**Системная и программная инженерия**

Лабораторная работа №5 «Интеграция RabbitMQ с Java Spring»

**Введение**

**В прошлой лабораторной работе мы познакомились с брокером сообщений RabbitMQ, а также реализовали два вида обменников: fanout и direct. В данной лабораторной работе мы интегрируем RabbitMQ в наше приложение, и реализуем fanout и direct обменник на уровне приложения. В качестве языка разработки мы выберем Java, а конкретно Spring Boot Framework.**

**Spring Boot - это популярный Java-фреймворк, который широко используется для создания веб-приложений и микросервисов. Он предоставляет надежный набор функций и инструментов, которые упрощают быструю и эффективную разработку и развертывание приложений. Одним из ключевых преимуществ Spring Boot является его поддержка систем обмена сообщениями, таких как RabbitMQ.**

Spring Boot предоставляет встроенную платформу обмена сообщениями под названием Spring AMQP, которая упрощает работу с RabbitMQ. С помощью Spring AMQP мы можем легко создавать и настраивать слушателей сообщений, отправлять и получать сообщения, а также обрабатывать ошибки обработки сообщений. Он также предоставляет такие функции, как преобразование и маршрутизация сообщений, а также их подтверждение, которое упрощает работу со сложными сценариями обмена сообщениями.

**Практическая часть**

**Запуск RabbitMQ контейнера**

Для запуска RabbitMQ необходимо создать docker-compose.yml файл с следующим содержимым:

**version: "2.1"**

**services:**

**rabbitmq:**

**image: rabbitmq:3.10.7-management**

**hostname: rabbitmq**

**restart: always**

**volumes:**

**- ./rabbitmq:/var/lib/rabbitmq**

**ports:**

**- 15672:15672**

**- 5672:5672**

**Создание Spring Boot приложения**

Для создания базового Spring Boot приложения нам необходимо открыть IntelijIDEA, и создать новый проект, нажав на File -> New -> Project…



Рис. 1 – Создание нового проекта

На экране создания нового проекта в меню «Generators» необходимо выбрать «Spring Initializr», и нажать на кнопку «Next»

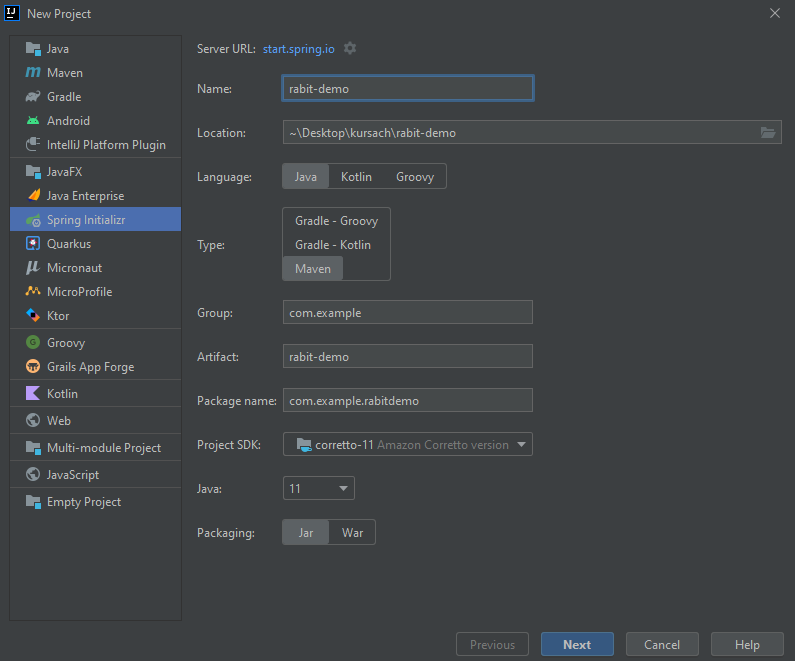


Рис. 2 – Создание Spring приложения

Далее перейдя в меню настройки Spring приложения в меню «Dependencies» необходимо выбрать «Messaging», и в выпавшем списке выбрать опцию «Spring for RabbitMQ».

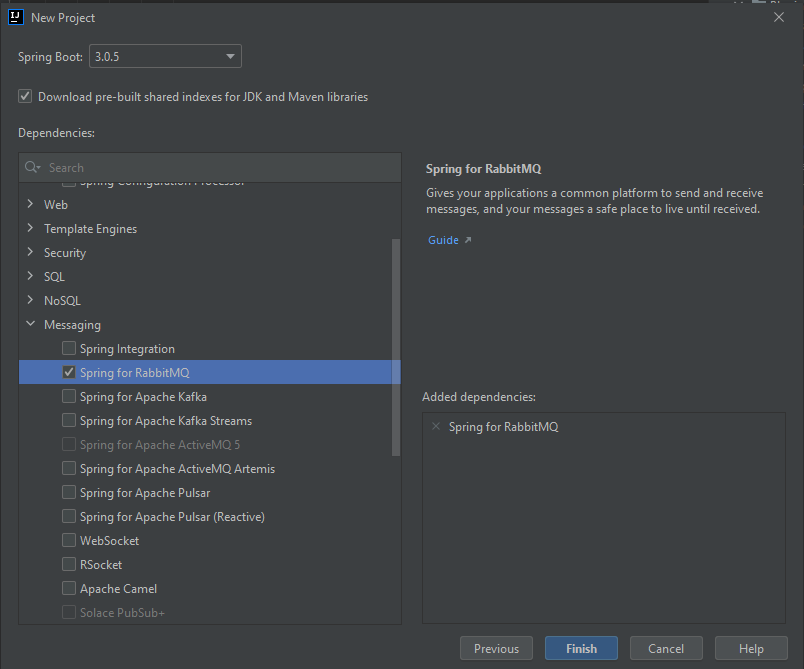


Рис. 3 – Настройка Spring приложения

После выбора опции необходимо нажать кнопку «Create»

**Создание** **RabbitMQ получателя**

В любом приложении, основанном на обмене сообщениями, вам необходимо создать получателя, который отвечает на опубликованные сообщения. Для этого нам необходимо создать класс **Receiver**, который определяет метод получения сообщений. Когда вы регистрируете его для получения сообщений, вы можете назвать его как угодно.:

import org.springframework.stereotype.Component;  
  
@Component  
public class Receiver {  
  
 public void receiveMessage(String message) {  
 System.*out*.println("Получено сообщение: " + message);  
 }  
  
}

Далее нажимаем кнопку «Add exchange», и видим, как в таблице появился созданный нами обменник.

**Регистрация слушателя**

**RabbitTemplate** от Spring AMQP предоставляет все необходимое для отправки и получения сообщений с помощью RabbitMQ. Однако для этого необходимо:

* Настроить контейнер для прослушивания сообщений.
* Создать очередь и обменник, а затем связать их между собой.
* Настроить компонент для отправки некоторых сообщений для тестирования прослушивателя.

Для этого в инициализирующем приложение классе мы должны зарегистрировать получателя в контейнере прослушивателя сообщений для получения сообщений. Фабрика подключений управляет обоими компонентами, позволяя им подключаться к серверу RabbitMQ, и настраивать их работу. Для этого дополним необходимыми компонентами:

import org.springframework.amqp.core.\*;  
import org.springframework.amqp.rabbit.connection.CachingConnectionFactory;  
import org.springframework.amqp.rabbit.connection.ConnectionFactory;  
import org.springframework.amqp.rabbit.core.RabbitTemplate;  
import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;  
import org.springframework.amqp.rabbit.listener.adapter.MessageListenerAdapter;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
@SpringBootApplication  
public class RabitDemoApplication {  
  
 static final String *rabbitHost* = "localhost";  
 static final String *rabbitUser*= "guest";  
 static final String *rabbitPassword* = "guest";  
  
 static final String *exchangeName* = "fanout-exchange";  
 static final String *queueName* = "fanout-queue";  
  
 @Bean  
 public ConnectionFactory connectionFactory() {  
 CachingConnectionFactory connectionFactory = new CachingConnectionFactory();  
 connectionFactory.setAddresses(*rabbitHost*);  
 connectionFactory.setUsername(*rabbitUser*);  
 connectionFactory.setPassword(*rabbitPassword*);  
 return connectionFactory;  
 }  
  
 @Bean  
 public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {  
 final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);  
 return rabbitTemplate;  
 }  
  
 @Bean  
 Queue queue() {  
 return new Queue(*queueName*, false);  
 }  
  
 @Bean  
 FanoutExchange exchange() {  
 return new FanoutExchange(*exchangeName*);  
 }  
  
 @Bean  
 Binding binding(Queue queue, FanoutExchange exchange) {  
 return BindingBuilder.*bind*(queue).to(exchange);  
 }  
  
 @Bean  
 SimpleMessageListenerContainer container(ConnectionFactory connectionFactory,  
 MessageListenerAdapter listenerAdapter) {  
 SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();  
 container.setConnectionFactory(connectionFactory);  
 container.setQueueNames(*queueName*);  
 container.setMessageListener(listenerAdapter);  
 return container;  
 }  
  
 @Bean  
 MessageListenerAdapter listenerAdapter(Receiver receiver) {  
 return new MessageListenerAdapter(receiver, "receiveMessage");  
 }  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(RabitDemoApplication.class, args);  
 }  
  
}

Остановимся на каждом компоненте:

**Переменные:**

* rabbitHost – Адрес сервера, на котором запущен RabbitMQ;
* rabbitUser – Логин пользователя RabbitMQ;
* rabbitPassword – Пароль пользователя RabbitMQ;
* exchangeName – Название обменника;
* queueName – Название очереди

**Bean:**

* connectionFactory - Метод, отвечающий за создание подключения к RabbitMQ;
* rabbitTemplate – Метод, отвечающий за отправку сообщений из приложения в RabbitMQ;
* quere **–** Метод, отвечающий за создание очереди;
* exchange – Метод, отвечающий за создание обменника. В качестве возвращаемого объекта мы указываем тип обменника;
* binding – Метод, отвечающий за связывание очереди с обменником. Так же мы можем указать routing key при помощи метода .with() в конце конструктора;
* container – Метод, создающий контейнер со слушателем;
* listenerAdapter – Метод, создающий слушателя

В данном случае мы создаем очередь с именем «fanout-queue» и обменник с именем «fanout-exchange», после чего связываем их между собой, и создаем контейнер для обмена сообщений.

**Отправка сообщений**

В качестве примера для отправки сообщений мы создадим класс Runner, который будет осуществлять отправку сообщений при помощи метода convertAndSend, в который мы передаем имя обменника, ключ, и текст сообщения.

import org.springframework.amqp.rabbit.core.RabbitTemplate;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Runner implements CommandLineRunner {  
  
 private final RabbitTemplate rabbitTemplate;  
  
 public Runner(RabbitTemplate rabbitTemplate) {  
 this.rabbitTemplate = rabbitTemplate;  
 }  
  
 @Override  
 public void run(String... args) throws Exception {  
 while(true) {  
 System.*out*.println("Sending message...");  
 rabbitTemplate.convertAndSend(RabitDemoApplication.*exchangeName*, "", "Hello from RabbitMQ!");  
 Thread.*sleep*(2000);  
 }  
 }  
  
}

Далее мы запускаем приложение. Как мы видим, раз в 2 секунды сообщение от отправителя попадает в fanout обменник, который направляет его в связанную с ним очередь, и отдает получателю.

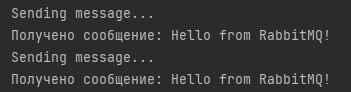


Рис. 4 – Прием сообщений

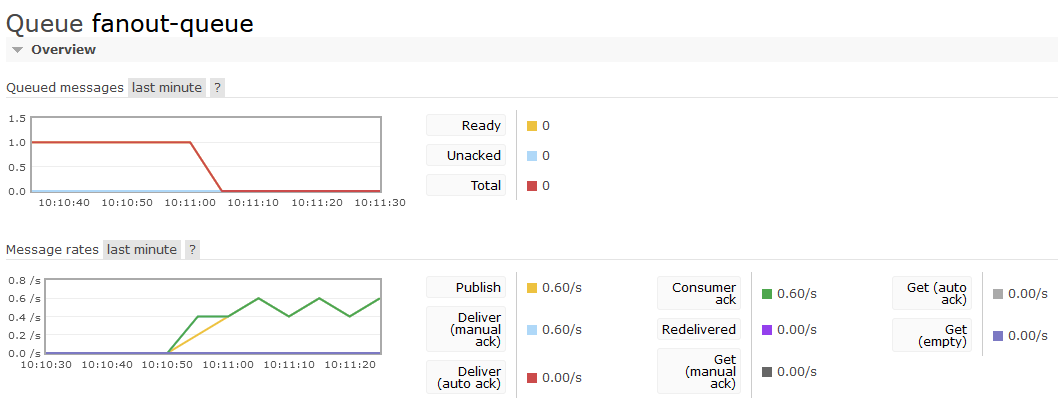


Рис. 4 – Статистика очереди fanout-queue

**Практическое задание**

1. Создать fanout обменник, в имени которого необходимо указать ФИО, и номер группы, создать очередь для него, связать их между собой, и отправить сообщение. Продемонстрировать отправку сообщения.
2. Создать direct обменник, в имени которого необходимо указать ФИО, и номер группы, создать очередь для него, связать их между собой, и отправить сообщение. Продемонстрировать отправку сообщения. (Подсказка: для реализации direct обменника необходимо использовать объект DirectExchange, а так же связывать его с очередью по ключу с помощью метода .with())