Оглавление

[Перечень принятых сокращений 3](#_Toc72424097)

[Введение 3](#_Toc72424098)

[Глава 1. Техническая часть 3](#_Toc72424099)

[Описание предметной области. 3](#_Toc72424100)

[Анализ подходов и методов решения задачи. 4](#_Toc72424101)

[Разработка технического задания 4](#_Toc72424102)

[Глава 2. Специальная часть 5](#_Toc72424103)

[Анализ предметной области 5](#_Toc72424104)

[Выбор среды разработки 5](#_Toc72424105)

[Анализ и выбор СУБД 6](#_Toc72424106)

[Выбор реляционной СУБД 8](#_Toc72424107)

[MS SQL Server 8](#_Toc72424108)

[Oracle Database 9](#_Toc72424109)

[PostgreSQL 9](#_Toc72424110)

[MySQL 10](#_Toc72424111)

[Выбор архитектуры приложения 11](#_Toc72424112)

[Монолитная архитектура приложения 11](#_Toc72424113)

[Микросервисная архитектура приложения 12](#_Toc72424114)

[Выбор средств проектирования 14](#_Toc72424115)

[Выбор языка программирования 14](#_Toc72424116)

[Язык программирования Go 14](#_Toc72424117)

[Язык программирования Python 14](#_Toc72424118)

[Язык программирования JavaScript 14](#_Toc72424119)

[Язык программирования Java 15](#_Toc72424120)

[Язык программирования Kotlin 15](#_Toc72424121)

[Выбор языка программирования 15](#_Toc72424122)

[Выбор библиотек для разработки 16](#_Toc72424123)

[Модули Spring для разработки приложения 17](#_Toc72424124)

[Spring MVC 17](#_Toc72424125)

[Spring JPA 17](#_Toc72424126)

[Spring Security 17](#_Toc72424127)

[Spring Cloud Eureka 17](#_Toc72424128)

[Описание архитектуры REST API 17](#_Toc72424129)

[Модель клиент-сервер 18](#_Toc72424130)

[Отсутствие состояния 18](#_Toc72424131)

[Кэширование 18](#_Toc72424132)

[Единообразие интерфейса. 19](#_Toc72424133)

[Многослойная система 19](#_Toc72424134)

[Глава 3. Проектная часть 20](#_Toc72424135)

[Проектирование 20](#_Toc72424136)

[Реализация 21](#_Toc72424137)

[Gateway. 21](#_Toc72424138)

[Eureka. 23](#_Toc72424139)

[Authorization service 25](#_Toc72424140)

[Структура БД 29](#_Toc72424141)

[REST API 29](#_Toc72424142)

[Product service 31](#_Toc72424143)

[REST API 31](#_Toc72424144)

[Структура БД 37](#_Toc72424145)

[Recommendation service 37](#_Toc72424146)

[REST API 38](#_Toc72424147)

[Структура БД 38](#_Toc72424148)

[Review service 39](#_Toc72424149)

[Cart service 39](#_Toc72424150)

[Order service 39](#_Toc72424151)

[Отладка программных модулей. 39](#_Toc72424152)

[Глава 4. Документация 39](#_Toc72424153)

[Описание технических требований к условиям эксплуатации программного продукта (модуля). 39](#_Toc72424154)

[Программа и методика испытаний. 39](#_Toc72424155)

[Руководство программиста и пользователя. 39](#_Toc72424156)

[Заключение 39](#_Toc72424157)

[Используемый источники 40](#_Toc72424158)

# 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.

Gateway

# Введение

Где как , актуальность

Цель

Задачи

# Глава 1. Техническая часть

## Описание предметной области.

В ходе выполнения работы рассматривается деятельность магазина электроники. Необходима выделить основные аспекты работы магазина и создать бекэнд для минимальной работы интернет-магазина

//TODO может быть дополнить

## Анализ подходов и методов решения задачи.

//Как вариант это - CMS vs нормальное приложение или вообще убрать этот раздел

## Разработка технического задания

Требования к системе:

Система должна предоставлять следующие операции:

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.
* Исправление ошибок.
* Техническая поддержка системы.

# 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 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

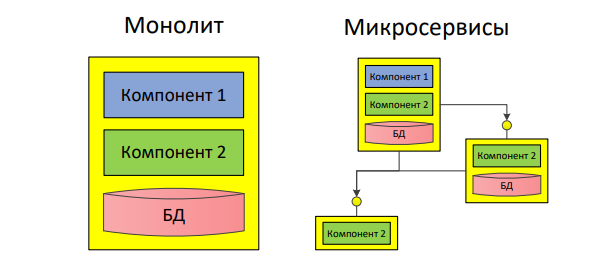


Рисунок 2 — Сравнение монолитной и микросервисной архитектур

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. 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.

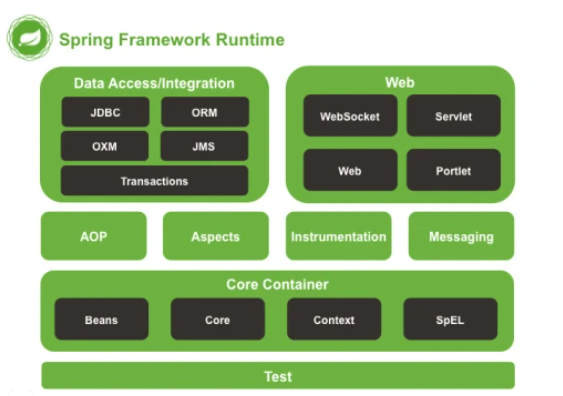


Рисунок 3 — Инфраструктура Spring Framework

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.

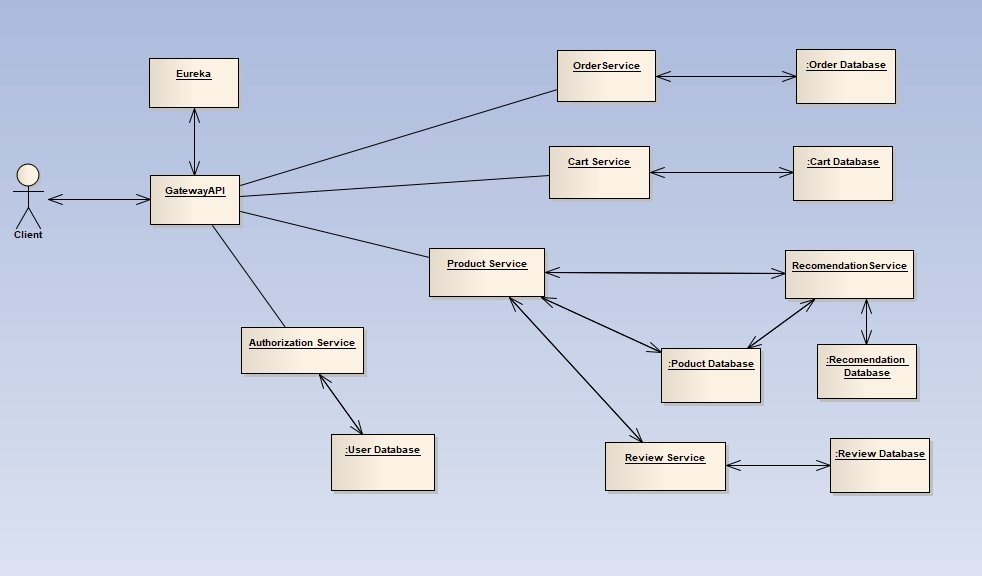


Рисунок 4 — Структурная схема интернет-магазина

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

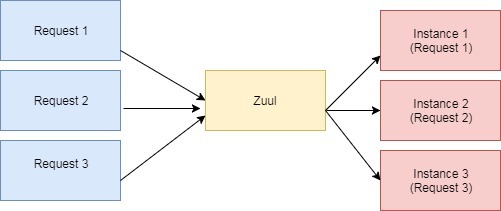


Рисунок 5 – Схема работы Zuul сервера

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}

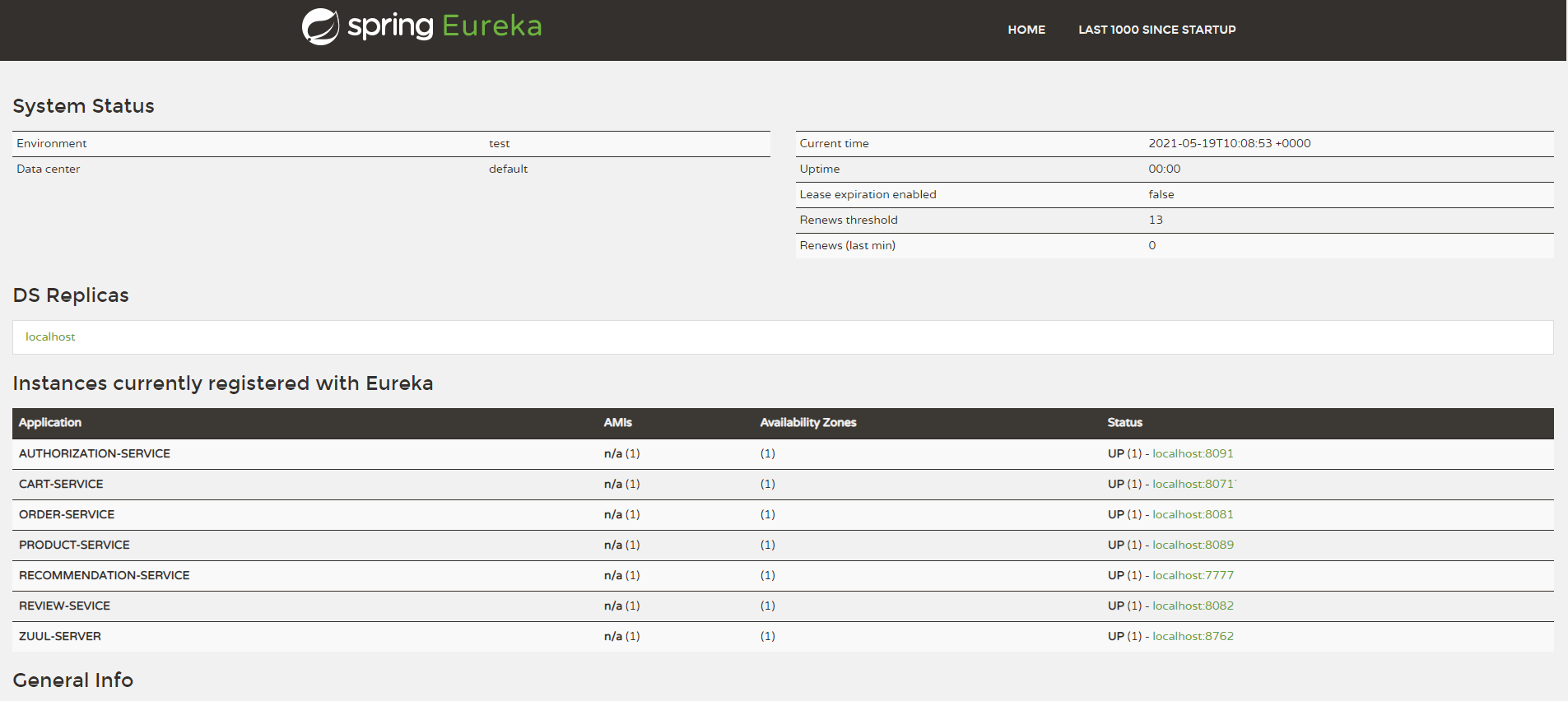


Рисунок 6 – Пример работы Eureka Server

На рисунке 6 показана работа еureka servera, 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"

}

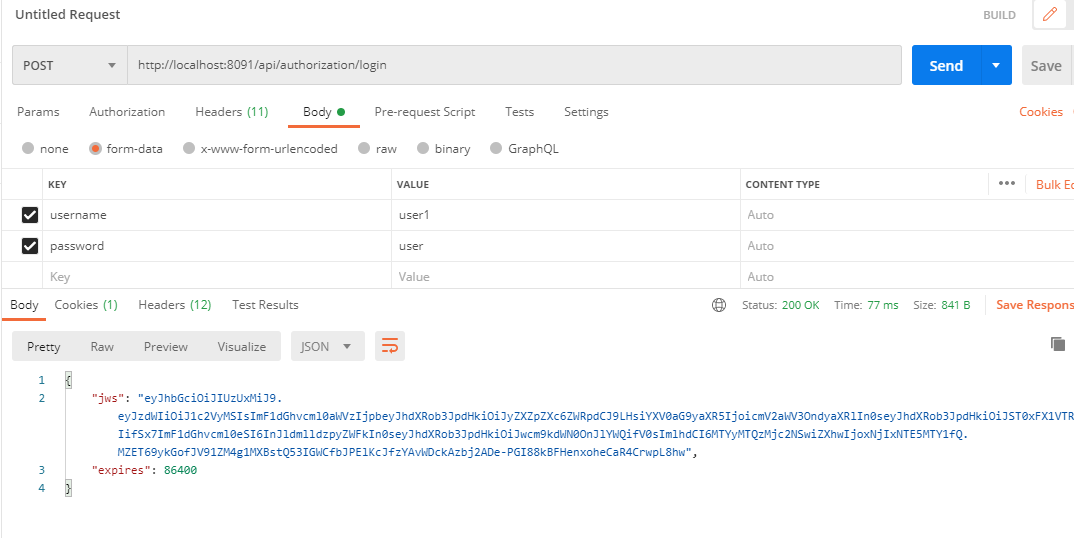
],

"iat": 1621428835,

"exp": 1621515235

}

Пример запроса на вход в систему и ответ от сервера



На рисунке 6 представлена диаграмма последовательности получения токена.

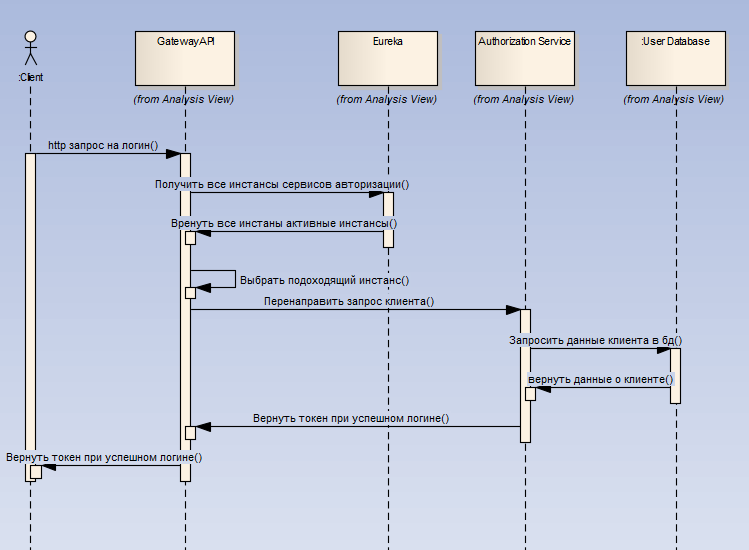


Рисунок 6 – Диаграмма последовательности получения JWT

#### 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) |

#### 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

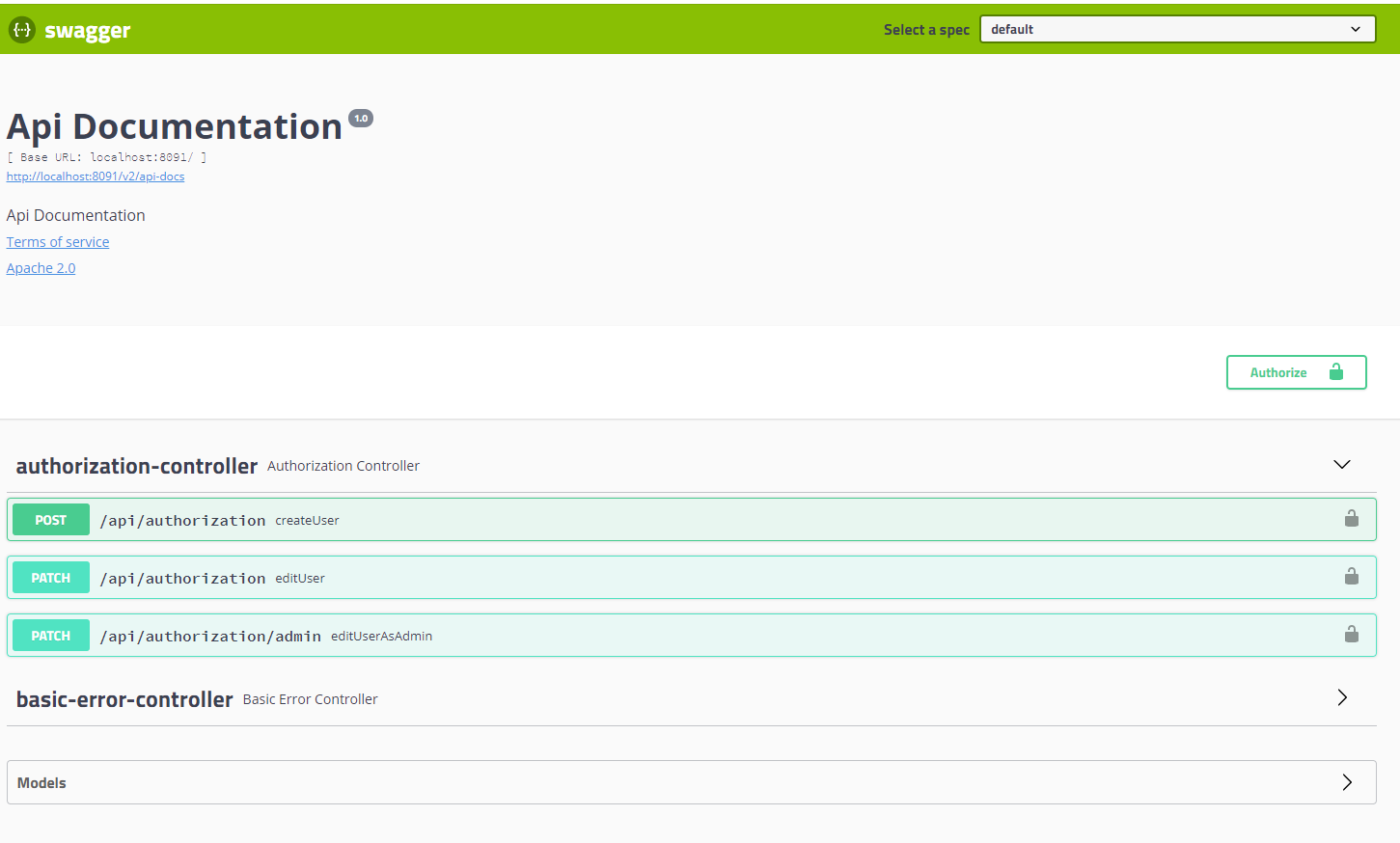


Рисунок 7 – Swagger страница для «Authorization service»

Как видно из рисунка 7 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».

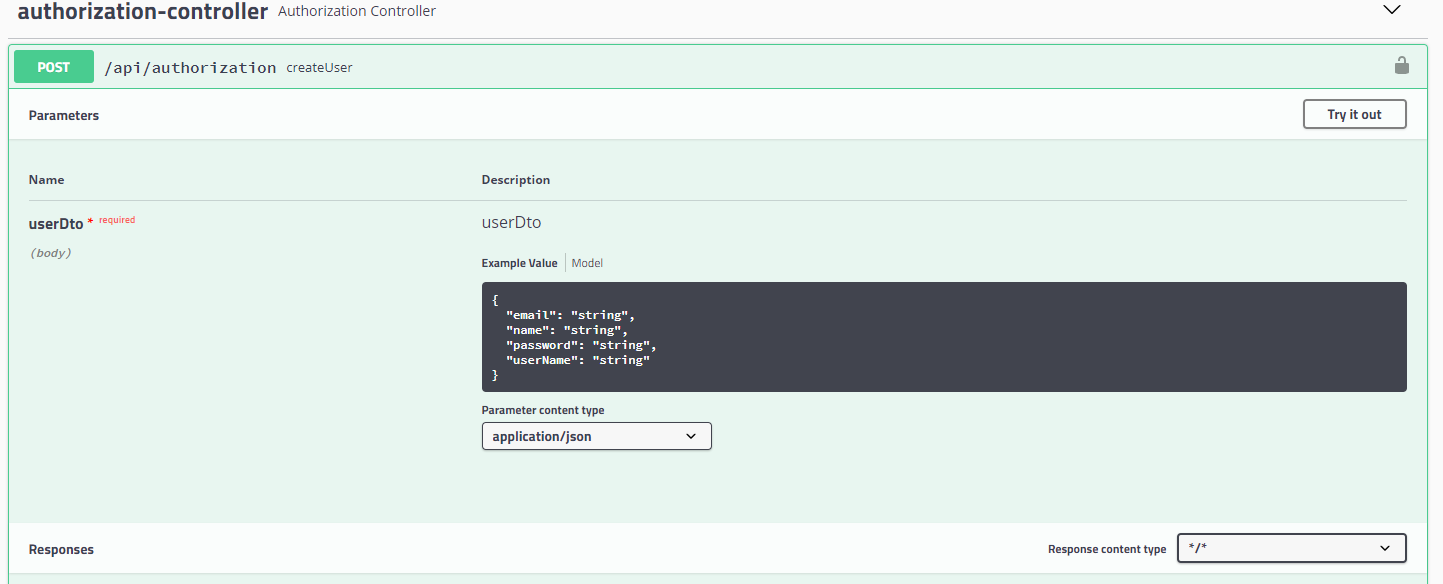


Рисунок 8 – Подробная информация об эндпоинте регистрации

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 9 – 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.

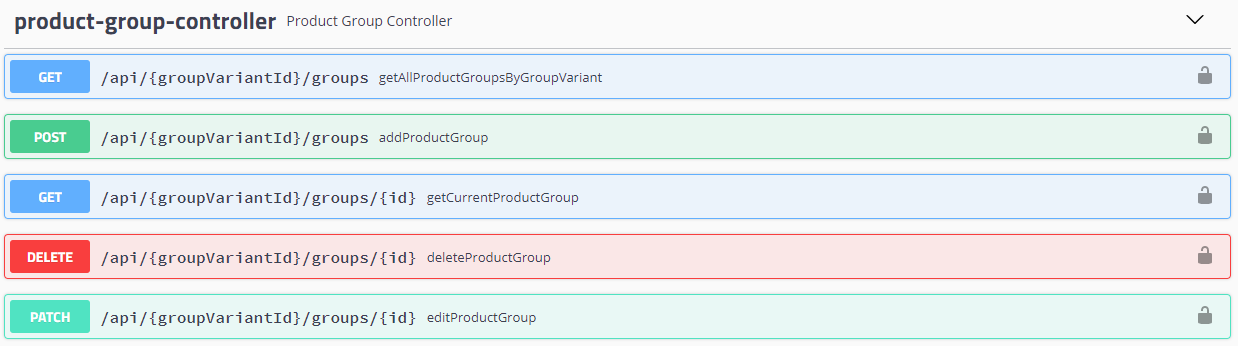


Рисунок 10 – API для работы с продуктовой группой

* 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.

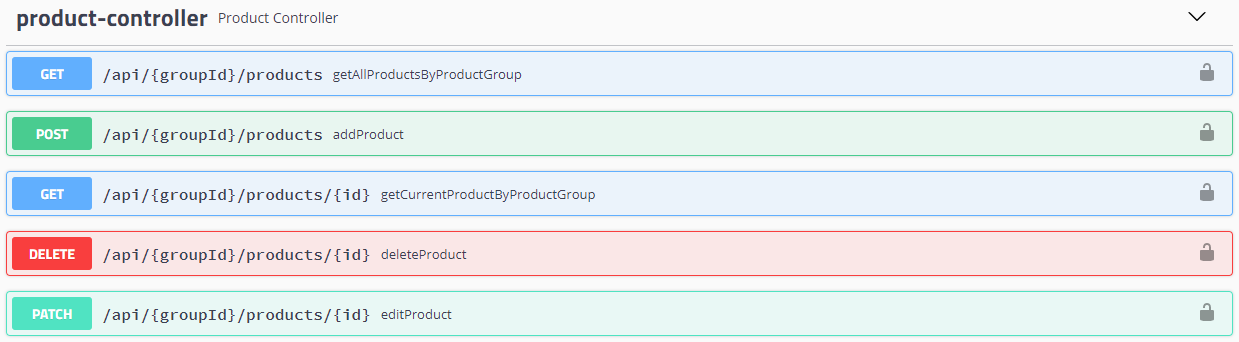


Рисунок 11 – API для работы с продуктовом

* 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

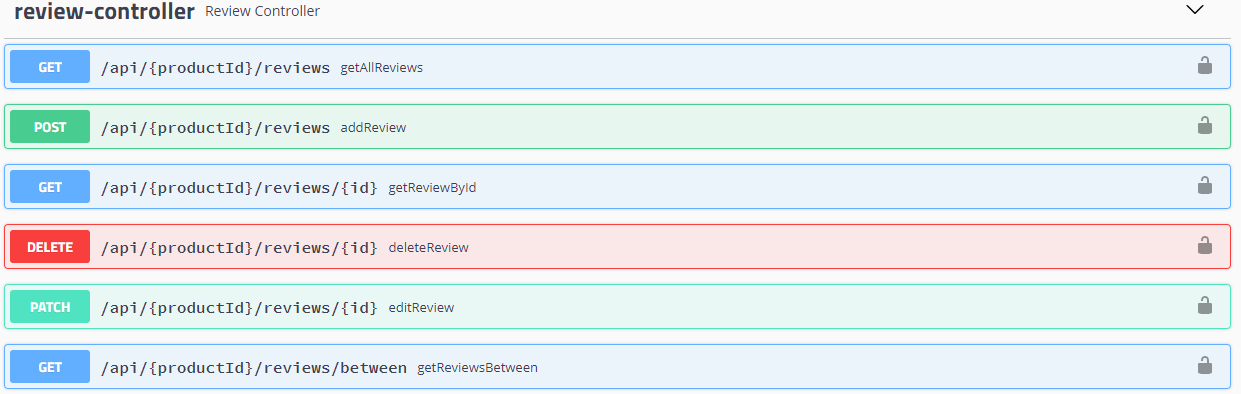


Рисунок 12 – API для работы с отзывами

* 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

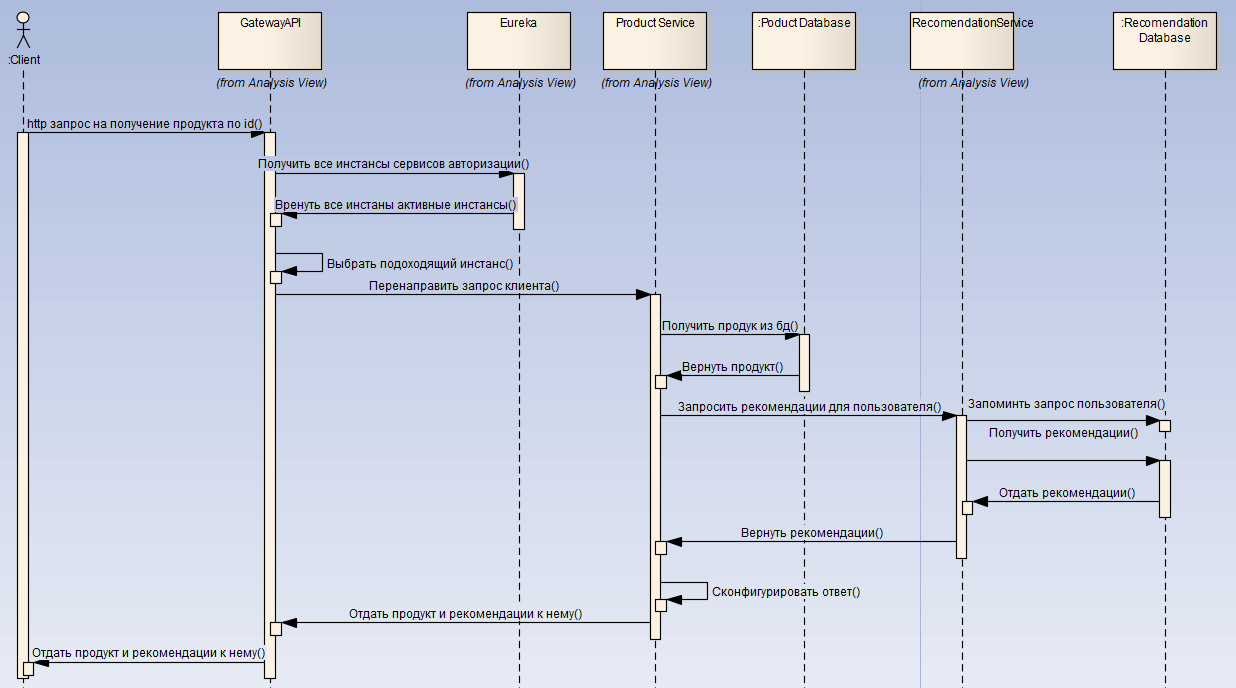


Рисунок 13 — Диаграмма последовательности запроса get запроса

* 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 |

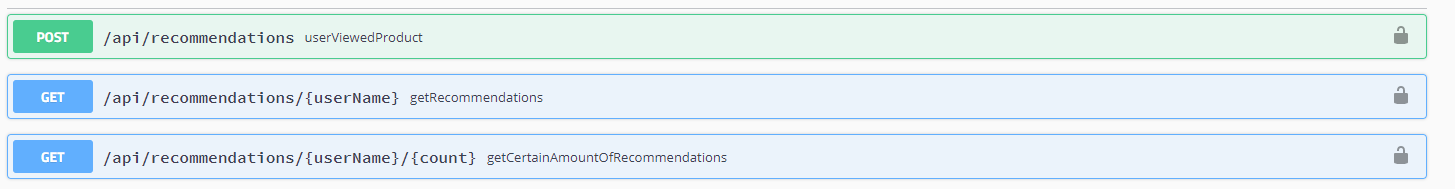
Таблица 3 — Структура таблицы «Продукт (product)».

|  |  |  |
| --- | --- | --- |
| 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 |

### Recommendation service

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

#### REST API



* 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 13 можем посмотреть 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 |

### Review service

### Cart service

### Order service

## Отладка программных модулей.

# Глава 4. Документация

## Описание технических требований к условиям эксплуатации программного продукта (модуля).

## Программа и методика испытаний.

## Руководство программиста и пользователя.

# 

# Заключение

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

При реализации поставленной задачи 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

<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>

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