САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Направление: 02.03.02 «Фундаментальная информатика и информационные технологии
ООП: Программирование и информационные технологии

Отчёт по производственной практике

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

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.

Выбранные технологии

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 и MongoDB.

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

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;

[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.

- 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.

Архитектура

Было решено использовать микросервисный подход ввиду его гибкости в разработке и поддерживании, простой масштабируемости и высокой доступности.

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.

Для авторизации и аутентификации в приложении используются JWT[8] токены.

Стоит отметить, что изначальная версия архитектуры была упрощена до той, что можно видеть на Рисунке 1, ввиду дороговизны и сложности её поддержания, отсутствия большого числа пользователей, а также отсутствия необходимости реализации некоторых функций на ранних этапах.

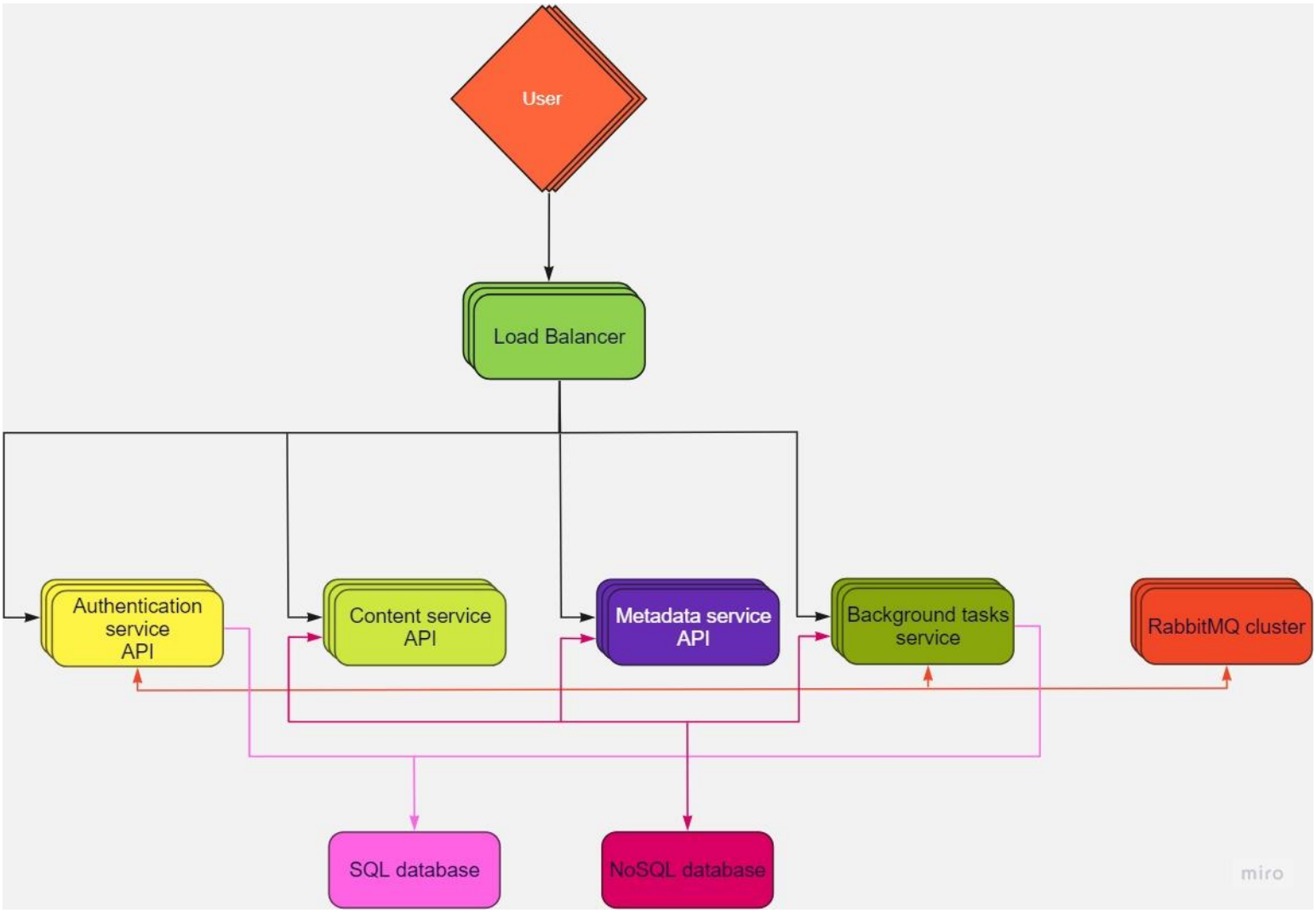

Рисунок 1. Архитектура приложения

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 информации
- CRUD операции с основным контентом приложения, которым являются скетчи, тэги и сопутствующие медиафайлы (фотографии, видео и прочее)
- CRUD операции с метаинформацией (оценки, комментарии, отзывы)

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

- 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 2.

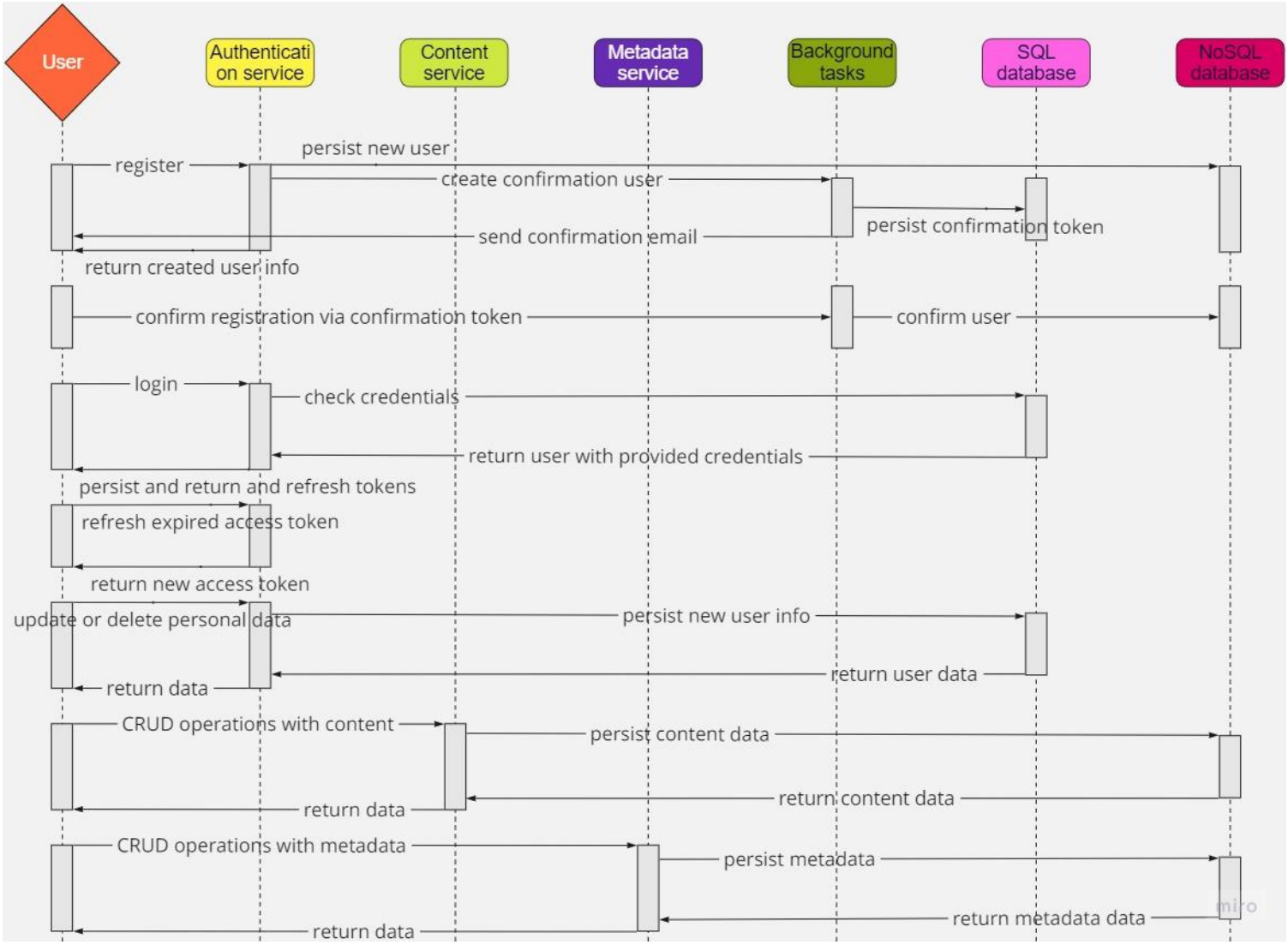

Рисунок 2. Поведенческая диаграмма приложения.

При определении способа разбиения приложения на микросервисы была создана структурная диаграмма, не только описывающая схему базы данных, но и отображающая ограниченные контексты приложения – изображена Рисунке 3. 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]).

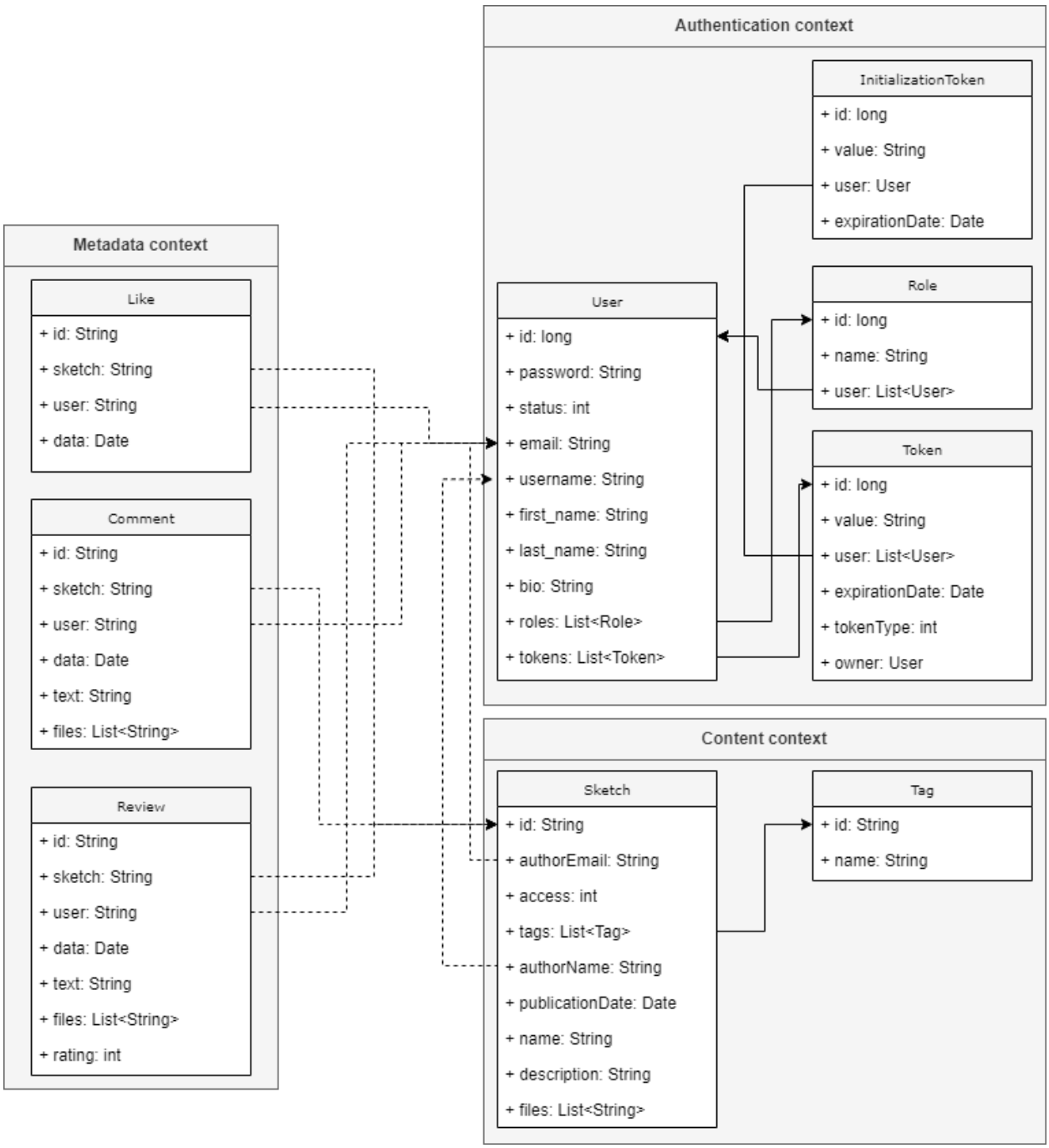

Рисунок 3. Структурная диаграмма приложения.

Глава 1. Слой данных

1.1 Реляционные СУБД

В нашем приложении слой данных, взаимодействующий с SQL СУБД, реализован с помощью технологии Spring Data JPA[3]. 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 лучшего понимания слоя данных нашего приложения можно обратиться к коду, описывающему сущность пользователя системы[27].

Класс User отображает таблицу пользователей. Он аннотирован[54] @Entity, что означает, что он является сущностью JPA, отображающей 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 указывает 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[28].

Первое что можно заметить – интерфейс наследуется от встроенного Spring Data JPA интерфейса. Это необходимо для автоматической генерации тел методов, который

[3] JPA – Java Persistence API[53]

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, 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 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.

1.2 Нереляционные СУБД

В нашем приложении помимо SQL СУБД используется документоориентированная NoSQL база данных MongoDB, которая информацию в виде документов BSON[4], 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[29].

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

[4] BSON – Binary JSON[55]

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[30].

В нашем приложении для взаимодействия с данными используются не только встроенные инструменты фреймворка, но и их имплементации[31]. Можно заметить, что данный интерфейс наследуется не только от интерфейса MongoRepository, но и от интерфейса SketchRepositoryCustom, который представляет собой интерфейс для взаимодействия с методами, описанными в классе SketchRepositoryImpl[32]. Внутри этих методов используется CriteriaAPI[56], 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 LikeRepositoryCustomImpl[33], 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.

1.3 Миграции

Миграция базы данных это изменение структуры базы данных от одной версии к 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[57] для миграций SQL баз данных и Mongock[58] для миграций баз данных MongoDB. Оба этих инструмента создают схемы в соответствующих СУБД, а также заполняют их начальными 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[34]. 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 мощным 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 2. Сервисный слой

2.1 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. Наше приложение использует JWT для 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.

Создание нового пользователя происходит по электронной почте и паролю. Также есть возможность предоставить дополнительные данные – имя, фамилия, описание, аватар и прочую информацию. 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 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, который хранит в своих полезных данных почтовый адрес пользователя. Он служит для авторизации при каждом запросе. Второй токен 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 годности.

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 токен успешно валидирован, то запрос выполняется и пользователю возвращается соответствующий ответ.

Рассмотрим код реализации механизма авторизации и аутентификации. В сервисе авторизации в пакете model[35] 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 security[36] 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[59], 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 отвечает за создание токенов (доступа и обновления) и их валидацию.

В классе SecurityConfiguration[37] 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[61] с помощью средств, предоставляемых компанией Google[60]. 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 OAuth2SuccessHandler[38].

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 UserController[39].

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.

2.2 Механизмы взаимодействия между сервисами

Ввиду того, что наше приложение является распределённой системой, возникает потребность взаимодействия между микросервисами. В нашем приложении было решено использовать брокер RabbitMQ.

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 AuthenticationConfirmationRabbitMQService[49] метод sendConfirmation добавляет через эксчейндж новое сообщение в очередь, отвечающую за запросы на создание подтверждений новых пользователей. Этот метод

вызывается в контроллере UserController[50] в методе register, отвечающем за эндпоинт регистрации. Далее в методе addUserForConfirmation класса AuthenticationRabbitMQMessageReceiver[51] 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 AuthenticationConfirmationController[52] происходит подтверждение регистрации.

2.3 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 скетчей[41]. 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, 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[42]. 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 3. 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 4 изображено это дерево путей.

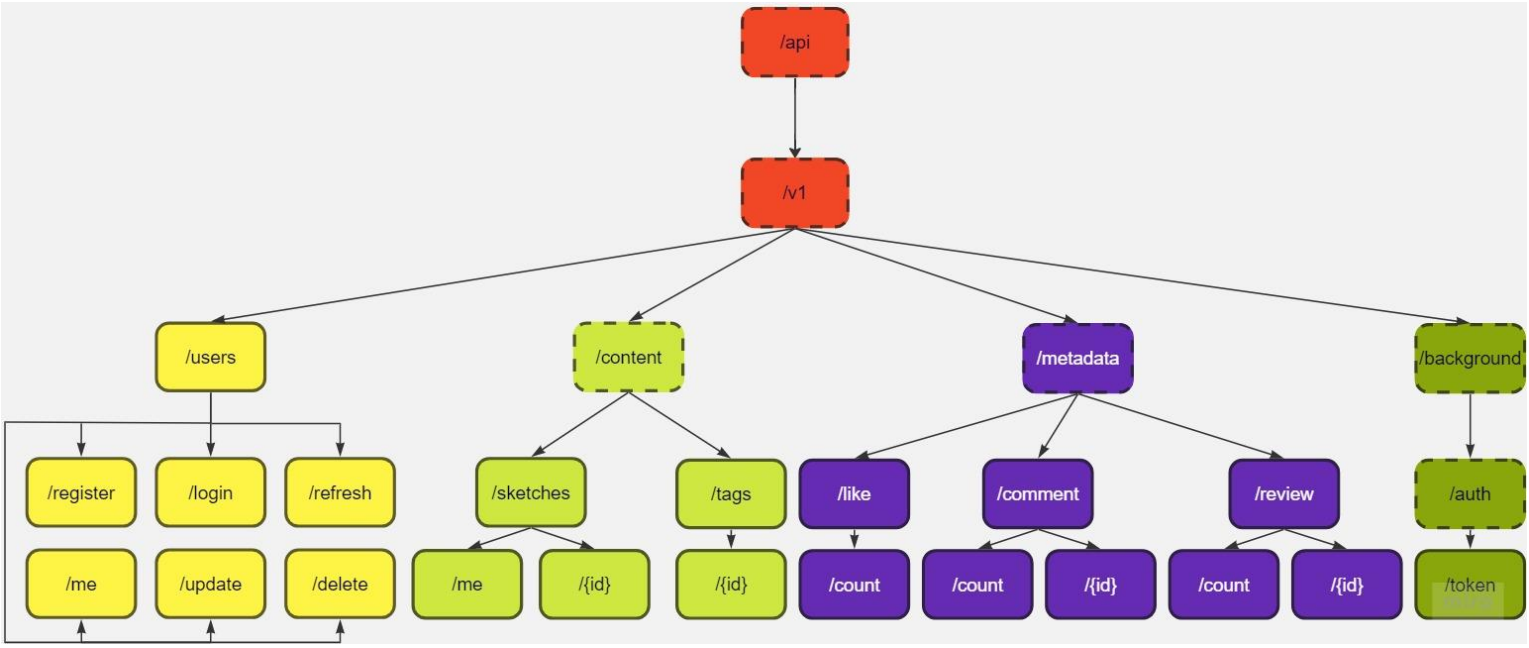

Рисунок 4. Дерево путей.

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 которым определяется типом HTTP запроса – GET, POST, PUT или DELETE.

Рассмотрим пример конкретного контроллера[43]. Аннотация @RestController помечает класс как REST[62] 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

Для развёртывания нашей системы было решено использовать контейнеры Docker[19] в связке с системой оркестрации Kubernetes[20].

Важно упомянуть про DockerHub[23] – облачный репозиторий, позволяющий 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 мы храним образы каждого сервиса, созданного в рамках работы над нашим приложением.

Для разворачивания приложения в кластере необходимо иметь как таковой кластер. Это может быть как облачный кластер (например GCP[17] или AWS[18]), так и локальный кластер minikube[16]. 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 отдельной папке в репозитории[44]. 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 отдельной директории[45] находятся скрипты для разворачивания всех СУБД, необходимых системе. Рассмотрим их на примере базы данных Postgres[46]. 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[47]. Как и при запуске СУБД, здесь присутствует 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

Таким образом в данной работе мы описали 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 мы смогли спроектировать, написать и развернуть защищённую, легко масштабируемую и высокопроизводительную систему, которую легко контролировать, поддерживать и обновлять.

Наше 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. В дальнейшем мы будем развивать систему и внедрять новый функционал. Этот процесс значительно упрощается благодаря 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[24] и непрерывного развёртывания[25], что позволит обновлять развёрнутую систему сразу после написания новой версии отдельного микросервиса или целого ряда микросервисов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. GitHub-репозиторий с кодом проекта – https://github.com/DaniilDDDDD/disign-workshop-backend
2. Эрик Эванс. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. Пер. с англ. – М.: ООО «И.Д. Вильямс», 2011.
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. Minikube – https://minikube.sigs.k8s.io/docs/
17. Google Cloud Platform – https://console.cloud.google.com/marketplace/product/google/container.googleapis.com?returnUrl=%2Fkubernetes%3Fproject%3Dworkshop-367510&project=workshop-367510
18. Amazon Web Services – https://aws.amazon.com/ru/eks/?did=ap_card&trk=ap_card
19. Docker – https://www.docker.com/
20. Kubernetes – https://kubernetes.io/
21. Контейнеризация – https://en.wikipedia.org/wiki/OS-level_virtualization
22. Оркестрирование – https://en.wikipedia.org/wiki/Orchestration_(computing)

23. DockerHub – https://hub.docker.com/
24. Непрерывная интеграция – https://en.wikipedia.org/wiki/Continuous_integration
25. Непрерывное развёртывание – https://en.wikipedia.org/wiki/Continuous_delivery
26. In-memory база данных – https://en.wikipedia.org/wiki/In-memory_database
27. Сущность пользователя системы – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/model/User.java
28. Репозиторий пользователей – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/repository/UserRepository.java
29. Сущность скетча – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/document/Sketch.java
30. Репозиторий тегов– https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/repository/TagRepository.java
31. Репозиторий скетчей – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/repository/sketch/SketchRepository.java
32. Пользовательская имплементация репозитория скетчей – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/repository/sketch/SketchRepositoryImpl.java
33. Пользовательская имплементация репозитория лайков – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/metadata-service/src/main/java/com/workshop/metadataservice/repository/metadata/like/LikeRepositoryCustomImpl.java
34. Миграция, инициализирующая коллекции – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-

service/src/main/java/com/workshop/contentservice/migration/FirstInitializeCollections.java

35. Пакет с моделями сервиса авторизации – https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/auth-service/src/main/java/com/workshop/authservice/model
36. Пакет с настройками безопасности сервиса авторизации – https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/auth-service/src/main/java/com/workshop/authservice/security
37. Конфигурация системы безопасности – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/configuration/SecurityConfiguration.java
38. Обработчик OAuth2 авторизации – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/handler/OAuth2SuccessHandler.java
39. Контроллер пользователей – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/controller/UserController.java
40. Пользовательская реализация интерфейса Authentication – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/security/JwtAuthentication.java
41. Сервис логики скетчей – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/java/com/workshop/contentservice/service/SketchService.java
42. Файл с переменными окружения – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/content-service/src/main/resources/application.yml
43. Контроллер пользователей – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/controller/UserController.java
44. Скрипты для развёртывания брокера сообщений – https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/rabbitmq

45. Скрипты для развёртывания СУБД – https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/database
46. Скрипты развёртывания СУБД Postgres – https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/database/postgres
47. Скрипты развёртывания сервиса авторизации – https://github.com/DaniilDDDDD/disign-workshop-backend/tree/master/deployment/kubernetes/application/auth
48. NFT – https://en.wikipedia.org/wiki/Non-fungible_token
49. RabbitMQService – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/service/messaging/AuthenticationConfirmationRabbitMQService.java
50. Контроллер пользователей сервиса фоновых задач – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/auth-service/src/main/java/com/workshop/authservice/controller/UserController.java
51. Обработчик сообщений от брокера сервиса фоновых задач – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/background-service/src/main/java/com/workshop/backgroundservice/handler/AuthenticationRabbitMQMessageReceiver.java
52. Контроллер подтверждения авторизации сервиса фоновых задач – https://github.com/DaniilDDDDD/disign-workshop-backend/blob/master/background-service/src/main/java/com/workshop/backgroundservice/controller/AuthenticationConfirmationController.java
53. The Java Persistence – https://www.oracle.com/technical-resources/articles/java/jpa.html
54. Java аннотации – https://docs.oracle.com/javase/6/docs/technotes/guides/language/annotations.html
55. BSON – https://bsonspec.org/
56. Criteria API – https://docs.oracle.com/cd/E19798-01/821-1841/gjitv/index.html
57. Flyway – https://flywaydb.org/
58. Mongock – https://mongock.io/

59. Встроенные в Spring фильтры – https://spring.io/guides/topicals/spring-security-architecture/
60. Google – https://en.wikipedia.org/wiki/Google
61. OAuth2.0 – https://en.wikipedia.org/wiki/OAuth
62. REST – https://en.wikipedia.org/wiki/Representational_state_transfer
63.