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**

[Введение 3](#__RefHeading___Toc1308_1500850277)

[Глава 1. Теоретическая часть 5](#__RefHeading___Toc1306_1500850277)

[1.1 Микросервисная архитектура плюсы и минусы 5](#__RefHeading___Toc1332_1500850277)

[1.2 Минусы микросервисного подхода 5](#__RefHeading___Toc1330_1500850277)

[1.3 Плюсы микросервисного подхода 6](#__RefHeading___Toc1328_1500850277)

[1.4 Что такое микросервисы? 7](#__RefHeading___Toc1326_1500850277)

[1.5 Этапность разработки 10](#__RefHeading___Toc1324_1500850277)

[Глава 2. Разработка приложения 16](#__RefHeading___Toc1304_1500850277)

[2.1 Подготовка базовой архитектуры приложения 16](#__RefHeading___Toc1310_1500850277)

[2.1.1 Развертывание базового приложения и настройка pom.xml 16](#__RefHeading___Toc1352_1500850277)

[2.1.2 Создание сервисных приложений 17](#__RefHeading___Toc1350_1500850277)

[2.1.3 Развертывание баз данных с использованием Docker и подключение сервисов 18](#__RefHeading___Toc1348_1500850277)

[2.2 Rest Template 20](#__RefHeading___Toc1322_1500850277)

[2.2.1 Настройка базовой коммуникации 20](#__RefHeading___Toc1346_1500850277)

[2.2.2 Eureka server 22](#__RefHeading___Toc1344_1500850277)

[2.2.3 Open Feign 25](#__RefHeading___Toc1342_1500850277)

[2.2.4 Трассировка микросервисов с использованием Micrometer и Zipkin 26](#__RefHeading___Toc1340_1500850277)

[2.2.5 API Gateway 29](#__RefHeading___Toc1338_1500850277)

[2.2.6 Брокер сообщений 30](#__RefHeading___Toc1334_1500850277)

[Глава 3. Сборка проекта 38](#__RefHeading___Toc1296_1500850277)

[3.1 Сборка проекта с использованием Maven Compiler Plugin / Spring boot maven plugin 38](#__RefHeading___Toc1320_1500850277)

[3.2 Сборка проекта с использованием Docker и Jib 40](#__RefHeading___Toc1318_1500850277)

[Заключение 44](#__RefHeading___Toc1300_1500850277)

[Основные выводы 44](#__RefHeading___Toc1316_1500850277)

[Оценка проведенного исследования 45](#__RefHeading___Toc1314_1500850277)

[Планы на дальнейшее развитие приложения 45](#__RefHeading___Toc1312_1500850277)

[Список используемой литературы 46](#__RefHeading___Toc1302_1500850277)

# 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 существующих 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) и логирования.

К приложению должен быть подключен балансировщик нагрузок. При изучении литературы сделал следующий вывод: микросервисные приложения должны включать в свой функционал как «внешний» так и «внутренний» балансировщики нагрузок. Под внешним балансировщиком понимается внешний отказоустойчивый сервис, позволяющий настроить коммуникацию между запросом от конкретного пользователя или пользователей к микросервисному приложению. «Best practices» это использование внешних провайдеров таких решений от таких компаний как Google, Yandex, Amazon и так далее. Поскольку разрабатываемое приложение не затрагивает изучение данного функционала, «внешний» балансировщик не планируется к реализации. Но будет реализован «внутренний» балансировщик нагрузок который будет настраивать распределение входящего запроса на конкретные микросервис/микросервисы. Таким образом, в случае необходимости всегда можно будет реализовать распределение запроса скажем по принципу «Round Robin». Дополнительно стоит сказать, что довольно часто используют балансировщики нагрузок как промежуточный посредник при коммуникации и между сервисами. В данной разработке внедрение таких балансировщиков не планируется, поскольку, на мой взгляд, это излишне для данного приложения.

Приложение должно реализовывать 2 механизма коммуникации: синхронный режим и асинхронный режим. Синхронный режим будет симулировать проверку, то есть жесткую связь работы двух отдельных сервисов, где от результатов работы одного сервиса зависит логика дальнейших действий второго сервиса. Асинхронный режим полезен, когда нет необходимости в жесткой связи между отдельными сервисами. Как пример, обычное уведомление, которое должно быть отправлено по результатам работы других сервисов. Взаимодействие между сервисами в асинхронном режиме должно быть реализовано посредством брокера сообщений. Таким образом я смогу еще более снизить связанность между сервисами и легко реализовать асинхронность работы, сохранение очереди сообщений, распределения сообщений по очередям без дополнительного кода.

**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, DrawIO

Состав команды: Зотов Алексей Викторович (программист)

# **Глава 1. Теоретическая часть**

## **1.1 Микросервисная архитектура плюсы и минусы**

Чем быстрее идея воплотится в новый проект, тем больше шансов занять нишу, завоевать лояльность пользователей и, как следствие, стать успешнее конкурентов. Ускорить разработку и сделать её более гибкой и управляемой помогает микросервисная архитектура.

Что такое микросервисная архитектура

Большинство проектов, особенно стартапов сначала выбирают монолит, так как нужен быстрый старт при минимальных ресурсах и не ясны перспективы. Но с увеличением размеров и сложности приложения управлять монолитом становится всё труднее. Любое изменение в монолитном приложении может иметь каскадный эффект, оказывая влияние на другие части приложения, что существенно усложняет интеграцию и развертывание в промышленной системе. Также присутствует риск отказа всего приложения.

Альтернатива монолиту — микросервисы.

**Микросервисная архитектура** — один из подходов проектирования, при котором единое приложение строится как набор небольших сервисов. Каждый модуль работает в собственном процессе и взаимодействует с другими модулями. Модули работают независимо и построены вокруг бизнес-потребностей и выполняют определённую функцию. Например, хранят данные пользователей или отвечают за аутентификацию.

Впервые о микросервисах заговорили в 2000-х годах, когда в США произошёл кризис электронных технологий. Тогда был бум на интернет-сервисы, но никто не задумывался о масштабировании, а оборудование не было таким мощным. В один момент крупные сервисы не смогли справиться с возросшей нагрузкой и потеряли своих пользователей и инвестиции. Это привело к переоценке многих вещей в индустрии ПО, и уже к началу 2010-х микросервисы сформировались как концепция архитектуры. А к 2014-ому их начали внедрять крупные компании вроде Netflix, Amazon и Twitter.

## 1.2 Минусы микросервисного подхода

При кажущейся простоте и логичности деление большого монолитного приложения на самостоятельные сервисы сам микросервисный подход не лишён недостатков, а его плюсы могут оборачиваться минусами:

* **Сложнее осуществлять мониторинг**. Монолит один, и отслеживать, как он работает проще. Микросервисов сотни, а иногда тысячи, и уследить за каждым физически невозможно. Поэтому приходится уделять много внимания системам управления и мониторинга.
* **Каждый микросервис может использовать тот язык программирования и те технологии, что удобны команде, которая его разрабатывает**. С одной стороны, это плюс — какие-то фреймворки и языки лучше заточены под конкретные вещи. Если мы делаем сервис обработки изображений или машинного обучения, можем не заострять внимание на технологиях монолита, а выбрать решение под задачу. С другой стороны, это минус — нужно как-то конфигурировать все микросервисы между собой и поддерживать.
* **Снижение доверия**. Когда у нас сотни узлов, могут возникать проблемы с аутентификацией и авторизацией, поскольку есть вероятность подсоединения мошенников.
* **Сложности развёртывания**. Чтобы требования по отказоустойчивости выполнялись, микросервисы нужно развёртывать на отдельных серверах. И здесь не работает подход: «Берём приложение, ставим и запускаем». Нужны системы оркестрации и деплоймента.

## 1.3 Плюсы микросервисного подхода

Один из основных плюсов микросервисов — повышение показателей доступности и отказоустойчивости. Когда монолит падает, он полностью перестаёт работать. В приложении с микросервисной архитектурой перестаёт работать только какая-то часть. Например, в интернет-магазине может сломаться корзина, но клиенты без проблем продолжат пользоваться каталогом, добавлять товары в избранное и т.д.

Ещё один плюс — повышение надёжности. В случае с монолитом падает сразу все приложение, а с микросервисом сломается только участок системы, остальное будет работать.

Также важно, что в приложении с микросервисной архитектурой локализуются сложность и риски отказов, а производительность системы масштабируется по горизонтальному пути. Компоненты (узлы) автономны, а значит легче поддаются тестированию.

## **1.4 Что такое микросервисы?**

Это архитектурный шаблон, в котором сервисы:

1. маленькие (small),
2. сфокусированные (focused),
3. слабосвязанные (loosely coupled),
4. высокосогласованные (highly cohesive).

«М**аленький**» сервис: Это значит, что сервис в микросервисной архитектуре не может разрабатываться больше чем одной командой. Обычно одна команда разрабатывает где-то 5 – 6 сервисов. При это каждый сервис решает одну бизнес-задачу, и его способен понять один человек. Потому что, если один человек способен поддерживать всю бизнес-логику одного сервиса, он построит действительно эффективное решение. И если сервис маленький, все намного проще.

Что значит «**сфокусированный**» сервис? Это значит, что сервис решает только одну бизнес-задачу, и решает ее хорошо. Такой сервис имеет смысл в отрыве от остальных сервисов. Другими словами, его можно выложить в интернет, дописав security-обертку, и он будет существовать отдельно.

«С**лабосвязанный**» сервис. Это когда изменение одного сервиса не требует изменений в другом. Вы связаны посредством интерфейсов, есть решение через DI и IoC — что сейчас стандартная практика, применять которую нужно обязательно.

Что такое «**высокосогласованный**» сервис? Это значит, что класс или компонент содержит все нужные методы решения поставленной задачи. Однако тут часто возникает вопрос, чем высокая согласованность (high cohesion) отличается от SRP? Допустим, у нас есть класс, отвечающий за управление кухней. В случае SRP такой класс работает только с кухней и больше ни с чем, но при этом он может содержать не все методы по управлению кухней. В случае же высокой согласованности, все методы по управлению кухней содержатся только в этом классе, и больше нигде.

Характеристики микросервисов

* Разделение на компоненты (сервисы).
* Группировка по бизнес-задачам.
* Сервисы имеют бизнес-смысл.
* Умные сервисы и простые коммуникации.
* Децентрализованное управление.
* Децентрализованное управление данными.
* Автоматизация развертывания и мониторинга.
* Design for failure.

Разделение на компоненты (сервисы)

Компоненты бывают двух видов: библиотеки и сервисы, которые взаимодействуют по сети. Мартин Фаулер определяет компоненты как независимо заменяемые и независимо развертываемые. То есть, если можно взять что-то и спокойно заменить на новую версию, — это компонент. А если что-то связано с другим и их независимо заменить нельзя — они вместе образуют один компонент. Если что-то нельзя развернуть независимо, и требуется логика откуда-то еще, это тоже не компонент.

Группировка по бизнес-задачам (сервисы имеют бизнес-смысл)Стандартная компоновка монолита:

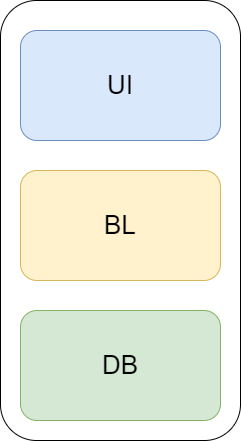


Рисунок 1.

Для повышения эффективности разработки зачастую разработчики вынуждены делить по этим слоям и команды: есть команда, которая занимается UI, есть команда, которая занимается ядром, и есть команда, которая разбирается в БД.

Если же использовать микросервисную архитектуру, сервисы и команды делятся по бизнес-задачам:

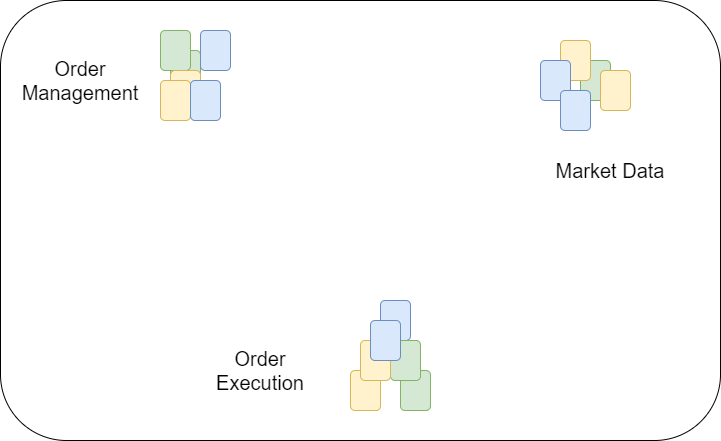


Рисунок 2.

Например, может быть группа, которая занимается управлением заказами, — она группа может обрабатывать транзакции, делать по ним отчеты и т. д. Такая группа будет заниматься и соответствующими БД, и соответствующей логикой. Как бы то ни было, даже если UI остается монолитным, все равно гораздо лучше, когда остальное разбито на компоненты.

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.

## 1.5 Этапность разработки

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 server. По сути посмотреть работу «записной книжки».

Eureka Server — это service discovery (обнаружение сервисов) для микросервисов. Клиентские приложения могут самостоятельно регистрироваться в нем, а другие микросервисы могут обращаться к Eureka Server для поиска необходимых им микросервисов.

Eureka Server также известен как Discovery Server и содержит такую информацию как IP-адрес и порт микросервиса.

Двигаясь далее затронуть такую библиотеку как Open Feign.

Feign — это средство связывания клиента Java с HTTP, созданное на основе Retrofit, JAXRS-2.0 и WebSocket. Первой целью Feign было упрощение единой привязки Denominator к HTTP API независимо от ReSTfulness. (<https://github.com/OpenFeign/feign?tab=readme-ov-file>)

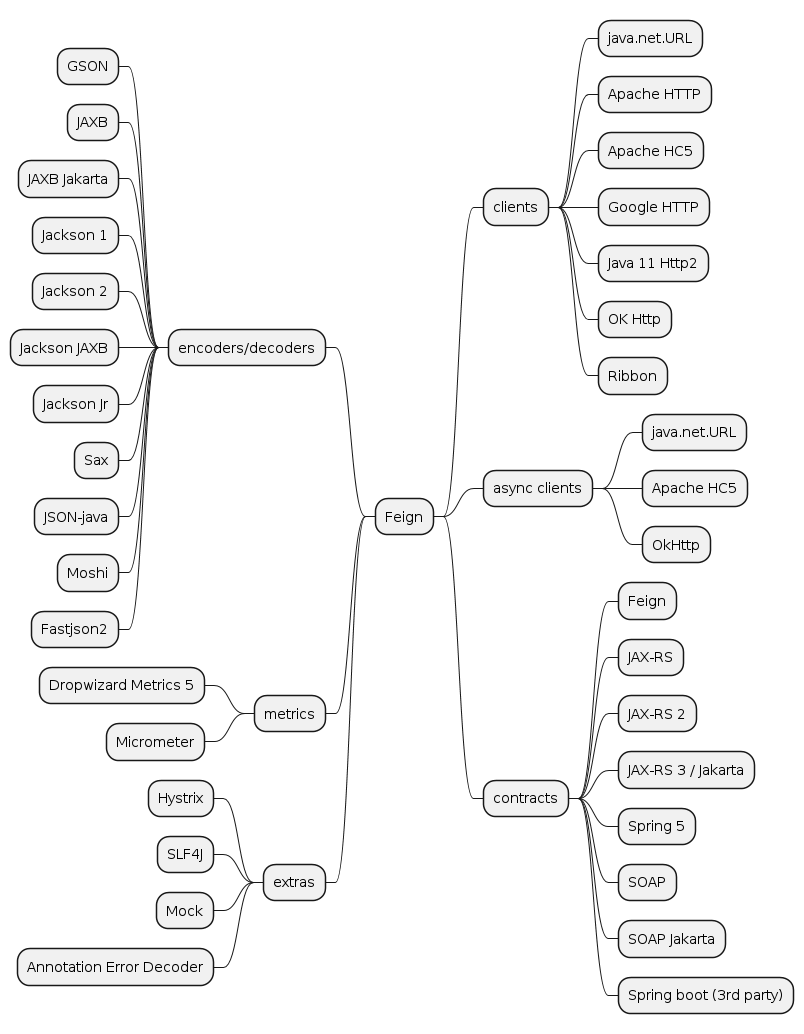


Рисунок 3.

На этом же этапе внедрить третий сервис, сервис трассировки. Этот этап подразумевает внедрение в проект таких технологий как Micrometer и Zipkin.

Zipkin — это распределенная система отслеживания, которая помогает собирать данные включая данные о времени, необходимые для устранения проблем с задержкой в ​​сервисных архитектурах. Функции включают в себя как сбор, так и поиск этих данных.  
Если есть идентификатор трассировки в файле журнала, можно перейти непосредственно к нему. В противном случае можно выполнить запрос на основе таких атрибутов, как услуга, имя операции, теги и продолжительность. И будут суммированы некоторые интересные данные, такие как процент времени, проведенного в сервисе, а также наличие сбоев в операциях.

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. Как было сказано выше, количество микросервисов в приложении может превышать сотни, а то и тысячи. Стандартными инструментами IDE быстро и качественно провести трассировки сигналов и ответов не возможно, поэтому необходимы сторонние библиотеки, которые помогут в режиме реального времени провести разработчика по всем сервисам, начиная от поступающего запроса до конечной точки.

Логичным продолжением по наполнению проекта инструментарием станет внедрение API Gateway и детальное рассмотрение балансировщика нагрузок.

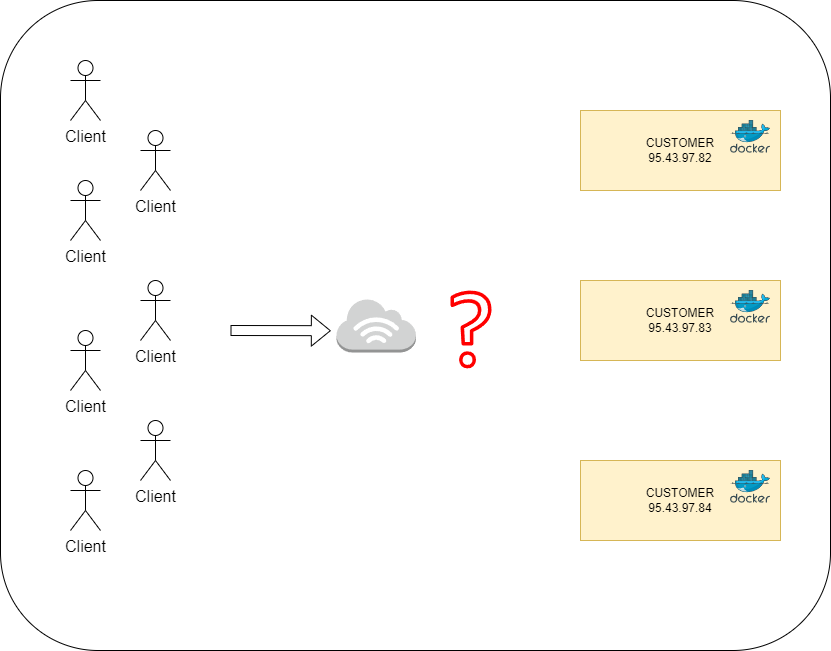


Рисунок 4.

Как показано на рисунке 4, 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. Более того, возникает вопрос, каким образом, автоматически распределить нагрузку и запросы на сервис *customer* если этих сервисов обработчиков, более чем один. В мою же задачу входит создание проекта в котором можно попробовать собственный сервис в условиях, близких к реальным, а балансировщик нагрузки позволит эмулировать распределение нагрузки на несколько образов сервиса утилитами по типу Apache Jmeter.

Следующим этапом будет реализация «брокера сообщений». На данный момент существует множество готовых решений — брокеров сообщений. На мой субъективный взгляд, наиболее популярен сейчас Kafka. Также следует упомянуть такие брокеры как RabbitMQ и SQS, все они работают с использованием Spring AMQP. Мной был выбран такой брокер сообщений как RabbitMQ (<https://www.rabbitmq.com/>). Причина этого проста:

RabbitMQ — это надежный брокер обмена сообщениями и потоковой передачи, который легко развернуть в облачных средах, локально и на вашем локальном компьютере. В настоящее время им пользуются миллионы людей по всему миру (источник: официальный сайт). Забегая вперед, все оказалось не очень просто, но тем не менее, решаемо. Основной проблемой было элементарное отсутствие однозначной информации по развертыванию приложений с использованием Docker в связке с RabbitMQ, хотя следует упомянуть, что развертывание приложения без использования Docker с использованием данного брокера сообщений, никаких сложностей не вызвало.

Последним этапом данной работы, было непосредственное развертывание приложения с использованием Maven, а также с использованием maven и библиотеки Jib от компании Google. Создание Docker образов с использованием данной библиотеки является по сути простейшей операцией убирающих всякую необходимость детального изучения создания Dockerfile.

Все сервисы и инструменты будут реализованы с помощью конейнеров Docker. Контейнеризация — новая идеология в IT. В литературе приводят много аналогий, когда поставщики, то есть разработчики, упаковывают всё в один контейнер, как груз в порту. Это очень хорошая аналогия. Ведь цифровой контейнер точно так же перевозится и распаковывается в готовый продукт.

Основной инструмент контейнеризации — Docker, на его основе даже выстраивают внутреннюю сервисную инфраструктуру. Docker используют в сочетании с оркестраторами, такими как [Kubernetes](https://kubernetes.io/) и [OpenShift](https://www.redhat.com/en/technologies/cloud-computing/openshift). Эти сервисы знаменуют переход от классических виртуалок к облачной инфраструктуре. С их помощью можно гибче, быстрее и динамичнее управлять ресурсами.

**Что такое Docker**

Docker — это платформа, которая позволяет упаковать в контейнер приложение со всем окружением и зависимостями, а затем доставить и запустить его в целевой системе.

Приложение, упакованное в контейнер, изолируется от операционной системы и других приложений. Поэтому разработчики могут не задумываться, в каком окружении будет работать их приложение, а инженеры по эксплуатации — единообразно запускать приложения и меньше заботиться о системных зависимостях.

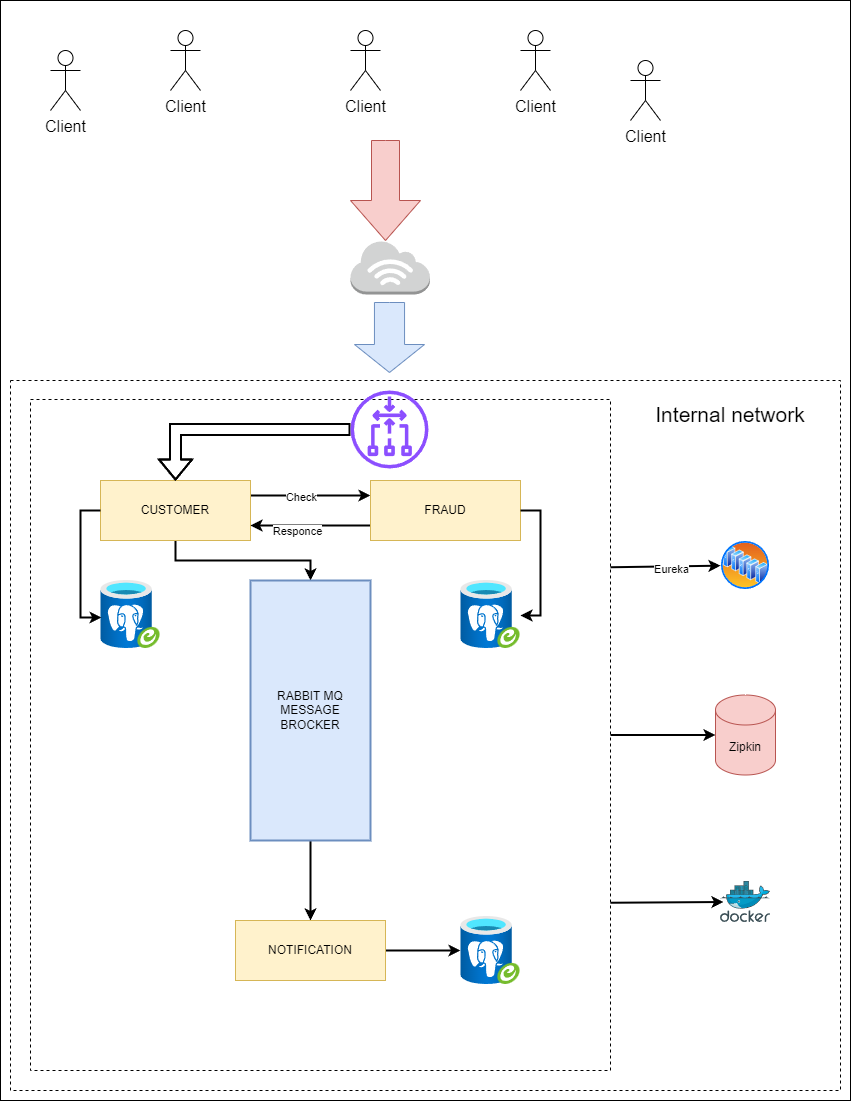
Docker разработали в 2008 году. Изначально это был внутренний проект компании dotCloud, которую впоследствии переименовали в Docker Inc. В 2013 году dotCloud открыла исходный код Docker для сообщества.

Ранние версии Docker представляли собой усовершенствованную обёртку LXC, а с 2015 года Docker уже использовал собственную библиотеку [libcontainer](https://github.com/opencontainers/runc/tree/main/libcontainer), абстрагирующую виртуализационные возможности ядра Linux. Так он превратился в самостоятельную технологию. Платформа неслучайно переехала на libcontainer: гибкость и управляемость LXC-контейнеров оставляла желать лучшего.

Популярность Docker продолжает расти, потому что его поддерживает большое сообщество. Платформа попала в мейнстрим на волне популярности [DevOps](https://skillbox.ru/media/code/chto_takoe_devops_i_zachem_on_nuzhen/?utm_source=media&utm_medium=link&utm_campaign=all_all_media_links_links_articles_all_all_skillbox), быстрых конвейеров доставки и автоматизации.

Основная задача данной работы это:

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

Рисунок 5. Общая схема проектируемого приложения

# **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. 6).

Рисунок 6.

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 собственную папку к проекту а именно: папка с проектом диплома.

«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, которые 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)

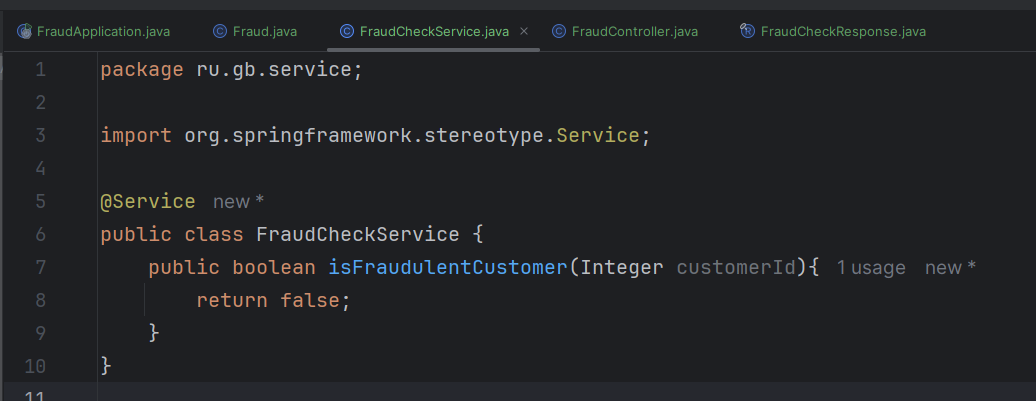
### **2.1.2 Создание сервисных приложений**

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.

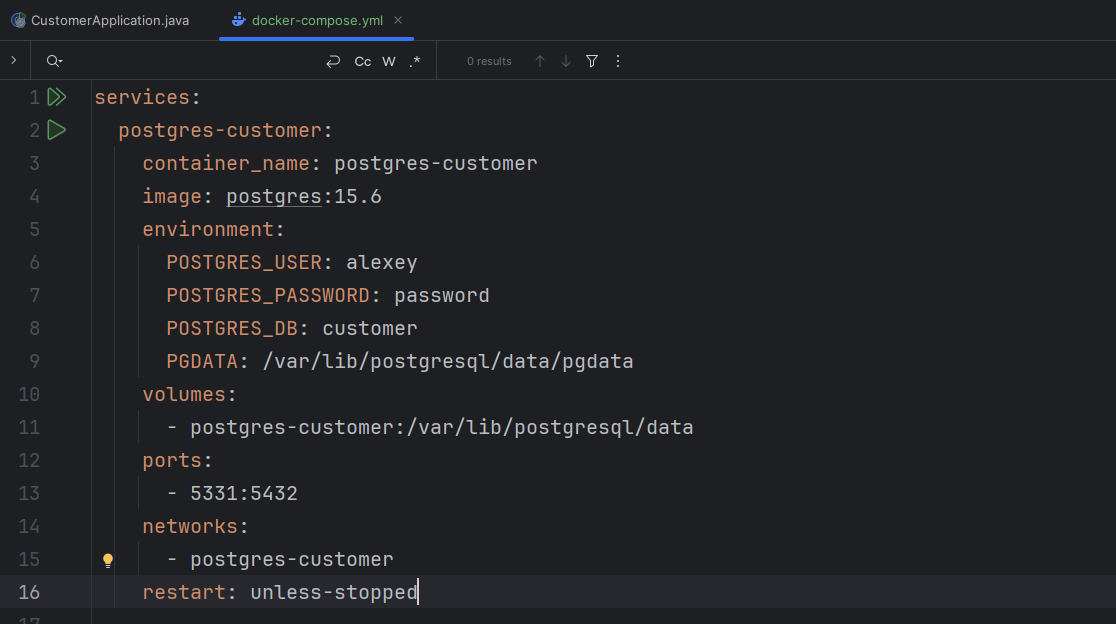
Поскольку данная работа в первую очередь направленна не на обработку персональных данных, а разработку взаимодействия между микросервисами, было решено максимально абстрагировать процесс проверки (рис. 7)

Рисунок 7.

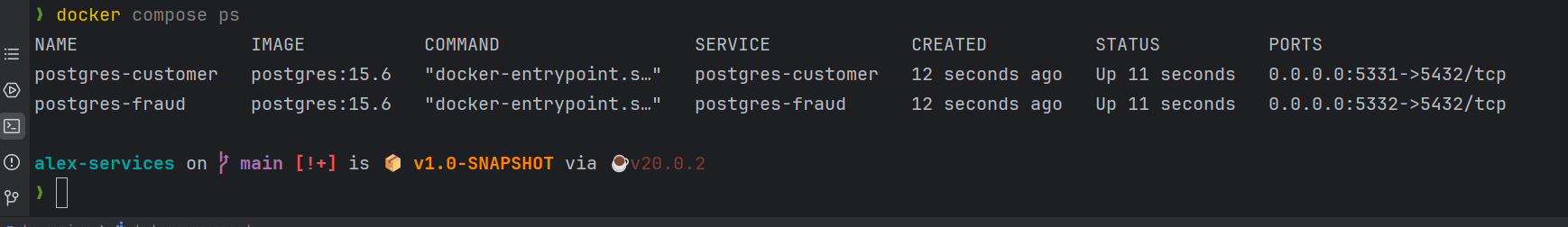
Помимо базовой настройки самих сервисов, мы добавим каждому микросервису отдельный порт, что бы избежать конфликтов. На этом этапе оба сервиса подготовлены для интегрирования с базами данных. [*Git Commit: 102d4d7*](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/102d4d7e9f4524be9ef9d2f30fff18dbf918f03c)

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

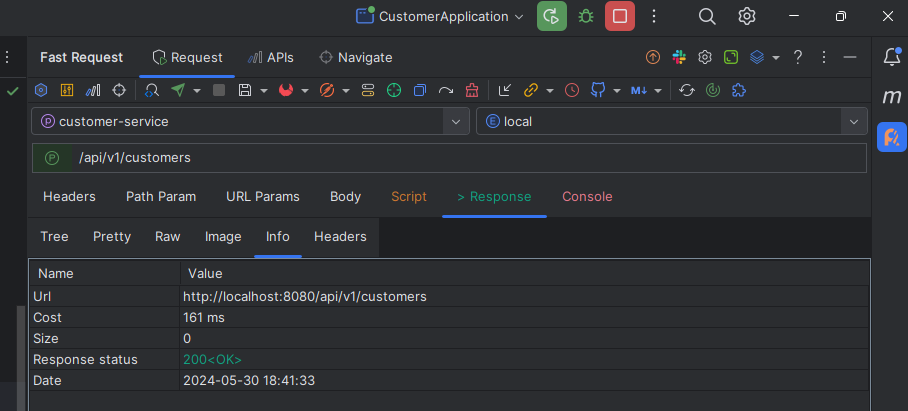
В корневом каталоге создадим файл docker-compose.yml в котором я настрою создание двух отдельных контейнеров для сервисов «customer» и «fraud». Версия PostgreSQL 15.6 (рис. 8)

Рисунок 8.

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 9)

Рисунок 9.

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 10.

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)

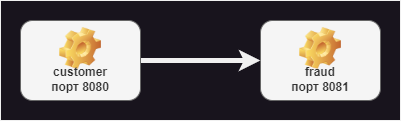
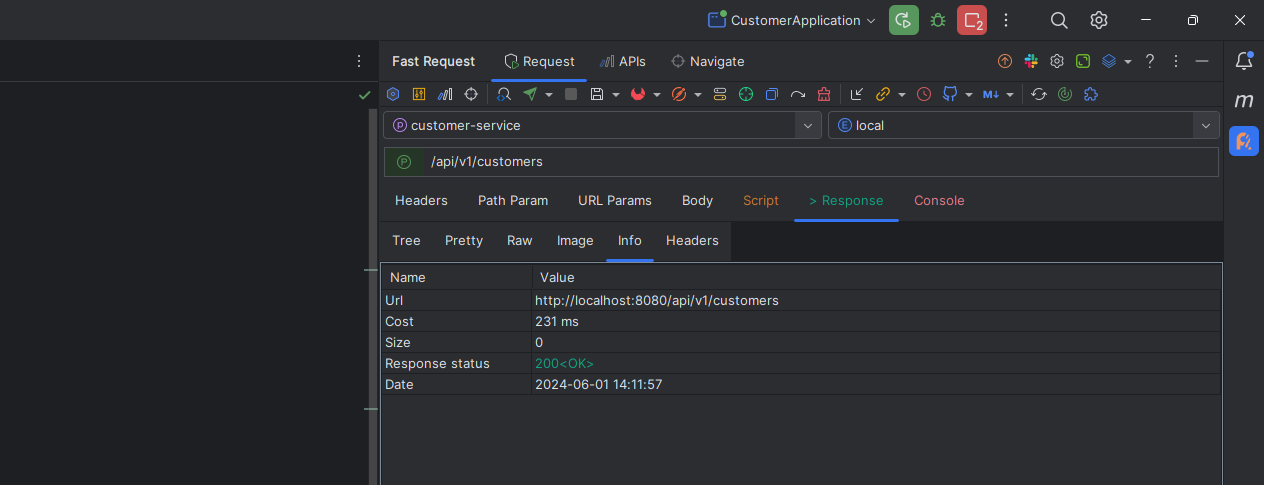


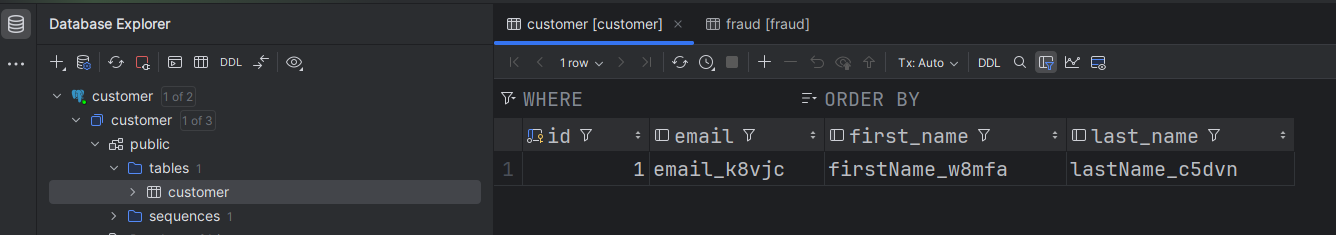
Рисунок 11.

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.

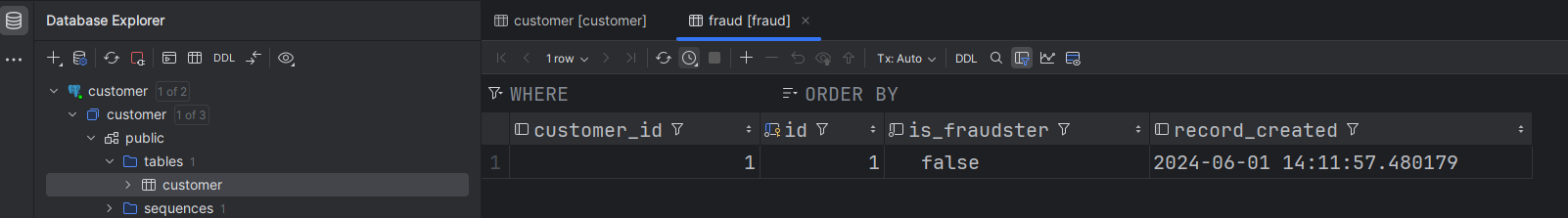
Далее в сервисе добавления нового «покупателя» мы обращаемся к RestTemplate через метод getForObject который направляет GET запрос на указанный URI и возвращает запрашиваемый объект. URI это полный путь к запросу GET fraud контроллера. Тут мы видим несколько неудобных моментов. Первое, это четкое указание 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 12. 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 13.

Запись добавлена в базу данных «покупателей»

Рисунок 14.

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 сущностней 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 как на рисунке 15.

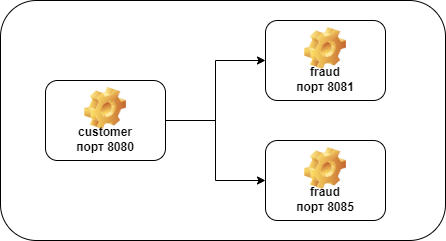


Рисунок 15.

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) (рис 16)

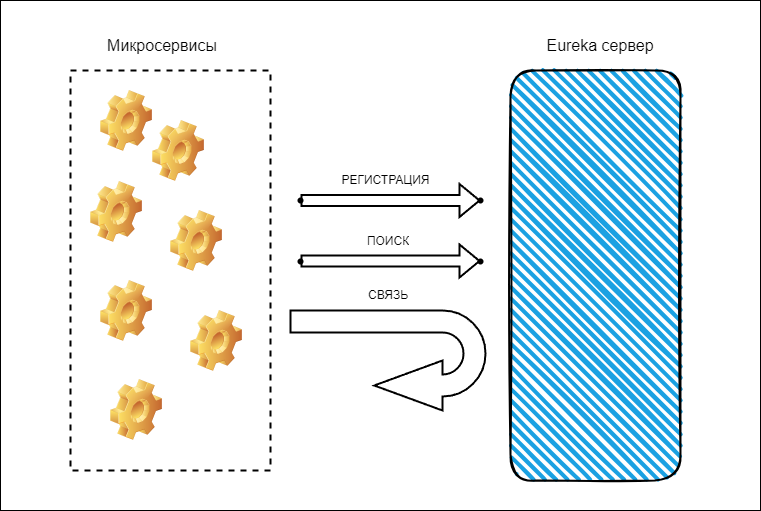
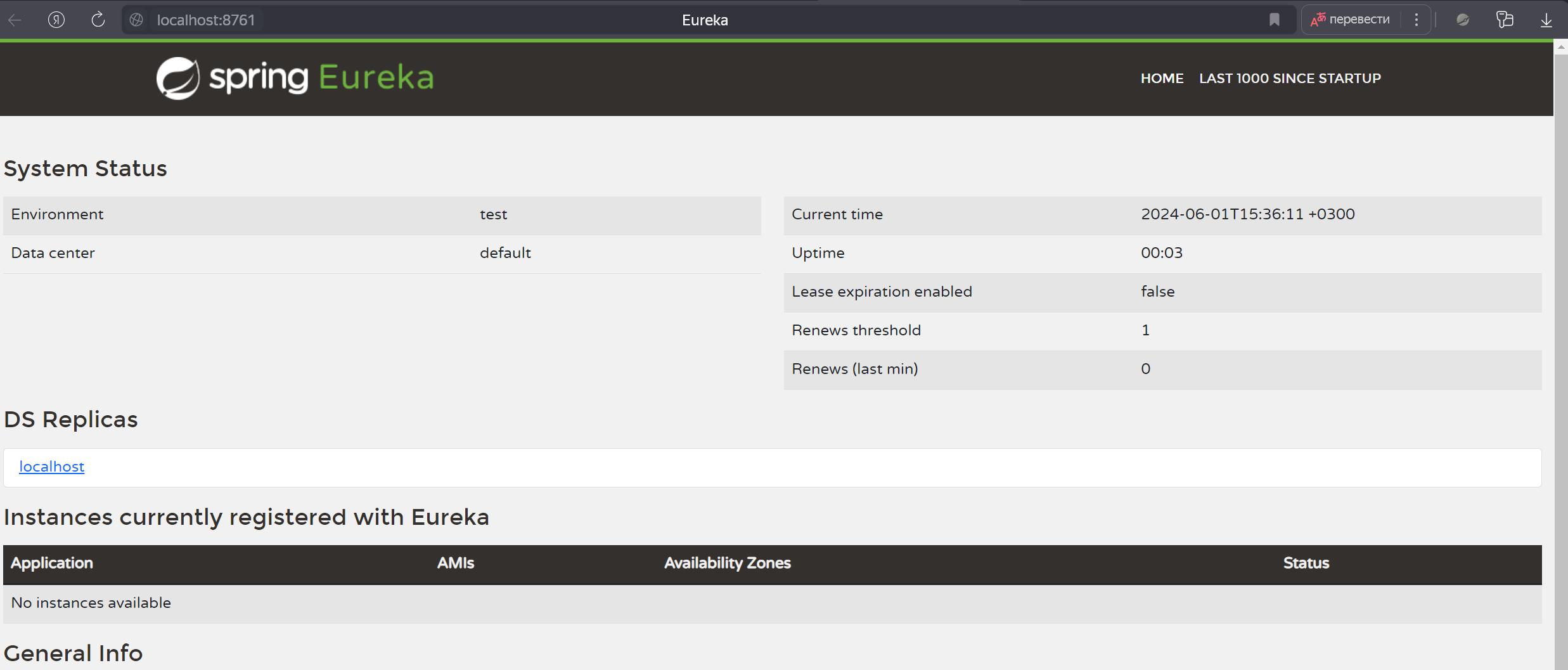


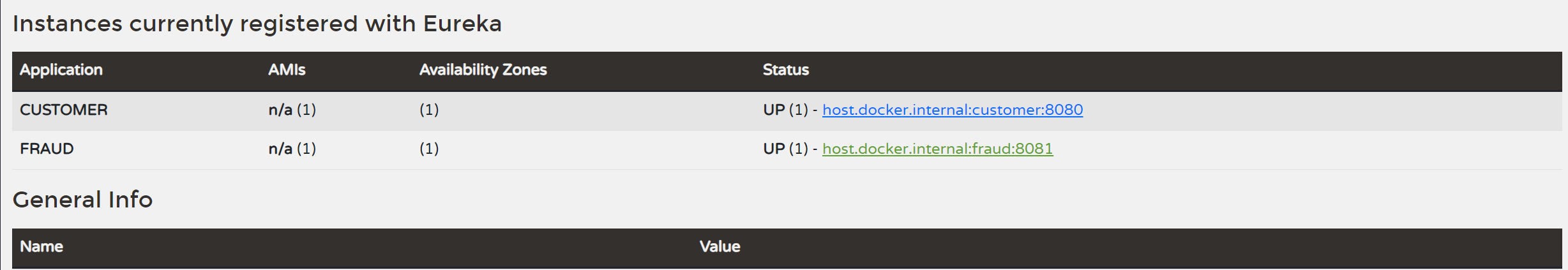
Рисунок 16.

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.

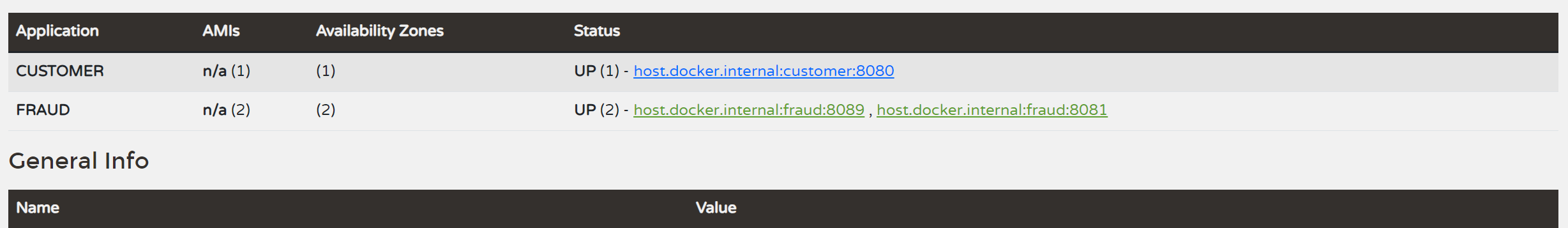
Для подключения Eureka сначала добавим в Dependency management pom файл проекта, зависимость Spring Cloud (2023.0.x). Добавим в проект новый модуль eureka-service. Сконфигурирем модуль как сервер, подключим зависимости, порт 8761 и запустим этот модуль. Через браузер по адресу localhost:8761 увидим панель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

Рисунок 17.

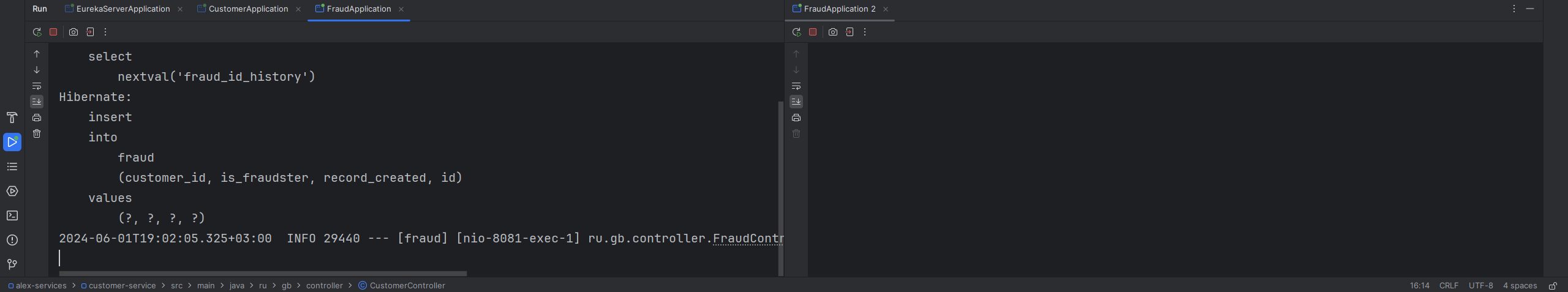
На рисунке 17 видно, что зарегистрированных сервисов пока нет. Поэтому сначала мы зарегистрируем 2 имеющихся микросервиса. Для этого в pom двух модулей добавим зависимости *spring-cloud-starter-netflix-eureka-client*, а также в файлах *application.yml* настроим регистрацию сервисов в eureka. После запуска двух сервисов можно увидеть 2 сервиса, зарегистрированных в eureka

Рисунок 18.

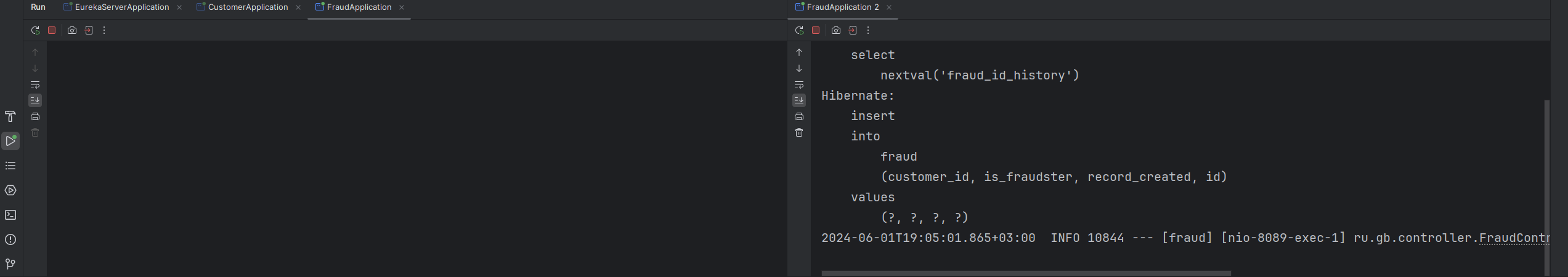
После того как сервисы зарегистрированы, можно использовать название сервиса вместо указания порта в URI при обращении к сервису «fraud». Дополнительно сконфигурируем класс CustomerConfig и добавим аннотацию @LoadBalanced. Запустим сервис eureka, сервис customer и сервис fraud, сделаем имитацию работы 2 сервисов проверки и запустим сервис fraud второй раз на любом свободном порту, я использовал порт 8089.

Рисунок 19.

На рисунке 19 видно что у нас запущено два сервиса fraud. Проверим работу балансировщика нагрузок. Отправим первый запрос на добавление нового «покупателя».

Рисунок 20.

Как видно из рисунка 20, 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 21.

Как видно из рисунка 21, 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 библиотеки 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.

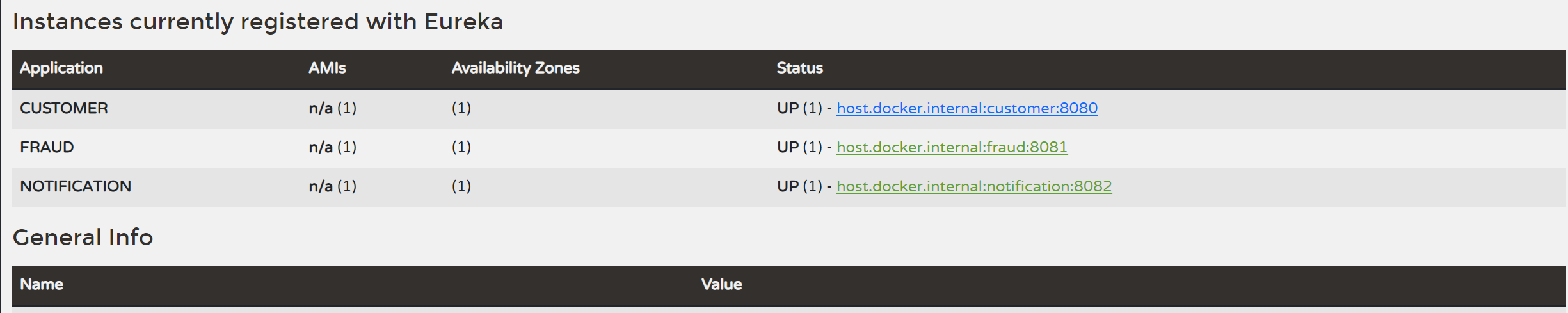
GitHub: [commit 273ac9f](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/273ac9f69b14bf482c1003c74ee6cfd33e232c40)

### 2.2.3 Open Feign

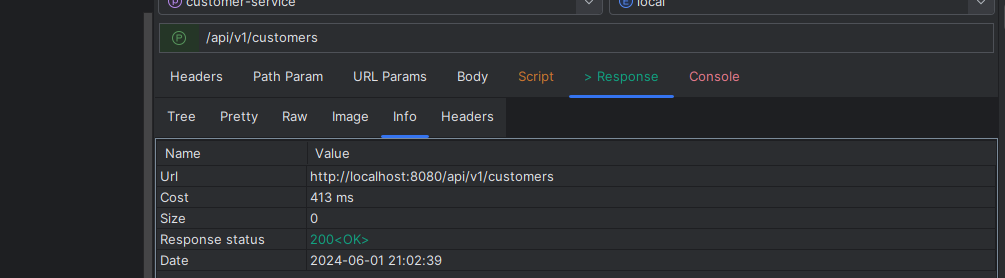
Оба разработанных микросервиса используют один и тот же объект «FraudCheckResponse» и реализуют его внутри своих модулей. Более того, необходимо постоянно помнить что в реализации метода registerCustomer класса CustomerService нам надо использовать сущность и если предположить что сервисов будет более 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». Добавим эту зависимость именно в проект, поскольку нам необходимо, что бы каждый из сервисов, по умолчанию, получал бы эту функциональность. 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 22.

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 23.

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

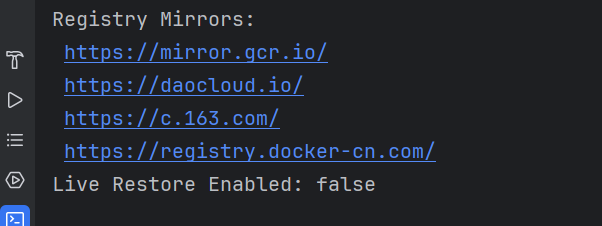
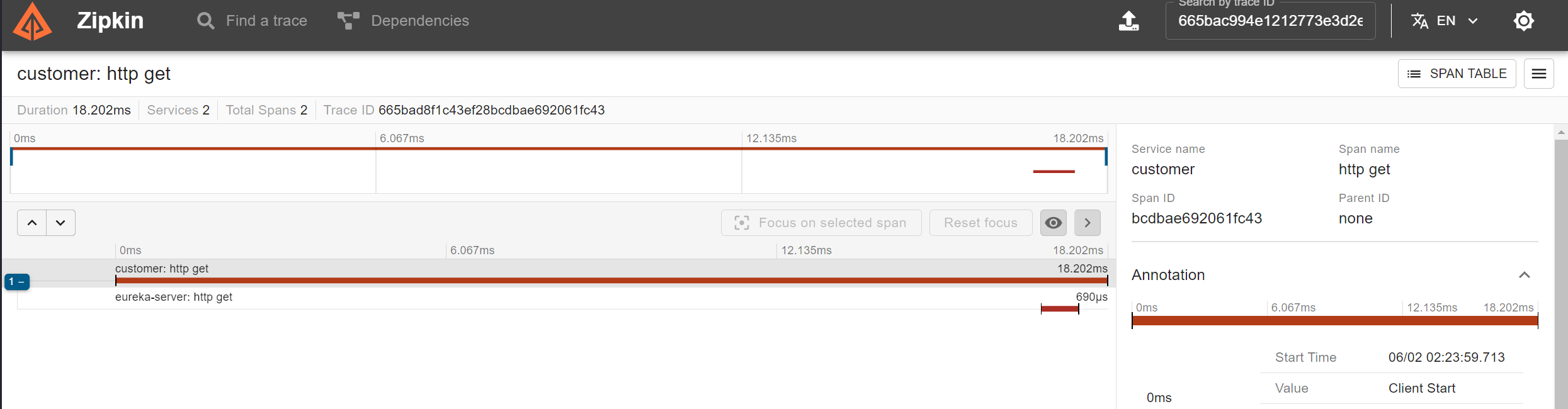


Рисунок 24.

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 25.

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 26.

Рисунок 26.

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.

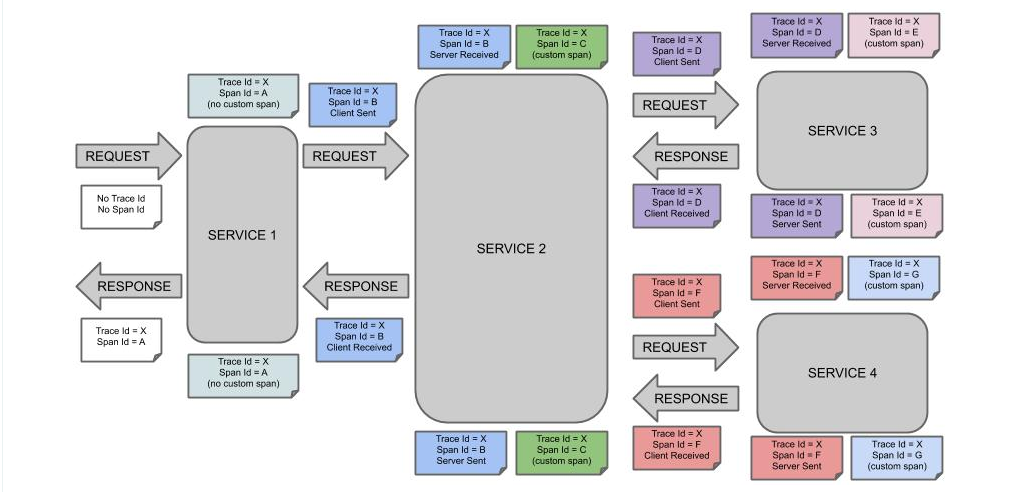


Рисунок 27.

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.

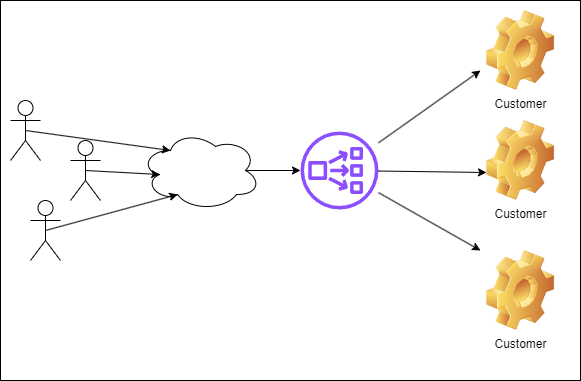
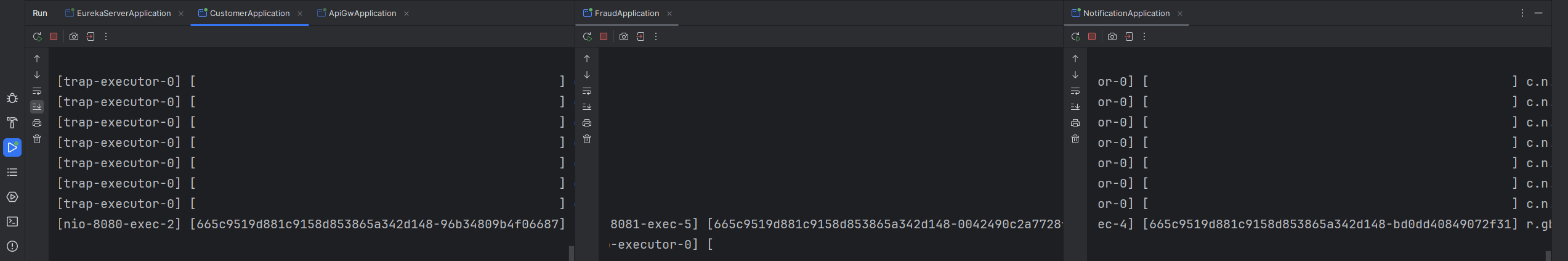
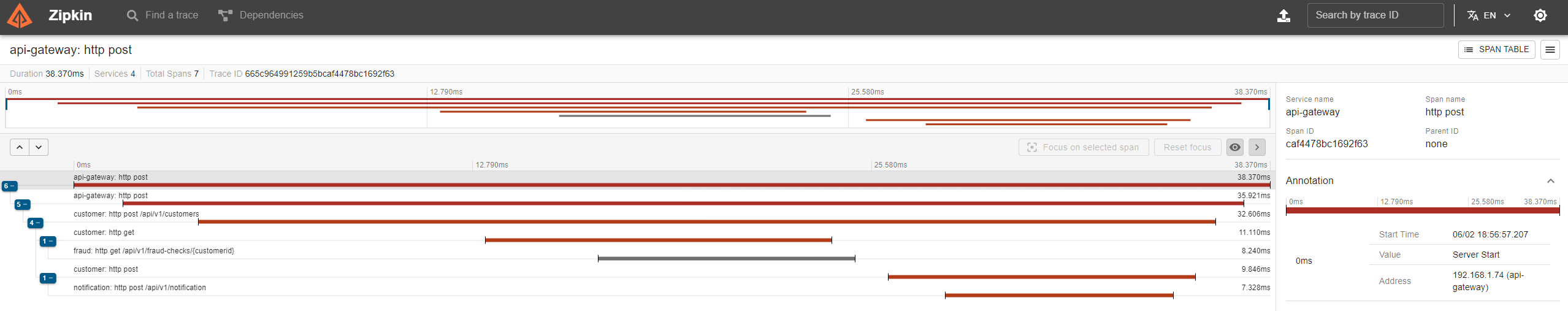


Рисунок 28.

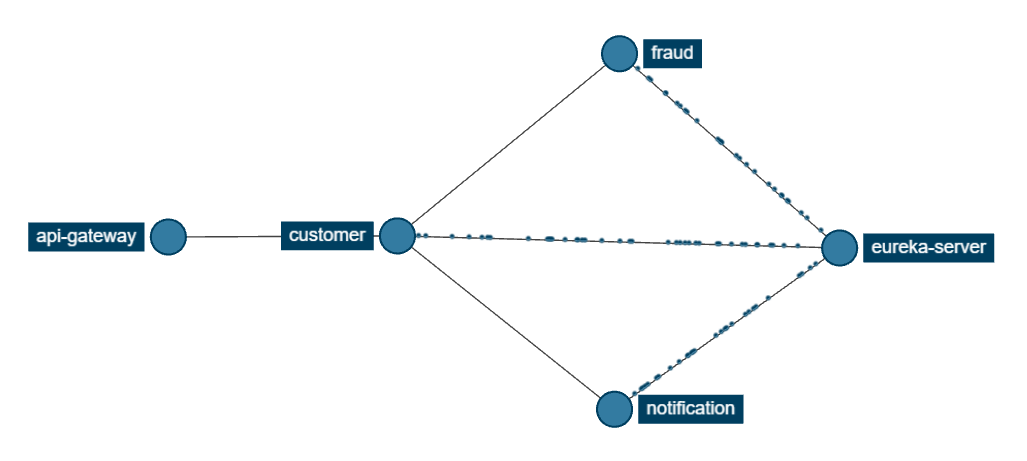
Но, будет подключен единый путь для добавления нового «покупателя» через библиотеку API Gateway. Что позволит обращаться к сервису по заданному нами порту. Для этого добавим в приложение новый модуль apigw. Пропишем путь к сервису регистрации «покупателей» через порт 8083. В дальнейшем все тестовые запросы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

Рисунок 29.

Как видно из рисунка 29 Trace Id сохраняется одинаковым по движению запроса добавления нового «покупателя».

Рисунок 30.

Как видно из рисунка 30 теперь все запросы стартуют с apigw сервиса. И после внедрения API Gateway схема взаимодействия между сервисами выглядит следующим образом

Рисунок 31.

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.

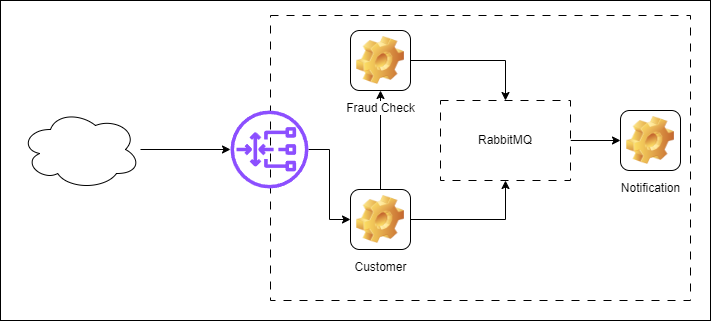


Рисунок 32.

Как видно из рисунка 32, 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…

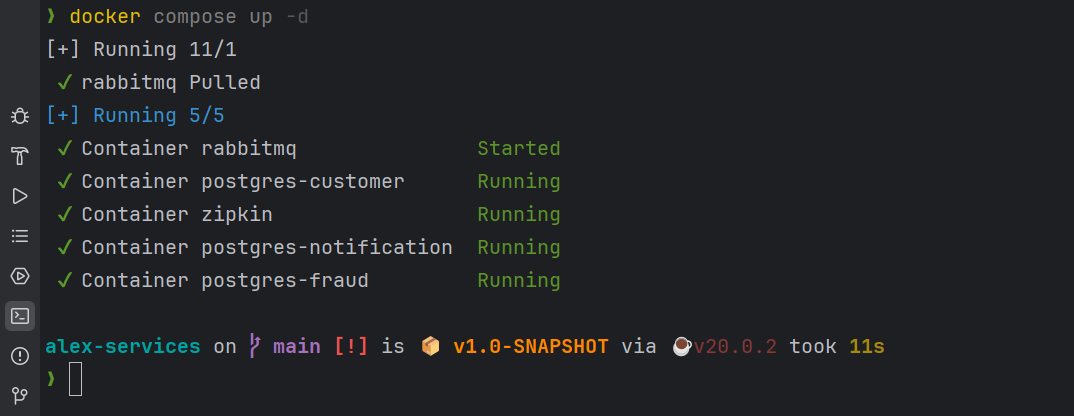
Для начала создадим docker контейнер с приложением. Добавим в файл docker-compose.yml образ rabbitmq:3.13-management-alpine. По указанным выше причинам, использую зеркало для закачки образа на локальную машину, указываем два порта 5672:5672 для обмена сообщениями и второй порт 15672:15672 для взаимодействия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

Рисунок 33.

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.

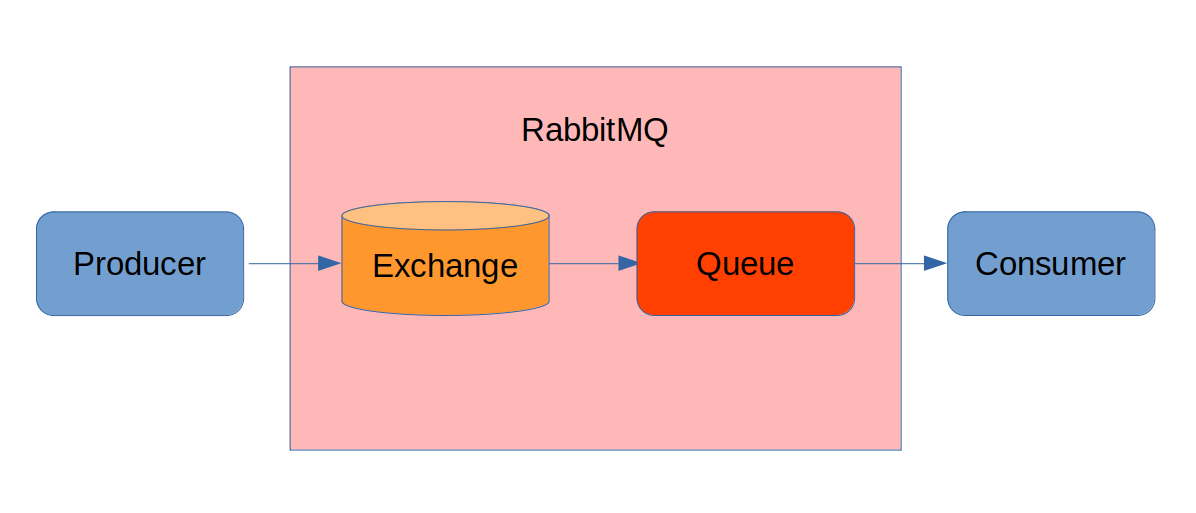
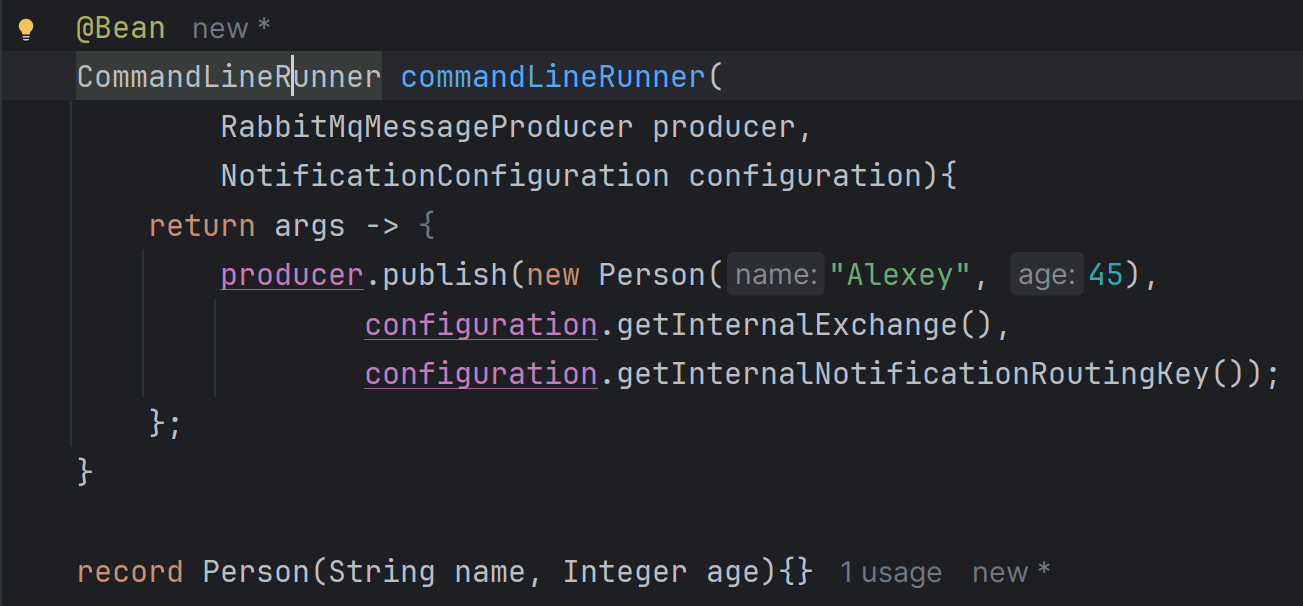
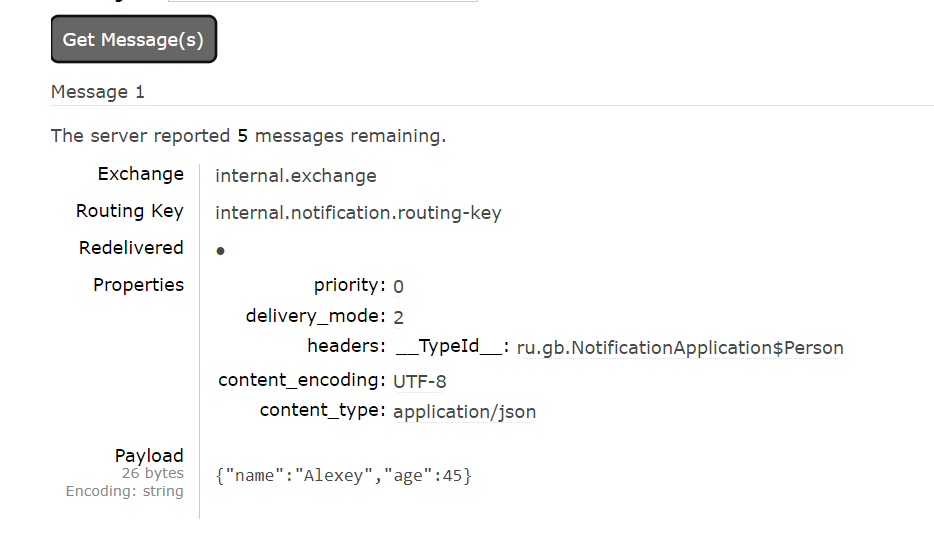


Рисунок 34. Источник — официальный сайт.

На рисунке 34 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 35)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 36).

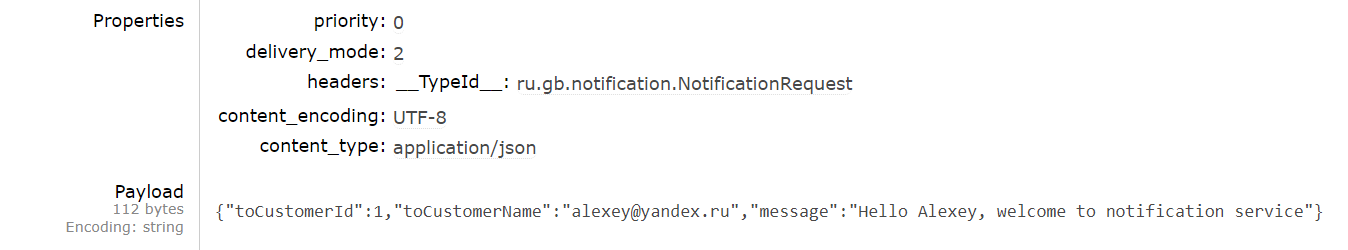
Рисунок 35.

Временный код для проверки работоспособности брокера сообщений.

Рисунок 36.

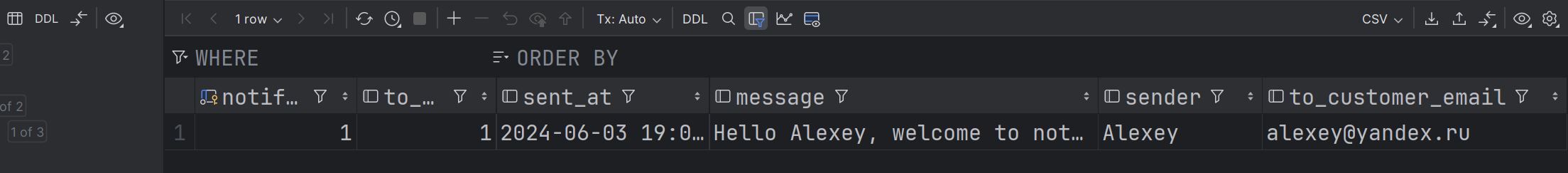
Как видно из рисунка 36, сообщение было успешно добавлено в очередь сообщений. Успешно отработал Object Json конвертер и класс Person был преобразован в JSON формат. Сам сервис работает успешно, поэтому далее необходимо добавить функционал обработки сообщений уже непосредственно от основного сервиса «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 37.

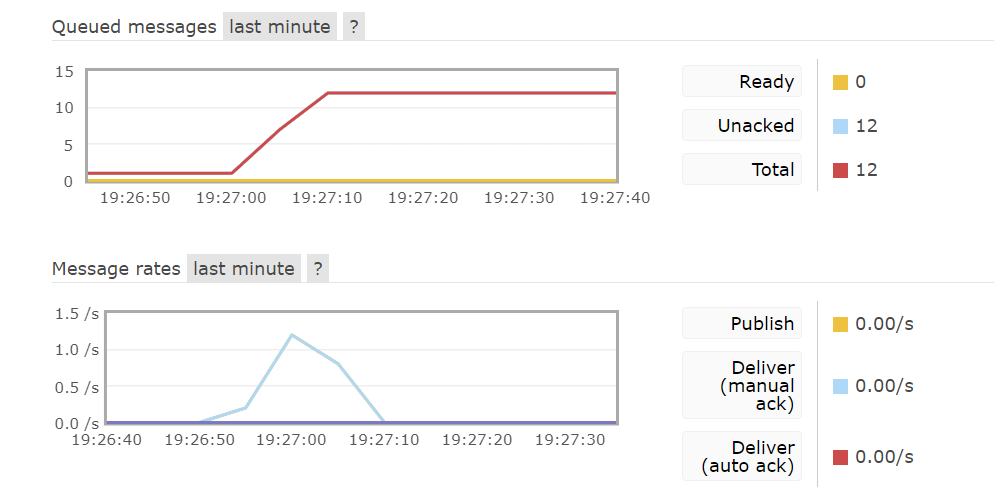
Как видно из рисунка 37, сообщение было успешно доставлено до брокера сообщений и размещено в очередь. Следующим этапом нужно добавить *message listener* в модуль *notification*

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.

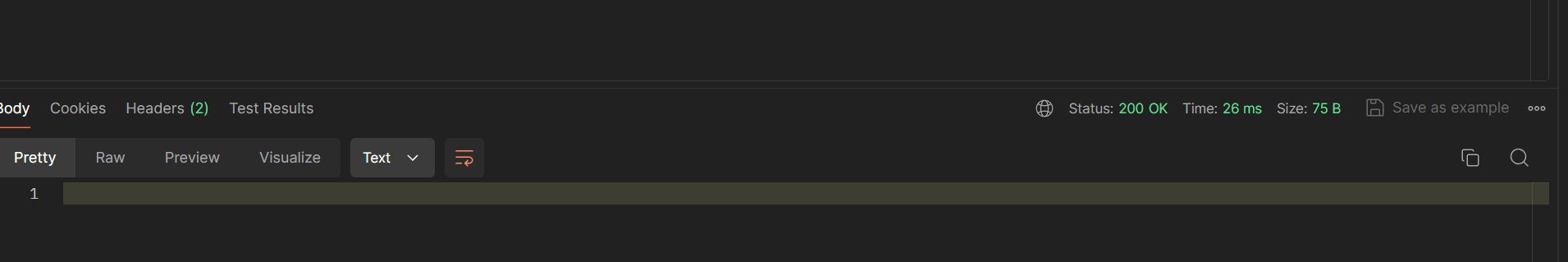
Рисунок 38.

По завершении перезагрузки сообщение пропало из очереди сообщений и было добавлено в базу данных что видно из рисунка 38.

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 39.

Как видно из рисунка 39, было отправлено 12 запросов на добавление «покупателя» в базу данных. 12 сообщений сейчас находятся в очереди сообщений поскольку сервис *notification* был приостановлен.

Рисунок 40.

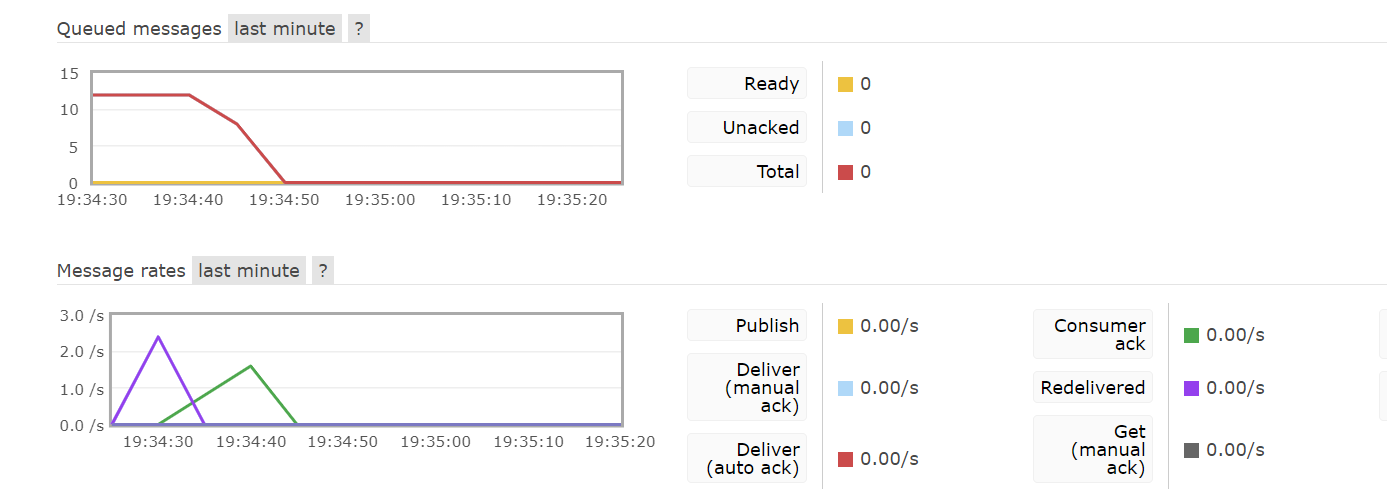
Как видно из рисунка 40 все запросы обработаны и приложение вернуло код 200. Значит, приложение работает так как задумано. После возобновления работы сервиса, все сообщения корректно обработаны и добавлены в базу данных Рисунок 41.

Рисунок 41.

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.

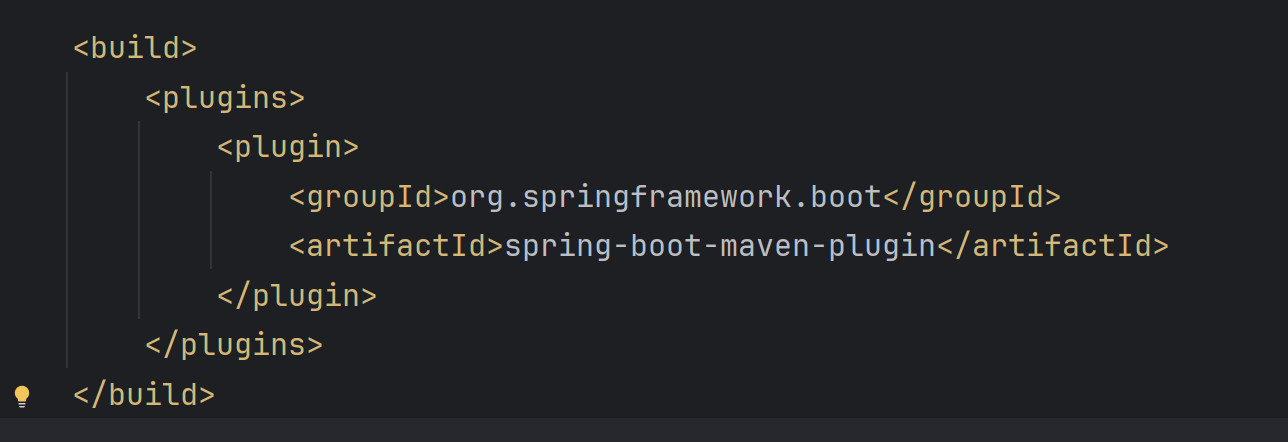
GitHub: [commit a921d58](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/a921d58698c2d6c2129a1790cb25248cf8108a87)

# **Глава 3. Сборка проекта**

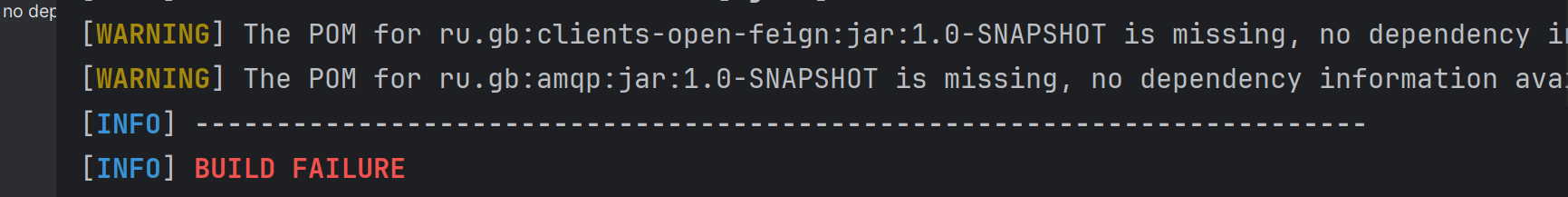
## **3.1 Сборка проекта с использованием Maven Compiler Plugin / Spring boot maven plugin**

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.

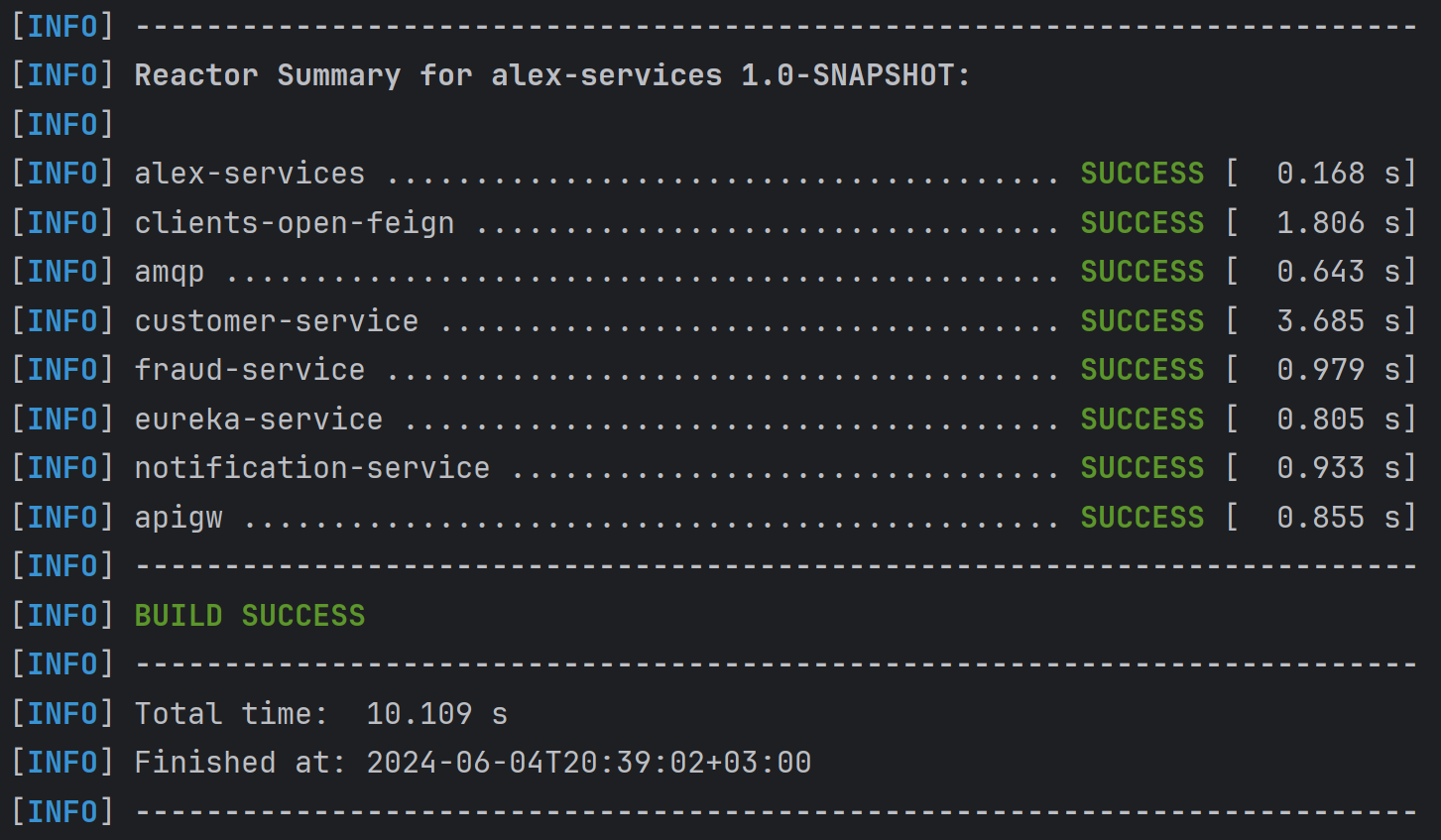
Поскольку остальные модули являются микросервисами Spring, их сборку сделаем с помощью Spring boot maven plugin, который уже подключен к родительскому pom файлу, для того что бы микросервисное приложение работало корректно. Для настройки сборки с помощью этого плагина, укажем выполнение сборки как *repackage*. 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 42.

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 43.

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 44)

Рисунок 44.

Как видно из рисунка 44, все модули приложения успешно установлены. 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)

## **3.2 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).

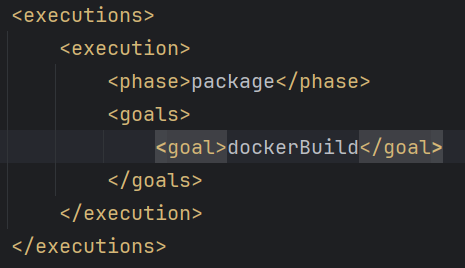
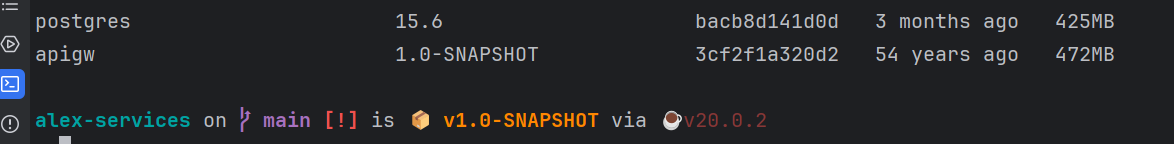
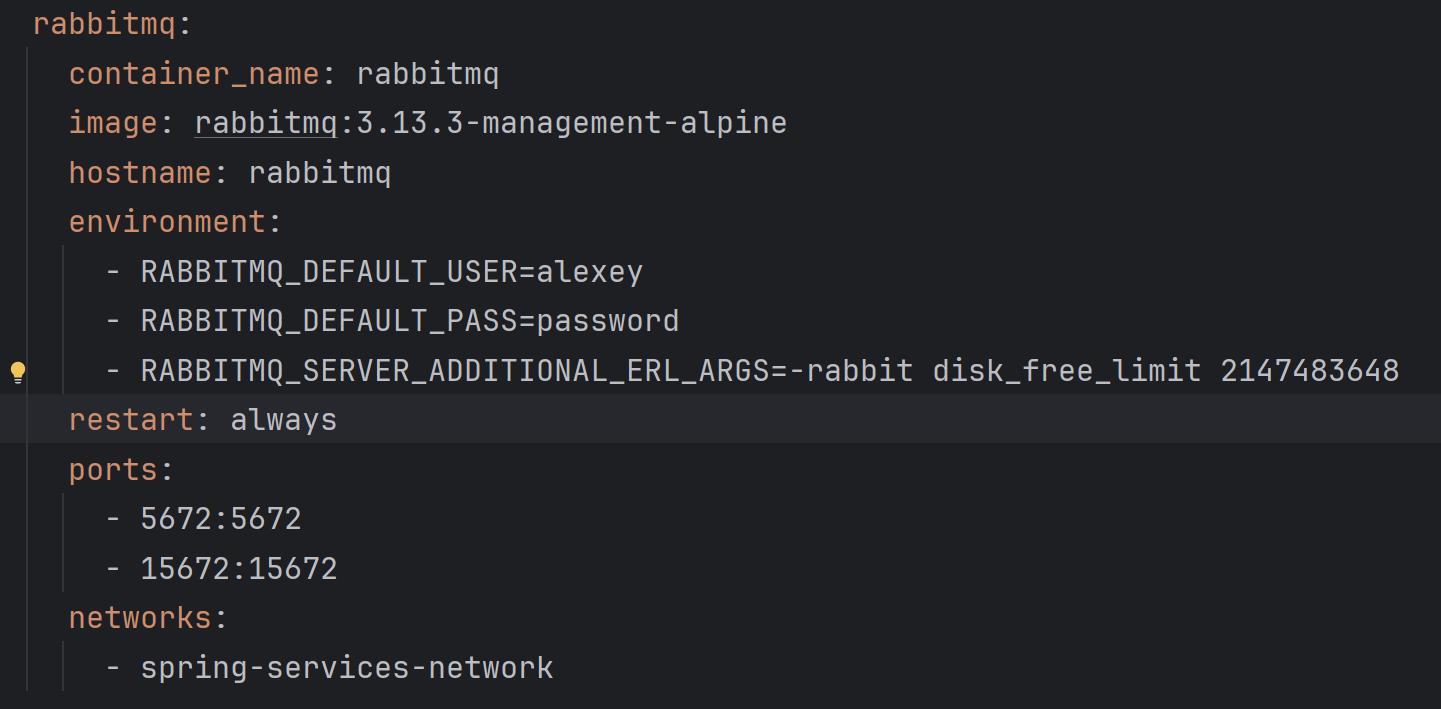
****

Рисунок 45.

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

Рисунок 46.

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 47)

Рисунок 47.

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 48)

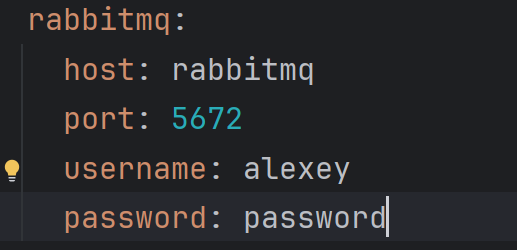
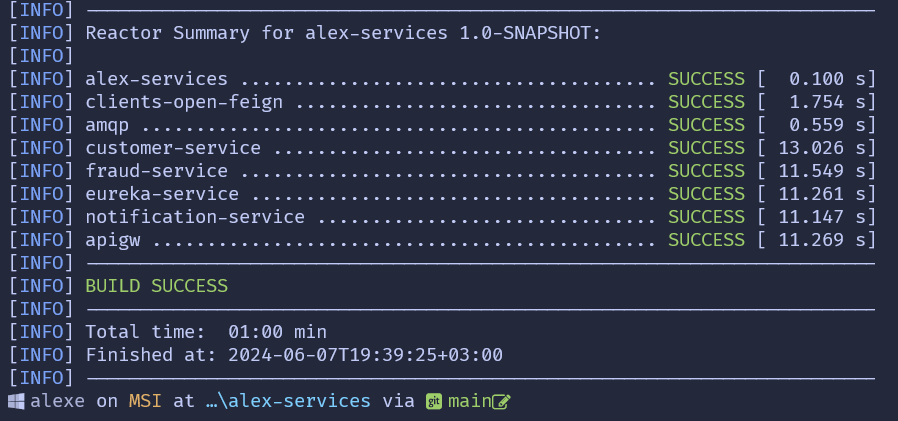
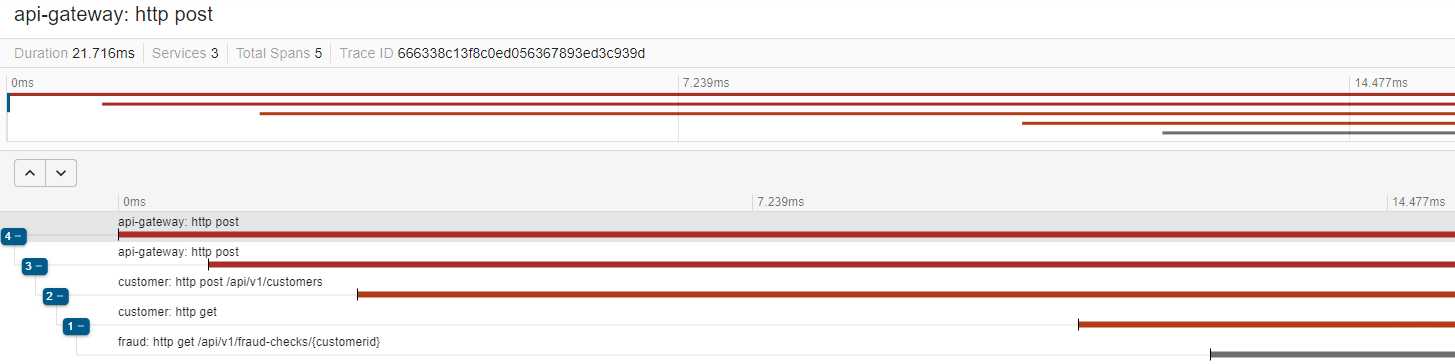


Рисунок 48.

Финально в конфигурационных файлах, для брокера сообщений, можно удалить указание порта 5672, поскольку контейнеры связаны через docker network. После финальных изменений пере собираю сборку и запускаем контейнерное приложение. Для проверки работоспособности добавим нового покупателя и проверим через Zipkin

Рисунок 49.

Рисунок 50.

Приложение работает.

Резюмируя внедрение *RabbitMQ* на базовом уровне можно выделить следующие основные положительные моменты, богатые возможности настройки и простота. К сожалению, по моему сугубо личному мнению, документация не самая обширная. Вопрос работы брокера при работе с сервисами в контейнерах описаны слабо.

GitHub: [commit 5a73715](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/5a737158a7799716476c69b0d4aea63382de4f3b)

# Заключение

## Основные выводы

1. Микросервисная архитектура разработки приложений на мой взгляд не «панацея» и не новое веяние, которое надо использовать при любом удобном случае.
2. Монолитные приложения, выигрывают именно на стадии разработки, в отличие от микросервисов и на мой взгляд тут основное преимущество это скорость внедрения.
3. Преимущества микросервисов становятся очевидными в больших приложениях, когда действительно приложение можно разделить на логические блоки и поддерживать их отдельно.
4. Необходимо четкое понимание на стадии проектирования, какие инструменты понадобятся в микросервисных приложениях. Предположить нагрузки на те или иные узлы, решить вопрос распределения таких нагрузок.
5. Отдельно нужно отметить что микросервисные приложения в своем большинстве связаны с теми или иными сторонними библиотеками, будь то брокеры сообщений, трейсеры и тп. Вопросы лицензирования и поддержки, в таком случае играют также не маловажную роль.
6. Облачные технологии играют немаловажную роль в микросервисных приложениях. Самым ярким примером тут может быть внешний балансировщик нагрузок. В большинстве случаев, внешний балансировщик нагрузок это подключаемый сервис от известных провайдеров и тут нельзя идти на компромисс, поскольку это ключевая точка входа в приложение.
7. Отдельно хочется отметить, что при реализации приложения к каждому из сервисов была подключена своя база данных. Но на практике, согласно рекомендациям из литературы, такой подход недопустим. Т.е. такая практика хороша только в момент разработки и тестирования, но в релизных вервсиях, такой подход должен быть пересмотрен в пользу сторонних подключаемых решений хранения данных.

## **Оценка проведенного исследования**

1. Считаю, что разработка приложения для изучения микросервисной архитектуры и использования стенда для внедрения и тестирования собственных сервисов закончена.
2. Основные взаимодействия между сервисами реализованы и могут быть использованы.

Практическая значимость работы, рекомендации и планы на дальнейшие исследования.

## Планы на дальнейшее развитие приложения

1. Рассмотреть и внедрить деплой проекта с помощью таких служб как containerD и   
   CRI-O
2. Внедрить Kubernetes
3. Рассмотреть и внедрить возможность использования Kafka как брокера сообщений
4. Внедрить автоматизации внедрения сервисов через frontend

# Список используемой литературы

* Микросервисы. Паттерны разработки и рефакторинга Автор: Крис Ричардсон издательство Manning, 2020
* Многопоточное программирование в Java Автор: Тимур Машнин издательство Ridero, 2021
* Запуск сервера Postgres через Docker-compose <https://www.youtube.com/watch?v=NkPvabFLKsg&t=14s>
* Микросервисы: плюсы, минусы, когда и зачем внедрять - https://habr.com/ru/companies/slurm/articles/674600/
* eureka-server - https://cloud.spring.io/spring-cloud-netflix/multi/multi\_spring-cloud-eureka-server.html
* Rest template - https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html
* Quick Guide to Micrometer - <https://www.baeldung.com/micrometer>
* spring cloud open-feign - https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/
* spring cloud open-feign - <https://github.com/OpenFeign/feign?tab=readme-ov-file>
* zipkin tracers and instrumentation - <https://zipkin.io/pages/tracers_instrumentation.html>