САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Направление: 02.03.02 «Фундаментальная информатика и информационные технологии

ООП: Программирование и информационные технологии

**Диплом**

**Тема задания: Построение серверной составляющей веб приложения для медиа-платформы магазина дизайнерских эскизов одежды с интеграцией фриланс-биржи портных**

**Выполнил: Панюшин Даниил Васильевич 19Б12-пу**

Фамилия И. О. номер группы

**Руководитель научно-  
исследовательской работы: Раевская Анастасия Павловна, доцент кафедры математической теории экономических решений Санкт-Петербургского Государственного Университета, кандидат физико-математических наук**

ФИО, должность, ученая степень

Санкт-Петербург  
2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3

Описание проекта 3

Проделанная работа 3

Глава 1. Архитектура кода 8

1.1 Бизнес процессы 8

Глава 2. Слой данных 11

2.1 Реляционные СУБД 11

2.2 Нереляционные СУБД 12

2.3 Миграции 14

Глава 3. Сервисный слой 17

3.1 Механизмы авторизации и аутентификации 17

3.2 Механизмы взаимодействия между сервисами 21

3.3 Механизмы основной бизнес логики 23

Глава 4. Слой контроллеров 25

Глава 5. Развертывание 26

ЗАКЛЮЧЕНИЕ 27

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 28

# ВВЕДЕНИЕ

## Описание проекта

Существует множество различных брендов, создающих вещи самых разных ценовых категорий. В таких компаниях трудится множество дизайнеров, чьи работы обречены выпускаться под именем компании их работодателя. Наша платформа позволит талантливым авторам выкладывать свои работы под собственным именем и зарабатывать этим.

Данная платформа позволит неизвестным дизайнерам сделать себе имя и предоставит простым пользователям возможность приобретать вещи, недоступные в прочих магазинах.

Стоит отметить, что платформа является прослойкой между тремя сторонами: покупателем, дизайнером и исполнителем. Последний необходим для избавления авторов эскизов от ручного создания продукта. Это позволит дизайнерам и портным заниматься своим делом.

Пользователь регистрируется как покупатель, дизайнер или исполнитель:

* Покупатель может добавлять эскизы в избранное, а также заказывать их. При заказе покупатель может связаться с дизайнером и задать ему все необходимые вопросы.
* Дизайнер выкладывает свои эскизы на платформу. После поступления заказа от покупателя, дизайнер отвечает на вопросы, связанные с заказом. После оплаты покупателем дизайнер приступает к выполнению заказа. Он может самостоятельно изготовить товар, либо воспользоваться услугами исполнителя, найдя его через нашу платформу.
* Исполнитель может просмотреть список доступных заказов от дизайнеров, связаться с ними и договориться об условиях.

После выполнения заказа дизайнер или исполнитель отправляет готовое изделие покупателю.

## Проделанная работа

На данный момент выбраны и разобраны технологии, используемые для создания приложения:

### Язык программирования – Java.

Причинами такого решения являются надёжность и безопасность, предоставляемая языком Java. Это необходимо для нашего проекта, так как имеет место обработка приватных данных клиентов.

Другой причиной такого решения – поддержка данного языка, сильно облегчающая работу с ним.

Система управления базами данных – PostgreSQL, MongoDB и Redis.

Первая СУБД – реляционная. Она выбрана для того, чтобы быть использованной как в аутентификации и авторизации, так и при создании заказов. Причинами такого выбора являются: строгая схема данных, большое количество связей между хранимыми сущностями и необходимость использования транзакций. PostgreSQL выбрана в качестве конкретной СУБД так как она масштабируема, имеет множество встроенных функций и графических интерфейсов, облегчающих использование.

Вторая СУБД – документоориентированная. Она выбрана так как: данные слабо связны, меняющие свою структуру по мере развития проекта (например, добавляются новые поля), необходима высокая скорость обработки запросов, которую предоставляют NoSQL базы данных, потенциальная необходимость серьёзного масштабирования (NoSQL базы лучше подходят для хранения больших объёмов данных и их легче горизонтально масштабировать как раз за счёт слабой связности данных и денормализации[[1]](#footnote-1)). Именно MongoDB выбрана, так как наиболее популярна и обладает расширенным функционалом по сравнению с аналогами[**Ошибка! Источник ссылки не найден.**].

Третья СУБД – in-memory[**Ошибка! Источник ссылки не найден.**] типа ключ-значение. Она служит для кэширования[[2]](#footnote-2) на уровне приложений. Данная база данных находится между веб приложением и постоянным хранилищем данных.

Фреймворки ­­­­– Spring, Swagger.

Spring – наиболее развитая и популярная экосистема для разработки веб приложений. Функционал позволяет создавать и настраивать практически все аспекты целевого приложения, в том числе работу с базами данных, защиту приложения, реактивное программирование и прочее.

Swagger – популярный фреймворк, предоставляющий удобный графический интерфейс для как визуализации сгенерированной документации, так и для создания различных запросов к сервису.

### Система сборки – Gradle.

* Было решено использовать её из-за следующих преимуществ:
  + позволяет выполнять инкрементные сборки;
  + имеет возможность безопасного кеширования;
  + позволяет определять пользовательские правила указания версии для динамической зависимости и разрешать конфликты версий;
  + имеет полностью настраиваемую модель выполнения;
  + имеет возможность использования пользовательских дистрибутивов;
  + позволяет настраивать среды сборки на основе версий без необходимости настраивать их вручную.

Средства разработки – IntelliJ IDEA, PgAdmin 4, MongoDBCompass, Postman.

InteliJ IDEA – самый популярная среда разработки на Java.

PgAdmin 4 – инструмент для управления базами данных на движке PostgreSQL.

MongoDBCompass – инструмент для управления базами данных на движке Mongo.

Postman – инструмент для создания HTTP запросов.

Помимо выбора технологий были выбраны и описаны подходы к созданию структуры приложения и построена архитектура реализуемого сервиса.

Было решено использовать микросервисный подход ввиду его гибкости в разработке и поддерживании, простой масштабируемости и высокой доступности. Достоинства микросервисов особенно ярко проявляются при разворачивании приложения в контейнерах в системах оркестрации, с помощью которых и будет работать система на облачных сервисах.

При микросервисном подходе компоненты системы могут взаимодействовать друг с другом с помощью обычных HTTP-запросов или брокеров сообщений. Именно последние были выбраны так как они не только позволяют обмениваться огромным числом сообщений, но и следят за тем, чтобы сообщения были доставлены, тем самым поддерживая целостность системы.

Сервисы используют в своей работе реляционные и нереляционные базы данных для хранения информации от пользователей, а также in-memory базу для операций кэширования.

Для авторизации и аутентификации в приложении используются JWT токены.

Система будет запущена в кластере Kubernetes с использованием контейнеров Docker. Такая комбинация выбрана так как именно эти технологии являются де-факто стандартом при разворачивании подобных приложений и предоставляют не только эффективные инструменты, но и весьма удобные ввиду простоты интерфейсов, качественной документации и широкой поддержки в среде разработчиков.

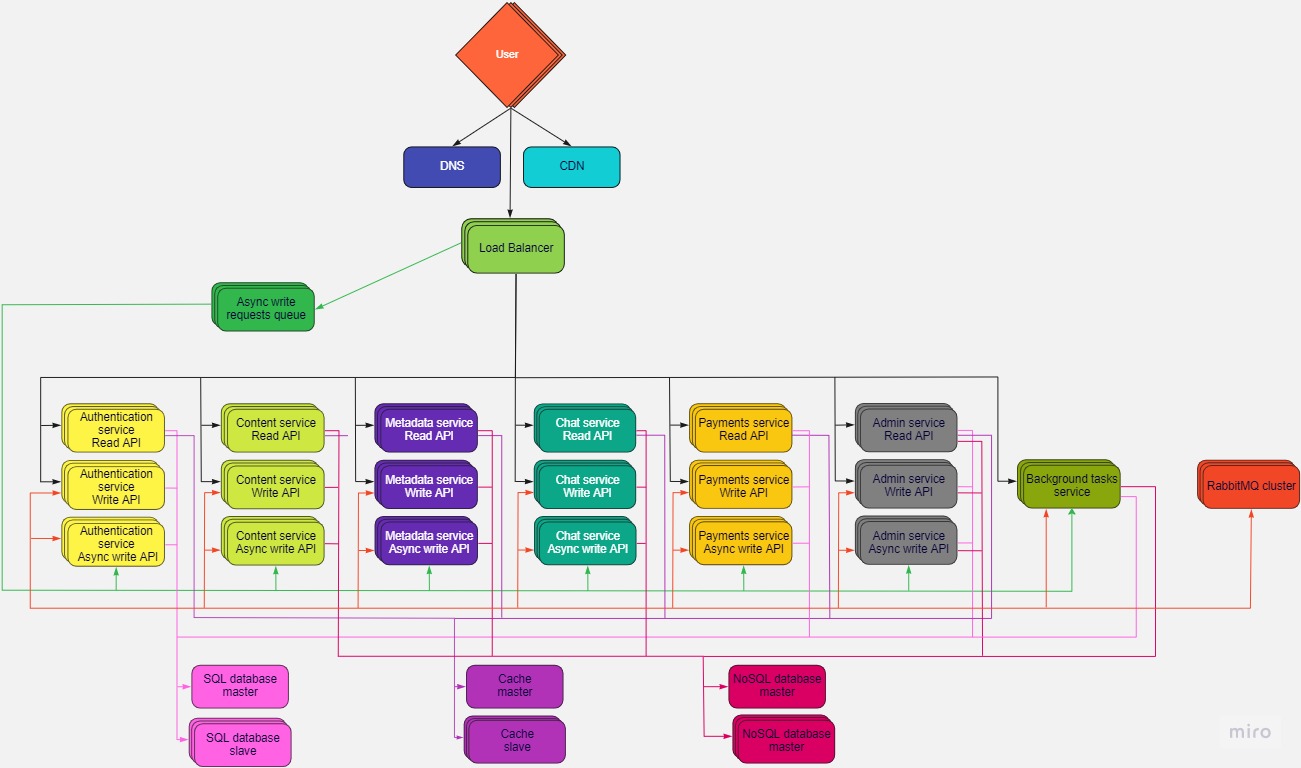
Отдельно стоит отметить, что при запуске приложения на «боевом» сервере, будут использоваться сети доставки содержимого (CDN) для ускорения получения пользователями контента и серверы доменных имён (DNS) для балансировки нагрузки и определения географически ближайшего сервера приложения.

Рисунок 1. Архитектура системы для высокой нагрузки.

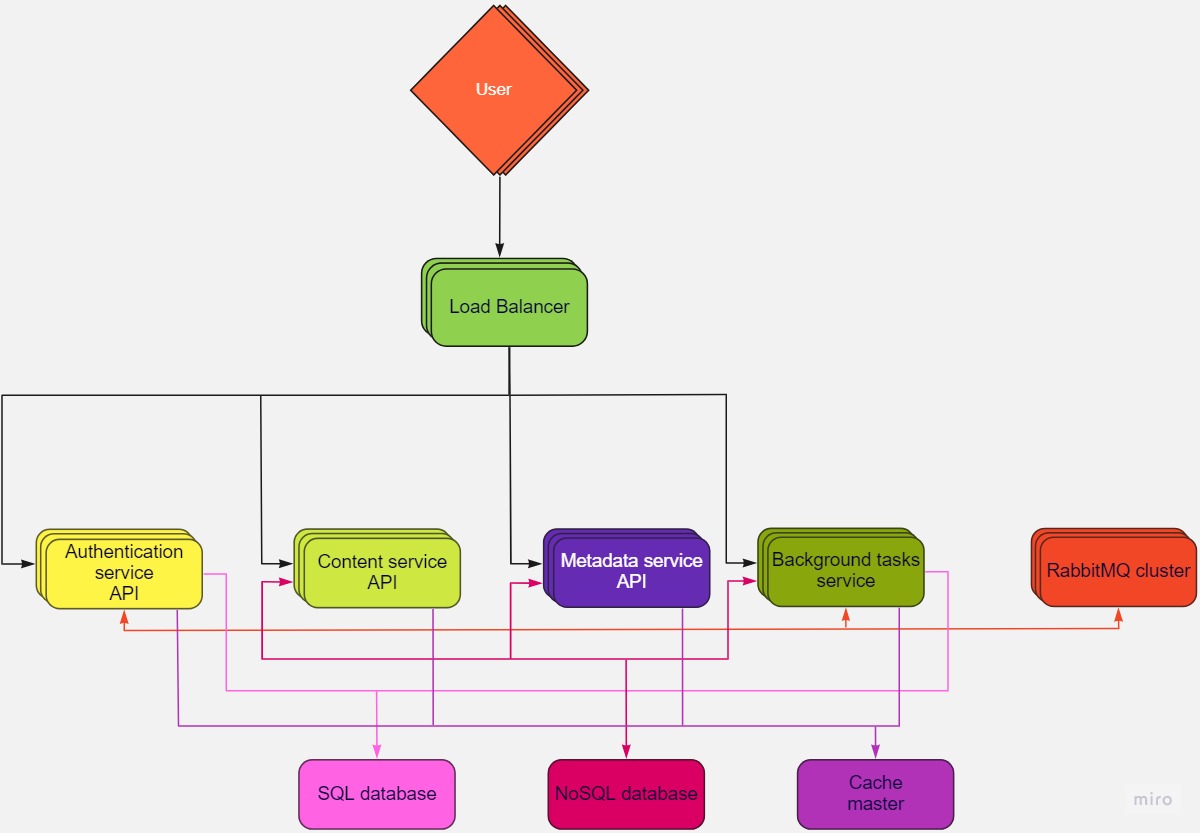
Данная архитектура подходит для приложения, обрабатывающего около одного миллиона постоянных пользователей. К сожалению, поддержание такой системы не только трудозатратно, но и дорого. К тому же на начальном этапе работы нашего сервиса количество пользователей будет значительно меньше. Поэтому в дальнейшей работе я опишу более простую версию приложения, но по-прежнему содержащую все ключевые элементы. Ввиду локального развертывания в данной версии отключены механизмы балансировки нагрузки через DNS, кэширования контента в CDN, репликации баз данных, а также, ввиду отсутствия крайне высокой нагрузки, и механизм асинхронной обработки запросов.

Рисунок 2. Архитектура приложения для малой нагрузки

# Глава 1. Архитектура кода

## Бизнес процессы

Понимание бизнес процессов очень важно при реализации системы. От этого зависит то, на сколько удачно произойдёт разбиение системы на микросервисы и, соответственно, предметной области на сущности баз данных. Приведённая выше архитектура была построена после осознания основных бизнес процессов приложения, которыми являются:

* Регистрация, авторизация и аутентификации
* CDUD[[3]](#footnote-3) операции с пользовательской информацей
* CRUD операции с основынм контентом приложения, которым являются скетчи, тэги и сопутствующие медиафайлы (фотографии, видео и прочее)
* CRUD операции с метаниформацией (лайки, комменты, отзывы)
* Операции поддержания целостности системы (отчистка от неактуальной информации и прочее)
* Операции уведомлений (например, письма для подтверждения авторизации)
* Чат между пользователями
* Операции оплаты
* CRUD операции администратора системы
* ERP[[4]](#footnote-4) операции (например сбор различного рода статистики).

Поведенческая диаграмма перечисленных выше процессов изображена на Рисунке 3.

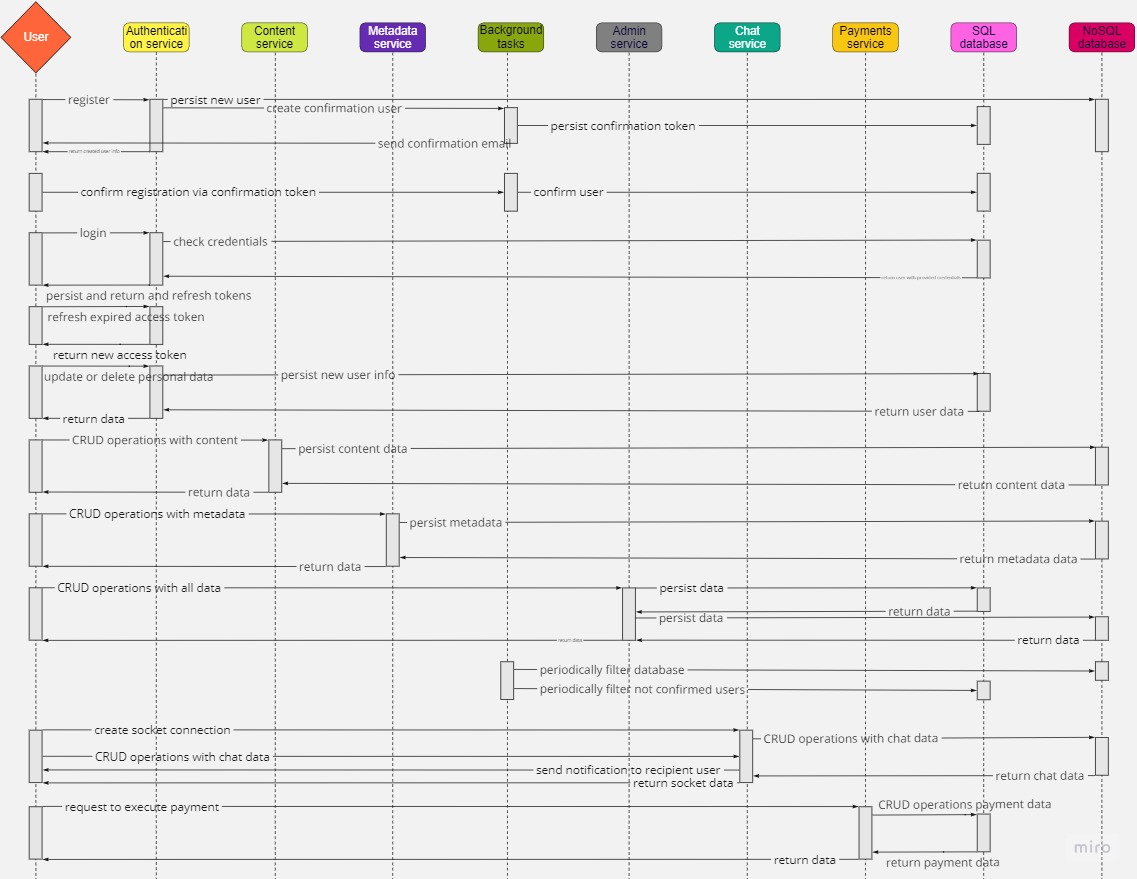
 При определении способа разбиения приложения на микросервисы была создана структурная диаграмма, не только описывающая схему базы данных, но и отображающая ограниченные контексты приложения – изображена данная диаграмма на Рисунке 4. Естественно с развитием приложения схема базы данных будет меняться, но ограниченные контексты менее сильно подвержены изменениям (естественно, до момента проведения углубляющего рефакторинга [2, стр. 287]).

Рисунок 3. Поведенческая диаграмма приложения.

## Работа с фреймворком Spring

При написании Java-приложения, использующего фреймворк, необходимо понимать принципы его работы. Фактический, Spring представляет собой контейнер для внедрения зависимостей – процесс предоставления внешней зависимости некоторому программному компоненту. Это является формой принципа инверсии управления. Таким образом, когда в программе, использующей внедрение зависимостей, создаётся объект, забота о построении требуемых ему зависимостей передаётся внешнему, специально предназначенному для этого механизму. В Spring таким механизмом и является ранее упомянутый контейнер.

Таким образом, работая с Spring, код пишется иным образом, нежели обычное Java-приложений. Создавая объекты, зависящие от других объектов, необходимо давать понять фреймворку о существовании зависимости. В начальных версиях Spring это происходило на основании специальных xml файлов, хранящих информацию о взаимосвязях между классами[16]. Затем в версии 3.1 разработчики фреймворка добавили возможность конфигурации приложения на основании внутри самого Java кода[17]. Для этого используются специальные аннотации[18][19] и классы[20].

Помимо создания объектов, Spring контейнер берёт на себя ответственность за инициализацию этих объектов, их конфигурацию и уничтожение. Для этого существуют специальные механизмы, вызывающие определённые методы на соответствующем этапе жизненного цикла объекта. То, где и какие методы будут вызваны определяется с помощью одного из видов конфигурации контекста, описанных ранее.

## 

Рисунок 4. Структурная диаграмма приложения.

# Глава 2. Слой данных

## 2.1 Реляционные СУБД

В нашем приложении слой данных, взаимодействующий с SQL СУБД, реализован с помощью технологии Spring Data JPA. Этот слой предоставляет абстракцию для взаимодействия с базой данных, предоставляя набор методов для выполнения различных операций с данными, таких как чтение, запись, обновление и удаление оных.

Внутри слоя данных существуют классы, которые отображают таблицы базы данных. Каждый класс представляет отдельную таблицу, и в нем определены поля, соответствующие столбцам этой таблицы.

Для работы с базой данных используется менеджеры транзакций, которые обеспечивают целостность данных и согласованность изменений, внесенных в базу данных.

Для лучшего понимания слоя данных нашего приложения можно обратиться к коду, описывающему сущность пользователя системы <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/model/User.java>.

Класс User отображает таблицу пользователей. Он аннотирован @Entity, что означает, что он является сущностью JPA, которая отображает таблицу базы данных. Аннотация @Table определяет имя таблицы в базе данных, которую отображает класс. Аннотация @Id указывает, что поле id является первичным ключом таблицы. Аннотация @GeneratedValue указывает, что значение id будет генерироваться автоматически. Аннотация @Column указывает на то, что поле класса соответствует колонке в таблице базы данных (имя которой указано в аргументе аннотации). Аннотация @ManyToMany указывают на то, что данное поле представляет собой связь типа многие ко многим на другую сущность и, соответственно, на другую таблицу в базе данных. Аннотация @JoinTable используется для настройки связи многие-ко-многим. Она указывает, какие столбцы должны быть использованы для связывания двух таблиц, и какое имя должна иметь таблица, которая будет создана для хранения этих связей, ведь, как мы знаем, связь многое ко многим реализовывается в реляционных базах данных с помощью дополнительной таблицы, столбцами которой являются первичные ключи исходных таблиц. Помимо @ManyToMany существуют и другие аннотации для описания связей других типов. Естественно, они так же настраиваются с помощью аннотаций @JoinTable. Схожим образом созданы все классы, представляющие сущности SQL СУДБ.

Рассмотрим код, описывающий программный интерфейс для выполнения CRUD операции с данными таблицы пользователей <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/repository/UserRepository.java>.

Первое что можно заметить – интерфейс наследуется от встроенного Spring Data Jpa интерфейса. Это необходимо для автоматической генерации тел методов, который используются для CRUD операций. При наследовании передаются два параметра с помощью «даймонд»[12] синтксиса – класс, представляющий сущность с которой будут происходить операции (в данном случае User) и тип первичного ключа этой сущности (тип поля, помеченного аннотацией @Id).

Рассмотрим метод getUserById. Можно предположить, что данный метод ищет пользователя по его уникальному идентификатору. Так и есть – при создании метода могут использоваться определённые шаблоны, такие как get, find, getAll, findAll. Они определяют будут ли искаться все объекты, подходящии описанию, или лишь один. Далее следуют поля (одно или несколько), по которым осуществляется поиск – в данном методе поиск происходит по полю Id. В аргументы метода необходимо добавить аргумент того же типа, то и поле сущности, по которому происходит поиск.

Так же можно осуществлять поиск не по полному совпадению, а по, например, вхождению (для строковых типов), по промежутку (для численных типов), по вхождению в коллекцию и прочее. Для этого в название метода необходимо добавить имя «типа поиска». Для вышеперечисленных типов поиска это, соответственно, contains, between, in. Естественно в аргументы метода добавляются соответствующие поля (для in этим полем является коллекция элементов, на вхождение в которую осуществляется проверка и выполнение метода). Примером такого метода является getAllByRolesIn.

## 2.2 Нереляционные СУБД

В нашем приложении помимо SQL СУБД используется документоориентированная NoSQL база данных MongoDB, которая информацию в виде документов BSON (Binary JSON), а не таблиц. Для работы с ней используются специальная библиотека Spring Data MongoDB, которая позволяет отображать Java-объекты в документы. Таким образом классы описывают не таблицу, как в реляционной базе данных, а коллекцию докуметов.

Рассмотрим пример сущности в MongoDB <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/document/Sketch.java>.

Аннотация @Document указывает, что данный класс представляет собой документ в некоторой коллекции (имя которой указывается в аргументе аннотации). Поле id должно быть типа String и помечено аннотацией @Id, чтобы быть использованным как уникальный идентификатор документа в MongoDB. Аннотация @Indexed создаёт индекс в базе данных. Аннотация @TextIndexed создаёт текстовый индекс, который позволяет выполнять операцию поиска по тексту. Аннотация @DocumentReference создаёт связь между документами коллекции. Это можно сравнить со связями между таблицами в SQL базах данных, за тем исключением, что в поле, помеченном этой аннотацией могут храниться любые типы данных, характеризующие связь. То есть если мы, например, хотим сделать связь один ко многим или многие ко многим, то в данном поле может храниться список всех других документов (или просто их идентификаторов). На обратной стороне связи так же может храниться любой доступный тип данных, или, например, поле вообще может отсутствовать (такие связи называют односторонними).

Можно заметить, что программные интерфейсы для CRUD операций документами практически идентичны интерфейсам для взаимодействия с данными, хранящимися в таблицах SQL баз данных – <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/repository/TagRepository.java>.

В нашем приложении для взаимодействия с данными используются не только встроенные инструменты фреймворка, но и их имплементации – <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/repository/sketch/SketchRepository.java>. Можно заметить что данный интерфейс наследуется не только от интерфейса MongoRepository, но и от интерфейса SketchRepositoryCustom, которые представляет собой интерфейс для взаимодействия с методами, описанными в классе SketchRepositoryImpl – <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/repository/sketch/SketchRepositoryImpl.java>. Внутри этих методов используется CriteriaAPI, позволяющее создавать запросы, использующие все возможности базы данных. Например, в методе findAllByTagsAndName используется поиск по тексту – один из инструментов движка базы данных MongoDB. Другим примером функционала Criteria API является возможность создания агрегирующих запросов, как это можно увидеть в методе countSketchesLikes класса <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/metadata-service/src/main/java/com/workshop/metadataservice/repository/metadata/like/LikeRepositoryCustomImpl.java>, где происходит подсчёт количества отметок «нравится» для каждой сущности эскиза.

Описание принципа того, как работает CriteriaAPI сводится к тому, что мы поэлементно программно строим запрос на языке манипулирования данными (SQL или его аналоги для взаимодействия с нереляционными СУБД) и выполняем его, получая результат.

Более подробно для агрегирующих запросов:

1. Создаём критерий агрегирования – GroupOperation
2. Создаём критерий фильтрации – MatchOperation
3. Объединяем всё в запрос – в данном случае в Aggregation
4. Выполняем запрос и получаем результат – AggregationResults.

Для запросов поиска по тексту:

1. Создаём запрос – Query
2. Создаём критерии фильтрации и добавляем их в запрос – Criteria
3. Создаём и добавляем критерий поиска по тексту – TextCriteria
4. Выполняем запрос используя абстракцию подключения к базе данных и получаем результат.

При необходимости есть возможность добавления пагинации в запросы – разбиение искомых данных на страницы, позволяющее получать не всю информацию в одном запросе, а в нескольких меньшего размера. Пагинация ускоряет получение части данных, но увеличивает количество обращений к СУБД при получения всего массива запрашиваемой информации.

## 2.3 Миграции

Миграция базы данных это изменение структуры базы данных от одной версии до другой. Они нужны для того, чтобы база данных находилась в том же состоянии, что и текущая версия кода приложения. Однако стоит заметить, что при использовании механизмов миграции можно не только менять структура БД, но и изменять хранимую в ней информацию, создавай, изменяя или удаляя записи.

В нашем приложении используется библиотека flyway для миграций SQL баз данных и Mongock для миграций баз данных MongoDB. Оба этих инструмента создают схемы в соответствующих СУБД, а также заполняют их изначальными данными. Кроме пользовательских таблиц или коллекций они поддерживают собственные сущности, содержащие информацию о выполненных миграциях. Это необходимо для поддержания согласованности версии базы данных с версией приложения.

Поговорим более детально про Flyway. При настройке данного инструмента в файле конфигурации сервиса необходимо указать директорию, в которой будут храниться SQL скрипты, а также текущую версию базы данных, которую и будет использовать проект. SQL скрипты именуются по следующему шаблону <Префикс><Версия>\_\_<Имя>.sql, где :

* <Префикс> – префикс, необходимый для определения файлов, используемых для создания миграций. По умолчанию равен «V», но может быть изменен.
* <Версия> – номер версии миграции. Мажорную и минорную версию можно разделить подчеркиванием. Отсчет версий начинается с 1.
* <Имя> – текстовое описание миграции. Отделено двойным нижним подчёркиванием от номера версии.

Таким образом, файл миграции может, называться, например, следующим образом – V1\_1\_0\_\_my\_first\_migration.sql. При запуске приложения Flyway сравнивает текущую версию базы данных (используя собственную таблицу, хранящую информацию о выполненных миграциях) с требуемой версией, которую мы указали в конфигурационном файле. Если версии не сходятся, то Flyway пытается выполнить миграционные файлы в порядке увеличения их версии, начиная с файла, версия которого следует за текущей версией схемы базы данных.

Рассмотрим миграции с Mongock. В отличии от Flyway, миграции базы данных прописываются не в отдельных скриптах на языке манипулирования данных, а непосредственно в коде программы. При старте приложения принцип работы тот же, что и у Flyway. В качестве примера рассмотрим следующую миграцию – <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/migration/FirstInitializeCollections.java>. Здесь создаются коллекции в базе данных Mongo, необходимые для работы приложения. Аннотация @ChangeUnit указывает на то, что данный класс является миграцией, а в качестве аргументов принимает идентификатор, номер в очереди и автора миграции. Аннотация @Execution помечает метод, который будет выполняться для осуществления миграции, а аннотация @RollbackExecution помечает метод, возвращающий базу данный в состояние, которое было до выполнения мигрирующего метода. Последний выполняется в случае ошибки при выполнении метода, помеченного @Execution.

Миграции являются очень сильным инструментом, необходимым для работы крупных приложений, но они не являются обязательными – компоненты фреймворка Spring способны автоматически приводить схему базы данных в необходимое состояние, если настроить определённые параметры в конфигурационном файле. Например, при использовании реляционной базы данных этот параметр – spring.jpa.hibernate.ddl-auto: update. Эта настройка будет обновлять схему базы данных чтобы она соответствовала структуре, описанной в слое данных. Данный механизм не используется в крупных продуктах так как определённая база данных может быть использована более чем одним сервисом и если один из них обновит схему базы данных, то это может негативно сказаться на работе другого сервиса.

# Глава 3. Сервисный слой

## 3.1 Механизмы авторизации и аутентификации

Сложно представить приложение, не нуждающееся в механизмах авторизации и аутентификации, ведь необходимо не только предоставлять разным пользователям разные полномочия, но и следить за безопасностью их данных. Наше приложение не является исключением, ведь система хранит личную информацию о пользователях, включая их адреса и контакты, а в перспективе и платёжные данные. Во многом надёжность разрабатываемого приложения зависит именно от этих механизмов.

Прежде чем говорить о реализации аутентификации и авторизации, опишем роли, которыми могут обладать пользователи и от которых зависят доступные им интерфейсы. Все роли делятся на публичные и административные. Первая группа ролей присваивается конечным пользователям, в то время как вторая необходима для поддержания работоспособности системы и её дальнейшего развития. К публичным ролям относятся:

* CUSTOMER – роль покупателя, присваиваемая автоматически всем зарегистрировавшимся пользователям. Этой роли доступны следующие действия:
  + Заполнение личных данных.
  + Выставление отметок «нравится».
  + Создании комментариев.
  + Создание отзывов.
  + Добавление продуктов в корзину.
  + Оформление заказов.
  + Чат с поддержкой.
* AUTHOR – роль для авторов скетчей. Добавляется к списку ролей у тех пользователей, которые пожелали стать авторами. Этой роли доступны следующие действия:
  + Публикация скетчей.
  + Настройки монетизации своих работ.
  + Статистика по своему аккаунту автора.
  + Поиск исполнителя своих заказов.
* EXECUTOR – роль исполнителя скетчей. Добавляется к списку ролей у тех пользователей, которые пожелали стать исполнителями. Этой роли доступны следующие действия:
  + Поиск свободного заказа.
  + Статистика по своему аккаунту исполнителя.

К административным ролям относятся:

* ADMIN – роль администратора. Фактически является ролью суперпользователя системы. Имеет полный доступ ко всем базам данных и их структурам.
* DEVELOPER – роль разработчика. Схожа с ролью ADMIN, за тем исключение, что имеет лишь ограниченный доступ к приватной информации о пользователях. Может изменять структуры баз данных.
* MODERATOR – роль модератора. Имеет ограниченный доступ к базам данных, содержащим приватную информацию о пользователях. Не может изменять структуру баз данных.
* SUPPORT – роль сотрудника поддержки системы. Обрабатывает пользовательские запросы через чат и прочие интерфейсы. Имеет ограниченный доступ к базам данных, содержащим приватную информацию о пользователях.

Поговорим про реализацию механизмов авторизации и аутентификации. Для начала необходимо понять, что такое JWT. JSON Web Token (JWT) представляет собой открытый стандарт (RFC 7519[3]) определяющий компактный и автономный способ безопасной передачи информации между сторонами в виде объекта JSON[7]. Эту информацию можно проверить и доверять ей, поскольку она имеет цифровую подпись. JWT могут быть подписаны с использованием секрета (с помощью алгоритма HMAC[4]) или пары открытого/закрытого ключа с использованием RSA[5] или ECDSA[6].

JWT состоит из трех частей:

* Заголовок (header) – содержит информацию о том, как вычисляется подпись. Представляет собой JSON объект.
* Полезные данные (payload) – так же, как и заголовок представляет собой JSON объект, хранящий пользовательскую информацию, так называемые заявки (claims).
* Подпись (signature) – зашифрованная строка, вычисляемая из заголовка и полезных данных с помощью определённого алгоритма шифрования и секрета.

Токен создаётся из этих трех частей посредством конкатенации через точку их base64[9] шифров. Важно понимать, что JWT не скрывает и не маскирует данные автоматически, а лишь предоставляет механизм цифровой подписи. То есть не стоит передавать важные данные (например, пароли) внутри JWT.

Наше приложение использует JWT при подтверждении аутентификации, так как мы хотим быть уверены в том, кто отправляет нам токен и можем ли мы предоставлять ему определённые полномочия. Для лучшего понимания опишем процесс регистрации пользователя и его дальнейшее взаимодействие с системой при помощи JWT.

Сразу стоит отметить что не все страницы эндпоинты[15] являются защищёнными – некоторые из них доступны без авторизации, например, просмотр скетчей или запрос на регистрацию.

При создании нового пользователя происходит по электронной почте и паролю. Также есть возможность предоставить дополнительные данные – имя, фамилия, описание, аватар и прочая информация. После этого на указанный адрес электронной почты отправляется письмо со ссылкой для подтверждения регистрации, пройдя по которой регистрация подтверждается и пользователю становятся доступны механизмы авторизации.

После подтверждения авторизации, пользователь должен автризоваться в системе – выполнить POST запрос на соответствующий эндпоинт и предоставить логин и пароль. Если данные верны, то пользователю возвращаются два токена. Один из них является JWT, хранит в своих полезных данных почтовый адрес пользователя и служит для авторизации при каждом запросе, а второй является UUID[11] и необходим для обновления первого токена, когда срок его годности заканчивается. Стоит отметить, что только второй токен сохраняется в базе данных вместе с его сроком доступа.

Когда у токена доступа истекает срок годности, его владельцу необходимо произвести обновление – с помощью POST запроса отправить токен обновления на соответствующий эндпоинт и получить новый токен доступа.

Наконец, когда пользователь желает обратиться к защищённому эндпоинту, то он вкладывает в заголовок «Authorization» HTTP запроса следующую строку: Bearer\_токен доступа (естественно, в эту строку вставляется первый токен). Если отправленный успешно валидирован, то запрос выполняется и пользователю возвращается соответствующий ответ.

Рассмотрим код реализации механизма авторизации и аутентификации. В пакете <https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/auth-service/src/main/java/com/workshop/authservice/model> хранятся классы, описывающие сущности пользователя, токена обновления и роли. Можно заметить, что класс пользователя (User) наследует реализовывает интерфейс UserDetails. Это необходимо для того, чтобы фреймворк понимал, какой класс будет являться представлением пользователя в системе.

Далее в пакете <https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/auth-service/src/main/java/com/workshop/authservice/security> находятся классы, отвечающие за настройку фильтров и обработку JWT. Важно понимать, как работает механизм авторизации Spring Security. Перед попаданием в контроллер, запрос проходит через двенадцать встроенных фильтров фреймворка, к которым также можно добавить собственные. Это и происходит в классах JwtTokenFilter и JwtFilterChainConfigurer. В первом из них создаётся фильтр, проверяющий наличие JWT в заголовке запроса, а также валидирующий его. При успешной валидации присутствующего токена в контекст приложения добавляется текущая авторизация, хранящая информацию о пользователе, отправившем запрос. В классе JwtFilterChainConfigurer фильтр добавляется в соответствующее место в списке фильтров. Класс [JwtTokenProvider](https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/security/JwtTokenProvider.java) отвечает за создание токенов (доступа и обновления) и их валидацию.

В классе <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/configuration/SecurityConfiguration.java> бин[13] SecurityFilterChain настраивает механизм обеспечения безопасности сервиса в целом. Здесь указываются, какие способы защиты должны работать и то, какие роли необходимы для доступа к каждому эндпоинту. Например, строка antMatchers("/docs/\*\*").hasAnyRole("DEVELOPER", "ADMIN") говорит о том, что доступ к эндпоинту с документацией доступен лишь администратору системы и разработчику, а строка csrf().disable() выключает механизм защиты от CSRF атак[10]. В этом классе можно заметить подключение OAuth2.0 с помощью средств, предоставляемых компанией Google. Работает это следующим образом: когда пользователь хочет авторизоваться с помощью своего аккаунта на стороннем сервисе, его переадресовывает на страницу авторизации этого сервиса, а затем, в случае успеха, данные о пользователе отправляются в нашу систему, где они обрабатываются в методе onAuthenticationSuccess класса <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/handler/OAuth2SuccessHandler.java>.

Наконец, если в каком-либо защищённом эндпоинте необходимо получить информацию о пользователе, то к аргументам метода нужно добавить интерфейс Authentication. Фреймворк автоматически добавит текущую пользовательскую аутентификацию в этот аргумент метода. Пример этого можно увидеть в методе me класса <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/controller/UserController.java>.

Стоит отметить, что код, не направленный на создание токенов дублируется в других микросервисах, требующих от пользователей авторизации. Это работает так как цифровую подпись можно проверять в любом сервисе системы, и это не требует доступа к базе данных с информацией о пользователях. Единственная разница заключается в том, что была создана собственная реализация интерфейса Authentication, лучше подходящая используемого механизма авторизации – <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/security/JwtAuthentication.java>.

## 3.2 Механизмы взаимодействия между сервисами

Ввиду того, что наше приложение является распределённой системой, возникает потребность взаимодействия между микросервисами. В современных приложениях эта задача может быть реализована принципиально различными способами – с помощью HTTP запросов между сервисами и с помощью брокеров сообщений. Каждый из этих подходов имеют как сильные, так и слабые стороны. Поговорим об этом подробнее:

* + - Поддержание очереди сообщение требует дополнительных ресурсов – любой брокер сообщений запускается как отдельных сервис и, соответственно, потребляющий ресурсы системы. В свою очередь HTTP сообщения встроены в каждый из уже работающих сервисов и не требуют дополнительных ресурсов.
    - Брокеры сообщений создают единую точку отказа взаимодействия между сервисами – так как брокер является отдельными сервисом, с которым связываются отправители и получатели сообщений, то его отказ может повлечь за собой нарушение целостности системы. С другой стороны, HTTP сообщения не нуждаются в поддержании стороннего сервиса, обрабатывающего сообщения.
    - Обмен сообщений на основании HTTP запросов не поддерживает журналирование – при отказе сервиса-получателя информация, передаваемая в сообщение будет потеряна, так как получатель её не обработал до конца (или даже не начал этого делать). Брокеры сообщений поддерживают журналирование. Это означает, что сообщение в конечном итоге будет доставлено до получателя, так как запись о нем хранится в журнале и удаляется из него только после того, как получатель обработает сообщение.
    - Брокеры сообщений асинхронны в отличии от HTTP запросов – при использовании HTTP запросов работа потока приостанавливается в виду необходимости получить ответ от запрашиваемого сервиса. Взаимодействие с очередью сообщений сводится к тому, что отправитель добавляет новую запись в определённую очередь, а получатель просматривает очереди и при наличии сообщений и обрабатывает их. Таким образом использование брокеров сообщений позволяет избавится от блокирующих операций, чего нельзя сказать про HTTP запросы. Особенно хорошо можно заметить разницу в случае появления цепочки запросов между сервисами, когда каждое звено обработки ожидает ответа от следующего. Конечный пользователь при этом длительное время ожидает ответа от системы в целом, что недопустимо в современных системах. А если в одном из звеньев цепи произошла ошибка, то пользователь после длительного ожидания ещё и не получает ответ.
    - Брокер сообщений позволяет отправлять сообщения многим сервисам одновременно, в то время как HTTP запросы придётся отправлять каждому получателю одновременно. Для отправки сообщения набору получателей необходимо лишь добавить записи в несколько очередей, что является довольно легковесной задачей с точки зрения потребления ресурсов.

В нашем приложении было решено использовать брокер сообщений ввиду того, что конечный пользователь должен получать ответ как можно быстрее, то есть цепочка запросов при взаимодействии между сервисами не должны быть блокирующей. В качестве брокера рассматривались две технологии – Apache Kafka и RabbitMQ, но был выбран последний, так как:

* нет необходимости использовать бесконечные потоки ввиду непостоянства потока сообщений;
* транзакционные данные в сообщениях;
* нет необходимости в длительном хранении сообщений в журнале;
* модель умный брокер/тупой консьюмер позволяет отслеживать исполнение сервисом задачи при получении сообщения;
* топология очереди способна эффективно обрабатывать поток сообщений системы.

Для лучшего понимания рассмотрим обмен сообщениями между сервисом авторизации и сервисом фоновых задач. Как было описано в предыдущей главе, при создании нового пользовательского аккаунта, на указанную электронную почту отправляется письмо информацией для подтверждения регистрации. Как это происходит? После успешной регистрации, сервис авторизации добавляет в определённую очередь сообщение, в котором хранится информация с адресом электронной почты пользователя. Далее сервис фоновых задач, видя новое сообщение в очереди обрабатывает его – генерирует UUID-токен, привязывающийся к записи о новом пользователе в базе данных, и отправляет на предоставленную электронную почту письмо со ссылкой, хранящей внутри себя созданный токен. Пройдя по ссылке, пользователь отправляет GET запрос, содержащий токен в качестве параметра пути и тем самым, обращается к контроллеру сервиса фоновых задач. Контроллер валидирует токен, получает идентификатор пользователя, ассоциирующегося с этим токеном и обновляет запись об полученном пользователе в базе данных, изменяя поле, отвечающее за статус пользователя (переводит из состояния «инициализирован» в статус «активен»).

Приведём примеры кода. В классе https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/service/messaging/AuthenticationConfirmationRabbitMQService.java метод sendConfirmation добавляет через эксчейндж новое сообщение в очередь, отвечающую за запросы на создание подтверждений новых пользователей. Этот метод вызывается в контроллере https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/controller/UserController.java в методе register, отвеающем за эндпоинт регистрации. Далее в методе addUserForConfirmation класса https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/background-service/src/main/java/com/workshop/backgroundservice/handler/AuthenticationRabbitMQMessageReceiver.java читается сообщение из вышеупомянутой очереди и создаётся запись в БД с UUID токеном пользователя. Затем, когда пользователь переходит по ссылке из электронного письма, в методе confirm контроллера https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/background-service/src/main/java/com/workshop/backgroundservice/controller/AuthenticationConfirmationController.java происходит подтверждение регистрации.

Помимо подтверждения авторизации, обмен сообщениями через брокер нужен и в других процессах системы, например, при совершении оплаты.

## 3.3 Механизмы основной бизнес логики

Механизмами основной бизнес логики являются функции, связывающие слой контроллеров со слоем данных. Это подразумевает собой CRUD операции с сущностями, входящими в различные контексты системы. Принцип работы этого слоя не меняется от микросервиса к микросервису, так как слой данных предоставляет достаточный уровень абстракции для того, чтобы не задумываться об используемых источниках данных.

Рассмотрим сервис, созданный для обработки скетычей – <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/service/SketchService.java>. В нем присутствуют различные методы для CRUD операций, функционал которыхлегко можно определить по названию самого метода. Помимо этого, стоит обратить внимание на сопутствующие настройки. Аннотация @Service, включающая в себя аннотацию @Component, говорит о том, что внутрь класса можно внедрять другие компоненты системы, коими в данном случае являются два репозитория – SketchRepository и TagRepository. Они необходимы для взаимодействия с базами данных, в которых хранятся соответствующие сущности. Аннотация @Autowired указывает на то, что аргументами метода являются бины, получаемые из контекста[14]. То, какой бин внедрить определяется типом данных аргумента. Если в контексте приложения существует несколько бинов одного и того же класса, то по умолчанию возникнет ошибка компиляции. Это можно исправить, используя аннотацию @Qualifier, в параметрах которой указывается имя бина, который необходимо внедрить. В данном классе можно ещё заметить аннотацию @Value. Ей помечается переменная, значение которой достаётся из переменной окружения. В нашем случае значение получается из файла application.yml – <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/resources/application.yml>. По данному файлу строятся и другие переменные окружения, которые фреймворк Spring использует для настройки собственных компонентов. Например, настройка server.port указывает какой порт будет слушать наше приложение, то есть куда следует отправлять HTTP запросы для взаимодействия с ним.

# Глава 4. Слой контроллеров

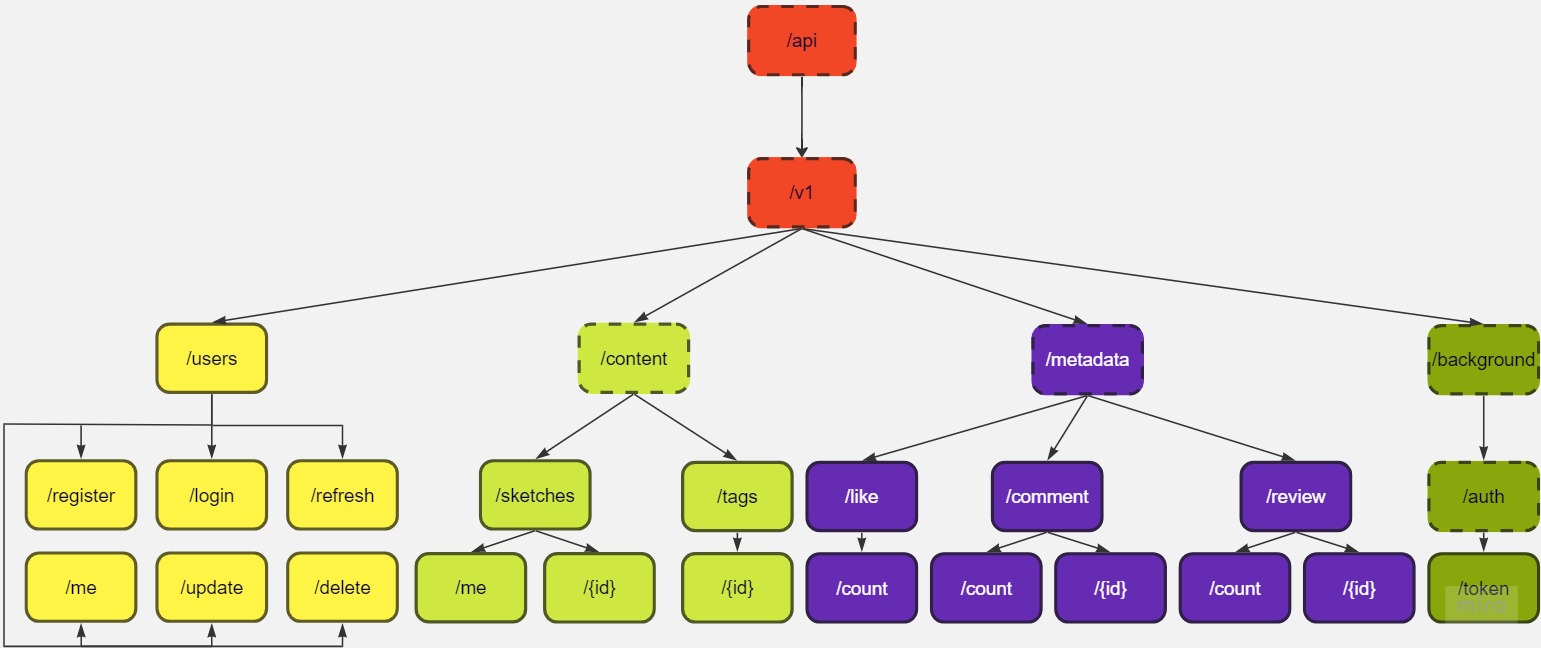
 Слой контроллеров приложения представляет собой интерфейс для взаимодействия пользователя с системой. Этот интерфейс представляет собой набор эндпоинтов, пути к каждому из которых в совокупности образуют дерево. На Рисунке 5 изображено это дерево путей.

Рисунок 5. Дерево путей.

Пунктирными границами обозначены объекты, к которым нельзя обратиться, то есть на обработку которых не создано эндпоинта ввиду особенностей архитектуры приложений или отсутствия необходимости. Стоит отметить, что по одному и тому пути можно могут находиться несколько эндпоинтов, доступ к которому определяется типом HTTP запроса – GET, POST, PUT или DELETE.

Рассмотрим пример конкретного контроллера – <https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/controller/UserController.java>. Аннотация @RestController помечает класс как контроллер. Так же аннотация включает в себя аннотацию @Component, которая помечает класс как бин. Аннотация @RequestMapping указывает корневой путь до контроллера. Важно понимать, что для каждого сервиса может быть указана переменная окружения server.servlet.context-path, значение которой добавляется в начало каждого корневого пути, заданного аннотаций @RequestMapping. Аннотации @GetMapping, @PostMapping, @PutMapping и @DeleteMapping определяют методы, являющиеся эндпоинтами, задают тип запроса, обрабатываемые ими и уточняют путь относительно корневого пути. Аннотация @RequestBody указывает, что данный аргумент эндпоинта является телом запроса, а аннотация @Valid сообщает фреймворку о необходимости валидации аргумента. Так же существуют аннотации @RequestParam и @PathVariable, отмечающие аргументы как переменные запроса и переменные пути, соответственно. Дополнительно к инструментам фреймворка Spring, в данном классе можно заметить аннотации @Tag и @Operation – они используются генерации документации, сервер которой запускается вместе с сервером приложения. Эти аннотации помечают контроллер и его методы соответственно и принимают параметры, обозначающие название и описание. Доступ к интерактивному интерфейсу документации находится по адресу, прописанному в параметре springdoc.swagger-ui.path и в нашем случае равному /docs/swagger.html. Пользовательский интерфейс документации не только содержит информацию об эндпоинтах (пути до них, тип принимаемых данных, тип возвращаемых данных и статусы ответов), но и позволяет выполнять запросы с произвольными данными к этим эндпоинтам. Отметим, что запросы, выполненные через интерфейс документации отправляются на работающее приложение, как если бы эти запросы выполнил бы конечный пользователь системы. Именно поэтому доступ к документации есть лишь у двух ролей – администратора и разработчика.

# Глава 5. Развертывание

Как было сказано в предыдущих главах, наше приложение является распределённой системой, то есть состоит из нескольких взаимосвязанных компонентов, работающих вместе ради достижения общей цели. Для управления и поддержания работоспособности всей системы, мы будем использовать такие инструменты как Docker[24] и Kubernetes[25]. Поговорим о них подробнее.

Контейнеры Docker – единые модули, упаковывающие само приложение и все его зависимости. Использование контейнеров Docker обеспечивает множество преимуществ при работе с микросервисами:

* Согласованность – контейнеры Docker обеспечивают согласованную среду для микросервисов, упрощая их развертывание и управление ими в кластере Kubernetes.
* Мобильность – контейнеры Docker спроектированы так, чтобы быть переносимыми, что означает, что их можно легко перемещать из одной среды в другую. Это упрощает разработку и тестирование микросервисов на локальном компьютере разработчика, а затем развертывание их в рабочем кластере Kubernetes.
* Изоляция. Контейнеры Docker обеспечивают изоляцию процессов, что помогает предотвратить конфликты между различными микросервисами, работающими в одном кластере. Это гарантирует, что каждый микросервис работает в своей собственной изолированной среде и не мешает другим.
* Масштабируемость. Контейнеры Docker легкие, поэтому их легко увеличивать или уменьшать по мере необходимости. Это особенно полезно в кластере Kubernetes, где микросервисы могут автоматически масштабироваться в зависимости от спроса.
* Управление версиями. Контейнеры Docker упрощают управление и управление версиями микросервисов, позволяя разработчикам при необходимости легко вернуться к предыдущей версии.

В целом, использование контейнеров Docker для микросервисов упрощает разработку, развертывание и управление системой, а также обеспечивает большую гибкость и масштабируемость приложений.

Управление контейнеризованными компонентами системы и их масштабирование является сложной задачей, особенно по мере роста их количества. Для решения этой проблемы используется система оркестрации контейнеризованными приложениями Kubernetes, к достоинствам которой относятся:

1. Масштабируемость – Kubernetes предоставляет гибкую и масштабируемую платформу для развертывания контейнеров и управления ими, которую можно использовать для увеличения или уменьшения количества узлов и компонентов в распределенной системе в зависимости от потребности.
2. Отказоустойчивость – Kubernetes предоставляет такие функции, как автоматический перезапуск контейнеров, последовательные обновления и возможности самовосстановления, обеспечивающие высокую доступность и отказоустойчивость системы.
3. Обнаружение служб и балансировка нагрузки – Kubernetes предоставляет встроенные возможности обнаружения служб и балансировки нагрузки, которые позволяют компонентам распределенной системы беспрепятственно взаимодействовать друг с другом.
4. Мобильность – Kubernetes предоставляет согласованную платформу для развертывания контейнеров и управления ими в различных средах, включая локальные центры обработки данных, общедоступные облака и гибридные облака, что упрощает перемещение распределенных систем и управление ими в разных средах.
5. Автоматизация: Kubernetes предоставляет декларативный API, который позволяет пользователям определять желаемое состояние своей распределенной системы, а Kubernetes автоматически управляет системой, чтобы гарантировать, что она всегда находится в желаемом состоянии.
6. Изоляция ресурсов – Kubernetes позволяет пользователям определять лимиты ресурсов и запросов для каждого контейнера, гарантируя, что контейнеры не превышают выделенные им ресурсы и не мешают другим компонентам в системе. Такая изоляция ресурсов повышает безопасность и стабильность распределенной системы.

Kubernetes обеспечивает иерархическую структуру для организации и управления контейнерными рабочими нагрузками, которая состоит из следующих компонентов:

* Кластер – набор физических или виртуальных машин, также называемых узлами, на которых выполняются контейнеризованные приложения. Кластер включает в себя плоскость управления, которая управляет состоянием кластера, и сервер API Kubernetes, предоставляющий доступ к API Kubernetes.
* Узел или, по-другому, node – физическая или виртуальная машина, на которой выполняются контейнеризованные приложения. На каждом узле в кластере Kubernetes работает среда выполнения контейнера, в нашем случае Docker, и kubelet, который является основным агентом узла, взаимодействующим с плоскостью управления Kubernetes.
* Pod, или, по-другому, под – самая маленькая и простая единица в объектной модели Kubernetes. Это логический хост для одного или нескольких контейнеров, развернутый на узле. Все контейнеры в модуле используют одно и то же сетевое пространство имен, поэтому они могут взаимодействовать друг с другом с помощью локального хоста.
* Deployment – объект Kubernetes, который управляет набором идентичных модулей, гарантируя, что желаемое количество реплик всегда работает. Deployment также предоставляет последовательные обновления и возможности отката для контейнеризованных приложений.
* Службы или, по-другому, service — это объект Kubernetes, определяющий логический набор модулей и политику доступа к ним. Службы обеспечивают балансировку нагрузки и обнаружение служб для контейнерных приложений, работающих в кластере Kubernetes.
* Namespace – виртуальный кластер внутри кластера Kubernetes. Он позволяет разделить кластер на несколько виртуальных кластеров, каждый со своими собственными ресурсами и средствами управления доступом.
* ConfigMap – используется для хранения данных конфигурации, которые могут использоваться контейнерами, работающими в поде. Он может хранить пары ключ-значение, файлы конфигурации или даже аргументы командной строки.

Нельзя не упомянуть про одну из самых мощных функций Kubernetes – способность автоматически масштабировать ресурсы в зависимости от спроса. Это помогает гарантировать, что у приложения всегда есть ресурсы, необходимые для обработки всплесков трафика и поддержания оптимальной производительности.

Kubernetes поддерживает два типа автоматического масштабирования:

* Горизонтальное автомасштабирование пода – масштабирует количество реплик пода на основе использования ЦП или пользовательских показателей. Он автоматически увеличивает или уменьшает количество реплик в соответствии с желаемым уровнем использования, что помогает гарантировать приложению наличие достаточного количества ресурсов для обработки всплесков трафика.
* Вертикальное автомасштабирование пода – регулирует запросы ЦП и памяти, а также ограничения контейнеров в поде в зависимости от их фактического использования. Он анализирует историю каждого контейнера и соответствующим образом корректирует ограничения ресурсов, что помогает предотвратить нехватку ресурсов и сбой контейнеров.

Kubernetes также поддерживает автоматическое масштабирование кластера, которое автоматически добавляет или удаляет узлы из кластера в зависимости от потребности. Это помогает гарантировать, что кластер всегда имеет достаточно ресурсов для обработки рабочей нагрузки, минимизируя затраты за счет масштабирования, когда ресурсы не нужны.

Для разворачивания приложения в кластере необходимо иметь как таковой кластер. Это может быть как облачный кластер (например GCP[22] или AWS[23]), так и локальный кластер minikube[21]. После подключения к такому кластеру можно работать с коммандной строкой Kubernetes.

Существует два способа развёртывания компонент кластера – через коммандную строку и через выполнение файлов формата .yaml. В нашем приложении используется именно второй способ ввиду его удобства и гибкости.

Перед разворачиванием непосредственно приложения нужно запустить базы данных и брокер сообщений. Скрипты для развёртывания RabbitMQ находятся в https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/rabbitmq. Там же присутствуют сценарии командной строки (.sh файлы), позволяющие быстро запустить или остановить соответствующие компоненты, не выполняя скрипты разворачивания по отдельности. В файле rabbitmq-storage.yaml создаются PersistentVolume и PersistentVolumeClaim, которые отвечают за хранение данных кластера RabbitMQ, а в файле rabbitmq-cluster.yaml находится скрипт для запуска самого брокера. Этот скрипт использует API, предоставленное разработчиками RabbitMQ.

В директории <https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/database> находятся скрипты для разворачивания всех СУБД, необходимых системе. Рассмотрим их на примере базы данных Postgres – <https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/database/postgres>. В файле postgres-storage.yaml создаются PersistentVolume и PersistentVolumeClaim, отвечающие за хранение информации в базе. В файле postgres-service.yaml создаётся сервис, который создаёт точку доступа для присоединения к СУБД извне контейнера, в котором она запущена. В файле postgres-deployment.yaml находится скрипт, создающий Deployment, необходимый для развёртывания СУБД. Важно упомянуть файл postgres-config.yaml, не хранящийся в репозитории, так как он содержит конфиденциальную информацию о системе. Скрипт внутри этого файла создаёт ConfigMap, которая задаёт настройки развёртываемой базы данных – её имя, а также логин и пароль для подключения. В данной директориии так же хранятся сценарии командной строки, запускающие вышеперечисленные скрипты.

Далее рассмотрим развёртывание разрабатываемого приложения на примере сервиса авторизации – <https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/application/auth>. Как и при запуске СУБД, здесь присутстует скрипт, создающий ConfigMap с переменнными окружения – эти значения перезаписывают соответствуюзие значения из файла application.yml. Этот механизм удобен для настройки приложения без изменения исходного кода. Действительно, если мы захотим, например, поменять секретный ключ для шифрования и дешифрования JWT, то нам достаточно изменить лишь файл, создающий ConfigMap, а не менять application.yml и пересоздавать Docker-образ приложения чтобы изменения вступили в силу. В фале application-auth-storage.yaml создаются PersistentVolume и PersistentVolumeClaim, а в application-auth-deployment.yaml – Deployment. В отличии от развёртывания СУБД, здесь при создании сервиса используется балансировщик нагрузки в качестве точки доступа к микросервису – это необходимо ввиду создания нескольких реплик данного компонента системы. Именно балансировщик нагрузки будет распределять входящие запросы между множеством реплик.

# ЗАКЛЮЧЕНИЕ

В заключение хотелось бы описать то, как развернуть приложение на localhost или в локальном кластере Kubernetes.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Репозиторий на сайте GitHub с кодом проекта ­­­– https://github.com/DaniilDDDDD/disign-workshop-backend
2. Эрик Эванс. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем.
3. RFC 7519 – https://www.rfc-editor.org/rfc/rfc7519
4. RFS 2104 – https://www.rfc-editor.org/rfc/rfc2104
5. RSA – https://en.wikipedia.org/wiki/RSA\_(cryptosystem)
6. ECDSA – https://en.wikipedia.org/wiki/Elliptic\_Curve\_Digital\_Signature\_Algorithm
7. JavaScript Object Notation (JSON) – https://www.rfc-editor.org/rfc/rfc4627
8. JWT – https://jwt.io/introduction
9. Base64 – https://www.rfc-editor.org/rfc/rfc4648
10. CSRF – https://en.wikipedia.org/wiki/Cross-site\_request\_forgery
11. UUID – https://en.wikipedia.org/wiki/Universally\_unique\_identifier
12. Дженерики в Java – https://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf
13. Spring bean – https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-definition
14. Spring container – https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-basics
15. Endpoint – https://en.wikipedia.org/wiki/Web\_API
16. XML конфигурация Spring – https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/xsd-configuration.html
17. Java конфигурация Spring – https://docs.spring.io/spring-framework/docs/3.0.0.M4/reference/html/ch03s11.html
18. Java аннотации – https://en.wikipedia.org/wiki/Java\_annotation
19. Spring аннотации – https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-java
20. Spring контекст – https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-java-instantiating-container
21. Minikube – https://minikube.sigs.k8s.io/docs/
22. Google Cloud Platform – https://console.cloud.google.com/marketplace/product/google/container.googleapis.com?returnUrl=%2Fkubernetes%3Fproject%3Dworkshop-367510&project=workshop-367510
23. Amazon Web Services – https://aws.amazon.com/ru/eks/?did=ap\_card&trk=ap\_card
24. Docker – https://www.docker.com/
25. Kubernetes – https://kubernetes.io/

1. Денормализация - попытка увеличить скорость чтения данных за счёт уменьшения скорости записи. Избыточные копии данных записываются в несколько таблиц для избежания сложных операций соединения данных. [↑](#footnote-ref-1)
2. Кэширование – метод разработки инфраструктуры системы, который улучшает время загрузки страницы и может уменьшить нагрузку на сервисы и базы данных. При таком подходе, диспетчер кэширования вначале проверяет, делался ли запрос ранее, чтобы найти ответ, который уже на него возвращался, сократив при этом время выполнения текущего запроса. Если такого запроса ранее не производилось, то полученный ответ сохраняется в кэш. [↑](#footnote-ref-2)
3. Create, Read, Update, Delete (CRUD) – создание, чтение, обновление, удаление. [↑](#footnote-ref-3)
4. Enterprise Resource Planning (ERP) - это программное обеспечение, помогающее предприятиям автоматизировать основные бизнес-процессы и управлять ими для достижения оптимальной производительности. [↑](#footnote-ref-4)