



UNIVERSIDAD PRIVADA ANTENOR ORREGO

INGENIERÍA DE COMPUTACIÓN Y SISTEMAS
FACULTAD DE INGENIERÍA

ARQUITECTURA DE SOFTWARE

TEMA

**Implementar el uso de Message
brokers**

DOCENTE

MENDOZA CORPUS, CARLOS ALFREDO;

INTEGRANTES

- ALFARO ASUNCIÓN JUAN CARLOS
- JACOBO ROMERO WENDY MITCHEL
- MIÑANO SICCHA BRAYAN DANIEL

TRUJILLO _ 2024

Message Brokers

INTRODUCCIÓN

En la arquitectura de microservicios, la comunicación eficiente entre los servicios es crucial para el rendimiento y la escalabilidad del sistema. Un "intermediario de mensajes" (message broker) desempeña un papel fundamental al facilitar esta comunicación. RabbitMQ es uno de los brokers de mensajes más populares y ampliamente utilizados, conocido por su confiabilidad y flexibilidad.

¿QUÉ ES RABBITMQ?

RabbitMQ es un intermediario de mensajes de código abierto que implementa el protocolo Advanced Message Queuing Protocol (AMQP). Actúa como un middleware que maneja el envío y recepción de mensajes entre los productores (servicios que envían mensajes) y los consumidores (servicios que reciben mensajes). Esto permite que los microservicios se comuniquen de manera asíncrona y desacoplada, mejorando la modularidad y la resiliencia del sistema.

VENTAJAS DE USAR RABBITMQ EN MICROSERVICIOS

- ✓ **Desacoplamiento:** Permite que los servicios se comuniquen sin conocer la existencia del otro, reduciendo las dependencias directas.
- ✓ **Escalabilidad:** Facilita la escalabilidad horizontal de los servicios al distribuir la carga de trabajo mediante colas.
- ✓ **Fiabilidad:** Garantiza la entrega de mensajes a través de mecanismos de confirmación y persistencia.
- ✓ **Flexibilidad:** Soporta múltiples patrones de mensajería como punto a punto, publicación/suscripción y enrutamiento basado en temas.
- ✓ **Resiliencia:** Mejora la tolerancia a fallos mediante la reintención y el almacenamiento persistente de mensajes en caso de errores.

CASO DE USO: Gestión de Productos y Comentarios

Para ilustrar el uso de RabbitMQ en una arquitectura de microservicios, consideremos un negocio que gestiona productos y comentarios. Este sistema consta de dos microservicios principales: `ms_product` y `ms_review`. La problemática es actualizar los detalles de los productos en tiempo real con el número de comentarios y el promedio de evaluación cada vez que se crea un nuevo comentario.

En este escenario:

- ✓ **ms_review** es responsable de manejar los comentarios y producir mensajes cuando se crea un nuevo comentario.
- ✓ **ms_product** es responsable de consumir estos mensajes y actualizar la información del producto en consecuencia.

Flujo de Trabajo con RabbitMQ

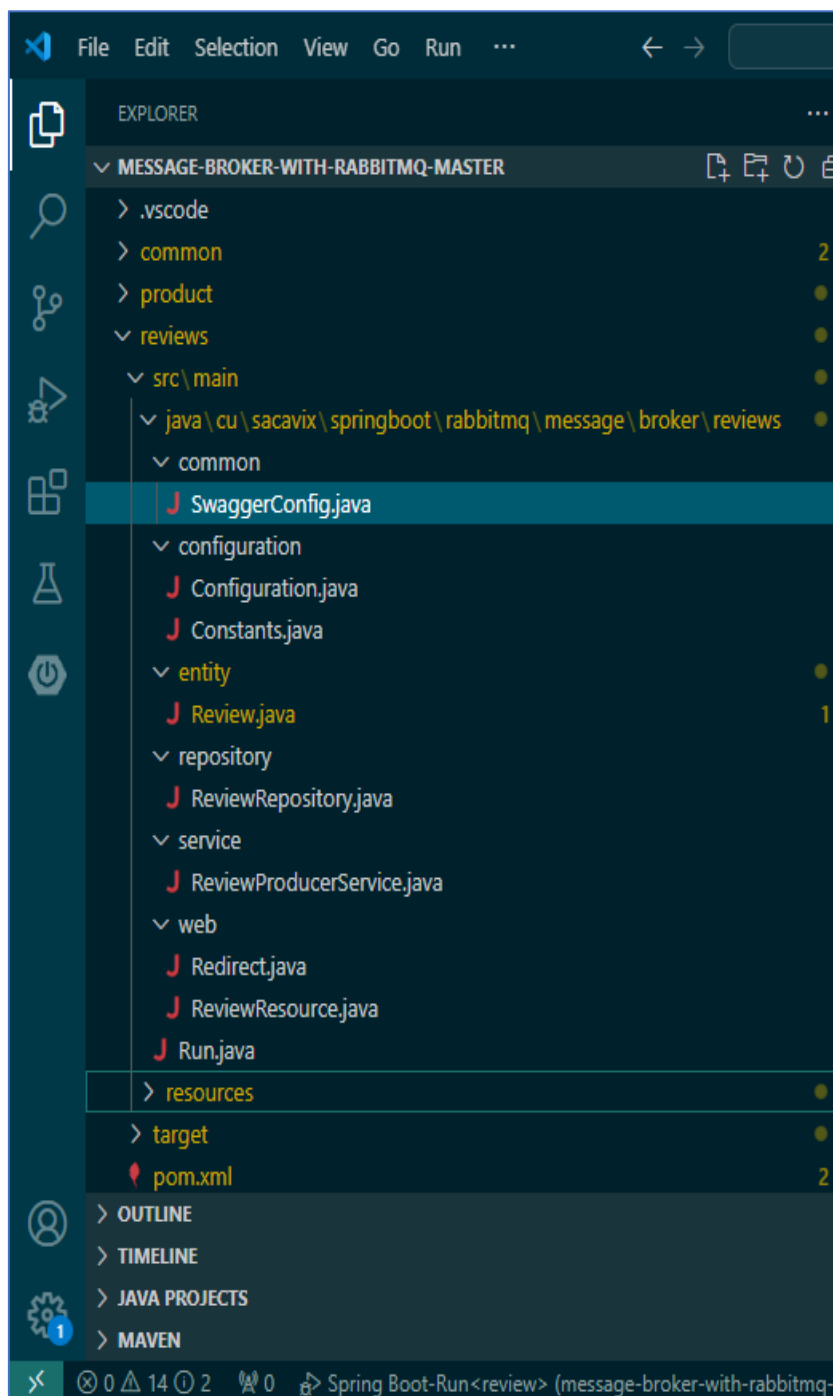
- ✓ **Creación de Comentarios:** Cuando un usuario crea un comentario en `ms_review`, se guarda en la base de datos y se envía un mensaje a RabbitMQ con los detalles del comentario (identificador del producto, número de comentarios y promedio de evaluación).

- ✓ **Procesamiento del Mensaje:** RabbitMQ encola el mensaje en la cola correspondiente (REVIEW_CREATED_QUEUE).
- ✓ **Actualización del Producto:** ms_product consume el mensaje desde la cola, procesa la información y actualiza los detalles del producto en su base de datos.

Ambos servicios estén desacoplados, ya que ms_review y ms_product no se comunican directamente entre sí. En su lugar, utilizan RabbitMQ como intermediario, lo que facilita la escalabilidad y el mantenimiento del sistema.

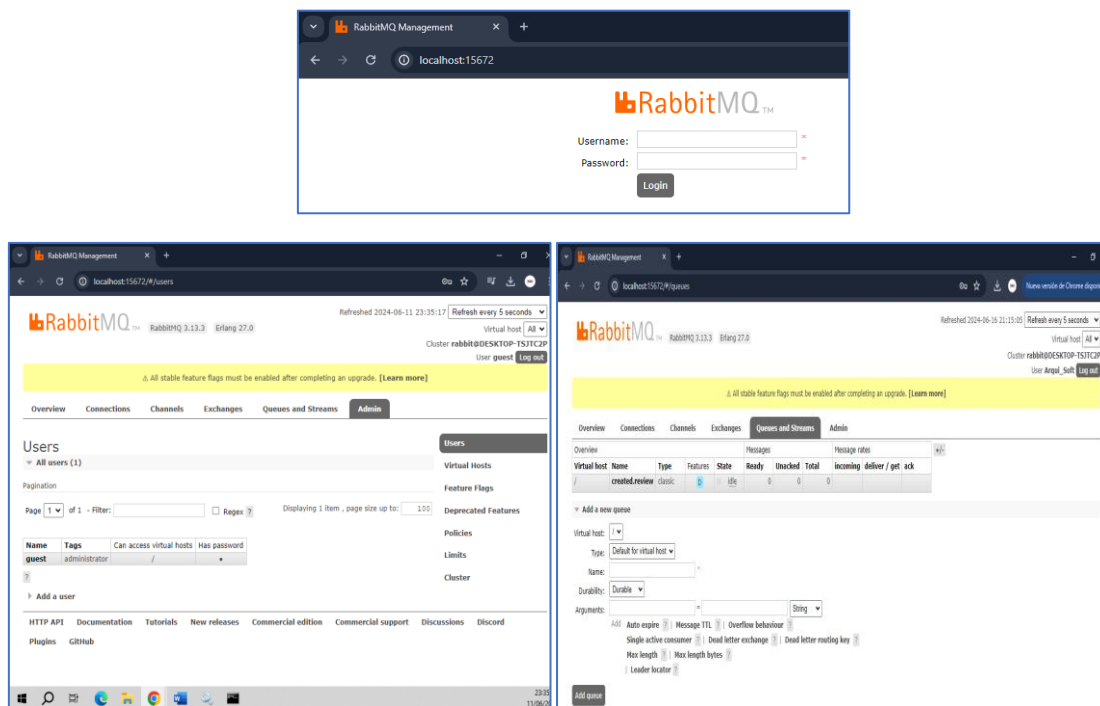
ESTRUCTURA DEL PROYETO

El proyecto se implementó en Visual Studio Code teniendo la siguiente estructura.



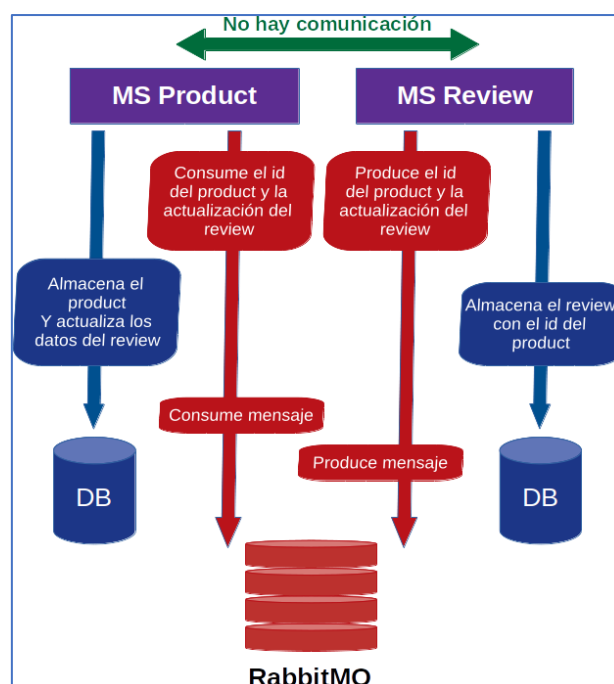
RabbitMQ

A continuación, se presentan los pasos esenciales para implementar esta solución utilizando RabbitMQ.



El negocio

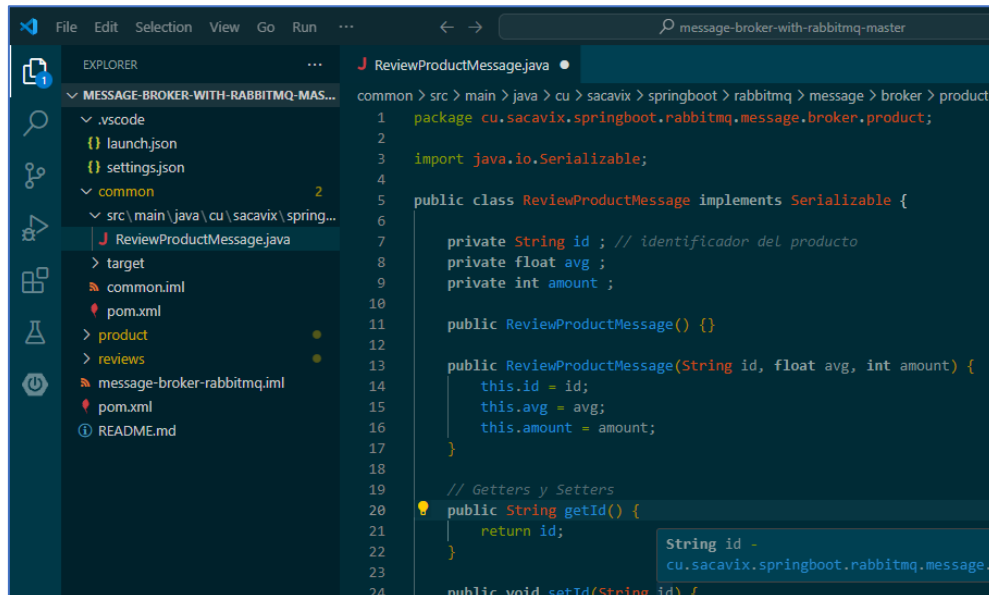
Para el ejemplo partimos de un negocio donde se gestionan productos y comentarios (los comentarios se le asignan a los productos con una evaluación), como arquitectura de microservicios tendríamos entonces dos de ellos, **ms_product** y **ms_review**. La problemática es que siempre queremos representar en productos (al menos), la cantidad de comentarios y el promedio de la valoración del producto (1 a 5). Es aquí donde juega un papel fundamental nuestro “intermediario de mensajes”. Analicemos la siguiente gráfica:



Veamos las principales clases que hacen posible el “intermediario de mensajes”

Paquete común “common”

Este paquete tiene la clase que manejará los datos del mensaje y que ambos microservicios utilizarán:

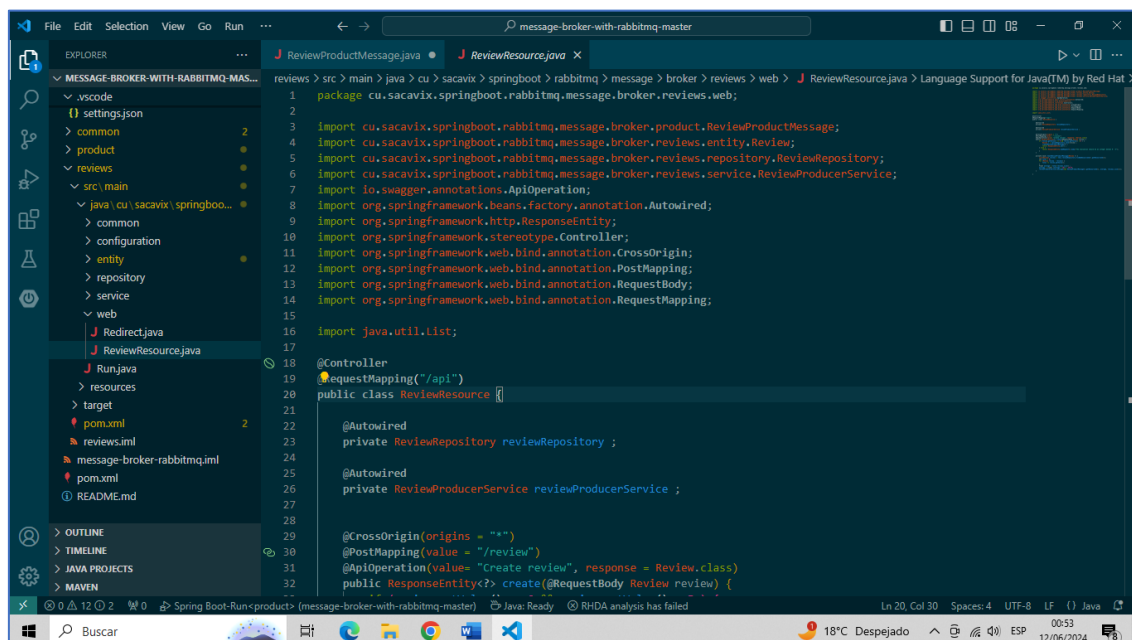


```
1 package cu.sacavix.springboot.rabbitmq.message.broker.product;
2
3 import java.io.Serializable;
4
5 public class ReviewProductMessage implements Serializable {
6
7     private String id; // identificador del producto
8     private float avg;
9     private int amount;
10
11     public ReviewProductMessage() {}
12
13     public ReviewProductMessage(String id, float avg, int amount) {
14         this.id = id;
15         this.avg = avg;
16         this.amount = amount;
17     }
18
19     // Getters y Setters
20     public String getId() {
21         return id;
22     }
23
24     public void setId(String id) {
```

Microservicio Review

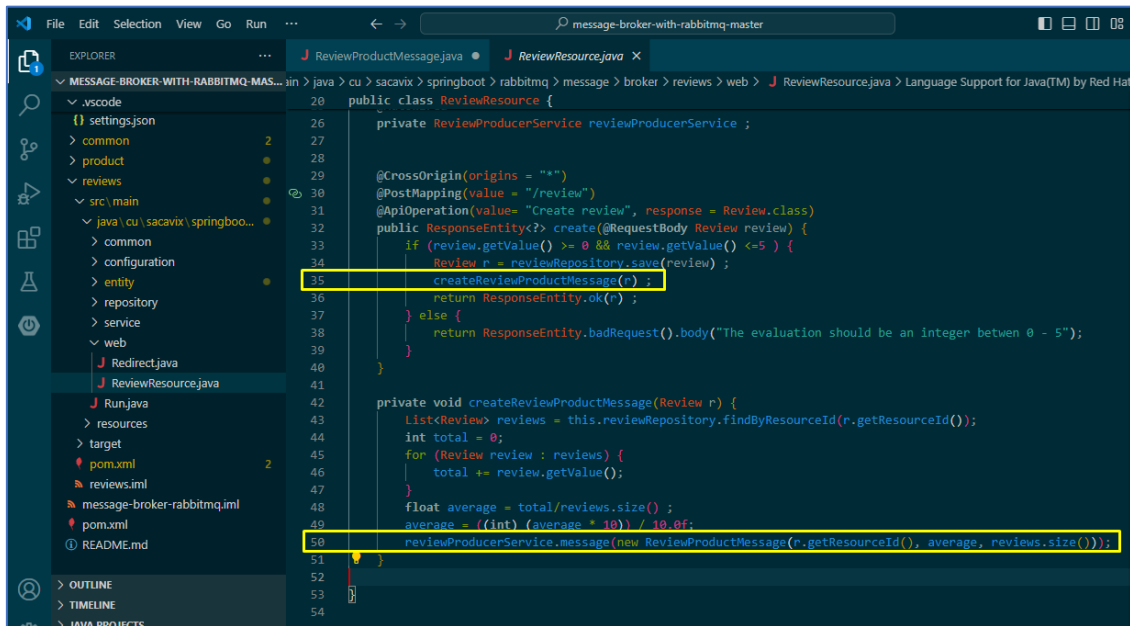
Este microservicio es el encargado de producir el mensaje *ReviewProductMessage*. En el recurso que brinda el endpoint para crear un review es donde se manda el mensaje:

Clase ReviewResource



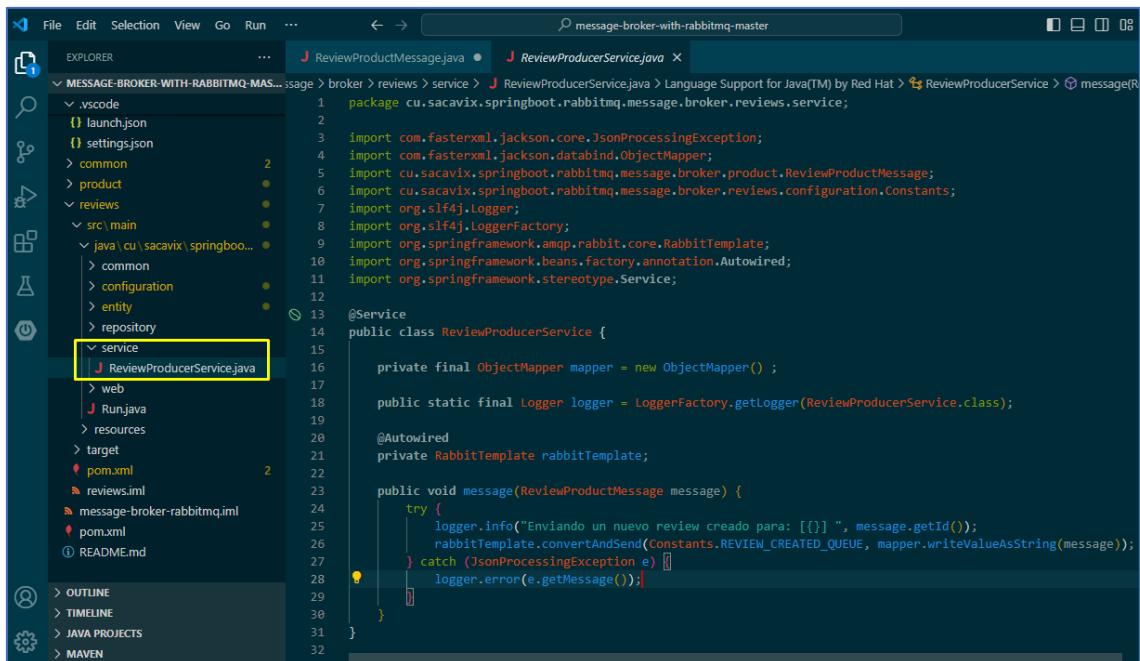
```
1 package cu.sacavix.springboot.rabbitmq.message.broker.reviews.web;
2
3 import cu.sacavix.springboot.rabbitmq.message.broker.product.ReviewProductMessage;
4 import cu.sacavix.springboot.rabbitmq.message.broker.reviews.entity.Review;
5 import cu.sacavix.springboot.rabbitmq.message.broker.reviews.repository.ReviewRepository;
6 import cu.sacavix.springboot.rabbitmq.message.broker.reviews.service.ReviewProducerService;
7 import io.swagger.annotations.ApiOperation;
8 import org.springframework.beans.factory.annotation.Autowired;
9 import org.springframework.http.ResponseEntity;
10 import org.springframework.stereotype.Controller;
11 import org.springframework.web.bind.annotation.CrossOrigin;
12 import org.springframework.web.bind.annotation.PostMapping;
13 import org.springframework.web.bind.annotation.RequestBody;
14 import org.springframework.web.bind.annotation.RequestMapping;
15
16 import java.util.List;
17
18 @Controller
19 @RequestMapping("/api")
20 public class ReviewResource {
21
22     @Autowired
23     private ReviewRepository reviewRepository;
24
25     @Autowired
26     private ReviewProducerService reviewProducerService;
27
28
29     @CrossOrigin(origins = "")
30     @PostMapping(value = "/review")
31     @ApiOperation(value = "Create review", response = Review.class)
32     public ResponseEntity<?> create(@RequestBody Review review) {
```

En la línea 35 se realiza la llamada de la funcionalidad *createReviewProductMessage* y en la línea 50 se llama el servicio encargado de enviar el mensaje.



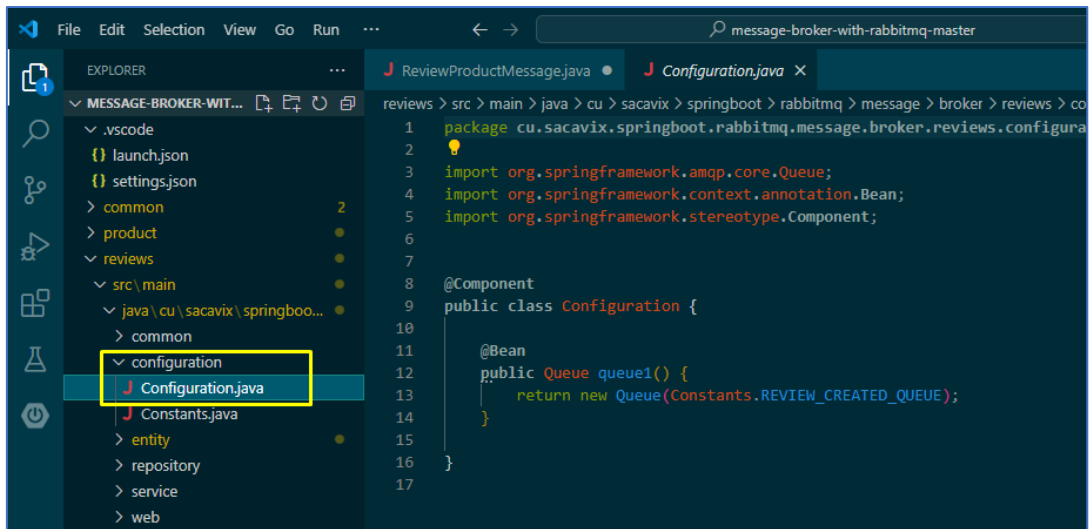
```
20 public class ReviewResource {
21     private ReviewProducerService reviewProducerService ;
22
23     @CrossOrigin(origins = "*")
24     @PostMapping(value = "/review")
25     @ApiOperation(value= "Create review", response = Review.class)
26     public ResponseEntity<?> create(@RequestBody Review review) {
27         if (review.getValue() >= 0 && review.getValue() <=5 ) {
28             Review r = reviewRepository.save(review) ;
29             createReviewProductMessage(r) ;
30             return ResponseEntity.ok(r) ;
31         } else {
32             return ResponseEntity.badRequest().body("The evaluation should be an integer between 0 - 5");
33         }
34     }
35
36     private void createReviewProductMessage(Review r) {
37         List<Review> reviews = this.reviewRepository.findByResourceId(r.getResourceId());
38         int total = 0;
39         for (Review review : reviews) {
40             total += review.getValue();
41         }
42         float average = total/reviews.size();
43         average = ((int) (average * 10)) / 10.0f;
44         reviewProducerService.message(new ReviewProductMessage(r.getResourceId(), average, reviews.size()));
45     }
46 }
```

Clase ReviewProducerService



```
1 package cu.sacavix.springboot.rabbitmq.message.broker.reviews.service;
2
3 import com.fasterxml.jackson.core.JsonProcessingException;
4 import com.fasterxml.jackson.databind.ObjectMapper;
5 import cu.sacavix.springboot.rabbitmq.message.broker.product.ReviewProductMessage;
6 import cu.sacavix.springboot.rabbitmq.message.broker.reviews.configuration.Constants;
7 import org.slf4j.Logger;
8 import org.slf4j.LoggerFactory;
9 import org.springframework.amqp.rabbit.core.RabbitTemplate;
10 import org.springframework.beans.factory.annotation.Autowired;
11 import org.springframework.stereotype.Service;
12
13 @Service
14 public class ReviewProducerService {
15
16     private final ObjectMapper mapper = new ObjectMapper() ;
17
18     public static final Logger logger = LoggerFactory.getLogger(ReviewProducerService.class);
19
20     @Autowired
21     private RabbitTemplate rabbitTemplate;
22
23     public void message(ReviewProductMessage message) {
24         try {
25             logger.info("Enviando un nuevo review creado para: [{}]", message.getId());
26             rabbitTemplate.convertAndSend(Constants.REVIEW_CREATED_QUEUE, mapper.writeValueAsString(message));
27         } catch (JsonProcessingException e) {
28             logger.error(e.getMessage());
29         }
30     }
31 }
```

Estas líneas de código son suficientes para enviar el mensaje para la cola *REVIEW_CREATED_QUEUE*, convirtiendo el mensaje *ReviewProductMessage* al formato **JSON**, la configuración de la cola se inicializa en la clase **Configuration**, donde *REVIEW_CREATED_QUEUE* es una constante literal con el valor “**created.review**”



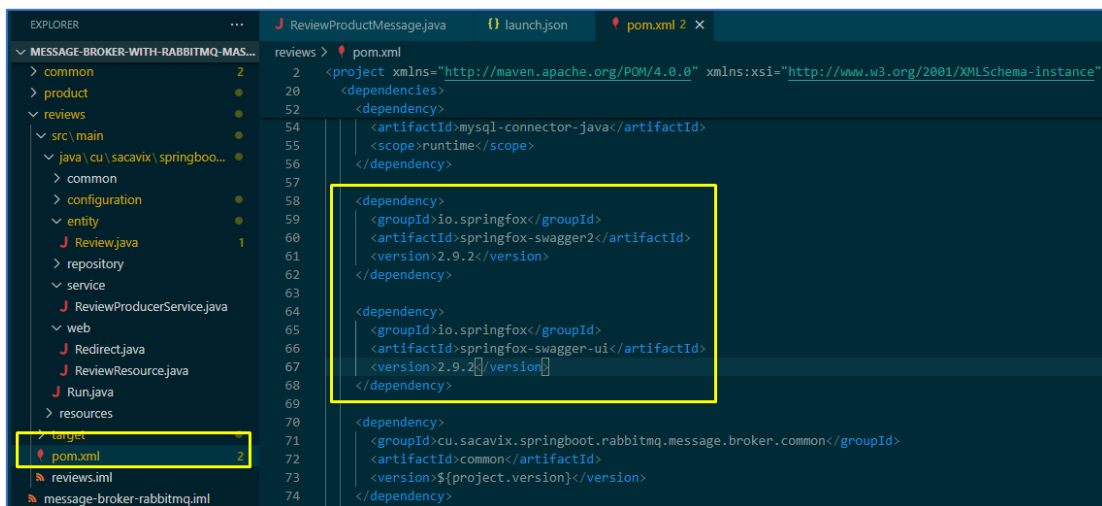
Si iniciamos este servicio sin tener el consumidor (***ms_product***) funcionando y creamos *review*, entonces los mensajes se encolarán hasta que sean consumidos, veamos esto en las siguientes tres imágenes:

Creación de review s través del Swagger en *ms_review*

Paso 1: Configuración de Swagger en *ms_review*

Dependencias de Maven

Incluir las dependencias necesarias para Swagger en tu pom.xml.



Configuración de Swagger

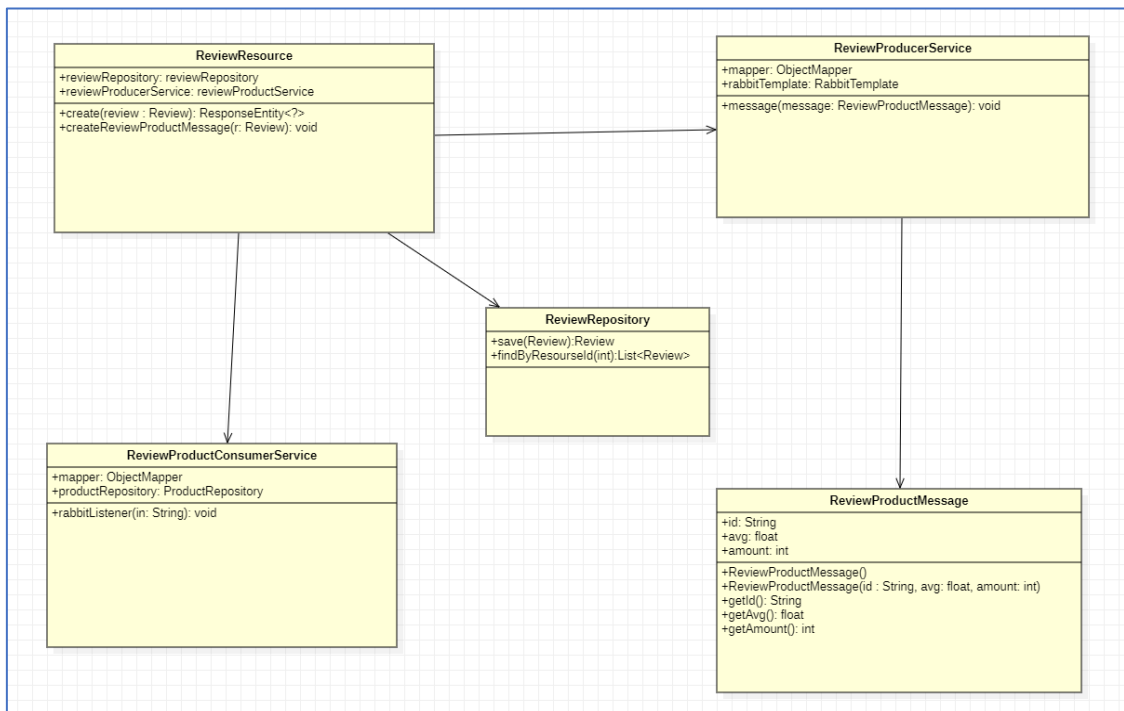
Crea una clase de configuración de Swagger en el paquete

```

14  /**
15   * Created by ups on 26/11/17.
16   */
17
18  @Configuration
19  @EnableSwagger2
20  public class SwaggerConfig {
21
22      @Bean
23      public Docket api() {
24          return new Docket(DocumentationType.SWAGGER_2)
25              .select()
26              .apis(RequestHandlerSelectors.basePackage(basePackage:"cu.sacavix."))
27              .build()
28              .apiInfo(new ApiInfo(title:"Review APIs",
29                                  description:"",
30                                  version:"",
31                                  termsOfServiceUrl:"",
32                                  new Contact(name:"", url:"", email:""),
33                                  license:"",
34                                  licenseUrl:"",
35                                  Collections.emptyList()));
36      }
37  }
38

```

DIAGRAMA DE CLASES:



GITBUB

https://github.com/WendyJacobo/ARQUITECTURA_SOFT

BIBLIOGRAFÍA

<https://sacavix.com/2020/11/message-broker-con-rabbitmq-para-microservicios/>