Tutorial

Ebanca Stefan Andrei

Using RabbitMQ as a Message Broker in Spring Boot

This is a tutorial on integrating RabbitMQ, a robust message broker, into a Spring Boot application. RabbitMQ facilitates communication between different components of an application by acting as a middleman for message exchange. In this tutorial, we'll explore how to configure RabbitMQ, create producers and consumers, and integrate them into a Spring Boot application to enable seamless communication between different parts of the system.

What is RabbitMQ?

RabbitMQ is a powerful open-source message broker that implements the Advanced Message Queuing Protocol (AMQP). It provides a reliable messaging solution for applications that need to communicate asynchronously and supports various messaging patterns such as publish/subscribe, point-to-point, and routing.

Why Use RabbitMQ with Spring Boot?

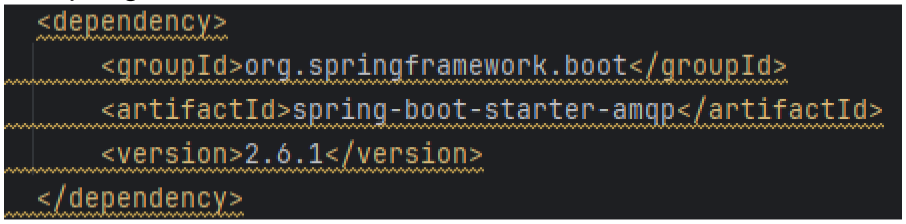
Integrating RabbitMQ with Spring Boot offers several advantages:

1. **Asynchronous Communication:** RabbitMQ enables decoupled communication between different parts of the application, allowing them to operate independently and asynchronously.
2. **Reliability:** RabbitMQ ensures reliable message delivery by providing features such as message acknowledgment, persistence, and delivery guarantees.
3. **Scalability:** RabbitMQ can handle large volumes of messages and scale horizontally to accommodate increased message traffic.
4. **Flexibility:** RabbitMQ supports various messaging patterns and can be integrated seamlessly with Spring Boot applications using Spring AMQP.

Step 1: Setting up RabbitMQ Configuration

Dependency Setup:

We need to make sure that we have the right dependencies in the pom.xml for RabbitMQ and Spring AMQP.

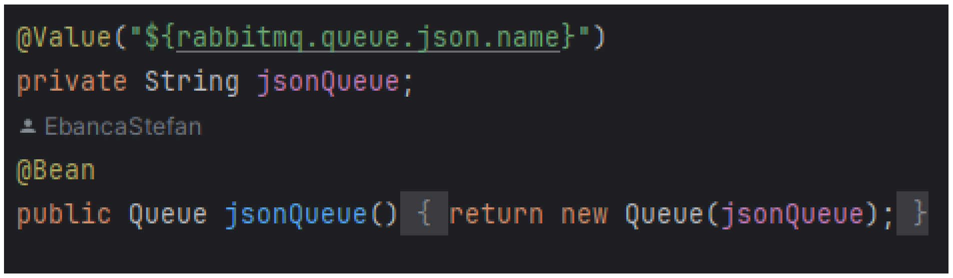


```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
    <version>2.6.1</version>
</dependency>
```

Configuration Class:

A configuration Class is needed for RabbitMQ in order to declare queues, exchanges, routing keys, bindings and message converters.

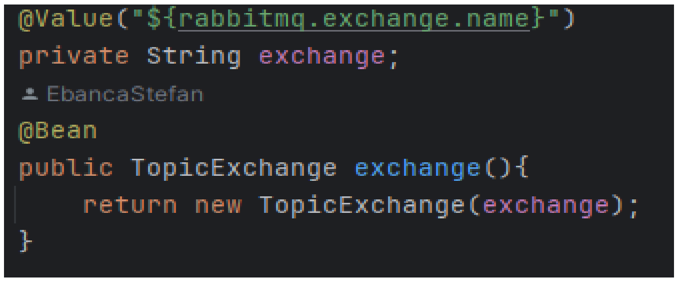
Queues are buffers that store messages. They are the primary mechanism through which messages are passed in RabbitMQ.



```java
@Value("${rabbitmq.queue.json.name}")
private String jsonQueue;
 EbancaStefan
@Bean
public Queue jsonQueue() { return new Queue(jsonQueue); }
```

This queue will store JSON messages that are sent by producers and consumed by consumers.

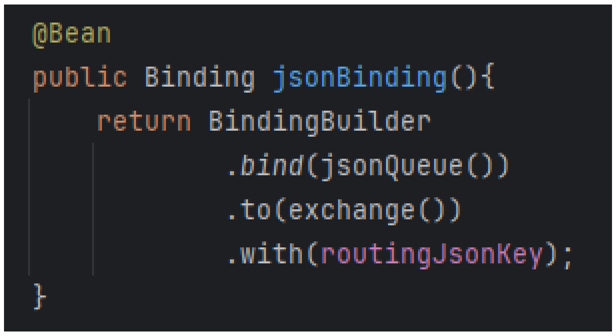
Exchanges receive messages from producers and route them to queues based on rules defined by bindings.



```java
@Value("${rabbitmq.exchange.name}")
private String exchange;
 EbancaStefan
@Bean
public TopicExchange exchange(){
    return new TopicExchange(exchange);
}
```

This code defines a topic exchange. Topic exchanges route messages to one or more queues based on matching between a message routing key and the routing pattern specified in the binding.

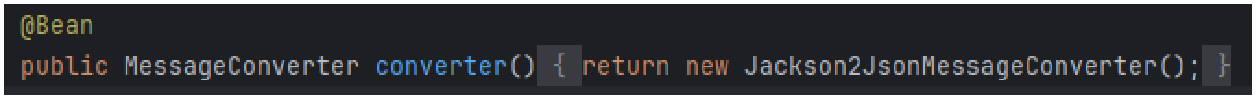
Bindings link queues to exchanges and specify the routing key used for message routing.



```java
@Bean
public Binding jsonBinding(){
    return BindingBuilder
            .bind(jsonQueue())
            .to(exchange())
            .with(routingJsonKey);
}
```

This code creates a binding between the **jsonQueue** and the **exchange**. It specifies that messages with a routing key matching **routingJsonKey** should be routed to the **jsonQueue**.

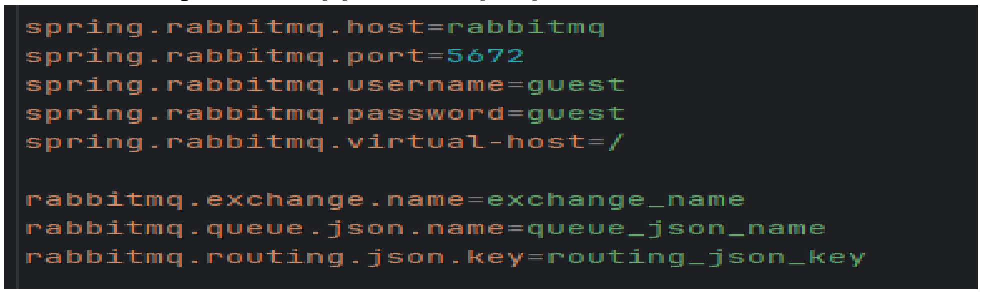
Message converters are used to convert between Java objects and message payloads when sending and receiving messages.



```java
@Bean
public MessageConverter converter() { return new Jackson2JsonMessageConverter(); }
```

This code configures a **Jackson2JsonMessageConverter** as the message converter. This converter is capable of converting Java objects to JSON format and vice versa, allowing the application to send and receive JSON messages.

We also need to configure the **application.properties** file like so.



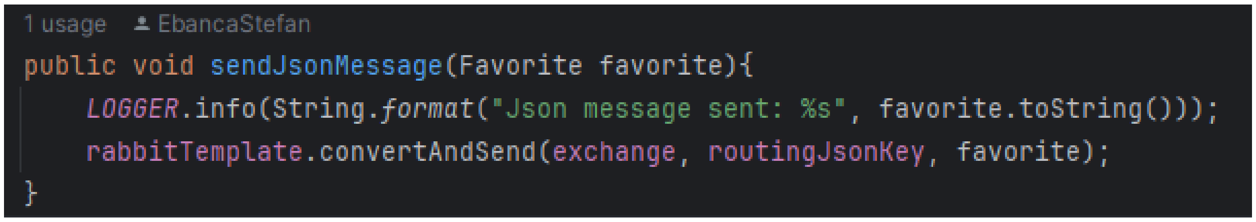
```
spring.rabbitmq.host=rabbitmq
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
spring.rabbitmq.virtual-host=/

rabbitmq.exchange.name=exchange_name
rabbitmq.queue.json.name=queue_json_name
rabbitmq.routing.json.key=routing_json_key
```

These properties specify the RabbitMQ host, port, credentials, virtual host, exchange name, queue name, and routing key used in the Spring Boot application.

Step 2: Creating RabbitMQ Producers

The **RabbitMQJsonProducer** class is responsible for sending JSON messages to RabbitMQ. It utilizes the **RabbitTemplate** provided by Spring AMQP to interact with RabbitMQ.

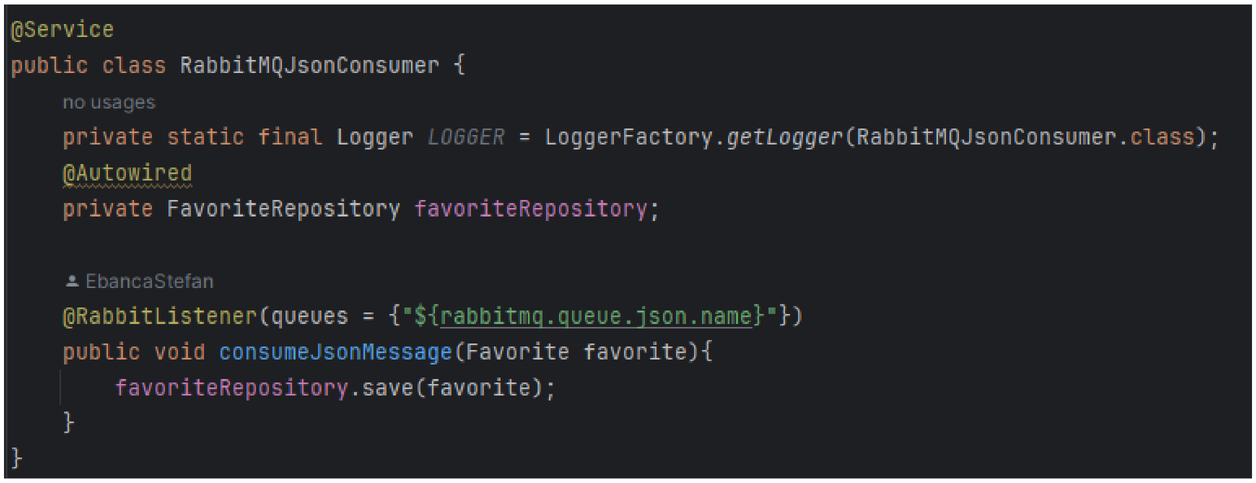


```
1 usage   EbancaStefan
public void sendJsonMessage(Favorite favorite){
    LOGGER.info(String.format("Json message sent: %s", favorite.toString()));
    rabbitTemplate.convertAndSend(exchange, routingJsonKey, favorite);
}
```

The **sendJsonMessage** method sends a JSON message to RabbitMQ using the **RabbitTemplate**.

Step 3: Creating RabbitMQ Consumers

The **RabbitMQJsonConsumer** class is responsible for consuming JSON messages from RabbitMQ. It utilizes the **@RabbitListener** annotation provided by Spring AMQP to listen for messages on specific queues.



```
@Service
public class RabbitMQJsonConsumer {
    no usages
    private static final Logger LOGGER = LoggerFactory.getLogger(RabbitMQJsonConsumer.class);
    @Autowired
    private FavoriteRepository favoriteRepository;

    EbancaStefan
    @RabbitListener(queues = {"${rabbitmq.queue.json.name}"})
    public void consumeJsonMessage(Favorite favorite){
        favoriteRepository.save(favorite);
    }
}
```

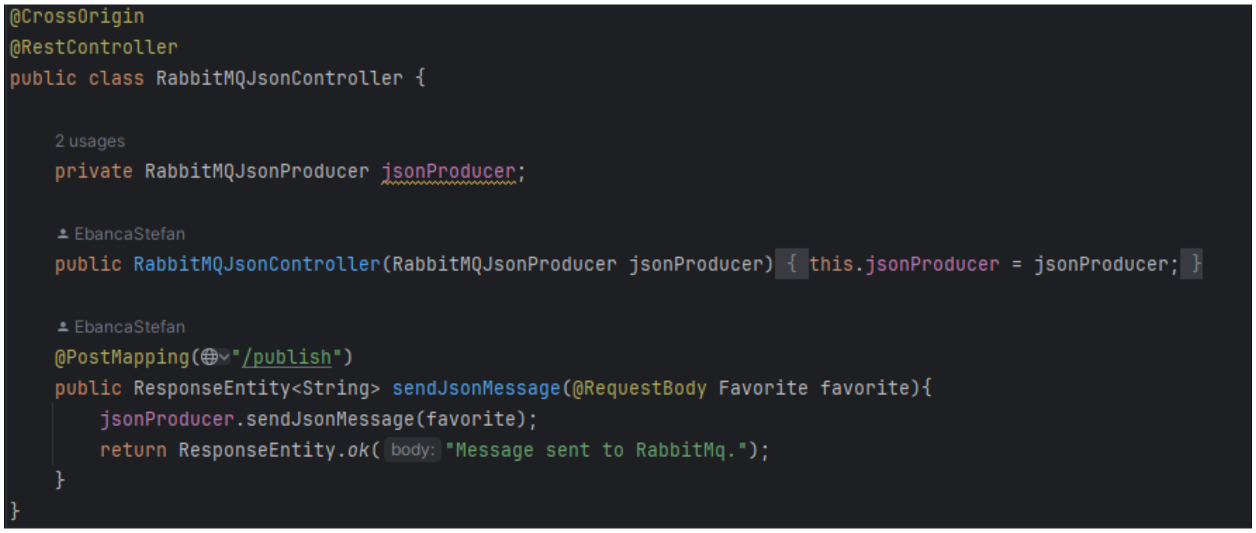
The **RabbitMQJsonConsumer** class is injected with a **FavoriteRepository** bean by Spring, allowing it to save **Favorite** objects to the database.

When a message is received on the RabbitMQ queue specified by **${rabbitmq.queue.json.name},** the **consumeJsonMessage** method is invoked. Inside this method, the received **Favorite** object is saved to the database using the injected **FavoriteRepository**.

The **RabbitMQJsonConsumer** class is a background service that listens for messages asynchronously. It doesn't need to be directly invoked from controllers.

Step 4: Integrating with Spring Boot Controllers

The **RabbitMQJsonController** class is responsible for handling HTTP requests from clients and invoking the RabbitMQ producer to send JSON messages to RabbitMQ.



```java
@CrossOrigin
@RestController
public class RabbitMQJsonController {

    2 usages
    private RabbitMQJsonProducer jsonProducer;

    EbancaStefan
    public RabbitMQJsonController(RabbitMQJsonProducer jsonProducer) { this.jsonProducer = jsonProducer; }

    EbancaStefan
    @PostMapping("/publish")
    public ResponseEntity<String> sendJsonMessage(@RequestBody Favorite favorite){
        jsonProducer.sendJsonMessage(favorite);
        return ResponseEntity.ok( body: "Message sent to RabbitMq.");
    }
}
```

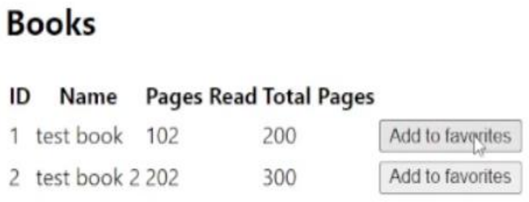
The **RabbitMQJsonController** class is injected with a **RabbitMQJsonProducer** bean by Spring, allowing it to send JSON messages to RabbitMQ.

When an HTTP POST request is sent to the **/publish** endpoint with a JSON payload representing a **Favorite** object, the **sendJsonMessage** method is invoked. Inside this method, the injected **RabbitMQJsonProducer** is used to send the **Favorite** object as a JSON message to RabbitMQ.

Frontend applications can send HTTP POST requests to the **/publish** endpoint of your Spring Boot application with a JSON payload representing a **Favorite** object. This will trigger the **sendJsonMessage** method in the controller, resulting in the JSON message being sent to RabbitMQ.

Example:

This is a quick example of how the RabbitMQ message is sent and consumed using a frontend React application.

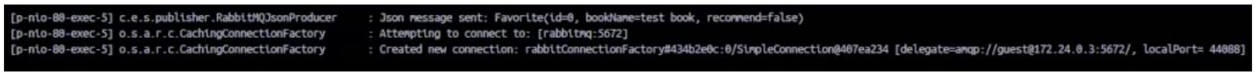


Books

ID	Name	Pages Read	Total Pages	
1	test book	102	200	Add to favorites
2	test book 2	202	300	Add to favorites

The webapp contains a table populated by books and each book has a button which sends a JSON message to the /publish endpoint. This message is being consumed by the RabbitMQ consumer class and creates a Favorite object.

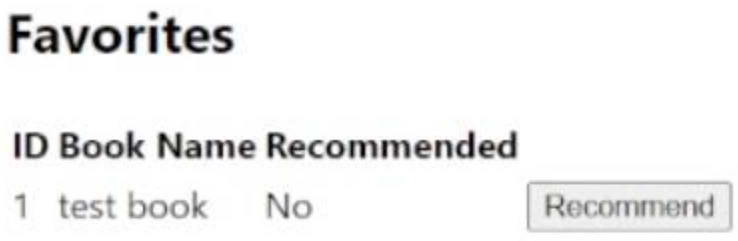
This is the response we get in the logs when pressing the button:



```
[p-nio-80-exec-5] c.e.s.publisher.RabbitMQJsonProducer      : Json message sent: Favorite(id=0, bookName=test book, recommend=false)
[p-nio-80-exec-5] o.s.a.r.c.CachingConnectionFactory        : Attempting to connect to: [rabbitmq:5672]
[p-nio-80-exec-5] o.s.a.r.c.CachingConnectionFactory        : Created new connection: rabbitConnectionFactory#434b2e0c:0/SimpleConnection@407ea234 [delegate=amqp://guest@172.24.0.3:5672/, localPort= 44088]
```

As you can see, the JSON message sent contains the name of the book associated to the button which was clicked along with the recommed field which is set to false.

And this is the newly created Favorite object in its own table:



Favorites

ID	Book Name	Recommended	
1	test book	No	Recommend