Spring Boot Microservices REST API Example

MICROSERVICES SPRING BOOT SPRING CLOUD

In this tutorial, we will create two Spring Boot microservices that communicate with each other via REST APIs. This guide is intended for beginners and includes detailed explanations for each step.

Introduction

In a microservices architecture, an application is composed of multiple loosely coupled services that communicate with each other. Each service is responsible for a specific piece of functionality and can be developed, deployed, and scaled independently. REST APIs are a common way for microservices to communicate.

Prerequisites

- JDK 17 or later
- Maven or Gradle
- IDE (IntelliJ IDEA, Eclipse, etc.)

Step 1: Create the Projects

We'll create two Spring Boot projects: `product-service` and `order-service`.

Step 2: Set Up `product-service`

2.1 Create the Project

Use Spring Initializr to create a new project with the following dependencies:

- Spring Web
- Spring Boot Actuator

2.2 Configure `application.properties`

Set up the application properties for `product-service`.

```
server.port=8081
spring.application.name=product-service
```

Explanation:

- `server.port=8081`: Sets the port for the Product Service.
- `spring.application.name=product-service`: Names the application.

2.3 Create a Product Model

Create a simple Product model to represent the product data.

```
package com.example.productservice;

public class Product {
    private String id;
    private String name;
    private double price;

    // Constructor, getters, and setters
    public Product(String id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}
```

Explanation:

- This class represents the product data with attributes `id`, `name`, and `price`.

2.4 Create a Controller

Create a controller to handle product-related requests.

```
package com.example.productservice;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ProductController {

    @GetMapping("/products/{id}")
    public Product getProduct(@PathVariable String id) {
        // In a real application, this data would come from a database
```

```
        return new Product(id, "Sample Product", 99.99);
    }
}
```

Explanation:

- `@RestController`: Marks this class as a REST controller.
- `@GetMapping("/products/{id}")`: Maps GET requests to `/products/{id}` to this method.
- `@PathVariable String id`: Extracts the `id` from the URL.

Step 3: Set Up `order-service`

3.1 Create the Project

Use Spring Initializr to create a new project with the following dependencies:

- Spring Web
- Spring Boot Actuator
- OpenFeign (for inter-service communication)

3.2 Configure `application.properties`

Set up the application properties for `order-service`.

```
server.port=8082
spring.application.name=order-service

# URL of the product service
product.service.url=http://localhost:8081
```

Explanation:

- `server.port=8082`: Sets the port for the Order Service.
- `spring.application.name=order-service`: Names the application.
- `product.service.url=http://localhost:8081`: Specifies the URL of the Product Service.

3.3 Enable Feign Clients

Enable Feign clients by adding the `@EnableFeignClients` annotation in the main application class.

```
package com.example.orderservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication
@EnableFeignClients
public class OrderServiceApplication {
```

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

Explanation:

- `@EnableFeignClients`: Enables Feign client functionality in your Spring Boot application.

3.4 Create a Product Client

Create a Feign client interface to communicate with the `product-service`.

```
package com.example.orderservice;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@FeignClient(name = "product-service", url = "${product.service.url}")
public interface ProductServiceClient {

    @GetMapping("/products/{id}")
    Product getProductById(@PathVariable String id);
}
```

Explanation:

- `@FeignClient(name = "product-service", url = "${product.service.url}")`: Declares this interface as a Feign client for the `product-service`.
- `@GetMapping("/products/{id}")`: Maps the GET request to the `/products/{id}` endpoint in `product-service`.

3.5 Create a Product Model

Create a Product model to match the one in `product-service`.

```
package com.example.orderservice;

public class Product {
    private String id;
    private String name;
    private double price;

    // Constructor, getters, and setters
    public Product() {}

    public Product(String id, String name, double price) {
        this.id = id;
```

```
        this.name = name;
        this.price = price;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}
```

Explanation:

- This class represents the product data with attributes `id`, `name`, and `price`.

3.6 Create a Controller

Create a controller to handle order-related requests and use the `ProductServiceClient` to fetch product details.

```
package com.example.orderservice;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    private final ProductServiceClient productServiceClient;

    public OrderController(ProductServiceClient productServiceClient) {
        this.productServiceClient = productServiceClient;
    }

    @GetMapping("/orders/{productId}")
    public String createOrder(@PathVariable String productId) {
        Product product = productServiceClient.getProductById(productId);
        return "Order created for product: " + product.getName() + " with price: $" + product.getPrice();
```

```
    }
}
```

Explanation:

- `@RestController`: Marks this class as a REST controller.
- `@GetMapping("/orders/{productId}")`: Maps GET requests to `/orders/{productId}` to this method.
- `ProductServiceClient productServiceClient`: Injects the Feign client for communication with `product-service`.

Step 4: Run the Microservices

1. **Start `product-service`**: Run the `ProductServiceApplication` class.
2. **Start `order-service`**: Run the `OrderServiceApplication` class.

Step 5: Test the Communication

Open your browser or use a tool like Postman to test the endpoints:

- **`product-service`**: `http://localhost:8081/products/1`
- **`order-service`**: `http://localhost:8082/orders/1`

The response from `order-service` should include the product details fetched from `product-service`.

Conclusion

You have successfully set up two Spring Boot microservices and enabled communication between them using REST APIs and Feign clients. This setup allows you to build scalable and maintainable microservices architecture. This example can be expanded to include more microservices and more complex inter-service communication patterns.