Red Hat Process Automation Manager 7.13

在 Red Hat Process Automation Manager 中开始使用 Red Hat build of Kogito

Last Updated: 2024-03-19

Red Hat Process Automation Manager 7.13 在 Red Hat Process Automation Manager 中开始使用 Red Hat build of Kogito

法律通告

摘要

本文档论述了如何在 Red Hat Process Automation Manager 中开始使用红帽构建 Kogito 来构建云原生业务应用程序。

目录

前言

作为业务决策的开发人员，您可以使用红帽构建 Kogi 来构建适合您业务域和工具的云原生应用程序。

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 *CTO Chris Wright* 的信息。

部分 I. 开始使用 RED HAT BUILD OF KOGITO 微服务

作为业务决策的开发人员，您可以使用 Red Hat build of Kogito Business Automation 来使用 Decision Model 和 Notation(DMN)模型、Eolus 规则语言(DRL)规则、预测模型标记语言(PMML)或所有三种方法的组合来开发决策服务。

先决条件

- JDK 11 或更高版本已经安装。
- 已安装 Apache Maven 3.6.2 或更高版本。

第 1 章 RED HAT PROCESS AUTOMATION MANAGER 中的 KOGITO 微服务构建

Red Hat build of Kogito 是用于构建云就绪业务应用程序的云原生业务自动化技术。名称 *Kogito* 从拉丁美洲"Cogito"生成，如 "Cogito, ergo sum"("I think, so I am")，它是 pronounced [时间为 **[时间为：acc-jee-to]（KO-jee-to]** (*KO-jee-to*)。字母 *K* 指 Kubernetes，Red Hat OpenShift Container Platform 的基础是 Red Hat Process Automation Manager 的目标云平台，以及 Red Hat building(KIE)的开源业务自动化项目。

Red Hat Process Automation Manager 中的 Red Hat build of Kogito 针对混合云环境进行了优化，并符合您的域和工具需求。红帽构建的 Kogito 微服务的核心目的是帮助您将一组决策放进您自己的域特定的云原生服务集合。

重要

在 Red Hat Process Automation Manager 7.13 版本中，Red Hat build of Kogito 的支持仅限于决策服务，包括 Decision Model 和 Notation(DMN)、Istio 规则语言(DRL)和 Predictive Model Markup Language(PMML)。以后的发行版本中，这个支持将改进并扩展到业务流程建模通知(BPMN)。

当您使用 Red Hat build of Kogito 时，您要构建云原生应用程序作为一组独立的域特定微服务，以实现一些业务价值。您用来描述目标行为的决策将作为您创建的微服务的一部分执行。生成的微服务具有高度分布式的可扩展，无集中式编配服务，您的微服务使用的运行时则针对所需的内容进行了优化。

作为业务规则开发人员，您可以使用红帽流程自动化管理器中的 Kogito 微服务构建适合您的业务域和工具的云原生应用程序。

1.1. CLOUD-FIRST 优先级

红帽构建的 Kogito 微服务旨在在云基础架构上运行和扩展。您可以使用 Red Hat Process Automation Manager 中的红帽构建 Kogito 微服务与最新的基于云的技术（如 Red Hat build of Quarkus）一起使用，以增加容器应用程序平台的启动时间和即时扩展，如 Red Hat OpenShift Container Platform。

例如，红帽构建的 Kogito 微服务与以下技术兼容：

- **Red Hat OpenShift Container Platform**基于 Kubernetes，是构建和管理容器化应用程序的目标平台。
- **Red Hat build of Quarkus**是一个用于 Kubernetes 的原生 Java 堆栈，您可以使用 Red Hat build of Kogito 微服务构建应用程序。
- **Spring Boot** 是一个应用程序框架，可用于使用 Red Hat Process Automation Manager 配置 Spring Framework。

1.2. RED HAT BUILD OF KOGITO MICROSERVICES ON RED HAT BUILD OF QUARKUS 和 SPRING BOOT

Red Hat build of Kogito 微服务支持的主要 Java 框架是 Red Hat build of Quarkus 和 Spring Boot。

Red Hat build of Quarkus 是一个 Kubernetes 原生 Java 框架，它带有一个容器先行的 Java 应用程序方法，特别是 OpenJDK HotSpot 等 Java 虚拟机(JVM)。红帽构建的 Quarkus 通过减少 Java 应用程序和容器镜像占用空间的大小，从上一代消除一些 Java 编程工作负载，并减少运行这些镜像所需的内存量来优化 Java。

对于 Red Hat build of Kogito 微服务，红帽构建的 Quarkus 是最佳 Kubernetes 兼容性和增强的开发人员功能的首选框架，如在开发模式中进行高级调试的实时重新加载。

Spring Boot 是一个基于 Java 的框架，用于构建独立生产就绪的 Spring 应用程序。Spring Boot 可让您使用最小配置和完整 Spring 配置设置开发 Spring 应用程序。

对于 Red Hat build of Kogito 微服务，对于需要在现有 Spring Framework 环境中使用 Red Hat Process Automation Manager 的开发人员支持 Spring Boot。

第 2 章 用于红帽构建 KOGITO 微服务的 DMN 型号者

Red Hat Process Automation Manager 提供了扩展或应用程序，您可以使用它们为使用图形模型器构建 Kogito 微服务构建红帽构建的 Decision Model 和 Notation(DMN)决策模型。

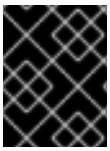
支持以下 DMN 模型器：

- **VS Code 扩展** ：允许您在 Visual Studio Code(VS Code)中查看和设计 DMN 模型。VS Code 扩展需要 VS Code 1.46.0 或更新版本。
  要直接在 VS Code 中安装 VS Code 扩展，请在 VS Code 中选择 **Extensions** 菜单选项，并搜索并安装 **Red Hat Business Automation Bundle**扩展。

- **商业模式的独立编辑器** ：使您能够查看和设计嵌入在 web 应用程序中的 DMN 模型。要下载必要的文件，您可以使用 Kogito 工具存储库中的 NPM 工件，或者直接下载基于 DMN 独立编辑器库的 JavaScript 文件，网址为 **https://kiegroup.github.io/kogito-online/standalone/dmn/index.js**。

2.1. 安装 RED HAT PROCESS AUTOMATION MANAGER VS CODE 扩展捆绑包

Red Hat Process Automation Manager 提供了一个 **Red Hat Network Automation Bundle**VS Code 扩展，它可让您设计决策模型和符号(DMN)决策模型、业务流程模型和符号(BPMN)2.0 业务流程并直接在 VS Code 中测试场景。VS Code 是开发新业务应用的首选集成开发环境(IDE)。Red Hat Process Automation Manager 还提供单独的 **DMN Editor** 和 **BPMN Editor** VS Code 扩展（如果需要）。



重要

VS Code 中的编辑器部分与 Business Central 中的编辑器兼容，并且 VS Code 不支持几个 Business Central 功能。

先决条件

- 安装了 VS Code 的最新稳定版本。

流程

1. 在 VS Code IDE 中，选择 **Extensions** 菜单选项，并搜索 **Red Hat Business Automation Bundle** for DMN、Harllse 和 test scenario 文件支持。
   对于 DMN 或 BPMN 文件支持，您还可以搜索单独的 **DMN Editor** 或 **BPMN Editor** 扩展。

2. 当 **Red Hat Business Automation Bundle**扩展出现在 VS Code 中时，选择它并点 **Install**。

3. 要获得最佳 VS Code 编辑器行为，请在扩展安装完成后，重新加载或关闭并重新启动 VS Code 实例。

安装 VS Code 扩展捆绑包后，任何 **.dmn**、**.bpmn** 或 **.bpmn2** 文件都会自动显示为图形模型。此外，您打开或创建的 **.scesim** 文件自动显示为表格测试场景模型，用于测试您的业务决策功能。

如果 DMN、CEP 或测试场景模型器只打开 DMN、Hardb 或 test scenario 文件的 XML 源，并显示错误消息，请查看报告的错误和模型文件，以确保定义所有元素。

注意

对于新的 DMN 或 BPMN 模型，您还可以在网页浏览器中输入 **dmn.new** 或 **BPMn.new**，以在在线模型程序中设计 DMN 或 BPMN 模型。完成创建模型后，您可以点击 **Download** in the online modeler 页面将 DMN 或 BPMN 文件导入到 VS Code 中的 Red Hat Process Automation Manager 项目中。

2.2. 配置 RED HAT PROCESS AUTOMATION MANAGER 独立编辑器

Red Hat Process Automation Manager 提供独立编辑器，这些编辑器在自包含的库中分发，为每个编辑器提供一个一体化 JavaScript 文件。JavaScript 文件使用全面的 API 来设置和控制编辑器。

您可以使用以下方法安装独立编辑器：

- 手动下载每个 JavaScript 文件
- 使用 NPM 软件包

流程

1. 使用以下方法之一安装独立编辑器：
   手动下载每个 JavaScript 文件 ：对于这个方法，请按照以下步骤操作：

   a. 下载 JavaScript 文件。

   b. 将下载的 Javascript 文件添加到您的托管应用程序中。

   c. 将以下 &lt **;script&** gt; 标签添加到 HTML 页面：

   DMN 编辑器的 HTML 页面标记

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

   BPMN 编辑器的 HTML 页面的脚本标签

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

   使用 NPM 软件包 ：对于这个方法，请按照以下步骤操作：

   a. 在 **package.json** 文件中添加 NPM 软件包：

   添加 NPM 软件包

   ```
   npm install @kie-tools/kie-editors-standalone
   ```

   b. 将每个编辑器库导入到 **TypeScript** 文件：

   导入每个编辑器

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

2. 安装独立编辑器后，使用提供的编辑器 API 打开所需的编辑器，如下例所示，打开 DMN 编辑器。每个编辑器的 API 相同。

打开 DMN 独立编辑器

```
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("")
      }
    ]
  ])
});
```

在 editor API 中使用以下参数：

表 2.1. 示例参数

参数	描述
container	附加编辑器的 HTML 元素。
initialContent	对 DMN 模型内容的承诺。这个参数可以为空，如下例所示： • **Promise.resolve("")** • **Promise.resolve("<DIAGRAM_CONTENT_DIRECTLY_HERE>")** • **fetch("MyDmnModel.dmn").then(content ⇒ content.text())**
ReadOnly （可选）	可让您在编辑器中允许更改。在编辑器中将 设置为 **false**（默认）以允许内容编辑和 **true**。
Origin （可选）	仓库的起源。默认值为 **window.location.origin**。
资源 （可选）	编辑器的资源映射。例如，这个参数用于为 DMN 编辑器提供包含的模型，或为 BPMN 编辑器提供工作项目定义。映射中的每个条目都包含一个资源名称和一个对象，它由 **content -type （文本 或二进制 ）和内容**组成（与 **初始 Content** 参数类似）。

返回的对象包含操作编辑器所需的方法。

表 2.2. 返回的对象方法

方法	描述
getContent（）：Promise<string>	返回包含编辑器内容的保证。
setContent(path: string, content: string): void	设置编辑器的内容。
getPreview（）：Promise<string>	返回包含当前图表的 SVG 字符串的保证。
subscribeToContentChanges（callback:(isDirty: boolean)IFL void）:(isDirty: boolean)void	当编辑器中的内容更改并返回用于 unsubscription 的回调时，设置要调用的回调。
subscribe toContentChanges（callback: (isDirty: boolean)IFL void）：void	当内容在编辑器中更改时，取消订阅传递的回调。
markAsSaved(): void	重置编辑器状态，这表示已保存编辑器中的内容。另外，它会激活与内容更改相关的订阅回调。
undo（）: void	在编辑器中取消上次更改。另外，它会激活与内容更改相关的订阅回调。
redo（）: void	在编辑器中恢复最近一次撤消的更改。另外，它会激活与内容更改相关的订阅回调。
close（）: void	关闭编辑器。
getElementPosition(selector: string): Promise<Rect>	提供了一种替代方式，可以在可清空或视频组件中的元素中时扩展标准查询选择器。**选择器** 参数必须遵循 <targetNamespaces **>:::<SELECT** > 格式，如 **Canvas:::MySquare** 或 **Video:::PresenterHand**。此方法返回一个代表元素位置的 **Rect**。
envelopeApi: MessageBusClientApi<KogitoEditorEnvelopeApi>	这是高级编辑器 API。有关高级编辑器 API 的更多信息，请参阅 MessageBusClientApi 和 KogitoEditorEnvelopeApi。

第 3 章 为红帽构建 KOGITO 微服务创建 MAVEN 项目

在开始开发 Red Hat build of Kogito 微服务之前，您需要创建一个 Maven 项目，在其中构建资产以及您的应用程序的任何其他相关资源。

流程

1. 在命令终端中，导航到要存储新项目的本地文件夹。
2. 输入以下命令在定义的文件夹中生成项目：

在 Red Hat build of Quarkus 上

```
$ mvn io.quarkus:quarkus-maven-plugin:create \
    -DprojectGroupId=org.acme -DprojectArtifactId=sample-kogito \
    -DprojectVersion=1.0.0-SNAPSHOT -Dextensions=kogito-quarkus
```

在 Spring Boot 上

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.kie.kogito \
    -DarchetypeArtifactId=kogito-spring-boot-archetype \
    -DgroupId=org.acme -DartifactId=sample-kogito \
    -DarchetypeVersion=1.11.0.Final \
    -Dversion=1.0-SNAPSHOT
```

此命令生成 **sample-kogito** Maven 项目，并导入扩展以获取所有需要的依赖关系和配置，以便为您的应用程序做好业务自动化准备。

如果要启用项目的 PMML 执行，请在包含 Red Hat build of Kogito 微服务的 Maven 项目的 **pom.xml** 文件中添加以下依赖项：

启用 PMML 执行的依赖项

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

在 Red Hat build of Quarkus 上，如果您计划在 OpenShift 上运行应用程序，还必须为 存活度和就绪度探测 导入 **smallrye-health** 扩展，如下例所示：

用于 OpenShift 上红帽构建的 Quarkus 应用程序的 SmallRye Health 扩展

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

此命令在 Red Hat Process Automation Manager 项目的 **pom.xml** 文件中生成以下依赖项：

在 OpenShift 中红帽构建的 Quarkus 应用程序的 SmallRye Health 依赖项

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

3. 在 VS Code IDE 中打开或导入项目，以查看内容。

3.1. 为 RED HAT BUILD OF KOGITO 微服务创建自定义 SPRING BOOT 项目

您可以使用 Spring Boot archetype 为 Red Hat build of Kogito 微服务创建自定义 Maven 项目。Spring Boot archetype 可让您将 Spring Boot starters 或 add-ons 添加到项目中。

Spring Boot starter 是项目的一体化描述符，需要由红帽构建 Kogito 提供的业务自动化引擎。Spring Boot starters 包括决策、规则和预测。

当项目包含所有资产且您希望使用 Red Hat build of Kogito 开始时，您可以使用 **kogito-spring-boot-starter** 启动程序。对于更细致的方法，您可以使用特定的初学者，如 **kogito-decisions-spring-boot-starter** 进行决策，或组合 starters。

Red Hat build of Kogito 支持以下 Spring Boot starters：

决策 Spring Boot starter

为 Spring Boot 项目提供 DMN 支持的入门者。以下是在项目中添加一个决策 Spring boot starter 的示例：

```
<dependencies>
  <dependency>
    <groupId>org.kie.kogito</groupId>
    <artifactId>kogito-decisions-spring-boot-starter</artifactId>
  </dependency>
</dependencies>
```

预测 Spring Boot starter

为 Spring Boot 项目提供 PMML 支持的入门者。以下是向项目添加 insights Spring boot starter 的示例：

```
<dependencies>
  <dependency>
    <groupId>org.kie.kogito</groupId>
    <artifactId>kogito-predictions-spring-boot-starter</artifactId>
  </dependency>
</dependencies>
```

规则 Spring Boot starter

为 Spring Boot 项目提供 DRL 支持入门。以下是在项目中添加规则 Spring boot starter 的示例：

```
<dependencies>
  <dependency>
    <groupId>org.kie.kogito</groupId>
```

```
    <artifactId>kogito-rules-spring-boot-starter</artifactId>
  </dependency>
</dependencies>
```

流程

1. 在命令终端中，导航到要存储新项目的本地文件夹。
2. 输入以下命令使用 **starters** 或 **addons** 属性来生成项目：
   - 要使用 starters 属性 **生成** 项目，请输入以下命令：

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.kie.kogito \
    -DarchetypeArtifactId=kogito-springboot-archetype \
    -DgroupId=org.acme -DartifactId=sample-kogito \
    -DarchetypeVersion=1.11.0.Final \
    -Dversion=1.0-SNAPSHOT
    -Dstarters=decisions
```

     新项目包含运行决策微服务所需的依赖项。您可以使用以逗号分隔的列表组合多个 Spring Boot starters，如 **starters=decisions,rules**。

   - 要使用 **addons** 属性生成包含 Prometheus 监控的项目，请输入以下命令：

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.kie.kogito \
    -DarchetypeArtifactId=kogito-springboot-archetype \
    -DgroupId=org.acme -DartifactId=sample-kogito \
    -DarchetypeVersion=1.11.0.Final \
    -Dversion=1.0-SNAPSHOT
    -Dstarters=descisions
    -Daddons=monitoring-prometheus,persistence-infinispan
```

注意

当您将附加组件传递给属性时，附加组件名称不需要 **kogito-addons-springboot** 前缀。另外，您还可以组合 **附加组件** 和 **starters** 属性来自定义项目。

3. 打开或导入 IDE 中的项目以查看内容。

第 4 章 带有红帽构建 KOGITO 微服务的应用程序示例

红帽构建的 Kogito 微服务包括 **rhpam-7.13.5-kogito-and-optaplanner-quickstarts.zip** 文件中的示例应用程序。这些示例应用程序包含 Red Hat build of Quarkus 或 Spring Boot 上的各种服务，以帮助您开发自己的应用程序。该服务使用一个或多个决策模型和 Notation(DMN)决策模型、Eward Rule Language(DRL)规则单元、预测模型标记语言(PMML)模型或 Java 类来定义服务逻辑。

有关每个示例应用程序以及使用方法的信息，请参阅相关应用程序文件夹中的 **README** 文件。

注意

当您在本地环境中运行示例时，请确保环境与相关应用程序文件夹 **README** 文件中列出的要求匹配。另外，这需要提供必要的网络端口，如红帽构建的 Quarkus、Spring Boot 和 docker-compose 配置。

以下列表介绍了红帽构建 Kogito 微服务提供的一些示例：

注意

这些快速启动示例展示了受支持的设置。未列出的其他快速入门可能只使用由上游社区提供的技术，因此不受红帽完全支持。

决策服务

- **dmn-quarkus-example** 和 **dmn-springboot-example**: decision service（在 Red Hat build of Quarkus 或 Spring Boot）中，使用 DMN 根据流量违反情况来确定驱动程序 penalty 和 suspension。
- **rules-quarkus-helloworld**: Red Hat build of Quarkus 带有单个 DRL 规则单元的 Hello World decision service。
- **ruleunit-quarkus-example** 和 **ruleunit-springboot-example**: 决策服务（在 Red Hat build of Quarkus 或 Spring Boot）中，它使用 DRL 及规则单元验证 shadow 应用程序，并公开 REST 操作来查看应用程序的状态。
- **dmn-pmml-quarkus-example** 和 **dmn-pmml-springboot-example**: A decision service（在 Red Hat build of Quarkus 或 Spring Boot）中，使用 DMN 和 PMML 来确定驱动程序 penalty 和 suspension。
- **dmn-drools-quarkus-metrics** 和 **dmn-drools-springboot-metrics**: decision service（由红帽构建 Quarkus 或 Spring Boot）在 Red Hat build of Kogito 中启用和使用运行时指标监控功能。
- **pmml-quarkus-example** 和 **pmml-springboot-example** ：决定服务（由红帽构建的 Quarkus 或 Spring Boot）使用 PMML。

如需更多信息，请参阅使用 DMN 模型设计决策服务、*使用* DRL 规则设计决策服务，*并使用 PMML 模式设计决策服务*。

第 5 章 使用 DMN 为红帽构建 KOGITO 微服务设计应用程序逻辑

创建项目后，您可以在项目的 **src/main/resources** 文件夹中创建或导入 Decision Model 和 Notation(DMN)决策模型和 Drools Rule Language(DRL)业务规则。您还可以在作为 Java 服务的项目的 **src/main/java** 文件夹中包含 Java 类，或者提供您从决策中调用的实施。

此流程示例为 Kogito 微服务的基本红帽构建，提供 REST 端点 **/persons**。此端点根据示例 **PersonDecisions.dmn** DMN 模型自动生成，以根据正在处理的数据做出决策。

业务决策包括红帽流程自动化管理器服务的决策逻辑。您可以通过不同的方法定义业务规则和决策，例如使用 DMN 模型或 DRL 规则。此流程示例使用 DMN 模型。

先决条件

- 您已创建了一个项目。有关创建 Maven 项目的更多信息，请参阅 *第 3 章 为红帽构建 Kogito 微服务创建 Maven 项目*。

流程

1. 在您为 Red Hat Process Automation Manager 服务生成的 Maven 项目中，导航到 **src/main/java/org/acme** 文件夹并添加以下 **Person.java** 文件：

   个人 Java 对象示例

```
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 + "]";
}

}
```

这个示例 Java 对象集并检索一个人的名称、年龄和 adult 状态。

2. 进入 **src/main/resources** 文件夹并添加以下 **PersonDecisions.dmn** DMN 决策模型：

图 5.1. PersonDecisions DMN 决策要求图(DRD)示例

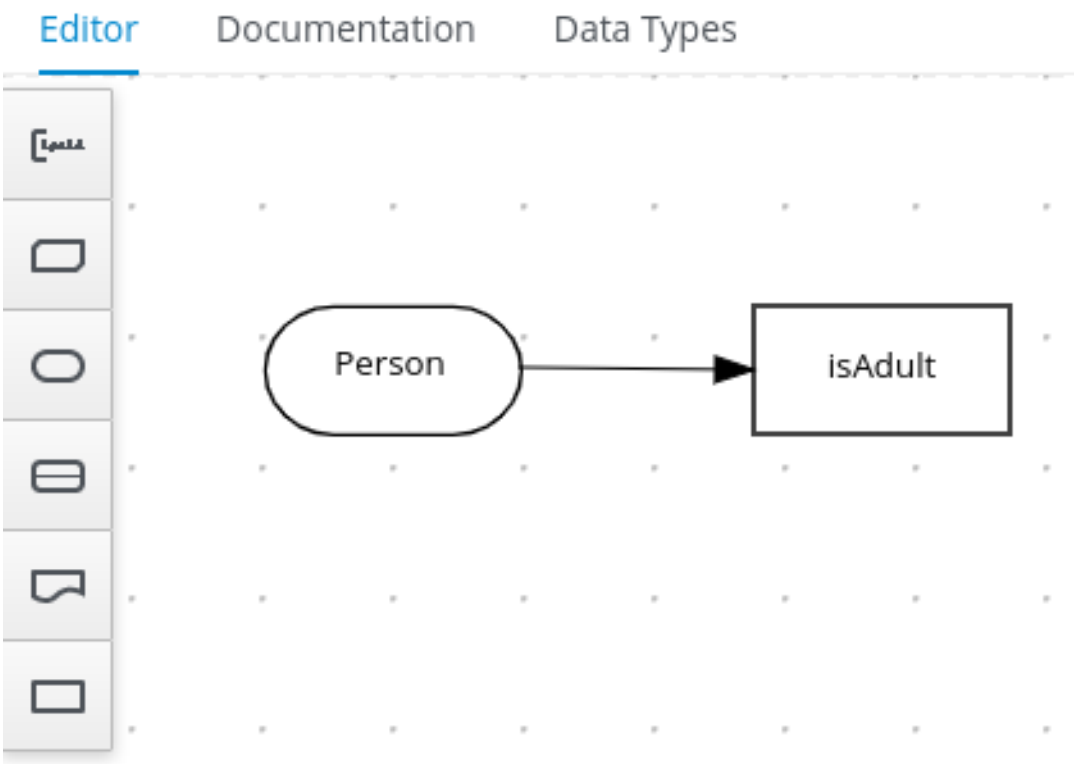


图 5.2. 适用于 isAdult 决策的 DMN 框表达式示例

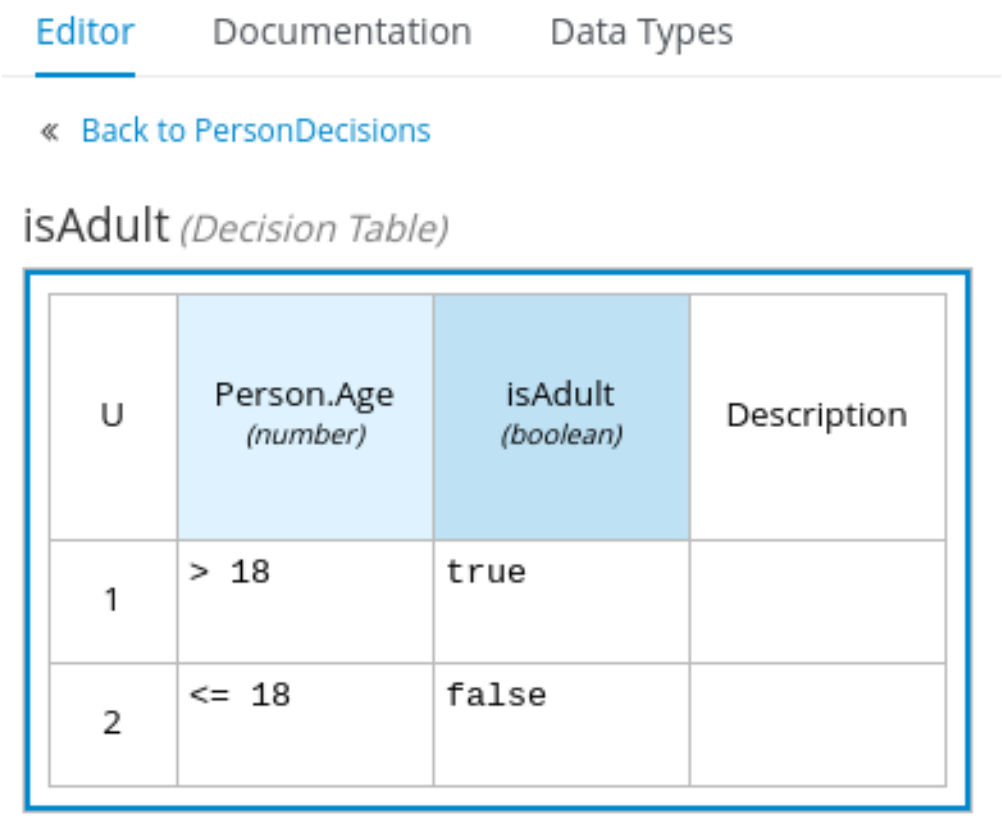
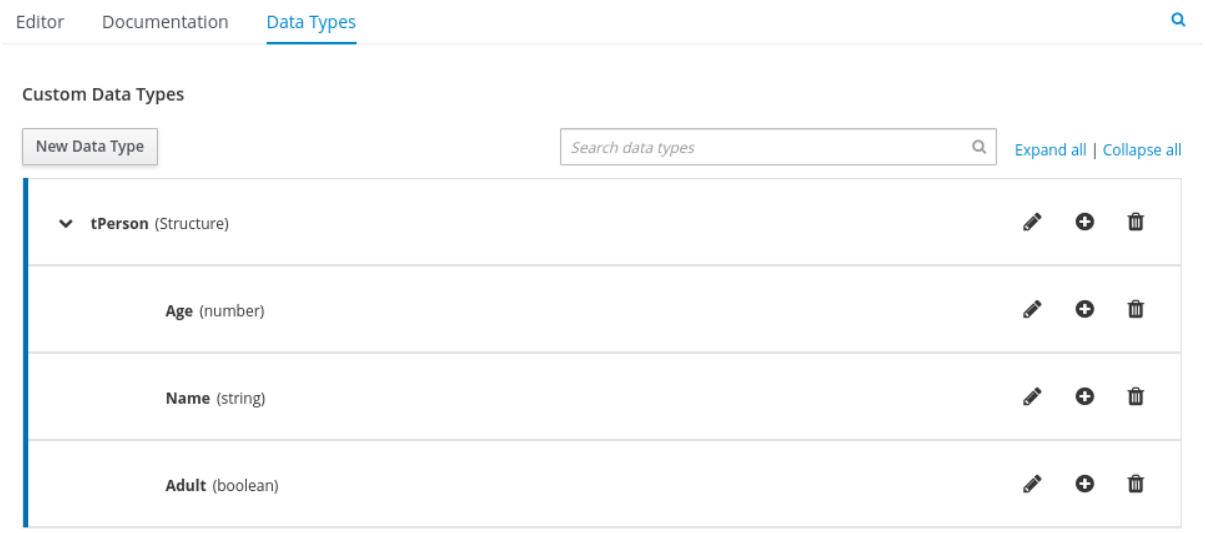


图 5.3. DMN 数据类型示例



这个示例 DMN 模型由基本 DMN 输入节点以及由 DMN 决策表定义的决定节点以及带有自定义结构化数据类型的 DMN 决策节点组成。

在 VS Code 中，您可以添加 **Red Hat Business Automation Bundle**VS Code 扩展，以通过 DMN 模型器设计决策要求图(DRD)、已框式表达式和数据类型。

要快速创建这个示例 DMN 模型，您可以复制以下 **PersonDecisions.dmn** 文件内容：

DMN 文件示例

```
<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>&amp;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>
```

要使用 DMN 型号在 VS Code 中创建此示例 DMN 模型，请按照以下步骤操作：

a. 打开空 **PersonDecisions.dmn** 文件，点 DMN 模型器右上角的 **Properties** 图标，并确认 DMN 模型名称已设为 **PersonDecisions**。

b. **在左侧面板中，选择 DMN Input Data，将节点拖到 canvas 中，然后双击该节点将其命名为 Person。**

c. **在左侧面板，将 DMN 决策节点拖放到 canvas 中，双击节点将其命名为 Adult，并从输入节点链接到其中。**

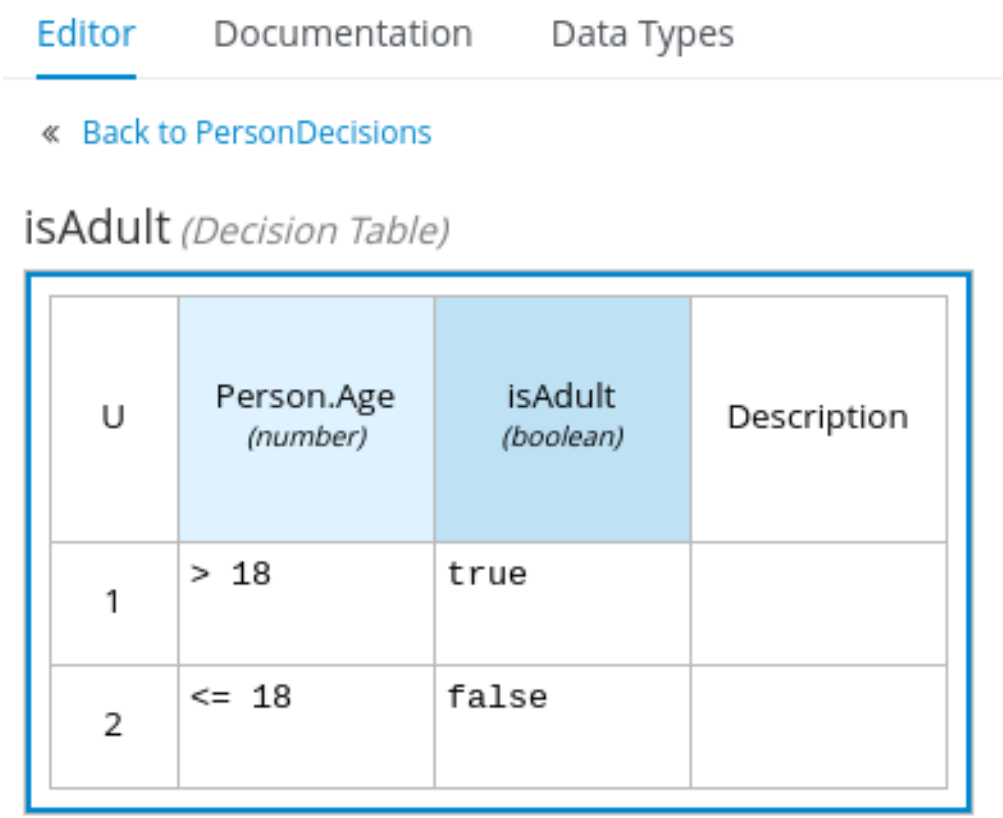
d. **选择决策节点以显示节点选项，并点击 Edit 图标打开 DMN 框的表达式编辑器，以定义节点的决策逻辑。**

e. **单击 *undefined* 表达式字段，再选择 Decision Table。**

f. **单击决策表的左上角，将点击策略设置为 unique。**

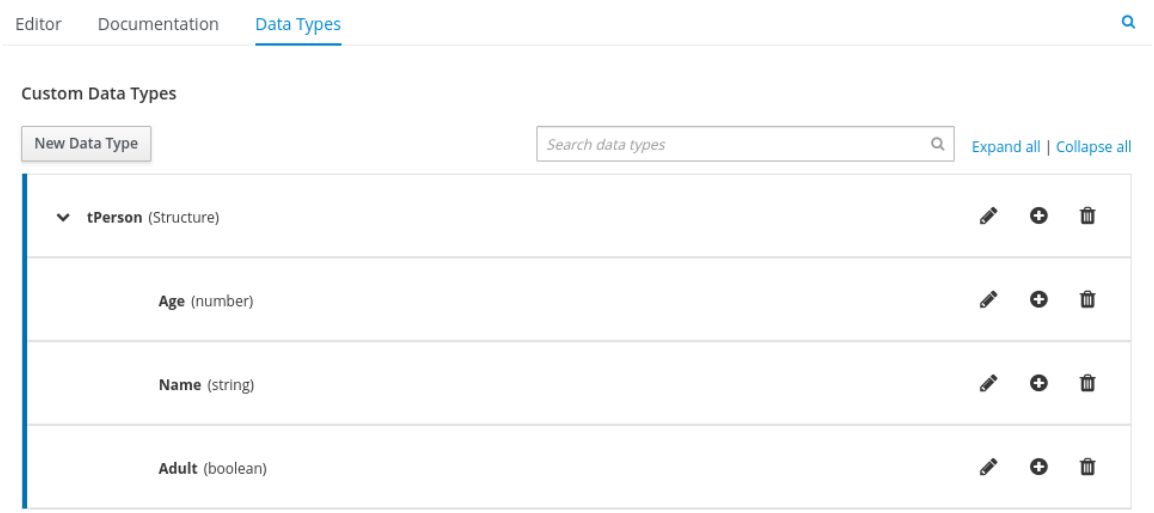
g. **设置输入和输出列，使输入源 Person.Age 使用类型编号 决定年龄限制，输出目标为：带有类型布尔值的 isAdult （类型 布尔值 ）决定了 dult 状态：**

图 5.4. 对 **isAdult** 决策的 DMN 决策示例



h. **在 upper 选项卡中，选择 Data Types 选项卡并添加以下 tPerson 结构化数据类型和嵌套的数据类型：**

图 5.5. DMN 数据类型示例



i. 定义数据类型后，选择 Editor 选项卡以返回到 DMN 模型器。

j. 选择 Person 输入节点，单击 Properties 图标，然后在 Information 项下，将数据类型设置为 tPerson。

k. 选择 isAdult 决策节点，单击 Properties 图标，然后在 Information 项下，确认数据类型是否仍然设置为布尔值。以前，在创建决策表时设置此数据类型。

l. 保存 DMN 决策文件。

5.1. 使用 DRL 规则单元作为替代决策服务

您还可以使用作为规则单元实施的 Drools 规则语言(DRL)文件来定义此示例决策服务，作为使用 Decision Model 和 Notation(DMN)的替代方案。

DRL 规则单元是规则以及执行单元的模块。规则单元收集一组规则，以及规则所针对的事实类型声明。规则单元还充当每个规则组的唯一命名空间。单个规则基础可以包含多个规则单元。您通常会在与单元声明相同的文件中存储单元的所有规则，以便该单元是自包含的。有关规则单元的更多信息，请参阅使用 *DRL 规则设计决策服务*。

先决条件

- 您已创建了一个项目。有关创建 Maven 项目的更多信息，请参阅 *第 3 章 为红帽构建 Kogito 微服务创建 Maven 项目*。

流程

1. 在示例项目的 src/main/resources 文件夹中，不使用 DMN 文件，添加以下PersonRules.drl 文件：

   PersonRules DRL 文件示例

```
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
```

这个示例规则确定任何超过 18 的人被归类为dult。规则文件还声明该规则属于规则单元 **PersonRules**。构建项目时，将生成规则单元并将其与 DRL 文件关联。

该规则还使用 OOPath 表示法定义条件。OOPath 是一个面向对象的语法扩展，用于导航相关的元素，同时处理集合和过滤限制。

您还可以使用传统规则模式语法以更明确的形式重写相同的规则条件，如下例所示：

使用传统表示法的 PersonRules DRL 文件示例

```
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 红帽构建的 KOGITO 事件附加组件

事件附加组件在支持的目标平台中为 EventEmitter 和 EventReceiver 接口提供默认实现。您可以使用 EventEmitter 和 EventReceiver 接口，通过处理、无服务器工作流事件和事件决策处理启用消息。

6.1. 为红帽构建的 KOGITO 事件附加组件实施消息有效负载 DECORATOR

任何依赖的附加组件都可以实施 *MessagePayloadDecorator*。

先决条件

- **您已在 Red Hat build of Kogito 中安装 Events 附加组件。**

流程

1. **在类路径中创建名为 META-INF/services/org.kie.kogito.add-on.cloudevents.message.MessagePayloadDecorator 的文件。**
2. **打开 文件。**
3. **在文件中输入您的实施类的完整名称。**
4. **保存该文件。**
   MessagePayloadDecoratorProvider 在应用程序启动时加载文件，并将该文件添加到 decoration 链中。当红帽构建的 Kogito 调用 MessagePayloadDecoratorProvider#decorate 时，您的实现是分离算法的一部分。
5. **要使用事件附加组件，请将以下代码添加到项目的 pom.xml 文件中：**

   {QAURKUS} 的事件小附加组件

```
<dependency>
  <groupId>org.kie.kogito</groupId>
  <artifactId>kogito-addons-quarkus-events-smallrye</artifactId>
  <version>1.15</version>
</dependency>
```

   {QAURKUS} 的事件决策附加组件

```
<dependency>
  <groupId>org.kie.kogito</groupId>
  <artifactId>kogito-addons-events-decisions</artifactId>
  <version>1.15</version>
</dependency>
```

   为 Spring Boot 的事件 Kafka 附加组件

```
<dependency>
  <groupId>org.kie.kogito</groupId>
  <artifactId>kogito-addons-springboot-events-kafka</artifactId>
  <version>1.15</version>
</dependency>
```

Spring Boot 的事件决策附加组件

```
<dependency>
  <groupId>org.kie.kogito</groupId>
  <artifactId>kogito-addons-springboot-events-decisions</artifactId>
  <version>1.15</version>
</dependency>
```

第 7 章 运行红帽构建 KOGITO 微服务

为 Red Hat build of Kogito 微服务设计业务决策后，您可以使用以下模式之一运行 Red Hat build of Quarkus 或 Spring Boot 应用程序：

- 开发模式 ：进行本地测试。在红帽构建的 Quarkus 上，开发模式还在运行的应用程序中实时重新载入您的决策，以进行高级调试。
- JVM 模式 ：与 Java 虚拟机(JVM)兼容。

流程

在命令终端中，进入包含 Red Hat build of Kogito 微服务的项目，并输入以下命令之一，具体取决于您首选的运行模式和应用程序环境：

- 对于开发模式：

  在 Red Hat build of Quarkus 上

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

  在 Sprint Boot 中

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

- 对于 JVM 模式：

  在 Red Hat build of Quarkus 和 Spring Boot 上

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

第 8 章 与正在运行的红帽构建 KOGITO 微服务交互

在 Red Hat build of Kogito 微服务运行后，您可以发送 REST API 请求以与应用程序进行交互并根据设置应用程序执行您的微服务。

这个示例测试了在 PersonDecisions.dmn 文件中自动生成决策的 /persons REST API 端点（如果您使用 DRL 规则单元，则为 PersonRules.drl 文件中的规则）。

在本例中，使用 REST 客户端、curl 程序或为应用程序配置的 Swagger UI（如 http://localhost:8080/q/swagger-ui 或 http://localhost:8080/swagger-ui.html）来发送带有以下组件的 API 请求：

- URL:http://localhost:8080/persons
- HTTP 标头 ：仅用于 POST 请求：
  - 接受:application/json
  - content-type:application/json
- HTTP 方法 ：GET、POST 或 DELETE

添加dult(JSON)的 POST 请求正文示例

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

添加 adult 的 curl 命令示例

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

响应(JSON)示例

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

这个示例步骤使用 curl 命令进行方便。

流程

在与正在运行的应用程序分开的命令终端窗口中，导航到包含 Red Hat build of Kogito 微服务的项目，并使用以下 curl 命令和 JSON 请求来与正在运行的微服务交互：

注意

在 Spring Boot 上，您可能需要修改应用程序如何公开 API 端点，以便这些示例请求正常工作。如需更多信息，请参阅您为本教程创建的 Spring Boot 项目示例中包含的 README 文件。

- **添加一个dult 用户：**

  请求示例

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

  响应示例

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

- **添加一个不足人：**

  请求示例

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

  响应示例

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

- **使用返回的 UUID 完成评估：**

  请求示例

  ```
  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 '{}'
  ```

部分 II. 在 RED HAT OPENSHIFT CONTAINER PLATFORM 上部署 RED HAT BUILD OF KOGITO 微服务

作为业务决策和流程的开发人员，您可以在 Red Hat OpenShift Container Platform 上部署 Kogito 微服务构建以实现云实施。RHPAM Kogito Operator 会自动为您完成部署过程进行多种部署步骤。

先决条件

- 安装了 Red Hat OpenShift Container Platform 4.6 或 4.7。
- 为部署创建 OpenShift 项目。

第 9 章 RED HAT BUILD OF KOGITO ON RED HAT OPENSHIFT CONTAINER PLATFORM

您可以在 Red Hat OpenShift Container Platform 上部署红帽构建 Kogito 微服务，以实现云实施。在这个架构中，红帽构建的 Kogito 微服务作为 OpenShift pod 进行部署，您可单独扩展和缩减，以根据特定服务所需的数量提供数个或数量。

为了帮助您在 OpenShift 上部署 Red Hat build of Kogito 微服务，Red Hat Process Automation Manager 提供了 Red Hat Process Automation Manager Kogito Operator。该 operator 指导您完成部署过程。Operator 基于 Operator SDK，并为您自动执行许多部署步骤。例如，当为 Operator 提供包含应用程序的 Git 存储库链接时，Operator 会自动配置从源构建项目所需的组件，并部署生成的服务。

要在 OpenShift Web 控制台中安装 Red Hat Process Automation Manager Kogito Operator，进入左侧菜单中的 Operators → OperatorHub，搜索 RHPAM Kogito Operator，并根据屏幕的说明安装最新的 Operator 版本。

第 10 章 带有 RHPAM KOGITO OPERATOR 的 OPENSHIFT 部署选项

在将 Kogito 微服务构建为业务应用程序的一部分后，您可以使用 Red Hat OpenShift Container Platform Web 控制台来部署微服务。OpenShift Web 控制台中的 RHPAM Kogito Operator 页面会指导您完成部署过程。

RHPAM Kogito Operator 支持以下选项，用于在 Red Hat OpenShift Container Platform 上构建和部署 Kogito 微服务：

- Git 源构建和部署
- 二进制构建和部署
- 自定义镜像构建和部署
- 文件构建和部署

10.1. 使用 GIT 源构建和 OPENSHIFT WEB 控制台在 OPENSHIFT 上部署 RED HAT BUILD OF KOGITO 微服务

RHPAM Kogito Operator 使用以下自定义资源来部署特定于域的微服务（您开发的微服务）：

- **KogitoBuild** 使用 Git URL 或其他源构建应用，并生成运行时镜像。
- **KogitoRuntime** 启动运行时镜像，并根据要求进行配置。

在大多数用例中，您可以使用标准运行时构建和部署方法，从 Git 存储库源在 OpenShift 上部署 Kogito 微服务，如以下步骤所示。

注意

如果您要在本地开发或测试 Red Hat build of Kogito 微服务，您可以使用二进制构建、自定义镜像构建或文件构建选项从本地源而不是从 Git 存储库构建和部署。

先决条件

- 已安装 RHPAM Kogito Operator。
- 带有 Red Hat build of Kogito 微服务的应用位于 Git 存储库中，可从您的 OpenShift 环境访问。
- 您可以访问 OpenShift Web 控制台，具有创建并编辑 **KogitoBuild** 和 **KogitoRuntime** 所需的权限。
- （仅红帽构建的 Quarkus）项目的 **pom.xml** 文件包含了对 **quarkus-smallrye-health** 扩展的以下依赖项。此扩展启用 OpenShift 中红帽构建 Quarkus 项目所需的 存活度和就绪度探测。

  在 OpenShift 中红帽构建的 Quarkus 应用程序的 SmallRye Health 依赖项

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

流程

1. 进入 Operators → Installed Operators 并选择 RHPAM Kogito Operator。
2. 要在 operator 页面上创建 Red Hat build of Kogito build 定义，请选择 Kogito Build选项卡，然后点击 Create KogitoBuild。
3. 在应用程序窗口中，使用 Form View 或 YAML View 来配置构建定义。
至少，定义以下示例 YAML 文件中显示的应用程序配置：

带有红帽构建的 Kogito 构建的红帽 Quarkus 应用程序的 YAML 定义示例

```
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
```

带有红帽构建的 Spring Boot 应用程序的 YAML 定义示例

```
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
```

注意

如果您配置了内部 Maven 存储库，您可以将其用作 Maven mirror 服务，并在 Red Hat build of Kogito 构建定义中指定 Maven mirror URL 来缩短构建时间：

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

有关内部 Maven 存储库的更多信息，请参阅 Apache Maven 文档。

4. 定义应用程序数据后，点 Create 生成 Red Hat build of Kogito 构建。
您的应用程序在 红帽构建的 KogitoBuilds 页面中列出。您可以选择应用程序名称来查看或修改应用程序设置和 YAML 详情。
5. 要创建 Red Hat build of Kogito 微服务定义，请在 operator 页面中选择 Kogito Runtime选项卡，然后点击 Create KogitoRuntime。

6. 在应用窗口中，使用 Form View 或 YAML View 来配置微服务定义。
至少，定义以下示例 YAML 文件中显示的应用程序配置：

带有红帽构建的 Kogito 微服务构建的 Red Hat build 的 YAML 定义示例

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

带有红帽构建的 Kogito 微服务的 Spring Boot 应用程序的 YAML 定义示例

```
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
```

注意

在这种情况下，应用程序是从 Git 构建并使用 KogitoRuntime 进行部署。您必须确保应用程序名称在 KogitoBuild 和 KogitoRuntime 中相同。

7. 定义应用程序数据后，点 Create 生成 Red Hat build of Kogito 微服务。
您的应用程序在红帽构建的 Kogito 微服务页面中列出。您可以选择应用程序名称来查看或修改应用程序设置以及 YAML 文件的内容。

8. 在 Web 控制台的左侧菜单中，转至 Builds → Builds 以查看应用程序构建的状态。
您可以选择特定的构建来查看构建详情。

注意

对于您为 OpenShift 部署创建的每个红帽构建 Kogito 微服务，Web 控制台的 Builds 页面会生成并列出两个构建：传统运行时构建和 Source-to-Image(S2I)构建，后缀 **-builder**。S2I 机制在 OpenShift 构建中构建应用，然后将构建的应用传递到下一 OpenShift 构建，以打包至运行时容器镜像。红帽构建的 Kogito S2I 构建配置还允许您直接从 OpenShift 平台上的 Git 存储库构建项目。

9. 应用程序构建完成后，进入 Workloads → Deployments 来查看应用程序部署、pod 状态和其他详情。

10. 部署完 Red Hat build of Kogito microservice 后，在 web 控制台的左侧菜单中，转至 Networking → Routes 以查看对部署的应用程序的访问链接。
您可以选择应用程序名称来查看或修改路由设置。

通过应用程序路由，您可以根据需要将 Kogito 微服务构建与业务自动化解决方案集成。

10.2. 使用二进制构建和 OPENSHIFT WEB 控制台在 OPENSHIFT 上部署 RED HAT BUILD OF KOGITO 微服务

OpenShift 构建可能需要大量时间。作为在 OpenShift 中构建和部署红帽构建的 Kogito 微服务的更快选择，您可以使用二进制构建。

Operator 使用以下自定义资源来部署特定于域的微服务（您开发的微服务）：

- **KogitoBuild** 处理上传的应用程序并生成运行时镜像。
- **KogitoRuntime** 启动运行时镜像，并根据要求进行配置。

先决条件

- 已安装 RHPAM Kogito Operator。
- 已安装 **oc** OpenShift CLI，并登录到相关的 OpenShift 集群。有关 **oc** 安装和登录说明，请参阅 OpenShift 文档。
- 您可以访问 OpenShift Web 控制台，具有创建并编辑 **KogitoBuild** 和 **KogitoRuntime** 所需的权限。
- （仅红帽构建的 Quarkus）项目的 **pom.xml** 文件包含了对 **quarkus-smallrye-health** 扩展的以下依赖项。此扩展启用 OpenShift 中红帽构建 Quarkus 项目所需的 存活度和就绪度探测。

  在 OpenShift 中红帽构建的 Quarkus 应用程序的 SmallRye Health 依赖项

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

流程

1. 本地构建应用程序。
2. 进入 Operators → Installed Operators 并选择 RHPAM Kogito Operator。
3. 要在 operator 页面上创建 Red Hat build of Kogito build 定义，请选择 Kogito Build 选项卡，然后点击 Create KogitoBuild。
4. 在应用程序窗口中，使用 Form View 或 YAML View 来配置构建定义。
   至少，定义以下示例 YAML 文件中显示的应用程序配置：

   带有红帽构建的 Kogito 构建的红帽 Quarkus 应用程序的 YAML 定义示例

```
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
```

   带有红帽构建的 Spring Boot 应用程序的 YAML 定义示例

```
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. 定义应用程序数据后，点 Create 生成 Red Hat build of Kogito 构建。
您的应用程序在 红帽构建的 KogitoBuilds 页面中列出。您可以选择应用程序名称来查看或修改应用程序设置和 YAML 详情。

6. 使用以下命令上传构建的二进制文件：

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

- **from-dir** 等于所构建应用程序的目标 文件夹路径。
- **namespace** 是创建 **KogitoBuild** 的命名空间。

7. 要创建 Red Hat build of Kogito 微服务定义，请在 operator 页面中选择 Kogito Runtime选项卡，然后点击 Create KogitoRuntime。

8. 在应用窗口中，使用 Form View 或 YAML View 来配置微服务定义。
至少，定义以下示例 YAML 文件中显示的应用程序配置：

带有红帽构建的 Kogito 微服务构建的 Red Hat build 的 YAML 定义示例

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

带有红帽构建的 Kogito 微服务的 Spring Boot 应用程序的 YAML 定义示例

```
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
```

注意

在这种情况下，应用程序在本地构建并使用 KogitoRuntime 进行部署。您必须确保应用程序名称在 **KogitoBuild** 和 **KogitoRuntime** 中相同。

9. 定义应用程序数据后，点 Create 生成 Red Hat build of Kogito 微服务。
您的应用程序在红帽构建的 Kogito 微服务页面中列出。您可以选择应用程序名称来查看或修改应用程序设置以及 YAML 文件的内容。

10. 在 Web 控制台的左侧菜单中，转至 Builds → Builds 以查看应用程序构建的状态。
您可以选择特定的构建来查看构建详情。

11. 应用程序构建完成后，进入 Workloads → Deployments 来查看应用程序部署、pod 状态和其他详情。

12. 部署完 Red Hat build of Kogito microservice 后，在 web 控制台的左侧菜单中，转至 Networking → Routes 以查看对部署的应用程序的访问链接。
您可以选择应用程序名称来查看或修改路由设置。

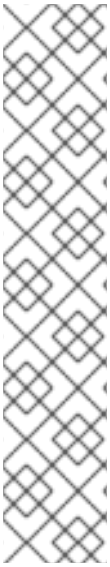
    通过应用程序路由，您可以根据需要将 Kogito 微服务构建与业务自动化解决方案集成。

10.3. 使用自定义镜像构建和 OPENSHIFT WEB 控制台在 OPENSHIFT 上部署 RED HAT KOGITO 微服务

您可以使用自定义镜像构建作为在 OpenShift 中构建和部署 Kogito 微服务的替代选择。

Operator 使用以下自定义资源来部署特定于域的微服务（您开发的微服务）：

- **KogitoRuntime** 启动运行时镜像，并根据要求进行配置。



注意

Red Hat Process Automation Manager 构建器镜像不支持原生构建。但是，您可以执行自定义构建并使用 **Containerfile** 构建容器镜像，如下例所示：

```
FROM registry.redhat.io/rhpam-7-tech-preview/rhpam-kogito-runtime-native-rhel8:7.13.5

ENV RUNTIME_TYPE quarkus

COPY --chown=1001:root target/*-runner $KOGITO_HOME/bin
```

此功能只是技术预览。

要使用 Mandrel 构建原生二进制文件，请参阅将 Quarkus 应用程序编译为原生可执行文件。

先决条件

- 已安装 RHPAM Kogito Operator。
- 您可以访问 OpenShift Web 控制台，具有创建并编辑 **KogitoRuntime** 所需的权限。
- （仅红帽构建的 Quarkus）项目的 **pom.xml** 文件包含了对 **quarkus-smallrye-health** 扩展的以下依赖项。此扩展启用 OpenShift 中红帽构建 Quarkus 项目所需的 存活度和就绪度探测。

  在 OpenShift 中红帽构建的 Quarkus 应用程序的 SmallRye Health 依赖项

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

流程

1. 本地构建应用程序。
2. 在项目根目录中创建 **Containerfile** 并包含以下内容：

   Red Hat build of Quarkus 应用程序的 Containerfile 示例

```
FROM registry.redhat.io/rhpam-7/rhpam-kogito-runtime-jvm-rhel8:7.13.5

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 应用程序的 Containerfile 示例

```
FROM registry.redhat.io/rhpam-7/rhpam-kogito-runtime-jvm-rhel8:7.13.5

ENV RUNTIME_TYPE springboot

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

- **application-jar-file 是应用的 JAR 文件的名称。**

3. **使用以下命令构建 Kogito 镜像的红帽构建：**

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

在上一命令中，final-image-name 是 Red Hat build of Kogito 镜像，Container-file 是您在上一步中创建的 Containerfile 的名称。

4. 另外，还可使用以下命令测试构建的镜像：

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

5. **使用以下命令将构建的 Red Hat build of Kogito 镜像推送到镜像 registry：**

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

6. **进入 Operators → Installed Operators 并选择 RHPAM Kogito Operator。**

7. **要创建 Red Hat build of Kogito 微服务定义，请在 operator 页面中选择 Kogito Runtime 选项卡，然后点击 Create KogitoRuntime。**

8. **在应用窗口中，使用 Form View 或 YAML View 来配置微服务定义。**
至少，定义以下示例 YAML 文件中显示的应用程序配置：

带有红帽构建的 Kogito 微服务构建的 Red Hat build 的 YAML 定义示例

```
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
```

带有红帽构建的 Kogito 微服务的 Spring Boot 应用程序的 YAML 定义示例

```
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. 定义应用程序数据后，点 Create 生成 Red Hat build of Kogito 微服务。
您的应用程序在红帽构建的 Kogito 微服务页面中列出。您可以选择应用程序名称来查看或修改应用程序设置以及 YAML 文件的内容。

10. 应用程序构建完成后，进入 Workloads → Deployments 来查看应用程序部署、pod 状态和其他详情。

11. 部署完 Red Hat build of Kogito microservice 后，在 web 控制台的左侧菜单中，转至 Networking → Routes 以查看对部署的应用程序的访问链接。
您可以选择应用程序名称来查看或修改路由设置。

    通过应用程序路由，您可以根据需要将 Kogito 微服务构建与业务自动化解决方案集成。

10.4. 使用文件构建和 OPENSHIFT WEB 控制台在 OPENSHIFT 上部署 RED HAT KOGITO 微服务

您可以从单个文件构建和部署 Kogito 微服务构建，如 Decision Model 和 Notation(DMN)、CEP Rule Language(DRL)或属性文件，或者从含有多个文件的目录构建和部署。您可以从本地文件系统路径中指定单个文件，或者只指定来自本地文件系统路径的文件目录。将文件或目录上传到 OpenShift 集群时，会自动触发新的 Source-to-Image(S2I)构建。

Operator 使用以下自定义资源来部署特定于域的微服务（您开发的微服务）：

- **KogitoBuild** 从文件中生成应用程序并生成运行时镜像。
- **KogitoRuntime** 启动运行时镜像，并根据要求进行配置。

先决条件

- 已安装 RHPAM Kogito Operator。
- 已安装 oc OpenShift CLI，并登录到相关的 OpenShift 集群。有关 oc 安装和登录说明，请参阅 OpenShift 文档。
- 您可以访问 OpenShift Web 控制台，具有创建并编辑 KogitoBuild 和 KogitoRuntime 所需的权限。

流程

1. 进入 Operators → Installed Operators 并选择 RHPAM Kogito Operator。
2. 要在 operator 页面上创建 Red Hat build of Kogito build 定义，请选择 Kogito Build选项卡，然后点击 Create KogitoBuild。

3. 在应用程序窗口中，使用 Form View 或 YAML View 来配置构建定义。
至少，定义以下示例 YAML 文件中显示的应用程序配置：

带有红帽构建的 Kogito 构建的红帽 Quarkus 应用程序的 YAML 定义示例

```
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
```

带有红帽构建的 Spring Boot 应用程序的 YAML 定义示例

```
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
```

注意

如果您配置了内部 Maven 存储库，您可以将其用作 Maven mirror 服务，并在 Red Hat build of Kogito 构建定义中指定 Maven mirror URL 来缩短构建时间：

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

有关内部 Maven 存储库的更多信息，请参阅 Apache Maven 文档。

4. 定义应用程序数据后，点 Create 生成 Red Hat build of Kogito 构建。
您的应用程序在 红帽构建的 KogitoBuilds 页面中列出。您可以选择应用程序名称来查看或修改应用程序设置和 YAML 详情。

5. 使用以下命令上传文件资产：

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

- file-asset-path 是您要上传的文件内容的路径。
- namespace 是创建 KogitoBuild 的命名空间。

6. 要创建 Red Hat build of Kogito 微服务定义，请在 operator 页面中选择 Kogito Runtime 选项卡，然后点击 Create KogitoRuntime。

7. 在应用窗口中，使用 Form View 或 YAML View 来配置微服务定义。
至少，定义以下示例 YAML 文件中显示的应用程序配置：

带有红帽构建的 Kogito 微服务构建的 Red Hat build 的 YAML 定义示例

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

带有红帽构建的 Kogito 微服务的 Spring Boot 应用程序的 YAML 定义示例

```
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
```

注意

在这种情况下，应用程序是从文件构建并使用 KogitoRuntime 进行部署。您必须确保应用程序名称在 KogitoBuild 和 KogitoRuntime 中相同。

8. 定义应用程序数据后，点 Create 生成 Red Hat build of Kogito 微服务。
您的应用程序在红帽构建的 Kogito 微服务页面中列出。您可以选择应用程序名称来查看或修改应用程序设置以及 YAML 文件的内容。

9. 在 Web 控制台的左侧菜单中，转至 Builds → Builds 以查看应用程序构建的状态。
您可以选择特定的构建来查看构建详情。

注意

对于您为 OpenShift 部署创建的每个红帽构建 Kogito 微服务，Web 控制台的 Builds 页面会生成并列出两个构建：传统运行时构建和 Source-to-Image(S2I)构建，后缀 **-builder**。S2I 机制在 OpenShift 构建中构建应用，然后将构建的应用传递到下一 OpenShift 构建，以打包至运行时容器镜像。

10. 应用程序构建完成后，进入 Workloads → Deployments 来查看应用程序部署、pod 状态和其他详情。

11. 部署完 Red Hat build of Kogito microservice 后，在 web 控制台的左侧菜单中，转至 Networking → Routes 以查看对部署的应用程序的访问链接。
您可以选择应用程序名称来查看或修改路由设置。

通过应用程序路由，您可以根据需要将 Kogito 微服务构建与业务自动化解决方案集成。

第 11 章 RED HAT BUILD OF KOGITO 服务属性配置

部署红帽 Kogito 微服务构建时，将为红帽构建 Kogito 微服务构建的 application.properties 配置创建一个 configMap 资源。

configMap 资源的名称包括红帽构建 Kogito 微服务的名称以及后缀-properties，如下例所示：

红帽构建 Kogito 微服务部署期间生成的 configMap 资源示例

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: kogito-travel-agency-properties
data:
  application.properties : |-
    property1=value1
    property2=value2
```

configMap 资源的 application.properties 数据挂载到红帽构建 Kogito 微服务的容器中。您添加到 application.properties 部分的任何运行时属性会覆盖 Red Hat build of Kogito 微服务的默认应用程序配置属性。

当更改 configMap 的 application.properties 数据时，滚动更新会修改 Red Hat build of Kogito 微服务的部署和配置。

第 12 章 RED HAT OPENSHIFT CONTAINER PLATFORM 上的 RED HAT BUILD OF KOGITO 探测

Red Hat OpenShift Container Platform 中的探测验证应用程序是否正常工作或需要重启。对于在红帽构建的 Quarkus 和 Spring Boot 上的红帽构建 Kogito 微服务构建，探测通过 HTTP 请求与应用程序交互，默认与扩展所公开的端点交互。因此，若要在 Red Hat OpenShift Container Platform 上运行 Red Hat build of Kogito 微服务，您必须导入扩展，以便为 存活度、就绪度和启动探测 提供应用程序可用性信息。

12.1. 在 RED HAT OPENSHIFT CONTAINER PLATFORM 上为 RED HAT BUILD OF QUARKUS 应用程序添加健康检查扩展

您可以为基于 Red Hat OpenShift Container Platform 上的 Quarkus 应用程序的 Kogito 服务添加健康检查扩展。

流程

在命令终端中，导航到项目的 pom.xml 文件，并为 quarkus-smallrye-health 扩展添加以下依赖项：

用于 Red Hat OpenShift Container Platform 上红帽构建的 Quarkus 应用程序的 SmallRye Health 依赖项

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

12.2. 在 RED HAT OPENSHIFT CONTAINER PLATFORM 上为 SPRING BOOT 应用程序添加健康检查扩展

您可以为基于 Red Hat OpenShift Container Platform 的 Spring Boot 的 Kogito 微服务构建添加健康检查扩展。

流程

在命令终端中，导航到项目的 pom.xml 文件，并添加以下 Spring Boot actuator 依赖项：

Red Hat OpenShift Container Platform 上的 Spring Boot actuator 依赖项

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

12.3. 为 RED HAT OPENSHIFT CONTAINER PLATFORM 上的红帽构建 KOGITO 微服务设置自定义探测

您还可以为存活度、就绪度和启动探测配置自定义端点。

流程

1. **在项目的 KogitoRuntime YAML 文件中定义探测，如下例所示：**

使用自定义探测端点构建的 Kogito 微服务自定义资源示例

```
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 13 章 RED HAT PROCESS AUTOMATION MANAGER KOGITO OPERATOR 与 PROMETHEUS 和 GRAFANA 交互

Red Hat Process Automation Manager 中的 Red Hat build of Kogito 提供了一个 **monitoring-prometheus-addon** 附加组件，它允许 Prometheus metrics 监控 Red Hat build of Kogito 微服务的红帽构建的 Grafana 仪表板，并使用附加组件导出的默认指标。RHPAM Kogito Operator 使用 Prometheus Operator 来公开项目的指标，以便 Prometheus 提取。由于这个依赖项，Prometheus Operator 必须安装到与项目相同的命名空间中。

如果要为 Red Hat build of Kogito 微服务启用 Prometheus 指标数据监控，根据您的项目中的 **pom.xml** 文件添加以下依赖关系，具体取决于您使用的框架：

Prometheus Red Hat build of Quarkus 附加组件的依赖项

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

Prometheus Spring Boot 附加组件的依赖关系

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

当您部署使用 **monitoring-prometheus-addon** 附加组件和 Prometheus Operator 的 Red Hat build 的 Red Hat build 时，Red Hat Process Automation Manager Kogito Operator 会创建一个 **ServiceMonitor** 自定义资源来公开 Prometheus 的指标，如下例所示：

Prometheus 的 ServiceMonitor 资源示例

```
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
```

您必须手动配置由 **Prometheus** Operator 管理的 Prometheus 自定义资源，以选择**ServiceMonitor** 资源：

Prometheus 资源示例

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

使用 ServiceMonitor 资源配置 Prometheus 资源后，您可以在 Prometheus web 控制台的Targets 页面中看到 Prometheus 提取的端点。Red Hat Process Automation Manager 服务公开的指标会出现在 Graph 视图中。

RHPAM Kogito Operator 还会为附加组件生成的每个 Grafana 仪表板创建一个 GrafanaDashboard 自定义资源。https://operatorhub.io/operator/grafana-operator仪表板 的应用程序标签是部署的红帽构建 Kogito 微服务的名称。您必须根据相关的红帽构建 Kogito 微服务来设置 Grafana 自定义资源的 dashboardLabelSelector 属性。

Grafana 资源示例

```
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 14 章 RED HAT PROCESS AUTOMATION MANAGER RED HAT BUILD OF KOGITO OPERATOR 与 KAFKA 交互

Red Hat Process Automation Manager Red Hat build of Kogito Operator 使用 AMQ Streams Operator 使用 Kafka 自动配置 Kogito 微服务构建。

当您通过 KogitoInfra 部署启用基础架构机制时，Red Hat Process Automation Manager Red Hat build of Kogito Operator 会使用相关的第三方 Operator 来配置基础架构。

您必须定义自定义基础架构资源，并将其链接到 KogitoInfra 文件中。您可以在 spec.resource.name 和 spec.resource.namespace 配置中指定您的自定义基础架构资源。

Red Hat Process Automation Manager Red Hat build of Kogito infrastructure resource for custom messaging

```
apiVersion: rhpam.kiegroup.org/v1 # Red Hat build of Kogito API for this microservice
kind: KogitoInfra # Application type
metadata:
  name: my-kafka-infra
spec:
  resource:
    apiVersion: kafka.strimzi.io/v1beta2 # AMQ Streams API
    kind: Kafka # AMQ Streams Application Type
    name: my-kafka-instance
    namespace: my-namespace
```

在本例中，Kog itoInfra 自定义资源从 my-namespace 用于事件消息传递连接到 Kafka 集群 my-kafka-instance。

要将 Red Hat build of Kogito 微服务连接到 Kafka，您需要定义 infra 配置以使用对应的基础架构。

使用消息传递的红帽构建 Kogito 微服务资源配置示例

```
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
  infra:
  - my-kafka-infra
```

Red Hat Process Automation Manager Red Hat build of Kogito Operator 配置必要的属性，以便应用程序可以连接到 Kafka 实例。

第 15 章 红帽构建的 KOGITO 微服务部署故障排除

使用本节中的信息对使用 operator 部署 Red Hat build of Kogito 微服务时可能会遇到的问题进行故障排除。以下信息会在新问题中更新，并发现临时解决方案。

没有构建正在运行

如果您没有看到运行任何构建以及相关命名空间中创建的任何资源，请输入以下命令来检索运行的 pod，并查看 pod 的 operator 日志：

查看指定 pod 的 RHPAM Kogito Operator 日志

```
// 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
```

验证 KogitoRuntime 状态

如果您创建，请使用以下 YAML 定义，带有非已存在的镜像的 KogitoRuntime 应用程序：

KogitoRuntime 应用程序的 YAML 定义示例

```
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
```

您可以使用 bash 控制台中的 oc describe KogitoRuntime 命令验证 KogitoRuntime 应用程序的状态。当您在 bash 控制台中运行 oc describe KogitoRuntime 示例 命令时，您会收到以下输出：

KogitoRuntime 状态示例

```
[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
```

在输出的末尾，您可以看到包含相关消息的 KogitoRuntime 状态。

部分 III. 迁移到红帽构建 KOGITO 微服务

作为业务决策和流程的开发人员，您可以将红帽流程自动化管理器中的决策服务迁移到 Red Hat build of Kogito 微服务。执行迁移时，您现有的业务决策成为您自己的域特定云原生服务的一部分。您可以迁移决策模型和 Notation(DMN)模型、预测模型标记语言(PMML)模型，或者 Drools 规则语言(DRL)规则。

先决条件

- JDK 11 或更高版本已经安装。
- 已安装 Apache Maven 3.6.2 或更高版本。

第 16 章 迁移到红帽构建的 KOGITO 微服务概述

您可以将您在 Business Central 中开发的决策服务构件迁移到红帽构建 Kogito 微服务。Red Hat build of Kogito 目前支持对以下决策服务的迁移：

- 决策模型和符号(DMN)模型 ：您可以通过将 DMN 资源从 KJAR 工件移至相应的 Red Hat build of Kogito archetype 来迁移基于 DMN 的决策服务。
- 预测模型标记语言(PMML)模式 ：您可以通过将 PMML 工件中的 PMML 资源移到相应的 Red Hat build of Kogito archetype 中，迁移 PMML 预测和预测服务。
- Drools Rule Language(DRL)规则 ：将基于 DRL 的决策服务包括在红帽 Quarkus REST 端点中。这个迁移的方法可让您使用主要的 Quarkus 功能，如热插拔和原生编译。Quarkus 功能和 Red Hat build of Kogito 的编程模型启用了在应用程序和服务中实施的 Quarkus REST 端点的自动生成。

第 17 章 将 DMN 服务迁移到红帽构建的 KOGITO 微服务

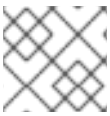
您可以通过将 DMN 资源从 KJAR 工件移到相应的 Red Hat build of Kogito 项目，将基于 DMN 的决策服务迁移到红帽构建 Kogito。在 Red Hat build of Kogito 微服务中，不再需要某些 KIE v7 功能。

17.1. 主要变化和迁移注意事项

下表描述了影响从 KIE 服务器 API 和 KJAR 到 Red Hat build of Kogito 部署的主要变化和功能：

表 17.1. DMN 迁移注意事项

功能	在 KIE 服务器 API 中	在红帽构建的 Kogito 工件中
DMN 模型	存储在 KJAR 的 **src/main/resources**.	将 复制到 **src/main/resources**。
KIE 服务器通用摘要所需的对象模型(OVA)	在 Business Central 中使用数据模型对象编辑器管理。	不再需要对象模型编辑。
DMNRuntimeListener	使用系统属性或 **kmodule.xml** 文件进行配置。	必须使用 CDI 配置，方法是使用 CDI 的 **@ApplicationScope** 注释为 **DMNRuntimeEventListener** 标注。
其他配置选项	使用系统属性或 **kmodule.xml** 文件进行配置。	除 **DMNRuntimeEventListener** 外，只考虑使用默认值，且不支持覆盖配置。
KIE 服务器客户端 API	与对象模型结合使用，与 KIE 服务器上部署的 KJAR 交互。	对于对象模型，不再需要此功能。  **注意** 您可以选择自己的 REST 库选择。
REST API	当在 KIE 服务器上部署 KJAR 时，与特定 DMN 模型端点交互的应用程序，请在红帽构建 Kogito 部署中使用相同的 API。	对特定 DMN 模型生成的高级支持。如需更多信息，请参阅 *DMN 模型执行*。
测试场景	使用 JUnit 激活器运行。	类似的 JUnit 激活器在红帽构建的 Kogito 上可用。



注意

上表中未提到的功能在云原生红帽构建的 Kogito 部署中不受支持或不需要。

17.2. 迁移策略

将 DMN 项目迁移到 Red Hat build of Kogito 项目时，首先您可以在 KIE 服务器上迁移与决策服务交互的

外部应用程序。您可以使用特定于 DMN 模型的 REST 端点。使用 REST 端点后，您可以将外部应用程序从 KIE 服务器迁移到红帽构建 Kogito 部署。有关特定于 DMN 模型的 REST 端点的更多信息，请参阅特定 *DMN 模型的 REST 端点*。

迁移策略包括以下步骤：

1. 使用 KIE 服务器 API 将现有外部应用程序从通用 KIE 服务器 API 迁移到特定的 DMN REST 端点。
2. 将 KIE 服务器上部署的 KJAR 迁移至 Red Hat build of Kogito 微服务。
3. 使用 Red Hat OpenShift Container Platform 部署红帽 Kogito 微服务构建。
4. 重新连接外部应用程序，并将 REST API 消耗从特定的 DMN REST 端点迁移到 Red Hat build of Kogito 部署。

17.3. 将外部应用程序迁移到特定于 DMN 模型的 REST 端点

要将 DMN 项目迁移到红帽构建 Kogito 部署，首先可以迁移使用特定 DMN REST 端点的外部应用程序，以便与 KIE 服务器上的决策服务交互。

流程

1. 如果您在外部应用程序中使用 REST 端点，请使用 **GET /server/containers/{containerId}/dmn/openapi.json(|.yaml)** 端点检索 KJAR 的 Swagger 或 OAS 规格文件。
   有关特定 DMN 模型的 REST 端点的更多信息，请参阅特定 *DMN 模型的 REST 端点*。
2. 在您的外部应用程序中，选择要与决策服务交互的 Java 或 JDK 库。您可以使用特定 KJAR 的 REST 端点与决策服务交互。

注意

KIE 服务器客户端 Java API 不支持迁移到 Red Hat build of Kogito 部署。

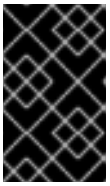
17.4. 将 DMN 模型 KJAR 迁移至 RED HAT BUILD OF KOGITO 微服务

迁移外部应用程序后，您需要将特定于 DMN 模型的 KJAR 迁移到 Red Hat build of Kogito 微服务。

流程

1. 为您的红帽构建 Kogito 微服务创建一个 Maven 项目。
   有关创建 Maven 项目的步骤，*请参阅为红帽构建 Kogito 微服务创建 Maven 项目*。

   Maven 项目创建 Kogito 工件。
2. 将 KJAR 的 **src/main/resources** 文件夹中的 DMN 模型复制到 Kogito 工件的 **src/main/resources** 文件夹。
3. 将 KJAR 的 **src/test/resources** 文件夹的测试场景复制到 Kogito 工件的**src/test/resources** 文件夹。



重要

您需要在项目 pom.xml 文件中导入 Kogito 依赖关系的 Kogito 依赖项，并使用 KIE 服务器 REST API 创建 JUnit 激活者。如需更多信息，请参阅使用 *测试场景 测试决策服务*。

4. **运行以下命令，并确保测试场景正在为指定的非出口测试运行。**

```
mvn clean install
```

运行 Red Hat build of Kogito 应用程序后，您可以检索 Swagger 或 OAS 规格文件。Swagger 或 OAS 规格提供与 REST 端点相同的信息，以及以下实施详情：

- **提供 API 的服务器的基本 URL**
- **引用架构名称**

当外部应用程序重新路由到新 URL 时，您可以使用提供的实现详情。

将 DMN 模型 KJAR 迁移到 Red Hat build of Kogito 微服务后，您需要使用 Red Hat OpenShift Container Platform 部署微服务。有关 Openshift 的部署选项，请参阅使用 *RHPAM Kogito Operator 的 OpenShift 部署选项*。

17.4.1. 将 DMN 模型 KJAR 迁移到 Red Hat build of Kogito 微服务的示例

以下是将 DMN 模型 KJAR 迁移到 Red Hat build of Kogito 微服务的示例：

图 17.1. 使用 DMN 模型实施的决策服务示例

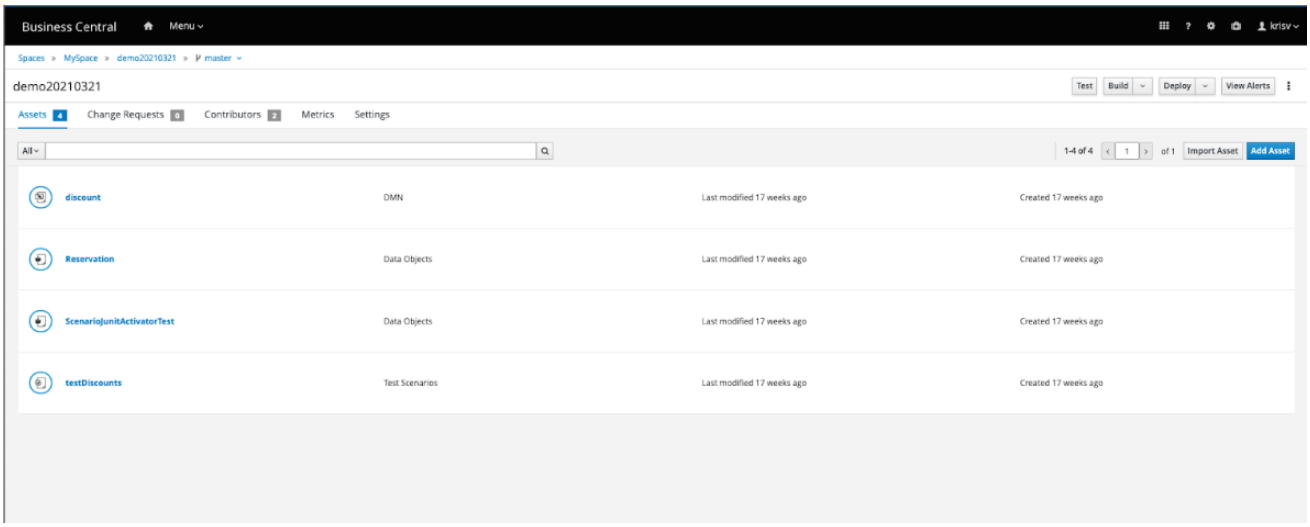
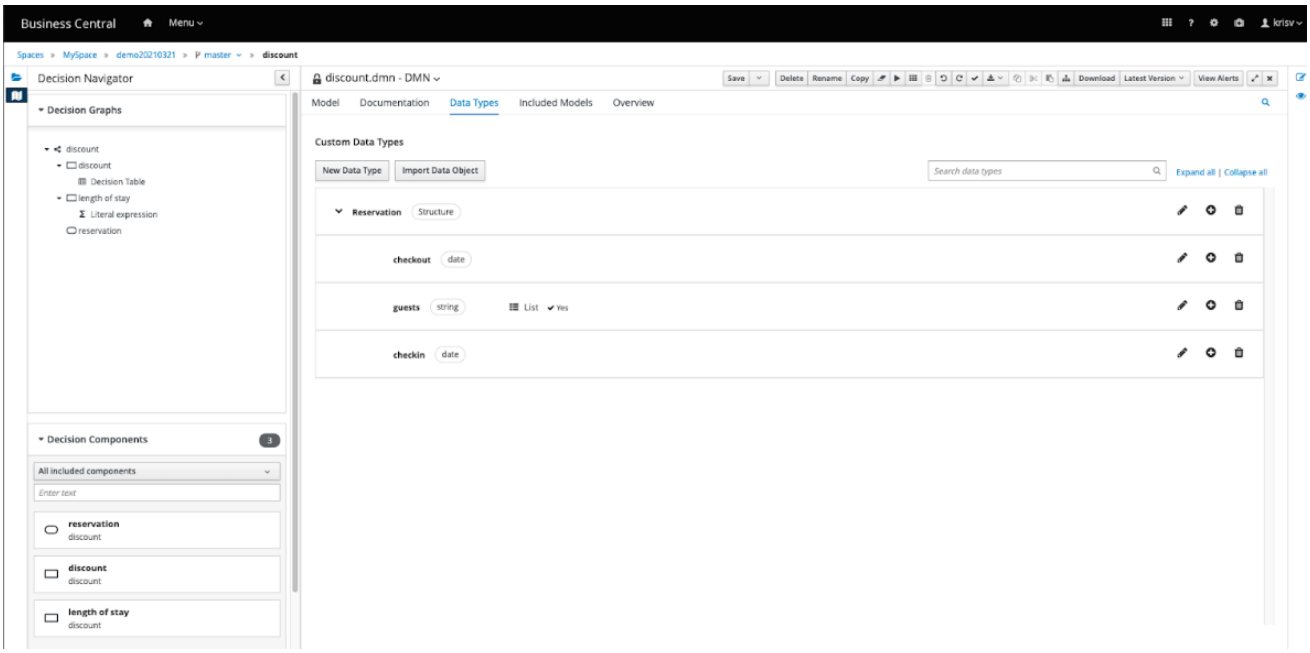


图 17.2. 使用特定 ItemDefinition 结构的 DMN 模型示例



您需要在 Business Central 中开发的现有 KJAR 中将对象模型(OVA)定义为 DTO。

在 KJAR 中定义为 DTO 的对象模型示例

```
package com.myspace.demo20210321;

/**
 * This class was automatically generated by the data modeler tool.
 */

public class Reservation implements java.io.Serializable {

 static final long serialVersionUID = 1L;
 @com.fasterxml.jackson.annotation.JsonFormat(shape =
com.fasterxml.jackson.annotation.JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
    @com.fasterxml.jackson.databind.annotation.JsonSerialize(using =
com.fasterxml.jackson.datatype.jsr310.ser.LocalDateSerializer.class)
 private java.time.LocalDate checkin;
 @com.fasterxml.jackson.annotation.JsonFormat(shape =
com.fasterxml.jackson.annotation.JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
    @com.fasterxml.jackson.databind.annotation.JsonSerialize(using =
com.fasterxml.jackson.datatype.jsr310.ser.LocalDateSerializer.class)
 private java.time.LocalDate checkout;

 private java.util.List<java.lang.String> guests;

 public Reservation() {
 }

 public java.time.LocalDate getCheckin() {
  return this.checkin;
 }

 public void setCheckin(java.time.LocalDate checkin) {
  this.checkin = checkin;
```

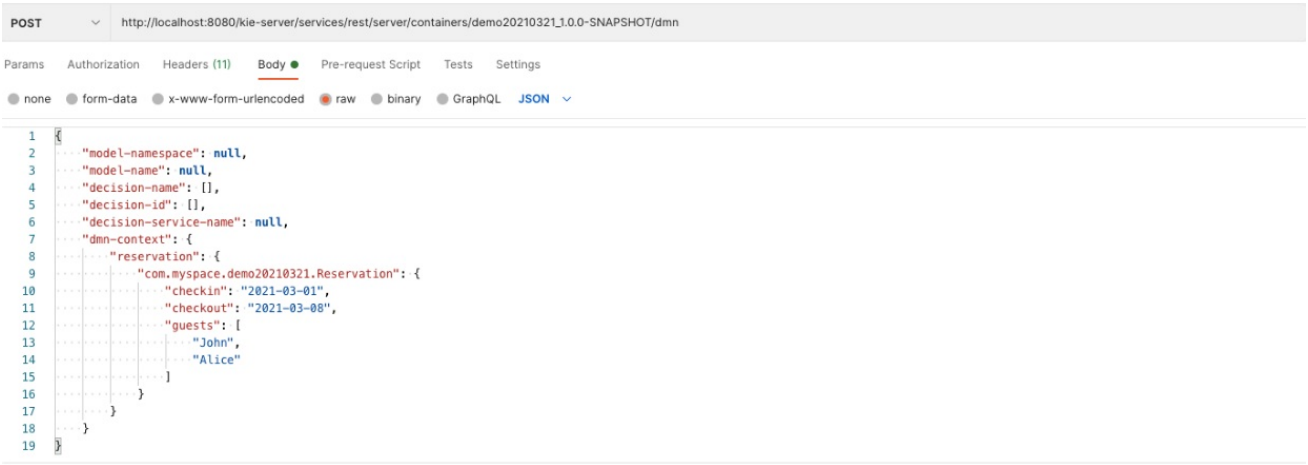
```
 }

 public java.time.LocalDate getCheckout() {
  return this.checkout;
 }

 public void setCheckout(java.time.LocalDate checkout) {
  this.checkout = checkout;
 }

 public java.util.List<java.lang.String> getGuests() {
  return this.guests;
 }

 public void setGuests(java.util.List<java.lang.String> guests) {
  this.guests = guests;
 }

 public Reservation(java.time.LocalDate checkin,
   java.time.LocalDate checkout,
   java.util.List<java.lang.String> guests) {
  this.checkin = checkin;
  this.checkout = checkout;
  this.guests = guests;
 }

}
```

在上例中，定义的 DTO 与 KIE 服务器客户端 Java API 结合使用。另外，您还可以在有效负载中指定 DTO，当一个非 Java 外部应用程序与 KIE 服务器上部署的 KJAR 交互时。

使用 KIE 服务器客户端 Java API 的示例

```
DMNServicesClient dmnClient =
kieServicesClient.getServicesClient(DMNServicesClient.class);
 DMNContext dmnContext = dmnClient.newContext();
 dmnContext.set("reservation", new
com.myspace.demo20210321.Reservation(LocalDate.of(2021, 3, 1),
                                                LocalDate.of(2021, 3, 8),
                                                Arrays.asList("John", "Alice")));
 run(dmnClient, dmnContext);
```

图 17.3. 在有效负载中手动指定 DTO 的示例



注意

在上例中，REST API 中对象模型的FQCN 用于通用 KIE 服务器 marshalling。

17.5. 将外部应用程序绑定到红帽构建 KOGITO 部署的示例

部署 Red Hat build of Kogito 微服务后，您需要将外部应用程序绑定到 Red Hat build of Kogito 微服务部署。

绑定外部应用程序包括重新定义外部应用程序，并将应用程序绑定到与红帽构建 Kogito 应用程序关联的服务器的新基本 URL。如需更多信息，请参阅以下示例：

图 17.4. KIE 服务器上 KJAR 的/discount REST 端点示例

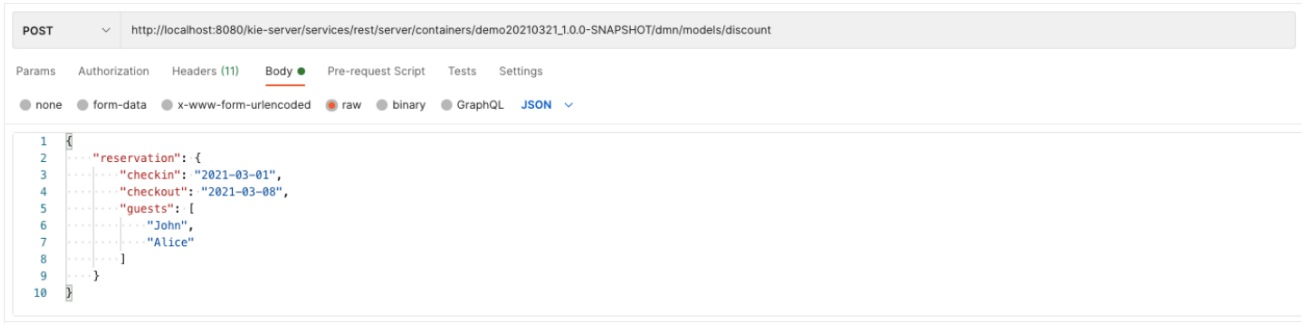


图 17.5. 本地红帽构建的 Kogito 上的/discount REST 端点示例

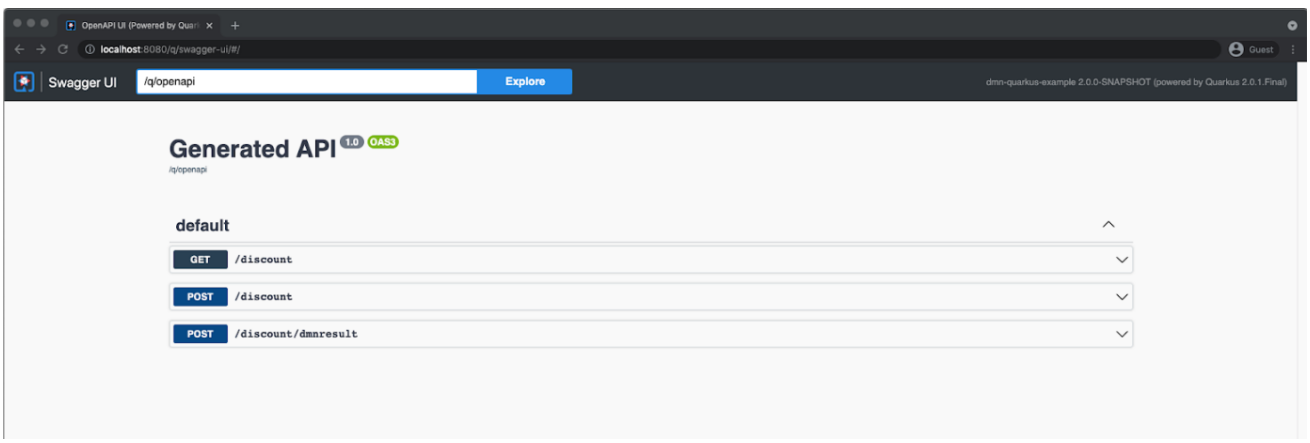
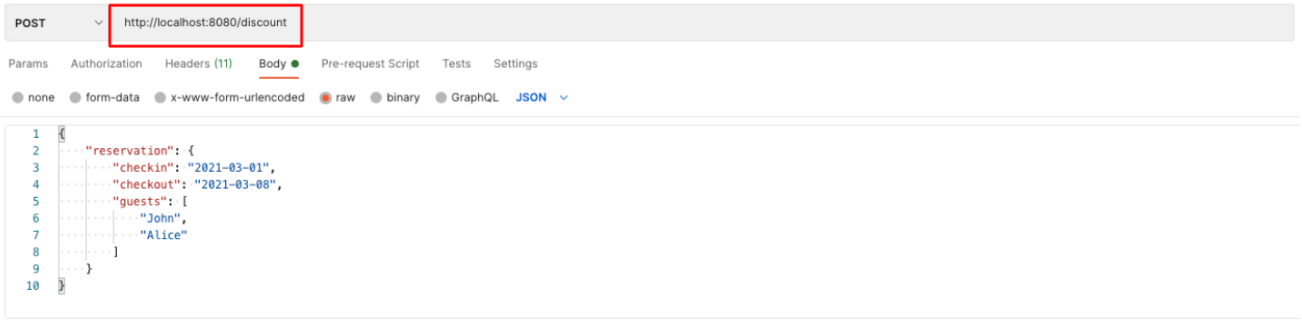


图 17.6. 绑定到新基础 URL 的/discount REST 端点示例



第 18 章 将 PMML 服务迁移到红帽构建的 KOGITO 微服务

您可以通过将 PMML 资源从 KJAR 工件移到相应的 Red Hat build of Kogito 项目，将 PMML 服务迁移到红帽构建的 Kogito 微服务。

18.1. 主要变化和迁移注意事项

下表描述了影响从 KIE 服务器 API 和 KJAR 到 Red Hat build of Kogito 部署的主要变化和功能：

表 18.1. PMML 迁移注意事项

功能	在 KIE 服务器 API 中	在红帽构建的 Kogito 工件中
PMML 型号	存储在 KJAR 的 **src/main/resources**.	将 复制到 **src/main/resources**。
其他配置选项	使用系统属性或 **kmodule.xml** 文件进行配置。	只支持默认值，且不支持覆盖配置。
基于命令的 REST API	使用 **ApplyPmmlModelCommand** 请求 PMML 评估。	不支持。
Domain-driven REST API	不支持。	对 PMML 模型进行特定的生成的高级支持。

注意

上表中未提到的功能在云原生红帽构建的 Kogito 部署中不受支持或不需要。

18.2. 迁移策略

迁移策略包括以下步骤：

1. **将 KIE 服务器上部署的 KJAR 迁移至 Red Hat build of Kogito 微服务。**
2. **使用 Red Hat OpenShift Container Platform 部署红帽 Kogito 微服务构建。**
3. **将 KIE 服务器上的客户端 PMML API 中的外部应用程序修改为 Red Hat 构建的 Kogito 部署的 REST API。**

18.3. 将 PMML 模型 KJAR 迁移到 RED HAT BUILD OF KOGITO 微服务

您可以将使用 PMML 模型实施的 KJAR 迁移至红帽构建 Kogito 微服务。

流程

1. **为您的红帽构建 Kogito 微服务创建一个 Maven 项目。**
   有关创建 Maven 项目的步骤， ***请参阅为红帽构建 Kogito 微服务创建 Maven 项目*****。**

   Maven 项目创建 Kogito 工件。

2. 将 KJAR 的 src/main/resources 文件夹下的 PMML 型号复制到 Kogito 工件的 src/main/resources 文件夹。

3. 运行以下命令执行 Red Hat build of Kogito 应用程序：

```
mvn clean install
```

运行 Red Hat build of Kogito 应用程序后，您可以检索 Swagger 或 OAS 规格文件。Swagger 或 OAS 规格提供与 REST 端点相同的信息，以及以下实施详情：

- 提供 API 的服务器的基本 URL
- 引用架构名称

当外部应用程序重新路由到新 URL 时，您可以使用提供的实现详情。

将 PMML 模型 KJAR 迁移到 Red Hat build of Kogito 微服务后，您需要使用 Red Hat OpenShift Container Platform 部署微服务。有关 Openshift 的部署选项，请参阅使用 *RHPAM Kogito Operator 的 OpenShift 部署选项*。

18.4. 将外部应用程序修改为红帽构建 KOGITO 微服务

部署 PMML 微服务后，您需要将外部应用程序修改为 Kogito 部署的红帽构建。

先决条件

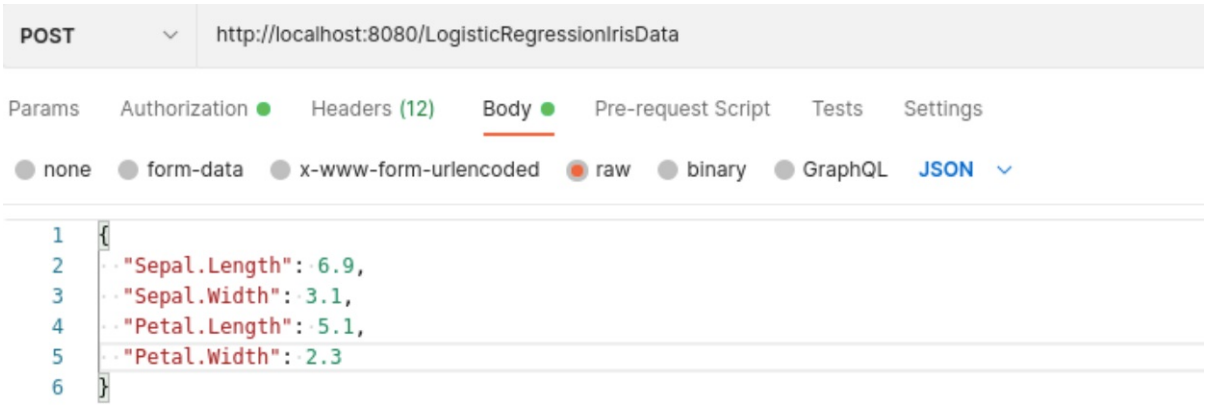
- 原始的外部应用程序必须在 KIE 服务器客户端 API 上实施。

图 18.1. KIE 服务器上的外部应用程序实现示例

```
protected void execute(String correlationId,
                       String containerId,
                       String modelName,
                       String fileName,
                       String targetField,
                       Object expectedResult,
                       Map<String, Object> inputData) {

    final PMMLRequestData request = new PMMLRequestData(correlationId, modelName);
    request.setSource(fileName);
    inputData.forEach(request::addRequestParam);

    final KieCommands commandsFactory = KieServices.Factory.get().getCommands();
    final ApplyPmmlModelCommand command = (ApplyPmmlModelCommand) commandsFactory.newApplyPmmlModel(request);
    final ServiceResponse<ExecutionResults> results = ruleClient.executeCommandsWithResults(containerId, command);
```

流程

1. 删除 KIE 服务器客户端 API 的所有用法，并将其替换为 HTTP 通信。以下是非 Java 请求的示例：



2. **确保外部应用中使用任何 HTTP 客户端 Java 库来创建类似的调用并解析结果。以下是 Java 11 HTTP 客户端和 Gson 将输入数据转换为 JSON 的示例：**

```
protected void execute(Map<String, Object> inputData) throws IOException, InterruptedException {
    Gson gson = new Gson();
    String requestBody = gson.toJson(inputData);
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("http://localhost:8080/LogisticRegressionIrisData"))
            .header( name: "Content-Type",  value: "application/json")
            .POST(HttpRequest.BodyPublishers.ofString(requestBody))
            .build();
    HttpClient client = HttpClient.newHttpClient();
    HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
```

注意

所有作为参数的值所需的参数都嵌入到调用的 URL 中。

第 19 章 将 DRL 服务迁移到红帽构建的 KOGITO 微服务

您可以在红帽构建 Kogito 中构建和部署示例项目，以在红帽 Quarkus REST 端点构建中公开决策引擎的无状态规则评估，并将 REST 端点迁移到 Red Hat build of Kogito。

无状态规则评估是红帽流程自动化管理器中设定的规则的单个执行，可识别为函数调用。在调用函数中，输出值会使用输入值来确定。另外，调用的功能将使用决策引擎来执行作业。因此，在这种情况下，函数通过 REST 端点公开，并转换为微服务。转换为微服务后，函数将部署为功能即服务环境，以消除 JVM 启动时间的成本。

19.1. 主要变化和迁移注意事项

下表描述了影响从 KIE 服务器 API 和 KJAR 到 Red Hat build of Kogito 部署的主要变化和功能：

表 19.1. DRL 迁移注意事项

功能	在 KIE 服务器 API 中	在 Red Hat build of Kogito 中，支持旧 API	在红帽构建的 Kogito 工件中
DRL 文件	存储在 KJAR 的 **src/main/resources** 文件夹中。	将 复制到 **src/main/resources** 文件夹。	使用规则单元和 OOPath 重写。
KieContainer	使用系统属性或 **kmodule.xml** 文件进行配置。	由 **KieRuntimeBuilder** 替代。	不需要。
KieBase 或 **KieSession**	使用系统属性或 **kmodule.xml** 文件进行配置。	使用系统属性或 **kmodule.xml** 文件进行配置。	被规则单元替代。

19.2. 迁移策略

在 Red Hat Process Automation Manager 中，您可以使用以下两种方式将规则评估迁移到 Red Hat build of Kogito 部署：

在 Red Hat build of Kogito 中使用旧的 API

在 Red Hat build of Kogito 中，**kogito-legacy-api** 模块使 Red Hat Process Automation Manager 的旧 API 可用，因此 DRL 文件保持不变。这个迁移规则评估的方法需要最小更改，并可让您使用针对 Quarkus 功能的主要红帽构建，如热重新加载和原生镜像创建。

迁移到红帽构建 Kogito 规则单元

迁移到 Red Hat build of Kogito 的规则单元包括红帽构建的 Kogito 的编程模型，这基于规则单元的概念。

Red Hat build of Kogito 中的规则单元包含一组规则和事实，与规则匹配。Red Hat build of Kogito 中的规则单元也附带数据源。规则单元数据源是由给定规则单元处理的数据源，代表入口点，用于评估规则单元。规则单元使用两种类型的数据源：

- datastream：这是仅附加的数据源，添加到 **DataStream** 中的事实无法更新或删除。
- **Datastore** ：此数据源用于可修改的数据。您可以使用对象添加到 **DataStore** 中时返回的事实来更新或删除对象。

总体而言，规则单元包含两个部分：评估的事实的定义以及评估事实的规则集合。

19.3. LOAN APPLICATION PROJECT 示例

在以下小节中，metering 应用程序项目作为一个示例用于将 DRL 项目迁移到 Red Hat build of Kogito 部署。loan Application 项目的域模型由两个类组成，即 LoanApplication 类和 Applicant 类：

LoanApplication 类示例

```
public class LoanApplication {

  private String id;
  private Applicant applicant;
  private int amount;
  private int deposit;
  private boolean approved = false;

  public LoanApplication(String id, Applicant applicant,
                 int amount, int deposit) {
   this.id = id;
   this.applicant = applicant;
   this.amount = amount;
   this.deposit = deposit;
  }
}
```

Applicant 类示例

```
public class Applicant {

  private String name;
  private int age;

  public Applicant(String name, int age) {
   this.name = name;
   this.age = age;
  }
}
```

规则集是利用业务决策创建来批准或拒绝应用程序，以及收集列表中所有已批准的应用程序的最后一个规则。

在 loan 应用程序中设置的规则示例

```
global Integer maxAmount;
global java.util.List approvedApplications;

rule LargeDepositApprove when
  $l: LoanApplication( applicant.age >= 20, deposit >= 1000, amount <= maxAmount )
then
  modify($l) { setApproved(true) }; // loan is approved
end

rule LargeDepositReject when
  $l: LoanApplication( applicant.age >= 20, deposit >= 1000, amount > maxAmount )
then
  modify($l) { setApproved(false) }; // loan is rejected
end

// ... more loans approval/rejections business rules ...

rule CollectApprovedApplication when
  $l: LoanApplication( approved )
then
  approvedApplications.add($l); // collect all approved loan applications
end
```

19.3.1. 使用红帽构建的 Quarkus 公开使用 REST 端点的规则评估

您可以使用红帽构建的 Quarkus 公开在 Business Central 中通过 REST 端点开发的规则评估。

流程

1. 根据包含规则和 Quarkus 库的模块创建新模块，提供 REST 支持：

创建新模块的依赖项示例

```
<dependencies>

 <dependency>
   <groupId>io.quarkus</groupId>
   <artifactId>quarkus-resteasy</artifactId>
 </dependency>
 <dependency>
   <groupId>io.quarkus</groupId>
   <artifactId>quarkus-resteasy-jackson</artifactId>
 </dependency>

 <dependency>
   <groupId>org.example</groupId>
   <artifactId>drools-project</artifactId>
   <version>1.0-SNAPSHOT</version>
 </dependency>

<dependencies>
```

2. 创建 **REST** 端点。

以下是创建 **REST** 端点的示例设置：

FindApprovedLoansEndpoint 端点设置示例

```
@Path("/find-approved")
public class FindApprovedLoansEndpoint {

  private static final KieContainer kContainer =
KieServices.Factory.get().newKieClasspathContainer();

  @POST()
  @Produces(MediaType.APPLICATION_JSON)
  @Consumes(MediaType.APPLICATION_JSON)
  public List<LoanApplication> executeQuery(LoanAppDto loanAppDto) {
   KieSession session = kContainer.newKieSession();

   List<LoanApplication> approvedApplications = new ArrayList<>();
   session.setGlobal("approvedApplications", approvedApplications);
   session.setGlobal("maxAmount", loanAppDto.getMaxAmount());

   loanAppDto.getLoanApplications().forEach(session::insert);
   session.fireAllRules();
```

```
    session.dispose();

    return approvedApplications;
  }
}
```

在上例中，将创建包含规则的 KieContainer 并添加到 static 字段中。KieContainer 中的规则是从类路径中的其他模块获得的。使用此方法，您可以重复使用与 FindApprovedLoansEndpoint 端点相关的后续调用的 KieContainer，而无需重新编译规则。

注意

两个模块将在下一流程中使用旧 API 将规则单元迁移到红帽构建 Kogito 微服务。如需更多信息，请参阅使用旧 *API 将 DRL 规则单元迁移到红帽构建 Kogito 微服务*。

调用 FindApprovedLoansEndpoint 端点时，会从 KieContainer 创建新的 KieSession。KieSession 填充了来自 LoanAppDto 的对象，从 JSON 请求的 unmarshalling 中生成。

LoanAppDto 类示例

```
public class LoanAppDto {

  private int maxAmount;

  private List<LoanApplication> loanApplications;

  public int getMaxAmount() {
   return maxAmount;
  }

  public void setMaxAmount(int maxAmount) {
   this.maxAmount = maxAmount;
  }

  public List<LoanApplication> getLoanApplications() {
   return loanApplications;
  }

  public void setLoanApplications(List<LoanApplication> loanApplications) {
```

```
    this.loanApplications = loanApplications;
  }
}
```

调用 **fireAllRules（） 方法时，会触发 KieSession d，并针对输入数据评估业务逻辑。在业务逻辑评估后，最后一条规则会在列表中收集所有已批准的应用程序，并且返回相同的列表作为输出。**

3. 启动 **Quarkus 应用程序的红帽构建。**

4. **使用 JSON 请求调用 FindApprovedLoansEndpoint 端点，该端点包含要检查的 loan 应用程序。**

maxAmount 的值在规则中使用，如下例所示：

curl 请求示例

```
curl -X POST -H 'Accept: application/json' -H 'Content-Type: application/json' -d
'{"maxAmount":5000,
"loanApplications":[
{"id":"ABC10001","amount":2000,"deposit":1000,"applicant":{"age":45,"name":"John"}},
{"id":"ABC10002","amount":5000,"deposit":100,"applicant":{"age":25,"name":"Paul"}},
{"id":"ABC10015","amount":1000,"deposit":100,"applicant":{"age":12,"name":"George"}}
]}' http://localhost:8080/find-approved
```

JSON 响应示例

```
[
  {
    "id": "ABC10001",
    "applicant": {
      "name": "John",
      "age": 45
    },
    "amount": 2000,
```

```
    "deposit": 1000,
    "approved": true
  }
]
```

注意

使用此方法时，您无法使用热重新加载功能，且无法创建项目的原生镜像。在接下来的步骤中，缺少的 Quarkus 功能由 Kogito 扩展提供，以便 Quarkus 了解 DRL 文件，并以类似的方式实现热重新加载功能。

19.3.2. 将规则评估迁移到使用旧 API 的红帽 Kogito 微服务构建

使用 REST 端点公开规则评估后，您可以使用传统 API 将规则评估迁移到红帽构建 Kogito 微服务。

流程

1. 在项目 pom.xml 文件中添加以下依赖项，以启用红帽构建的 Quarkus 和旧 API：

使用 Quarkus 和旧 API 的依赖项示例

```
<dependencies>
 <dependency>
  <groupId>org.kie.kogito</groupId>
  <artifactId>kogito-quarkus-rules</artifactId>
 </dependency>
 <dependency>
  <groupId>org.kie.kogito</groupId>
  <artifactId>kogito-legacy-api</artifactId>
 </dependency>
</dependencies>
```

2. 重写 REST 端点实现：

REST 端点实现示例

```
@Path("/find-approved")
public class FindApprovedLoansEndpoint {

  @Inject
  KieRuntimeBuilder kieRuntimeBuilder;

  @POST()
  @Produces(MediaType.APPLICATION_JSON)
  @Consumes(MediaType.APPLICATION_JSON)
  public List<LoanApplication> executeQuery(LoanAppDto loanAppDto) {
   KieSession session = kieRuntimeBuilder.newKieSession();

   List<LoanApplication> approvedApplications = new ArrayList<>();
   session.setGlobal("approvedApplications", approvedApplications);
   session.setGlobal("maxAmount", loanAppDto.getMaxAmount());

   loanAppDto.getLoanApplications().forEach(session::insert);
   session.fireAllRules();
   session.dispose();

   return approvedApplications;
  }
}
```

在重写的 REST 端点实施中，而不是从 KieContainer 创建 KieSession，而是使用集成的 KieRuntimeBuilder 自动创建 KieSession。

KieRuntimeBuilder 是一个由 kogito-legacy-api 模块提供的接口，它取代了 KieContainer。使用 KieRuntimeBuilder，您可以以类似您在 KieContainer 中创建的方式创建 KieBases 和 KieSessions。Red Hat build of Kogito 在编译时自动生成 KieRuntimeBuilder 接口的实施，并将 KieRuntimeBuilder 整合到类中，该函数实施 FindApprovedLoansEndpoint REST 端点。

3. 以开发模式启动您的 Red Hat build of Quarkus 应用程序。

您还可以使用热重新加载对应用到正在运行的应用程序的规则文件进行更改。另外，您可以创建基于规则的应用程序的原生镜像。

19.3.3. 实施规则单元和自动 REST 端点生成

将规则单元迁移到 Red Hat build of Kogito microservice 后，您可以实施规则单元和 REST 端点的自动生成。

在 Red Hat build of Kogito 中，规则单元包含一组规则和事实，与规则匹配。Red Hat build of Kogito 中的规则单元也附带数据源。规则单元数据源是由给定规则单元处理的数据源，代表入口点，用于评估规则单元。规则单元使用两种类型的数据源：

- data stream：这是仅附加数据源。在 DataStream 中，订阅者接收新的和过去的信息，流可以是热或冷的。另外，添加到 DataStream 中的事实无法更新或删除。
- Datastore ：此数据源用于可修改的数据。您可以使用对象添加到 DataStore 中时返回的事实来更新或删除对象。

总体而言，规则单元包含两个部分：评估的事实的定义以及评估事实的一组规则。

流程

1. 使用 POJO 实施事实定义：

使用 POJO 实施事实定义示例

```
package org.kie.kogito.queries;

import org.kie.kogito.rules.DataSource;
import org.kie.kogito.rules.DataStore;
import org.kie.kogito.rules.RuleUnitData;

public class LoanUnit implements RuleUnitData {

  private int maxAmount;
  private DataStore<LoanApplication> loanApplications;

  public LoanUnit() {
   this(DataSource.createStore(), 0);
  }

  public LoanUnit(DataStore<LoanApplication> loanApplications, int maxAmount) {
   this.loanApplications = loanApplications;
   this.maxAmount = maxAmount;
  }

  public DataStore<LoanApplication> getLoanApplications() { return loanApplications;
```

```
}

  public void setLoanApplications(DataStore<LoanApplication> loanApplications) {
   this.loanApplications = loanApplications;
  }

  public int getMaxAmount() { return maxAmount; }
  public void setMaxAmount(int maxAmount) { this.maxAmount = maxAmount; }
}
```

在上例中，而不是使用 LoanAppD 到 LoanUnit 类被直接绑定。LoanAppDto 用于 marshall 或 unmarshall JSON 请求。另外，上例实施了 org.kie.kogito.rules.RuleUnitData 接口，并使用 DataStore 包含要批准的 loan 应用程序。

org.kie.kogito.rules.RuleUnitData 是一个标记接口，用于通知决定引擎 LoanUnit 类是规则单元定义的一部分。另外，DataStore 负责通过触发新规则并触发其他规则来响应更改。

另外，规则的结果会修改上例中的 approved 属性。在 contrary 中，maxAmount 值被视为规则单元的配置参数，不会修改。maxAmount 会在规则评估过程中自动处理，并从 JSON 请求中传递的值自动设置。

2. 实施 DRL 文件：

DRL 文件的实现示例

```
package org.kie.kogito.queries;
unit LoanUnit; // no need to using globals, all variables and facts are stored in the rule unit

rule LargeDepositApprove when
  $l: /loanApplications[ applicant.age >= 20, deposit >= 1000, amount <= maxAmount ] //
oopath style
then
  modify($l) { setApproved(true) };
end

rule LargeDepositReject when
  $l: /loanApplications[ applicant.age >= 20, deposit >= 1000, amount > maxAmount ]
then
  modify($l) { setApproved(false) };
end
```

```
// ... more loans approval/rejections business rules ...

// approved loan applications are now retrieved through a query
query FindApproved
  $l: /loanApplications[ approved ]
end
```

您创建的 DRL 文件必须声明与事实定义实施相同的软件包，以及一个具有相同 Java 类名称的单元。Java 类将 RuleUnitData 接口实施为接口属于同一规则单元的状态。

另外，上例中的 DRL 文件会使用 OOPath 表达式重写。在 DRL 文件中，数据源充当入口点，OOPath 表达式包含数据源名称作为 root。但是，限制在方括号中添加，如下所示：

```
$L: /loanApplications[ applicant.age >= 20, stored >= 1000, amount maxAmount ]
```

或者，您可以使用标准 DRL 语法，在其中指定数据源名称作为入口点。但是，您需要再次指定匹配对象的类型，如下例所示，即使决策引擎能够推断数据源的类型：

```
$L: LoanApplication(applicant.age >= 20, stored >= 1000, amount gap
maxAmount)from entry-point loanApplications
```

在前面的示例中，收集所有批准的 loan 应用的最后一个规则将被替换为检索列表的查询。规则单元定义要在输入中用于评估规则的事实，而查询则定义了规则评估中预期的输出。使用这个方法，红帽构建的 Kogito 可以自动生成一个执行查询的类，并如下例所示：

LoanUnitQueryFindApproved 类示例

```
public class LoanUnitQueryFindApproved implements
org.kie.kogito.rules.RuleUnitQuery<List<org.kie.kogito.queries.LoanApplication>> {

  private final RuleUnitInstance<org.kie.kogito.queries.LoanUnit> instance;

  public
LoanUnitQueryFindApproved(RuleUnitInstance<org.kie.kogito.queries.LoanUnit>
instance) {
   this.instance = instance;
  }
```

```
  @Override
  public List<org.kie.kogito.queries.LoanApplication> execute() {
   return
instance.executeQuery("FindApproved").stream().map(this::toResult).collect(toList());
  }

  private org.kie.kogito.queries.LoanApplication toResult(Map<String, Object> tuple) {
   return (org.kie.kogito.queries.LoanApplication) tuple.get("$l");
  }
}
```

以下是一个 REST 端点示例，它使用规则单元作为输入并将输入传递给查询执行器返回输出：

LoanUnitQueryFindApprovedEndpoint 端点示例

```
@Path("/find-approved")
public class LoanUnitQueryFindApprovedEndpoint {

  @javax.inject.Inject
  RuleUnit<org.kie.kogito.queries.LoanUnit> ruleUnit;

  public LoanUnitQueryFindApprovedEndpoint() {
  }

  public
LoanUnitQueryFindApprovedEndpoint(RuleUnit<org.kie.kogito.queries.LoanUnit>
ruleUnit) {
   this.ruleUnit = ruleUnit;
  }

  @POST()
  @Produces(MediaType.APPLICATION_JSON)
  @Consumes(MediaType.APPLICATION_JSON)
  public List<org.kie.kogito.queries.LoanApplication>
executeQuery(org.kie.kogito.queries.LoanUnit unit) {
   RuleUnitInstance<org.kie.kogito.queries.LoanUnit> instance =
ruleUnit.createInstance(unit);
   return instance.executeQuery(LoanUnitQueryFindApproved.class);
  }
}
```

注意

您还可以添加多个查询和每个查询，并生成不同的 REST 端点。例如，为 find-approved 生成 FindApproved REST 端点。

第 20 章 其他资源

- *使用 DMN 模型设计决策服务*
- *使用 DRL 规则设计决策服务*
- *使用 PMML 型号设计决策服务*

附录 A. 版本信息

文档最新更新于 2024 年 3 月 14 日星期四。

附录 B. 联系信息

Red Hat Process Automation Manager 文档团队 ：brms-docs@redhat.com