Spring Boot Drools Rule Engine Example

Thameem Ansari · Follow

Published in Javarevisited · 6 min read · Oct 21, 2021

Rule engines can be used to implement complex business rules. The Drools is an open-source business rule management system that can be integrated easily with the spring boot application.

The KIE project supports the integration of the drools with other technologies like the spring boot framework.

In this article, we will learn how to integrate the Drools rule engine and implement a simple rule engine service using spring boot. We will use the DRL rules by creating the .drl file inside the spring boot application.

Technologies used in the article:

- Spring boot version: 2.5.5
- Drools
- Java version 1.8

Create a spring boot application

Create a spring boot application with the required dependencies. Add the spring boot starter web dependency. Also, include drools dependencies to the spring boot application's pom XML configuration file.

The below is the complete content of the pom.xml file.

pom.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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.5</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.demo.example</groupId>
    <artifactId>spring-boot-drools-example</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-boot-drools-example</name>
    <description>spring-boot-drools-example</description>
    <properties>
        <java.version>1.8</java.version>
        <maven-jar-plugin.version>3.1.1</maven-jar-plugin.version>
        <drools.version>7.59.0.Final</drools.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```xml
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.drools</groupId>
            <artifactId>drools-core</artifactId>
            <version>${drools.version}</version>
        </dependency>
        <dependency>
            <groupId>org.drools</groupId>
            <artifactId>drools-compiler</artifactId>
            <version>${drools.version}</version>
        </dependency>
        <dependency>
            <groupId>org.drools</groupId>
            <artifactId>drools-decisiontables</artifactId>
            <version>${drools.version}</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Configure the drools application

Create a configuration java class with the name **DroolsConfig.**

Add the below configuration to the java class.

DroolsConfig.java

```
package com.demo.example.config;

import org.kie.api.KieServices;
import org.kie.api.builder.KieBuilder;
import org.kie.api.builder.KieFileSystem;
import org.kie.api.builder.KieModule;
import org.kie.api.runtime.KieContainer;
import org.kie.internal.io.ResourceFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DroolsConfig {

    private static final String RULES_CUSTOMER_RULES_DRL =
"rules/customer-discount.drl";
    private static final KieServices kieServices =
KieServices.Factory.get();

    @Bean
    public KieContainer kieContainer() {
        KieFileSystem kieFileSystem = kieServices.newKieFileSystem();

kieFileSystem.write(ResourceFactory.newClassPathResource(RULES_CUSTOM
ER_RULES_DRL));
        KieBuilder kb = kieServices.newKieBuilder(kieFileSystem);
        kb.buildAll();
        KieModule kieModule = kb.getKieModule();
        KieContainer kieContainer =
kieServices.newKieContainer(kieModule.getReleaseId());
        return kieContainer;
    }
}
```

The configuration class defines a spring bean **KieContainer**. The KieContainer is used to build the rule engine by loading the rule files under the application's **/resources** folder.

We create the **KieFileSystem** instance and configure the rule engine and load the DRL file from the application's resources directory.

Also, we can use the **KieBuilder** instance to build the drools module. We can use the **KieSerive** singleton instance to create the KieBuilder instance.

Finally, we use KieService to create a KieContainer and configure it as a spring bean.

Add the model classes

Create a Pojo class with the name **OrderRequest** and define the below fields.

We receive this class as a request object to the rule engine and also we send these fields as the input to defined rules to calculate the discount amount for the given customer order.

OrderRequest.java

```
package com.demo.example.model;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class OrderRequest {

    private String customerNumber;
    private Integer age;
    private Integer amount;
    private CustomerType customerType;
}
```

Also, define a java enum with the name **CustomerType** as shown below. The enum holds the customer types and based on the value, the rule engine calculates the customer order discount percentage.

CustomerType.java

```
package com.demo.example.model;

public enum CustomerType {
    LOYAL, NEW, DISSATISFIED;

    public String getValue() {
        return this.toString();
    }
}
```

Finally, create a response POJO class with the name OrderDiscount as shown below.

OrderDiscount.java

```
package com.demo.example.model;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class OrderDiscount {

    private Integer discount = 0;
}
```

We will return the calculated discount using the above response object.

Define the drools rules

Create the drools rules inside a file with the name **customer-discount.drl**.

Add the file under the directory **/src/main/resources/rules**.

```
import com.demo.example.model.OrderRequest;
import com.demo.example.model.CustomerType;
global com.demo.example.model.OrderDiscount orderDiscount;

dialect "mvel"

rule "Age based discount"
    when
        OrderRequest(age < 20 || age > 50)
    then
        System.out.println("==========Adding 10% discount for Kids/
senior customer=============");
        orderDiscount.setDiscount(orderDiscount.getDiscount() + 10);
end

rule "Customer type based discount - Loyal customer"
    when
        OrderRequest(customerType.getValue == "LOYAL")
    then
        System.out.println("==========Adding 5% discount for LOYAL
customer=============");
        orderDiscount.setDiscount(orderDiscount.getDiscount() + 5);
end

rule "Customer type based discount - others"
    when
        OrderRequest(customerType.getValue != "LOYAL")
    then
```

```
        System.out.println("==========Adding 3% discount for NEW or
DISSATISFIED customer=============");
        orderDiscount.setDiscount(orderDiscount.getDiscount() + 3);
end

rule "Amount based discount"
    when
        OrderRequest(amount > 1000L)
    then
        System.out.println("==========Adding 5% discount for amount
more than 1000$=============");
        orderDiscount.setDiscount(orderDiscount.getDiscount() + 5);
end
```

We need to import the models that are being used in the DRL file.

We are also using a global parameter with the name **orderDiscount**. The global parameter can be shared between multiple rules.

The DRL file can contain one or multiple rules. We can use the mvel syntax to specify the rules. Also, each rule can be described with a description using the **rule** keyword.

We can use **when-then** syntax to define the conditions for a rule.

Based on the input values of the Order request, we are adding discount to the result. Every rule adds additional discount to the global result variable if the rule expression matches.

Add the service layer

Create a service java class with the name **OrderDiscountService**, and add the below content.

```java
package com.demo.example.service;

import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.demo.example.model.OrderDiscount;
import com.demo.example.model.OrderRequest;

@Service
public class OrderDiscountService {

    @Autowired
    private KieContainer kieContainer;

    public OrderDiscount getDiscount(OrderRequest orderRequest) {
        OrderDiscount orderDiscount = new OrderDiscount();
        KieSession kieSession = kieContainer.newKieSession();
        kieSession.setGlobal("orderDiscount", orderDiscount);
        kieSession.insert(orderRequest);
        kieSession.fireAllRules();
        kieSession.dispose();
        return orderDiscount;
    }
}
```

We are injecting the **KieContainer** instance and creating a **KieSession** instance.

We are also setting a global parameter of type **OrderDiscount**, that will hold the rules execution result.

To pass the request object to the DRL file, we can use the **insert()** method.

Then we fire all the rules by calling the **fireAllRules()** method. and finally terminate the session by calling the **dispose()** method of the KieSession.

Add the REST API

create a REST API class with the name **OrderDiscountController** and add the below content.

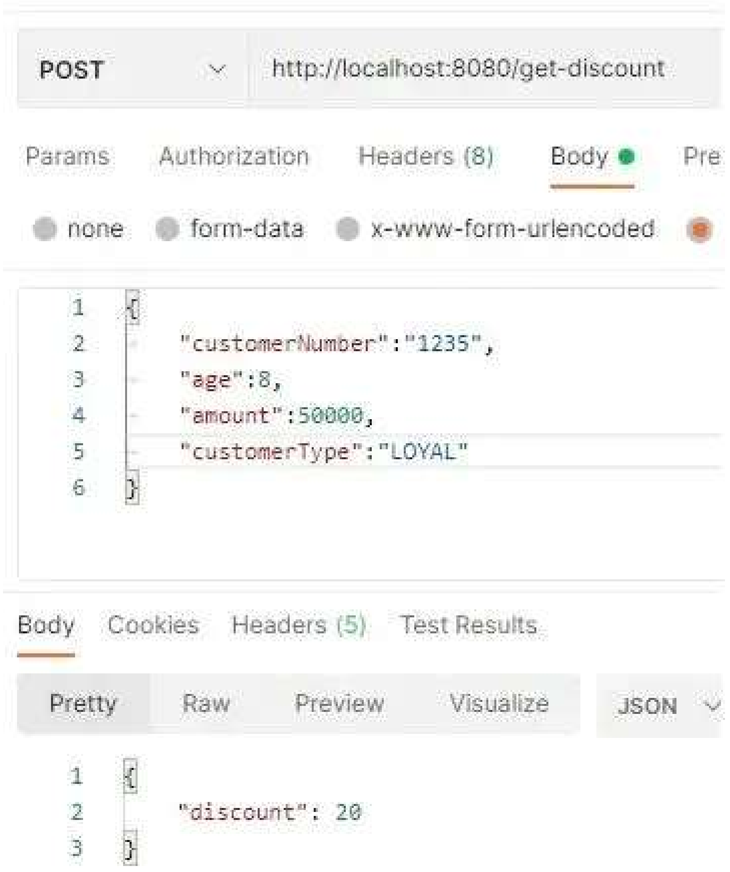
The controller exposes a POST API with the endpoint **/get-discount**. The endpoint expects an OrderRequest object and returns the calculated OrderDiscount response.

```
package com.demo.example.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.demo.example.model.OrderDiscount;
import com.demo.example.model.OrderRequest;
import com.demo.example.service.OrderDiscountService;

@RestController
public class OrderDiscountController {

    @Autowired
    private OrderDiscountService orderDiscountService;

    @PostMapping("/get-discount")
    public ResponseEntity<OrderDiscount> getDiscount(@RequestBody
OrderRequest orderRequest) {
        OrderDiscount discount =
orderDiscountService.getDiscount(orderRequest);
        return new ResponseEntity<>(discount, HttpStatus.OK);
    }
}
```

Testing the application

Run the spring boot application and access the REST API endpoint by sending the customer order request JSON.

For the LOYAL customer type with age < 20 and amount > 1000, we should get 20% discount according to our defined rules.

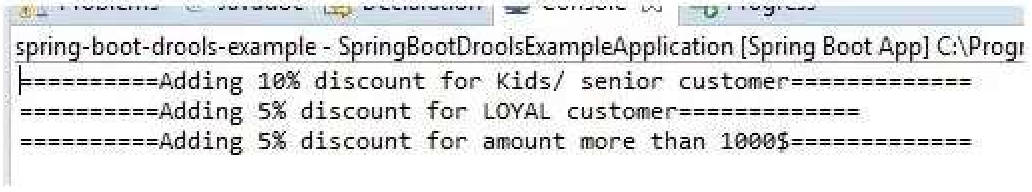


```
http://localhost:8080/get-discount
```

```
{
    "customerNumber":"1235",
    "age":8,
    "amount":50000,
    "customerType":"LOYAL"
}
```

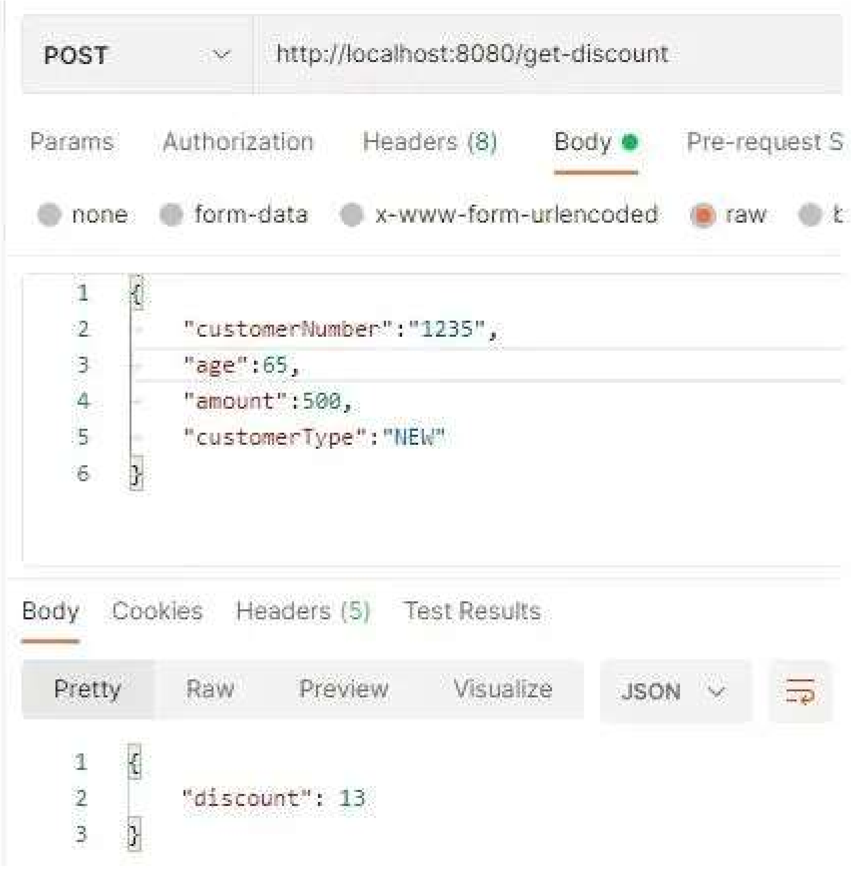
```
{
    "discount": 20
}
```

We can also observe the applied discounts for the input request.



```
spring-boot-drools-example - SpringBootDroolsExampleApplication [Spring Boot App] C:\Progr
==========Adding 10% discount for Kids/ senior customer============
==========Adding 5% discount for LOYAL customer============
==========Adding 5% discount for amount more than 1000$============
```

Now, try with the different values as shown below.

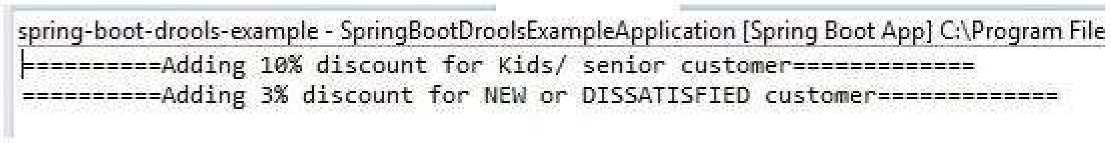


```
http://localhost:8080/get-discount
```

```
{
    "customerNumber":"1235",
    "age":65,
    "amount":500,
    "customerType":"NEW"
}
```

```
{
    "discount": 13
}
```

We can observe that only 2 rules are applied this time.



```
spring-boot-drools-example - SpringBootDroolsExampleApplication [Spring Boot App] C:\Program File
==========Adding 10% discount for Kids/ senior customer=============
==========Adding 3% discount for NEW or DISSATISFIED customer=============
```

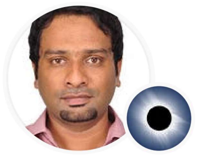
Conclusion

In this article, we learned how to create the drools rule engine using the spring boot framework.

We also learned how we can use the DRL files to define the business rules.

Happy coding..

Spring Boot



Written by Thameem Ansari

291 Followers · Writer for Javarevisited

Follow

Technology Expert| Coder| Sharing Experience| Spring | Java | Docker | K8s| DevOps|
https://reachansari.com

More from Thameem Ansari and Javarevisited

Thameem Ansari

How to avoid SSL validation in Spring Boot RestTemplate?

Spring Boot RestTemplate is an extremely versatile tool for making HTTP Requests....

2 min read · Oct 12, 2021

4

Semyon Kirekov in Javarevisited

Rich Domain Model with Hibernate

How to implement Rich Domain Model in Java? What pros and cons does it have in...

28 min read · Sep 11

349 8

50 Key Spring Boot Interview Questions for Java Developers
Ace your interview with these Interview question and answer
By - Ajay Rathod
twitter - ajtheory

Ajay Rathod in Javarevisited

50+ Key Spring Boot Interview Questions for Programmers and...

“Explore 50 Crucial Spring Boot Interview Questions for Excelling in Your Java Spring...

10 min read · Sep 15

203

Thameem Ansari in Javarevisited

Spring Boot 2 + JUnit 5 + Mockito

Introduction

5 min read · Nov 2, 2022

43 2

See all from Thameem Ansari

See all from Javarevisited

Recommended from Medium

Devalère T KAMGUIA

Master Data Processing With Spring Batch: Best Practices

Best practices that will empower you to create efficient and robust batch processing…

9 min read · Jun 5

20

Iftekhar Hossain

How to Use Java Records to Write Better and More Efficient Code

JAVA is one of the most popular and widely used programming languages. In many…

6 min read · Aug 20

105 1

Lists

Staff Picks

470 stories · 338 saves

Stories to Help You Level-Up at Work

19 stories · 240 saves

Self-Improvement 101

20 stories · 694 saves

Productivity 101

20 stories · 631 saves

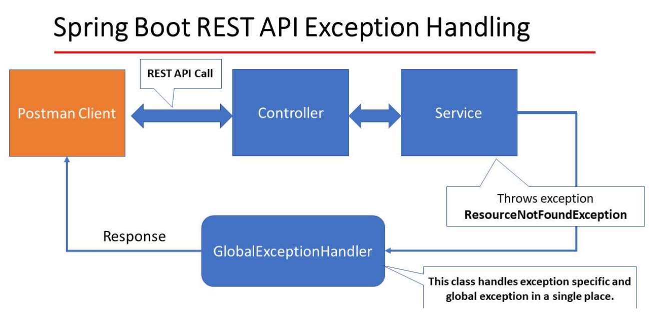
Adrian Chlebo... in the-stepstone-group-tech-bl...

Integration testing with Spring Boot 3, DynamoDB and LocalStack

Amazon DynamoDB is a popular NoSQL database. Additionally, it's still gaining...

6 min read · Jul 3

13



Spring Boot REST API Exception Handling
REST API Call
Postman Client
Controller
Service
Throws exception
ResourceNotFoundException
Response
GlobalExceptionHandler
This class handles exception specific and global exception in a single place.

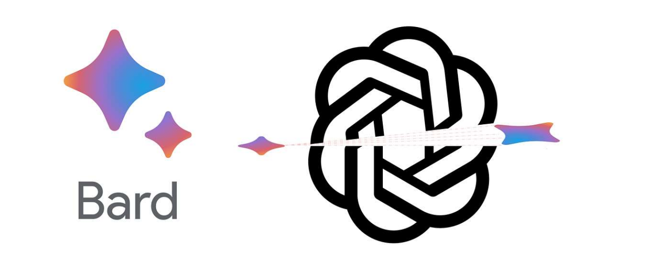
The Java Trail

Spring Boot: Exception Handling Best Practices

Use @ControllerAdvice for global exception handling

5 min read · Sep 3

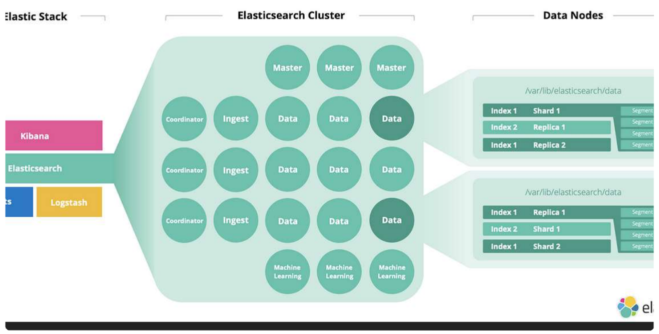
143 1



AL Anany

The ChatGPT Hype Is Over — Now Watch How Google Will Kill...

It never happens instantly. The business game is longer than you know.



sambsv

Advanced ELASTIC SEARCH Using SpringBoot Java ElasticSearch Ja...

Brief

· 6 min read · Sep 1

13K 394

5 min read · May 20

4

See more recommendations