Geekbrains

**Исследование особенностей разработки небольших масштабируемых приложений с использованием микросервисной архитектуры**

Программа: Разработчик - Программист

Специализация: Веб-разработка на JAVA

Зотов Алексей Викторович

Москва

2024

**Содержание**

[Содержание](#_heading=h.7mt9x4b37ce)

[Введение](#_heading=h.1fob9te)

[Теоретическая и практическая главы](#_heading=h.hst8x8f1r8tz)

[Заключение](#_heading=h.dhwmmf3wq1kk)

[Список используемой литературы](#_heading=h.m8jvbwhv7ujg)

[Приложения](#_heading=h.s6k7t2lwcp0k)

# Как оформить содержание?

В содержании дипломного проекта необходимо озаглавить основные блоки и подпункты работы с указанием страниц.

### Пример

*Данный пример не является эталонным, лишь один из вариаций.*

Введение (2–3 стр.)

Глава 1. Основы ручного тестирования веб-приложений (~15 стр.)

1.1 Что такое веб-приложение, его особенности

1.2 Что такое тестирование, кто такой тестировщик. Зачем нужно тестирование и когда оно начинается

1.3 Этапы ручного тестирования веб-приложений

1.4 Источники требований: что такое техническое задание и как проводить ручное тестирование, если техническое задание отсутствует

Глава 2. Подготовка к ручному тестированию веб-приложения (~15 стр)

2.1 Составление чек-листов

2.2 Составление тест-кейсов

2.3 Разработка тест-плана

2.4 Тест-дизайн и тест-аналитика

2.5 Баг-репорты и багтрекинговая система

Глава 3. Проведение тестирования и разработка предложений по улучшению тестирования Интернет-магазина РИВ ГОШ (~20 стр.)

3.1 Тестирование функциональности

3.2 Тестирование юзабилити

3.3 Тестирование интерфейса

3.4 Тестирование совместимости

3.5 Тестирование производительности

3.6 Тестирование безопасности

3.7 Подведение итогов. Разработка предложений по улучшению ручного тестирования веб-приложения РИВ ГОШ

Заключение (~ 4 стр.)

Список используемой литературы

Приложения

# Введение

**Тема проекта:** Исследование особенностей разработки и развертывания небольших масштабируемых приложений с использованием микросервисной архитектуры

**Цель:** Рассмотреть основные подходы к дизайну и развертыванию приложений и используемому доступному инструментарию. Рассмотреть механизмы развертывания приложений. Создать шаблон приложения, предусматривающий легкую адаптацию к конкретным нуждам разработчиков при развертывании небольших сервисов (pet projects)

**Какие задачи решает:** В процессе обучения или же в процессе разработки отдельных микросервисов для небольших стартапов, частных проектов и тп. может возникать потребность протестировать или проверить взаимодействие своего разработанного микросервиса с другими сервисами именно на этапе разработки, выбрать те или иные библиотеки и определится с инструментарием. Задача, разработать приложение, позволяющее «подменять» один из сервисов или добавлять в один из сервисов интересующую разработчика логику и смотреть за ее работой, без необходимости создавать такое приложение с нуля, снабжать его необходимым инструментарием. Предполагаемое к разработке приложение планируется подготовить максимально абстрактным с целью легкой подмены шаблонных сервисов на кастомные. Приложение должно содержать базовый набор инструментов для отслеживания (трейсинга), логирования. К приложению должен быть подключен балансировщик нагрузок. Приложение должно реализовывать 2 механизма коммуникации: синхронный режим и асинхронный режим. Взаимодействие между сервисами в асинхронном режиме должно быть реализовано посредством брокера сообщений.

**Задачи:**

1. Изучение литературы
2. Подготовить схему приложения и взаимодействие сервисов
3. Разработать первичную архитектуру приложения
4. Внедрить базовый инструментарий
5. Внедрить асинхронную коммуникацию между двумя сервисами с помощью брокера сообщений
6. Изучить процесс развертывания приложений

**Инструменты:** Postman, FastRequest, PostgreSQL, Docker, Docker-Compose, Spring, Eureka, Open Feign, API Gateway, RabbitMQ, Micrometer, Zipkin, Jib

Состав команды: Зотов Алексей Викторович (программист)

# **Глава 1. Теоретическая часть**

Добавить общие слова по по микросервисам

Данная работа и данное исследование в первую очередь направлено на разработку небольшого микросервисного приложения для небольшой аудитории «энтузиастов» которые на своем опыте и самостоятельно как разрабатывают собственные микросервисы так и изучают доступные инструментарии.

В процессе собственного обучения, захотелось создать небольшое приложение, которое бы моделировало основные принципы взаимодействия между сервисами. Развивая эту мысль дальше, создав такое приложение, можно было бы продумать дальнейшие шаги по его усовершенствованию. Как например, создать возможность подменять сами сервисы исходя из потребностей конкретного пользователя. И далее уже предусмотреть автоматизацию, нагрузочное тестирование, frontend и так далее.

Исходя из вышесказанного появилось понимание, что конкретно необходимо сделать в первую очередь, на чем сфокусироваться - разработать микросервисное приложение, состоящие из трех сервисов. Сервис эмулирующий сохранение неких «покупателей» в собственной базе данных, Сервиса проверки данных «покупателей» и Сервиса уведомлений. Причем сервис проверки должен работать синхронно с сервисом регистрации и сохранять собственную информацию в собственной базе данных, сервис уведомлений же, наооборот спроектировать с асинхронной коммуникацией. Как итог мы получим одного «клиента» и двух «подписчиков», один из которых будет обмениваться информацией с «клиентом» синхронно эмулируя работу, от которой зависит то или иное дальнейшее действие «клиента», второй же будет работать асинхронно эмулируя работу условного сервиса уведомлений, которые не обязательно должны приходить в момент поступления запроса. Более того, каждый из сервисов, независимо, будет работать с собственной базой данных.

Поскольку в процессе разработки планируется не только создать конечную рабочую версию приложения, но и посмотреть различные инструменты и подходы, разработка будет вестись поэтапно. Основные этапы будут фиксироваться в виде коммитов в репозитории на [GitHub](https://github.com/).

Первым этапом будет создание двух микросервисов и рассмотреть взаимодействие между ними с использованием RestTemplate. Далее внедрить Eureka. По сути посмотреть работу «записной книжки».

Двигаясь далее затронуть такой сервис как Open Feign и на этом же этапе внедрить третий сервис, сервис трассировки. Этот этап подразумевает внедрение в проект таких технологий как Micrometer и Zipkin.

Поскольку данное приложение планируется использовать и для демонстрации работы микросервисов, считаю, что функционал отслеживания взаимодействий между микросервисами, включая и скорость ответа, это критически важный элемент для всей дальнейшей разработки.

Логичным продолжением по наполнению проекта инструментарием станет внедрение API Gateway и детальное рассмотрение балансировщика нагрузок. Мы понимаем, что при обращении внешнего клиента к нашему сервису появится базовая нагрузка на наш внутренний микросервис регистрации. При кратном увеличении нагрузки на этот сервис, время отклика будет увеличиваться, что в конечном счете может привести к нежелательным последствиям. В нашу же задачу входит создание проекта в котором можно попробовать собсвенный сервис в условиях, близких к реальным, а балансировщик нагрузки позволит эмулировать распределение нагрузки на несколько образов сервиса утилитами по типу Apache Jmeter.

Основная задача данной работы это:

* Спроектировать 3 микросервиса
* Внедрить необходимый инструментарий
* Внедрить брокер сообщений
* Рассмотреть подходы к развертыванию приложения с помощью Docker

# **Глава 2. Разработка приложения**

**2.1 Подготовка базовой архитектуры приложения**

Разработка приложения будет вестись с использованием IntellyJ IDEA Community edition с использованием Maven framework.

Приложение будет разрабатываться как мультимодульное, в составе которого будет «родительский» pom.xml файл с основными зависимостями. Сами сервисы будут включены в проект в качестве отдельных модулей. Каждый из модулей будет включать в себя как собственные, необходимые зависимости, так и «родительские» зависимости. Проект будет обновляться и совершенствоваться в процессе разработки, поэтому было принято решение создать пустой Maven проект без использования “spring initializr” – [https://start.spring.io](https://start.spring.io/)

**2.1.1 Развертывание базового приложения и настройка pom.xml**

В папке с проектом через консольную команду создаем базовый пустой maven проект (рис. 1).

Рисунок 1

Далее настраиваем проект. Версия Java: Oracle open jdk 17.0.10. Поскольку данный проект будет являться «родительским» для остальных модулей, удаляем папку src, а также добавляем 2 собственные папки к проекту а именно: папка с проектом диплома и папка с диаграммами.

«Родительский» pom файл настраиваем с использованием dependency Management блока в который подключим артефакт spring dependency и зафиксируем версию 3.2.5 на уровне свойств. Основная задача, фиксирование версий используемых зависимостей и внедрение на уровень проекта основных зависимостей, который абсолютно точно понадобятся во всех дочерних модулях. Для удобства работы подключим на уровень проекта Lombok и spring boot test. В подраздел plugin подключим арефакт maven-plugin.

На этом, этап подготовки базового проекта завершен, собираем pom файл, подгружаем зависимости и размещаем первый коммит на GitHub -[*Initial commit*](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/df8f748734800529b78367ae7ebf1ee3d667ce10)

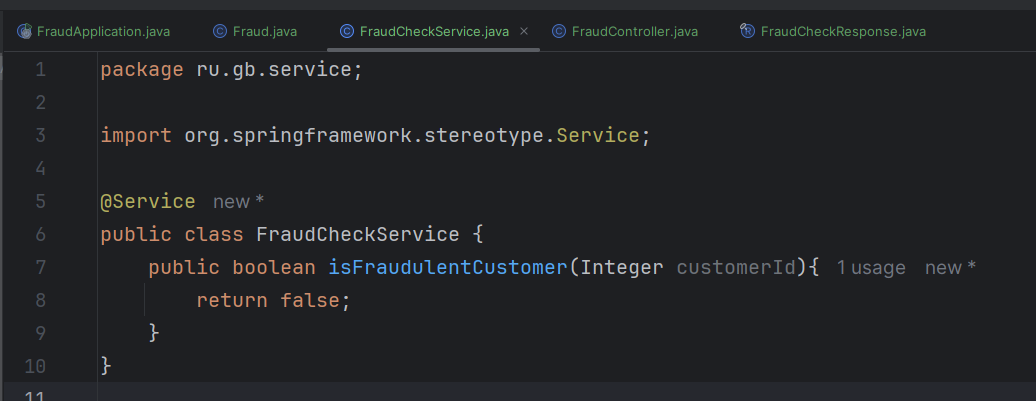
**2.1.2 Создание сервисных приложений**

Как упоминалось в первой главе, дизайн приложения будет состоять из 3 сервисов и ряда сопутствующих модулей, с минимальным наполнением самих сервисов, предполагая что эти сервисы будут использоваться как шаблоны для тестирования или подмены необходимыми разработчику сервисами. Для эмуляции каждый из сервисов будет работать с собственной базой данных развернутой на хостовой машине в docker контейнере

Первым сервисом будет сервис *customer-service*. Мы хотим использовать базовое наполнение самого сервиса и реализовать базовый набор функционала для работы с репозиторием развернутом на PostgreSQL, с JPA драйвером, а также контроллер. Для этого мы добавим новый модуль в проект. В зависимости мы добавляем Spring web. Для удобства идентификации запущенных сервисов, добавим в ресурсы проекта файл *banner.txt* с названием сервиса, а также файл *application.yml* с базовыми настройками названия проекта, портом, в дальнейшем же добавим сюда настройки для работы сервиса с собственной базой данных.

Второй сервис, который мы также добавим в проект *fraud-service* будет максимально похож на первый сервис. Более того тут важно подчеркнуть что оба эти сервиса будут зависеть друг от друга и будут взаимодействовать только синхронно. Первый сервис будет регистрировать покупателей в собственной базе данных и направлять запрос во второй сервис, который будет имитировать проверку, является ли добавленный пользователь «настоящим» или валидным, либо это «мошенник» и дополнительно фиксировать данные в своей базе данных.

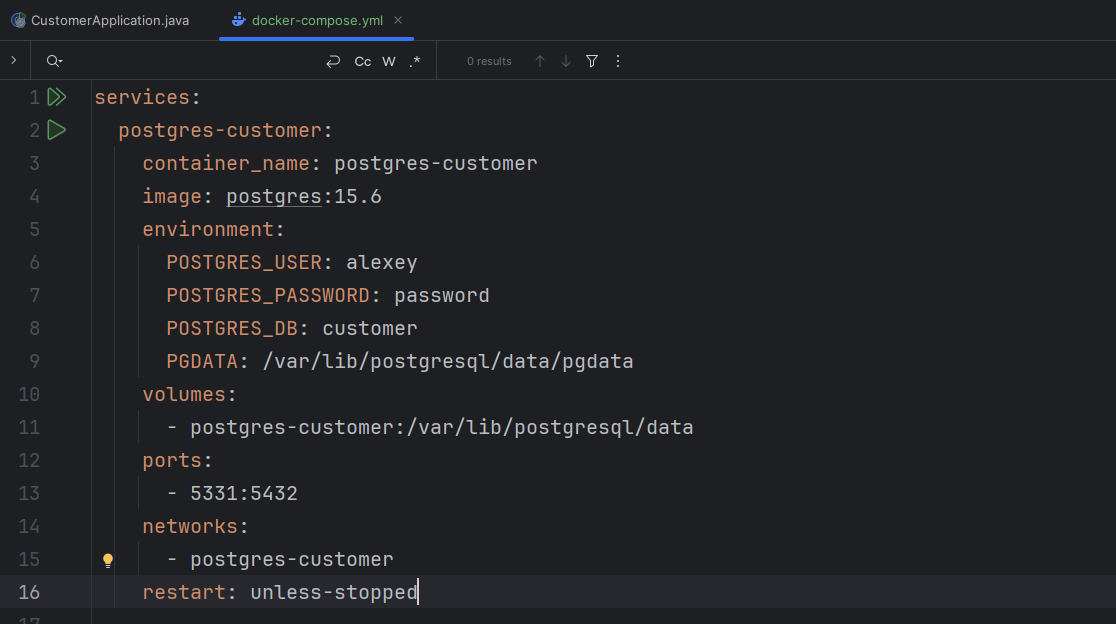
Поскольку данная работа в первую очередь направленна не на обработку персональных данных, а разработку взаимодействия между микросервисами, было решено максимально абстрагировать процесс проверки (рис. 2)

Рисунок 2

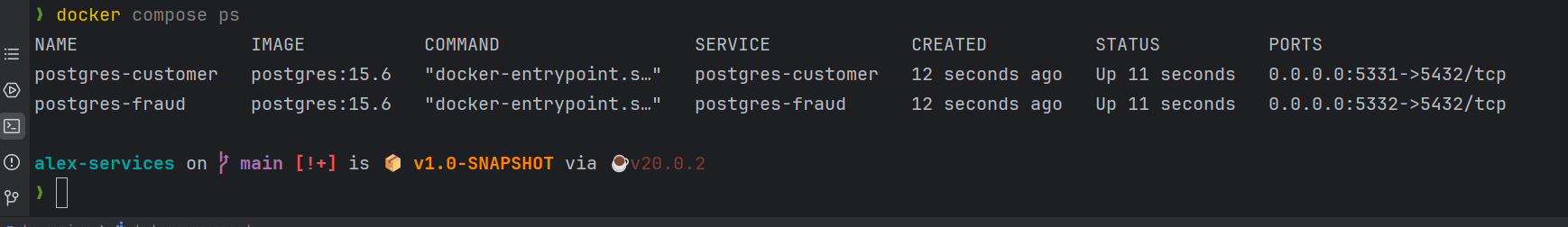
Помимо базовой настройки самих сервисов, мы добавим каждому микросервису отдельный порт, что бы избежать конфликтов. На этом этапе оба сервиса подготовлены для интегрирования с базами данных. [*Git Commit: 102d4d7*](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/102d4d7e9f4524be9ef9d2f30fff18dbf918f03c)

**2.1.3 Развертывание баз данных с использованием Docker и подключение сервисов**

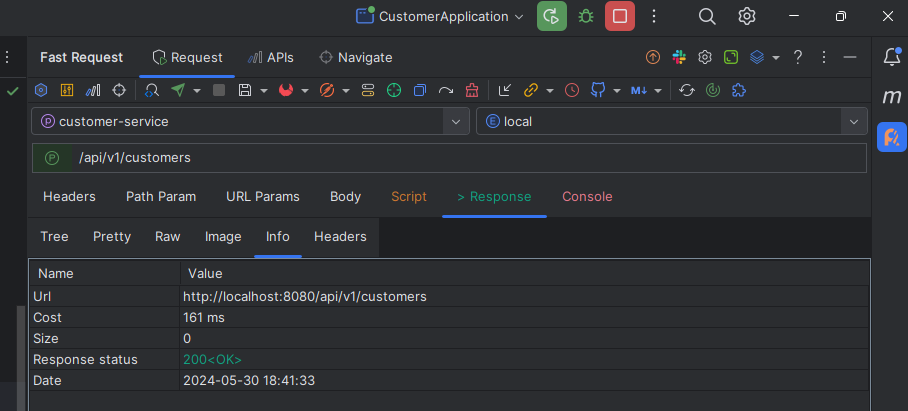
В корневом каталоге создадим файл docker-compose.yml в котором я настрою создание двух отдельных контейнеров для сервисов «customer» и «fraud». Версия PostgreSQL 15.6 (рис. 3)

Рисунок 3

Для каждого из сервисов настроим свой отдельный порт доступа. В целях демонстрации логин и пароль у всех образов будет один и тот же. Создадим 2 контейнера командой *docker compose up -d* и проверим что оба контейнера запущены (рис 4)

Рисунок 4

Для каждого из сервисом подключим зависимости *postgresql и JPA driver* и настроим оба файла *application.yml* и добавим отображение *sql* запросов строчкой *show-sql: true*, а также настроим оба *domain* класса для создания таблиц в базах данных с использованием структуры данных этих классов. Далее добавим в оба сервиса интерфейс репозитория. После успешного подключения двух сервисов к базам данных проверим работу контроллера (рис 5) и сохраним изменения в удаленном репозитории. GitHub коммит: - [d7f2b90](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/d7f2b909c288498db0ef10cd18431207a521adb8)

Рисунок 5

На данном этапе подготовка базовой архитектуры завершена. Поэтому можно внедрять Rest Template

**2.2 Rest Template**

**2.2.1 Настройка базовой коммуникации**

Первой задачей будет простой запрос с сервиса регистрации «покупателей», на проверку в сервис «мошенник» с прямым указанием портов каждого сервиса (диаграмма 1)

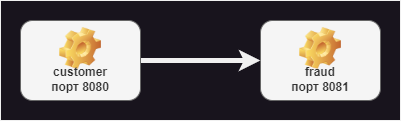
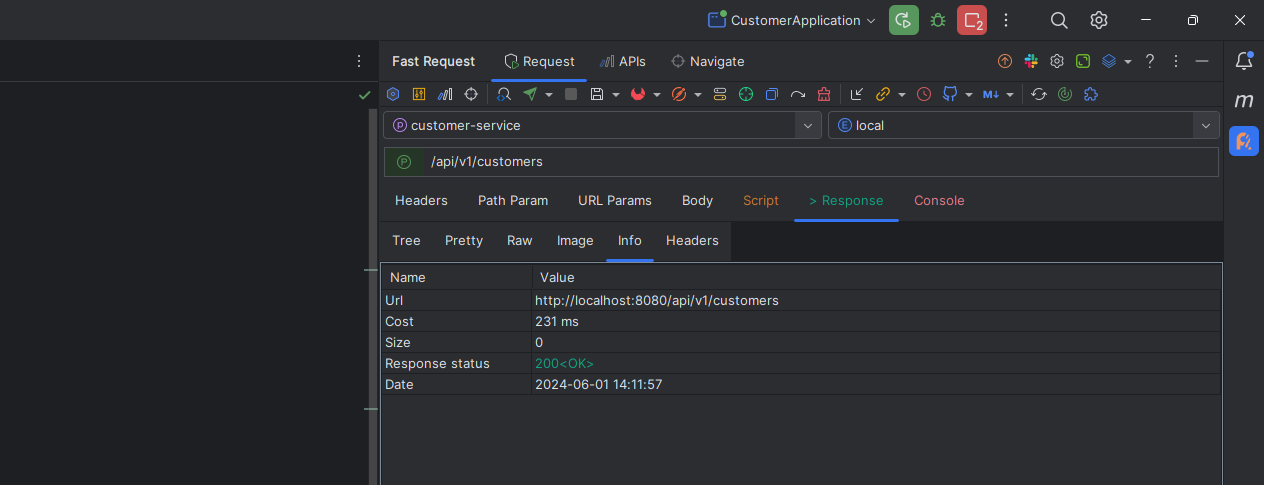


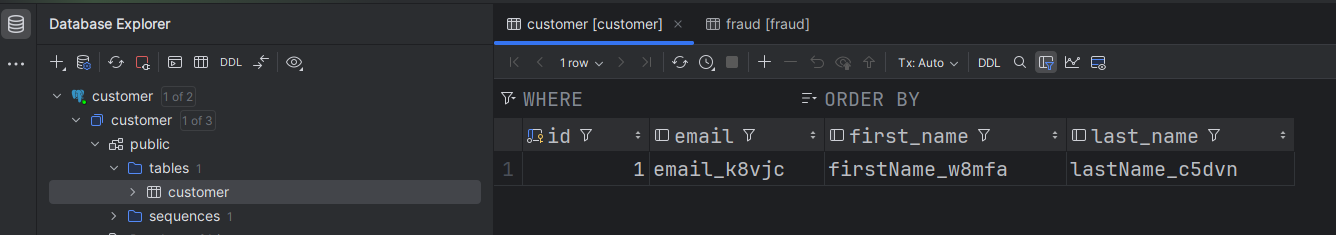
Диаграмма 1

В сервисе «покупатель» создадим новый конфигурационный класс и внутри класса создадим один простой метод, возвращающий RestTemplate объект. Дополнительных конфигураций, в этом методе мы не планируем, поскольку следующим этапом развития программы, будет внедрение более удобных сервисов. Данный шаг нам необходим для простой проверки работоспособности коммуникации между микросервисами.

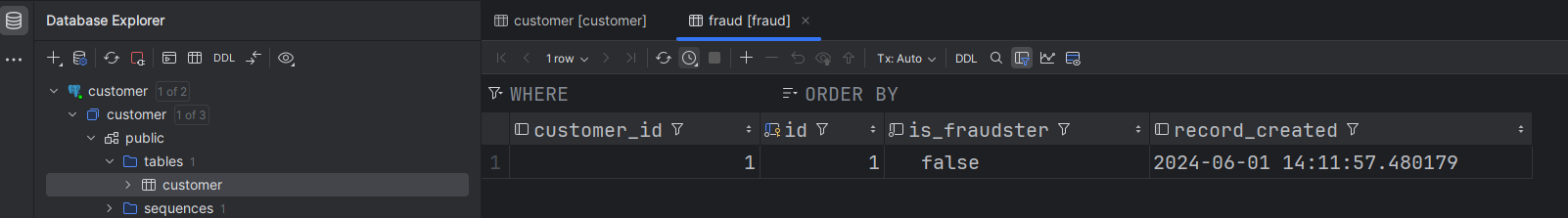
Далее в сервисе добавления нового «покупателя» мы обращаемся к RestTemplate через метод getForObject который направляет GET запрос на указанный URI и возвращает запрашиваемый объект. URI это полный путь к запросу GET fraud контроллера. Тут мы видим несколько неудобных моментов. Первое, это четкое указание порта в URI, что связывает наш сервис, более того, метод возвращает объект FraudCheckResponse, которого у нас нет в сервисе покупатель. Поскольку, задача стоит не только создать сервис, но и выяснить плюсы и минусы ряда подходов, временно добавим этот объект в сервис покупатель, далее же, решим эту проблему с внедрением отдельного модуля и Open Feign. Направим запрос на регистрацию нового «покупателя» и проверим прошел ли запрос и добавлены ли сущности в обе базы данных.

Рисунок 6. Получен ответ со кодом «200»

Для проверки баз данных, для удобства решил использовать DataGrip. Через эту IDE соединяемся с двумя базами данных и проверяем, добавленный ли данные.

Рисунок 7

Запись добавлена в базу данных «покупателей»

Рисунок 8

Также, зафиксирован результат проверки, не является ли «покупатель» «мошенником».

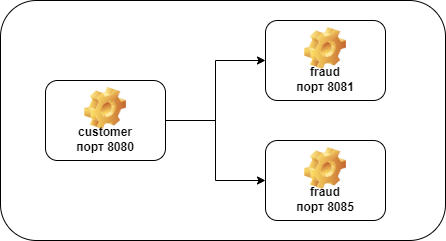
При использовании данного подхода выявлен ряд существенных недостатков, которые нам не позволят универсализировать программу, для подключения сторонних сервисов:

* Жесткая привязка к портам
* Добавление сторонних сущностей из других сервисов в сервис, который отправляет запрос и который не должен их содержать

GitHub: [commit 2f5655e](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/2f5655ef4d5fd644963fc43a74ef5e145dded7ef)

**2.2.2 Eureka server**

Предположим, что наш сервис проверки является ли «покупатель» «мошенником», перегружен и необходимо подключить второй такой же сервис, но имеющий другой порт.



Использовать разработанную архитектуру мы уже не сможем, поскольку мы отправляем запрос на порт а не по имени сервиса или сервисов. Поэтому первым шагом, добавим в проект «адресную книгу» - Eureka или сервис регистрации модулей (микросервисов).

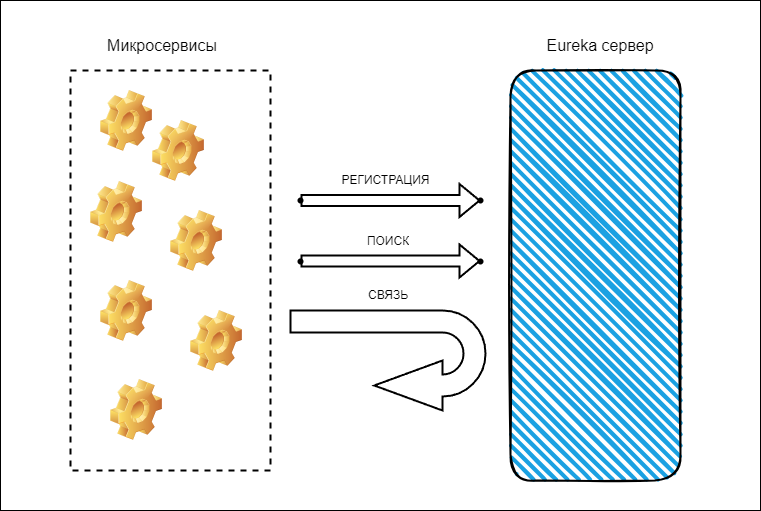
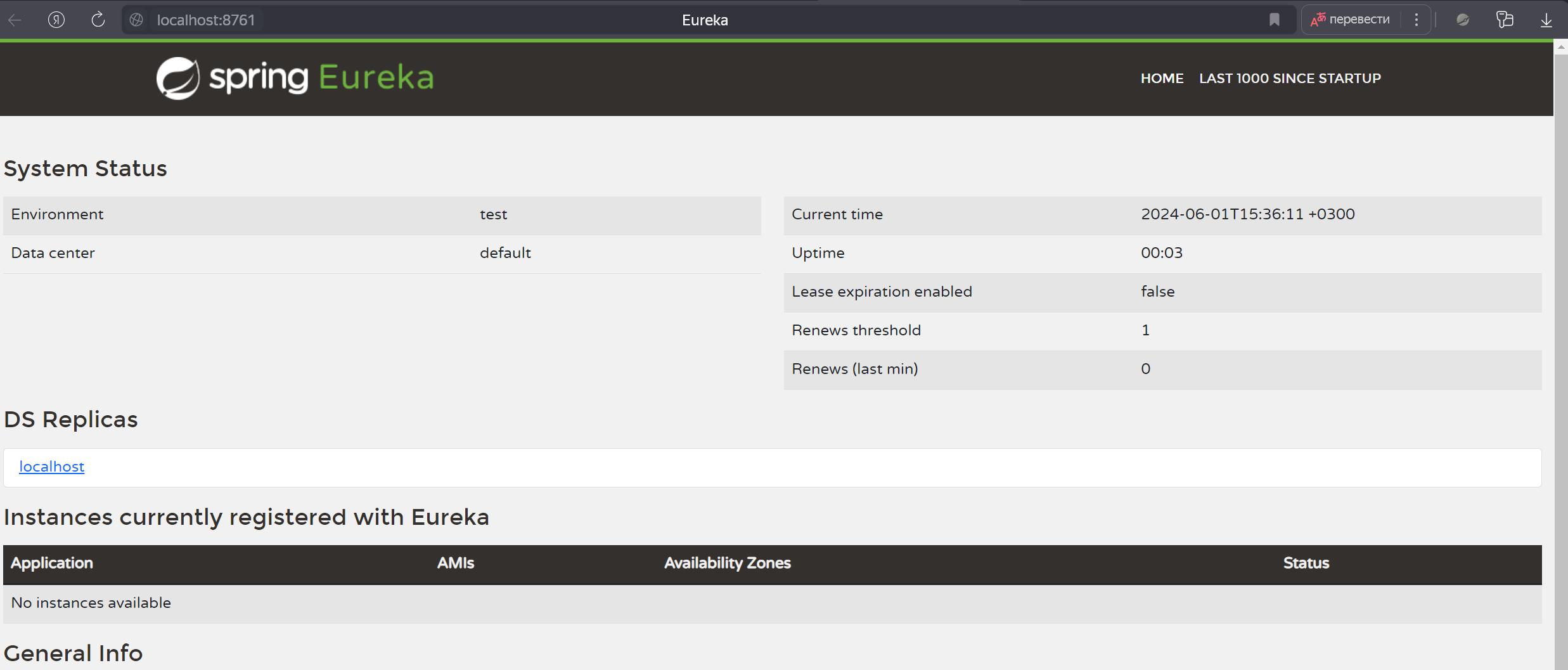


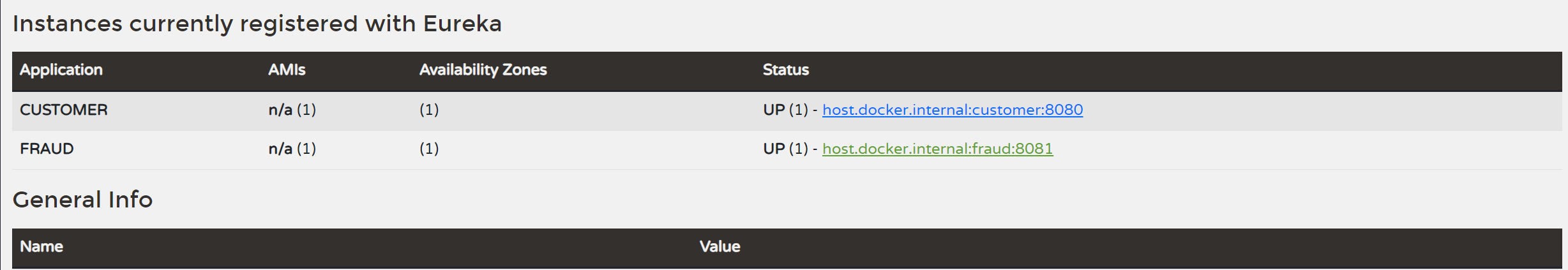
Рисунок 10

Механизм взаимодействия в Eureka довольно простой. Сервис «покупатель» и предположим 2 сервиса проверки должны зарегистрироваться в сервисе регистрации. Тогда каждый из сервисов сможет обратится к другому.

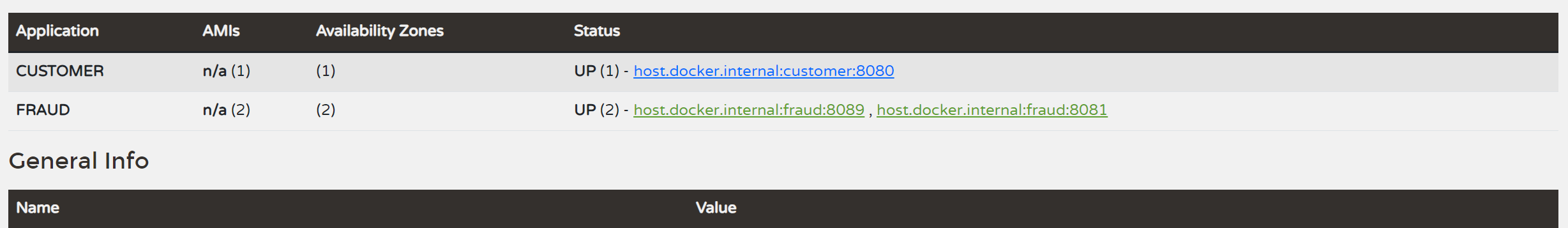
Для подключения Eureka сначала добавим в Dependency management pom файл проекта, зависимость Spring Cloud (2023.0.x). Добавим в проект новый модуль eureka-service. Сконфигурирем модуль как сервер, подключим зависимости, порт 8761 и запустим этот модуль. Через браузер по адресу localhost:8761 увидим панель управления сервиса eureka.

Рисунок 11

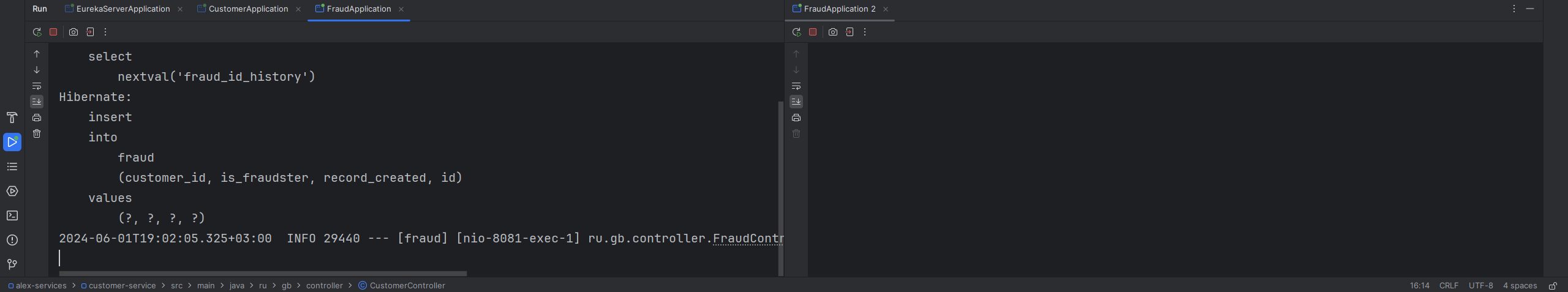
На рисунке 11 видно, что зарегистрированных сервисов пока нет. Поэтому сначала мы зарегистрируем 2 имеющихся микросервиса. Для этого в pom двух модулей добавим зависимости *spring-cloud-starter-netflix-eureka-client*, а также в файлах *application.yml* настроим регистрацию сервисов в eureka. После запуска двух сервисов можно увидеть 2 сервиса, зарегистрированных в eureka

Рисунок 12

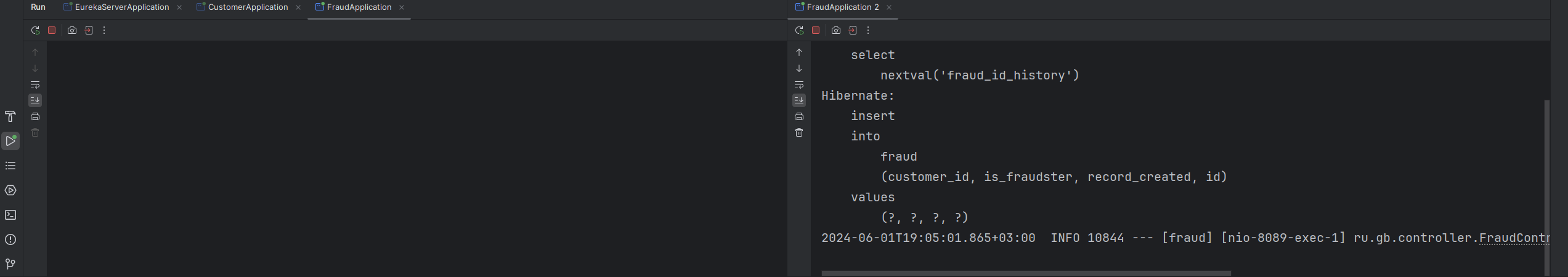
После того как сервисы зарегистрированы, можно изспользовать название сервера вместо указания порта в URI при обращении к сервису «fraud». Дополнительно сконфигурируем класс CustomerConfig и добавим аннотацию @LoadBalanced. Запустим сервис eureka, сервис customer и сервис fraud, сделаем имитацию работы 2 сервисов проверки и запустим сервис fraud второй раз на любом свободном порту.

Рисунок 13

На рисунке 13 видно что у нас запущено 2 сервиса fraud. Проверим работу балансировщика нагрузок. Отправим первый запрос на добавление нового «покупателя».

Рисунок 14

Как видно из рисунка 14, запрос пришел на 1 из двух сервисов «fraud». Это говорит о том что вся система работает и также задействован балансировщик нагрузок. Можно очистить лог консоли и направить повторный запрос.

Рисунок 15

Как видно из рисунка 15, повторный запрос был направлен на второй сервис. Это говорит о том что балансировщик нагрузок работает в режиме «Round Robin». Таким образом решили вопрос регистрации сервисов и связки сервисов через прямое указание сервисов. Это решение лишь частично помогает решить вопрос независимости сервисов между собой. Далее необходимо реализовать функционал open feign для того что бы еще больше изолировать сервисы друг от друга. Поскольку общая задача это разработка программы в которой можно было бы подключать независимые модули, то чем менее сервисы связаны между собой, тем проще будет подключать сторонние модули.

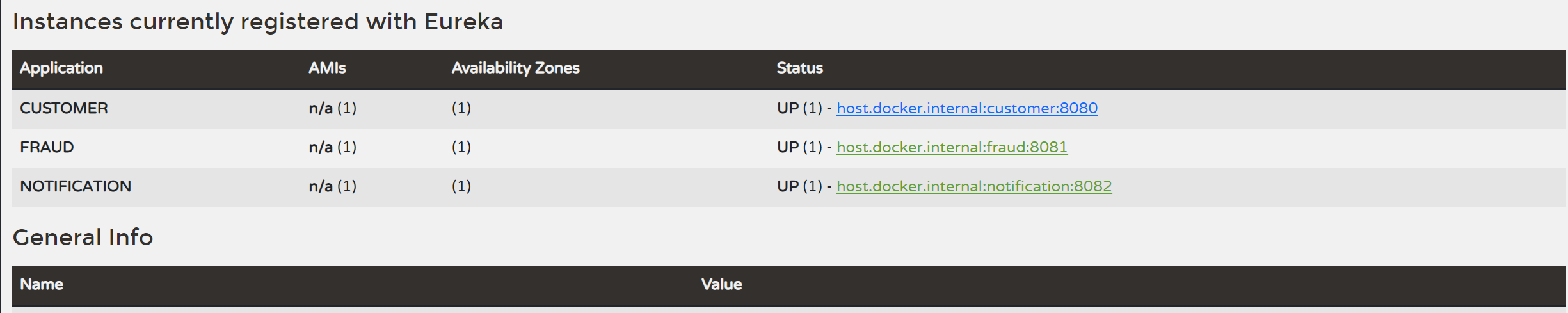
GitHub: [commit 273ac9f](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/273ac9f69b14bf482c1003c74ee6cfd33e232c40)

**2.2.3 Open Feign**

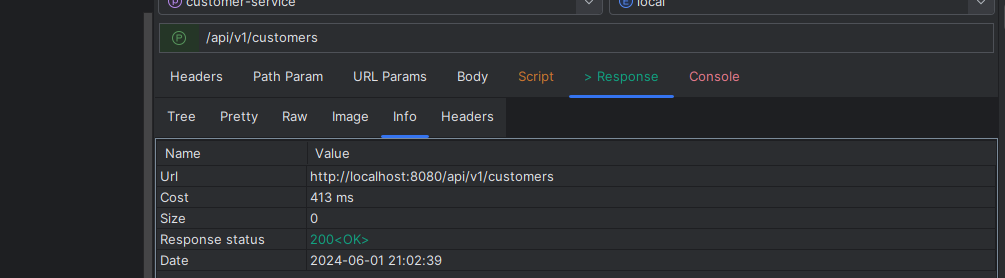
Оба разработанных микросервиса используют один и тот же объект «FraudCheckResponse» и реализуют ее внутри. Более того, необходимо постоянно помнить что в реализации метода registerCustomer класса CustomerService нам надо использовать сущность и если предположить что сервисов будет более чем 2 или 3, то нужно упростить работу с сервисами и вынести эти зависимости наружу сервисов. Для этого я решил разобраться с Open Feign и с его помощью сделать сервисы «слабо связанными».   
[Open Feign documents](https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/)

Добавим новый модуль в проект «clients-open-feign» в pom файл проекта добавим зависимость «spring-cloud-starter-openfeign». Сначала добавим интерфейс «Fraud Client». Скопируем методы (в случае этого приложения только один метод) из контроллера fraud в этот интерфейс с указанием полного URI. Таким образом разработанный интерфейс нацелен на Fraud Controller модуля fraud. Любой микросервис теперь может использовать этот интерфейс, если будет нужна функциональность fraud сервиса. Все что для этого нужно, это в pom файле сервиса в зависимостях прописать разработанный модуль clients-open-feign и в main классе включить open feign с указанием пути до модуля. Теперь можно перенести класс Fraud Check Response в новый модуль и удалить его в двух микросервисах customer и fraud. Теперь этот класс находится в единственном месте, что снижает сложность сервисов и количество зависимостей.

Теперь когда удалось снизить связанность между сервисами, можно добавить третий микросервис, сервис уведомлений. Как и планировалось, в дальнейшем данный сервис будет работать в асинхронном режиме обмена сообщениями. Но на данном этапе мы просто его добавим в наш проект и создадим по образу сервиса fraud и подключим к нему собственную базу данных и сконфигурируем в соответствии файл *application.yml* и *docker-compose.yml.* Запустим все три сервиса и проверим что все они видны в eureka server.

Рисунок 16

Все три сервиса удачно запущены. Добавим функционал обмена сообщениями в отдельный модуль clients-open-feign и проверим работу приложения. Теперь можно отправить запрос и получить ответ от сервиса fraud и сервиса notification.

Рисунок 17

Ответ получен, сервис удачно подключен и работает со своей отдельной базой данных. GitHub: [commit 0988519](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/098851947ca9171f56639ebc1d9a50701781d8b3)

**2.2.4 Трассировка микросервисов с использованием Micrometer и Zipkin**

Для отслеживания микросервисов, удобства работы, понимания связей между ними и получения базовой информации по времени ответа сервисов необходим инструментарий. Для этого мы будем использовать docker образ zipkin. В связи с региональными ограничениями, буду использовать docker «зеркало» для закачки образа. Для этого добавим ссылки на самые популярные зеркала в docker deamon.json

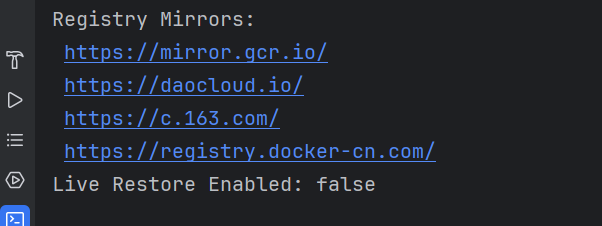
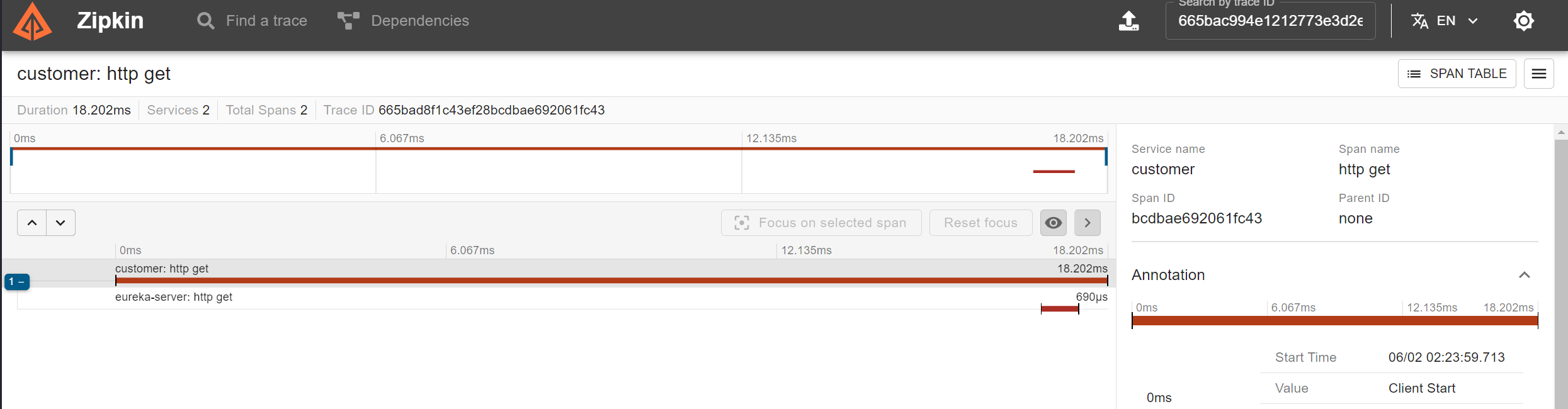


Рисунок 18

Далее добавим зависимости в наши сервисы для связки ziplin – micrometer. Zipkin UI находится по адресу: [http://127.0.0.1:9411](http://127.0.0.1:9411/) и имеет простой интерфейс. Основная задача это получить понимание о том как запрос проходит через все наши сервисы, время ответа, а также каким образом взаимосвязаны микросервисы между собой.

Рисунок 19

Поскольку с версии Spring Cloud Sleuth 3.1.x была произведена миграция на сервис micrometer, необходимо более детально рассмотреть работу micrometer: Основные зависимости, которые нужно добавить указаны на рисунке 20

Рисунок 20

Zipkin это инструмент, который помогает отслеживать запросы в разных частях распределенной системы. Это отображение пути, по которому запрос перемещается между различными микросервисами. Таким образом можно выявить узкие места в работе системы и работать над их оптимизацией.

Micrometer – это инструмент, для отслеживания работы микросервисов. Он делает это путем сбора и отображения данных о таких вещах как, время отклика, ошибки, количество выполняемых запросов и т. п. Используя Micrometer, разработчики могут получать представление о производительности своих микросервисов и принимать обоснованные решения об оптимизации своих систем. Библиотека позволяет легко отслеживать пользовательские показатели, предоставляя разработчикам больший контроль над мониторингом конкретных аспектов производительности их приложений.

С помощью зависимости micrometer-observation, можно собирать показатели из своего приложения и передавать их различным серверным системам мониторинга.

Зависимость ***micrometer-tracing-bridge-brave***используется для отслеживания приложения *Spring Boot*. Отслеживание - это способ увидеть, что произошло и сколько времени потребовалось для этого, например, при выполнении веб-запроса или обращении к базе данных. Отслеживание помогает отлаживать и оптимизировать производительность приложения. Зависимость micrometer-tracing-bridge-brave использует интерфейс трассировки *Micrometer* для использования *Brave,* решения для трассировки от *Zipkin.* Используя зависимость micrometer-tracing-bridge-brave, позволяет интегрировать приложение с Zipkin и видеть трассировки в веб-интерфейсе.

Зависимость ***zipkin-reporter-brave***используется для отправки трассировок, собранных с помощью зависимости micrometer-tracing-bridge-brave, на серверную часть *Zipkin*.

Чтобы использовать Micrometer в приложении *Spring Boot*, нужно добавить зависимость под названием *spring-boot-starter-actuator*. Эта зависимость предоставляет ряд функций мониторинга и управления микросервисами, таких как возможность предоставления метрик и информации о работоспособности.

Также, при использовании Open Feign вместе с *Zipkin* необходимо добавить зависимость *feign-micrometer*

Визуализировать работу трассировщика можно с использованием блок схемы с официального сайта разработчика.

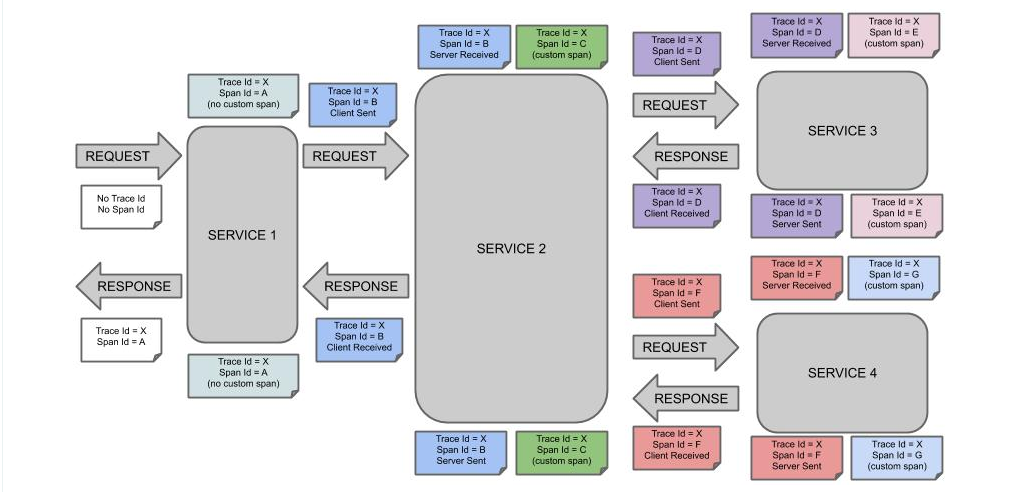


Рисунок 21

Особое внимание можно обратить на Trace Id, значение которого не меняется и который позволяет отследить всю «жизнь» запроса.

Еще одним инструментом который надо добавить будет API Gateway

**2.2.5 API Gateway**

Разрабатываемое приложение должно иметь понятную точку входа для внешних запросов. Согласно открытым источникам, для этого используются внешние балансировщики нагрузок от таких компаний как Amazone, Google, Yandex и так далее. Это отказоустойчивые решения, с механизмом распределения нагрузок. В рамках данной работы внешний балансировщик подключать не планируется.

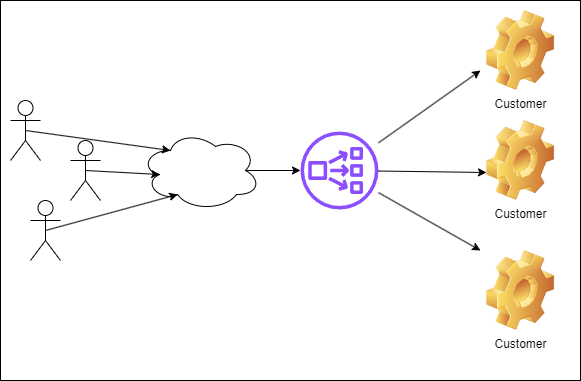
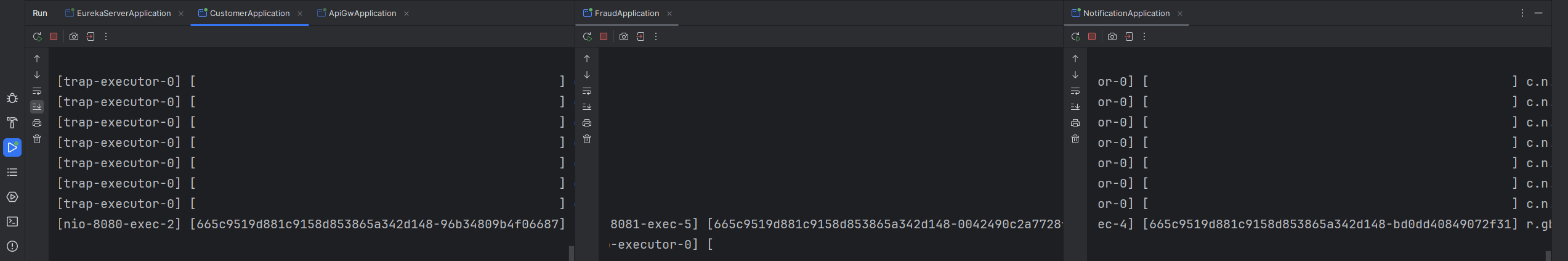
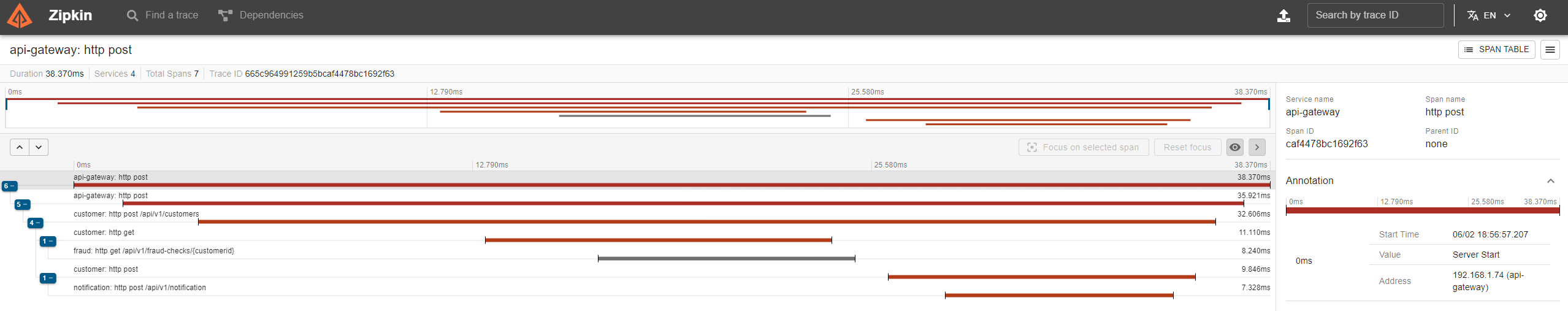


Рисунок 22

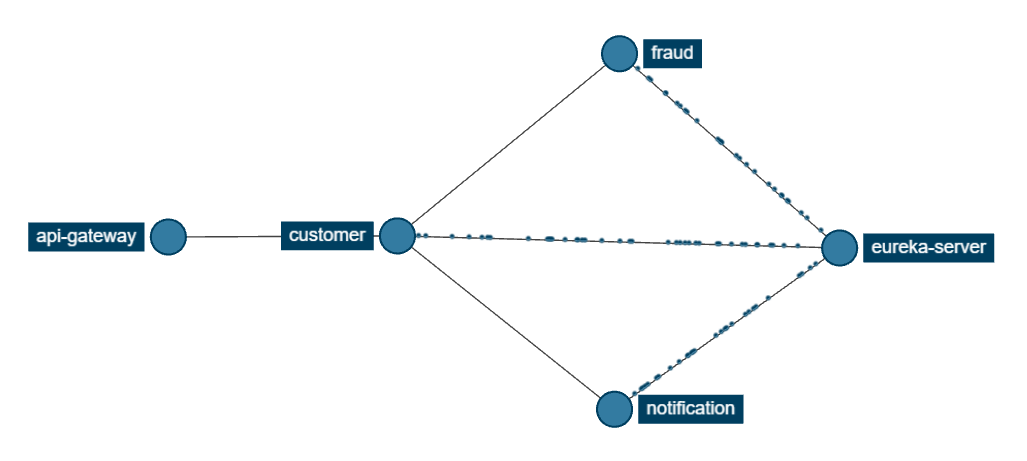
Но, будет подключен единый путь для добавления нового «покупателя» через библиотеку API Gateway. Что позволит обращаться к сервису по заданному нами порту. Для этого добавим в приложение новый модуль apigw. Пропишем путь к сервису регистрации «покупателей» через порт 8083. В дальнейшем все тестовые запросы через Postman будут отправляться именно по этому пути localhost:8083/api/v1/customers.   
Направим новый запрос на добавление нового покупателя.

Рисунок 23

Как видно из рисунка 23 Trace Id сохраняется одинаковым по движению запроса добавления нового «покупателя».

Рисунок 24

Как видно из рисунка 24 теперь все запросы стартуют с apigw сервиса. И после внедрения API Gateway схема взаимодействия между сервисами выглядит следующим образом

Рисунок 25

На этом настройка основных инструментов завершена.

GitHub: [commit 04f51d9](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/04f51d932a3b61b043b9c42588aa38e5563787ae)

**2.2.6 Брокер сообщений**

Поскольку в проект приложения был запланирован функционал взаимодействия между сервисами как в синхронном так и асинхронном режимах, необходимо подготовить приложение к внедрению асинхронного способа взаимодействия между сервисами. Планировалось, что сервис notification будет обрабатывать сообщения в асинхронном режиме. Задача внедрить AMPQ (advanced message queuing protocol), который позволить наладить коммуникацию между издателем и подписчиком посредством брокера сообщений, который будет выполнять функцию связующего звена. Поскольку издатель, подписчик и брокер используют сетевой протокол, они все могут находится на разных машинах. При работе брокера сообщений в асинхронном режиме, в случае задержки в работе сервиса сообщений, по разным причинам, брокер сообщений будет накапливать запросы и по мере восстановления работы, будет передавать их в сервис уведомлений, таким образом устойчивость системы повысится.

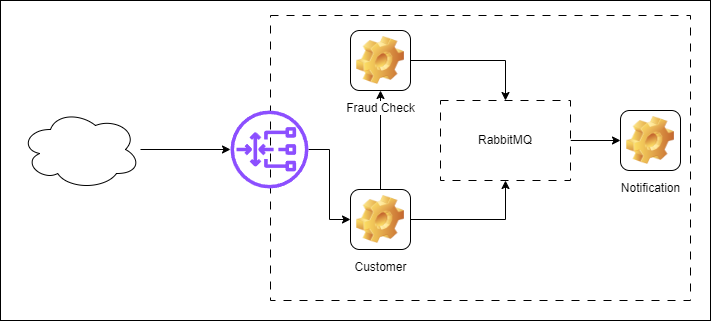


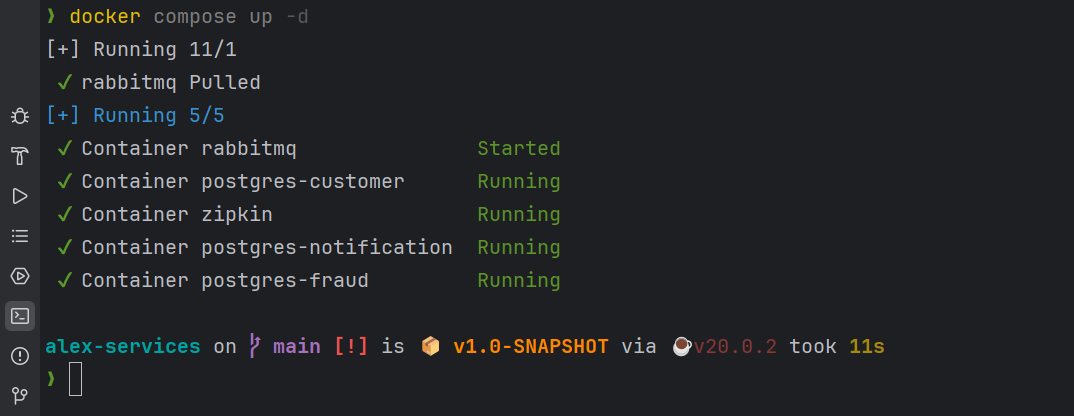
Рисунок 26

Как видно из рисунка 26, планируется использовать брокер сообщений RabbitMQ <https://www.rabbitmq.com/>. К основным преимуществам данного сервиса можно отнести следующее:

* Возможность работы в асинхронном режиме
* Снижение связанности сервисов
* Производительность
* Собственный UI

и так далее...

Для начала создадим docker контейнер с приложением. Добавим в файл docker-compose.yml образ rabbitmq:3.13-management-alpine. По указанным выше причинам, использую зеркало для закачки образа на локальную машину, указываем два порта 5672:5672 для обмена сообщениями и второй порт 15672:15672 для взаимодействия с UI RabbitMQ.

Рисунок 27

И далее заходим в графический интерфейс брокера. В приложение добавим новый модуль amqp (advanced message queuing protocol). В pom файл нового модуля добавим зависимость *spring-boot-starter-amqp.* Добавим базовый конфигурационный файл модуля где настроим работу AMPQ Template, конвертера сообщений, а также настроим конфигурацию «слушателя» сообщений. Дополнительно, сообщим приложению, что планируем использовать jackson object конвертер. Также необходимо добавить зависимости на новый модуль и библиотеку *spring-boot-starter-amqp* в сервис customer и notification, поскольку «покупатель» будет отправлять запрос на сообщение, а сервис сообщений будет их обрабатывать.

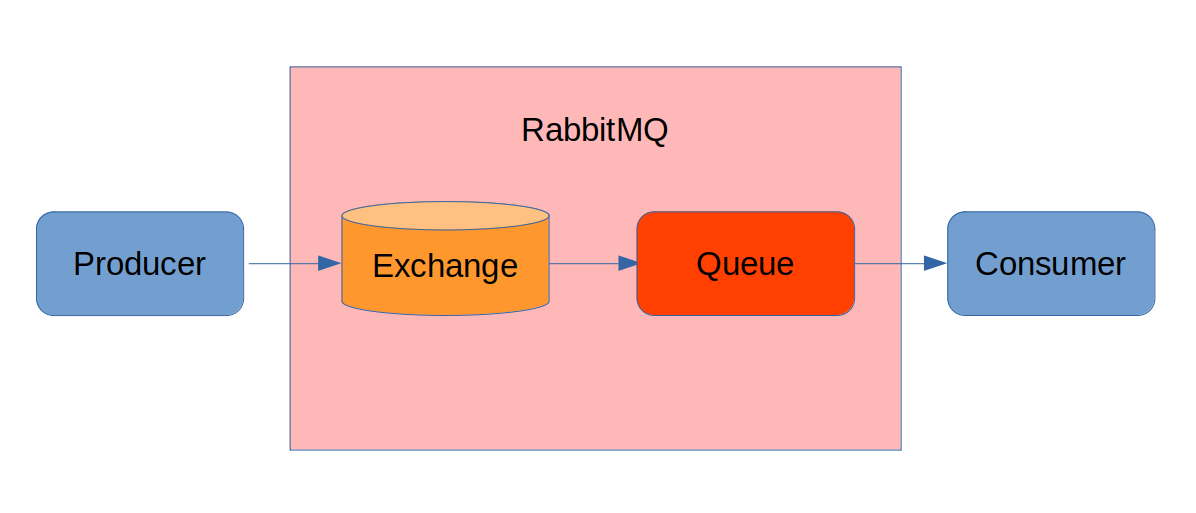
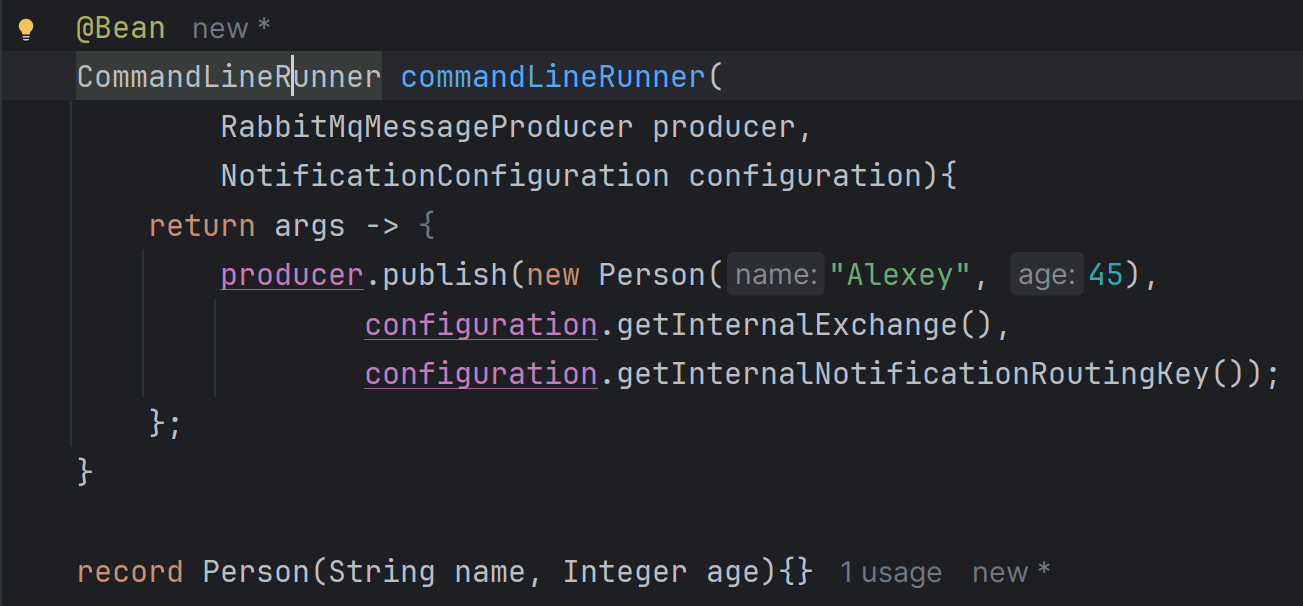
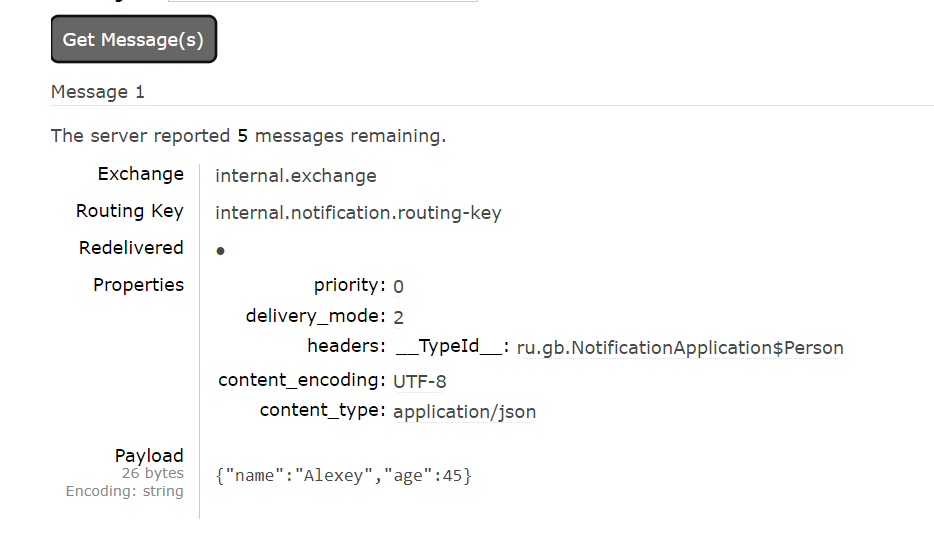


Рисунок 28. Источник — официальный сайт.

На рисунке 28 показана базовая архитектура работы брокера. Куда включены 2 модуля. Модуль обмена или распределения сообщений, а также «очередь», куда доставляются сообщения, для последующей пересылки на подписчика. К одному сервису обмена может подключено несколько очередей с сообщениями, причем каждая очередь должна быть привязана к своему подписчику. Далее создадим простой класс с методом публикации сообщений. На данном этапе можно проверить, работоспособность модуля распределения сообщений и добавляются ли сообщения в очередь. Для этого через интерфейс *CommandLineRunner* модуля *notification* *(рисунок 29)* передадим сообщение в обработчик и проверим появилось ли это сообщение в очереди сообщений, для чего проверим появилось ли сообщение в очереди через панель управления RabbitMQ (Рисунок 30).

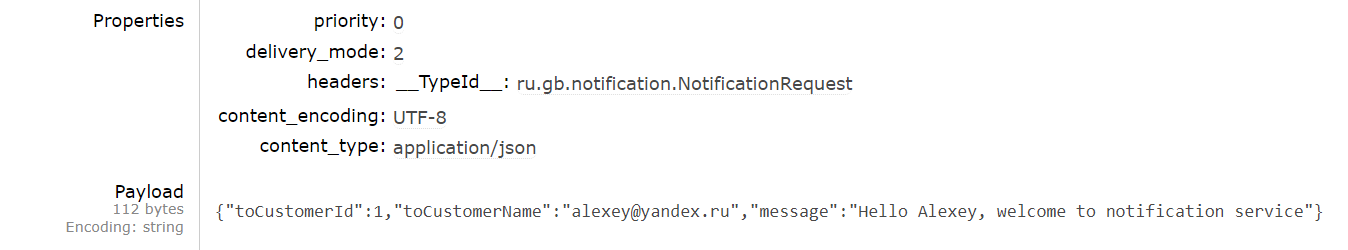
Рисунок 29

Временный код для проверки работоспособности брокера сообщений.

Рисунок 30

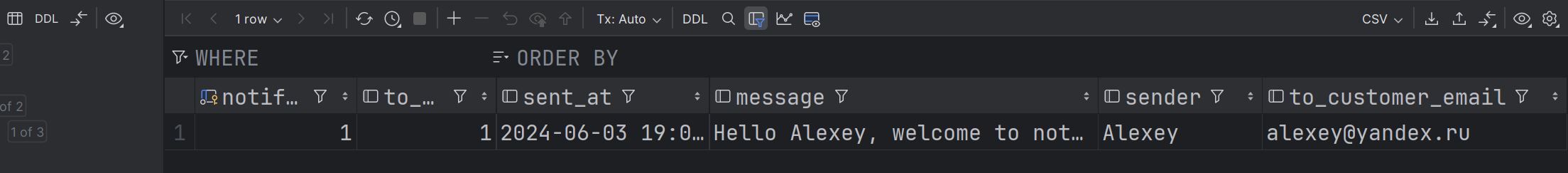
Как видно из рисунка 30, сообщение было успешно добавлено в очередь сообщений. Успешно отработал Object Json конвертер и класс Person был преобразован в JSON формат. Сам сервис работает успешно, поэтому далее необходимо добавить функционал обработки сообщений уже непосредственно от основного сервиса «customer» через брокер сообщений в сервис «notification».

Модуль «customer» сконфигурируем следующим образом. В сервисный класс модуля добавим зависимость *message producer* из модуля *amqp* и перепишем метод *register customer* использую теперь брокер сообщений, вместо прямого обращения к сервису *notification.* Проверку доставки сообщений к брокеру и далее в очередь проверим запустив все сервисы приложения. И отправим запрос на добавление нового «покупателя».

Рисунок 31

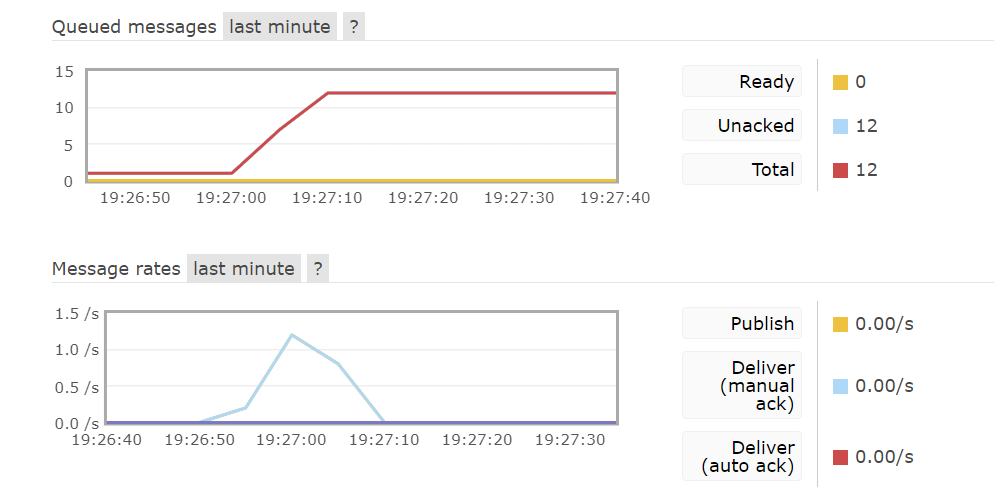
Как видно из рисунка 31, сообщение было успешно доставлено до брокера сообщений и размещено в очередь. Следующим этапом нужно добавить *message listener* в модуль *notification*

В модуле *notification* добавим новый класс, функционалом которого будет просматривать очередь сообщений и доставлять эти сообщения. В классе будет реализован функционал один метод, аннотацией которого будет @RabbitListener который будет взаимодействовать с базой данных модуля *notification* и будет просто сохранять сообщения в базе данных. После перезагрузки модуля *notification*, сообщение, которое находится сейчас в очереди сообщений будет немедленно обработано и добавлено в базу данных.

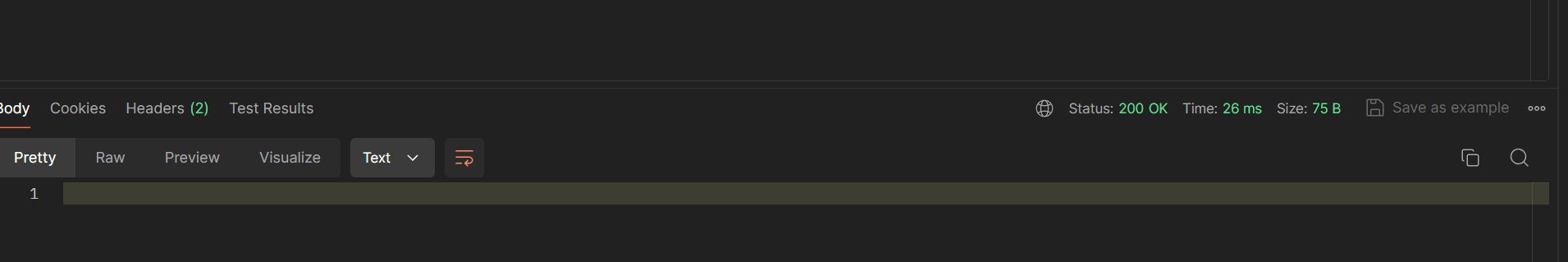
Рисунок 32

По завершении перезагрузки сообщение пропало из очереди сообщений и было добавлено в базу данных что видно из рисунка 32.

Одной из основных целей данного диплома было сделать один из сервисов асинхронным, тем самым не зависеть от скорости обработки ли, работоспособности ли, самого сервиса. Т.е. даже если сервис не отвечает или время ответа сервиса значительно увеличено, это не должно приводить к ошибкам или простою всего приложения. Воссоздать увеличенное время ответа можно просто перезапустить сервис *notification* в режиме отладки и поставить точку остановки в классе *Notification Consumer*. Если Postman вернет код 200 и в это время сервис *notification* будет искусственно приостановлен, значит сервис работает в асинхронном режиме и приложение не зависит от производительности и работоспособности одного конкретного сервиса, который не жизненно необходим для реализации логики всего приложения.

Рисунок 33

Как видно из рисунка 33, было отправлено 12 запросов на добавление «покупателя» в базу данных. 12 сообщений сейчас находятся в очереди сообщений поскольку сервис *notification* был приостановлен.

Рисунок 34

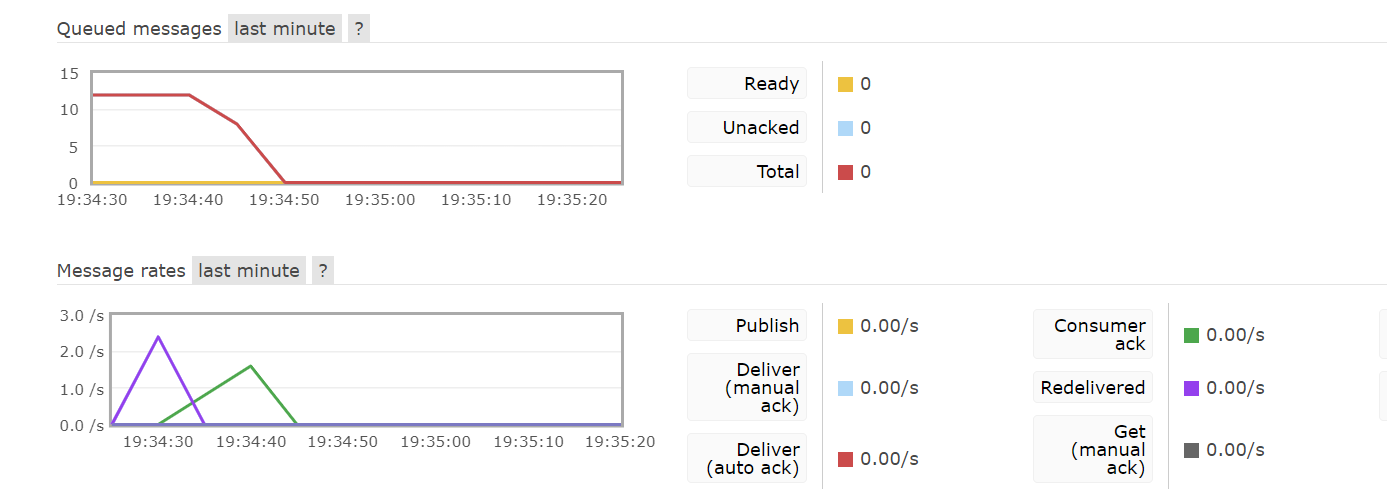
Как видно из рисунка 34 все запросы обработаны и приложение вернуло код 200. Значит, приложение работает так как задумано. После возобновления работы сервиса, все сообщения корректно обработаны и добавлены в базу данных Рисунок 35.

Рисунок 35

Задача проектирования завершена. Приложение работает по задуманной логике. Есть балансировщик нагрузок, который принимает запросы и направляет их по определенному пути к сервису *customer* далее запрос на добавление обрабатывается в сервисе проверки не мошенник ли этот покупатель, причем эти 2 сервиса логически связаны между собой, поскольку нельзя добавить в базу аккаунт мошенника. Но также сервис *customer*, направляет запрос и на сервис уведомлений, который имитирует работу службы уведомлений, при замедлении которой основной функционал должен работать, поэтому данный сервис работает в асинхронном режиме при помощи брокера сообщений.

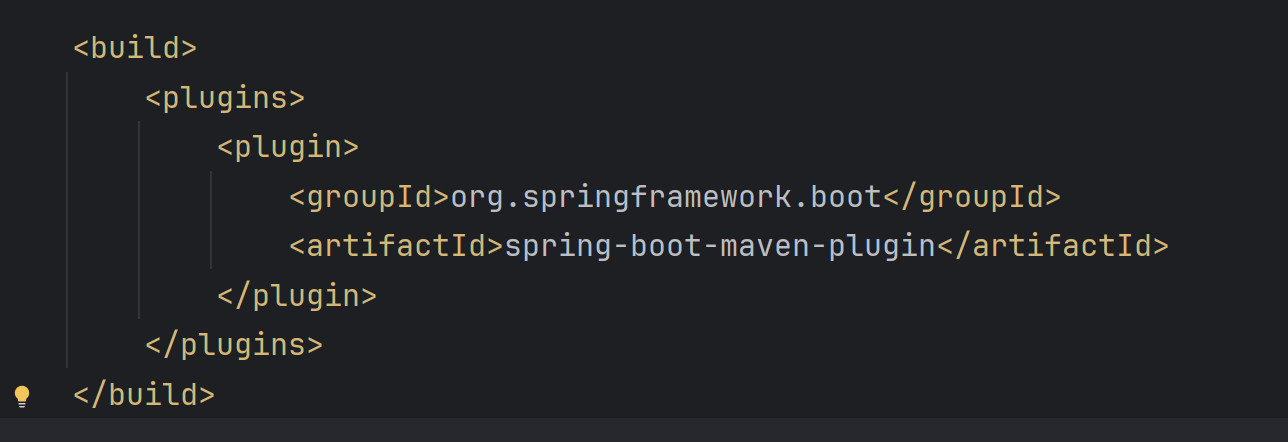
GitHub: [commit a921d58](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/a921d58698c2d6c2129a1790cb25248cf8108a87)

**Глава 3. Сборка проекта**

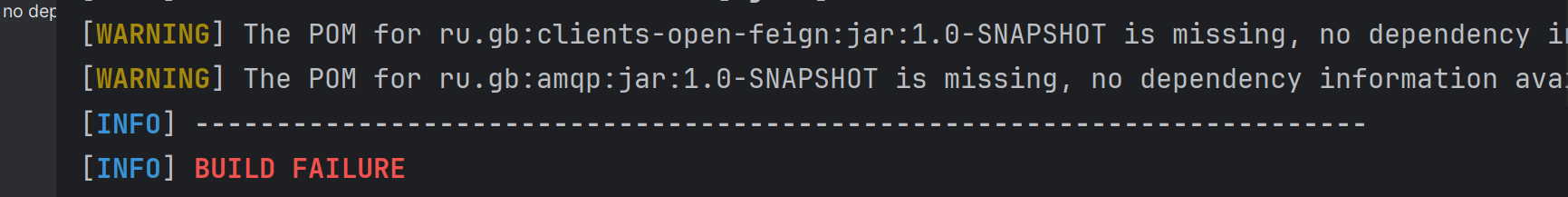
**3.1 Сборка проекта с использованием Maven Compiler Plugin / Spring boot maven plugin**

В данной части проект будет собран в jar файлы, которые можно будет запустить. В родительский pom файл проекта добавим plugin *Maven compiler plugin* и настроим версию языка для сборки. Укажем 17 версию. Для проверки работы упаковщика и сборки обычных модулей, соберем 2 jar файла для модулей *clients-open-feign* и *amqp.* Для этого сначала запросим очистку командой *clean* далее скомпилируем модуль и потом командой *mvn package* получим jar файлы для каждого из модулей.

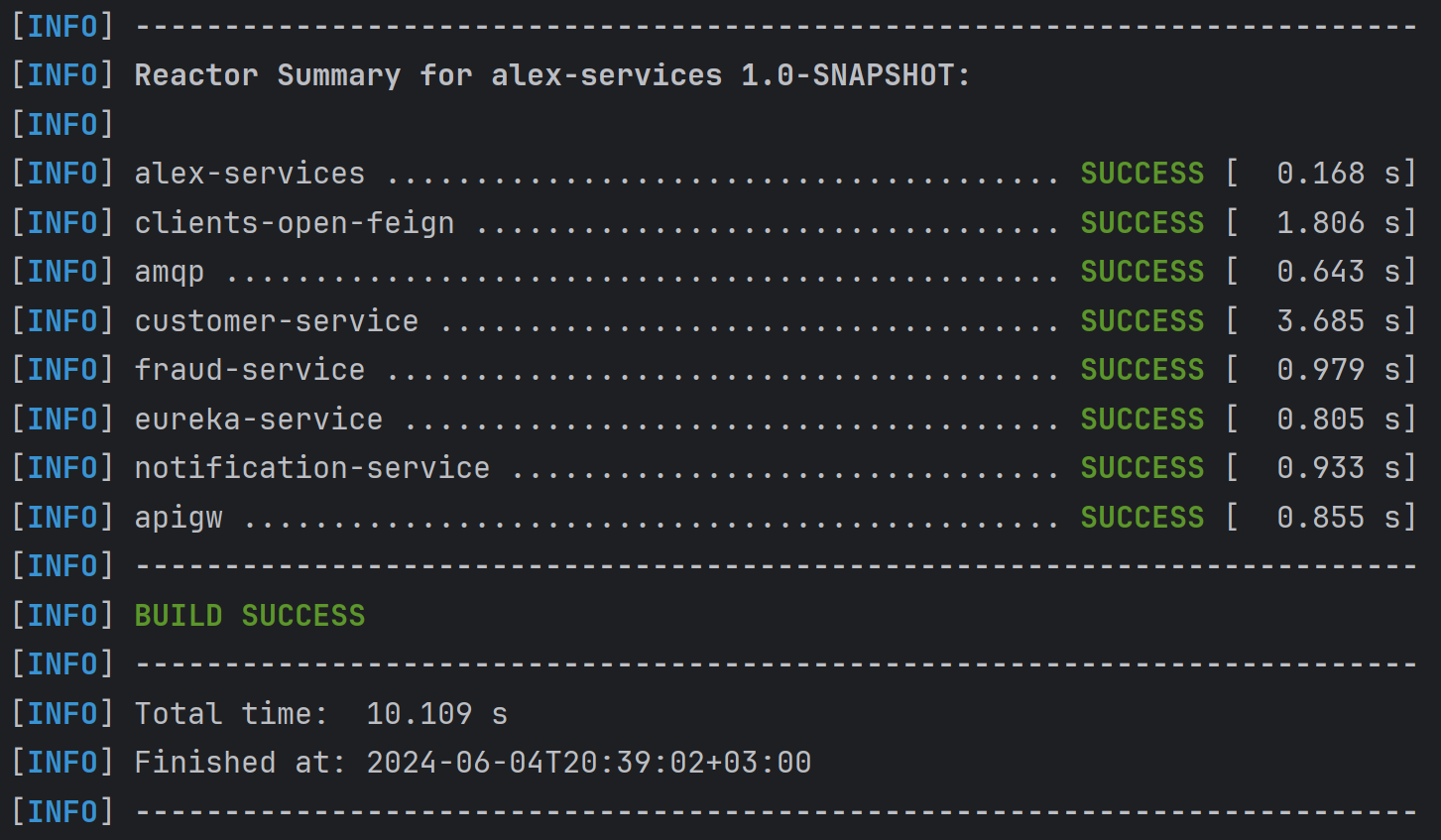
Поскольку остальные модули являются микросервисами Spring, их сборку сделаем с помощью Spring boot maven plugin, который уже подключен к родительскому pom файлу, для того что бы микросервисное приложение работало корректно. Для настройки сборки с помощью этого плагина, укажем выполнение сборки как *repackage*. Механизм работает следующим образом. Сначала упаковщик соберет Spring модули с помощью *maven compiler plugin* и далее переупакует их с помощью s*pring boot maven plugin*. Что позволит в дальнейшем их запустить как Spring приложение. В каждый из микросервисов, *apigw, customer-service, eureka-service, notification-service* и *fraud-service* в pom файл модулей добавим следующую настройку сборки (рисунок 36)

Рисунок 36

После добавления этого плагина, сборщик будет знать что необходимо все настройки подтянуть из родительского pom файла всего приложения. Сами настройки уже сделаны. Более того в каждом из pom файлов укажем что упаковывать необходимо в jar архив. Если сейчас попробовать упаковать любой из микросервисов сборщик выдаст ошибку.

Рисунок 37

Согласно официальной документации к maven [maven lifecycle](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html) для того что бы упаковщик имел доступ ко всем зависимостям приложения, их необходимо установить локально. Таким образом необходимо инсталлировать все зависимости всего приложения через команду *mvn clean install* (рисунок 38)

Рисунок 38

Как видно из рисунка 38, все модули приложения успешно установленны. Само приложение можно запустить через команду maven spring-boot: run или классическим способом через java -jar. Успешный запуск приложения можно проверить через отправку запроса через Postman.

GitHub: [commit 92f5a20](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/92f5a20f10eac94162a003671d4bec392ebe537c)

**3.1 Сборка проекта с использованием Docker и Jib**

Поскольку приложение использует *docker-compose.yml* файл для развертывания баз данных, а также развертывания Zipkin и Eureka-server было принято решение попробовать развернуть и сервисы в виде отдельных контейнеров, настроить порты и протестировать работоспособность. Большое количетсво разработчиков используют Docker и контейнеризацию приложений для совместной работы, особоенно на этапе отладки. Когда очень важно что бы приложение на любой машине запускалось в одинаковых условиях. Мне известно 2 библиотеки, которые позволяют создавать Docker образы. Spring boot maven plagin OCI images, а также Jib от компании Google. Jib создает оптимизированные образы Docker и [OCI](https://github.com/opencontainers/image-spec) для Java-приложений без демона Docker и без глубокого изучения рекомендаций Docker. Доступно в виде плагинов для [Maven](https://github.com/GoogleContainerTools/jib/blob/master/jib-maven-plugin) и [Gradle](https://github.com/GoogleContainerTools/jib/blob/master/jib-gradle-plugin), а также в виде библиотеки Java. Источник: [Jib](https://github.com/GoogleContainerTools/jib/tree/master). Из основных преимуществ, возможность создавать образы и сразу отправлять их на многие из репозиториев хранения образов, а не только Docker Hub. Плагин позиционирует себя как простой в настройках, а также быстрый и не требует Docker deamon на локальной машине. По этой причине было решено использовать именно его.

Первоначальная настройка начинается с pom файла проекта. Необходимо добавить plugin в менеджер плагинов приложения *jib-maven-plugin* и добавить в конфигурацию рекомендованный базовый образ *eclipse-temurin:17* и также архитектуру для работы под *windows & macOS*. Также добавим на какой стадии и как будет формироваться образ, выбрано, что образ будет создаваться на локальной машине (рисунок 39).

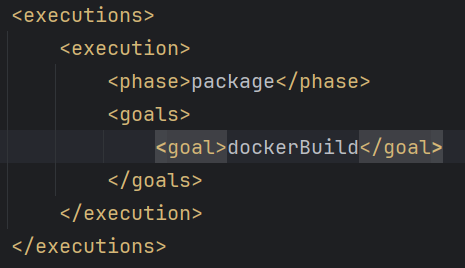
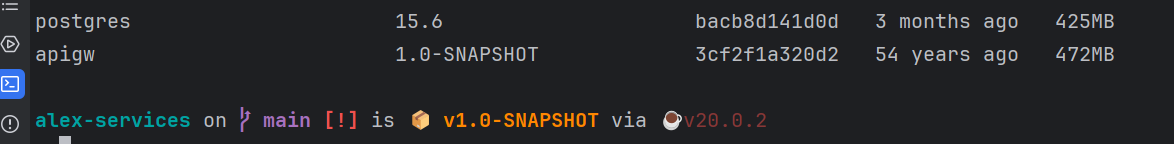
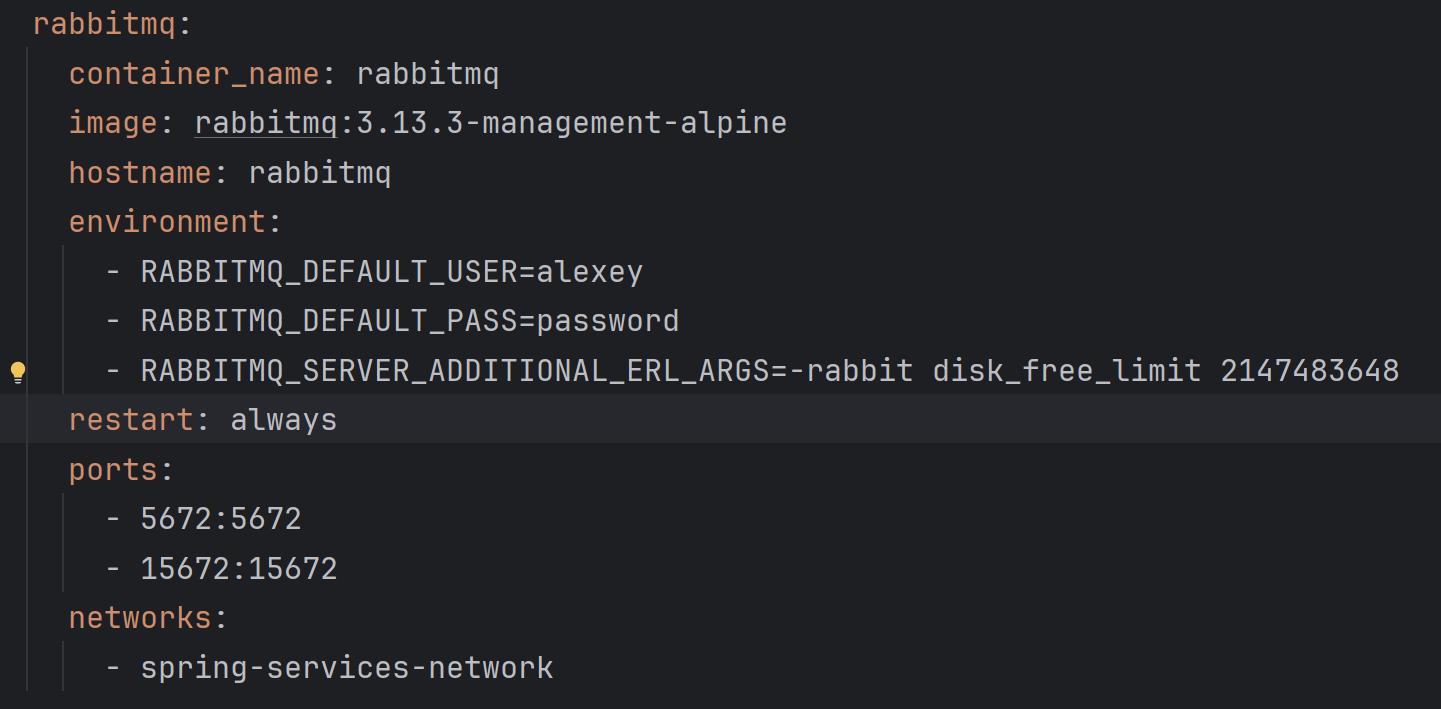
****

Рисунок 39

Пробуем собрать первый сервис *apigw*. В pom файл необходимо добавить профиль докер. И далее собрать модуль.

Рисунок 40

Далее, необходимо добавить тот же профиль во все оставшиеся миросервисы и собрать их все. Как и стояла задача, необходимо настроить *docker-compose.yml* файл для удобства и простоты запуска приложения. Для чего пропишем сами сервисы, порты и зависимости для каждого из сервисов. Поскольку сервисы будут работать в контейнерах, через localhost к ним уже не обратится, поэтому необходимо изменить файл *application.yml*. С другой стороны, для сохранения функциональности без развертывания приложения в контейнерах и внесения изменений в рабочую версию приложения необходимо создать дополнительный файл *application-docker.yml.* Причем название после дефиса будет являтся названием профиля. Таким образом скопировав и переименовав файл *application.yml* в *application-docker.yml* получится дополнительный профиль, который можно сконфигурировать отдельно от основного профиля. Данные конфигурации называются *Spring profiles.* Для каждого сервиса в файле *docker-compose.yml* добавим переменную среды *SPRING\_PROFILES\_ACTIVE=docker.* После чего пересоберем приложение с использованием профиля *docker-build-image* что бы пересобрать именно сборку docker котейнеров. К сожалению, в документации к RabbitMQ нет достаточного количества информации, каким образом развернуть и настроить взаимодействие между контейнерами и контейнером с RabbitMQ. Для себя я решил проблему ошибки *docker image forSpring/RabbitMQ tutorial results in connection refused* следующим образом. При создании контейнера с образом *rabbitMQ, в файле docker-compose.yml* указал название хоста и добавил переменные среды имени пользователя и пароля (рисунок 41)



Дополнительно в файлах *application-docker.yml* тех сервисов, которые взаимодействуют с *rabbitMQ* прописал путь к хосту и указал имя пользователя и пароль, в моем случае это были сервисы *customer & notification* (рисунок 42)

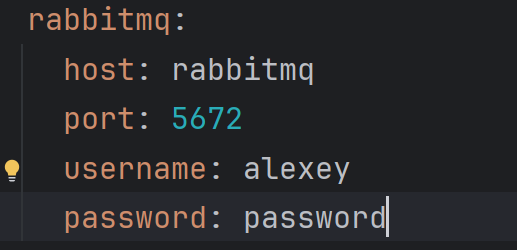


Рисунок 42

# Заключение

В заключение необходимо включить следующее:

1. Краткие и ёмкие теоретические и практические выводы, которые были получены во время анализа теоретической базы и практического исследования.
2. Оценка проведённого исследования, описание его результатов.
3. Практическая значимость работы, рекомендации и планы на дальнейшие исследования.
4. Общий итог — достижение цели, выполнение задач, доказательство гипотезы.
5. Предложения по совершенствованию объекта исследования.

# Список используемой литературы

Здесь нужно будет указатьсписок используемой литературы, ссылки на все ресурсы, которые нужны были для создания проектной работы.

Основные правила оформления использованной литературы и ресурсов:

1. Каждый источник упоминается единожды, независимо от того, насколько часто на него ссылаются.
2. Список литературы оформляется в алфавитном порядке по фамилии автора, сначала русскоязычная литература, затем иностранная, далее интернет-сайты.
3. Библиографическая запись обязательно включает:
   * Фамилию автора или фамилии их группы, инициалы (при наличии).
   * Название статьи, книги, справочника, закона, иного документа.
   * Населённый пункт, в котором был издан источник, наименование издательства.
   * Год публикации.
   * Число страниц.

## Пример

* *Книга: Автор. Название книги. Город: Издательство, Год.*
* *Статья: Автор. "Заголовок статьи." Название журнала Том, номер (Год): страницы.*

# Приложения

В **приложения** обычно входят артефакты, получившиеся в процессе создания проекта:

1. Объёмные графики и таблицы, которые не помещаются на лист А4.
2. Длинные математические формулы и расчёты по ним.
3. Характеристики аппаратуры, которая использовалась для проведения исследования.
4. Авторские методики.
5. Вспомогательный материал: тесты, карточки, схемы, рисунки.
6. Материалы, полученные на предприятии: отчёты, прочие документы.

**Основные требования к оформлению дипломного проекта**

1. Текст проекта оформляйте только с одной стороны листа формата А4. При наборе используйте межстрочный интервал 1,5.  
   Объемные таблицы/иллюстрации возможно размещать на листах А3, А2 и выносить в приложения.
2. При наборе информации, используйте шрифт Times New Roman с выравниванием текста по ширине. Рекомендуем кегль — 12–14, для некоторых структурных единиц работы можно выбрать полужирное выделение.
3. Поля страницы, где размещается текст, выставляйте по определенным показателям: нижнее/верхнее – 20 мм, правое – 10 мм, левое (для последующей прошивки) – 30 мм.
4. Все листы дипломного проекта нумеруются по порядку, начиная с введения (используется сквозная нумерация). Номер указывайте в центре нижней части страницы без точки.
5. Заголовок раздела с одним подразделом или пунктом не нумеруется.
6. После текста перед следующим заголовком установите одиночный интервал.
7. После заголовка перед подзаголовком проставьте двойной интервал.
8. Имена собственные — наименования компаний/изделий, фамилии учёных и т. д., указывайте на языке оригинала.
9. Каждый структурный элемент начинается с нового листа.

*Вся работа должна составлять 50+ страниц формата А4, включая приложения.*

|  |
| --- |
| 💡 Инструкция по работе с шаблоном.   1. Создайте копию данного шаблона. Файл - Создать копию. 2. Изучите шаблон и начинайте работу. |