Developing Decision Microservices with Spring Boot and OpenRules

OpenRules, Inc.

www.openrules.com

May-2019

TABLE OF CONTENTS

Introduction

Nowadays microservices quickly become a highly popular architectural approach. They have shown a great deal of benefits over the legacy style of monolithic single applications. Spring Boot is an open source Java-based framework that is commonly used to create microservices. It offers the following advantages:

- Easy deployment
- Simple scalability
- Compatible with Containers and cloud environments
- Minimum configuration
- Lesser production time.

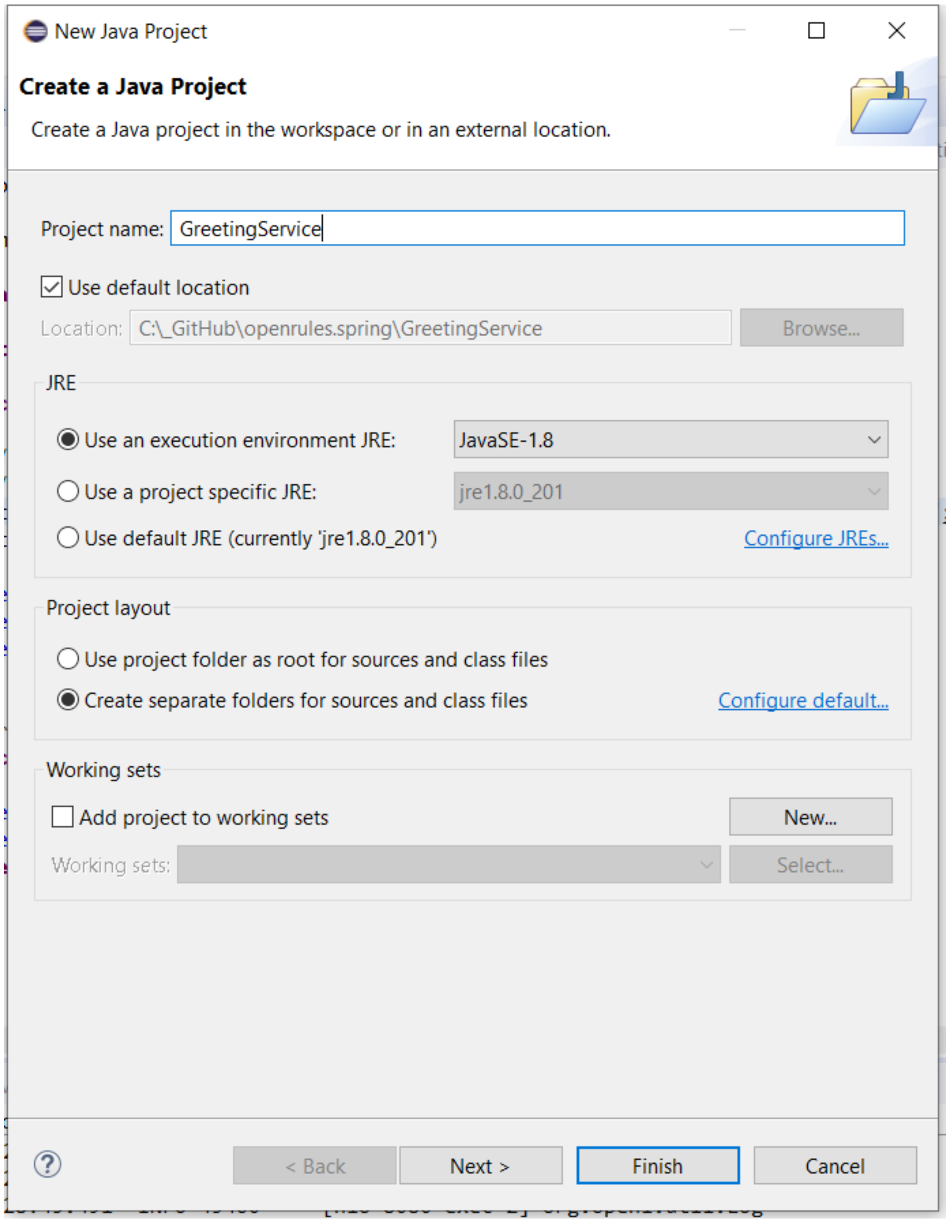
This tutorial provides a sampling of how to build Decision Microservices with Spring Boot and OpenRules.

What You'll Build

You will build a simple greeting service that greets a customer. Initially, you will create it in Java and then add it to a simple web application built using Spring Boot. You will be able to test it as a service using the standard JSON tests send over http requests. Then you will make your greeting more sophisticated by moving the greeting logic to OpenRules-based Excel files. Then you will make necessary changes in the Spring Boot web application to call the new OpenRules-based greeting service. We will explain all installations and development details not assuming any preliminary knowledge of the Spring framework – the only assumption is that you are familiar with Java and Eclipse IDE. You will end up with a working decision microservice and will be ready to create and deploy more OpenRules-based microservices.

Creating Simple Java-based Greeting Service in Eclipse

We will assume that you've already installed Java 1.8 or later and Eclipse IDE. Start Eclipse with a new workspace, and select File+New+Java Project:


New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: GreetingService

[✓] Use default location

Location: C:_GitHub\openrules.spring\GreetingService

JRE

(●) Use an execution environment JRE: JavaSE-1.8

( ) Use a project specific JRE: jre1.8.0_201

( ) Use default JRE (currently 'jre1.8.0_201')

Project layout

( ) Use project folder as root for sources and class files

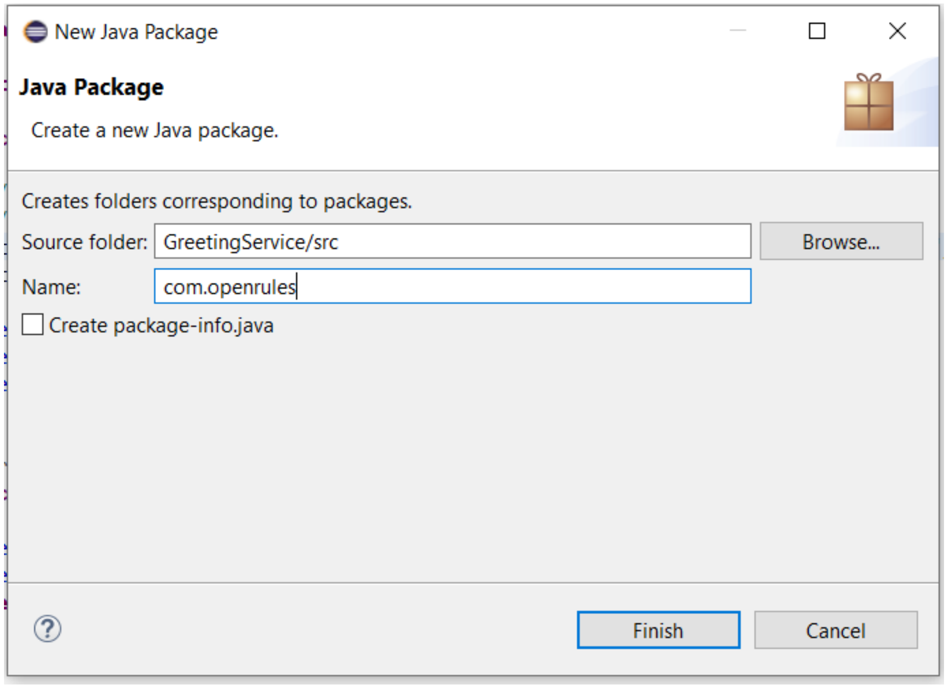
(●) Create separate folders for sources and class files

Working sets

[ ] Add project to working sets

Working sets:

When you click on "Finish" a new project GreetingService will be created. Right-click on the subfolder "GreetingService/src", select New Java Package and enter "com.openrules" as the Name:


New Java Package

Java Package

Create a new Java package.

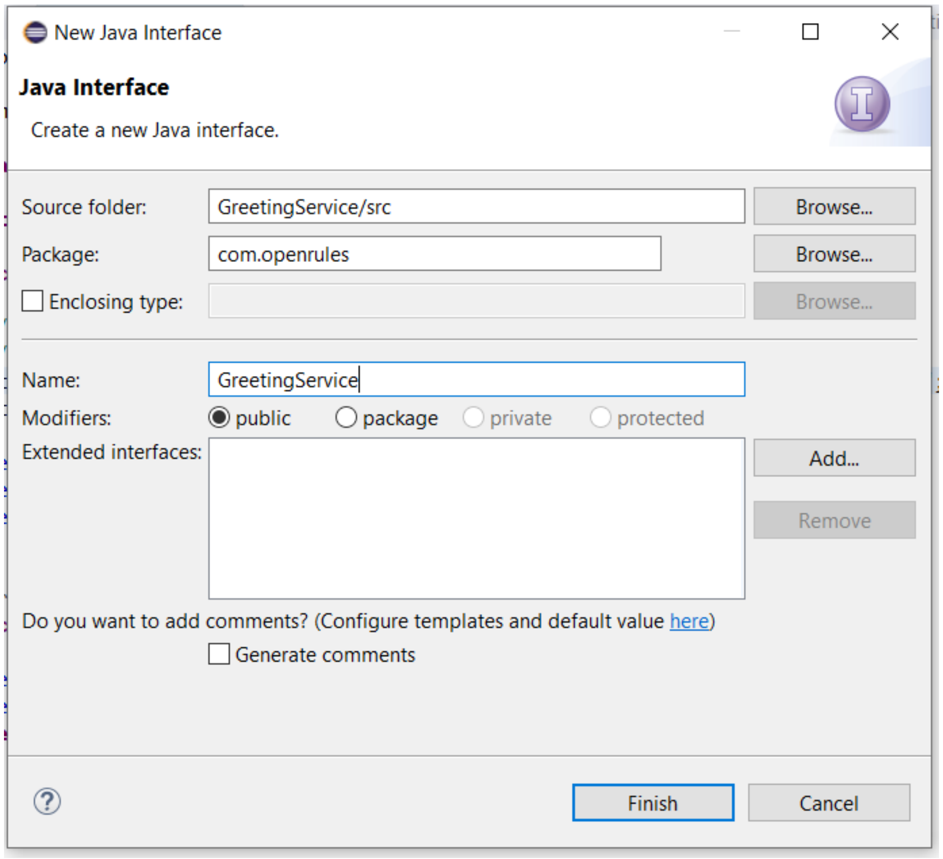
Creates folders corresponding to packages.

Source folder: GreetingService/src

Name: com.openrules

☐ Create package-info.java

Right-click on this package, select New Java Interface:


New Java Interface

Java Interface

Create a new Java interface.

Source folder: GreetingService/src

Package: com.openrules

☐ Enclosing type:

Name: GreetingService

Modifiers: [✓ public] package private protected

Extended interfaces:

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

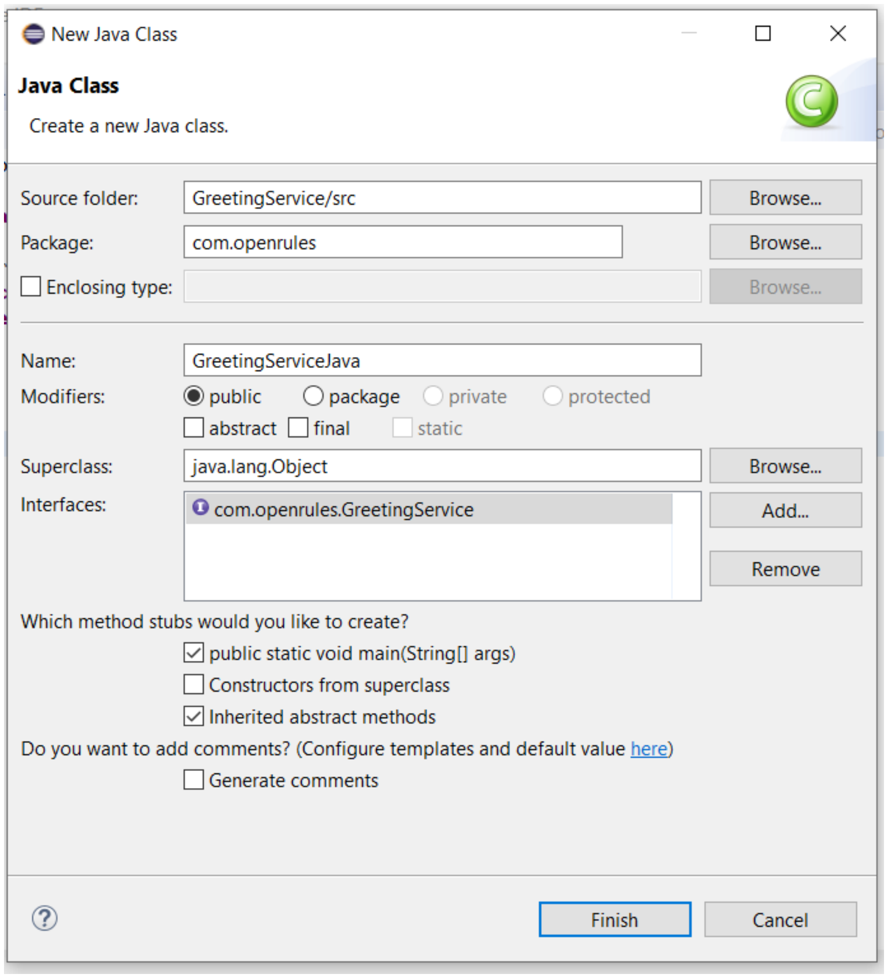
Double-click to the created file a GreetingService.java and add to it one method "generateGreetingFor":

```
package com.openrules;

public interface GreetingService {

    public String generateGreetingFor(Customer customer);

}
```

Create a New Java Class "Customer" in the same package "com.openrules":

```
package com.openrules;

public class Customer {

    String name;

    public String getName() {
        return name;
    }

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

}
```

Right-click on "com.openrules", select New+Class to create the class GreetingServiceJava that implements the interface GreetingService:



Source folder: GreetingService/src
Package: com.openrules
☐ Enclosing type:
Name: GreetingServiceJava
Modifiers: [✓ public] package private protected
☐ abstract ☐ final ☐ static
Superclass: java.lang.Object
Interfaces: com.openrules.GreetingService
Which method stubs would you like to create?
☑ public static void main(String[] args)
☐ Constructors from superclass
☑ Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
☐ Generate comments

Modify the created file “GreetingServiceJava.java” as follows:

```
package com.openrules;

public class GreetingServiceJava implements GreetingService {

    @Override
    public String generateGreetingFor(Customer customer) {
        return "Hello, " + customer.getName() + "!";
    }
}
```

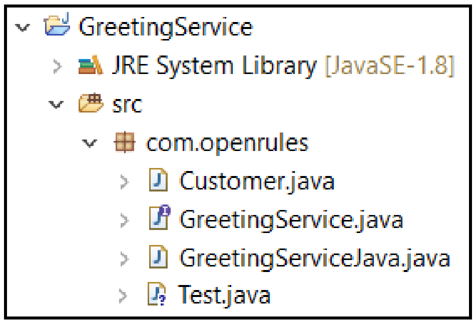
To test this service as a stand-alone program, we may create another class Test.java:

```
public class Test {

    public static void main(String[] args) {

        GreetingServiceJava service = new GreetingServiceJava();

        Customer customer = new Customer();
        customer.setName("Robinson");
        customer.setGender("Female");
        customer.setMaritalStatus("Married");
        customer.setCurrentHour(20);

        String helloStatement = service.generateGreetingFor(customer);

        System.out.println(helloStatement);
    }
}
```

Right-click on the file "Test.java" and select "Run As Java Application". It will generate the greeting:

```
Hello, Robinson!
```

Our simple greeting service in Java is completed and tested. Your Eclipse project looks as below:

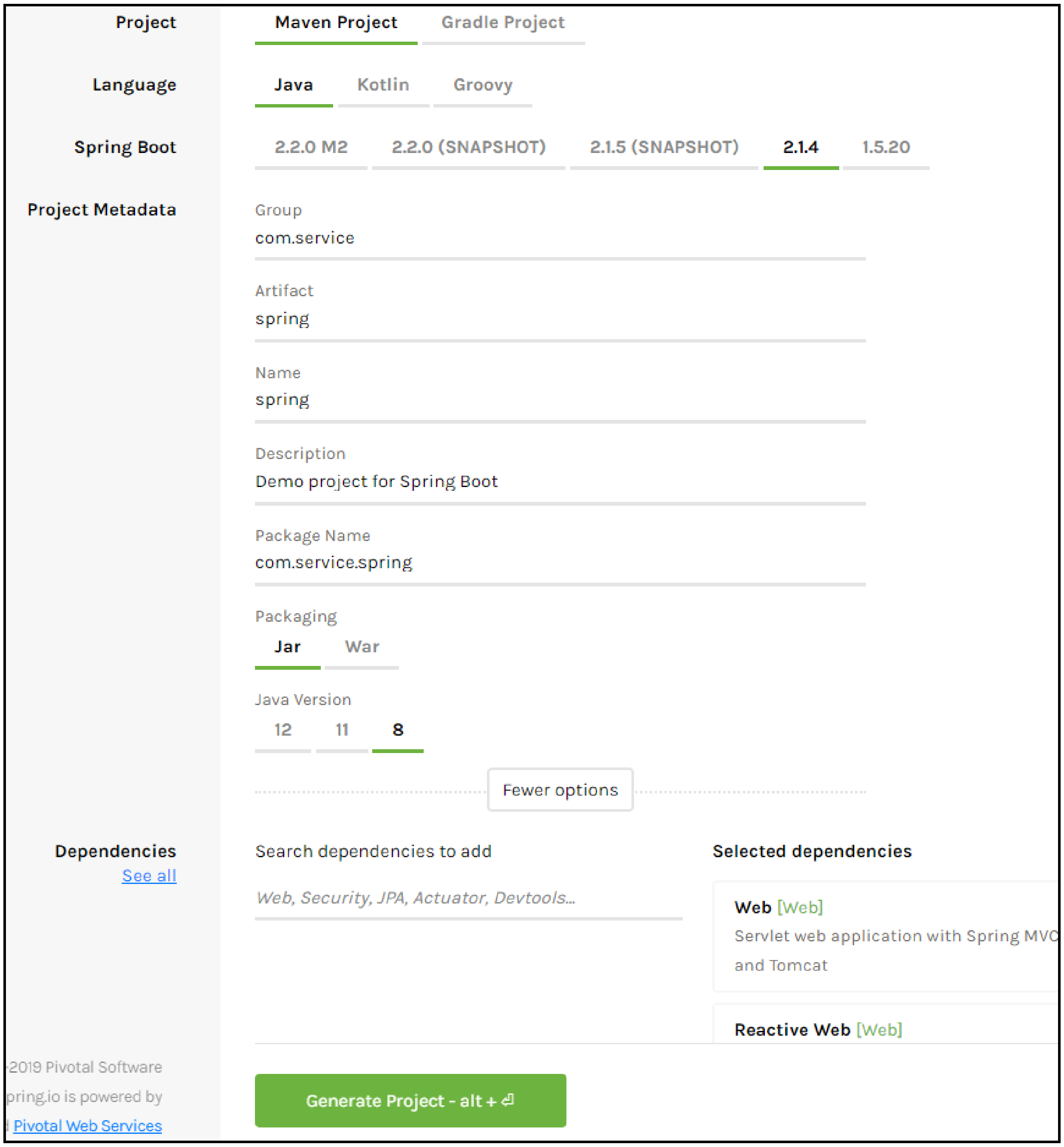


```
GreetingService
  JRE System Library [JavaSE-1.8]
  src
    com.openrules
      Customer.java
      GreetingService.java
      GreetingServiceJava.java
      Test.java
```

Now it's time to migrate this service to a web-based microservice using Spring Boot.

Creating Spring Boot Web Application

This section will give you a quick taste of Spring Boot by creating your own Spring Boot-based microservice. The simplest way to create a Spring Boot project is to use Spring Initializr. To bootstrap your new Spring Boot project got to https://start.spring.io/ and enter the following data:



Project	Maven Project (selected), Gradle Project
Language	Java (selected), Kotlin, Groovy
Spring Boot	2.2.0 M2, 2.2.0 (SNAPSHOT), 2.1.5 (SNAPSHOT), 2.1.4 (selected), 1.5.20
Project Metadata	
Group	com.service
Artifact	spring
Name	spring
Description	Demo project for Spring Boot
Package Name	com.service.spring
Packaging	Jar (selected), War
Java Version	12, 11, 8 (selected)

Fewer options

Dependencies
See all

Search dependencies to add
Web, Security, JPA, Actuator, Devtools...

Selected dependencies

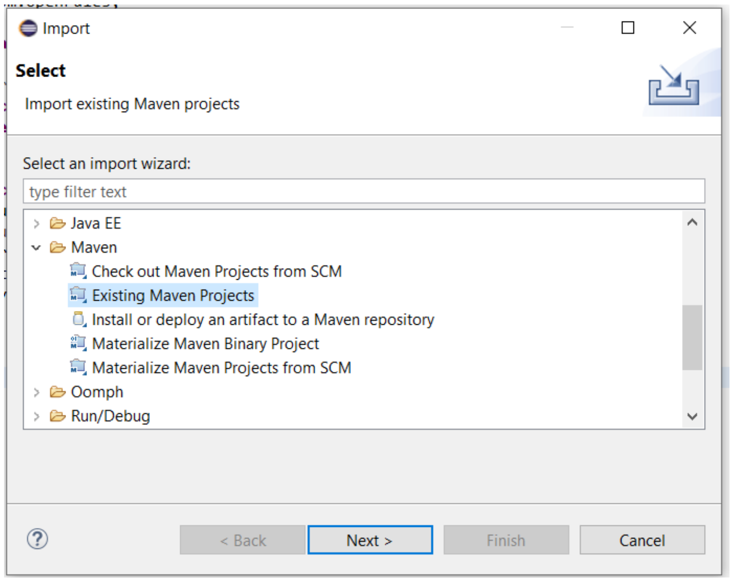
Web [Web]
Servlet web application with Spring MVC and Tomcat

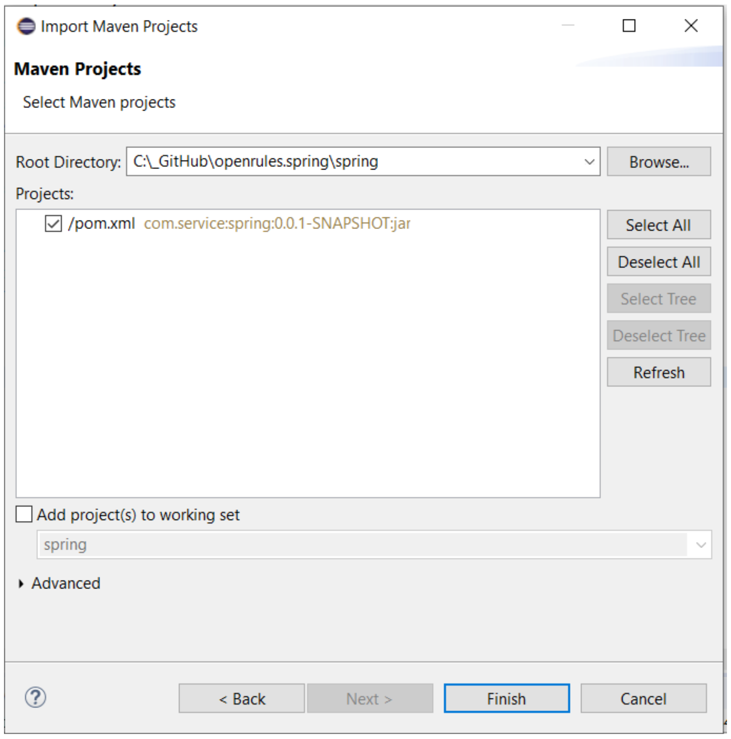
Reactive Web [Web]

2019 Pivotal Software
pring.io is powered by
Pivotal Web Services

Generate Project - alt + ⏎

When you click on “Generate Project”, Spring Initializr will create and download the file “spring.zip” into your Downloads folder. Extract the downloaded zip file into your Eclipse workspace folder. In your Eclipse select ‘Import Project”/ “Existing Maven Projects”:


Import
Select
Import existing Maven projects
Select an import wizard:
type filter text
Java EE
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Binary Project
Materialize Maven Projects from SCM
Oomph
Run/Debug
< Back
Next >
Finish
Cancel



Root Directory: C:_GitHub\openrules.spring\spring

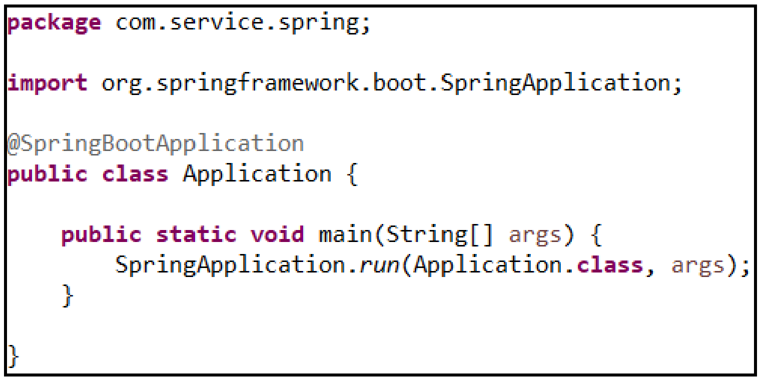
Projects:

☑ /pom.xml com.service:spring:0.0.1-SNAPSHOT:jar

☐ Add project(s) to working set

spring

It will create a new project “spring”. This project already contains the file “Application.java”:

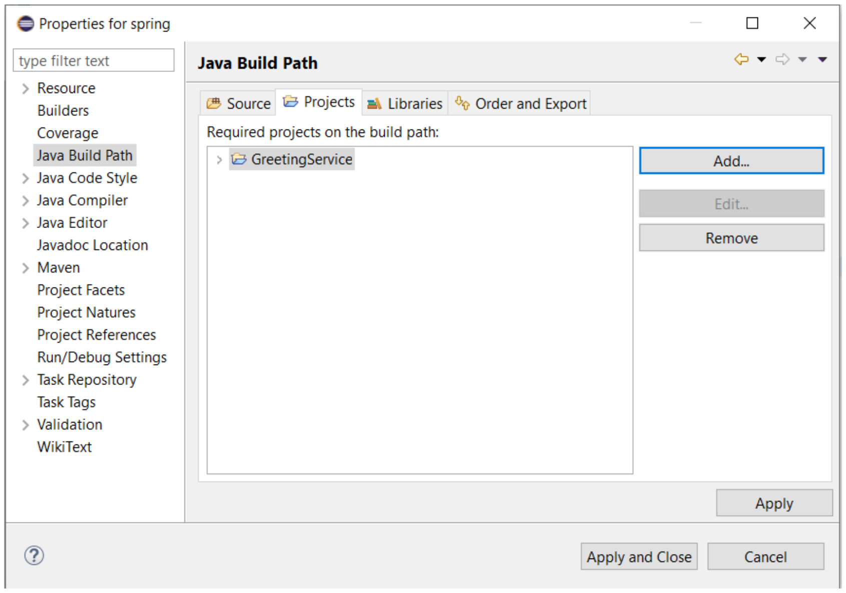


```
package com.service.spring;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

This is the main application class with *@SpringBootApplication* annotation for our future Spring Boot application. If you right-click on this file "Application.java" and select "Run as Java Application" the Spring Boot will start the embedded Tomcat server, deploy the application on the Tomcat, and will wait for http requests on the port "localhost:8080". But this application doesn't have any services yet. In the next section we will add our Java-based GreetingService to this web application.

Adding Java-based Greeting Microservice

To add different services to this application, you need to create a Java class called a REST Controller that may include different services waiting to be executed upon the proper http request. We will call our REST Controller "GreetingController" and it will include our GreetingService defined in the project "GreetingService". To make sure that the Spring Boot project "spring" is aware of our regular Java project "GreetingService" we need to add it to the classpath of our project "spring". Right-click on "spring", select "Properties", and add GreetingService as shown below:



Required projects on the build path:

GreetingService

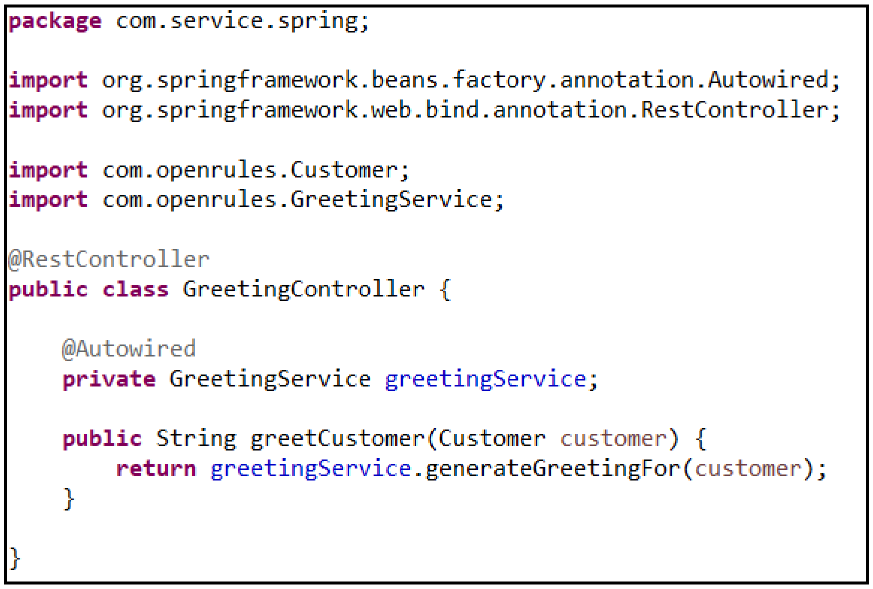
Now create a new Java class "GreetingController" in the same folder "com.service.spring" where SpringBoot placed the above class Application. Here is the initial version:

```
package com.service.spring;

import com.openrules.GreetingService;

public class GreetingController {

    private GreetingService greetingService;

}
```

We will use Spring Boot dependency injection facilities by adding an annotation *@Autowired* to the definition of our service. When you type Eclipse will automatically add the proper import:

```
package com.service.spring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;

import com.openrules.GreetingService;

@RestController
public class GreetingController {

    @Autowired
    private GreetingService greetingService;

}
```

To handle the incoming http requests for a greeting service, this controller should include a method that will accept a Customer object as a parameter and returns a generate greeting message as a response. Let’s add such a method by calling it “greetCustomer”:



```
package com.service.spring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;

import com.openrules.Customer;
import com.openrules.GreetingService;

@RestController
public class GreetingController {

    @Autowired
    private GreetingService greetingService;

    public String greetCustomer(Customer customer) {
        return greetingService.generateGreetingFor(customer);
    }

}
```

Now, we need to enhance this simple Java method as we want it to be automatically called when our web application receives an http POST request through the URL “*/greeting*” with a JSON object that has the same properties as the class Customer (now it is just a Customer’s name). Sprint Boot allows us to do it by adding the following annotations to the method “greetCustomer”:

@RequestMapping(path=”/greeting”, method={RequestMethod.POST})

public String greetCustomer(@RequestBody Customer customer) {...}

Here is the complete implementation of the GreetingController (when you type the annotations, Eclipse will automatically add the corresponding imports):

```
package com.service.spring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.openrules.Customer;
import com.openrules.GreetingService;

@RestController
public class GreetingController {

    @Autowired
    private GreetingService greetingService;

    @RequestMapping( path="/greeting", method= {RequestMethod.POST} )
    public String greetCustomer(@RequestBody Customer customer) {
        return greetingService.generateGreetingFor(customer);
    }
}
```

We are not done yet. Our “greetingService” will be @Autowired by Spring using a special class annotated of the type @Configuration. So, you need to create a new class (call it “DecisionFactory”) annotated it with @Configuration. For each service this class should include a method annotated with @Bean that returns an instance of the service. In our case it will be the method “buildGreetingService” as described below:

```
package com.service.spring;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.openrules.GreetingService;
import com.openrules.GreetingServiceJava;

@Configuration
public class DecisionFactory {

        @Bean
        public GreetingService buildGreetingService() {
            return new GreetingServiceJava();
        }
}
```

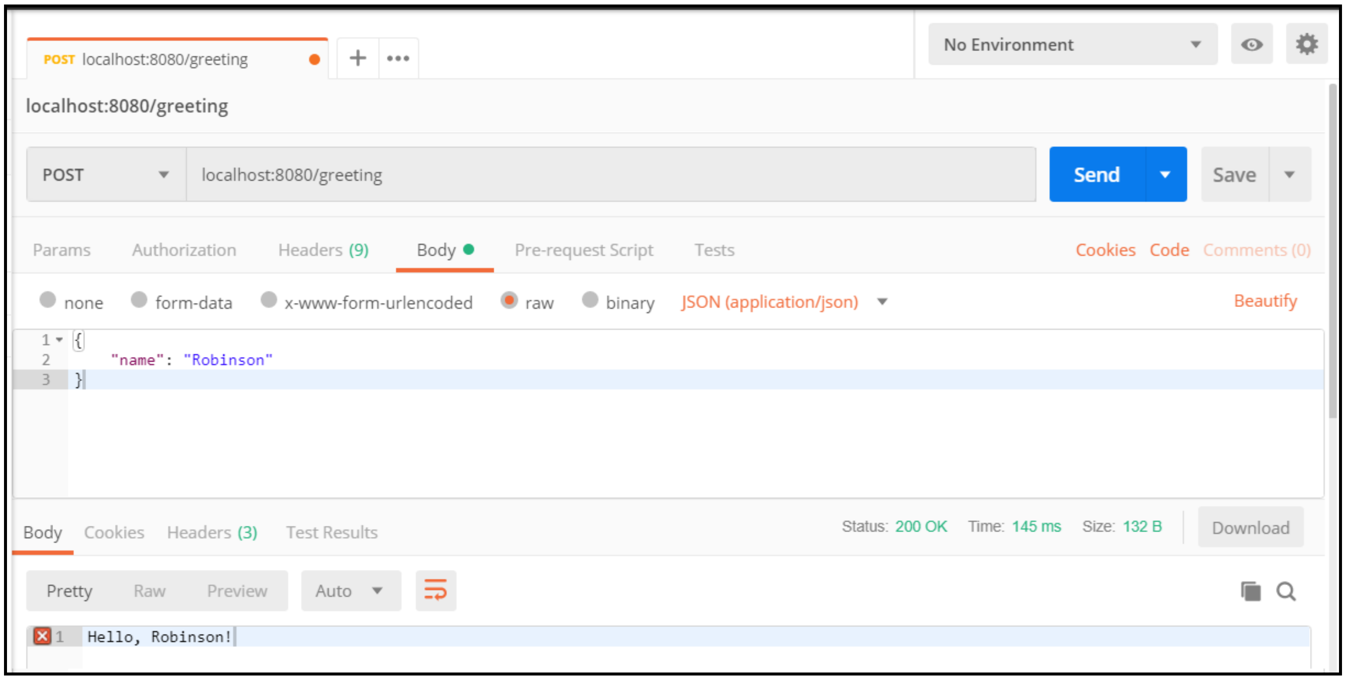
This completes the development of our Spring Boot application with one greeting that will be called using URL “*/greeting*”.

To test our web application, right-click on Application and select "Run As Java Application". It will start the embedded Tomcat, deploy the latest version of our spring project, and will wait to http-request. Here is the protocol from the Eclipse's Console view:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.4.RELEASE)

2019-04-28 17:49:52.379  INFO 20200 --- [           main] com.service.spring.Application           : Starting Application on DESKTOP-A
2019-04-28 17:49:52.382  INFO 20200 --- [           main] com.service.spring.Application           : No active profile set, falling ba
2019-04-28 17:49:53.475  INFO 20200 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s):
2019-04-28 17:49:53.499  INFO 20200 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2019-04-28 17:49:53.499  INFO 20200 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache
2019-04-28 17:49:53.591  INFO 20200 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebA
2019-04-28 17:49:53.591  INFO 20200 --- [           main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initi
2019-04-28 17:49:53.841  INFO 20200 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'app
2019-04-28 17:49:54.292  INFO 20200 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (
2019-04-28 17:49:54.295  INFO 20200 --- [           main] com.service.spring.Application           : Started Application in 2.229 seco
2019-04-28 17:50:06.461  INFO 20200 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherSer
2019-04-28 17:50:06.461  INFO 20200 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherS
2019-04-28 17:50:06.466  INFO 20200 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 4 ms
```

Testing Spring Boot Web Application

To create http-requests for this web application, we will use POSTMAN, a popular tool that can be downloaded for free from https://www.getpostman.com/. After installation and start, you may fill out this POSTMAN's form:



localhost:8080/greeting

POST localhost:8080/greeting

```
{
    "name": "Robinson"
}
```

Status: 200 OK Time: 145 ms Size: 132 B

```
Hello, Robinson!
```

You should select the method POST (from the drop-down list), type the URL "localhost:8080/greeting", enter a simple JSON structure

```
{
    "name": "Robinson"
}
```

and click on "Send". The POSTMAN will send the proper http-request to our web application, that will execute our GreetingService and will return the string "Hello, Robinson!" at the bottom of the form.

Thus, our Spring Boot web application with a Java-based GreetingService works fine. Naturally, Spring can deploy the same application not only with the embedded Tomcat but using any container and any cloud environment such as AWS. While our application is too simple, but it already demonstrates how to create Java-based microservices with Spring Boot.

Moving Greeting Service to OpenRules

Our Java-based greeting service doesn't need any business logic to produce a greeting message like "Hello, Robinson!". Let's make it a bit more sophisticated. Instead of "Hello, Robinson!" this service should be able to produce greetings like "Good Morning, Mrs. Robinson!" if it is morning and the customer Robinson is a married woman or "Good Afternoon, Mr. Robinson!" if it is afternoon and Robinson is a man. The latest OpenRules installation "openrules.models" already includes a sample decision project called "Greeting" that provides this functionality. However, this is a stand-alone project, and we will need to convert it to a decision microservice that will replace our GreetingServiceJava inside the Spring Boot web application. First, let's migrate the standard OpenRules project "Greeting" to our GreetingService project.

We will assume that you already installed an OpenRules evaluation version by downloading the standard folder "openrules.models". Let's import two OpenRules projects "Greeting" and "openrules.config" into our workspace using Eclipse File+Import+General+Existing Projects into Workspace. Now you should see two new folders "Greeting" and "openrules.config" in our Eclipse workspace. Follow these steps to migrate the project Greeting to our GreetingService.

Step 1. Copy "Customer.java" from the project Greeting to our project GreetingService. Now along with the "name" the class Customer will also include the following attributes:

```
public class Customer {

    String name;
    String gender;
    String maritalStatus;
    int    currentHour;

    ....
```

along with their getters and setters.

Step 2. Copy "GreetingResponse.java" from the project Greeting to our project GreetingService.

It will be used to save the results or our greeting decision in the following attributes:

```
public class GreetingResponse {

    String greeting;
    String salutation;
    String helloStatement;

    ....
```

Step 3. Copy the folder "rules" from the folder Greeting to our folder GreetingService. It contains the decision model in the form of the following Excel files:

DecisionModel.xls

It contains only one Environment table that defines the structure of this decision model:

Environment	
include	Rules.xls
	Glossary.xls
	../../openrules.config/DecisionTemplates.xls

Glossary.xls

It contains two tables. The table Glossary

Glossary glossary		
Variable	**Business Concept**	**Attribute**
Name	Customer	name
Gender		gender
Marital Status		maritalStatus
Current Hour		currentHour
Greeting	GreetingResponse	greeting
Salutation		salutation
Hello Statement		helloStatement

defines all decision variables distributed between two business concepts "Customer" and "GreetingResponse" with attributes that correspond to our Java classes with the same names.

The table DecisionObject maps these business concepts and the corresponding Java objects:

DecisionObject decisionObjects	
Business Concept	**Business Object**
Customer	:= decision.get("Customer")
GreetingResponse	:= decision.get("GreetingResponse")

Rules.xls

It contains three decision tables which define our greeting logic:

DecisionTable DefineGreeting	
If	Then
Current Hour	**Greeting**
[0..11)	Good Morning
[11..17)	Good Afternoon
[17..22)	Good Evening
[22-24]	Good Night

DecisionTable DefineSalutation					
Condition		Condition		Conclusion	
Gender		**Marital Status**		**Salutation**	
Is	Male			Is	Mr.
Is	Female	Is	Married	Is	Mrs.
Is	Female	Is	Single	Is	Ms.

DecisionTable DefineHelloStatement	
Conclusion	
Hello Statement	
Is	Greeting + ", " + Salutation + " " + Name + "!"

It also contains the file "**Goals.xls**" (that was automatically generated by build.bat) that defines the top-level decision "DecisionHelloStatement":

Decision DecisionHelloStatement
ActionExecute
Decision Tables
DefineGreeting
DefineSalutation
DefineHelloStatement

Step 4. Copy "Main.java" from the project Greeting to our project GreetingService. It was used to test our stand-alone decision model:

```
import com.openrules.ruleengine.Decision;

public class Main {

    public static void main(String[] args) {

        String fileName = "classpath:/Goals.xls";
        String decisionName = "DecisionHelloStatement";
        Decision decision = new Decision(decisionName,fileName);
        decision.put("FEEL", "On");
        decision.put("report", "On");

        Customer customer = new Customer();
        customer.setName("Robinson");
        customer.setGender("Female");
        customer.setMaritalStatus("Married");
        customer.setCurrentHour(20);

        GreetingResponse response = new GreetingResponse();

        decision.put("Customer", customer);
        decision.put("GreetingResponse", response);
        decision.execute();

        decision.log("\nProduced Hello Statement: " + response.getHelloStatement());
    }
}
```

Note that this method uses URL "**classpath:/Goals.xls**". It means that the rules repository folder "rules" should be in the project's classpath. To make sure that it is true, check if the folder "rules" has the same Eclipse icon as the folder "src". If not, right-click on the folder "rules" and select "Build Path + Use as Source Folder".

Step 5. Copy "run.bat" from Greeting to GreetingService. Double-click on this file to make sure that the decision model still works inside the new folder.

So, after these 5 steps our folder GreetingService includes a working OpenRules-based decision project.

CONVERTING OPENRULES DECISION PROJECT TO SPRING BOOT MICROSERVICE

Our Spring Boot web application uses the greeting service define in the class GreetingServiceJava. Now you will use Main.java as a prototype to create a new Java class

GreetingServiceOpenRules that will replace GreetingServiceJava in our web application. Here is its code:

```
package com.openrules;

import com.openrules.ruleengine.Decision;

public class GreetingServiceOpenRules implements GreetingService {

    Decision decision;

    public GreetingServiceOpenRules() {

        String fileName = "classpath:/Goals.xls";
        String decisionName = "DecisionHelloStatement";
        decision = new Decision(decisionName,fileName);
        decision.put("FEEL", "On");
        decision.put("report", "On");
    }

    @Override
    public String generateGreetingFor(Customer customer) {
        decision.put("Customer", customer);
        GreetingResponse response = new GreetingResponse();
        decision.put("GreetingResponse", response);
        decision.execute();
        return response.getHelloStatement();
    }
}
```

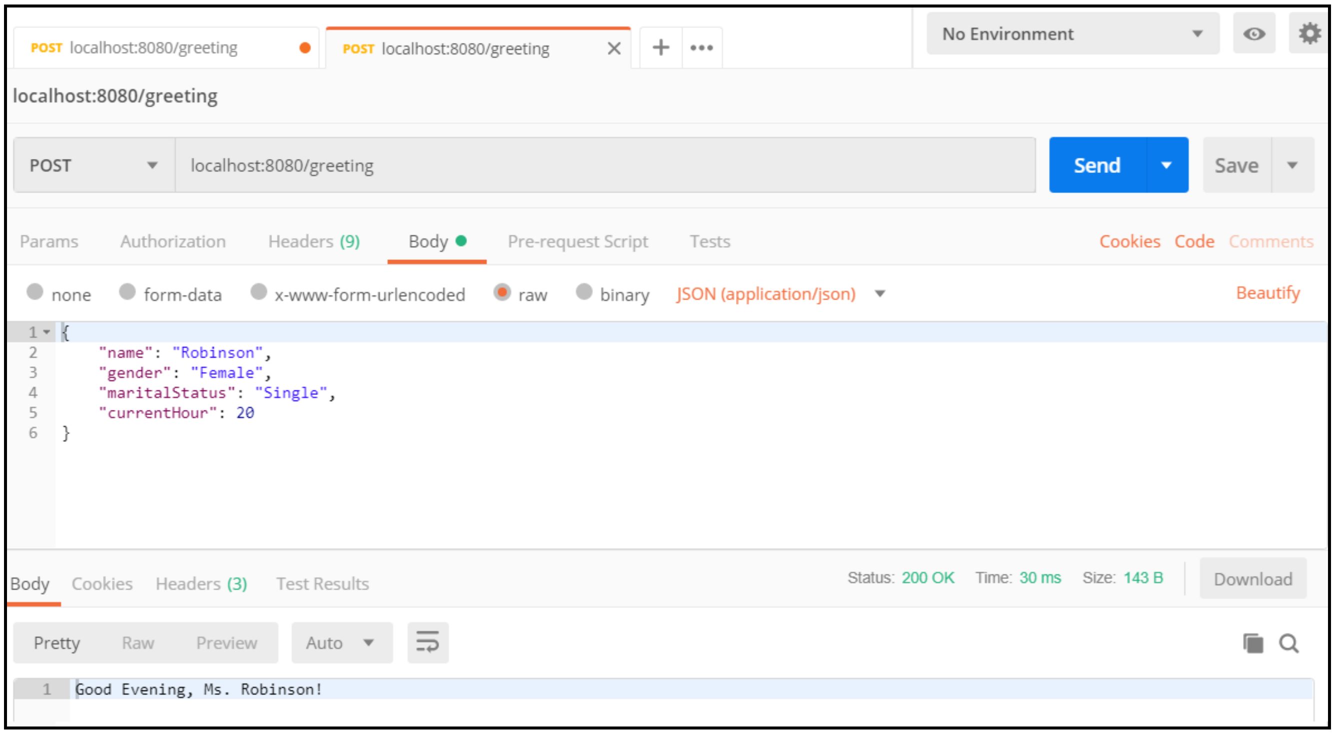
As you can see, the constructor GreetingServiceOpenRules creates an instance of the OpenRules class Decision similarly as it was done in Main.java. Then the overridden method generateGreetingFor(Customer customer) runs this decision using the instance of the class Customer that comes as a parameter (from an http request) and a new instance of GreetingResponse to save the decision's results.

To be able to run this microservice we only need to modify the class DecisionFactory in the project "spring":

```java
package com.service.spring;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.openrules.GreetingService;
import com.openrules.GreetingServiceOpenRules;

@Configuration
public class DecisionFactory {

        @Bean
        public GreetingService buildGreetingService() {
            //return new GreetingServiceJava();
            return new GreetingServiceOpenRules();
        }
}
```

We simply commented out GreetingServiceJava and replaced it with GreetingServiceOpenRules. Now we can run our Spring Boot application with the new OpenRules service. To do that, right-click on the class Application and select "Run as Java Application". If you receive an error, you may select "Debug as Java Application", add breakpoints to suspicious Java classes and do regular Eclipse-based debugging.

To test our decision microservice, open again PORTMAN, enter the JSON code as on the picture below and click on "Send":



```json
{
    "name": "Robinson",
    "gender": "Female",
    "maritalStatus": "Single",
    "currentHour": 20
}
```

```
Good Evening, Ms. Robinson!
```

As you can see, our OpenRules-based microservice produced the expected result "Good Evening, Ms. Robinson!".

Conclusion

In this tutorial we demonstrated how to create a Sprint Boot web application with a simple Java-based greeting service. Then we replaced it to a little bit more sophisticated greeting service that utilizes OpenRules. In a similar manner we can add any OpenRules-based service and take advantage of the powerful Spring framework for creating decision microservices available from any server or a cloud environment.

Technical Support

Direct all your technical questions to support@openrules.com or to this Discussion Group.