Spring Boot Microservices with RabbitMQ Example

MICROSERVICES SPRING BOOT SPRING CLOUD

In this tutorial, we will create two Spring Boot microservices that communicate with each other using RabbitMQ. We'll use RabbitMQ as a message broker to enable asynchronous communication between the microservices. This guide is intended for beginners and includes detailed explanations for each step.

Introduction to RabbitMQ

RabbitMQ is an open-source message broker software that implements the Advanced Message Queuing Protocol (AMQP). It allows applications to communicate with each other using messages, enabling asynchronous and decoupled communication. RabbitMQ is often used to build scalable and fault-tolerant distributed systems.

Prerequisites

- JDK 17 or later
- Maven or Gradle
- IDE (IntelliJ IDEA, Eclipse, etc.)
- RabbitMQ server (You can run RabbitMQ using Docker)

Step 1: Set Up RabbitMQ Server

You can run RabbitMQ using Docker with the following command:

```
docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

This command will start RabbitMQ with the management UI accessible at `http://localhost:15672` (default username and password: guest/guest).

Step 2: Create the Projects

We'll create two Spring Boot projects: `producer-service` and `consumer-service`.

Step 3: Set Up `producer-service`

3.1 Create the Project

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

- Spring Web
- Spring Boot Actuator
- Spring for RabbitMQ

3.2 Configure `application.properties`

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

```
spring.application.name=producer-service
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

Explanation:

- `spring.application.name=producer-service`: Names the application.
- `spring.rabbitmq.host=localhost`: Specifies the RabbitMQ server host.
- `spring.rabbitmq.port=5672`: Specifies the RabbitMQ server port.
- `spring.rabbitmq.username=guest`: Specifies the RabbitMQ username.
- `spring.rabbitmq.password=guest`: Specifies the RabbitMQ password.

3.3 Configure RabbitMQ

Create a configuration class to define the RabbitMQ components.

```
package com.example.producerservice;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMQConfig {

    public static final String QUEUE_NAME = "exampleQueue";

    @Bean
    public Queue exampleQueue() {
        return new Queue(QUEUE_NAME, false);
    }
}
```

Explanation:

- `@Configuration`: Indicates that this class contains Spring configuration.
- `@Bean`: Marks this method as a bean producer.
- `Queue exampleQueue()`: Creates a new queue named `exampleQueue`.

3.4 Create a Message Producer

Create a service to send messages to the queue.

```
package com.example.producerservice;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Service;

@Service
public class MessageProducer {

    private final RabbitTemplate rabbitTemplate;

    public MessageProducer(RabbitTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }

    public void sendMessage(String message) {
        rabbitTemplate.convertAndSend(RabbitMQConfig.QUEUE_NAME, message);
    }
}
```

Explanation:

- `@Service`: Marks this class as a service component.
- `RabbitTemplate rabbitTemplate`: Injects the `RabbitTemplate` for sending messages to RabbitMQ.
- `sendMessage(String message)`: Sends a message to the `exampleQueue`.

3.5 Create a Controller

Create a controller to handle HTTP requests and send messages to the queue.

```
package com.example.producerservice;

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

@RestController
public class MessageController {

    private final MessageProducer messageProducer;

    public MessageController(MessageProducer messageProducer) {
        this.messageProducer = messageProducer;
    }

    @GetMapping("/send")
    public String sendMessage(@RequestParam String message) {
        messageProducer.sendMessage(message);
```

```
        return "Message sent: " + message;
    }
}
```

Explanation:

- `@RestController`: Marks this class as a REST controller.
- `@GetMapping("/send")`: Maps GET requests to `/send` to this method.
- `@RequestParam String message`: Extracts the `message` parameter from the request.
- `messageProducer.sendMessage(message)`: Sends the message to the queue.

Step 4: Set Up `consumer-service`

4.1 Create the Project

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

- Spring Web
- Spring Boot Actuator
- Spring for RabbitMQ

4.2 Configure `application.properties`

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

```
spring.application.name=consumer-service
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

Explanation:

- `spring.application.name=consumer-service`: Names the application.
- `spring.rabbitmq.host=localhost`: Specifies the RabbitMQ server host.
- `spring.rabbitmq.port=5672`: Specifies the RabbitMQ server port.
- `spring.rabbitmq.username=guest`: Specifies the RabbitMQ username.
- `spring.rabbitmq.password=guest`: Specifies the RabbitMQ password.

4.3 Configure RabbitMQ

Create a configuration class to define the RabbitMQ components.

```
package com.example.consumerservice;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```java
@Configuration
public class RabbitMQConfig {

    public static final String QUEUE_NAME = "exampleQueue";

    @Bean
    public Queue exampleQueue() {
        return new Queue(QUEUE_NAME, false);
    }

    @RabbitListener(queues = QUEUE_NAME)
    public void listen(String message) {
        System.out.println("Received message: " + message);
    }
}
```

Explanation:

- `@Configuration`: Indicates that this class contains Spring configuration.
- `@Bean`: Marks this method as a bean producer.
- `Queue exampleQueue()`: Creates a new queue named `exampleQueue`.
- `@RabbitListener(queues = QUEUE_NAME)`: Annotates a method to listen to messages from the `exampleQueue`.
- `listen(String message)`: Processes the received message and prints it to the console.

Step 5: Run the Microservices

1. **Start RabbitMQ**: Ensure RabbitMQ is running using the Docker command mentioned above.
2. **Start `producer-service`**: Run the `ProducerServiceApplication` class.
3. **Start `consumer-service`**: Run the `ConsumerServiceApplication` class.

Step 6: Test the Communication

1. Open your browser or use a tool like Postman to send a GET request to `producer-service`:
    - URL: `http://localhost:8081/send?message=Hello`
    - This will send the message "Hello" to the queue.
2. Check the console logs of `consumer-service` to see the received message:
    - You should see `Received message: Hello` in the logs.

Conclusion

You have successfully set up two Spring Boot microservices that communicate asynchronously using RabbitMQ. This setup allows you to decouple the services and enable asynchronous processing, which can improve the scalability and resilience of your system. This example can be expanded to include more complex message handling, additional microservices, and advanced RabbitMQ configurations.