);
 /**
 * Migrates given process instances that belong to the source deployment into a target process ID that belongs to the target deployment.
 * The following rules are enforced:
 * <ul>
```

```
 * <li>the source deployment ID must point to an existing deployment</li>
 * <li>the process instance ID must point to an existing and active process instance</li>
 * <li>the target deployment must exist</li>
 * <li>the target process ID must exist in the target deployment</li>
 * </ul>
 * Returns a migration report regardless of migration being successful or not; examine the report for
the outcome of the migration.
 * @param sourceDeploymentId deployment to which the process instances to be migrated belong
 * @param processInstanceIds list of process instance IDs to be migrated
 * @param targetDeploymentId ID of the deployment to which the target process belongs
 * @param targetProcessId ID of the process to which the process instances should be migrated
 * @return returns complete migration report
 */
 List<MigrationReport> migrate(String sourceDeploymentId, List<Long> processInstanceIds, String
targetDeploymentId, String targetProcessId);
 /**
 * Migrates given process instances (with node mapping) that belong to the source deployment into a
target process ID that belongs to the target deployment.
 * The following rules are enforced:
 * <ul>
 * <li>the source deployment ID must point to an existing deployment</li>
 * <li>the process instance ID must point to an existing and active process instance</li>
 * <li>the target deployment must exist</li>
 * <li>the target process ID must exist in the target deployment</li>
 * </ul>
 * Returns a migration report regardless of migration being successful or not; examine the report for
the outcome of the migration.
 * @param sourceDeploymentId deployment to which the process instances to be migrated belong
 * @param processInstanceIds list of process instance ID to be migrated
 * @param targetDeploymentId ID of the deployment to which the target process belongs
 * @param targetProcessId ID of the process to which the process instances should be migrated
 * @param nodeMapping node mapping - source and target unique IDs of nodes to be mapped - from
process instance active nodes to new process nodes
 * @return returns list of migration reports one per each process instance
 */
 List<MigrationReport> migrate(String sourceDeploymentId, List<Long> processInstanceIds, String
targetDeploymentId, String targetProcessId, Map<String, String> nodeMapping);
}
```

KIE Server のプロセスインスタンスを移行するには、以下の実装を使用します。これらのメソッドは **ProcessInstanceMigrationService** インターフェイスのメソッドと類似しており、KIE Server デプロイメントに同じ移行実装を提供します。

KIE Server デプロイメントの移行サービスを実装する ProcessAdminServicesClient インターフェイスのメソッド

```
public interface ProcessAdminServicesClient {

    MigrationReportInstance migrateProcessInstance(String containerId, Long processInstanceId,
String targetContainerId, String targetProcessId);

    MigrationReportInstance migrateProcessInstance(String containerId, Long processInstanceId,
String targetContainerId, String targetProcessId, Map<String, String> nodeMapping);

    List<MigrationReportInstance> migrateProcessInstances(String containerId, List<Long>
```

```
processInstancesId, String targetContainerId, String targetProcessId);

    List<MigrationReportInstance> migrateProcessInstances(String containerId, List<Long>
processInstancesId, String targetContainerId, String targetProcessId, Map<String, String>
nodeMapping);
}
```

1 つのプロセスインスタンスまたは複数のプロセスインスタンスを一度に移行することができます。複数のプロセスインスタンスを移行する場合は、各インスタンスを別のトランザクションに移行し、移行が相互に影響しないようにします。

移行が完了すると、**migrate** メソッドは以下の情報が含まれる **MigrationReport** オブジェクトを返します。

- 移行の開始日および終了日。
- 移行の結果 (成功または失敗)。
- **INFO**、**WARN**、または **ERROR** タイプのログエントリー。**ERROR** メッセージは移行を終了します。

プロセスインスタンスの移行の例を以下に示します。

KIE Server デプロイメントのプロセスインスタンスの移行

```
import org.kie.server.api.model.admin.MigrationReportInstance;
import org.kie.server.api.marshalling.MarshallingFormat;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;

public class ProcessInstanceMigrationTest{

 private static final String SOURCE_CONTAINER = "com.redhat:MigrateMe:1.0";
  private static final String SOURCE_PROCESS_ID = "MigrateMe.MigrateMev1";
 private static final String TARGET_CONTAINER = "com.redhat:MigrateMe:2";
  private static final String TARGET_PROCESS_ID = "MigrateMe.MigrateMeV2";

 public static void main(String[] args) {

  KieServicesConfiguration config =
KieServicesFactory.newRestConfiguration("http://HOST:PORT/kie-server/services/rest/server",
"USERNAME", "PASSWORD");
  config.setMarshallingFormat(MarshallingFormat.JSON);
  KieServicesClient client = KieServicesFactory.newKieServicesClient(config);

  long sourcePid = client.getProcessClient().startProcess(SOURCE_CONTAINER,
SOURCE_PROCESS_ID);

    // Use the 'report' object to return migration results.
  MigrationReportInstance report =
client.getAdminClient().migrateProcessInstance(SOURCE_CONTAINER,
sourcePid,TARGET_CONTAINER, TARGET_PROCESS_ID);

  System.out.println("Was migration successful:" + report.isSuccessful());

  client.getProcessClient().abortProcessInstance(TARGET_CONTAINER, sourcePid);
```

```
  }
}
```

プロセスインスタンス移行の既知の制限
以下の状況では、移行の失敗や移行の誤った状況が発生する可能性があります。

- 新規または変更されたタスクには、移行したプロセスインスタンスでは利用できない入力が必要です。
- 変更がさらなる処理に影響を与えるアクティブなタスクの前にタスクを変更します。
- 現在アクティブなヒューマンタスクを削除します。ヒューマンタスクを置き換えるには、別のヒューマンタスクにマップする必要があります。
- 単一のアクティブなタスクと並行して新しいタスクを追加します。**AND** ゲートウェイのすべての分岐がアクティブになっていないため、プロセスは停止します。
- アクティブなタイマーイベントを削除します (これらのイベントはデータベースでは変更しません)。
- アクティブなタスクでの入力および出力を修正または更新します (タスクデータは移行されません)。

タスクノードへのマッピングを適用する場合は、タスクノード名と説明のみがマッピングされます。**TaskName** 変数を含む他のタスクフィールドは、新規タスクにマップされません。

66.3.11. デプロイメントおよび異なるプロセスバージョン

デプロイメントサービスは、ビジネスアセットを実行環境に配置します。ただし、場合によっては、正しいコンテキストでアセットを使用できるようにするために追加の管理が必要になる場合があります。特に、同じプロセスを複数バージョンにデプロイする場合は、プロセスインスタンスが正しいバージョンを使用するようにする必要があります。

デプロイメントのアクティブ化および非アクティブ化
ケースによっては、デプロイメントで複数のプロセスインスタンスが実行し、ランタイム環境に同じプロセスの新しいバージョンを追加する場合があります。

既存のアクティブなインスタンスが以前のバージョンを続行する必要がある間、このプロセス定義の新規インスタンスは新しいバージョンを使用する必要があります。

このシナリオを有効にするには、デプロイメントサービスの以下の方法を使用します。

- **activate**: 対話に使用できるデプロイメントを有効にします。そのプロセス定義を一覧表示し、このデプロイメントの新規プロセスインスタンスを開始できます。
- **deactivate**: デプロイメントを非アクティブにします。プロセス定義を一覧表示し、デプロイメント内のプロセスの新規プロセスインスタンスを開始するオプションを無効にします。ただし、シグナルイベントやユーザータスクとの対話など、すでにアクティブなプロセスインスタンスを引き続き使用できます。

この機能を使用して、プロセスインスタンスの移行を必要とせずに、プロジェクトバージョン間のスムーズな移行を行うことができます。

プロセスの最新バージョンの呼び出し

プロジェクトのプロセスの最新版を使用する必要がある場合は、**latest** のキーワードを使用して、サービス内の複数の操作と対話することができます。このアプローチは、全バージョンでプロセス ID が同じままである場合にのみサポートされます。

以下の例では、機能について説明します。

最初のデプロイメントユニットは **org.jbpm:HR:1.0** です。これには、採用プロセスの最初のバージョンが含まれています。

数週間後、新しいバージョンを開発し、**org.jbpm:HR.2.0** として実行サーバーにデプロイします。これには、採用プロセスのバージョン 2 が含まれています。

プロセスを呼び出して最新バージョンを使用する場合は、以下のデプロイメント ID を使用することができます。

```
org.jbpm.HR:latest
```

このデプロイメント ID を使用する場合、プロセスエンジンは、プロジェクトの利用可能な最新バージョンを見つけます。以下の識別子を使用します。

- **groupId**: **org.jbpm**
- **artifactId**: **HR**

バージョン番号は、Maven ルールによって照合され、最新バージョンを見つけることができます。

以下のコード例は、複数のバージョンのデプロイメントと最新バージョンの操作を示しています。

プロセスの複数のバージョンをデプロイし、最新バージョンと対話する

```
KModuleDeploymentUnit deploymentUnitV1 = new KModuleDeploymentUnit("org.jbpm", "HR", "1.0");
deploymentService.deploy(deploymentUnitV1);

long processInstanceId = processService.startProcess("org.jbpm:HR:LATEST", "customtask");
ProcessInstanceDesc piDesc = runtimeDataService.getProcessInstanceById(processInstanceId);

// We have started a process with the project version 1
assertEquals(deploymentUnitV1.getIdentifier(), piDesc.getDeploymentId());

// Next we deploy version 2
KModuleDeploymentUnit deploymentUnitV2 = new KModuleDeploymentUnit("org.jbpm", "HR", "2.0");
deploymentService.deploy(deploymentUnitV2);

processInstanceId = processService.startProcess("org.jbpm:HR:LATEST", "customtask");
piDesc = runtimeDataService.getProcessInstanceById(processInstanceId);

// This time we have started a process with the project version 2
assertEquals(deploymentUnitV2.getIdentifier(), piDesc.getDeploymentId());
```

注記

この機能は、KIE Server REST API でも利用できます。デプロイメント ID でリクエストを送信する場合は、**LATEST** をバージョン識別子として使用できます。

関連情報

- KIE API を使った Red Hat Process Automation Manager の操作

66.3.12. デプロイメントの同期

プロセスエンジンサービスには、すべてのデプロイメントのデプロイメント記述子を含め、使用可能なデプロイメントをデータベースに格納するデプロイメントシンクロナイザーが含まれています。

また、シンクロナイザーはこのテーブルを監視して、同じデータソースを使用している可能性のある他のインストールと同期させます。この機能は、クラスターで実行している場合や、Business Central とカスタムアプリケーションが同じアーティファクト上で動作する必要がある場合に特に重要です。

デフォルトでは、コアサービスを実行する場合は同期を設定する必要があります。EJB および CDI の拡張機能では、同期は自動的に有効になります。

以下のコードサンプルは同期を設定します。

同期の設定

```
TransactionalCommandService commandService = new TransactionalCommandService(emf);

DeploymentStore store = new DeploymentStore();
store.setCommandService(commandService);

DeploymentSynchronizer sync = new DeploymentSynchronizer();
sync.setDeploymentService(deploymentService);
sync.setDeploymentStore(store);

DeploymentSyncInvoker invoker = new DeploymentSyncInvoker(sync, 2L, 3L, TimeUnit.SECONDS);
invoker.start();
....
invoker.stop();
```

この設定では、デプロイメントは 3 秒ごとに同期され、初期の遅延は 2 秒となっています。

66.4. プロセスエンジンのスレッド

論理 および **技術** の 2 種類のマルチスレッドを参照できます。**技術的なマルチスレッド** には、Java や C プログラムなどにより開始される複数のスレッドまたはプロセスが必要です。**論理マルチスレッド** は、たとえば、プロセスが並列ゲートウェイに到達すると、BPM プロセスで実行されます。実行ロジックでは、元のプロセスは並行方式で実行する 2 つのプロセスに分割されます。

プロセスエンジンコードは、1 つの技術スレッドを使用して論理マルチスレッドを実装します。

この設計の理由は、複数の (技術的な) スレッドが同じプロセスで作業している場合に状態情報を相互に通信できる必要があることです。この要件により、多くの問題が発生します。スレッド間の安全な通信に必要な追加のロジックや、競合状態やデッドロックを回避するために追加のオーバーヘッドが発生すると、このようなスレッドの使用によるパフォーマンス上のメリットがなくなります。

一般的に、プロセスエンジンは連続してアクションを実行します。たとえば、プロセスエンジンがプロセスでスクリプトタスクに遭遇すると、スクリプトが同期的に実行し、スクリプトが完了するのを待ってから、プロセスエンジンの実行を続行します。同様に、プロセスが並列ゲートウェイに遭遇すると、プロセスエンジンは各発信ブランチを順番にトリガーします。

これは、実行がほぼ常にインスタンス化されるため可能です。つまり、非常に高速でオーバーヘッドがほぼ発生しないためです。その結果、順次実行を実行しても、ユーザーに通知できる影響は作成されません。

指定したプロセスのコードはすべて同期的に実行され、プロセスエンジンは完了するまで待機してからプロセスを続行します。たとえば、カスタムスクリプトの一部として **Thread.sleep(...)** を使用する場合、プロセスエンジンスレッドはスリープ期間中にブロックされます。

プロセスがサービスタスクに到達すると、プロセスエンジンはタスクのハンドラーも同期的に呼び出し、**completeWorkItem(...)** メソッドが戻るのを待ってから実行を継続します。サービスハンドラーがインスタンス化されていない場合は、コードに非同期実行を個別に実装します。

たとえば、サービスタスクが外部サービスを呼び出す場合があります。このサービスをリモートで呼び出し、その結果を待機する遅延は大きな可能性があります。したがって、このサービスを非同期で呼び出します。ハンドラーはサービスのみを呼び出してからメソッドを返し、その結果が利用可能になったらプロセスエンジンに通知します。その間、プロセスエンジンはプロセスの実行を継続できます。

ヒューマンタスクは、非同期で呼び出す必要のあるサービスの典型的な例です。ヒューマンタスクでは、要求に応答するためのヒューマンアクターが必要で、プロセスエンジンはこの応答を待ちません。

ヒューマンタスクノードが開始すると、ヒューマンタスクハンドラーは割り当てられたアクターのタスクリストに新しいタスクのみを作成します。その後、プロセスエンジンは、必要に応じて残りのプロセスで実行を継続できます。ハンドラーは、ユーザーがタスクを完了するとプロセスエンジンに非同期的に通知します。

66.5. プロセスエンジンの実行エラー

タスクサービスを含むプロセスエンジン実行のどの部分でも、例外を発生することができます。例外は、**java.lang.Throwable** を拡張する任意のクラスとすることができます。

一部の例外は、プロセスレベルで処理されます。注目すべき点は、ワークアイテムハンドラーは、エラー処理のためのサブプロセスを指定するカスタム例外を発生することができることです。ワークアイテムハンドラーの開発については、**カスタムタスクおよびワークアイテムハンドラー** を参照してください。

例外が処理されずにプロセスエンジンに到達した場合、**実行エラー** となります。実行エラーが発生した場合、プロセスエンジンは現在のトランザクションをロールバックし、プロセスを以前の安定した状態にします。その後、プロセスエンジンは、その時点からプロセスの実行を継続します。

実行エラーは、プロセスエンジンにリクエストを送った呼び出し元が見ることができます。プロセスエンジンは、実行エラーを処理し、その情報を保存するための拡張可能なメカニズムも備えています。このメカニズムは、以下のコンポーネントで設定されています。

- **ExecutionErrorManager**: エラー処理のエントリーポイント。このクラスは **RuntimeManager** と統合され、基盤となる **KieSession** および **TaskService** に提供されます。**ExecutionErrorManager** は、実行エラー処理メカニズムの他のクラスへのアクセスを提供します。
  プロセスエンジンが **RuntimeManager** インスタンスを作成すると、それに対応する **ExecutionErrorManager** インスタンスも作成されます。

- **ExecutionErrorHandler**: エラー処理のためのメインのクラス。このクラスはプロセスエンジンに実装されており、通常、直接カスタマイズや拡張を行う必要はありません。**ExecutionErrorHandler** は、エラーフィルターを呼び出して特定のエラーを処理し、**ExecutionErrorStorage** を呼び出してエラー情報を格納します。
  ExecutionErrorHandler は、**RuntimeEngine** のライフサイクルにバインドされ、新しいランタイムエンジンの生成時に生成され、**RuntimeEngine** が破棄されたときに破棄されます。ある

実行コンテキストもしくはトランザクションでは、単一インスタンスの **ExecutionErrorHandler** が使用されます。**KieSession** および **TaskService** の両方がそのインスタンスを使用して、処理されたノードまたはタスクに関するエラー処理を通知します。**ExecutionErrorHandler** には、以下のイベントが通知されます。

- ノードインスタンスの処理の開始
- ノードインスタンスの処理の完了
- タスクインスタンスの処理の開始
- タスクインスタンスの処理の完了
  特にエラー自体がプロセスコンテキスト情報を提供しない場合、**ExecutionErrorHandler** はこの情報を使用してエラーのコンテキストを記録します。たとえば、データベース例外にはプロセス情報がありません。

- **ExecutionErrorStorage**: 実行エラー情報のための、プラグ可能なストレージクラス
  プロセスエンジンが **RuntimeManager** インスタンスを作成すると、それに対応する **ExecutionErrorStorage** インスタンスも作成されます。続いて、**ExecutionErrorHandler** クラスはこの **ExecutionErrorStorage** インスタンスを呼び出して、すべての実行エラーの情報を保存します。

  デフォルトのストレージ実装では、データベーステーブルを使用して、すべてのエラーについて利用可能なすべての情報を格納します。一部のエラーでは、詳細な情報を抽出できない場合があるため、エラーの種類によって、利用できる詳細レベルが異なる場合があります。

- 特定の種類の実行エラーを処理する多数の **フィルター**。カスタムフィルターを追加することができます。

デフォルトでは、すべての実行エラーは **未承認** として記録されます。Business Central を使用すると、記録されたすべての実行エラーを表示し、それを承認することができます。すべてまたは一部の実行エラーを自動的に承認するジョブを作成することもできます。

Business Central を使用して実行エラーを表示する、エラーを自動的に承認するジョブを作成する方法については、**Business Central でのビジネスプロセスの管理および監視** を参照してください。

66.5.1. 実行エラーのタイプとフィルター

実行エラーの処理では、どのような種類のエラーも補足して処理を試みます。しかし、エラーごとに異なる方法で処理しなければならない可能性があります。また、エラーの種類によって、利用できる詳細情報が異なります。

エラー処理メカニズムは、プラグイン可能な**フィルター** をサポートしています。どのフィルターも、特定の種類のエラーを処理します。特定のエラーを異なる方法で処理するフィルターを追加し、デフォルトの処理を上書きすることができます。

フィルターは、**ExecutionErrorFilter** インターフェイスの実装です。このインターフェイスは **ExecutionError** のインスタンスをビルドし、これは後で **ExecutionErrorStorage** クラスを使用して保存されます。

ExecutionErrorFilter インターフェイスは、以下のメソッドを持ちます。

- **accept**: エラーがフィルターで処理可能かどうかを示します
- **filter**: エラーを処理し、**ExecutionError** インスタンスを返します
- **getPriority**: このフィルターの優先度を示します

実行エラーハンドラーは、各エラーを個別に処理します。エラーが発生するたびに、登録されているすべてのフィルターの **accept** メソッドを、優先度値の低いものから順に呼び出し始めます。あるフィルターの **accept** メソッドが **true** を返した場合、ハンドラーはそのフィルターの **filter** メソッドを呼び出し、他のフィルターを呼び出しません。

優先順位があるため、どのようなエラーでも 1 つのフィルターだけが処理します。より特殊なフィルターほど、優先度の値は低くなります。どの特殊なフィルターも受け付けないエラーは、より高い優先度の値を持つ汎用フィルターに到達します。

ServiceLoader メカニズムは、**ExecutionErrorFilter** インスタンスを提供します。カスタムフィルターを登録するには、その完全修飾クラス名をサービスプロジェクトの **META-INF/services/org.kie.internal.runtime.error.ExecutionErrorFilter** ファイルに追加します。

Red Hat Process Automation Manager には、以下の実行エラーフィルターが付属しています。

表66.1 ExecutionErrorFilters

クラス名	タイプ	優先順位
org.jbpm.runtime.manager.impl.error.filters.ProcessExecutionErrorFilter	Process	100
org.jbpm.runtime.manager.impl.error.filters.TaskExecutionErrorFilter	タスク	80
org.jbpm.runtime.manager.impl.error.filters.DBExecutionErrorFilter	DB	200
org.jbpm.executor.impl.error.JobExecutionErrorFilter	Job	100

フィルターには、優先度の値の低いものが実行順序の高いものとして与えられます。したがって、実行エラーハンドラーは、以下の順序でこれらのフィルターを呼び出します。

1. タスク
2. Process
3. Job
4. DB

66.6. プロセスエンジンのイベントリスナー

プロセスやタスクがライフサイクルの異なるポイントに移行するたびに、プロセスエンジンはイベントを生成します。このようなイベントを受信して処理するクラスを開発することができます。このクラスを **イベントリスナー** と呼びます。

プロセスエンジンは、イベントオブジェクトをこのクラスに渡します。オブジェクトは、関連情報へのアクセスを提供します。例えば、イベントがプロセスノードに関連する場合、オブジェクトはプロセスインスタンスとノードインスタンスへのアクセスを提供します。

66.6.1. イベントリスナーのインターフェイス

プロセスエンジンのイベントリスナーを開発するために、以下のインターフェイスを使用することができます。

66.6.1.1. プロセスイベントリスナー用インターフェイス

ProcessEventListener インターフェイスを実装するクラスを開発できます。このクラスは、プロセスの開始や完了、またはノードへの入出力などのプロセス関連のイベントをリッスンできます。

以下のソースコードは、**ProcessEventListener** インターフェイスのさまざまなメソッドを示しています。

ProcessEventListener インターフェイス

```
public interface ProcessEventListener
    extends
    EventListener {

    /**
     * This listener method is invoked right before a process instance is being started.
     * @param event
     */
    void beforeProcessStarted(ProcessStartedEvent event);

    /**
     * This listener method is invoked right after a process instance has been started.
     * @param event
     */
    void afterProcessStarted(ProcessStartedEvent event);

    /**
     * This listener method is invoked right before a process instance is being completed (or aborted).
     * @param event
     */
    void beforeProcessCompleted(ProcessCompletedEvent event);

    /**
     * This listener method is invoked right after a process instance has been completed (or aborted).
     * @param event
     */
    void afterProcessCompleted(ProcessCompletedEvent event);

    /**
     * This listener method is invoked right before a node in a process instance is being triggered
     * (which is when the node is being entered, for example when an incoming connection triggers it).
     * @param event
     */
    void beforeNodeTriggered(ProcessNodeTriggeredEvent event);

    /**
     * This listener method is invoked right after a node in a process instance has been triggered
     * (which is when the node was entered, for example when an incoming connection triggered it).
     * @param event
     */
    void afterNodeTriggered(ProcessNodeTriggeredEvent event);

    /**
```

```
     * This listener method is invoked right before a node in a process instance is being left
     * (which is when the node is completed, for example when it has performed the task it was
     * designed for).
     * @param event
     */
    void beforeNodeLeft(ProcessNodeLeftEvent event);

    /**
     * This listener method is invoked right after a node in a process instance has been left
     * (which is when the node was completed, for example when it performed the task it was
     * designed for).
     * @param event
     */
    void afterNodeLeft(ProcessNodeLeftEvent event);

    /**
     * This listener method is invoked right before the value of a process variable is being changed.
     * @param event
     */
    void beforeVariableChanged(ProcessVariableChangedEvent event);

    /**
     * This listener method is invoked right after the value of a process variable has been changed.
     * @param event
     */
    void afterVariableChanged(ProcessVariableChangedEvent event);

    /**
     * This listener method is invoked right before a process/node instance's SLA has been violated.
     * @param event
     */
    default void beforeSLAViolated(SLAViolatedEvent event) {}

    /**
     * This listener method is invoked right after a process/node instance's SLA has been violated.
     * @param event
     */
    default void afterSLAViolated(SLAViolatedEvent event) {}

    /**
     * This listener method is invoked when a signal is sent
     * @param event
     */
    default void onSignal(SignalEvent event) {}

    /**
     * This listener method is invoked when a message is sent
     * @param event
     */
    default void onMessage(MessageEvent event) {}
}
```

これらのメソッドのいずれかを実装することで、対応するイベントを処理することができます。

プロセスエンジンがメソッドに渡すイベントクラスの定義については、Java ドキュメントの **org.kie.api.event.process** パッケージを参照してください。

イベントクラスのメソッドを使用して、イベントに関与するエンティティーに関するすべての情報が含まれる他のクラスを取得することができます。

次の例は、**afterNodeLeft()** などのノード関連イベントの一部で、プロセスインスタンスおよびノードタイプを取得します。

ノード関連イベントのプロセスインスタンスおよびノードタイプの取得

```
WorkflowProcessInstance processInstance = event.getNodeInstance().getProcessInstance()
NodeType nodeType = event.getNodeInstance().getNode().getNodeType()
```

66.6.1.2. タスクライフサイクルイベントリスナー用インターフェイス

TaskLifecycleEventListener インターフェイスを実装するクラスを開発できます。このクラスは、タスクのライフサイクルに関連するイベント (オーナーの割り当てやタスクの完了など) をリッスンすることができます。

以下のソースコードは、**TaskLifecycleEventListener** インターフェイスのさまざまなメソッドを示しています。

TaskLifecycleEventListener インターフェイス

```
public interface TaskLifeCycleEventListener extends EventListener {

    public enum AssignmentType {
        POT_OWNER,
        EXCL_OWNER,
        ADMIN;
    }

    public void beforeTaskActivatedEvent(TaskEvent event);
    public void beforeTaskClaimedEvent(TaskEvent event);
    public void beforeTaskSkippedEvent(TaskEvent event);
    public void beforeTaskStartedEvent(TaskEvent event);
    public void beforeTaskStoppedEvent(TaskEvent event);
    public void beforeTaskCompletedEvent(TaskEvent event);
    public void beforeTaskFailedEvent(TaskEvent event);
    public void beforeTaskAddedEvent(TaskEvent event);
    public void beforeTaskExitedEvent(TaskEvent event);
    public void beforeTaskReleasedEvent(TaskEvent event);
    public void beforeTaskResumedEvent(TaskEvent event);
    public void beforeTaskSuspendedEvent(TaskEvent event);
    public void beforeTaskForwardedEvent(TaskEvent event);
    public void beforeTaskDelegatedEvent(TaskEvent event);
    public void beforeTaskNominatedEvent(TaskEvent event);
    public default void beforeTaskUpdatedEvent(TaskEvent event){};
    public default void beforeTaskReassignedEvent(TaskEvent event){};
    public default void beforeTaskNotificationEvent(TaskEvent event){};
    public default void beforeTaskInputVariableChangedEvent(TaskEvent event, Map<String, Object>
variables){};
    public default void beforeTaskOutputVariableChangedEvent(TaskEvent event, Map<String,
Object> variables){};
    public default void beforeTaskAssignmentsAddedEvent(TaskEvent event, AssignmentType type,
List<OrganizationalEntity> entities){};
    public default void beforeTaskAssignmentsRemovedEvent(TaskEvent event, AssignmentType
```

```
type, List<OrganizationalEntity> entities){};

    public void afterTaskActivatedEvent(TaskEvent event);
    public void afterTaskClaimedEvent(TaskEvent event);
    public void afterTaskSkippedEvent(TaskEvent event);
    public void afterTaskStartedEvent(TaskEvent event);
    public void afterTaskStoppedEvent(TaskEvent event);
    public void afterTaskCompletedEvent(TaskEvent event);
    public void afterTaskFailedEvent(TaskEvent event);
    public void afterTaskAddedEvent(TaskEvent event);
    public void afterTaskExitedEvent(TaskEvent event);
    public void afterTaskReleasedEvent(TaskEvent event);
    public void afterTaskResumedEvent(TaskEvent event);
    public void afterTaskSuspendedEvent(TaskEvent event);
    public void afterTaskForwardedEvent(TaskEvent event);
    public void afterTaskDelegatedEvent(TaskEvent event);
    public void afterTaskNominatedEvent(TaskEvent event);
    public default void afterTaskReassignedEvent(TaskEvent event){};
    public default void afterTaskUpdatedEvent(TaskEvent event){};
    public default void afterTaskNotificationEvent(TaskEvent event){};
    public default void afterTaskInputVariableChangedEvent(TaskEvent event, Map<String, Object>
variables){};
    public default void afterTaskOutputVariableChangedEvent(TaskEvent event, Map<String, Object>
variables){};
    public default void afterTaskAssignmentsAddedEvent(TaskEvent event, AssignmentType type,
List<OrganizationalEntity> entities){};
    public default void afterTaskAssignmentsRemovedEvent(TaskEvent event, AssignmentType type,
List<OrganizationalEntity> entities){};

}
```

これらのメソッドのいずれかを実装することで、対応するイベントを処理することができます。

プロセスエンジンがメソッドに渡すイベントクラスの定義については、Java ドキュメントの **org.kie.api.task** パッケージを参照してください。

イベントクラスのメソッドを用いて、タスク、タスクコンテキスト、タスクメタデータを表すクラスを取得することができます。

66.6.2. イベントリスナーへの呼び出しのタイミング

イベントの **before** および **after** に多数のイベントリスナーの呼び出しがあります (例: **beforeNodeLeft()** および **afterNodeLeft()**、**beforeTaskActivatedEvent()** および **afterTaskActivatedEvent()**)。

通常、**before** イベントおよび **after** イベントの呼び出しは、スタックのように動作します。イベント A が直接イベント B を発生する場合は、以下の呼び出しシーケンスが実行されます。

- Before A
- Before B
- After B
- After A

たとえば、ノード X を残すと、ノード Y のトリガーに関連するすべてのイベント呼び出しが、ノード X の **beforeNodeLeft** 呼び出しと **afterNodeLeft** 呼び出しの間で実行されます。

同様に、プロセスを開始すると一部のノードが直接開始する場合に、すべての **nodeTriggered** イベントおよび **nodeLeft** イベント呼び出しは、**beforeProcessStarted** 呼び出しと **afterProcessStarted** 呼び出しの間に発生します。

このアプローチは、イベント間の原因と効果の関係を反映しています。ただし、イベントコール **後** のタイミングと順序は常に直感的には限りません。たとえば、**afterProcessStarted** 呼び出しは、プロセス内の一部のノードを **afterNodeLeft** 呼び出しの後に発生する可能性があります。

通常、特定のイベントの発生時に通知するには、**before** 呼び出しをイベントに使用します。たとえば、特定のプロセスインスタンスの開始に関連するすべての手順が完了したときに、このイベントに関連するすべての処理が終了する場合のみ、**after** の呼び出しを使用してください。

ノードのタイプによって、一部のノードは **nodeLeft** 呼び出しのみを生成する可能性があり、**nodeTriggered** 呼び出しのみを生成する可能性があります。たとえば、catch 中間イベントノードは、別のプロセスノードによってトリガーされないため、**nodeTriggered** 呼び出しを生成しません。同様に、throw 中間イベントノードには別のノードへの外向き接続がないため、これらのノードは **nodeLeft** 呼び出しを生成しません。

66.6.3. イベントリスナー開発のためのプラクティス

プロセスエンジンは、イベントやタスクの処理中にイベントリスナーを呼び出します。呼び出しは、プロセスエンジンのトランザクションとブロック実行の中で行われます。したがって、イベントリスナーはプロセスエンジンのロジックや性能に影響を与える可能性があります。

中断を最小限に抑えるため、以下のガイドラインに従ってください。

- どんなアクションでも、できるだけ短くしなければならない。
- リスナークラスは状態を持ってはいけない。プロセスエンジンは、いつでもリスナークラスを破棄し、再作成することができます。
- リスナーがリスナーメソッドのスコープ外に存在するリソースを変更する場合、そのリソースが現在のトランザクションに登録されていることを確認する。トランザクションがロールバックされる可能性があります。この場合、変更されたリソースがトランザクションの一部でない場合、リソースの状態の一貫性が失われます。
  Red Hat JBoss EAP が提供するデータベース関連のリソースは、常に現在のトランザクションに登録されます。それ以外の場合は、使用している実行環境の JTA 情報を確認してください。
- 異なるイベントリスナーの実行順序に依存するロジックは使用しない。
- リスナー内にプロセスエンジン外の異なるエンティティーとのインタラクションを含めない。例えば、イベント通知のための REST 呼び出しを含めないでください。代わりに、プロセスノードを使用してそのような呼び出しを完了します。例外はロギング情報の出力です。しかし、ロギングリスナーは可能な限りシンプルにする必要があります。
- リスナーを使用して、イベントに関与するプロセスやタスクの状態を変更することができる (例: その変数を変更する)。
- リスナーを使用して、プロセスエンジンと対話することができる (例: イベントに関与していないプロセスインスタンスにシグナルを送信する、プロセスインスタンスと対話する)。

66.6.4. イベントリスナーの登録

KieSession クラスは、以下のリストのようにイベントリスナーの登録、削除、および一覧表示を行うメソッドを提供する **RuleRuntimeEventManager** インターフェイスを実装します。

RuleRuntimeEventManager インターフェイスのメソッド

```
void addEventListener(AgendaEventListener listener);
void addEventListener(RuleRuntimeEventListener listener);
void removeEventListener(AgendaEventListener listener);
void removeEventListener(RuleRuntimeEventListener listener);
Collection<AgendaEventListener> getAgendaEventListeners();
Collection<RuleRuntimeEventListener> getRuleRintimeEventListeners();
```

ただし、典型的な場合には、これらのメソッドは使用しないでください。

RuntimeManager インターフェイスを使用している場合は、**RuntimeEnvironment** クラスを使用してイベントリスナーを登録します。

サービス API を使用している場合は、プロジェクトの **META-INF/services/org.jbpm.services.task.deadlines.NotificationListener** ファイルに、イベントリスナーの完全修飾クラス名を追加できます。また、Services API は、イベントの電子メール通知を送信できる **org.jbpm.services.task.deadlines.notifications.impl.email.EmailNotificationListener** などのデフォルトのリスナーも登録します。

デフォルトのリスナーを除外するには、リスナーの完全修飾名を JVM システムプロパティー **org.kie.jbpm.notification_listeners.exclude** に追加します。

66.6.5. KieRuntimeLogger イベントリスナー

KieServices パッケージには、KIE セッションに追加できる **KieRuntimeLogger** イベントリスナーが含まれます。このリスナーを使用して監査ログを作成できます。このログには、起動時に発生した異なるイベントがすべて含まれます。

注記

これらのロガーはデバッグの目的で使用されます。ビジネスレベルのプロセス分析では詳細すぎる可能性があります。

リスナーは以下のロガータイプを実装します。

- コンソールロガー: このロガーはすべてのイベントをコンソールに書き込みます。このロガーの完全修飾クラス名は **org.drools.core.audit.WorkingMemoryConsoleLogger** です。
- ファイルロガー: このロガーは XML 表現を使用してすべてのイベントをファイルに書き込みます。IDE でログファイルを使用して、実行時に発生したイベントのツリーベースの視覚化を生成できます。このロガーの完全修飾クラス名は **org.drools.core.audit.WorkingMemoryFileLogger** です。
  ファイルロガーは、ロガーを閉じるときや、ロガーのイベント数が事前定義レベルに達した場合にのみ、イベントをディスクに書き込みます。したがって、ランタイム時のプロセスのデバッグには適していません。
- スレッドファイルロガー: このロガーは、指定した時間間隔の後にイベントをファイルに書き込みます。このロガーを使用して、プロセスのデバッグ中に進捗をリアルタイムで視覚化することができます。このロガーの完全修飾クラス名は **org.drools.core.audit.ThreadedWorkingMemoryFileLogger** です。

ロガーの作成時に、KIE セッションを引数として渡す必要があります。ファイルロガーでは、ログファイルの名前を作成する必要もあります。スレッド化されたファイルロガーには、イベントが保存される間隔 (ミリ秒単位) が必要です。

アプリケーションの末尾で常にロガーを閉じます。

以下の例は、ファイルロガーの使用例を示しています。

ファイルロガーの使用

```
import org.kie.api.KieServices;
import org.kie.api.logger.KieRuntimeLogger;
...
KieRuntimeLogger logger = KieServices.Factory.get().getLoggers().newFileLogger(ksession, "test");
// add invocations to the process engine here,
// e.g. ksession.startProcess(processId);
...
logger.close();
```

ファイルベースのロガーによって作成されるログファイルには、プロセスのランタイム中に発生したすべてのイベントの XML ベースの概要が含まれます。

66.7. プロセスエンジンの設定

お使いの環境の要件に応じて、プロセスエンジンのデフォルト動作を変更するのに利用できる制御パラメーターをいくつか使用できます。

これらのパラメーターを JVM システムプロパティーとして設定します。通常、アプリケーションサーバーなどのプログラムを開始するときに **-D** オプションを使用します。

表66.2 コントロールパラメーター

名前	使用できる値	デフォルト値	説明
jbpm.ut.jndi.lookup	String		デフォルト名 (**java:comp/UserTransaction**) にアクセスできない場合に使用される代替 JNDI 名。 注記: 指定のランタイム環境に対して、名前が有効である必要があります。デフォルトのユーザートランザクション JNDI 名にアクセスできない場合は、この変数を使用しないでください。
jbpm.enable.multi.con	**true|false**	**false**	アクティビティーに対して複数の内向きおよび外向きのシーケンスフローサポートを有効にします。

名前	使用できる値	デフォルト値	説明
jbpm.business.calendar.properties	String	**/jbpm.business.calendar.properties**	ビジネスカレンダー設定ファイルの代替クラスパスの場所
jbpm.overdue.timer.delay	Long	2000	適切な初期化を可能にするための期限切れタイマーの遅延をミリ秒単位で指定します。
jbpm.process.name.comparator	String		名前でプロセスを開始できるようにする代替コンパレータークラス。デフォルトでは **NumberVersionComparator** コンパレーターが使用されます。
jbpm.loop.level.disabled	**true**|**false**	**true**	XOR ゲートウェイを使用する場合の高度なループサポートのループ反復追跡を有効または無効にします。
org.kie.mail.session	String	**mail/jbpmMailSession**	Task Deadlines が使用するメールセッションの代替 JNDI 名
jbpm.usergroup.callback.properties	String	**/jbpm.usergroup.callback.properties**	ユーザーグループコールバック実装の代替クラスパスの場所 (LDAP、DB)
jbpm.user.group.mapping	String	**${jboss.server.config.dir}/roles.properties**	**JBossUserGroupCallbackImpl** の **roles.properties** ファイルの場所
jbpm.user.info.properties	String	**/jbpm.user.info.properties**	ユーザー情報設定の代替クラスパスの場所 (**LDAPUserInfoImpl** で使用)
org.jbpm.ht.user.separator	String	**,**	ユーザータスクのアクターとグループの代替セパレーター

名前	使用できる値	デフォルト値	説明
org.quartz.properties	String		Quartz ベースのタイマーサービスをアクティブにする Quartz 設定ファイル の場所
jbpm.data.dir	String	利用可能な場合は **${jboss.server.data.dir}**。それ以外は **${java.io.tmpdir}**。	プロセスエンジンが生成したデータファイルを保存する場所
org.kie.executor.pool.size	Integer	**1**	プロセスエンジンエグゼキューターのスレッドプールサイズ
org.kie.executor.retry.count	Integer	3	エラーが発生した場合のプロセスエンジンエグゼキューターの再試行回数
org.kie.executor.interval	Integer	0	プロセスエンジンエグゼキューターが保留中のジョブをチェックする頻度 (秒単位)。値が **0** の場合は、エグゼキューターの起動時にチェックが 1 回実行されます。
org.kie.executor.disabled	**true**|**false**	**true**	プロセスエンジンエグゼキューターの無効化
org.kie.store.services.class	String	**org.drools.persistence.jpa.KnowledgeStoreServiceImpl**	**KieSession** インスタンスのブートストラップを行う **KieStoreServices** を実装するクラスの完全修飾名

名前	使用できる値	デフォルト値	説明
org.kie.jbpm.notification_listeners.exclude	String		他の方法で使用される場合でも除外する必要があるイベントリスナーの完全修飾名。複数の名前はコンマで区切ります。たとえば、**org.jbpm.services.task.deadlines.notifications.impl.email.EmailNotificationListener** を追加して、デフォルトのメール通知リスナーを除外できます。
org.kie.jbpm.notification_listeners.include	String		組み込む必要があるイベントリスナーの完全修飾名。複数の名前はコンマで区切ります。このプロパティーを設定すると、このプロパティーのリスナーのみが含まれ、他のすべてのリスナーは除外されます。

第67章 プロセスエンジンの永続性およびトランザクション

プロセスエンジンは、プロセスの状態に永続性を実装します。この実装では、SQL データベースバックエンドと JPA フレームワークを使用します。また、監査ログ情報をデータベースに保存することもできます。

プロセスエンジンは、トランザクションをサポートする永続バックエンドに依存する、JTA フレームワークを使用したプロセスのトランザクション実行も有効にします。

67.1. プロセスランタイム状態の永続性

プロセスエンジンは、実行中のプロセスインスタンスのランタイム状態の永続ストレージをサポートします。ランタイムの状態を保存するため、プロセスエンジンが停止したり、問題が発生した場合はプロセスインスタンスの実行を継続できます。

また、プロセスエンジンは、プロセス定義と現在のプロセス状態および以前のプロセス状態の履歴ログを永続的に格納します。

JPA フレームワークによって指定された **persistence.xml** ファイルを使用して、SQL データベースで永続性を設定できます。さまざまな永続ストラテジーをプラグインできます。**persistence.xml** ファイルの詳細は、「**persistence.xml** ファイルの設定」を参照してください。

デフォルトでは、プロセスエンジンで永続性を設定しないと、プロセスインスタンスの状態を含むプロセス情報は永続化されません。

プロセスエンジンがプロセスを開始すると、**プロセスインスタンス** が作成され、特定のコンテキストでのプロセスの実行を表します。たとえば、売上注文を処理するプロセスを実行する場合は、各営業リクエストに対してプロセスインスタンスが 1 つ作成されます。

プロセスインスタンスには、プロセス変数の現在の値など、プロセスの現在のランタイム状態とコンテキストが含まれます。ただし、この情報はプロセスの継続的な実行には必要ないため、プロセスの過去の状態の履歴に関する情報は含まれていません。

プロセスインスタンスのランタイム状態が永続化されると、プロセスエンジンが失敗したり停止した場合に、実行中のプロセスの実行状態をすべて復元できます。メモリーから特定のプロセスインスタンスを削除して、後で復元することもできます。

永続性を使用するようにプロセスエンジンを設定すると、ランタイム状態がデータベースに自動的に保存されます。コードで永続性をトリガーする必要はありません。

データベースからプロセスエンジンの状態を復元すると、すべてのインスタンスが自動的に最後に記録された状態に復元されます。プロセスインスタンスは、たとえば期限切れタイマー、プロセスインスタンスから要求されたタスクの完了、プロセスインスタンスに送信されたシグナルなど、トリガーされた場合に実行を自動的に再開します。別のインスタンスを読み込み、手動で実行をトリガーする必要はありません。

プロセスエンジンは、オンデマンドでプロセスインスタンスを自動的に再読み込みします。

67.1.1. 永続性のセーフポイント

プロセスエンジンは、プロセスの実行中に **安全な時点** でプロセスインスタンスのステータスを永続ストレージに保存します。

プロセスインスタンスが以前の待機状態から開始または再開すると、このプロセスエンジンは、他のアクションが実行するまで実行を継続します。これ以外のアクションを実行できない場合は、プロセスが完了するか、待機状態に達したことを意味します。プロセスに複数の並列パスが含まれる場合は、すべ

てのパスが待機状態である必要があります。

プロセスの実行におけるこのポイントは、安全なポイントと見なされます。この時点で、プロセスエンジンはプロセスインスタンスのステータスと、実行の影響を受けるその他のプロセスインスタンスのステータスを永続ストレージに保存します。

67.2. 永続的な監査ログ

プロセスエンジンは、インスタンスの過去の状態など、プロセスインスタンスの実行に関する情報を保存できます。

この情報は多くの場合で役に立ちます。たとえば、特定のプロセスインスタンスに対して実行されたアクションを確認したり、特定のプロセスの効率を監視して分析したりできます。

ただし、ランタイムデータベースに履歴情報を保存すると、データベースのサイズが急増し、永続レイヤーのパフォーマンスにも影響を及ぼします。そのため、履歴ログ情報は別々に保存されます。

プロセスエンジンは、プロセスの実行時に生成するイベントに基づいてログを作成します。イベントリスナーのメカニズムを使用してイベントを受け取り、必要な情報を抽出してから、この情報をデータベースに永続化します。**jbpm-audit** モジュールには、JPA を使用してデータベースにプロセス関連の情報を保存するイベントリスナーが含まれています。

フィルターを使用してログ情報の範囲を制限できます。

67.2.1. プロセスエンジン監査ログデータモデル

プロセスエンジンの監査ログ情報をクエリーして、異なるシナリオで使用することができます。たとえば、1 つの特定のプロセスインスタンスに対して履歴ログを作成したり、特定のプロセスの全インスタンスのパフォーマンスを分析したりできます。

監査ログデータモデルはデフォルトの実装です。ユースケースによっては、必要な情報を格納するための独自のデータモデルを定義することもできます。プロセスイベントリスナーを使用して情報を抽出できます。

データモデルには、プロセスインスタンス情報用のエンティティー、ノードインスタンス情報用のエンティティー、およびプロセス変数インスタンス情報用のエンティティーが含まれます。

ProcessInstanceLog テーブルには、プロセスインスタンスに関する基本的なログ情報が含まれます。

表67.1 ProcessInstanceLog テーブルフィールド

フィールド	説明	Null 許容型
id	ログエンティティーのプライマリーキーおよび ID	Null 不可
correlationKey	このプロセスインスタンスの相関	
duration	このプロセスインスタンスの開始日からの実際の時間	
end_date	プロセスインスタンスの終了日 (該当する場合)	

フィールド	説明	Null 許容型
externalId	一部の要素 (デプロイメント ID など) に関連付けるのに使用される任意の外部 ID	
user_identity	プロセスインスタンスを開始したユーザーの任意の識別子	
outcome	プロセスインスタンスの結果。プロセスインスタンスがエラーイベントで終了した場合に、このフィールドにはエラーコードが含まれます。	
parentProcessInstanceId	親プロセスインスタンスのプロセスインスタンス ID (該当する場合)	
processid	プロセス ID	
processinstanceid	プロセスインスタンス ID	Null 不可
processname	プロセスの名前	
processtype	インスタンスのタイプ (プロセスまたはケース)	
processversion	プロセスのバージョン	
sla_due_date	サービスレベルアグリーメント (SLA) に基づくプロセスの期日	
slaCompliance	SLA への準拠のレベル	
start_date	プロセスインスタンスの開始日	
status	プロセスインスタンスの状態にマップするプロセスインスタンスのステータス	

NodeInstanceLog テーブルには、各プロセスインスタンス内で実行されたノードに関する詳細情報が含まれます。ノードインスタンスが内向き接続のいずれかから入力したり、外向き接続のいずれかを介して終了するたびに、イベントに関する情報がこのテーブルに保存されます。

表67.2 NodeInstanceLog テーブルフィールド

フィールド	説明	Null 許容型
id	ログエンティティーのプライマリーキーおよび ID	Null 不可
connection	このノードインスタンスの原因となったシーケンスフローの実際の識別子	
log_date	イベントの日付	
externalId	一部の要素 (デプロイメント ID など) に関連付けるのに使用される任意の外部 ID	
nodeid	プロセス定義の対応するノードのノード ID	
nodeinstanceid	ノードインスタンス ID	
nodename	ノードの名前	
nodetype	ノードのタイプ	
processid	プロセスインスタンスが実行しているプロセスの ID	
processinstanceid	プロセスインスタンス ID	Null 不可
sla_due_date	サービスレベルアグリーメント (SLA) に基づくノードの期日	
slaCompliance	SLA への準拠のレベル	
type	イベントのタイプ (0 = enter、1 = exit)	Null 不可
workItemId	(任意、特定のノードタイプのみ) ワークアイテムの識別子	
nodeContainerId	ノードが埋め込みサブプロセスノード内にある場合のコンテナーの識別子	
referenceId	参照識別子	

フィールド	説明	Null 許容型
observation	ノードがスケジューリングされたイベントタイプである場合、元のノードのインスタンス ID およびジョブ ID。この情報を使用して、再度ジョブを起動させることができます。	

VariableInstanceLog テーブルには、変数インスタンスの変更に関する情報が含まれます。デフォルトでは、変数が値を変更すると、プロセスエンジンはログエントリーを生成します。プロセスエンジンは、変更前にエントリーをログに記録することもできます。

表67.3 VariableInstanceLog テーブルフィールド

フィールド	説明	Null 許容型
id	ログエンティティーのプライマリーキーおよび ID	Null 不可
externalId	一部の要素 (デプロイメント ID など) に関連付けるのに使用される任意の外部 ID	
log_date	イベントの日付	
processid	プロセスインスタンスが実行しているプロセスの ID	
processinstanceid	プロセスインスタンス ID	Null 不可
oldvalue	ログ作成時の変数の以前の値	
value	ログが作成された時点の変数の値	
variableid	プロセス定義の変数 ID	
variableinstanceid	変数インスタンスの ID	

AuditTaskImpl テーブルには、ユーザータスクに関する情報が含まれます。

表67.4 AuditTaskImpl テーブルフィールド

フィールド	説明	Null 許容型
id	タスクログエンティティーのプライマリーキーおよび ID	

フィールド	説明	Null 許容型
activationTime	このタスクがアクティベートされた時間	
actualOwner	このタスクに割り当てられている実際の所有者。この値は、所有者がタスクを要求した場合にのみ設定されます。	
createdBy	このタスクを作成したユーザー	
createdOn	タスクの作成日	
deploymentId	このタスクが含まれるデプロイメントの ID	
description	タスクの説明	
dueDate	このタスクに設定した期日	
name	タスクの名前	
parentId	親タスク ID	
priority	タスクの優先順位	
processId	このタスクが属するプロセス定義 ID	
processInstanceId	このタスクが関連付けられているプロセスインスタンス ID	
processSessionId	このタスクの作成に使用する KIE セッション ID	
status	タスクの現在の状態	
taskId	タスクの識別子	
workItemId	プロセス側でこのタスク ID に割り当てられたワークアイテムの識別子	
lastModificationDate	プロセスインスタンスの状態が永続データベースに最後に記録された日時	

BAMTaskSummary テーブルは、チャートとダッシュボードの構築に、BAM エンジンが使用するタスクに関する情報を収集します。

表67.5 BAMTaskSummary テーブルフィールド

フィールド	説明	Null 許容型
pk	ログエンティティーのプライマリーキーおよび ID	Null 不可
createdDate	タスクの作成日	
duration	タスクの作成後の期間	
endDate	タスクが終了状態に達した日付 (complete、exit、fail、skip)	
processinstanceid	プロセスインスタンス ID	
startDate	タスクの開始日	
status	タスクの現在の状態	
taskId	タスクの識別子	
taskName	タスクの名前	
userId	タスクに割り当てられたユーザー ID	
optlock	最適なロック値として機能する version フィールド	

TaskVariableImpl テーブルには、タスク変数インスタンスに関する情報が含まれます。

表67.6 TaskVariableImpl テーブルフィールド

フィールド	説明	Null 許容型
id	ログエンティティーのプライマリーキーおよび ID	Null 不可
modificationDate	変数が直近で変更された日	
name	タスクの名前	
processid	プロセスインスタンスが実行しているプロセスの ID	

フィールド	説明	Null 許容型
processinstanceid	プロセスインスタンス ID	
taskId	タスクの識別子	
type	変数のタイプ: タスクの入力または出力のいずれか	
value	変数値	

TaskEvent テーブルには、タスクインスタンスの変更に関する情報が含まれます。**claim**、**start**、**stop** などの操作は、指定のタスクに発生したイベントのタイムラインビューを提供するためにこの表に保存されます。

表67.7 TaskEvent テーブルフィールド

フィールド	説明	Null 許容型
id	ログエンティティーのプライマリーキーおよび ID	Null 不可
logTime	このイベントが保存された日付	
message	ログイベントメッセージ	
processinstanceid	プロセスインスタンス ID	
taskId	タスクの識別子	
type	イベントのタイプ。タイプはタスクのライフサイクルフェーズに対応します。	
userId	タスクに割り当てられたユーザー ID	
workItemId	タスクが割り当てられているワークアイテムの識別子	
optlock	最適なロック値として機能する version フィールド	
correlationKey	プロセスインスタンスの相関キー	
processType	プロセスインスタンスのタイプ (プロセスまたはケース)	

フィールド	説明	Null 許容型
currentOwner	タスクの現在のオーナー	

67.2.2. プロセスイベントログをデータベースに格納するための設定

デフォルトのデータモデルを使用してデータベースにプロセス履歴情報のログを記録するには、セッションでロガーを登録する必要があります。

KIE セッションでのロガーの登録

```
KieSession ksession = ...;
ksession.addProcessEventListener(AuditLoggerFactory.newInstance(Type.JPA, ksession, null));

// invoke methods for your session here
```

情報を保存するデータベースを指定するには、**persistence.xml** ファイルを変更して監査ログクラス (**ProcessInstanceLog**、**NodeInstanceLog**、および **VariableInstanceLog**) を追加する必要があります。

監査ログクラスが含まれる変更された persistence.xml ファイル

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<persistence
  version="2.0"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
  http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</class>

    <class>org.jbpm.process.audit.ProcessInstanceLog</class>
    <class>org.jbpm.process.audit.NodeInstanceLog</class>
    <class>org.jbpm.process.audit.VariableInstanceLog</class>

    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
```

```
      <property name="hibernate.connection.release_mode" value="after_transaction"/>
      <property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform"/>
    </properties>
  </persistence-unit>
</persistence>
```

67.2.3. プロセスイベントログを JMS キューに送信する設定

プロセスエンジンがデフォルトの監査ログ実装でデータベースにイベントを保存すると、データベース操作はプロセスインスタンスの実際の実行と同じトランザクション内で同期的に完了します。この操作には時間がかかり、負荷の高いシステムでは特に履歴ログとランタイムデータの両方が同じデータベースに格納される場合に、データベースのパフォーマンスに影響する可能性があります。

または、プロセスエンジンが提供する JMS ベースのロガーを使用できます。このロガーは、プロセスログエントリーをデータベースで直接永続化するのではなく、JMS キューにメッセージとして送信するように設定できます。

プロセスエンジントランザクションのロールバック時にデータの不整合を回避するために、JMS ロガーをトランザクションとして設定できます。

JMS 監査ロガーの使用

```
ConnectionFactory factory = ...;
Queue queue = ...;
StatefulKnowledgeSession ksession = ...;
Map<String, Object> jmsProps = new HashMap<String, Object>();
jmsProps.put("jbpm.audit.jms.transacted", true);
jmsProps.put("jbpm.audit.jms.connection.factory", factory);
jmsProps.put("jbpm.audit.jms.queue", queue);
ksession.addProcessEventListener(AuditLoggerFactory.newInstance(Type.JMS, ksession,
jmsProps));

// invoke methods one your session here
```

これは、JMS 監査ロガーの設定を可能にする方法の 1 つです。**AuditLoggerFactory** クラスを使用して、追加の設定パラメーターを設定できます。

67.2.4. 変数の監査

デフォルトでは、プロセス変数およびタスク変数の値は文字列表現として監査テーブルに保存されます。文字列以外の変数型の文字列表現を作成するには、プロセスエンジンが **variable.toString()** メソッドを呼び出します。変数にカスタムクラスを使用する場合は、このメソッドをクラスに実装できます。多くの場合は、この表現で十分です。

ただし、特にプロセス変数またはタスク変数による効率的なクエリーが必要な場合は、ログで文字列表現が不十分な場合があります。たとえば、変数の値として使用する **Person** オブジェクトは、以下の構造を持つ場合があります。

プロセスまたはタスク変数の値として使用される Person オブジェクトの例

```
public class Person implements Serializable {

    private static final long serialVersionUID = -5172443495317321032L;
```

```
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}
```

toString() メソッドは、人間が判読できる形式を提供します。ただし、検索には十分ではない場合があります。サンプル文字列の値は **Person [name="john", age="34"]** です。このような文字列を多数検索して 34 の年齢を見つけると、データベースクエリーが非効率になります。

より効率的な検索を有効にするには、**VariableIndexer** オブジェクトを使用して変数を監査し、監査ログのストレージの関連する部分を抽出します。

VariableIndexer インターフェイスの定義

```
/**
 * Variable indexer that transforms a variable instance into another representation (usually string)
 * for use in log queries.
 *
 * @param <V> type of the object that will represent the indexed variable
 */
public interface VariableIndexer<V> {

    /**
     * Tests if this indexer can index a given variable
     *
     * NOTE: only one indexer can be used for a given variable
     *
     * @param variable variable to be indexed
     * @return true if the variable should be indexed with this indexer
     */
    boolean accept(Object variable);
```

```
    /**
     * Performs an index/transform operation on the variable. The result of this operation can be
     * either a single value or a list of values, to support complex type separation.
     * For example, when the variable is of the type Person that has name, address, and phone fields,
     * the indexer could build three entries out of it to represent individual fields:
     * person = person.name
     * address = person.address.street
     * phone = person.phone
     * this configuration allows advanced queries for finding relevant entries.
     * @param name name of the variable
     * @param variable actual variable value
     * @return
     */
    List<V> index(String name, Object variable);
}
```

デフォルトのインデクサーは **toString()** メソッドを使用して、単一の変数の単一の監査エントリーを生成します。他のインデクサーは、単一の変数をインデックス化してオブジェクトの一覧を返すことができます。

Person タイプの効率的なクエリーを有効にするために、**Person** インスタンスを個別の監査エントリーにインデックス付けするカスタムインデクサーを構築できます。1 つは名前を表し、もう 1 つは年齢を表します。

Person タイプのインデクサーのサンプル

```
public class PersonTaskVariablesIndexer implements TaskVariableIndexer {

    @Override
    public boolean accept(Object variable) {
        if (variable instanceof Person) {
            return true;
        }
        return false;
    }

    @Override
    public List<TaskVariable> index(String name, Object variable) {

        Person person = (Person) variable;
        List<TaskVariable> indexed = new ArrayList<TaskVariable>();

        TaskVariableImpl personNameVar = new TaskVariableImpl();
        personNameVar.setName("person.name");
        personNameVar.setValue(person.getName());

        indexed.add(personNameVar);

        TaskVariableImpl personAgeVar = new TaskVariableImpl();
        personAgeVar.setName("person.age");
        personAgeVar.setValue(person.getAge()+"");

        indexed.add(personAgeVar);

        return indexed;
```

```
    }

}
```

プロセスエンジンは、**Person** タイプの場合は、このインデクサーを使用して値をインデックス化し、その他の変数はすべてデフォルトの **toString()** メソッドでインデックス化されます。これで、年齢が 34 のプロセスインスタンスまたはタスクについてクエリーするには、以下のクエリーを使用できます。

- 変数名: **person.age**
- 変数値: **34**

LIKE タイプのクエリーが使用されていないため、データベースサーバーはクエリーを最適化し、大量のデータに対して効率化できます。

カスタムインデクサー
プロセスエンジンは、プロセス変数とタスク変数の両方のインデクサーをサポートします。ただし、変数の監査ビューを表すさまざまなタイプのオブジェクトを生成する必要があるため、インデクサーに異なるインターフェイスを使用します。

以下のインターフェイスを実装してカスタムインデクサーを構築する必要があります。

- プロセス変数の場合: **org.kie.internal.process.ProcessVariableIndexer**
- タスク変数: **org.kie.internal.task.api.TaskVariableIndexer**

インターフェイスのいずれかに 2 つのメソッドを実装する必要があります。

- **accept**: タイプがこのインデクサーによって処理されるかどうかを示します。プロセスエンジンは、1つのインデクサーのみが指定の変数値をインデックス化できることを想定し、タイプを受け入れる最初のインデクサーを使用します。
- **index**: 値にインデックス化し、監査ログに含まれるオブジェクト (通常は文字列) を生成します。

インターフェイスを実装したら、この実装を JAR ファイルとしてパッケージ化し、以下のファイルのいずれかで実装の一覧を作成する必要があります。

- プロセス変数の場合は **META-INF/services/org.kie.internal.process.ProcessVariableIndexer** ファイル。これは、プロセス変数インデクサーの完全修飾クラス名を一覧表示 (1 行に 1 クラス名) します。
- タスク変数の場合は **META-INF/services/org.kie.internal.task.api.TaskVariableIndexer** ファイル。タスク変数インデクサーの完全修飾クラス名を一覧表示 (1 行に 1 クラス名) します。

ServiceLoader のメカニズムは、これらのファイルを使用してインデクサーを検出します。プロセスまたはタスク変数をインデックス化する場合、プロセスエンジンは登録されたインデクサーを調べ、変数の値を受け入れるインデクサーを見つけます。他のインデクサーがこの値を受け入れない場合、プロセスエンジンは **toString()** メソッドを使用するデフォルトのインデクサーを適用します。

67.3. プロセスエンジンのトランザクション

プロセスエンジンは、Java Transaction API (JTA) トランザクションをサポートします。

プロセスエンジンの現在のバージョンは、純粋なローカルトランザクションをサポートしません。

アプリケーション内にトランザクション境界を指定しないと、プロセスエンジンは、別のトランザクションでプロセスエンジンで各メソッド呼び出しを自動的に実行します。

必要に応じて、アプリケーションコードでトランザクション境界を指定して、複数のコマンドを 1 つのトランザクションに統合できます。

67.3.1. トランザクションマネージャーの登録

ユーザー定義トランザクションを使用するには、環境でトランザクションマネージャーを登録する必要があります。

以下のサンプルコードはトランザクションマネージャーを登録し、JTA 呼び出しを使用してトランザクション境界を指定します。

トランザクションマネージャーの登録およびトランザクションの使用

```
// Create the entity manager factory
EntityManagerFactory emf =
EntityManagerFactoryManager.get().getOrCreate("org.jbpm.persistence.jpa");
TransactionManager tm = TransactionManagerServices.getTransactionManager();

// Set up the runtime environment
RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
.newDefaultBuilder()
.addAsset(ResourceFactory.newClassPathResource("MyProcessDefinition.bpmn2"),
ResourceType.BPMN2)
    .addEnvironmentEntry(EnvironmentName.TRANSACTION_MANAGER, tm)
    .get();

// Get the KIE session
RuntimeManager manager =
RuntimeManagerFactory.Factory.get().newPerRequestRuntimeManager(environment);
RuntimeEngine runtime = manager.getRuntimeEngine(ProcessInstanceIdContext.get());
KieSession ksession = runtime.getKieSession();

// Start the transaction
UserTransaction ut = InitialContext.doLookup("java:comp/UserTransaction");
ut.begin();

// Perform multiple commands inside one transaction
ksession.insert( new Person( "John Doe" ) );
ksession.startProcess("MyProcess");

// Commit the transaction
ut.commit();
```

UserTransaction、**TransactionManager**、**TransactionSynchronizationRegistry** などのトランザクション関連のオブジェクトが JNDI に登録されているため、root クラスパスに **jndi.properties** ファイルを指定して JNDI **InitialContextFactory** オブジェクトを作成する必要があります。

プロジェクトに **jbpm-test** モジュールが含まれる場合、このファイルはすでにデフォルトで含まれています。

それ以外の場合は、以下の内容で **jndi.properties** ファイルを作成してください。

jndi.properties ファイルの内容

```
java.naming.factory.initial=org.jbpm.test.util.CloseSafeMemoryContextFactory
org.osjava.sj.root=target/test-classes/config
org.osjava.jndi.delimiter=/
org.osjava.sj.jndi.shared=true
```

この設定では、**simple-jndi:simple-jndi** アーティファクトがプロジェクトのクラスパスにあることを前提としています。異なる JNDI 実装を使用することもできます。

デフォルトでは、Narayana JTA トランザクションマネージャーが使用されます。別の JTA トランザクションマネージャーを使用する場合は、**persistence.xml** ファイルを変更して必要なトランザクションマネージャーを使用できます。たとえば、アプリケーションが Red Hat JBoss EAP バージョン 7 以降で実行する場合は、JBoss トランザクションマネージャーを使用できます。この場合は、**persistence.xml** ファイルのトランザクションマネージャープロパティーを変更します。

JBoss トランザクションマネージャーの persistence.xml ファイルのトランザクションマネージャープロパティー

```
<property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform" />
```

警告

JTA トランザクション (**UserTransaction** または CMT) で **RuntimeManager** クラスの Singleton ストラテジーを使用すると競合状態が作成されます。この競合状態により、**Process instance XXX is disconnected** と同様のメッセージを含む **IllegalStateException** 例外が発生する場合があります。

この競合状態を回避するには、ユーザーアプリケーションコードでトランザクションを呼び出す際に、**KieSession** インスタンスを明示的に同期します。

```
synchronized (ksession) {
   try {
      tx.begin();

      // use ksession
      // application logic

      tx.commit();
   } catch (Exception e) {
      //...
   }
}
```

67.3.2. コンテナー管理トランザクションの設定

EJB Bean などの CMT (container-managed transaction) モードで実行するアプリケーションにプロセスエンジンを埋め込む場合は、追加の設定を完了する必要があります。この設定は、アプリケーション

が CMT アプリケーションが JNDI から **UserTransaction** インスタンス (WebSphere Application Server など) にアクセスできないアプリケーションサーバーで実行される場合に特に重要になります。

プロセスエンジンのデフォルトトランザクションマネージャーの実装は、**UserTransaction** を使用してトランザクションの状態をクエリーし、ステータスを使用してトランザクションを開始するかどうかを判断します。**UserTransaction** インスタンスにアクセスできない環境では、この実装は失敗します。

CMT 環境で適切な実行を有効にするために、プロセスエンジンは専用のトランザクションマネージャーの実装 (**org.jbpm.persistence.jta.ContainerManagedTransactionManager**) を提供します。このトランザクションマネージャーはトランザクションがアクティブであることを想定し、**getStatus()** メソッドが呼び出されると常に **ACTIVE** を返します。トランザクションマネージャーはコンテナー管理トランザクションモードでこれらの操作に影響を与えることができないため、**begin**、**commit**、**rollback** などの操作は操作できません。

注記

プロセス中にコードはエンジンによって発生した例外をコンテナーに伝播し、コンテナーがトランザクションを必要に応じてロールバックするようにする必要があります。

このトランザクションマネージャーを設定するには、以下の手順を行います。

手順

1. コードで、セッションを作成または読み込む前に、トランザクションマネージャーと永続コンテキストマネージャーを環境に挿入します。

   トランザクションマネージャーと永続コンテキストマネージャーの環境への挿入

   ```
   Environment env = EnvironmentFactory.newEnvironment();
   env.set(EnvironmentName.ENTITY_MANAGER_FACTORY, emf);
   env.set(EnvironmentName.TRANSACTION_MANAGER, new
   ContainerManagedTransactionManager());
   env.set(EnvironmentName.PERSISTENCE_CONTEXT_MANAGER, new
   JpaProcessPersistenceContextManager(env));
   env.set(EnvironmentName.TASK_PERSISTENCE_CONTEXT_MANAGER, new
   JPATaskPersistenceContextManager(env));
   ```

2. **persistence.xml** ファイルで、JPA プロバイダーを設定します。以下の例では、**hibernate** および WebSphere Application Server を使用します。

   persistence.xml ファイルでの JPA プロバイダーの設定

   ```
   <property name="hibernate.transaction.factory_class"
   value="org.hibernate.transaction.CMTTransactionFactory"/>
   <property name="hibernate.transaction.jta.platform"
   value="org.hibernate.service.jta.platform.internal.WebSphereJtaPlatform"/>
   ```

3. KIE セッションを破棄する場合は、直接破棄しないでください。代わりに、**org.jbpm.persistence.jta.ContainerManagedTransactionDisposeCommand** コマンドを実行します。このコマンドは、現在のトランザクションの完了時にセッションが確実に破棄されるようにします。以下の例の **ksession** は、破棄する **KieSession** オブジェクトです。

   ContainerManagedTransactionDisposeCommand コマンドを使用した KIE セッションの破棄

```
ksession.execute(new ContainerManagedTransactionDisposeCommand());
```

プロセスエンジンは、トランザクションの同期を登録してセッション状態をクリーンアップするため、セッションを直接破棄すると、トランザクションの完了時に例外が発生します。

67.3.3. トランザクションの再試行

プロセスエンジンがトランザクションのコミット時に別のトランザクションが同時にコミットされることが原因で、コミット操作に失敗することがあります。この場合、プロセスエンジンはトランザクションを再試行する必要があります。

再試行に複数回失敗すると、トランザクションは完全に失敗します。

JVM システムプロパティーを使用して、再試行プロセスを制御できます。

表67.8 トランザクションのコミットを再試行するシステムプロパティー

プロパティー	値	デフォルト	説明
org.kie.optlock.retries	Integer	5	このプロパティーでは、完全に失敗となる前に、プロセスエンジンがトランザクションを再試行する回数を記述します。
org.kie.optlock.delay	Integer	50	最初の再試行までの時間 (ミリ秒単位)。
org.kie.optlock.delayFactor	Integer	4	後続の再試行ごとに遅延時間を長くする乗数。デフォルト値の場合には、プロセスエンジンは、1回目の再試行まで 50 ミリ秒、2 番目の再試行まで 200 ミリ秒、3 番目の再試行まで 800 ミリ秒待機します。

67.4. プロセスエンジンでの永続性の設定

永続性を設定せずにプロセスエンジンを使用する場合は、ランタイムデータをデータベースに保存しません。インメモリーデータベースはデフォルトで利用できません。パフォーマンス上の理由で必要な場合や永続性を独自に管理する場合に、このモードを使用できます。

プロセスエンジンで JPA 永続性を使用するには、これを設定する必要があります。

設定には、通常、必要な依存関係の追加、データソースの設定、および永続性が設定されたプロセスエンジンクラスの作成が必要になります。

67.4.1. persistence.xml ファイルの設定

JPA 永続性を使用するには、**persistence.xml** 永続性設定をクラスパスに追加して、Hibernate および H2 データベース (または任意の他のデータベース) を使用するように JPA を設定する必要があります。このファイルをプロジェクトの **META-INF** ディレクトリーに配置します。

persistence.xml サンプルファイル

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence
```

```
    version="2.0"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
    http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</class>

    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.connection.release_mode" value="after_transaction"/>
          <property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform"/>
    </properties>
  </persistence-unit>
</persistence>
```

この例では **jdbc/jbpm-ds** データソースを参照します。データソースの設定方法については、「プロセスエンジンの永続性のデータソースの設定」を参照してください。

67.4.2. プロセスエンジンの永続性のデータソースの設定

プロセスエンジンで JPA 永続性を設定するには、データベースバックエンドを表すデータソースを指定する必要があります。

Red Hat JBoss EAP などのアプリケーションサーバーでアプリケーションを実行する場合、たとえば、データソース設定ファイルを **deploy** ディレクトリーに追加することで、アプリケーションサーバーを使用してデータソースを設定できます。データソースの作成方法は、アプリケーションサーバーのドキュメントを参照してください。

アプリケーションを Red Hat JBoss EAP にデプロイする場合は、**deploy** ディレクトリーに設定ファイルを作成してデータソースを作成できます。

Red Hat JBoss EAP のデータソース設定ファイルの例

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/jbpm-ds</jndi-name>
    <connection-url>jdbc:h2:tcp://localhost/~/test</connection-url>
    <driver-class>org.h2.jdbcx.JdbcDataSource</driver-class>
```

```
    <user-name>sa</user-name>
    <password></password>
  </local-tx-datasource>
</datasources>
```

アプリケーションがプレーンな Java 環境で実行される場合は、Red Hat Process Automation Manager が提供する **kie-test-util** モジュールの **DataSourceFactory** クラスを使用すると、Narayana および Tomcat DBCP を使用できます。以下のコードフラグメントを参照してください。この例では、Narayana および Tomcat DBCP と H2 インメモリーデータベースを使用します。

H2 インメモリーデータベースデータソースの設定コード例

```
Properties driverProperties = new Properties();
driverProperties.put("user", "sa");
driverProperties.put("password", "sa");
driverProperties.put("url", "jdbc:h2:mem:jbpm-db;MVCC=true");
driverProperties.put("driverClassName", "org.h2.Driver");
driverProperties.put("className", "org.h2.jdbcx.JdbcDataSource");
PoolingDataSourceWrapper pdsw = DataSourceFactory.setupPoolingDataSource("jdbc/jbpm-ds",
driverProperties);
```

67.4.3. 永続性の依存関係

永続性には、特定の JAR アーティファクトの依存関係が必要です。

jbpm-persistence-jpa.jar ファイルが常に必要になります。このファイルには、必要に応じてランタイム状態を保存するコードが含まれます。

使用している永続ソリューションおよびデータベースによっては、追加の依存関係が必要になる場合があります。デフォルト設定の組み合わせには、以下のコンポーネントが含まれます。

- Hibernate を JPA 永続プロバイダーとする
- H2 インメモリーデータベース
- JTA ベースのトランザクション管理用の Narayana
- 接続プール機能用の Tomcat DBCP

この設定には、以下の追加の依存関係が必要です。

- **jbpm-persistence-jpa** (**org.jbpm**)
- **drools-persistence-jpa** (**org.drools**)
- **persistence-api** (**javax.persistence**)
- **hibernate-entitymanager** (**org.hibernate**)
- **hibernate-annotations** (**org.hibernate**)
- **hibernate-commons-annotations** (**org.hibernate**)
- **hibernate-core** (**org.hibernate**)
- **commons-collections** (**commons-collections**)

- **dom4j** (**org.dom4j**)
- **jta** (**javax.transaction**)
- **narayana-jta** (**org.jboss.narayana.jta**)
- **tomcat-dbcp** (**org.apache.tomcat**)
- **jboss-transaction-api_1.2_spec** (**org.jboss.spec.javax.transaction**)
- **javassist** (**javassist**)
- **slf4j-api** (**org.slf4j**)
- **slf4j-jdk14** (**org.slf4j**)
- **simple-jndi** (**simple-jndi**)
- **h2** (**com.h2database**)
- **jbpm-test** (**org.jbpm**) はテスト専用で、実稼働アプリケーションにこのアーティファクトを含めないでください。

67.4.4. 永続性のある KIE セッションの作成

コードで KIE セッションを直接作成した場合は、**JPAKnowledgeService** クラスを使用して KIE セッションを作成できます。この方法では、基礎となる設定への完全アクセスが可能です。

手順

1. KIE ベース、KIE セ ッション設定 (必要な場合)、および環境に基づいて、**JPAKnowledgeService** クラスを使用して KIE セッションを作成します。環境には、永続性に使用する Entity Manager Factory への参照が含まれている必要があります。

   永続性のある KIE セッションの作成

   ```
   // create the entity manager factory and register it in the environment
   EntityManagerFactory emf =
       Persistence.createEntityManagerFactory( "org.jbpm.persistence.jpa" );
   Environment env = KnowledgeBaseFactory.newEnvironment();
   env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );

   // create a new KIE session that uses JPA to store the runtime state
   StatefulKnowledgeSession ksession =
   JPAKnowledgeService.newStatefulKnowledgeSession( kbase, null, env );
   int sessionId = ksession.getId();

   // invoke methods on your method here
   ksession.startProcess( "MyProcess" );
   ksession.dispose();
   ```

2. 特定のセッション ID に基づいてデータベースからセッションを再作成するには、**JPAKnowledgeService.loadStatefulKnowledgeSession()** メソッドを使用します。

   永続データベースからの KIE セッションの再作成

```
// re-create the session from database using the sessionId
ksession = JPAKnowledgeService.loadStatefulKnowledgeSession(sessionId, kbase, null, env
);
```

67.4.5. ランタイムマネージャーの永続性

コードで **RuntimeManager** クラスを使用する場合は、**RuntimeEnvironmentBuilder** クラスを使用して永続性の環境を設定します。デフォルトでは、ランタイムマネージャーは **org.jbpm.persistence.jpa** 永続ユニットを検索します。

以下の例は、空のコンテキストで **KieSession** を作成します。

ランタイムマネージャーを使用した空のコンテキストでの KIE セッションの作成

```
RuntimeEnvironmentBuilder builder = RuntimeEnvironmentBuilder.Factory.get()
        .newDefaultBuilder()
        .knowledgeBase(kbase);
RuntimeManager manager = RuntimeManagerFactory.Factory.get()
        .newSingletonRuntimeManager(builder.get(), "com.sample:example:1.0");
RuntimeEngine engine = manager.getRuntimeEngine(EmptyContext.get());
KieSession ksession = engine.getKieSession();
```

上の例では、**kbase** パラメーターとして KIE ベースが必要です。クラスパスで **kmodule.xml** KJAR 記述子を使用して、KIE ベースを設定できます。

KJAR 記述子 kmodule.xml からの KIE ベースの構築

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();
KieBase kbase = kContainer.getKieBase("kbase");
```

kmodule.xml 記述子ファイルには、スキャンしてプロセスエンジンのワークフローを検索およびデプロイするリソースパッケージの属性を含めることができます。

kmodule.xml 記述子ファイルのサンプル

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="kbase" packages="com.sample"/>
</kmodule>
```

永続性を制御するには、**RuntimeEnvironmentBuilder::entityManagerFactory** メソッドを使用できます。

ランタイムマネージャーでの永続性の設定の制御

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("org.jbpm.persistence.jpa");

RuntimeEnvironment runtimeEnv = RuntimeEnvironmentBuilder.Factory
        .get()
        .newDefaultBuilder()
        .entityManagerFactory(emf)
        .knowledgeBase(kbase)
        .get();
```

```
StatefulKnowledgeSession ksession = (StatefulKnowledgeSession)
RuntimeManagerFactory.Factory.get()
        .newSingletonRuntimeManager(runtimeEnv)
        .getRuntimeEngine(EmptyContext.get())
        .getKieSession();
```

この例では、KIE セッション **ksession** を作成したら、**ksession** でメソッドを呼び出すことができます (例: **StartProcess()**)。プロセスエンジンは、設定されたデータソースのランタイム状態を永続化します。

プロセスインスタンス ID を使用して永続ストレージからプロセスインスタンスを復元できます。ランタイムマネージャーが必要なセッションを自動的に再作成します。

プロセスインスタンス ID を使用した永続データベースからの KIE セッションの再作成

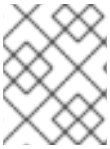
```
RuntimeEngine runtime =
manager.getRuntimeEngine(ProcessInstanceIdContext.get(processInstanceId));

KieSession session = runtime.getKieSession();
```

67.5. RED HAT PROCESS AUTOMATION MANAGER の個別のデータベーススキーマにおけるプロセス変数の永続化

プロセス変数を作成して、定義したプロセス内で使用する場合に、Red Hat Process Automation Manager はこれらのプロセス変数を、デフォルトのデータベーススキーマにバイナリーデータとして保存します。別のデータベーススキーマでプロセス変数を永続化して、プロセスデータの管理と実装に柔軟性をもたせることができます。

たとえば、別のデータベーススキーマで、プロセス変数を永続化すると、以下のタスクを行うのに役立ちます。

- 人間が解読可能な形式でのプロセス変数を管理する
- Red Hat Process Automation Manager 外のサービスに対して変数を使用可能にする
- プロセス変数データを損失せずに Red Hat Process Automation Manager のデフォルトのデータベーステーブルのログを消去する



注記

この手順は、プロセス変数にのみ適用されます。この手順では、ケース変数には適用されません。

前提条件

- 変数の実装先の Red Hat Process Automation Manager でプロセスを定義している。
- Red Hat Process Automation Manager 外部のデータベーススキーマで変数を永続化する場合は、データソースと、使用するデータベーススキーマを別に作成している。データソース作成の詳細は、**Business Central 設定とプロパティーの設定** を参照してください。

手順

1. プロセス変数として使用するデータオブジェクトファイルで、以下の要素を追加して変数の永続性を設定します。

変数を永続化するように設定した Person.java オブジェクトの例

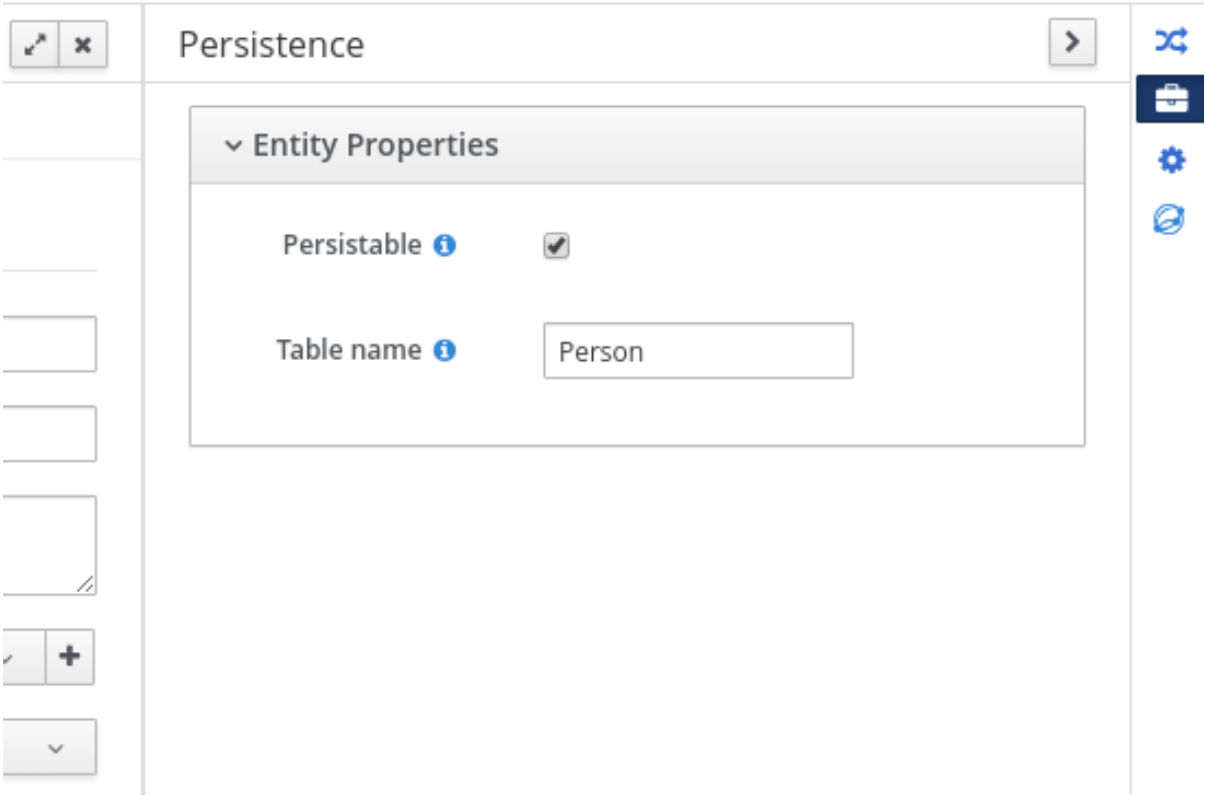
```
@javax.persistence.Entity (1)
@javax.persistence.Table(name = "Person") (2)
public class Person extends org.drools.persistence.jpa.marshaller.VariableEntity (3)
implements java.io.Serializable { (4)

 static final long serialVersionUID = 1L;

 @javax.persistence.GeneratedValue(strategy = javax.persistence.GenerationType.AUTO,
generator = "PERSON_ID_GENERATOR")
 @javax.persistence.Id (5)
 @javax.persistence.SequenceGenerator(name = "PERSON_ID_GENERATOR",
sequenceName = "PERSON_ID_SEQ")
 private java.lang.Long id;

 private java.lang.String name;

 private java.lang.Integer age;

 public Person() {
 }

 public java.lang.Long getId() {
  return this.id;
 }

 public void setId(java.lang.Long id) {
  this.id = id;
 }

 public java.lang.String getName() {
  return this.name;
 }

 public void setName(java.lang.String name) {
  this.name = name;
 }

 public java.lang.Integer getAge() {
  return this.age;
 }

 public void setAge(java.lang.Integer age) {
  this.age = age;
 }

 public Person(java.lang.Long id, java.lang.String name,
   java.lang.Integer age) {
  this.id = id;
  this.name = name;
  this.age = age;
```

```
    }

}
```

1. データオブジェクトを永続エンティティーとして設定します。
2. データオブジェクトに使用するデータベーステーブル名を定義します。
3. このデータオブジェクトと関連のあるプロセスインスタンスの関係を管理する **MappedVariable** マッピングテーブルを別に作成します。関係の管理が必要ない場合は、**VariableEntity** クラスを継承する必要はありません。この継承がない場合、データオブジェクトは永続化されますが、追加のデータは含まれません。
4. 並列化可能なオブジェクトとしてデータオブジェクトを設定します。
5. オブジェクトの永続 ID を設定します。

Business Central を使用して、データオブジェクトを永続化するには、プロジェクトのデータオブジェクトファイルに移動し、ウィンドウの右上隅の **Persistence** アイコンをクリックして、永続性の動作を設定します。

図67.1 Business Central での永続性設定



2. プロジェクトの **pom.xml** ファイルで、永続性のサポートを提供するために、以下の依存関係を追加します。この依存関係には、データオブジェクトで設定した **VariableEntity** クラスが含まれます。

永続性のプロジェクトの依存関係

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-persistence-jpa</artifactId>
```

```
  <version>${rhpam.version}</version>
  <scope>provided</scope>
</dependency>
```

3. プロジェクトの ~/**META-INF/kie-deployment-descriptor.xml** ファイルで、JPA マーシャリングストラテジーと、マーシャラーで使用する永続ユニットを設定します。オブジェクトをエンティティーとして定義するには、JPA マーシャリングストラテジーと永続ユニットが必要です。

kie-deployment-descriptor.xml ファイルで設定する JPA マーシャラーと永続ユニット

```
<marshalling-strategy>
  <resolver>mvel</resolver>
  <identifier>new
org.drools.persistence.jpa.marshaller.JPAPlaceholderResolverStrategy("myPersistenceUnit",
classLoader)</identifier>
  <parameters/>
</marshalling-strategy>
```

4. プロジェクトの ~/**META-INF** ディレクトリーで、**persistence.xml** ファイルを作成し、プロセス変数を永続化するデータソースを指定します。

データソース設定を含む persistence.xml ファイルの例

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_2_0.xsd">
   <persistence-unit name="myPersistenceUnit" transaction-type="JTA">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source> (1)
      <class>org.space.example.Person</class>
      <exclude-unlisted-classes>true</exclude-unlisted-classes>
      <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect"/>
         <property name="hibernate.max_fetch_depth" value="3"/>
         <property name="hibernate.hbm2ddl.auto" value="update"/>
         <property name="hibernate.show_sql" value="true"/>
         <property name="hibernate.id.new_generator_mappings" value="false"/>
         <property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform"/>
      </properties>
   </persistence-unit>
</persistence>
```

(1) データソースを設定して、プロセス変数を永続化します。

Business Central を使用してマーシャリングストラテジー、永続ユニット、データソースを設

Business Central を使用してマーシャリングストラテジー、永続ユニット、データソースを設定するには、プロジェクトの **Settings** → **Deployments** → **Marshalling Strategies** に移動し、プロジェクトの **Settings** → **Persistence** に移動します。

図67.2 Business Central での JPA マーシャラー設定

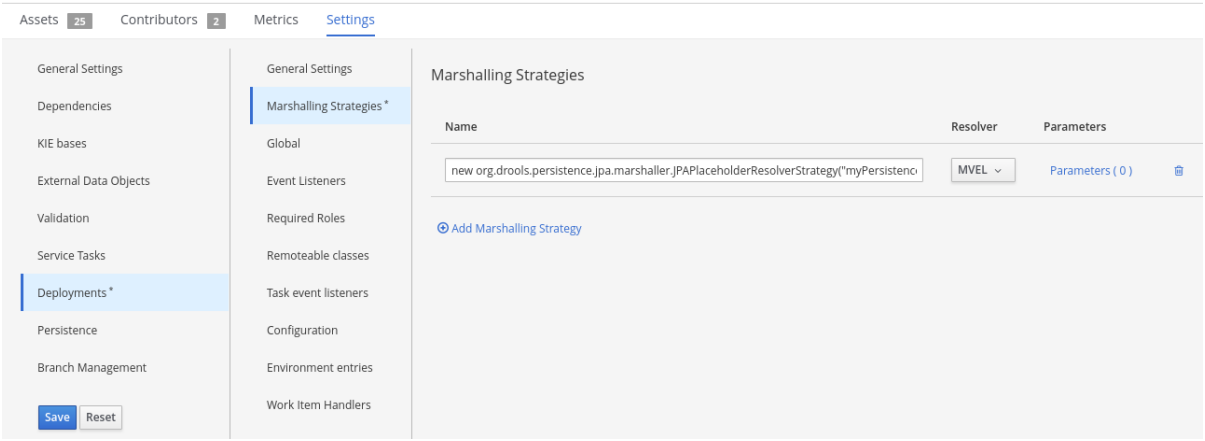
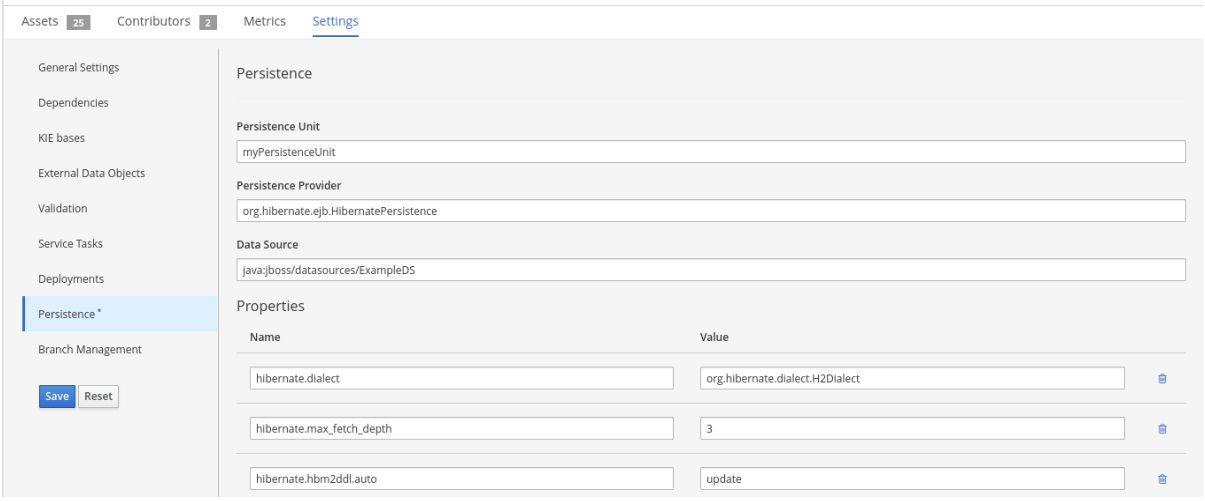


図67.3 Business Central での永続ユニットとデータソース設定



第68章 JAVA フレームワークとの統合

プロセスエンジンは、Apache Maven、CDI、Spring、EJB などの業界標準の Java フレームワークと統合できます。

68.1. APACHE MAVEN との統合

プロセスエンジンは、2 つの主な目的で Maven を使用します。

- KJAR アーティファクト (プロセスエンジンが実行するためのランタイム環境にインストールできるデプロイメントユニット) を作成する
- プロセスエンジンが組み込まれたアプリケーションを構築する依存関係を管理するには

68.1.1. デプロイメントユニットとしての Maven アーティファクト

プロセスエンジンは、Apache Maven アーティファクトからプロセスをデプロイするメカニズムを提供します。これらのアーティファクトは JAR ファイル形式で、**KJAR ファイル** (非公式には **KJAR**) として知られています。KJAR ファイルには、KIE ベースおよび KIE セッションを定義する記述子が含まれています。また、プロセスエンジンは KIE ベースに読み込めるプロセス定義を含むビジネスアセットも含まれます。

KJAR ファイルの記述子は、**kie-deployment-descriptor.xml** という名前の XML ファイルで表されます。記述子は空にすることができます。この場合は、デフォルト設定が適用されます。KIE ベースおよび KIE セッションのカスタム設定も提供できます。

空の kie-deployment-descriptor.xml 記述子

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<deployment-descriptor xsi:schemaLocation="http://www.jboss.org/jbpm deployment-descriptor.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <persistence-unit>org.jbpm.domain</persistence-unit>
  <audit-persistence-unit>org.jbpm.domain</audit-persistence-unit>
  <audit-mode>JPA</audit-mode>
  <persistence-mode>JPA</persistence-mode>
  <runtime-strategy>SINGLETON</runtime-strategy>
  <marshalling-strategies/>
  <event-listeners/>
  <task-event-listeners/>
  <globals/>
  <work-item-handlers />
  <environment-entries/>
  <configurations/>
  <required-roles/>
  <remoteable-classes/>
</deployment-descriptor>
```

空の **kie-deployment-descriptor.xml** 記述子を使用すると、以下のデフォルト設定が適用されます。

- 1 つのデフォルト KIE ベースが、以下の特性で作成されます。
  - KJAR ファイル内のすべてのパッケージからのアセットをすべて含む
  - そのイベント処理モードは **cloud** に設定されている

    - この等価動作が **identity** に設定されている
    - 宣言型アジェンダが無効である
    - CDI アプリケーションの場合、そのスコープは **ApplicationScope** に設定されます。
- 単一のデフォルトステートレス KIE セッションが作成され、以下の特徴があります。
    - 単一の KIE ベースにバインドされます
    - クロックタイプは、**real time** に設定される
    - CDI アプリケーションの場合、そのスコープは **ApplicationScope** に設定されます。
- 単一のデフォルトのステートフル KIE セッションが作成され、以下の特徴があります。
    - 単一の KIE ベースにバインドされます
    - クロックタイプは、**real time** に設定される
    - CDI アプリケーションの場合、そのスコープは **ApplicationScope** に設定されます。

デフォルトを使用しない場合は、**kie-deployment-descriptor.xml** ファイルを使用してすべての設定を変更することができます。このファイルに対するすべての要素の完全な仕様は、XSD スキーマ にあります。

以下の例は、ランタイムエンジンを設定するカスタムの **kie-deployment-descriptor.xml** ファイルを示しています。この例では、最も一般的なオプションを設定し、1 つのワークアイテムハンドラーが含まれます。**kie-deployment-descriptor.xml** ファイルを使用して他のオプションを設定することもできます。

カスタム kie-deployment-descriptor.xml ファイルのサンプル

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<deployment-descriptor xsi:schemaLocation="http://www.jboss.org/jbpm deployment-descriptor.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <persistence-unit>org.jbpm.domain</persistence-unit>
  <audit-persistence-unit>org.jbpm.domain</audit-persistence-unit>
  <audit-mode>JPA</audit-mode>
  <persistence-mode>JPA</persistence-mode>
  <runtime-strategy>SINGLETON</runtime-strategy>
  <marshalling-strategies/>
  <event-listeners/>
  <task-event-listeners/>
  <globals/>
  <work-item-handlers>
    <work-item-handler>
      <resolver>mvel</resolver>
      <identifier>new org.jbpm.process.workitem.bpmn2.ServiceTaskHandler(ksession,
classLoader)</identifier>
      <parameters/>
      <name>Service Task</name>
    </work-item-handler>
  </work-item-handlers>
  <environment-entries/>
  <configurations/>
```

```
    <required-roles/>
    <remoteable-classes/>
</deployment-descriptor>
```

注記

RuntimeManager クラスを使用する場合、このクラスは **KieContainer** クラスではなく **KieSession** インスタンスを作成します。ただし、**kie-deployment-descriptor.xml** モデルは常に構築プロセスのベースとして使用されます。**KieContainer** クラスは常に **KieBase** インスタンスを作成します。

GAV (グループ、アーティファクト、バージョン) の値を使用して、他の Maven アーティファクトなどの KJAR アーティファクトを参照できます。KJAR ファイルからユニットをデプロイする場合、プロセスエンジンは GAV 値を KIE API のリリース ID として使用します。GAV 値を使用して、KJAR アーティファクトをランタイム環境 (例: KIE Server) にデプロイできます。

68.1.2. Maven を使用した依存関係管理

プロセスエンジンを組み込むプロジェクトをビルドする場合は、Apache Maven を使用してプロセスエンジンが必要とするすべての依存関係を設定します。

プロセスエンジンは、アーティファクトの依存関係を宣言するための一連の BOM (Bills of Material) を提供します。

プロジェクトの上部の **pom.xml** ファイルを使用して、以下の例のように、プロセスエンジンを埋め込む依存関係管理を定義します。この例には、アプリケーションがアプリケーションサーバーにデプロイされているか、サーブレットコンテナーにデプロイされているか、スタンドアロンアプリケーションとしてデプロイされているかに関係なく適用できる、メインのランタイム依存関係が含まれています。

この例には、プロセスエンジンを使用するアプリケーションが一般的に必要とするコンポーネントのバージョンプロパティーも含まれています。必要に応じて、コンポーネントおよびバージョンの一覧を調整します。製品チームが Github リポジトリーの親の **pom.xml** ファイル でテストしたサードパーティーの依存関係バージョンを確認することができます。

プロセスエンジンの埋め込み用 Maven 依存関係管理の設定

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <version.org.drools>
  </version.org.drools>
  <version.org.jbpm>7.59.0.Final-redhat-00006</version.org.jbpm>
  <hibernate.version>5.3.17.Final</hibernate.version>
  <hibernate.core.version>5.3.17.Final</hibernate.core.version>
  <slf4j.version>1.7.26</slf4j.version>
  <jboss.javaee.version>1.0.0.Final</jboss.javaee.version>
  <logback.version>1.2.9</logback.version>
  <h2.version>1.3.173</h2.version>
  <narayana.version>5.9.0.Final</narayana.version>
  <jta.version>1.0.1.Final</jta.version>
  <junit.version>4.13.1</junit.version>
</properties>
<dependencyManagement>
  <dependencies>
    <!-- define Drools BOM -->
    <dependency>
```

```
      <groupId>org.drools</groupId>
      <artifactId>drools-bom</artifactId>
      <type>pom</type>
      <version>${version.org.drools}</version>
      <scope>import</scope>
    </dependency>
    <!-- define jBPM BOM -->
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-bom</artifactId>
      <type>pom</type>
      <version>${version.org.jbpm}</version>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

プロセスエンジン Java API (KIE API) を使用するモジュールでは、以下の例のように、必要なプロセスエンジンの依存関係やモジュールが必要とする他のコンポーネントを宣言します。

KIE API を使用するモジュールの依存関係

```
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-flow</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-flow-builder</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-bpmn2</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-persistence-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-human-task-core</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-runtime-manager</artifactId>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>${slf4j.version}</version>
</dependency>
```

アプリケーションが永続性およびトランザクションを使用する場合は、JTA フレームワークおよび JPA フレームワークを実装するアーティファクトを追加する必要があります。実際のデプロイメントの前にワークフローコンポーネントをテストするには、追加の依存関係が必要です。

以下の例は、JPA 用の Hibernate、永続性用の H2 データベース、JTA 用の Narayana、およびテストに必要なコンポーネントが含まれる依存関係を定義します。この例では、**test** スコープを使用します。アプリケーションに合わせて、この例を調整します。実稼働環境で使用する場合は、**test** スコープを削除します。

プロセスエンジンのテストモジュール依存関係の例

```
<!-- test dependencies -->
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-shared-services</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>${logback.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>${hibernate.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>${hibernate.core.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>${h2.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>jboss-transaction-api_1.2_spec</groupId>
  <artifactId>org.jboss.spec.javax.transaction</artifactId>
  <version>${jta.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.narayana.jta</groupId>
  <artifactId>narayana-jta</artifactId>
  <version>${narayana.version}</version>
  <scope>test</scope>
</dependency>
```

この設定では、アプリケーションにプロセスエンジンを埋め込むことができ、KIE API を使用してプロセス、ルール、およびイベントと対話できます。

Maven リポジトリー
Maven 依存関係の Red Hat 製品バージョンを使用するには、トップレベルの **pom.xml** ファイルで Red Hat JBoss Enterprise Maven リポジトリーを設定する必要があります。このリポジトリーに関する詳細は、JBoss Enterprise Maven リポジトリー を参照してください。

または、Red Hat カスタマーポータルの Software Downloads ページから製品配信可能ファイル **rhpam-7.12.0-maven-repository.zip** をダウンロードし、このファイルの内容をローカルの Maven リポジトリーとして利用可能にします。

68.2. CDI との統合

プロセスエンジンは、CDI との統合を自動的にサポートします。CDI フレームワークでは、ほとんどの API を変更せずに使用できます。

プロセスエンジンは、CDI コンテナー用に特別に設計された専用のモジュールも提供します。最も重要なモジュールは、プロセスエンジンサービスの CDI ラッパーを提供する **jbpm-services-cdi** です。これらのラッパーを使用して、CDI アプリケーションでプロセスエンジンを統合することができます。モジュールは、以下のサービスセットを提供します。

- **DeploymentService**
- **ProcessService**
- **UserTaskService**
- **RuntimeDataService**
- **DefinitionService**

これらのサービスは、他の CDI Bean の挿入に使用できます。

68.2.1. CDI のデプロイメントサービス

DeploymentService サービスは、ランタイム環境でデプロイメントユニットをデプロイし、デプロイ解除します。このサービスを使用してユニットをデプロイすると、デプロイメントユニットの実行が準備され、そのユニットに **RuntimeManager** インスタンスが作成されます。**DeploymentService** を使用して以下のオブジェクトを取得することもできます。

- 特定のデプロイメント ID の **RuntimeManager** インスタンス
- 指定のデプロイメント ID の完全なデプロイメントユニットを表す **DeployedUnit** インスタンス
- デプロイメントサービスで認識されている全ユニットの一覧

デフォルトでは、デプロイメントサービスは、デプロイされたユニットに関する情報を永続ストレージに保存しません。CDI フレームワークでは、サービスを使用するコンポーネントは、データベース、ファイル、システム、リポジトリーなどを使用して、デプロイメントユニット情報を保存し、復元できます。

デプロイメントサービスは、デプロイメントおよびデプロイメント解除で CDI イベントを実行します。サービスを使用するコンポーネントは、これらのイベントを処理してデプロイメントを保存し、デプロイの解除時にそれらをストアから削除できます。

- **@Deploy** 修飾子を含む **DeploymentEvent** は、ユニットのデプロイメントで実行されます。
- **@Undeploy** 修飾子を持つ **DeploymentEvent** は、ユニットのデプロイ解除で実行されます。

CDI オブザーバーメカニズムを使用して、これらのイベントの通知を取得できます。

以下の例は、ユニットのデプロイメントで通知を受け取り、デプロイメントを保存できます。

デプロイメントイベントの処理例

```
public void saveDeployment(@Observes @Deploy DeploymentEvent event) {
    // Store deployed unit information
    DeployedUnit deployedUnit = event.getDeployedUnit();
}
```

以下の例は、ユニットのデプロイメント時に通知を受け取り、デプロイメントをストレージから削除できます。

アンデプロイメントイベントの処理例

```
public void removeDeployment(@Observes @Undeploy DeploymentEvent event) {
    // Remove deployment with the ID event.getDeploymentId()
}
```

DeploymentService サービスの複数の実装が可能であるため、修飾子を使用して CDI コンテナーに特定の実装を挿入するように指示する必要があります。**DeploymentUnit** の一致する実装は、**DeploymentService** の各実装に対して存在している必要があります。

プロセスエンジンは、**KmoduleDeploymentService** 実装を提供します。この実装は、KJAR ファイルに含まれる小規模記述子である **KmoduleDeploymentUnits** と連携するように設計されています。この実装は、ほとんどのユースケースの一般的なソリューションです。この実装の修飾子は **@Kjar** です。

68.2.2. CDI のフォームプロバイダーサービス

FormProviderService サービスは、フォーム表現へのアクセスを提供します。これは通常、プロセスフォームとユーザータスクフォームの両方のユーザーインターフェイスに表示されます。

サービスは、異なる機能を提供し、異なるテクノロジーでサポートされる、分離されたフォームプロバイダーの概念に依存します。**FormProvider** インターフェイスは、フォームプロバイダーの実装のコントラクトを記述します。

FormProvider インターフェイスの定義

```
public interface FormProvider {

    int getPriority();

    String render(String name, ProcessDesc process, Map<String, Object> renderContext);

    String render(String name, Task task, ProcessDesc process, Map<String, Object> renderContext);
}
```

FormProvider インターフェイスの実装は、優先度の値を定義する必要があります。**FormProviderService** サービスがフォームをレンダリングする必要がある場合は、利用可能なプロバイダーを優先度順に呼び出します。

優先度の値が低いほど、プロバイダーが取得する優先度が高くなります。たとえば、優先度が 5 のプロバイダーは、優先度が 10 のプロバイダーの前に評価されます。必要な形式ごとに、サービスはいずれかのコンテンツを提供するまで、優先度順に利用可能なプロバイダーを繰り返し処理します。小文字のシナリオでは、単純なテキストベースのフォームが返されます。

プロセスエンジンは、**FormProvider** の以下の実装を提供します。

- Form Modeller ツールで作成されたフォームを提供するプロバイダー。優先度は 2 です。
- プロセスフォームおよびタスクフォームをサポートする FreeMarker ベースの実装 (優先度は 3)
- 単純なテキストベースのフォームを返すデフォルトのフォームプロバイダー。他のプロバイダーがコンテンツを配信しない場合の最後の手段として使用され、優先度は 1000 です。

68.2.3. CDI のランタイムデータサービス

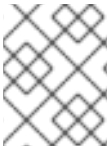
RuntimeDataService サービスは、以下のデータを含め、ランタイム時に利用可能なデータへのアクセスを提供します。

- さまざまなフィルターで実行される利用可能なプロセス
- アクティブなプロセスインスタンス (さまざまなフィルターを含む)
- プロセスインスタンスの履歴
- プロセスインスタンス変数
- プロセスインスタンスのアクティブノードおよび完了ノード

RuntimeDataService のデフォルト実装はデプロイメントイベントを監視し、デプロイされたすべてのプロセスをインデックス化して呼び出しコンポーネントに公開します。

68.2.4. CDI の定義サービス

DefinitionService サービスは、BPMN2 XML 定義の一部として保存されるプロセスの詳細へのアクセスを提供します。



注記

情報を提供するメソッドを使用する前に、**buildProcessDefinition()** メソッドを呼び出して、リポジトリーに BPMN2 コンテンツから取得したプロセス情報を設定します。

BPMN2DataService 実装は、以下のデータへのアクセスを提供します。

- 指定のプロセス定義のプロセス全体の説明
- プロセス定義にある全ユーザータスクのコレクション
- ユーザータスクノードに定義された入力に関する情報
- ユーザータスクノードに定義された出力に関する情報

- 特定のプロセス定義内で定義される再利用可能なプロセス (コールアクティビティー) の ID
- 指定のプロセス定義内で定義したプロセス変数に関する情報
- プロセス定義に含まれるすべての組織エンティティー (ユーザーおよびグループ) に関する情報。特定のプロセス定義に応じて、ユーザーおよびグループに返される値には、以下の情報が含まれます。
  - 実際のユーザーまたはグループの名前
  - ランタイム時に実際のユーザーもしくはグループの名前を取得するのに使用するプロセス変数 (例: **#{manager}**)

68.2.5. CDI 統合の設定

CDI フレームワークで **jbpm-services-cdi** モジュールを使用するには、追加するサービス実装の依存関係に対応するために Bean を提供する必要があります。

使用方法によっては、複数の Bean が必要になることがあります。

- エンティティーマネージャーおよびエンティティーマネージャーファクトリー
- ヒューマンタスクのユーザーグループコールバック
- 認証されたユーザー情報をサービスに渡すアイデンティティープロバイダー

Red Hat JBoss EAP などの JEE 環境で実行する場合、以下のプロデューサー Bean は **jbpm-services-cdi** モジュールの全要件を満たす必要があります。

JEE 環境内の jbpm-services-cdi モジュールの全要件を満たすプロデューサー Bean

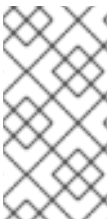
```
public class EnvironmentProducer {

    @PersistenceUnit(unitName = "org.jbpm.domain")
    private EntityManagerFactory emf;

    @Inject
    @Selectable
    private UserGroupInfoProducer userGroupInfoProducer;

    @Inject
    @Kjar
    private DeploymentService deploymentService;

    @Produces
    public EntityManagerFactory getEntityManagerFactory() {
        return this.emf;
    }

    @Produces
    public org.kie.api.task.UserGroupCallback produceSelectedUserGroupCalback() {
        return userGroupInfoProducer.produceCallback();
    }

    @Produces
    public UserInfo produceUserInfo() {
```

```
        return userGroupInfoProducer.produceUserInfo();
    }

    @Produces
    @Named("Logs")
    public TaskLifeCycleEventListener produceTaskAuditListener() {
        return new JPATaskLifeCycleEventListener(true);
    }

    @Produces
    public DeploymentService getDeploymentService() {
        return this.deploymentService;
    }

    @Produces
    public IdentityProvider produceIdentityProvider {
        return new IdentityProvider() {
            // implement IdentityProvider
        };
    }
}
```

アプリケーションの **beans.xml** ファイルは、ユーザーグループ情報コールバックの適切な代替を有効にする必要があります。この代替は **@Selectable** 修飾子に基づいて実行されます。

beans.xml ファイルのユーザーグループ情報コールバックの代替の定義

```
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee https://docs.jboss.org/cdi/beans_1_0.xsd">

  <alternatives>
    <class>org.jbpm.kie.services.cdi.producer.JAASUserGroupInfoProducer</class>
  </alternatives>

</beans>
```



注記

org.jbpm.kie.services.cdi.producer.JAASUserGroupInfoProducer はサンプル値です。この値は、サーバーが使用するセキュリティー方式 (LDAP やデータベースなど) に関係なく、アプリケーションサーバーのセキュリティー設定を再利用するため、通常は Red Hat JBoss EAP に適しています。

必要に応じて、他のプロデューサーを複数提供して、イベントリスナー **WorkItemHandlers**、**Process**、**Agenda**、および **WorkingMemory** を配信できます。以下のインターフェイスを実装して、これらのコンポーネントを提供できます。

CDI とのプロセスエンジン統合用のワークアイテムハンドラープロデューサーインターフェイス

```
/**
 * Enables providing custom implementations to deliver WorkItem name and WorkItemHandler
instance pairs
```

```
 * for the runtime.
 * <br/>
 * This interface is invoked by the RegisterableItemsFactory implementation (in particular
InjectableRegisterableItemsFactory
 * in the CDI framework) for every KieSession. Always return new instances of objects to avoid
unexpected
 * results.
 *
 */
public interface WorkItemHandlerProducer {

    /**
     * Returns map of work items(key = work item name, value = work item handler instance)
     * to be registered on KieSession
     * <br/>
     * The following parameters might be given:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     *
     * @param identifier - identifier of the owner - usually the RuntimeManager. This parameter allows
the producer to filter out
     * and provide valid instances for a given owner
     * @param params - the owner might provide some parameters, usually KieSession, TaskService,
RuntimeManager instances
     * @return map of work item handler instances (always return new instances when this method is
invoked)
     */
    Map<String, WorkItemHandler> getWorkItemHandlers(String identifier, Map<String, Object>
params);
}
```

CDI とのプロセスエンジン統合用のイベントリスナープロデューサーインターフェイス

```
/**
 * Enables defining custom producers for known EventListeners. There might be several
 * implementations that might provide a different listener instance based on the context in which they
are executed.
 * <br/>
 * This interface is invoked by the RegisterableItemsFactory implementation (in particular,
InjectableRegisterableItemsFactory
 * in the CDI framework) for every KieSession. Always return new instances of objects to avoid
unexpected results.
 *
 * @param <T> type of the event listener - ProcessEventListener, AgendaEventListener,
WorkingMemoryEventListener
 */
public interface EventListenerProducer<T> {

    /**
     * Returns list of instances for given (T) type of listeners
     * <br/>
     * Parameters that might be given are:
     * <ul>
```

```
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     * @param identifier - identifier of the owner - usually RuntimeManager. This parameter allows the
producer to filter out
     * and provide valid instances for given owner
     * @param params - the owner might provide some parameters, usually KieSession, TaskService,
RuntimeManager instances
     * @return list of listener instances (always return new instances when this method is invoked)
     */
    List<T> getEventListeners(String identifier, Map<String, Object>  params);
}
```

これらの 2 つのインターフェイスを実装する Bean はランタイム時に収集され、**RuntimeManager** クラスが **KieSession** インスタンスをビルドすると呼び出されます。

68.2.5.1. CDI Bean としてのランタイムマネージャー

RuntimeManager クラスを CDI Bean としてアプリケーション内の他の CDI Bean に挿入できます。**RuntimeManager** インスタンスを正しく初期化できるようにするには、**RuntimeEnvironment** クラスを適切に生成する必要があります。

以下の CDI 修飾子は既存のランタイムマネージャーストラテジーを参照します。

- **@Singleton**
- **@PerRequest**
- **@PerProcessInstance**

ランタイムマネージャーの詳細は、「ランタイムマネージャー」を参照してください。

注記

RuntimeManager クラスを直接挿入できますが、CDI、EJB、Spring などのフレームワークのほとんどのユースケースでは、サービスが使用されます。プロセスエンジンサービスは、ランタイムマネージャーを使用するための多くのベストプラクティスを実装します。

ランタイムマネージャーを使用するには、「CDI 統合の設定」セクションに定義されたプロデューサーに **RuntimeEnvironment** クラスを追加する必要があります。

RuntimeEnvironment クラスを提供するプロデューサー Bean

```
public class EnvironmentProducer {

    //Add the same producers as for services

    @Produces
    @Singleton
    @PerRequest
    @PerProcessInstance
    public RuntimeEnvironment produceEnvironment(EntityManagerFactory emf) {
```

```
        RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
                .newDefaultBuilder()
                .entityManagerFactory(emf)
                .userGroupCallback(getUserGroupCallback())
                .registerableItemsFactory(InjectableRegisterableItemsFactory.getFactory(beanManager,
null))
                .addAsset(ResourceFactory.newClassPathResource("BPMN2-ScriptTask.bpmn2"),
ResourceType.BPMN2)
                .addAsset(ResourceFactory.newClassPathResource("BPMN2-UserTask.bpmn2"),
ResourceType.BPMN2)
                .get();
        return environment;
    }
}
```

この例では、メソッドレベルですべての修飾子を指定して、単一のプロデューサーメソッドがすべてのランタイムマネージャーストラテジーに **RuntimeEnvironment** クラスを提供できます。

完全なプロデューサーが利用できる状態になったら、**RuntimeManager** クラスはアプリケーションの CDI Bean に挿入できます。

RuntimeManager クラスの挿入

```
public class ProcessEngine {

    @Inject
    @Singleton
    private RuntimeManager singletonManager;

    public void startProcess() {

        RuntimeEngine runtime = singletonManager.getRuntimeEngine(EmptyContext.get());
        KieSession ksession = runtime.getKieSession();

        ProcessInstance processInstance = ksession.startProcess("UserTask");

        singletonManager.disposeRuntimeEngine(runtime);
    }
}
```

RuntimeManager クラスを挿入する場合は、アプリケーションに **RuntimeManager** のインスタンスが 1 つだけ存在する可能性があります。通常、必要に応じて **RuntimeManager** インスタンスを作成する **DeploymentService** サービスを使用します。

DeploymentService の代わりに、**RuntimeManagerFactory** クラスを挿入し、アプリケーションがそれを使用して **RuntimeManager** インスタンスを作成できます。この場合、**EnvironmentProducer** 定義は依然として必要になります。以下の例は、単純な ProcessEngine Bean を示しています。

ProcessEngine Bean の例

```
public class ProcessEngine {

    @Inject
    private RuntimeManagerFactory managerFactory;
```

```
    @Inject
    private EntityManagerFactory emf;

    @Inject
    private BeanManager beanManager;

    public void startProcess() {
        RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
                .newDefaultBuilder()
                .entityManagerFactory(emf)
                .addAsset(ResourceFactory.newClassPathResource("BPMN2-ScriptTask.bpmn2"),
ResourceType.BPMN2)
                .addAsset(ResourceFactory.newClassPathResource("BPMN2-UserTask.bpmn2"),
ResourceType.BPMN2)
                .registerableItemsFactory(InjectableRegisterableItemsFactory.getFactory(beanManager,
null))
                .get();

        RuntimeManager manager = managerFactory.newSingletonRuntimeManager(environment);
        RuntimeEngine runtime = manager.getRuntimeEngine(EmptyContext.get());
        KieSession ksession = runtime.getKieSession();

        ProcessInstance processInstance = ksession.startProcess("UserTask");

        manager.disposeRuntimeEngine(runtime);
        manager.close();
    }

}
```

68.3. SPRING との統合

Spring フレームワークでプロセスエンジンを使用する方法はいくつかありますが、最も頻繁に使用されるのは 2 つのアプローチです。

- Runtime Manager API の直接使用
- プロセスエンジンサービスの使用

どちらのアプローチもテストされ、有効です。

アプリケーションが 1 つのランタイムマネージャーのみを使用する必要がある場合は、ダイレクト Runtime Manager API を使用します。これは、Spring アプリケーションでプロセスエンジンを使用する最も簡単な方法となるためです。

アプリケーションがランタイムマネージャーの複数のインスタンスを使用する必要がある場合は、プロセスエンジンサービスを使用して、動的ランタイム環境を提供することでベストプラクティスをカプセル化します。

68.3.1. Spring でのランタイムマネージャー API の直接使用

ランタイムマネージャーは、プロセスエンジンとタスクサービスを同期して管理します。ランタイムマネージャーの詳細は、「ランタイムマネージャー」を参照してください。

Spring フレームワークでランタイムマネージャーを設定するには、以下のファクトリー Bean を使用します。

- **org.kie.spring.factorybeans.RuntimeEnvironmentFactoryBean**
- **org.kie.spring.factorybeans.RuntimeManagerFactoryBean**
- **org.kie.spring.factorybeans.TaskServiceFactoryBean**

これらのファクトリー Bean は、Spring アプリケーションの **spring.xml** ファイルを設定する標準的な方法を提供します。

68.3.1.1. RuntimeEnvironmentFactoryBean Bean

RuntimeEnvironmentFactoryBean ファクトリー Bean は **RuntimeEnvironment** のインスタンスを生成します。これらのインスタンスは、**RuntimeManager** インスタンスの作成に必要です。

Bean は、異なるデフォルト設定を持つ以下のタイプの **RuntimeEnvironment** インスタンスの作成をサポートします。

- **DEFAULT**: ランタイムマネージャーのデフォルトまたは最も一般的な設定
- **EMPTY**: 手動で設定できる、完全に空の環境
- **DEFAULT_IN_MEMORY**: ランタイムエンジンの永続性がない DEFAULT と同じ設定
- **DEFAULT_KJAR**: DEFAULT と同じ設定ですが、アセットは KJAR アーティファクトから読み込まれます。これは、リリース ID または GAV 値によって識別されます。
- **DEFAULT_KJAR_CL**: 設定は KJAR アーティファクトの **kmodule.xml** 記述子から構築されます。

必須プロパティーは選択されたタイプによって異なります。ただし、全タイプに対するナレッジ情報が必要です。この要件は、以下のいずれかの情報が提供される必要があることを意味します。

- **knowledgeBase**
- **assets**
- **releaseId**
- **groupId, artifactId, version**

タイプ **DEFAULT**、**DEFAULT_KJAR**、および **DEFAULT_KJAR_CL** については、以下のパラメーターを指定して永続性を設定する必要もあります。

- エンティティーマネージャーファクトリー
- トランザクションマネージャー

永続性やトランザクションサポートがこのトランザクションマネージャーを基に設定されるため、トランザクションマネージャーは Spring トランザクションマネージャーである必要があります。

任意で、**EntityManagerFactory** から新規インスタンスを作成する代わりに **EntityManager** インスタンスを指定できます。たとえば、Spring から共有エンティティーマネージャーを使用できます。

その他のプロパティーはすべて任意です。ランタイム環境の選択したタイプで決定されるデフォルト値を上書きできます。

68.3.1.2. RuntimeManagerFactoryBean Bean

RuntimeManagerFactoryBean ファクトリー Bean は、提供された **RuntimeEnvironment** インスタンスに基づいて、指定したタイプの **RuntimeManager** インスタンスを生成します。

サポートされるタイプは、ランタイムマネージャーの戦略に対応します。

- **SINGLETON**
- **PER_REQUEST**
- **PER_PROCESS_INSTANCE**

タイプが指定されていない場合のデフォルトタイプは **SINGLETON** です。

すべてのランタイムマネージャーは一意に特定する必要があるため、識別子は必須プロパティーです。このファクトリーによって作成されたすべてのインスタンスがキャッシュされるため、destroy メソッド (**close()**) を使用して適切に破棄できます。

68.3.1.3. TaskServiceFactoryBean Bean

TaskServiceFactoryBean ファクトリー Bean は、指定のプロパティーに基づいて **TaskService** のインスタンスを生成します。以下の必須プロパティーを指定する必要があります。

- エンティティーマネージャーファクトリー
- トランザクションマネージャー

永続性やトランザクションサポートがこのトランザクションマネージャーを基に設定されるため、トランザクションマネージャーは Spring トランザクションマネージャーである必要があります。

任意で、**EntityManagerFactory** から新規インスタンスを作成する代わりに **EntityManager** インスタンスを指定できます。たとえば、Spring から共有エンティティーマネージャーを使用できます。

タスクサービスインスタンスに、任意の追加プロパティーを設定することもできます。

- **userGroupCallback**: タスクサービスが使用する必要のある **UserGroupCallback** の実装。デフォルト値は **MVELUserGroupCallbackImpl** です。
- **userInfo**: タスクサービスが使用する必要のある **UserInfo** の実装。デフォルト値は **DefaultUserInfo** です。
- **listener**: タスクのさまざまな操作で通知する必要がある **TaskLifeCycleEventListener** リスナーのリスト

このファクトリー Bean は、タスクサービスの単一のインスタンスを作成します。設計上、このインスタンスは Spring 環境のすべての Bean 間で共有される必要があります。

68.3.1.4. Spring アプリケーションを使用したランタイムマネージャーの設定

以下の手順は、Spring アプリケーション内の単一のランタイムマネージャーの完全な設定例になります。

手順

1. エンティティーマネージャーファクトリーおよびトランザクションマネージャーを設定します。

   spring.xml ファイルでのエンティティーマネージャーファクトリーおよびトランザクションマネージャーの設定

   ```
   <bean id="jbpmEMF"
   class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
     <property name="persistenceUnitName" value="org.jbpm.persistence.spring.jta"/>
   </bean>

   <bean id="jbpmEM"
   class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
     <property name="entityManagerFactory" ref="jbpmEMF"/>
   </bean>

   <bean id="narayanaUserTransaction" factory-method="userTransaction"
   class="com.arjuna.ats.jta.UserTransaction" />

   <bean id="narayanaTransactionManager" factory-method="transactionManager"
   class="com.arjuna.ats.jta.TransactionManager" />

   <bean id="jbpmTxManager"
   class="org.springframework.transaction.jta.JtaTransactionManager">
     <property name="transactionManager" ref="narayanaTransactionManager" />
     <property name="userTransaction" ref="narayanaUserTransaction" />
   </bean>
   ```

   これらの設定は、以下の永続性設定を定義します。

   - JTA トランザクションマネージャー (Narayana JTA でサポート) (ユニットテスト用またはサーブレットコンテナー用)
   - **org.jbpm.persistence.spring.jta** 永続ユニットのエンティティーマネージャーファクトリー

2. ビジネスプロセスリソースを設定します。

   spring.xml ファイルでのビジネスプロセスリソースの設定

   ```
   <bean id="process" factory-method="newClassPathResource"
   class="org.kie.internal.io.ResourceFactory">
     <constructor-arg>
       <value>jbpm/processes/sample.bpmn</value>
     </constructor-arg>
   </bean>
   ```

   これらの設定は、実行可能な1つのプロセスを定義します。リソースの名前は **sample.bpmn** で、クラスパスで利用可能でなければなりません。クラスパスを単純な方法で使用して、プロセスエンジンを試すためのリソースを追加できます。

3. エンティティーマネージャー、トランザクションマネージャー、およびリソースを使用して **RuntimeEnvironment** インスタンスを設定します。

spring.xml ファイルでの RuntimeEnvironment インスタンスの設定

```
<bean id="runtimeEnvironment"
class="org.kie.spring.factorybeans.RuntimeEnvironmentFactoryBean">
  <property name="type" value="DEFAULT"/>
  <property name="entityManagerFactory" ref="jbpmEMF"/>
  <property name="transactionManager" ref="jbpmTxManager"/>
  <property name="assets">
    <map>
      <entry key-ref="process"><util:constant static-
field="org.kie.api.io.ResourceType.BPMN2"/></entry>
    </map>
  </property>
</bean>
```

これらの設定は、ランタイムマネージャーのデフォルトのランタイム環境を定義します。

4. 環境に基づいて **RuntimeManager** インスタンスを作成します。

```
<bean id="runtimeManager"
class="org.kie.spring.factorybeans.RuntimeManagerFactoryBean" destroy-method="close">
  <property name="identifier" value="spring-rm"/>
  <property name="runtimeEnvironment" ref="runtimeEnvironment"/>
</bean>
```

結果

これらの手順を完了した後、**EntityManagerFactory** クラスおよび JTA トランザクションマネージャーを使用して、ランタイムマネージャーを使用して Spring 環境でプロセスを実行できます。

リポジトリー では、異なるストラテジーの完全 Spring 設定ファイルを確認できます。

68.3.1.5. Spring フレームワークのランタイムマネージャーの追加設定オプション

「Spring アプリケーションを使用したランタイムマネージャーの設定」で説明されているように、**EntityManagerFactory** クラスと JTA トランザクションマネージャーの設定の他に、Spring フレームワークのランタイムマネージャーに他の設定オプションを使用できます。

- JTA および **SharedEntityManager** クラス
- ローカル永続ユニットおよび **EntityManagerFactory** クラス
- ローカル永続ユニットおよび **SharedEntityManager** クラス

アプリケーションがローカル永続ユニットで設定され、**AuditService** サービスを使用してプロセスエンジン履歴データをクエリーする場合は、**org.kie.api.runtime.EnvironmentName.USE_LOCAL_TRANSACTIONS** 環境エントリーを **RuntimeEnvironment** インスタンス設定に追加する必要があります。

spring.xml ファイルのローカル永続ユニットの RuntimeEnvironment インスタンス設定

```
<bean id="runtimeEnvironment"
class="org.kie.spring.factorybeans.RuntimeEnvironmentFactoryBean">
...
    <property name="environmentEntries" ref="env" />
```

```
</bean>
...

<util:map id="env" key-type="java.lang.String" value-type="java.lang.Object">
<entry>
 <key>
  <util:constant
   static-field="org.kie.api.runtime.EnvironmentName.USE_LOCAL_TRANSACTIONS" />
 </key>
 <value>true</value>
</entry>
</util:map>
```

リポジトリーの設定オプションの例は、設定ファイル および テストケース で確認できます。

68.3.2. Spring を使用したプロセスエンジンサービス

動的な Spring アプリケーションを作成し、アプリケーションを再起動しなくても、プロセス定義、データモデル、ルール、フォームなどのビジネスアセットを追加および削除できます。

この場合は、プロセスエンジンサービスを使用します。プロセスエンジンサービスはフレームワークに依存しないように設計され、必要なフレームワーク固有のアドオンに個別のモジュールが追加されます。

jbpm-kie-services モジュールには、サービスのコードロジックが含まれます。Spring アプリケーションはこれらの純粋な Java サービスを使用できます。

プロセスエンジンサービスを設定するために Spring アプリケーションに追加する必要がある唯一のコードは、**IdentityProvider** インターフェイスの実装です。この実装は、セキュリティー設定によって異なります。以下の実装例では Spring Security を使用していますが、Spring アプリケーションで利用可能なすべてのセキュリティー機能に対応していない可能性があります。

Spring Security を使用した IdentityProvider インターフェイスの実装

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.kie.internal.identity.IdentityProvider;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;

public class SpringSecurityIdentityProvider implements IdentityProvider {

 public String getName() {

  Authentication auth = SecurityContextHolder.getContext().getAuthentication();
  if (auth != null && auth.isAuthenticated()) {
   return auth.getName();
  }
  return "system";
 }

 public List<String> getRoles() {
```

```
 Authentication auth = SecurityContextHolder.getContext().getAuthentication();
 if (auth != null && auth.isAuthenticated()) {
  List<String> roles = new ArrayList<String>();

  for (GrantedAuthority ga : auth.getAuthorities()) {
   roles.add(ga.getAuthority());
  }

  return roles;
 }

 return Collections.emptyList();
}

public boolean hasRole(String role) {
 return false;
}

}
```

68.3.2.1. Spring アプリケーションを使用したプロセスエンジンサービスの設定

以下の手順は、Spring アプリケーション内のプロセスエンジンサービスの完全な設定例です。

手順

1. トランザクションを設定します。

   spring.xml ファイルでのトランザクションの設定

   ```
   <context:annotation-config />
   <tx:annotation-driven />
   <tx:jta-transaction-manager />

   <bean id="transactionManager"
   class="org.springframework.transaction.jta.JtaTransactionManager" />
   ```

2. JPA および永続性を設定します。

   spring.xml ファイルでの JPA および永続性の設定

   ```
   <bean id="entityManagerFactory"
   class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean" depends-
   on="transactionManager">
     <property name="persistenceXmlLocation" value="classpath:/META-INF/jbpm-
   persistence.xml" />
   </bean>
   ```

3. セキュリティーおよびユーザーおよびグループの情報プロバイダーを設定します。

   spring.xml ファイルでのセキュリティー、ユーザー、およびグループの情報プロバイダーの設定

   ```
   <util:properties id="roleProperties" location="classpath:/roles.properties" />
   ```

```
<bean id="userGroupCallback"
class="org.jbpm.services.task.identity.JBossUserGroupCallbackImpl">
  <constructor-arg name="userGroups" ref="roleProperties"></constructor-arg>
</bean>

<bean id="identityProvider" class="org.jbpm.spring.SpringSecurityIdentityProvider"/>
```

4. ランタイムマネージャーファクトリーを設定します。このファクトリーは Spring コンテキストを認識するため、トランザクションコマンドサービスやタスクサービスなどの必要なサービスをサポートすることで Spring コンテナーと正しい方法で対話できます。

spring.xml ファイルでのランタイムマネージャーファクトリーの設定

```
<bean id="runtimeManagerFactory"
class="org.kie.spring.manager.SpringRuntimeManagerFactoryImpl">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="userGroupCallback" ref="userGroupCallback"/>
</bean>

<bean id="transactionCmdService"
class="org.jbpm.shared.services.impl.TransactionalCommandService">
  <constructor-arg name="emf" ref="entityManagerFactory"></constructor-arg>
</bean>

<bean id="taskService" class="org.kie.spring.factorybeans.TaskServiceFactoryBean"
destroy-method="close">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
  <property name="transactionManager" ref="transactionManager"/>
  <property name="userGroupCallback" ref="userGroupCallback"/>
  <property name="listeners">
    <list>
      <bean class="org.jbpm.services.task.audit.JPATaskLifeCycleEventListener">
        <constructor-arg value="true"/>
      </bean>
    </list>
  </property>
</bean>
```

5. プロセスエンジンサービスを Spring Bean として設定します。

spring.xml ファイルでプロセスエンジンサービスを Spring Bean として設定

```
<!-- Definition service -->
<bean id="definitionService"
class="org.jbpm.kie.services.impl.bpmn2.BPMN2DataServiceImpl"/>

<!-- Runtime data service -->
<bean id="runtimeDataService" class="org.jbpm.kie.services.impl.RuntimeDataServiceImpl">
  <property name="commandService" ref="transactionCmdService"/>
  <property name="identityProvider" ref="identityProvider"/>
  <property name="taskService" ref="taskService"/>
</bean>

<!-- Deployment service -->
```

```
<bean id="deploymentService"
class="org.jbpm.kie.services.impl.KModuleDeploymentService" depends-
on="entityManagerFactory" init-method="onInit">
  <property name="bpmn2Service" ref="definitionService"/>
  <property name="emf" ref="entityManagerFactory"/>
  <property name="managerFactory" ref="runtimeManagerFactory"/>
  <property name="identityProvider" ref="identityProvider"/>
  <property name="runtimeDataService" ref="runtimeDataService"/>
</bean>

<!-- Process service -->
<bean id="processService" class="org.jbpm.kie.services.impl.ProcessServiceImpl" depends-
on="deploymentService">
  <property name="dataService" ref="runtimeDataService"/>
  <property name="deploymentService" ref="deploymentService"/>
</bean>

<!-- User task service -->
<bean id="userTaskService" class="org.jbpm.kie.services.impl.UserTaskServiceImpl"
depends-on="deploymentService">
  <property name="dataService" ref="runtimeDataService"/>
  <property name="deploymentService" ref="deploymentService"/>
</bean>

<!-- Register the runtime data service as a listener on the deployment service so it can
receive notification about deployed and undeployed units -->
<bean id="data"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean" depends-
on="deploymentService">
  <property name="targetObject" ref="deploymentService"></property>
  <property name="targetMethod"><value>addListener</value></property>
  <property name="arguments">
  <list>
     <ref bean="runtimeDataService"/>
  </list>
  </property>
</bean>
```

結果

Spring アプリケーションはプロセスエンジンサービスを使用できます。

68.4. EJB との統合

プロセスエンジンは、Enterprise Java Bean (EJB) の完全な統合レイヤーを提供します。このレイヤーは、ローカルおよびリモートの EJB 対話の両方をサポートします。

以下のモジュールは EJB サービスを提供します。

- **jbpm-services-ejb-api**: **jbpm-services-api** モジュールを EJB 固有のインターフェイスおよびオブジェクトで拡張する API モジュール
- **jbpm-services-ejb-impl**: コアサービスの EJB 拡張
- **jbpm-services-ejb-timer**: EJB Timer サービスに基づくプロセスエンジンスケジューラーサービスの実装

- **jbpm-services-ejb-client**: デフォルトでは Red Hat JBoss EAP をサポートするリモート対話の EJB リモートクライアント実装

EJB レイヤーはプロセスエンジンサービスに基づいています。リモートインターフェイスを使用する場合はいくつかの制限がありますが、コアモジュールとほぼ同じ機能を提供します。

デプロイメントサービスの主な制限は、リモート EJB サービスとして使用されている場合に、以下の方法のみをサポートします。

- **deploy()**
- **undeploy()**
- **activate()**
- **deactivate()**
- **isDeployed()**

他のメソッドは、リモートインターフェイスに使用できない **RuntimeManager** などのランタイムオブジェクトのインスタンスを返すため除外されます。

他のすべてのサービスは、コアモジュールに含まれているバージョンと同じ機能を EJB 上で提供します。

68.4.1. EJB サービスの実装

プロセスエンジンのコアサービスの拡張として、EJB サービスは EJB ベースの実行セマンティクスを提供し、さまざまな EJB 固有の機能をベースにしています。

- **DeploymentServiceEJBImpl** は、コンテナー管理の同時実行性を備えた EJB シングルトンとして実装されます。ロックタイプは **write** に設定されます。
- **DefinitionServiceEJBImpl** は、コンテナー管理の同時実行性を備えた EJB シングルトンとして実装されます。この全体的なロックタイプは **read** に設定されます。**buildProcessDefinition()** メソッドの場合は ロックタイプが **write** に設定されます。
- **ProcessServiceEJBImpl** は、ステートレスセッション Bean として実装されます。
- **RuntimeDataServiceEJBImpl** は、EJB シングルトンとして実装されます。ほとんどのメソッドでは、ロックタイプが **read** に設定されます。以下のメソッドでは、ロックタイプが **write** に設定されます。
  - **onDeploy()**
  - **onUnDeploy()**
  - **onActivate()**
  - **onDeactivate()**
- **UserTaskServiceEJBImpl** は、ステートレスセッション Bean として実装されます。

トランザクション
EJB コンテナーは EJB サービスでトランザクションを管理します。このため、アプリケーションコード内でトランザクションマネージャーまたはユーザートランザクションを設定する必要はありません。

アイデンティティープロバイダー

デフォルトのアイデンティティープロバイダーは **EJBContext** インターフェイスをベースとし、名前とロールの両方に呼び出し元プリンシパル情報に依存します。**IdentityProvider** インターフェイスは、ロールに関連する 2 つのメソッドを提供します。

- **EJBContext** インターフェイスは特定ユーザーの全ロールを取得するオプションを提供しないため、**getRoles()** によって空のリストが返されます。
- **hasRole()** はコンテキストの **isCallerInRole()** メソッドに委譲されます。

EJB 環境で有効な情報を利用できるようにするには、標準の JEE セキュリティープラクティスに従ってユーザーを認証および承認する必要があります。EJB サービスに認証または承認が設定されていない場合、匿名ユーザーは常に仮定されます。

別のセキュリティーモデルを使用する場合は、EJB サービスの **IdentityProvider** オブジェクトに CDI 形式の挿入を使用できます。この場合は、**org.kie.internal.identity.IdentityProvider** インターフェイスを実装する有効な CDI Bean を作成し、アプリケーションでの挿入でこの Bean を利用できるようにします。この実装は、**EJBContext** ベースのアイデンティティープロバイダーよりも優先されます。

デプロイメントの同期

デプロイメントの同期はデフォルトで有効になり、3 秒ごとにすべてのデプロイメントの同期を試みます。コンテナー管理コンカレンシーを使用する EJB シングルトンとして実装されます。ロックタイプは **write** に設定されます。EJB タイマーサービスを使用して同期ジョブをスケジュールします。

EJB スケジューラーサービス

プロセスエンジンは、スケジューラーサービスを使用してタイマーイベントやデッドラインなどの時間ベースのアクティビティーを処理します。EJB 環境で実行する場合、プロセスエンジンは EJB タイマーサービスに基づいてスケジューラーを使用します。すべての **RuntimeManager** インスタンスに対してこのスケジューラーを登録します。

クラスター操作に対応するためにアプリケーションサーバーに固有の設定を使用しないといけない場合があります。

UserGroupCallback および UserInfo 実装の選択

UserGroupCallback インターフェイスおよび **UserInfo** インターフェイスに必要な実装は、さまざまなアプリケーションで異なる場合があります。これらのインターフェイスに EJB を直接挿入することはできません。以下のシステムプロパティーを使用して、既存の実装を選択するか、プロセスエンジンにこれらのインターフェイスのカスタム実装を使用できます。

- **org.jbpm.ht.callback**: このプロパティーは、**UserGroupCallback** インターフェイスの実装を選択します。
  - **mvel**: 通常、テストに使用するデフォルトの実装です。
  - **ldap**: LDAP ベースの実装。この実装には、**jbpm.usergroup.callback.properties** ファイルで追加の設定が必要です。
  - **db**: データベースベースの実装。この実装には、**jbpm.usergroup.callback.properties** ファイルで追加の設定が必要です。
  - **jaas**: コンテナーからユーザー情報を要求する実装。
  - **props**: 単純なプロパティーベースのコールバック。この実装には、ユーザーおよびグループすべてが含まれる追加のプロパティーファイルが必要です。
  - **custom**: カスタム実装。実装の完全修飾クラス名を **org.jbpm.ht.custom.callback** システムプロパティーに指定する必要があります。

- **org.jbpm.ht.userinfo**: このプロパティーは **UserInfo** インターフェイスの実装を選択します。
  - **ldap**: LDAP ベースの実装。この実装には、**jbpm-user.info.properties** ファイルで追加の設定が必要です。
  - **db**: データベースベースの実装。この実装には、**jbpm-user.info.properties** ファイルで追加の設定が必要です。
  - **props**: 単純なプロパティーベースの実装。この実装には、すべてのユーザー情報が含まれる追加のプロパティーファイルが必要です。
  - **custom**: カスタム実装。実装の完全修飾クラス名を **org.jbpm.ht.custom.userinfo** システムプロパティーに指定する必要があります。

通常、アプリケーションサーバーまたは JVM の起動時にシステムプロパティーを設定します。サービスを使用する前に、コードでプロパティーを設定することもできます。たとえば、これらのシステムプロパティーを設定するカスタムの **@Startup** Bean を指定できます。

68.4.2. ローカル EJB インターフェイス

以下のローカル EJB サービスインターフェイスはコアサービスを拡張します。

- **org.jbpm.services.ejb.api.DefinitionServiceEJBLocal**
- **org.jbpm.services.ejb.api.DeploymentServiceEJBLocal**
- **org.jbpm.services.ejb.api.ProcessServiceEJBLocal**
- **org.jbpm.services.ejb.api.RuntimeDataServiceEJBLocal**
- **org.jbpm.services.ejb.api.UserTaskServiceEJBLocal**

これらのインターフェイスを挿入ポイントとして使用し、**@EJB** アノテーションを付ける必要があります。

ローカルの EJB サービスインターフェイスの使用

```
@EJB
private DefinitionServiceEJBLocal bpmn2Service;

@EJB
private DeploymentServiceEJBLocal deploymentService;

@EJB
private ProcessServiceEJBLocal processService;

@EJB
private RuntimeDataServiceEJBLocal runtimeDataService;
```

これらのインターフェイスを挿入した後に、コアモジュールと同じ方法で操作を呼び出します。ローカルインターフェイスの使用には制限がありません。

68.4.3. リモート EJB インターフェイス

以下の専用のリモート EJB インターフェイスはコアサービスを拡張します。

- **org.jbpm.services.ejb.api.DefinitionServiceEJBRemote**
- **org.jbpm.services.ejb.api.DeploymentServiceEJBRemote**
- **org.jbpm.services.ejb.api.ProcessServiceEJBRemote**
- **org.jbpm.services.ejb.api.RuntimeDataServiceEJBRemote**
- **org.jbpm.services.ejb.api.UserTaskServiceEJBRemote**

これらのインターフェイスは、カスタムタイプの処理を除き、ローカルインターフェイスと同じ方法で使用できます。

カスタムタイプは 2 つの方法で定義できます。**グローバル** に定義されたタイプは、アプリケーションクラスパスで利用でき、エンタープライズアプリケーションに含まれます。**ローカルでデプロイメントユニット** にタイプを定義する場合、タイプはプロジェクトの依存関係 (KJAR ファイルなど) で宣言され、デプロイメント時に解決されます。

グローバルで利用可能なタイプには、特別な処理は必要ありません。EJB コンテナーは、リモートリクエストの処理時にデータを自動的にマーシャルします。ただし、ローカルカスタムタイプは、デフォルトでは EJB コンテナーに表示されません。

プロセスエンジン EJB サービスは、カスタムタイプと連携するメカニズムを提供します。他にも、以下の 2 つのタイプが提供されます。

- **org.jbpm.services.ejb.remote.api.RemoteObject**: シングル値パラメーターのシリアライズ可能なラッパークラス
- **org.jbpm.services.ejb.remote.api.RemoteMap**: カスタムオブジェクト入力を受け入れるサービスメソッドのリモート呼び出しを単純化する専用の **java.util.Map** 実装。マップの内部実装は、送信時に追加のシリアライズを回避するために、すでにシリアライズされているコンテンツを保持します。
  この実装には、通常データ送信時には使用されない **java.util.Map** のメソッドが含まれません。

これらの特別なオブジェクトは、**ObjectInputStream** オブジェクトを使用してバイトに対してシリアライズを実行します。EJB クライアント/コンテナーでのデータのシリアライズに必要なものを削除します。シリアライズは必要ないため、カスタムデータモデルを EJB コンテナーと共有する必要はありません。

以下のコード例は、ローカルタイプおよびリモート EJB サービスと動作します。

リモート EJB サービスでのローカルタイプの使用

```
// Start a process with custom types via remote EJB

Map<String, Object> parameters = new RemoteMap();
Person person = new org.jbpm.test.Person("john", 25, true);
parameters.put("person", person);

Long processInstanceId = processService.startProcess(deploymentUnit.getIdentifier(), "custom-data-project.work-on-custom-data", parameters);

// Fetch task data and complete a task with custom types via remote EJB
Map<String, Object> data = userTaskService.getTaskInputContentByTaskId(taskId);

Person fromTaskPerson = data.get("_person");
fromTaskPerson.setName("John Doe");
```

```
RemoteMap outcome = new RemoteMap();
outcome.put("person_", fromTaskPerson);

userTaskService.complete(taskId, "john", outcome);
```

同様に、**RemoteObject** クラスを使用してイベントをプロセスインスタンスに送信できます。

```
// Send an event with a custom type via remote EJB
Person person = new org.jbpm.test.Person("john", 25, true);

RemoteObject myObject = new RemoteObject(person);

processService.signalProcessInstance(processInstanceId, "MySignal", myObject);
```

68.4.4. リモート EJB クライアント

リモートクライアントサポートは、アプリケーションサーバー固有のコード向けのファサードである **ClientServiceFactory** インターフェイスの実装によって提供されます。

ClientServiceFactory インターフェイスの定義

```
/**
 * Generic service factory used for remote lookups that are usually container specific.
 *
 */
public interface ClientServiceFactory {

 /**
  * Returns unique name of given factory implementation
  * @return
  */
 String getName();

 /**
  * Returns remote view of given service interface from selected application
  * @param application application identifier on the container
  * @param serviceInterface remote service interface to be found
  * @return
  * @throws NamingException
  */
 <T> T getService(String application, Class<T> serviceInterface) throws NamingException;
}
```

ServiceLoader メカニズムを使用して実装を動的に登録できます。デフォルトでは、Red Hat JBoss EAP で利用可能な実装は 1 つだけです。

各 **ClientServiceFactory** 実装は名前を指定する必要があります。この名前は、クライアントレジストリー内で登録するために使用されます。名前で実装を検索できます。

以下のコードは、デフォルトの Red Hat JBoss EAP リモートクライアントを取得します。

デフォルトの Red Hat JBoss EAP リモートクライアントの取得

```
// Retrieve a valid client service factory
ClientServiceFactory factory = ServiceFactoryProvider.getProvider("JBoss");

// Set the application variable to the module name
String application = "sample-war-ejb-app";

// Retrieve the required service from the factory
DeploymentServiceEJBRemote deploymentService = factory.getService(application,
DeploymentServiceEJBRemote.class);
```

サービスを取得したら、そのメソッドを使用できます。

Red Hat JBoss EAP とリモートクライアントを使用する際に、以下の Maven 依存関係を追加して、すべての EJB クライアントライブラリーを取り込むことができます。

```
<dependency>
  <groupId>org.jboss.as</groupId>
  <artifactId>jboss-as-ejb-client-bom</artifactId>
  <version>7.4.1.Final</version> <!-- use the valid version for the server you run on -->
  <optional>true</optional>
  <type>pom</type>
</dependency>
```

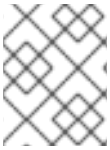
68.5. OSGI との統合

すべてのコアプロセスエンジン JAR ファイルとコアの依存関係は、OSGi 対応です。以下の追加のプロセスエンジン JAR ファイルも OSGI 対応です。

- jbpm-flow
- jbpm-flow-builder
- jbpm-bpmn2

OSGi 対応 JAR ファイルには、**META-INF** ディレクトリーに **MANIFEST.MF** ファイルが含まれます。これらのファイルには、必要な依存関係などのデータが含まれます。このような JAR ファイルを OSGi 環境に追加できます。

OSGi インフラストラクチャーに関する補足情報は、OSGi ドキュメント を参照してください。



注記

OSGi フレームワークとの統合のサポートは非推奨になりました。新しい拡張機能や機能は提供されておらず、将来のリリースで削除される予定です。

付録A バージョン情報

本ドキュメントの最終更新日: 2023 年 2 月 1 日

付録B お問い合わせ先

Red Hat Process Automation Manager のドキュメントチーム: brms-docs@redhat.com