

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное автономное образовательное учреждение высшего  
образования «Самарский национальный исследовательский университет имени  
академика С.П. Королева» (Самарский университет)

Институт информатики и кибернетики

Кафедра информационных систем и технологий

КУРСОВОЙ ПРОЕКТ ПО ДИСЦИПЛИНЕ  
«Разработка WEB-приложений»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

ТЕМА:

«Создание микроблога»

Обучающийся \_\_\_\_\_ С. В. Тиликанов  
*(подпись)*

Руководитель работы \_\_\_\_\_ И.В. Лёзина  
*(подпись)*

Оценка \_\_\_\_\_

Самара 2026

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» (Самарский университет)

Институт информатики и кибернетики

Кафедра информационных систем и технологий

**ЗАДАНИЕ НА КУРСОВОЙ ПРОЕКТ**

Обучающемуся Тиликанову Сергею Валерьевичу группы 6596-090301Z

Тема проекта: «Создание микроблога»

Планируемые результаты освоения образовательной программы (компетенции)	Планируемые результаты практики	Содержание задания
ПК-3 Способен осуществлять интеграцию программных модулей и компонент и верификацию выпусков программного продукта ПК-3.1. Разрабатывает процедуры интеграции программных модулей	<b>знать:</b> теоретические основы проектирования и создания моделей баз данных и программного обеспечения при разработке web-приложений в рамках технологии Java Enterprise Edition; <b>уметь:</b> создавать и проверять работоспособность моделей баз данных и программного обеспечения при разработке web-приложений в рамках технологии Java Enterprise Edition; <b>владеть:</b> современными программными продуктами для создания моделей баз данных и программного обеспечения при разработке	1. Разработка логической модели базы данных для микроблога Разработка автоматизированной информационной системы микроблога. 2. Отладка и тестирование разработанной автоматизированной информационной системы микроблога. 3. Проведение экспериментов и анализ результатов.

	web-приложений в рамках технологии Java Enterprise Edition.	
--	---	--

Дата выдачи задания 25 сентября 2025 г.

Срок представления на кафедру пояснительной записки 22 января 2026 г.

Руководитель курсового проекта

доцент кафедры ИСТ, к.т.н., доцент \_\_\_\_\_ И.В. Лёзина

*(подпись)*

Задание принял к исполнению

обучающийся группы № 6596-090301Z \_\_\_\_\_ С. В. Тиликанов

*(подпись)*

## РЕФЕРАТ

**Пояснительная записка к курсовому проекту:** 40 страниц, 24 рисунка, 9 источников, 1 приложение.

ИНФОРМАЦИОННАЯ СИСТЕМА, JAKARTAEE-ТЕХНОЛОГИИ, СУБД, POSTGRESQL, TOMCAT, WEB-ИНТЕРФЕЙС, SPA, RESTAPI, SERVLETS, УПРАВЛЕНИЕ МИКРОБЛОГОМ.

Цель работы – разработать автоматизированную информационную систему управления новостным порталом.

В ходе курсового проекта разработана информационная система на основе JakartaEE-технологий с использованием клиент-серверной архитектуры REST. В качестве серверной части использован фреймворк Spring Boot, отвечающий за бизнес-логику и взаимодействие с базой данных. Клиентская часть реализована в виде одностраничного приложения (SPA) с использованием фреймворка Angular, который формирует динамический интерфейс на HTML и TypeScript. Взаимодействие между клиентом и сервером организовано через RESTful API с аутентификацией на основе JWT-токенов.

В качестве системы управления базами данных (СУБД) использован PostgreSQL. Серверная часть развёрнута на встроенном сервере приложений, совместимом со стандартами Tomcat.

Система предоставляет возможность управления данными микроблога. А именно: регистрацию и авторизацию пользователей, публикацию и удаление пользовательских постов с возможностью прикрепления фотографий, комментирование постов, оценку постов («лайки»), а также ведение пользовательских профилей. Вся работа с системой производится через адаптивный веб-интерфейс, представленный в виде клиентского SPA-приложения.

Программа написана на языке Java в среде IntelliJ IDEA 2025.1.1 и функционирует под управлением операционной системы Windows 10/11.

## ОГЛАВЛЕНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ.....	6
ВВЕДЕНИЕ.....	8
1. Структура базы данных .....	10
1.1.Описание предметной области. Логическая структура данных. ....	10
1.2.Физическая схема базы данных.....	11
2. Архитектура приложения .....	13
2.1.Архитектурная модель.....	13
2.1.1. Принципы независимой архитектуры веб-приложений .....	13
2.1.2. Слои Web-приложения. ....	14
2.1.3. Внедрение зависимостей .....	16
2.1.4. Масштабирование приложения. ....	16
2.2.Выбор и обоснование использования технологий. ....	17
2.2.1. Выбор языка программирования и среды разработки .....	17
2.2.2. Выбор backend-фреймворка.....	18
2.2.3. Выбор системы управления базами данных .....	19
2.2.4. Выбор способа работы с базой данных .....	19
2.2.5. Используемые средства и библиотеки:.....	20
2.2.6. Выбор frontend-фреймворка.....	21
2.3.Авторизация и аутентификация. ....	23
2.3.1. JWT-токен .....	23
2.3.2. Процедура генерации токена. ....	25
2.3.3. Процедура валидации токена.....	26
2.4.Средства моделирования.....	27
2.4.1. Диаграмма вариантов использования .....	27
2.4.2. Схемы основных алгоритмов.....	27
3. Пользовательский интерфейс .....	30
ЗАКЛЮЧЕНИЕ .....	34
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ .....	35
Приложение А: Взаимодействие слоёв	
приложения при работе с сущностью User. ....	36

## ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

- JakartaEE-технологии - набор спецификаций и соответствующей документации для языка Java, описывающей архитектуру серверной платформы для задач средних и крупных предприятий.
- API (Application Programming Interface – интерфейс программирования приложений) – это интерфейс, который одна программа или сервис предоставляет другим программам для безопасного и структурированного использования своих данных и функций.
- REST – (англ. Representational state transfer - передача состояния представления) – архитектурный стиль взаимодействия компонентов распределённого приложения в сети. Для веб-служб, построенных с учётом REST (то есть не нарушающих накладываемых им ограничений), применяют термин «RESTful».
- JWT (англ. JSON Web Token) — это открытый стандарт (RFC 7519) для создания токенов доступа, основанный на формате JSON. Как правило, его используют в передаче данных для аутентификации или авторизации в клиент-серверных приложениях.
- SPA – (англ. Single Page Application) - это веб-приложение, которое загружается один раз и затем динамически обновляет контент на той же странице, не загружая полностью новые HTML-страницы с сервера.
- Фреймворк (англ. framework – каркас, структура) - заготовка, готовая модель в программировании для быстрой разработки, на основе которой можно дописать собственный код. Он задает структуру, определяет правила и предоставляет необходимый набор инструментов для создания проекта.
- Фронтенд (англ. frontend) – это клиентская часть продукта (интерфейс, с которым взаимодействует пользователь). В случае с сайтом ее формирует и выводит на экран браузер, который работает с HTML, CSS и JavaScript.

- Бэкэнд (англ. backend) — это внутренняя часть продукта, которая находится на сервере и скрыта от пользователей. Для ее разработки могут использоваться самые разные языки, например, Python, PHP, Java, C#.
- ORM (англ. Object Relational Mapping - объектно-реляционное отображение) - это программная технология, которая создает «мост» между объектно-ориентированными языками программирования и реляционными базами данных, преобразуя данные между двумя принципиально разными парадигмами.
- СУБД — это система управления базами данных. Так называют программное обеспечение, которое требуется, чтобы создавать базы данных, изменять их, получать из них информацию и контролировать версии.
- UI (англ. User Interface – пользовательский интерфейс) — представляет собой комплекс визуальных элементов, через которые пользователь взаимодействует с программным продуктом[1].

## ВВЕДЕНИЕ

В современном цифровом мире социальные сети и микроблоги стали неотъемлемой частью коммуникации, обмена информацией и формирования общественного мнения. Все более актуален запрос на более легковесные, кастомизируемые и контролируемые решения, которые можно адаптировать под конкретные задачи – будь то корпоративная социальная сеть, платформа для сообщества или учебный проект для освоения современных веб-технологий. Лучший способ изучить работу какого-либо приложения – разработать собственную версию этого приложения. Такой метод позволит комплексно изучить ключевые аспекты создания современных распределенных информационных систем: от проектирования серверной логики и баз данных до реализации динамического пользовательского интерфейса. Разработка функционального прототипа микроблога предоставляет уникальную возможность применить теоретические знания на практике, столкнувшись с реальными архитектурными решениями и проблемами интеграции, которые остаются «за кадром» при простом использовании готовых платформ.

Целью данной курсовой работы является разработка и практическая реализация прототипа автоматизированной информационной системы – микроблога – с использованием современных стеков технологий клиент-серверной архитектуры. В рамках достижения этой цели планируется не только создать работоспособное приложение, но и провести сравнительный анализ выбранных инструментов, исследовать их синергию при построении RESTful API, а также освоить ключевые практики современной веб-разработки, такие как JWT-авторизация, управление состоянием на клиенте и работа с реляционными данными через ORM.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Провести анализ предметной области, определить функциональные и нефункциональные требования к системе.

2. Спроектировать логическую и физическую модель базы данных, обеспечивающую хранение информации о пользователях, постах, комментариях, медиафайлах и социальных взаимодействиях.

3. Реализовать серверную часть (backend) приложения на основе фреймворка Spring Boot, обеспечивающую RESTful API, бизнес-логику, безопасность (аутентификацию и авторизацию) и взаимодействие с СУБД PostgreSQL.

4. Провести анализ доступных frontend-фреймворков, и выбрать наиболее подходящий для наших целей. Разработать клиентскую часть (frontend) в виде одностраничного приложения (SPA) с использованием данного фреймворка.

5. Провести интеграционное тестирование системы, отладить взаимодействие компонентов и проанализировать полученные результаты.

## 1. Структура базы данных

### 1.1. Описание предметной области. Логическая структура данных.

Центральным элементом нашего приложения является возможность для пользователей публиковать короткие сообщения, обмениваться мнениями и формировать социальные связи в режиме реального времени. Каждый пользователь имеет персональный профиль для самовыражения и может взаимодействовать с контентом других через комментарии и реакции. Исходя из этого, выберем следующие сущности для базы данных:

- Пользователь – сущность, которая представляет человека, использующего сервис. Имеет уникальный идентификатор, логин, имя, фамилию, краткую информацию о себе, зашифрованный пароль, путь к изображению пользователя в хранилище и дата создания.
- Пост – пост пользователя. У него есть заголовок, локация, содержимое поста, привязанный к нему автор и дата создания.
- Комментарий – это реакция пользователя на публикацию. Каждый комментарий создаётся конкретным пользователем, привязывается к определённому посту, содержит уникальный идентификатор, текст сообщения и метку времени создания.
- Лайки – сущность, выражающая связь между пользователем и постом, регистрирующая положительную реакцию данного пользователя на данный пост. Помимо этого, также содержит уникальный идентификатор.
- Изображения – сущность, содержащая данные о пути к файлу изображения. Содержит собственный идентификатор, путь к файлу в хранилище и обязательно связана с одним конкретным постом.

Теперь мы можем построить логическую схему структуры данных (Рис 1).



Рис 1. Логическая схема базы данных.

## 1.2. Физическая схема базы данных.

Дополним логическую модель деталями, необходимыми для СУБД.

Таблица 1. Таблица сущностей физической модели базы данных.

Сущность	Поле	Тип данных	Ограничения
users	id	BIGINT	PRIMARY KEY
	username	VARCHAR(64)	NOT NULL UNIQUE
	firstname	VARCHAR(64)	NOT NULL
	lastname	VARCHAR(64)	NOT NULL
	email	VARCHAR(64)	NOT NULL UNIQUE
	bio	TEXT	
	avatar	VARCHAR(128)	
	password	TEXT	NOT NULL
	created_date	DATE	
post	id	BIGINT	PRIMARY KEY
	title	VARCHAR(64)	NOT NULL
	caption	TEXT	
	location	VARCHAR(64)	
	user_id	BIGINT	FOREIGN KEY REFERENCES users ON DELETE CASCADE
	created_date	DATE	

Таблица 1 (Продолжение). Табл. сущностей физической модели базы данных.

Сущность	Поле	Тип данных	Ограничения
comment	id	BIGINT	PRIMARY KEY
	user_id	BIGINT	NOT NULL REFERENCES users
	message	TEXT	NOT NULL
	created_date	DATE	
	post_id	BIGINT	NOT NULL REFERENCES post ON DELETE CASCADE
image_model	id	BIGINT	PRIMARY KEY
	imageURL	VARCHAR(128)	UNIQUE
	post_id	BIGINT	
likes_post	id	BIGINT	PRIMARY KEY
	user_id	BIGINT	UNIQUE (user_id, post_id) REFERENCES users ON DELETE CASCADE
	post_id	BIGINT	REFERENCES post ON DELETE CASCADE

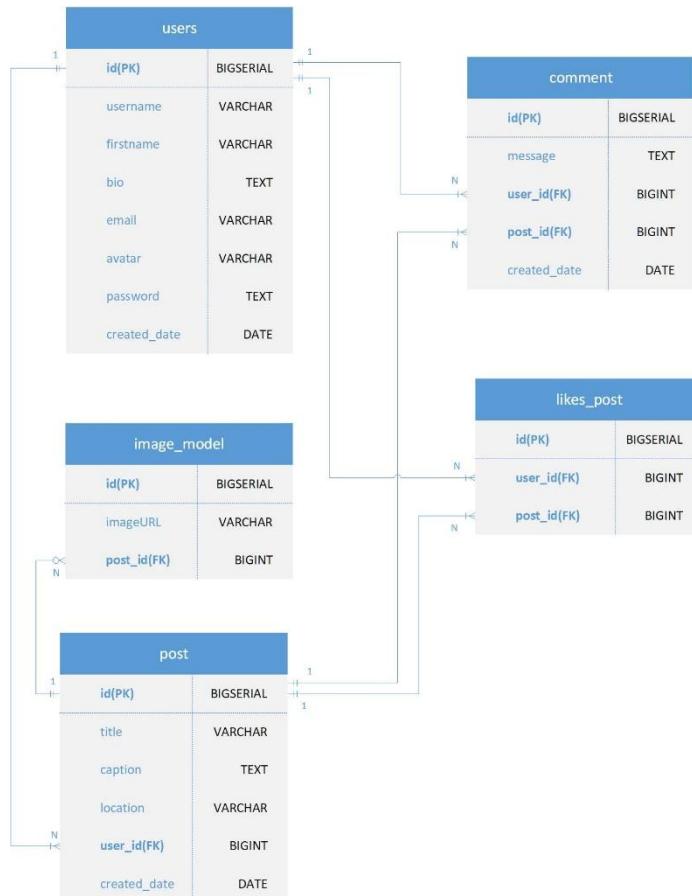


Рис 2. Физическая схема базы данных

## 2. Архитектура приложения

### 2.1. Архитектурная модель

#### 2.1.1. Принципы независимой архитектуры веб-приложений

Основные идеи, заложенные в архитектуру Web-приложения:

1. Независимость от фреймворка. Архитектура не зависит от существования какой-либо библиотеки. Это позволяет использовать фреймворк в качестве инструмента вместо того, чтобы втискивать свою систему в рамки его ограничений.

2. Тестируемость. Бизнес-правила могут быть протестированы без пользовательского интерфейса, базы данных, веб-сервера или любого другого внешнего компонента.

3. Независимость от UI. Пользовательский интерфейс можно легко изменить, не изменяя остальную систему. Например, веб-интерфейс может быть заменен на консольный, без изменения бизнес-правил.

4. Независимость от базы данных. Можно поменять Oracle или SQL Server на MongoDB, BigTable, CouchDB или что-то еще. Бизнес-правила не должны быть связаны с базой данных.

5. Независимость от какого-либо внешнего сервиса. По факту наши бизнес-правила просто ничего не знают о внешнем мире.

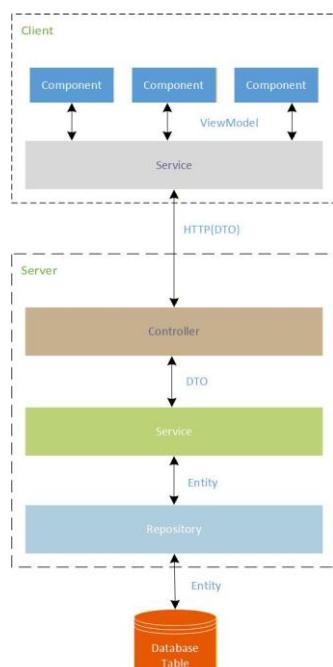


Рис 3. Слои Web- приложения

Достигается такая гибкость за счет разделения приложения на слои Service, Repository, Model (Рис 3) [2]. Рассмотрим каждый из этих слоев поподробнее.

### 2.1.2. Слои Web-приложения.

- Entity — отвечают за работу с базой данных и представляют из себя структуру повторяющую таблицу или документ в базе данных.
- Repository — отвечает за общение с хранилищем данных. В качестве хранилища может выступать сервер, база данных, память, localstorage, sessionstorage или любое другое хранилище. Его задача абстрагировать слой Service от конкретной реализации хранилища. В нашем случае роль хранилища исполняет база данных.
- Service — отвечает за всю бизнес логику приложения. Если Controller'у понадобилось получить, обработать, отправить какие-то данные — он делает это через Service. Если нескольким контроллерам понадобилась одна и та же логика, они работают с Service. Но сам слой Service ничего не должен знать о слое Controller и окружении, в котором он работает.
- DTO (Data Transfer Object) — служат для переноса данных между разными слоями приложения. Зачастую, в клиент-серверных приложениях, данные на клиенте и на сервере структурируются по-разному. На стороне сервера это дает нам возможность комфортно хранить данные в базе данных или оптимизировать использование данных в угоду производительности, в то же время заниматься “user-friendly” отображением данных на клиенте[3]. Таким образом, одной из задач Service-слоя является преобразование Entity данных в DTO данные и наоборот. В нашем приложении такой задачей занимаются классы, реализующие интерфейс Mapper:

```
public interface Mapper<F, T> {  
    T map(F from);  
}
```

- Контроллер (Controller) — отвечает за обработку HTTP-запросов и управление потоком данных между клиентом и бизнес-логикой. Он принимает входящие запросы, выполняет валидацию входных данных (например, с

использованием DTO или различных схем валидации), вызывает соответствующие методы сервисного слоя для выполнения бизнес-операций и формирует структурированные HTTP-ответы (успешные или ошибки). Контроллер не должен содержать бизнес-логику или напрямую взаимодействовать с базой данных — его задача состоит лишь в координации: преобразовании сетевых запросов в вызовы сервисов и возврате результатов в удобном для клиента формате, в нашем случае — JSON[4].

— HTTP-запросы: когда клиент отправляет запрос, он сериализует свои данные (например, объекты JavaScript либо TypeScript) в строковый формат JSON, XML или FormData, который помещается в тело HTTP-запроса с указанием соответствующего Content-Type. Сервер, получив запрос, выполняет десериализацию — парсит эти строковые данные обратно в структуры, понятные серверному языку, с одновременной валидацией на корректность и безопасность. В обратном направлении сервер сериализует результаты обработки в подходящий формат (чаще всего JSON), устанавливает заголовки ответа и отправляет клиенту, где данные снова десериализуются.

Эти процессы обеспечивают единый язык общения между разнородными системами, позволяя им обмениваться сложными структуризованными данными через текстовые HTTP-сообщения. Такой обмена данных между клиентом и сервером позволяет добиться существенного снижения нагрузки на сервер и трафика, так как передаются только минимально необходимые данные (например, 2 КБ JSON вместо 50 КБ HTML с разметкой и стилями). Кроме того, это позволяет добиться интерактивности и бесшовного UI: клиент (JavaScript/TypeScript в браузере) может динамически обновлять отдельные части интерфейса без перезагрузки всей страницы, создавая плавные SPA-приложения.

— В клиент-слое Service данные десериализуются, преобразуются во ViewModel, и далее отсылаются тому компоненту, который их запросил. В контроллерах эти данные обрабатываются, и отрисовываются на странице.

В приложении А приведен вертикальный срез описанных классов на примере сущности User.

### 2.1.3. Внедрение зависимостей

Для обеспечения масштабирования приложения применяют паттерн Dependency Injection, который служит нескольким ключевым целям:

- Обеспечение слабой связанности (loose coupling) между слоями: контроллер не зависит от конкретной реализации сервиса, а только от его интерфейса. Это позволяет легко заменять реализации (например, MongoUserRepository на PostgresUserRepository) без изменения кода контроллера.
- Централизация управления жизненным циклом объектов: DI-контейнер (инжектор) контролирует создание и время жизни зависимостей (синглтоны, экземпляры на запрос), что уменьшает дублирование кода и управляет ресурсами.
- Обеспечение чистоты и наглядности кода: зависимости явно объявлены в конструкторе, что упрощает понимание, от чего зависит компонент, и соблюдает принцип единой ответственности (SRP).

### 2.1.4. Масштабирование приложения.

По сравнению с другими архитектурами (например, Redux) архитектура слоев обеспечивает сбалансированную масштабируемость (Рис 4).

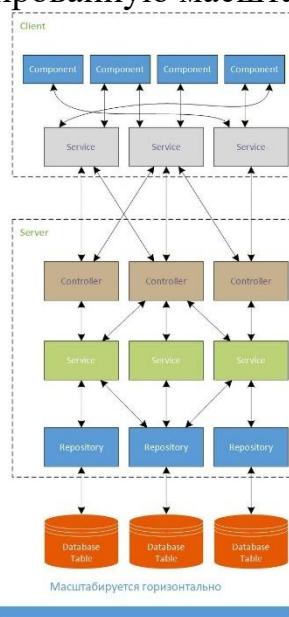


Рис 4. Пример масштабирования слоев Web-приложений

Как мы можем увидеть, за счет разделения логики и обязанностей между слоями приложение масштабируется горизонтально (Рис 4). Если какому-либо компоненту понадобится обработать данные, он обратится к соответствующему сервису, не трогая не связанные с его задачей сервисы. После получения данных произойдет перерисовка только одного компонента. Цепочка обработки данных очень короткая и очевидная, и максимум за 4 перехода по коду можно найти нужную логику и отладить ее. Кроме того, если провести аналогию с кирпичной стеной, то мы можем изъять любой кирпичик из этой стены и заменить его на другой, никак не повлияв на устойчивость этой стены.

## 2.2. Выбор и обоснование использования технологий.

### 2.2.1. Выбор языка программирования и среды разработки

Java — это один из самых популярных языков программирования, который активно используется в веб-разработке уже более двух десятилетий. Его универсальность, надежность и богатый набор инструментов делают его отличным выбором для создания масштабируемых и высокопроизводительных веб-приложений. Рассмотрим основные преимущества Java в веб-разработке:

- Код, написанный на Java, может быть выполнен на любой платформе, где установлена Java Virtual Machine (JVM). Веб-приложения на Java могут легко развертываться на серверах с различными операционными системами, будь то Windows, Linux или macOS, без необходимости значительных изменений кода.
- Java предлагает широкий набор фреймворков и библиотек, которые позволяют быстро и эффективно разрабатывать надежные и масштабируемые веб-приложения. Некоторые из них мы рассмотрим ниже.
- JVM обеспечивает безопасное выполнение кода, а встроенные механизмы безопасности, такие как контроль доступа и шифрование, позволяют защитить приложения от различных угроз.

- Многопоточность позволяет веб-приложениям на Java обрабатывать множество запросов одновременно, что значительно повышает их производительность и масштабируемость[5].

В качестве основной IDE выберем IntelliJ IDEA — наиболее популярную и продвинутую среду разработки среди Java-разработчиков. Это решение обусловлено несколькими ключевыми преимуществами:

- Богатая экосистема плагинов, позволяющая расширять функциональность среды и адаптировать её под конкретные задачи проекта.
- Глубокая интеграция с системами контроля версий (Git, SVN, Mercurial), что обеспечивает эффективную командную работу и управление кодом.
- Поддержка широкого спектра языков (Kotlin, Scala, Groovy, JavaScript, HTML, CSS и др.) помимо Java, что делает среду универсальной для разнотипных проектов.
- Расширенные возможности отладки, включая отображение значений переменных, вычисление выражений и многое другое. Это помогает эффективно выявлять и устранять ошибки.

### 2.2.2. Выбор backend-фреймворка

Spring Boot является стандартным выбором для создания современных web-приложений на Java. Рассмотрим главные достоинства Spring Boot для работы с такими проектами:

- Автоконфигурация – фреймворк самостоятельно настраивает приложение на основе подключенных зависимостей.
- Встроенные серверы — приложение включает в себя Tomcat, Jetty или Undertow, не требуя отдельной установки.
- Spring Boot Starters — готовые наборы зависимостей для типовых сценариев разработки.
- Широкая поддержка баз данных и технологий — удобная работа с SQL (через JPA/Hibernate) и NoSQL базами, кэшированием, очередями и современными протоколами (REST, WebSocket, gRPC).

### 2.2.3. Выбор системы управления базами данных

PostgreSQL — одна из самых популярных реляционных СУБД в мире.

Главными целями системы были: расширяемость, корректность данных и соответствие стандартам языка SQL. Академическая ДНК сделала СУБД функциональной и надежной, что привлекло внимание компаний, ищущих альтернативу коммерческим СУБД вроде Oracle. Рассмотрим основные причины популярности PostgreSQL:

- Одним из главных преимуществ PostgreSQL является его статус открытого исходного кода. Это означает, что любой желающий может свободно использовать, модифицировать и распространять СУБД без каких-либо лицензионных отчислений.
- PostgreSQL известен своей исключительной надежностью и стабильностью. Он строго придерживается принципов ACID, что гарантирует целостность и согласованность данных даже в случае сбоев системы.
- PostgreSQL стремится к максимальному соответствию стандартам SQL, что облегчает миграцию с других СУБД и обеспечивает высокую переносимость кода.
- PostgreSQL поддерживается большим и активным глобальным сообществом разработчиков, пользователей и энтузиастов. Это сообщество постоянно работает над улучшением СУБД, выпуская новые версии с расширенным функционалом и исправлениями ошибок[6].

### 2.2.4. Выбор способа работы с базой данных

Наиболее распространенным ORM-фреймворком для работы с базами данных в Java является Hibernate. Фреймворк используют при создании информационных систем: приложений, крупных программ и сетей, которые работают с информацией и базами данных. Основная идея фреймворка – создать «виртуальную» базу данных из программных объектов и работать с ней как с реальной базой. Поэтому им часто пользуются для упрощения работы. Он берет на себя взаимодействие с реляционной БД, а разработчику остается работать с кодом.

Преимущества Hibernate:

- ускоряет и облегчает написание кода;
- позволяет создать удобную модель для отображения базы данных в коде;
- дает возможность быстро и в читаемом виде записывать информацию из кода в базу.
- Реализует базовые CRUD-операции (create, read, update, delete), а также поддерживает сложные связи, транзакции, кэширование и миграции.

Таким образом, Hibernate снижает порог входа в работу с базами данных и повышает продуктивность разработки, оставаясь стандартом для Java-приложений, работающих с реляционными СУБД.

#### 2.2.5. Используемые средства и библиотеки:

Помимо уже описанных систем и фреймворков в проекте задействованы некоторые другие системы и библиотеки, которые тоже стоило бы упомянуть:

1) Gradle — это система автоматизации сборки (build automation system), которая используется для управления жизненным циклом разработки проекта. Она предоставляет удобный и гибкий способ описания процесса сборки, зависимостей и других задач, связанных с разработкой программного обеспечения. Вот некоторые из основных возможностей:

- Управление зависимостями: Система обеспечивает простой способ управления зависимостями проекта. Зависимости могут быть определены в файле сборки, и Gradle автоматически загрузит их из удаленных репозиториев.
- Плагины: Gradle расширяется с помощью плагинов, которые добавляют дополнительную функциональность.
- Интеграция с IDE: Система интегрируется с IntelliJ IDEA. Это обеспечивает единый опыт разработки для разработчиков.
- Gradle поддерживает инкрементальную сборку, что означает, что он пересобирает только те части проекта, которые изменились, ускоряя процесс сборки [1].

## 2) Система контроля версий Git:

Системы контроля версий являются фундаментальным инструментом в разработке программного обеспечения. В данный проект Git была внедрена в целях обеспечения надёжного хранения и отслеживания истории изменений исходного кода проекта. Каждое изменение фиксируется с комментарием, что позволяет в любой момент восстановить предыдущее состояние, понять, кто, когда и почему внёс конкретные правки, и откатить ошибочные изменения [7].

## 3) Библиотека Lombok:

Lombok — это библиотека для сокращения кода в классах и расширения функциональности языка Java. Lombok предоставляет обширные возможности для автоматической генерации геттеров, сеттеров, конструкторов, методов `toString()` и др. Использование Lombok позволяет многократно сократить шаблонный код, сэкономить время разработки, улучшить читаемость кода.

## 4) Библиотека Liquibase:

Liquibase — это система управления изменениями базы данных (Database Migration Tool), которая решает критически важную проблему контроля и версионирования структуры БД в процессе разработки и развёртывания приложения. Liquibase позволяет хранить скрипты миграций (changesets) в системе контроля версий (Git) вместе с кодом приложения. Это обеспечивает синхронизацию изменений кода и структуры БД, даёт возможность отслеживать историю изменений схемы и откатываться к предыдущим версиям[1].

### 2.2.6. Выбор frontend-фреймворка

Выбор подходящего frontend-фреймворка — один из ключевых этапов планирования веб-проекта. Сегодня на рынке доминируют три решения: Vue, React и Angular. Каждый из них имеет свои преимущества, ограничения и экосистему. Кратко рассмотрим каждый из них:

1) Vue.js — прогрессивный JavaScript-фреймворк, разработанный для постепенного внедрения. Он сочетает простоту и мощь, позволяя создавать как маленькие виджеты, так и сложные SPA.

Достоинства:

- Легкий вход: понятный синтаксис, минимальное количество boilerplate-кода.
- Документация: одна из самых качественных и понятных.
- Реактивность: реактивная система, основанная на data binding.
- Компонентный подход: разделение UI на переиспользуемые компоненты.

Недостатки:

- Меньшая популярность и экосистема по сравнению с React и Angular.
- Недостаток крупных корпораций, использующих Vue на уровне enterprise.

2) React — библиотека для построения пользовательских интерфейсов от Facebook. Популярность объясняется гибкостью, широким сообществом и огромным количеством сопутствующих инструментов.

Преимущества:

- JSX: сочетание JavaScript и HTML в одном файле.
- Virtual DOM: высокая производительность за счёт виртуализации изменений.
- Большая экосистема: Redux, MobX, Next.js.
- Поддержка сообществом: активная разработка и множество ресурсов.

Недостатки:

- Неопределенность экосистемы: множество альтернативных решений (разные способы стейт-менеджмента).
- Кривая обучения: освоение JSX и комбинация библиотек.

3) Angular – фреймворк от Google, предлагающий полный набор инструментов «из коробки». Подходит для крупных корпоративных приложений.

Преимущества:

- TypeScript изначально: строгая типизация.
- Opinionated-подход: чёткая структура приложения.
- Встроенные инструменты: маршрутизация, DI (Dependency Injection), формы, Http-клиент.
- Поддержка: долгосрочная и мощная поддержка от Google.

Недостатки:

- Большой размер: фреймворк весит больше, чем Vue или React.
- Сложность: кривая обучения выше, чем у Vue.
- Версионная несовместимость: при больших обновлениях могут потребоваться значительные правки кода [8].

Из трех фреймворков выберем Angular по двум главным причинам:

- Angular изначально построен на TypeScript с полной типизацией, строгой проверкой на этапе компиляции и продвинутым инструментарием для работы с типами. Поэтому, несмотря на всю сложность языка, работа на Angular зачастую похожа на работу с типичным ООП-языком.
- Angular CLI предоставляет готовые команды для генерации кода, сборки, тестирования, развёртывания и обновления версий, значительно ускоряя разработку и обеспечивая стандартизацию. React Create App и Vue CLI также предлагают инструменты, но они менее всеобъемлющи в плане архитектурных решений.

### 2.3. Авторизация и аутентификация.

#### 2.3.1. JWT-токен

В нашем проекте аутентификация производится с помощью JWT токена. Такой метод имеет несколько преимуществ перед другими методами (например, обычной защитой паролем):

- Учетные данные пользователя, как правило хранятся долго (месяцы). Как бы хорошо не был зашифрован запрос, при достаточном количестве времени его можно расшифровать. Если запрос, содержащий учетные данные перехвачен злоумышленником, у него будет много времени на расшифровку. Токены доступа имеют ограниченный срок годности (обычно ~15 минут). Этого времени недостаточно, чтобы расшифровать надежный шифр. К тому времени, когда зловредный алгоритм расшифрует запрос, токен уже выйдет из обращения и будет бесполезен.
- Использовать учетные данные - медленно. Для валидации учетных данных сервер должен запросить их сохраненную копию из БД и сравнить с данными, которые пришли в запросе. Обращение к БД — дорогостоящая процедура, она сильно увеличивает время обработки запроса. Токены, с другой стороны, не требуют обращения к БД для валидации. Это позволяет снизить нагрузку на БД и ускорить обработку запросов сервером [9].

Токен состоит из 3 частей, разделенных точкой (Рис 5):

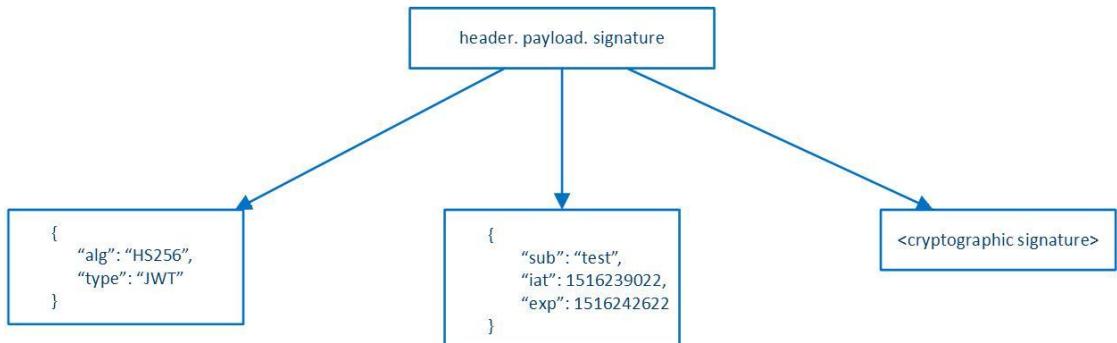


Рис 5. Структура JWT

- 1) header — содержит информацию об алгоритме шифрования и типе токена (JWT)
- 2) payload — данные токена. Мы добавили следующие поля в наш токен:
  - sub (Subject) — собственник токена. Как правило — uuid пользователя.
  - iat (Issued At) — время создания токена.
  - exp (Expiration Time) — время, в течение которого токен считается валидным.

Помимо этого, в токен можно добавлить следующие поля:

- iss (Issuer) — издатель токена. Как правило — uuid приложения, выпустившего токен.
- aud (Audience) — массив url серверов, для которых предназначен токен.
- nbf (Not Before) — временная метка, до которой токен не считается валидным

3) signature — строка, полученная из частей токена (header + payload) при помощи шифрования.

### 2.3.2. Процедура генерации токена.

Рассмотрим алгоритм генерации токена на сервере при прохождении процедуры авторизации (Рис 6).

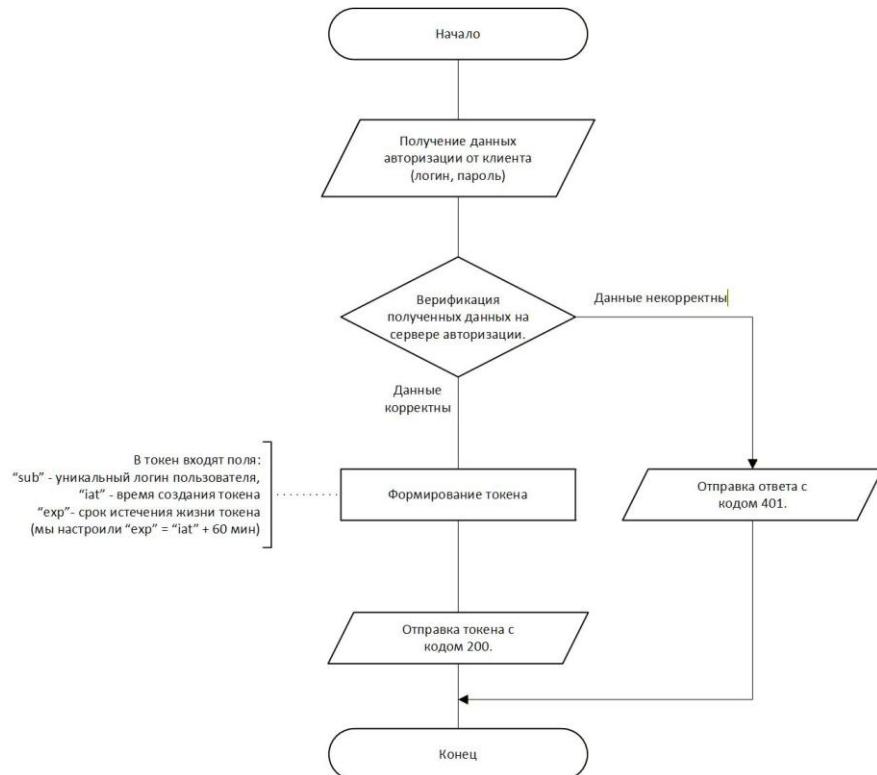


Рис 6. Блок схема авторизации пользователя

Описание алгоритма авторизации пользователя:

- Пользователь вводит логин и пароль и отправляет их на сервер.
- Сервер сверяет данные с имеющимися пользователями в таблице users. В случае корректности данных генерируется токен и отправляется на клиент в теле ответа на запрос.

- Клиент сохраняет токен в sessionStorage и добавляет его в заголовок каждого последующего запроса к серверу.

### 2.3.3. Процедура валидации токена.

Рассмотрим алгоритм валидации токена на сервере при получении запроса от клиента (Рис 7).

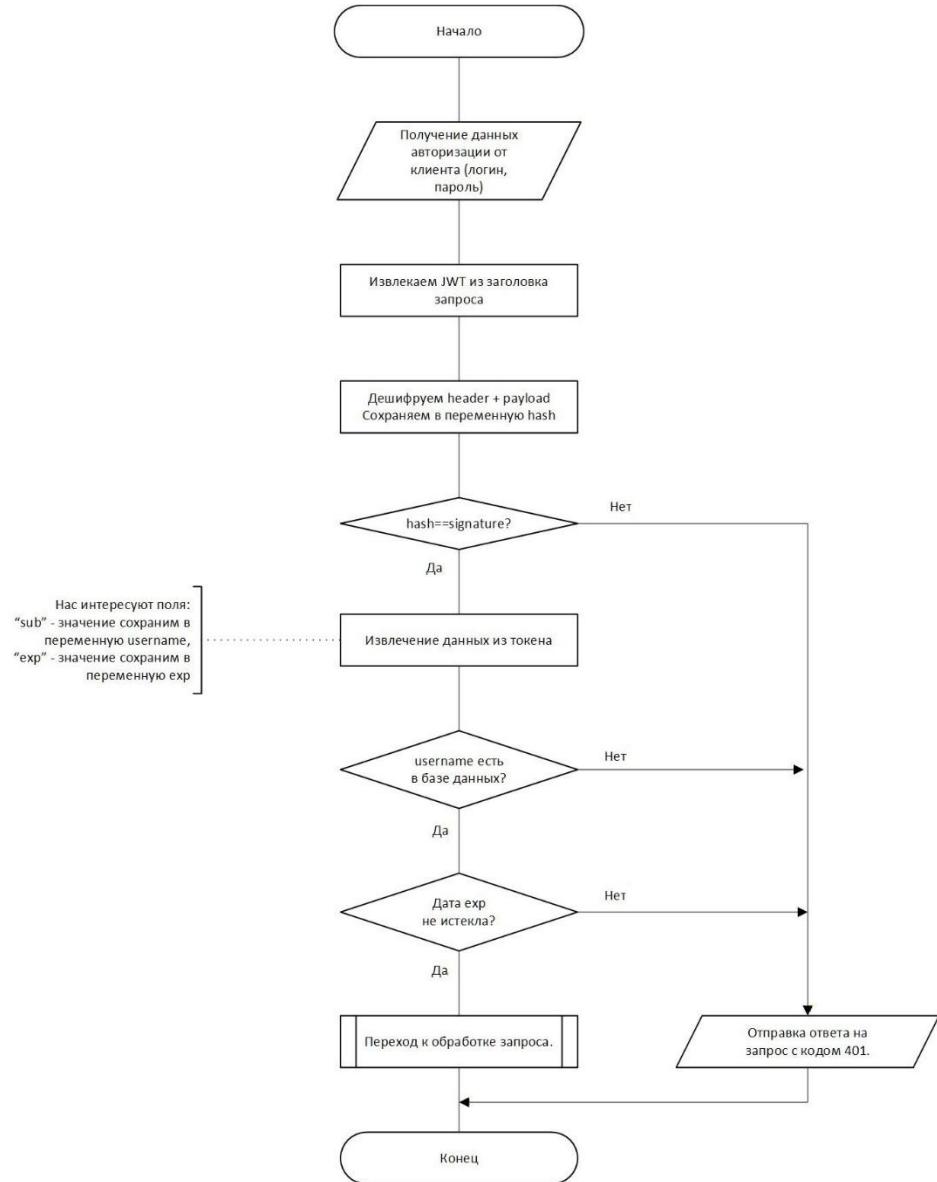


Рис 7. Блок схема валидации пользователя

### Описание алгоритма авторизации пользователя:

- При формировании запроса на стороне клиента из sessionStorage извлекается токен, и прикрепляется к заголовку запроса с ключом “Authorization”.

- Принимаем запрос на стороне сервера. Извлекаем и дешифруем токен. Проверяем содержимое. В случае прохождения проверки переходим к обработке запроса. В противном случае возвращаем ответ с кодом 401.

## 2.4. Средства моделирования.

### 2.4.1. Диаграмма вариантов использования

Для наглядного описания возможностей системы воспользуемся диаграммой вариантов использования. Каждый вариант использования определяет типичный сценарий взаимодействия между пользователем (актором) и системой, описывая последовательность действий, выполняемых системой в ответ на действия пользователя.

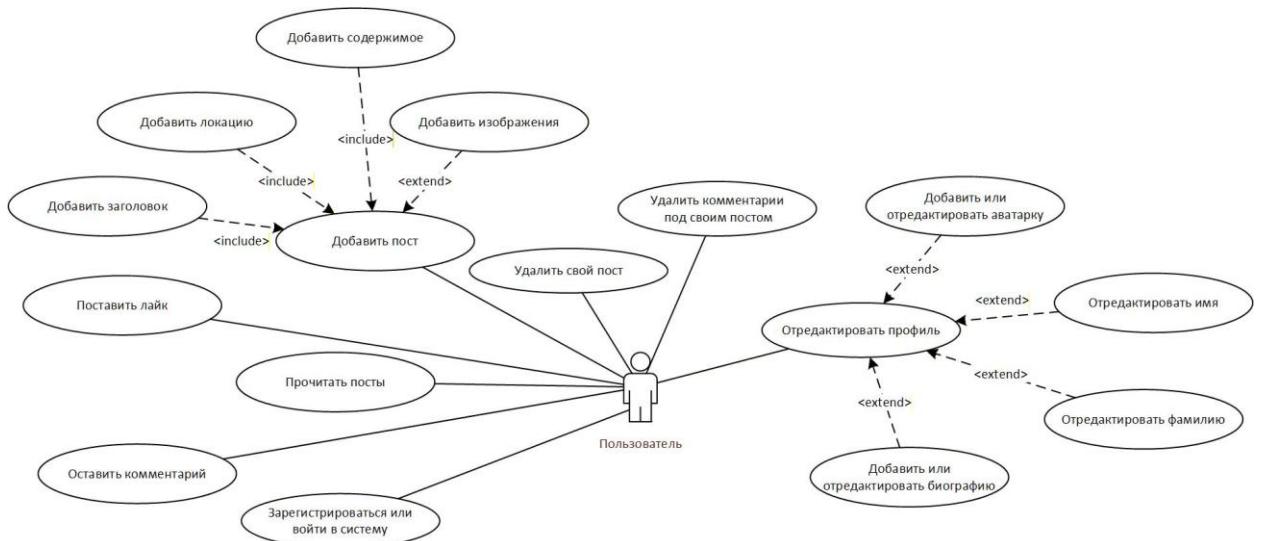


Рис 8. Диаграмма вариантов использования

### 2.4.2. Схемы основных алгоритмов

Основные действия в системе — создание и просмотр сущностей. Поэтому покажем основные схемы алгоритмов на примерах запроса и вывода постов, а также добавления нового поста в базу данных.

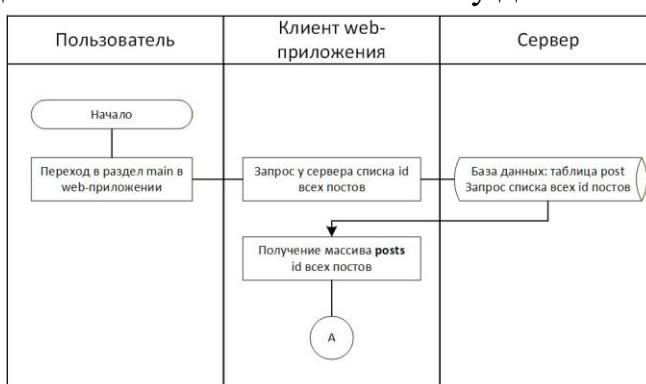


Рис 9а – Алгоритм вывода всех постов

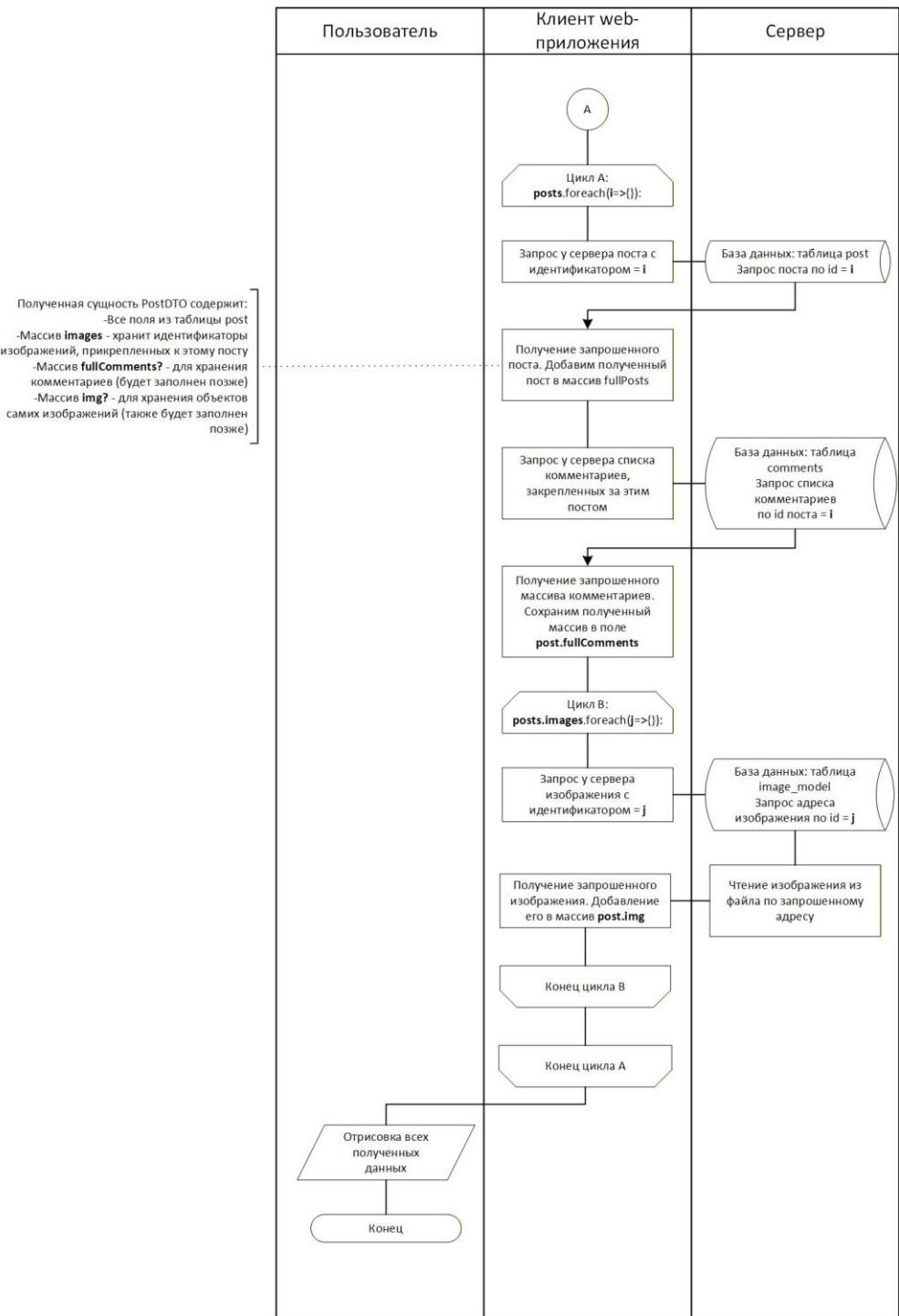


Рис 9б – Алгоритм вывода всех постов

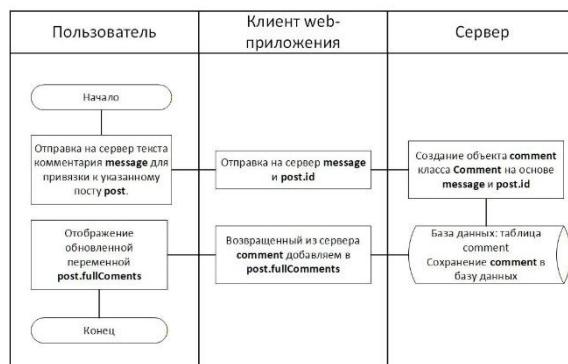


Рис 10. Алгоритм сохранения и вывода нового комментария под постом.

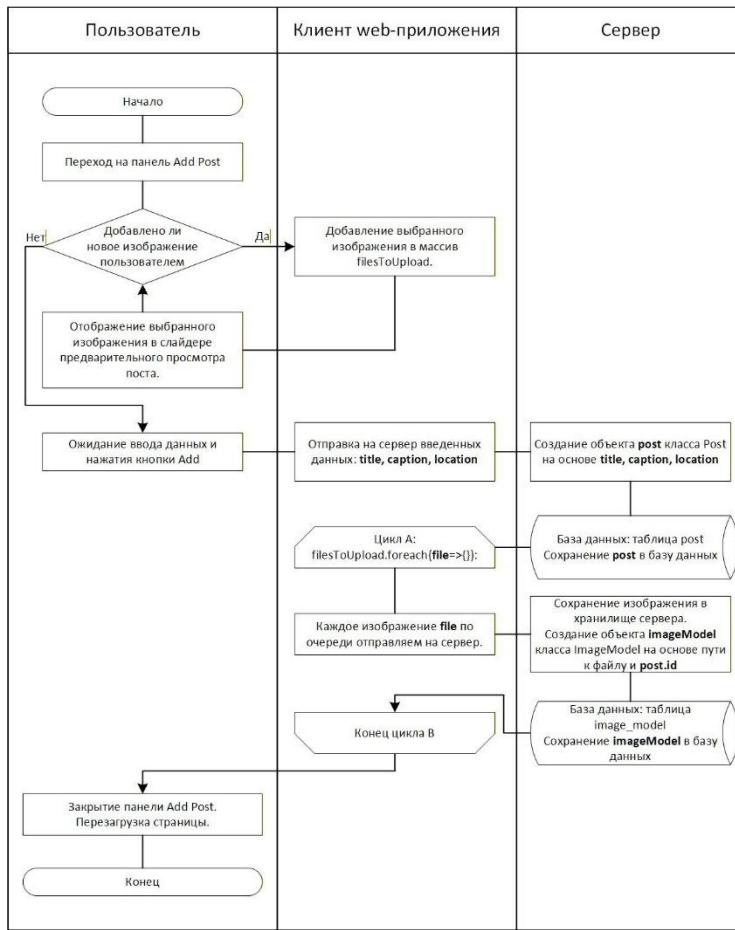


Рис 11. Алгоритм сохранения нового поста.

На рисунках 9а и 9б показан полный цикл работы с постами: от инициации запроса клиентом до получения ответа от сервера и финального отображения данных пользователю.

На рисунке 10 показан алгоритм сохранения комментария в базе данных.

На рисунке 11 показан алгоритм создания нового поста в базе данных вместе с изображениями.

### 3. Пользовательский интерфейс

Проверим работу UI:

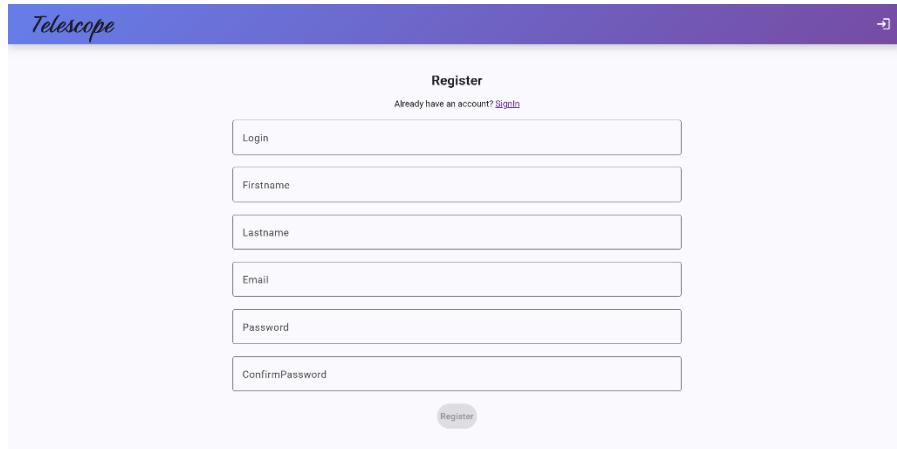


Рис 12. Окно регистрации нового пользователя.

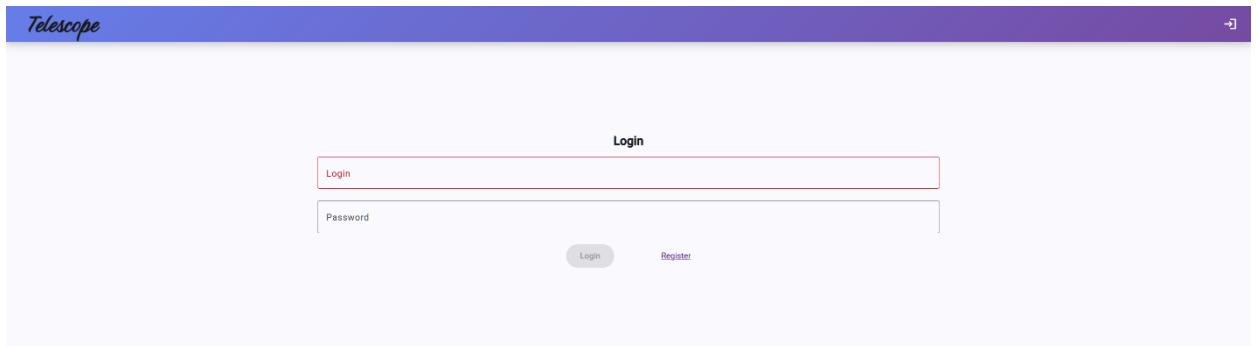


Рис 13. Окно авторизации.

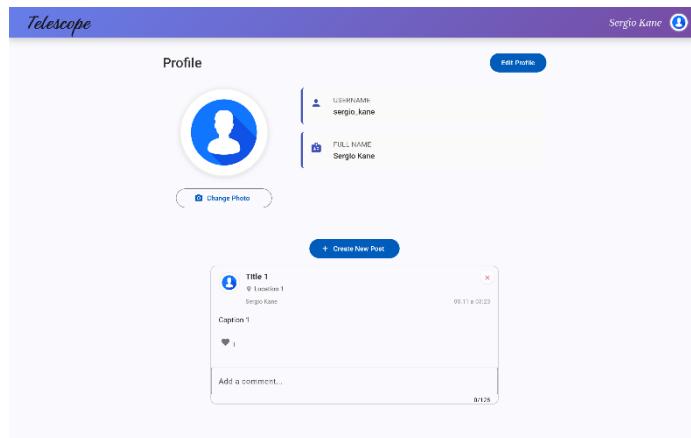


Рис 14. Профиль пользователя

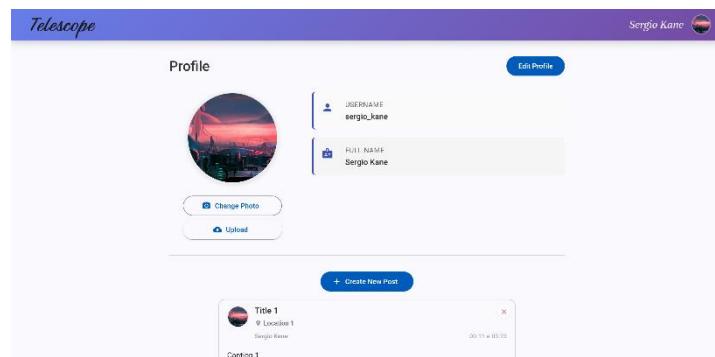


Рис 15. Профиль пользователя после добавления аватарки

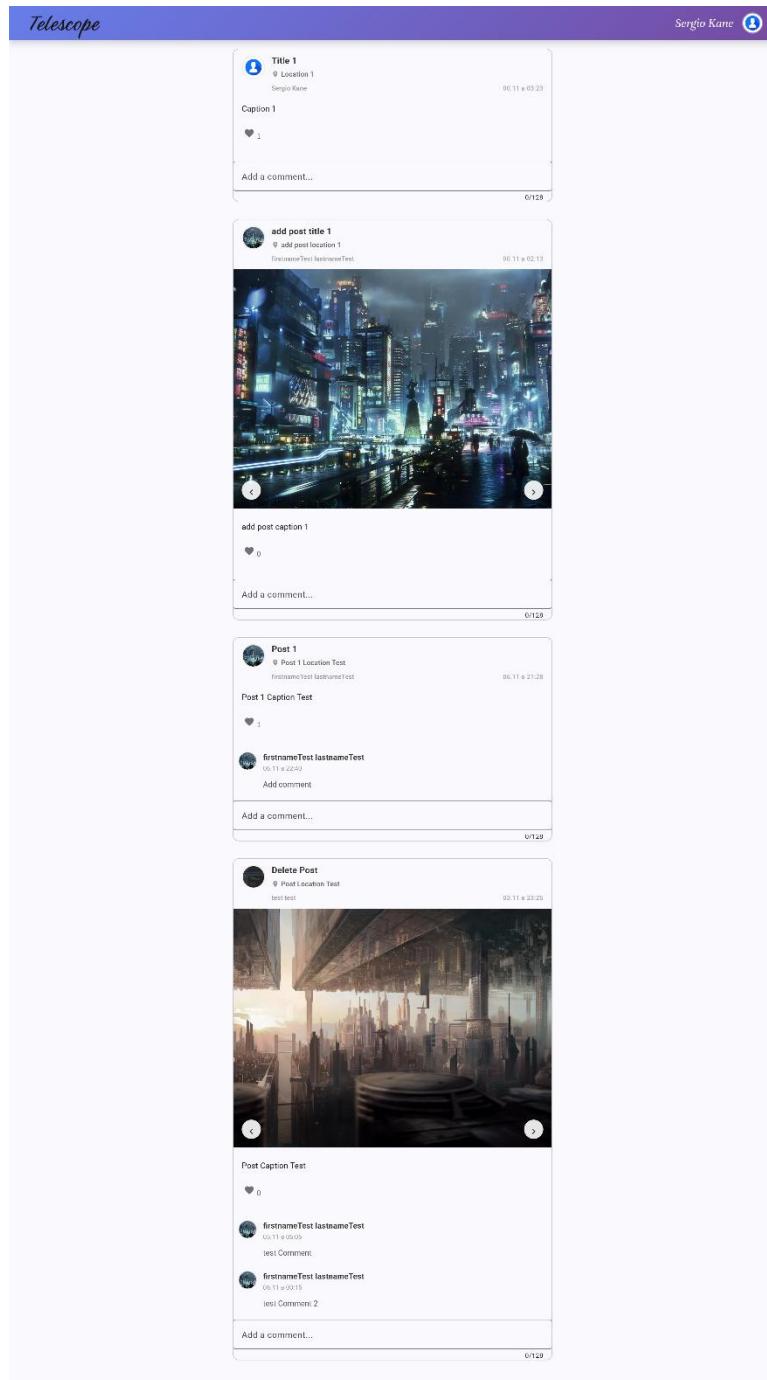


Рис 16. Основная лента постов.

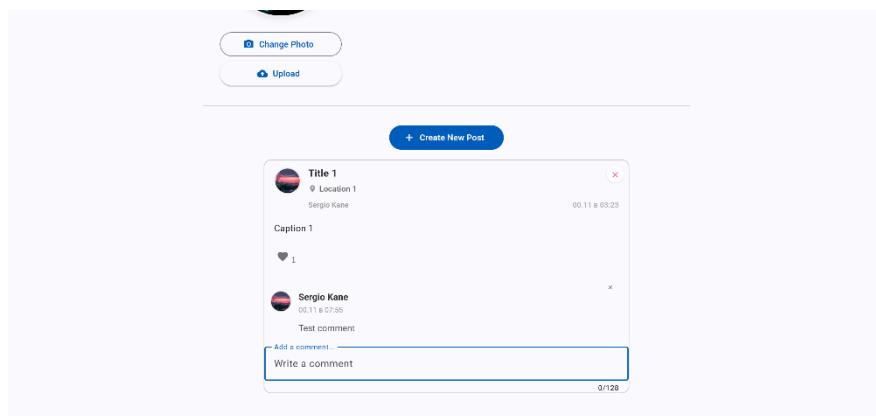


Рис 17. Добавление комментария к посту

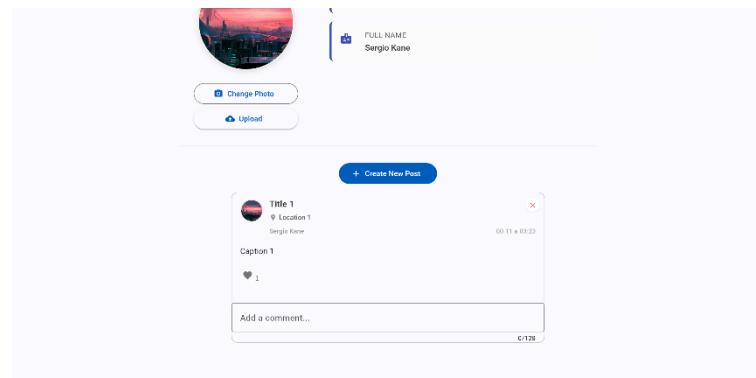


Рис 18. Удаление комментария из поста

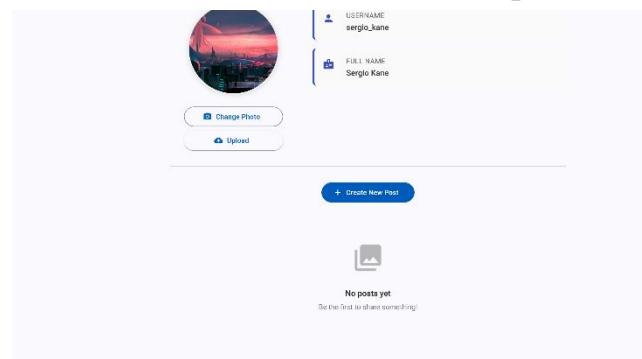


Рис 19. Удаление поста

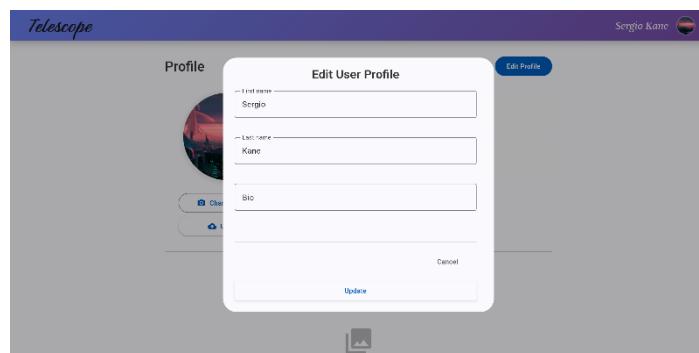


Рис 20. Форма редактирования профиля

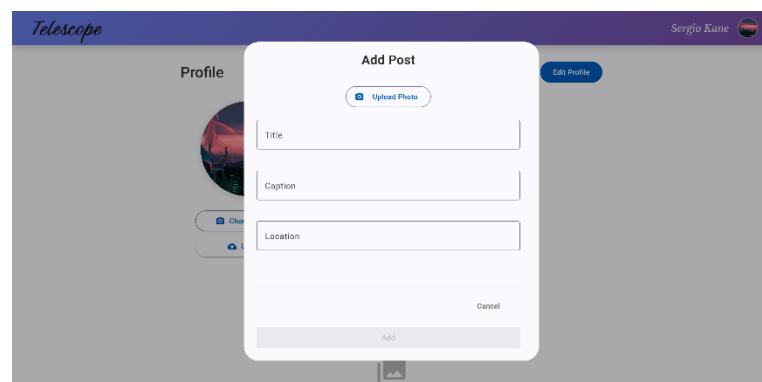


Рис 21. Форма добавления поста

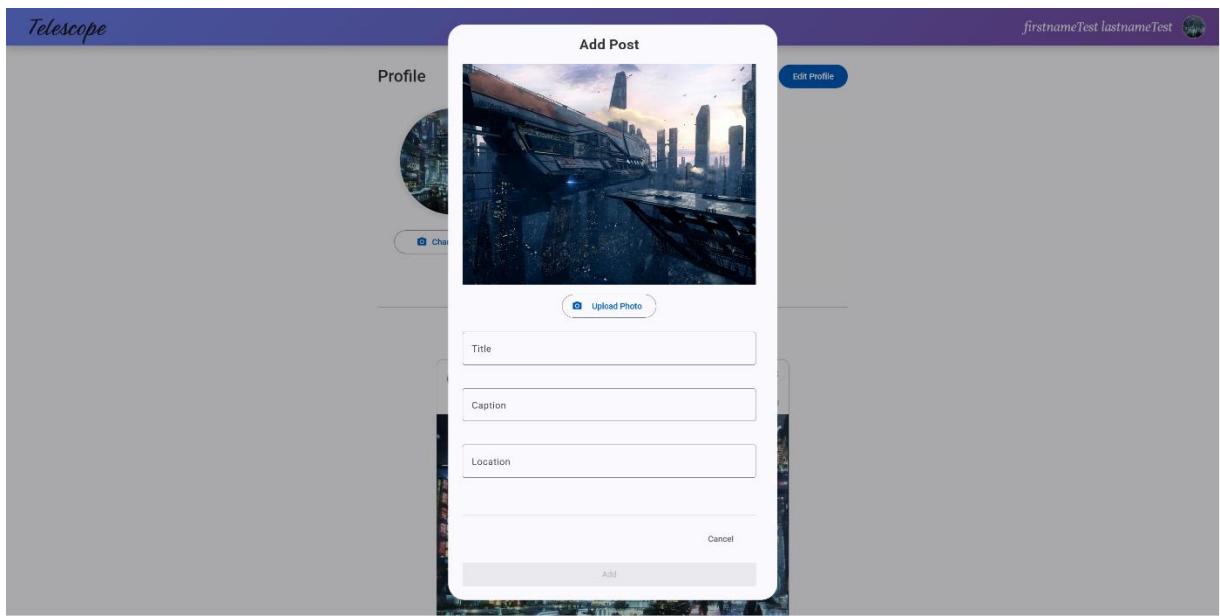


Рис 22. Форма добавления поста после добавления изображения

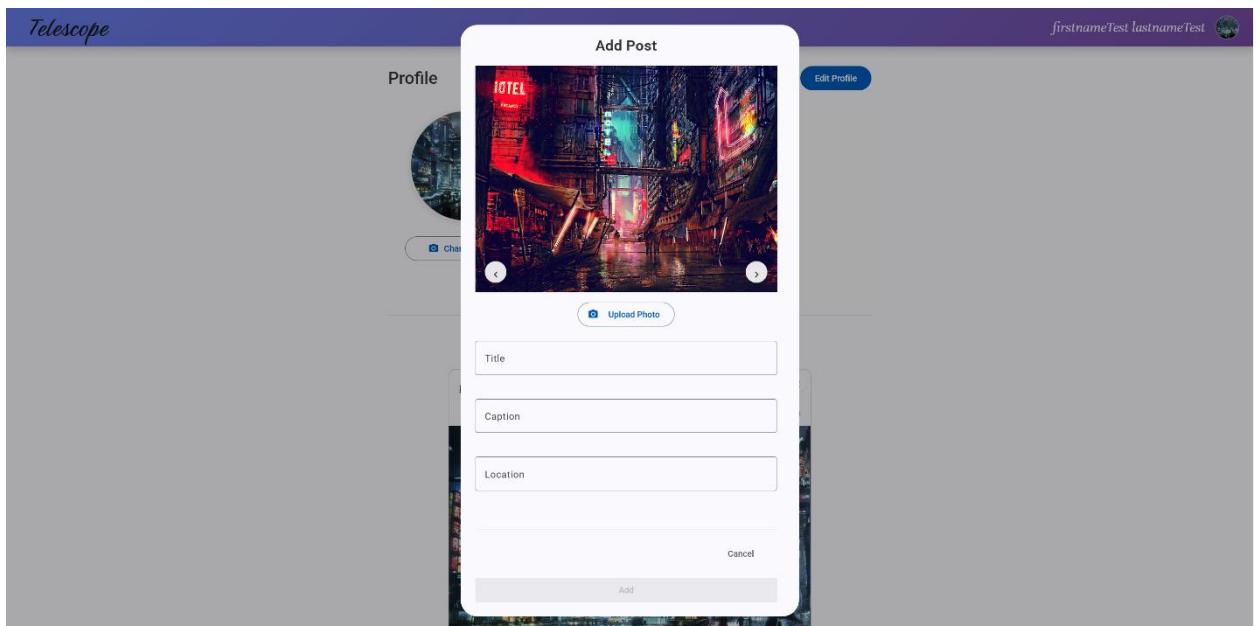


Рис 23. Форма добавления поста после добавления второго изображения

## ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы была достигнута поставленная цель: разработан и практически реализован полнофункциональный прототип микроблога.

Основные результаты и выводы по этапам проекта:

1) В ядре приложения была заложена нормализованная логическая модель базы данных в PostgreSQL, которая эффективно отражает связи между основными сущностями: пользователями, постами, комментариями, лайками и изображениями.

2) Реализация серверной части. На основе фреймворка Spring Boot был разработан RESTful API, предоставляющий клиенту полный набор операций для работы с данными.

3) Реализация клиентской части. С помощью фреймворка Angular создано современное одностраничное приложение (SPA) с интуитивно понятным и отзывчивым интерфейсом.

4) Приложение нуждается в развитии и доработке. В первую очередь, это касается работы с данными: механизмы загрузки и отображения постов и изображений реализованы в базовом варианте и не рассчитаны на высокие нагрузки. Ключевыми направлениями для доработки являются:

- Оптимизация взаимодействия с базой данных.

Существующие SQL-запросы, особенно в областях, связанных с загрузкой связанных сущностей (например, постов с комментариями и лайками), требуют пересмотра. Требуется устранить проблему N+1 запроса путём перехода на более эффективные методы выборки.

- Внедрение пагинации и оконного вывода. Вместо загрузки

всех записей единовременно необходимо реализовать постраничный вывод (pagination) как на серверной, так и на клиентской стороне. Это позволит разбивать большие наборы данных на «пачки» и подгружать их по мере необходимости (например, при скроллинге), что кардинально улучшит производительность фронтенда и снизит объем передаваемых данных.

## СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Словарь ИТ терминов [Электронный ресурс]:  
<https://blog.skillfactory.ru/> образовательный портал. – URL:  
<https://blog.skillfactory.ru/glossary/> (дата обращения 23.12.2025)
2. Чистая Архитектура для веб-приложений [Электронный ресурс]:  
<https://habr.com/> ИТ-форум – URL: <https://habr.com/ru/articles/493430/> - Дата публикации: 03.05.2020
3. Rethinking the Java DTO [Электронный ресурс]:  
<https://blog.scottlogic.com/> IT-блог – URL:  
<https://blog.scottlogic.com/2020/01/03/rethinking-the-java-dto.html> - Дата публикации: 03.12.2020
4. Robert C. Martin. Clean Architecture: A Craftsman's Guide to Software Structure and Design[текст]. – Pearson, 2017.-C 432.
5. Преимущества Java в веб-разработке[Электронный ресурс]:  
<https://astanahub.com/> ИТ-форум – URL:  
<https://astanahub.com/ru/blog/preimushchestva-java-v-veb-razrabotke> - Дата публикации: 23.08.2024
6. PostgreSQL: сравнение, возможности и экосистема [Электронный ресурс]: <https://bi-data.ru/> IT-блог – URL: <https://bi-data.ru/blog/2025/06/18/postgresql-сравнение-возможности-и-экосистем/> - Дата публикации: 18.06.2025
7. Введение в системы контроля версий [Электронный ресурс]:  
<https://htmlacademy.ru/> образовательный портал - URL:  
<https://htmlacademy.ru/blog/git/version-control-system> - Дата публикации: 03.08.2018
8. Vue vs React vs Angular: что выбрать для вашего проекта [Электронный ресурс]: <https://itanddigital.ru/> ИТ-форум – URL:  
<https://itanddigital.ru/bloghrconsulting/tpost/kl3u260yr1-vue-vs-react-vs-angular-chto-vibrat-dlya> (дата обращения 24.12.2025)
9. Подробно про JWT [Электронный ресурс]: <https://habr.com/> ИТ-форум - URL: <https://habr.com/ru/articles/842056/> - Дата публикации: 10.09.2024

## Приложение А:

Взаимодействие слоёв приложения при работе с сущностью User.

### 1. Сервер (Java).

#### 1.1. Entity – слой

```
@Data
@Entity
@Table(name = "users")
@NoArgsConstructor
@AllArgsConstructor
@ToString(of = {"id", "username", "firstname", "bio", "password", "email", "createdDate"})
@EqualsAndHashCode(of = {"id", "username", "firstname", "bio", "password", "email",
"createdDate"})
@Builder
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(updatable = false)
    private String username;

    private String firstname;
    private String lastname;

    @Column(updatable = false)
    private String email;

    private String bio;
    private String password;
    @Column(updatable = true)
    private String avatar;

    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", 
    timezone = "UTC")
    @Column(updatable = false)
    private LocalDateTime createdDate;

    @OneToMany(
        cascade = CascadeType.REFRESH, //ALL
        fetch = FetchType.EAGER,
        mappedBy = "user",
        orphanRemoval = true
    )
    @Builder.Default
    private List<Post> posts = new ArrayList<>();

    @PrePersist
    private void onCreate() {
        this.createdDate = LocalDateTime.now();
    }

    public void addPost(Post post) {
        this.posts.add(post);
        post.setUser(this);
    }
    public void removePost(Post post) {
        this.posts.remove(post);
    }
}
```

#### 1.2. Repository-слой

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);

    default User getUserByPrincipal(Principal principal){
        String username = principal.getName();
        return findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
    }

    Optional<User> getUserId(Long id);
}
```

### 1.3. DTO

```
@Data  
@Builder  
@NoArgsConstructor  
@AllArgsConstructor  
public class UserShowNameDTO {  
    private Long id;  
    private String username;  
    private String firstname;  
    private String lastname;  
}  
  
@Builder  
@Data  
public class UserEditDTO {  
    private Long id;  
    private String firstname;  
    private String lastname;  
    private String bio;  
}  
  
@Data  
@Builder  
@NoArgsConstructor  
@AllArgsConstructor  
public class UserProfileDTO {  
    Long id;  
    String username;  
    String email;  
    String firstname;  
    String lastname;  
    String bio;  
}
```

### 1.4. UserMapper

```
@Component  
@RequiredArgsConstructor  
public class UserMapper implements Mapper<UserEditDTO, User> {  
    @Override  
    public User map(UserEditDTO from) {  
        return User.builder()  
            .id(from.getId())  
            .firstname(from.getFirstname())  
            .lastname(from.getLastname())  
            .bio(from.getBio())  
            .build();  
    }  
  
    public User map(UserEditDTO from, User to){  
        to.setBio(Optional.ofNullable(from.getBio()).orElse(to.getBio()));  
        to.setFirstname(Optional.ofNullable(from.getFirstname()).orElse(to.getFirstname()));  
        to.setLastname(Optional.ofNullable(from.getLastname()).orElse(to.getLastname()));  
  
        return to;  
    }  
}
```

### 1.5. Service-слой

```
@Slf4j  
@Service  
@RequiredArgsConstructor  
@Transactional(readOnly = true)  
public class UserService {  
    private static final String DEFAULT_AVATAR = "user/default/307ce493-b254-4b2d-8ba4-d12c080d6651.png";  
  
    private final UserRepository userRepository;  
    private final UserMapper userMapper;  
    private final BCryptPasswordEncoder bCryptPasswordEncoder;  
  
    @Transactional  
    public User createUser(SignupRequest userIn){  
        var user = User.builder()  
            .firstname(userIn.getFirstname())  
            .lastname(userIn.getLastname())  
            .email(userIn.getEmail())  
            .username(userIn.getUsername())  
            .password(bCryptPasswordEncoder.encode(userIn.getPassword()))  
            .build();  
    }
```

```

        try {
            log.info("Creating user {}", userIn.getUsername());
            return userRepository.save(user);
        } catch (Exception e) {
            log.error("Error creating user {}", userIn.getUsername(), e);
            throw new UserExistException("User %s already exists. Please check credentials".formatted(user.getUsername()));
        }
    }

    @Transactional
    public Optional<User> updateUser(UserEditDTO userDTO, Principal principal) {
        return Optional.of(userRepository.getUserByPrincipal(principal))
            .map(u->userMapper.map(userDTO, u))
            .map(userRepository::save);
    }

    public User getCurrentUser(Principal principal) {
        return userRepository.getUserByPrincipal(principal);
    }

    public Optional<User> getUserId(Long userId) {
        return userRepository.getUserId(userId);
    }

    @Transactional
    public void uploadAvatar(MultipartFile file, Principal principal) {
        User user = userRepository.getUserByPrincipal(principal);
        var pref = ImagePrefix.USER.toString();
        if (!ObjectUtils.isEmpty(user.getAvatar())) {
            delete(user.getAvatar());
        }
        var imgPath = saveImage(file, user.getId(), pref);
        user.setAvatar(imgPath);
        userRepository.save(user);
    }

    public Optional<String> getAvatar(Principal principal) {
        return Optional.of(userRepository.getUserByPrincipal(principal).getAvatar());
    }

    public Optional<String> getAvatarById(Long userId) {
        return userRepository.getUserId(userId).map(User::getAvatar);
    }

    public String getDefaultAvatar() {
        return DEFAULT_AVATAR;
    }
}

```

## 1.6. Controller слой

```

@RestController
@RequestMapping("api/user")
@CrossOrigin
@RequiredArgsConstructor
public class UserController {
    private final UserService userService;
    private final PostService postService;

    private final UserShowNameMapper userShowNameMapper;
    private final UserProfileMapper userProfileMapper;
    private final UserMapper userMapper;
    private final ResponseErrorValidation responseErrorValidation;

    @GetMapping("/")
    public ResponseEntity<UserShowNameDTO> getCurrentUser(Principal principal) {
        var userDTO = Optional.of(userService.getCurrentUser(principal))
            .map(userShowNameMapper::map)
            .orElseThrow(()->new UsernameNotFoundException("User not found"));
        return ResponseEntity.ok(userDTO);
    }

    @PutMapping("/update")
    public ResponseEntity<Object> updateUser(@RequestBody UserEditDTO userDTO,
                                              BindingResult bindingResult,
                                              Principal principal){
        ResponseEntity<Object> errors =
        responseErrorValidation.mapValidationService(bindingResult);
        if(!ObjectUtils.isEmpty(errors)) return errors;
    }
}

```

```

        var userUpd = userService.updateUser(userDTO, principal)
            .map(userShowNameMapper::map)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
        return ResponseEntity.ok(userUpd);
    }

    @GetMapping("/profile")
    public ResponseEntity<UserProfileDTO> getCurrentUserProfile(Principal principal) {
        var user = Optional.of(userService.getCurrentUser(principal))
            .map(userProfileMapper::map)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
        return ResponseEntity.ok(user);
    }

    @GetMapping("/like/{postId}")
    public ResponseEntity<List<Long>> likePost(@PathVariable Long postId,
                                                Principal principal) {
        var user = userService.getCurrentUser(principal);
        var postDTO = postService.likePost(postId, user.getId());
        return ResponseEntity.ok(postDTO);
    }

    @GetMapping("/image")
    public ResponseEntity<byte[]> getUserImage(Principal principal) {
        var avatar = Optional.ofNullable(userService.getCurrentUser(principal).getAvatar())
            .orElse(userService.getDefaultAvatar());
        return processAvatar(avatar);
    }

    @GetMapping("/image/{userId}")
    public ResponseEntity<byte[]> getAvatarByUserId(@PathVariable Long userId) {
        var avatar = userService.getAvatarByUserId(userId)
            .orElse(userService.getDefaultAvatar());
        return processAvatar(avatar);
    }

    public ResponseEntity<byte[]> processAvatar(String avatar) {
        return Optional.of(avatar)
            .flatMap(ImageUploadService::get)
            .map(img -> ResponseEntity.ok()
                .contentType(MediaType.IMAGE_PNG)
                .body(img))
            .orElse(ResponseEntity.ok(null));
    }

    @PutMapping("/image")
    public ResponseEntity<MessageResponse> uploadImage(@RequestParam MultipartFile file,
                                                       Principal principal) {
        userService.uploadAvatar(file, principal);
        return ResponseEntity.ok(new MessageResponse("Message uploaded successfully"));
    }
}

```

## 2. Клиент (TypeScript)

### 2.1. DTO

```

export interface UserShowNameDto{
    id:number;
    username:string;
    firstname:string;
    lastname: string;
}

export interface UserProfileDto{
    id:number;
    username:string;
    email:string;
    firstname:string;
    lastname:string;
    bio?:string;
}

export interface UserEditDto{
    id:number;
    firstname:string;
    lastname:string;
    bio?:string;
}

```

## 2.2. Service

```
@Injectable({
  providedIn: 'root',
})
export class UserService{
  private readonly http = inject(HttpClient);
  private readonly userAPI = new EndpointBuilder('user');

  getCurrentUser(): Observable<UserShowNameDto>{
    return this.http.get<UserShowNameDto>(this.userAPI.build());
  }

  updateUser(user: UserEditDto): Observable<UserShowNameDto>{
    return this.http.put<UserShowNameDto>(this.userAPI.build("update"), user)
  }

  getCurrentUserProfile(): Observable<UserProfileDto>{
    return this.http.get<UserProfileDto>(this.userAPI.build("profile"))
  }

  likePost(postId: number): Observable<number[]>{
    return this.http.get<number[]>(this.userAPI.build("like", String(postId)))
  }

  getAvatar(): Observable<Blob>{
    return this.http.get(this.userAPI.build("image"), {
      responseType: 'blob'
    });
  }

  postAvatar(file: Blob): Observable<any>{
    const uploadData = new FormData();
    uploadData.append('file', file);
    return this.http.put(this.userAPI.build("image"), uploadData)
  }

  getAvatarByUserId(userId: number): Observable<Blob>{
    return this.http.get(this.userAPI.build("image", String(userId)), {
      responseType: 'blob'
    });
  }
}
```