Red Hat Process Automation Manager 7.5

Process Server の管理とモニターリング

ガイド

Last Updated: 2022-09-17

Red Hat Process Automation Manager 7.5 Process Server の管理とモニターリング

ガイド

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

法律上の通知

概要

本書では、Red Hat Process Automation Manager 7.5 をインストール、設定し、パフォーマンスのチューニングをする方法について説明します。

目次

前書き

システム管理者として、Red Hat Process Automation Manager を実稼働環境にインストール、設定、アップグレードし、システム障害にすばやくかつ容易に対応できるようになり、システムが最適に稼働するようにできます。

前提条件

- Red Hat JBoss Enterprise Application Platform 7.2 がインストールされている。詳細は『Red Hat JBoss Enterprise Application Platform 7.2 インストールガイド』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、**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1章 RED HAT PROCESS AUTOMATION MANAGER コンポーネント

Red Hat Process Automation Manager は、Business Central および Process Server で設定されます。

- Business Central は、ビジネスルールを作成して管理するグラフィカルユーザーインターフェイスです。Business Central は、Red Hat JBoss EAP インスタンスまたは Red Hat OpenShift Container Platform (OpenShift) にインストールできます。
  Business Central は、スタンドアロンの JAR ファイルとしても使用できます。Business Central スタンドアロンの JAR ファイルとして使用して、アプリケーションサーバーにデプロイせずに Business Central を実行できます。

- Process Server は、ルールおよびその他のアーティファクトを実行するサーバーです。これは、ルールをインスタンス化して実行し、計画の問題を解決するために使用されます。Process Server を Red Hat JBoss EAP インスタンス、OpenShift、Oracle WebLogic Server インスタンス、IBM WebSphere Application Server インスタンスに、または Spring Boot アプリケーションの一部として、インストールできます。
  Process Server は、管理モードまたは非管理モードで動作するように設定できます。Process Server が非管理モードにある場合は、手動で KIE コンテナーを作成および維持する必要があります (デプロイメントユニット)。KIE コンテナーは、プロジェクトの特定のバージョンです。Process Server が管理モードにある場合は、Process Automation Manager コントローラーが Process Server 設定を管理するため、ユーザーがコントローラーと対話して KIE コンテナーの作成と維持を行います。

第2章 MAVEN を使用したシステム統合

Red Hat Process Automation Manager は、Red Hat JBoss Middleware Maven Repository と Maven Central リポジトリーを依存関係ソースとして使用するように作られています。これら両方の依存関係がプロセスビルドに利用可能になるようにしてください。

ご自分のプロジェクトがアーティファクトの特定バージョンに依存していることを確認してください。**LATEST** または **RELEASE** は、一般的に、アプリケーションの依存関係バージョンの特定と管理に使用されます。

- **LATEST** は、アーティファクトの最新デプロイ (スナップショット) バージョンになります。
- **RELEASE** は、リポジトリー内の最新の非スナップショットバージョンリリースになります。

LATEST または **RELEASE** を使用することで、サードパーティーのライブラリーの新リリース時にバージョン番号を更新する必要がなくなります。ただし、ソフトウェアリリースに影響を受けるビルドに対するコントロールができなくなることになります。

2.1. ローカルプロジェクトの PREEMPTIVE (先行) 認証

お使いの環境にインターネットアクセスがない場合には、Maven Central や他のパブリックリポジトリーの代わりに社内リポジトリーを設定します。Red Hat Process Automation Manager サーバーのリモート Maven リポジトリーからローカル Mave プロジェクトに JAR をインポートするには、リポジトリーサーバーの先行認証をオンにします。**pom.xml** ファイルの **guvnor-m2-repo** 用の認証を設定することでこれが実行できます。以下に例を示します。

```
<server>
  <id>guvnor-m2-repo</id>
  <username>admin</username>
  <password>admin</password>
  <configuration>
    <wagonProvider>httpclient</wagonProvider>
     <httpConfiguration>
       <all>
         <usePreemptive>true</usePreemptive>
       </all>
     </httpConfiguration>
  </configuration>
</server>
```

別の方法では、Authorization HTTP ヘッダーを Base64 でエンコードされた認証情報で設定できます。

```
<server>
  <id>guvnor-m2-repo</id>
  <configuration>
   <httpHeaders>
      <property>
        <name>Authorization</name>
        <!-- Base64-encoded "admin:admin" -->
        <value>Basic YWRtaW46YWRtaW4=</value>
      </property>
    </httpHeaders>
  </configuration>
</server>
```

2.2. BUSINESS CENTRAL における重複した GAV の検出

Business Central のすべての Maven リポジトリーで、プロジェクトの **GroupId**、**ArtifactId**、および **Version** (GAV) の各値が重複しているかどうかが確認されます。GAV が重複していると、実行された操作が取り消されます。

注記

重複する GAV の検出は、**Development Mode** のプロジェクトでは無効になっています。Business Central で重複する GAV 検出を有効にするには、プロジェクトの **Settings** → **General Settings** → **Version** に移動して、**Development Mode** オプションを **OFF** (該当する場合) に切り替えます。

重複した GAV の検出は、以下の操作を実行するたびに実行されます。

- プロジェクトのプロジェクト定義の保存。
- **pom.xml** ファイルの保存。
- プロジェクトのインストール、ビルド、またはデプロイメント。

以下の Maven リポジトリーで重複の GAV が確認されます。

- **pom.xml** ファイルの **<repositories>** 要素および **<distributionManagement>** 要素で指定されたリポジトリー。
- Maven の **settings.xml** 設定ファイルに指定されたリポジトリー。

2.3. BUSINESS CENTRAL における重複した GAV 検出設定の管理

admin ロールを持つ Business Central ユーザーは、プロジェクトで **GroupId** 値、**ArtifactId** 値、および **Version** 値 (GAV) が重複しているかどうかを確認するリポジトリーの一覧を修正できます。

注記

重複する GAV の検出は、**Development Mode** のプロジェクトでは無効になっています。Business Central で重複する GAV 検出を有効にするには、プロジェクトの **Settings** → **General Settings** → **Version** に移動して、**Development Mode** オプションを **OFF** (該当する場合) に切り替えます。

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. プロジェクトの **Settings** タブをクリックし、**Validation** をクリックしてリポジトリーの一覧を開きます。
3. 一覧表示したリポジトリーオプションの中から選択するか選択を解除して、重複した GAV の検出を有効または無効にします。
   今後、重複した GAV の報告は、検証を有効にしたリポジトリーに対してのみ行われます。

注記

この機能を無効にするには、システムの起動時に Business Central の **org.guvnor.project.gav.check.disabled** システムプロパティーを **true** に設定します。

```
$ ~/EAP_HOME/bin/standalone.sh -c standalone-full.xml
-Dorg.guvnor.project.gav.check.disabled=true
```

第3章 RED HAT PROCESS AUTOMATION MANAGER へのパッチ更新およびマイナーリリースアップグレードの適用

大抵の場合は、Business Central、Process Server、ヘッドレス Process Automation Manager コントローラーなど、Red Hat Process Automation Manager の特定コンポーネントの更新を容易にする自動更新ツールが Red Hat Process Automation Manager のパッチ更新と新規マイナーバージョンで提供されます。デシジョンエンジンやスタンドアロンの Business Central など、その他の Red Hat Process Automation Manager アーティファクトは、各マイナーリリースが含まれる新しいアーティファクトとしてリリースされるため、再インストールして更新を適用する必要があります。

この自動更新ツールを使ってパッチ更新とマイナーリリースアップグレードの両方を Red Hat Process Automation Manager 7.5 に適用することができます。バージョン 7.5 から 7.5.1 への更新といった Red Hat Process Automation Manager のパッチ更新には、最新のセキュリティー更新とバグ修正が含まれます。バージョン 7.5.x から 7.5 へのアップグレードといった Red Hat Process Automation Manager のマイナーリリースアップグレードには、機能強化、セキュリティー更新、バグ修正が含まれます。

注記

Red Hat Process Automation Manager への更新だけが、Red Hat Process Automation Manager パッチ更新に含まれます。Red Hat JBoss EAP への更新は、Red Hat JBoss EAP パッチ配信を使用して適用する必要があります。詳細は『Red Hat **JBoss EAP パッチおよびアップグレードガイド**』 を参照してください。

前提条件

- Red Hat Process Automation Manager インスタンスおよび Process Server インスタンスが実行していない。Red Hat Process Automation Manager または Process Server のインスタンスを実行している間は更新を適用しないでください。

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選択およびバージョンを選択します。
以下に例を示します。

   - **製品:** Process Automation Manager
   - **バージョン:** 7.5.1

   バージョン 7.5.x から 7.5 などのように Red Hat Process Automation Manager のマイナーリリースにアップグレードする場合は、お使いの Red Hat Process Automation Manager に最新のパッチ更新を適用してから、以下の手順にしたがって新たなマイナーリリースにアップグレードしてください。

2. **Patches** をクリックし、**Red Hat Process Automation Manager [VERSION] Patch Update**をダウンロードし、ダウンロードした **rhpam-$VERSION-update.zip** ファイルを一時ディレクトリーに展開します。
この更新ツールは、Business Central、Process Server、およびヘッドレス Process Automation Manager コントローラーなど、Red Hat Process Automation Manager の一定のコンポーネントの更新を自動化します。この更新ツールを使用して最初に更新を適用し、Red Hat Process Automation Manager ディストリビューションに関連するその他の更新、または新しいリリースアーティファクトをインストールします。

3. 更新ツールにファイルが更新されないようにするには、展開した **rhpam-$VERSION-update** フォルダーに移動し、**blacklist.txt** ファイルを開き、更新しないファイルの相対パスを追加します。
ファイルが **blacklist.txt** ファイルの一覧に追加されていると、更新スクリプトは、そのファイルを新しいバージョンに置き換えずにそのまま残し、新しいバージョンのファイルに **.new** 接尾辞を付けて追加します。ブラックリストのファイルが配布されなくなると、更新ツールは、**.removed** 接尾辞の付いた、空のマーカーファイルを作成します。次に、これらの新しいファイルを手動で保持、マージ、または削除することを選択できます。

   blacklist.txt ファイルで除外されるファイルの例:

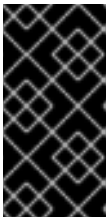
   ```
   WEB-INF/web.xml  // Custom file
   styles/base.css  // Obsolete custom file kept for record
   ```

   更新後の、ブラックリストに指定されたファイルディレクトリー内のコンテンツ:

   ```
   $ ls WEB-INF
   web.xml web.xml.new
   ```

   ```
   $ ls styles
   base.css base.css.removed
   ```

4. コマンドの端末で、**rhpam-$VERSION-update.zip** ファイルから展開した一時ディレクトリーに移動し、以下の形式で **apply-updates** スクリプトを実行します。

   重要

   更新を行う前に、Red Hat Process Automation Manager インスタンスおよび Process Server インスタンスが実行していないことを確認します。Red Hat Process Automation Manager または Process Server のインスタンスを実行している間は更新を適用しないでください。

   Linux システムまたは Unix ベースのシステムの場合:

   ```
   $ ./apply-updates.sh $DISTRO_PATH $DISTRO_TYPE
   ```

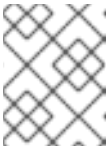
   Windows の場合:

   ```
   $ .\apply-updates.bat $DISTRO_PATH $DISTRO_TYPE
   ```

   $DISTRO_PATH の部分は、関連するディストリビューションディレクトリーへのパスで、**$DISTRO_TYPE** の部分は、更新しているディストリビューションの種類となります。

   Red Hat Process Automation Manager 更新ツールでは、以下のディストリビューションの種類がサポートされます。

   - **rhpam-business-central-eap7-deployable**: Business Central (**business-central.war**) を更新します。
   - **rhpam-kie-server-ee7**: Process Server (**kie-server.war**) を更新します。



注記

この更新ツールで、Red Hat JBoss EAP EE7 から Red Hat JBoss EAP EE8 に更新されます。

- **rhpam-controller-ee7**: ヘッドレス Process Automation Manager controller (**controller.war**) を更新します。

Red Hat JBoss EAP で、Red Hat Process Automation Manager の完全ディストリビューションに対する Business Central および Process Server への更新の例:

```
./apply-updates.sh ~EAP_HOME/standalone/deployments/business-central.war rhpam-business-central-eap7-deployable

./apply-updates.sh ~EAP_HOME/standalone/deployments/kie-server.war rhpam-kie-server-ee8
```

ヘッドレス Process Automation Manager コントローラーへの更新例 (使用している場合):

```
./apply-updates.sh ~EAP_HOME/standalone/deployments/controller.war rhpam-controller-ee7
```

この更新スクリプトは、展開した **rhpam-$VERSION-update** ディレクトリーに、指定したディストリビューションのコピーを含む **backup** ディレクトリーを作成してから、更新を行います。

5. 更新ツールが完了したら、更新ツールをダウンロードした、Red Hat カスタマーポータルの **Software Downloads** ページに戻り、Red Hat Process Automation Manager ディストリビューションに関するその他の更新または新しいリリースアーティファクトをインストールします。デシジョンエンジンまたはその他のアドオンに関する **.jar** ファイルなど、Red Hat Process Automation Manager ディストリビューションにすでに存在しているファイルについては、ファイルの既存のバージョンを Red Hat カスタマーポータルから取得した新しいバージョンに置き換えます。

6. エアギャップ環境など、スタンドアロンの **Red Hat Process Automation Manager 7.5.1 Maven Repository** アーティファクト (**rhpam-7.5.1-maven-repository.zip**)を使用している場合は、**Red Hat Process Automation Manager 7.5.x Maven Repository**をダウンロードして、ダウンロードした **rhpam-7.5.x-maven-repository.zip** ファイルを既存の **~/maven-repository** ディレクトリーに展開して、関連するコンテンツを更新します。
Maven リポジトリーの更新例:

```
$ unzip -o rhpam-7.5.1-maven-repository.zip 'rhba-7.5.1.GA-maven-repository/maven-repository/*' -d /tmp/rhbaMavenRepoUpdate

$ mv /tmp/rhbaMavenRepoUpdate/rhba-7.5.1.GA-maven-repository/maven-repository/ $REPO_PATH/
```

注記

更新が完了したら **/tmp/rhbaMavenRepoUpdate** ディレクトリーを削除してください。

7. 関連する更新をすべて適用したら、Red Hat Process Automation Manager および Process Server を起動して、Business Central にログインします。

8. Business Central 内のすべてのプロジェクトデータが存在して正確であることを確認し、Business Central ウィンドウの右上隅でプロファイル名をクリックし、**About** をクリックして、更新した製品バージョン番号を確認します。
   Business Central でエラーが発生したり、データが不足していることが通知されたら、**rhpam-$VERSION-update** ディレクトリーの **backup** ディレクトリーにコンテンツを復元し、更新ツールへの変更を戻します。Red Hat カスタマーポータルで Red Hat Process Automation Manager の以前のバージョンから、関連するリリースアーティファクトを再インストールできます。以前のディストリビューションを復元したら、更新を再実行してください。

第4章 PROCESS SERVER の設定と起動

Process Server の場所、ユーザー名、パスワード、その他の関連プロパティーは、Process Server の起動時に必要な設定を定義することで設定できます。

手順

Red Hat Process Automation Manager 7.5 の **bin** ディレクトリーに移動し、以下のプロパティーで新しい Process Server を起動します。お使いの環境に応じて特定のプロパティーを調整します。

```
$ ~/EAP_HOME/bin/standalone.sh --server-config=standalone-full.xml (1)
-Dorg.kie.server.id=myserver (2)
-Dorg.kie.server.user=process_server_username (3)
-Dorg.kie.server.pwd=process_server_password (4)
-Dorg.kie.server.controller=http://localhost:8080/business-central/rest/controller (5)
-Dorg.kie.server.controller.user=controller_username (6)
-Dorg.kie.server.controller.pwd=controller_password (7)
-Dorg.kie.server.location=http://localhost:8080/kie-server/services/rest/server (8)
-Dorg.kie.server.persistence.dialect=org.hibernate.dialect.PostgreSQLDialect (9)
-Dorg.kie.server.persistence.ds=java:jboss/datasources/psjbpmDS (10)
```

(1) **standalone-full.xml** サーバープロファイルの開始コマンド

(2) サーバー ID (Business Central で定義したサーバー設定名に一致させる必要がある)

(3) Process Automation Manager コントローラーから Process Server に接続する際のユーザー名

(4) Process Automation Manager コントローラーから Process Server に接続する際のパスワード

(5) Process Automation Manager コントローラーの場所 (**/rest/controller** 接尾辞が付いた Business Central URL)

(6) Process Automation Manager コントローラー REST API に接続するユーザー名

(7) Process Automation Manager コントローラー REST API に接続するパスワード

(8) プロセスサーバーの場所 (この例では Business Central と同じ場所)

(9) 使用する Hibernate の方言

(10) 以前の Red Hat JBoss BPM Suite データベースに使用されるデータソースの JNDI 名

注記

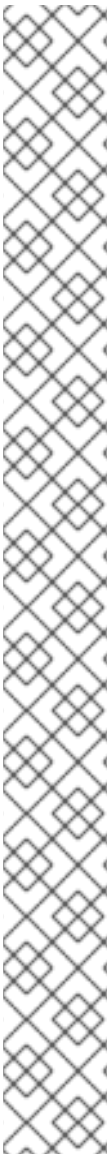
Business Central と Process Server が別々のアプリケーションサーバーインスタンス (Red Hat JBoss EAP など) にインストールされている場合は、Business Central とポートが競合しないように、Process Server の場所には別のポートを使用します。別の Process Server ポートが設定されていない場合は、ポートオフセットを追加して、Process Server プロパティーに従って Process Server のポート値を調整します。

以下に例を示します。

```
-Djboss.socket.binding.port-offset=150
-Dorg.kie.server.location=http://localhost:8230/kie-server/services/rest/server
```

この例のように、Business Central ポートが 8080 の場合は、定義したオフセットが 150 の Process Server ポートは 8230 です。

Process Server は、新しい Business Central に接続し、デプロイするデプロイメントユニット (KIE コンテナー) の一覧を収集します。



注記

依存関係の JAR ファイルでクラスを使用して Process Server クライアントから Process Server にアクセスすると、Business Central では **ConversionException** および **ForbiddenClassException** が発生します。Business Central でこれらの例外を発生させないようにするには、次のいずれかを実行します。

- クライアント側で例外が発生する場合は、kie-server クライアントに次のシステムプロパティーを追加します。

```
System.setProperty("org.kie.server.xstream.enabled.packages", "org.example.**");
```

- サーバー側で例外が発生する場合は、Red Hat Process Automation Manager インストールディレクトリーから **standalone-full.xml** を開き、<system-properties> タグに以下のプロパティーを設定します。

```
<property name="org.kie.server.xstream.enabled.packages" value="org.example.**"/>
```

- 以下の JVM プロパティーを設定します。

```
-Dorg.kie.server.xstream.enabled.packages=org.example.**
```

KJAR に存在するクラスは、これらのシステムプロパティーを使用して設定しないように想定されています。システムプロパティーでは既知のクラスのみを使用し、脆弱性を回避するようにしてください。

org.example はパッケージ例で、使用するパッケージを何でも定義できます。**org.example1.* * , org.example2.* * , org.example3.* *** などのように、コンマ区切りで、複数のパッケージを指定できます。

org.example1.Mydata1, org.example2.Mydata2 など、特定のクラスも追加できます。

第5章 PROCESS SERVER への JDBC データソースの設定

データソースは、アプリケーションサーバーなど、Java Database Connectivity (JDBC) クライアントを有効にするオブジェクトで、データベースへの接続を確立します。アプリケーションは、JNDI (Java Naming and Directory Interface) ツリーまたはローカルのアプリケーションコンテキストでデータソースを検索し、データベース接続を要求してデータを取得します。Process Server にデータソースを設定して、サーバーと、指定したデータベースとの間で適切なデータ交換を行う必要があります。

注記

実稼働環境の場合は、実際のデータソースを指定します。実稼働環境で、データソースの例は使用しないでください。

前提条件

- 『Red Hat JBoss Enterprise Application Server 設定ガイド』の「データソースの作成」と「JDBC ドライバー」のセクションで説明されているように、データベース接続の作成に使用する JDBC プロバイダーが、Process Server をデプロイするすべてのサーバーに設定 されている。
- Red Hat Process Automation Manager 7.5.1 Add Ons(rhpam-7.5.1-add-ons.zip)ファイルは、Red Hat カスタマーポータルの Software Downloads ページからダウンロードされている。

手順

1. 以下の手順を実行して、データベースを準備します。
   a. **TEMP_DIR** などの一時ディレクトリーに **rhpam-7.5-add-ons.zip** を展開します。
   b. **TEMP_DIR/rhpam-7.5-migration-tool.zip** を展開します。
   c. 現在のディレクトリーから、**TEMP_DIR/rhpam-7.5-migration-tool/ddl-scripts** ディレクトリーに移動します。このディレクトリーには、複数のデータベースタイプの DDL スクリプトが含まれています。
   d. 使用するデータベースに、お使いのデータベースタイプの DDL スクリプトをインポートします。以下に例を示します。

```
psql jbpm < /ddl-scripts/postgresql/postgresql-jbpm-schema.sql
```

2. テキストエディターで **EAP_HOME/standalone/configuration/standalone-full.xml** を開き、**<system-properties>** タグの場所を特定します。
3. 以下のプロパティーを **<system-properties>** タグに追加します。**<DATASOURCE>** はデータソースの JNDI 名で、**<HIBERNATE_DIALECT>** はデータベースの Hibernate 方言です。

注記

org.kie.server.persistence.ds プロパティーのデフォルト値は **java:jboss/datasources/ExampleDS** です。**org.kie.server.persistence.dialect** プロパティーのデフォルト値は **org.hibernate.dialect.H2Dialect** です。

```
<property name="org.kie.server.persistence.ds" value="<DATASOURCE>"/>
<property name="org.kie.server.persistence.dialect" value="<HIBERNATE_DIALECT>"/>
```

以下の例では、PostgreSQL hibernate 方言のデータソースの設定方法を紹介しています。

```
<system-properties>
  <property name="org.kie.server.repo" value="${jboss.server.data.dir}"/>
  <property name="org.kie.example" value="true"/>
  <property name="org.jbpm.designer.perspective" value="full"/>
  <property name="designerdataobjects" value="false"/>
  <property name="org.kie.server.user" value="rhpamUser"/>
  <property name="org.kie.server.pwd" value="rhpam123!"/>
  <property name="org.kie.server.location" value="http://localhost:8080/kie-
server/services/rest/server"/>
  <property name="org.kie.server.controller" value="http://localhost:8080/business-
central/rest/controller"/>
  <property name="org.kie.server.controller.user" value="kieserver"/>
  <property name="org.kie.server.controller.pwd" value="kieserver1!"/>
  <property name="org.kie.server.id" value="local-server-123"/>

  <!-- Data source properties. -->
  <property name="org.kie.server.persistence.ds"
value="java:jboss/datasources/KieServerDS"/>
  <property name="org.kie.server.persistence.dialect"
value="org.hibernate.dialect.PostgreSQLDialect"/>
</system-properties>
```

以下の方言がサポートされます。

- DB2: **org.hibernate.dialect.DB2Dialect**
- MSSQL: **org.hibernate.dialect.SQLServer2012Dialect**
- MySQL: **org.hibernate.dialect.MySQL5InnoDBDialect**
- MariaDB: **org.hibernate.dialect.MySQL5InnoDBDialect**
- Oracle: **org.hibernate.dialect.Oracle10gDialect**
- PostgreSQL: **org.hibernate.dialect.PostgreSQL82Dialect**
- PostgreSQL plus: **org.hibernate.dialect.PostgresPlusDialect**
- Sybase: **org.hibernate.dialect.SybaseASE157Dialect**

第6章 統合 PROCESS AUTOMATION MANAGER コントローラーを使用する PROCESS SERVER の設定

注記

本セクションの変更は、Process Server を Business Central で管理し、Red Hat Process Automation Manager を ZIP ファイルからインストールしている場合にのみ、実行してください。Business Central をインストールしていない場合は、「*7章 ヘッドレス Process Automation Manager コントローラーのインストールおよび実行*」の記載通りに、ヘッドレス Process Automation Manager コントローラーを使用して Process Server を管理することができます。

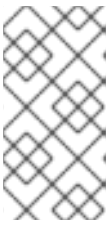
Process Server は管理モードにすることも、非管理モードにすることもできます。Process Server が非管理モードにある場合は、手動で KIE コンテナーを作成および維持する必要があります (デプロイメントユニット)。Process Server が管理モードにある場合は、Process Automation Manager コントローラーが Process Server 設定を管理するため、ユーザーがコントローラーと対話して KIE コンテナーの作成と維持を行います。

Process Automation Manager コントローラーは Business Central と統合します。Business Central をインストールしている場合は、Business Central の **Execution Server** ページを使用して Process Automation Manager コントローラーと対話します。

ZIP ファイルから Red Hat Process Automation Manager をインストールした場合は、Process Server および Business Central のインストールの **standalone-full.xml** ファイルを編集して、統合 Process Automation Manager コントローラーを持つ Process Server を設定する必要があります。

前提条件

- Business Central と Process Server が Red Hat JBoss EAP インストールのベースディレクトリー (**EAP_HOME**) にインストールされている。



注記

実稼働環境では、Business Central と Process Server を異なるサーバーにインストールすることを推奨します。ただし、たとえば開発環境で、Process Server と Business Central を同じサーバーにインストールする場合は、本セクションの説明に従って、共有の **standalone-full.xml** ファイルを変更します。

- Business Central サーバーノードに、**rest-all** ロールを持つユーザーが作成されている。

手順

1. Business Central の **EAP_HOME/standalone/configuration/standalone-full.xml** ファイルで、**<system-properties>** セクションの以下のプロパティーのコメントを解除し、**<USERNAME>** および **<USER_PWD>** を、**kie-server** ロールを持つユーザーの認証情報に置き換えます。

```
<property name="org.kie.server.user" value="<USERNAME>"/>
<property name="org.kie.server.pwd" value="<USER_PWD>"/>
```

2. Process Server の **EAP_HOME/standalone/configuration/standalone-full.xml** ファイルで、**<system-properties>** セクションの以下のプロパティーのコメントを解除します。

```
<property name="org.kie.server.controller.user" value="<CONTROLLER_USER>"/>
<property name="org.kie.server.controller.pwd" value="<CONTROLLER_PWD>"/>
<property name="org.kie.server.id" value="<KIE_SERVER_ID>"/>
<property name="org.kie.server.location" value="http://<HOST>:<PORT>/kie-server/services/rest/server"/>
<property name="org.kie.server.controller" value="<CONTROLLER_URL>"/>
```

3. 以下の値を置き換えてください。

- **<CONTROLLER_USER>** および **<CONTROLLER_PWD>** を **rest-all** ロールを持つユーザーの認証情報に置き換えます。
- &lt **;KIE_SERVER_ID** > を Process Server システムの ID または名前に置き換えます（例：**rhpam-7.5.1-process_server-1**）。
- **<HOST>** を Process Server ホストの ID または名前に置き換えます (例: **localhost** または **192.7.8.9**)。
- **<PORT>** を Process Server ホストのポートに置き換えます (例: **8080**)。

注記

org.kie.server.location プロパティーで Process Server の場所を指定します。

- **<CONTROLLER_URL>** を Business Central の URL に置き換えます。起動中に Process Server がこの URL に接続します。
  - インストーラーまたは Red Hat JBoss EAP zip ファイルを使用して Business Central をインストールした場合、**<CONTROLLER_URL>** は以下のようになります。
    http://<HOST>:<PORT>/business-central/rest/controller
  - **standalone.jar** ファイルを使用して Business Central を実行している場合、**<CONTROLLER_URL>** は以下のようになります。
    http://<HOST>:<PORT>/rest/controller

第7章 ヘッドレス PROCESS AUTOMATION MANAGER コントローラーのインストールおよび実行

Process Server は、管理モードまたは非管理モードで動作するように設定できます。Process Server が非管理モードにある場合は、手動で KIE コンテナーを作成および維持する必要があります (デプロイメントユニット)。Process Server が管理モードにある場合は、Process Automation Manager コントローラーが Process Server 設定を管理するため、ユーザーがコントローラーと対話して KIE コンテナーの作成と維持を行います。

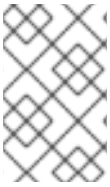
Business Central には Process Automation Manager コントローラーが組み込まれています。Business Central をインストールしている場合は、**Execution Server** ページを使用して KIE コンテナーを作成および維持します。Business Central なしで Process Server の管理を自動化するには、ヘッドレス Process Automation Manager コントローラーを使用することで可能になります。

7.1. インストーラーを使用した、PROCESS AUTOMATION MANAGER コントローラーを使用する PROCESS SERVER の設定

Process Server は Process Automation Manager コントローラーによる管理モードにすることも、非管理モードにすることもできます。Process Server が非管理モードにある場合は、手動で KIE コンテナーを作成および維持する必要があります (デプロイメントユニット)。Process Server が管理モードにある場合は、Process Automation Manager コントローラーが Process Server 設定を管理するため、ユーザーがコントローラーと対話して KIE コンテナーの作成と維持を行います。

Process Automation Manager コントローラーは Business Central と統合します。Business Central をインストールしている場合は、Business Central の **Execution Server** ページを使用して Process Automation Manager コントローラーと対話します。

インタラクティブモードまたは CLI モードでインストーラーを使用して Business Central および Process Server をインストールし、Process Automation Manager コントローラーで Process Server を設定します。



注記

Business Central をインストールしない場合は、「*7章 ヘッドレス Process Automation Manager コントローラーのインストールおよび実行*」でヘッドレス Process Automation Manager の使用方法を参照してください。

前提条件

- バックアップを作成済みの Red Hat JBoss EAP 7.2 サーバーインストールが設定された 2 台のコンピューターが利用できる。
- インストールを完了するのに必要なユーザーパーミッションが付与されている。

手順

1. 1 台目のコンピューターで、インタラクティブモードまたは CLI モードでインストーラーを実行します。詳細は **『Red Hat JBoss EAP 7.2 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。
2. **Component Selection** ページで、**Process Server** チェックボックスを外します。
3. Business Central インストールを完了します。

4. 2 台目のコンピューターで、インタラクティブモードまたは CLI モードでインストーラーを実行します。
5. **Component Selection** ページで **Business Central** チェックボックスを外します。
6. **Configure Runtime Environment** ページで **Perform Advanced Configuration** を選択します。
7. **Customize Process Server properties** を選択し、**Next** をクリックします。
8. Business Central のコントローラー URL を入力し、Process Server に追加のプロパティーを設定します。コントローラー URL は、以下の形式を取ります。**<HOST:PORT>** は、2 台目のコンピューターの Business Central のアドレスに置き換えます。

```
<HOST:PORT>/business-central/rest/controller
```

9. インストールを完了します。
10. Process Automation Manager コントローラーが Business Central と統合されていることを確認するには、Business Central の **Execution Servers** ページに移動して、設定した Process Server が **REMOTE SERVERS** に表示されていることを確認します。

7.2. ヘッドレス PROCESS AUTOMATION MANAGER コントローラーのインストール

ヘッドレス Process Automation Manager コントローラーをインストールして、REST API または Process Server Java Client API を使用して対話します。

前提条件

- バックアップを作成済みの Red Hat JBoss EAP システム (バージョン 7.2) が利用できる。Red Hat JBoss EAP システムのベースディレクトリーを **EAP_HOME** とする。
- インストールを完了するのに必要なユーザーパーミッションが付与されている。

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。
   - **製品:** Process Automation Manager
   - **バージョン:** 7.5
2. **Red Hat Process Automation Manager 7.5.1 Add Ons**（**rhpam-7.5.1-add-ons.zip** ファイル）をダウンロードします。
3. **rhpam-7.5.1-add-ons.zip** ファイルを展開します。**rhpam-7.5-controller-ee7.zip** ファイルは展開したディレクトリーにあります。
4. **rhpam-7.5-controller-ee7** 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-7.5-controller-ee7/controller.war** ディレクトリーを **EAP_HOME/standalone/deployments/** にコピーします。

警告

コピーするヘッドレス Process Automation Manager コントローラー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. **TEMP_DIR/rhpam-7.5-controller-ee7/SecurityPolicy/** ディレクトリーの内容を **EAP_HOME/bin** にコピーします。ファイルの上書きを確認するメッセージが表示されたら、**Yes** を選択します。

7. **EAP_HOME/standalone/deployments/** ディレクトリーに、**controller.war.dodeploy** という名前で空のファイルを作成します。このファイルにより、サーバーが起動するとヘッドレス Process Automation Manager コントローラーが自動的にデプロイされます。

7.2.1. ヘッドレス Process Automation Manager コントローラーのユーザー作成

ヘッドレス Process Automation Manager コントローラーを使用する前に、**kie-server** ロールを持つユーザーを作成する必要があります。

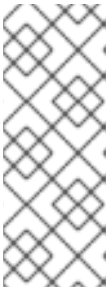
前提条件

- ヘッドレス Process Automation Manager コントローラーが Red Hat JBoss EAP インストールのベースディレクトリー (**EAP_HOME**) にインストールされている。

手順

1. 端末アプリケーションで **EAP_HOME/bin** ディレクトリーに移動します。
2. 以下のコマンドを入力し、**<USER_NAME>** および **<PASSWORD>** を、作成するユーザー名およびパスワードに置き換えます。

```
$ ./add-user.sh -a --user <USER_NAME> --password <PASSWORD> --role kie-server
```



注記

必ず、既存のユーザー、ロール、またはグループとは異なるユーザー名を指定してください。たとえば、**admin** という名前のユーザーは作成しないでください。

パスワードは 8 文字以上で、数字と、英数字以外の文字をそれぞれ 1 文字以上使用する必要があります。ただし & の文字は使用できません。

3. ユーザー名とパスワードを書き留めておきます。

7.2.2. Process Server およびヘッドレス Process Automation Manager コントローラーの設定

Process Server をヘッドレス Process Automation Manager コントローラーで管理する場合は、本セクションの説明に従って Process Server インストールの **standalone-full.xml** ファイルとヘッドレス

Process Automation Manager コントローラーの **standalone.xml** ファイルを編集する必要があります。

前提条件

- Process Server が Red Hat JBoss EAP インストールのベースディレクトリー (**EAP_HOME**) にインストールされている。
- ヘッドレス Process Automation Manager コントローラーが **EAP_HOME** にインストールされている。

注記

実稼働環境では Process Server およびヘッドレス Process Automation Manager コントローラーを異なるサーバーにインストールすることを推奨します。ただし、開発環境など、Process Server およびヘッドレス Process Automation Manager コントローラーを同じサーバーにインストールする場合は、併せて共有の **standalone-full.xml** ファイルを変更します。

- Process Server ノードに、**kie-server** ロールを持つユーザーが作成されている。
- サーバーノードに、**kie-server** ロールのあるユーザーが作成されている。

手順

1. **EAP_HOME/standalone/configuration/standalone-full.xml** ファイルの **<system-properties>** セクションに以下のプロパティーを追加し、**<USERNAME>** および **<USER_PWD>** を、**kie-server** ロールを持つユーザーの認証情報に置き換えます。

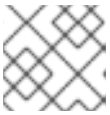
```
<property name="org.kie.server.user" value="<USERNAME>"/>
<property name="org.kie.server.pwd" value="<USER_PWD>"/>
```

2. Process Server の **EAP_HOME/standalone/configuration/standalone-full.xml** ファイルの **<system-properties>** セクションに以下のプロパティーを追加します。

```
<property name="org.kie.server.controller.user" value="<CONTROLLER_USER>"/>
<property name="org.kie.server.controller.pwd" value="<CONTROLLER_PWD>"/>
<property name="org.kie.server.id" value="<KIE_SERVER_ID>"/>
<property name="org.kie.server.location" value="http://<HOST>:<PORT>/kie-server/services/rest/server"/>
<property name="org.kie.server.controller" value="<CONTROLLER_URL>"/>
```

3. このファイルで、以下の値を置き換えます。

- **<CONTROLLER_USER>** および **<CONTROLLER_PWD>** を **kie-server** ロールを持つユーザーの認証情報に置き換えます。
- &lt **;KIE_SERVER_ID** > を Process Server システムの ID または名前に置き換えます（例：**rhpam-7.5.1-process_server-1**）。
- **<HOST>** を Process Server ホストの ID または名前に置き換えます (例: **localhost** または **192.7.8.9**)。
- **<PORT>** を Process Server ホストのポートに置き換えます (例: **8080**)。



注記

org.kie.server.location プロパティーで Process Server の場所を指定します。

- **<CONTROLLER_URL>** をヘッドレス Process Automation Manager コントローラー の URL で置き換えます。

    1. 起動中に Process Server がこの URL に接続します。

7.3. ヘッドレス PROCESS AUTOMATION MANAGER コントローラーの実行

ヘッドレス Process Automation Manager コントローラーを Red Hat JBoss EAP にインストールしたら、以下の手順に従ってヘッドレス Process Automation Manager コントローラーを実行します。

前提条件

- ヘッドレス Process Automation Manager コントローラーが Red Hat JBoss EAP インストールのベースディレクトリー (**EAP_HOME**) にインストールされ設定されている。

手順

1. ターミナルアプリケーションで **EAP_HOME/bin** に移動します。
2. ヘッドレス Process Automation Manager コントローラーを、Process Server をインストールした Red Hat JBoss EAP インスタンスと同じ Red Hat JBoss EAP インスタンスにインストールしている場合は、以下のいずれかのコマンドを実行します。

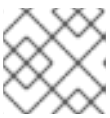
    - Linux または UNIX ベースのシステムの場合:

    ```
    $ ./standalone.sh -c standalone-full.xml
    ```

    - Windows の場合:

    ```
    standalone.bat -c standalone-full.xml
    ```

3. ヘッドレス Process Automation Manager コントローラーを、Process Server をインストールした Red Hat JBoss EAP インスタンスとは別の Red Hat JBoss EAP インスタンスにインストールしている場合は、**standalone.sh** スクリプトで Process Automation Manager コントローラーを開始できます。

    注記

    この場合は、**standalone.xml** ファイルに必要な設定変更を加えます。

    - Linux または UNIX ベースのシステムの場合:

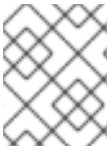
    ```
    $ ./standalone.sh
    ```

    - Windows の場合:

    ```
    standalone.bat
    ```

4. ヘッドレス Process Automation Manager コントローラーが Red Hat JBoss EAP 上で動作していることを確認するには、以下のコマンドを入力します。**<CONTROLLER>** はユーザー名で、**<CONTROLLER_PWD>** はパスワードになります。ここで、<CONTROLLER> と <CONTROLLER_PWD> は、で作成したユーザー名とパスワードの組み合わせです。このコマンドにより、Process Server インスタンスに関する情報が出力されます。

```
curl -X GET "http://<HOST>:<PORT>/controller/rest/controller/management/servers" -H "accept: application/xml" -u '<CONTROLLER>:<CONTROLLER_PWD>'
```



注記

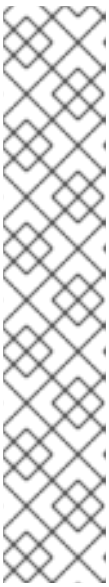
あるいは、Process Server Java API Client を使用して、ヘッドレス Process Automation Manager コントローラーにアクセスすることもできます。

7.4. ヘッドレス PROCESS AUTOMATION MANAGER コントローラーを使用した PROCESS SERVER のクラスターリング

Process Automation Manager コントローラーは Business Central と統合します。ただし、Business Central をインストールしない場合は、ヘッドレス Process Automation Manager コントローラーをインストールし、REST API または Process Server Java Client API を使用してそのコントローラーと対話します。

前提条件

- バックアップを作成してある Red Hat JBoss EAP システム (バージョン 7.2 またはそれ以降) が利用できる。Red Hat JBoss EAP システムのベースディレクトリーを **EAP_HOME** とする。
- インストールを完了するのに必要なユーザーパーミッションが付与されている。
- マウントしたパーティションが存在する NFS サーバーが利用できる。



注記

マウントしたパーティションで NFS 共有を設定するには、以下の手順を実行します。

1. NFS サーバーを設定します。詳細は、「RHEL 7 で NFS を設定する」を参照してください。
2. 共有フォルダーを追加するには、以下のコマンドを実行します。

```
# vi /etc/exports
/data/shared *(rw,sync,no_root_squash)
```

/data/shared は共有フォルダー、* は NFS サーバーに接続可能な IP アドレス、**(rw,sync,no_root_squash)** は NFS に最小限必要なオプションを指します。

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**

- **バージョン: 7.5**

2. Red Hat Process Automation Manager 7.5.1 Add Ons（**rhpam-7.5.1-add-ons.zip** ファイル）をダウンロードします。

3. **rhpam-7.5.1-add-ons.zip** ファイルを展開します。**rhpam-7.5-controller-ee7.zip** ファイルは展開したディレクトリーにあります。

4. **rhpam-7.5-controller-ee7** 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-7.5-controller-ee7/controller.war** ディレクトリーを **EAP_HOME/standalone/deployments/** にコピーします。

警告

コピーするヘッドレス Process Automation Manager コントローラー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. **TEMP_DIR/rhpam-7.5-controller-ee7/SecurityPolicy/** ディレクトリーの内容を **EAP_HOME/bin** にコピーします。ファイルの上書きを確認するメッセージが表示されたら、**Yes** を選択します。

7. **EAP_HOME/standalone/deployments/** ディレクトリーに、**controller.war.dodeploy** という名前で空のファイルを作成します。このファイルにより、サーバーが起動するとヘッドレス Process Automation Manager コントローラーが自動的にデプロイされます。

8. テキストエディターで **EAP_HOME/standalone/configuration/standalone.xml** ファイルを開きます。

9. 以下のプロパティーを **<system-properties>** 要素に追加し、**<NFS_STORAGE>** を、テンプレート設定が保存されている NFS ストレージへの絶対パスに置き換えます。

```
<system-properties>
  <property name="org.kie.server.controller.templatefile.watcher.enabled" value="true"/>
  <property name="org.kie.server.controller.templatefile" value="<NFS_STORAGE>"/>
</system-properties>
```

テンプレートファイルには、特定のデプロイメントシナリオのデフォルト設定が含まれます。

org.kie.server.controller.templatefile.watcher.enabled プロパティーの値を true に設定すると、別のスレッドが開始してテンプレートファイルの修正を監視します。この確認の間隔はデフォルトで 30000 ミリ秒になり、**org.kie.server.controller.templatefile.watcher.interval** システムプロパティーで制御できます。このプロパティーの値を false に設定すると、テンプレートファイルへの変更の検出が、サーバーの再起動時に制限されます。

10. ヘッドレス Process Automation Manager コントローラーを開始するには、**EAP_HOME/bin** に移動して、以下のコマンドを実行します。

- Linux または UNIX ベースのシステムの場合:

-

```
$ ./standalone.sh
```

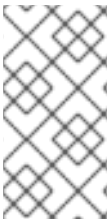
- Windows の場合:

```
standalone.bat
```

Red Hat JBoss Enterprise Application Platform のクラスター環境で Red Hat Process Automation Manager を稼働する方法の詳細情報は、**『Red Hat JBoss EAP クラスター環境での 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。

第8章 BUSINESS CENTRAL に接続するように PROCESS SERVER の設定

Process Server を Red Hat Process Automation Manager 環境に設定していない場合、または Red Hat Process Automation Manager 環境に Process Server を追加する必要がある場合は、Process Server を設定して Business Central に接続する必要があります。



注記

Red Hat OpenShift Container Platform に Process Server をデプロイする場合は、『Red Hat OpenShift Container Platform への Red Hat Process Automation Manager フリーフォーム管理サーバー環境のデプロイ』で、Business Central に接続する設定手順を参照してください。

前提条件

- Process Server がインストールされている。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。

手順

1. Red Hat Process Automation Manager インストールディレクトリーで、**standalone-full.xml** ファイルに移動します。たとえば、Red Hat Process Automation Manager に Red Hat JBoss EAP インストールを使用する場合は **$EAP_HOME/standalone/configuration/standalone-full.xml** に移動します。

2. **standalone-full.xml** を開き、**<system-properties>** タグの下に、以下のプロパティーを設定します。

   - **org.kie.server.controller.user:** Business Central にログインするユーザーのユーザー名。
   - **org.kie.server.controller.pwd:** Business Central にログインするユーザーのパスワード。
   - **org.kie.server.controller:** Business Central の API に接続する URL。通常、URL は **http://<centralhost>:<centralport>/business-central/rest/controller** です。**<centralhost>** と **<centralport>** はそれぞれ Business Central のホスト名とポートになります。Business Central を OpenShift にデプロイしている場合は、URL から **business-central/** を削除します。
   - **org.kie.server.location:** Process Server の API に接続する URL。通常、URL は **http://<serverhost>:<serverport>/kie-server/services/rest/server** (**<serverhost>** および **<serverport>** はそれぞれ Process Server のホスト名およびポート) になります。
   - **org.kie.server.id:** サーバー設定の名前。このサーバー設定が Business Central に存在しない場合は、Process Server が Business Central に接続する時に自動的に作成されます。

   以下に例を示します。

```
<property name="org.kie.server.controller.user" value="central_user"/>
<property name="org.kie.server.controller.pwd" value="central_password"/>
<property name="org.kie.server.controller" value="http://central.example.com:8080/business-central/rest/controller"/>
<property name="org.kie.server.location" value="http://kieserver.example.com:8080/kie-server/services/rest/server"/>
<property name="org.kie.server.id" value="production-servers"/>
```

-

3. Process Server を起動または再起動します。

第9章 PROCESS SERVER および BUSINESS CENTRAL での環境モードの設定

Process Server は、**production** (実稼働) モードと **development** (開発) モードでの実行が設定可能です。開発モードでは、柔軟な開発ポリシーが提供され、小規模な変更の場合はアクティブなプロセスインスタンスを維持しながら、既存のデプロイメントユニット (KIE コンテナー) を更新できます。また、大規模な変更の場合は、アクティブなプロセスインスタンスを更新する前に、デプロイメントユニットの状態をリセットすることも可能です。実稼働モードは、各デプロイメントで新規デプロイメントユニットが作成される実稼働環境に最適です。

開発環境では、**Deploy** をクリックすると、（該当する場合）実行中のインスタンスを **中止** することなくビルドした KJAR ファイルを Process Server にデプロイすることができます。または Redeploy をクリックして、ビルドした KJAR ファイルをデプロイして実行中のインスタンスを中止することもできます。ビルドした KJAR ファイルを次回に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 Process Server で自動的に更新されます。

実稼働環境では、Business Central の **Redeploy** オプションは無効になり、**Deploy** をクリックして、ビルドした KJAR ファイルを Process 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。

手順

1. Process Server 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。

2. Business Central のプロジェクトにデプロイメントの動作を設定するには、プロジェクトの **Settings → General Settings → Version** に移動して、**Development Mode** オプションを切り替えます。

注記

デフォルトでは、Process Server と Business Central の新規プロジェクトはすべて、開発モードになっています。

Development Mode がオンのプロジェクトや、実稼働モードの Process 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することはできません。

第10章 BUSINESS CENTRAL で管理する PROCESS SERVER の設定

警告

このセクションでは、テスト目的で使用可能なサンプルの設定を紹介します。一部の値は、実稼働環境には適しておらず、その旨を記載しています。

以下の手順を使用して、Process Server インスタンスを管理するように Business Central を設定します。

前提条件

- 以下のロールを持つユーザーが存在している
  - Business Central: **rest-all** ロールを持つユーザー
  - Process Server: **kie-server** ロールを持つユーザー

注記

実稼働環境では、2 人の異なるユーザーを使用し、それぞれロールを 1 つ割り当ててください。このサンプルでは、**rest-all** と **kie-server** の両ロールを持つ **controllerUser** という名前のユーザー 1 人のみを使用します。

手順

1. 以下の JVM プロパティーを設定します。
   Business Central と Process Server の場所は異なる可能性があります。このような場合、正しいサーバーインスタンスのプロパティーを設定するようにしてください。
   - Red Hat JBoss EAP で、以下のファイルの **<system-properties>** セクションを変更します。
     - スタンドアロンモードの場合: **EAP_HOME/standalone/configuration/standalone*.xml**
     - ドメインモードの場合: **EAP_HOME/domain/configuration/domain.xml**

表10.1 Process Server インスタンスの JVM プロパティー

プロパティー	値	注記
org.kie.server.id	**default-kie-server**	Process Server の ID
org.kie.server.controller	**http://localhost:8080/business-central/rest/controller**	Business Central の場所

プロパティー	値	注記
org.kie.server.controller.user	**controllerUser**	前のステップで説明した **rest-all** ロールを持つユーザーの名前
org.kie.server.controller.pwd	**controllerUser1234;**	前のステップで説明したユーザーのパスワード
org.kie.server.location	**http://localhost:8080/kie-server/services/rest/server**	Process Server の場所

表10.2 Business Central インスタンスの JVM プロパティー

プロパティー	値	注記
org.kie.server.user	**controllerUser**	前のステップで説明した **kie-server** ロールを持つユーザーの名前
org.kie.server.pwd	**controllerUser1234;**	前のステップで説明したユーザーのパスワード

2. **http://SERVER:PORT/kie-server/services/rest/server/** に GET リクエストを送信して Process Server が正常に起動したことを確認します。認証が終わると、以下のような XML 応答が返されます。

```
<response type="SUCCESS" msg="Kie Server info">
  <kie-server-info>
    <capabilities>KieServer</capabilities>
    <capabilities>BRM</capabilities>
    <capabilities>BPM</capabilities>
    <capabilities>CaseMgmt</capabilities>
    <capabilities>BPM-UI</capabilities>
    <capabilities>BRP</capabilities>
    <capabilities>DMN</capabilities>
    <capabilities>Swagger</capabilities>
    <location>http://localhost:8230/kie-server/services/rest/server</location>
    <messages>
      <content>Server KieServerInfo{serverId='first-kie-server', version='7.5.1.Final-redhat-
1', location='http://localhost:8230/kie-server/services/rest/server', capabilities=[KieServer,
BRM, BPM, CaseMgmt, BPM-UI, BRP, DMN, Swagger]}started successfully at Mon Feb 05
15:44:35 AEST 2018</content>
      <severity>INFO</severity>
      <timestamp>2018-02-05T15:44:35.355+10:00</timestamp>
    </messages>
    <name>first-kie-server</name>
    <id>first-kie-server</id>
    <version>7.5.1.Final-redhat-1</version>
  </kie-server-info>
</response>
```

3. 登録が正常に完了したことを確認します。

   a. Business Central にログインします。

   b. **Menu → Deploy → Execution Servers** の順にクリックします。
   正常に登録されている場合には、登録されたサーバーの ID が表示されます。

10.1. TLS 対応の SMART ROUTER の設定

TLS 対応の Smart Router (以前の KIE Server Router) 設定か可能となり、HTTPS トラフィックが使用できます。

手順

- ターミナルを開いて以下のコマンドを実行し、TLS 対応の Smart Router を起動します。

```
java -Dorg.kie.server.router.tls.keystore=PATH_TO_YOUR_KEYSTORE
    -Dorg.kie.server.router.tls.keystore.password=YOUR_KEYSTORE_PASSWD
    -Dorg.kie.server.router.tls.keystore.keyalias=YOUR_KEYSTORE_ALIAS
    -jar kie-server-router-proxy-YOUR_VERSION.jar
```

PATH_TO_YOUR_KEYSTORE、**YOUR_KEYSTORE_PASSWD**、**YOUR_KEYSTORE_ALIAS**、および **YOUR_VERSION** をそれぞれ関連データで置き換えます。

第11章 管理対象 PROCESS SERVER

管理対象インスタンスには、Process Server を起動するために利用可能な Process Automation Manager コントローラーが必要です。

Process Automation Manager コントローラーは、Process Server の設定を一元的に管理します。各 Process Automation Manager コントローラーは複数の設定を一度に管理でき、環境内に複数の Process Automation Manager コントローラーを配置できます。管理対象の Process Server に複数の Process Automation Manager コントローラーを設定できますが、一度に接続できるのは 1 台だけです。

重要

どの Process Automation Manager コントローラーに接続されても同じ設定セットがサーバーに提供されるように、Process Automation Manager コントローラーはすべて同期する必要があります。

Process Server に複数のコントローラーが設定されている場合には、いずれかのコントローラーとの接続が正常に確立されるまで、起動時に各コントローラーに対して接続を試みます。接続を確立できない場合は、設定でローカルのストレージが利用可能な場合でもサーバーは起動しません。こうすることで、整合性を保ち、冗長設定でサーバーが実行されるのを回避します。

注記

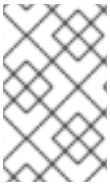
Process Automation Manager コントローラーに接続せずにスタンドアロンモードで Process Server を実行する方法については、12章*非管理対象 Process Server* を参照してください。

第12章 非管理対象 PROCESS SERVER

管理対象外の Process Server はスタンドアロンインスタンスであるため、Process Server 自体から REST/JMS API を使用して個別に設定する必要があります。再起動時には、サーバーが自動的に設定をファイルに永続化し、そのファイルが内部のサーバーの状態として使用されます。

以下の操作を実行中に、設定が更新されます。

- KIE コンテナーのデプロイ
- KIE コンテナーのデプロイ解除
- KIE コンテナーの起動
- KIE コンテナーの停止



注記

Process Server が再起動すると、シャットダウン前に永続化された状態を再度確立しようと試みます。そのため、実行していた KIE コンテナー (デプロイメントユニット) は起動しますが、停止していたコンテナーは起動しません。

第13章 PROCESS SERVER での KIE コンテナーのアクティブ化および非アクティブ化

特定のコンテナーを非アクティブにすることで、既存のプロセスインスタンスおよびタスクの作業を継続したまま、新規プロセスインスタンスの作成を停止できます。非アクティブ化が一時的な場合は、後でコンテナーを再度アクティブにできます。KIE コンテナーのアクティブ化および非アクティブ化には、KIE Server の再起動は必要ありません。

前提条件

- Business Central で Kie コンテナーが作成および設定されている。

手順

1. Business Central にログインします。
2. メインメニューで、**Menu → Deploy → Execution Servers** の順にクリックします。
3. パージの左側にある **Server Configurations** ペインからサーバーを選択します。
4. **Deployment Units** ペインから、アクティブ化または非アクティブ化するデプロイメントユニットを選択します。
5. デプロイメントユニットペインの右上にある **Activate** または **Deactivate** をクリックします。非アクティブ化されたら、KIE コンテナーからプロセスインスタンスを作成できません。

第14章 デプロイメント記述子

プロセスとルールは Apache Maven ベースのパッケージに保存され、ナレッジアーカイブ、または KJAR と呼ばれます。ルール、プロセス、アセット、およびその他のプロジェクトアーティファクトは、Maven がビルドおよび管理する JAR ファイルの一部です。**kmodule.xml** と呼ばれる、KJAR の **META-INF** ディレクトリー内に保存されるファイルを使用して、KIE ベースとセッションを定義できます。デフォルトでは、この **kmodule.xml** ファイルは空です。

Business Central のようなランタイムコンポーネントが KJAR を処理しようとする際には、ランタイム表記のビルドのために **kmodule.xml** を検索します。

デプロイメント記述子は **kmodule.xml** ファイルを補い、デプロイメントにおいてより詳細な制御を提供します。このような記述子は任意で、記述子がなくてもデプロイメントは正常に行われます。記述子を使用して、persistence、auditing、runtime strategy といったメタ値を含む技術的属性を設定することができます。

記述子を使用すると、(サーバーレベルのデフォルト、KJAR ごとに異なるデプロイメント記述子、その他のサーバー設定など) 複数レベルで Process Server を設定できるようになります。記述子を使用して、デフォルトの Process Server 設定にシンプルなカスタマイズが可能になります (KJAR ごとなど)。

記述子は **kie-deployment-descriptor.xml** と呼ばれるファイルで定義し、**META-INF** ディレクトリーの **kmodule.xml** ファイルの隣に置くことができます。このデフォルトの場所とファイル名は、システムパラメーターとして指定すると変更できます。

```
-Dorg.kie.deployment.desc.location=file:/path/to/file/company-deployment-descriptor.xml
```

14.1. デプロイメント記述子の設定

デプロイメント記述子を使用すると、ユーザーは以下の複数レベルで実行サーバーを設定することができるようになります。

- **サーバーレベル**: メインのレベルで、サーバーにデプロイされているすべての KJAR に適用されます。
- **KJAR レベル**: このレベルでは、KJAR ベースで記述子を設定できます。
- **デプロイ時レベル**: KJAR のデプロイ時に適用される記述子です。

デプロイメント記述子で指定されたより詳細な設定アイテムは、マージされるコレクションベースの設定アイテムを除いて、サーバーレベルのものよりも優先されます。優先順位は、**デプロイ時設定 > KJAR 設定 > サーバー設定** となります。

注記

デプロイ時の設定は、REST API によるデプロイメントに適用されます。

たとえば、サーバーレベルで定義された (設定可能なアイテムの 1 つである) persistence mode が **NONE** で、同じモードが KJAR レベルでは **JPA** と指定されている場合、その KJAR の実際のモードは **JPA** になります。その KJAR についてデプロイメント記述子で persistence mode に何も指定されていない場合 (またはデプロイメント記述子がない場合) は、サーバーレベルの設定にフォールバックします。このケースでは、**NONE** (またはサーバーレベルのデプロイメント記述子がない場合は **JPA**) になります。

設定内容

デプロイメント記述子では、高度な技術的設定が可能です。以下の表では、設定可能な詳細と、それぞれの許容値とデフォルト値を掲載しています。

表14.1 デプロイメント記述子

設定	XML エントリー	許容値	デフォルト値
ランタイムデータの永続ユニット名	persistence-unit	有効な永続パッケージ名	org.jbpm.domain
監査データの永続ユニット名	audit-persistence-unit	有効な永続パッケージ名	org.jbpm.domain
永続モード	persistence-mode	JPA, NONE	JPA
監査モード	audit-mode	JPA、JMS、または NONE	JPA
ランタイムストラテジー	runtime-strategy	SINGLETON、PER_REQUEST、または PER_PROCESS_INSTANCE	SINGLETON
登録するイベントリスナー一覧	event-listeners	**ObjectModel** のような有効なリスナークラス名	デフォルト値なし
登録するタスクイベントリスナー一覧	task-event-listeners	**ObjectModel** のような有効なリスナークラス名	デフォルト値なし
登録する作業アイテムハンドラー一覧	work-item-handlers	**NamedObjectHandler** のような有効な作業アイテムハンドラークラス	デフォルト値なし
登録するグローバル一覧	globals	**NamedObjectModel** のような有効なグローバル変数	デフォルト値なし
登録するマーシャリングストラテジー (プラグ可能変数永続)	marshalling-strategies	有効な **ObjectModel** クラス	デフォルト値なし
KJAR のリソースにアクセス可能となるために必要なロール	required-roles	文字列のロール名	デフォルト値なし
KIE セッションの追加の環境エントリー	environment-entries	有効な **NamedObjectModel**	デフォルト値なし
KIE セッションの追加の設定オプション	configurations	有効な **NamedObjectModel**	デフォルト値なし

設定	XML エントリー	許容値	デフォルト値
リモートサービスのシリアル化に使用するクラス	remoteable-class	有効な **CustomClass**	デフォルト値なし

警告

EJB Timer スケジューラー (Process Server のデフォルトのスケジューラー) では、Singleton ランタイム戦略を使用しないでください。この組み合わせを使用すると、負荷がかかると、Hibernate で問題が発生する可能性があります。具体的に他のストラテジーを使用する理由がない限り、プロセスインスタンス別のランタイムストラテジーの使用を推奨します。この制約に関する詳細情報は、Hibernate issues with Singleton strategy and EJBTimerScheduler を参照してください。

14.2. デプロイメント記述子の管理

デプロイメント記述子を設定するには、Business Central で **Menu → Design → $PROJECT_NAME → Settings → Deployments** と移動します。

プロジェクトが作成されるたびに、ストックの **kie-deployment-descriptor.xml** ファイルがデフォルト値で生成されます。

すべての KJAR で完全なデプロイメント記述子を提供する必要はありません。部分的なデプロイメント記述子の提供は可能で、かつ推奨されるものです。たとえば、異なる監査モードを使用する必要がある場合は、その KJAR のみにそれを指定し、残りの属性はサーバーレベルのデフォルト値で定義します。

OVERRIDE_ALL マージモードの使用時には、すべての設定アイテムを指定する必要があります。関連する KJAR は常に指定された設定を使用し、階層内の他のデプロイメント記述子とマージしないためです。

14.3. ランタイムエンジンへのアクセス制限

required-roles 設定アイテムは、デプロイメント記述子で編集できます。このプロパティーが定義するグループに属するユーザーにのみ特定プロセスへのアクセスを付与することで、KJAR ごとまたはサーバーレベルごとにランタイムエンジンへのアクセスを制限します。

セキュリティーロールを使用してプロセス定義へのアクセスを制限したり、ランタイムでのアクセスを制限することができます。

リポジトリーの制限に基づいてこのプロパティーに必要なロールを追加するのがデフォルトの動作になります。必要な場合は、セキュリティーレルムで定義されている実際のロールに合致するロールを提供することで、このプロパティーを手動で変更できます。

手順

1. プロジェクトのデプロイメント記述子設定を開くには、Business Central で **Menu → Design → $PROJECT_NAME → Settings → Deployments** の順に選択します。

2. 設定一覧から、**Required Roles** をクリックし、次に **Add Required Role** をクリックします。
3. **Add Required Role** ウィンドウで、このデプロイメントにアクセスをするパーミッションのロール名を入力し、**Add** をクリックします。
4. デプロイメントにアクセスする権限を持つロールをさらに追加するには、前の手順を繰り返します。
5. すべてのロールを追加したら、**Save** をクリックします。

第15章 BUSINESS CENTRAL からのランタイムデータへのアクセス

Business Central の以下のページでは、Process Server のランタイムデータを表示できます。

- **プロセスレポート**
- **タスクレポート**
- **プロセス定義**
- **プロセスインスタンス**
- **実行エラー**
- **ジョブ**
- **タスク**

これらのページは、現在ログインしているユーザーの認証情報を使用して、Process Server からデータを読み込みます。したがって、Business Central でランタイムデータを表示できるようにするには、以下の条件を満たしていることを確認してください。

- Business Central アプリケーションを実行している KIE コンテナー (デプロイメントユニット) にユーザーが存在している。このユーザーはランタイムデータへのフルアクセスと **kie-server** ロールのほかに、**admin**、**analyst**、**developer** のいずれかのロールが必要です。**manager** および **process_admin** のロールでも、Business Central のランタイムデータページにアクセスできます。
- Process Server を実行している KIE コンテナー (デプロイメントユニット) にユーザーが存在し、**kie-server** ロールがあること。
- Business Central と Process Server の通信が確立されていること。つまり、 Business Central の一部である Process Automation Manager コントローラーに Process Server が登録されていること。
- Business Central を実行しているサーバーの **standalone.xml** 設定に以下の **deployment.business-central.war** ログインモジュールが存在する。

```
<login-module code="org.kie.security.jaas.KieLoginModule" flag="optional"
module="deployment.business-central.war"/>
```

第16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。この痕跡はログにのみ表示され、通常は、プロセスエンジンに要求を送信した人にしか表示されません。

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が、Business Central の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

設定可能なエラー処理は、プロセスエンジンの実行 (タスクサービスを含む) 時に発生した技術エラーに対応します。以下の技術例外が適用されます。

- **java.lang.Throwable** を拡張するすべてのもの
- プロセスレベルのエラー処理およびその他の例外が事前に処理されていない

エラー処理メカニズムを設定し、その機能を拡張するプラグ可能なアプローチが可能なコンポーネントが複数あります。

エラー処理を行うプロセスエンジンのエントリーポイントは **ExecutionErrorManager** です。これは、**RuntimeManager** と統合され、基盤となる **KieSession** および **TaskService** に渡します。

API の観点からすると、**ExecutionErrorManager** で次のコンポーネントにアクセスできます。

- **ExecutionErrorHandler**: エラー処理の主なメカニズム
- **ExecutionErrorStorage**: 実行エラー情報のための、プラグ可能なストレージ

16.1. 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。エラーは、確認されたかどうかに基づいてフィルターリングされます。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。いつでも、**Error Management** ビューにアクセスできます。

手順

1. Business Central で、**Menu → Manage → Execution Errors** の順に移動します。
2. 一覧からエラーを選択し、**Details** タブを開きます。これにより、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、エラーを承認して削除します。**Manage Execution Errors** ページの **Acknowledged** フィルターで **Yes** を選択すれば、後からその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. 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。

16.2. EXECUTIONERRORHANDLER

ExecutionErrorHandler は、すべてのプロセスエラー処理の主要メカニズムです。これは RuntimeEngine に結びついているため、新規 RuntimeEngine が作成されるとこれも作成され、RuntimeEngine が破棄されるとこれも破棄されます。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** と **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/タスクに関するエラー処理を通知します。**ExecutionErrorHandler** には、以下の情報について通知されます。

- 特定のノードインスタンスの処理の開始。
- 特定のノードインスタンスの処理の完了。
- 特定のタスクインスタンスの処理の開始。
- 特定のタスクインスタンスの処理の完了。

この情報は主に、未知のタイプのエラーに使用されます。つまり、プロセスコンテキストに関する情報を提供しないエラーです。たとえば、コミット時にデータベース例外にはプロセス情報がありません。

16.3. 実行エラーの保存

ExecutionErrorStorage はプラグ可能な戦略で、実行エラーに関する情報の永続化を各種の方法で可能にします。ストレージは、ストアのインスタンス作成時 (RuntimeEngine の作成時) にこれを取得するハンドラーが直接使用します。デフォルトのストレージ実装はデータベーステーブルをベースにしており、これはすべてのエラーを保存し、利用可能な全情報を含めるものです。エラーによっては詳細を含まないものもあります。これは、エラーのタイプや特定情報を抽出可能かどうかによって異なるためです。

16.4. エラータイプとフィルター

エラー処理ではどのような種類のエラーも捕まえて処理しようとするため、エラーをカテゴリー分けする方法が必要になります。こうすることで、エラーから適切に情報を抽出し、プラグ可能とすることができます。これは、ユーザーによっては、デフォルトとは違う方法で特定タイプのエラーの出力と処理方法を必要とするためです。

エラーのカテゴリー分けとフィルターリングは、**ExecutionErrorFilters** をベースにしています。このインターフェイスは **ExecutionError** のインスタンス構築を担っており、これは後で **ExecutionErrorStorage** 戦略で保存されます。これには以下のメソッドがあります。

- **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**: 実際のフィルターリング、処理などが発生する場所です。
- **getPriority**: フィルター呼び出し時に使用される優先度を示します。

フィルターは、一度に処理するエラーは 1 つで、優先順位の仕組みを使用して、複数のフィルターで、別のビューで同じエラーが返されないようにします。優先順位を使用すると、より特化したフィルターで、エラーが受け入れられるかどうか、または別のフィルターで処理可能かが判断されます。

ExecutionErrorFilter は **ServiceLoader** メカニズムを使用することで提供され、これによりエラー処理の機能が容易に拡張できます。

Red Hat Process Automation Manager には以下の **ExecutionErrorFilters** が同梱されています。

表16.1 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。上記のテーブルでは、以下の順序でフィルターが実行されます。

1. タスク
2. Process
3. Job
4. DB

16.5. 実行エラーの自動承認

実行エラーが発生すると、デフォルトでは承認されず、承認されるには手動での作業が必要になります。承認が行われてないと、実行エラーは注意が必要な情報としてみなされます。ボリュームが大きい場合は、手動作業には時間がかかるため、状況によっては適当ではありません。

自動承認によりこの問題が解消されます。これは **jbpm-executor** を使用したスケジュールジョブをベースとするため、以下の 3 つのタイプのジョブが利用できます。

org.jbpm.executor.commands.error.JobAutoAckErrorCommand

1 回失敗したものの、別の実行でキャンセル、完了、もしくは再スケジュールされたジョブを探します。このジョブは **Job** タイプの実行エラーのみを承認します。

org.jbpm.executor.commands.error.TaskAutoAckErrorCommand

1 回失敗したものの、いずれかの終了状態 (completed、failed、exited、obsolete) にあるタスクのユーザータスク実行エラーを自動承認します。このジョブは、**Task** タイプの実行エラーのみを承認します。

org.jbpm.executor.commands.error.ProcessAutoAckErrorCommand

エラーがアタッチされたプロセスインスタンスを自動承認します。プロセスインスタンスが既に終了してる (completed または aborted) エラー、もしくはエラーの発生元であるタスクがすでに終了しているエラーを承認します。これは **init_activity_id** 値をベースにしています。このジョブは、これらの条件に合致する実行エラーのタイプすべてを承認します。

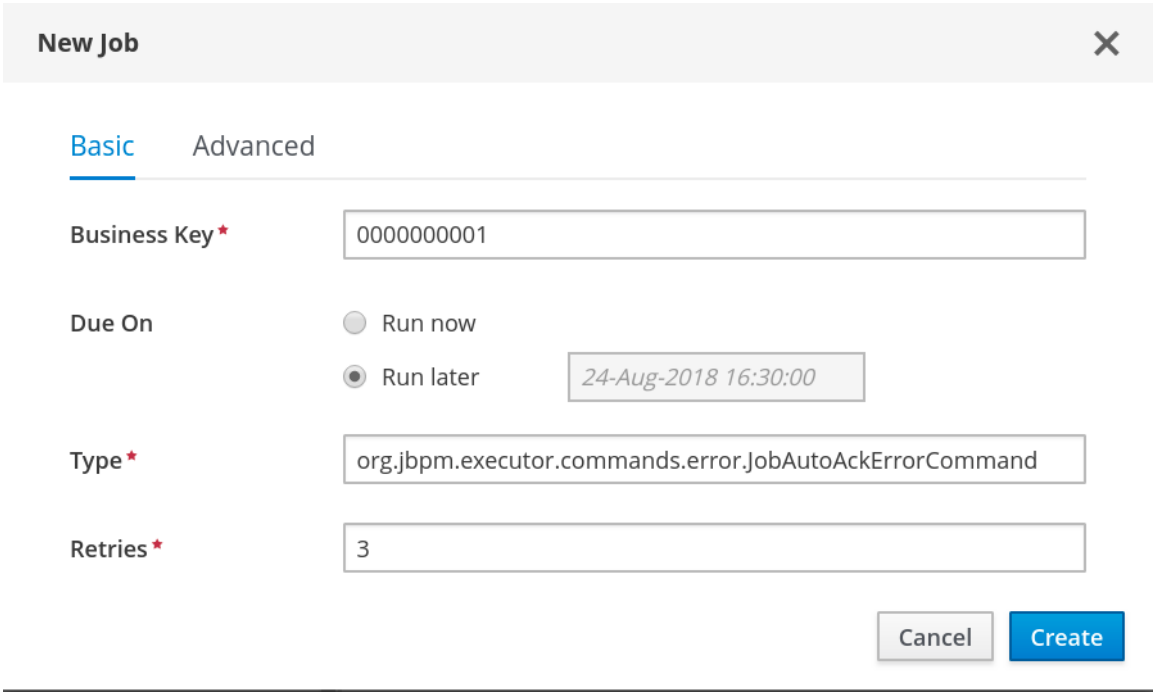
ジョブは Process Server で登録できます。Business Central では、以下のようにしてエラーに対する自動承認ジョブを設定できます。

前提条件

- プロセス実行中に 1 つ以上のタイプの実行エラーが発生したが、さらなる注意を必要としていない。

手順

1. Business Central で、**Menu → Manage → Jobs** の順にクリックします。
2. 画面右上の **New Job** をクリックします。
3. **Business Key** フィールドにプロセス相関キーを入力します。
4. **Type** フィールドに上記のリストから自動承認ジョブタイプを追加します。
5. **Due On** でジョブの完了時間を選択します。
   a. 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。
   b. 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. **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。

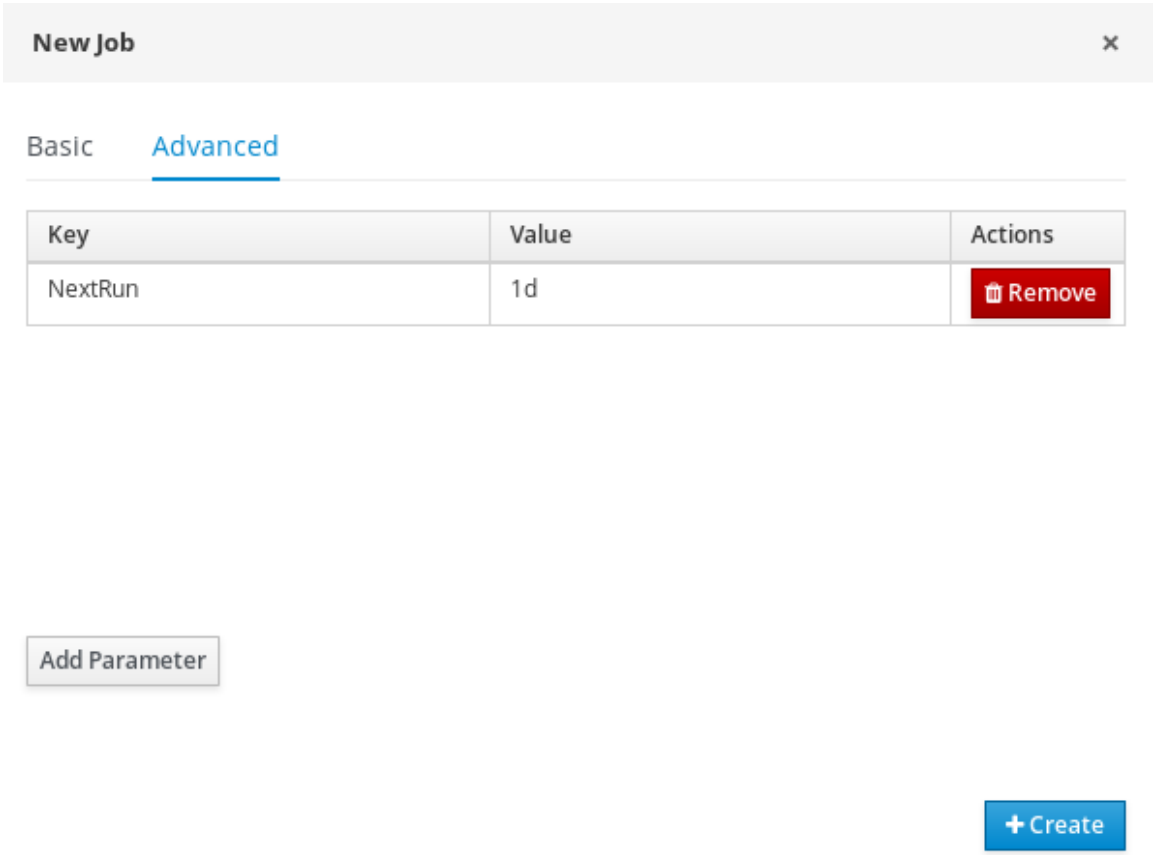
以下のステップは任意となり、自動承認ジョブを 1 回のみ (**SingleRun**) または特定の間隔 (**NextRun**) で実行するか、承認するジョブの検索にエンティティーマネジャーファクトリーのカスタム名を使用 (**EmfName**) して実行するように設定できます。

1. **Advanced** タブをクリックします。
2. **Add Parameter** ボタンをクリックします。
3. ジョブに適用する設定パラメーターを入力します。

a. **SingleRun**: **true** または **false**

b. **NextRun**: 2h、5d、1m などの時間表示。

c. **EmfName**: カスタムのエンティティーマネージャーファクトリーの名前。



16.6. エラー一覧のクリーンアップ

ExecutionErrorInfo エラーリストテーブルは、クリーンアップして冗長情報を削除できます。プロセスのライフサイクルによっては、エラーがリストにしばらく残る場合があり、そのリストをクリーンアップするための直接的な API はありません。代わりに、**ExecutionErrorCleanupCommand** コマンドをスケジュールして、エラーを定期的にクリーンアップできます。

クリーンアップコマンドには、次のパラメーターを設定できます。このコマンドは、すでに完了または中断済みのプロセスインスタンスの実行エラーしか削除できません。

- **DateFormat**
  - 日付関連パラメーター用の日付形式 - 指定しない場合は、**yyyy-MM-dd** が使用されます (**SimpleDateFormat** クラスのパターン)。
- **EmfName**
  - クエリーに使用するエンティティーマネジャーファクトリーの名前 (有効な永続的ユニット名)。
- **SingleRun**
  - 実行が 1 回のみかどうかを指定します (**true**|**false**)。
- **NextRun**

- 次回の実行時間を指定します (有効な時間表記。例: 1d、5h など)。

- **OlderThan**
  - 削除するエラーを指定します。指定した日付より古いものが削除されます。
- **OlderThanPeriod**
  - 指定した時間表記よりも古いエラーを削除することを指定します (有効な時間表記。例: 1d、5h など)
- **ForProcess**
  - 指定したプロセス定義のみのエラーを削除します。
- **ForProcessInstance**
  - 特定のプロセスインスタンスに対してのみ削除されるエラーを示します。
- **ForDeployment**
  - 指定したデプロイメント ID から削除されるエラーを示します。

第17章 RED HAT PROCESS AUTOMATION MANAGER の PROMETHEUS メトリクスの監視

Prometheus は、オープンソースのシステム監視ツールキットで、Red Hat Process Automation Manager と連携して、ビジネスルール、プロセス、Decision Model and Notation (DMN) モデル、その他の Red Hat Process Automation Manager アセットの実行に関するメトリクスを収集して保存できます。Process Server への REST API 呼び出しや、Prometheus 表現ブラウザー、Grafana などのデータグラフツールを使用して 、保存したメトリクスにアクセスできます。

オンプレミスの Process Server、Spring Boot の Process Server、Red Hat OpenShift Container Platform の Process Server デプロイメントに、Prometheus メトリクス監視を設定できます。

Process Server が Prometheus を使用して公開するメトリクスで、利用可能なものの一覧については、**Red Hat カスタマーポータル から Red Hat Process Automation Manager 7.5.1 Source Distribution をダウンロードし、～/rhpam-7.5.1-sources/src/droolsjbpm-integration-$VERSION/kie-server-parent/kie-server-services/kie-server-services-prometheus/src/main/java/org/kie/server/services/prometheus に移動してください。**

重要

Prometheus に対する Red Hat の サポートは、Red Hat 製品ドキュメントに記載の設定および設定の推奨事項に限定されます。

17.1. PROCESS SERVER のモニターリングを行う PROMETHEUS メトリックスの設定

Process Server インスタンスが Prometheus を使用して、Red Hat Process Automation Manager で、ビジネスアセットアクティビティーに関連するメトリクスを収集して保存するように設定できます。Process Server が Prometheus を使用して公開するメトリクスで、利用可能なものの一覧については、**Red Hat カスタマーポータル から Red Hat Process Automation Manager 7.5.1 Source Distribution をダウンロードし、～/rhpam-7.5.1-sources/src/droolsjbpm-integration-$VERSION/kie-server-parent/kie-server-services/kie-server-services-prometheus/src/main/java/org/kie/server/services/prometheus に移動してください。**

前提条件

- Process Server がインストールされている。
- **kie-server** ユーザーロールで Process Server にアクセスできる。
- Prometheus がインストールされている。Prometheus のダウンロードおよび使用に関する情報は、Prometheus ドキュメントページ を参照してください。

手順

1. Process Server インスタンスで、**org.kie.prometheus.server.ext.disabled** システムプロパティーを **false** に設定して、Prometheus 拡張機能を有効にします。このプロパティーは、Process Server の起動時、または Red Hat Process Automation Manager ディストリビューションの **standalone.xml** または **standalone-full.xml** ファイルで定義できます。
2. Spring Boot で Red Hat Process Automation Manager を実行している場合には、Maven プロジェクトの **pom.xml** ファイルに次の依存関係を追加し、**application.properties** システムプロパティーで必要なキーを設定します。

Prometheus の Spring Boot pom.xml 依存関係

Prometheus の Spring Boot pom.xml 依存関係

```
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-services-prometheus</artifactId>
  <version>${rhpam.version}</version>
</dependency>

<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-rest-prometheus</artifactId>
  <version>${rhpam.version}</version>
</dependency>
```

Red Hat Process Automation Manager および Prometheus の Spring Boot application.properties キー

```
kieserver.jbpm.enabled=true
kieserver.drools.enabled=true
kieserver.dmn.enabled=true
kieserver.prometheus.enabled=true
```

3. Prometheus ディストリビューションの **prometheus.yaml** ファイルで、**scrape_configs** セクションに、以下の設定を追加して、Prometheus が Process Server からメトリクスを収集 (scrape) するように設定します。

prometheus.yaml ファイルの Scrape 設定

```
scrape_configs:
  - job_name: 'kie-server'
    metrics_path: /SERVER_PATH/services/rest/metrics
    basic_auth:
      username: USER_NAME
      password: PASSWORD
    static_configs:
      - targets: ["HOST:PORT"]
```

Spring Boot の prometheus.yaml ファイルでの Scrape 設定 (該当する場合)

```
scrape_configs:
  - job_name: 'kie'
    metrics_path: /rest/metrics
    static_configs:
      - targets: ["HOST:PORT"]
```

Process Server の場所と設定に合わせて、値を置き換えます。

4. Process Server インスタンスを起動します。

Red Hat JBoss EAP での Red Hat Process Automation Manager の起動コマンド例

```
$ cd ~/EAP_HOME/bin
$ ./standalone.sh --c standalone-full.xml
```

設定済みの Process Server インスタンスを起動すると、Prometheus はメトリクスの収集を開始し、Process Server はメトリクスを REST API エンドポイント **http://HOST:PORT/SERVER/services/rest/metrics** (または Spring Boot では **http://HOST:PORT/rest/metrics**) に公開します。

5. REST クライアントまたは curl ユーティリティーで、以下のコンポーネントを含む REST API 要求を送信し、Process Server がメトリクスを公開していることを確認します。
REST クライアントの場合:

- **Authentication**: **kie-server** ロールを持つ Process Server ユーザーのユーザー名とパスワードを入力します。
- **HTTP Headers**: 以下のヘッダーを設定します。
  - **Accept**: **application/json**
- **HTTP method**: **GET** に設定します。
- **URL**: Process Server REST API ベース URL とメトリクスエンドポイントを入力します。たとえば、**http://localhost:8080/kie-server/services/rest/metrics** (または Spring Boot では **http://localhost:8080/rest/metrics**) となります。

curl ユーティリティーの場合:

- **-u**: **kie-server** ロールを持つ Process Server ユーザーのユーザー名とパスワードを入力します。
- **-H**: 以下のヘッダーを設定します。
  - **accept**: **application/json**
- **-X**: **GET** に設定します。
- **URL**: Process Server REST API ベース URL とメトリクスエンドポイントを入力します。たとえば、**http://localhost:8080/kie-server/services/rest/metrics** (または Spring Boot では **http://localhost:8080/rest/metrics**) となります。

```
curl -u 'baAdmin:password@1' -H "accept: application/json" -X GET
"http://localhost:8080/kie-server/services/rest/metrics"
```

サーバーの応答例

```
# HELP kie_server_container_started_total Kie Server Started Containers
# TYPE kie_server_container_started_total counter
kie_server_container_started_total{container_id="task-assignment-kjar-1.0",} 1.0
# HELP solvers_running Number of solvers currently running
# TYPE solvers_running gauge
solvers_running 0.0
# HELP dmn_evaluate_decision_nanosecond DMN Evaluation Time
# TYPE dmn_evaluate_decision_nanosecond histogram
# HELP solver_duration_seconds Time in seconds it took solver to solve the constraint
problem
# TYPE solver_duration_seconds summary
solver_duration_seconds_count{solver_id="100tasks-5employees.xml",} 1.0
solver_duration_seconds_sum{solver_id="100tasks-5employees.xml",} 179.828255925
solver_duration_seconds_count{solver_id="24tasks-8employees.xml",} 1.0
```

```
solver_duration_seconds_sum{solver_id="24tasks-8employees.xml",} 179.995759653
# HELP drl_match_fired_nanosecond Drools Firing Time
# TYPE drl_match_fired_nanosecond histogram
# HELP dmn_evaluate_failed_count DMN Evaluation Failed
# TYPE dmn_evaluate_failed_count counter
# HELP kie_server_start_time Kie Server Start Time
# TYPE kie_server_start_time gauge
kie_server_start_time{name="myapp-kieserver",server_id="myapp-
kieserver",location="http://myapp-kieserver-demo-
monitoring.127.0.0.1.nip.io:80/services/rest/server",version="7.4.0.redhat-20190428",}
1.557221271502E12
# HELP kie_server_container_running_total Kie Server Running Containers
# TYPE kie_server_container_running_total gauge
kie_server_container_running_total{container_id="task-assignment-kjar-1.0",} 1.0
# HELP solver_score_calculation_speed Number of moves per second for a particular solver
solving the constraint problem
# TYPE solver_score_calculation_speed summary
solver_score_calculation_speed_count{solver_id="100tasks-5employees.xml",} 1.0
solver_score_calculation_speed_sum{solver_id="100tasks-5employees.xml",} 6997.0
solver_score_calculation_speed_count{solver_id="24tasks-8employees.xml",} 1.0
solver_score_calculation_speed_sum{solver_id="24tasks-8employees.xml",} 19772.0
# HELP kie_server_case_started_total Kie Server Started Cases
# TYPE kie_server_case_started_total counter
kie_server_case_started_total{case_definition_id="itorders.orderhardware",} 1.0
# HELP kie_server_case_running_total Kie Server Running Cases
# TYPE kie_server_case_running_total gauge
kie_server_case_running_total{case_definition_id="itorders.orderhardware",} 2.0
# HELP kie_server_data_set_registered_total Kie Server Data Set Registered
# TYPE kie_server_data_set_registered_total gauge
kie_server_data_set_registered_total{name="jbpmProcessInstanceLogs::CUSTOM",uuid="jbp
mProcessInstanceLogs",} 1.0
kie_server_data_set_registered_total{name="jbpmRequestList::CUSTOM",uuid="jbpmRequest
List",} 1.0
kie_server_data_set_registered_total{name="tasksMonitoring::CUSTOM",uuid="tasksMonitorin
g",} 1.0
kie_server_data_set_registered_total{name="jbpmHumanTasks::CUSTOM",uuid="jbpmHuman
Tasks",} 1.0
kie_server_data_set_registered_total{name="jbpmHumanTasksWithUser::FILTERED_PO_TA
SK",uuid="jbpmHumanTasksWithUser",} 1.0
kie_server_data_set_registered_total{name="jbpmHumanTasksWithVariables::CUSTOM",uuid
="jbpmHumanTasksWithVariables",} 1.0
kie_server_data_set_registered_total{name="jbpmProcessInstancesWithVariables::CUSTOM",
uuid="jbpmProcessInstancesWithVariables",} 1.0
kie_server_data_set_registered_total{name="jbpmProcessInstances::CUSTOM",uuid="jbpmPr
ocessInstances",} 1.0
kie_server_data_set_registered_total{name="jbpmExecutionErrorList::CUSTOM",uuid="jbpmEx
ecutionErrorList",} 1.0
kie_server_data_set_registered_total{name="processesMonitoring::CUSTOM",uuid="processe
sMonitoring",} 1.0
kie_server_data_set_registered_total{name="jbpmHumanTasksWithAdmin::FILTERED_BA_TA
SK",uuid="jbpmHumanTasksWithAdmin",} 1.0
# HELP kie_server_execution_error_total Kie Server Execution Errors
# TYPE kie_server_execution_error_total counter
# HELP kie_server_task_completed_total Kie Server Completed Tasks
# TYPE kie_server_task_completed_total counter
# HELP kie_server_container_running_total Kie Server Running Containers
```

```
# TYPE kie_server_container_running_total gauge
kie_server_container_running_total{container_id="itorders_1.0.0-SNAPSHOT",} 1.0
# HELP kie_server_job_cancelled_total Kie Server Cancelled Jobs
# TYPE kie_server_job_cancelled_total counter
# HELP kie_server_process_instance_started_total Kie Server Started Process Instances
# TYPE kie_server_process_instance_started_total counter
kie_server_process_instance_started_total{container_id="itorders_1.0.0-
SNAPSHOT",process_id="itorders.orderhardware",} 1.0
# HELP solver_duration_seconds Time in seconds it took solver to solve the constraint
problem
# TYPE solver_duration_seconds summary
# HELP kie_server_task_skipped_total Kie Server Skipped Tasks
# TYPE kie_server_task_skipped_total counter
# HELP kie_server_data_set_execution_time_seconds Kie Server Data Set Execution Time
# TYPE kie_server_data_set_execution_time_seconds summary
kie_server_data_set_execution_time_seconds_count{uuid="jbpmProcessInstances",} 8.0
kie_server_data_set_execution_time_seconds_sum{uuid="jbpmProcessInstances",}
0.05600000000000001
# HELP kie_server_job_scheduled_total Kie Server Started Jobs
# TYPE kie_server_job_scheduled_total counter
# HELP kie_server_data_set_execution_total Kie Server Data Set Execution
# TYPE kie_server_data_set_execution_total counter
kie_server_data_set_execution_total{uuid="jbpmProcessInstances",} 8.0
# HELP kie_server_process_instance_completed_total Kie Server Completed Process
Instances
# TYPE kie_server_process_instance_completed_total counter
# HELP kie_server_job_running_total Kie Server Running Jobs
# TYPE kie_server_job_running_total gauge
# HELP kie_server_task_failed_total Kie Server Failed Tasks
# TYPE kie_server_task_failed_total counter
# HELP kie_server_task_exited_total Kie Server Exited Tasks
# TYPE kie_server_task_exited_total counter
# HELP dmn_evaluate_decision_nanosecond DMN Evaluation Time
# TYPE dmn_evaluate_decision_nanosecond histogram
# HELP kie_server_data_set_lookups_total Kie Server Data Set Running Lookups
# TYPE kie_server_data_set_lookups_total gauge
kie_server_data_set_lookups_total{uuid="jbpmProcessInstances",} 0.0
# HELP kie_server_process_instance_duration_seconds Kie Server Process Instances
Duration
# TYPE kie_server_process_instance_duration_seconds summary
# HELP kie_server_case_duration_seconds Kie Server Case Duration
# TYPE kie_server_case_duration_seconds summary
# HELP dmn_evaluate_failed_count DMN Evaluation Failed
# TYPE dmn_evaluate_failed_count counter
# HELP kie_server_task_added_total Kie Server Added Tasks
# TYPE kie_server_task_added_total counter
kie_server_task_added_total{deployment_id="itorders_1.0.0-
SNAPSHOT",process_id="itorders.orderhardware",task_name="Prepare hardware spec",}
1.0
# HELP drl_match_fired_nanosecond Drools Firing Time
# TYPE drl_match_fired_nanosecond histogram
# HELP kie_server_container_started_total Kie Server Started Containers
# TYPE kie_server_container_started_total counter
kie_server_container_started_total{container_id="itorders_1.0.0-SNAPSHOT",} 1.0
# HELP kie_server_process_instance_sla_violated_total Kie Server Process Instances SLA
Violated
```

```
# TYPE kie_server_process_instance_sla_violated_total counter
# HELP kie_server_task_duration_seconds Kie Server Task Duration
# TYPE kie_server_task_duration_seconds summary
# HELP kie_server_job_executed_total Kie Server Executed Jobs
# TYPE kie_server_job_executed_total counter
# HELP kie_server_deployments_active_total Kie Server Active Deployments
# TYPE kie_server_deployments_active_total gauge
kie_server_deployments_active_total{deployment_id="itorders_1.0.0-SNAPSHOT",} 1.0
# HELP kie_server_process_instance_running_total Kie Server Running Process Instances
# TYPE kie_server_process_instance_running_total gauge
kie_server_process_instance_running_total{container_id="itorders_1.0.0-
SNAPSHOT",process_id="itorders.orderhardware",} 2.0
# HELP solvers_running Number of solvers currently running
# TYPE solvers_running gauge
solvers_running 0.0
# HELP kie_server_work_item_duration_seconds Kie Server Work Items Duration
# TYPE kie_server_work_item_duration_seconds summary
# HELP kie_server_job_duration_seconds Kie Server Job Duration
# TYPE kie_server_job_duration_seconds summary
# HELP solver_score_calculation_speed Number of moves per second for a particular solver
solving the constraint problem
# TYPE solver_score_calculation_speed summary
# HELP kie_server_start_time Kie Server Start Time
# TYPE kie_server_start_time gauge
kie_server_start_time{name="sample-server",server_id="sample-
server",location="http://localhost:8080/kie-server/services/rest/server",version="7.22.0-
SNAPSHOT",} 1.557285486469E12
```

Process Server でメトリクスが利用できない場合には、このセクションで説明されている Process Server および Prometheus の設定を確認します。

また、**http://HOST:PORT/graph** の Prometheus の expression browser で収集したメトリクスと対話したり、Prometheus データソースを Grafana などのデータグラフ作成ツールと統合したりすることもできます。

図17.1 Prometheus の expression browser と Process Server メトリクス

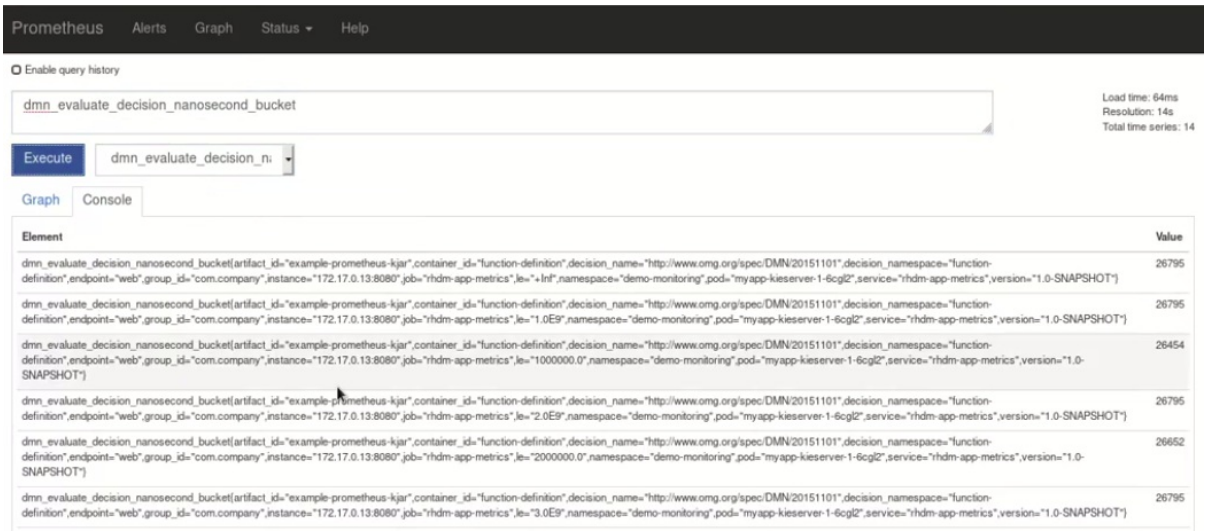


図17.2 Prometheus の expression browser と Process Server ターゲット

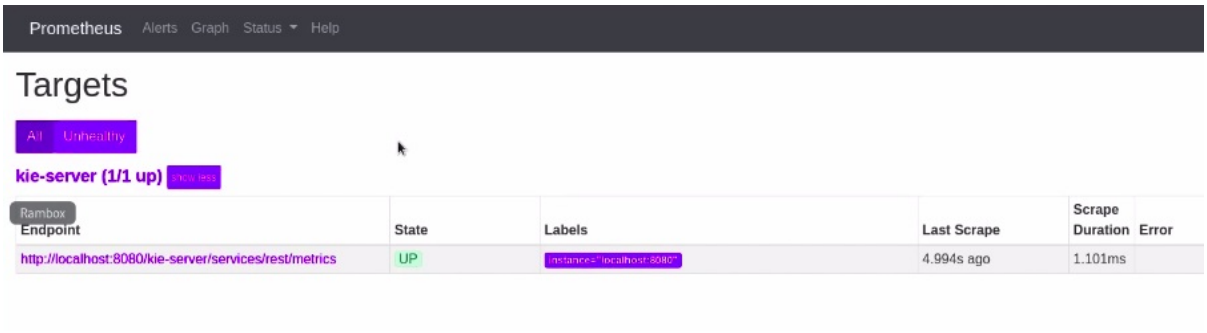


図17.3 Grafana ダッシュボードと DMN モデルの Process Server メトリクス

図17.4 Grafana ダッシュボードとソルバーの Process Server メトリクス

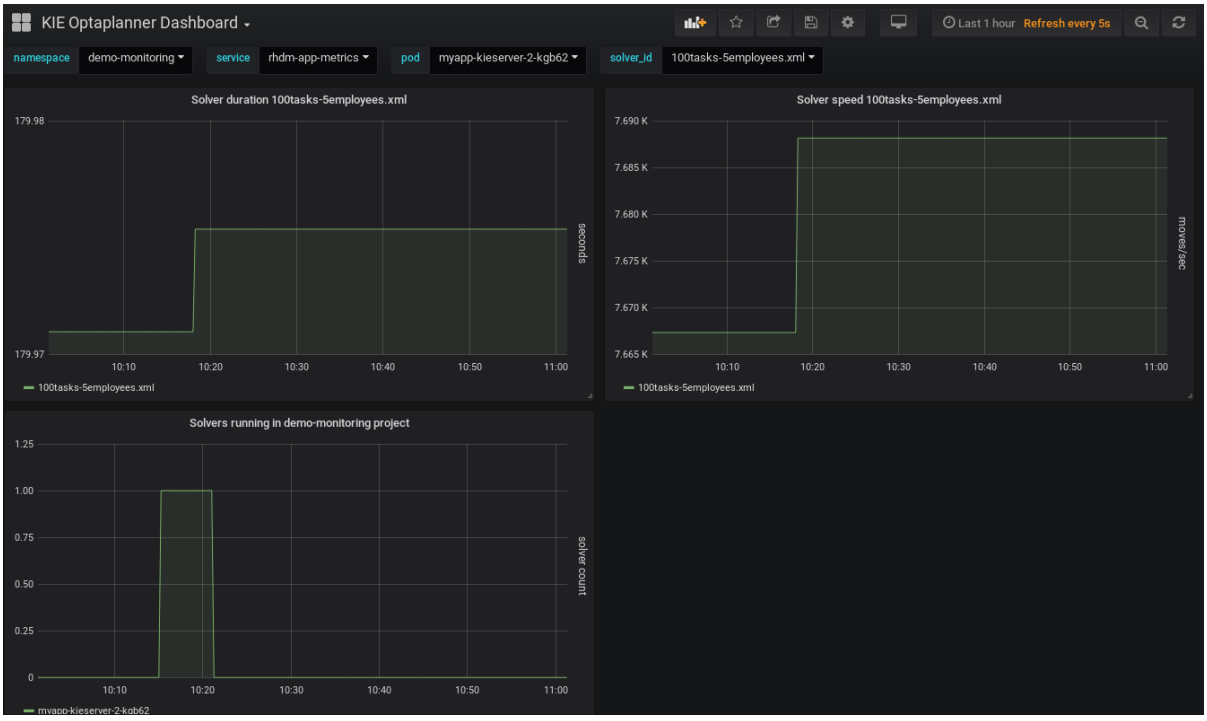
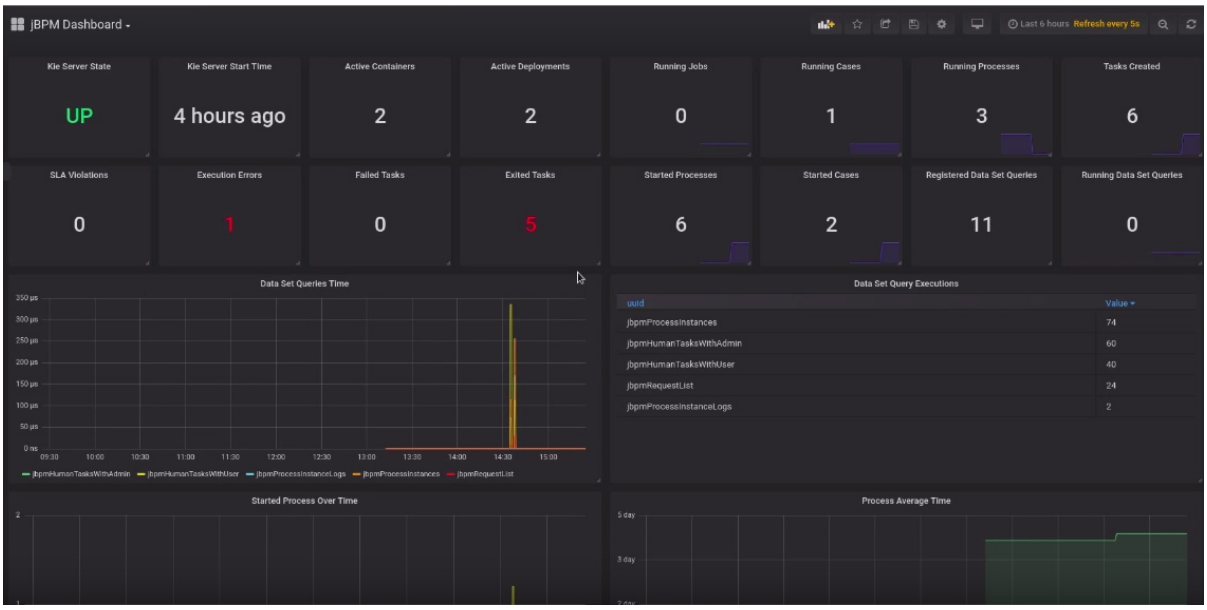


図17.5 Grafana ダッシュボードとプロセス、ケースおよびタスクの Process Server メトリクス



関連情報

- Getting Started with Prometheus
- Grafana Support for Prometheus
- Using Prometheus in Grafana

17.2. RED HAT OPENSHIFT CONTAINER PLATFORM の PROCESS SERVER の PROMETHEUS メトリクスモニターリングの設定

Prometheus を使用して Red Hat Process Automation Manger で、ビジネスアセットアクティビティー関連のメトリクスを収集して保存するように、Red Hat OpenShift Container Platform 上の Process Server デプロイメントを設定できます。Process Server が Prometheus を使用して公開するメトリクスで、利用可能なものの一覧については、**Red Hat カスタマーポータル から Red Hat Process Automation Manager 7. 5.1 Source Distribution をダウンロードし、～/rhpam-7.5.1-sources/src/droolsjbpm-integration-$VERSION/kie-server-parent/kie-server-services/kie-server-services-prometheus/src/main/java/org/kie/server/services/prometheus** に移動してください。

前提条件

- Process Server が、Red Hat OpenShift Container Platform にインストールおよびデプロイメントされている。OpenShift 上の Process Server の詳細については、Red Hat Process Automation Manager 7.5 の製品ドキュメント で関連する OpenShift デプロイメントオプションを参照してください。
- **kie-server** ユーザーロールで Process Server にアクセスできる。
- Prometheus Operator がインストールされている。Prometheus Operator のダウンロードおよび使用についての詳細は、GitHub の Prometheus Operator プロジェクトを参照してください。

手順

1. OpenShift 上の Process Server デプロイメントの **DeploymentConfig** オブジェクトで、**PROMETHEUS_SERVER_EXT_DISABLED** 環境変数を **false** に設定して、Prometheus 拡張機能を有効にします。この変数は、OpenShift Web コンソールを使用するか、コマンド端末で **oc** コマンドを使用してこの変数を設定してください。

```
oc set env dc/<dc_name> PROMETHEUS_SERVER_EXT_DISABLED=false -n
<namespace>
```

OpenShift に Process Server をまだデプロイしていない場合には、OpenShift デプロイメントに使用予定の OpenShift テンプレートで（例： **rhpam75-prod-immutable-kieserver.yaml**）、**PROMETHEUS_SERVER_EXT_DISABLED** テンプレートパラメーターを **false** に設定して、Prometheus 拡張機能を有効にします。

OpenShift Operator を使用して Process Server を OpenShift にデプロイする場合には、Process Server の設定で、**PROMETHEUS_SERVER_EXT_DISABLED** 環境変数を **false** に設定して、Prometheus 拡張機能を有効にします。

```
apiVersion: app.kiegroup.org/v1
kind: KieApp
metadata:
  name: enable-prometheus
spec:
  environment: rhpam-trial
  objects:
    servers:
    - env:
      - name: PROMETHEUS_SERVER_EXT_DISABLED
        value: "false"
```

2. **service-metrics.yaml** ファイルを作成して、Process Server から Prometheus にメトリクスを公開するサービスを追加します。

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    description: RHPAM Prometheus metrics exposed
  labels:
    app: myapp-kieserver
    application: myapp-kieserver
    template: myapp-kieserver
    metrics: rhpam
  name: rhpam-app-metrics
spec:
  ports:
    - name: web
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    deploymentConfig: myapp-kieserver
  sessionAffinity: None
  type: ClusterIP
```

3. コマンドターミナルで、**oc** コマンドを使用して、**service-metrics.yaml** ファイルを OpenShift デプロイメントに適用します。

```
oc apply -f service-metrics.yaml
```

4. **metrics-secret** など、OpenShift シークレットを作成して、Process Server の Prometheus メトリクスにアクセスします。シークレットには username と password 要素と Process Server ユーザー資格情報が含める必要があります。OpenShift シークレットの詳細は、OpenShift **開発者ガイド**の シークレット の章を参照してください。

5. **ServiceMonitor** オブジェクトを定義する **service-monitor.yaml** ファイルを作成します。サービスモニターにより Prometheus を Process Server メトリクスサービスに接続できます。

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: rhpam-service-monitor
  labels:
    team: frontend
spec:
  selector:
    matchLabels:
      metrics: rhpam
  endpoints:
    - port: web
      path: /services/rest/metrics
      basicAuth:
        password:
          name: metrics-secret
          key: password
        username:
          name: metrics-secret
          key: username
```

6. コマンド端末で、**oc** コマンドを使用して、**service-monitor.yaml** ファイルを OpenShift デプロイメントに適用します。

```
oc apply -f service-monitor.yaml
```

上記の設定を完了すると、Prometheus はメトリクスの収集を開始し、Process Server は REST API エンドポイント **http://HOST:PORT/kie-server/services/rest/metrics** にメトリクスを公開します。

http://HOST:PORT/graph の Prometheus expression browser で収集したメトリクスと対話したり、Prometheus データソースを Grafana などのデータグラフ作成ツールと統合したりすることができます。

Prometheus expression browser の場所のホストとポートである **http://HOST:PORT/graph** は、Prometheus Operator をインストールしたときに Prometheus Web コンソールを公開したルートで定義されています。OpenShift ルートの詳細は、OpenShift **アーキテクチャ**ドキュメントの ルート の章を参照してください。

図17.6 Prometheus の expression browser と Process Server メトリクス

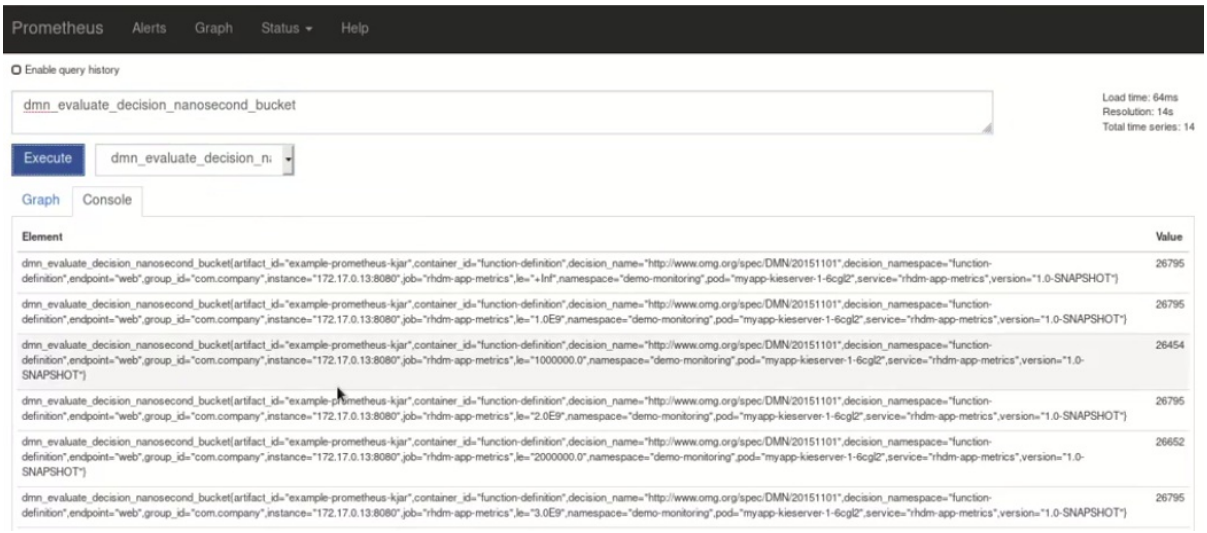


図17.7 Prometheus の expression browser と Process Server ターゲット

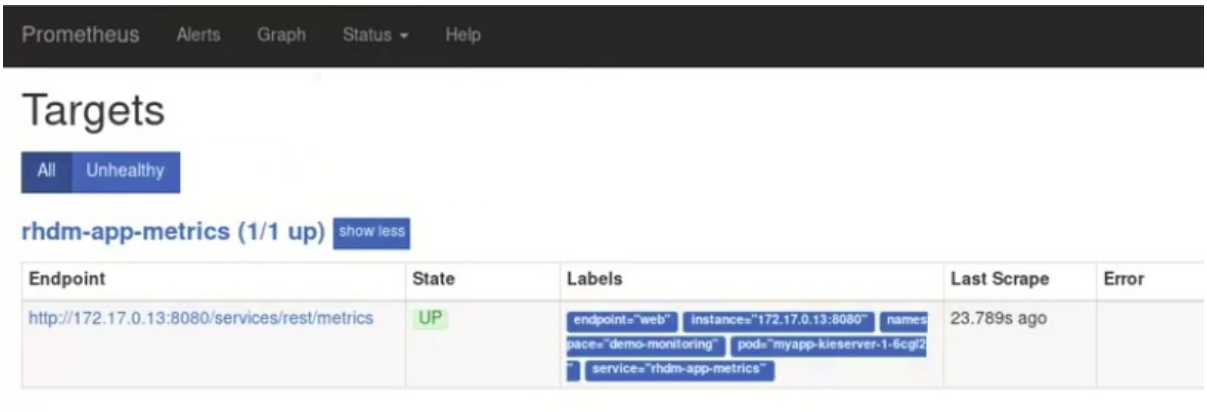


図17.8 Grafana ダッシュボードと DMN モデルの Process Server メトリクス

図17.9 Grafana ダッシュボードとソルバーの Process Server メトリクス

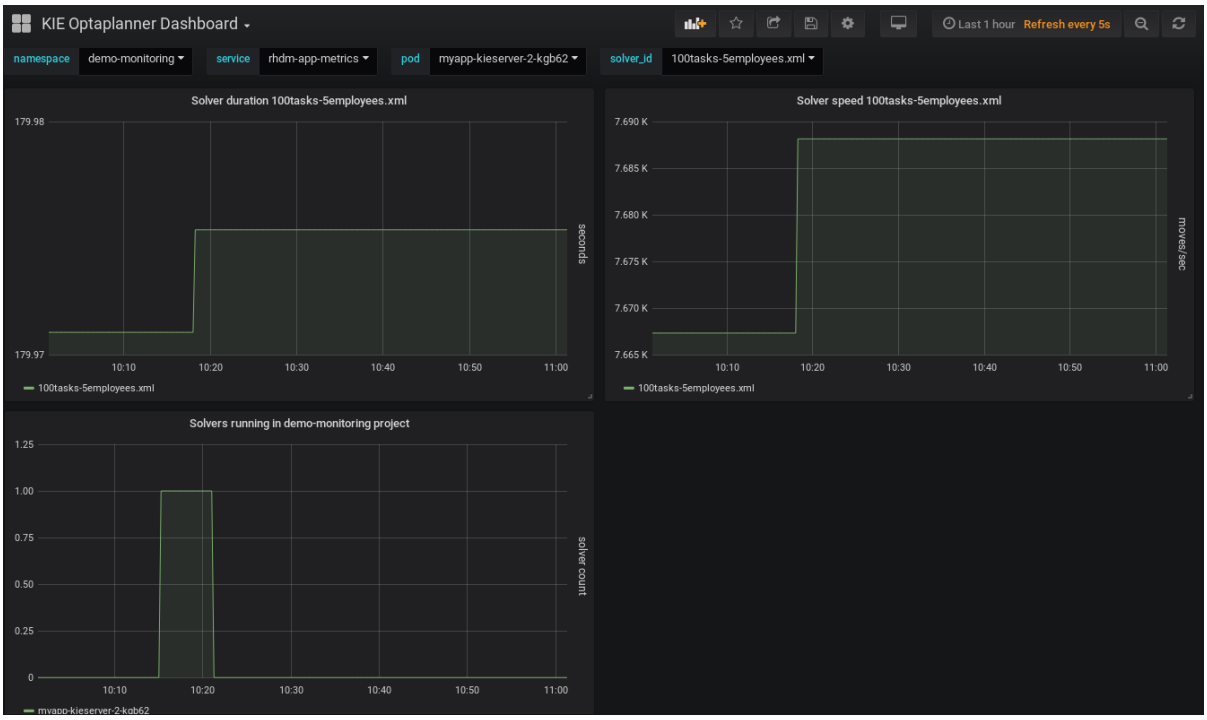
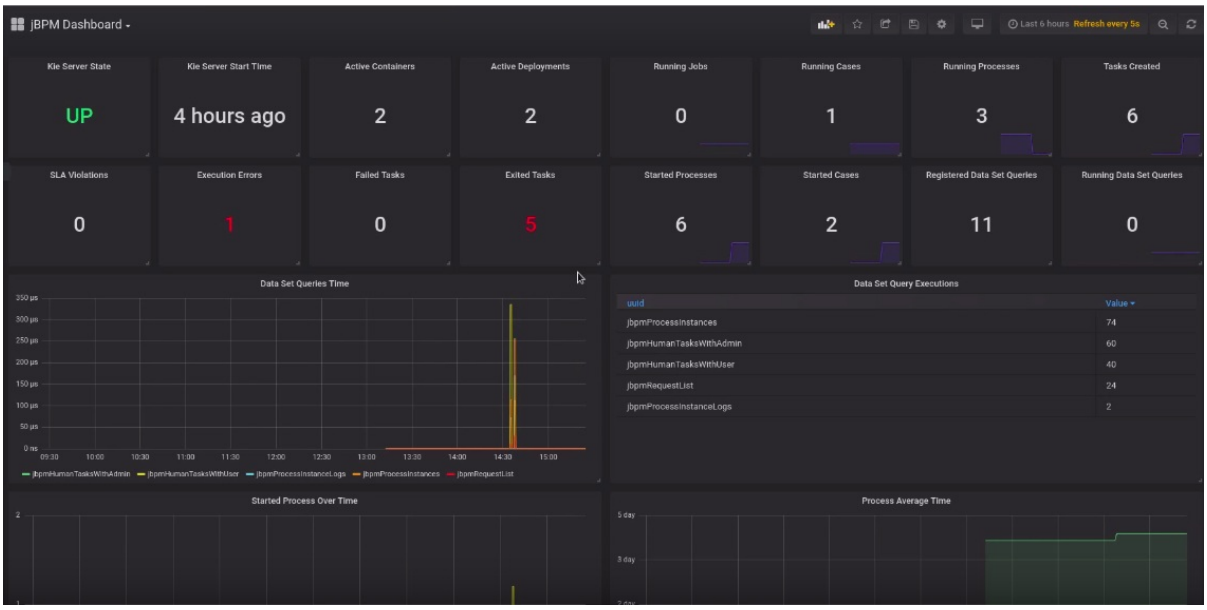


図17.10 Grafana ダッシュボードとプロセス、ケースおよびタスクの Process Server メトリクス



関連情報

- Prometheus Operator
- Getting started with the Prometheus Operator
- Prometheus RBAC
- Grafana Support for Prometheus
- Using Prometheus in Grafana

- Red Hat Process Automation Manager 7.5 の製品ドキュメント の OpenShift デプロイメントオプション

17.3. カスタムメトリクスを使用した PROCESS SERVER の PROMETHEUS メトリクスモニタリングの拡張

Process Server インスタンスが Prometheus メトリクスモニタリングを使用するように設定後に、ビジネス要件に合わせてカスタムのメトリクスを使用するように Process Server の Prometheus 機能を拡張できます。Prometheus は、Process Server が Prometheus に公開するデフォルトのメトリクスと、カスタムメトリクスを収集して、保存します。

たとえば、以下の手順では、Prometheus で収集して保存するように、カスタムの Decision Model and Notation (DMN) メトリックを定義します。

前提条件

- Prometheus メトリクスモニタリングが、Process Server インスタンス用に設定されている。オンプレミスの Process Server と Prometheus の設定に関する情報は、「Process Server のモニターリングを行う Prometheus メトリックスの設定」 を参照してください。Red Hat OpenShift Container Platform の Process Server と Prometheus の設定に関する情報は、「Red Hat OpenShift Container Platform の Process Server の Prometheus メトリクスモニターリングの設定」 を参照してください。

手順

1. 空の Maven プロジェクトを作成して、以下のパッケージタイプと依存関係を、プロジェクトの **pom.xml** ファイルに定義します。

   サンプルプロジェクトの pom.xml ファイルの例

```
<packaging>jar</packaging>

<properties>
  <version.org.kie>7.26.0.Final-redhat-00005</version.org.kie>
</properties>

<dependencies>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-api</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-api</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-services-common</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-services-drools</artifactId>
```

```
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-services-prometheus</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-dmn-api</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-dmn-core</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.jbpm</groupId>
    <artifactId>jbpm-services-api</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.jbpm</groupId>
    <artifactId>jbpm-executor</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.optaplanner</groupId>
    <artifactId>optaplanner-core</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient</artifactId>
    <version>0.5.0</version>
  </dependency>
</dependencies>
```

2. 以下の例にあるように、カスタムの Prometheus メトリクスを定義する、カスタムのリスナークラスの一部として、**org.kie.server.services.prometheus.PrometheusMetricsProvider** インターフェイスを実装します。

カスタムのリスナークラス内の DMNRuntimeEventListener リスナーの実装例

```
package org.kie.server.ext.prometheus;

import io.prometheus.client.Gauge;
import org.kie.dmn.api.core.ast.DecisionNode;
import org.kie.dmn.api.core.event.AfterEvaluateBKMEvent;
import org.kie.dmn.api.core.event.AfterEvaluateContextEntryEvent;
import org.kie.dmn.api.core.event.AfterEvaluateDecisionEvent;
import org.kie.dmn.api.core.event.AfterEvaluateDecisionServiceEvent;
import org.kie.dmn.api.core.event.AfterEvaluateDecisionTableEvent;
import org.kie.dmn.api.core.event.BeforeEvaluateBKMEvent;
import org.kie.dmn.api.core.event.BeforeEvaluateContextEntryEvent;
```

```
import org.kie.dmn.api.core.event.BeforeEvaluateDecisionEvent;
import org.kie.dmn.api.core.event.BeforeEvaluateDecisionServiceEvent;
import org.kie.dmn.api.core.event.BeforeEvaluateDecisionTableEvent;
import org.kie.dmn.api.core.event.DMNRuntimeEventListener;
import org.kie.server.api.model.ReleaseId;
import org.kie.server.services.api.KieContainerInstance;

public class ExampleCustomPrometheusMetricListener implements
DMNRuntimeEventListener {

    private final KieContainerInstance kieContainer;

    private final Gauge randomGauge = Gauge.build()
            .name("random_gauge_nanosecond")
            .help("Random gauge as an example of custom KIE Prometheus metric")
            .labelNames("container_id", "group_id", "artifact_id", "version",
"decision_namespace", "decision_name")
            .register();

    public ExampleCustomPrometheusMetricListener(KieContainerInstance
containerInstance) {
        kieContainer = containerInstance;
    }

    public void beforeEvaluateDecision(BeforeEvaluateDecisionEvent e) {
    }

    public void afterEvaluateDecision(AfterEvaluateDecisionEvent e) {
        DecisionNode decisionNode = e.getDecision();
        ReleaseId releaseId = kieContainer.getResource().getReleaseId();
        randomGauge.labels(kieContainer.getContainerId(), releaseId.getGroupId(),
                releaseId.getArtifactId(), releaseId.getVersion(),
                decisionNode.getModelName(), decisionNode.getModelNamespace())
            .set((int) (Math.random() * 100));
    }

    public void beforeEvaluateBKM(BeforeEvaluateBKMEvent event) {
    }

    public void afterEvaluateBKM(AfterEvaluateBKMEvent event) {
    }

    public void beforeEvaluateContextEntry(BeforeEvaluateContextEntryEvent event) {
    }

    public void afterEvaluateContextEntry(AfterEvaluateContextEntryEvent event) {
    }

    public void beforeEvaluateDecisionTable(BeforeEvaluateDecisionTableEvent event) {
    }

    public void afterEvaluateDecisionTable(AfterEvaluateDecisionTableEvent event) {
    }

    public void beforeEvaluateDecisionService(BeforeEvaluateDecisionServiceEvent event) {
    }
```

```
    public void afterEvaluateDecisionService(AfterEvaluateDecisionServiceEvent event) {
    }
}
```

PrometheusMetricsProvider インターフェイスには、Prometheus メトリクス収集に必要なリスナーが含まれます。このインターフェイスは、プロジェクトの **pom.xml** ファイルで宣言した **kie-server-services-prometheus** 依存関係に組み込まれます。

以下の例では、**ExampleCustomPrometheusMetricListener** クラスは、(**PrometheusMetricsProvider** インターフェイスからの) **DMNRuntimeEventListener** リスナーを実装し、Prometheus で収集、保存するカスタムの DMN メトリクスを定義します。

3. 以下の例にあるように、**PrometheusMetricsProvider** インターフェイスとカスタムのリスナーを関連付ける、カスタムのメトリクスプロバイダーの一部として **PrometheusMetricsProvider** インターフェイスを実装します。

カスタムメトリクスプロバイダークラスの PrometheusMetricsProvider インターフェイスの実装例

```
package org.kie.server.ext.prometheus;

import org.jbpm.executor.AsynchronousJobListener;
import org.jbpm.services.api.DeploymentEventListener;
import org.kie.api.event.rule.AgendaEventListener;
import org.kie.api.event.rule.DefaultAgendaEventListener;
import org.kie.dmn.api.core.event.DMNRuntimeEventListener;
import org.kie.server.services.api.KieContainerInstance;
import org.kie.server.services.prometheus.PrometheusMetricsProvider;
import org.optaplanner.core.impl.phase.event.PhaseLifecycleListener;
import org.optaplanner.core.impl.phase.event.PhaseLifecycleListenerAdapter;

public class MyPrometheusMetricsProvider implements PrometheusMetricsProvider {

    public DMNRuntimeEventListener createDMNRuntimeEventListener(KieContainerInstance
kContainer) {
        return new ExampleCustomPrometheusMetricListener(kContainer);
    }

    public AgendaEventListener createAgendaEventListener(String kieSessionId,
KieContainerInstance kContainer) {
        return new DefaultAgendaEventListener();
    }

    public PhaseLifecycleListener createPhaseLifecycleListener(String solverId) {
        return new PhaseLifecycleListenerAdapter() {
        };
    }

    public AsynchronousJobListener createAsynchronousJobListener() {
        return null;
    }

    public DeploymentEventListener createDeploymentEventListener() {
```

```
        return null;
    }
}
```

以下の例では、**MyPrometheusMetricsProvider** クラスは **PrometheusMetricsProvider** インターフェイスを実装し、このクラスには、カスタムの **ExampleCustomPrometheusMetricListener** リスナークラスが含まれます。

4. 新規メトリクスプロバイダーを Process Server で検出できるようにするには、Maven プロジェクトに **META-INF/services/org.kie.server.services.prometheus.PrometheusMetricsProvider** ファイルを作成し、このファイルに **PrometheusMetricsProvider** 実装クラスの完全修飾クラス名を追加します。以下の例では、このファイルに **org.kie.server.ext.prometheus.MyPrometheusMetricsProvider** の 1 行が含まれています。

5. プロジェクトを構築して、作成された JAR ファイルをプロジェクトの **~/kie-server.war/WEB-INF/lib** ディレクトリーにコピーします。たとえば、Red Hat JBoss EAP ではこのディレクトリーへのパスは **EAP_HOME/standalone/deployments/kie-server.war/WEB-INF/lib** です。

6. Process Server を起動して、実行中の Process Server に構築したプロジェクトをデプロイします。プロジェクトは、Business Central インターフェースまたは Process Server REST API（**http://SERVER:PORT/kie-server/services/rest/server/containers/{containerId}**への **PUT** 要求）を使用してデプロイできます。
実行中の Process Server にプロジェクトをデプロイした後に、Prometheus はメトリクスの収集を開始し、Process Server はメトリクスを REST API エンドポイント **http://HOST:PORT/SERVER/services/rest/metrics** （または Spring Boot では **http://HOST:PORT/rest/metrics**）に公開します。

第18章 OPENSHIFT 接続タイムアウトの設定

デフォルトでは、OpenShift のルートは 30 秒を超えた HTTP リクエストをタイムアウトするように設定されています。これにより Business Central でセッションタイムアウト問題が発生し、以下の動作につながるおそれがあります。

- "Unable to complete your request.The following exception occurred: (TypeError) : Cannot read property 'indexOf' of null."
- "Unable to complete your request.The following exception occurred: (TypeError) : b is null."
- Business Central で **Project** リンクまたは **Server** リンクをクリックすると、空白ページが表示される。

すべての Business Central テンプレートには拡張タイムアウト設定が含まれています。

Business Central OpenShift ルートのタイムアウトを長く設定するには、ターゲットルートに **haproxy.router.openshift.io/timeout: 60s** の注釈を追加します。

```
- kind: Route
  apiVersion: v1
  id: "$APPLICATION_NAME-rhpamcentr-http"
  metadata:
    name: "$APPLICATION_NAME-rhpamcentr"
    labels:
      application: "$APPLICATION_NAME"
    annotations:
      description: Route for Business Central's http service.
      haproxy.router.openshift.io/timeout: 60s
  spec:
    host: "$BUSINESS_CENTRAL_HOSTNAME_HTTP"
    to:
      name: "$APPLICATION_NAME-rhpamcentr"
```

グローバルのルート固有のタイムアウト注釈の完全一覧は、OpenShift ドキュメント を参照してください。

第19章 永続性

バイナリーの永続性、もしくはマーシャリングは、プロセスインスタンスのステータスをバイナリーのデータセットに変換します。バイナリーの永続性は、情報を永続的に保存、取得する際に使用するメカニズムです。同じメカニズムがセッションステータスや作業アイテムのステータスにも適用されています。

プロセスインスタンスの永続性を有効にすると、以下のようになります。

- Red Hat Process Automation Manager はプロセスインスタンス情報をバイナリーデータに変換します。パフォーマンスの理由から、Java の直列化ではなくカスタムの直列化が使用されます。
- バイナリーデータはプロセスインスタンスに関する他のメタデータと併せて保存されます。このメタデータには、プロセスインスタンス ID、プロセス ID、プロセスの開始日が含まれます。

セッションには、タイマージョブのステータスや、ビジネスルールの評価に必要なデータなど、他の形式のステータスを格納することもできます。セッション状態は、セッションの ID およびメタデータとともにバイナリーデータセットとして別途保存されます。指定された ID でセッションを再読み込みすることにより、セッション状態を復元できます。セッション ID は、**ksession.getId()** を使用して取得します。

永続性が設定されていれば、Red Hat Process Automation Manager は以下を維持します。

- **Session state**: セッション ID、最終変更日、ビジネスルールによる評価に必要なセッションデータ、タイマージョブのステータス。
- **Process instance state**: プロセスインスタンス ID、プロセス ID、最終変更日、最終読み取りアクセス日、プロセスインスタンスの開始日、ランタイムデータ (実行されているノード、変数値、その他のプロセスインスタンスデータを含む実行ステータス)、およびイベントタイプ。
- **Work item runtime state**: ワークアイテム ID、作成日、名前、プロセスインスタンス ID、およびワークアイテムステータス。

永続化したデータを基に、障害発生時にはすべての実行中のプロセスインスタンスの実行ステータスを復元したり、メモリーから実行中のインスタンスを一時的に削除し、それらを後で復元することができます。

19.1. PROCESS SERVER の永続性設定

Hibernate または JPA パラメーターをシステムプロパティーとして渡すと、Process Server の永続性を設定できます。

Process Server は以下の接頭辞でシステムプロパティーを確認でき、これらの接頭辞を持つ Hibernate または JPA パラメーターをすべて使用できます。

- **javax.persistence**
- **hibernate**

手順

1. Process Server の永続性を設定するには、以下のタスクのいずれかを実行します。
   Red Hat JBoss EAP 設定ファイルを使用して Process Server の永続性を設定する場合は、以下のタスクを実行します。

i. Red Hat Process Automation Manager インストールディレクトリーで、**standalone-full.xml** ファイルに移動します。たとえば、Red Hat Process Automation Manager に Red Hat JBoss EAP インストールを使用する場合は **$EAP_HOME/standalone/configuration/standalone-full.xml** に移動します。

ii. **standalone-full.xml** ファイルを開き、**<system-properties>** タグの下に、システムプロパティーとして、Hibernate または JPA パラメーターを設定します。

Hibernate パラメーターを使用した Process Server 永続性の設定例

```
<system-properties>
    ...
        <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    ...
<system-properties>
```

JPA パラメーターを使用した Process Server 永続性の設定例

```
<system-properties>
    ...
        <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://mysql.db.server:3306/my_database?
useSSL=false&serverTimezone=UTC"/>
    ...
<system-properties>
```

コマンドラインを使用して Process Server の永続性を設定する場合は、以下のタスクを実行します。

i. 以下のように **-Dkey=value** を使用してコマンドラインからパラメーターを直接渡します。

Hibernate パラメーターを使用して Process Server の永続性を設定する例：

```
$EAP_HOME/bin/standalone.sh -Dhibernate.hbm2ddl.auto=create-drop
```

JPA パラメーターを使用して Process Server の永続性を設定する例：

```
$EAP_HOME/bin/standalone.sh -
Djavax.persistence.jdbc.url=jdbc:mysql://mysql.db.server:3306/my_database?
useSSL=false&serverTimezone=UTC
```

19.2. セーフポイントの設定

永続化を有効にするには、**jbpm-persistence** JAR ファイルをアプリケーションのクラスパスに追加し、プロセスエンジンが永続化を使用するように設定します。プロセスエンジンはセーフポイントに到達すると、自動的にランタイムステータスをストレージに保存します。

セーフポイントとは、プロセスインスタンスが一時停止するポイントです。プロセスエンジンでプロセスインスタンスの呼び出しがセーフポイントに到達すると、プロセスエンジンはプロセスインスタンスの変更をプロセスランタイムデータのスナップショットとして保存します。ただし、プロセスインスタンスが完了すると、永続化されたプロセスインスタンスランタイムデータのスナップショットが自動的に削除されます。

BPMN2 セーフポイントノードを使用すると、プロセスエンジンにより、実行が停止してトランザクションがコミットされた時点のプロセス定義の状態が保存されます。以下の BPMN2 ノードがセーフポイントとみなされます。

- すべての中間キャッチイベント
  - タイマー中間イベント
  - エラー中間イベント
  - 条件中間イベント
  - 補償中間イベント
  - シグナル中間イベント
  - エスカレーション中間イベント
  - メッセージ中間イベント
- ユーザータスク
- ハンドラーでタスクが完了されないカスタム (ユーザーが定義) のサービスタスク。

障害が発生し、ストレージからプロセスエンジンのランタイムを復元する必要がある場合は、プロセスインスタンスは自動的に復元され、それらの実行が再開されるので、手動でプロセスインスタンスを再読み込みしたり、開始したりする必要はありません。

ランタイムの永続データはプロセスエンジン内にあるとみなします。永続的なランタイムデータにアクセスしたり、直接変更したりしないでください。予期しない結果になる場合があります。

現在の実行ステータスについての詳細は、履歴ログを確認してください。本当に必要な場合にのみ、ランタイムデータのデータベースにクエリーしてください。

19.3. セッション永続化エンティティー

セッションは、**SessionInfo** エンティティーとして維持されます。これらはランタイム KIE セッションのステータスを維持し、以下のデータを保存します。

表19.1 SessionInfo

フィールド	説明	Null 許容型
id	プライマリーキー	Null 不可
lastModificationDate	エンティティーがデータベースに最後に保存された時間	
rulesByteArray	セッションのステータス。	Null 不可
startDate	セッションの開始時間。	
OPTLOCK	ロック値を含むバージョンフィールド	

19.4. プロセスインスタンス永続化エンティティー

プロセスインスタンスは、**ProcessInstanceInfo** エンティティーとして維持されます。これはランタイムのプロセスインスタンスのステータスを維持し、以下のデータを保存します。

表19.2 ProcessInstanceInfo

フィールド	説明	Null 許容型
instanceId	プライマリーキー	Null 不可
lastModificationDate	エンティティーがデータベースに最後に保存された時間	
lastReadDate	エンティティーがデータベースから最後に取得された時間。	
processId	プロセス ID	
processInstanceByteArray	バイナリーデータセット形式のプロセスインスタンスのステータス	Null 不可
startDate	プロセスの開始時間	
state	プロセスインスタンスのステータスを示す整数。	Null 不可
OPTLOCK	ロック値を含むバージョンフィールド	

ProcessInstanceInfo には 1:N の関係が **EventTypes** エンティティーとあります。

EventTypes エンティティーには以下のデータが含まれます。

表19.3 EventTypes

フィールド	説明	Null 許容型
instanceId	**ProcessInstanceInfo** プライマリーキーおよびこのコラムの外部キー制約への参照。	Null 不可
element	プロセスで終了したイベント	

19.5. ワークアイテム永続エンティティー

ワークアイテムは **workiteminfo** エンティティーとして維持され、ランタイムのワークアイテムインスタンスのステータスを維持し、以下のデータを保存します。

表19.4 WorkItemInfo

フィールド	説明	Null 許容型
workItemId	プライマリーキー	Null 不可
name	ワークアイテムの名前。	
processInstanceId	プロセスの (プライマリーキー) ID。このフィールドには外部キーの制約はありません。	Null 不可
state	ワークアイテムのステータス。	Null 不可
OPTLOCK	ロック値を含むバージョンフィールド	
workitembytearay	バイナリーデータセットとしてのワークアイテムのステータス。	Null 不可

19.6. 相関キーエンティティー

CorrelationKeyInfo エンティティーには、指定したプロセスインスタンスに割り当てた相関キーに関する情報が含まれます。以下のテーブルはオプションになります。相関機能が必要な場合にのみ、使用してください。

表19.5 CorrelationKeyInfo

フィールド	説明	Null 許容型
keyId	プライマリーキー	Null 不可
name	割り当てられた相関キーの名前。	
processInstanceId	相関キーに割り当てられたプロセスインスタンスの ID	Null 不可
OPTLOCK	ロック値を含むバージョンフィールド	

CorrelationPropertyInfo エンティティーには、プロセスインスタンスに割り当てられた相関キーの相関プロパティーに関する情報が含まれます。

表19.6 CorrelationPropertyInfo

フィールド	説明	Null 許容型
propertyId	プライマリーキー	Null 不可
name	プロパティー名。	

フィールド	説明	Null 許容型
value	プロパティーの値。	Null 不可
OPTLOCK	ロック値を含むバージョンフィールド	
correlationKey_keyId	相関キーにマッピングされた外部キー。	Null 不可

19.7. コンテキストマッピングエンティティー

ContextMappingInfo エンティティーには、**KieSession** にマッピングされたコンテキスト情報に関する情報が含まれます。これは **RuntimeManager** の内部の部分となり、**RuntimeManager** を使用しない場合は任意となります。

表19.7 ContextMappingInfo

フィールド	説明	Null 許容型
mappingId	プライマリーキー	Null 不可
CONTEXT_ID	コンテキスト識別子。	Null 不可
KSESSION_ID	**KieSession** 識別子。	Null 不可
OPTLOCK	ロック値を含むバージョンフィールド	
OWNER_ID	マッピングが関連付けられているデプロイメントユニットの識別子を保持	

19.8. PESSIMISTIC ロックのサポート

プロセスの永続性に関するデフォルトのロックメカニズムは、**optimistic** です。同一プロセスインスタンスにマルチスレッドの同時実行が行われると、このロック戦略はパフォーマンスに悪影響を与えます。

これについては、プロセスベースでユーザーがロックをランタイムで設定できるように変更可能で、**pessimistic** (この変更はプロセスレベルだけでなく、KIE Session レベルやランタイムマネジャーレベルでも可能です) にします。

プロセスが pessimistic ロックを使用するようにするには、ランタイム環境で以下の設定を使用します。

```
import org.kie.api.runtime.Environment;
import org.kie.api.runtime.EnvironmentName;
import org.kie.api.runtime.manager.RuntimeManager;
import org.kie.api.runtime.manager.RuntimeManagerFactory;
```

```
...

env.set(EnvironmentName.USE_PESSIMISTIC_LOCKING, true); (1)

RuntimeManager manager =
RuntimeManagerFactory.Factory.get().newPerRequestRuntimeManager(environment); (2)
```

(1) **env** は **org.kie.api.runtime.Environment** のインスタンスです。

(2) この環境を使用してランタイムマネジャーを作成します。

19.9. 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

Red Hat Process Automation Manager で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 設定および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

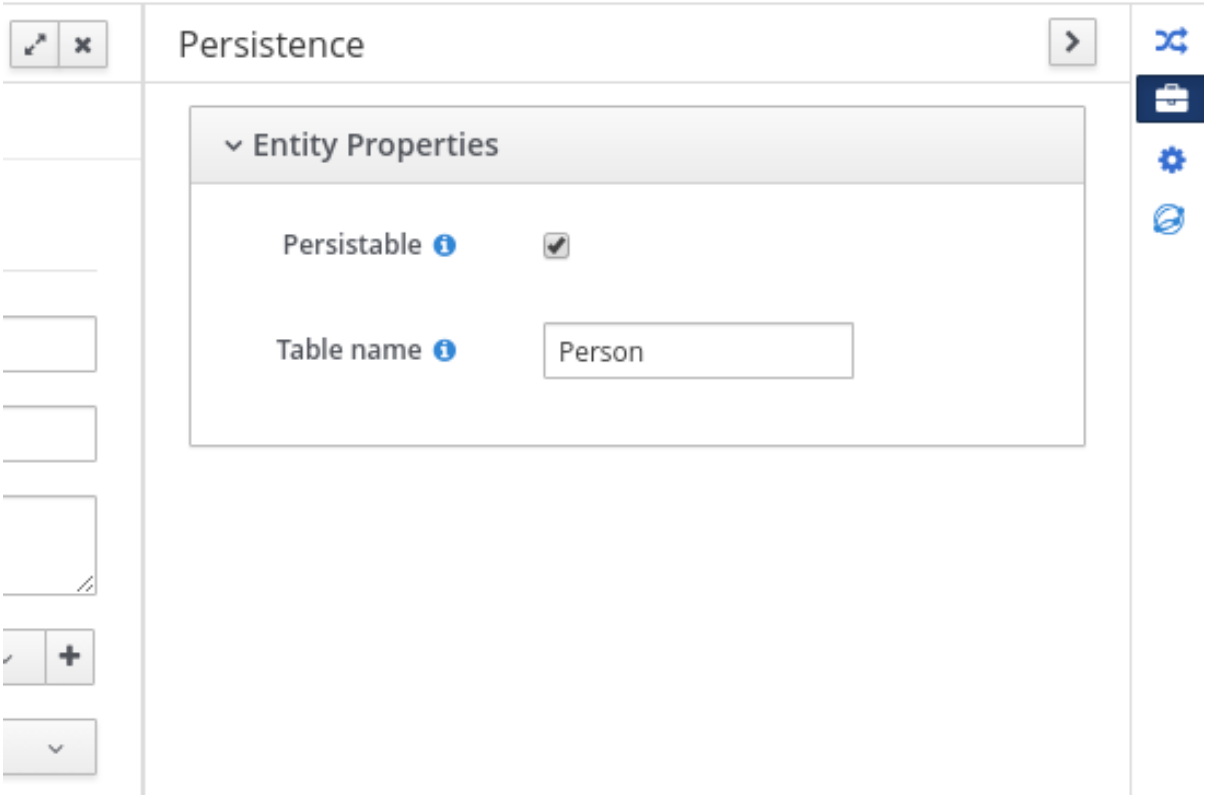
MappedVariable マッピングテーブルを別に作成します。関係の管理が必要ない場合は、**VariableEntity** クラスを継承する必要はありません。この継承がない場合、データオブジェクトは永続化されますが、追加のデータは含まれません。

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19.1 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

kie-deployment-descriptor.xml ファイルで設定するカスタムシリアライザーと永続ユニット

```
<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19.2 Business Central での JPA マーシャラー設定

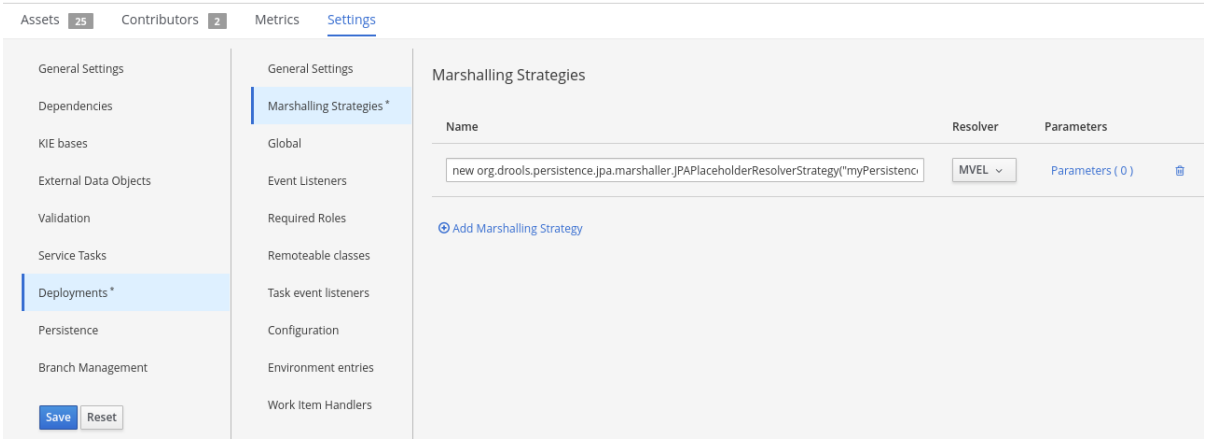
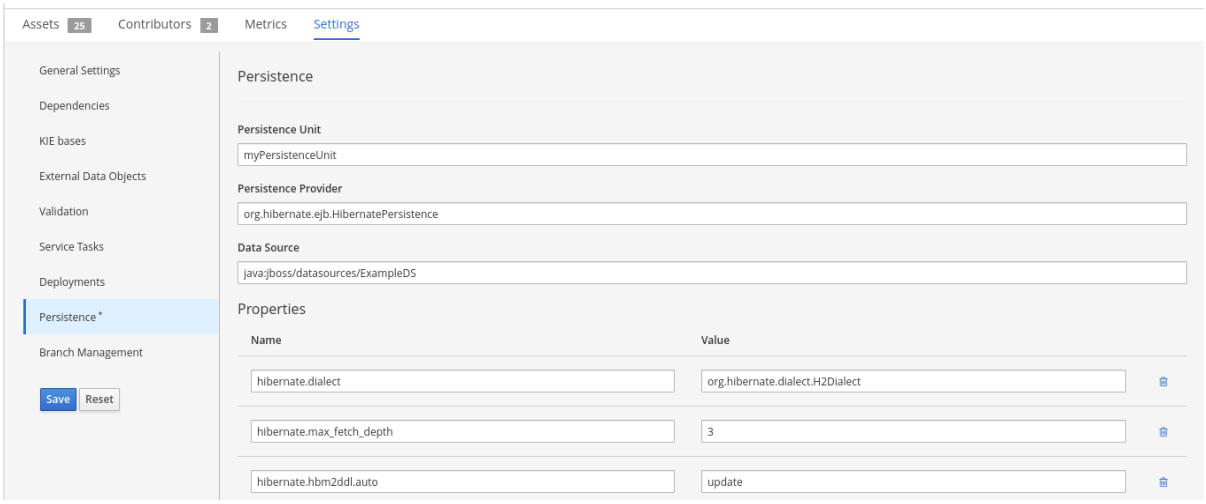


図19.3 Business Central での永続ユニットとデータソース設定



第20章 LDAP ログインドメインの定義

Red Hat Process Automation Manager が認証と承認に LDAP を使用するように設定するには、LDAP ログインドメインを定義します。これは、Git SSH 認証が別のセキュリティードメインを使用する可能性があるためです。

LDAP ログインドメインを定義するには、**org.uberfire.domain** システムプロパティーを使用します。たとえば、Red Hat JBoss Enterprise Application Platform 上でこのプロパティーを以下のように **standalone.xml** ファイルに追加します。

```
<system-properties>
  <!-- other system properties -->
  <property name="org.uberfire.domain" value="LDAPAuth"/>
</system-properties>
```

認証されたユーザーが、LDAP で適切なロール (**admin**、**analyst**、**reviewer**) に関連付けられているようにしてください。

第21章 RH-SSO を使用したサードパーティークライアントの認証

Business Central または Process Server が提供するさまざまなリモートサービスを使用するには、curl、wget、Web ブラウザー、カスタムの REST クライアントなどのクライアントが、RH-SSO サーバー経由で認証を受け、要求を実行するために有効なトークンを取得する必要があります。リモートのサービスを使用するには、認証済みのユーザーに以下のロールを割り当てる必要があります。

- **rest-all**: Business Central リモートサービスを使用する場合
- **kie-server** Process Server リモートサービスを使用する場合

RH-SSO 管理コンソールを使用してこれらのロールを作成し、リモートサービスを使用するユーザーに割り当てます。

クライアントは、以下のオプションのいずれかを使用して RH-SSO 経由で認証できます。

- クライアントでサポートされている場合は Basic 認証
- トークンベースの認証

21.1. BASIC 認証

Business Central および Process Server の両方に対して RH-SSO クライアントアダプターの設定で Basic 認証を有効にした場合には、以下の例のようにトークンの付与/更新の呼び出しをせずにサービスを呼び出すことができます。

- Web ベースのリモートリポジトリーエンドポイントの場合:

```
curl http://admin:password@localhost:8080/business-central/rest/repositories
```

- Process Server の場合:

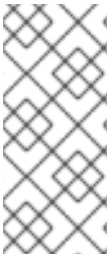
```
curl http://admin:password@localhost:8080/kie-execution-server/services/rest/server/
```

第22章 PROCESS SERVER システムプロパティー

Process Server では、以下のシステムプロパティー（ブートストラップスイッチ）を使用してサーバーの動作を設定できます。

表22.1 Process Server 拡張機能を無効に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.drools.server.ext.disabled	**true**、**false**	**false**	**true** に設定した場合は、(ルールのサポートなど) Business Rule Management (BRM) のサポートが無効になります。
org.jbpm.server.ext.disabled	**true**、**false**	**false**	**true** に設定した場合は、(プロセスのサポートなど) Red Hat Process Automation Manager のサポートが無効になります。
org.jbpm.ui.server.ext.disabled	**true**、**false**	**false**	**true** に設定した場合には、Red Hat Process Automation Manager UI 拡張が無効になります。
org.jbpm.case.server.ext.disabled	**true**、**false**	**false**	**true** に設定すると、Red Hat Process Automation Manager ケース管理の拡張が無効になります。
org.optaplanner.server.ext.disabled	**true**、**false**	**false**	**true** に設定した場合は、Red Hat Business Optimizer のサポートが無効になります。
org.kie.prometheus.server.ext.disabled	**true**、**false**	**true**	**true** に設定した場合は、Prometheus Server 拡張が無効になります。
org.kie.dmn.server.ext.disabled	**true**、**false**	**false**	**true** に設定した場合には、Process Server DMN サポートが無効になります。
org.kie.swagger.server.ext.disabled	**true**、**false**	**false**	**true** に設定した場合には、Process Server swagger のドキュメントサポートが無効になります。



注記

以下の表に記載した Process Automation Manager コントローラーのプロパティーの中で、必須と印がついているものがあります。Business Central で Process Server コンテナーを作成または削除する場合に、このプロパティーを設定してください。Business Central との対話なしに Process Server を別個で使用する場合には、必須のプロパティーを設定する必要はありません。

表22.2 Process Automation Manager コントローラーに必要な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.server.id	String	該当なし	サーバーに割り当てる任意の ID。ヘッドレス Process Automation Manager コントローラーが Business Central 外に設定されている場合は、この ID を使用して、サーバーはヘッドレス Process Automation Manager コントローラーと接続し、KIE コンテナー設定をフェッチします。指定されていない場合、ID は自動で生成されます。
org.kie.server.user	String	**kieserver**	Process Automation Manager コントローラーから Process Server への接続に使用するユーザー名。このプロパティーは、Business Central のシステムプロパティーで設定します。Process Automation Manager コントローラーを使用する場合は、このプロパティーを設定します。
org.kie.server.pwd	String	**kieserver1!**	Process Automation Manager コントローラーから Process Server への接続に使用するパスワード。このプロパティーは、Business Central のシステムプロパティーで設定します。Process Automation Manager コントローラーを使用する場合は、このプロパティーを設定します。
org.kie.server.token	String	該当なし	このプロパティーにより、Process Automation Manager コントローラーと Process Server 間の認証に、ユーザー名/パスワードを使用する Basic 認証ではなく、トークンベースの認証を使用できます。Process Automation Manager コントローラーは、要求ヘッダーのパラメーターとしてトークンを送信します。トークンは更新されないため、サーバーには有効期限の長いアクセストークンが必要です。
org.kie.server.location	URL	該当なし	Process Automation Manager コントローラーが Process Server インスタンスをコールバックするのに使用する URL（例：**http://localhost:8230/kie-server/services/rest/server**）。Process Automation Manager コントローラーを使用する場合は、このプロパティーの設定が必須です。

プロパティー	値	デフォルト	説明
org.kie.server.controller	コンマ区切りのリスト	該当なし	Process Automation Manager コントローラー REST エンドポイントへの URL のコンマ区切りリスト（例：**http://localhost:8080/business-central/rest/controller**）。Process Automation Manager コントローラーを使用する場合は、このプロパティーの設定が必須です。
org.kie.server.controller.user	String	**kieserver**	Process Automation Manager コントローラー REST API に接続するユーザー名。Process Automation Manager コントローラーを使用する場合は、このプロパティーの設定が必須です。
org.kie.server.controller.pwd	String	**kieserver1!**	Process Automation Manager コントローラー REST API に接続するためのパスワード。Process Automation Manager コントローラーを使用する場合は、このプロパティーの設定が必須です。
org.kie.server.controller.token	String	該当なし	このプロパティーにより、Process Server と Process Automation Manager コントローラーとの間の認証に、ユーザー名/パスワードを使用する Basic 認証ではなく、トークンベースの認証を使用できます。このサーバーは、要求ヘッダーのパラメーターとしてトークンを送信します。トークンは更新されないため、サーバーには有効期限の長いアクセストークンが必要です。
org.kie.server.controller.connect	Long	**10000**	サーバーの起動時に Process Server を Process Automation Manager コントローラーに接続することを試み、次に試みるまでの待機時間（ミリ秒単位）。

表22.3 永続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.server.persistence.ds	String	該当なし	データソースの JNDI 名。このプロパティーは、BPM サポートを有効する場合に設定します。

プロパティー	値	デフォルト	説明
org.kie.server.persistence.tm	String	該当なし	Hibernate プロパティーのトランザクションマネージャープラットフォーム。このプロパティーは、BPM サポートを有効する場合に設定します。
org.kie.server.persistence.dialect	String	該当なし	使用する Hibernate 方言。このプロパティーは、BPM サポートを有効する場合に設定します。
org.kie.server.persistence.schema	String	該当なし	使用するデータベーススキーマ

表22.4 エグゼキューターの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.executor.interval	Integer	**0**	Red Hat Process Automation Manager エグゼキューターがジョブを完了してから、新しいジョブを開始するまでの時間。時間の単位は **org.kie.executor.timeunit** プロパティーで指定します。
org.kie.executor.timeunit	**java.util.concurrent.TimeUnit** 定数	**SECONDS**	**org.kie.executor.interval** プロパティーで指定する時間の単位。
org.kie.executor.pool.size	Integer	**1**	Red Hat Process Automation Manager エグゼキューターで使用するスレッド数。
org.kie.executor.retry.count	Integer	**3**	Red Hat Process Automation Manager エグゼキューターが失敗したジョブをリトライする回数。
org.kie.executor.jms.queue	String	**queue/KIE.SERVER.EXECUTOR**	Process Server へのジョブエグゼキューターの JMS キュー。
org.kie.executor.disabled	**true**、**false**	**false**	**true** に設定した場合には、Process Server エグゼキューターが無効になります。

表22.5 ヒューマンタスクのシステムプロパティー

プロパティー	値	デフォルト	説明

プロパティー	値	デフォルト	説明
org.jbpm.ht.callback	**mvel** **ldap** **db** **jaas** **props** **custom**	**jaas**	使用するユーザーグループコールバックの実装を指定するプロパティー ● **mvel**: デフォルト。主にテストで使用します。 ● **ldap**: LDAP。**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。**jbpm.usergroup.callback.properties** ファイルで追加の設定が必要です。 ● **jaas**: JAAS。コンテナーにユーザーデータの情報をフェッチするように委譲します。 ● **props**: 単純なプロパティーファイル。全情報を格納する追加のファイルが必要がです (ユーザーおよびグループ)。 ● **custom**: カスタムの実装。**org.jbpm.ht.custom.callback** プロパティーでクラスの完全修飾名を指定します。
org.jbpm.ht.custom.callback	完全修飾名	該当なし	**org.jbpm.ht.callback** プロパティーが **custom** に設定されている場合の **UserGroupCallback** インターフェイスのカスタム実装
org.jbpm.task.cleanup.enabled	**true**、**false**	**true**	タスク消去のジョブリスナーを有効にして、プロセスインスタンスが完了した時点でタスクを削除します。
org.jbpm.task.bam.enabled	**true**、**false**	**true**	タスクの BAM モジュールを有効にして、タスク関連の情報を保存します。
org.jbpm.ht.admin.user	String	**Administrator**	Process Server からの全タスクにアクセスできるユーザー。
org.jbpm.ht.admin.group	String	**管理者**	Process Server からの全タスクを表示するためにユーザーが所属するグループ。

表22.6 キーストアを読み込むためのシステムプロパティー

プロパティー	値	デフォルト	説明
kie.keystore.keyStoreURL	URL	該当なし	Java Cryptography Extension KeyStore (JCEKS) の読み込みに使用する URL。例: **file:///home/kie/keystores/keystore.jceks**
kie.keystore.keyStorePwd	String	該当なし	JCEKS に使用するパスワード
kie.keystore.key.server.alias	String	該当なし	パスワードの保存先となる REST サービスのキーのエイリアス名
kie.keystore.key.server.pwd	String	該当なし	REST サービスのエイリアスのパスワード
kie.keystore.key.ctrl.alias	String	該当なし	デフォルトの REST Process Automation Manager コントローラー用のキーのエイリアス。
kie.keystore.key.ctrl.pwd	String	該当なし	デフォルトの REST Process Automation Manager コントローラー用のエイリアスのパスワード。

表22.7 その他のシステムプロパティー

プロパティー	値	デフォルト	説明
kie.maven.settings.custom	パス	該当なし	Maven 設定のカスタム **settings.xml** ファイルの場所。
kie.server.jms.queues.response	String	**queue/KIE.SERVER.RESPONSE**	JMS に対する応答キューの JNDI 名。
org.drools.server.filter.classes	**true**、**false**	**false**	**true** に設定した場合、Drools Process Server の拡張機能が受け入れるのは **XmlRootElement** または **Remotable** のアノテーションが付いたカスタムクラスのみです。
org.kie.server.bypass.auth.user	**true**、**false**	**false**	クエリーなど、タスク関連の操作では認証済みユーザーをバイパスできるようにするプロパティー

プロパティー	値	デフォルト	説明
org.jbpm.rule.task.firelimit	Integer	**10000**	このプロパティーは、実行ルールの最大数を指定して、ルールが無限ループに陥って、サーバーが完全に応答不能な状態にならないようにします。
org.jbpm.ejb.timer.local.cache	**true**、**false**	**true**	このプロパティーでは、EJB タイマーのローカルキャッシュをオフにします。
org.kie.server.domain	String	該当なし	JMS を使用する場合にユーザーの認証に使う JAAS **LoginContext** ドメイン。
org.kie.server.repo	パス	**.**	Process Server の状態ファイルが保存される場所。

プロパティー	値	デフォルト	説明
org.kie.server.sync.deploy	**true**、**false**	**false**	Process Server に対して、Process Automation Manager コントローラーがコンテナーのデプロイメント設定を提供するまでデプロイメントを保持するように指示します。このプロパティーは、管理モードで実行するサーバーのみが対象です。以下のオプションが利用できます。 * **false**: Process Automation Manager コントローラーへの接続は非同期です。アプリケーションが起動して、Process Automation Manager コントローラーに接続し、成功すると、コンテナーをデプロイします。アプリケーションはコンテナーが利用可能になる前でもリクエストを受け付けます。* **true**: サーバーアプリケーションのデプロイメントは、Process Automation Manager コントローラーの接続スレッドと、メインのデプロイメントを結合し、完了するまで待機します。このオプションを使用すると、複数のアプリケーションが同じサーバー上にある場合に、デッドロックになる可能性があります。1台のサーバーで使用するアプリケーションは1つだけにしてください。
org.kie.server.startup.strategy	**ControllerBasedStartupStrategy**、**LocalContainersStartupStrategy**	**ControllerBasedStartupStrategy**	デプロイした KIE コンテナーの制御に使用する Process Server の起動ストラテジー、およびデプロイする順番
org.kie.server.mgmt.api.disabled	**true**、**false**	**false**	**true** に設定した場合には、Process Server 管理 API が無効になります。

プロパティー	値	デフォルト	説明
org.kie.server.xstream.enabled.packages	**org.kie.example** などの Java パッケージ。**org.kie.example.*** のようにワイルドカード表現を指定することも可能です。	該当なし	XStream を使用してマーシャリングのホワイトリスト化を行うための追加パッケージを指定するプロパティー
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** を実装する完全修飾クラス名

第23章 PROCESS SERVER の機能と拡張

Process Server の機能は、ビジネスニーズに合わせて有効化、無効化、または拡張可能なプラグインにより決まります。Process Server は以下の機能および拡張をサポートします。

表23.1 Process Server の機能と拡張

機能名	拡張名	説明
KieServer	**KieServer**	サーバーインスタンスでの KIE コンテナーの作成や破棄など、Process Server のコア機能を提供します。
BRM	**Drools**	ファクトの挿入やビジネスルールの実行など、ビジネスルール管理 (BRM) 機能を提供します。
BPM	**jBPM**	ユーザータスクの管理や、ビジネスプロセスの実行など、Business Process Management (BPM) 機能を提供します。
BPM-UI	**jBPM-UI**	XML フォームや SVG イメージをプロセスダイアグラムにレンダリングするなど、ビジネスプロセスに関連するユーザーインターフェイス機能を提供します。
CaseMgmt	**Case-Mgmt**	ケースの定義やマイルストーン管理など、ビジネスプロセスのケース管理機能を提供します。
BRP	**OptaPlanner**	ソルバーの実装など、ビジネスリソースプランニング (BRP) 機能を提供します。
DMN	**DMN**	DMN データ型の管理や DMN モデルの実行など、Decision Model and Notation (DMN) 機能を提供します。
Swagger	**Swagger**	Process Server REST API と対話するための Swagger の Web インターフェース機能を提供します。

実行中の Process Server インスタンスに対応する拡張を表示するには、以下の REST API エンドポイントに **GET** リクエストを送信し、XML または JSON サーバーの応答を確認します。

Process Server の情報に対する GET 要求のベース URL

```
http://SERVER:PORT/kie-server/services/rest/server
```

Process Server の情報を含む JSON 応答の例

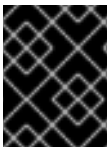
```
{
  "type": "SUCCESS",
  "msg": "Kie Server info",
  "result": {
    "kie-server-info": {
      "id": "test-kie-server",
      "version": "7.26.0.20190818-050814",
      "name": "test-kie-server",
```

```
      "location": "http://localhost:8080/kie-server/services/rest/server",
      "capabilities": [
        "KieServer",
        "BRM",
        "BPM",
        "CaseMgmt",
        "BPM-UI",
        "BRP",
        "DMN",
        "Swagger"
      ],
      "messages": [
        {
          "severity": "INFO",
          "timestamp": {
            "java.util.Date": 1566169865791
          },
          "content": [
            "Server KieServerInfo{serverId='test-kie-server', version='7.26.0.20190818-050814',
name='test-kie-server', location='http:/localhost:8080/kie-server/services/rest/server', capabilities=
[KieServer, BRM, BPM, CaseMgmt, BPM-UI, BRP, DMN, Swagger]', messages=null',
mode=DEVELOPMENT}started successfully at Sun Aug 18 23:11:05 UTC 2019"
          ]
        }
      ],
      "mode": "DEVELOPMENT"
    }
  }
}
```


Process Server 拡張機能を有効または無効にするには、関連する Process Server システムプロパティー (***.server.ext.disabled**)を設定します。たとえば、**BRM** 機能を無効にするには、**org.drools.server.ext.disabled=true** システムプロパティーを設定します。全 Process Server システムプロパティーについては、*22章Process Server システムプロパティー* を参照してください。

デフォルトでは、Process Server 拡張機能は REST または JMS データトランスポートで公開され、事前定義済みのクライアント API を使用します。追加の REST エンドポイントで既存の Process Server 機能を拡張するか、REST または JMS 以外の対応のトランスポートメソッドを拡張するか、Process Server クライアントの機能を拡張できます。

Process Server 機能は柔軟であるため、デフォルトの Process Server 機能にビジネスニーズを合わせるのではなく、Process Server インスタンスをビジネスニーズに適合できます。



重要

Process Server 機能を拡張した場合には、Red Hat では、カスタムの実装や拡張の一部として使用したカスタムコードをサポートしません。

23.1. カスタム REST API エンドポイントを使用した既存の PROCESS SERVER 機能の拡張

Process Server REST API を使用すると、Business Central ユーザーインターフェイスを使わずに Red Hat Process Automation Manager の KIE コンテナーやビジネスアセット (ビジネスルールやプロセス、ソルバーなど) を操作することができます。利用可能な REST エンドポイントは、Process Server シス

テムプロパティーで有効にした機能により決まります（例： **BRM** 機能は **org.drools.server.ext.disabled=false** ）。既存の Process Server 機能は、カスタムの REST API エンドポイントで拡張し、ビジネスニーズに合わせて Process Server REST API を適合できます。

たとえば、この手順では、以下のカスタム REST API エンドポイントで **Drools** Process Server 機能（ **BRM** 機能向け）を拡張します。

カスタム REST API エンドポイントの例

```
/server/containers/instances/{containerId}/ksession/{ksessionId}
```

このカスタムのエンドポイントの例では、デシジョンエンジンの作業メモリーに挿入するファクト一覧を受け入れ、自動的に全ルールを実行して、指定の KIE コンテナーで KIE セッションからのオブジェクトをすべて取得します。

手順

1. 空の Maven プロジェクトを作成して、以下のパッケージタイプと依存関係を、プロジェクトの **pom.xml** ファイルに定義します。

   サンプルプロジェクトの pom.xml ファイルの例

```
<packaging>jar</packaging>

<properties>
  <version.org.kie>7.26.0.Final-redhat-00005</version.org.kie>
</properties>

<dependencies>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-api</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-internal</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-api</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-services-common</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-services-drools</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
```

```
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-rest-common</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-core</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.25</version>
  </dependency>
</dependencies>
```

2. 以下の例のように、プロジェクトの Java クラスに **org.kie.server.services.api.KieServerApplicationComponentsService** インターフェイスを実装します。

KieServerApplicationComponentsService インターフェイスの実装例

```
public class CusomtDroolsKieServerApplicationComponentsService implements
KieServerApplicationComponentsService {  1

    private static final String OWNER_EXTENSION = "Drools";  2

    public Collection<Object> getAppComponents(String extension, SupportedTransports
type, Object... services) {  3
        // Do not accept calls from extensions other than the owner extension:
        if ( !OWNER_EXTENSION.equals(extension) ) {
            return Collections.emptyList();
        }

        RulesExecutionService rulesExecutionService = null;  4
        KieServerRegistry context = null;

        for( Object object : services ) {
            if( RulesExecutionService.class.isAssignableFrom(object.getClass()) ) {
                rulesExecutionService = (RulesExecutionService) object;
                continue;
            } else if( KieServerRegistry.class.isAssignableFrom(object.getClass()) ) {
                context = (KieServerRegistry) object;
                continue;
            }
        }

        List<Object> components = new ArrayList<Object>(1);
        if( SupportedTransports.REST.equals(type) ) {
            components.add(new CustomResource(rulesExecutionService, context));  5
```

```
        }

        return components;
    }

}
```

1. アプリケーションの起動時にデプロイされる Process Server インフラストラクチャーに REST エンドポイントを提供します。
2. この例の **Drools** 拡張など、拡張する機能を指定します。
3. REST コンテナーがデプロイする必要のある全リソースを返します。Process Server インスタンスで有効化した各拡張で **getAppComponents** メソッドを呼び出して、指定した **OWNER_EXTENSION** 以外の拡張の空のコレクションを、**if（!OWNER_EXTENSION.equals(extension)）** の呼び出しで返します。
4. この例の **Drools** 拡張の **RulesExecutionService** や **KieServerRegistry** サービスなど、指定の拡張から使用するサービスを表示します。
5. **components** リストの一部としてリソースを返す **CustomResource** クラスと、拡張のトランスポートタイプを **REST** または **JMS** に指定します (この例では **REST**)。

3. 以下の例のように、Process Server を使用して新規の REST リソースの機能を追加する **CustomResource** クラスを実装します。

CustomResource クラスの実装例

```
// Custom base endpoint:
@Path("server/containers/instances/{containerId}/ksession")
public class CustomResource {

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

    private KieCommands commandsFactory = KieServices.Factory.get().getCommands();

    private RulesExecutionService rulesExecutionService;
    private KieServerRegistry registry;

    public CustomResource() {

    }

    public CustomResource(RulesExecutionService rulesExecutionService, KieServerRegistry
registry) {
        this.rulesExecutionService = rulesExecutionService;
        this.registry = registry;
    }

    // Supported HTTP method, path parameters, and data formats:
    @POST
    @Path("/{ksessionId}")
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response insertFireReturn(@Context HttpHeaders headers,
```

```
            @PathParam("containerId") String id,
            @PathParam("ksessionId") String ksessionId,
            String cmdPayload) {

        Variant v = getVariant(headers);
        String contentType = getContentType(headers);

        // Marshalling behavior and supported actions:
        MarshallingFormat format = MarshallingFormat.fromType(contentType);
        if (format == null) {
            format = MarshallingFormat.valueOf(contentType);
        }
        try {
            KieContainerInstance kci = registry.getContainer(id);

            Marshaller marshaller = kci.getMarshaller(format);

            List<?> listOfFacts = marshaller.unmarshall(cmdPayload, List.class);

            List<Command<?>> commands = new ArrayList<Command<?>>();
            BatchExecutionCommand executionCommand =
commandsFactory.newBatchExecution(commands, ksessionId);

            for (Object fact : listOfFacts) {
                commands.add(commandsFactory.newInsert(fact, fact.toString()));
            }
            commands.add(commandsFactory.newFireAllRules());
            commands.add(commandsFactory.newGetObjects());

            ExecutionResults results = rulesExecutionService.call(kci, executionCommand);

            String result = marshaller.marshall(results);


            logger.debug("Returning OK response with content '{}'", result);
            return createResponse(result, v, Response.Status.OK);
        } catch (Exception e) {
            // If marshalling fails, return the `call-container` response to maintain backward
compatibility:
            String response = "Execution failed with error : " + e.getMessage();
            logger.debug("Returning Failure response with content '{}'", response);
            return createResponse(response, v,
Response.Status.INTERNAL_SERVER_ERROR);
        }

    }
}
```

この例では、カスタムエンドポイントの **CustomResource** クラスで、以下のデータと動作を指定します。

- **server/containers/instances/{containerId}/ksession** のベースポイントを使用します。
- **POST** HTTP メソッドを使用します。
- REST 要求で以下のデータを指定する必要があります。

- パスの引数として **containerId**
- パスの引数として **ksessionId**
- メッセージペイロードとしてファクトの一覧

- 全 Process Server データ形式をサポートします。
  - XML (JAXB、XStream)
  - JSON

- **List<?>** コレクションにペイロードをアンマーシャリングして、リスト内のアイテムごとに、**InsertCommand** インスタンスを作成し、その後に **FireAllRules** と **GetObject** コマンドを追加します。
- デシジョンエンジンを呼び出す **BatchExecutionCommand** インスタンスに全コマンドを追加します。

4. 新規エンドポイントを Process Server で検出できるようにするには、Maven プロジェクトに **META-INF/services/org.kie.server.services.api.KieServerApplicationComponentsService** ファイルを作成して、このファイルに **KieServerApplicationComponentsService** 実装クラスの完全修飾名を追加します。たとえば、このファイルには、**org.kie.server.ext.drools.rest.CusomtDroolsKieServerApplicationComponentsService** の 1 行が含まれます。

5. プロジェクトを構築して、作成された JAR ファイルをプロジェクトの **~/kie-server.war/WEB-INF/lib** ディレクトリーにコピーします。たとえば、Red Hat JBoss EAP ではこのディレクトリーへのパスは **EAP_HOME/standalone/deployments/kie-server.war/WEB-INF/lib** です。

6. Process Server を起動して、実行中の Process Server に構築したプロジェクトをデプロイします。プロジェクトは、Business Central インターフェースまたは Process Server REST API（**http://SERVER:PORT/kie-server/services/rest/server/containers/{containerId}**への **PUT** 要求）を使用してデプロイできます。
実行中の Process Server にプロジェクトを追加した後に、新しい REST エンドポイントとの対話を開始します。

   今回の例では、以下の情報を使用して新規エンドポイントを呼び出すことができます。

   - **要求 URL 例: http://localhost:8080/kie-server/services/rest/server/containers/instances/demo/ksession/defaultKieSession**
   - **HTTP メソッド: POST**
   - **HTTP ヘッダー:**
     - **Content-Type: application/json**
     - **Accept: application/json**
   - **メッセージペイロードの例:**

```
[
  {
    "org.jbpm.test.Person": {
      "name": "john",
      "age": 25
```

```
    }
   },
   {
    "org.jbpm.test.Person": {
     "name": "mary",
     "age": 22
    }
   }
]
```

- **サーバーの応答例: 200** (success)
- **サーバーのログ出力例:**

```
13:37:20,347 INFO  [stdout] (default task-24) Hello mary
13:37:20,348 INFO  [stdout] (default task-24) Hello john
```

23.2. カスタムデータトランスポートを使用するための PROCESS SERVER の拡張

デフォルトでは、Process Server 拡張機能は REST または JMS データトランスポートを使用して公開されます。Process Server を拡張して、カスタムのデータトランスポートのサポートを追加し、Process Server トランスポートプロトコルをビジネスニーズに適合します。

たとえば、以下の手順では、**Drools** 拡張を使用し、Apache MINA（オープンソースの Java ネットワークアプリケーションフレームワーク）をベースとする Process Server にカスタムのデータトランスポートを追加します。カスタムの MINA トランスポートの例では、既存のマーシャリング操作に依存し、JSON 形式のみをサポートする文字列ベースのデータを変換します。

手順

1. 空の Maven プロジェクトを作成して、以下のパッケージタイプと依存関係を、プロジェクトの **pom.xml** ファイルに定義します。

サンプルプロジェクトの pom.xml ファイルの例

```
<packaging>jar</packaging>

<properties>
 <version.org.kie>7.26.0.Final-redhat-00005</version.org.kie>
</properties>

<dependencies>
 <dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-api</artifactId>
  <version>${version.org.kie}</version>
 </dependency>
 <dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-internal</artifactId>
  <version>${version.org.kie}</version>
 </dependency>
 <dependency>
```

```
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-api</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-services-common</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-services-drools</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-core</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.25</version>
  </dependency>
  <dependency>
    <groupId>org.apache.mina</groupId>
    <artifactId>mina-core</artifactId>
    <version>2.1.3</version>
  </dependency>
</dependencies>
```

2. 以下の例のように、プロジェクトの Java クラスに **org.kie.server.services.api.KieServerExtension** インターフェイスを実装します。

KieServerExtension インターフェイスの実装例

```
public class MinaDroolsKieServerExtension implements KieServerExtension {

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

    public static final String EXTENSION_NAME = "Drools-Mina";

    private static final Boolean disabled =
Boolean.parseBoolean(System.getProperty("org.kie.server.drools-mina.ext.disabled",
"false"));
    private static final String MINA_HOST = System.getProperty("org.kie.server.drools-
mina.ext.port", "localhost");
    private static final int MINA_PORT =
Integer.parseInt(System.getProperty("org.kie.server.drools-mina.ext.port", "9123"));
```

```
    // Taken from dependency on the `Drools` extension:
    private KieContainerCommandService batchCommandService;

    // Specific to MINA:
    private IoAcceptor acceptor;

    public boolean isActive() {
        return disabled == false;
    }

    public void init(KieServerImpl kieServer, KieServerRegistry registry) {

        KieServerExtension droolsExtension = registry.getServerExtension("Drools");
        if (droolsExtension == null) {
            logger.warn("No Drools extension available, quiting...");
            return;
        }

        List<Object> droolsServices = droolsExtension.getServices();
        for( Object object : droolsServices ) {
            // If the given service is null (not configured), continue to the next service:
            if (object == null) {
                continue;
            }
            if( KieContainerCommandService.class.isAssignableFrom(object.getClass()) ) {
                batchCommandService = (KieContainerCommandService) object;
                continue;
            }
        }
        if (batchCommandService != null) {
            acceptor = new NioSocketAcceptor();
            acceptor.getFilterChain().addLast( "codec", new ProtocolCodecFilter( new
TextLineCodecFactory( Charset.forName( "UTF-8" ))));

            acceptor.setHandler( new TextBasedIoHandlerAdapter(batchCommandService) );
            acceptor.getSessionConfig().setReadBufferSize( 2048 );
            acceptor.getSessionConfig().setIdleTime( IdleStatus.BOTH_IDLE, 10 );
            try {
                acceptor.bind( new InetSocketAddress(MINA_HOST, MINA_PORT) );

                logger.info("{} -- Mina server started at {} and port {}", toString(), MINA_HOST,
MINA_PORT);
            } catch (IOException e) {
                logger.error("Unable to start Mina acceptor due to {}", e.getMessage(), e);
            }

        }
    }

    public void destroy(KieServerImpl kieServer, KieServerRegistry registry) {
        if (acceptor != null) {
            acceptor.dispose();
            acceptor = null;
        }
        logger.info("{} -- Mina server stopped", toString());
    }
```

```
    public void createContainer(String id, KieContainerInstance kieContainerInstance,
Map<String, Object> parameters) {
        // Empty, already handled by the `Drools` extension

    }

    public void disposeContainer(String id, KieContainerInstance kieContainerInstance,
Map<String, Object> parameters) {
      // Empty, already handled by the `Drools` extension

    }

    public List<Object> getAppComponents(SupportedTransports type) {
        // Nothing for supported transports (REST or JMS)
        return Collections.emptyList();
    }

    public <T> T getAppComponents(Class<T> serviceType) {

        return null;
    }

    public String getImplementedCapability() {
        return "BRM-Mina";
    }

    public List<Object> getServices() {
        return Collections.emptyList();
    }

    public String getExtensionName() {
        return EXTENSION_NAME;
    }

    public Integer getStartOrder() {
        return 20;
    }

    @Override
    public String toString() {
        return EXTENSION_NAME + " KIE Server extension";
    }
}
```

KieServerExtension インターフェースは、新規の MINA トランスポートの機能を追加する時に Process Server が使用する主要な拡張インターフェースです。このインターフェイスには、以下のコンポーネントが含まれます。

KieServerExtension インターフェイスの概要

```
public interface KieServerExtension {

    boolean isActive();

    void init(KieServerImpl kieServer, KieServerRegistry registry);
```

```
    void destroy(KieServerImpl kieServer, KieServerRegistry registry);

    void createContainer(String id, KieContainerInstance kieContainerInstance, Map<String,
Object> parameters);

    void disposeContainer(String id, KieContainerInstance kieContainerInstance, Map<String,
Object> parameters);

    List<Object> getAppComponents(SupportedTransports type);

    <T> T getAppComponents(Class<T> serviceType);

    String getImplementedCapability(); (1)

    List<Object> getServices();

    String getExtensionName(); (2)

    Integer getStartOrder(); (3)
}
```

(1) この拡張で対応している機能を指定します。この機能は、Process Server 内で一意でなければなりません。

(2) 拡張は、人間が解読可能な名前に定義します。

(3) 指定した拡張の起動のタイミングを決定します。他の拡張と依存関係がある拡張の場合、この設定は親の設定と競合しないようにしてください。たとえば、今回の場合、このカスタムの拡張は **Drools** 拡張に依存しており、Drool 拡張の **StartOrder** は **0** に設定されているため、このカスタムのアドオン拡張は **0** を超える値でなければなりません (サンプルの実装では **20** に設定)。

このインターフェイスの先程の **MinaDroolsKieServerExtension** 実装例では、**init** メソッドが主に、**Drools** 拡張からサービスを収集して、MINA サーバーをブートストラップ化する要素となっています。**KieServerExtension** インターフェイスの他のメソッドは、標準の実装のままで、インターフェイスの要件を満たします。

TextBasedIoHandlerAdapter クラスは、受信要求に対応する MINA サーバーにあるハンドラーです。

3. 以下の例のように、MINA サーバーの **TextBasedIoHandlerAdapter** ハンドラーを実装します。

TextBasedIoHandlerAdapter ハンドラーの実装例

```
public class TextBasedIoHandlerAdapter extends IoHandlerAdapter {

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

    private KieContainerCommandService batchCommandService;

    public TextBasedIoHandlerAdapter(KieContainerCommandService
batchCommandService) {
```

```
        this.batchCommandService = batchCommandService;
    }

    @Override
    public void messageReceived( IoSession session, Object message ) throws Exception {
        String completeMessage = message.toString();
        logger.debug("Received message '{}'", completeMessage);
        if( completeMessage.trim().equalsIgnoreCase("quit") ||
completeMessage.trim().equalsIgnoreCase("exit") ) {
            session.close(false);
            return;
        }

        String[] elements = completeMessage.split("\\|");
        logger.debug("Container id {}", elements[0]);
        try {
            ServiceResponse<String> result = batchCommandService.callContainer(elements[0],
elements[1], MarshallingFormat.JSON, null);

            if (result.getType().equals(ServiceResponse.ResponseType.SUCCESS)) {
                session.write(result.getResult());
                logger.debug("Successful message written with content '{}'", result.getResult());
            } else {
                session.write(result.getMsg());
                logger.debug("Failure message written with content '{}'", result.getMsg());
            }
        } catch (Exception e) {

        }
    }
}
```

この例では、ハンドラークラスはテキストメッセージを受信して、**Drools** サービスでこのメッセージを実行します。

TextBasedIoHandlerAdapter ハンドラー実装を使用する場合は、以下のハンドラー要件と動作を考慮してください。

- 各受信トランスポート要求が 1 行であるため、ハンドラーに送信する内容は、1 行でなければなりません。
- ハンドラーで **containerID|payload** の形式が想定されるように、この 1 行に KIE コンテナー ID を渡す必要があります。
- マーシャラーで生成される方法で応答を設定できます。応答は複数行にすることができます。
- このハンドラーは **stream mode** をサポートし、Process Server セッションを切断せずにコマンドを送信できます。ストリームモードで Process Server セッションを終了するには、サーバーに **exit** コマンドまたは **quit** コマンドを送信してください。

4. 新規のデータトランスポートを Process Server で検出できるようにするには、Maven プロジェクトで **META-INF/services/org.kie.server.services.api.KieServerExtension** ファイルを作成し、このファイルに **KieServerExtension** 実装クラスの完全修飾名を追加します。たとえば、このファイルには **org.kie.server.ext.mina.MinaDroolsKieServerExtension** の 1 行が含まれます。

5. プロジェクトを構築して、作成された JAR ファイルと **mina-core-2.0.9.jar** ファイル (今回の例でこの拡張が依存) をプロジェクトの ~/**kie-server.war**/**WEB-INF**/**lib** ディレクトリーにコピーします。たとえば、Red Hat JBoss EAP ではこのディレクトリーへのパスは **EAP_HOME**/**standalone**/**deployments**/**kie-server.war**/**WEB-INF**/**lib** です。

6. Process Server を起動して、実行中の Process Server に構築したプロジェクトをデプロイします。プロジェクトは、Business Central インターフェースまたは Process Server REST API（**http://SERVER:PORT/kie-server/services/rest/server/containers/{containerId}**への **PUT** 要求）を使用してデプロイできます。
プロジェクトを実行中の Process Server にデプロイした後に、Process Server ログで新規データトランスポートのステータスを表示して、新しいデータトランスポートの使用を開始できます。

サーバーログの新規データトランスポート

```
Drools-Mina KIE Server extension -- Mina server started at localhost and port 9123
Drools-Mina KIE Server extension has been successfully registered as server extension
```

この例では、Telnet を使用して Process Server の新しい MINA ベースのデータトランスポートと対話できます。

コマンドターミナルでの Telnet の開始およびポート 9123 での Process Server の接続

```
telnet 127.0.0.1 9123
```

コマンドターミナルでの Process Server との対話例

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

# Request body:
demo|{"lookup":"defaultKieSession","commands":[{"insert":{"object":{"org.jbpm.test.Person":
{"name":"john","age":25}}}},{"fire-all-rules":""}]}

# Server response:
{
  "results" : [ {
    "key" : "",
    "value" : 1
  } ],
  "facts" : [ ]
}

demo|{"lookup":"defaultKieSession","commands":[{"insert":{"object":{"org.jbpm.test.Person":
{"name":"mary","age":22}}}},{"fire-all-rules":""}]}
{
  "results" : [ {
    "key" : "",
    "value" : 1
  } ],
  "facts" : [ ]
}

demo|{"lookup":"defaultKieSession","commands":[{"insert":{"object":{"org.jbpm.test.Person":
```

```
{"name":"james","age":25}}}},{"fire-all-rules":""}]}
{
  "results" : [ {
    "key" : "",
    "value" : 1
  } ],
  "facts" : [ ]
}
exit
Connection closed by foreign host.
```

サーバーログの出力例

```
16:33:40,206 INFO  [stdout] (NioProcessor-2) Hello john
16:34:03,877 INFO  [stdout] (NioProcessor-2) Hello mary
16:34:19,800 INFO  [stdout] (NioProcessor-2) Hello james
```

23.3. カスタムクライアント API を使用した PROCESS SERVER クライアントの拡張

Process Server は、Process Server サービスの使用時に対話可能な、事前定義済みのクライアント API を使用します。カスタムのクライアント API で Process Server クライアントを拡張して、ビジネスのニーズに Process Server サービスを適合させます。

たとえば、以下の手順では、カスタムのクライアント API を Process Server に追加して、Apache MINA（オープンソースの Java ネットワークアプリケーションフレームワーク）をもとにしたカスタムのデータトランスポートに対応します（このシナリオ向けにすでに設定済み）。

手順

1. 空の Maven プロジェクトを作成して、以下のパッケージタイプと依存関係を、プロジェクトの **pom.xml** ファイルに定義します。

サンプルプロジェクトの pom.xml ファイルの例

```
<packaging>jar</packaging>

<properties>
  <version.org.kie>7.26.0.Final-redhat-00005</version.org.kie>
 </properties>

 <dependencies>
  <dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-api</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
  <dependency>
     <groupId>org.kie.server</groupId>
     <artifactId>kie-server-client</artifactId>
     <version>${version.org.kie}</version>
   </dependency>
  <dependency>
    <groupId>org.drools</groupId>
```

```
    <artifactId>drools-compiler</artifactId>
    <version>${version.org.kie}</version>
  </dependency>
</dependencies>
```

2. 以下の例のように、プロジェクトの Java クラスに、関連する **ServicesClient** インターフェイスを実装します。

RulesMinaServicesClient インターフェイスの例

```
public interface RulesMinaServicesClient extends RuleServicesClient {

}
```

インターフェイスをもとにクライアントの実装を登録する必要があるため、特定のインターフェイスが必要です。また、指定のインターフェイスには実装は 1 つしか指定できません。

この例では、カスタムの MINA ベースのデータトランスポートが **Drools** 拡張を使用し、この **RulesMinaServicesClient** インターフェイスの例は、**Drools** 拡張から、既存の **RuleServicesClient** クライアント API を拡張します。

3. 以下の例のように、新規の MINA トランスポートのクライアント機能を追加するのに Process Server が使用可能な **RulesMinaServicesClient** インターフェースを実装します。

RulesMinaServicesClient インターフェイスの実装例

```
public class RulesMinaServicesClientImpl implements RulesMinaServicesClient {

    private String host;
    private Integer port;

    private Marshaller marshaller;

    public RulesMinaServicesClientImpl(KieServicesConfiguration configuration, ClassLoader
classloader) {
        String[] serverDetails = configuration.getServerUrl().split(":");

        this.host = serverDetails[0];
        this.port = Integer.parseInt(serverDetails[1]);

        this.marshaller = MarshallerFactory.getMarshaller(configuration.getExtraJaxbClasses(),
MarshallingFormat.JSON, classloader);
    }

    public ServiceResponse<String> executeCommands(String id, String payload) {

        try {
            String response = sendReceive(id, payload);
            if (response.startsWith("{")) {
                return new ServiceResponse<String>(ResponseType.SUCCESS, null, response);
            } else {
                return new ServiceResponse<String>(ResponseType.FAILURE, response);
            }
        } catch (Exception e) {
            throw new KieServicesException("Unable to send request to KIE Server", e);
```

```
        }
    }

    public ServiceResponse<String> executeCommands(String id, Command<?> cmd) {
        try {
            String response = sendReceive(id, marshaller.marshall(cmd));
            if (response.startsWith("{")) {
                return new ServiceResponse<String>(ResponseType.SUCCESS, null, response);
            } else {
                return new ServiceResponse<String>(ResponseType.FAILURE, response);
            }
        } catch (Exception e) {
            throw new KieServicesException("Unable to send request to KIE Server", e);
        }
    }

    protected String sendReceive(String containerId, String content) throws Exception {

        // Flatten the content to be single line:
        content = content.replaceAll("\\n", "");

        Socket minaSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        StringBuffer data = new StringBuffer();
        try {
            minaSocket = new Socket(host, port);
            out = new PrintWriter(minaSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(minaSocket.getInputStream()));

            // Prepare and send data:
            out.println(containerId + "|" + content);
            // Wait for the first line:
            data.append(in.readLine());
            // Continue as long as data is available:
            while (in.ready()) {
                data.append(in.readLine());
            }

            return data.toString();
        } finally {
            out.close();
            in.close();
            minaSocket.close();
        }
    }
}
```

この実装例は、以下のデータおよび動作を指定します。

- ソケットベースの通信を使用して簡素化します。
- Process Server クライアントのデフォルト設定に依存し、**ServerUrl** を使用して MINA サーバーのホストとポートを提供します。

- マーシャリング形式で JSON を指定します。
- 受信メッセージは左波括弧 **{** で始まる JSON オブジェクトでなければなりません。
- 応答の最初の行を待機中に、ブロッキング API と直接、ソケット通信を使用してから、利用可能なすべての行を読み取ります。
- **ストリームモード** を使用しないので、コマンドの呼び出し後に Process Server セッションを切断します。

4. 以下の例のように、プロジェクトの Java クラスに **org.kie.server.client.helper.KieServicesClientBuilder** インターフェイスを実装します。

KieServicesClientBuilder インターフェイスの実装例

```
public class MinaClientBuilderImpl implements KieServicesClientBuilder { (1)

    public String getImplementedCapability() { (2)
        return "BRM-Mina";
    }

    public Map<Class<?>, Object> build(KieServicesConfiguration configuration, ClassLoader
classLoader) { (3)
        Map<Class<?>, Object> services = new HashMap<Class<?>, Object>();

        services.put(RulesMinaServicesClient.class, new
RulesMinaServicesClientImpl(configuration, classLoader));

        return services;
    }

}
```

(1) 一般の Process Server クライアントインフラストラクチャーにクライアント API を追加できます。

(2) クライアントが使用する Process Server 機能（拡張）を定義します。

(3) クライアントの実装のマッピングを提供します。キーはインターフェイス、値は完全な初期実装です。

5. 新規のクライアント API を Process Server クライアントで検出できるようにするには、Maven プロジェクトで **META-INF/services/org.kie.server.client.helper.KieServicesClientBuilder** ファイルを作成し、このファイルに **KieServicesClientBuilder** 実装クラスの完全修飾名を追加します。たとえば、このファイルには **org.kie.server.ext.mina.client.MinaClientBuilderImpl** の 1 行が含まれます。

6. プロジェクトを構築して、作成された JAR ファイルをプロジェクトの ~/**kie-server.war/WEB-INF/lib** ディレクトリーにコピーします。たとえば、Red Hat JBoss EAP ではこのディレクトリーへのパスは **EAP_HOME/standalone/deployments/kie-server.war/WEB-INF/lib** です。

7. Process Server を起動して、実行中の Process Server に構築したプロジェクトをデプロイします。プロジェクトは、Business Central インターフェースまたは Process Server REST API（**http://SERVER:PORT/kie-server/services/rest/server/containers/{containerId}**への **PUT** 要求）を使用してデプロイできます。

実行中の Process Server にプロジェクトを追加した後に、新しい Process Server クライアントと対話を開始できます。標準の Process Server クライアントと同じ方法で、クライアント設定とクライアントインスタンスを作成し、タイプ別にサービスクライアントを取得し、クライアントメソッドを呼び出して、新しいクライアントを使用します。

たとえば、**RulesMinaServiceClient** クライアントインスタンスを作成して、MINA トランスポートを使用して Process Server で操作を呼び出すことができます。

RulesMinaServiceClient クライアント作成の実装例

```
protected RulesMinaServicesClient buildClient() {
    KieServicesConfiguration configuration =
KieServicesFactory.newRestConfiguration("localhost:9123", null, null);
    List<String> capabilities = new ArrayList<String>();
    // Explicitly add capabilities (the MINA client does not respond to `get-server-info`
requests):
    capabilities.add("BRM-Mina");

    configuration.setCapabilities(capabilities);
    configuration.setMarshallingFormat(MarshallingFormat.JSON);

    configuration.addJaxbClasses(extraClasses);

    KieServicesClient kieServicesClient =
KieServicesFactory.newKieServicesClient(configuration);

    RulesMinaServicesClient rulesClient =
kieServicesClient.getServicesClient(RulesMinaServicesClient.class);

    return rulesClient;
}
```

MINA トランスポートを使用して Process Server で操作を呼び出す設定例

```
RulesMinaServicesClient rulesClient = buildClient();

List<Command<?>> commands = new ArrayList<Command<?>>();
BatchExecutionCommand executionCommand =
commandsFactory.newBatchExecution(commands, "defaultKieSession");

Person person = new Person();
person.setName("mary");
commands.add(commandsFactory.newInsert(person, "person"));
commands.add(commandsFactory.newFireAllRules("fired"));

ServiceResponse<String> response = rulesClient.executeCommands(containerId,
executionCommand);
Assert.assertNotNull(response);

Assert.assertEquals(ResponseType.SUCCESS, response.getType());

String data = response.getResult();

Marshaller marshaller = MarshallerFactory.getMarshaller(extraClasses,
MarshallingFormat.JSON, this.getClass().getClassLoader());
```

```
ExecutionResultImpl results = marshaller.unmarshall(data, ExecutionResultImpl.class);
Assert.assertNotNull(results);

Object personResult = results.getValue("person");
Assert.assertTrue(personResult instanceof Person);

Assert.assertEquals("mary", ((Person) personResult).getName());
Assert.assertEquals("JBoss Community", ((Person) personResult).getAddress());
Assert.assertEquals(true, ((Person) personResult).isRegistered());
```

第24章 関連情報

- 『Red Hat JBoss EAP 7.2 への Red Hat Process Automation Manager のインストールおよび設定』
- 『Red Hat Process Automation Manager インストールの計画』
- 『Red Hat JBoss EAP 7.2 への Red Hat Process Automation Manager のインストールおよび設定』
- 『Red Hat OpenShift Container Platform への Red Hat Process Automation Manager イミュータブルサーバー環境のデプロイメント』
- 『Red Hat OpenShift Container Platform への Red Hat Process Automation Manager オーサリング環境のデプロイ』
- Red Hat OpenShift Container Platform への Red Hat Process Automation Manager フリーフォーム管理サーバー環境のデプロイ
- 『Automation Broker を使用した Red Hat OpenShift Container Platform への Red Hat Process Automation Manager 環境のデプロイメント』
- 『Operator を使用した Red Hat OpenShift Container Platform への Red Hat Process Automation Manager 環境のデプロイメント』

付録A バージョン情報

本書の最終更新日： 2021 年 7 月 19 日（月）