CREATING OPENRULES DECISION MICROSERVICES

WITH

MAVEN, SPRINGBOOT, AND DOCKER

OpenRules, Inc.

www.openrules.com

July-2019

TABLE OF CONTENTS

Introduction 3
What You'll Do 3
What Should Be Pre-Installed 4
Mavenizing OpenRules Configuration 4
Creating a Maven OpenRules Project in Eclipse 6
Defining Decision Model 7
Adding a Java Interface 11
Creating Spring Boot Web Application 16
Adding Service to Spring Boot Application 18
Testing Decision Service with POSTMAN 22
Testing Decision Service with a Java Client 24
Adding Another OpenRules Decision Service 25
Deploying Decision Service to Docker 35
Conclusion 36
Technical Support 36

INTRODUCTION

Nowadays microservices quickly become a highly popular architectural approach. They have shown essential advantages over the legacy style of monolithic single applications:

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

It's only natural to deploy Business Decision Models created and tested by business users as **decision microservices**. This tutorial provides a sampling of how to build Decision Microservices with Spring Boot and OpenRules and containerize them with Docker.

WHAT YOU'LL DO

We will explain what you need to do to create, test, and deploy an OpenRules-based decision service using SpringBoot and Docker. We will assume that you are familiar with Java and Eclipse IDE and have a high-level understanding of the Spring framework and Docker.

Following step-by-step instructions below, you do the following:

- "Mavenize" the standard OpenRules configuration
- Create a simple Maven project in Eclipse which will be used to invoke OpenRules-based service with business logic represented in Excel decision tables
- Test this from Java
- Convert this decision service to a simple REST web application built using Spring Boot and test it with POSTMAN using JSON
- Containerize this decision service using Docker and test it using POSTMAN and/or a Java-based client.

In the end, you will be ready to create and containerize your own OpenRules decision services.

What Should Be Pre-Installed

We assume that you've already installed:

- Java 1.8 or later
- Eclipse IDE
- Maven
- OpenRules evaluation (or complete) version by downloading the workspace "openrules.models"
- Docker Desktop.

When all these products are installed, start Eclipse with a new workspace called "**openrules.services**".

Mavenizing OpenRules Configuration

The standard OpenRulesinstallation workspace "openrules.models" contains a special configuration project called "openrules.config" and a set of sample projects such as "VacationDays". Import the standard OpenRules configuration project "openrules.config" from the workspace "openrules.models" to the already opened workspace "openrules.services".

We are going to use Maven as our main building tool, so first we need to "mavenize" this OpenRules configuration. We will install all jar-files from "openrules.config/lib" to the local Maven's repository, that is a directory on your computer where Maven holds all artifacts and dependencies. To do that, first we will add the following "**pom.xml**" file to "openrules.config":

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.openrules</groupId>
    <artifactId>config</artifactId>
    <version>7.0.1-SNAPSHOT</version>
    <packaging>pom</packaging>
    <name>OpenRules Config</name>
    <description>Config project for Openrules Library</description>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <openrules.version>7.0.1-SNAPSHOT</openrules.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>com.openrules</groupId>
            <artifactId>openrules.all</artifactId>
            <version>${openrules.version}</version>
        </dependency>
        <dependency>
            <groupId>com.openrules</groupId>
            <artifactId>openrules.tools</artifactId>
            <version>${openrules.version}</version>
        </dependency>

        <!-- apache -->
        <dependency>
            <groupId>org.apache.poi</groupId>
            <artifactId>poi</artifactId>
            <version>3.10-FINAL</version>
        </dependency>
        <dependency>
            <groupId>org.apache.poi</groupId>
            <artifactId>poi-ooxml</artifactId>
            <version>3.10-FINAL</version>
        </dependency>
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.6</version>
        </dependency>
        <dependency>
            <groupId>commons-lang</groupId>
            <artifactId>commons-lang</artifactId>
            <version>2.3</version>
        </dependency>
    </dependencies>

    <build>
       <resources>
         <resource>
           <directory>rules</directory>
         </resource>
       </resources>
    </build>
</project>
```

Note that along with necessary jar-files, we configured a rule repository called “rules” as a “resource” folder, so that the content of this folder will be packaged into created jars along with classes, properties files and other resources, and will be used by all decision services we plan to add to this Maven configuration.

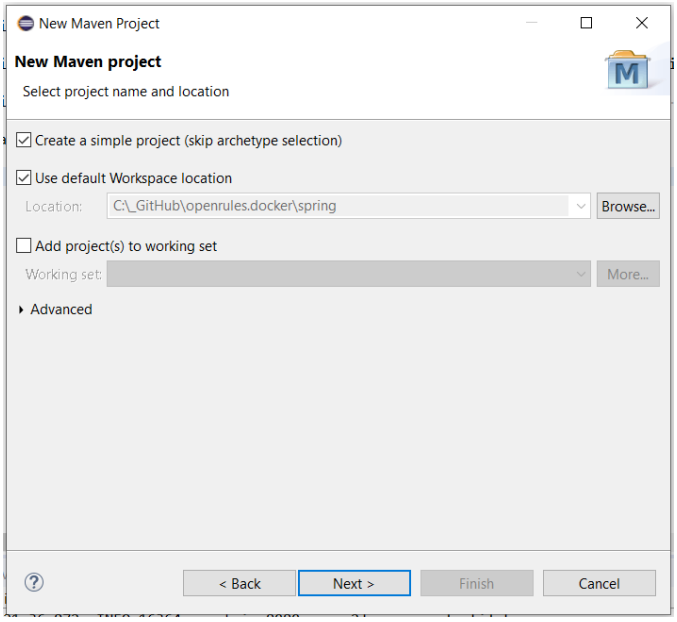
Now we will add the following file “**install.bat**” to “openrules.config”:

```
@echo off
cd %~dp0

set VERSION=7.0.1-SNAPSHOT

call mvn install -Dversion=%VERSION%

call mvn install:install-file -Dfile=lib/openrules.all.jar -DgroupId=com.openrules
-DartifactId=openrules.all  -Dversion=%VERSION% -Dpackaging=jar -DgeneratePom=true

call mvn install:install-file -Dfile=lib/com.openrules.tools.jar -DgroupId=com.openrules
-DartifactId=openrules.tools -Dversion=%VERSION% -Dpackaging=jar -DgeneratePom=true

echo Initialized
pause
```

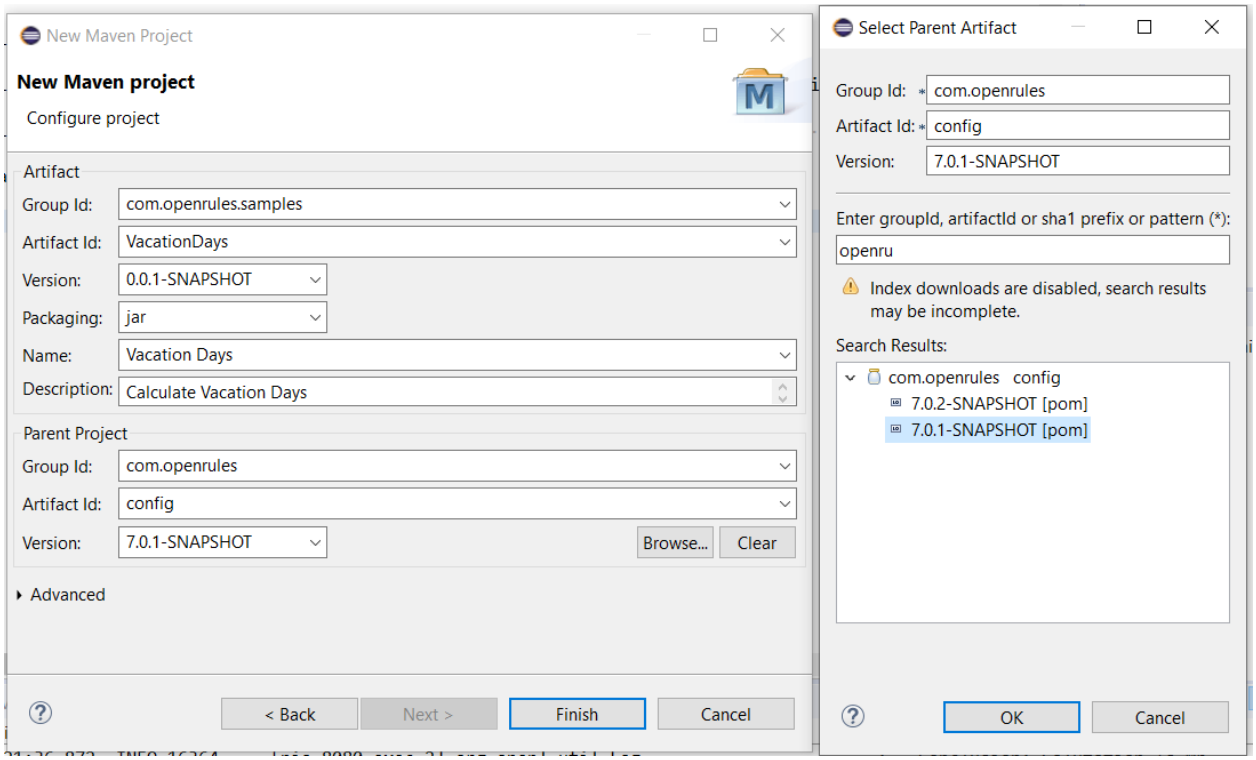
Now you may right-click on this file and select “Open With System Editor” – alternatively you may double-click on this file from File Explorer or just execute 4 commands used inside this file directly from a command line. It will build OpenRules Config 7.0.1-SNAPSHOT in your local Maven repository (but you don’t even need to look at it). Your Maven is ready to work with OpenRules.

Creating a Maven OpenRules Project in Eclipse

Now we will create a Maven project for the OpenRules service “VacationDays”. From you Eclipse File-menu, select File+New Project+Maven Project:



Click “Next” and enter the Artifact data as below. For the Parent project click on “Browse..”, start typing “openrules” and make the selections as on the right image:



Defining Decision Model

We want to build a simple decision model that calculates the number of vacation days given to an employee based on the age and years of service. Here are the business rules:

The number of vacation days depends on age and years of service.

Every employee receives at least 22 days.
Additional days are provided according to the following criteria:

1) Only employees younger than 18 or at least 60 years, or employees with at least 30 years of service will receive 5 extra days.

2) Employees with at least 30 years of service and also employees of age 60 or more, receive 3 extra days, on top of possible additional days already given.

3) If an employee has at least 15 but less than 30 years of service, 2 extra days are given. These 2 days are also provided for employees of age 45 or more. These 2 extra days can not be combined with the 5 extra days.

You already can find the proper decision model implemented in the standard OpenRules workspace "openrules.models" as a stand-alone project "VacationDays". In that project the decision model was defined in the rules repository called "rules" and use the standard OpenRules templates defined in the configuration project "openrules.config".

For the Maven-based decision services we already decided to use the common rules repository called "rules" which we added as a dependency to the file "openrules.config/pom.xml". So, now we will create the folder "**rules**" in our new Maven's project "VacationDays". Then we will add two sub-folders to the folder "rules":

- **templates** – a placeholder for the standard OpenRules templates
- **vacationDays** – a placeholder for our decision model "VacationDays".

So, now we will copy files "openrules.config/DecisionTemplates.xls" and "openrules.config/DecisionTableExecuteTemplates.xls" into the subfolder "templates". Then we will copy all Excel files from the folder "openrules.models/VacationDays/rules" to the subfolder "vacationDays". Here are Excel files and tables that implement vacation days calculation logic.

File "rules/vacationDays/**Rules.xls**":

DecisionTableMultiHit CalculateVacationDays				
If	If	If	Conclusion	
Eligible for Extra 5 Days	**Eligible for Extra 3 Days**	**Eligible for Extra 2 Days**	**Vacation Days**	
			=	22
TRUE			+=	5
	TRUE		+=	3
FALSE		TRUE	+=	2

DecisionTable SetEligibleForExtra5Days		
If	If	Then
Age in Years	**Years of Service**	**Eligible for Extra 5 Days**
< 18		TRUE
>= 60		TRUE
	>= 30	TRUE
		FALSE

DecisionTable SetEligibleForExtra3Days		
If	If	Then
Age in Years	**Years of Service**	**Eligible for Extra 3 Days**
	>= 30	TRUE
>= 60		TRUE
		FALSE

DecisionTable SetEligibleForExtra2Days		
If	If	Then
Age in Years	**Years of Service**	**Eligible for Extra 2 Days**
	[15..30)	TRUE
>= 45		TRUE
		FALSE

All decision goals, variables, and user Decision objects are defined in the file "rules/ vacationDays/**Glossary.xls**":

Glossary glossary		
Variable Name	**Business Concept**	**Attribute**
Age in Years	Employee	age
Years of Service		service
Eligible for Extra 5 Days		eligibleForExtra5Days
Eligible for Extra 3 Days		eligibleForExtra3Days
Eligible for Extra 2 Days		eligibleForExtra2Days
Vacation Days		vacationDays

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

The structure of this decision model was defined in the file "**DecisionModel.xls**" in this Environment table that used to look as below:

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

The third include-statement referred to the standard templates located two levels above the old project "VacationDays". Now we keep these templates in the subfolder "templates" that is on the same level as the folder "vacationDays". So, we need to adjust the Environment table as follows:

Environment	
include	Glossary.xls
	Rules.xls
	../templates/DecisionTemplates.xls

The folder "rules/vacationDays" also include the file "**Test.xls**" that specifies Datatype "Employee" and creates several test-cases with expected results. To make sure that the new decision model still works, copy file "build.bat" and "run.bat" from openrules.models/VacationDays/ to our new project "VacationDays", and make the following changes in them:

File "build.bat":

```
set DECISION_NAME="Vacation Days"
set INPUT_FILE_NAME=rules/vacationDays/DecisionModel.xls
set OUTPUT_FILE_NAME=rules/vacationDays/Goals.xls
@echo off
cd %~dp0
call ..\openrules.config\projectBuild
pause
```

File "run.bat":

```
set DECISION_NAME="Vacation Days"
set FILE_NAME=rules/vacationDays/Test.xls
cd %~dp0
call ..\openrules.config\projectRun
pause
```

These bat-files use openrules.config's file projectBuild.bat and projectRun.bat. Both these files referred to compiles classes using set CLASSES=./bin. However, in the Maven's projects compiled classes are created not in "bin", but rather in "target". So, we need in both these bat-

files replace the setting to
set CLASSES=./target/classes;./target/test-classes

Now, we can double-click on "**build.bat**" and it will build an execution path for this model and will save it in the file "**Goals.xls**". Then double-click on "**run.bat**" and it will execute all test-cases producing the following results:

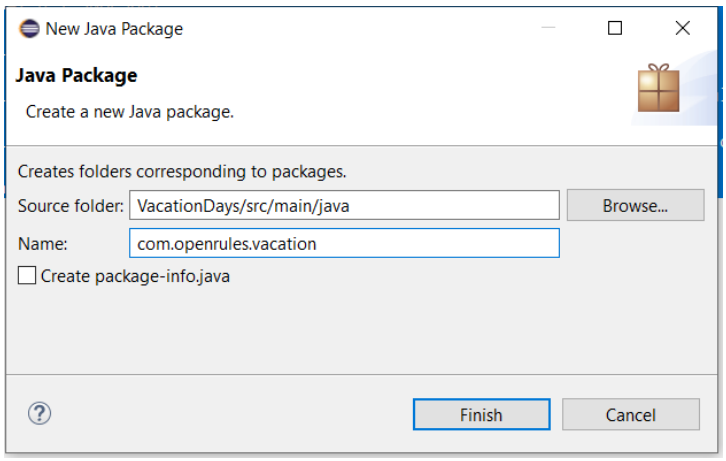
```
Execute SetEligibleForExtra3Days
  Assign: Eligible for Extra 3 Days  = true
Execute SetEligibleForExtra2Days
  Assign: Eligible for Extra 2 Days  = true
Execute CalculateVacationDays
  Conclusion: Vacation Days  = 22
  Conclusion: Vacation Days  += 27
  Conclusion: Vacation Days  += 30
Validating results for the test <Test E>
Test E was successful
Executed test Test E in 13 ms

RUN TEST: Test F 2019-06-22 13:25:36.378
Execute SetEligibleForExtra5Days
  Assign: Eligible for Extra 5 Days  = true
Execute SetEligibleForExtra3Days
  Assign: Eligible for Extra 3 Days  = true
Execute SetEligibleForExtra2Days
  Assign: Eligible for Extra 2 Days  = true
Execute CalculateVacationDays
  Conclusion: Vacation Days  = 22
  Conclusion: Vacation Days  += 27
  Conclusion: Vacation Days  += 30
Validating results for the test <Test F>
Test F was successful
Executed test Test F in 15 ms
1 test(s) out of 6 failed!
Executed all tests cases in 125 ms - 2019-06-22 13:25:36.396
done
```

Adding a Java Interface

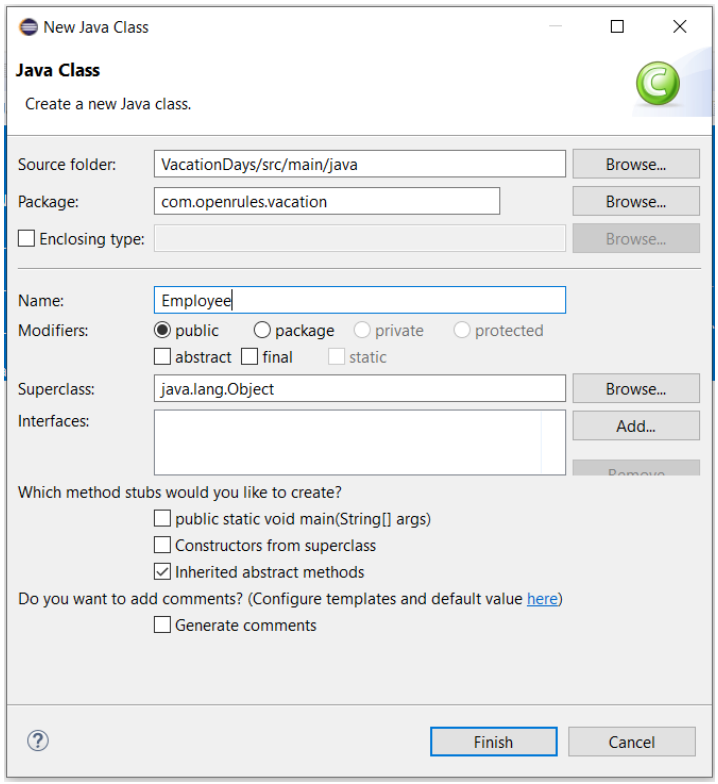
To execute the same model from Java, we will create 3 Java classes:

- **Employee**.java – to define test-employees
- **VacationDaysService**.java – to specify our service
- **Test**.java – to test the service locally.

So first, we create a new Java package "com.openrules.vacation" in the folder "src/maim/java":



and then add a new class Employee:



The class "**Employee**" will contain the same attributes that were used in the above Glossary and the Datatype table "Employee":

```
public class Employee {
    
    String id;
    int vacationDays;
    boolean eligibleForExtra5Days;
    boolean eligibleForExtra3Days;
    boolean eligibleForExtra2Days;
    int age;
    int service;
```

Then we will right-click on “Employee.java” and use “Source” + ”Generate Getters and Setters” + “Generate toString()” to complete this class. Here is the class “**VacationDaysService**”:

```
package com.openrules.vacation;

import com.openrules.ruleengine.Decision;
import com.openrules.ruleengine.OpenRulesEngine;
import com.openrules.vacation.Employee;

public class VacationDaysService {

    OpenRulesEngine engine;

    public VacationDaysService() {

        String fileName = "classpath:/vacationDays/Goals.xls";
        engine = new OpenRulesEngine(fileName);
    }

    public int run(Employee employee) {
        String decisionName = "Vacation Days";
        Decision decision = new Decision(decisionName,engine);
        decision.put("FEEL", "On");
        decision.put("Employee", employee);
        decision.execute();
        return employee.getVacationDays();
    }
}
```

Note, that we refer to main Excel file in the rules repository as “classpath:/Goals.xls”. It means all Excel files that our folder “rules” also should be in the Maven classpath. This has been already guaranteed when we added the dependency for “rules” to the “pom.xml”.

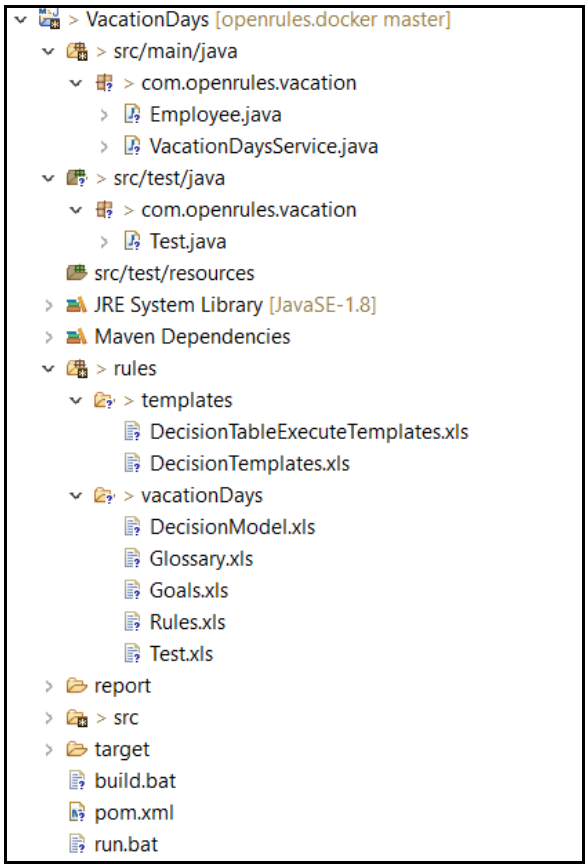
Now, we are ready to test our modified decision model “VacationDaysService”. The tests should be placed to the automatically created folder “src/test/java”. So, we will add here a new package “com.openrules.vacation”, and then we will add to this package a new class “**Test**.java”:

```
package com.openrules.vacation;

import com.openrules.vacation.Employee;

public class Test {

public static void main(String[] args) {
        VacationDaysService service = new VacationDaysService()
        Employee employee = new Employee();
        employee.setId("Robinson");
        employee.setAge(47);
        employee.setService(20);
        service.run(employee);
        System.out.println("VacationDaysService: " + employee);
    }
}
```

Right-click on the file “Test.java” and select “Run As Java Application”. It will produce the results that look like below:

```
[INFO] Log - -- SetEligibleForExtra5Days
[INFO] Log - -- SetEligibleForExtra3Days
[INFO] Log - -- SetEligibleForExtra2Days
[INFO] Log - -- CalculateVacationDays
[INFO] Log - -Execute SetEligibleForExtra5Days
[INFO] Log - -  Assign: Eligible for Extra 5 Days  = false
[INFO] Log - -Execute SetEligibleForExtra3Days
[INFO] Log - -  Assign: Eligible for Extra 3 Days  = false
[INFO] Log - -Execute SetEligibleForExtra2Days
[INFO] Log - -  Assign: Eligible for Extra 2 Days  = true
[INFO] Log - -Execute CalculateVacationDays
[INFO] Log - -  Conclusion: Vacation Days  = 22
[INFO] Log - -  Conclusion: Vacation Days  += 24
VacationDaysService: Employee [id=Robinson, vacationDays=24,
```

We may consider that our decision service “VacationDays” has been tested as a stand-alone application and is ready for further deployment. Our Eclipse project now looks as below:

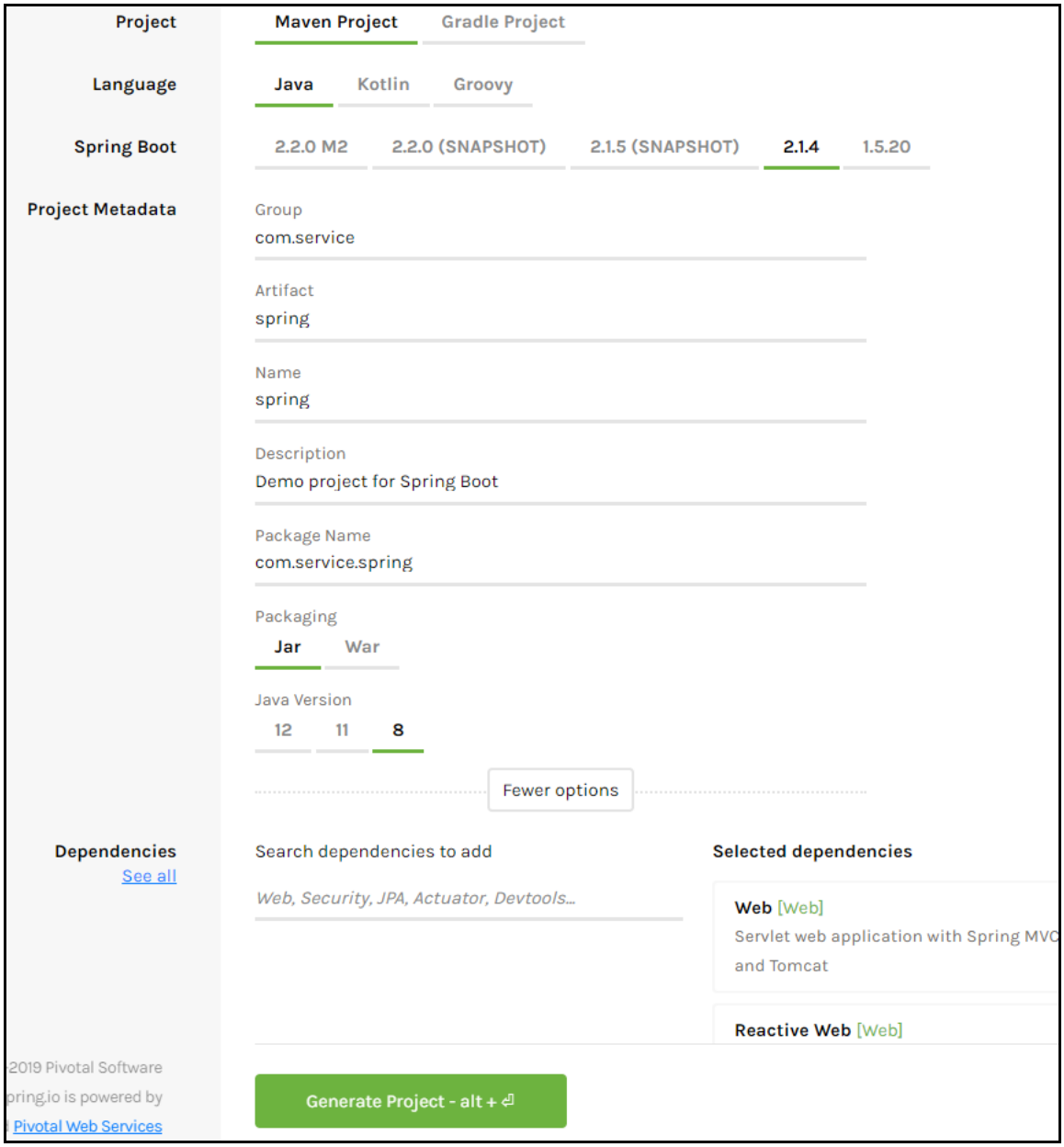


To complete its Maven's installation, we should right-click on "VacationDays/pom.xml" and select "Run As" + "Maven Install".

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

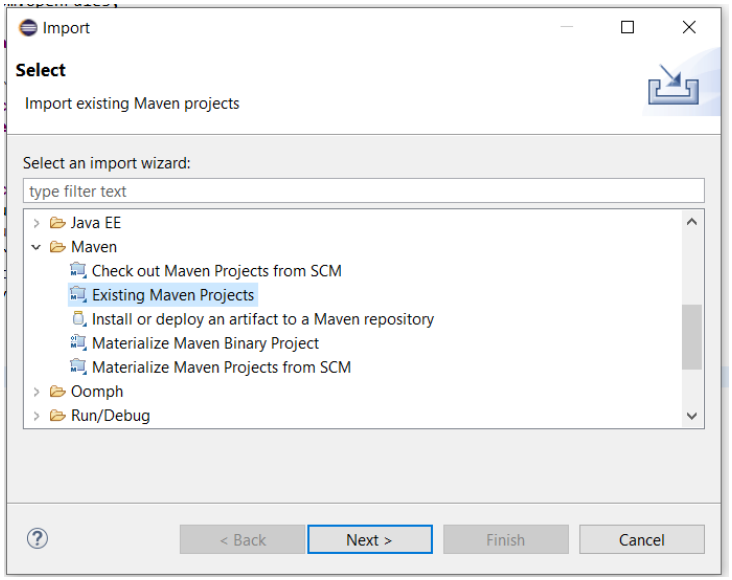
CREATING SPRING BOOT WEB APPLICATION

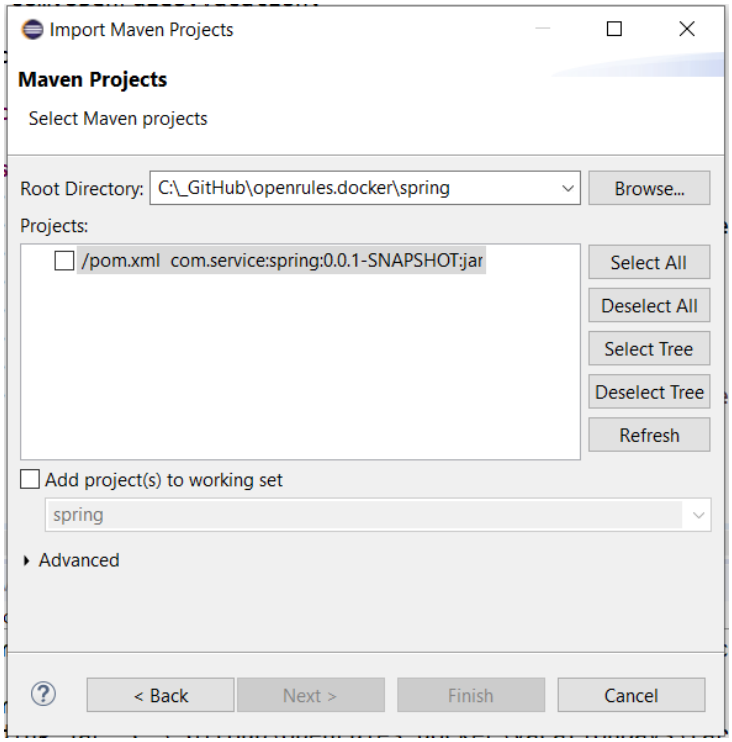
The simplest way to create a Spring Boot project is to use Spring Initializr. To bootstrap your new Spring Boot project, open https://start.spring.io/ and enter the following data:



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 “openrules.services”. In the Eclipse select “File + Import Project + Existing Maven Projects”:





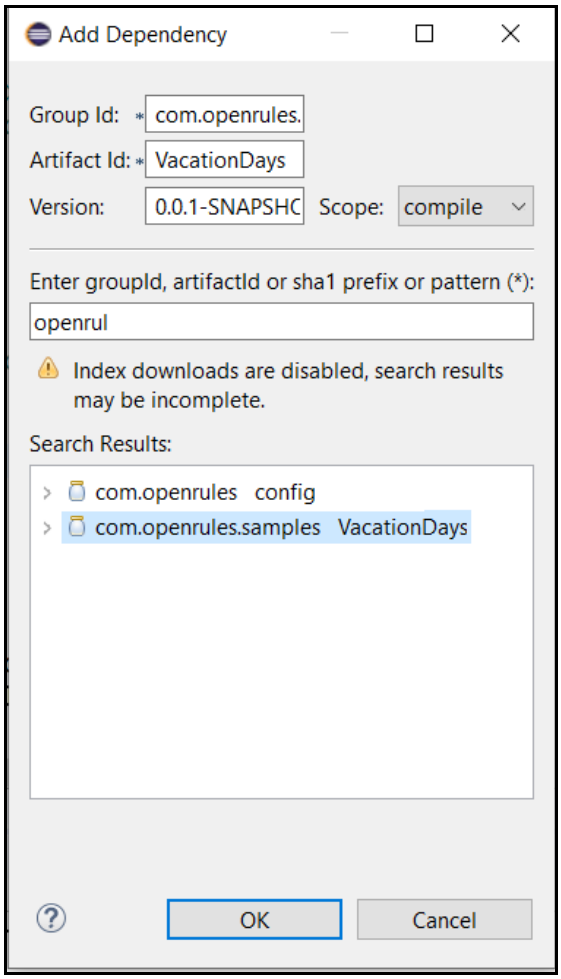
It will create a new project “spring”. This project already contains the file “Application.java”:

```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 VacationDaysService to this web application.

Adding Service to Spring Boot Application

To add different services to this application, we 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 "VacationDaysController" and it will include our service defined in the project "VacationDaysService". To make sure that the Spring Boot project "spring" is aware of our project "VacationDaysService", right-click on "spring/pom.xml" and select Maven + Add Dependency. It will open this dialog:



Start typing “openrul” in the box “Enter groupId,…” and it will show available results. Select “com.openrules.samples VacationDays” and Group ID, Artifact Id” and “Version” will be filled out automatically. After you click OK, Eclipse will add the following dependency to the file “spring/pom.xml”:

```
<dependency>
    <groupId>com.openrules.samples</groupId>
    <artifactId>VacationDays</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

Of course, you could add them manually as well.

Now we can create a new Java class “VacationDaysController” in the same package “com.service.spring” where SpringBoot placed the above class Application. Here is the initial version:

```
package com.service.spring;

import org.springframework.beans.factory.annotation.Autowired;

@RestController
public class VacationDaysController {

    @Autowired
    private VacationDaysService vacationDaysService;

    @RequestMapping( path="/vacationDays", method= {RequestMethod.POST} )
    public int calculateVacationDays(@RequestBody Employee employee) {
        System.out.println("/vacationDays for " + employee);
        return vacationDaysService.run(employee);
    }
}
```

Here we 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 corresponding imports.

To handle the incoming HTTP requests for our service, this controller should include a method that will accept an Employee object as a parameter and returns a calculated number of vacation days for this employee.

The method "calculateVacationDays" will be automatically called when our web application receives a POST request through the URL "*/vacationDays*" with a JSON object that has the same properties as the class Employee. To define this functionality, we used Sprint Boot annotations @RequestMapping and @RequestBody:

@RequestMapping(path="/vacationDays", method={RequestMethod.POST})
public int calculateVacationDays(**@RequestBody** Employee employee) {...}

We are not done yet. To "autowire" this service, we need to inform Spring how to create an instance of the class VacationDaysService. It's usually done within a special class annotated of the type @Configuration. So, we need to create a new class "ServiceFactory" annotated it with **@Configuration**. For each service this class should include a method annotated with **@Bean** that creates and returns an instance of the service. In our case it will be the method "newVacationDaysService" as described below:

```
package com.service.spring;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.openrules.vacation.VacationDaysService;

@Configuration
public class ServiceFactory {
    
    @Bean
    public VacationDaysService newVacationDaysService() {
        return new VacationDaysService();
    }

}
```

This completes the development of our Spring Boot application with one service that will be called using URL "*/vacationDays*".

To test our web application, right-click on "Application.java" 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 requests. Here is the protocol from the Eclipse's Console view:

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

2019-06-23 08:14:49.064  INFO 30844 --- [           main] com.service.spring.Application
2019-06-23 08:14:49.069  INFO 30844 --- [           main] com.service.spring.Application
2019-06-23 08:14:50.365  INFO 30844 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
```

…

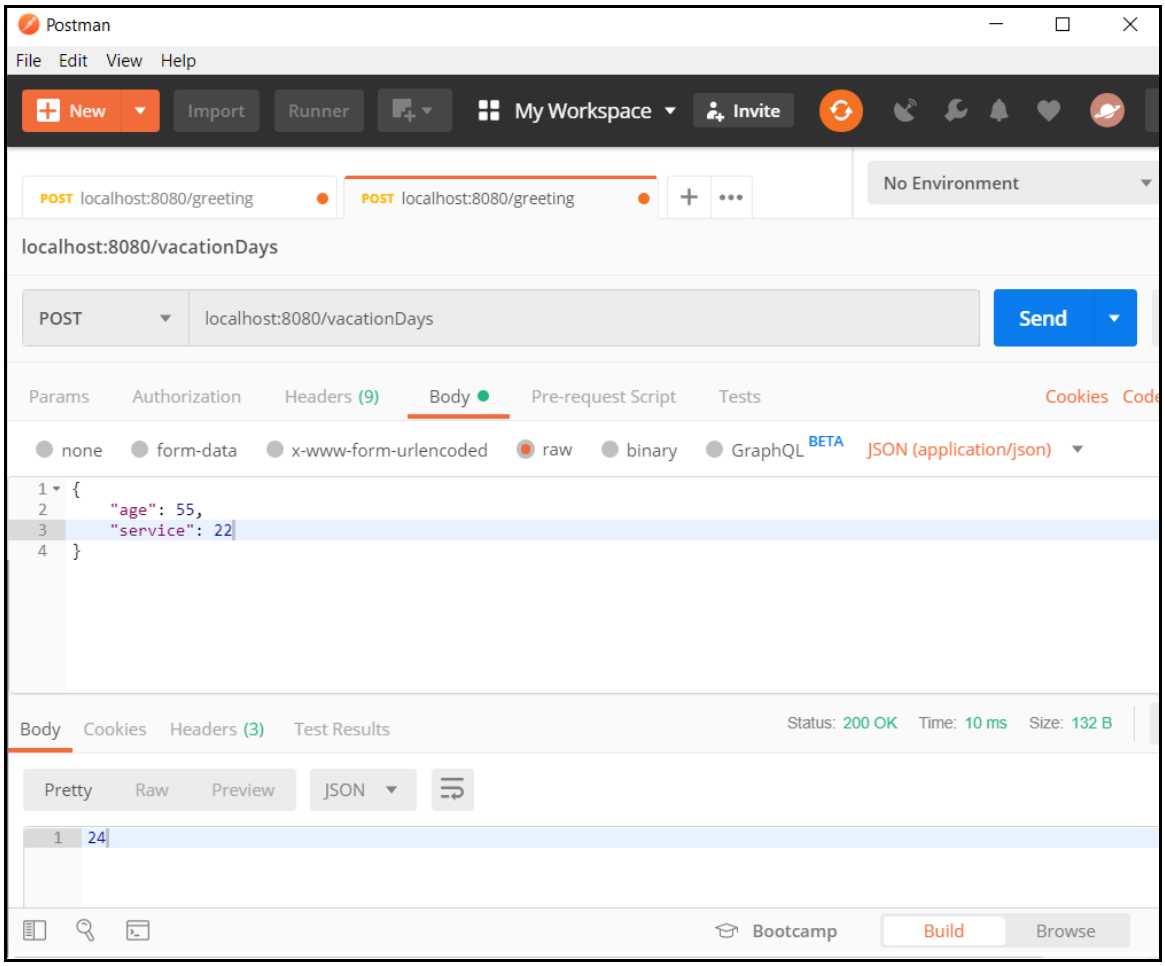
At the end of the protocol in the right bottom corner you should see that Spring started our service:

```
: Starting service [Tomcat]
: Starting Servlet engine: [Apache Tomcat/9.0.17]
: Initializing Spring embedded WebApplicationContext
: Root WebApplicationContext: initialization completed in 1372 m
: INITIALIZE OPENRULES ENGINE 7.0.1 (build 06112018) for [classp
: INCLUDE=DecisionModel.xls
: [DecisionModel.xls] has been resolved to [classpath:/vacationD
: Processing classpath:/vacationDays/DecisionModel.xls
: INCLUDE=Glossary.xls
: [Glossary.xls] has been resolved to [classpath:/vacationDays/G
: Processing classpath:/vacationDays/Glossary.xls
: INCLUDE=Rules.xls
: [Rules.xls] has been resolved to [classpath:/vacationDays/Rule
: Processing classpath:/vacationDays/Rules.xls
: INCLUDE=../templates/DecisionTemplates.xls
: [../templates/DecisionTemplates.xls] has been resolved to [cla
: Processing classpath:/templates/DecisionTemplates.xls
: INCLUDE=DecisionTable${OPENRULES_MODE}Templates.xls
: [DecisionTable${OPENRULES_MODE}Templates.xls] has been resolve
: Processing classpath:/templates/DecisionTableExecuteTemplates.
: *** Decision Vacation Days ***
: Decision Vacation Days has been initialized
: Initializing ExecutorService 'applicationTaskExecutor'
: Tomcat started on port(s): 8080 (http) with context path ''
: Started Application in 3.247 seconds (JVM running for 3.567)
```

Now our web application is running and waiting for HTTP requests.

TESTING DECISION SERVICE WITH POSTMAN

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:



In this form, we selected the method “POST” (from the drop-down list), typed the URL “localhost:8080/vacationDays”, enter a simple JSON structure

```
{
    “age”: 55
    “service”: 22
}
```

After, a click on “Send”, the POSTMAN sent the proper HTTP request to our web application, that executed our VacationDaysService and returned the calculated number of vacation days “24” at the bottom of the form. We may enter different combinations of “age” and “service” to make sure that our OpenRules-based VacationDaysService works as expected.

TESTING DECISION SERVICE WITH A JAVA CLIENT

Now we may test our running service from any Java program similar to what we did with POSTMAN. Spring has already prepared for us the place for all Java tests: the package "com.service.spring" in the folder "src/test/java". Let's add a new Java class "VacationDaysClient" to this package. It may look as below:

```
package com.service.spring;

import java.io.BufferedReader;

public class VacationDaysClient {

    public static void main(String[] args) throws Exception {
        URL url = new URL("http://localhost:8080/vacationDays");

        String json = "{"
                + "\"age\":55,"
                + "\"service\":22"
                + "}";

        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type", "application/json;utf-8");
        connection.setDoOutput(true);
        try(OutputStream os = connection.getOutputStream()) {
            os.write(json.getBytes("utf-8"));
        }
        try(InputStream is = connection.getInputStream()) {
            BufferedReader reader = new BufferedReader(new InputStreamReader(is));
            String line;
            while( (line = reader.readLine()) != null ) {
                System.out.println(line);
            }
        }
    }
}
```

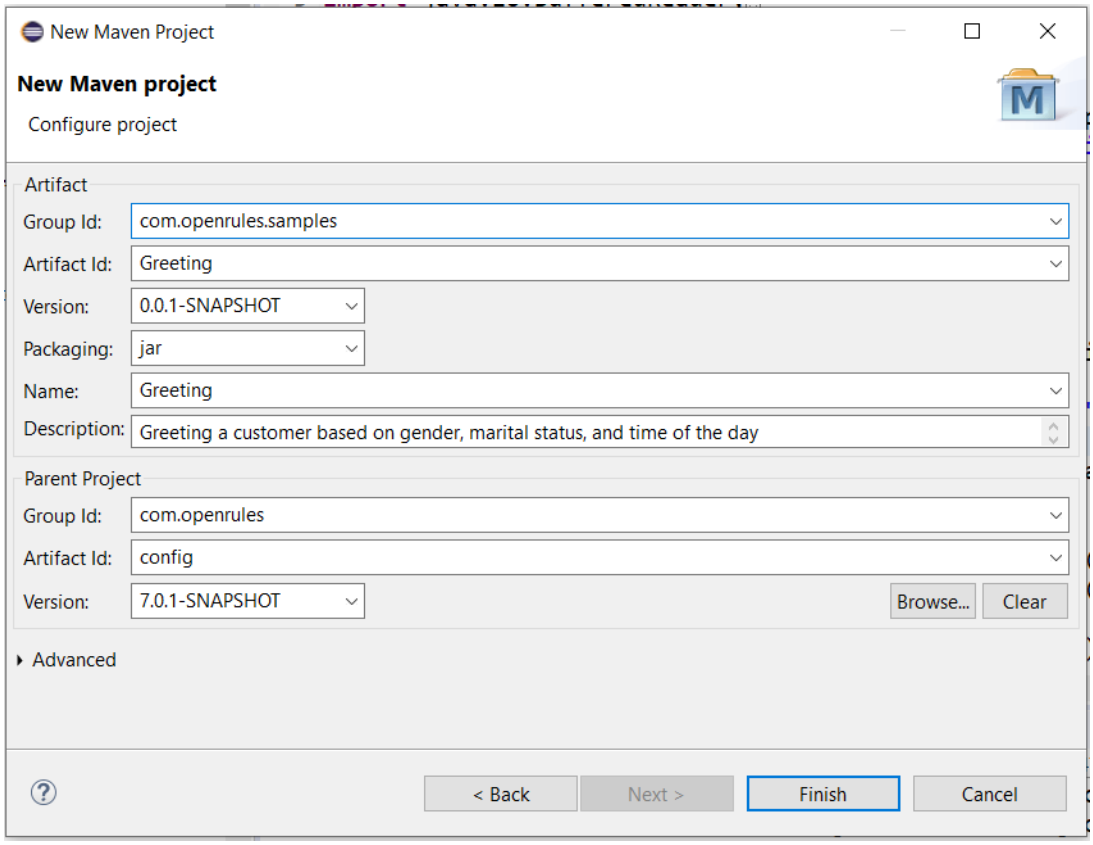
As you can see, we create a string with the same JSON data as we used in POSTMAN. Then we open a connection using the URL "http://localhost:8080/VacationDays". When we write our JSON data to the connection's output stream, it sends the proper HTTP request to our service. And then we simply read the produced results from the connection's input stream.

As our service is still up and waiting, we may simply right-click on the "VacationDaysClient.java" and select "Run As Java Application". After executing the request, it will display the same 24 days.

ADDING ANOTHER OPENRULES DECISION SERVICE

Similarly to the service “VacationDays”, we can move more services from “openrules.models” to our workspace “openrules.services”. Let’s start with the rules project Hello.

We will create a simple Maven project “Greeting”:



Defining Decision Model

We can find the proper decision model implemented in the standard OpenRules workspace “openrules.models” as a stand-alone project “Hello”. First, we create a new rules repository in the folder “**rules**” in our new Maven’s project “Greeting”. We may copy the subfolder “**templates**” from the VacationDays/rules/templates into “rules”. Then we will create a subfolder “greeting” inside “rules”. Then we will copy all Excel files from the folder “openrules.models/Hello/rules” to the subfolder “greeting”. Here are Excel files and tables that implement vacation days calculation logic.

File “rules/greeting/**Rules.xls**”:

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 + "!"

All decision goals, variables, and decision objects are defined in the file "rules/greeting/**Glossary.xls**":

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

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

The file "rules/greeting/**DecisionModel.xls**" contains the modified Environment table:

Environment	
include	Glossary.xls
	Rules.xls
	../templates/DecisionTemplates.xls

The folder “rules/greeting” also includes the file “**Test.xls**” that specifies Datatype “Customer” and creates several test-cases with expected results. To make sure that the new decision model still works, we copy file “build.bat” and “run.bat” from openrules.models/Hello/ to our new project “Greeting”, and make the following changes in them:

File “build.bat”:

```
set INPUT_FILE_NAME=rules/greeting/DecisionModel.xls
set DECISION_NAME="Hello Statement"
set OUTPUT_FILE_NAME=rules/greeting/Goals.xls
cd %~dp0
call ..\openrules.config\projectBuild
pause
```

File “run.bat”:

```
set DECISION_NAME="Hello Statement"
set FILE_NAME=rules/greeting/Test.xls
cd %~dp0
call ..\openrules.config\projectRun
pause
```

Double-click on “**build.bat**” and it will build an execution path for this model and will save it in the file “**Goals.xls**”. Then double-click on “**run.bat**” and it will execute all test-cases producing the expected results.

Adding a Java Interface

To execute the same model from Java, we will create 3 Java classes:

- **Customer**.java – to define test-employees
- **GreetingService**.java – to specify our service
- **Test**.java – to test the service locally.

So first, we create a new Java package “com.openrules.greeting” in the folder “src/main/java” and then add a new class Customer:

```
package com.openrules.greeting;

public class Customer {
»
»   String name;
»   String gender;
»   String maritalStatus;
»   int    currentHour;
```

Then we will use Eclipse to generate Getters and Setters and toString() methods for this class.

Then we add a new class "**GreetingService**":

```
package com.openrules.greeting;

import com.openrules.ruleengine.Decision;
import com.openrules.ruleengine.OpenRulesEngine;

public class GreetingService {

    OpenRulesEngine engine;

    public GreetingService() {

        String fileName = "classpath:/greeting/Goals.xls";
        engine = new OpenRulesEngine(fileName);
    }

    public String run(Customer customer) {
        String decisionName = "Hello Statement";
        Decision decision = new Decision(decisionName,engine);
        decision.put("FEEL", "On");
        decision.put("Customer", customer);
        decision.execute();
        return customer.getHelloStatement();
    }
}
```

To test this modified decision model, we will add here a new package "com.openrules.greeting" to "src/test/java/, and then we will add to this package a new class "**Test**.java":

```
public class Test {

public static void main(String[] args) {
        GreetingService service = new GreetingService();

        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 produce the results that look like below:

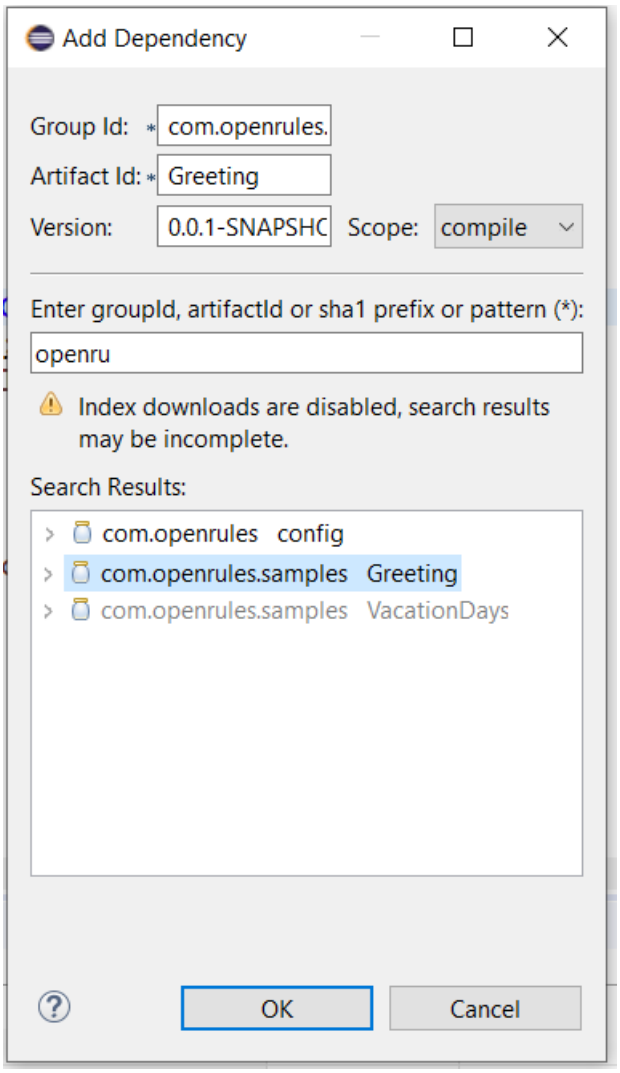
```
[INFO] Log - -AUTOMATICALLY DETERMINED EXECUTION PATH for Hello Statement:
[INFO] Log - -- DefineGreeting
[INFO] Log - -- DefineSalutation
[INFO] Log - -- DefineHelloStatement
[INFO] Log - -Execute DefineGreeting
[INFO] Log - -  Assign: Greeting = Good Evening
[INFO] Log - -Execute DefineSalutation
[INFO] Log - -  Conclusion: Salutation Is Mrs.
[INFO] Log - -Execute DefineHelloStatement
[INFO] Log - -  Conclusion: Hello Statement  Is Good Evening, Mrs. Robinson!
Good Evening, Mrs. Robinson!
```

We may consider that our decision service “Greeting” has been tested as a stand-alone application and is ready for further deployment. To complete Maven’s installation for this project, we should right-click on “Greeting/pom.xml” and select “Run As” + “Maven Install”.

Now it’s time to migrate this service to a REST web-based application using Spring Boot.

Adding Greeting Service to Spring Boot Application

We will continue to use the same Spring Boot project ”spring” that we created earlier.
The file “Application.java” remains without changes. Now we want to add a new “GreetingService” based on our project “Greeting”. To make sure that the Spring Boot project “spring” is aware of our project “Greeting”, right-click on “spring/pom.xml”, select Maven + Add Dependency, and fill out this dialog:



After you click OK, Eclipse will add the following dependency to the file “spring/pom.xml”:

```
<dependency>
    <groupId>com.openrules.samples</groupId>
    <artifactId>Greeting</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
dependencies>
```

Now we can create a new Java class “GreetingController” in the same package “com.service.spring” where we placed “VacationDaysController”. Here is this class:

```
package com.service.spring;

import org.springframework.beans.factory.annotation.Autowired;

@RestController
public class GreetingController {

    @Autowired
    private GreetingService greetingService;

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

The method “produceGreetingFor” will be automatically called when our web application receives a POST request through the URL “*/greeting*” with a JSON object that has the same properties as the class Customer.

To “autowire” this service, we need to add the method “newGreetingService” (the name is up to us) to the class “ServiceFactory”. This method will create and return an instance of GreetingService. Here is the modified class “ServiceFactory”:

```
package com.service.spring;

import org.springframework.context.annotation.Bean;

@Configuration
public class ServiceFactory {

    @Bean
    public VacationDaysService newVacationDaysService() {
        return new VacationDaysService();
    }

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

Now our Spring Boot application can handle two services: VacationDays and Greeting.

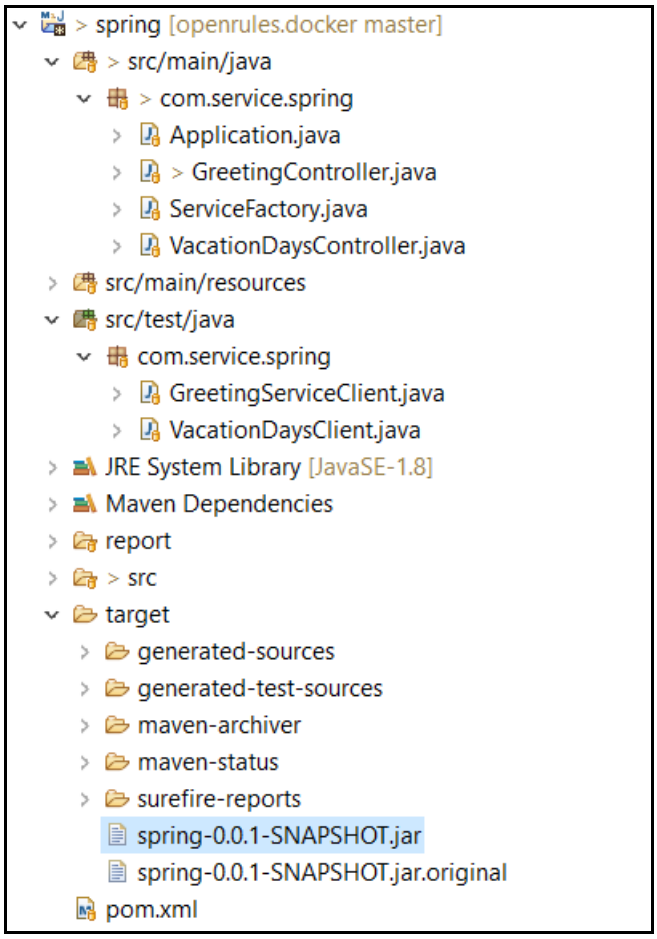
To test these services, we will start our application by right-clicking on “Application.java” and select “Run As Java Application” – make sure that you stopped previous applications which use the same port.

Now our web application is running and waiting for HTTP requests for services with URLs: “localhost:8080/vacationDays” and “localhost:8080/greeting”. If you run POSTMAN with these URLs and the proper JSON data, it will produce the expected results. We also may add a Java client in the class GrretingServiceClient in src/test/java/:

```
package com.service.spring;

import java.io.BufferedReader;

public class GreetingServiceClient {

    public static void main(String[] args) throws Exception {
        URL url = new URL("http://localhost:8080/greeting");

        String json = "{"
                + "\"name\":\"Robinson\","
                + "\"gender\":\"Female\","
                + "\"maritalStatus\":\"Married\","
                + "\"currentHour\":7"
                + "}";

        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type", "application/json;utf-8");
        connection.setDoOutput(true);
        try(OutputStream os = connection.getOutputStream()) {
            os.write(json.getBytes("utf-8"));
        }
        try(InputStream is = connection.getInputStream()) {
            BufferedReader reader = new BufferedReader(new InputStreamReader(is));
            String line;
            while( (line = reader.readLine()) != null ) {
                System.out.println(line);
            }
        }
    }
}
```

When you run this class as Java Application, it will produce: ”`Good Morning, Mrs. Robinson!`”.

To prepare our SpringBoot application for further deployment, we need to right-click on “spring/pom.xml” and select “Run As” + “Maven install”. It will install openrules.services\spring\target\spring-0.0.1-SNAPSHOT.jar to the Maven’s repository. We just need to check the “Maven install” will be completed with the message “BUILD SUCCESS”. Here is the latest structure of the project “spring”:



Please note that the jar-file openrules.services\spring\target\spring-0.0.1-SNAPSHOT.jar completely encapsulates everything we need to run our deployed decision services "VacationDays", "Greeting", and any other services we may add. And you don't need to run "Application.java" from Eclipse. Let's stop this application in Eclipse. Let's open a command line window in the folder "openrules.services\spring". Now we may enter the following command:

```
C:\_GitHub\openrules.services\spring>java -jar target\spring-0.0.1-SNAPSHOT.jar
```

It will start our REST web application, initialize both "Greeting" and "VacationDays" services, and wait for HTTP requests. Here is the start protocol:

```
Starting Application v0.0.1-SNAPSHOT on DESKTOP-AORGQJG with PID 15888 (C:\_GitHub\openrules.dock

No active profile set, falling back to default profiles: default
Tomcat initialized with port(s): 8080 (http)
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/9.0.17]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 1501 ms
INITIALIZE OPENRULES ENGINE 7.0.1 (build 06112018) for [classpath:/greeting/Goals.xls]
INCLUDE=DecisionModel.xls
[DecisionModel.xls] has been resolved to [classpath:/greeting/DecisionModel.xls]
Processing classpath:/greeting/DecisionModel.xls
INCLUDE=Rules.xls
[Rules.xls] has been resolved to [classpath:/greeting/Rules.xls]
Processing classpath:/greeting/Rules.xls
INCLUDE=Glossary.xls
[Glossary.xls] has been resolved to [classpath:/greeting/Glossary.xls]
Processing classpath:/greeting/Glossary.xls
INCLUDE=../templates/DecisionTemplates.xls
[../templates/DecisionTemplates.xls] has been resolved to [classpath:/templates/DecisionTemplates
Processing classpath:/templates/DecisionTemplates.xls
INCLUDE=DecisionTable${OPENRULES_MODE}Templates.xls
[DecisionTable${OPENRULES_MODE}Templates.xls] has been resolved to [classpath:/templates/Decision

Processing classpath:/templates/DecisionTableExecuteTemplates.xls
*** Decision Hello Statement ***
Decision Hello Statement has been initialized
INITIALIZE OPENRULES ENGINE 7.0.1 (build 06112018) for [classpath:/vacationDays/Goals.xls]
INCLUDE=DecisionModel.xls
[DecisionModel.xls] has been resolved to [classpath:/vacationDays/DecisionModel.xls]
Processing classpath:/vacationDays/DecisionModel.xls
INCLUDE=Glossary.xls
[Glossary.xls] has been resolved to [classpath:/vacationDays/Glossary.xls]
Processing classpath:/vacationDays/Glossary.xls
INCLUDE=Rules.xls
[Rules.xls] has been resolved to [classpath:/vacationDays/Rules.xls]
Processing classpath:/vacationDays/Rules.xls
INCLUDE=../templates/DecisionTemplates.xls
[../templates/DecisionTemplates.xls] has been resolved to [classpath:/templates/DecisionTemplates
Processing classpath:/templates/DecisionTemplates.xls
INCLUDE=DecisionTable${OPENRULES_MODE}Templates.xls
[DecisionTable${OPENRULES_MODE}Templates.xls] has been resolved to [classpath:/templates/Decision

Processing classpath:/templates/DecisionTableExecuteTemplates.xls
*** Decision Vacation Days ***
Decision Vacation Days has been initialized
Initializing ExecutorService 'applicationTaskExecutor'
Tomcat started on port(s): 8080 (http) with context path ''
Started Application in 3.978 seconds (JVM running for 4.341)
```

Now again we may send HTTP requests from POSTMAN or from our Java clients or from similar programs, and they will work as before. You actually may move spring-0.0.1-SNAPSHOT.jar to any other location and it will work as well. It's also ready to be uploaded to AWS or another cloud repository, and invoke our decision services remotely.

DEPLOYING DECISION SERVICE TO DOCKER

Now it's time to containerize our decision service using Docker. We are going to use the same port, so let's stop the running Application by clicking on the red rectangle in the Eclipse bar with "Console".

We need to add the following file "Dockerfile" to the folder "openrules.services/spring/":

```
FROM openjdk:8-jdk-alpine

COPY target/spring-0.0.1-SNAPSHOT.jar /OpenRulesSampleServices.jar

EXPOSE 8080

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/OpenRulesSampleServices.jar"]
```

This file will be used to build a Docker image from the command line. To do this, we will use a command line starting from the "openrules.services/spring/".

Enter the following command:

```
C:\_GitHub\openrules.services\spring>docker build -t openrules.samples .
```

It will build the Docker image of our Spring application and will call it "openrules.samples". Here is the execution protocol:

```
C:\_GitHub\openrules.docker\spring>docker build -t openrules.samples .
Sending build context to Docker daemon  36.27MB
Step 1/4 : FROM openjdk:8-jdk-alpine
 ---> a3562aa0b991
Step 2/4 : COPY target/spring-0.0.1-SNAPSHOT.jar /OpenRulesSampleServices.jar
 ---> 14ff13ca0584
Step 3/4 : EXPOSE 8080
 ---> Running in 1f8760dfaecf
Removing intermediate container 1f8760dfaecf
 ---> 272e9d2cad51
Step 4/4 : ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/OpenRulesSampleServices.jar"]
 ---> Running in c0ac57a7bc5e
Removing intermediate container c0ac57a7bc5e
 ---> e74b8177b640
Successfully built e74b8177b640
Successfully tagged openrules.samples:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and
ies added to build context will have '-rwxr-xr-x' permissions. It is recommended to double check and reset permi
or sensitive files and directories.
```

To run our application from the newly created container, we may enter the command:

```
C:\_GitHub\openrules.services\spring>docker run -d -p 8080:8080 openrules.samples
```

It will show the temporary name of the started docker's process, and our application is ready again to handle HTTP requests. We can do it either from POSTMAN or from VacationDaysClient and still will receive the same results as before.

Now you can use orchestration tools such as Kubernetes for of the Docker containers with OpenRules decision services. The created Docker images can be deployed to any cloud environment that supports Docker containers: AWS, Google Cloud, MS Azure, IBM Cloud, Rackspace, and many others.

CONCLUSION

In this tutorial we demonstrated how to migrate OpenRules decision models to Maven. Then we created a Sprint Boot REST application with several OpenRules-based decision models deployed as decision services. We had shown how to test this decision services using by sending HTTP requests with JSON data using the POSTMAN or Java-based clients. And finally, we containerized this Spring Boot web application using Docker.

TECHNICAL SUPPORT

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