

Практическая работа №5. Межсервисное взаимодействие

Цель работы

Целью данной работы является закрепление практических навыков организации межсервисного взаимодействия, в частности, работы с брокером сообщений RabbitMQ и отправкой асинхронных сообщений между различными программами.

Теоретическое введение

В предыдущих работах уже были рассмотрены теоретические аспекты межсервисного взаимодействия. В данной работе внимание будет уделено неблокирующему (асинхронному) взаимодействию между сервисами.

Когда будет использоваться асинхронный обмен сообщениями? Данный подход используется тогда, когда у клиента нет необходимости в ожидании ответа сервера. Отправив сообщение, клиент может спокойно продолжить выполнять свой код. Более того, может возникнуть ситуация, когда ответ от сервера даже не предполагается. Такой стиль общения также называется неблокирующим, потому что отправка сообщения не блокирует выполнение кода на клиенте.

Для организации такого стиля общения достаточно часто используются брокеры сообщений. Брокер сообщений представляет собой отдельный сервис, который занимается маршрутизацией сообщений между различными сервисами, а также некоторыми другими вспомогательными функциями, которые могут использоваться опционально.

В данной работе будет рассмотрена работа с RabbitMQ. Это брокер сообщений с открытым исходным кодом, работающий по протоколу AMQP.

Основными действующими лицами в RabbitMQ являются издатель (producer) и потребитель (consumer). Издатель занимается отправкой сообщений, а потребитель отвечает за их получение. Ключевым элементом, который находится между поставщиком и потребителем, является очередь

сообщений. Именно в очереди сообщений осуществляется хранение сообщений.

Тем не менее поставщик не работает напрямую с очередью сообщений. Для взаимодействия с очередями у поставщика есть обменник (exchange). Как правило поставщик не знает, было ли его сообщений доставлено в очередь.

Обменник необходим для того, чтобы маршрутизировать сообщения по очередям. Подходы для распределения бывают разными: сообщение может быть отправлено в какую-то конкретную очередь или, например, сразу в несколько очередей. Именно для выполнения таких правил и выполняется обменником.

Сами правила для обменников называются связями (binding). Они регламентируют, в какую из очередей должны попадать сообщения. При этом один обменник может быть связан с очередью более чем одной связью.

Далее будут рассмотрены некоторые варианты межсервисного взаимодействия на основе RabbitMQ.

1. Базовое взаимодействие между сервисами

Рассмотрим немного упрощённое взаимодействие между сервисами. Для этого понадобится источник сообщений, очередь и потребитель. Эта модель взаимодействия достаточно сильно похожа на почту. Рассмотрим роли немного подробнее.

Поставщик в этой модели имеет роль отправителя, который хочет отправить кому-то письмо.

Получатель – это тот, кому это письмо необходимо получить.

Между ними имеется очередь, которая играет роль почтового ящика. В очереди хранятся сообщения, отправленные поставщиком. Сама по себе очередь не имеет ограничений на количество хранимых сообщений. При этом сообщения в неё могут отправлять самые разные поставщики, а принимать также может любое количество потребителей.

На Рисунке 5.1 приведён пример того, как будут обозначаться все три роли на схемах.

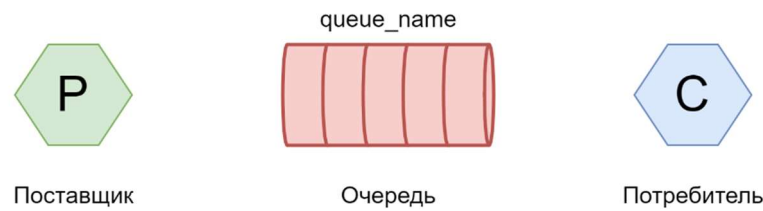


Рисунок 5.1 — Условные обозначения поставщика, очереди и потребителя на схемах

В первом примере будет рассматриваться один из самых простых вариантов взаимодействия. Система будет состоять из одного поставщика, одной очереди и одного потребителя (Рисунок 5.2). Для реализации такого взаимодействия понадобится две программы: одна будет отправлять сообщения в очередь, а другая будет эти сообщения принимать.

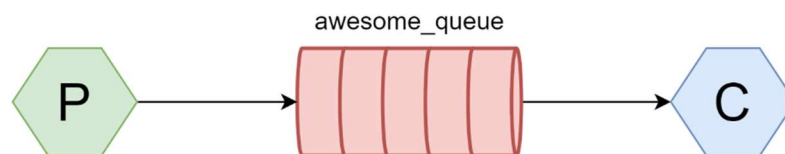


Рисунок 5.2 — Схема простейшего асинхронного взаимодействия

Данная работа будет выполняться на языке программирования Python 3. Для дальнейшей работы понадобится установить библиотеку *pika* для работы с RabbitMQ из Python. Это делается командой:

```
pip install pika
```

Вместе с этим нам понадобится запущенный RabbitMQ. Способ, используемый здесь не требует установки брокера на устройство. Для выполнения данной работы брокер сообщений будет запущен в контейнере Docker:

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

С помощью вышеприведённой команды будет запущена версия RabbitMQ, имеющая веб интерфейс, предназначенный для управления

брокером, а также его мониторинга. Он доступен по адресу <http://localhost:15672>. Логин и пароль по умолчанию – guest.

Отправка сообщений

Составим программу, которая будет выполнять отправку сообщений в очередь. Если точнее, она будет подключаться к брокеру, отправлять одно сообщение с очередь и отключаться. Для начала этого вполне достаточно.

В первую очередь нам необходимо подключиться к брокеру сообщений. Выше, запустив локально RabbitMQ, мы открыли для него порты 5672 и 15672. Первый нам необходим для работы с сообщениями, в то время как через второй можно получить доступ к панели управления.

Установить соединение с брокером достаточно просто: необходимо создать подключение, а затем в нём создать канал, по которому и будет происходить общение с RabbitMQ.

```
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost',
5672))
channel = connection.channel()
```

С помощью кода выше было создано подключение к брокеру сообщений, который находится на хосте с именем localhost, через порт 5672. При необходимости работы с удалённым брокером сообщений вместо localhost можно указать ip-адрес необходимой машины или имя хоста.

Для дальнейшей отправки сообщений необходимо убедиться, что очередь, куда будет происходить отправка сообщений существует. В противном случае RabbitMQ проигнорирует отправленное сообщение. Создадим очередь, куда будем слать сообщения. Назовём её `awesome_queue`.

```
channel.queue_declare(queue='awesome_queue')
```

На данном этапе уже можно совершать отправку сообщений в очередь. Для того, чтобы не углубляться в данный момент в работу обменников, будем использовать обменник по умолчанию, указывая пустую строку при

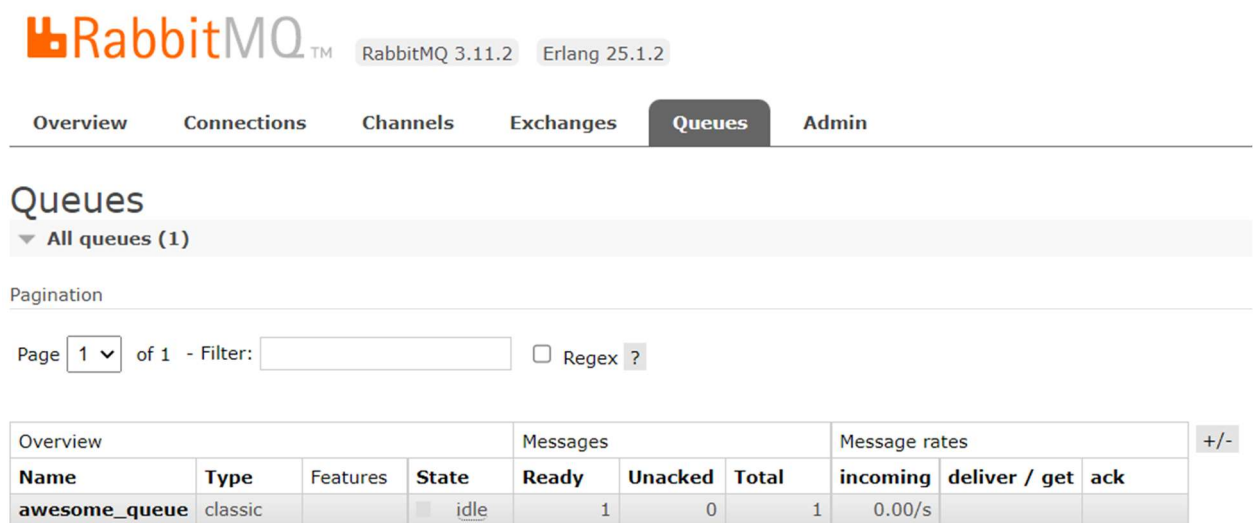
отправке сообщения. Данный способ позволяет отправлять сообщения по имени очереди, указав его в параметре `routing_key`.

```
channel.basic_publish(
    exchange='',
    routing_key='awesome_queue',
    body='My first message'
)
print('SENT: My first message')

connection.close()
```

Также в конце программы добавим вывод в консоль, чтобы видеть, что было отправлено в очередь, и безопасное закрытие соединения с брокером.

Если сейчас выполнить данный скрипт, то будет создана очередь `awesome_queue`, а затем в неё через обменник по умолчанию будет отправлено сообщение `My first message`. В данный момент скрипта для приёма сообщений ещё нет, но есть возможность посмотреть на состояние новой очереди через веб интерфейс (Рисунок 5.3).



RabbitMQ 3.11.2 Erlang 25.1.2

Overview Connections Channels Exchanges **Queues** Admin

Queues

▼ All queues (1)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview				Messages			Message rates			+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
awesome_queue	classic		idle	1	0	1	0.00/s			

Рисунок 5.3 — Интерфейс панели управления RabbitMQ

Приём сообщений

Теперь необходимо создать ещё один скрипт, который будет принимать сообщения из очереди. По аналогии с предыдущим скриптом необходимо создать подключение, канал и очередь `awesome_queue` с помощью функции `queue_declare`. Функция создания очереди работает так, что, если

очередь уже существует, она не будет создаваться заново, поэтому очередь так и будет оставаться одна.

Для чего производится создание очереди в обоих скриптах? Это необходимо, так как нет точной уверенности, что получатель будет запущен после поставщика. Таким образом, кто бы первым не запустился, у нас есть гарантия, что очередь будет создана до её использования.

Рассмотрим процесс приёма сообщений более подробно. Этот процесс немного сложнее чем отправка сообщений. Создаётся бесконечный процесс, который ожидает сообщения из очереди, а при каждом их появлении вызывает определённую функцию для обработки сообщения.

Создадим простейшую функцию для обработки сообщений. Она имеет сигнатуру, определённую RabbitMQ. В нашем случае функция `callback` будет просто выводить текст сообщения в консоль.

```
def process_message(channel, method, properties, body):  
    print(f'RECEIVED: {body}')
```

После определения обработчика сообщений необходимо подписаться на необходимую очередь, а также запустить процесс ожидания сообщений.

```
channel.basic_consume(  
    queue='awesome_queue',  
    auto_ack=True,  
    on_message_callback=process_message  
)  
  
print('Waiting for messages...')  
channel.start_consuming()
```

Проверка работоспособности

Чтобы убедиться в том, что наш обмен сообщениями работает достаточно несколько раз запустить скрипт с поставщиком и один раз с потребителем. В консоль с запущенным потребителем должно прийти ровно столько сообщений, сколько раз был запущен поставщик.

Данный пример имеет сильно упрощённый вид. Он призван показать основы работы с RabbitMQ. Почти всегда на практике требуется производить

более тщательную настройку компонентов взаимодействия, а также правил маршрутизации сообщений.

2. Распределение задач по очередям

В первом примере был разобран простейший случай взаимодействия сервисов через брокера сообщений. В этом же мы создадим очередь, в которая будет распределять ресурсоёмкие задачи между несколькими подписчиками.

Очереди в RabbitMQ представляют из себя достаточно большой буфер, в котором может накапливаться весьма много сообщений. Весьма часто сообщения, пересылаемые между сервисами, запускают выполнение каких-то задач. Так как мы имеем дело с асинхронным общением, и как правило ответа никто не ждёт, нет необходимости обрабатывать сообщения прямо в момент их получения. Выполнение задачи вполне можно отложить на какое-то время.

В данном примере мы рассмотрим распределение ресурсоёмких задач между несколькими потребителями. Если все они будут заняты, то сообщения будут накапливаться в очереди, пока какой-нибудь из потребителей не освободится. Ниже представлена схема взаимодействия сервисов (Рисунок 5.4):

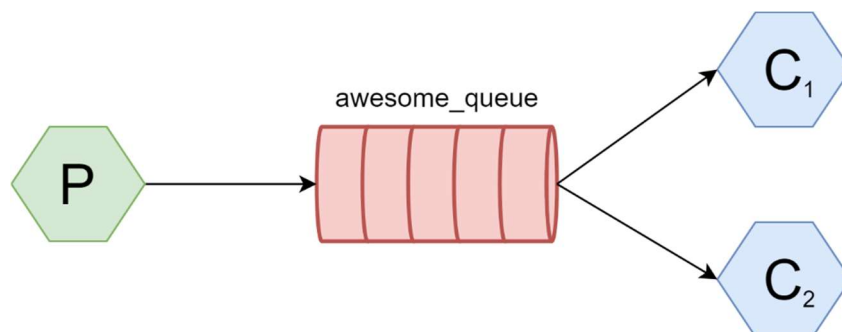


Рисунок 5.4 — Схема подключения двух потребителей к одной очереди

Добавление нагрузки

Данный пример предполагает наличие ресурсоёмких задач. Так как наша основная задача разобраться с работой RabbitMQ, создавать какие-то

реальные задания мы не будем. Вместо этого будем их симулировать, создавая задержку при обработке сообщений.

Общение будет строиться таким образом: передаём поставщику сообщение для отправки, отправляем его в очередь, принимаем сообщение потребителем, приостанавливаем его выполнение на столько секунд, сколько точек в сообщении.

Для начала изменим код поставщика. Добавим в него использование аргументов в качестве текста сообщения.

```
import sys

message_text = ' '.join(sys.argv[1:]) or 'Empty message'
channel.basic_publish(
    exchange='',
    routing_key=queue_name,
    body=message_text
)
```

Теперь мы можем при запуске поставщика передавать ему сообщения, которые хотим отправить.

Далее необходимо видоизменить обработчик сообщений в потребителе. Мы будем считать количество точек в полученном сообщении, а затем на такое же количество секунд останавливать выполнение кода. По выполнении задачи будем выводить об этом сообщение в консоль.

```
from time import sleep

def process_message(channel, method, properties, body):
    print(f'RECEIVED: {body.decode()}')
    sleep(body.count(b'.'))
    print('INFO: Done')
```

С помощью уже написанного кода мы добавили симуляцию полезной нагрузки. Каким образом сейчас работает распределение сообщений между задачами? На данном этапе RabbitMQ отдаёт сообщения потребителям по очереди, проходя по всем подряд, а затем возвращаясь к первому. Фактически он отдаёт каждое n-ое сообщение каждому n-ому подписчику.

Подтверждение сообщений

Сейчас мы работаем с сообщениями, обработка которых занимает некоторое время. Текущая реализация межсервисного взаимодействия имеет заметный изъян. Если вдруг работа одного из потребителей прервётся, задача не будет выполнена, а сообщение будет утеряно. Это произойдёт из-за того, что RabbitMQ удалит сообщение сразу, когда оно будет доставлено потребителю.

Конечно, мы не хотим терять сообщения, поэтому необходимо сделать так, чтоб в случае непредвиденного завершения работы нагруженного потребителя его задачи передавались другому потребителю.

Именно для таких ситуаций в RabbitMQ есть механизм подтверждения выполнения сообщений (ack). Это подтверждение отправляется подписчиком по выполнении задачи, информируя RabbitMQ о том, что задача выполнена.

В случае, когда потребитель закрыл соединение и не отправил подтверждений, RabbitMQ передаст сообщение другому потребителю, поняв, что оно не было обработано.

В коде потребителя у нас указан флаг `auto_ack=True`. Это включает режим автоматического подтверждения, в котором сообщение считается успешно доставленным сразу после его получения.

Для того, чтобы подтверждение производилось после обработки сообщения необходимо отключить автоматический режим и начать самостоятельно отправлять подтверждение из обработчика сообщения.

```
def process_message(channel, method, properties, body):
    print(f'RECEIVED: {body.decode()}')
    sleep(body.count(b'.'))
    print('INFO: Done')
    channel.basic_ack(delivery_tag=method.delivery_tag)

channel.basic_consume(
    queue=queue_name,
    on_message_callback=process_message
)
```

Если протестировать полученный код, то даже если один из потребителей будет остановлен во время обработки сообщений, это сообщение будет отправлено следующему обработчику.

Если не используется автоматический режим подтверждения сообщений, важно следить за тем, чтобы подтверждение в итоге отсылалось на каждое сообщение. В противном случае потребление памяти RabbitMQ будет постоянно расти.

Немного ранее мы сделали обработку сообщений более стабильными. Тем не менее этого недостаточно, потому что в случае, если произойдёт падение сервера RabbitMQ, будут потеряны и все очереди, и все сообщения.

Для начала необходимо сделать очередь устойчивой. Для этого необходимо объявить её как устойчивую (с флагом `durable`). Важно отметить, что попытки объявления существующей очереди с отличными параметрами будут вызывать ошибки, поэтому стоит либо удалить старую очередь, либо дать ей другое имя.

```
channel.queue_declare(queue=queue_name, durable=True)
```

Перейдём к сохранению сообщений. Для этого необходимо при их отправке добавить свойство, с указанием режима.

```
channel.basic_publish(
    exchange='',
    routing_key=queue_name,
    body=message_text,
    properties=
        pika.BasicProperties(delivery_mode=pika.spec.PERSISTENT_DELIVERY_MODE)
)
```

Настройка подтверждения доставки сообщений в конце обработки нам была необходима для того, чтобы мы могли распределять сообщения по потребителям с учётом их занятости.

RabbitMQ старается распределить все сообщения по потребителям в тот момент, когда они поступают в очередь. Сейчас он не учитывает, сколько неподтверждённых сообщений есть у поставщика.

Мы можем настраивать максимальное количество неподтверждённых сообщений, которые могут быть переданы одному потребителю. Для этого достаточно добавить одну строчку в потребителе.

```
channel.basic_qos(prefetch_count=1)
```

Проверка работоспособности

Для проверки написанного кода потребуется запустить несколько потребителей сразу, а затем отправить подряд одно сообщение, которое будет обрабатываться долго и ещё несколько маленьких. Это позволит увидеть, что один из потребителей получит несколько сообщений подряд, потому что второй будет занят.

3. Отправка сообщений в несколько очередей сразу

В предыдущем примере мы рассмотрели общение, основная задача которого распараллелить выполнение одинаковых задач между подписчиками. Сейчас же мы займёмся рассылкой сообщений сразу нескольким потребителям одновременно.

В этом примере мы реализуем паттерн publish/subscribe. Поставщик будет публиковать сообщения, а потребители будут подписываться на получение всех сообщений.

Для реализации такой системы понадобится разобраться в работе обменников (exchange).

Поставщик не занимается отправкой сообщений в очередь на прямую. На самом деле между ним и очередью всегда есть обменник, который занимается распределением сообщений по очередям.

Именно обменник хранит в себе правила маршрутизации сообщений. При получении нового сообщения он самостоятельно решает в какие очереди необходимо перенаправить сообщение.

Существует несколько типов обменников. В этом примере мы рассмотрим работу с типом `fanout`. Данный тип обменников весьма прост. Он просто отправляет сообщения во все доступные очереди. Этот тип обменников понадобится для реализации системы логирования, представленной в этом примере. Ниже представлена схема взаимодействия, которую мы реализуем (Рисунок 5.5).

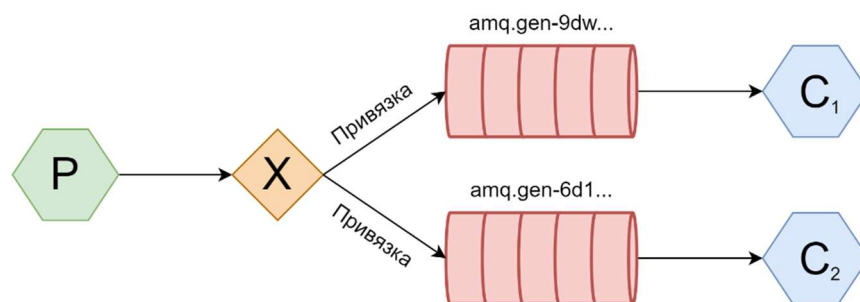


Рисунок 5.5 — Пример использования обменника типа `fanout`

Настройка обменника

В этом примере мы не будем определять очереди со стороны поставщика, так как ни источник, ни обменник не могут знать заранее, кто будет получать сообщения. Более того в этом примере нас будут слабо интересовать названия очередей, поэтому позволим RabbitMQ сгенерировать их самостоятельно.

Для того чтобы использовать собственный обменник, необходимо его объявить в коде поставщика. При объявлении надо указать его имя, по которому мы сможем его идентифицировать, а также тип `fanout`.

```
exchange = channel.exchange_declare(exchange='logs', exchange_type='fanout')
```

Далее при отправке сообщения нам необходимо указывать имя обменника, в который хотим отправить сообщение. Так, мы будем использовать не обменник по умолчанию, а тот, который был настроен.

```
channel.basic_publish(
```

```
exchange='logs',  
routing_key='',  
body=message_text  
)
```

Теперь все наши сообщения будут отсылаться в настроенный обменник.

Работа с очередями

Как было описано ранее, в этом примере мы не будем создавать именованные очереди, так как мы всё равно будем отправлять сообщения во все очереди.

Это значит, что каждый раз, когда потребителю надо будет подключиться к RabbitMQ для получения сообщений, которые раскидывает обменник, у нас будет создаваться новая временная очередь. После отключения потребителя очередь будет удалена.

```
queue = channel.queue_declare(queue='', exclusive=True)
```

После создания очереди необходимо привязать её к обменнику, чтобы он мог отправлять в неё сообщения. Для этого нам необходимы имена обменника и очереди.

```
queue_name = queue.method.queue  
exchange_name = 'logs'  
channel.queue_bind(exchange=exchange_name, queue=queue_name)
```

В настройках приёма сообщений также необходимо поменять имя очереди на сгенерированное RabbitMQ.

Проверка работоспособности

Для проверки написанного кода потребуется запустить несколько потребителей сразу. Вслед за этим можно отправлять сообщения с помощью поставщика. Отправленные сообщения будут получены всеми потребителями. Если проследить в графическом интерфейсе RabbitMQ за тем, как меняется набор очередей в зависимости от того, сколько

потребителей работает, можно заметить, что очереди удаляются сразу после отключения потребителя.

4. Использование ключа маршрутизации

В прошлом примере мы сделали небольшую систему логирования. Достаточно часто у нас есть разные уровни логов: информация, предупреждения, ошибки и т.д. Изменим в этом примере код так, чтобы потребители могли получать только те логи, которые им необходимы. Ниже представлена схема взаимодействия, которую мы реализуем (Рисунок 5.7).

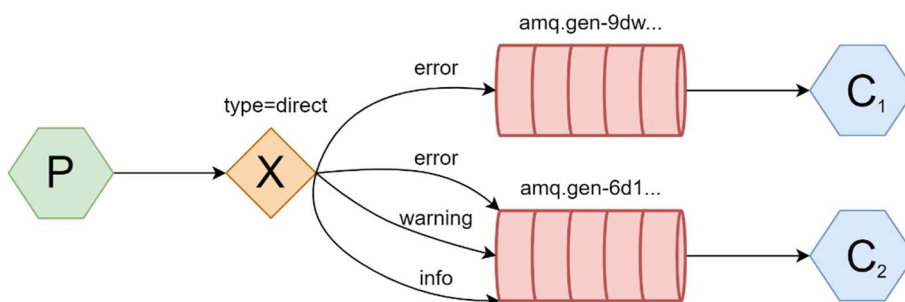


Рисунок 5.7 — Пример использования обменника типа `direct`

Настройка обменника

В этом случае тип обменника `fanout` нам уже не подойдёт, так как он без разбора шлёт сообщения во все очереди. Для решения нашей задачи существует другой тип обменника `direct`.

```
exchange = channel.exchange_declare(exchange='logs', exchange_type='direct')
```

Для использования данного типа обменника нам понадобятся ключи маршрутизации, которые позволяют настроить правила отправки сообщений по очередям более гибко. Ключ маршрутизации (`routing_key`) относится к связи (`binding`) между очередью и обменником и задаётся на этапе её создания.

```
channel.queue_bind(exchange='logs', queue=queue_name, routing_key='info')
```

Создав связи с ключом маршрутизации, мы указали обменнику, какие сообщения хотим от него получать. Теперь надо только указать необходимый

ключ маршрутизации при отправке сообщения поставщиком. Тогда в случае, если ключи отправителя и связки совпадут, сообщение будет отправлено в очередь.

```
channel.basic_publish(  
    exchange=exchange_name,  
    routing_key='info',  
    body=message_text  
)
```

Осталось добавить немного кода для того, чтоб можно было регулировать из консоли ключи маршрутизации и система будет готова.

Проверка работоспособности

Для проверки написанного кода потребуется запустить несколько потребителей сразу. Отправляя сообщения с разными ключами маршрутизации, можно заметить, что получатели принимают и обрабатывают лишь те, чей ключ маршрутизации совпадает с ключом используемых связей.

5. Использование тематик для маршрутизации сообщений

В последнем примере мы рассмотрим самый гибкий способ маршрутизации сообщений с помощью RabbitMQ.

Здесь мы будем использовать ещё один тип обменника `topic`. Данный тип обменника позволяет задавать правила маршрутизации сразу по нескольким критериям.

Здесь будут минимальные изменения в коде. Нам только необходимо поменять тип обменника на `topic`, что мы уже делали ранее. Сообщения также будут направляться различным потребителям в соответствии с `routing_key`.

Для ключа маршрутизации нового типа обменника существуют некоторые правила.

Во-первых, ключ маршрутизации представляет собой набор слов, разделённых символом точки. Слова могут использоваться совершенно

любые. Обычно их подбирают таким образом, чтобы они описывали какие-то признаки сообщения. Логика проверки ключа маршрутизации такая же, как и в прошлом примере с небольшой разницей: мы проверяем на соответствие не весь ключ сразу, а последовательно сравниваем соответствующие слова.

Примеры ключей: `russian.male.brown: <гражданство>.<пол>.<цвет волос>`,
`stock.etf.usa.usd: <класс активов>.<вид>.<рынок>.<валюта>`.

Во-вторых, в ключе маршрутизации могут быть использованы специальные символы. С помощью `*` можно заменить ровно одно слово в ключе, а с помощью `#` можно обозначить любое количество слов, начиная с нуля.

Рассмотрим, как это работает, на примере (Рисунок 5.9). В примере будем работать с биржевыми активами. Выделим несколько свойств, которые могут быть у сообщений: класс (акции, облигации), рынок (США, Россия), валюта (доллар, рубль). В общем виде ключ маршрутизации будет выглядеть так: `<класс активов>.<рынок>.<валюта>`. Составим схему взаимодействия сервисов.

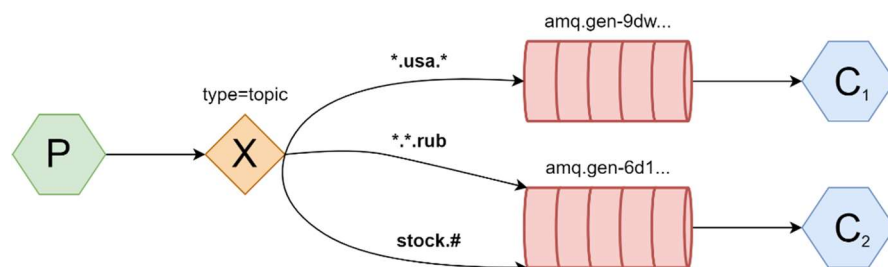


Рисунок 5.9 — Пример использования обменника типа `topic`

На схеме выше представлены три связи с ключами маршрутизации.

Рассмотрим каждую из связей. По первому маршруту будут отправлены сообщения, где второе слово из трёх будет соответствовать `usa`. Значения остальных слов нас не интересуют. По центральному маршруту мы отправим лишь те сообщения, где третье слово будет `rub`. Последний маршрут примет в себя все акции (`stock`). При этом сообщение может состоять хоть из 25 слов, но, если первое будет `stock`, оно будет отправлено в очередь.

Для того чтобы реализовать данный пример, достаточно в коде 4 части данной работы поменять тип обменника, а также ключи маршрутизации. В остальном всё останется без изменений.

Задание на самостоятельную работу

Необходимо выполнить самостоятельно ряд заданий, имеющих варианты. Варианты считаются в виде остатка от деления последней цифры билета на количество вариантов.

Для выполнения практических работ можно использовать развёрнутый учебный сервер с RabbitMQ (хост: 51.250.26.59, порты: 5672 и 15672, логин: guest, пароль: guest123)

Задание 1. Реализовать схему простейшего взаимодействия сервисов (Рисунок 5.2). Название очереди должно иметь формат <группа>_<фамилия> (inbo-04_laptev).

1. Создать эксклюзивную очередь.
2. Создать очередь, сохраняемую при перезапуске сервера RabbitMQ.
3. Создать автоудаляемую очередь.

Задание 2. Реализовать схему взаимодействия сервисов с нагрузкой.

1. Символ для обозначения времени сна: #. Тип обменника: fanout.
Сообщения должны храниться при выключении RabbitMQ.
2. Символ для обозначения времени сна: *. Тип обменника: direct.
Сообщения могут не храниться при выключении RabbitMQ.
3. Символ для обозначения времени сна: -. Тип обменника: topic.
Сообщения должны храниться при выключении RabbitMQ.