Red Hat Process Automation Manager 7.5

デシジョンエンジンを使用した高可用性イベント駆動型デシジョン機能の Red Hat OpenShift Container Platform への実装

ガイド

Last Updated: 2022-09-17

Red Hat Process Automation Manager 7.5 デシジョンエンジンを使用した高可用性イベント駆動型デシジョン機能の Red Hat OpenShift Container Platform への実装

ガイド

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

法律上の通知

概要

本書は、Red Hat Process Automation Manager 7.5 で、Red Hat OpenShift Container Platform に、デシジョンエンジンを使用して高可用性イベント駆動型デシジョン機能を実装する方法を説明します。

目次

前書き 3

第1章 RED HAT OPENSHIFT CONTAINER PLATFORM での高可用性イベント駆動型デシジョン機能 4

第2章 HA CEP サーバーの実装 5

第3章 HA CEP クライアントの作成 7

第4章 HA CEP クライアントコードの要件 8
kie-remote API 8
明示的なタイムスタンプ 8
メモリー以外のアクションの Lambda 式 8

付録A バージョン情報 10

前書き

ビジネスルール開発者は、デシジョンエンジンを使用するコードで、複合イベント処理 (CAP: Complex Event Processing) など、高可用性イベント駆動型デシジョン機能を使用できます。高可用性イベント駆動型デシジョン機能は、Red Hat OpenShift Container Platform に実装できます。

『Operator を使用した Red Hat OpenShift Container Platform への Red Hat Process Automation Manager 環境のデプロイメント』の記載のとおり、Red Hat OpenShift Container Platform では、Red Hat Process Automation Manager の標準デプロイメントを使用して、高可用性イベント駆動型デシジョン機能を実装できません。標準デプロイメントは、ステートレス処理しかサポートしないためです。そのため、指定の参照実装を使用して、カスタム実装を作成する必要があります。

前提条件

- Red Hat OpenShift Container Platform 4.1 環境が利用でき、プロジェクトが作成されている。
- Red Hat AMQ Streams を含む OpenShift 環境に、Kafka Cluster がデプロイされている。
- OpenJDK Java 開発環境がインストールされている。
- Maven、Docker、および kubectl がインストールされている。
- OpenShift コマンドラインツール **oc** がインストールされている。

第1章 RED HAT OPENSHIFT CONTAINER PLATFORM での高可用性イベント駆動型デシジョン機能

デシジョンエンジンを使用して、Red Hat OpenShift Container Platform に高可用性イベント駆動型デシジョン機能を実装できます。

イベント は、特定の時点で発生するファクトをモデル化します。デシジョンエンジンは、一時オペレーターが豊富にあり、イベントの比較、相関、累積ができます。イベント駆動型のデシジョン機能では、デシジョンエンジンがイベントをもとに一連の複雑なデシジョンを処理します。イベントはすべて、エンジンの状態を変更でき、後続のイベントのデシジョンに影響を与えます。

『Operator を使用した Red Hat OpenShift Container Platform への Red Hat Process Automation Manager 環境のデプロイメント』の記載のとおり、Red Hat OpenShift Container Platform では、Red Hat Process Automation Manager の標準デプロイメントを使用 して、高可用性イベント駆動型デシジョン機能を実行できません。デプロイメントには Process Server(KIE Server)Pod が含まれており、スケーリング時も Pod ごとに独立したままになります。Pod の状態は同期されません。そのため、ステートレス呼び出しのみを確実に処理できます。

複合イベント処理 (CEP) API は、デシジョンエンジンを含むイベント駆動型デシジョン機能で便利です。デシジョンエンジンは、CEP を使用してイベントコレクションにある複数のイベントを検出して処理し、イベント間に存在する関係を明確にして、このようなイベントや関係をもとに新規データを推測します。デシジョンエンジンでの CEP に関する情報は、「**Red Hat Process Automation Manager のデシジョンエンジン**」を参照してください。

Red Hat Process Automation Manager が提供する参照実装をもとに、Red Hat OpenShift Container Platform に高可用性イベント駆動型デシジョン機能を実装できます。この実装を使用すると、安全にフェイルオーバーできる環境が実現できます。

この参照実装では、処理コードを使用して Pod をスケーリングできます。Pod のレプリカは独立していません。レプリカの 1 つが自動的に **リーダー** として指定されます。リーダーが機能を停止した場合には、別のリーダーが自動的にリーダーになり、中断やデータの損失なしに、処理が続行されます。

リーダーの選択は、Kubernetes ConfigMaps で実装されます。リーダーと他のレプリカは、Kafka を介してメッセージを交換することで連携します。リーダーが必ず、最初にイベントを処理します。処理が完了したら、リーダーは他のレプリカに通知します。リーダーではないレプリカは、リーダーでの処理が完了してからでないとイベントは実行されません。

新規レプリカがクラスターに参加すると、このレプリカは、リーダーから、現在の Drools セッションのスナップショットを要求します。Kafka トピックで利用可能なスナップショットがある場合に、リーダーは既存で最新のスナップショットを使用できます。最新のスナップショットがない場合はリーダーがオンデマンドで新しいスナップショットを生成します。スナップショットを受信後に、新しいレプリカはそのスナップショットをデシリアライズし、最終的に、スナップショットに含まれていない最後のイベントを実行し、その後にリーダーと連携して新規イベントの処理は開始されません。

第2章 HA CEP サーバーの実装

高可用性 (HA) CEP サーバーは、Red Hat OpenShift Container Platform 環境で実行します。このサーバーには、必要なすべての Drools ルールと、イベント処理に必要なその他のコードが含まれています。

ソースを準備して、ビルドし、Red Hat OpenShift Container Platform にデプロイする必要がある。

手順

1. Red Hat カスタマーポータルの Software Downloads ページから製品配信可能ファイル **rhpam-7.5.1-reference-implementation.zip** をダウンロードします。

2. ファイルの内容を展開して、さらに **rhpam-7.5.1-openshift-drools-hacep-distribution.zip** ファイルを展開します。

3. **openshift-drools-hacep-distribution/sources** ディレクトリーに移動します。

4. **sample-hacep-project/sample-hacep-project-kjar** ディレクトリー内のサンプルプロジェクトをもとに、サーバーのコードを確認して変更します。複合イベント処理のロジックは、**src/main/resources/org.drools.cep** サブディレクトリーの DRL ルールで定義します。

5. 標準の Maven コマンドを使用してプロジェクトをビルドします。

```
mvn clean install -DskipTests
```

6. OpenShift operator インフラストラクチャーを使用して、Red Hat AMQ Streams をインストールします。Red Hat AMQ Streams のインストールに関する情報は、『Using AMQ Streams on OpenShift Container Platform』を参照してください。

7. Red Hat OpenShift Container Platform の **KafkaTopic** リソースを使用して、**kafka-topics** サブディレクトリーの全 YAML ファイルからトピックを作成します。
KafkaTopic リソースを使用したトピックの作成に関する詳細は、Red Hat AMQ ドキュメントの「Using the topic operator」を参照してください。

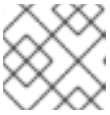
8. アプリケーションが、リーダーの選択に使用する ConfigMap にアクセスできるように、ロールベースのアクセス制御を設定する必要があります。**springboot** ディレクトリーに移動して、以下のコマンドを入力します。

```
oc create -f kubernetes/service-account.yaml
oc create -f kubernetes/role.yaml
oc create -f kubernetes/role-binding.yaml
```

Red Hat OpenShift Container Platform のロールベースのアクセス制御に関する詳細は、Red Hat OpenShift Container Platform ドキュメントの「RBAC を使用したパーミッションの定義および適用」を参照してください。

9. **springboot** ディレクトリーで、**Dockerfile** ファイルを編集します。**microdnf** コマンドを含む行を以下の行に置き換えます。

```
RUN microdnf install java-1.8.0-openjdk-headless && microdnf clean all
```



注記

microdnf で最近問題が発見 されたため、回避策が必要です。

10. **springboot** ディレクトリーで、以下のコマンドを入力して Docker イメージをビルドして、お使いのシステムで設定されている Docker レジストリーにプッシュします (このコマンドを実行する前に非公開のレジストリーを設定することをご検討ください)。このビルドは、Maven の依存関係として構築された t sample-hacep-project-kjar コードをインポートし、openshift-kie-springboot.jar ファイルの BOOT-INF/lib ディレクトリーに追加します。このビルドは、ビルドした **sample-hacep-project-kjar** コードを Maven 依存関係としてインポートし、**openshift-kie-springboot.jar** ファイルの **BOOT-INF/lib** ディレクトリーに追加します。その後、Docker ビルドは JAR ファイルを使用してイメージを作成します。

```
docker login --username=<user username>
docker build -t <user_username>/openshift-kie-springboot:<tag> .
docker push <user_username>/openshift-kie-springboot:<tag>
```

または、ビルドの目的で Podman を使用するには、コマンドで **docker** を **podman** に置き換えます。

11. OpenShift ユーザーインターフェイスで YAML ソースを作成します。**kubernetes/deployment.yaml** の内容を貼り付けて、必要に応じて Docker のイメージ名を変更します。**Deploy** をクリックしてサーバーをデプロイします。

第3章 HA CEP クライアントの作成

CEP クライアントコードを HA CEP サーバーイメージと通信できるように、適応する必要があります。お使いのクライアントコード向けの参照実装に含まれるサンプルプロジェクトを使用できます。また、OpenShift 環境内外を問わず、クライアントコードを実行できます。

手順

1. Red Hat カスタマーポータルの Software Downloads ページから製品配信可能ファイル **rhpam-7.5.1-reference-implementation.zip** をダウンロードします。
2. ファイルの内容を展開して、さらに **rhpam-7.5.1-openshift-drools-hacep-distribution.zip** ファイルを展開します。
3. **openshift-drools-hacep-distribution/sources** ディレクトリーに移動します。
4. **sample-hacep-project/sample-hacep-project-client** ディレクトリーのサンプルプロジェクトをもとにクライアントコードをレビューし、変更します。このコードが *4章HA CEP クライアントコードの要件* に記載の追加要件を満たしていることを確認します。
5. **sample-hacep-project/sample-hacep-project-client** ディレクトリーで、パスワードに **password** と指定してキーストアを生成します。以下のコマンドを入力します。

```
keytool -genkeypair -keyalg RSA -keystore src/main/resources/keystore.jks
```

6. OpenShift 環境から HTTPS 証明書を展開して、キーストアーに追加します。以下のコマンドを実行します。

```
oc extract secret/my-cluster-cluster-ca-cert --keys=ca.crt --to=- > src/main/resources/ca.crt
keytool -import -trustcacerts -alias root -file src/main/resources/ca.crt -keystore
src/main/resources/keystore.jks -storepass password -noprompt
```

7. プロジェクトの **src/main/resources** サブディレクトリーに、以下のコンテンツを含めた **configuration.properties** ファイルを作成します。

```
ssl.keystore.location=keystore.jks
ssl.truststore.location=keystore.jks
ssl.keystore.password=password
ssl.truststore.password=password
bootstrap.servers=http://<serveraddress>
```

<serveraddress> は、Kafka サーバーのルートが使用するアドレスに置き換えます。

8. 標準の Maven コマンドを使用してプロジェクトをビルドします。

```
mvn clean install
```

9. **sample-hacep-project-client** プロジェクトのディレクトリーに移動して、以下のコマンドを入力し、クライアントを実行します。

```
mvn exec:java -Dexec.mainClass="org.kie.hacep.sample.client.ClientProducerDemo
```

このコマンドは、**ClientProducerDemo** クラスの **main** メソッドを実行します。

第4章 HA CEP クライアントコードの要件

高可用性 CEP のクライアントコードを開発する場合には、以下のような特定の追加要件に準拠する必要があります。

kie-remote API

コードは、**kie** ではなく **kie-remote** API を使用する必要があります。**kie-remote** API は、**org.kie:kie-remote** Maven アーティファクトに指定します。また、ソースコードは、Maven モジュール **kie-remote** にあります。

明示的なタイムスタンプ

デシジョンエンジンは、イベントの発生する順番を決定する必要があります。このような理由から、イベントには必ず、タイムスタンプを割り当てます。高可用性環境では、イベントをモデル化する JavaBean のプロパティーに、このタイムスタンプを指定します。次に、イベントクラスに **@Timestamp** アノテーションをつける必要があります。以下の例のように、ここではタイムスタンプ属性自体の名前がパラメーターとなります。

```
@Role(Role.Type.EVENT)
@Timestamp("myTime")
public class StockTickEvent implements Serializable {

    private String company;
    private double price;
    private long myTime;
}
```

タイムスタンプ属性を指定しない場合には、クライアントがリモートセッションにイベントを挿入するタイミングをもとに、Drools が全イベントにタイムスタンプを割り当てます。ただし、このメカニズムは、クライアントマシンのクロックにより異なります。異なるクライアント間でクロックにずれがある場合は、このようなホストが挿入したイベント間で不整合が発生する可能性があります。

メモリー以外のアクションの Lambda 式

作業メモリーアクション (デシジョンエンジンの作業メモリー内の情報を挿入、変更、または削除するアクション) は、クラスターの全ノードで処理する必要があります。メモリーアクションではないアクションは、リーダーでのみ実行する必要があります。

たとえば、今回のコードには、以下のルールが含まれます。

```
rule FindAdult when
  $p : Person(age >= 18)
then
  modify($p) { setAdult(true) }; // working memory action
  sendEmailTo($p); // side effect
end
```

このルールがトリガーされると、対象となる人は、すべてのノードで大人としてマークする必要があります。ただし、送信されるメール数が 1 通だけとなるように、リーダーだけがメールを送信できます。

そのため、以下の例のように、lambda 式のメールアクション (**副作用** と呼ばれる) をラップする必要があります。

```
rule FindAdult when
  $p : Person(age >= 18)
then
```

```
  modify($p) { setAdult(true) };
  DroolsExecutor.getInstance().execute( () -> sendEmailTo($p) );
end
```

付録A バージョン情報

本書の最終更新日： 2021 年 6 月 25 日（金）