Geekbrains

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

Программа: Разработчик - Программист

Специализация: Веб-разработка на JAVA

Зотов Алексей Викторович

Москва

2024

**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. |

**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 небольших масштабируемых приложений с использованием микросервисной архитектуры

**Цель:** Рассмотреть основные подходы к дизайну и развертыванию приложений и используемому инструментарию. 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 инструментарии и определится с инструментарием. Будь то сервис уведомлений, аутентификации и так далее. Предполагаемое к разработке приложение планируется подготовить максимально абстрактным с целью легкой подмены шаблонных сервисов на собственные.

**Задачи:**

1. Изучение литературы

2. Создать базовый проект

3. Выбрать инструментарий для разработки, мониторинга и управления микросервисным приложением

4. Изучить инструментарий принципы внедрения и внедрить в базовое приложение

5. Изучить процесс развертывания таких приложений с использованием Docker

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

Состав команды: Зотов Алексей Викторович (программист)

# **Глава 1. Теоретическая часть**

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. Выявить плюсы и минусы данного подхода. Далее внедрить Eureka. По сути посмотреть работу «записной книжки».

Двигаясь далее затронуть такой сервис как Open Feign и попытаться сравнить его с подходом с использованием RestTemplate и на этом же этапе внедрить третий сервис, сервис уведомлений. Этот этап подразумевает внедрение в проект таких технологий как Spring Cloud Sleuth 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 сервиса и насколько это возможно уйти от деталей именно в самих сервисах и просто обозначить их функционал

# **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)

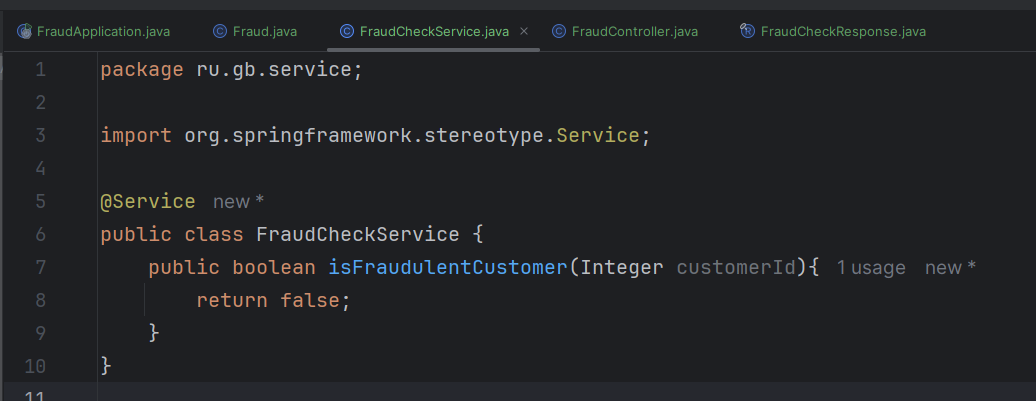
**2.1.2 Создание сервисных приложений**

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.

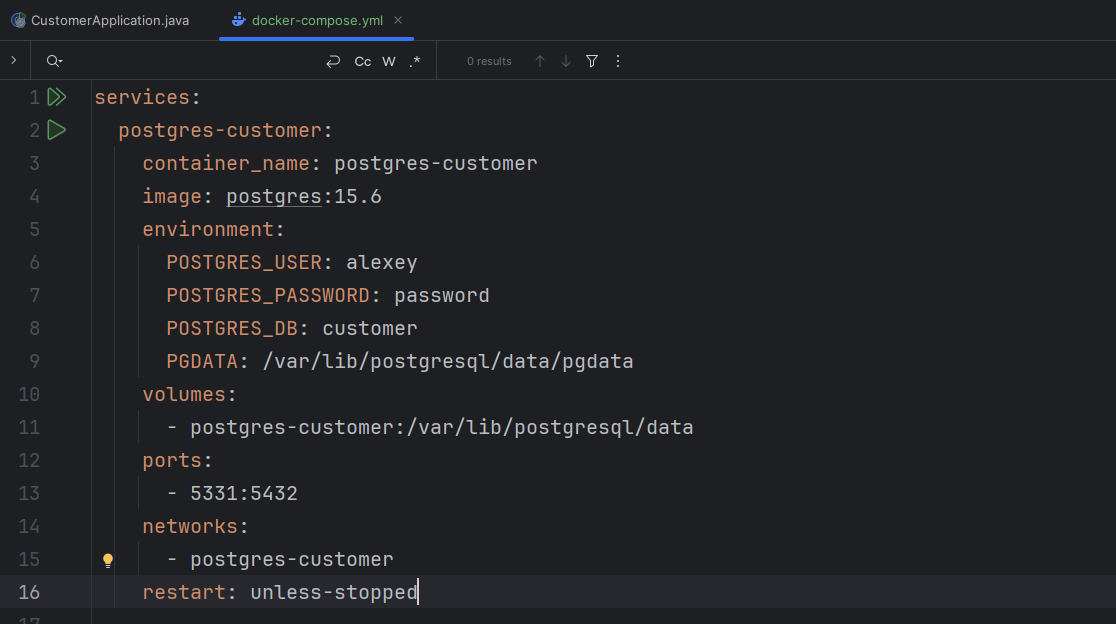
Поскольку данная работа в первую очередь направленна не на обработку персональных данных, а разработку взаимодействия между микросервисами, было решено максимально абстрагировать процесс проверки (рис. 2)

Рисунок 2

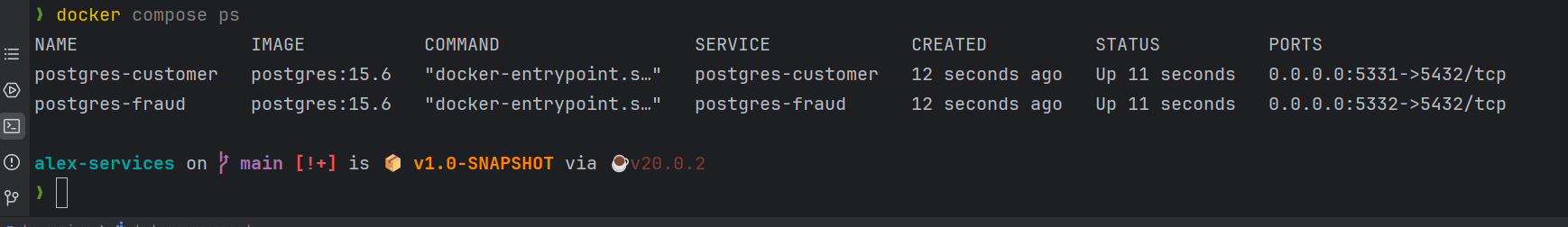
Помимо базовой настройки самих сервисов, мы добавим каждому микросервису отдельный порт, что бы избежать конфликтов. На этом этапе оба сервиса подготовлены для интегрирования с базами данных. [*Git Commit: 102d4d7*](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/102d4d7e9f4524be9ef9d2f30fff18dbf918f03c)

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

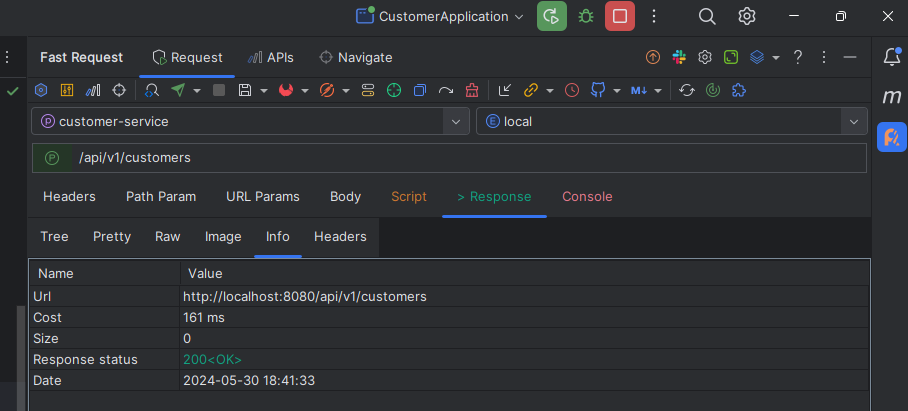
В корневом каталоге создадим файл docker-compose.yml в котором я настрою создание двух отдельных контейнеров для сервисов «customer» и «fraud». Версия PostgreSQL 15.6 (рис. 3)

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)

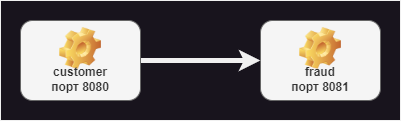
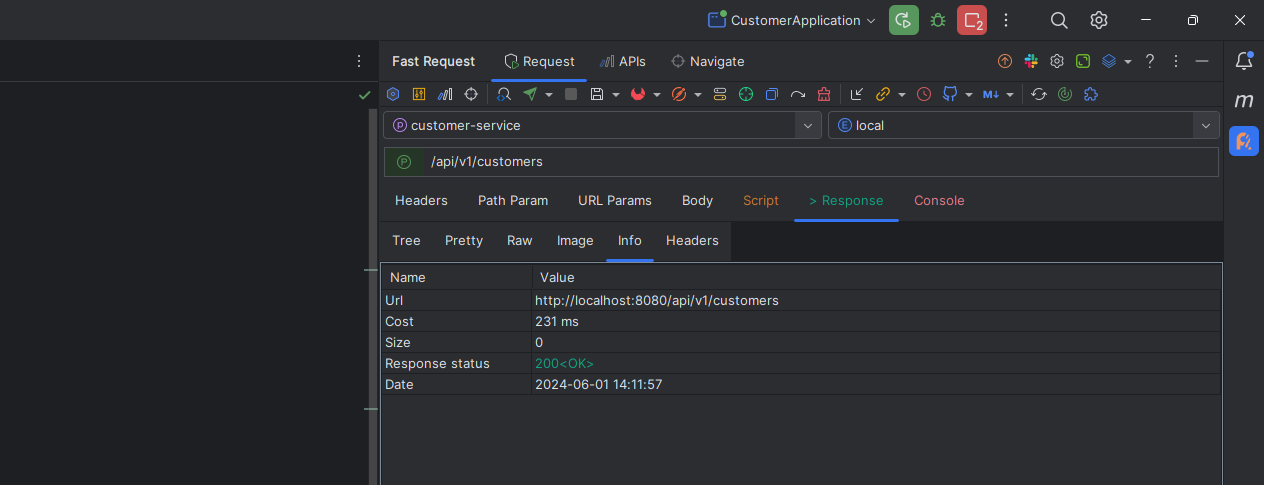


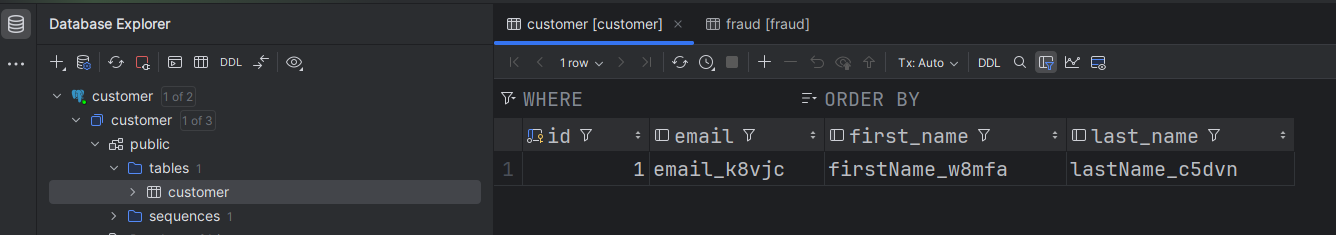
Диаграмма 1

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.

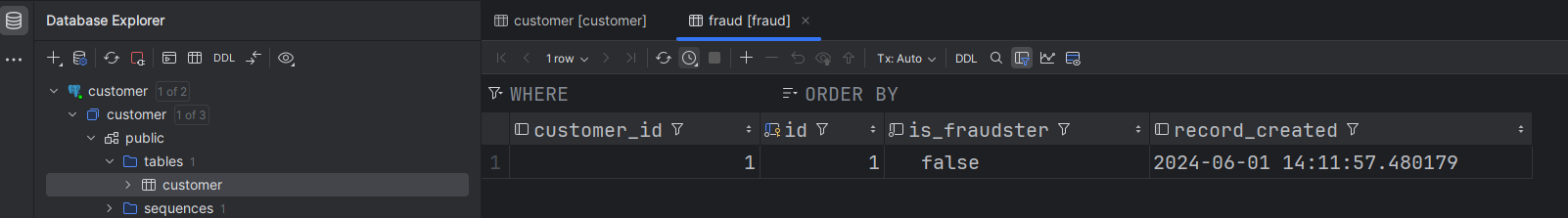
Далее в сервисе добавления нового «покупателя» мы обращаемся к RestTemplate через метод getForObject который направляет GET запрос на указанный URI и возвращает запрашиваемый объект. URI это полный путь к запросу GET fraud контроллера. Тут мы видим несколько неудобных моментов. Первое, это четкое указание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

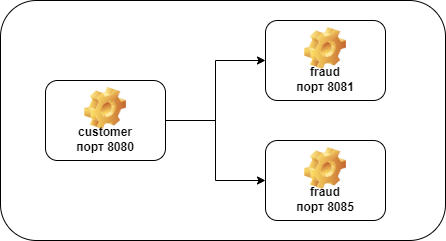
При использовании данного подхода выявлен ряд существенных недостатков, которые нам не позволят универсализировать программу, для подключения сторонних сервисов:

* 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).

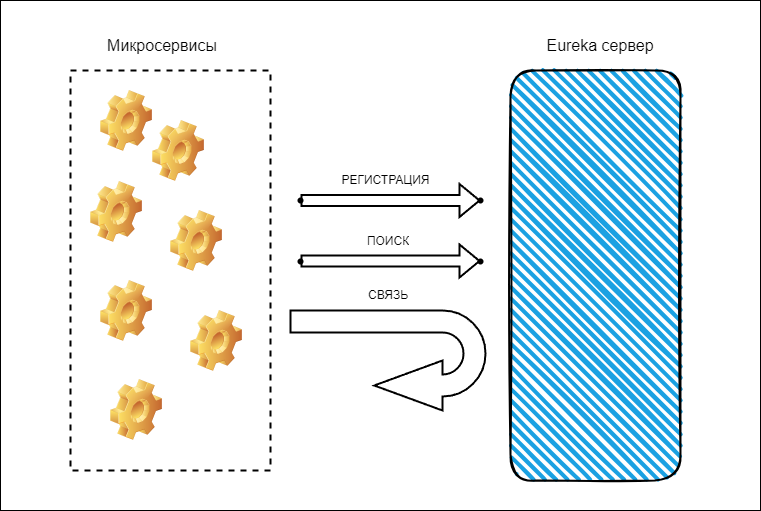
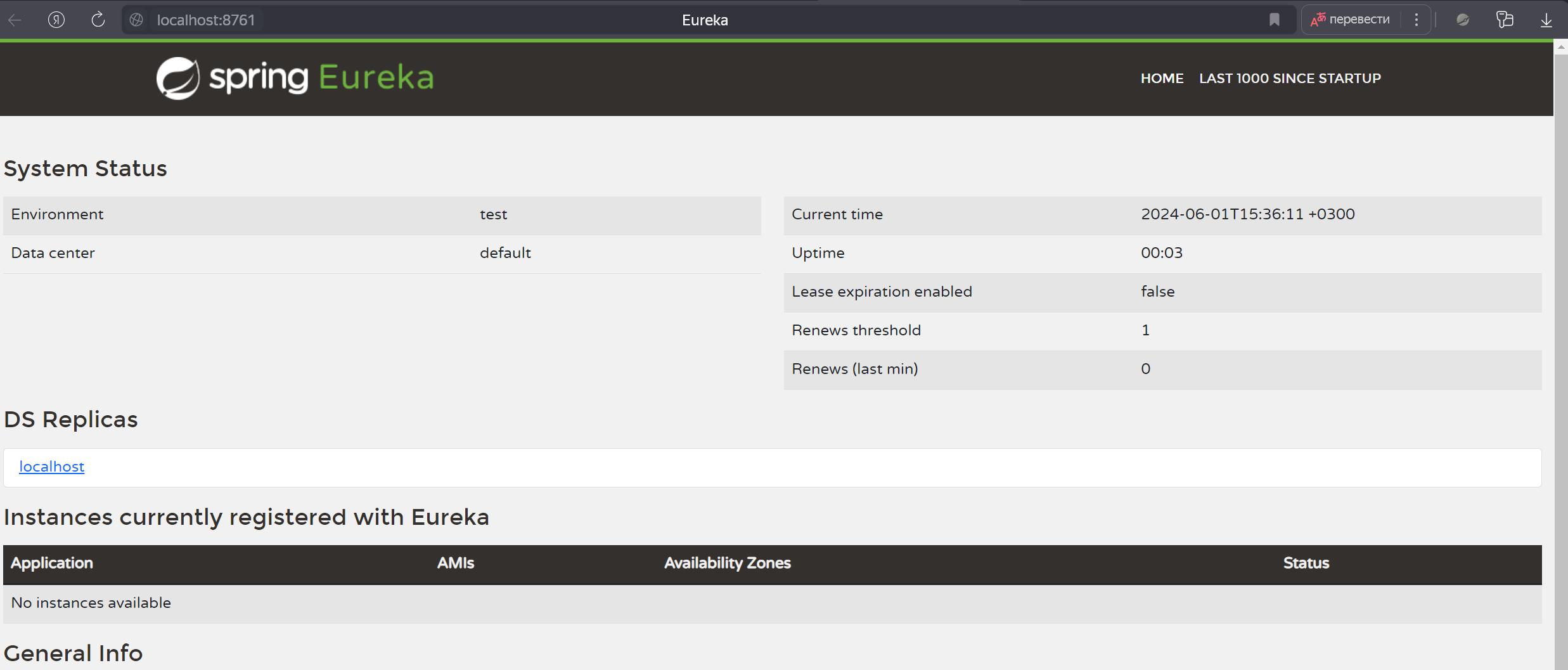


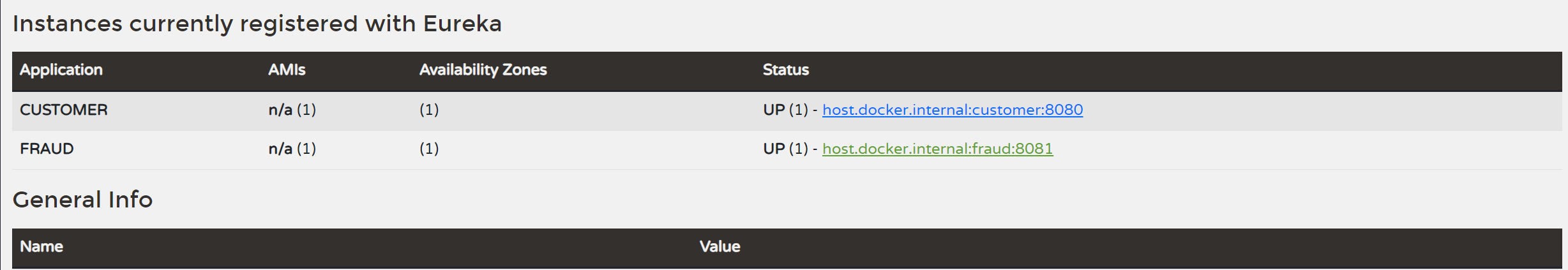
Рисунок 10

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.

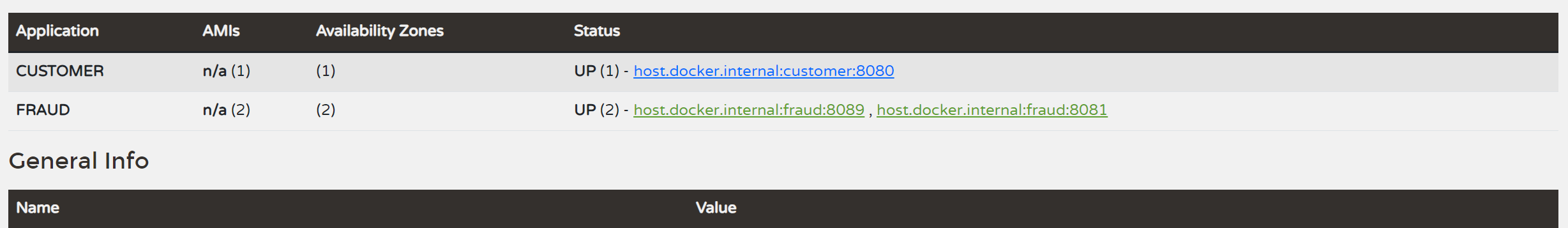
Для подключения Eureka сначала добавим в Dependency management pom файл проекта, зависимость Spring Cloud (2023.0.x). Добавим в проект новый модуль eureka-service. Сконфигурирем модуль как сервер, подключим зависимости, порт 8761 и запустим этот модуль. Через браузер по адресу localhost:8761 увидим панель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

Рисунок 11

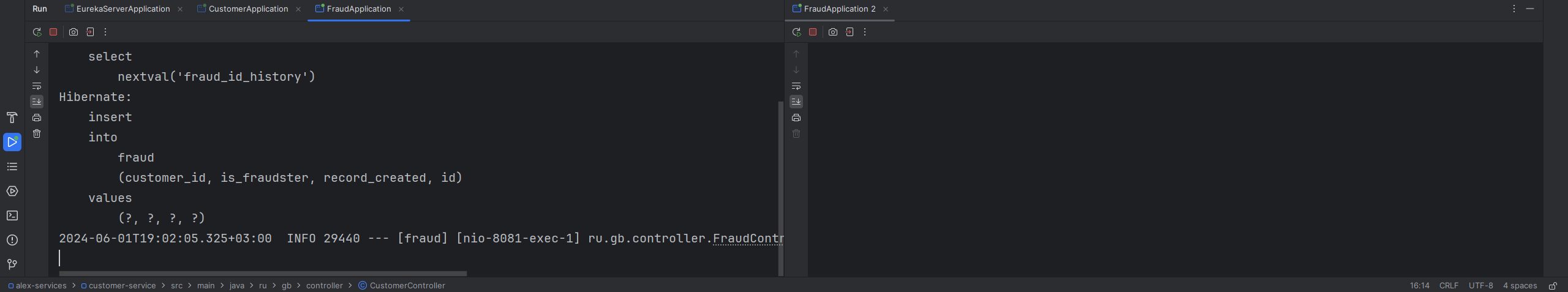
На рисунке 11 видно, что зарегистрированных сервисов пока нет. Поэтому сначала мы зарегистрируем 2 имеющихся микросервиса. Для этого в pom двух модулей добавим зависимости *spring-cloud-starter-netflix-eureka-client*, а также в файлах *application.yml* настроим регистрацию сервисов в eureka. После запуска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

Рисунок 12

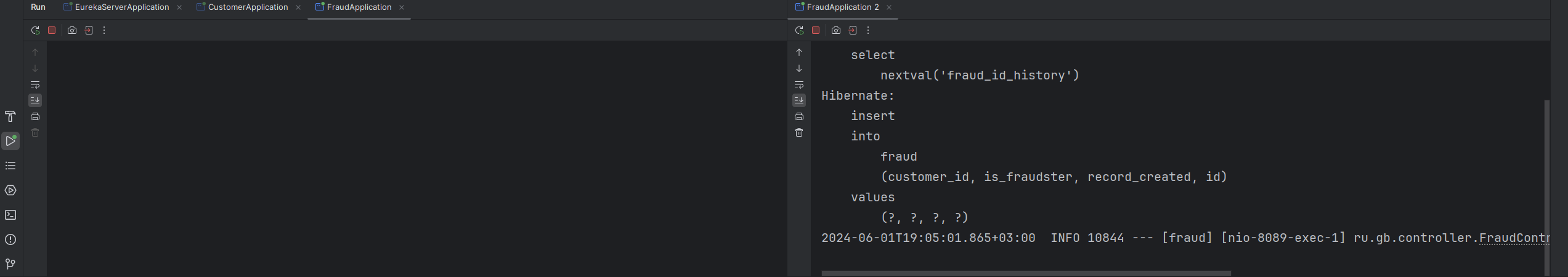
После того как сервисы зарегистрированы, можно изспользовать название сервера вместо указания порта в URI при обращении к сервису «fraud». Дополнительно сконфигурируем класс CustomerConfig и добавим аннотацию @LoadBalanced. Запустим сервис eureka, сервис customer и сервис fraud, сделаем имитацию работы 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.

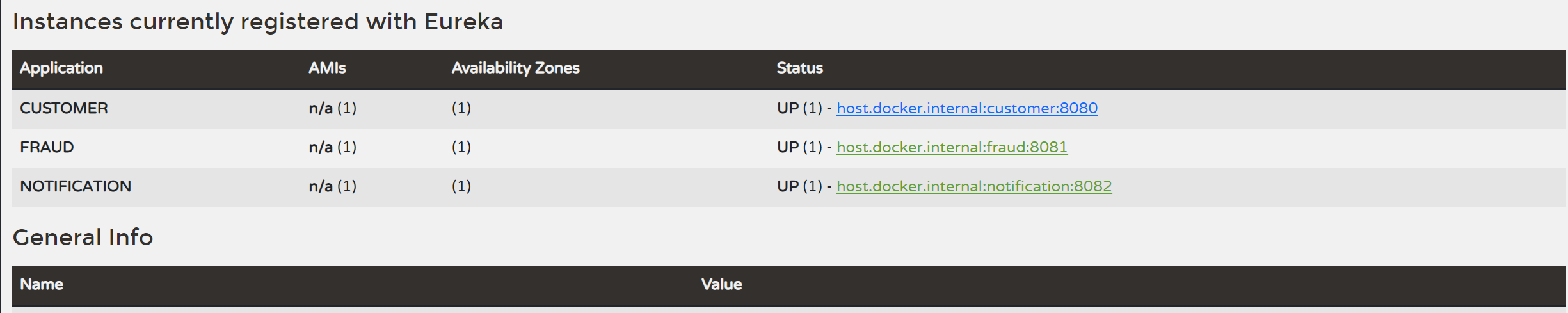
GitHub: [commit 273ac9f](https://github.com/AVZotov/Tech-Specialization-diploma-microservices/commit/273ac9f69b14bf482c1003c74ee6cfd33e232c40)

**2.2.3 Open Feign**

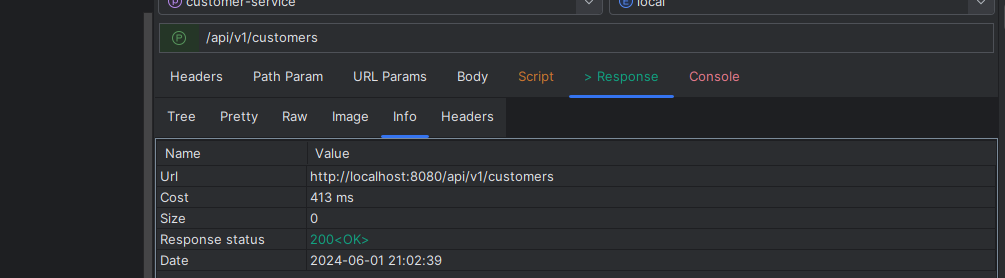
Оба разработанных микросервиса используют один и тот же объект «FraudCheckResponse» и реализуют ее внутри. Более того, необходимо постоянно помнить что в реализации метода registerCustomer класса CustomerService нам надо использовать сущность и если предположить что сервисов будет более чем 2 или 3, 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 минимизировать зависимости 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 Micrometr 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

# 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.