МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Тверской государственный технический университет»

(ТвГТУ)

Кафедра «Программного обеспечения»

ДОПУСТИТЬ К ЗАЩИТЕ

ЗАВЕДУЮЩИЙ КАФЕДРОЙ

\_\_\_\_\_\_\_\_\_\_\_\_\_А. Л. КАЛАБИН

«\_\_\_\_\_»\_\_\_\_\_\_\_\_\_20\_\_\_Г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

На тему: Разработка серверной части интернет магазина «Электроника» на основе микросервисной архитектуры

Направление: 09.03.04- Программная инженерия

Профиль: Разработка программно-информационных систем

Студент (ка): Иванов Роман Вячеславович

(Ф.И.О.) (подпись)

Форма обучения: очная Группа: Б.ПИН.РИС. 17.06

Руководитель: канд. тех. наук, доцент, Мальков Александр Анатольевич (ученая степень, звание, фамилия, инициалы) (подпись)

Сильченко Александр Фёдорович

Нормоконтроль (ученая степень, звание, фамилия, инициалы) (подпись)

Тверь 2021

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Тверской государственный технический университет»

(ТвГТУ)

З А Д А Н И Е

на выпускную квалификационную работу бакалавра

Студент(ка) Иванов Роман Вячеславович (фамилия, имя, отчество)

Направление подготовки бакалавров 09.03.04- Программная инженерия (код и наименование)

Профиль: Разработка программно-информационных систем

Тема Разработка серверной части интернет магазина «Электроника» на основе микросервисной архитектуры

(утверждена приказом ректора от « 26 » апреля 2021 г. № 286-У ).

Дата выдачи задания « 20 » января 2021 г.

Срок представления студентом ВКР на кафедру: «\_\_15\_\_\_»\_\_\_\_июня\_\_\_ \_2021\_\_г.

Руководитель: канд. тех. наук, доцент, Мальков Александр Анатольевич (ученая степень, звание, фамилия, инициалы) (подпись)

Задание получено: Иванов Роман Вячеславович

(Ф.И.О.) (подпись)

Тверь 2021

Аннотация

Дипломный проект на тему: Разработка серверной части интернет магазина «Электроника» на основе микросервисной архитектуры содержит 54 страницы текста, 23 рисунка, 15 таблиц , 8 – текстовых блоков с примерами настроек приложения, 9 – источников .

Объект исследования – деятельность интернет магазина.

Предмет – основной функционал интернет магазина.

Целью - разработка серверной части интернет магазина «Электроника» на основе микросервисной архитектуры.

Дипломный проект состоит из введения, четырех глав, заключения и списка используемой литературы.

Во введении раскрывается актуальность выбранной темы , описываются цели и задачи работы.

В первой главе изложена техническая часть, а именно описание предметной области и разработка технического задания.

Вторая глава посвящена технологическим аспектам, необходимые для реализации программного продукта. В частности, производится выбор среды разработки, СУБД, архитектуры приложения, языка программирования, библиотек и архитектуры API. В каждом подразделе обосновывается выбор того или иного технического решения задачи.

В третей глава рассказывается о реализации каждого из сервисов. А именно описывается его API, структура базы данных и взаимодействие с другими сервисами. Заключением данной главы является показ работы системы , а также результаты тестирования.

В четвертой главе изложена информация о документирование системы

В заключении обобщается проделанная работа и формулируются выводы о проделанной работе

Оглавление

[Перечень принятых сокращений 6](#_Toc74498355)

[Введение 7](#_Toc74498356)

[Цель 8](#_Toc74498357)

[Задачи 8](#_Toc74498358)

[Глава 1. Техническая часть 8](#_Toc74498359)

[Описание предметной области. 8](#_Toc74498360)

[Разработка технического задания 9](#_Toc74498361)

[Глава 2. Специальная часть 10](#_Toc74498362)

[Анализ предметной области 10](#_Toc74498363)

[Выбор среды разработки 11](#_Toc74498364)

[Анализ и выбор СУБД 12](#_Toc74498365)

[Выбор реляционной СУБД 13](#_Toc74498366)

[MS SQL Server 13](#_Toc74498367)

[Oracle Database 14](#_Toc74498368)

[PostgreSQL 15](#_Toc74498369)

[MySQL 15](#_Toc74498370)

[Выбор архитектуры приложения 16](#_Toc74498371)

[Монолитная архитектура приложения 16](#_Toc74498372)

[Микросервисная архитектура приложения 17](#_Toc74498373)

[Выбор средств проектирования 19](#_Toc74498374)

[Выбор языка программирования 19](#_Toc74498375)

[Язык программирования Go 19](#_Toc74498376)

[Язык программирования Python 19](#_Toc74498377)

[Язык программирования JavaScript 20](#_Toc74498378)

[Язык программирования Java 20](#_Toc74498379)

[Язык программирования Kotlin 20](#_Toc74498380)

[Выбор языка программирования 21](#_Toc74498381)

[Выбор библиотек для разработки 21](#_Toc74498382)

[Модули Spring для разработки приложения 22](#_Toc74498383)

[Spring MVC 22](#_Toc74498384)

[Spring JPA 23](#_Toc74498385)

[Spring Security 23](#_Toc74498386)

[Spring Cloud Eureka 23](#_Toc74498387)

[Описание архитектуры REST API 23](#_Toc74498388)

[Модель клиент-сервер 23](#_Toc74498389)

[Отсутствие состояния 24](#_Toc74498390)

[Кэширование 24](#_Toc74498391)

[Единообразие интерфейса. 24](#_Toc74498392)

[Многослойная система 25](#_Toc74498393)

[Глава 3. Проектная часть 25](#_Toc74498394)

[Проектирование 25](#_Toc74498395)

[Реализация 26](#_Toc74498396)

[Gateway. 26](#_Toc74498397)

[Eureka. 29](#_Toc74498398)

[Authorization service 32](#_Toc74498399)

[Структура БД 35](#_Toc74498400)

[REST API 36](#_Toc74498401)

[Product service 38](#_Toc74498402)

[REST API 38](#_Toc74498403)

[Структура БД 45](#_Toc74498404)

[Recommendation service 46](#_Toc74498405)

[REST API 46](#_Toc74498406)

[Структура БД 47](#_Toc74498407)

[Review service 48](#_Toc74498408)

[REST API 48](#_Toc74498409)

[Структура БД 49](#_Toc74498410)

[Cart service 50](#_Toc74498411)

[REST API 50](#_Toc74498412)

[Структура БД 51](#_Toc74498413)

[Order service 52](#_Toc74498414)

[REST API 52](#_Toc74498415)

[Структура БД 53](#_Toc74498416)

[Тестирование и результаты 54](#_Toc74498417)

[Глава 4. Документация 59](#_Toc74498418)

[KDoc. 59](#_Toc74498419)

[Развертывание системы 60](#_Toc74498420)

[Заключение 62](#_Toc74498421)

[Используемый источники 63](#_Toc74498422)

# Перечень принятых сокращений

API (Application Programming Interface) – программный интерфейс приложения.

JVM (Java Virtual Machine) - Виртуальная машина Java, которая служит для исполнения Java байт-кода.

OS – (Operation System) – Операционная система

GC – (Garbage Collector) – Сборщик мусора

HTTP (HyperText Transfer Protocol) – протокол передачи гипертекста.

ID (Identifier) – идентификатор.

JSON (JavaScript Object Notation) – текстовый формат обмена данными, основанный на JavaScript.

REST (Representational State Transfer) – передача состояния представления.

SQL (Structured Query Language) – язык структурированных запросов.

UML (Unified Modeling Language) – унифицированный язык моделирования.

БД – База данных.

# Введение

В настоящее время покупки через интернет являются неотъемлемой частью жизни современного человека. Различные фирмы создают свои уникальные сайты чтоб привлеч покупателей, а так же облегчить поиск нужного для них товара. Чтобы обеспечить бесперебойную и стабильную работу магазина, многие разработчики используют различные подходы к организации архитектуры приложения.

Одной из самых распространенных и удобных архитектур является микросервисная архитектура. Удобство заключается в том, что приложение разбито на функциональные блоки – микросервисы, каждый из которых может масштабироваться, тем самым распределяя нагрузку, повышая отказоустойчивость всего приложения.

В данной работе рассматривается реализация серверной части интернет магазина.Каждый из микросервисов отвечаете за функцианал определенной части системы. В контексте данной работы не рассматривается реализация фронтенд части приложения.

## Цель

Целью работы реализовать основной функционал серверной части интернет магазина «Электроника» с предоставлением API для дальнейшей разработки.

## Задачи

1. Исследовать предметную область
2. Спроектировать приложение
3. Разрабоать приложения используя микросервисную архитектуру
4. Разработать и отладить прикладной интерейс (REST API) для каждого сервиса
5. Наладить взаимодействие микросервисов между собой
6. Обеспечить приложение должным уровнем безопасности
7. Провести тестирование и отладку

# Глава 1. Техническая часть

## Описание предметной области.

В ходе выполнения работы рассматривается деятельность интернет-магазина электроники. Рассмотрим основные аспекты работы интернет-магазина.

Любой пользователь может просматривать товары, а также отзовы к ним, во время просмотра товаров пользователь может получать рекомендации других товаром, обычно из той же группы товаров, которую он просмотривает, пользователи, которые авторизиовались в системе могут добавлять товары в коризину, а так же осуществлять заказы.

Diagram, schematic

Description automatically generated

## Разработка технического задания

Объектом разработки является приложение, обеспечивающее выполнение следующих функций:

Для не авторизированных пользователей:

* Регистрации в системе.
* Получение всех продуктов для данной группы.
* Получение конкретного продукта.
* Получение отзывов для продукта.
* Получение рекомендаций к продукту.

Для авторизированных пользователей:

* Все возможности не авторизированных пользователей.
* Добавление отзывов для продукта.
* Редактирование своих отзывов.
* Удаление своих отзывов.
* Получение рекомендация основанных на предпочтениях пользователя.
* Возможность добавлять продукты в корзину.
* Возможность редактировать количество продуктов в корзине.
* Возможность осуществлять заказ продуктов.
* Возможность отменять заказ.
* Возможность следить за статусом заказа.

Для администраторов системы:

* Все возможности не авторизированных пользователей.
* Добавление продуктов в систему.
* Удаление продуктов из системы.
* Редактирование продуктов.
* Удаление отзывов других пользователей.
* Редактирование отзывов других пользователей.
* Возможность отменять заказы других пользователей.

Общие требование для системы

* Система должна быть масштабируемая.
* Все данные должны храниться в Базе данных.
* Система должна быть проста в развертывание на сервере.
* API системы должен быть задокументирован.
* Описание внутренней структуры базы данных.
* Описание процесса установки системы.

Требования к сопровождению системы Этапы внедрения системы

* Установка базы данных на сервере.
* Тестирование работы системы.
* Техническая поддержка системы.

Хранение информации о пользователях, продуктках,заказах и корзине должно осуществляться в БД. Также необходимо реализовать API для взаимодействия микросервисов между собой.

# Глава 2. Специальная часть

## Анализ предметной области

При разработке программных продуктов необходимо провести анализ существующих вариантов для решения задачи. В данном разделе рассматриваются общие теоретические вопросы, освещающие технологические аспекты, необходимые для реализации программного продукта корпоративного класса. В частности, производится выбор среды разработки, СУБД, архитектуры приложения, языка программирования, библиотек и архитектуры API. В каждом подразделе обосновывается выбор того или иного технического решения задачи.

## Выбор среды разработки

В качестве основной платформы для написания дипломной работы была выбрана Java Virtual Machine.

Виртуальная машина Java это абстрактная вычислительная машина. Как и реальная вычислительная машина, она имеет набор инструкций и манипулирует разными участками памяти во время своей работы.

Первая реализации виртуальной машины Java был представлена компанией Sun Microsystems, Inc в 1994 году

Непосредственно виртуальная машина Java «не знает» ничего о языке программирования, на котором написан исходный код приложения, ей лишь известен заданный формат двоичных файлов – файлов, имеющих расширение .class. Эти файлы содержат инструкции виртуальной машины (байткод), таблицы символов и другую вспомогательную информацию. Из соображений безопасности виртуальная машина Java предъявляет строгие синтаксические и структурные требования на код, расположенный в .class файле. Тем не менее, любой язык, функциональность которого может быть выражена средствами корректного .class файла, может быть интерпретирован для виртуальной машины Java. Привлечённые общедоступностью и платформенной независимостью, разработчики компиляторов других языков могут использовать виртуальную машину как удобную платформу для своих реализаций.

JVM имеет две основные функции:

1. Позволяет запускать Java-приложения на любых устройствах или операционных системах (принцип — «Написал один раз, запускай везде»)
2. Управляет и оптимизирует память, используемую приложением

Таким образом JVM – это одна из лучший платформ для разработки приложений из-за совей кроссплатформенности нам не нужно заботиться о том на какой OS запущено наше приложение и об организации работы с памятью GC очистит память за нас, а из того, что JVM – языка независимая платформа, то допускается смешивания языков программирования, например в Java проекте может быть код на Kotlin, Groovy или Scala. Поскольку JVM довольно старая платформа, то под нее написано довольно много библиотек, что является еще одним плюсом в качестве выбора этой платформы.

## Анализ и выбор СУБД

Существуют два вида СУБД: реляционные (SQL) и нереляционные (NoSQL).

Что бы выбрать какой вид базы данных нам использовать нужно знать характерные отличия их друг от друга.

Можно выделить следующий отличительные признаки:

1. Разница в структуре и типе хранимых данных. В реляционных СУБД должна быть однозначно определена структура хранения данных, в то время как в NoSQL БД нет таких строгих ограничений, и они допускают хранение любого типа данных. Для наших микросервисов подходит структура данных с однозначно определенными связями между объектами таблиц.
2. Различия в структуре запросов. В SQL БД данные могут быть получены только посредством языка SQL, так как реляционные СУБД соответствуют стандартам SQL. В нереляционных СУБД нет четкого определения, как должен выглядеть запрос и поэтому для каждой NoSQL БД реализуется свой способ работы с данными. Для нашей системы требуется построение множества запросов с выборками по условию, и использование нереляционной СУБД будет сопряжено с проблемами с созданием запросов, поэтому выбор реляционной СУБД для работы здесь очевиден.
3. Масштабируемость. В большинстве случаев SQL базы данных вертикально масштабируемые, то есть вы можете увеличивать нагрузку на отдельно взятый сервер, наращивая мощность центральных процессоров, объёмы ОЗУ или системы хранения данных. А NoSQL базы данных горизонтально масштабируемы. Хотя NoSQL базы данных лучше масштабируют горизонтально, для нашей систем более важным является строгая структура хранения данных, а при увеличении нагрузки можно прибегнуть кластеризации базы данных.
4. Надежность. SQL базы данных являются более надежными, по сравнению с NoSQL БД. Поэтому реляционные БД более предпочтительны для выбора в качестве решения для микросервиса
5. Поддержка. Реляционные СУБД появились гораздо раньше NoSQL, поэтому существуют уже готовые платные и бесплатные решения и поддержка, и найти ответ на свой вопрос гораздо проще, чем когда речь идет о сложном решении в нереляционной СУБД. И снова выбор в пользу SQL.
6. Хранение и доступ к сложным структурам данных. SQL базы данных выигрывают в производительности по сравнению с нереляционными СУБД, так как изначально предполагалось, что они будут работать именно со сложными структурами данных.

Подводя итоги анализа выбора между реляционных и нереляционных СУБД для нашей системы, можно с уверенностью сказать, что очевидным выборам является реляционная СУБД.

## Выбор реляционной СУБД

Для выбора наиболее подходящей PСУБД необходимо проанализировать имеющиеся на рынке популярные и развивающиеся SQL базы данных. А именно: MS SQL Server, Oracle Database, MySQL, PostgreSQL.

### MS SQL Server

Microsoft SQL Server — система управления реляционными базами данных (РСУБД), разработанная корпорацией Microsoft. Основной используемый язык запросов — Transact-SQL, создан совместно Microsoft и Sybase. Transact-SQL является реализацией стандарта ANSI/ISO по структурированному языку запросов (SQL) с расширениями. Используется для работы с базами данных размером от персональных до крупных баз данных масштаба предприятия; конкурирует с другими СУБД в этом сегменте рынка.

Достоинства РСУБД MS SQL Server:

* простота использования;
* возможность регулировки и отслеживания уровня производительности, чтобы уменьшить загрузку;
* последняя версия продукта работает стабильно;
* возможность интеграции с другими продуктами Microsoft;
* возможность визуализации на мобильных устройствах.

Недостатки РСУБД MS SQL Server:

* очень высокая стоимость продукта для юридических лиц;
* возможны проблемы в работе служб интеграции импорта файлов;
* высокая ресурсоемкость SQL Server (даже при настройке производительности).

При всех упомянутых достоинствах, у данной РСУБД есть очень существенный недостаток, который делает ее недоступной для использования – ее высокая цена для юридических лиц. Бесплатная версия не обладает достаточными возможностями для полной реализации базы данных для микросервиса

### Oracle Database

Oracle Database - это объектно-реляционная система поддерживающая некоторые технологии, реализующие объектно-ориентированный подход, то есть обеспечивающих управление создания и использования баз данных.

Достоинства ОРСУБД Oracle Database:

* надежность;
* современный функционал и новые разработки, доступные в последней версии;

Недостатки ОРСУБД Oracle Database:

* высокая стоимость продукта для юридических лиц;
* для системы необходимы аппаратные ресурсы, обладающие достаточной мощностью и производительностью;

К сожалению, высокая стоимость не позволяет использовать данную ОРСУБД, бесплатная версия позволяет развертывать небольшие по объему базы данных

### PostgreSQL

PostgreSQL – свободная объектно-реляционная система управления базами данных с открытым исходным кодом, разработчиком которой является сообщество PostgreSQL

Достоинства PostgreSQL:

* бесплатная и для личного и для коммерческого использования;
* хорошо масштабируется;
* возможность обрабатывать терабайты данных;
* поддержка формата JSON;
* существует множество готовых, встроенных функций;
* хорошая документация;
* большое сообщество с поддержкой.

Недостатки PostgreSQL:

* неэффективная репликация данных;
* трудности с обновлением;
* Производительность: В простых операциях чтения PostgreSQL может уступать своим соперникам.

Поскольку нам важна скорость операции чтения, например получение продуктов или отзывов, то PostgreSQL проигрывает по этому параметру.

### MySQL

MySQL – свободная реляционная система управления базами данных. С 2010 года права на торговую марку получила компания Oracle. MySQL является бесплатным программным обеспечением, регулярно выходят новые версии, дополняющие функционал. Существуют и платные версии для коммерческих организаций. В бесплатной версии сделан упор на надежность и скорость работы, но не на полноту функционала. MySQL является довольно гибкой СУБД за счёт возможности использовать множество типов таблиц.

Достоинства MySQL:

* бесплатная для личного использования;
* имеет хорошую документацию;
* безопасность;
* возможность использования множества функций в бесплатной версии;
* возможность взаимодействия с другими базами данных, например DB2 и Oracle.
* Реплицируемость: Базу данных MySQL можно распределять между несколькими узлами, таким образом уменьшая нагрузку и улучшая масштабируемость и доступность приложения;
* Шардинг: В то время как шардинг невозможен на большинстве SQL баз данных, MySQL является исключением.

Недостатки MySQL:

* проблемы с надежностью;
* платная для коммерческого использования
* некоторые ограничения функционала;

Поскольку MySQL весьма гибкий и простой в обращении инструмент, который к тому же обладает высокой степенью безопасности, а также поддерживает репликацию и шардинг было принято решение использовать именно эту СУБД.

## Выбор архитектуры приложения

Существует два основных типа архитектуры для проектирования веб-приложения: монолитная и микросервисная. Чтобы понять какую архитектуру выбрать нужно проанализировать особенности каждой архитектуры.

### Монолитная архитектура приложения

Разработка нормального приложения начинается с модульной многоуровневой или шестиугольной архитектуры. Эта архитектура состоит из следующих типов слоев:

1. **Уровень представления.** Это уровень графического интерфейса пользователя, который обрабатывает запросы протокола передачи гипертекста (HTTP) с использованием HTML или XML / JSON.
2. **Уровень** бизнес-логики: бизнес-логика приложения присутствует на этом уровне.
3. **Уровень доступа** к базе данных: все обращения к базе данных, включая SQL и NoSQL приложений, происходят на этом уровне.
4. **Уровень интеграции приложений**: все интеграции программного обеспечения с другими системами происходят на этом уровне.

Несмотря на то, что монолитная архитектура имеет логическую многоуровневую архитектуру, конечные приложения будут упакованы в один монолит и затем развернуты таким образом. Монолитным приложениям не хватает надлежащей модульности, и она имеет только одну кодовую базу. На рисунке 1 показано графическое представление монолитной архитектуры.



Рисунок 1 — графическое представление монолитной архитектуры

Преимущества монолитной архитектуры:

* сравнительно простая реализация, развертывание и управление;
* проще поддерживать согласованность кода и обрабатывать ошибки;
* легко добавить типичный функционал к компонентам;
* достаточно высокая производительность для малых и средних приложений.

Но при кажущихся преимуществах монолитной архитектуре присущи достаточно серьезные недостатки:

* возрастающая сложность и запутанность приложения;
* с увеличением приложения все сложнее поддерживать изолированность сервисов, чтобы масштабировать каждый из них независимо или осуществлять поддержку кода;
* внесение изменений сопряжено с проблемами и занимает много времени;
* сбой одного сервиса приводит к сбою всего приложения.

Последний недостаток, при котором от сбоя в одном сервисе начинает сбоить всё приложение, является неприемлемым, если рассматривать его в контексте работы интернет-магазина.

### Микросервисная архитектура приложения

Микросервисная архитектура — распространенный подход к разработке программного обеспечения, когда приложение разбивается на небольшие автономные компоненты (микросервисы), каждый из которых может независимо развиваться и масштабироваться, и взаимодействовать со своими собственными ресурсами. Отдельные микросервисы могут располагаться на разных серверных узлах. Схематичное сравнение микросервисной и монолитной архитектуры представлено на рисунке 2

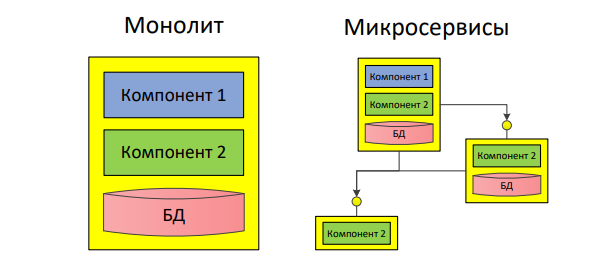


Рисунок 2 — Сравнение монолитной и микросервисной архитектур

Преимущества использования микросервисов:

* возможность обновления приложения по частям, независимое развертывание;
* У микросервисов доступность выше: даже если один из них сбоит, это не приводит к сбою всего приложения;
* возможность независимого масштабирования отдельных микросервисов;
* микросервисная архитектура позволяет размещать разные модули на разных серверах, тем самым обеспечивая их логическое разделение;
* возможность использования нескольких языков программирования за счет расположения модулей на разных серверах;
* разнообразие технологий, которые можно использовать, чтобы подобрать наиболее подходящий функционал;
* децентрализованное управление;
* децентрализованное управление данными;

Как и у всякой технологии и подхода к разработке у микросервисов есть и ряд недостатков:

* сложность разработки;
* требуют тщательного управления, так как используют API и развертываются на разных серверных узлах;
* требуют четкого разграничения модулей;
* увеличение сетевых задержек Вывод: несмотря на сложность разработки и управления, микросервисная архитектура отлично подходит как для реализации функционала интернет-магазина электроники.

## Выбор средств проектирования

//Кратко написать про Enterprise Architect либо вообще убрать этот раздел

## Выбор языка программирования

При разработки программного продукта одним из важных этапов является выбор языка программирования. От этого зависит скорость разработки и чистота кода, а также длительность поддержки продукта.

В настоящее время наиболее популярными языками программирования для разработки веб-приложений являются: Go, Java, JavaScript, Python,Kotlin. Такие языки как С++, Php и Ruby в последнее время теряют популярность в веб-разработке.

### Язык программирования Go

Go (также известный как Golang) — компилируемый многопоточный язык программирования, разработанный компанией Google. Язык разрабатывался для проектирования высокопроизводительных программ и для решения проблем большинства языков. Язык проектировался с целью нивелировать такие проблемы при разработке как медленная сборка программы, сложность при разработке инструментария, проблемы межъязыкового взаимодействия и другие. Но данный язык не является объектно-ориентированным в полном смысле этого понятия, так как в язык сознательно не была включена концепция наследования. Язык обладает меньшей переносимостью на разные платформы, так как компилируется в машинный код.

### Язык программирования Python

Python – высокоуровневый язык программирования, который ориентирован на повышение производительности разработчика и читаемости кода. Этот язык очень гибкий и обладает широким функционалом, а код высокой степенью читаемости. Но у данного языка есть существенный недостаток – он обладает низким быстродействием и поэтому часто используется как язык второго плана.

### Язык программирования JavaScript

JavaScript – мультипарадигменный язык программирования. Язык не принадлежит какой-либо организации или компании. Наиболее часто используется в качестве языка сценариев веб-страниц. JavaScript поддерживает императивный, декларативный и объектно-ориентированный стили программирования. JavaScript часто используется в таких программных продуктах, которые в свою очередь используются в веб-разработке. Так как язык является интерпретируемым, то производительность и безопасность зависит от конкретной реализации интерпретатора («движка»).

### Язык программирования Java

Java – сильно типизированный объектно-ориентированный язык программирования. В настоящее время разработкой занимается компания Oracle. Основной особенностью данного языка является то, что программы на Java транслируются в байт-код, который выполняется на виртуальной машине Java. Виртуальная машина Java в свою очередь является интерпретатором и передает инструкции оборудованию, что обеспечивает легкую переносимость программы на разные платформы. В настоящее время Java является одним из самых популярных языков программирования.

### Язык программирования Kotlin

**Kotlin**— это язык программирования с открытым исходным кодом, который может работать на виртуальной машине Java (JVM). Это язык, который сочетает в себе объектно-ориентированное программирование (ООП) и функциональное программирование на неограниченной, самодостаточной и самобытной платформе.

Программы Kotlin не требуют точки с запятой в своей программе. Это делает код простым и более читабельным.

* Программы Kotlin не требуют точки с запятой в своей программе. Это делает код простым и более читабельным.
* Этот язык позволяет осуществлять обмен и использование информации из Java различными способами. Более того, код Java и Kotlin могут сосуществовать в одном проекте.
* Система типов Kotlin нацелена на исключение NullPointerException из кода.
* Написание нового кода на Kotlin займет у вас меньше времени. Еще проще развернуть код kotlin и поддерживать его в масштабе.

Выбор языка программирования

В качестве языка программирования был выбран язык Kotlin. В отличие от программ написанных на Python программы на Kotlin обладают большей производительностью, по сравнению с языком Go. Хотя Kotlin достаточно молодой язык, он имеет большое сообщество, хорошую документацию. JavaScript проигрывает Koltin в безопасности. Java проигрывает Kotlin в гибкость и простоте синтаксиса. Для упрощения и ускорения процесса разработки был выбран фреймворк Spring. Данный фрейворк является универсальным и используется для создания бизнес-приложений.

## Выбор библиотек для разработки

В разработки веб приложений бесспорным лидером является Spring Framework.

Spring-фреймворк обеспечивает комплексную программную и конфигурационную модель для современных Java-enterprise приложений. Ключевой элемент Spring это инфраструктурная поддержка приложений таким образом, чтобы команда разработчиков могла сфокусироваться на бизнес-логике приложения, которое они пишут, а не на технических деталях платформы, на которую они устанавливают приложение.

Но спринг — это не один какой-то конкретный фреймворк. Это скорее общее названия для целого ряда небольших фреймворков, каждый из которых выполняет какую-то свою работу.

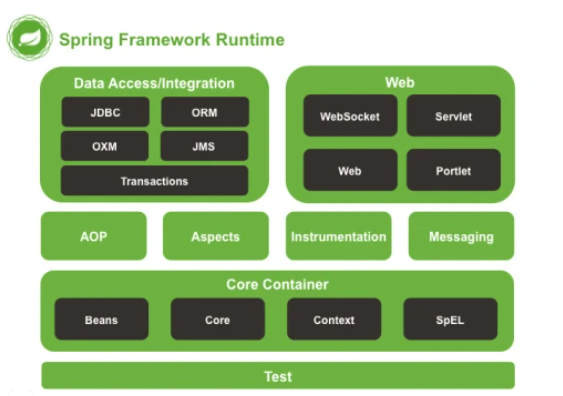


Рисунок 3 — Инфраструктура Spring Framework

Как видно, у спринга модульная структура. Это позволяет подключать только те модули, что нам нужны для нашего приложения и не подключать те, которыми мы заведомо не будем пользоваться.

### Модули Spring для разработки приложения

#### Spring MVC

**Spring MVC** обеспечивает архитектуру паттерна Model — View — Controller (Модель — Отображение (далее — Вид) — Контроллер) при помощи слабо связанных готовых компонентов. Паттерн MVC разделяет аспекты приложения (логику ввода, бизнес-логику и логику UI), обеспечивая при этом свободную связь между ними.

* **Model** (Модель) инкапсулирует (объединяет) данные приложения, в целом они будут состоять из POJO.
* **View** (Отображение, Вид) отвечает за отображение данных Модели, — как правило, генерируя HTML, которые мы видим в своём браузере.
* **Controller** (Контроллер) обрабатывает запрос пользователя, создаёт соответствующую Модель и передаёт её для отображения в Вид.

#### Spring JPA

Поскольку наши сервисы осуществляют запросы к базе дынных, мы используем Spring JPA.

Spring JPA – это модуль отвечает за работу с базой данных. JPA это стандартизированный программный интерфейс (API) позволяющий задать информацию необходимую для осуществления преобразования объектов к модели реляционных БД посредством аннотаций, а также там описаны интерфейсы объектов предназначенных для выполнения Create, Read, Update, Delete (CRUD) операций с БД. В этот модуль так же входит и реализация для JPA - Hibernate.

#### Spring Security

Spring Security – это фреймворк, который сфокусирован на обеспечение как аутентификации, так и авторизации в Java-приложениях. Как и все Spring проекты, настоящая сила Spring Security в том, что он может быть легко дополнен нужным функционалом.

#### Spring Cloud Eureka

Сервер Eureka — это приложение, которое содержит информацию обо всех клиентских сервисных приложениях. Каждый микросервис регистрируется на сервере Eureka, и Eureka знает все клиентские приложения, работающие на каждом порту и IP-адресе.

## Описание архитектуры REST API

REST – это архитектурный стиль взаимодействия компонентов распределенного веб-приложения. Принципы REST сформулировал Рой Филдинг в своей диссертации в 2000 году, но не существует официально принятого стандарта или спецификации REST (в отличие, например, от протокола SOAP). API-интерфейс может считаться RESTful только в том случае, если соблюдены все требования. При создании приложения «Интернет-магазин» были учтены данные требования, и поэтому веб-приложение является RESTful. Существует шесть обязательных ограничений для проектирования REST-приложений:

### Модель клиент-сервер

Система должна быть поделена на клиентов и серверов, то есть в основе данного ограничения лежит принцип разграничения потребностей. Это позволяет упростить серверную часть приложения, что в будущем упрощает ее масштабируемость. При этом у клиентского интерфейса повышается переносимость кода на другие платформы. Например, микросервис «Product service» выступает в качестве сервера, но клиенты общаются с ним через сервер посредник (прокси). Клиентами могут являться любые приложения способные осуществить http запрос.

### Отсутствие состояния

Взаимодействие между клиентом и сервером строится по следующему условию: в моменты времени между запросами никакая информация о клиенте не должна храниться на сервере, а каждый запрос от клиента должен быть составлен таким образом, чтобы на сервер поступила вся необходимая информация о транзакции для ее завершения. При этом информация о состоянии сессии сохраняется у клиента. В то время как происходит обработка запросов от клиента, считается, что он находится в переходном состоянии. Данное ограничение позволяет системе лучше масштабироваться, так как отсутствие необходимости хранить информацию о состояниях высвобождает ресурсы сервера, которые могут быть направлены на обработку одновременно большего количества клиентов. Микросервис «Product service» не сохраняет сессии клиентов на своей стороне и не знает в каком состоянии находятся клиенты.

### Кэширование

Клиенты могут выполнять кэширование ответов сервера. В этом случае ответы сервера должны быть обозначены как кэшируемые, если поступила новая информация, или некэшируемые, если поступившие данные уже передавались ранее. Грамотно построенное кэширование способно уменьшить взаимодействие между клиентом и сервером, что увеличивает производительность и масштабируемость распределенной веб-системы в целом. В микросервисе «Product service» невозможно сделать кэширование запросов на данном этапе разработки, в дальнейшем планируется рассмотрение этого вопроса.

### Единообразие интерфейса.

Для эффективного взаимодействия компонентов и кэширования в сети необходим унифицированный интерфейс, позволяющий независимо развиваться отдельным сервисам. В свою очередь для унифицированных интерфейсов существует четыре условия:

* 1. Идентификация ресурсов. Любая информация может быть ресурсом – для этого она должна иметь имя. Каждый ресурс должен быть идентифицирован с помощью идентификатора, не меняющегося при изменении состояния ресурса. Идентификатором в REST является URI. В микросервисе «Poroduct service» для каждой сущности существует свой идентификатор. Пример для сущности «Product» идентификатор выглядит как «/api/product/{productId}».
  2. Манипуляция ресурсами через представление. Ресурсы могут быть представлены различными способами, например: JSON, HTML, XML описание и так далее. Представление является описанием текущего состояния ресурса и используется для выполнения операций над ресурсом. Взаимодействия клиентов с ресурсами происходит посредством представлений. В «Product service» в качестве способа представления используется описание в формате JSON.
  3. «Самоописываемые» сообщения. Каждый запрос (ответ) хранит в себе всю необходимую информацию, чтобы понять, как его нужно обработать. Для обработки одного запроса не должно быть дополнительных сообщений. Каждый из микросервисов в моем приложении полностью удовлетворяет данное условие.
  4. Гипермедиа, как средство изменения состояния приложения. Для навигации по API должен быть использован гипертекст, что позволяет клиентам обнаруживать ресурсы посредством гиперссылок. В микросервисе «Product service» используются гиперссылки.

### Многослойная система

В архитектуре REST возможно разделение системы на иерархию слоев, но с условием: отдельный компонент системы может видеть только компоненты следующего уровня. Использование промежуточных прокси-серверов позволяет увеличить масштабируемость, сбалансировав нагрузку и распределив кэширование. Также это позволяет использовать политику безопасности для обеспечения конфиденциальности данных.

В нашем приложении клиент не общается напрямую с ни с одним сервисом, взаимодействие идет через zuul proxy сервер, который получает данные с eurka сервера и перенаправляет клиента на нужный сервер. Для распределения нагрузки используется Ribbon.

# Глава 3. Проектная часть

## Проектирование

Перед тем, как создавать приложение, необходимо продумать его архитектуру, а именно составить диаграмму компонентов, обозначить связь компонентов.

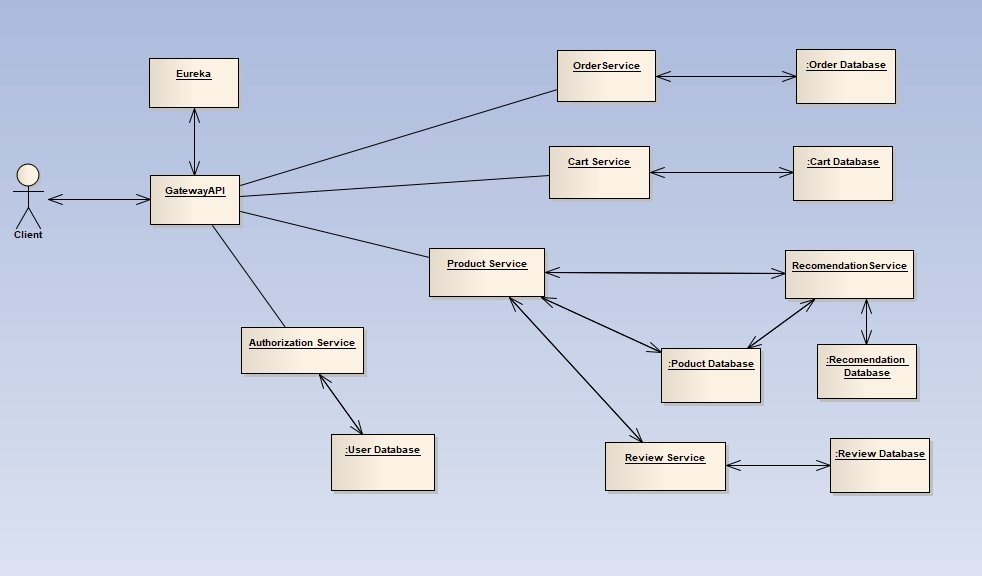


Рисунок 4 — Структурная схема интернет-магазина

Схема, представленная на рисунке 4 описывает связь микросервисов между собой.

В качестве «Client» - может выступать как браузер, так и десктоп или смартфон, то есть любое приложение способное отправлять http запросы.

Общая картина выглядит следующим образом

Клиент делает запрос на Gateway, который в свою очередь перенаправляет запрос на один из сервисов.

Давайте более подробно рассмотрим каждый из сервисов

## Реализация

Для обеспечения совместной работы сервисов будем использовать набор основных паттернов и практик Микросервисной архитектуры. Многие из них реализованы в Spring Cloud (в частности, посредством интеграции с продуктами Netflix OSS)

### Gateway.

Гипотетически, клиент могло бы запрашивать каждый из сервисов самостоятельно. Но такой подход сразу натыкается на массу ограничений — необходимость знать адрес каждого эндпоинта, делать запрос за каждым куском информации отдельно и самостоятельно мерджить результат. Кроме того, не все приложения не бэкенде могут поддерживать дружественные вебу протоколы.

Для решения такого рода проблем применяют API Gateway — единую точку входа. Ее используют для приема внешних запросов и маршрутизации в нужные сервисы внутренней инфраструктуры, отдачи статического контента, аутентификации, стресс тестирования, канареечного развертывания, миграции сервисов, динамического управления трафиком.

В нашем случае имплементацией паттерна API Gateway является Zuul.Zuul используется для роутинга запросов и балансирования запросов.

Вот так выглядит префиксная маршрутизация в данном проекте:

routes:  
 cart-service:  
 path: /api/cart/\*\*  
 serviceId: ${cart-service.id}  
 stripPrefix: false  
 product-service:  
 path: /api/products/\*\*  
 serviceId: ${product-service.id}  
 stripPrefix: false  
 order-service:  
 path: /api/orders/\*\*  
 serviceId: ${order-service.id}  
 stripPrefix: false  
 authorization-service:  
 path: /api/authorization/\*\*  
 serviceId: ${authorization-service.id}  
 stripPrefix: false

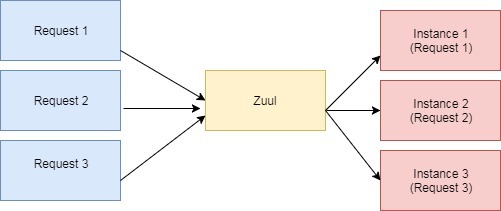


Рисунок 5 – Схема работы Zuul сервера

Обычно для распределенных систем характерна работа ее компонентов на разных компьютерах. С увеличением количества пользователей приложение обычно создает разные реплики, каждая реплика работает на отдельном компьютере. В это время появляется «Load Balancer» (Балансировка нагрузки), которая помогает распределять входящий траффик равно между серверами.

Есть 2 вида LoadBalancer:

**Server side Load Balancer**: Расположен на стороне сервера. Когда запросы поступают от Client они придут к балансировке нагрузки, и она определит один сервер для этого запроса. Самый простой алгоритм, используемый балансировкой нагрузки, это случайное распределение.

**Client-Side Load Balancer**: Балансировка нагрузки находится на стороне клиента, она сама решает к какому серверу отправить запрос, основываясь на некоторых критериях. Балансировка на клиентской стороне обычно отправляет запросы к серверам одной зоны (Zone), или имеет быстрый ответ.

В нашем случае это в качестве балансировщика нагрузки выступает Ribbon. **Ribbon** — это client-side балансировщик. По сравнению с традиционным, здесь запросы проходят напрямую по нужному адресу, что исключает лишний узел при вызове. Из коробки он интегрирован с механизмом Service Discovery, который предоставляет динамический список доступных инстансов для балансировки между ними по правилу «Best Rule». Алгоритм этого правила такой: мы делаем запросы на каждый инстанс необходимого нам микросервиса с целью узнать какой из инстансов имеет наименьшую очередь из запросов, как только нужный нам инстанс найдем мы перенаправляем запрос на него.

### Eureka.

Еще один широко известный паттерн для распределенных систем Service discovery. Service discovery позволяет автоматически определять сетевые адреса для доступных инстансов приложений, которые могут динамически изменяться по причинам масштабирования, падений и обновлений.  
  
Ключевым звеном здесь является Registry service. В этом проекте я использую Netflix Eureka.

Для создания Eureka сервиса был создан Spring-Boot проект, в который была добавлена зависимость на «spring-cloud-starter-eureka-server», сам сервер подключается через добавление аннотации @EnableEurekaServer

Так же была выполнена настройки сервера через .yml файл

Настройка на стороне сервера

server:  
 port: 8761  
 address: localhost  
---  
spring:  
 profiles: docker  
server:  
 port: 8761  
 address: eureka-server  
---  
eureka:  
 client:  
 registerWithEureka: false  
 fetchRegistry: false  
 server-url:  
 defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka  
 server:  
 waitTimeInMsWhenSyncEmpty: 0  
 maxThreadsForPeerReplication: 0  
 healthcheck:  
 enabled: true  
 instance:  
 hostname: ${server.address}  
 lease-expiration-duration-in-seconds: 15  
 lease-renewal-interval-in-seconds: 5

Помимо настройки сервера нужно так же настроить и клиентов, то есть микросервисы которые будут регистрироваться в Eureka. Для этого в каждый из микросервисов была подключена зависимость «spring-cloud-starter-eureka-server» а также аннотация @EnableDiscoveryClient

Настройка на стороне клиента

eureka:  
 client:  
 registerWithEureka: true  
 fetchRegistry: true  
 service-url:  
 defaultZone: http://${eureka.address}:${eureka.port}/eureka  
 server:  
 waitTimeInMsWhenSyncEmpty: 0  
 healthcheck:  
 enabled: true  
 instance:  
 prefer-ip-address: true  
 lease-expiration-duration-in-seconds: 15  
 lease-renewal-interval-in-seconds: 5  
 ip-address: ${server.address}  
 instance-id: ${server.address}:${server.port}

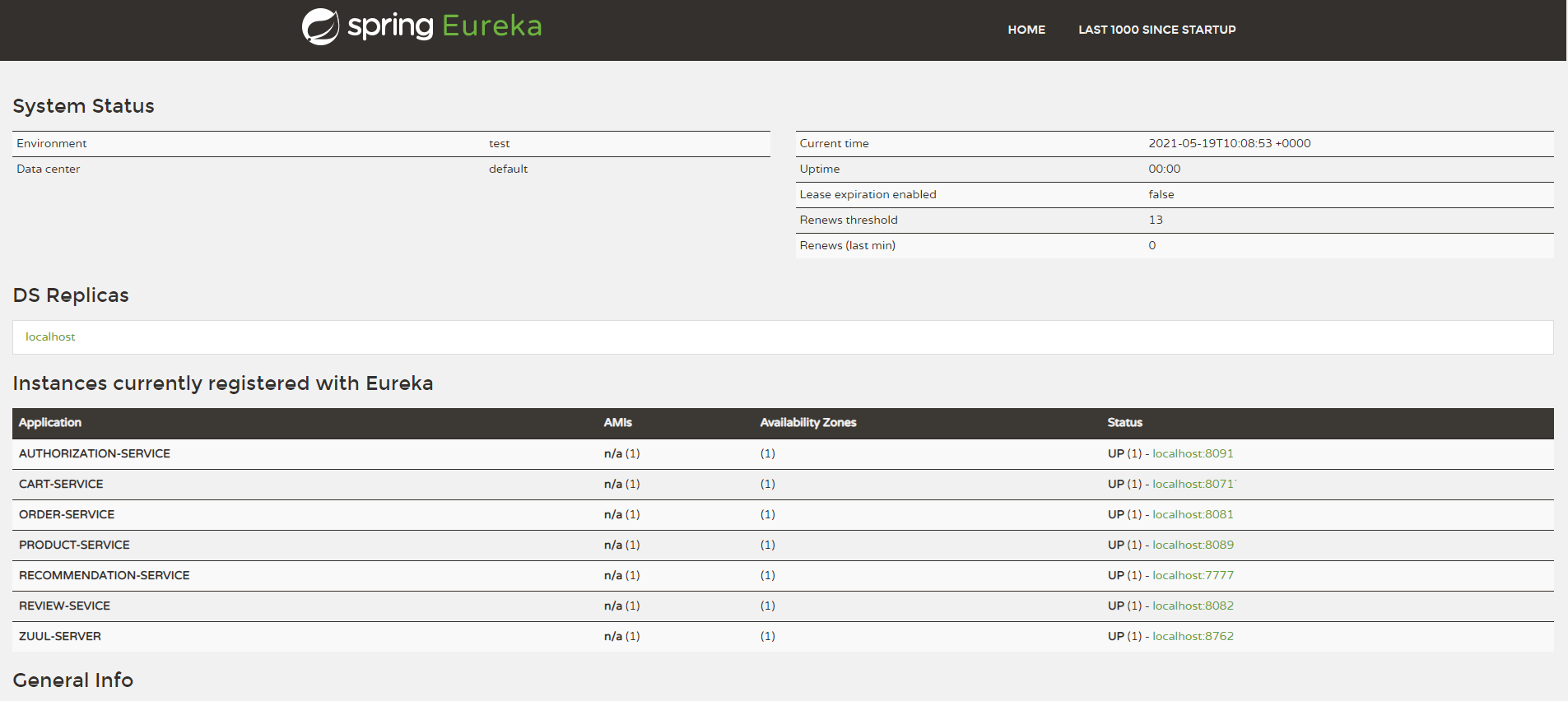


Рисунок 6 – Пример работы Eureka Server

На рисунке 6 показана работа еureka server, можно увидеть, что все инстансы успешно подняты и готовы к работе.

### Authorization service

Для того, чтобы взаимодействовать с приложением, оно должно понимать — кем является текущий пользователь.

Для этого существует два процесса:

* Аутентификация — процесс подтверждения, что этот человек именно тот, за кого себя выдает.
* Авторизация — процесс принятия решения о том, что именно этой аутентифицированной персоне разрешается делать.

Для процессов аутентификации и авторизации был разработан микросервис «Authorization service».

Аутентификация производится с использованием Spring Security и JWT-токенов

JSON Web Token (JWT) — это JSON объект, который определен в открытом стандарте RFC 7519. Он считается одним из безопасных способов передачи информации между двумя участниками. Для его создания необходимо определить заголовок (header) с общей информацией по токену, полезные данные (payload), такие как id пользователя, его роль и т.д. и подписи (signature).

Использование токена позволяет серверу не заботиться о сохранении состояния между запросами (HTTP-сессии), уменьшить количество запросов к БД — необходимые для восстановления данные могут сохраняться в токене.

Рассмотрим процесс получения токена пользователем, а таже саму структуру токена.

Клиент совершает http POST запрос на сервер авторизации, в параметры запроса он передает username и password. Сервер получает запрос клиента и выполняет поиск пользователя по username в базе данных, если такой пользователь найдет, то мы кодируем пароль, который он прислал с помощью, шифруется с помощью BCryptPasswordEncoder и сравнивается с паролем, полученным из базы данных.

Использование BCryptPasswordEncoder позволяет повыситьё степень защищённости приложения так, например, при утечке данных у злоумышленников не будет полноценного пароля пользователя, а всего лишь его хеш. Если хеши паролей совпали, то происходит генерация токена.

В токен входит имя пользователя, список его разрешений, а также время жизни токена

Пример токена

{

"sub": "user1",

"authorities": [

{

"authority": "review:edit"

},

{

"authority": "review:write"

},

{

"authority": "ROLE\_USER"

},

{

"authority": "review:read"

},

{

"authority": "product:read"

}

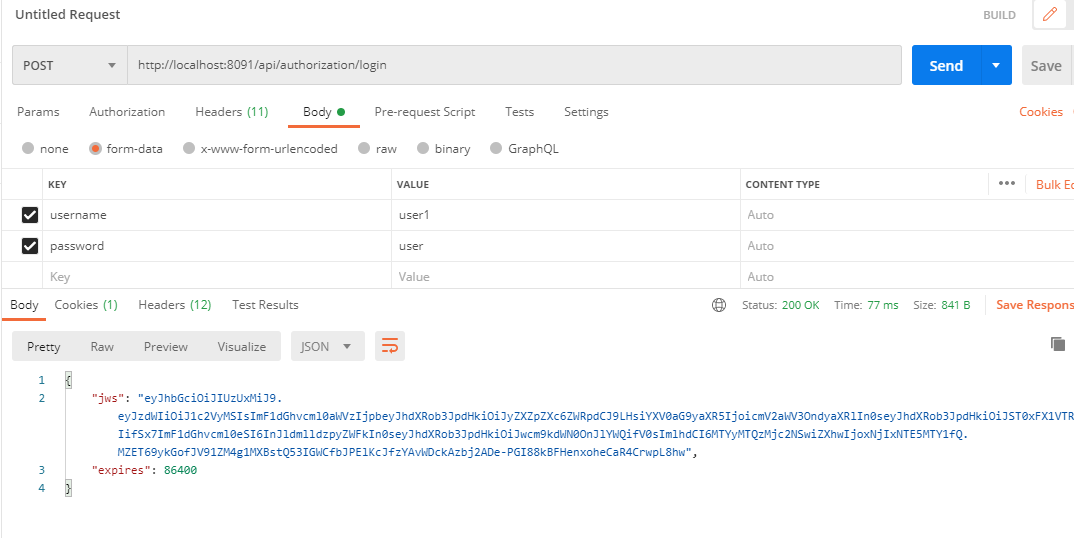
],

"iat": 1621428835,

"exp": 1621515235

}

Пример запроса на вход в систему и ответ от сервера



На рисунке 6 представлена диаграмма последовательности получения токена.

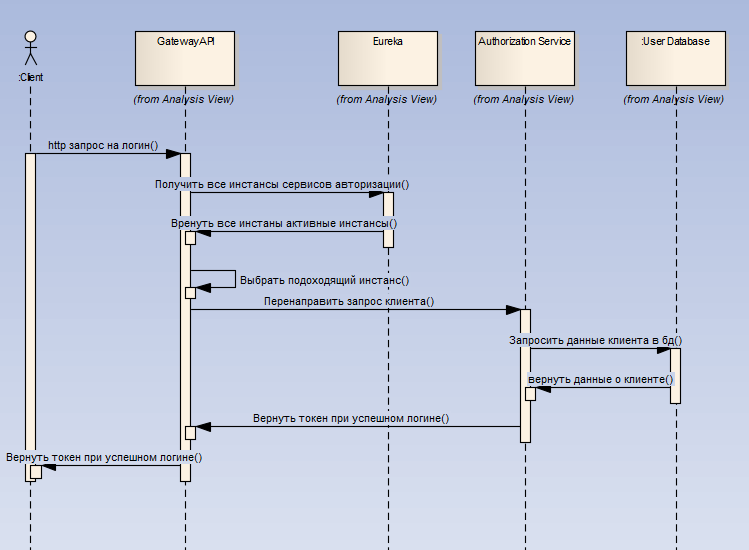


Рисунок 6 – Диаграмма последовательности получения JWT

#### Структура БД

При разработке «Authorization service» была использована СУБД MySQL. Далее представлено описание структуры разработанной БД. Таблица 1 содержит описание структуры таблицы «Пользователь (user)».

Таблица 1 — Структура таблицы «Пользователь (user)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| username | Уникальное имя пользователя, которое является идентификатором | VarChar(255) |
| email | Email- адрес пользователя | VarChar(255) |
| is\_account\_non\_expired | Не истек ли срок действия учетной записи | BIT |
| is\_account\_non\_locked | Заблокирован ли аккаунт | BIT |
| is\_credentials\_non\_expired | Не истек ли срок действия учетных данных | BIT |
| is\_enabled | Активный ли аккаунт | BIT |
| name | Имя пользователя | VarChar(255) |
| password | Пароль | VarChar(255) |
| role | Роль пользователя в системе | VarChar(255) |

Text

Description automatically generated

Рисунок 7 – Схема базы данных для сервиса авторизации

#### REST API

Разработка API один из важнейших этапов создания микросервисов, так как от полноты реализации функционала API зависит удобство реализации остальных микросервисов. API, описанное в данной выпускной работе, соответствует архитектурному стилю REST.

Поскольку техническое задание требует документации API было принято решение использовать Swagger.

**Swagger** - это фреймворк для спецификации RESTful API. Он дает возможность не только интерактивно просматривать спецификацию, но и отправлять запросы.

После запуска микросервиса можно перейти по данной ссылке <http://localhost:8091/swagger-ui.html#/> и посмотреть на все доступные эндпоинты в данном микросервисе

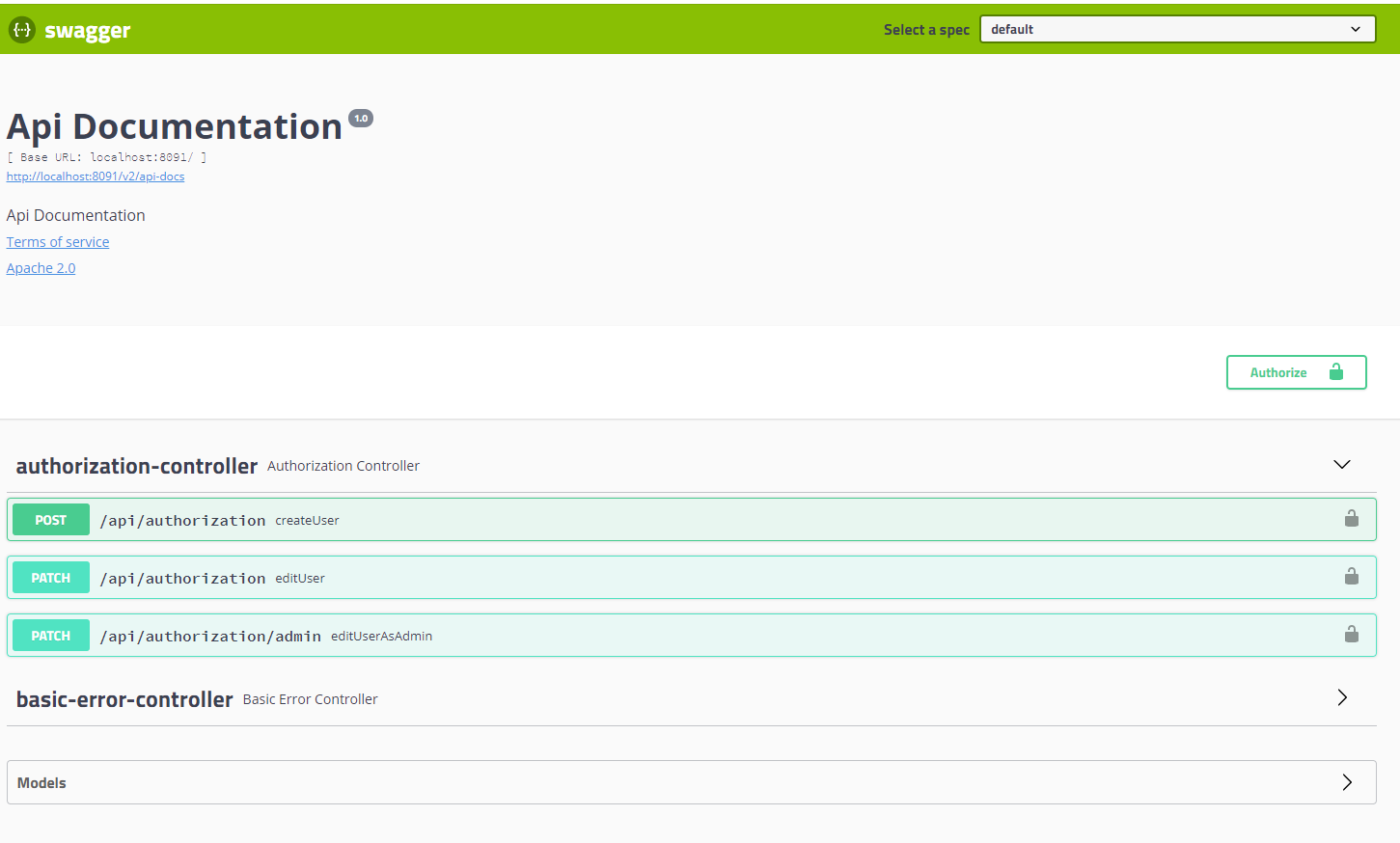


Рисунок 8 – Swagger страница для «Authorization service»

Как видно из рисунка 8 у «Authorization service» три эндпоинта, это:

* POST /api/authorization Запрос на регистрацию пользователя.
* PATCH /api/authorization Запрос на редактирования пользователя, самим пользователем.
* PATCH /api/authorization Запрос на редактирования пользователя Администратором, для данного запроса требуется токен с правами администратора.

Если раскрыть какой-либо из эндпоинтов, то можно увидеть пример JSON-а который должен быть в теле запроса а также, можно выполнить запрос прямо из сваггера, нажав на кнопку «Try it out».

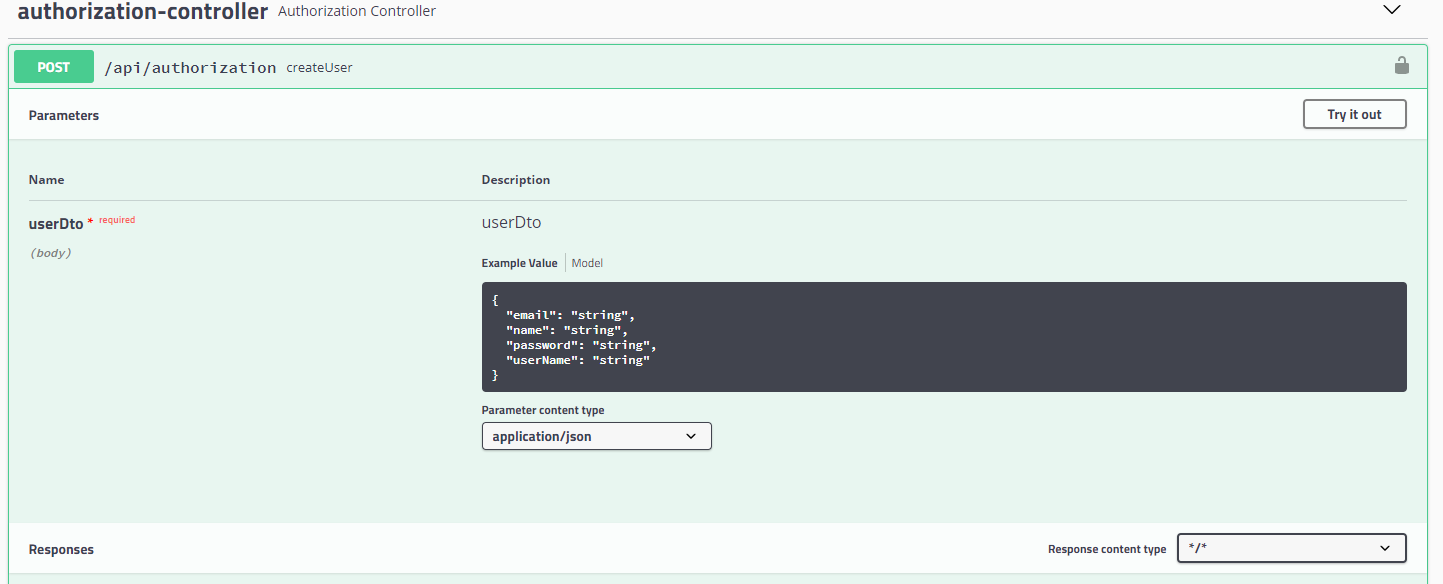


Рисунок 9 – Подробная информация об эндпоинте регистрации

Помимо этих эндпоинтов есть еще один, который отвечает за вход в систему, он находится по такому пути <http://localhost:8091/api/authorization/login> , он не попал в swagger документацию, потому что является стандартным эндпоинтов spring security модуля.

### Product service

Основной микросервис, отвечающий за работу с продуктами и выступающий в качестве маршрутизатора для сервиса с отзывами.

Рассмотрим возможности этого микросервиса, для этого обратимся к его API

#### REST API



Рисунок 10 – API для работы с групп вариантами

* GET /api/group-variants – эндпоинт для получения всех групп -вариантов.
* POST /api/group-variants – эндпоинт для добавления групп-вариантов.
* GET /api/group-variants/{id} – эндпоинт для получения конкретного групп-варианта.
* DELETE /api/group-variants/{id} – эндпоинт для удаления конкретного групп варианта групп-варианта.
* PATCH /api/group-variants/{id} – эндпоинт для редактирования групп-варианта.

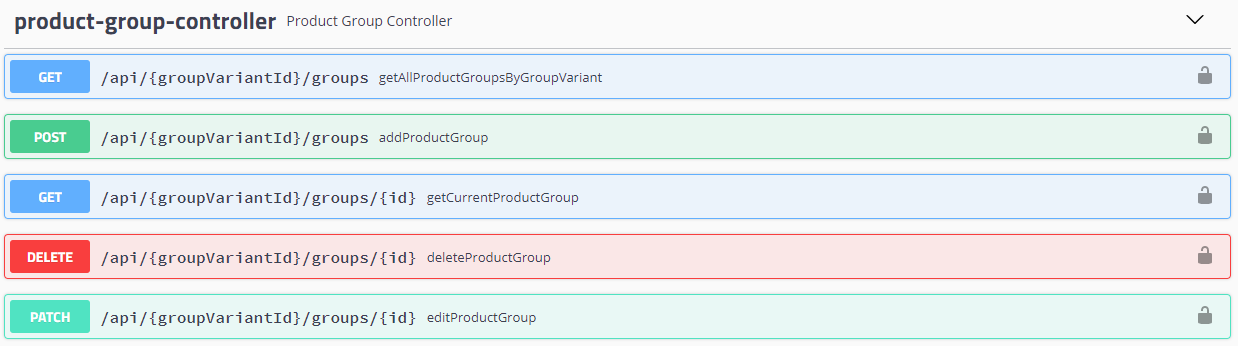


Рисунок 11 – API для работы с продуктовой группой

* GET /api/{groupVariantId}/groups – эндпоинт для получения всех продуктовых групп.
* POST /api/{groupVariantId}/groups – эндпоинт для добавления продуктовых групп.
* GET /api/{groupVariantId}/groups/{id} – эндпоинт для получения конкретной продуктовой группы.
* DELETE /api/{groupVariantId}/groups/{id} – эндпоинт для удаления конкретной продуктовой группы.
* PATCH /api/{groupVariantId}/groups/{id} – эндпоинт для редактирования конкретной продуктовой группы.

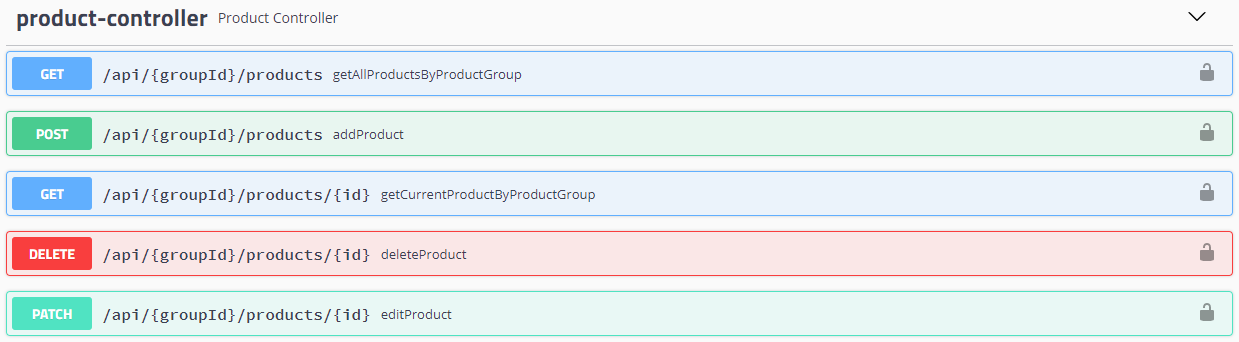


Рисунок 12 – API для работы с продуктовом

* GET /api/{groupId}/products – эндпоинт для получения всех продуктов.
* POST /api/{groupId}/products – эндпоинт для добавления продукта.
* GET /api/{groupId}/products/{id} – эндпоинт для получения конкретного продукта.
* DELETE /api/{groupId}/products/{id} – эндпоинт для удаления конкретного продукта.
* PATCH /api/{groupId}/products/{id} – эндпоинт для редактирования конкретного продукта.

Как было описано выше «Product service» выступает в качестве маршрутизатора для сервиса с отзывами. Это осуществляется за счет Feign Client.

Feign — простой и гибкий http-клиент, который нативно интегрирован с Ribbon.

Для работы Feign клиента мы должны описать API сервиса к котором мы хотим обратиться и указать id сервиса в настройка конфигурации. Далее Feign обращается к Eureka Server и спрашивает, где находится находиться данные сервис и осуществляет запрос к нему, если сервис по каким-либо причина недоступен, то в дело вступает CircuitBreaker.

В нашем случае реализаций CircuitBreaker является Hystrix.

Hystrix - библиотека задержек и отказоустойчивости, которая помогает контролировать взаимодействие между службами, обеспечивая отказоустойчивость и устойчивость к задержкам, благодаря чему повышается устойчивость всей системы в целом.

Конфигурация hystrix настроена следующим образом, после трех неудачных запросов к микросервису, мы начинаем возвращаем значение по умолчанию, через минуту мы повторяем запросы к микросервису, если ситуация стабилизировалось, то начинаем направлять запросы на микросервис, если сервис все еще не доступен, то мы продолжаем возвращать значения по умолчанию.

Конфигурация hystrix

hystrix:  
 command:  
 default:  
 execution:  
 isolation:  
 strategy: SEMAPHORE  
 thread:  
 timeoutInMilliseconds: 10000circuitBreaker:  
 enabled: true  
 requestVolumeThreshold: 3  
 sleepWindowInMilliseconds: 60000  
 fallback:  
 enable: true  
 isolation:  
 semaphore:  
 maxConcurrentRequests: 3

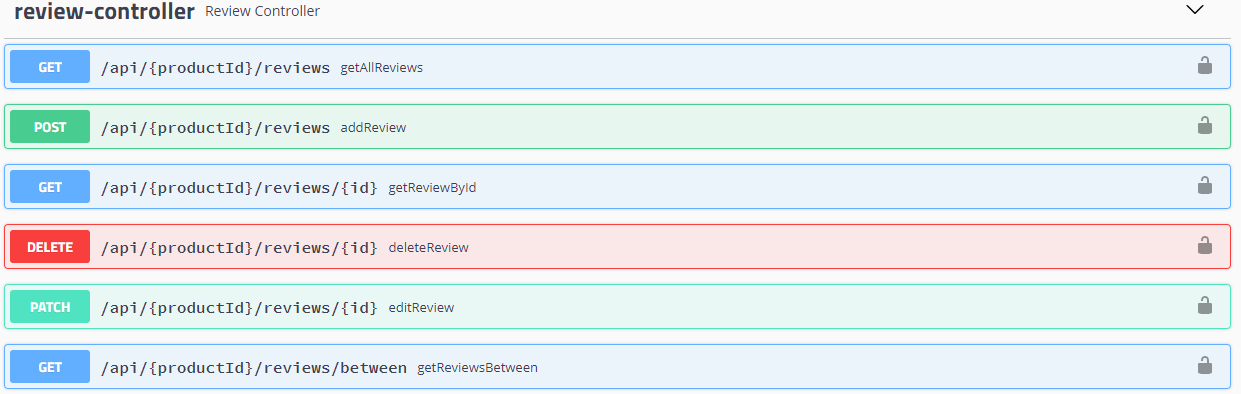


Рисунок 13 – API для работы с сервисом отзывов

* GET /api/{productId}/reviews – эндпоинт для получения всех отзывов.
* POST /api/{productId}/reviews – эндпоинт для добавления отзыва.
* GET /api/{productId}/reviews/{id} – эндпоинт для получения конкретного отзыва.
* DELETE /api/{productId}/reviews/{id} – эндпоинт для удаления конкретного отзыва.
* PATCH /api/{productId}/reviews/{id} – эндпоинт для редактирования конкретного отзыва.
* GET /api/{productId}/reviews/between – эндпоинт для части отзывов.

Более подробно работу «Product service» можно рассмотреть на примере Get запроса на получение продукта по id.

Для этого построим диаграмму последовательности

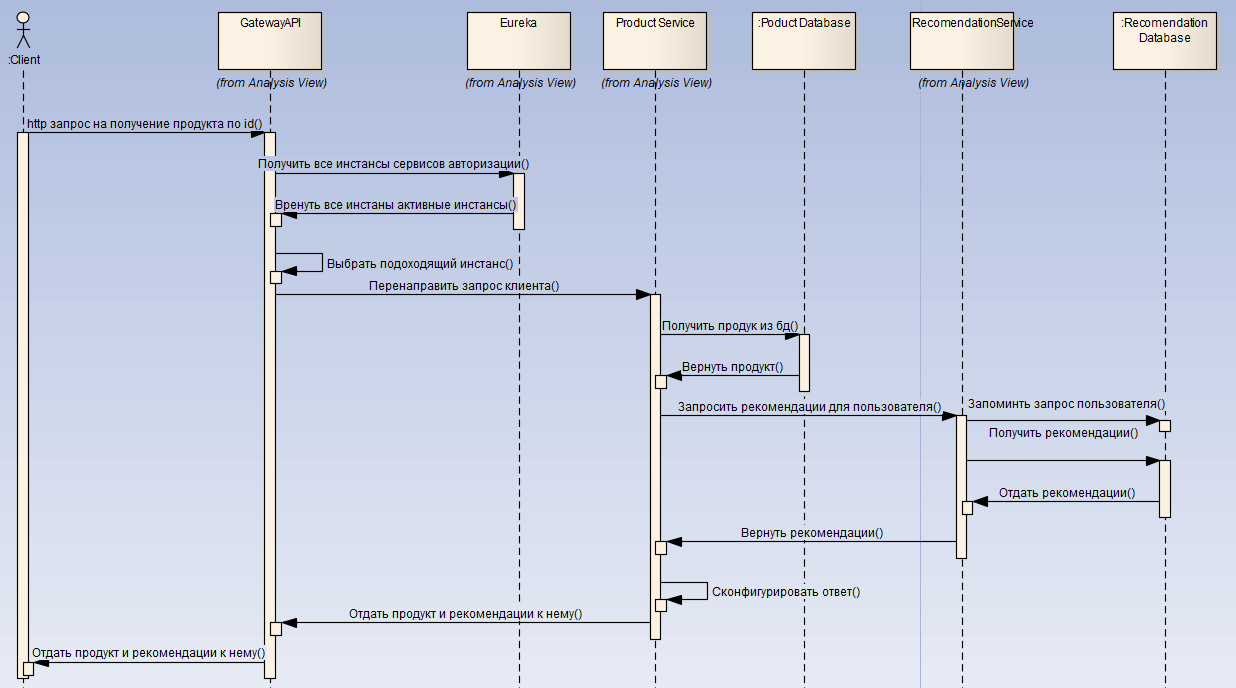


Рисунок 14 — Диаграмма последовательности запроса get запроса

* Клиент делает запрос на получение продукта по id.
* Запрос клиента попадает на Getaway – которым является Zuul сервер.
* Getaway обращается к Eureka сервису, который хранит все информацию о поднятых инстансах всех микросервисов.
* После получения всех нужных инстансов в дело вступает Балансировщик нагрузки
* После выбора инстанса происходит перенаправления запроса клиента на это инстанс.
* После получения запроса «Product service»-ом происходит два действия:
  + Запрашиваем продукт с заданным id из базы данных
  + Делаем асинхронный запрос на «Recommendation Service»
    - На случай если «Recommendation Service» будет не доступен предусмотрен так называемый «CircuitBreaker»
* Формируем ответ и отдаем пользователю

Пример запроса: <http://localhost:8089/api/1/products/1>

Пример ответа на запрос:

{

"Product": {

"id": 1,

"name": "Smart",

"amount": 0,

"price": 200,

"created": "2021-01-02T11:22:28.057+00:00",

"description": "",

"productGroupId": 1,

"isAvailable": false

},

"recommends": [

{

"id": 1,

"name": "Smart",

"created": "2021-01-02T11:22:28.057+00:00",

"description": "",

"productGroup": {

"id": 1,

"name": "Phone",

"groupVariants": {

"id": 1,

"name": "technique"

}

},

"amount": 0,

"price": 200

}, ….

]

}

#### Структура БД

Таблица 1 — Структура таблицы «Групп-вариант (group\_variant)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| group\_variant\_id | Уникальный идентификатор | Уникальный идентификатор |
| name | Имя групп варианта | VarChar(255) |

Таблица 2 — Структура таблицы «Продукт групп (product\_group)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| group\_variant\_id | Уникальный идентификатор | BIGINT |
| name | Имя групп варианта | VarChar(255) |
| group\_variant | Уникальный идентификатор, связанный с таблицей 1 | BIGINT |

Таблица 3 — Структура таблицы «Продукт (product)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| product\_id | Уникальный идентификатор | BIGINT |
| amount | Количество доступных продуктов | INT |
| created | Дата создания продукта | DATETIME |
| description | Описание продукта | VarChar(255) |
| name | Имя продукта | VarChar(255) |
| price | Цена продукта | INT |
| product\_group | Уникальный идентификатор, связанный с таблицей 2 | BIGINT |

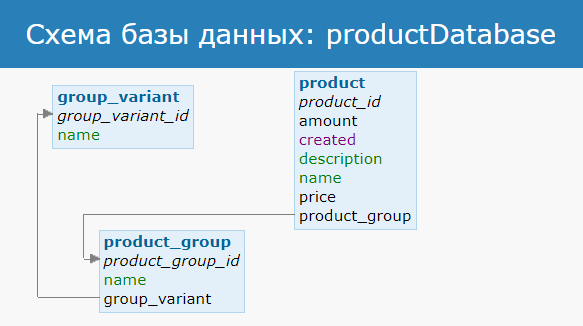


Рисунок 15 — Схема баз данных продукт сревиса

### Recommendation service

Сервис рекомендаций – вспомогательный сервис для «Product service» служит для выдачи рекомендаций для конкретного пользователя.

#### REST API

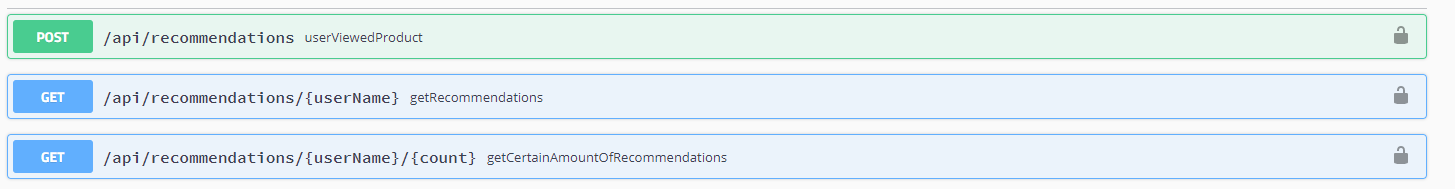


Рисунок 16 – API для взаимодействия с сервисом рекомендаций

* POST /api/recommendations– эндпоинт для добавления информации о пользователе и группе товаров которую он запросил.
* GET /api/recommendations/{userName} – эндпоинт для получения рекомендации для конкретного пользователя по умолчанию возвращает 5 рекомендаций.
* GET /api/recommendations/{userName}/{count} – эндпоинт для получения рекомендации для конкретного пользователя можно указать количество возвращаемых рекомендаций.

На рисунок 14 показан пример получения рекомендаций для пользователя, запросившего продукт по id**.**

#### Структура БД

Таблица 1 — Структура таблицы «Продуктовая-группа (product\_group)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| productGroupId | Номер группы продукта | BIGINT |

Таблица 2 — Структура таблицы «Продуктовая-группа (user)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| userName | Имя пользователя | VarChar(255) |

Таблица 3 — Структура таблицы «Заинтересованность пользователя в группе (user\_interested\_in\_group)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| visitTime | Количество раз, которое посетил пользователь | BIGINT |
| product\_group | Уникальный идентификатор, связанный с таблицей 1 | BIGINT |
| user | Уникальный идентификатор, связанный с таблицей 2 | BIGINT |

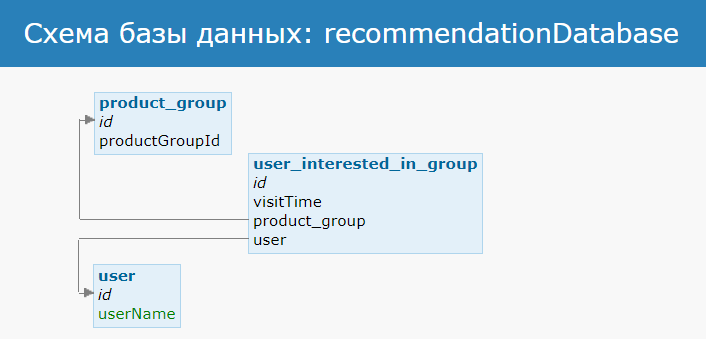


Рисунок 17 – Схема базы данных сервиса рекомендаций

### Review service

Сервис отзывов - служит для получения и оставления отзывов к продуктам. Доступ к этому сервису осуществляется через сервис продуктов. API для работы с эти сервисом представлен на рисунке 18

#### REST API

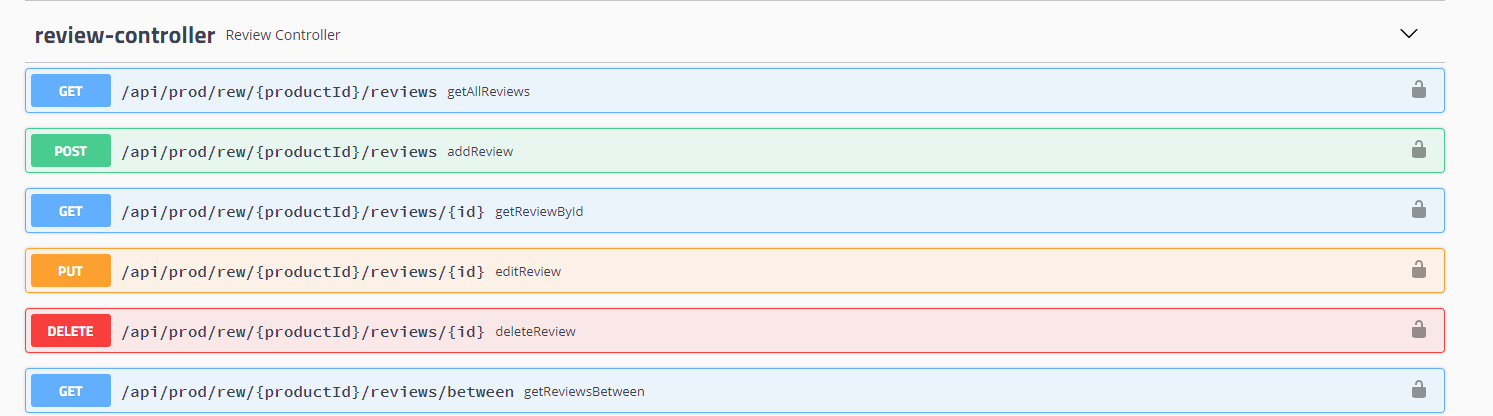


Рисунок 18 – API для работы с сервисом отзывов

* GET /api/prod/rew/{productId}/reviews – получение всех отзывов для одного продукта.
* POST /api/prod/rew/{productId}/reviews – добавление отзыва для продукта.
* GET /api/prod/rew/{productId}/reviews/{id} – получение конкретного отзыва по id.
* PUT /api/prod/rew/{productId}/reviews/{id} – редактирование отзыва.
* DELETE /api/prod/rew/{productId}/reviews/{id} – удаление отзыва по id.
* GET /api/prod/rew/{productId}/reviews/between – получение определенного количества отзывов.

#### Структура БД

Таблица 1 — Структура таблицы «Продукт (product)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| product\_id | Идентификатор продукта | BIGINT |

Таблица 2 — Структура таблицы «Пользователь (user)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| user\_name | Имя пользователя | VarChar(255) |

Таблица 3 — Структура таблицы «Отзыв (review)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| content | Содержание отзыва | VarChar(255) |
| date | Дата создание отзыва | DATETIME |
| product | Уникальный идентификатор, связанный с таблицей 1 | BIGINT |
| user | Уникальный идентификатор, связанный с таблицей 2 | BIGINT |

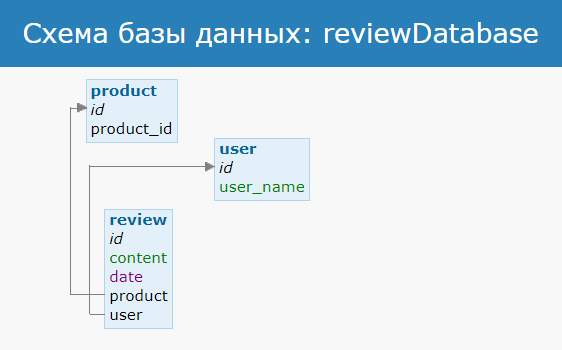


Рисунок 19 – Схема базы данных сервиса отзывов

### Cart service

Сервис корзины – служит для добавления пользователем товаров, которые он собирается купить.

#### REST API

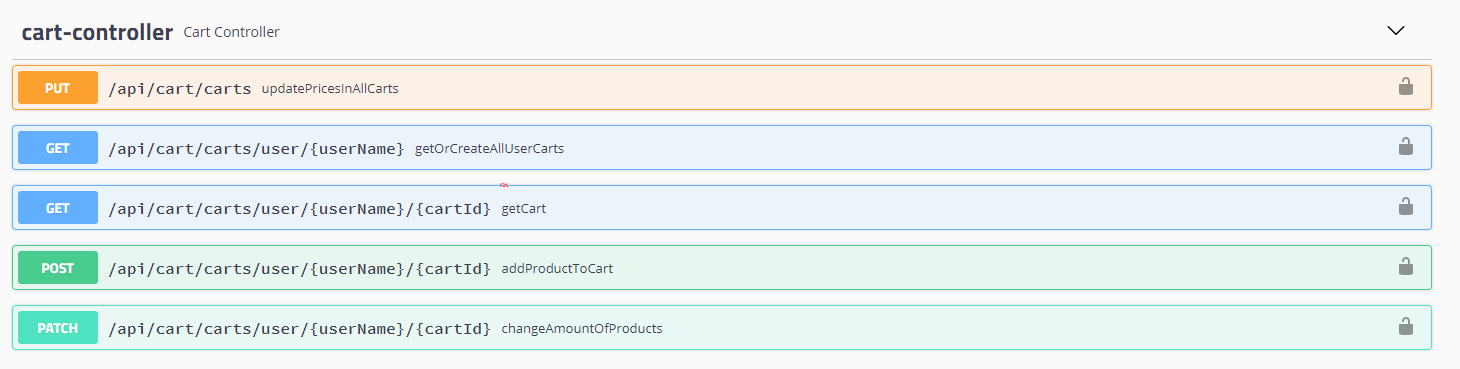


Рисунок 20 – API для взаимодействия с сервисом корзины

* PUT /api/cart/carts – обновление всех цен на сервисе.
* GET /api/cart/carts/user/{userName} – получение всех корзин для всех пользователей.
* GET /api/cart/carts/user/{userName}/{cartId}– получение конкретной корзины.
* POST /api/cart/carts/user/{userName}/{cartId} –добавление продукта в корзину.
* PATCH /api/cart/carts/user/{userName}/{cartId} – редактирование количества продуктов в корзине

#### Структура БД

Таблица 1 — Структура таблицы «Корзина (cart)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| user\_name | Имя пользователя | VarChar(255) |

Таблица 2 — Структура таблицы «Элемент корзины (cart\_item)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| amount | Количество предметов | INT |
| price | Цена | INT |
| product\_group\_id | Идентификатор группы продукта | BIGINT |
| product\_id | Идентификатор продукта | BIGINT |

Таблица 3— Структура таблицы «Элемент в корзине (item\_to\_cart)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| cart\_id | Уникальный идентификатор корзины, связанный с таблицей 1 | BIGINT |
| item\_id | Уникальный идентификатор предмета, связанный с таблицей 2 | BIGINT |

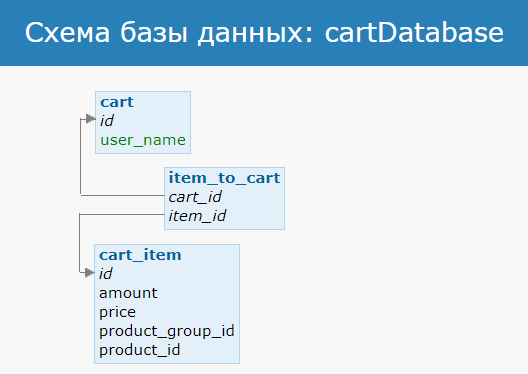


Рисунок 21 – Схема базы данных сервиса корзины

### Order service

Сервис заказов – служит для формирования пользовательского заказа.

#### REST API

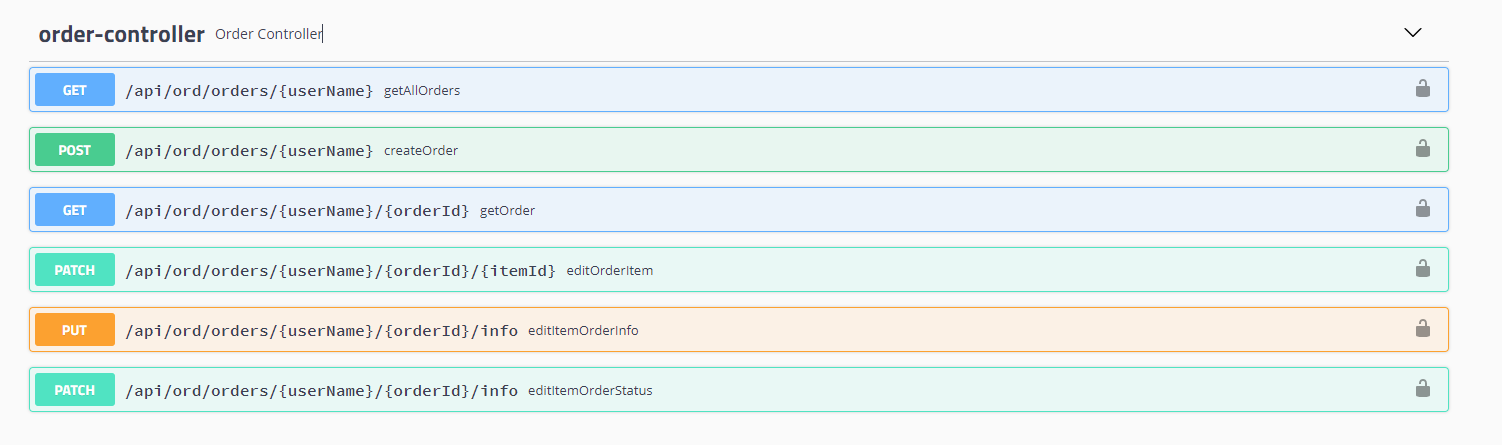


Рисунок 22 – API для взаимодействия с сервисом

* GET /api/ord/orders/{userName} – обновление всех заказов пользователя.
* POST /api/ord/orders/{userName} – создание заказа пользователем.
* GET /api/ord/orders/{userName}/{orderId}– получение конкретного заказа.
* PATCH /api/ord/orders/{userName}/{orderId}/{itemId} – редактирование предмета в заказе.
* PUT /api/ord/orders/{userName}/{orderId}/info– редактирование основоной информации о заказе.
* PATCH /api/ord/orders/{userName}/{orderId}/info– редактирование основоной статуса заказа.

#### Структура БД

Таблица 1 — Структура таблицы «Заказ (order\_table)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| address | Адресс | VarChar(255) |
| city | Город | VarChar(255) |
| comment | Комментарий к заказу | VarChar(255) |
| created | Дата создания | DATETIME |
| name | Имя пользователя | VarChar(255) |
| status | Статус заказ | VarChar(255) |
| total\_price | Стоимость всего заказа | INT |
| user name\_ | Ник нейм пользователя | VarChar(255) |

Таблица 2 — Структура таблицы «Элемент заказа (order\_item)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| id | Уникальный идентификатор | BIGINT |
| amount | Количество предметов | INT |
| price | Цена | INT |
| product\_id | Идентификатор продукта | BIGINT |

Таблица 3— Структура таблицы «Элемент в заказе (order\_table\_cart)».

|  |  |  |
| --- | --- | --- |
| Название поля | Описание поля | Тип данных |
| cart\_id | Уникальный идентификатор корзины, связанный с таблицей 1 | BIGINT |
| item\_id | Уникальный идентификатор предмета, связанный с таблицей 2 | BIGINT |

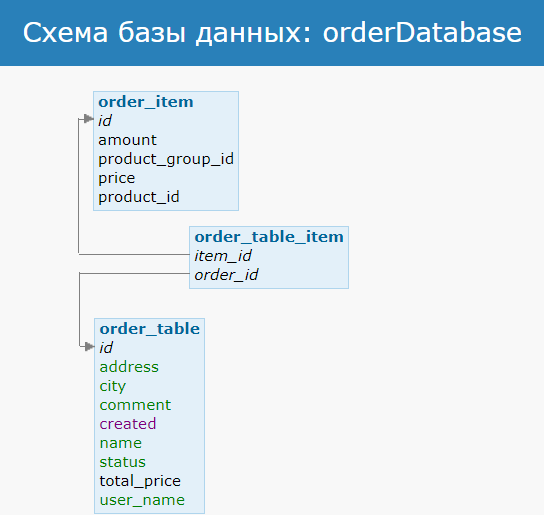


Рисунок 21 – Схема базы данных сервиса заказов

## Тестирование и результаты

В результате компиляции программы было получено восемь исполняемых файлов

«authorization-service-0.0.1-SNAPSHOT.jar», «eureka-server-0.0.1-SNAPSHOT.jar», «gateway-0.0.1-SNAPSHOT.jar», «product-service-0.0.1-SNAPSHOT.jar», «cart-service-0.0.1-SNAPSHOT.jar», «order-service-0.0.1-SNAPSHOT.jar», « recommendation-service-0.0.1-SNAPSHOT.jar», «review-service-0.0.1-SNAPSHOT.jar». Для его запуска можно использовать команду «java –jar product-service-0.0.1-SNAPSHOT.jar». Результат запуска представлен на рисунке 24.

Text

Description automatically generated

Рисунок 24 — Результат работы исполняемого файла «product-service-0.0.1-SNAPSHOT.jar» сразу после запуска.

Для проверки работоспособности приложения используется программа Postman, поскольку она бесплатная и обладает широким функционалом по отладке REST API. Чтобы пользоваться полным функцианалом системы, система должна понимать с кем работает , для этого мы должны пройти процесс регистрации и авторизации, для этого сделаем POST запрос на регистрацию в системе «/api/authorization», а затем POST запрос на вход в систему «/api/authorization/login».

Graphical user interface, text, application, email

Description automatically generated

Рисунок 25 — Результат регистрации пользователя

Graphical user interface, text, application, email

Description automatically generated

Рисунок 26 — Результат входа в систему

После успешного входа в систему нам выдаетмя JWT который служит для индификации пользователя в систему. Давайте проверим основной функцианал системы, а имменно просмотрим существующие продукты , добавим их в корзину и сделаем заказ. Получим все продукты первой группы для этого сделаем GET запрос «/api/prod/1/products».

Graphical user interface, text, application, email

Description automatically generated

Рисунок 27 — Получения продуктов определенной группы

Для работы с сервисами корзины и заказов в заголовках http запроса должен быть уникальный токен пользователя.

Создадим корзину и положим туда продукт, для этого воспользуемя GET запросом на создании или получение корзины «/api/cart/carts/user/{userName}»

Graphical user interface, text, application, email

Description automatically generated

Рисунок 28 — Получения или создание корзины

Для добавление продукта в корзину отправим POST запрос «/api/cart/carts/user/{userName}/{cartId}»

Graphical user interface, text, application, email

Description automatically generated

Рисунок 29 — Добавление продукта в корзины

Для создания заказа отправим POST запрос «/api/ord/orders/{userName}»

Graphical user interface, text, application, email

Description automatically generated

Рисунок 30 — Создание заказа

Для проверки работоспособности системы были написаны автотесты.

Автотест (автоматизированный тест) – это скрипт, имитирующий взаимодействия пользователя с приложением, цель которого – локализация ошибок в работе программного обеспечения.

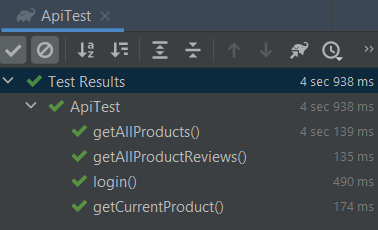


Рисунок 22 – Результат запуска автотестов

# Глава 4. Документация

## KDoc.

Документация кода является неотъемлемой частью разработки программного обеспеченья. В своем проекте я использую KDoc. Это язык, используемый для документирования кода Kotlin (эквивалент JavaDoc в Java). По сути, KDoc объединяет синтаксис JavaDoc для блочных тегов (расширенный для поддержки специфических конструкций Kotlin) и Markdown для встроенной разметки.

Рассмотрим пример документации для «Common» модуля

Graphical user interface, application, email

Description automatically generated

Рисунок 23 – Пример документации KDoc.

На рисунке 23 показана главная страница документации.

1. Название модуля, для которого мы сгенирировали документацию.
2. Список пакетов.
3. Список методов и классов которые находятся в данном пакете.

Graphical user interface, text, application, email

Description automatically generated

Рисунок 23 – Пример описания методов пакета

Для генерации документации для определенного сервиса можно воспользоваться gradle таской : gradlew dokka

## Развертывание системы

Развертывание системы это процесс, который является частью жизненного цикла программного обеспечения. По сути это все действия, которые делают программную систему готовой к использованию.

Было бы не удобно запускать и настривать каждый сервис отдельно поэтому в своем проекте я использую докер. Докер — это открытая платформа для разработки, доставки и эксплуатации приложений. Docker позволяет разработчикам использовать локальные контейнеры с приложениями и сервисами. Что в последствии позволяет интегрироваться с процессом постоянной интеграции и выкладывания (continuous integration and deployment workflow). Также используется Docker Compose - это инструментальное средство, входящее в состав Docker. Оно предназначено для решения задач, связанных с развёртыванием проектов. Технология Docker Compose, если описывать её упрощённо, позволяет, с помощью одной команды, запускать множество сервисов.

Часть docker-compose файла используемого для запуска всех сервисов

version: "3.1"  
  
services:  
 product-database:  
 image: mysql:5.7  
 restart: always  
 environment:  
 MYSQL\_ROOT\_PASSWORD: root  
 MYSQL\_DATABASE: productDatabase  
 MYSQL\_TCP\_PORT: 3305  
 ports:  
 - 3305:3305volumes:  
 - D:/GitHub/graduate-project/databases/product/var/lib/mysql57-data:/var/lib/mysql

eureka-server:  
 build: eureka-server/  
 container\_name: eureka-server  
 restart: always  
 ports:  
 - 8761:8761  
  
gateway:  
 build: gateway/  
 container\_name: gateway  
 restart: always  
 hostname: gateway  
 ports:  
 - 8762:8762  
 depends\_on:  
 - eureka-server  
  
product-service:  
 build: product-service/  
 container\_name: product-service  
 ports:  
 - 8089:8089  
 depends\_on:  
 - gateway  
 - eureka-server  
 links:  
 - product-database

Пример файл для создания докер образа для определенного сервиса

FROM openjdk:11.0.6-jre  
  
RUN mkdir /recommendation-service  
  
COPY ./build/libs/recommendation-service-0.0.1-SNAPSHOT.jar /recommendation-service/recommendation-service.jar  
  
WORKDIR /recommendation-service  
  
CMD ["java", "-server", "-jar", "recommendation-service.jar", "--spring.profiles.active=docker"]

Для запуска всех сервисов можно воспользоваться командой: «docker-compose up -d»

# Заключение

В ходе выполнения данной работы была изучена методология проектирования приложения с микросервисной архитектурой и приобретены навыки работы с фрейворком Spring.

При реализации поставленной задач были получены следующие результаты:

- разработана структура базы данных для СУБД MySQL

- разработан прикладной интерфейс для взаимодействия с остальными микросервисами приложения (REST API)

- Разработано и отлажено приложение «Интернет магазин электроники».

Функционал каждого сервиса реализован с учетом технических требований, сервисы интегрированы между собой

# Используемый источники

Книжные источники

Леон Шкляр, Рич Розен. Архитектура веб-приложений. — М.: «Эксмо», 2010. — С.35-150.

Ньюмен С. Создание микросервисов = Building Microservices. — СПб.: «Питер», 2016. — С. 56-246

Крис Ричардсон: Микросервисы. Паттерны разработки и рефакторинга - СПб.: «Питер»,2019. – C 63-343

Козмина Юлиана, Харроп Роб. Spring 5 для профессионалов - Диалектика-Вильямс, 2019. – 29-307

Интернет источники

<https://en.wikipedia.org/wiki/Java_virtual_machine>

<http://javacogito.net/index.php/Спецификация_виртуальной_машины_Java#.D0.9D.D0.B5.D0.BC.D0.BD.D0.BE.D0.B3.D0.BE_.D0.B8.D1.81.D1.82.D0.BE.D1.80.D0.B8.D0.B8>

<https://ru.wikipedia.org/wiki/Microsoft_SQL_Server>

<https://tproger.ru/translations/sqlite-mysql-postgresql-comparison/>

<https://medium.com/@kirill.sereda/spring-cloud-netflix-eureka-по-русски-5b7829481717>