Quarkus - Using Kogito to add business automation capabilities to an application

This guide demonstrates how your Quarkus application can use Kogito to add business automation to power it up with business processes and rules.

Kogito is a next generation business automation toolkit that originates from well known Open Source projects Drools (for business rules) and jBPM (for business processes). Kogito aims at providing another approach to business automation where the main message is to expose your business knowledge (processes, rules and decisions) in a domain specific way.

This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.

For a full list of possible extension statuses, check our FAQ entry.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE (Eclipse is preferred with the BPMN modeller plugin)
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- Docker

Install modelling plugins in your IDE

Kogito Tooling is currently supported in Eclipse and VSCode:

- Eclipse

To be able to make use of visual modelling of your processes, download Eclipse IDE and install from Market place Eclipse BPMN2 Modeller plugin (with jBPM Runtime Extension)

- VSCode

Download and install the VSCode Extension from Kogito Tooling release page to edit and model

process definitions from VSCode IDE.

- Online

To avoid any modeler installation you can use directly use BPMN.new to design and model your process through your favorite web browser.

Architecture

In this example, we build a very simple microservice which offers one REST endpoint:

- `/persons`

This endpoint will be automatically generated based on business process, that in turn will make use of business rules to make certain decisions based on the data being processed.

Business process

The business process will be responsible for encapsulating business logic of our microservice. It should provide complete set of steps to achieve a business goal. At the same time this is the entry point to the service that can be consumed by clients.

Business rule

A business rule allows to externalise decision logic into reusable pieces that can be easily used in declarative way. There are multiple ways of writing rules like decision tables, decision trees, rules, etc. For this example we focus on the rule format backed by DRL (Drools Rule Language).

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the complete example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `kogito-quickstart` directory.

Creating the Maven Project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.2.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=kogito-quickstart \
    -Dextensions="kogito"
cd kogito-quickstart
```

This command generates a Maven project, importing the `kogito` extension that comes with all needed dependencies and configuration to equip your application with business automation.

If you already have your Quarkus project configured, you can add the `kogito` extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="kogito"
```

This will add the following to your `pom.xml`:

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

Writing the application

Let's start by implementing the simple data object `Person`. As you can see from the source code below it is just a POJO:

```java
package org.acme.kogito.model;

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 + "]";
    }

}
```

Next, we create a rule file `person-rules.drl` inside the `src/main/resources/org/acme/kogito` folder of the generated project.

```
package org.acme.kogito

import org.acme.kogito.model.Person;


rule "Is adult" ruleflow-group "person"

when
    $person: Person(age > 18)
then
    modify($person) {
        setAdult(true)
    };
end
```
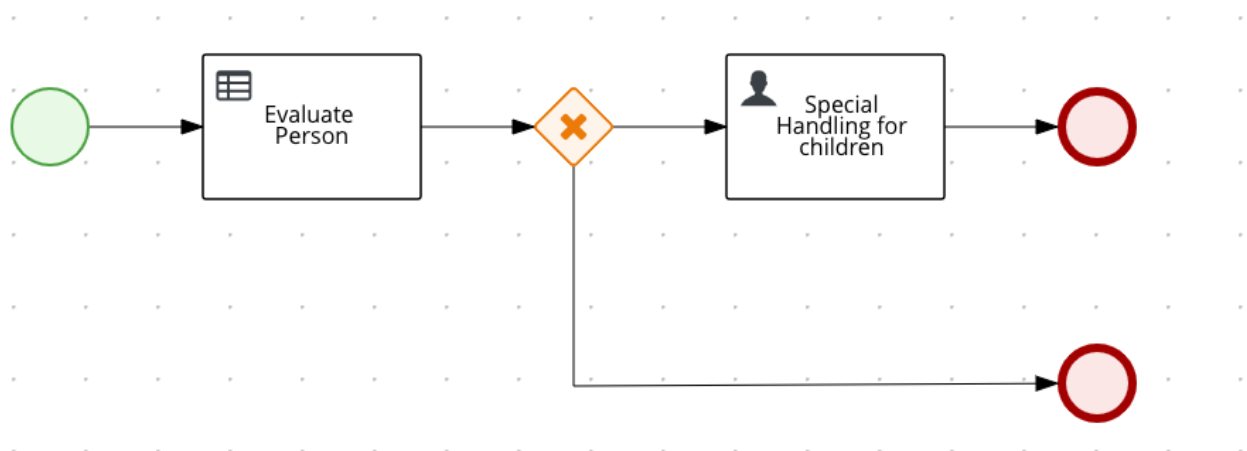
This is really a simple rule that marks a person who is older that 18 years as an adult.

Finally we create a business process that will make use of this rule and some other activities to approve a given person. Using new item wizard (File → New → Other → BPMN2 Model) create `persons.bpmn` inside `src/main/resources/org/acme/kogito` folder of the generated project.

This process should consist of

- start event

- business rule task

- exclusive gateway

- user task

- end events

And should look like

To get started quickly copy the process definition from the quickstart

To model this process yourself, just follow these steps (start event should be automatically added)

- define a process variable with name `person` of type `org.acme.kogito.model.Person`

- drag the Tasks → Business Rule Task from the palette and drop it next to start event, link it with start event

  - double click on the business rule task

    - on tab I/O Parameters, set data input and output (map `person` process variable to input data with name `person` and same for data output)

    - on tab Business Rule Task, set rule flow group to the value defined in the drl file (`person`)

- drag the Gateways → XOR gateway from the palette and drop it next to the business rule task, link it with rule task

- drag the Tasks → User Task from the palette and drop it next to the gateway, link it with gateway

  - double click on the user task

    - on tak User Task, set task name to `ChildrenHandling`

    - on tab I/O Parameters, set data input (map `person` process variable to input data with name `person`)

- drag the End Events → End from the palette and drop it next to the user task, link it with the user task

- drag the End Events → End from the palette and drop it next to the gateway, link it with the user task

- double click on the gateway

  - on tab Gateway, set the diverging direction for the gateway

  - on tab Gateway, set conditions on sequence flow list

    - → going to end event `return person.isAdult() == true;` with language `Java`

    - → going to user task `return person.isAdult() == false;` with language `Java`

- save the file

Running and Using the Application

Running in Developer Mode

To run the microservice in dev mode, use `./mvnw clean compile quarkus:dev`.

Running in JVM Mode

When you're done playing with "dev-mode" you can run it as a standard Java application.

First compile it:

```
./mvnw package
```

Then run it:

```
java -jar ./target/kogito-quickstart-runner.jar
```

Running in Native Mode

This same demo can be compiled into native code: no modifications required.

This implies that you no longer need to install a JVM on your production environment, as the runtime technology is included in the produced binary, and optimized to run with minimal resource overhead.

Compilation will take a bit longer, so this step is disabled by default; let's build again by enabling the native profile:

```
./mvnw package -Dnative
```

After getting a cup of coffee, you'll be able to run this binary directly:

```
./target/kogito-quickstart-runner
```

Testing the Application

To test your application, just send request to the service with giving the person as JSON payload.

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

In the response, the person should be approved as an adult and that should also be visible in the response payload.

```
{"id":"dace1d6a-a5fa-429d-b253-
d6b66e265bbc","person":{"adult":true,"age":20,"name":"John Quark"}}
```

You can also verify that there are no more active instances

```
curl -X GET http://localhost:8080/persons \
    -H 'content-type: application/json' \
    -H 'accept: application/json'
```

To verify the non adult case, send another request with the age set to less than 18

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

this time there should be one active instance, replace {uuid} with the id attribute taken from the response

```
curl -X GET http://localhost:8080/persons/{uuid}/tasks \
    -H 'content-type: application/json' \
    -H 'accept: application/json'
```

You can get the details of the task by calling another endpoint, replace uuids with the values taken from the responses (uuid-1 is the process instance id and uuid-2 is the task instance id). First corresponds to the process instance id and the other to the task instance id.

```
curl -X GET http://localhost:8080/persons/{uuid-
1}/ChildrenHandling/{uuid-2} \
    -H 'content-type: application/json' \
    -H 'accept: application/json'
```

You can complete this person evaluation process instance by calling the same endpoint but with POST, replace uuids with the values taken from the responses (uuid-1 is the process instance id and uuid-2 is the task instance id).

```
curl -X POST http://localhost:8080/persons/{uuid-
1}/ChildrenHandling/{uuid-2} \
    -H 'content-type: application/json' \
    -H 'accept: application/json' \
    -d '{}'
```

Enabling persistence

Since 0.3.0 of Kogito, there is an option to enable persistence to preserve process instance state across application restarts. That supports long running process instances that can be resumed at any point in time.

Prerequisites

Kogito uses Infinispan as the persistence service so you need to have Infinispan server installed and running. Version of the Infinispan is aligned with Quarkus BOM so make sure the right version is installed.

Add dependencies to project

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-infinispan-client</artifactId>
</dependency>
<dependency>
  <groupId>org.kie.kogito</groupId>
  <artifactId>infinispan-persistence-addon</artifactId>
  <version>${kogito.version}</version>
</dependency>
```

Configure connection with Infinispan server

Add following into the src/main/resources/application.properties file (create the file if it does not exist)

```
quarkus.infinispan-client.server-list=localhost:11222
```

> ℹ️ Adjust the host and port number according to your Infinispan server installation.

Test with enabled persistence

After configuring persistence on the project level, you can test and verify that the process instance state is preserved across application restarts.

- start Infinispan server

- build and run your project

- execute non adult use case

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

You can also verify that there is active instance

```
curl -X GET http://localhost:8080/persons \
    -H 'content-type: application/json' \
    -H 'accept: application/json'
```

Restart your application while keeping Infinispan server up and running.

Check if you can see active instance which should have exactly the same id

```
curl -X GET http://localhost:8080/persons \
    -H 'content-type: application/json' \
    -H 'accept: application/json'
```

To learn more about persistence in Kogito visit this page

Using decision tables

Kogito allows to define business rules as decision tables using the Microsoft Excel file formats. To be able to use such assets in your application, an additional dependency is required:

```
<dependency>
    <groupId>org.kie.kogito</groupId>
    <artifactId>drools-decisiontables</artifactId>
</dependency>
```

Once the dependency is added to the project, decision tables in `xls` or `xlsx` format can be properly handled.

References

- Kogito Website

- Kogito Documentation