

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**«Дальневосточный федеральный университет»**

**ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ**

**Департамент информационных компьютерных систем**

**КУРСОВАЯ РАБОТА**

по дисциплине

«Технологии программирования информационных систем»

на тему

«Проектирование серверной части функционала гибридной платформы для публикации и монетизации творческого контента с элементами социальной сети»

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Курсовая работа защищена:  с оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  подпись И. О. Фамилия  «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.  число месяц  Регистрационный № \_\_\_\_\_\_  «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.  число месяц  \_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  подпись И. О. Фамилия |  | Выполнил студент  гр. Б9123-09.03.02прс  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /Спирин Н. В./  подпись  Руководитель курсовой работы старший преподаватель  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /**Полищук С. В./  подпись |

Владивосток

2025

Оглавление

[Введение 3](#_Toc122709447)

[1 Аналитический обзор современных платформ для создателей контента 5](#_Toc122709448)

[2 Основные требования и функциональные блоки разрабатываемой системы 7](#_Toc122709449)

[2.1 Архитектурный подход разработки 8](#_Toc122709450)

[2.2 Выбор технологического стека и логика архитектуры 9](#_Toc122709451)

[3 Проектирование и реализация 12](#_Toc122709452)

[3.1 Архитектура функционала серверной части 12](#_Toc122709453)

[3.2 Структура серверной части 14](#_Toc122709454)

[3.3 Реализация ключевых маршрутов и бизнес-логики 18](#_Toc122709454)

[3.3.1 Регистрации пользователя 19](#_Toc122709454)

[3.3.2 Хеширование данных 19](#_Toc122709454)

[3.3.3 Реализация взаимодействи с базой данных 20](#_Toc122709454)

[Заключение 23](#_Toc122709455)

[Список источников 25](#_Toc122709456)

[Приложение А 26](#_Toc122709457)

Введение

В настоящее время современное общество невозможно представить без активного использования цифровых технологий. Особое значение в этой цифровой среде приобретает творческий контент: музыка, графика, литература, видео, игровые и мультимедийные проекты. Миллионы пользователей по всему миру создают, публикуют и распространяют свои работы, стремясь не только к самовыражению, но и к получению обратной связи, признания и материального вознаграждения.

Однако существующие платформы, как правило, ориентированы либо на размещение контента (хостинг изображений, видео или любого творческого контента), либо на монетизацию (донат-сервисы, NFT-маркеты), либо на социальное взаимодействие (социальные сети общего назначения). Это приводит к разделению цифровой среды, в которой создателю контента приходится использовать сразу несколько независимых ресурсов для охвата всех этапов жизненного цикла контента: от создания и публикации до продвижения и получения прибыли. Это не только усложняет работу авторов, но и мешает построению устойчивого творческого сообщества.

В связи с этим возрастает потребность в комплексных, гибридных решениях, сочетающих в себе функции социальной сети, платформы для публикации и эффективного инструмента монетизации. Такие инструменты должны предоставлять пользователям возможность не только делиться результатами своего труда, но и выстраивать вокруг них аудиторию, получать отзывы, строить бренд, а также зарабатывать, не покидая пределы одной экосистемы. Особенно актуально это становится в условиях децентрализации медиа и роста числа независимых авторов.

Кроме того, важную роль играет удобство интерфейса, кроссплатформенность, безопасность и масштабируемость подобных решений. Необходимо учитывать потребности различных категорий пользователей – как начинающих авторов, так и профессионалов, заинтересованных в автоматизации процессов, глубокой аналитике и интеграции с другими сервисами.

Таким образом, в рамках данной курсовой работы ставится цель – **проектирование серверной части функционала** **гибридной платформы для публикации и монетизации творческого контента с элементами социальной сети**, которая объединяет в себе возможности для размещения мультимедийных материалов, взаимодействия между пользователями и получения дохода за счет встроенных механизмов поддержки и распространения.

Для достижения указанной цели в работе необходимо решить следующие задачи:

* Провести аналитический обзор существующих решений в сфере цифровых платформ для авторов контента;
* Определить ключевые требования и функциональные блоки разрабатываемой системы;
* Спроектировать архитектуру платформы с учетом принципов модульности, масштабируемости и безопасности;
* Реализовать прототип платформы, включающий основные модули: публикации, социального взаимодействия и монетизации;
* Протестировать работу платформы на предмет удобства, производительности и надежности;
* Провести оценку перспектив развития платформы и ее возможной интеграции с другими сервисами.

Актуальность выбранной темы обусловлена растущей цифровизацией творческой деятельности, необходимостью создания удобных инструментов для авторов и спросом на унифицированные решения, способные заменить разрозненные сервисы. Разработка такой платформы представляет собой интересную задачу, требующую комплексного подхода к проектированию и реализации современных веб-систем.

1 Аналитический обзор современных платформ для создателей контента

В последние годы наблюдается стремительное развитие цифровых площадок, позволяющих авторам публиковать и монетизировать свое творчество. Среди наиболее известных можно выделить платформы вроде Patreon, Boosty, Behance, SoundCloud, YouTube, DeviantArt и TikTok. Каждая из них предлагает специфический набор функций и возможностей, однако ни одна из них не сочетает в себе полной мере все необходимые компоненты: инструменты публикаций, функции социальной сети для обеспечения обратной связи с пользователями и развитую систему монетизации.

* Patreon и Boosty представляют авторам возможность получить регулярную финансовую поддержку от подписчиков. Однако эти платформы слабо развиты с точки зрения социальной составляющей и продвижения контента, также эти платформы обладают большой комиссией без региональных особенностей и разделения на крупных и начинающих создателей контента.
* Behance, DeviantArt и SoundCloud являются платформами для размещения и демонстрации работ, но не представляют автору гибкой системы монетизации и ограничивают тип контента публикуемого автором.
* YouTube и TikTok совмещают элементы социальной сети и монетизации, но они ориентированы прежде всего на видеоформаты и имеют высокую конкуренцию и требования к алгоритмическому продвижению.

Таким образом, на рынке ощущается дефицит платформы, которая в равной степени предоставляла бы функции для публикации, социализации и монетизации контента вне зависимости от его формата. Это подтверждает актуальность идеи создания гибридной системы, способной закрыть потребность как начинающих, так и профессиональных авторов.

Однако стоит учитывать, что создание многофункциональной и гибридной платформы сопряженно с риском перегрузки интерфейса и снижения удобства для пользователей. Опыт экспериментальных проектов, таких как попытка VK превратить свою социальную сеть в универсальную площадку со множеством функций, показал, что излишняя сложность и перегруженность системы могут отпугнуть пользователей и снизить эффективность взаимодействия с контентом. В то же время компактные и лаконичные решения, например, Telegram, остаются популярными благодаря удобству и быстроте использования несмотря на то, что они не обладают встроенными инструментами продвижения и монетизации контента внутри платформы. Поэтому при разработке гибридной платформы для творцов критически важно найти оптимальный баланс между функциональностью и эргономикой, обеспечив компактный, интуитивный и не перегруженный интерфейс, который позволит пользователям эффективно взаимодействовать друг с другом.

2 Основные требования и функциональные блоки разрабатываемой системы

При проектировании гибридной цифровой платформы для творчества авторов необходимо определить ключевые требования, которые обеспечат удобство, стабильность и эффективность работы системы. Платформа должна удовлетворять запросы как начинающих, так и профессиональных пользователей, представляя возможности для публикации, взаимодействия и монетизации контента в одном интегрированном пространстве.

**К числу важнейших требований относятся:**

* **Удобство и простота использования**. Интерфейс должен быть интуитивно понятным и не создавать препятствий для пользователей. Это позволит максимально расширить аудиторию и снизить порог входа.
* **Разнообразие инструментов публикации**. Платформа должна поддерживать различные форматы контента – текст, изображения, аудио, видео, а также гибко управлять параметрами публикации (например, приватность, доступ к подписке, ограничения по просмотрам)
* **Развитая социальная составляющая**. Важной функцией является возможность создавать сообщества, подписываться на автора, комментировать и взаимодействовать с контентом, что способствует формированию активного пользовательского сообщества.
* **Механизация монетизации**. Платформа должна представлять разнообразные способы заработка для авторов – подписки, донаты, продажу контента, а также аналитику доходов. При этом платежные инструменты должны быть надежными и безопасными.
* **Система рекомендаций и продвижения**. Эффективные алгоритмы подбора и продвижения контента помогут авторам найти свою аудиторию, а пользователям – легко находить интересующий материал.
* **Высокая производительность и масштабируемость**. Платформа должна обеспечивать стабильную работу при увеличении числа пользователей и объема контента, а также поддерживать быстрый отклик интерфейса.
* **Безопасность и конфиденциальность**. Защита персональных данных и предотвращение мошенничества – обязательные требования к системе.

В ходе архитектуры платформы выделяются основные функциональные блоки: модуль управления контентом и публикациями, социальные модули для взаимодействия пользователей, платежный модуль, аналитический блок, а также административная панель для контроля и управления системой. Каждый из этих блоков должен быть разработан с учетом высокой интеграции, но при этом сохранять модульность для упрощения поддержки и развития платформы.

2.1 Архитектурный подход разработки

Реализация платформы будет старится на модульной архитектуре [[1]](https://www.hackfrontend.com/docs/architecture/module): каждый функциональный блок – публикация, профили, монетизация, социальные функции – разрабатывается как самостоятельный компонент с чёткими интерфейсами взаимодействия. Это обеспечит гибкость, масштабируемость и упрощенную поддержку в бедующем.

В приоритете – последовательная разработка: сначала – базовая функциональность (аутентификация, публикация, простые формы взаимодействия), затем – аналитика, рекомендации и расширенные социальные модули [[2]](https://habr.com/ru/articles/458336/). Такой подход позволяет быстро запустить жизнеспособный продукт, протестировать пользовательские сценарии и поэтапно расширять возможности платформы.

2.2 Выбор технологического стека и логика архитектуры

Для реализации гибридной платформы, объединяющей функции публикации, социализации и монетизации, важно выбрать надёжный и масштабируемый технологический стек, который позволит обеспечивать стабильную работу системы, быструю разработку новых функций и комфортный пользовательский опыт.

Фронтенд платформы будут разрабатываться с использованием библиотек **ReactJS** в связке с **TypeScript**, что обеспечит компонентный подход и строгую типизацию. Это позволит быстро масштабировать проект, уменьшить количество ошибок и упростить поддержку кода. Основой визуального оформления станет **Material UI** – современная библиотека компонентов, следящая за UX/UI-стандартами и поддерживающая адаптивность, что критически важно для кроссплатформенности [[3]](https://javarush.com/quests/lectures/ru.javarush.web.fullstack.lecture.level06.lecture04).

Управление состоянием приложения будет построено с использованием **Redux** и **Redux Saga**, что обеспечит централизированный и предсказуемый поток данных, а также упростит работу с асинхронными запросами. Для оптимизации производительности подключается **Reselect**, а иммутабельность работы с данными обеспечивается через **Immer**. Работа с формами будет организована через **Formik**, а взаимодействие с **REST API** – с помощью **Axios**. Библиотека **date-fns** облегчит управление временными метками и календарными функциями в интерфейсе публикаций, подписок и аналитики.

Бэкенд платформы будут реализованы на **NodeJS** с использованием фреймворка **Express**, который обеспечит маршрутизацию и быструю обработку API-запросов. Робота с базой данных **MongoDB** будет осуществляться через **Mongoose**, что позволит безопасно управлять схемами данных и взаимодействием с коллекциями. Для валидации входящих данных используется **Express Validator** [**[4]**](https://express-validator.github.io/docs/).

Аутентификация пользователей реализуется через связку **Passport** и **JWT**, что позволит безопасно управлять сессиями, а также реализовать гибкую систему доступа. Загрузка медиафайлов (изображения, видео, аудио) будет происходить через **Multer**, а хранение – на облачной платформе **Cloudinary**, которая дополнительно поддерживает адаптивную оптимизацию контента. Для связи с пользователями по email будет использоваться **Nodemailer** – он обеспечит подтверждение аккаунтов, уведомления и оповещения.

Таким образом весь стек будет выглядеть следующим образом:

**Frontend:**

* TypeScript
* ReactJS
* Redux / Redux Saga
* Immer
* Reselect
* React Router
* Material UI
* Axios
* Date-fns
* Formik

**Backend:**

* TypeScript
* NodeJS
* Express / express Validator
* Mongoose
* Passport
* JWT
* Multer
* Cloudinary
* Nodemailer

Такой стек позволит выстроить надёжную архитектуру с акцентом на модульность, отказоустойчивость и масштабируемость, а также создать прочный фундамент для расширения функциональности платформы в будущем.

3 Проектирование и реализация

В рамках данной работы основное внимание уделяется серверной части разработки, включая проектирование логики обработки данных, реализацию API и работу с базами данных. Хотя фронтенд часть проекта была рассмотрена в предыдущем разделе для более полного представления архитектуры системы, она не является основной целью данной разработки.

В этом разделе будут подробно рассмотрены этапы создания бекенд части платформы, включая проектирование моделей данных, настройку серверной логики на **Node.js** и **Express**, а также реализацию системы аутентификации, валидации, загрузки медиафайлов и отправки уведомлений.

3.1 Архитектура функционала серверной части

Курсовая работа посвящена разработке серверной части платформы для публикации и взаимодействия с авторским контентом. В данном разделе описана архитектура основного функционала с точки зрения **REST API** и взаимодействия клиента с сервером.

Архитектура построена по принципу **REST** [**[5]**](https://blog.skillfactory.ru/glossary/rest-api/) и включает маршруты, которые реализуют бизнес-логику системы: регистрацию пользователя, создание постов, оценку контента, управление профилем и отправку донатов. Сервер принимает HTTP-запросы и обрабатывает их соответствующими контроллерами и сервисами.

**Основные типы запросов:**

* **GET** – получение данных (лента постов, информация о пользователе);
* **POST** – создание новых сущностей (регистрация, пост, донат, лайк);
* **PATCH** – частичное обновление информации (редактирование поста или профиля);
* **DELETE** – удаление сущностей (собственный пост)

**Разделение пользователей:**

* **Гости (неавторизованные пользователи)** могут просматривать посты и искать авторов.
* **Зарегистрированные пользователи** получают доступ к расширенному функционалу: они могут публиковать посты, редактировать и удалять их, загружать файлы, подтверждать email, отправлять лайки, репосты, донаты.

**Примеры запросов:**

* **GET /post** – загрузка ленты постов;
* **POST /posts/:id/like** – лайк поста, где id: – id пользователя;
* **PATCH /users/:id** – обновление профиля.

Таким образом, каждый элемент функционала представлен отдельным маршрутом API и реализован через соответствующий модуль серверной части. Такое распределение позволяет легко масштабировать и поддерживать проект. Полная архитектура представлена в следующей схеме (Рисунок 1).

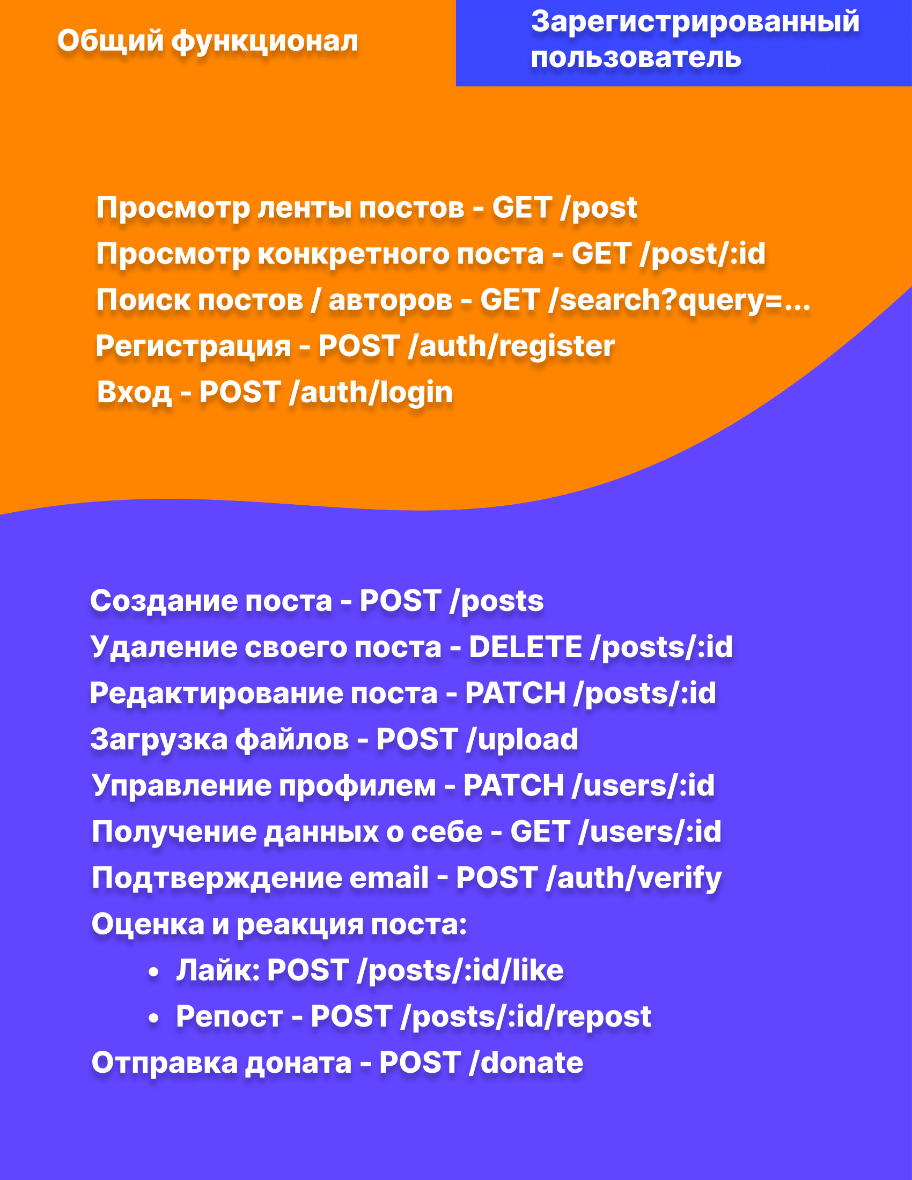


Рисунок 1 – Схема функционала серверной части

3.2 Структура серверной части

Проект реализован по модульной архитектуре, с чётким разделением логики на уровни: контроллеры, модели, вспомогательные модули и сервисы. Главный акцент сделан на серверную часть, которая обеспечивает **API-интерфейс**, аутентификацию, работу с базой данных **MongoDB** и внешними сервисами (облачное хранилище и email-рассылка)

Серверная часть платформы реализована на **Node.js** с использованием фреймворка **Express**. Архитектура проекта построена по модульному принципу и разделена на логические компоненты, каждый из которых отвечает за определенный участок функциональности: обработку маршрутов, работу с базой данных, валидацию, загрузку файлов, авторизацию и отправку уведомлений.

На схеме (Рисунок 2) показана архитектура серверной части проекта с пояснением ключевых связей между файлами и модулями [[6]](https://ru.hexlet.io/courses/internet-fundamentals/lessons/client-server/theory_unit). Далее приведено подробное текстовое описание её компонентов.

**Общие компоненты серверной архитектуры:**

* **Контроллеры** – обрабатывают группы маршрутов, вызывают логику бизнес-процессов.
* **Модели данных** – описывают структуру коллекций MongoDB.
* **Утилиты** – служебные функции (хеширование пароля, валидация ID).
* **Сторонние сервисы** – Cloudinary (для загрузки файлов) и SendGrid (для отправки email).

Основной файл **server.ts** – точка входа в приложение. В этом файле инициализируются все маршруты, middleware и подключаются необходимые модули. Здесь происходит подключение контроллеров, маршрутов, базы данных и вспомогательных функций.

**Контроллеры:**

**UserContoller** – основной контроллер для работы с пользователями. Он обрабатывает входящие запросы, связанные с регистрацией, авторизацией и управлением пользовательскими данными. Подключает функции:

* Валидация регистрационных данных (**register.ts**),
* Хеширование паролей (**generateHash.ts**),
* Аутентификации с использованием JWT (**passport.ts**),
* Отправки уведомлений на email (**mailer.ts, SendGrid**),
* Обращения к моделям пользователя (**UserModel**),
* Взаимодействия с базой данных (**db.ts, MongoDB**).

**UploadFileController** – контроллер для обработки загрузки файлов, который обеспечивает:

* Загрузку изображений на облачное хранилище (**cloudinary.ts, Cloudinary**),
* Валидацию содержимого постов (**createPost.ts**),
* Передачу информации о посте в базу данных (**PostModel, db.ts**).

**Модели:**

Модели реализованы с использованием библиотек **Mongose**, и каждая отражает структуру определенной сущности в базе данных.

* **UserModel** – определяет схему пользователя: логин, email, хеш пароля, дата регистрации, статус верификации.
* **PostModel** – описывает структуру поста: автора, текстовое описание, прикрепленные изображения, дата публикации и т.д.

**Работа с базой данных:**

* **db.ts** – файл конфигурации и инициализации подключения к **MongoDB**. Используется в контроллерах и сервисах для выполнения CRUD-операций.
* **MongoDB** – облачная база данных, в которой хранятся все сущности проекта: пользователи, посты, донаты и т.д.

**Аутентификация и безопасность:**

* **passport.ts** – файл настройки стратегии авторизации. Включает использование JWT для авторизации пользователей. Настраивается и подключается в основном файле server.ts.
* **OAuth / JWT** – механизм авторизации и проверка токенов, гарантирующий доступ к защищенным маршрутам только для авторизованных пользователей.

**Отправка уведомлений:**

* **mailer.ts** – модуль для отправки писем, подключаемый в контроллерах.
* **SendGrif** – сторонний сервис, используемый для доставки email-уведомлений пользователям (подтверждение регистрации, уведомления о событиях).
* **sendEmail.ts** – вспомогательная функция, обёртка над mailer.ts для унифицированного вызова.

**Загрузка медиафайлов:**

* **cloudinary.ts** – модуль для загрузки файлов в облачное хранилище Cloudinary.
* Используется в **UploadFileController** при публикации постов с изображением

**Валидация данных:**

* **Register.ts** – middleware-функция валидации данных при регистрации (проверка длины логина, формат email, соответствия пароля и его подтверждения).
* **createPost.ts** – проверяет валидность полей при создании поста.
* **isValidObjectId.ts** – утилита, проверяющая корректность **Mongo OjectId** перед выполнением запросов к базе данных.

**Утилиты:**

* **generateHash.ts** – используется для хеширования паролей с помощью **bcrypt** перед их сохранением в базе.
* **isValidObjectId.ts** – проверка идентификаторов на корректность (используется в маршрутах, принимающих **id**).
* **sendEmail.ts** – модульная реализация отправки email через **mailer.ts**.

**Конфигурационные файлы:**

* **passport.ts** – настройка JWT-авторизации.
* **db.ts** – подключение к базе данных **MongoDB**.
* **mailer.ts** – подключение и конфигурация отправки почты через **SendGrid**.
* **Cloudinary.ts** – конфигурация доступа к облачному хранилищу **Cloudinary**.

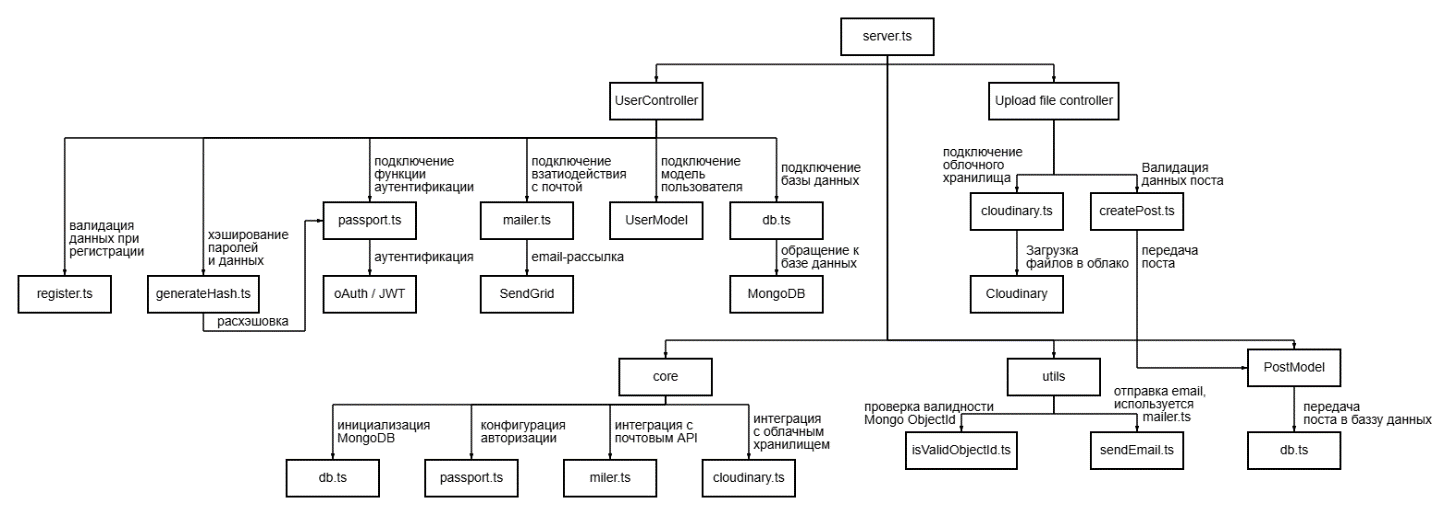


Рисунок 2 – Схема функционала серверной части

Таким образом, структура проекта обеспечивает четкое разделение ответственности между модулями, благодаря чему упрощается тестирование, повторное использование компонентов и дальнейшая поддержка проекта. Каждый функциональный блок (аутентификация, работа с данными, медиа, уведомления) реализован изолированно и связан с другими через четко определенные интерфейсы.

3.3 Реализация ключевых маршрутов и логики проекта

В этом разделе подробно рассматривается реализация основных маршрутов серверной части платформы, а также логика, лежащая в их основе. Каждый маршрут отвечает за конкретный функциональный блок-системы: от регистрации и авторизации пользователя до взаимодействия с контентом, профилем и системой донатов.

Для обработки запросов используются контроллеры, валидаторы, модели данных и вспомогательные сервисы, объединенные в модульную структуру проекта. Такой подход обеспечит гибкость, читаемость и возможность масштабирования проекта без переработки всей архитектуры.

Подпункты ниже раскрывают, как реализованы конкретные функции на уровне **API**, включая структуру маршрутов, валидацию данных, бизнес-логику и взаимодействие с базой данных **MongoDB** через библиотеку Mongoose.

3.3.1 Регистрация пользователя

Регистрация пользователя реализована с применением middleware-валидации [[7]](https://vc.ru/dev/1546651-middleware-v-nodejs-express) на стороне сервера, обеспечивающей первичную проверку корректности входных данных до их обработки. На данном этапе пользователь предоставляет email, имя, логин, пароль и подтверждение пароля. После отправляется email-подтверждение и ответ сервера клиенту о результате.

Валидация данных:

Для обработки запроса регистрации используется модуль **express-validator**, который предоставляет цепочку проверок, определённых в файле **register.ts**. Он экспортирует массив r**egisterValidations**, содержащий последовательные правила для каждого поля формы регистрации.

Валидация включает в себя следующие аспекты: обязательное поле, валидный адрес электронной почты, длина строки, а для пароля костюмная проверка на совпадение с паролем password2, содержащим подтверждение пароля. Полный код можно посмотреть в приложении А.

3.3.2 Хеширование данных

Для безопасного хранения пользовательских данных и паролей в системе используется механизм хеширования, реализованный в модуле **generateHash.ts**. Основной задачей данного этапа является защита данных и паролей от компрометации в случае утечки базы данных.

**Генерация хеша:**

В модуле используется встроенная библиотека crypto из **Node.js**. Функция **generateMD5** [[8]](https://www.md5hashgenerator.com/) принимает строковое значение и возвращает его хеш, вычисленный по алгоритму **MD5** (Рисунок 3):

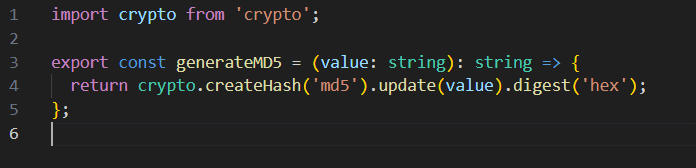


Рисунок 3 – Создание хеша

Этот алгоритм преобразует входные данные в уникальную строку фиксированной длины (32 символа), что позволяет безопасно сохранять пароли без риска хранения их в открытом виде, таким образом, даже если злоумышленник получит доступ к базе, он не сможет узнать реальные данные пользователей.

3.3.3 Реализация взаимодействия с базой данных

Для взаимодействия с базой данных **MongoDB** в проекте используется библиотека **Mongoose**, которая обеспечивает удобный интерфейс для создания моделей, валидации данных и выполнения операций с коллекциями. Она предоставляет декларативную схему описания структуры документов и позволяет абстрагироваться от низкоуровневых операций **MongoDB**.

**Подключение к базе данных:**

Подключение к **MongoDB** происходит на этапе запуска сервера в корневом файле **db.ts**. Там указывается **URI** для подключения и обрабатываются события подключения или ошибки (Рисунок 4):

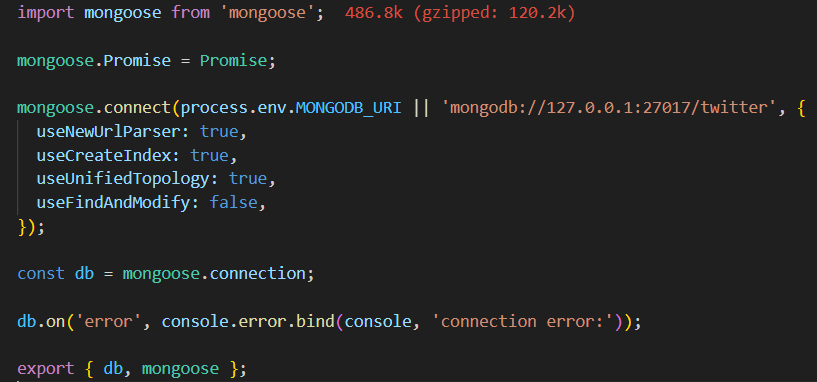


Рисунок 4 – Подключение к базе данных

Подключение осуществляется один раз при старте приложения, после чего mongoose используется во всех частях проекта.

**Запросы и логика:**

Вся логика работы с базой реализована в контроллерах и сервисах, которые вызывают методы **Mongoose** [**[9]**](https://my-js.org/docs/guide/mongoose):

* **UserModel.findOne({email})** – для поиска пользователя по email;
* **UserModel.create({…}))** – для создания нового пользователя;
* **PostModel.find().populate(‘user’)** – для получения постов вместе с данными автора;
* **UserModel.findByIdAndUpdate(id, {…})** – для обновления данных пользователя;

Эти методы позволяют реализовать **REST-запросы** [**[10]**](https://habr.com/ru/articles/193458/) в контроллерах без необходимости вручную писать Mongo-запросы.

**Связь между коллекциями:**

**Mongoose** позволяет описывать связи между сущностями. Например, поле **posts** в модели пользователя содержит массив **ObjectId**, ссылающихся на коллекцию **post**. При необходимости можно использовать метод **.populate()** для загрузки связанных данных.

**Преимущества такой архитектуры:**

* Централизация логики структуры данных в моделях;
* Упрощенная работа с БД через абстракции;
* Возможность повторного использования моделей в разных частях приложения;
* Безопасность за счет встроенной валидации данных на уровне модели.

Заключение

Целью данной курсовой работы являлась разработка архитектурной основы и серверной части гибридной платформы для публикации и монетизации творческого контента с элементами социальной сети. Основной задачей было не создание готового продукта, а проектирование и реализация ключевых серверных компонентов, способных служить фундаментом для для последующего расширения платформы.

В ходе работы были достигнуты следующие результаты:

* Проведён анализ существующих цифровых платформ и выявлены их сильные и слабые стороны;
* Определены требования к системе, обеспечивающие удобство, безопасность, масштабируемость и поддержку различных форматов контента;
* Спроектирована архитектура серверной части на основе REST API;
* Разработана структура базы данных и реализованы модели пользователей и контента с использованием MongoDB и Mongoose;
* Настроена система регистрации, валидации и авторизации пользователей (JWT, passport);
* Реализованы маршруты для работы с постами, загрузки изображений, отправки email и поддержки пользовательского взаимодействия;
* Обеспечено разделение доступа между гостями и авторизованными пользователями;
* Вся серверная логика написана на TypeScript с применением Express и современной модульной архитектуры.

Таким образом, в рамках курсовой работы была построена работоспособная и расширяемая серверная основа, на базе которой может быть реализована полноценная гибридная платформа. Полученный результат может быть использован как стартовая точка для последующей разработки пользовательского интерфейса, интеграции платежной системы, системы рекомендаций, аналитики и других модулей, необходимых для полноценного функционирования современной платформы для авторов контента.

Список источников

1. Сайт "Hack Frontend". Статья о модульной архитектуре [Электронный ресурс]. – Режим доступа: <https://www.hackfrontend.com/docs/architecture/module>. – Дата доступа: 14.06.2025.
2. Сайт "Habr". Полный цикл разработки IT продуктов на примере проекта: роли в команде, задачи заказчика, этапы [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/458336/>. – Дата доступа: 14.06.2025.
3. Сайт "Javarush". Создание React компонентов с TypeScript [Электронный ресурс]. – Режим доступа: <https://javarush.com/quests/lectures/ru.javarush.web.fullstack.lecture.level06.lecture04>. – Дата доступа: 14.06.2025.
4. Сайт "express-validator". Introduction into express-validator [Электронный ресурс]. – Режим доступа: <https://express-validator.github.io/docs/>. – Дата доступа: 14.06.2025.
5. Сайт "skillfactory media". Словарь IT – REST API [Электронный ресурс]. – Режим доступа: <https://blog.skillfactory.ru/glossary/rest-api/>. – Дата доступа: 14.06.2025.
6. Сайт "hexlet.io". Введение в интернет – клиент-серверная архитектура [Электронный ресурс]. – Режим доступа: <https://ru.hexlet.io/courses/internet-fundamentals/lessons/client-server/theory_unit>. – Дата доступа: 14.06.2025.
7. Сайт "vc.ru". Middleware в Node.js/EXPRESS [Электронный ресурс]. – Режим доступа: <https://vc.ru/dev/1546651-middleware-v-nodejs-express>. – Дата доступа: 14.06.2025.
8. Сайт "Dan’s Tools". MD5 Hash Generator [Электронный ресурс]. – Режим доступа: <https://www.md5hashgenerator.com/>. – Дата доступа: 14.06.2025.
9. Сайт "MyJavaScript". Руководство Mongoose [Электронный ресурс]. – Режим доступа: <https://my-js.org/docs/guide/mongoose>. – Дата доступа: 14.06.2025.
10. Сайт "Habr". RESTful API на Node.js + MongoDB [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/458336/>. – Дата доступа: 14.06.2025.

Приложение А

**(обязательное)**

**Реализованный код серверной части**

**Cloudinary.ts:**

import cloudinary from 'cloudinary';

if (!process.env.CLOUDINARY\_NAME) {

  throw new Error('Отсутствуют конфигурации для Cloudinary');

}

// @ts-ignore

cloudinary.config({

  cloud\_name: process.env.CLOUDINARY\_NAME,

  api\_key: process.env.CLOUDINARY\_API\_KEY,

  api\_secret: process.env.CLOUDINARY\_API\_SECRET,

});

export default cloudinary;

**mailer.ts:**

import nodemailer from 'nodemailer';

const options = {

  host: process.env.NODEMAILER\_HOST || 'smtp.mailtrap.io',

  port: Number(process.env.NODEMAILER\_PORT) || 2525,

  auth: {

    user: process.env.NODEMAILER\_USER,

    pass: process.env.NODEMAILER\_PASS,

  },};

export const mailer = nodemailer.createTransport(options);

**passport.ts:**

import passport from 'passport';

import { Strategy as LocalStrategy } from 'passport-local';

import { Strategy as JWTstrategy, ExtractJwt } from 'passport-jwt';

import { UserModel, UserModelInterface } from '../models/UserModel';

import { generateMD5 } from '../utils/generateHash';

passport.use(

  new LocalStrategy(

    async (username, password, done): Promise<void> => {

      try {

        const user = await UserModel.findOne({ $or: [{ email: username }, { username }] }).exec();

        if (!user) {

          return done(null, false);

        }

        if (user.confirmed && user.password === generateMD5(password + process.env.SECRET\_KEY)) {

          done(null, user);

        } else {

          done(null, false);

        }

      } catch (error) {

        done(error, false);

}},),);

passport.use(

  new JWTstrategy(

    {

      secretOrKey: process.env.SECRET\_KEY || '123',

      jwtFromRequest: ExtractJwt.fromHeader('token'),

    },

    async (payload: { data: UserModelInterface }, done): Promise<void> => {

      try {

        const user = await UserModel.findById(payload.data.\_id).exec();

        if (user) {

          return done(null, user);

        }

        done(null, false);

      } catch (error) {

        done(error, false);

      }},),);

passport.serializeUser((user: UserModelInterface, done) => {

  done(null, user?.\_id);

});

passport.deserializeUser((id, done) => {

  UserModel.findById(id, (err, user) => {

    done(err, user);

  });});

export { passport };

**PostsModel.ts:**

import { model, Schema, Document } from 'mongoose';

import { UserModelDocumentInterface } from './UserModel';

export interface TweetModelInterface {

  \_id?: string;

  text: string;

  user: UserModelDocumentInterface;

  images?: string[];

}

export type TweetModelDocumentInterface = TweetModelInterface & Document;

const TweetSchema = new Schema<TweetModelInterface>(

  {

    text: {

      required: true,

      type: String,

      maxlength: 280,

    },

    user: {

      required: true,

      ref: 'User',

      type: Schema.Types.ObjectId,

    },

    images: [

      {

        type: String,

      },],},

  {

    timestamps: true,

  },);

export const TweetModel = model<TweetModelDocumentInterface>('Tweet', TweetSchema);

**UserModel.ts:**

import { model, Schema, Document } from 'mongoose';

export interface UserModelInterface {

  \_id?: string;

  email: string;

  fullname: string;

  username: string;

  password: string;

  confirmHash: string;

  confirmed?: boolean;

  location?: string;

  about?: string;

  website?: string;

  tweets?: string[];

}

export type UserModelDocumentInterface = UserModelInterface & Document;

const UserSchema = new Schema<UserModelInterface>(

  {

    email: {

      unique: true,

      required: true,

      type: String,

    },

    fullname: {

      required: true,

      type: String,

    },

    username: {

      unique: true,

      required: true,

      type: String,

    },

    password: {

      required: true,

      type: String,

    },

    confirmHash: {

      required: true,

      type: String,

    },

    confirmed: {

      type: Boolean,

      default: false,

    },

    location: String,

    about: String,

    website: String,

    tweets: [{ type: Schema.Types.ObjectId, ref: 'Tweet' }],

  },

  {

    timestamps: true,

  },);

UserSchema.set('toJSON', {

  transform: function (\_, obj) {

    delete obj.password;

    delete obj.confirmHash;

    return obj;

  },});

export const UserModel = model<UserModelDocumentInterface>('User', UserSchema);

**IsValidObjectId.ts:**

import { mongoose } from '../core/db';

export const isValidObjectId = mongoose.Types.ObjectId.isValid;

**sendEmail.ts:**

import { SentMessageInfo } from 'nodemailer/lib/sendmail-transport';

import { mailer } from '../core/mailer';

interface SendEmailProps {

  emailFrom: string;

  emailTo: string;

  subject: string;

  html: string;

}

export const sendEmail = (

  { emailFrom, emailTo, subject, html }: SendEmailProps,

  callback?: (err: Error | null, info: SentMessageInfo) => void,

) => {

  mailer.sendMail(

    {

      from: emailFrom,

      to: emailTo,

      subject: subject,

      html: html,

    },

    callback ||

      function (err: Error | null, info: SentMessageInfo) {

        if (err) {

          console.log(err);

        } else {

          console.log(info);

        }},);};

**createPosts.ts:**

import { body } from 'express-validator';

export const createTweetValidations = [

  body('text', 'Введите текст твита')

    .isString()

    .isLength({

      max: 280,

    })

    .withMessage('Максимальная длина твита 280 символов'),

];

**register.ts:**

import { body } from 'express-validator';

export const registerValidations = [

  body('email', 'Введите E-Mail')

    .isEmail()

    .withMessage('Неверный E-Mail')

    .isLength({

      min: 10,

      max: 40,

    })

    .withMessage('Допустимое кол-во символов в почте от 10 до 40.'),

  body('fullname', 'Введите имя')

    .isString()

    .isLength({

      min: 2,

      max: 40,

    })

    .withMessage('Допустимое кол-во символов в имени от 2 до 40.'),

  body('username', 'Укажите логин')

    .isString()

    .isLength({

      min: 2,

      max: 40,

    })

    .withMessage('Допустимое кол-во символов в логине от 2 до 40.'),

  body('password', 'Укажите пароль')

    .isString()

    .isLength({

      min: 6,

    })

    .withMessage('​Минимальная длина пароля 6 символов')

    .custom((value, { req }) => {

      if (value !== req.body.password2) {

        throw new Error('Пароли не совпадают');

      } else {

        return value;

      }}),];

**server.ts:**

import dotenv from 'dotenv';

dotenv.config();

import './core/db';

import express from 'express';

import multer from 'multer';

import bodyParser from 'body-parser';

import { registerValidations } from './validations/register';

import { passport } from './core/passport';

import { createTweetValidations } from './validations/createTweet';

import { TweetsCtrl } from './controllers/TweetsController';

import { UserCtrl } from './controllers/UserController';

import { UploadFileCtrl } from './controllers/UploadFileController';

const app = express();

// const storage = multer.diskStorage({

//   destination: function (\_, \_\_, cb) {

//     cb(null, \_\_dirname + '/uploads');

//   },

//   filename: function (\_, file, cb) {

//     const ext = file.originalname.split('.').pop();

//     cb(null, 'image-' + Date.now() + '.' + ext);

//   },

// });

const storage = multer.memoryStorage();

const upload = multer({ storage });

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.use(passport.initialize());

app.get('/users', UserCtrl.index);

app.get('/users/me', passport.authenticate('jwt', { session: false }), UserCtrl.getUserInfo);

app.get('/users/:id', UserCtrl.show);

app.get('/tweets', TweetsCtrl.index);

app.get('/tweets/:id', TweetsCtrl.show);

app.get('/tweets/user/:id', TweetsCtrl.getUserTweets);

app.delete('/tweets/:id', passport.authenticate('jwt'), TweetsCtrl.delete);

app.patch('/tweets/:id', passport.authenticate('jwt'), createTweetValidations, TweetsCtrl.update);

app.post('/tweets', passport.authenticate('jwt'), createTweetValidations, TweetsCtrl.create);

app.get('/auth/verify', registerValidations, UserCtrl.verify);

app.post('/auth/register', registerValidations, UserCtrl.create);

app.post('/auth/login', passport.authenticate('local'), UserCtrl.afterLogin);

app.post('/upload', upload.single('image'), UploadFileCtrl.upload);

app.listen(process.env.PORT, (): void => {

  console.log('SERVER RUNNING!');

});