**SPRING + RABBIT**



RabbitMQ – где сообщения прыгают быстрее, чем кролики!

Rabbit Message Queue (Реббит Мессдже Киіу)

RabbitMQ является незаметным дирижёром, оркестрирующим гармоничную симфонию распределённых систем, обеспечивая беспроблемное взаимодействие между различными компонентами и принося мелодию эффективности в мир асинхронного обмена сообщениями.

Битлабжан Джавабекович - сениор Java разработчик

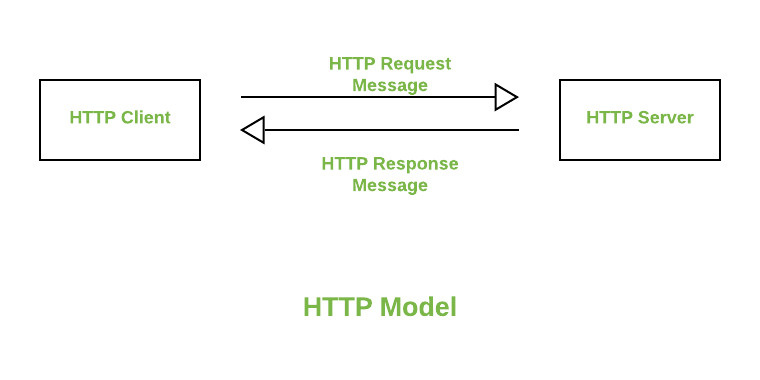
**RabbitMQ** — это асинхронный брокер сообщений, который широко используется в распределенных системах для обмена сообщениями между различными компонентами. Он реализует протокол **AMQP** (Advanced Message Queuing Protocol), который предоставляет стандартный интерфейс для передачи и обработки сообщений между различными приложениями.

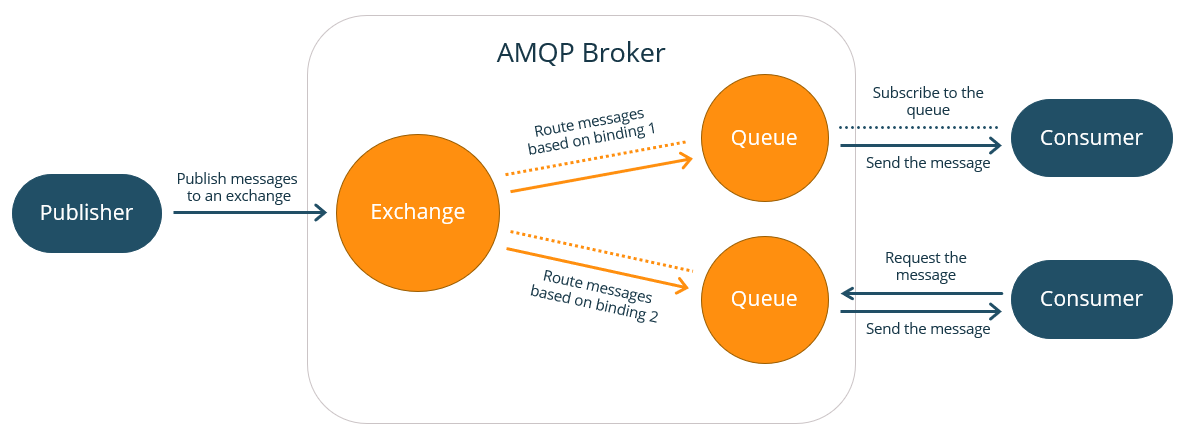
В самом начале нужно разъяснить, что такое протокол **AMQP**, и сравнить его с протоколом **HTTP**.

**HTTP** — протокол запрос-ответ, предназначенный для взаимодействия веб-приложений. Он характеризуется короткими соединениями и отсутствием сохранения состояния между запросами.

**AMQP** — протокол для обеспечения надежной и упорядоченной передачи сообщений через очередь, характеризующийся долгими соединениями и поддержкой состояния.

Давайте теперь визуально рассмотрим, как функционируют оба протокола.

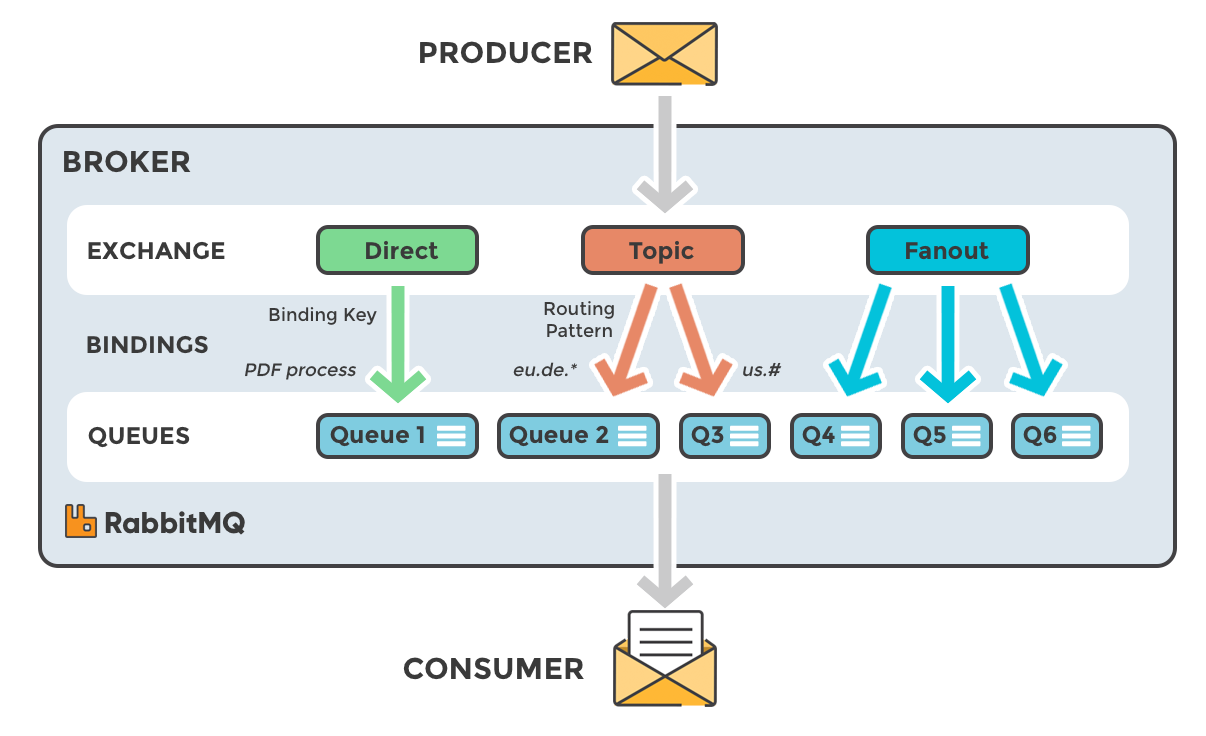




Давайте теперь рассмотрим принципы AMQP:

1) *publisher-subscriber* принцип (*publisher* не знает кто получатель, то есть малое связывание)

2) гарантированная доставка запросов (не всегда, но более надежно по сравнению с HTTP)



1) EXCHANGE (Обменник): это способ отправки сообщений. Существуют три вида:

* Direct (Прямой): сообщение отправляется на основе ключа.
* Topic (Тематический): сообщение отправляется в соответствии с паттерном.
* Fanout (Широковещательный): сообщение отправляется всем очередям.

2) QUEUE (Очередь): это место, где сообщения ожидают обработки.

3) BINDING (Привязка): это правило, которое указывает обменнику, как отправленные сообщения должны быть направлены в очереди.

Давайте начнем с **docker-compose**.

**docker-compose.yaml**

rabbitmq:

image: rabbitmq:3-management-alpine

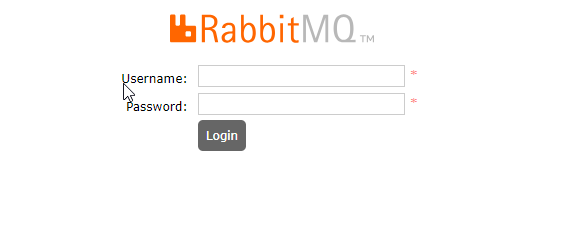
ports:

- 5672:5672

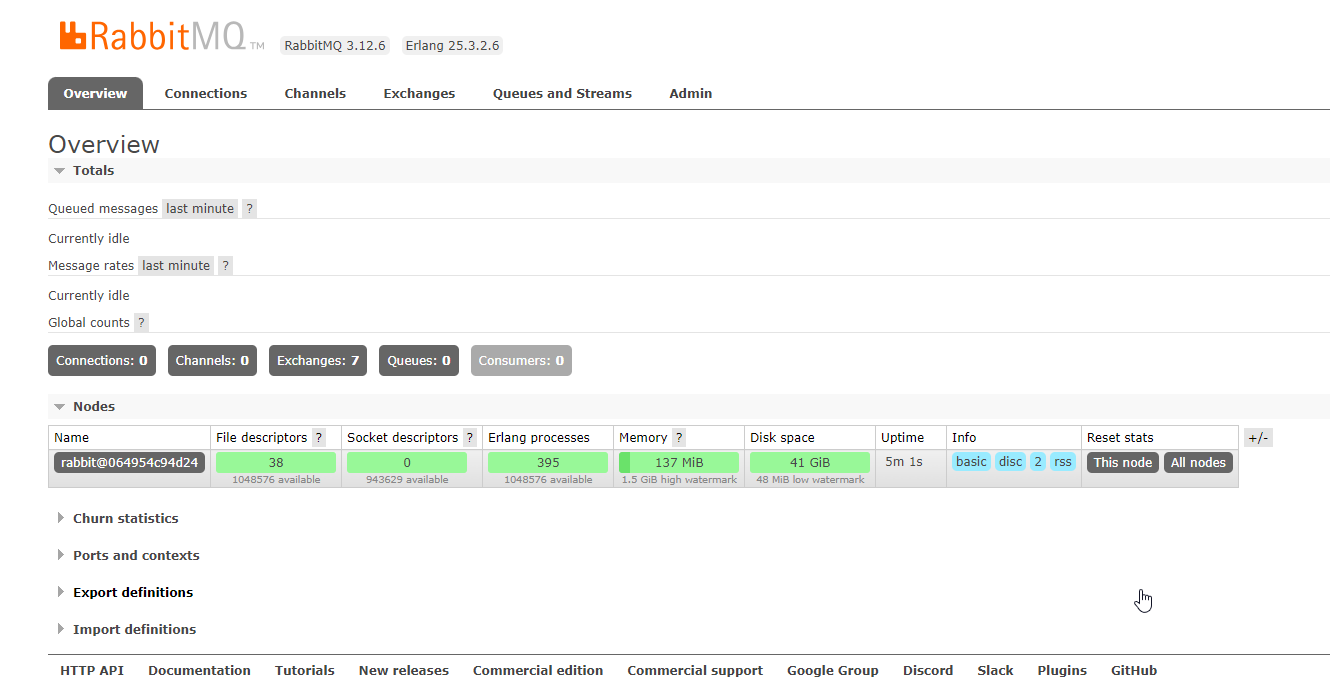
- 15672:15672

Теперь мы перейдем к ознакомлению веб-интерфейса RabbitMQ. Именно для этого используется второй порт - 15672.

Открываем **http://localhost:15672**



Вводим *username/password*: guest/guest



Далее мы можем просто пройтись по вкладкам и объяснить, для чего предназначены различные разделы. По сути, некоторые разделы понятны, например, **Connections** (Подключения).

Затем мы расскажем, что будет изменяться здесь в процессе разработки.

Теперь сама разработка. Добавляем в **build.gradle** зависимость AMQP.

**build.gradle**

implementation 'org.springframework.boot:spring-boot-starter-amqp'

Далее, соединяем наш Spring App с RabbitMQ:

**application.properties**

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

Сейчас мы будем отправлять *String* сообщение, и наш сервис будет одновременно и издателем (*publisher*) и потребителем (*consumer*). Одновременно мы будем следить за изменениями в веб-интерфейсе *RabbitMQ*.

**Примечание!** Это всего лишь простой пример для демонстрации передачи строки (String). На практике передаются классы DTO (Data Transfer Object).

Создаем контроллер.

**RabbitController.java**

@RestController

@RequiredArgsConstructor

public class RabbitController {

private final MessageSender sender;

@PostMapping("/send")

public ResponseEntity<String> sendMessage(@RequestBody String message) {

try {

sender.sendMessage(message);

return new ResponseEntity<>("Message sent to RabbitMQ: " + message, HttpStatus.OK);

} catch (Exception e) {

return new ResponseEntity<>("Failed to send message: " + e.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

}

Создаём отправитель.

**MessageSender.java**

@Service

@RequiredArgsConstructor

public class MessageSender {

private final RabbitTemplate rabbitTemplate;

public void sendMessage(String message) {

rabbitTemplate.convertAndSend("message-exchange", "key123", message);

}

}

Создаём получатель.

**MessageListener.java**

@Slf4j

@Component

public class MessageListener {

@RabbitListener(bindings = @QueueBinding(

exchange = @Exchange(value = "message-exchange", type = ExchangeTypes.DIRECT),

value = @Queue(value = "message-queue"), key = "key123"))

public void receiveMessage(String message) {

log.info("Received message: {}", message);

}

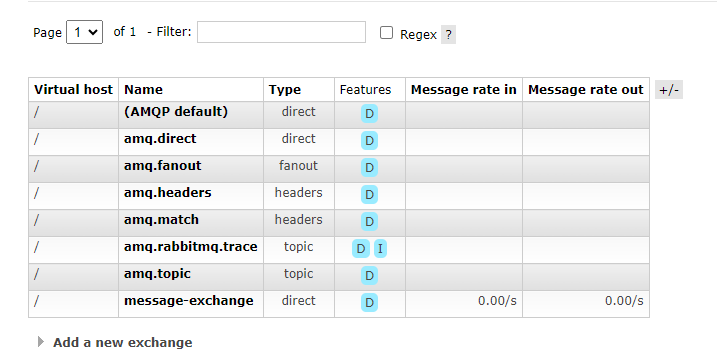
}

При запуске проекта в логах отобразится:

.....  
Attempting to connect to: [localhost:5672]  
Created new connection: rabbitConnectionFactory#75b3ef1a:0/SimpleCon  
.....

И вы можете увидеть изменения в веб-интерфейсе, где добавились новые записи:

1) **Connection** (подключение)  
2) **Exchanges** (обмены) для сообщений (*message-exchange*)



**TOPIC, FANOUT exchange types**

Давайте в качестве примера рассмотрим приложение GLOVO (сервис доставки еды).

Создадим два *Spring* проекта:

1. order-service
2. notification-service

Начнем с **order-service**.

Добавляем в **build.gradle** зависимость AMQP.

**build.gradle**

implementation 'org.springframework.boot:spring-boot-starter-amqp'

Создаём **OrderDTO**.

**OrderDTO.java**

@Data

@AllArgsConstructor

@NoArgsConstructor

public class OrderDTO implements Serializable {

private String restaurant;

private String courier;

private List<String> foods;

private String status;

}

OrderDTO должен обязательно реализовывать интерфейс **Serializable**, так как мы будем отправлять объекты в виде **byte[ ]**

Далее, настраиваем **application.properties**.

**application.properties**

mq.order.topic.exchange=order-create-exchange

mq.order.fanout.exchange=order-update-exchange

Далее, создаем **OrderPublisher**.

**OrderPublisher.java**

@Slf4j

@Service

@RequiredArgsConstructor

public class OrderPublisher {

private final RabbitTemplate rabbitTemplate;

@Value("${mq.order.topic.exchange}")

private String topicExchange;

@Value("${mq.order.fanout.exchange}")

private String fanoutExchange;

public void sendOrderToPrepare(OrderDTO order, String region) {

String routingKey = "order." + region;

rabbitTemplate.convertAndSend(topicExchange, routingKey, order);

}

public void updateOrderStatus(OrderDTO orderDTO, String status) {

orderDTO.setStatus(status);

rabbitTemplate.convertAndSend(fanoutExchange, "", orderDTO);

}

}

Теперь создадим второй проект — **notification-service**.

Также, в **build.gradle** добавляем зависимость AMQP.

**build.gradle**

implementation 'org.springframework.boot:spring-boot-starter-amqp'

Проект **notification-service** также должен иметь OrderDTO, чтобы принимать листенер.

**OrderDTO.java**

@Data

@AllArgsConstructor

@NoArgsConstructor

public class OrderDTO implements Serializable {

private String restaurant;

private String courier;

private List<String> foods;

private String status;

}

Настраиваем **application.properties**.

**application.properties**

mq.order.topic.exchange=order-create-exchange

mq.order.fanout.exchange=order-update-exchange

Теперь создадим наши Listeners: на данный момент они настроены на тип **"fanout"**, что означает, что все очереди будут получать эти сообщения.

**OrderNotificationListener.java**

@Slf4j

@Service

public class OrderNotificationListener {

@RabbitListener(bindings = @QueueBinding(

value = @Queue(value = "almaty\_orders\_queue"),

exchange = @Exchange(value = "${mq.order.topic.exchange}", type = ExchangeTypes.TOPIC),

key = "order.#"))

public void receiveAlmatyOrder(OrderDTO order) {

log.info("Received order - ORDER:{}", order);

}

// Listener для клиента обновления статусов заказов

@RabbitListener(bindings = @QueueBinding(

value = @Queue(value = "customer\_updates\_queue"),

exchange = @Exchange(value = "${mq.order.fanout.exchange}", type = ExchangeTypes.FANOUT),

key = ""))

public void receiveCustomerOrderStatusUpdate(OrderDTO order) {

log.info("Customer notification - ORDER:{} status updated: {}", order, order.getStatus());

}

// Listener для курьеров: обновления статусов заказов

@RabbitListener(bindings = @QueueBinding(

value = @Queue(value = "courier\_updates\_queue"),

exchange = @Exchange(value = "${mq.order.fanout.exchange}", type = ExchangeTypes.FANOUT),

key = ""))

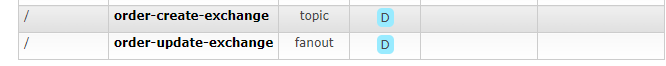
public void receiveCourierOrderStatusUpdate(OrderDTO order) {

log.info("Courier notification - ORDER:{} status updated: {}", order, order.getStatus());

}

}

Запускаем проект и проверяем:



Как мы видим, у нас создались подключения (*Connections*) и обменники (*Exchanges*).

Создаем контроллер:

**OrderController.java**

@RestController

@RequiredArgsConstructor

@RequestMapping("/order")

public class OrderController {

private final OrderPublisher orderPublisher;

@PostMapping

public ResponseEntity<String> createOrder(@RequestBody OrderDTO orderDTO) {

try {

orderPublisher.sendOrderToPrepare(orderDTO, "almaty");

return new ResponseEntity<>("Order created", HttpStatus.OK);

} catch (Exception e) {

return new ResponseEntity<>("Order failed to create", HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

@PutMapping("/{status}")

public ResponseEntity<String> updateOrder(@RequestBody OrderDTO orderDTO, @PathVariable String status) {

try {

orderPublisher.updateOrderStatus(orderDTO, status);

return new ResponseEntity<>("Order updated", HttpStatus.OK);

} catch (Exception e) {

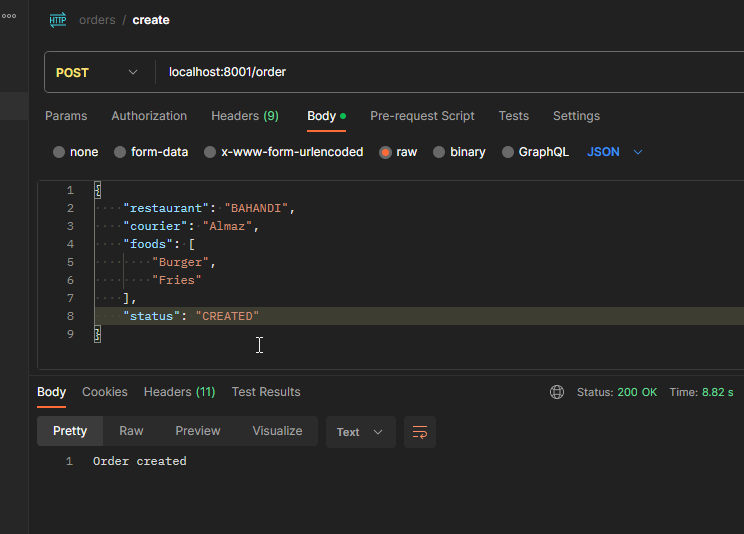
return new ResponseEntity<>("Order failed to updated", HttpStatus.INTERNAL\_SERVER\_ERROR);

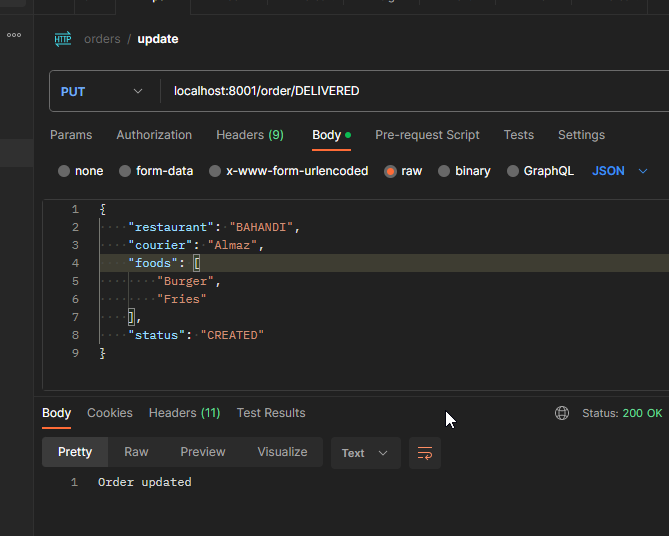
}

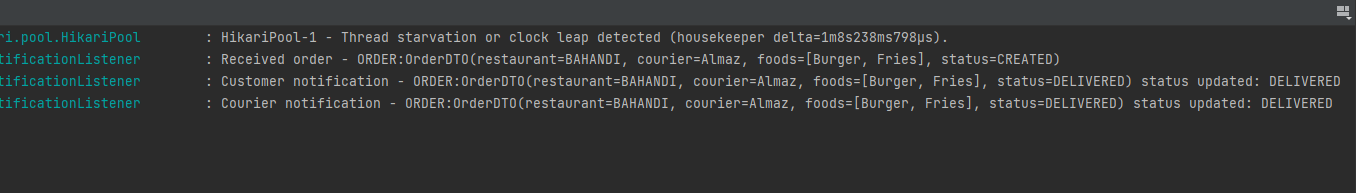
}

}

А теперь давайте выполним проверку с использованием Postman.





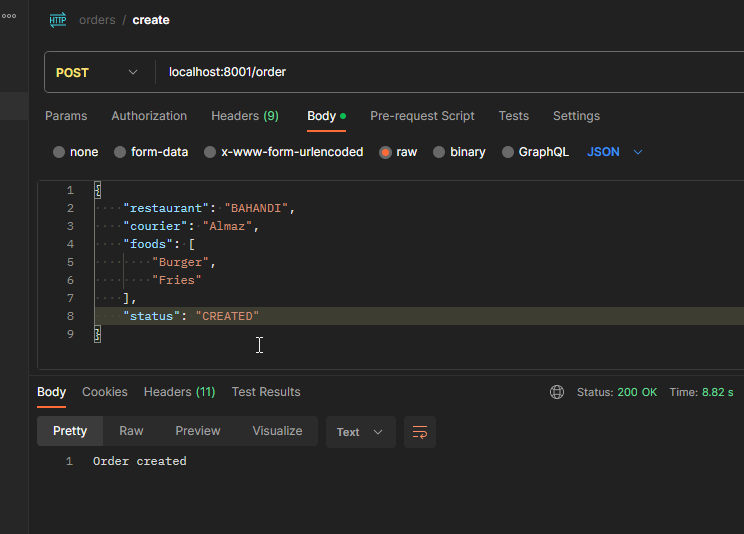


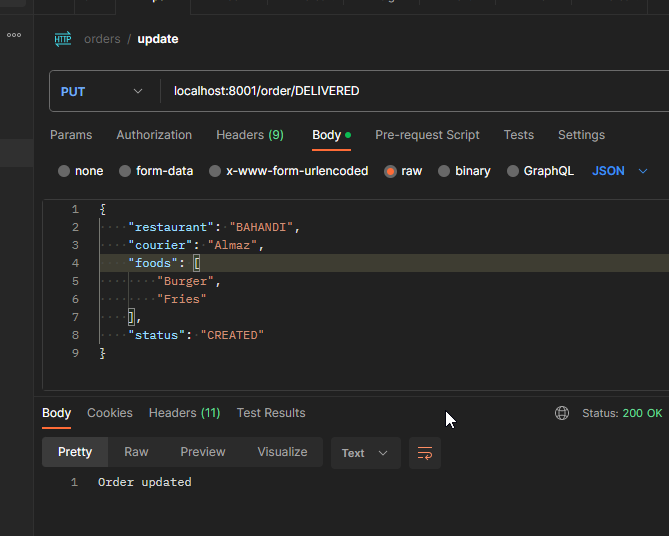
**Retry механизм и DLQ (dead letter queue)**

Retry механизм в системах обмена сообщениями, таких как RabbitMQ, позволяет автоматически повторять обработку сообщений при возникновении временных ошибок, тем самым увеличивая надежность системы и минимизируя потерю данных.

Мы будем использовать предыдущий пример с **order** и **notification** сервисами.

Для начала нужно ещё раз проверить, работают ли наши *listeners*





Затем мы создаем конфигурацию для механизма повторной попытки (*Retry*). Как обычно, существует два способа:

1) через конфиг **application.properties**  
2) через конфиг класс

Мы настроим механизма повторной попытки (*Retry*) через конфигурационный файл **application.properties**.

**application.properties**

spring.rabbitmq.listener.simple.retry.enabled=true

spring.rabbitmq.listener.simple.retry.max-attempts=3

spring.rabbitmq.listener.simple.retry.initial-interval=5000

spring.rabbitmq.listener.simple.retry.multiplier=2.0

spring.rabbitmq.listener.simple.retry.max-interval=10000

И мы сами сделаем имитацию ошибки в коде, чтобы перенаправлять сообщения. Для этого изменяем наш метод **receiveAlmatyOrder()**:

@RabbitListener(bindings = @QueueBinding(

value = @Queue(value = "almaty\_orders\_queue"),

exchange = @Exchange(value = "${mq.order.topic.exchange}", type = ExchangeTypes.TOPIC),

key = "order.#"))

public void receiveAlmatyOrder(OrderDTO order) {

try {

log.info("Received order - ORDER:{}", order);

processOrder(order);

} catch (Exception e) {

log.error("Error processing order - ORDER:{}, ERROR:{}", order, e.getMessage());

throw e;

}

}

private void processOrder(OrderDTO order) {

if (someConditionFails()) { // ИМИТАЦИЯ ОШИБКИ

throw new RuntimeException("Failed to process order");

}

// обработка запроса order

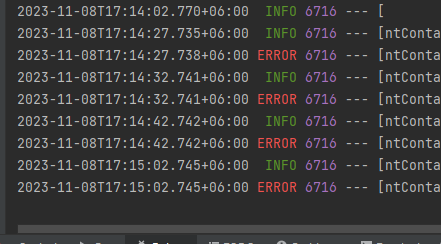
}

private boolean someConditionFails() {

return true;

}

Теперь проверяем запрос и получаем ошибку



Как видим, после возникновения ошибки, мы ожидаем 5 секунд (**.initial-interval=5000**).  
Затем увеличиваем этот интервал в 2 раза (**.multiplier=2.0**) и ждем 10 секунд.  
После этого продолжаем ждать ещё 10 секунд, так как максимальный интервал установлен в 10 секунд (**.max-interval=10000**).



Можно экспериментировать (“поиграть”) с этими значениями в файле **application.properties** и одновременно следить за админ-панелью RabbitMQ.

**Что такое Dead Letter Queue (DLQ)?**

**DLQ** — это специальная очередь, предназначенная для хранения сообщений, которые не могут быть обработаны из-за какой-либо ошибки. Это важно для:

* **Избежание потери сообщений:** Если сообщение не может быть обработано, оно не теряется, а сохраняется в DLQ.
* **Анализ проблем:** DLQ позволяет анализировать ошибки обработки сообщений.
* **Повторная обработка:** Сообщения в DLQ могут быть возвращены в основную очередь после исправления проблемы, которая вызвала ошибку.

В DLQ мы можем сохранять необработанные сообщения, и администратор может запустить процесс их повторной обработки. Таким образом, сообщение может быть переотправлено в определенный *RabbitMQ listener* после внесения необходимых корректив.

Для добавления **Dead Letter Queue (DLQ)** мы создадим бины в классе **RabbitConfig**. Почему же мы не используем аннотации, такие как **@RabbitListener** с **@QueueBinding** и т.д.?  
Это связано с тем, что через бины предоставляется более гибкая настройка и полный контроль над процессом. В аннотациях, таких как **@RabbitListener**, некоторые настройки ограничены, и использование бинов предоставляет больше возможностей для тонкой настройки и управления.

**RabbitConfig.java**

@Bean

public Queue deadLetterQueue() {

return QueueBuilder.durable("almaty\_orders\_queue.dlq").build();

}

@Bean

public TopicExchange deadLetterExchange() {

return ExchangeBuilder.topicExchange("dlx").durable(true).build();

}

@Bean

public Binding DLQbinding() {

return BindingBuilder

.bind(deadLetterQueue())

.to(deadLetterExchange())

.with("dlx.almaty\_orders");

}

Далее, меняем как раз-таки наш метод **receiveAlmatyOrders()**:

@RabbitListener(bindings = @QueueBinding(

value = @Queue(value = "almaty\_orders\_queue",

arguments = {

@Argument(name = "x-dead-letter-exchange", value = "dlx"),

@Argument(name = "x-dead-letter-routing-key", value = "dlx.almaty\_orders")

}),

exchange = @Exchange(value = "${mq.order.topic.exchange}", type = ExchangeTypes.TOPIC),

key = "order.#"))

public void receiveAlmatyOrder(OrderDTO order) {

try {

log.info("Received order - ORDER:{}", order);

processOrder(order);

} catch (Exception e) {

log.error("Error processing order - ORDER:{}, ERROR:{}", order, e.getMessage());

throw e;

}

}

В контексте аннотации **@RabbitListener** важно удалить **almaty\_orders\_queue**. Это необходимо для того, чтобы RabbitMQ автоматически пересоздал очередь и подключил ее к Dead Letter Queue (DLQ).

После получения ошибки, повторные вызовы отправляются в *Dead Letter Queue (DLQ)*, и вы можете просматривать это в *RabbitMQ Admin Web*.



Вот и все!

Если мы хотим отправить сообщение снова из *Dead Letter Queue (DLQ)*, мы можем добавить аннотацию **@RabbitListener** специально для DLQ в коде. Затем, используя *Rabbit CLI*, мы можем получить сообщение и отправить его снова (воспользоваться командой SEND).