Apache Camel with Spring Boot

javadevjournal.com/spring-boot/apache-camel-spring-boot

June 9, 2020

In this article, we will look at how to ***integrate Apache Camel with Spring Boot***.

1. Introduction

Apache Camel is an integration framework that aims to put different systems together to work robustly. In the enterprise systems, there is always work to connect different systems. Apache Camel provides a way for the developer to focus on their business logic without converting your data to a canonical format. Camel makes it by supporting over 80 API implementation of protocols and data types. So as a developer you only need to know how Camel glues everything together.In this article, we will look at the steps to integrate *Apache Camel with Spring Boot*.

Before showing an example with Spring Boot, it is good to know the core concepts and terminology of Camel.

1.1. Message

An entity used by the systems to communicate with each other.

1.2. Exchange

Exchange encapsulates a message and provides interaction between the systems. It is the container of the messages that determine the messaging type.

1.3. Camel Context

Camel Context is Camel's core model that provides access to services like Routes, Endpoints, etc.

1.4. Routes

An abstraction that allows clients and servers to work independently. We create routes with Domain-Specific Languages and they are a chain of function calls (processors).

1.5. Domain-Specific Language (DSL)

Processors, endpoints are wired together by writing them with DSL which at the end forms routes. In our case DSL is JAVA fluent API but if we use Camel with other languages/frameworks, then DSL could be XML etc. too.

1.6. Processor

Processors performs exchange operations. We can think of routes as a logical unit that connects correct processors to process a message.

1.7. Component

Components are the extension units of Apache Camel. They are the units that make Camel so easy to integrate with other systems. Have a look at core components for the full list of supported components.Components work as a factory of Endpoints by creating them with given URI.

1.8. Endpoint

Endpoints are the connection point of services that connects systems to other systems. We create endpoints via components with a given URI. For instance, to create a FTP connection provide following URI in a route: `<em>ftp://[username@]hostname[:port]/directoryname[?options]</em>` and Components create an endpoint of FTP with given configurations.

1.9. Producer

Producers are the units of Camel that create and send messages to an endpoint.

1.10. Consumer

Consumers are the units of Camel that receive messages created by producer, wrap them in exchange and send them to processors.

We summarized the major parts of Camel so far. It is unnecessary to get details of every concept but it is good to have an architectural overview of Camel which helps us to use it properly. In our following example, we will show how they are integrated into Spring Boot.

2. Application Overview

We will create an application that;

- Have products and discounts entities
- We insert products on startup
- Discounts that automatically applied to products with some periods (Camel Timer + Camel JPA)
- REST Endpoint to list all products and discounts (Camel REST)
- Swagger Documentation (Camel Swagger)

For this, we will use H2, Spring Web, Spring JPA, and Apache Camel.

3. Setting up Application

Create your maven project with the following dependencies. You can use your IDE or Spring Initializr to bootstrap your application.Here is the complete pom.xml with their explanations:

```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>
   <!--Get required dependencies from a parent-->
   <parent>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-dependencies</artifactId>
      <version>3.3.0</version>
   </parent>
   <artifactId>spring-boot-camel</artifactId>
   <name>spring-boot-camel</name>
   <description>Spring Boot Camel integration tutorial</description>
   <properties>
      <spring-boot-version>2.2.7.RELEASE</spring-boot-version>
      <run.profiles>dev</run.profiles>
   </properties>
   <dependencyManagement>
      <dependencies>
         <!--Import as a pom to let spring-boot to manage spring-boot dependencies
version -->
         <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>${spring-boot-version}</version>
            <type>pom</type>
            <scope>import</scope>
         </dependency>
         <!--Import as a pom to let camel manage camel-spring-boot dependencies
version-->
         <dependency>
            <groupId>org.apache.camel.springboot</groupId>
            <artifactId>camel-spring-boot-dependencies</artifactId>
            <version>${project.version}</version>
            <type>pom</type>
            <scope>import</scope>
         </dependency>
      </dependencies>
   </dependencyManagement>
   <dependencies>
      <!--Spring boot dependencies to enable REST, JPA and Core features-->
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-data-jpa</artifactId>
      </dependency>
      <!--Camel Spring Boot Dependencies to enable REST, JSON, SWAGGER, JPA
features-->
      <dependency>
         <groupId>org.apache.camel.springboot</groupId>
         <artifactId>camel-spring-boot-starter</artifactId>
      </dependency>
      <dependency>
         <groupId>org.apache.camel.springboot</groupId>
```

```
            <artifactId>camel-servlet-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.camel.springboot</groupId>
            <artifactId>camel-jackson-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.camel.springboot</groupId>
            <artifactId>camel-swagger-java-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.camel.springboot</groupId>
            <artifactId>camel-jpa-starter</artifactId>
        </dependency>
        <!--In memory database-->
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <!--Spring boot testing-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>${spring-boot-version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

4. Set Up Entities

Before dealing with Apache Camel, we need to set up some entities, services, and repositories.

4.1. Product

Create a Product Entity with id, name, price, and discounted fields. We are also creating a named query which can be called from Camel with its name and returns query result. Our discounted-products named query returns all products that have a discount on them.

```
@Entity
@Table(name = "products")
@NamedQuery(name = "discounted-products", query = "select product from Product
product where product.discounted IS NOT NULL")
public class Product {

 @Id
 @GeneratedValue
 private int id;

 private String name;

 private Integer price;

 private Integer discounted;

 // Getters and setters
}
```

Create a ProductRepository class which extends from `CrudRepository` of Spring Data. This extension provides us ready to call queries like findAll, findById, save, etc.

```
public interface ProductRepository extends CrudRepository<Product, Integer> {
}
```

4.2. Service Class

Create a `ProductService` class and annotate it with Service annotation. Use Constructor Injection to retrieve ProductRepository from the Spring Context. We provide basic `findById<code>, <code>findAll` , and save self-explanatory functions.

```
@Service
public class ProductService {

    private final ProductRepository products;

    @Autowired
    public ProductService(ProductRepository products) {
        this.products = products;
    }

    public Product findById(Integer id) {
        Optional < Product > product = products.findById(id);
        if (!product.isPresent()) {
            throw new IllegalStateException("Product could not found for given
id:" + id);
        }
        return product.get();
    }

    public Iterable < Product > findAll() {
        return products.findAll();
    }

    public void save(Product product) {
        products.save(product);
    }
}
```

As a last step, create a `data.sql` file in `src/main/resources` and insert 3 products as follows. Spring will run `data.sql` automatically on startup. Read more about init scripts

```
INSERT INTO products (id, name, price, discounted)
  VALUES
      (1, 'Book', 25, NULL),
      (2, 'Watch', 100, NULL),
      (3, 'Shoes', 40, NULL);
```

4.3. Discount

Create Discount Entity with id, amount, and product fields. One discount may happen in a given time on a product, so make an OneToOne relation on the product field.

```
@Entity
@Table(name = "discounts")
public class Discount {

    @Id
    @GeneratedValue
    private int id;

    private Integer amount;

    @OneToOne
    private Product product;

    // Getters and setters
}
```

Create DiscountRepository as we did.

```
public interface DiscountRepository extends CrudRepository<Discount, Integer> {}
```

Create DiscountService class, similar to ProductService. Besides `findDiscount` method which works like `findProduct`, we also have `makeDiscount` function. This function generates a random discount, gets the random product from the database, and applies the discount to that product.

```
@Service
public class DiscountService {

    private final DiscountRepository discounts;
    private final ProductService productService;

    private final Random random = new Random();

    @Autowired
    public DiscountService(DiscountRepository discounts,
        ProductService productService) {
        this.discounts = discounts;
        this.productService = productService;
    }

    public Discount makeDiscount() {
        // create a discount
        Discount discount = new Discount();
        int discountRate = this.random.nextInt(100);
        discount.setAmount(discountRate);

        // select random product
        int productId = this.random.nextInt(3) + 1;
        Product product = productService.findById(productId);

        // set the discount to product and save
        int discountedPrice = product.getPrice() - (discountRate *
product.getPrice() / 100);
        product.setDiscounted(discountedPrice);
        productService.save(product);

        discount.setProduct(product);
        return discount;
    }

    public Discount findDiscount(Integer id) {
        Optional < Discount > discount = discounts.findById(id);
        if (!discount.isPresent()) {
            throw new IllegalStateException("Discount could not found for given
id:" + id);
        }
        return discount.get();
    }
}
```

5. Application Configuration

Create `application-dev.yml` to configure `contextPath` mapping need for Camel. Add custom discount properties that will be used in our routes.

```
camel:
  component:
    servlet:
      mapping:
        contextPath: /javadevjournal/*

discount:
  newDiscountPeriod: 2000
  listDiscountPeriod: 6000/pre>
```

6. Apache Camel Integration

So far we configured our data before dealing with Apache Camel. Now let's use it.

6.1. Create Routes

Camel provides RouteBuilder as a base class to create routes. We need to extend it and annotate it with `@Component`. As we mentioned earlier Apache Camel uses its context to reference the objects. But when working with SpringBoot, Camel searches SpringBoot context first then injects found objects from there to its `CamelContext`, like `RouteBuilder` in our example.

After creating our Routes class extends from RouteBuilder, we need to override its configure method. We want to have a logic that autogenerates discounts with some given period. Let's add the following route to our configure function first and explain it:

```
@Component
class TimedJobs extends RouteBuilder {

@Override
public void configure() {
        from("timer:new-discount?delay=1000&period=
{{discount.newDiscountPeriod:2000}}")
            .routeId("make-discount")
            .bean("discountService", "makeDiscount")
            .to("jpa:org.apache.camel.example.spring.boot.rest.jpa.Discount")
            .log("Created %${body.amount} discount for ${body.product.name}");

        // additional route will be added in the next step
}
```

It is better to think about our Camel terminology here while using it with Spring Boot. We are creating **Routes** using **Java DSL**. Then we are using timer **Component,** which is an extension provided by Camel. Under the hood, Camel reaches timer **Endpoint** to start its **Producer** with our initial delay and run period configurations.

Before going to further usage, it is good to mention that A*pache Camel supports using Spring Boot properties* as we used here. You can directly refer to them with its name and a default value like `{{property_name:default_value}}.`

Then defining the make-discount route, which should be unique and can be referred later. Then we are calling our makeDiscount function in discountService bean. The **Message** is **Exchanged** which can be referred with body prefix and **Consumed** by the logger to log.

Refer to Simple Language for the full list of expressions you can use. Let's also add another route below to the previous one to list all the products with their updated prices.

```
from("jpa:org.apache.camel.example.spring.boot.rest.jpa.Product"
    + "?namedQuery=discounted-products"
    + "&delay={{discount.listDiscountPeriod:6000}}"
    + "&consumeDelete=false")
    .routeId("list-discounted-products")
    .log(
        "Discounted product ${body.name}. Price dropped from ${body.price} to
${body.discounted}");
```

We are using JPA Component for our Product entity and calling it `namedQuery`. We also configure our JPA with a delay so there could be some discounted created before we list products. `consumeDelete` query means we don't want to delete processed Product entity, check JPA Component for the full list of configurations. Here are the logs of our job:

```
Created %27 discount for Watch
Created %84 discount for Book
Created %92 discount for Shoes
Discounted product Book. Price dropped from 25 to 4
Discounted product Watch. Price dropped from 100 to 73
Discounted product Shoes. Price dropped from 40 to 4
```

6.2. Create REST Endpoints

So far we configured timer component to trigger our functions. Let's also integrate with REST endpoints and generate Swagger documentation. Create a new Route extending `RouteBuilder` ,we need to call Camel's `restConfiguration` function to configure our application.

```
@Component
class RestApi extends RouteBuilder {

@Override
public void configure() {
        restConfiguration()
            .contextPath("/javadevjournal")
            .apiContextPath("/api-doc")
            .apiProperty("api.title", "JAVA DEV JOURNAL REST API")
            .apiProperty("api.version", "1.0")
            .apiProperty("cors", "true")
            .apiContextRouteId("doc-api")
            .port(env.getProperty("server.port", "8080"))
            .bindingMode(RestBindingMode.json);

        rest("/products").description("Details of products")
            .get("/").description("List of all products")
            .route().routeId("products-api")
            .bean(ProductService.class, "findAll")
            .endRest()
            .get("discounts/{id}").description("Discount of a product")
            .route().routeId("discount-api")
            .bean(DiscountService.class, "findDiscount(${header.id})");
    }
}
```

We set our `contextPath` to javadevjournal and API context path to `api-doc` which is used for Swagger. Binding mode is off by default. Since we added json-jackson to our pom.xml, we can use json binding format. See here for the full list of configurations.In the second part of our configuration, we define `/products` endpoint and returning `Productservice`.findAll result. Also, we extend `/products` endpoint with /discounts/{id} and calling Discountservice.findDiscount function with retrieved id from the query. `{header}` refers to incoming input as mentioned in Simple Language previously for `{body}` placeholder.

If you visit `http://localhost:8080/javadevjournal/api-doc` you will get Swagger response.Hit `http://localhost:8080/javadevjournal/products` and you will get:

```
[
    {
        "id": 1,
        "name": "Book",
        "price": 25,
        "discounted": 4
    },
    {
        "id": 2,
        "name": "Watch",
        "price": 100,
        "discounted": 73
    },
    {
        "id": 3,
        "name": "Shoes",
        "price": 40,
        "discounted": 4
    }
]
```

Similarly visit `http://localhost:8080/javadevjournal/products/discounts/1` and you will get

```
{
    "id": 1,
    "amount": 92,
    "product": {
        "id": 3,
        "name": "Shoes",
        "price": 40,
        "discounted": 4
    }
}
```

Summary

In this article, we saw how to ***integrate Apache Camel with Spring Boot***.We briefly describe what is Apache Camel, how to integrate it with Spring Boot using real-life scenarios. The source code of this application is available on Github.

Advertisements