Red Hat Process Automation Manager 7.11

Red Hat Process Automation Manager での Red Hat ビルドの Kogito の使用

Last Updated: 2023-04-08

Red Hat Process Automation Manager 7.11 Red Hat Process Automation Manager での Red Hat ビルドの Kogito の使用

法律上の通知

概要

本書では、クラウドネイティブのビジネスアプリケーションを構築するために Red Hat Process Automation Manager で Red Hat ビルドの Kogito を使い始める方法を説明します。

目次

はじめに

ビジネスデシジョンおよびプロセスの開発者は、Red Hat ビルドの Kogito を使用して、ビジネスドメインおよびツールを適応するクラウドネイティブアプリケーションを構築できます。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みにより、これらの変更は今後の複数のリリースに対して段階的に実施されます。詳細は、**弊社の CTO である Chris Wright のメッセージ**を参照してください。

パート I. RED HAT ビルドの KOGITO マイクロサービスの使用

ビジネスデシジョンの開発者は、Red Hat ビルドの Kogito ビジネス自動化を使用して、DMN (Decision Model and Notation) モデル、DRL (Drools Rule Language) ルール、PMML (Predictive Model Markup Language) または 3 つのメソッドすべての組み合わせを使用して、デシジョンサービスを開発できます。

前提条件

- JDK 11 以降がインストールされている。
- Apache Maven 3.6.2 以降がインストールされている。

第1章 RED HAT PROCESS AUTOMATION MANAGER での RED HAT ビルドの KOGITO マイクロサービス

Red Hat ビルドの Kogito は、クラウド対応のビジネスアプリケーションを構築するクラウドネイティブのビジネス自動化テクノロジーです。名前 **Kogito** は、"Cogito, ergo sum" ("I think, therefore I am") で使用されるラテン語 Cogito を由来とし、**['koːˌd͡ʒi.to]** (**コジート**) と発音します。文字 **K** は Kubernetes を指し、Red Hat Process Automation のターゲットクラウドプラットフォームとして Red Hat OpenShift Container Platform のベースと、Red Hat ビルドの Kogito が起点となるオープンソースビジネス自動化プロジェクトに対応します。

Red Hat Process Automation Manager の Kogito の Red Hat ビルドはハイブリッドクラウド環境向けに最適化されており、ドメインおよびツールのニーズに順応します。Red Hat ビルドの Kogito マイクロサービスは主に、独自のドメイン固有のクラウドネイティブセットにデシジョンのセットを形成しやすくすることを目的とします。

重要

Red Hat Process Automation Manager 7.11 バージョンでは、Red Hat ビルドの Kogito のサポートは Decision Model and Notation (DMN)、Drools Rule Language (DRL)、Predictive Model Markup Language (PMML) などのデシジョンサービスに制限されます。このサポートは、今後のリリースで Business Process Modeling Notation (BPMN) にも展開され、向上されます。

Red Hat ビルドの Kogito を使用する場合は、クラウドネイティブアプリケーションを独立したドメイン固有のマイクロサービスのセットとしてビルドし、ビジネスバリューを実現します。ターゲットの動作の記述に使用するデシジョンは、作成するマイクロサービスの一部として実行されます。作成されるマイクロサービスは、集約オーケストレーションサービスなしで、非常に分散化されており、スケーラブルです。また、マイクロサービスが使用するランタイムは要件に合わせて最適化されています。

ビジネスルール開発者は、Red Hat Process Automation Manager で Kogito マイクロサービスの Red Hat ビルドを使用して、ビジネスドメインおよびツールに適合するクラウドネイティブアプリケーションを構築できます。

1.1. CLOUD-FIRST の優先度

Red Hat ビルドの Kogito マイクロサービスは、クラウドインフラストラクチャー上で実行され、スケーリングできるように設計されています。Red Hat ビルドの Quarkus などの最新のクラウドベースのテクノロジーが含まれる Red Hat Process Automation Manager で Red Hat ビルドの Kogito マイクロサービスを使用し、Red Hat OpenShift Container Platform などのコンテナーアプリケーションプラットフォームでの起動時間および即時スケーリングを向上できます。

たとえば、Red Hat ビルドの Kogito マイクロサービスは以下のテクノロジーと互換性があります。

- **Red Hat OpenShift Container Platform** は Kubernetes をベースとしており、コンテナー化されたアプリケーションをビルドし、管理するためのターゲットプラットフォームです。
- **Red Hat ビルドの Quarkus** は、Red Hat ビルドの Kogito マイクロサービスを使用してアプリケーションをビルドするために使用できる Kubernetes のネイティブ Java スタックです。
- **Spring Boot** は、Red Hat Process Automation Manager で Spring Framework を設定するために使用できるアプリケーションフレームワークです。

1.2. RED HAT ビルドの QUARKUS および SPRING BOOT での RED HAT ビルドの KOGITO マイクロサービス

Red Hat ビルドの Kogito マイクロサービスがサポートする主な Java フレームワークは、Red Hat ビルドの Quarkus および Spring Boot です。

Red Hat ビルドの Quarkus は、特に OpenJDK HotSpot などの Java 仮想マシン (JVM) 向けの、コンテナーファーストアプローチを使用した Kubernetes ネイティブ Java フレームワークです。Red Hat ビルドの Quarkus は、Java アプリケーションとコンテナーイメージフットプリントの両方のサイズを小さくし、Java プログラミングのワークロードの一部を以前の世代から排除し、それらのイメージの実行に必要なメモリー量を減らすことで、Java 専用の Kubernetes を最適化します。

Red Hat ビルドの Kogito マイクロサービスにとっては、Red Hat ビルドの Quarkus は高度なデバッグを行う開発モードでのライブリロードなど、Kubernetes の互換性および強化された開発者機能を最適化するための推奨フレームワークです。

Spring Boot は、スタンドアロンで実稼働対応の Spring アプリケーションをビルドするための Java ベースのフレームワークです。Spring Boot では、Spring 設定すべてを設定せずに最小設定で Spring アプリケーションを開発できます。

Kogito マイクロサービスの Red Hat ビルドを使用する場合に、Spring Boot は既存の Spring Framework 環境で Red Hat Process Automation Manager を使用する必要のある開発者向けにサポートされます。

第2章 RED HAT ビルドの KOGITO マイクロサービスの DMN モデラー

Red Hat Process Automation Manager は、グラフィカルモデラーを使用して Red Hat ビルドの Kogito マイクロサービスに Decision Model and Notation (DMN) デシジョンモデルを設計するために使用できる拡張機能またはアプリケーションを提供します。

以下の DMN モデラーは、Business Modeler Hub デスクトップアプリケーションで利用できます。

- **VSCode 拡張機能**: Visual Studio Code(VSCode) で DMN モデルを表示し、設計できます。VSCode 拡張機能には VSCode 1.46.0 以降が必要です。
  VSCode に直接拡張機能をインストールするには、VSCode で **Extensions** メニューオプションを選択して、**Red Hat Business Automation Bundle**拡張を検索し、インストールします。

- **Business Modeler スタンドアロンエディター**: Web アプリケーションに組み込まれた DMN モデルを表示して、作成できます。必要なファイルをダウンロードするには、Kogito ツールリポジトリー から NPM アーティファクトを使用するか、**https://kiegroup.github.io/kogito-online/standalone/dmn/index.js** の DMN スタンドアロンエディターライブラリーに直接 JavaScript ファイルをダウンロードできます。

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

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

重要

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

前提条件

- VSCode の最新の安定版がインストールされている。

手順

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

2. **Red Hat Business Automation Bundle**拡張機能が VSCode に表示されたら、これを選択して **Install** をクリックします。

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

VSCode 拡張バンドルをインストールした後、VSCode で開くか作成するすべての **.dmn** ファイル、**.bpmn** ファイル、または **.bpmn2** ファイルがグラフィカルモデルとして自動的に表示されます。

さらに、開くまたは作成する **.scesim** ファイルは、ビジネスデシジョンの機能をテストするテーブルテストシナリオモデルとして自動的に表示されます。

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

注記

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

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

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

スタンドアロンのエディターは、以下の 3 つの方法でインストールできます。

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

手順

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

   a. JavaScript ファイルをダウンロードします。

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

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

   DMN エディターの HTML ページのスクリプトタグ

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

   BPMN エディターの HTML ページのスクリプトタグ

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

   NPM パッケージの使用: この方法の場合は、以下の手順に従います。

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

   NPM パッケージの追加

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

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

各エディターのインポート

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

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

DMN スタンドアロンのエディターを開く

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

エディター API で以下のパラメーターを使用します。

表2.1 パラメーターの例

パラメーター	説明
container	エディターが追加される HTML 要素。
initialContent	DMN モデルのコンテンツへの Pomise。以下の例のように、このパラメーターは空にすることができます。 • **Promise.resolve("")** • **Promise.resolve("<DIAGRAM_CONTENT_DIRECTLY_HERE>")** • **fetch("MyDmnModel.dmn").then(content ⇒ content.text())**
readonly (任意)	エディターでの変更を許可します。コンテンツの編集を許可する場合は **false** (デフォルト)、エディターで読み取り専用モードの場合は **true** に設定します。

パラメーター	説明
origin (任意)	リポジトリーの起点。デフォルト値は **window.location.origin** です。
resources (任意)	エディターのリソースのマッピング。たとえば、このパラメーターを使用して、BPMN エディターの DMN エディターまたは作業アイテム定義に含まれるモデルを提供します。マップの各エントリーには、リソース名と、**content-type** (**text** または **binary**) および **content** (**initialContent** パラメーターと同様) で設定されるオブジェクトが含まれています。

返されるオブジェクトには、エディターの操作に必要なメソッドが含まれます。

表2.2 返されたオブジェクトメソッド

メソッド	説明
getContent(): Promise<string>	エディターのコンテンツを含む promise を返します。
setContent(content: string): void	エディターの内容を設定します。
getPreview(): Promise<string>	現在のダイアグラムの SVG 文字列が含まれる promise を返します。
subscribeToContentChanges(callback: (isDirty: boolean) ⇒ void): (isDirty: boolean) ⇒ void	エディターでコンテンツを変更し、サブスクライブ解除に使用されるのと同じコールバックを返す際に呼び出されるコールバックを設定します。
unsubscribeToContentChanges(callback: (isDirty: boolean) ⇒ void): void	エディターでコンテンツが変更される際に渡されたコールバックのサブスクライブを解除します。
markAsSaved(): void	エディターの内容が保存されることを示すエディターの状態をリセットします。また、コンテンツの変更に関連するサブスクライブされたコールバックをアクティベートします。
undo(): void	エディターの最後の変更を元に戻します。また、コンテンツの変更に関連するサブスクライブされたコールバックをアクティベートします。
redo(): void	エディターで、最後に元に戻した変更をやり直します。また、コンテンツの変更に関連するサブスクライブされたコールバックをアクティベートします。
close(): void	エディターを終了します。

メソッド	説明
getElementPosition(selector: string): Promise<Rect>	要素をキャンバスまたはビデオコンポーネント内に置いた場合に、標準のクエリーセレクターを拡張する方法を提供します。**selector** パラメーターは、**Canvas:::MySquare**、**Video:::PresenterHand** などの **<PROVIDER>:::<SELECT>** 形式に従う必要があります。このメソッドは、要素の位置を表す **Rect** を返します。
envelopeApi: MessageBusClientApi<KogitoEditorEnvelopeApi>	これは高度なエディター API です。高度なエディター API の詳細は、MessageBusClientApi および KogitoEditorEnvelopeApi を参照してください。

第3章 RED HAT ビルドの KOGITO マイクロサービスの MAVEN プロジェクトの作成

Red Hat ビルドの Kogito マイクロサービスの開発を開始する前に、アセットおよびその他の関連リソースをビルドできる Maven プロジェクトを作成する必要があります。

手順

1. コマンドターミナルで、新しプロジェクトを保存するローカルディレクトリーに移動します。
2. 以下のコマンドを入力して、定義したディレクトリーにプロジェクトを生成します。

Red Hat ビルドの Quarkus の場合

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.kie.kogito \
    -DarchetypeArtifactId=kogito-quarkus-dm-archetype \
    -DgroupId=org.acme -DartifactId=sample-kogito \
    -DarchetypeVersion=1.5.0.redhat-00002 \
    -Dversion=1.0-SNAPSHOT
```

Spring Boot の場合

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.kie.kogito \
    -DarchetypeArtifactId=kogito-springboot-dm-archetype \
    -DgroupId=org.acme -DartifactId=sample-kogito \
    -DarchetypeVersion=1.5.0.redhat-00002 \
    -Dversion=1.0-SNAPSHOT
```

このコマンドは、**sample-kogito** の Maven プロジェクトを生成し、ビジネスアプリケーション自動化用のアプリケーション準備に必要な依存関係および設定の拡張をインポートします。

プロジェクトの PMML 実行を有効にする場合は、Red Hat ビルドの Kogito マイクロサービスが含まれる Maven プロジェクトの **pom.xml** ファイルに以下の依存関係を追加します。

PMML 実行を有効にするための依存関係

```
<dependency>
  <groupId>org.kie.kogito</groupId>
  <artifactId>kogito-pmml</artifactId>
</dependency>
<dependency>
  <groupId>org.jpmml</groupId>
  <artifactId>pmml-model</artifactId>
</dependency>
```

Red Hat ビルドの Quarkus でアプリケーションを OpenShift で実行する予定の場合には、以下の例のように liveness および readiness Probe の **smallrye-health** 拡張もインポートする必要があります。

OpenShift 上の Red Hat ビルドの Quarkus アプリケーションの SmallRye Health 拡張

```
$ mvn quarkus:add-extension -Dextensions="smallrye-health"
```

このコマンドにより、Quarkus 上の Red Hat ビルドの Red Hat Process Automation Manager プロジェクトの **pom.xml** ファイルに、以下の依存関係を生成します。

OpenShift 上の Red Hat ビルドの Quarkus アプリケーションの SmallRye Health 依存関係

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-health</artifactId>
</dependency>
```

3. VSCode IDE でプロジェクトを開くか、インポートしてコンテンツを表示します。

第4章 RED HAT ビルドの KOGITO マイクロサービスのあるアプリケーションの例

Red Hat ビルドの Kogito マイクロサービスでは、**rhpam-7.11.0-decision-services-quickstarts.zip** ファイルにサンプルアプリケーションが含まれます。これらのサンプルアプリケーションには、独自のアプリケーションの開発に役立つ Red Hat ビルドの Quarkus または Spring Boot のさまざまなタイプのサービスが含まれます。このサービスは、1 つ以上の Decision Model and Notation (DMN) デシジョンモデル、Drools Rule Language (DRL) ルールユニット、Predictive Model Markup Language (PMML) モデル、または Java クラスを使用してサービスロジックを定義します。

アプリケーションの例および使用方法については、関連するアプリケーションフォルダーの **README** ファイルを参照してください。

注記

ローカル環境でサンプルを実行する場合は、関連するアプリケーションフォルダーの **README** ファイルに記載されている要件に環境が対応していることを確認します。また、これには Red Hat ビルドの Quarkus、Spring Boot、および docker-compose の設定に必要なネットワークポートを利用できるようにする必要がある場合があります。

以下は、Red Hat ビルドの Kogito マイクロサービスで提供される例の一部です。

デシジョンサービス

- **dmn-quarkus-example** および **dmn-springboot-example**: DMN を使用するデシジョンサービス (Red Hat ビルドの Quarkus または Spring Boot) を使用して、交通違反をもとに運転手の罰則や免許停止を判断します。
- **rules-quarkus-helloworld**: DRL ルールユニットが 1 つある、Red Hat ビルドの Quarkus 上の Hello World のデシジョンサービス。
- **RuleUnit-quarkus-example** および **ruleunit-springboot-example**: ルールユニットで DRL を使用して、ローン申し込みを検証するデシジョンサービス (Red Hat ビルドの Quarkus または Spring Boot) を使用し、REST 操作を公開してアプリケーションの状態を表示します。
- **dmn-pmml-quarkus-example** および **dmn-pmml-springboot-example**: DMN および PMML を使用するデシジョンサービス (Red Hat ビルドの Quarkus または Spring Boot) を使用して、交通違反をもとに運転手の罰則や免許停止を判断します。
- **dmn-drools-quarkus-metrics** および **dmn-drools-springboot-metrics**: Red Hat ビルドの Kogito のランタイムメトリクス監視機能を有効化および消費するデシジョンサービス (Red Hat ビルドの Quarkus または Spring Boot 上)。
- **PMML-quarkus-example** および **pmml-springboot-example**: PMML を使用するデシジョンサービス (Red Hat ビルドの Quarkus または Spring Boot 上)。

Red Hat Process Automation Manager での DMN、DRL、および PMML モデルの使用に関する詳細は **DMN モデルを使用したデシジョンサービスの作成**、**DRL ルールを使用したデシジョンサービスの作成**、および **PMML モデルでのデシジョンサービスの作成** を参照してください。

第5章 DMN を使用した RED HAT ビルドの KOGITO マイクロサービスのアプリケーションロジックの設計

プロジェクトを作成したら、プロジェクトの **src/main/resources** ディレクトリーに、Decision Model and Notation (DMN) デシジョンモデルと Drools Rule Language (DRL) ビジネスルールを作成またはインポートできます。また、Java サービスとして動作するプロジェクトの **src/main/java** ディレクトリーに Java クラスを含めるか、デシジョンから呼び出す実装を指定することもできます。

この手順の例では、基本的な Red Hat ビルドの Kogito マイクロサービスが REST エンドポイント **/persons** を提供します。このエンドポイントは、**PersonDecisions.dmn** DMN モデルのサンプルをもとに自動生成され、処理されたデータをベースに意思決定を行います。

ビジネスデシジョンには、Red Hat Process Automation Manager サービスのデシジョンロジックが含まれます。DMN モデルや DRL ルールなど、ビジネスルールやデシジョンをさまざまな方法で定義できます。この手順の例では、DMN モデルを使用します。

前提条件

- プロジェクトを作成している。Maven プロジェクトの作成方法は *3章Red Hat ビルドの Kogito マイクロサービスの Maven プロジェクトの作成* を参照してください。

手順

1. Red Hat Process Automation Manager サービス用に生成した Maven プロジェクトで **src/main/java/org/acme** フォルダーに移動し、以下の **Person.java** ファイルを追加します。

   person Java オブジェクトの例

```
package org.acme;

import java.io.Serializable;

public class Person {

 private String name;
 private int age;
 private boolean adult;

 public String getName() {
  return name;
 }

 public void setName(String name) {
  this.name = name;
 }

 public int getAge() {
  return age;
 }

 public void setAge(int age) {
  this.age = age;
 }

 public boolean isAdult() {
```

```
 return adult;
}

public void setAdult(boolean adult) {
 this.adult = adult;
}

@Override
public String toString() {
 return "Person [name=" + name + ", age=" + age + ", adult=" + adult + "]";
}

}
```

この例では、Java オブジェクトの名前、年齢、成人ステータスを設定して取得します。

2. **src/main/resources** フォルダーに移動し、以下の **PersonDecisions.dmn** DMN デシジョンモデルを追加します。

図5.1 PersonDecisions の DMN 意思決定要件ダイアグラム (DRD) 例

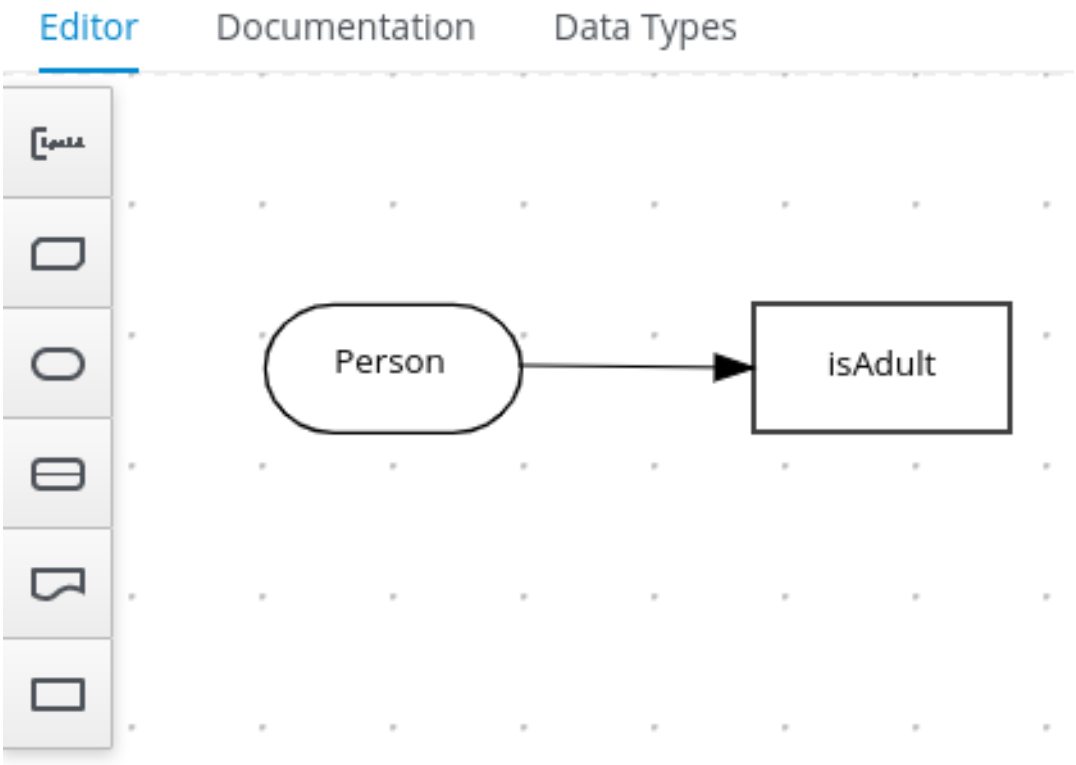


図5.2 isAdult デシジョン の DMN ボックス式の例

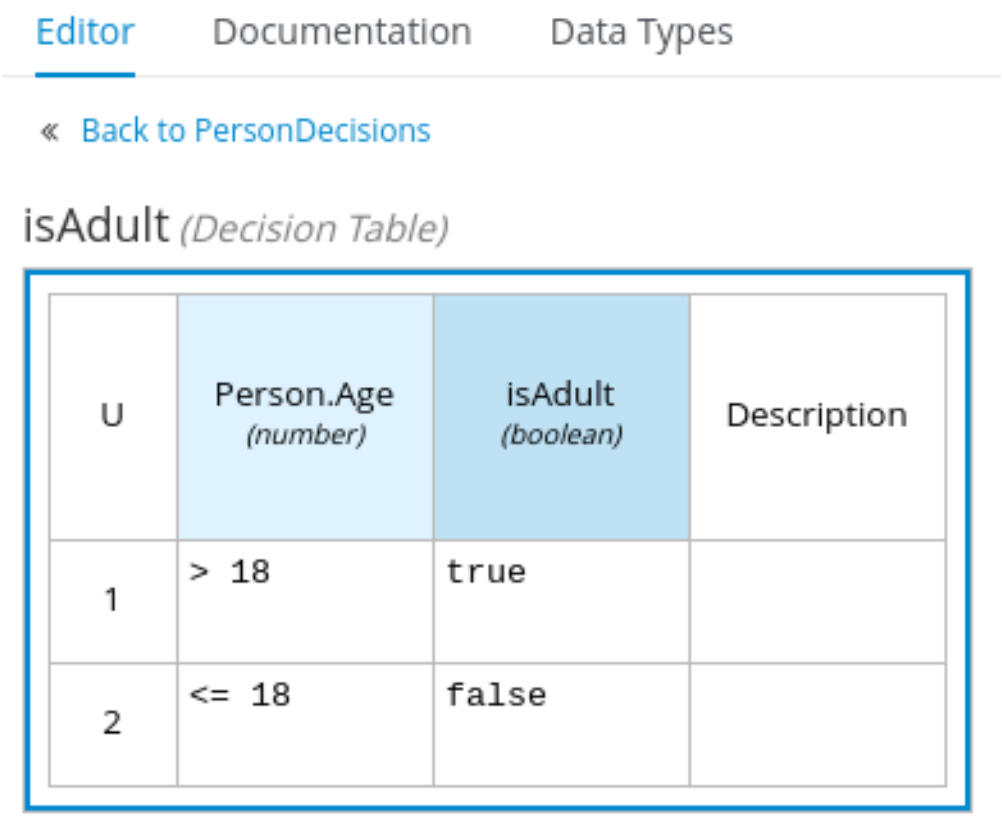
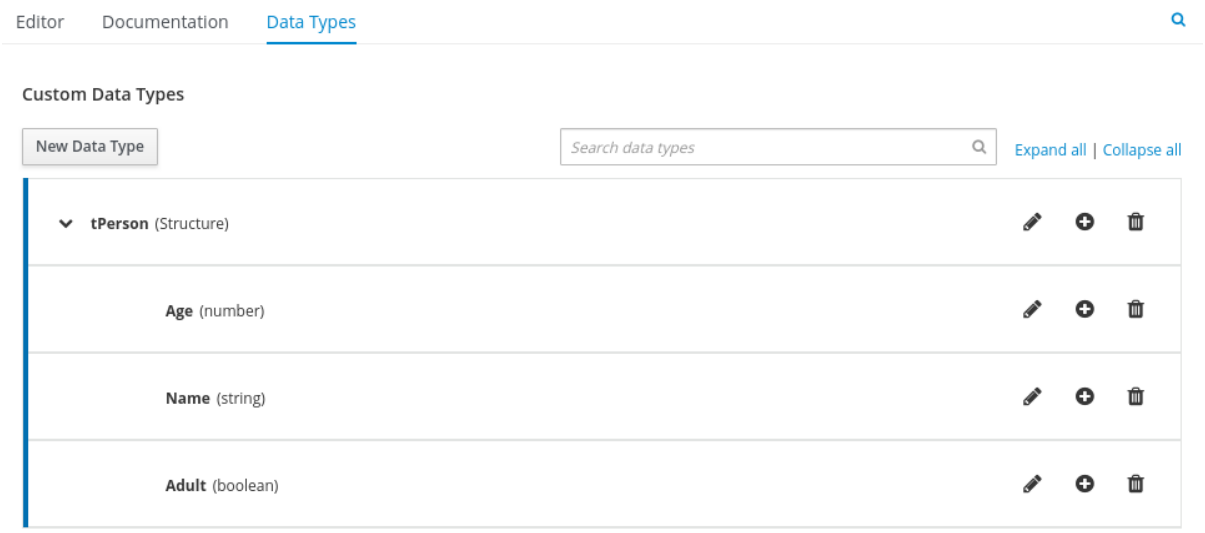


図5.3 DMN データ型の例



この DMN モデルは、基本的な DMN 入力ノードと、カスタムの構造化データ型で DMN デシジョンテーブルが定義したデシジョンノードで設定されます。

VSCode では、**Red Hat Business Automation Bundle**VSCode 拡張機能を追加して、DMN モデラーで意思決定要件ダイアグラム (DRD)、ボックス式、およびデータタイプを設計できます。

この DMN モデルの例を迅速に作成するには、以下の **PersonDecisions.dmn** ファイルの内容をコピーします。

DMN ファイルの例

```
<dmn:definitions xmlns:dmn="http://www.omg.org/spec/DMN/20180521/MODEL/"
xmlns="https://kiegroup.org/dmn/_52CEF9FD-9943-4A89-96D5-6F66810CA4C1"
xmlns:di="http://www.omg.org/spec/DMN/20180521/DI/"
xmlns:kie="http://www.drools.org/kie/dmn/1.2"
xmlns:dmndi="http://www.omg.org/spec/DMN/20180521/DMNDI/"
xmlns:dc="http://www.omg.org/spec/DMN/20180521/DC/"
xmlns:feel="http://www.omg.org/spec/DMN/20180521/FEEL/" id="_84B432F5-87E7-43B1-
9101-1BAFE3D18FC5" name="PersonDecisions"
typeLanguage="http://www.omg.org/spec/DMN/20180521/FEEL/"
namespace="https://kiegroup.org/dmn/_52CEF9FD-9943-4A89-96D5-6F66810CA4C1">
  <dmn:extensionElements/>
  <dmn:itemDefinition id="_DEF2C3A7-F3A9-4ABA-8D0A-C823E4EB43AB" name="tPerson"
isCollection="false">
    <dmn:itemComponent id="_DB46DB27-0752-433F-ABE3-FC9E3BDECC97" name="Age"
isCollection="false">
      <dmn:typeRef>number</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_8C6D865F-E9C8-43B0-AB4D-3F2075A4ECA6" name="Name"
isCollection="false">
      <dmn:typeRef>string</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_9033704B-4E1C-42D3-AC5E-0D94107303A1" name="Adult"
isCollection="false">
      <dmn:typeRef>boolean</dmn:typeRef>
    </dmn:itemComponent>
  </dmn:itemDefinition>
  <dmn:inputData id="_F9685B74-0C69-4982-B3B6-B04A14D79EDB" name="Person">
    <dmn:extensionElements/>
    <dmn:variable id="_0E345A3C-BB1F-4FB2-B00F-C5691FD1D36C" name="Person"
typeRef="tPerson"/>
  </dmn:inputData>
  <dmn:decision id="_0D2BD7A9-ACA1-49BE-97AD-19699E0C9852" name="isAdult">
    <dmn:extensionElements/>
    <dmn:variable id="_54CD509F-452F-40E5-941C-AFB2667D4D45" name="isAdult"
typeRef="boolean"/>
    <dmn:informationRequirement id="_2F819B03-36B7-4DEB-AED6-2B46AE3ADB75">
      <dmn:requiredInput href="#_F9685B74-0C69-4982-B3B6-B04A14D79EDB"/>
    </dmn:informationRequirement>
    <dmn:decisionTable id="_58370567-05DE-4EC0-AC2D-A23803C1EAAE"
hitPolicy="UNIQUE" preferredOrientation="Rule-as-Row">
      <dmn:input id="_ADEF36CD-286A-454A-ABD8-9CF96014021B">
        <dmn:inputExpression id="_4930C2E5-7401-46DD-8329-EAC523BFA492"
typeRef="number">
          <dmn:text>Person.Age</dmn:text>
        </dmn:inputExpression>
      </dmn:input>
      <dmn:output id="_9867E9A3-CBF6-4D66-9804-D2206F6B4F86" typeRef="boolean"/>
      <dmn:rule id="_59D6BFF0-35B4-4B7E-8D7B-E31CB0DB8242">
        <dmn:inputEntry id="_7DC55D63-234F-497B-A12A-93DA358C0136">
          <dmn:text>&amp;gt; 18</dmn:text>
        </dmn:inputEntry>
        <dmn:outputEntry id="_B3BB5B97-05B9-464A-AB39-58A33A9C7C00">
          <dmn:text>true</dmn:text>
        </dmn:outputEntry>
      </dmn:rule>
      <dmn:rule id="_8FCD63FE-8AD8-4F56-AD12-923E87AFD1B1">
```

```
      <dmn:inputEntry id="_B4EF7F13-E486-46CB-B14E-1D21647258D9">
        <dmn:text>&lt;= 18</dmn:text>
      </dmn:inputEntry>
      <dmn:outputEntry id="_F3A9EC8E-A96B-42A0-BF87-9FB1F2FDB15A">
        <dmn:text>false</dmn:text>
      </dmn:outputEntry>
    </dmn:rule>
  </dmn:decisionTable>
 </dmn:decision>
 <dmndi:DMNDI>
  <dmndi:DMNDiagram>
   <di:extension>
    <kie:ComponentsWidthsExtension>
     <kie:ComponentWidths dmnElementRef="_58370567-05DE-4EC0-AC2D-
A23803C1EAAE">
      <kie:width>50</kie:width>
      <kie:width>100</kie:width>
      <kie:width>100</kie:width>
      <kie:width>100</kie:width>
     </kie:ComponentWidths>
    </kie:ComponentsWidthsExtension>
   </di:extension>
   <dmndi:DMNShape id="dmnshape-_F9685B74-0C69-4982-B3B6-B04A14D79EDB"
dmnElementRef="_F9685B74-0C69-4982-B3B6-B04A14D79EDB" isCollapsed="false">
    <dmndi:DMNStyle>
     <dmndi:FillColor red="255" green="255" blue="255"/>
     <dmndi:StrokeColor red="0" green="0" blue="0"/>
     <dmndi:FontColor red="0" green="0" blue="0"/>
    </dmndi:DMNStyle>
    <dc:Bounds x="404" y="464" width="100" height="50"/>
    <dmndi:DMNLabel/>
   </dmndi:DMNShape>
   <dmndi:DMNShape id="dmnshape-_0D2BD7A9-ACA1-49BE-97AD-19699E0C9852"
dmnElementRef="_0D2BD7A9-ACA1-49BE-97AD-19699E0C9852" isCollapsed="false">
    <dmndi:DMNStyle>
     <dmndi:FillColor red="255" green="255" blue="255"/>
     <dmndi:StrokeColor red="0" green="0" blue="0"/>
     <dmndi:FontColor red="0" green="0" blue="0"/>
    </dmndi:DMNStyle>
    <dc:Bounds x="404" y="311" width="100" height="50"/>
    <dmndi:DMNLabel/>
   </dmndi:DMNShape>
   <dmndi:DMNEdge id="dmnedge-_2F819B03-36B7-4DEB-AED6-2B46AE3ADB75"
dmnElementRef="_2F819B03-36B7-4DEB-AED6-2B46AE3ADB75">
    <di:waypoint x="504" y="489"/>
    <di:waypoint x="404" y="336"/>
   </dmndi:DMNEdge>
  </dmndi:DMNDiagram>
 </dmndi:DMNDI>
</dmn:definitions>
```

DMN モデラーを使用して VSCode でこの DMN モデルのサンプルを作成するには、以下の手順に従います。

a. 空の **PersonDecisions.dmn** ファイルを開き、DMN モデラーの右上隅にある **Properties** アイコンをクリックし、DMN モデル **Name** が **PersonDecisions** に設定されていることを確認します。

b. 左側のパレットで **DMN Input Data** を選択し、ノードをキャンバスにドラッグしてから、ノードをダブルクリックして **Person** という名前を付けます。

c. 左側のパレットで **DMN Decision** ノードをキャンバスにドラッグし、ノードをダブルクリックして 名前を **isAdult** に入力ノードからリンクします。

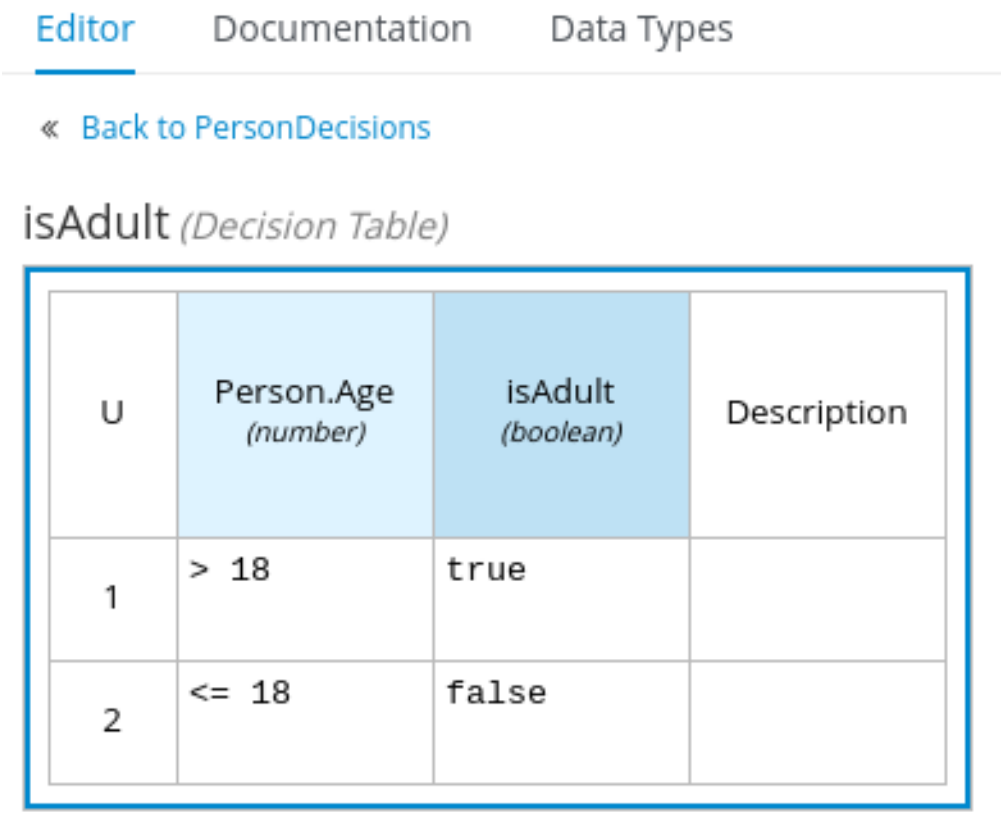
d. ノードオプションを表示するデシジョンノードを選択して **Edit** アイコンをクリックし、DMN ボックス式エディターを開き、ノードのデシジョンロジックを定義します。

e. **undefined expression** フィールドをクリックし、**Decision Table** を選択します。

f. デシジョンテーブルの左上隅をクリックして、ヒットポリシーを **Unique** に設定します。

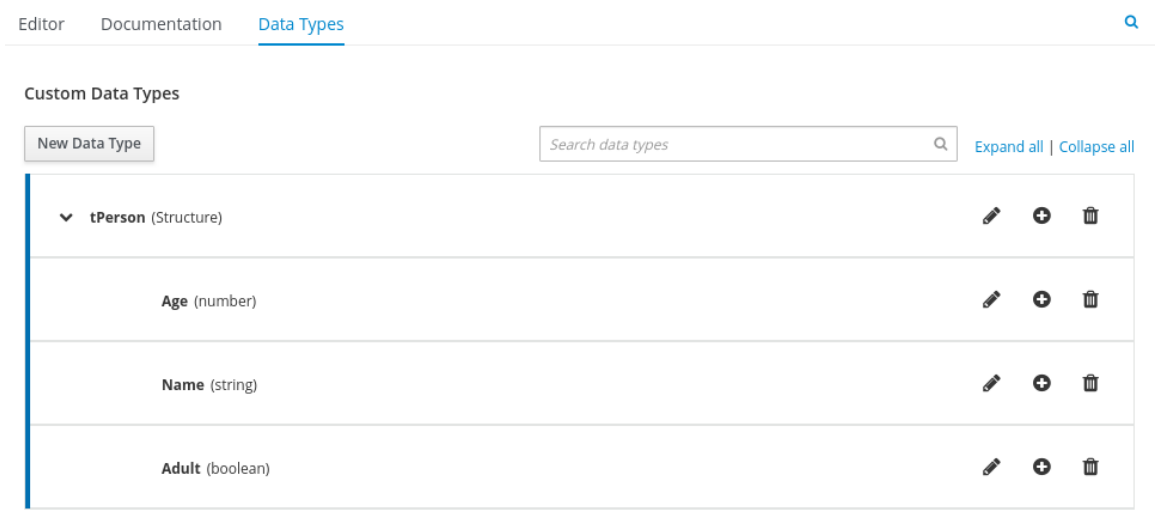
g. タイプが **number** の入力ソース **Person.Age** で、年齢制限を、タイプが **boolean** の出力ターゲット **isAdult** で成人ステータスを判断できるように、入出力コラムを設定します。

図5.4 デシジョンが isAdult の DMN デシジョンテーブルの例



h. 右上のタブで **Data Types** タブを選択して、以下の **tPerson** 構造のデータ型とネスト化データタイプを追加します。

図5.5 DMN データ型の例



i. データタイプを定義したら、**Editor** タブを選択して、DMN モデラーキャンバスに戻ります。

j. **Person** 入力ノードを選択し、**Properties** アイコンをクリックし、**Information item** の下にある **Data type** を **tPerson** に設定します。

k. **isAdult** デシジョンノードを 選択し、**Properties** アイコンをクリックして、**Information 項目** で **Data type** がまだ **boolean** に設定されていることを確認します。デシジョンテーブルの作成時に、このデータ型を設定してください。

l. DMN デシジョンサービスを保存します。

5.1. 別のデシジョンサービスとしての DRL ルールユニットの使用

また、ルールユニットとして実装した Drools Rule Language (DRL) ファイルを使用して、Decision Model and Notation (DMN) を使用する代わりに、このサンプルデシジョンサービスを定義することもできます。

DRL ルールユニットは、ルールのモジュールおよび、実行ユニットです。ルールユニットは、ルール実行のファクトタイプの宣言を使用して、一連のルールを収集します。また、ルールユニットは、各ルールグループに固有の namespace としても機能します。1 つのルールベースには、複数のルールユニットを含めることができます。通常、ユニットだけで自己完結できるように、ユニットの全ルールはユニット宣言と同じファイルに保存します。ルールユニットの作成方法は、**DRL ルールを使用したデシジョンサービスの作成** を参照してください。

前提条件

- プロジェクトを作成している。Maven プロジェクトの作成方法は *3章Red Hat ビルドの Kogito マイクロサービスの Maven プロジェクトの作成* を参照してください。

手順

1. DMN ファイルを使用する代わりに、サンプルプロジェクトの **src/main/resources** ディレクトリーに、以下の **PersonRules.drl** ファイルを追加します。

PersonRules DRL ファイルの例

```
package org.acme
```

```
unit PersonRules;

import org.acme.Person;

rule isAdult
 when
  $person: /person[ age > 18 ]
 then
    modify($person) {
     setAdult(true)
    };
 end

 query persons
  $p : /person[ adult ]
  end
```

このルールの例では、18 歳以上を成人として分類しているかどうかを判断します。また、ルールファイルは、ルールがルールユニット **PersonRules** に属することを宣言します。プロジェクトをビルドすると、ルールユニットが生成され、DRL ファイルに関連付けられます。

ルールは OOPath 表記を使用して条件も定義します。OOPath は、Xpath のオブジェクト指向の構文拡張で、コレクションおよびフィルター制約を処理しながら、関連要素間を移動します。

以下の例のように、従来のルールパターン構文を使用して、より明示的な形式で同じルール条件を再書き込みすることもできます。

従来の表記を使用した PersonRules DRL ファイルの例

```
package org.acme
unit PersonRules;

import org.acme.Person;

rule isAdult
 when
  $person: Person(age > 18) from person
 then
    modify($person) {
     setAdult(true)
    };
 end

 query persons
  $p : /person[ adult ]
  end
```

第6章 RED HAT ビルドの KOGITO マイクロサービスの実行

Red Hat ビルドの Kogito マイクロサービスのビジネスデシジョンの設計後に、以下のいずれかのモードで Red Hat ビルドの Quarkus または Spring Boot アプリケーションを実行できます。

- **開発モード**: ローカルテスト用。Red Hat ビルドの Quarkus の開発モードでは、実行中のアプリケーションでデシジョンをライブリロードし、詳細にわたるデバッグを行います。
- **JVM モード**: Java 仮想マシン (JVM) との互換性用。

手順

コマンドターミナルで、Red Hat ビルドの Kogito マイクロサービスが含まれるプロジェクトに移動し、選択した実行モードやアプリケーション環境に合わせて、以下のコマンドのいずれかを入力します。

- 開発モードの場合:

  Red Hat ビルドの Quarkus の場合

  ```
  $ mvn clean compile quarkus:dev
  ```

  Sprint Boot の場合

  ```
  $ mvn clean compile spring-boot:run
  ```

- JVM モードの場合:

  Red Hat ビルドの Quarkus および Spring Boot の場合

  ```
  $ mvn clean package
  $ java -jar target/sample-kogito-1.0-SNAPSHOT-runner.jar
  ```

第7章 実行中の RED HAT ビルドの KOGITO マイクロサービスとの対話

Red Hat ビルドの Kogito マイクロサービスの実行後、REST API 要求を送信し、アプリケーションとの対話やアプリケーションの設定方法に合わせて、マイクロサービスを実行できます。

この例では、**/persons** REST API エンドポイントをテストし、 **PersonDecisions.dmn** ファイル (または DRL ルールユニットを使用した場合には **PersonRules.drl** ファイルのルール) に意思決定を自動生成します。

この例では、アプリケーションに設定された REST クライアント、curl ユーティリティー、またはアプリケーションに設定された Swagger UI (例: **http://localhost:8080/q/swagger-ui** or **http://localhost:8080/swagger-ui.html**) を使用して、以下のコンポーネントで API 要求を送信します。

- URL: **http://localhost:8080/persons**
- HTTP ヘッダー: **POST** 要求のみ:
  - **accept**: **application/json**
  - **content-type**: **application/json**
- HTTP メソッド: **GET**、**POST**、または **DELETE**

成人 (JSON) を追加するための POST 要求ボディーの例

```
{
  "person": {
    "name": "John Quark",
    "age": 20
  }
}
```

成人を追加する curl コマンドの例

```
curl -X POST http://localhost:8080/persons -H 'content-type: application/json' -H 'accept: application/json' -d '{"person": {"name":"John Quark", "age": 20}}'
```

応答例 (JSON)

```
{
  "id": "3af806dd-8819-4734-a934-728f4c819682",
  "person": {
    "name": "John Quark",
    "age": 20,
    "adult": false
  },
  "isAdult": true
}
```

以下の手順では、便宜上 curl コマンドを使用します。

手順

実行中のアプリケーションから分離されているコマンドターミナルウィンドウで、Red Hat ビルドの Kogito マイクロサービスが含まれるプロジェクトに移動し、以下の curl コマンドのいずれかを JSON 要求と共に使用して、実行中のマイクロサービスと対話します。

注記

Spring Boot では、これらのサンプル要求を機能させるには、アプリケーションによる API エンドポイントの公開方法の変更が必要となる場合があります。詳細は、このチュートリアルで作成した Spring Boot プロジェクトのサンプルに含まれる **README** ファイルを参照してください。

- 成人を追加します。

  要求の例

  ```
  curl -X POST http://localhost:8080/persons -H 'content-type: application/json' -H 'accept: application/json' -d '{"person": {"name":"John Quark", "age": 20}}'
  ```

  応答例

  ```
  {"id":"3af806dd-8819-4734-a934-728f4c819682","person":{"name":"John Quark","age":20,"adult":false},"isAdult":true}
  ```

- 未成年を追加します。

  要求の例

  ```
  curl -X POST http://localhost:8080/persons -H 'content-type: application/json' -H 'accept: application/json' -d '{"person": {"name":"Jenny Quark", "age": 15}}'
  ```

  応答例

  ```
  {"id":"8eef502b-012b-4628-acb7-73418a089c08","person":{"name":"Jenny Quark","age":15,"adult":false},"isAdult":false}
  ```

- 返された UUID を使用して評価を完了します。

  要求の例

  ```
  curl -X POST http://localhost:8080/persons/8eef502b-012b-4628-acb7-73418a089c08/ChildrenHandling/cdec4241-d676-47de-8c55-4ee4f9598bac -H 'content-type: application/json' -H 'accept: application/json' -d '{}'
  ```

パート II. RED HAT OPENSHIFT CONTAINER PLATFORM での RED HAT ビルドの KOGITO マイクロサービスのデプロイ

ビジネスデシジョンおよびプロセスの開発者は、クラウド実装用に Red Hat OpenShift Container Platform に Red Hat ビルドの Kogito マイクロサービスをデプロイできます。RHPAM Kogito Operator は、デプロイメント手順の多数を自動化するか、デプロイメントプロセスを支援します。

前提条件

- Red Hat OpenShift Container Platform 4.6 または 4.7 がインストールされている。
- デプロイメントする OpenShift プロジェクトが作成されている。

第8章 RED HAT OPENSHIFT CONTAINER PLATFORM での RED HAT ビルドの KOGITO

Red Hat OpenShift Container Platform に対して、クラウド実装用に Red Hat ビルドの Kogito マイクロサービスをデプロできます。このアーキテクチャーでは、Red Hat ビルドの Kogito マイクロサービスは OpenShift Pod としてデプロイされ、個別にスケールアップおよびスケールダウンして、特定のサービスに必要な数のコンテナーを提供できます。

Red Hat Process Automation Manager では、**Red Hat Process Automation Manager Kogito Operator** が同梱されており、Red Hat ビルドの Kogito マイクロサービスを OpenShift にデプロイするのに役立ちます。この Operator は、デプロイメントプロセス関連の設定をサポートします。Operator は Operator SDK をベースとしており、多くのデプロイメント手順を自動化します。たとえば、アプリケーションが含まれる Git リポジトリーへのリンクを Operator に指定すると、Operator はソースからプロジェクトのビルドに必要なコンポーネントを自動的に設定し、生成されるサービスをデプロイします。

OpenShift Web コンソールで Red Hat Process Automation Manager Kogito Operator をインストールするには、左側のメニューで **Operators → OperatorHub** に移動し、**RHPAM Kogito Operator** を検索して選択し、最新の Operator バージョンをインストールします。

第9章 RHPAM KOGITO OPERATOR を使用した OPENSHIFT デプロイメントオプション

ビジネスアプリケーションの一部として Red Hat ビルドの Kogito マイクロサービスを作成したら、Red Hat OpenShift Container Platform Web コンソールを使用してマイクロサービスをデプロイすることができます。OpenShift Web コンソールの RHPAM Kogito Operator ページで、デプロイメントプロセスを進めます。

RHPAM Kogito Operator は、Red Hat OpenShift Container Platform で Red Hat ビルドの Kogito マイクロサービスをビルドしてデプロイするための以下のオプションをサポートします。

- Git ソースビルドおよびデプロイメント
- バイナリービルドおよびデプロイメント
- カスタムイメージのビルドおよびデプロイメント
- ファイルビルドおよびデプロイメント

9.1. GIT ソースビルドおよび OPENSHIFT WEB コンソールを使用した OPENSHIFT への RED HAT ビルドの KOGITO マイクロサービスのデプロイ

RHPAM Kogito Operator は以下のカスタムリソースを使用して、開発するマイクロサービス (ドメイン固有のマイクロサービス) をデプロイします。

- **KogitoBuild** は Git URL または他のソースを使用してアプリケーションをビルドし、ランタイムイメージを作成します。
- **KogitoRuntime** はランタイムイメージを起動し、要件に応じて設定します。

ほとんどのユースケースでは、以下の手順のように、標準のランタイムビルドおよびデプロイメント方法を使用して、Git リポジトリーソースから OpenShift に Red Hat ビルドの Kogito マイクロサービスをデプロイできます。

注記

Red Hat ビルドの Kogito マイクロサービスをローカルで開発またはテストする場合は、バイナリービルド、カスタムイメージビルド、またはファイルビルドオプションを使用して、Git リポジトリーではなく、ローカルソースからビルドおよびデプロイできます。

前提条件

- RHPAM Kogito Operator がインストールされている。
- Red Hat ビルドの Kogito マイクロサービスを使用するアプリケーションが OpenShift 環境からアクセス可能な Git リポジトリーにある。
- OpenShift Web コンソールにアクセスでき、**KogitoBuild** および **KogitoRuntime** の作成および編集に必要なパーミッションが設定されている。
- (Red Hat ビルドの Quarkus のみ) プロジェクトの **pom.xml** ファイルに **quarkus-smallrye-health** 拡張機能の依存関係が含まれる。この拡張機能は、OpenShift の Red Hat ビルドの Quarkus プロジェクトに必要な Liveness および Readiness Probe を有効する。

OpenShift 上の Red Hat ビルドの Quarkus アプリケーションの SmallRye Health 依存関係

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-health</artifactId>
</dependency>
```

手順

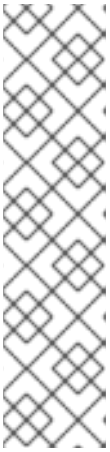
1. **Operators → Installed Operators** に移動し、**RHPAM Kogito Operator** を選択します。
2. Red Hat ビルドの Kogito ビルド定義を作成するには、Operator ページで **Kogito Build** タブを選択し、**Create KogitoBuild** をクリックします。
3. アプリケーションウィンドウで、**Form View** または **YAML View** を使用してビルド定義を設定します。
   少なくとも、以下の YAML ファイルのサンプルに示されるようにアプリケーション設定を定義します。

Red Hat ビルドの Kogito ビルドを使用した Red Hat ビルドの Quarkus アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this service
kind: KogitoBuild  # Application type
metadata:
  name: example-quarkus  # Application name
spec:
  type: RemoteSource
  gitSource:
    uri: 'https://github.com/kiegroup/kogito-examples'  # Git repository containing application (uses default branch)
    contextDir: dmn-quarkus-example  # Git folder location of application
```

Red Hat ビルドの Kogito ビルドを使用した Spring Boot アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this service
kind: KogitoBuild  # Application type
metadata:
  name: example-springboot  # Application name
spec:
  runtime: springboot
  type: RemoteSource
  gitSource:
    uri: 'https://github.com/kiegroup/kogito-examples'  # Git repository containing application (uses default branch)
    contextDir: dmn-springboot-example  # Git folder location of application
```



注記

内部 Maven リポジトリーを設定した場合は、Maven ミラーサービスとして使用でき、Red Hat ビルドの Kogito ビルド定義に Maven ミラー URL を指定して、ビルド時間を大幅に短縮できます。

```
spec:
  mavenMirrorURL: http://nexus3-nexus.apps-crc.testing/repository/maven-public/
```

内部 Maven リポジトリーの詳細は、Apache Maven ドキュメントを参照してください。

4. アプリケーションデータを定義したら、**Create** をクリックして Red Hat ビルドの Kogito ビルドを生成します。
アプリケーションが **Red Hat ビルドの KogitoBuilds** ページに一覧表示されます。アプリケーション名を選択して、アプリケーション設定および YAML の詳細を表示または変更できます。

5. Red Hat ビルドの Kogito マイクロサービス定義を作成するには、Operator ページで **Kogito Runtime** タブを選択し、**Create KogitoRuntime** をクリックします。

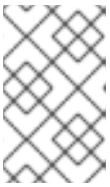
6. アプリケーションウィンドウで、**Form View** または **YAML View** を使用してマイクロサービス定義を設定します。
少なくとも、以下の YAML ファイルのサンプルに示されるようにアプリケーション設定を定義します。

Red Hat ビルドの Kogito マイクロサービスを使用した Red Hat ビルドの Quarkus アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoRuntime  # Application type
metadata:
  name: example-quarkus  # Application name
```

Red Hat ビルドの Kogito マイクロサービスを使用した Spring Boot アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoRuntime  # Application type
metadata:
  name: example-springboot  # Application name
spec:
  runtime: springboot
```



注記

この場合、アプリケーションは Git からビルドされ、KogitoRuntime を使用してデプロイされます。アプリケーション名が **KogitoBuild** および **KogitoRuntime** で同じである必要があります。

7. アプリケーションデータを定義したら、**Create** をクリックして Red Hat ビルドの Kogito マイクロサービスを生成します。

アプリケーションが Red Hat ビルドの Kogito マイクロサービスページに一覧表示されます。アプリケーション名を選択して、アプリケーション設定や YAML ファイルの内容を表示または変更できます。

8. Web コンソールの左側のメニューで **Builds → Builds** に移動して、アプリケーションのビルドのステータスを表示します。
特定のビルドを選択して、ビルドの詳細を表示できます。

注記

OpenShift デプロイメント用に作成するすべての Red Hat ビルドの Kogito マイクロサービスでは、従来のランタイムビルドと S2I (接尾辞 **-builder**) ビルドの 2 つのビルドが生成され、Web コンソールの **Builds** ページに一覧表示されます。S2I メカニズムは OpenShift ビルドでアプリケーションを構築し、このアプリケーションを次の OpenShift ビルドに渡してランタイムコンテナーイメージにパッケージ化します。Red Hat ビルドの Kogito S2I ビルド設定では、OpenShift プラットフォームの Git リポジトリーから直接プロジェクトをビルドできます。

9. アプリケーションのビルドが完了したら、**Workloads → Deployments** に移動して、アプリケーションのデプロイメント、Pod のステータスなどの情報を表示します。

10. Red Hat ビルドの Kogito マイクロサービスをデプロイしたら、Web コンソールの左側のメニューの **Networking → Routes** に移動し、デプロイされたアプリケーションへのアクセスリンクを表示します。
アプリケーション名を選択して、ルート設定を表示または変更できます。

    必要に応じてアプリケーションルートを使用すると、Red Hat ビルドの Kogito マイクロサービスはビジネス自動化ソリューションと統合できます。

9.2. バイナリービルドおよび OPENSHIFT WEB コンソールを使用した OPENSHIFT への RED HAT ビルドの KOGITO マイクロサービスのデプロイ

OpenShift ビルドには、長時間かかる可能性があります。OpenShift での Red Hat ビルドの Kogito マイクロサービスのビルドおよびデプロイを短時間で行う別の方法として、バイナリービルドを使用できます。

Operator は以下のカスタムリソースを使用して、開発するマイクロサービス (ドメイン固有のマイクロサービス) をデプロイします。

- **KogitoBuild** はアップロードしたアプリケーションを処理し、ランタイムイメージを作成します。
- **KogitoRuntime** はランタイムイメージを起動し、要件に応じて設定します。

前提条件

- RHPAM Kogito Operator がインストールされている。
- **oc** OpenShift CLI がインストールされており、関連する OpenShift クラスターにログインしている。**oc** のインストールおよびログインの手順については、OpenShift ドキュメント を参照してください。

- OpenShift Web コンソールにアクセスでき、**KogitoBuild** および **KogitoRuntime** の作成および編集に必要なパーミッションが設定されている。
- (Red Hat ビルドの Quarkus のみ) プロジェクトの **pom.xml** ファイルに **quarkus-smallrye-health** 拡張機能の依存関係が含まれる。この拡張機能は、OpenShift の Red Hat ビルドの Quarkus プロジェクトに必要な Liveness および Readiness Probe を有効する。

OpenShift 上の Red Hat ビルドの Quarkus アプリケーションの SmallRye Health 依存関係

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-health</artifactId>
</dependency>
```

手順

1. アプリケーションをローカルに構築します。
2. **Operators → Installed Operators** に移動し、**RHPAM Kogito Operator** を選択します。
3. Red Hat ビルドの Kogito ビルド定義を作成するには、Operator ページで **Kogito Build** タブを選択し、**Create KogitoBuild** をクリックします。
4. アプリケーションウィンドウで、**Form View** または **YAML View** を使用してビルド定義を設定します。
   少なくとも、以下の YAML ファイルのサンプルに示されるようにアプリケーション設定を定義します。

   Red Hat ビルドの Kogito ビルドを使用した Red Hat ビルドの Quarkus アプリケーションの YAML 定義の例

   ```
   apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this service
   kind: KogitoBuild  # Application type
   metadata:
     name: example-quarkus  # Application name
   spec:
     type: Binary
   ```

   Red Hat ビルドの Kogito ビルドを使用した Spring Boot アプリケーションの YAML 定義の例

   ```
   apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this service
   kind: KogitoBuild  # Application type
   metadata:
     name: example-springboot  # Application name
   spec:
     runtime: springboot
     type: Binary
   ```

5. アプリケーションデータを定義したら、**Create** をクリックして Red Hat ビルドの Kogito ビルドを生成します。
   アプリケーションが **Red Hat ビルドの KogitoBuilds** ページに一覧表示されます。アプリケーション名を選択して、アプリケーション設定および YAML の詳細を表示または変更できます。

6. 以下のコマンドを使用して、ビルドしたバイナリーをアップロードします。

```
$ oc start-build example-quarkus --from-dir=target/ -n namespace
```

- **from-dir** は、ビルドされたアプリケーションの **ターゲット** フォルダーパスと同じです。
- **Namespace** は、**KogitoBuild** の作成先の namespace に置き換えます。

7. Red Hat ビルドの Kogito マイクロサービス定義を作成するには、Operator ページで **Kogito Runtime** タブを選択し、**Create KogitoRuntime** をクリックします。

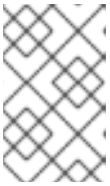
8. アプリケーションウィンドウで、**Form View** または **YAML View** を使用してマイクロサービス定義を設定します。
少なくとも、以下の YAML ファイルのサンプルに示されるようにアプリケーション設定を定義します。

Red Hat ビルドの Kogito マイクロサービスを使用した Red Hat ビルドの Quarkus アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoRuntime  # Application type
metadata:
  name: example-quarkus  # Application name
```

Red Hat ビルドの Kogito マイクロサービスを使用した Spring Boot アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoRuntime  # Application type
metadata:
  name: example-springboot  # Application name
spec:
  runtime: springboot
```



注記

この場合、アプリケーションは、KogitoRuntime を使用してローカルにビルドされ、デプロイされます。アプリケーション名が **KogitoBuild** および **KogitoRuntime** で同じである必要があります。

9. アプリケーションデータを定義したら、**Create** をクリックして Red Hat ビルドの Kogito マイクロサービスを生成します。
アプリケーションが Red Hat ビルドの Kogito マイクロサービスページに一覧表示されます。アプリケーション名を選択して、アプリケーション設定や YAML ファイルの内容を表示または変更できます。

10. Web コンソールの左側のメニューで **Builds → Builds** に移動して、アプリケーションのビルドのステータスを表示します。
特定のビルドを選択して、ビルドの詳細を表示できます。

11. アプリケーションのビルドが完了したら、**Workloads → Deployments** に移動して、アプリケーションのデプロイメント、Pod のステータスなどの情報を表示します。

12. Red Hat ビルドの Kogito マイクロサービスをデプロイしたら、Web コンソールの左側のメニューの **Networking → Routes** に移動し、デプロイされたアプリケーションへのアクセスリンクを表示します。
アプリケーション名を選択して、ルート設定を表示または変更できます。

    必要に応じてアプリケーションルートを使用すると、Red Hat ビルドの Kogito マイクロサービスはビジネス自動化ソリューションと統合できます。

9.3. カスタムイメージビルドおよび OPENSHIFT WEB コンソールを使用した OPENSHIFT への RED HAT ビルドの KOGITO マイクロサービスのデプロイ

OpenShift で Red Hat ビルドの Kogito マイクロサービスをビルドし、デプロイする代わりに、カスタムイメージビルドを使用できます。

Operator は以下のカスタムリソースを使用して、開発するマイクロサービス (ドメイン固有のマイクロサービス) をデプロイします。

- **KogitoRuntime** はランタイムイメージを起動し、要件に応じて設定します。

前提条件

- RHPAM Kogito Operator がインストールされている。
- OpenShift Web コンソールにアクセスでき、**KogitoRuntime** の作成および編集に必要なパーミッションが設定されている。
- (Red Hat ビルドの Quarkus のみ) プロジェクトの **pom.xml** ファイルに **quarkus-smallrye-health** 拡張機能の依存関係が含まれる。この拡張機能は、OpenShift の Red Hat ビルドの Quarkus プロジェクトに必要な Liveness および Readiness Probe を有効する。

  OpenShift 上の Red Hat ビルドの Quarkus アプリケーションの SmallRye Health 依存関係

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-health</artifactId>
</dependency>
```

手順

1. アプリケーションをローカルに構築します。
2. 以下の内容を含むプロジェクトルートディレクトリーに **Containerfile** を作成します。

   Red Hat ビルドの Quarkus アプリケーションの Containerfile の例

```
FROM registry.redhat.io/rhpam-7/rhpam-kogito-runtime-jvm-rhel8:7.11.0

ENV RUNTIME_TYPE quarkus

COPY target/quarkus-app/lib/ $KOGITO_HOME/bin/lib/
```

```
COPY target/quarkus-app/*.jar $KOGITO_HOME/bin
COPY target/quarkus-app/app/ $KOGITO_HOME/bin/app/
COPY target/quarkus-app/quarkus/ $KOGITO_HOME/bin/quarkus/
```

Spring Boot アプリケーションの Containerfile の例

```
FROM registry.redhat.io/rhpam-7/rhpam-kogito-runtime-jvm-rhel8:7.11.0

ENV RUNTIME_TYPE springboot

COPY target/<application-jar-file> $KOGITO_HOME/bin
```

- **application-jar-file** は、アプリケーションの JAR ファイルの名前です。

3. 以下のコマンドを使用して Red Hat ビルドの Kogito イメージをビルドします。

```
podman build --tag <final-image-name> -f <Container-file>
```

このコマンドでは、**final-image-name** は Red Hat ビルドの Kogito イメージの名前、**Container-file** は直前の手順で作成した **Containerfile** に置き換えます。

4. 必要に応じて、以下のコマンドを使用してビルドイメージをテストします。

```
podman run --rm -it -p 8080:8080 <final-image-name>
```

5. 以下のコマンドを使用して、ビルドした Red Hat ビルドの Kogito イメージをイメージレジストリーにプッシュします。

```
podman push <final-image-name>
```

6. **Operators → Installed Operators** に移動し、**RHPAM Kogito Operator** を選択します。

7. Red Hat ビルドの Kogito マイクロサービス定義を作成するには、Operator ページで **Kogito Runtime** タブを選択し、**Create KogitoRuntime** をクリックします。

8. アプリケーションウィンドウで、**Form View** または **YAML View** を使用してマイクロサービス定義を設定します。
少なくとも、以下の YAML ファイルのサンプルに示されるようにアプリケーション設定を定義します。

Red Hat ビルドの Kogito マイクロサービスを使用した Red Hat ビルドの Quarkus アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoRuntime # Application type
metadata:
  name: example-quarkus # Application name
spec:
  image: <final-image-name> # Kogito image name
  insecureImageRegistry: true # Can be omitted when image is pushed into secured registry with valid certificate
```

Red Hat ビルドの Kogito マイクロサービスを使用した Spring Boot アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoRuntime # Application type
metadata:
  name: example-springboot # Application name
spec:
  image: <final-image-name> # Kogito image name
  insecureImageRegistry: true # Can be omitted when image is pushed into secured registry with valid certificate
  runtime: springboot
```

9. アプリケーションデータを定義したら、**Create** をクリックして Red Hat ビルドの Kogito マイクロサービスを生成します。
アプリケーションが Red Hat ビルドの Kogito マイクロサービスページに一覧表示されます。アプリケーション名を選択して、アプリケーション設定や YAML ファイルの内容を表示または変更できます。

10. アプリケーションのビルドが完了したら、**Workloads → Deployments** に移動して、アプリケーションのデプロイメント、Pod のステータスなどの情報を表示します。

11. Red Hat ビルドの Kogito マイクロサービスをデプロイしたら、Web コンソールの左側のメニューの **Networking → Routes** に移動し、デプロイされたアプリケーションへのアクセスリンクを表示します。
アプリケーション名を選択して、ルート設定を表示または変更できます。

    必要に応じてアプリケーションルートを使用すると、Red Hat ビルドの Kogito マイクロサービスはビジネス自動化ソリューションと統合できます。

9.4. ファイルビルドおよび OPENSHIFT WEB コンソールを使用した OPENSHIFT への RED HAT ビルドの KOGITO マイクロサービスのデプロイ

Decision Model and Notation (DMN)、Drools Rule Language (DRL)、プロパティーファイルなどの 1 つのファイルから、または複数のファイルを含むディレクトリーから Red Hat ビルドの Kogito マイクロサービスをビルドおよびデプロイできます。ローカルファイルシステムパスからファイル を 1 つまたは、ディレクトリーを 1 つのみ指定できます。ファイルまたはディレクトリーを OpenShift クラスターにアップロードすると、新しい Source-to-Image (S2I) ビルドが自動的に開始されます。

Operator は以下のカスタムリソースを使用して、開発するマイクロサービス (ドメイン固有のマイクロサービス) をデプロイします。

- **KogitoBuild** はファイルからアプリケーションを生成し、ランタイムイメージを作成します。
- **KogitoRuntime** はランタイムイメージを起動し、要件に応じて設定します。

前提条件

- RHPAM Kogito Operator がインストールされている。
- **oc** OpenShift CLI がインストールされており、関連する OpenShift クラスターにログインしている。**oc** のインストールおよびログインの手順については、OpenShift ドキュメント を参照してください。

- OpenShift Web コンソールにアクセスでき、**KogitoBuild** および **KogitoRuntime** の作成および編集に必要なパーミッションが設定されている。

手順

1. **Operators → Installed Operators** に移動し、**RHPAM Kogito Operator** を選択します。
2. Red Hat ビルドの Kogito ビルド定義を作成するには、Operator ページで **Kogito Build** タブを選択し、**Create KogitoBuild** をクリックします。
3. アプリケーションウィンドウで、**Form View** または **YAML View** を使用してビルド定義を設定します。
   少なくとも、以下の YAML ファイルのサンプルに示されるようにアプリケーション設定を定義します。

   Red Hat ビルドの Kogito ビルドを使用した Red Hat ビルドの Quarkus アプリケーションの YAML 定義の例

   ```
   apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this service
   kind: KogitoBuild  # Application type
   metadata:
     name: example-quarkus  # Application name
   spec:
     type: LocalSource
   ```

   Red Hat ビルドの Kogito ビルドを使用した Spring Boot アプリケーションの YAML 定義の例

   ```
   apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this service
   kind: KogitoBuild  # Application type
   metadata:
     name: example-springboot  # Application name
   spec:
     runtime: springboot
     type: LocalSource
   ```

   注記

   内部 Maven リポジトリーを設定した場合は、Maven ミラーサービスとして使用でき、Red Hat ビルドの Kogito ビルド定義に Maven ミラー URL を指定して、ビルド時間を大幅に短縮できます。

   ```
   spec:
     mavenMirrorURL: http://nexus3-nexus.apps-crc.testing/repository/maven-public/
   ```

   内部 Maven リポジトリーの詳細は、Apache Maven ドキュメントを参照してください。
4. アプリケーションデータを定義したら、**Create** をクリックして Red Hat ビルドの Kogito ビルドを生成します。
   アプリケーションが **Red Hat ビルドの KogitoBuilds** ページに一覧表示されます。アプリケーション名を選択して、アプリケーション設定および YAML の詳細を表示または変更できます。

5. 以下のコマンドを使用して、ファイルアセットをアップロードします。

```
$ oc start-build example-quarkus-builder --from-file=<file-asset-path> -n namespace
```

- **file-asset-path** は、アップロードするファイルハンドラーのパスに、
- **Namespace** は、**KogitoBuild** の作成先の namespace に置き換えます。

6. Red Hat ビルドの Kogito マイクロサービス定義を作成するには、Operator ページで **Kogito Runtime** タブを選択し、**Create KogitoRuntime** をクリックします。

7. アプリケーションウィンドウで、**Form View** または **YAML View** を使用してマイクロサービス定義を設定します。
少なくとも、以下の YAML ファイルのサンプルに示されるようにアプリケーション設定を定義します。

Red Hat ビルドの Kogito マイクロサービスを使用した Red Hat ビルドの Quarkus アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoRuntime  # Application type
metadata:
  name: example-quarkus  # Application name
```

Red Hat ビルドの Kogito マイクロサービスを使用した Spring Boot アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoRuntime  # Application type
metadata:
  name: example-springboot  # Application name
spec:
  runtime: springboot
```

注記

この場合、アプリケーションはファイルからビルドされ、KogitoRuntime を使用してデプロイされます。アプリケーション名が **KogitoBuild** および **KogitoRuntime** で同じである必要があります。

8. アプリケーションデータを定義したら、**Create** をクリックして Red Hat ビルドの Kogito マイクロサービスを生成します。
アプリケーションが Red Hat ビルドの Kogito マイクロサービスページに一覧表示されます。アプリケーション名を選択して、アプリケーション設定や YAML ファイルの内容を表示または変更できます。

9. Web コンソールの左側のメニューで **Builds → Builds** に移動して、アプリケーションのビルドのステータスを表示します。
特定のビルドを選択して、ビルドの詳細を表示できます。

注記

OpenShift デプロイメント用に作成するすべての Red Hat ビルドの Kogito マイクロサービスでは、従来のランタイムビルドと S2I (接尾辞 **-builder**) ビルドの 2 つのビルドが生成され、Web コンソールの **Builds** ページに一覧表示されます。S2I メカニズムは OpenShift ビルドでアプリケーションを構築し、このアプリケーションを次の OpenShift ビルドに渡してランタイムコンテナーイメージにパッケージ化します。

10. アプリケーションのビルドが完了したら、**Workloads → Deployments** に移動して、アプリケーションのデプロイメント、Pod のステータスなどの情報を表示します。

11. Red Hat ビルドの Kogito マイクロサービスをデプロイしたら、Web コンソールの左側のメニューの **Networking → Routes** に移動し、デプロイされたアプリケーションへのアクセスリンクを表示します。
アプリケーション名を選択して、ルート設定を表示または変更できます。

    必要に応じてアプリケーションルートを使用すると、Red Hat ビルドの Kogito マイクロサービスはビジネス自動化ソリューションと統合できます。

第10章 RED HAT OPENSHIFT CONTAINER PLATFORM での RED HAT ビルドの KOGITO マイクロサービスのプローブ

Red Hat OpenShift Container Platform のプローブは、アプリケーションが機能していることを検証します。検証されない場合には、再起動する必要があります。Red Hat ビルドの Quarkus および Spring Boot の Red Hat ビルドの Kogito マイクロサービスの場合には、プローブは HTTP 要求 (拡張機能によって公開されるエンドポイントにデフォルト設定) を使用してアプリケーションと対話します。そのため、Red Hat OpenShift Container Platform で Red Hat ビルドの Kogito マイクロサービスを実行するには、拡張機能をインポートして liveness、readiness、および startup probe のアプリケーション可用性の情報を提供する必要があります。

10.1. RED HAT OPENSHIFT CONTAINER PLATFORM での RED HAT ビルドの QUARKUS アプリケーションのヘルスチェック拡張機能の追加

Red Hat OpenShift Container Platform の Red Hat ビルドの Quarkus をベースとする Red Hat ビルドの Kogito サービスのヘルスチェック拡張機能を追加できます。

手順

コマンドターミナルで、プロジェクトの **pom.xml** ファイルに移動し、**quarkus-smallrye-health** 拡張機能の依存関係を追加します。

Red Hat OpenShift Container Platform の Red Hat ビルドの Quarkus アプリケーションの SmallRye Health 依存関係

```
<dependencies>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-health</artifactId>
  </dependency>
</dependencies>
```

10.2. RED HAT OPENSHIFT CONTAINER PLATFORM での SPIRNG BOOT アプリケーションのヘルスチェック拡張機能の追加

Red Hat OpenShift Container Platform の Spring Boot をベースとする Red Hat ビルドの Kogito マイクロサービスのヘルスチェック拡張機能を追加できます。

手順

コマンドターミナルで、プロジェクトの **pom.xml** ファイルに移動し、以下の Spring Boot アクチュエーターの依存関係を追加します。

Red Hat OpenShift Container Platform における Spring Boot アプリケーションの Spring Boot のアクチュエーター依存関係

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
</dependencies>
```

10.3. RED HAT OPENSHIFT CONTAINER PLATFORM での RED HAT ビルドの KOGITO マイクロサービスのカスタムプローブ設定

liveness、readiness、および startup probe のカスタムエンドポイントを設定することもできます。

手順

1. 以下の例のように、プロジェクトの **KogitoRuntime** YAML ファイルでプローブを定義します。

カスタムプローブエンドポイントを含む Red Hat ビルドの Kogito マイクロサービスカスタムリソースの例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this service
kind: KogitoRuntime
metadata:
  name: process-quarkus-example # Application name
spec:
  replicas: 1
  probes:
    livenessProbe:
      httpGet:
        path: /probes/live # Liveness endpoint
        port: 8080
    readinessProbe:
      httpGet:
        path: /probes/ready # Readiness endpoint
        port: 8080
    startupProbe:
      tcpSocket:
        port: 8080
```

第11章 RED HAT PROCESS AUTOMATION MANAGER KOGITO OPERATOR と PROMETHEUS および GRAFANA との対話

Red Hat Process Automation Manager の Red Hat ビルドの Kogito は、Red Hat ビルドの Kogito マイクロサービスの Prometheus メトリクス監視を有効にする **monitoring-prometheus-addon** アドオンを提供し、アドオンによってエクスポートされたデフォルトのメトリクスを使用する Grafana ダッシュボードを生成します。RHPAM Kogito Operator は Prometheus Operator を使用して、Prometheus からスクレープするプロジェクトからメトリクスを公開します。この依存関係により、Prometheus Operator はプロジェクトと同じ namespace にインストールする必要があります。

Red Hat ビルドの Kogito マイクロサービスの Prometheus メトリクス監視を有効にする場合は、使用するフレームワークに応じて、プロジェクトの **pom.xml** ファイルに以下の依存関係を追加します。

Prometheus Red Hat ビルドの Quarkus アドオンの依存関係

```
<dependency>
    <groupId>org.kie.kogito</groupId>
    <artifactId>monitoring-prometheus-quarkus-addon</artifactId>
</dependency>
```

Prometheus Spring Boot アドオンの依存関係

```
<dependency>
    <groupId>org.kie.kogito</groupId>
    <artifactId>monitoring-prometheus-springboot-addon</artifactId>
</dependency>
```

monitoring-prometheus-addon アドオンを使用し、Prometheus Operator がインストールされている Red Hat ビルドの Kogito マイクロサービスをデプロイする場合、Red Hat Process Automation Manager Kogito Operator は **ServiceMonitor** カスタムリソースを作成し、以下の例のように Prometheus のメトリクスを公開します。

Prometheus の ServiceMonitor リソースの例

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app: onboarding-service
  name: onboarding-service
  namespace: kogito
spec:
  endpoints:
  - path: /metrics
    targetPort: 8080
    scheme: http
  namespaceSelector:
    matchNames:
    - kogito
  selector:
    matchLabels:
      app: onboarding-service
```

Prometheus Operator によって管理される **Prometheus** カスタムリソースを手作業で設定し、**ServiceMonitor** リソースを選択する必要があります。

Prometheus リソースの例

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus
spec:
  serviceAccountName: prometheus
  serviceMonitorSelector:
    matchLabels:
      app: dmn-drools-quarkus-metrics-service
```

ServiceMonitor リソースで Prometheus リソースを設定した後に、Prometheus Web コンソールの **Targets** ページで Prometheus によってスクレープされたエンドポイントを表示できます。Red Hat Process Automation Manager サービスが公開するメトリクスが **Graph** ビューに表示されます。

RHPAM Kogito Operator は、アドオンによって生成された Grafana ダッシュボードごとに、Grafana Operator によって定義される **GrafanaDashboard** カスタムリソースも作成します。ダッシュボードの **app** ラベルは、デプロイされた Red Hat ビルドの Kogito マイクロサービスの名前です。Kogito マイクロサービスの関連する Red Hat ビルドに従い、**Grafana** カスタムリソースの **dashboardLabelSelector** プロパティーを設定する必要があります。

Grafana リソースの例

```
apiVersion: integreatly.org/v1alpha1
kind: Grafana
metadata:
  name: example-grafana
spec:
  ingress:
    enabled: true
  config:
    auth:
      disable_signout_menu: true
    auth.anonymous:
      enabled: true
    log:
      level: warn
      mode: console
    security:
      admin_password: secret
      admin_user: root
  dashboardLabelSelector:
    - matchExpressions:
        - key: app
          operator: In
          values:
            - my-kogito-application
```

第12章 RED HAT ビルドの KOGITO マイクロサービスのデプロイメントのトラブルシューティング

Operator を使用して Red Hat ビルドの Kogito マイクロサービスをデプロイする時に発生する可能性のある問題のトラブルシューティングを行うには、本セクションの情報を使用します。以下の情報は、新しい問題および回避策が検出されるたびに更新されています。

実行中のビルドがない

実行中のビルドや関連する namespace で作成されたリソースが表示されない場合は、以下のコマンドを実行して実行中の Pod を取得し、Pod の Operator ログを表示します。

指定の Pod に関する RHPAM Kogito Operator ログの表示

```
// Retrieves running pods
$ oc get pods

NAME                                READY   STATUS      RESTARTS   AGE
kogito-operator-6d7b6d4466-9ng8t   1/1     Running     0          26m

// Opens RHPAM Kogito Operator log for the pod
$ oc logs -f kogito-operator-6d7b6d4466-9ng8t
```

KogitoRuntime のステータスを確認します。

たとえば、以下の YAML 定義を使用して、存在しないイメージを含めて **KogitoRuntime** アプリケーションを作成します。

KogitoRuntime アプリケーションの YAML 定義の例

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoRuntime # Application type
metadata:
  name: example # Application name
spec:
  image: 'not-existing-image:latest'
  replicas: 1
```

bash コンソールで **oc describe KogitoRuntime example** コマンドを使用して、**KogitoRuntime** アプリケーションのステータスを確認できます。bash コンソールで **oc describe KogitoRuntime** サンプルコマンドを実行すると、以下の出力が表示されます。

KogitoRuntime ステータスの例

```
[user@localhost ~]$ oc describe KogitoRuntime example
Name:         example
Namespace:    username-test
Labels:       <none>
Annotations:  <none>
API Version:  rhpam.kiegroup.org/v1
Kind:         KogitoRuntime
Metadata:
  Creation Timestamp:  2021-05-20T07:19:41Z
  Generation:          1
  Managed Fields:
```

```
    API Version:  rhpam.kiegroup.org/v1
    Fields Type:  FieldsV1
    fieldsV1:
      f:spec:
        .:
        f:image:
        f:replicas:
    Manager:      Mozilla
    Operation:    Update
    Time:         2021-05-20T07:19:41Z
    API Version:  rhpam.kiegroup.org/v1
    Fields Type:  FieldsV1
    fieldsV1:
      f:spec:
        f:monitoring:
        f:probes:
          .:
          f:livenessProbe:
          f:readinessProbe:
        f:resources:
        f:runtime:
      f:status:
        .:
        f:cloudEvents:
        f:conditions:
    Manager:         main
    Operation:       Update
    Time:            2021-05-20T07:19:45Z
  Resource Version:  272185
  Self Link:         /apis/rhpam.kiegroup.org/v1/namespaces/ksuta-test/kogitoruntimes/example
  UID:               edbe0bf1-554e-4523-9421-d074070df982
Spec:
  Image:     not-existing-image:latest
  Replicas:  1
Status:
  Cloud Events:
  Conditions:
    Last Transition Time:  2021-05-20T07:19:44Z
    Message:
    Reason:                NoPodAvailable
    Status:                False
    Type:                  Deployed
    Last Transition Time:  2021-05-20T07:19:44Z
    Message:
    Reason:                RequestedReplicasNotEqualToAvailableReplicas
    Status:                True
    Type:                  Provisioning
    Last Transition Time:  2021-05-20T07:19:45Z
    Message:               you may not have access to the container image "quay.io/kiegroup/not-
existing-image:latest"
    Reason:                ImageStreamNotReadyReason
    Status:                True
    Type:                  Failed
```

出力の最後に、**KogitoRuntime** ステータスと該当するメッセージが表示されます。

第13章 関連情報

- DMN モデルを使用したデシジョンサービスの作成
- DRL ルールを使用したデシジョンサービスの作成
- PMML モデルでのデシジョンサービスの作成

付録A バージョン情報

本書の最終更新日: 2022 年 3 月 8 日 (火)

付録B お問い合わせ先

Red Hat Process Automation Manager のドキュメントチーム: brms-docs@redhat.com