Санкт-Петербургский государственный университет

Панюшин Даниил Васильевич

Выпускная квалификационная работа

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

Уровень образования: бакалавриат

Направление 02.03.02 «Фундаментальная информатика и информационные технологии»

Основная образовательная программа СВ.5003.2019 «Программирование и информационные технологии»

Научный руководитель: кандидат физ.-мат. наук, доцент Раевская А. П.

Рецензент: Java-разработчик, ПАО «Газпром», Крешков И. П.

Санкт-Петербург

2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

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. Платформа позволит талантливым авторам выкладывать свои работы под собственным именем и зарабатывать этим.

Данная платформа позволит неизвестным дизайнерам создать собственный бренд и предоставит простым пользователям возможность приобретать вещи, недоступные в прочих магазинах.

Пользователь регистрируется как покупатель или дизайнер:

- 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 создаёт продукт и отправляет его покупателю.

Целью выпускной квалификационной работы является реализация вышеописанного проекта, а именно:

- Описание разработанной архитектуры;
- Описание использованных технологий и необходимости их применения;
- Описание реализации проекта.

Для решения поставленных задач были изучены различные источники информации, такие как официальные документации, технические спецификации, книги и статьи опытных специалистов в данной сфере. Также были изучены материалы из различных конференций, посвященных используемым технологиям.

В данной выпускной квалификационной работе указана ссылка на основную страницу репозитория[1] с кодом проекта в списке источников. Примеры кода приводятся в виде ссылок так же на список источников.

Глава 1. Использованные технологии

Технологии, используемые для создания системы:

Язык программирования – Java[2].

Причинами такого решения являются надёжность и безопасность, предоставляемая языком Java.

Система управления базами данных – PostgreSQL[3], MongoDB[4] и Redis[5].

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. Такой выбор сделан ввиду особенностей данных, обрабатываемых приложением:

- Слабая связность;
- Изменчивость структуры по мере развития проекта (например, добавляются новые поля);
- Необходимость в высокой скорости 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[6]).

Именно MongoDB выбрана, так как наиболее популярна и обладает расширенным функционалом по сравнению с аналогами.

Третья СУБД – in-memory[7] база данных. Она необходима для работы системы кэширования – процесса временного сохранения данных в

специальном хранилище (кэше) для ускорения доступа к этим данным в будущем. В данном случае кэшем является СУБД Redis.

Фреймворки[1] – Spring[8], Swagger[9].

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[10].

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[11], PgAdmin 4[12], MongoDBCompass[13], Postman[14].

InteliJ IDEA – самая популярная среда разработки на Java.

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

[1] Фрейвморк – готовый набор инструментов, который помогает разработчику быстро создать продукт.

MongoDBCompass – инструмент для управления базами данных на движке Mongo.

Postman – инструмент для создания HTTP [15] запросов.

Средства развёртывания – Docker[16], Kubernetes[17].

Docker – система контейнеризации приложений, позволяющая быстро разворачивать приложения на любых платформах.

Kubernetes – система оркестрирования контейнеризованных приложений, необходимая для настройки взаимодействия между компонентами микросервисами системы.

Глава 2. Архитектура

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[18] в системах оркестрации[19], 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[20]. 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.

Для авторизации и аутентификации в приложении используются JWT[21].

Система будет запущена в кластере Kubernetes с использованием контейнеров Docker. Данная 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[22]) для ускорения получения пользователями контента и серверы доменных имён (DNS[23]) для балансировки нагрузки и определения географически ближайшего сервера приложения.

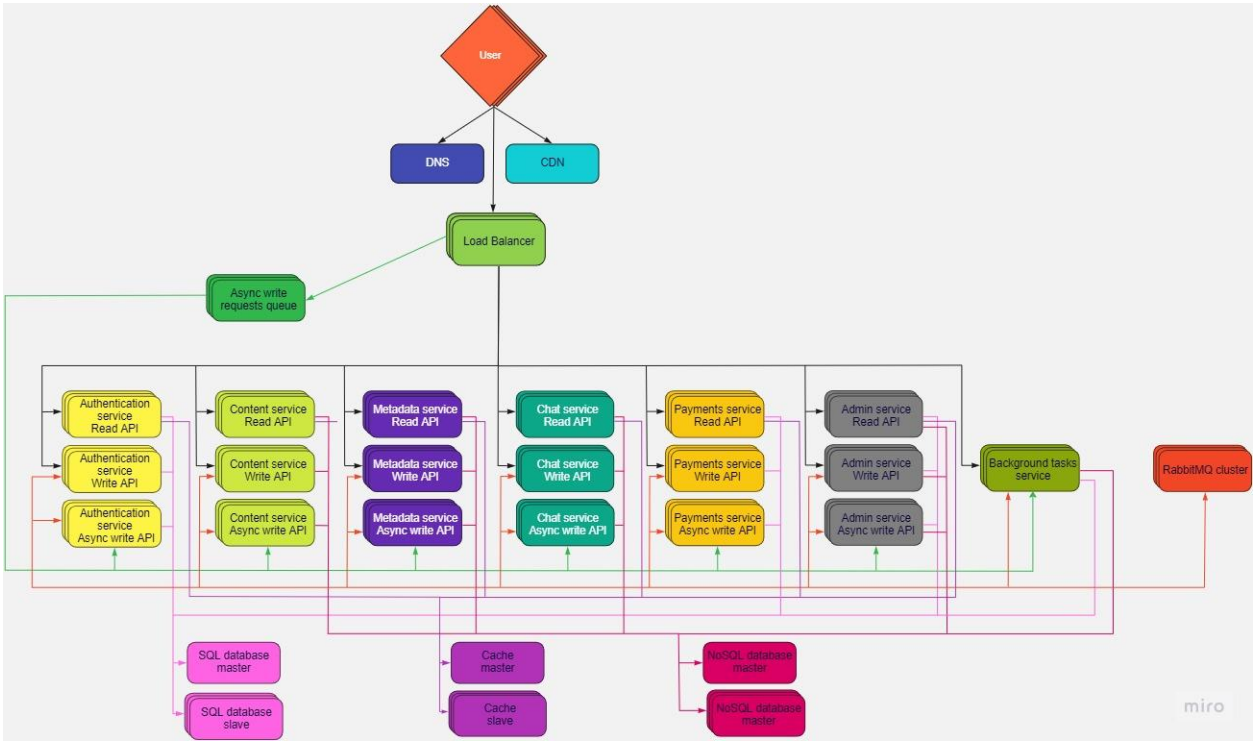

Рисунок 1. Архитектура системы для высокой нагрузки.

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 сервиса количество пользователей будет значительно меньше. Поэтому в дальнейшей работе будет описана более простая версия приложения. Ввиду локального развертывания в данной версии отключены механизмы балансировки нагрузки через DNS, кэширования контента в CDN, репликации баз данных, а также, из-за отсутствия крайне высокой нагрузки, и механизм асинхронной обработки запросов.

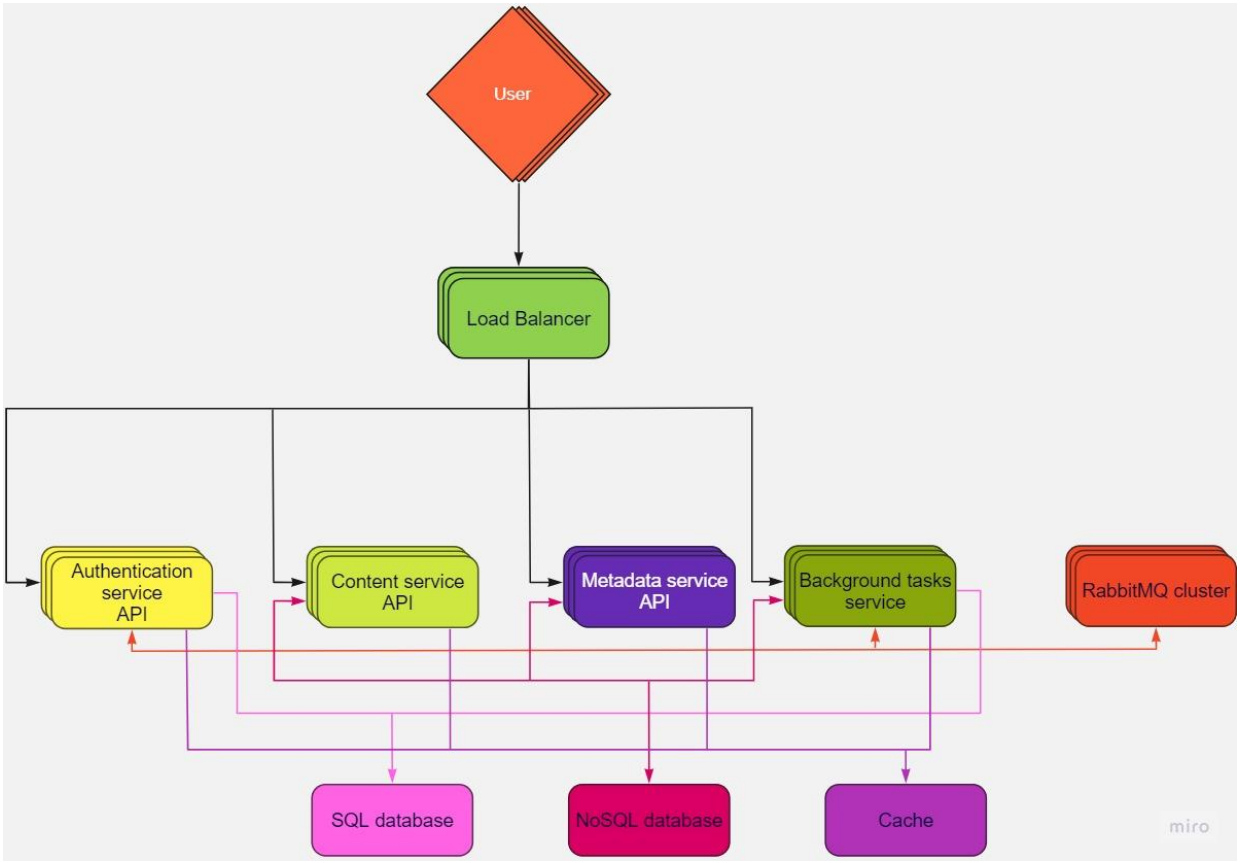

Рисунок 2. Архитектура приложения для малой нагрузки

Глава 3. 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[2] операции с пользовательской информаций;
- CRUD операции с основным контентом приложения, которым являются скетчи, тэги и сопутствующие медиафайлы (фотографии, видео и прочее);
- CRUD операции с метаинформацией (оценки, комментарии, отзывы);
- операции поддержания целостности системы (отчистка от неактуальной информации и прочее);
- операции уведомлений (например, письма для подтверждения авторизации).

Поведенческая диаграмма перечисленных выше процессов изображена на Рисунке 3.

[2] Create, Read, Update, Delete (CRUD) – создание, чтение, обновление, удаление.

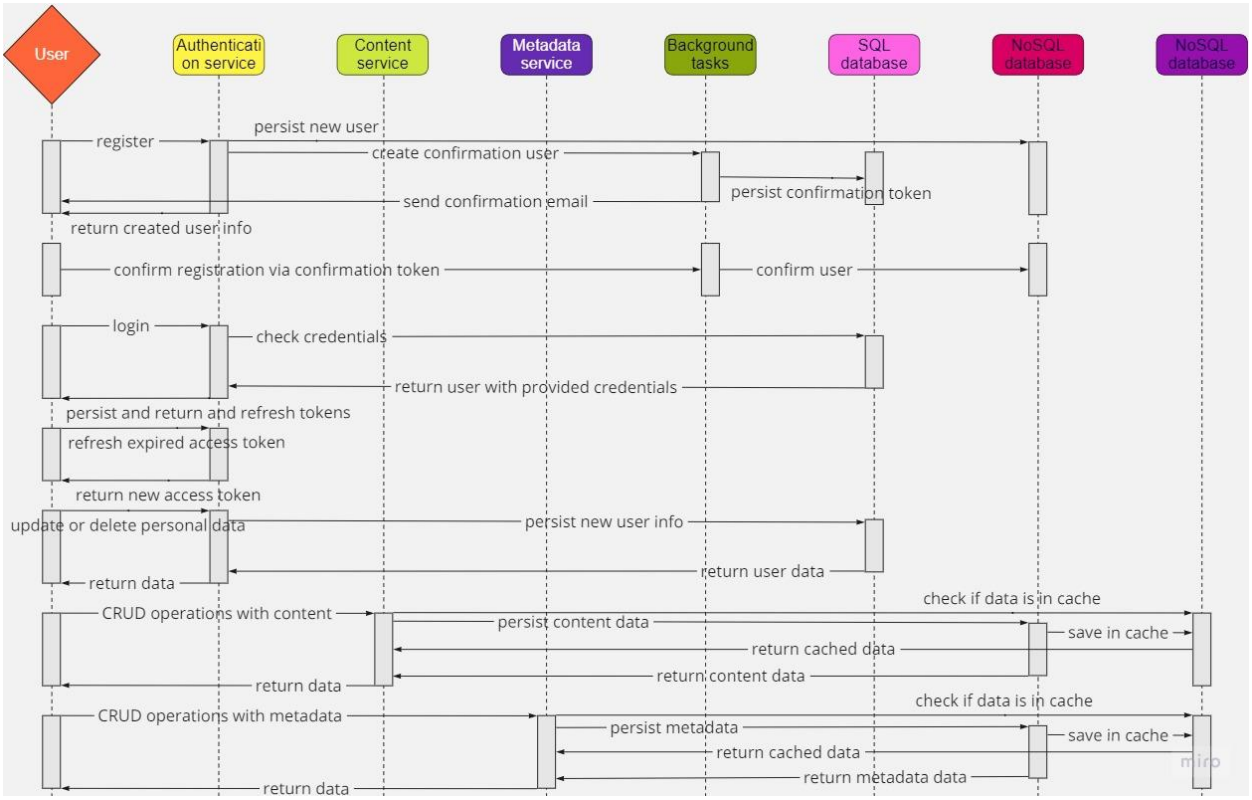

Рисунок 3. Поведенческая диаграмма приложения.

При определении способа разбиения приложения на микросервисы была создана структурная диаграмма, не только описывающая схему базы данных, но и отображающая ограниченные контексты приложения – изображена на Рисунке 4. По мере развития приложения схема базы данных будет меняться, но ограниченные контексты менее сильно подвержены изменениям (естественно, до момента проведения углубляющего рефакторинга[24, стр. 287]).

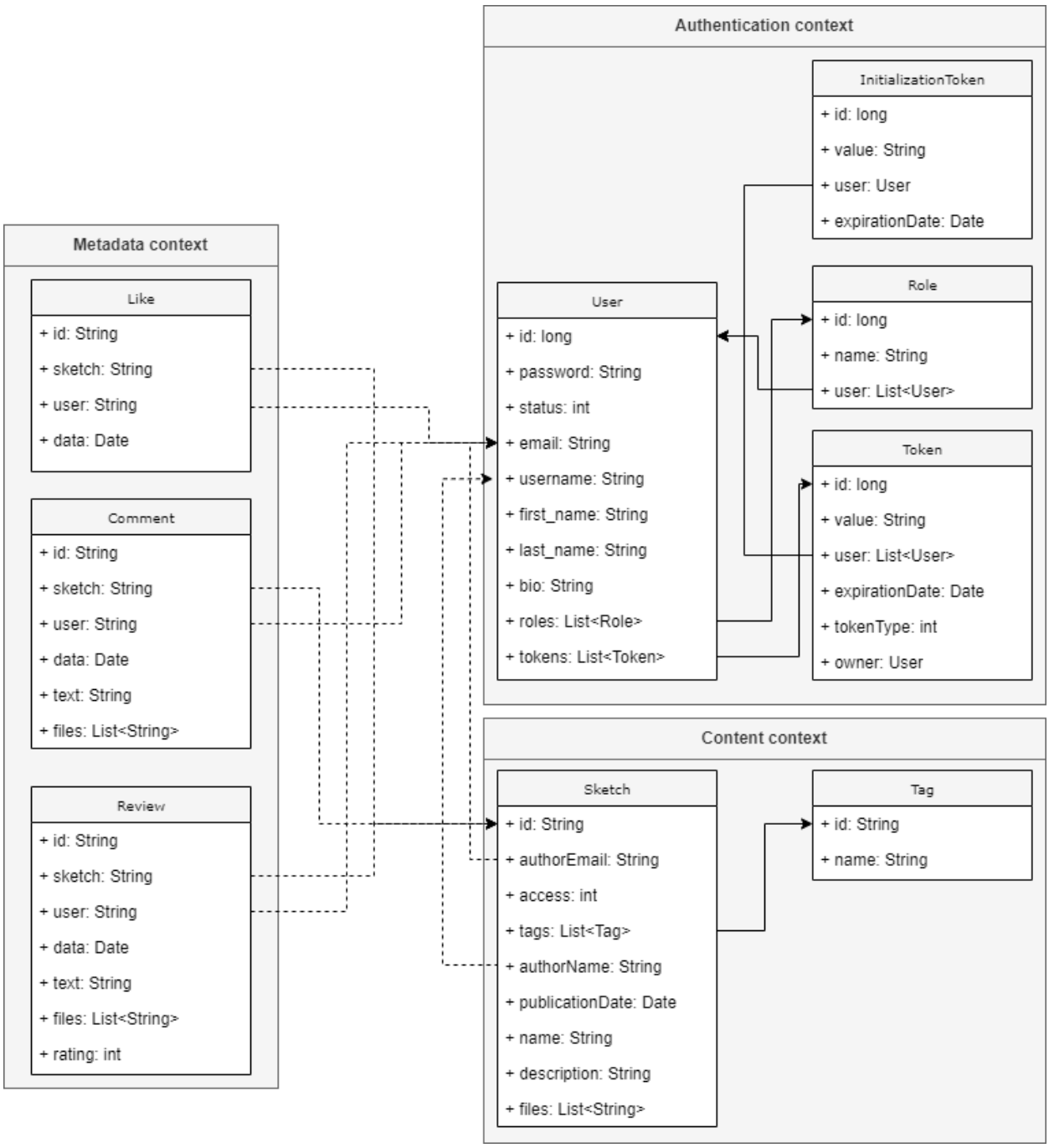

Рисунок 4. Структурная диаграмма приложения.

Глава 4. Работа с фреймворком Spring и сборка проекта

При написании Java-приложения, использующего фреймворк, необходимо понимать принципы его работы. Фактически, 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, в сравнении с обычным Java-приложением, работая со Spring, код пишется иначе. Создавая объекты, зависящие от других объектов, необходимо давать понять фреймворку о существовании зависимости. В ранних версиях Spring это происходило на основании специальных xml файлов[25], хранящих информацию о взаимосвязях между классами. Затем, в версии 3.1 разработчики фреймворка добавили возможность конфигурации приложения внутри самого Java кода[26]. Для этого используются специальные аннотации[27][28] и классы[29].

Помимо создания объектов, Spring контейнер берёт на себя ответственность за инициализацию этих объектов, их конфигурацию и уничтожение. 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.

Для сборки каждого компонента, написанного с помощью данного фреймворка, в исполняемые файлы используется Gradle – система автоматизации сборки приложенний, написанных на Java и других языках, использующих JVM[30].

При работе с системами сборки в каждом модуле проекта поддерживается скрипт, содержащий информацию о том, в какой последовательности производить сборку, какие сторонние зависимости скачивать и упаковывать, какие тесты запускать и прочее. Также Gradle позволяет настраивать автоматическое развёртывание приложений на удалённом сервере.

Рассмотрим пример – скрипт build.gradle[31] для микросервиса авторизации. В данном файле в секции plugin содержится список плагинов, используемых для сборки. В секции repositories указывают репозитории, из которых будут загружаться зависимости. Секция test определяет зависимости и настройки для запуска тестов. Наконец, в разделе depedencies перечислены зависимости микросервиса от сторонних библиотек. Также в каждой зависимости указывается когда их применять[32], например, строка compileOnly говорит о том, что код зависимости будет доступен только на стадии компиляции.

Глава 5. Слой данных

5.1 Реляционные СУБД

В приложении слой данных, взаимодействующий с SQL[33] СУБД, реализован с помощью технологии Spring Data JPA[34]. 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.

Внутри слоя данных существуют классы, отображающие 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 приложения можно обратиться к коду, описывающему сущность пользователя системы[35].

Класс User отображает таблицу пользователей. Он аннотирован @Entity – это означает, что он является сущностью JPA, которая отображает таблицу базы данных. Аннотация @Table определяет в базе данных имя таблицы, 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 связь многие 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 реляционных СУДБ.

Рассмотрим код, описывающий программный интерфейс для выполнения CRUD операции с данными таблицы пользователей[36].

Первое что можно заметить – интерфейс наследуется от встроенного в Spring Data JPA интерфейса. Это необходимо для автоматической генерации тел методов, который используются для CRUD операций. При наследовании передаются два параметра с помощью даймонд-синтксиса[37] – 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 определении метода могут использоваться определённые шаблоны, такие как get, find, getAll, findAll. Они определяют будут ли искаться все объекты, подходящие под описание, 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 переменные того же типа, что и поле сущности, по которому происходит поиск.

Так же можно осуществлять поиск не по полному совпадению, а по 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 (например, для in этим полем является

коллекция элементов, на вхождение в которую осуществляется проверка). Примером такого метода является getAllByRolesIn.

5.2 Нереляционные СУБД

В приложении помимо реляционной СУБД используется ещё и документоориентированная NoSQL база данных MongoDB, хранящая информацию в виде документов BSON[38], 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 документов.

Рассмотрим пример сущности в MongoDB[39].

Аннотация @Document указывает, что данный класс представляет собой документ в некоторой коллекции (имя которой указывается в аргументе аннотации). Поле id должно быть помечено аннотацией @Id, чтобы быть использованным как уникальный идентификатор документа в MongoDB. Аннотация @Indexed создаёт индекс в базе данных[40]. 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[41].

В приложении для взаимодействия с данными используются не только встроенные инструменты фреймворка, но и их имплементации[42]. 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[43]. Внутри этих методов используется CriteriaAPI, позволяющий 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[44] является возможность создания агрегирующих запросов, как это можно увидеть в методе countSketchesLikes классе LikeRepositoryCustom[45], где происходит подсчёт количества отметок «нравится» для каждой сущности эскиза.

Принцип работы CriteriaAPI сводится к тому, что поэлементно программно строится запрос на языке манипулирования данными (SQL или его аналоги для взаимодействия с нереляционными СУБД) и выполняется, возвращая 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 информацию не в одном объемном запросе, а в нескольких меньшего размера. Пагинация ускоряет получение части данных, но увеличивает количество обращений к СУБД.

5.3 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 структуру БД, но и изменять хранимую в ней информацию, создавая, изменяя или удаляя записи.

В приложении используется библиотека Flyway[46] для миграций SQL баз данных и Mongock[47] 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 разделять 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, указанной 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[48]. Здесь создаются коллекции в базе данных MongoDB, 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 этим параметром является 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 других сервисов.

Глава 6. Сервисный слой

6.1 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. Реализуемое 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[49], определяющий компактный и автономный способ безопасной передачи информации между сторонами в виде объекта JSON[50]. Эту информацию можно проверить и доверять ей,

поскольку она имеет цифровую подпись. JWT могут быть подписаны с использованием секрета (с помощью алгоритма HMAC[51]) или пары открытого/закрытого ключа с использованием RSA[52] или ECDSA[53].

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[54] 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.

Проект использует JWT при подтверждении аутентификации, так как необходимо точно знать, кто отправляет токен и можно ли 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 эндпоинты[55] являются защищёнными – некоторые из них доступны без авторизации, например, просмотр скетчей или запрос на регистрацию.

Создание нового пользователя происходит по электронной почте и паролю. Также есть возможность предоставить дополнительные данные – имя, фамилию, описание, аватар и прочую информацию. После этого на указанный адрес электронной почты отправляется письмо со ссылкой, пройдя по которой

регистрация подтверждается и пользователю становятся доступны механизмы авторизации.

После подтверждения авторизации, пользователь должен авторизоваться 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[56] 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 (естественно, в эту строку вставляется первый токен). Если отправленный токен успешно валидирован, то запрос выполняется и пользователю возвращается соответствующий ответ.

Рассмотрим код реализации механизма авторизации и аутентификации. В пакете model сервиса авторизации[57] хранятся классы, описывающие сущности пользователя, токена обновления и роли. Можно заметить, что класс пользователя (User) реализовывает интерфейс UserDetails. Это необходимо для того, чтобы фреймворк понимал, какой класс будет являться представлением пользователя в системе.

Далее, в пакете security сервиса авторизации[58] 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[59] 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 токена в контекст приложения добавляется текущая авторизация, хранящая информацию о пользователе, отправившем запрос. В классе JwtFilterChainConfigurer добавляется фильтр в соответствующее место в списке фильтров. Класс JwtTokenProvider отвечает за создание токенов (доступа и обновления) и их валидацию.

В классе SecurityConfiguration[60] бин[61] 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[62]. В этом классе можно заметить подключение OAuth2.0[63] 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 систему, где они обрабатываются в методе onAuthenticationSuccess класса OAuth2SuccessHandler[64].

Наконец, если в каком-либо защищённом эндпоинте необходимо получить информацию о пользователе, то к аргументам метода добавляется интерфейс Authentication. Фреймворк автоматически внедрит текущую пользовательскую аутентификацию в этот аргумент метода. Пример этого можно увидеть в методе me класса контроллера пользователей[65].

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[66].

6.2 Механизмы взаимодействия между сервисами

Ввиду того, что реализуемое 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 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 получит 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 проекте 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 должна быть блокирующей. В качестве брокера рассматривались две технологии – Apache Kafka[67] и RabbitMQ[68], 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»[69] 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. Это происходит следующим образом. 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 AuthenticationConfirmationRabbitMQService[70] метод sendConfirmation

добавляет через эксчейндж[71] новое сообщение в очередь, отвечающую за запросы на создание подтверждений новых пользователей. Этот метод вызывается в контроллере пользователей[65] в методе register, отвеающем за эндпоинт регистрации. Далее в методе addUserForConfirmation класса AuthenticationRabbitMQMessageReceiver[72] 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 подтверждения авторизации[73] происходит подтверждение регистрации.

6.3 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 скетечей[74]. В нем присутствуют различные методы для CRUD операций, функционал которых легко 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, которыми 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[75]. 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 внедряемого бина. В данном классе можно заметить аннотацию @Value. Ей помечается переменная, значение которой получается из переменной окружения. В данном случае значение приходит из файла application.yml[76]. 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 работающее приложение, то есть куда следует отправлять HTTP запросы для взаимодействия с ним.

Стоит отдельно поговорить про кеширование. Аннотация @CacheConfig позволяет установить общие настройки кеширования для всех методов этого класса или интерфейса. В рассматриваемом сервисе эта аннотация устанавливает имя кэша для всех методов. Аннотация @Cacheable помечает метод, результат выполнения которого должен быть сохранён в кэше, чтобы в будущем можно было получить быстрый доступ к нему. В этой аннотации есть возможность указать ключ, по которому будет происходить кеширование. По умолчанию ключом является весь список аргументов метода. Например, @Cacheable(key = “#id”) указывает, что результаты метода должны быть закешированы и извлекаться по значению аргумента id. С помощью атрибута condition можно определить условие, при котором результаты метода будут кешироваться. Например, @Cacheable(cacheNames = "myCache", condition = "#result != null") указывает, что результаты метода будут кешироваться только если результат не является null. Далее, аннотация @CacheEvict используется для удаления данных из кеша. Она применяется к методу и указывает, что при вызове этого метода соответствующие данные в кеше должны быть удалены. Также существует аннотация @CachePut, указывающая на то, что результат выполнения метода должен обновлять соответствующее значение в кеше.

Глава 7. 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 – Рисунок 5.

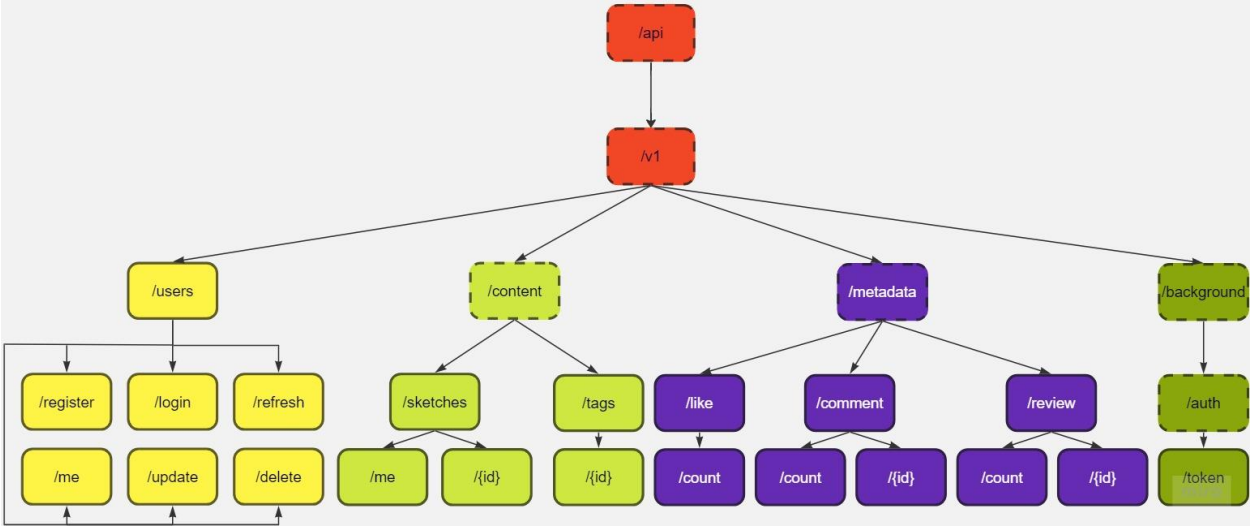

Рисунок 5. Дерево путей.

Пунктирными границами обозначены объекты, к которым нельзя обратиться, то есть на обработку которых не создано эндпоинта ввиду особенностей архитектуры приложения. Стоит отметить, что по одному и тому пути можно могут находиться несколько эндпоинтов, доступ к которым определяется типом HTTP. Запрос GET используется для получения данных с сервера. Клиент отправляет запрос на определенный ресурс, и сервер возвращает запрошенные данные в ответ. GET-запросы могут быть кэшированы и сохранены в истории браузера. Запрос POST используется для отправки данных на сервер. Обычно используется для отправки данных формы или загрузки файлов на сервер. POST-запросы не кэшируются и не сохраняются в истории браузера. Запрос PUT используется для загрузки или замены существующего ресурса на сервере. Он отправляет данные, которые должны заменить текущее состояние ресурса на сервере. Запрос DELETE используется для удаления указанного ресурса на сервере. После успешного выполнения запроса сервер удаляет указанный ресурс.

Рассмотрим пример конкретного контроллера[65]. Аннотация @RestController помечает класс как контроллер. Эта аннотация включает в себя @Component, помечающая 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. Аннотация @CrossOrigin помечающая класс контроллера или отдельные его эндпоинты, необходима для настройки CORS-механизма[77], позволяющего ограничивать или разрешать доступ к ресурсам на сервере из других источников (доменов), список которых передаётся в параметры аннотации. 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.

Пользователи могут обмениваться с сервером не только текстовыми данными, но и файлами, отправляемыми в телах запросов и ответов. Для реализации такого функционала существует аннотация @ModelAttribute. Ей помечают аргументы методов котроллера, которые являются файлами или, являясь дата-классами, могут содержать в себе файлы. При необходимости вернуть пользователю файл, можно применять специальные типы данных, например, HTTP-ответ ResponseEntity, с параметром Resource. Класс Resource предоставляет абстракцию для доступа к ресурсам различных типов, включая файлы, и предлагает удобные методы для получения доступа к содержимому ресурса. Таким же образом возможно реализовывать стримминг видео при необходимости.

Дополнительно к инструментам фреймворка Spring, в данном классе можно заметить аннотации @Tag и @Operation, принимающие параметры название и описание. Они помечают контроллер и его методы соответственно. Эти аннотации, вместе со многими другими аннотациями, используются для генерации документации на основе спецификации OpenAPI[78] библиотекой Swagger, сервер которой запускается вместе с сервером приложения. Такая документация содержит информацию о каждом эндпоинте, его параметрах, возможных ответах и других деталях, что облегчает понимание и использование API. Существует графический интерфейс Swagger, который позволяет разработчикам взаимодействовать с API без необходимости написания собственных HTTP-запросов. В интерфейсе Swagger отображается список доступных эндпоинтов, параметры запроса, возможные значения и другие детали. Доступ к графическому интерфейсу документации находится по адресу, прописанному в параметре springdoc.swagger-ui.path и в данном случае равному /docs/swagger.html. Отметим, что запросы, выполненные через графический интерфейс документации, отправляются на работающее приложение, как если бы эти запросы выполнил конечный пользователь системы. Именно поэтому доступ к документации есть лишь у двух ролей – администратора и разработчика.

Глава 8. Развертывание

Как было сказано в предыдущих главах, создаваемое приложение является распределённой системой, то есть состоит из нескольких взаимосвязанных компонентов, работающих вместе ради достижения общей цели. Хотя микросервисы и предлагают ряд преимуществ (масштабируемость, гибкость и динамичность), они также сопряжены с определенными проблемами при развертывании и поддержании работоспособности. Вот некоторые их этих проблем:

1. Обнаружение сервисов – микросервисы системы полагаются на обнаружение других сервисов для связи с ними. Это может быть сложной задачей, особенно когда количество сервисов увеличивается, а разные сервисы имеют отличные ограничения доступа.
2. Управление системой – мониторинг, ведение журнала, отслеживание и устранение неполадок в одном или нескольких сервисах одновременно являются сложными задачами при работе с распределённой системой;
3. Управление данными – при использовании микросервисной архитектуры, каждая служба владеет своими данными, что создает проблемы с их управлением. Может быть трудно обеспечить согласованность и синхронизацию между сервисами, а миграция данных может быть проблематичной.
4. Управление конфигурацией – микросервисы обычно содержат много сервисов и их реплик, каждый со своей конфигурацией. Управление конфигурациями нескольких служб может быть сложным процессом, подверженным ошибкам.
5. Порядок развёртывания – обеспечение правильного развертывания всех служб в правильном порядке, а также их откат является сложной задачей.

6. Управление версиями – микросервисы имеют версии, необходимые для обеспечения совместимости между сервисами. Управление версиями в нескольких сервисах особенно затруднительно при наличии большого числа зависимостей между ними.
7. Тестирование – тестирование микросервисов может быть сложной задачей, так как нужно тестировать множество сервисов, и у каждого сервиса могут быть свои зависимости и требования.

В целом развертывание и поддержка микросервисов может быть сложной задачей, но преимущества такой архитектуры часто перевешивают недостатки. Эффективное управление и мониторинг микросервисов могут помочь преодолеть большинство проблем и гарантировать, что архитектура выполняет свои обещания гибкости и масштабируемости.

Для решения вышеперечисленных проблем было решено использовать контейнеры Docker и систему оркестрации Kubernetes.

Контейнер Docker – единый модуль, упаковывающий 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.

Использование контейнеров Docker упрощает разработку, развертывание и управление микросервисной системой, а также обеспечивает большую гибкость и масштабируемость приложений.

Важно упомянуть и про существование DockerHub[79] – облачного репозитория, позволяющего хранить образы контейнеров не только в памяти компьютера, принадлежащего пользователю Docker, но и на удалённом сервере. Это предоставляет возможность запускать контейнеры, образы которых созданы на других устройствах, что особенно полезно при развертывании приложений на удалённых серверах, позволяя не пересоздавать образ локально на каждом используемом узле системы. Именно в DockerHub хранятся образы каждого сервиса, созданного в рамках работы над приложением.

Управление контейнеризованными компонентами системы и их масштабирование является сложной задачей, особенно по мере роста их количества. Для решения этой проблемы используется система оркестрации 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 приложения. На каждом узле в кластере Kubernetes работает среда выполнения контейнера 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 поде 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 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.
- PersistentVolume или, по другому, постоянные тома – механизм сохранения данных, находящихся внутри контейнера. Так как при удалении контейнера удаляются хранившиеся в нем данные, то необходимо отображать файловую систему контейнера на внешнюю файловую систему. Для этого как раз им служат тома.

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:

1. 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.
2. 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[80] или AWS[81]), так и локальный кластер Minikube[82]. После подключения к такому кластеру можно работать с командной строкой Kubernetes.

Существует два способа развёртывания компонент кластера – через командную строку и через выполнение файлов формата yaml[83]. В создаваемом 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 директории rabbitmq[84]. Там же присутствуют сценарии командной строки (файлы shell script), 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 database[85] находятся скрипты для разворачивания всех СУБД, необходимых системе. Рассмотрим их на примере базы данных Postgres[86]. 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[87]. Как и при запуске СУБД, здесь присутствует скрипт, создающий ConfigMap с переменными окружения – эти значения перезаписывают соответствующие значения из файла application.yml. Этот механизм удобен для настройки приложения без изменения исходного кода. Действительно, если необходимо, например, поменять секретный ключ для шифрования и дешифрования JWT, то 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.

ВЫВОД

В рамках данной выпускной квалификационной работы были выполнены следующие задачи:

1. Описана архитектура разработанной системы;
2. Описана дерево путей для взаимодействия с эндпоинтами системы;
3. Описаны использованные технологии и особенности работы с ними, а именно:
    a. Работа с реляционными и нереляционными базами данных через инструменты Hibernate и Java Persistence API;
    b. Создание миграций для реляционных и нереляционных баз данных и помощью Flyway и Mongock;
    c. Создание веб-приложений, с помощью Spring MVC и защита их с помощью Spring Security и JWT;
    d. Настройка кэширования в СУБД Redis;
    e. Настройка обмена сообщениями между микросервисами с помощью RabbitMQ;
    f. Запуск приложений в Docker контейнерах;
    g. Оркестрация контейнеров с микросервисами в кластере Kubernetes.
4. Реализованы микросервисы авторизации, контента, метаданных и фоновых задач с учётов вышеописанных особенностей.

ЗАКЛЮЧЕНИЕ

Таким образом в данной работе описаны различные аспекты всех этапов разработки веб-приложения – от проектирования до развёртывания.

Благодаря реализации архитектуры микросервисов и использованию фреймворка Spring, брокера сообщений RabbitMQ, JWT, контейнеров Docker и системы оркестрации Kubernetes спроектирована, написана и развернута защищённая, легко масштабируемая и высокопроизводительная система, которую легко контролировать, поддерживать и обновлять.

Созданное приложение имеет потенциал для расширения не только с точки зрения масштабирования и открытия точек доступа на географически отличных серверах, но и с точки зрения функциональных возможностей:

- внедрение интерфейса для исполнителей – клиентов сервиса, способных реализовывать вещи, созданные дизайнерами в виде эскизов, чтобы отправлять их покупателю;
- настройка сервисов оплаты;
- внедрение механизмов дополненной реальности для предоставления пользователям возможности взаимодействия с продуктом;
- внедрение голосового чата с использованием peer-to-peer[88] соединений;
- создание браузерного редактора эскизов, а также собственного формата данных, для их хранения;
- разработка и внедрение систем автоматизация логистики;
- внедрение машинного обучения с целью автоматической генерации продуктов на основании пользовательских предпочтений;
- внедрение NFT (Non Fungible Tokens)[89], привязывающихся к каждому уникальному эскизу, выложенному в систему.

В дальнейшем планируется развивать систему и внедрять новый функционал. Этот процесс значительно упрощается ввиду использования не только фремворка Spring, позволяющего писать эффективный код, но и настроенной системе оркестрации, облегчающей развертывание новых компонент системы. Разумеется, для ещё большей эффективности работы над приложением будут настроены процессы непрерывной интеграции[90] и непрерывного развёртывания[91], что позволит обновлять систему сразу после написания новой версии отдельного микросервиса или целого ряда микросервисов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Репозиторий с кодом проекта // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend (дата обращения: 08.05.2023).
2. Java // Официальный сайт компании Oracle URL: https://www.oracle.com/java/ (дата обращения: 08.05.2023).
3. PostgreSQL: The World's Most Advanced Open Source Relational Database // Официальный сайт технологии PostgreSQL URL: https://www.postgresql.org/ (дата обращения: 08.05.2023).
4. What is MongoDB? // Официальный сайт технологии MongoDB URL: https://www.mongodb.com/docs/manual/?_ga=2.40846946.1520107555.1683497062-767325683.1677707143 (дата обращения: 08.05.2023).
5. Redis // Официальный сайт технологии Redis URL: https://redis.io/ (дата обращения: 08.05.2023).
6. G. Lawrence Sanders, Seungkyoon Shin Denormalization Effects on Performance of RDBMS. - Proceedings of the 34th Hawaii International Conference on System Sciences, 2001.
7. Definition: in-memory database // WhatIs URL: https://www.techtarget.com/whatis/definition/in-memory-database (дата обращения: 08.05.2023).
8. Spring Framework Overview // Официальный сайт технологии Spring Framework URL: https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html (дата обращения: 08.05.2023).
9. What Is Swagger? // Официальный сайт технологии Swagger URL: https://swagger.io/docs/specification/2-0/what-is-swagger/ (дата обращения: 08.05.2023).

10. What is Gradle? // Официальный сайт технологии Gradle URL: https://docs.gradle.org/current/userguide/what_is_gradle.html (дата обращения: 08.05.2023).
11. What is IntelliJ IDEA? // Официальный сайт компании JetBrains URL: https://www.jetbrains.com/idea/features/ (дата обращения: 08.05.2023).
12. pgAdmin // Официальный сайт технологии PostgreSQL URL: https://www.pgadmin.org/ (дата обращения: 08.05.2023).
13. What is MongoDB Compass? // Официальный сайт технологии MongoDB URL: https://www.mongodb.com/docs/compass/current/ (дата обращения: 08.05.2023).
14. What is Postman? // Официальный сайт технологии Postman URL: https://www.postman.com/product/what-is-postman/ (дата обращения: 08.05.2023).
15. RFC 7230 // Библиотека RFC документов URL: https://www.ietf.org/rfc/rfc7230.html (дата обращения: 08.05.2023).
16. Develop faster. Run anywhere. // Официальный сайт технологии Docker URL: https://www.docker.com/ (дата обращения: 08.05.2023).
17. Why you need Kubernetes and what it can do // Официальный сайт технологии Kubernetes URL: https://kubernetes.io/docs/concepts/overview/ (дата обращения: 08.05.2023).
18. Software Containers: Used More Frequently than Most Realize // Networkworld URL: https://www.networkworld.com/article/2226996/software-containers--used-more-frequently-than-most-realize.html (дата обращения: 08.05.2023).
19. What is orchestration? // Официальный сайт компании Rad Hat URL: https://www.redhat.com/en/topics/automation/what-is-orchestration (дата обращения: 08.05.2023).
20. What are message brokers? // Официальный сайт компании IBM URL: https://www.ibm.com/topics/message-brokers (дата обращения: 08.05.2023).

21. Introduction to JSON Web Tokens // Официальный сайт технологии JWT URL: https://jwt.io/introduction (дата обращения: 08.05.2023).
22. How Does a CDN Work? // CDNetworks URL: https://www.cdnetworks.com/web-performance-blog/how-content-delivery-networks-work/ (дата обращения: 08.05.2023).
23. RFC 1034 // Библиотека RFC документов URL: https://www.ietf.org/rfc/rfc1034.txt (дата обращения: 08.05.2023).
24. Эрик Эванс Предметно-ориентированное проектирование (DDD). - Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. изд. - Москва, Санкт-Петербург, Киев: ООО «И.Д. Вильямс», 2011. - 443 с.
25. XML Schema-based configuration // Официальный сайт технологии Spring Framework URL: https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/xsd-configuration.html (дата обращения: 08.05.2023).
26. Java-based container configuration // Официальный сайт компании Oracle URL: https://docs.spring.io/spring-framework/docs/3.0.0.M4/reference/html/ch03s11.html (дата обращения: 08.05.2023).
27. Annotations Basics // Официальный сайт компании Oracle URL: https://docs.oracle.com/javase/tutorial/java/annotations/basics.html (дата обращения: 08.05.2023).
28. Java-based Container Configuration // Официальный сайт технологии Spring Framework URL: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-java (дата обращения: 08.05.2023).
29. Instantiating the Spring Container by Using AnnotationConfigApplicationContext // Официальный сайт технологии Spring Framework URL: https://docs.spring.io/spring-

framework/docs/current/reference/html/core.html#beans-java-instantiating-container (дата обращения: 08.05.2023).
30. The Java Virtual Machine Specification // Официальный сайт компании Oracle URL: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html (дата обращения: 08.05.2023).
31. Скрипт для сборки сервиса авторизации // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/build.gradle (дата обращения: 08.05.2023).
32. Declaring dependencies // Официальный сайт технологии Gradle URL: https://docs.gradle.org/current/userguide/declaring_dependencies.html (дата обращения: 08.05.2023).
33. Introduction to SQL // Официальный сайт компании Oracle URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/sql.html#GUID-CBD8FE77-BA6F-4241-A71C-2ADDDF43EA7F (дата обращения: 08.05.2023).
34. Overview of Spring Data JPA // Официальный сайт технологии Spring Framework URL: https://spring.io/projects/spring-data-jpa#overview (дата обращения: 08.05.2023).
35. Сущность пользователя системы // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/model/User.java (дата обращения: 08.05.2023).
36. Репозиторий пользователей в сервисе авторизации // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/repository/UserRepository.java (дата обращения: 08.05.2023).
37. Gilad Bracha Generics in the Java Programming Language // 2004

38. BSON // Официальный сайт технологии BSON URL: https://bsonspec.org/ (дата обращения: 08.05.2023).
39. Сущность эскиза в сервисе контента // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/document/Sketch.java (дата обращения: 08.05.2023).
40. Indexes // Официальный сайт технологии MongoDB URL: https://www.mongodb.com/docs/manual/indexes/ (дата обращения: 08.05.2023).
41. Репозиторий тегов в сервисе контента // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/repository/TagRepository.java (дата обращения: 08.05.2023).
42. Репозиторий скетчей в сервисе контента // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/repository/sketch/SketchRepository.java (дата обращения: 08.05.2023).
43. Пользовательская имплементация репозитория скетчей в сервисе контента // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/repository/sketch/SketchRepositoryImpl.java (дата обращения: 08.05.2023).
44. Overview of the Criteria and Metamodel APIs // Официальный сайт компании Oracle URL: https://docs.oracle.com/cd/E19798-01/821-1841/gjitv/index.html (дата обращения: 08.05.2023).
45. Пользовательская имплементация репозитория скетчей в сервисе метадаты // GitHub URL: https://github.com/DaniilDDDDD/disign-

workshop-backend/blob/master/metadata-service/src/main/java/com/workshop/metadataservice/repository/metadata/like/LikeRepositoryCustomImpl.java (дата обращения: 08.05.2023).
46. Quickstart - How Flyway Works // Официальный сайт технологии Flyway URL: https://documentation.red-gate.com/fd/quickstart-how-flyway-works-184127223.html (дата обращения: 08.05.2023).
47. Introduction into Mongock // Официальный сайт технологии Mongock URL: https://docs.mongock.io/#introduction (дата обращения: 08.05.2023).
48. Миграция инициализации коллекций в сервисе контента // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/migration/FirstInitializeCollections.java (дата обращения: 08.05.2023).
49. RFC 7519 // Библиотека RFC документов URL: https://www.rfc-editor.org/rfc/rfc7519 (дата обращения: 08.05.2023).
50. RFC 2627 // Библиотека RFC документов URL: https://www.rfc-editor.org/rfc/rfc4627 (дата обращения: 08.05.2023).
51. RFS 2104 // Библиотека RFC документов URL: https://www.rfc-editor.org/rfc/rfc2104 (дата обращения: 08.05.2023).
52. RFS 8017 // Библиотека RFC документов URL: https://www.rfc-editor.org/rfc/rfc8017 (дата обращения: 08.05.2023).
53. ASC X9 Issues New Standard for Public Key Cryptography/ECDSA // Accredited Standarts Commitee X9 URL: https://x9.org/asc-x9-issues-new-standard-for-public-key-cryptography-ecdsa/ (дата обращения: 08.05.2023).
54. RFC 4648 // Библиотека RFC документов URL: https://www.rfc-editor.org/rfc/rfc4648 (дата обращения: 08.05.2023).
55. What is a Server-Side Endpoint? // Официальный сайт компании Oracle URL: https://docs.oracle.com/cd/E17802_01/webservices/webservices/reference/tutorials/wsit/doc/Initialization2.html (дата обращения: 08.05.2023).

56. RFC 4122 // Библиотека RFC документов URL: https://www.ietf.org/rfc/rfc4122.txt (дата обращения: 08.05.2023).
57. Пакет моделей сервиса авторизации // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/auth-service/src/main/java/com/workshop/authservice/model (дата обращения: 08.05.2023).
58. Пакет безопасности сервиса авторизации // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/auth-service/src/main/java/com/workshop/authservice/security (дата обращения: 08.05.2023).
59. Spring Security Architecture // Официальный сайт технологии Spring Framework URL: https://spring.io/guides/topicals/spring-security-architecture/ (дата обращения: 08.05.2023).
60. Класс настройки безопасности сервиса авторизации // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/configuration/SecurityConfiguration.java (дата обращения: 08.05.2023).
61. Bean Overview // Официальный сайт технологии Spring Framework URL: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-definition (дата обращения: 08.05.2023).
62. Cross Site Request Forgery (CSRF) // Официальный сайт организации The Open Worldwide Application Security Project (OWASP) URL: https://owasp.org/www-community/attacks/csrf (дата обращения: 08.05.2023).
63. RFC 6819 // Библиотека RFC документов URL: https://www.ietf.org/rfc/rfc6819.txt (дата обращения: 08.05.2023).

64. Обработчик успешной авторизации на стороннем сервисе // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/handler/OAuth2SuccessHandler.java (дата обращения: 08.05.2023).
65. Контроллер пользователей в сервисе авторизации // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/controller/UserController.java (дата обращения: 08.05.2023).
66. Пользовательская имплементация интерфейса авторизации // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/security/JwtAuthentication.java (дата обращения: 08.05.2023).
67. Introduction into Apache Kafka // Официальный сайт технологии Apache Kafka URL: https://kafka.apache.org/documentation/#introduction (дата обращения: 08.05.2023).
68. What can RabbitMQ do for you? // Официальный сайт технологии RabbitMQ URL: https://www.rabbitmq.com/features.html (дата обращения: 08.05.2023).
69. Apache Kafka и RabbitMQ: в чем разница и что лучше изучать? // Хабр URL: https://habr.com/ru/companies/southbridge/articles/666326/ (дата обращения: 08.05.2023).
70. Класс для отправки сообщений через RabbitMQ в сервисе авторизации // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/service/messaging/AuthenticationConfirmationRabbitMQService.java (дата обращения: 08.05.2023).

71. AMQP 0-9-1 Model Explained // Официальный сайт технологии RabbitMQ URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html#amqp-model (дата обращения: 08.05.2023).
72. Обработчик сообщений от RabbitMQ в сервисе фоновых задач // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/background-service/src/main/java/com/workshop/backgroundservice/handler/AuthenticationRabbitMQMessageReceiver.java (дата обращения: 08.05.2023).
73. Контроллер подтверждения авторизации сервиса фоновых задач // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/background-service/src/main/java/com/workshop/backgroundservice/controller/AuthenticationConfirmationController.java (дата обращения: 08.05.2023).
74. Сервис для работы со скетчами // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/service/SketchService.java (дата обращения: 08.05.2023).
75. Container Overview // Официальный сайт технологии Spring Framework URL: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-basics (дата обращения: 08.05.2023).
76. Файл переменных окружения сервиса контента // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/resources/application.yml (дата обращения: 08.05.2023).
77. Cross-Origin Resource Sharing // W3C URL: https://www.w3.org/TR/2020/SPSD-cors-20200602/ (дата обращения: 08.05.2023).

78. OpenAPI Specification // Официальный сайт технологии Swagger URL: https://swagger.io/specification/ (дата обращения: 08.05.2023).
79. DockerHub // Официальный сайт технологии Docker URL: https://www.docker.com/products/docker-hub/ (дата обращения: 08.05.2023).
80. Kubernetes Engine API // Официальный сайт компании Google URL: https://console.cloud.google.com/marketplace/product/google/container.googleapis.com?returnUrl=%2Fkubernetes%3Fproject%3Dworkshop-367510&project=workshop-367510 (дата обращения: 08.05.2023).
81. Amazon Elastic Kubernetes Service // Официальный сайт компании Amazon URL: https://aws.amazon.com/ru/eks/?did=ap_card&trk=ap_card (дата обращения: 08.05.2023).
82. Minikube overview // Официальный сайт технологии Minikube URL: https://minikube.sigs.k8s.io/docs/ (дата обращения: 08.05.2023).
83. YAML 1.2 // Официальный сайт технологии YAML URL: https://yaml.org/ (дата обращения: 08.05.2023).
84. Скрипты для развёртывания RabbitMQ // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/rabbitmq (дата обращения: 08.05.2023).
85. Скрипты для развёртывания баз данных // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/database (дата обращения: 08.05.2023).
86. Скрипты для развёртывания базы данных Postgres // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/database/postgres (дата обращения: 08.05.2023).
87. Скрипты для развёртывания сервиса авторизации // GitHub URL: https://github.com/DaniilDDDDD/disign-workshop-

backend/tree/master/deployment/kubernetes/application/auth (дата обращения: 08.05.2023).

88. What's a Peer-to-Peer (P2P) Network? // Computerworld URL: https://www.computerworld.com/article/2588287/networking-peer-to-peer-network.html (дата обращения: 08.05.2023).
89. What Is An NFT? Non-Fungible Tokens Explained // Forbes Advisor URL: https://www.forbes.com/advisor/investing/cryptocurrency/nft-non-fungible-token/ (дата обращения: 08.05.2023).
90. What is Continuous Integration? // Официальный сайт компании Amazon URL: https://aws.amazon.com/devops/continuous-integration/?nc1=h_ls (дата обращения: 08.05.2023).
91. What is continuous deployment? // Официальный сайт компании IBM URL: https://www.ibm.com/topics/continuous-deployment (дата обращения: 08.05.2023).