**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_Toc202185880)

[1 Обзор существующих решений 6](#_Toc202185881)

[1.1 Общая характеристика социальных сетей 6](#_Toc202185882)

[1.2 Сравнение популярных реализаций 6](#_Toc202185883)

[1.3 Существующие фреймворки и инструменты 7](#_Toc202185884)

[1.4 Причины отказа от готовых CMS и движков 8](#_Toc202185885)

[1.5 Обоснование выбора технологий 8](#_Toc202185886)

[1.6 Целесообразность собственной реализации 9](#_Toc202185887)

[1.7 Архитектурные подходы к построению серверной части 9](#_Toc202185888)

[1.8 Безопасность серверной части 10](#_Toc202185889)

[2. ОПИСАНИЕ ПРОГРАММИРУЕМОЙ СИСТЕМЫ 11](#_Toc202185890)

[2.1 Архитектура приложения 11](#_Toc202185891)

[2.2 Диаграммы 12](#_Toc202185892)

[2.3 Используемые технологии 15](#_Toc202185893)

[3. результаты разработки программного продукта 16](#_Toc202185894)

[ЗАКЛЮЧЕНИЕ 21](#_Toc202185895)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 22](#_Toc202185896)

[ПРИЛОЖЕНИЯ 23](#_Toc202185897)

[Приложение А – Исходный код программы 24](#_Toc202185898)

# ВВЕДЕНИЕ

В настоящее время социальные сети стали неотъемлемой частью повседневной жизни миллионов пользователей по всему миру. Они предоставляют возможности для общения, публикации контента, комментирования и взаимодействия между людьми в онлайн-пространстве.

**Цель работы:** разработка серверной части социальной сети, которая обеспечивает функциональность регистрации пользователей, создания и получения публикаций (постов), добавления комментариев, а также авторизации с использованием токенов доступа.

**Задачи работы:**

1. разработка API-интерфейса для работы с пользователями (авторы профилей);
2. реализация регистрации и аутентификации пользователей с помощью JWT;
3. создание CRUD-интерфейса для постов и комментариев;
4. обеспечение защиты маршрутов через middleware (аутентификация);
5. документирование API с использованием Swagger UI;
6. использование СУБД PostgreSQL и ORM-библиотеки GORM для хранения и обработки данных.

**Объектом исследования** в данной работе является процесс проектирования и разработки серверных компонентов информационных систем, в частности — веб-приложений, основанных на архитектуре REST API.

**Предметом исследования** является разработка серверной части социальной сети с использованием языка программирования Go, фреймворка Gin и системы управления базами данных PostgreSQL.

В качестве основных методов исследования применяются анализ, синтез, абстрагирование, сравнение, моделирование, а также применение объектно-ориентированного и компонентного подходов.

Практическая реализация поставленной задачи осуществляется с применением инструментов индустриальной разработки: GORM (ORM для Go), JWT (аутентификация), Swagger (документирование API), что позволяет обеспечить масштабируемость и расширяемость архитектуры.

**Информационной базой** исследования послужили открытые источники, официальная документация используемых технологий, а также материалы курса «Технологии индустриального программирования», представленные в системе дистанционного обучения РТУ МИРЭА.

В отчёте представлен процесс проектирования и разработки REST-сервиса, его архитектура, описание используемых технологий, схемы взаимодействия компонентов и результаты практической реализации серверной части социальной сети.

# 1 Обзор существующих решений

Развитие социальных сетей как явления привело к появлению множества технических решений, связанных с проектированием и реализацией их серверной архитектуры. Современные приложения подобного рода требуют высокой надёжности, масштабируемости, поддержки взаимодействия между пользователями и обработки больших объёмов данных в реальном времени. В данной главе будет рассмотрен обзор существующих технологий, подходов и платформ, используемых для реализации серверной части социальных сетей, а также приведено обоснование выбора собственной реализации.

## 1.1 Общая характеристика социальных сетей

Социальная сеть — это программно-аппаратная система, предоставляющая пользователям возможность обмена сообщениями, публикации и взаимодействия с контентом (постами, комментариями, медиа), а также формирования персональных профилей. В современных реализациях социальные сети реализуются в виде распределённых клиент-серверных приложений, где серверная часть отвечает за:

1. управление данными пользователей;
2. реализацию бизнес-логики;
3. обеспечение безопасности (аутентификация, авторизация);
4. обработку запросов от фронтенда (мобильного или веб-клиента);
5. логирование и интеграцию с внешними API.

## 1.2 Сравнение популярных реализаций

Рассмотрим некоторые крупные технологические решения (Таблица 1.1), применяемые в реальных социальных платформах:

Таблица 1.1 – Платформы и их решения

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Платформа | Языки сервера | СУБД | Аутентификация | Архитектура |
| Facebook | PHP (HHVM), C++, Erlang | MySQL, RocksDB | OAuth 2.0 | Микросервисная |
| Twitter | Scala, Java | MySQL, Redis | OAuth 1.0a | Микросервисы |
| VK | PHP, Go, Python | PostgreSQL, Tarantool | JWT, OAuth | Монолит + API |

Большинство решений используют REST или GraphQL API для связи клиента и сервера, масштабируются с помощью облачных решений, а для хранения данных предпочитают PostgreSQL или MySQL.

## 1.3 Существующие фреймворки и инструменты

В реальной практике разработчики не создают сервер с нуля, а используют надёжные фреймворки. Ниже перечислены некоторые популярные инструменты для создания API:

1. Node.js (Express.js) — удобен для быстрого старта, но может проигрывать по производительности.
2. Django (Python) — мощный фреймворк с ORM и встроенной панелью администратора, часто используется в MVP.
3. Spring Boot (Java) — корпоративный стандарт, устойчив к высоким нагрузкам, но требует большого количества кода.
4. Ruby on Rails — удобен для быстрой сборки REST-сервисов, активно используется в стартапах.
5. Go (Gin) — высокопроизводительный, компилируемый язык с минимализмом и хорошей поддержкой параллельности. С фреймворком Gin — подходит для лёгких REST-сервисов.

## 1.4 Причины отказа от готовых CMS и движков

Существуют готовые решения (Elgg, Oxwall, HumHub), которые позволяют быстро собрать социальную сеть. Однако они имеют ряд ограничений:

1. сложность модификации кода и архитектуры под нестандартные задачи;
2. устаревшие технологии и слабая поддержка;
3. ограниченные возможности по масштабированию;
4. отсутствие гибкости в выборе базы данных и авторизации.

Поэтому при создании учебного или исследовательского проекта, требующего полной прозрачности логики и архитектуры, целесообразнее реализовывать серверную часть самостоятельно.

## 1.5 Обоснование выбора технологий

В качестве базового стека для разработки выбраны следующие инструменты:

1. язык программирования Go – благодаря своей скорости, статической типизации, встроенной поддержке параллелизма и минимализму, идеально подходит для разработки производительных веб-сервисов;
2. фреймворк Gin – лёгкий и быстрый HTTP-фреймворк для Go, обеспечивающий маршрутизацию, middleware, JSON-парсинг и валидацию;
3. ORM GORM – библиотека для работы с PostgreSQL через Go, поддерживающая миграции, связи между таблицами, транзакции;
4. JWT – JSON Web Token для безопасной авторизации и хранения сессий;
5. Swagger – инструмент для документирования REST API, удобный для тестирования и взаимодействия с фронтенд-разработчиками;
6. PostgreSQL – мощная реляционная база данных с открытым исходным кодом, поддерживающая индексы, связи, JSON-поля, транзакции.

## 1.6 Целесообразность собственной реализации

Создание собственной серверной части даёт разработчику полное понимание архитектуры социальной платформы, позволяет на практике применить знания в области баз данных, работы с API, реализации авторизации, логирования и проектирования модели данных. Кроме того, такая реализация легко расширяема, может быть доработана в будущем и используется как часть более крупного программного комплекса.

Обзор существующих решений показал, что при наличии простых и понятных фреймворков, таких как Gin и GORM, разработка собственной серверной части социальной сети является эффективным, обучающим и гибким решением, особенно в контексте курсовой работы, где требуется полное понимание всех компонентов системы.

## 1.7 Архитектурные подходы к построению серверной части

Наиболее распространёнными архитектурными подходами при разработке социальных сетей являются:

**Монолитная архитектура, которая п**редставляет собой единую кодовую базу, где все компоненты (регистрация, посты, комментарии, авторизация и т.д.) находятся в одном приложении.

Преимущества:

* простота разработки и деплоя;
* быстрая реализация.

Недостатки:

* трудности при масштабировании;
* сложность поддержки при росте проекта.

Микросервисная архитектура, где каждый компонент системы реализован как отдельный сервис, взаимодействующий с другими через API.

Преимущества:

* масштабируемость и отказоустойчивость;
* независимость разработки и обновления.

Недостатки:

* более высокая сложность, необходимость в сервисной шине, авторизации между сервисами.

Для целей данной курсовой работы выбран монолитный подход, как наиболее подходящий для прототипа с ограниченным числом компонентов.

## 1.8 Безопасность серверной части

Важным аспектом любой социальной сети является безопасность. Современные серверные решения реализуют следующие механизмы:

1. Аутентификация через JWT – безопасная и масштабируемая передача токена авторизации между клиентом и сервером.
2. Шифрование паролей – с использованием устойчивых к атаке алгоритмов (например, SHA-256, bcrypt, Argon2).
3. Защита от SQL-инъекций – достигается использованием ORM (например, GORM), которая применяет параметризованные запросы.
4. CORS и CSRF-защита – настройка заголовков доступа к API и защита от подделки запросов.
5. Валидация входящих данных – обязательна для всех точек входа (через middleware).

# 2. ОПИСАНИЕ ПРОГРАММИРУЕМОЙ СИСТЕМЫ

В данной главе рассматривается состав и архитектура разрабатываемой программной системы, которая реализует серверную часть социальной сети. Основное назначение сервиса – предоставление REST API, обеспечивающего клиентам (веб или мобильным приложениям) доступ к операциям регистрации, авторизации, управления профилем, публикации и чтения постов, а также комментирования.

Ниже в таблице 2.1 представлены основные требования к функциональности и характеристикам программной системы:

Таблица 2.1 – Требования к продукту

|  |  |  |
| --- | --- | --- |
| № | Требование | Значение |
| 1 | Язык программирования | Go (Golang) |
| 2 | Корректность работы | Приложение запускается, обрабатывает HTTP-запросы и стабильно функционирует до остановки |
| 3 | Использование архитектурных принципов | Используется REST-архитектура, принципы слоения (handlers, services, models, routes) |
| 4 | Документированное API | Swagger UI подключен и содержит описание всех маршрутов |
| 5 | Авторизация | JWT (JSON Web Token) реализован для защиты маршрутов и идентификации пользователей |
| 6 | Хранение данных | PostgreSQL в связке с GORM ORM |
| 7 | Интерфейс взаимодействия | JSON API с поддержкой всех CRUD-операций по HTTP |
| 8 | Структурированное логирование | Используется log.Logger с уровнями логирования и сохранением в файл |
| 9 | Безопасность хранения паролей | Пароли хэшируются с использованием sha256 с солью |
| 10 | Система контроля версий | Проект отслеживается с помощью Git |

## 2.1 Архитектура приложения

Приложение реализовано на языке Go с использованием фреймворка Gin. В структуре проекта чётко выделены слои:

1. handlers (controllers) – обрабатывают HTTP-запросы;
2. services – реализуют бизнес-логику;
3. models – описывают структуры данных и связи;
4. routes – задают маршруты API;
5. middleware – обеспечивает авторизацию и логирование;
6. database – отвечает за подключение к PostgreSQL и миграции через GORM.

## 2.2 Диаграммы

Для описания работы системы были использованы диаграммы, описывающие работу системы в различных ее аспектах. Были спроектированы диаграмма состояний (Рисунок 2.1), диаграмма классов (Рисунок 2.2) и диаграмма последовательности (Рисунок 2.3). Они позволяют наглядно представить структуру данных и последовательность взаимодействий между компонентами.

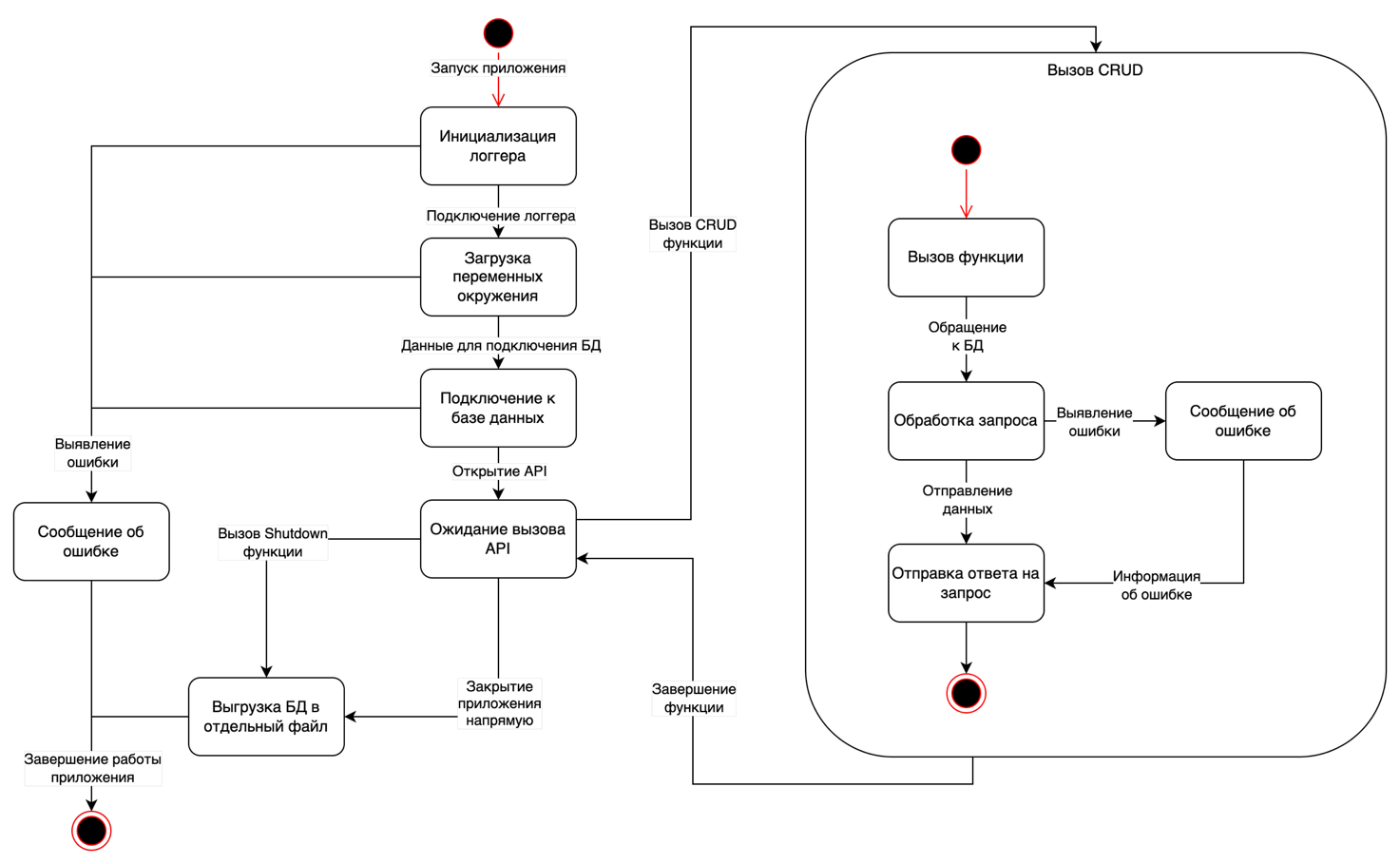


Рисунок 2.1 — Диаграмма состояний системы

Диаграмма состояний описывает жизненный цикл взаимодействия клиента с системой. Приложение запускается, инициализирует подключения, ожидает входящие HTTP-запросы, обрабатывает их (включая маршрутизацию, валидацию, аутентификацию), и отправляет ответы клиенту. При получении сигнала завершения (например, SIGINT) – сервис закрывает соединения и завершает работу. От момента старта до остановки, включая обработку исключений и graceful shutdown.

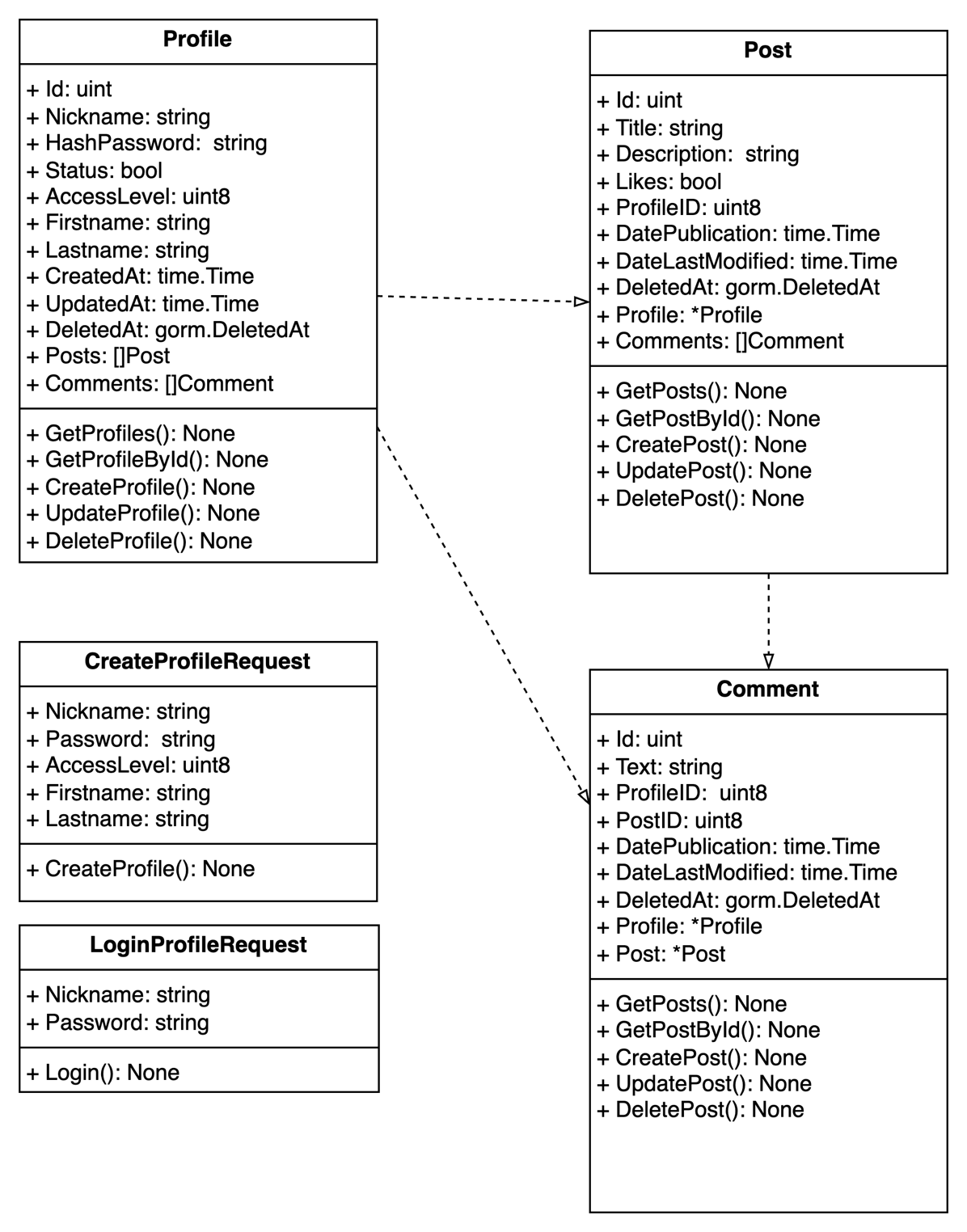


Рисунок 2.2 — Диаграмма классов (структур данных)

Диаграмма классов включает следующие основные сущности:

1. Profile – пользователь (автор), имеет связи с Post и Comment
2. Post – публикация, связана с автором и комментариями
3. Comment – комментарий, содержит связь с постом и автором, поддерживает вложенность.

Все структуры определены в виде struct и связаны через GORM, поддерживаются миграции, внешние ключи и soft delete.

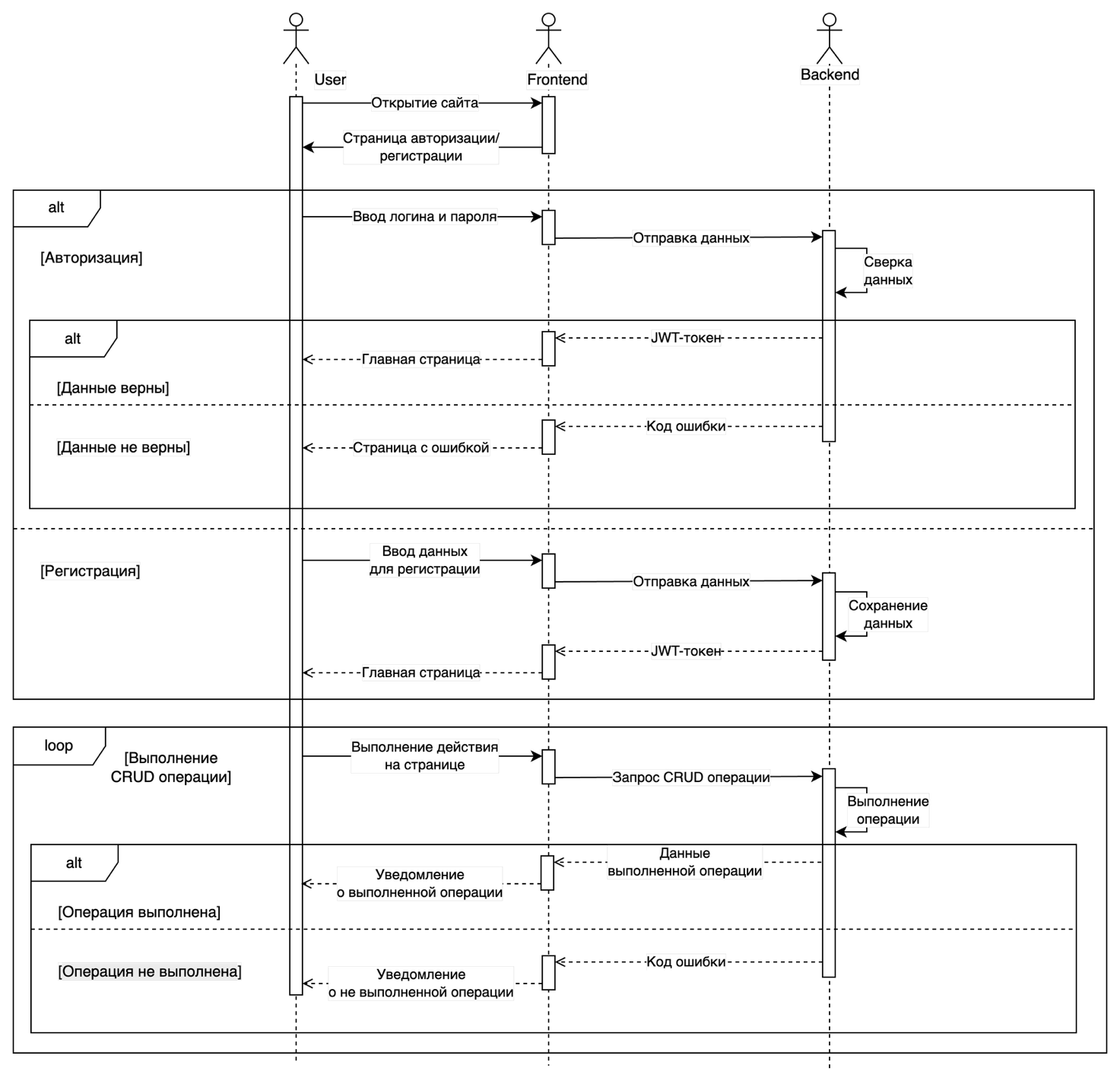


Рисунок 2.3 — Диаграмма последовательности

Диаграмма иллюстрирует взаимодействие между пользователем, frontend-интерфейсом и backend-сервером в процессе работы. Рассматриваются сценарии авторизации, регистрации, выполнения CRUD-операций и обработки ошибок. Отображены альтернативные потоки выполнения (например, при неверных данных), а также логика возврата JWT-токена для подтверждения успешной аутентификации.

## 2.3 Используемые технологии

В таблице 2.2 представлены технологии программируемой системе.

Таблица 2.2 – Технология продукта

|  |  |  |
| --- | --- | --- |
| **№** | **Технология/инструмент** | **Назначение** |
| 1 | Go (Golang) | Язык программирования, выбранный за счёт своей производительности и простоты |
| 2 | Gin | HTTP-фреймворк для Go, обеспечивает маршрутизацию, обработку запросов и middleware |
| 3 | GORM | ORM-библиотека для работы с PostgreSQL через Go |
| 4 | PostgreSQL | Реляционная СУБД для хранения пользователей, постов и комментариев |
| 5 | JWT | Технология аутентификации через токены, применяемая для защиты маршрутов |
| 6 | bcrypt | Алгоритм хеширования паролей для безопасного хранения |
| 7 | Swagger (swaggo/gin-swagger) | Генерация и отображение документации API |
| 8 | Log.Logger + файл логирования | Ведение логов работы сервера с сохранением в файл |
| 9 | dotenv (.env) | Загрузка конфигурационных переменных (порты, ключи и т.д.) |
| 10 | Git | Система контроля версий, используемая для отслеживания изменений |

В результате проведённого проектирования была сформирована чёткая структура программируемой системы, определены её основные компоненты и взаимодействие между ними. Используемые технологии обеспечивают надёжность, масштабируемость и безопасность серверной части социальной сети. Разработанные диаграммы наглядно иллюстрируют архитектуру, внутреннюю логику и поведение системы в различных сценариях. Представленные решения стали основой для последующей реализации и тестирования функциональности.

# 3. результаты разработки программного продукта

**Реализация требований к системе**

1. **Язык программирования**

Для разработки использовался язык **Go (Golang)** – современный компилируемый язык, обладающий высокой производительностью, встроенной поддержкой конкурентности и лаконичным синтаксисом, идеально подходящий для серверной логики и сетевого программирования.

1. **Корректность работы**

Приложение запускается, подключается к базе данных, открывает HTTP-сервер и стабильно функционирует до получения сигнала завершения. Обработка ошибок осуществляется централизованно с выводом в лог, предусмотрено корректное завершение работы через механизм shutdown.

1. **Использование архитектурных принципов**

В проекте реализована REST-архитектура. Код структурирован по слоям: обработчики (handlers), бизнес-логика (services), модели (models), маршруты (routes). Такое разделение улучшает читаемость, сопровождаемость и масштабируемость кода.

1. **Документированное API**

Документация к REST API сгенерирована автоматически с помощью библиотеки **swaggo**, и доступна пользователю через **Swagger UI**. Все маршруты снабжены аннотациями и комментариями, что облегчает тестирование и интеграцию с внешними сервисами.

1. **Авторизация**

В проекте реализована система авторизации на основе **JWT (JSON Web Token)**. При успешной аутентификации пользователю выдается токен, который требуется для доступа к защищённым маршрутам. Токен проверяется через middleware.

1. **Хранение данных**

В качестве СУБД используется **PostgreSQL**. Для взаимодействия с базой применена ORM-библиотека **GORM**, обеспечивающая автоматические миграции, работу с транзакциями и поддержку связей между моделями.

1. **Интерфейс взаимодействия**

Сервер предоставляет RESTFUL JSON API, поддерживающий все основные CRUD-операции. Обмен данными с клиентом осуществляется в формате JSON, что соответствует современным стандартам API–взаимодействия.

1. **Структурированное логирование**

Для логирования используется стандартный log.Logger с разделением по уровням (Info, Error, Panic). Все логи записываются в отдельный лог-файл, что обеспечивает удобство при отладке и мониторинге работы приложения.

1. **Безопасность хранения паролей**

Пароли пользователей не хранятся в открытом виде. Перед сохранением выполняется их хэширование с использованием алгоритма **SHA-256 с солью**, что исключает возможность восстановления исходного пароля при утечке базы.

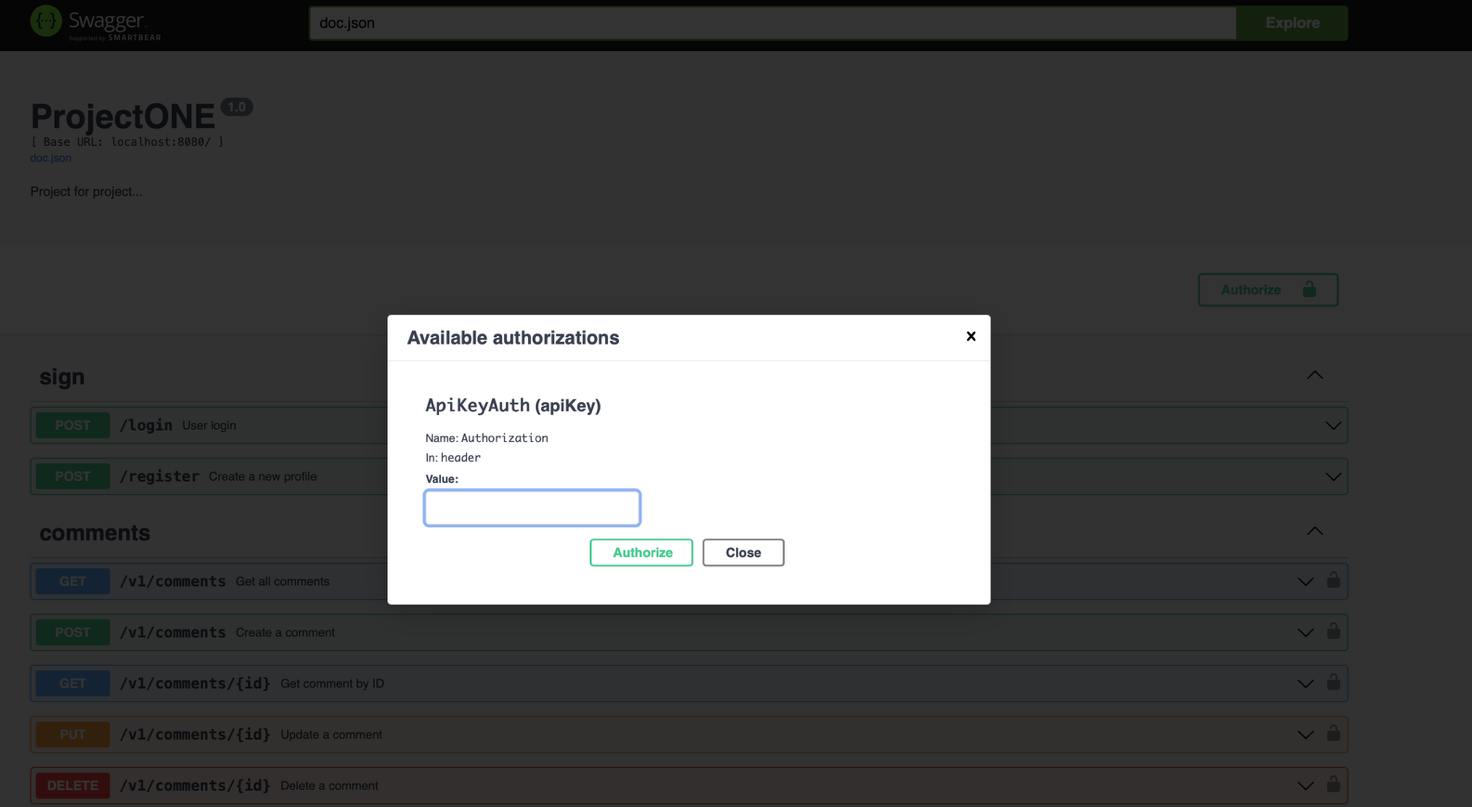
1. **Система контроля версий**

Вся разработка велась с использованием системы контроля версий **Git**. История изменений отслеживается в репозитории, что обеспечивает стабильность проекта, возможность отката и командную работу.

**Функциональное тестирование программного продукта**

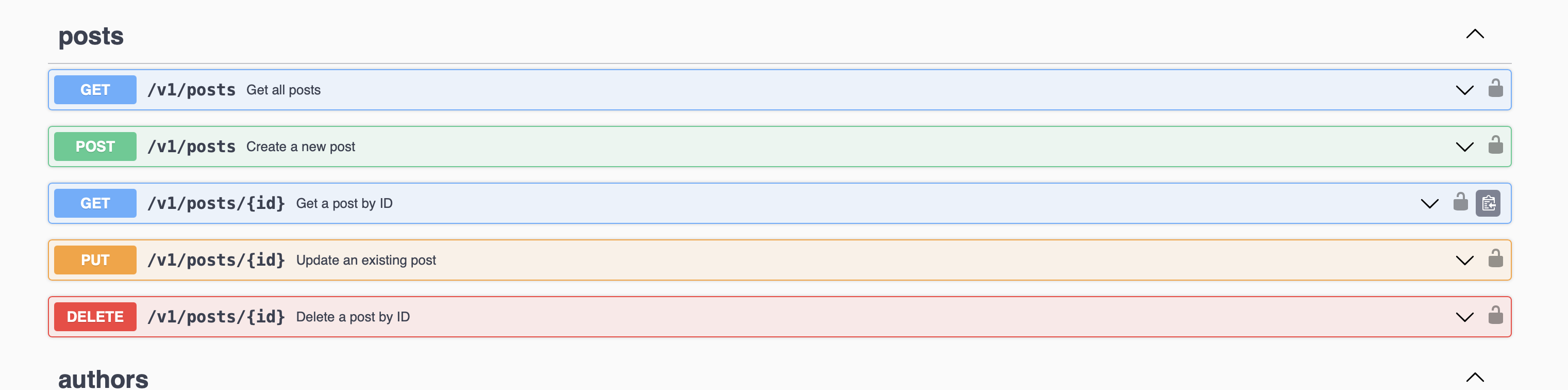
Работу серверного приложения координирует основная точка входа — функция main(), где запускается Run() (Приложение А, Листинг A.2), которая инициализирует подключение к базе данных, логгер, а также запускает HTTP-сервер с настройкой маршрутов. Вся бизнес-логика реализована через слоистую архитектуру: контроллеры (handlers), сервисы (services), модели (models) и маршруты (routes), что позволяет обеспечить модульность и удобство тестирования.

Работа приложения начинается с регистрации или входа пользователя (Рисунок 3.1). После успешной авторизации система возвращает **JWT-токен**, который пользователь использует для выполнения всех защищённых операций, включая работу с постами и комментариями. Верификация токена осуществляется через middleware, что демонстрирует корректную работу механизма авторизации.



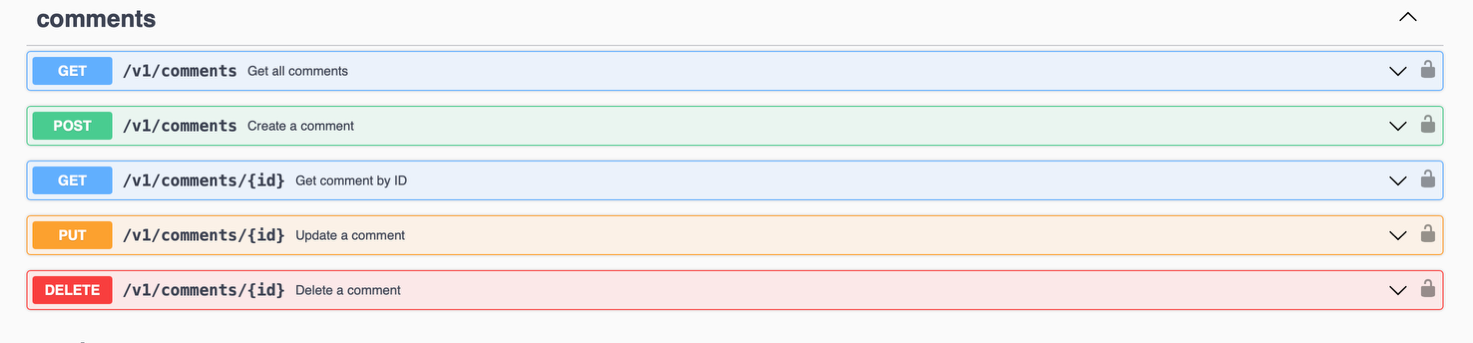
****Рисунок 3.1 — Авторизация и регистрация через Swagger UI****

После входа пользователь может выполнять CRUD-операции с постами (Рисунок 3.2), включая создание, просмотр, редактирование и удаление. Тестирование проводилось через интерфейс **Swagger UI**, где каждый маршрут сопровождён примером тела запроса и ожидаемыми кодами ответа. Все операции возвращают статусы 200, 201, 400, 404 или 500 в зависимости от ситуации, что подтверждает корректную реализацию обработки ошибок.



****Рисунок 3.2 — Создание и получение постов****

Также протестирована работа с комментариями: пользователь может оставить комментарий к посту, получить комментарий по ID, изменить или удалить его (Рисунок 3.3). Комментарии связаны с постами и авторами через внешние ключи, а структура базы данных корректно отражает эти связи, что также было подтверждено при миграции через GORM.



****Рисунок 3.3 — Работа с комментариями через Swagger****

Дополнительно протестирована работа функции **Shutdown**, которая корректно завершает работу сервера при получении сигнала SIGINT или SIGTERM, выгружая данные из базы и завершая все активные соединения (Приложение A, Листинг A.3). Это подтверждает устойчивость приложения при непредвиденных остановках.

В ходе тестирования также была проверена система логирования. Все ошибки, HTTP-запросы и события фиксируются в лог-файл, обеспечивая прозрачность и возможность отладки приложения.

Результаты тестирования подтверждают соответствие функционала требованиям и стабильную работу всех модулей. Сервер успешно справляется с многократными запросами, корректно обрабатывает ошибки и обеспечивает надёжную защиту данных.

**Инструкция по эксплуатации**

После запуска приложения пользователь (или разработчик) может взаимодействовать с серверной частью через графический интерфейс Swagger UI или с помощью инструментов типа Postman и curl. Программа автоматически инициализирует подключение к базе данных, настраивает маршруты и начинает прослушивать HTTP-запросы на заданном порту (по умолчанию localhost:8080).

При открытии Swagger UI пользователю становятся доступны следующие основные действия:

**Регистрация профиля.** Для создания нового пользователя необходимо перейти в раздел POST /register, нажать **Try it out** и заполнить поля:

1. nickname – уникальное имя пользователя;
2. password – пароль (будет автоматически захеширован);

**Вход в систему.** Для авторизации используется маршрут POST /login. Необходимо указать nickname и пароль. В ответ сервер возвращает **JWT-токен**, который используется для доступа к защищённым маршрутам.

**Работа с профилями, постами и комментариями.** Все CRUD-операции (создание, получение, изменение, удаление) реализованы по REST-принципам. Примеры:

1. GET /v1/profiles — получить список пользователей;
2. POST /v1/posts — создать новый пост;
3. POST /v1/comments — добавить комментарий.

Для выполнения этих запросов необходимо авторизоваться, вставив JWT-токен в поле **Authorize** (в верхней части Swagger UI).

**Завершение работы сервера.** Сервер поддерживает корректное завершение работы (graceful shutdown) при получении сигнала Ctrl + C (или через системные сигналы SIGINT, SIGTERM). Также реализован специальный маршрут GET /shutdown, при вызове которого производится выгрузка данных и остановка сервера.

Все ошибки и действия логируются в файл app.log, который можно использовать для отладки или анализа работы приложения. Программа устойчива к ошибочным запросам и при необходимости возвращает корректные HTTP-статусы (400, 401, 404, 500) с сообщением об ошибке.

После завершения работы (например, остановки сервера через Ctrl + C) все соединения с базой данных закрываются автоматически.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была разработана серверная часть социальной сети, включающая в себя основные функции: регистрацию, авторизацию, управление постами и комментариями. В качестве языка программирования выбран Go – современный инструмент для построения быстрых и надёжных веб-приложений.

Архитектура проекта основана на принципах REST и разделении на слои: обработчики, сервисы, модели и маршруты. Для взаимодействия с базой данных использована PostgreSQL в связке с GORM. Все данные структурированы, а связи между сущностями реализованы через внешние ключи.

Аутентификация пользователей выполнена с использованием JWT-токенов, что обеспечивает защиту маршрутов и удобство взаимодействия. Пароли проходят хеширование с применением безопасного алгоритма, а все действия фиксируются в лог-файл для последующего анализа.

Дополнительно реализована документация API через Swagger, что облегчает тестирование и взаимодействие с системой. Работа приложения проверена через функциональное тестирование, подтвердившее корректность реализации и стабильность работы.

Разработанный сервер может быть расширен и использован в более крупной системе. Проект продемонстрировал эффективность применения языка Go и связанных технологий в задачах построения надёжной серверной архитектуры.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Go (язык программирования) / [Электронный ресурс] // Википедия : [сайт]. – URL: [https://ru.wikipedia.org/wiki/Go\_(язык\_программирования)](https://ru.wikipedia.org/wiki/Go_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) (дата обращения: 18.06.2025).
2. Gin Web Framework / [Электронный ресурс] // Официальный сайт Gin : [сайт]. – URL: <https://gin-gonic.com/> (дата обращения: 10.05.2025).
3. GORM – The fantastic ORM library for Golang / [Электронный ресурс] // GORM : [сайт]. – URL: <https://gorm.io/> (дата обращения: 12.05.2025).
4. JSON Web Token (JWT) / [Электронный ресурс] // Википедия : [сайт]. – URL: <https://ru.wikipedia.org/wiki/JSON_Web_Token> (дата обращения: 12.05.2025).
5. PostgreSQL – мощная объектно-реляционная система управления базами данных / [Электронный ресурс] // Официальный сайт PostgreSQL: [сайт]. – URL: <https://www.postgresql.org/> (дата обращения: 14.05.2025).
6. REST API: принципы и реализация / [Электронный ресурс] // Хабр : [сайт]. – URL: <https://habr.com/ru/post/501552/> (дата обращения: 15.05.2025).
7. Swagger UI – документация REST API / [Электронный ресурс] // Swagger : [сайт]. – URL: <https://swagger.io/tools/swagger-ui/> (дата обращения: 15.05.2025).
8. Hashing passwords securely / [Электронный ресурс] // OWASP : [сайт]. – URL: <https://owasp.org/www-community/controls/Password_Storage_Cheat_Sheet> (дата обращения: 10.05.2025).
9. Log package – Go standard library / [Электронный ресурс] // pkg.go.dev : [сайт]. – URL: <https://pkg.go.dev/log> (дата обращения: 15.05.2025).
10. Использование .env файлов в Go / [Электронный ресурс] // Dev.to : [сайт]. – URL: <https://dev.to/mikegeo/use-env-files-in-go-5h5f> (дата обращения: 16.05.2025).

# ПРИЛОЖЕНИЯ

Приложение А – Исходный код программы

## Приложение А – Исходный код программы

Листинг кода А.1 – Функция main

package main

import (

"ProjectONE/cmd"

"fmt"

)

//@title ProjectONE

//@version 1.0

//@description Project for project...

//@host localhost:8080

//@BasePath /

// @securityDefinitions.apikey ApiKeyAuth

// @in header

// @name Authorization

func main() {

if err := cmd.Run(); err != nil {

fmt.Printf("Запуск программы не сработал!!!\n%v", err)

}

}

Листинг кода А.2 – Функция Run

package cmd

import (

v1 "ProjectONE/internal/api/v1"

database "ProjectONE/internal/database/postgres"

"ProjectONE/internal/models"

"ProjectONE/pkg/utils"

"github.com/joho/godotenv"

)

func Run() error {

// Запуск логгера

utils.InitLogger("pkg/utils/app.log")

// Загружаем переменные окружения из файла .env

if err := godotenv.Load(); err != nil {

utils.Logger.Fatalf("Error loading .env file")

return err

}

if err := database.Connect(database.LoadConfigFromEnv()); err != nil {

return err

}

defer database.Close()

utils.Logger.Info("Передача моделей")

database.CreateObjDB(&models.Profile{}, &models.Post{}, &models.Comment{})

utils.Logger.Info("Успешная передача моделей")

v1.Apies()

return nil

}

Листинг кода А.3 – Функция Apies

package v1

import (

"ProjectONE/internal/service"

"ProjectONE/pkg/utils"

"context"

"net/http"

"os"

"os/signal"

"syscall"

"time"

\_ "ProjectONE/docs"

"github.com/gin-gonic/gin"

swaggerFiles "github.com/swaggo/files" // Swagger JSON files

ginSwagger "github.com/swaggo/gin-swagger" // Swagger UI

)

func Apies() {

router := gin.Default()

// Маршрут для Swagger UI

router.GET("/swagger/\*any", ginSwagger.WrapHandler(swaggerFiles.Handler))

// Создание нового профиля

router.POST("/register", service.CreateProfile)

// Проверка профиля

router.POST("/login", login)

router.GET("/shutdown")

routerv1 := router.Group("/v1")

// Получение всех профилей

//router.GET("/profiles", service.GetProfiles)

profiles := routerv1.Group("/profiles")

profiles.Use(authMiddleware())

{

// Получение всех профилей

profiles.GET("", service.GetProfiles)

// Получение поста по ID

profiles.GET("/:id", service.GetProfileById)

// Обновление существующего профиля

profiles.PUT("/:id", service.UpdateProfile)

// Удаление профиля

profiles.DELETE("/:id", service.DeleteProfile)

}

posts := routerv1.Group("/posts")

posts.Use(authMiddleware())

{

// Получение всех постов

posts.GET("", service.GetPosts)

// Получение профиля по ID

posts.GET("/:id", service.GetPostById)

// Создание новой поста

posts.POST("", service.CreatePost)

Продолжение листинг кода А.3

// Обновление существующего поста

posts.PUT("/:id", service.UpdatePost)

// Удаление поста

posts.DELETE("/:id", service.DeletePost)

}

comments := routerv1.Group("/comments")

comments.Use(authMiddleware())

{

// Получение всех постов

comments.GET("", service.GetComments)

// Получение профиля по ID

comments.GET("/:id", service.GetCommentById)

// Создание новой поста

comments.POST("", service.CreateComment)

// Обновление существующего поста

comments.PUT("/:id", service.UpdateComment)

// Удаление поста

comments.DELETE("/:id", service.DeleteComment)

}

//router.Run(":8080") // Это прошлое

// Создаём кастомный сервер

srv := &http.Server{

Addr: ":8080",

Handler: router,

}

// Запуск сервера в горутине

go func() {

if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {

utils.Logger.Fatalf("ListenAndServe error: %v", err)

}

}()

// Канал для сигналов завершения

quit := make(chan os.Signal, 1)

signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)

// Блокировка до получения сигнала

sig := <-quit // программа здесь "ждёт" сигнал

utils.Logger.Warn("Завершение работы сервера...")

if sig == os.Interrupt {

utils.Logger.Info("Пойман сигнал (Ctrl + C):", sig)

} else {

utils.Logger.Info("Вызов из вне (Shutdown()):", sig)

}

if err := service.DumpDataToFile(); err != nil {

utils.Logger.Error("Ошибка при выгрузке данных:", err)

}

// Таймаут для graceful shutdown

Продолжение листинг кода А.3

ctx, cancel := context.WithTimeout(

context.Background(),

5\*time.Second,

)

defer cancel()

// Остановка сервера

if err := srv.Shutdown(ctx); err != nil {

utils.Logger.Fatal("Server forced to shutdown:", err)

}

utils.Logger.Println("Server exiting")

}

Листинг кода А.4 – Файл middle.go

package v1

import (

database "ProjectONE/internal/database/postgres"

"ProjectONE/internal/models"

"ProjectONE/pkg/utils"

"fmt"

"net/http"

"os"

"strings"

"time"

"github.com/dgrijalva/jwt-go"

"github.com/gin-gonic/gin"

password "github.com/vzglad-smerti/password\_hash"

)

var jwtKey = []byte(os.Getenv("JWT\_SECRET"))

type Credentials struct {

Nickname string `json:"nickname"`

Password string `json:"password"`

}

type Claims struct {

Nickname string `json:"nickname"`

jwt.StandardClaims

}

// generateToken создает новый JWT токен с данными пользователя и временем истечения

func generateToken(nickname string) (string, error) {

expirationTime := time.Now().Add(20 \* time.Hour)

claims := &Claims{

Nickname: nickname,

StandardClaims: jwt.StandardClaims{

ExpiresAt: expirationTime.Unix(),

},

}

token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)

ss, \_ := token.SignedString(jwtKey)

fmt.Println("\n\n", ss)

return token.SignedString(jwtKey)

}

// @Summary User login

// @Description Login using nickname and password to generate a JWT token

Продолжение листинг кода А.4

// @Tags sign

// @Accept json

// @Produce json

// @Param creds body Credentials true "User credentials"

// @Success 200 {object} statusResponse "JWT token"

// @Failure 400 {object} errorResponse "Invalid request"

// @Failure 401 {object} errorResponse "Unauthorized error"

// @Failure 500 {object} errorResponse "Internal server error"

// @Router /login [post]

func login(c \*gin.Context) {

var req models.LoginProfileRequest

// Привязываем JSON и валидируем

if err := c.ShouldBindJSON(&req); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"message": "Invalid input"})

return

}

var p models.Profile

// Ищем пользователя по nickname

if err := database.DbPostgres.

Where("nickname = ?", req.Nickname).

First(&p).Error; err != nil {

c.JSON(http.StatusUnauthorized, gin.H{"message": "Profile not found"})

return

}

fmt.Println(p.Nickname, "and", p.HashPassword, "and", req.Password)

// Проверяем, совпадает ли введенный пароль с сохраненным хешом

// if ok, err := password.Verify("ZzP5RstQI4RRETvy-CVKqYqLO6LFfeE=$#$16$#$1b7832c4a2be040c782b7dad3bfd78446af6be9db90331955276f452$#$afe31e3d2d01d7ce1279bf2a3aa7c1ae27c276a94088a759cced899bf34e3e15",

// "2"); !ok || err != nil {

if ok, err := password.Verify(p.HashPassword, req.Password); !ok || err != nil {

c.JSON(http.StatusUnauthorized, gin.H{"message": "Password error!!!"})

utils.Logger.Warn("Bad with password(middle.go|login|): ", ok, "||", err)

return

}

// Генерируем токен

token, err := generateToken(p.Nickname)

if err != nil {

utils.Logger.Error(err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "could not create token"})

utils.Logger.Warn("Could not create token(middle.go|login|): ", err)

return

}

// Отправляем ответ с токеном

c.JSON(http.StatusOK, gin.H{

"token": token,

"nickname": p.Nickname,

})

}

Продолжение листинг кода А.4

// authMiddleware - middleware для проверки валидности JWT токена

func authMiddleware() gin.HandlerFunc {

return func(c \*gin.Context) {

// Получаем JWT токен из заголовка

tokenString := strings.Split(c.GetHeader("Authorization"), " ")

if len(tokenString) != 2 {

c.JSON(http.StatusUnauthorized, gin.H{"message": "Unauthorized"})

c.Abort()

return

}

// Инициализируем структуру для хранения данных токена

claims := &Claims{}

// Пытаемся распарсить токен

token, err := jwt.ParseWithClaims(tokenString[1], claims, func(token \*jwt.Token) (interface{}, error) {

return jwtKey, nil

})

// Если токен невалиден или произошла ошибка, отклоняем запрос

if err != nil || !token.Valid {

c.JSON(http.StatusUnauthorized, gin.H{"message": "unauthorized"})

c.Abort()

return

}

// Если токен валиден, передаем выполнение дальше

c.Next()

}

}

Листинг кода А.5 – Файл postgres.go

package database

import (

"ProjectONE/pkg/utils"

"database/sql"

"fmt"

"log"

"os"

\_ "github.com/lib/pq" // Импортируем PostgreSQL драйвер

"gorm.io/driver/postgres"

"gorm.io/gorm"

"gorm.io/gorm/logger"

)

// DB — глобальная переменная для хранения подключения к базе данных

var DbPostgres \*gorm.DB

var sqlDB \*sql.DB

// Config — структура для хранения конфигурации подключения

type Config struct {

User string

Password string

Host string

Port string

DBName string

SSLMode string

}

Продолжение листинг кода А.5

// LoadConfigFromEnv загружает настройки базы данных из переменных окружения

func LoadConfigFromEnv() Config {

cfg := Config{

User: os.Getenv("DB\_USER"),

Password: os.Getenv("DB\_PASSWORD"),

Host: os.Getenv("DB\_HOST"),

Port: os.Getenv("DB\_PORT"),

DBName: os.Getenv("DB\_NAME"),

SSLMode: os.Getenv("DB\_SSLMODE"),

}

// utils.Logger.Printf("Проверка загрузки\nuser=%s password=%s host=%s port=%s dbname=%s sslmode=%s",

// cfg.User, cfg.Password, cfg.Host, cfg.Port, cfg.DBName, cfg.SSLMode)

return cfg

}

// Connect устанавливает соединение с базой данных

func Connect(cfg Config) error {

dsn := fmt.Sprintf(

"user=%s password=%s host=%s port=%s dbname=%s sslmode=%s",

cfg.User, cfg.Password, cfg.Host, cfg.Port, cfg.DBName, cfg.SSLMode,

)

utils.Logger.Printf("Проверка подключения\n%s", dsn)

var err error

DbPostgres, err = gorm.Open(postgres.Open(dsn), &gorm.Config{

Logger: logger.Default.LogMode(logger.Info),

})

if err != nil {

return fmt.Errorf("Ошибка подключения к базе данных: %w", err)

}

// Проверяем соединение

sqlDB, err := DbPostgres.DB()

if err != nil {

log.Fatal("Ошибка получения sql.DB: ", err)

}

if err := sqlDB.Ping(); err != nil {

log.Fatal("БД недоступна: ", err)

}

utils.Logger.Info("Успешное подключение к базе данных")

return nil

}

func CreateObjDB(dst ...interface{}) {

// dst = &models.Profile{}, &models.Post{}, &models.Comment{}

// fmt.Println("ЫЫЫЫЫ ЫЫЫЫЫЫЫ ЫЫЫЫ ЫЫЫЫЫ", dst)

// for \_, obj := range dst {

// // Пример с reflection

// val := reflect.ValueOf(obj)

// if val.Kind() == reflect.Ptr {

// val = val.Elem()

// }

// fmt.Printf("Тип: %v, Значение: %+v\n", val.Type(), val.Interface())

// }

if err := DbPostgres.AutoMigrate(dst...); err != nil {

log.Fatalf("Ошибка миграции: %v", err)

}

}

// Close закрывает соединение с базой данных

func Close() error {

Продолжение листинг кода А.5

if DbPostgres != nil && sqlDB != nil {

if err := sqlDB.Close(); err != nil {

return err

}

}

return nil

}

Листинг кода А.6 – Модель Profile

package models

import (

"time"

"gorm.io/gorm"

)

type Profile struct {

Id uint `json:"id" gorm:"primaryKey"`

Nickname string `json:"nickname" gorm:"type:varchar(30);not null;unique"`

HashPassword string `json:"hashpassword" gorm:"type:text;not null"`

Status bool `json:"status" gorm:"default:true"`

AccessLevel uint8 `json:"accesslevel" gorm:"default:1;index"`

Firstname string `json:"firstname" gorm:"type:varchar(100);not null"`

Lastname string `json:"lastname" gorm:"type:varchar(100);not null"`

CreatedAt time.Time `json:"createdat" gorm:"autoCreateTime"`

UpdatedAt time.Time `json:"updatedat" gorm:"autoUpdateTime"`

DeletedAt gorm.DeletedAt `json:"deletedat" gorm:"index"`

Posts []Post `json:"posts" gorm:"foreignKey:ProfileID"`

Comments []Comment `json:"comments" gorm:"foreignKey:ProfileID"`

}

type CreateProfileRequest struct {

Nickname string `json:"nickname"`

Password string `json:"password"`

AccessLevel uint8 `json:"access\_level"`

Firstname string `json:"firstname"`

Lastname string `json:"lastname"`

}

type LoginProfileRequest struct {

Nickname string `json:"nickname" binding:"required"`

Password string `json:"password" binding:"required"`

}

Листинг кода А.7 – Модель Post

package models

import (

"time"

"gorm.io/gorm"

)

type Post struct {

Id uint `json:"id" gorm:"primaryKey"`

Title string `json:"title" gorm:"type:varchar(255);not null"`

Продолжение листинг кода А.7

Description string `json:"description" gorm:"type:text;not null"`

Likes int `json:"likes" gorm:"default:0"`

ProfileID uint `json:"profile\_id"` // Внешний ключ для профиля

DatePublication time.Time `json:"date\_publication" gorm:"autoCreateTime"`

DateLastModified time.Time `json:"date\_last\_modified" gorm:"autoUpdateTime"`

DeletedAt gorm.DeletedAt `json:"deletedat" gorm:"index"`

Profile \*Profile `json:"profile" gorm:"foreignKey:ProfileID"`

Comments []Comment `json:"comments" gorm:"foreignKey:PostID"`

}

Листинг кода А.8 – Модель Comment

package models

import (

"time"

"gorm.io/gorm"

)

type Comment struct {

Id uint `json:"id" gorm:"primaryKey"`

Text string `json:"text\_comment" gorm:"type:text;not null"`

ProfileID uint `json:"profile\_id"` // Внешний ключ для профиля

PostID uint `json:"post\_id"` // Внешний ключ для поста

DatePublication time.Time `json:"date\_publication" gorm:"autoCreateTime"`

DateLastModified time.Time `json:"date\_last\_modified" gorm:"autoUpdateTime"`

DeletedAt gorm.DeletedAt `json:"deletedat" gorm:"index"`

Profile \*Profile `json:"profile" gorm:"foreignKey:ProfileID"`

Post \*Post `json:"post" gorm:"foreignKey:PostID"`

}

Листинг кода А.9 – Сервис Profile

package service

import (

database "ProjectONE/internal/database/postgres"

"ProjectONE/internal/models"

"ProjectONE/pkg/utils"

"errors"

"net/http"

"strconv"

password "github.com/vzglad-smerti/password\_hash"

"gorm.io/gorm"

"github.com/gin-gonic/gin"

)

var profiles = []models.Profile{}

// @Summary Get profiles

// @Security ApiKeyAuth

Продолжение листинг кода А.8

// @Description Retrieve a list of profiles for a specific account by account ID with pagination

// @Tags authors

// @Accept json

// @Produce json

// @Param page query int false "Page number (default: 1)"

// @Param limit query int false "Number of profiles per page (default: 5)"

// @Success 200 {array} models.Profile

// @Failure 400 {object} errorResponse

// @Failure 404 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/profiles [get]

func GetProfiles(c \*gin.Context) {

// Получение параметров из запроса

page, \_ := strconv.Atoi(c.DefaultQuery("page", "1")) // Номер страницы, по умолчанию 1

limit, \_ := strconv.Atoi(c.DefaultQuery("limit", "5")) // Количество элементов на странице, по умолчанию 5

if page < 1 {

page = 1

}

if limit < 1 {

limit = 5

}

offset := (page - 1) \* limit // Вычисление смещения

var profiles []models.Profile

// Использование GORM для выборки с лимитом и смещением

err := database.DbPostgres.Limit(limit).Offset(offset).Find(&profiles).Error

if err != nil {

utils.Logger.Panic(err)

return

}

//utils.Logger.Printf("%v", profiles)

c.JSON(http.StatusOK, profiles)

}

// @Summary Get profile by ID

// @Security ApiKeyAuth

// @Description Retrieve a specific profile by its ID

// @Tags authors

// @Accept json

// @Produce json

// @Param id path int true "Account ID"

// @Success 200 {object} models.Profile

// @Failure 400 {object} errorResponse

// @Failure 404 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/profiles/{id} [get]

func GetProfileById(c \*gin.Context) {

// Получаем параметр id из запроса

id := c.Param("id")

// Использование GORM для поиска профиля по ID

Продолжение листинг кода А.8

var profile models.Profile

err := database.DbPostgres.First(&profile, id).Error

if err != nil {

if err == gorm.ErrRecordNotFound {

c.JSON(http.StatusNotFound, gin.H{"message": "profile not found"})

} else {

c.JSON(http.StatusInternalServerError, gin.H{"message": "internal server error"})

}

utils.Logger.Panic("Неудачный запрос|(profile\_handler.go|GetProfileById|):", err)

return

}

// Возвращаем профиль в ответе

c.JSON(http.StatusOK, profile)

}

// @Summary Create a new profile

// @Description Creates a new profile by accepting profile details in the request body

// @Tags sign

// @Accept json

// @Produce json

// @Param profile body models.Profile true "Profile data"

// @Success 201 {object} models.Profile

// @Failure 400 {object} errorResponse

// @Failure 409 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /register [post]

func CreateProfile(c \*gin.Context) {

req := models.CreateProfileRequest{}

// Парсим JSON из тела запроса в структуру Profile

if err := c.BindJSON(&req); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"message": "invalid request"})

utils.Logger.Panic("Data is bad|(profile\_handler.go|CreateProfile|)|:", err)

return

}

// Проверка, есть ли уже такой nickname

var existing models.Profile

if err := database.DbPostgres.Where("nickname = ?", req.Nickname).First(&existing).Error; err == nil {

// Нашли совпадение

c.JSON(http.StatusConflict, gin.H{"message": "nickname already taken"})

return

} else if !errors.Is(err, gorm.ErrRecordNotFound) {

// Ошибка при запросе

c.JSON(http.StatusInternalServerError, gin.H{"message": "database error"})

utils.Logger.Error("DB error when checking nickname (profile\_handler.go|CreateProfile):", err)

return

}

// Хеширование пароля

hash, err := password.Hash(req.Password)

Продолжение листинг кода А.9

if err != nil {

c.JSON(http.StatusBadRequest, gin.H{"message": "Problem with password hashing"})

utils.Logger.Panic("Hash wasn't working(profile\_handler.go|CreateProfile|):", err)

return

}

p := models.Profile{

Nickname: req.Nickname,

HashPassword: hash,

AccessLevel: req.AccessLevel,

Firstname: req.Firstname,

Lastname: req.Lastname,

}

// Создаем новый профиль в базе данных с использованием GORM

if err := database.DbPostgres.Create(&p).Error; err != nil {

utils.Logger.Panic("Insert isn't done(profile\_handler.go|CreateProfile|):", err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "internal server error"})

return

}

// Отправляем успешный ответ с созданным профилем

c.JSON(http.StatusCreated, p)

}

// @Summary Update an existing profile

// @Security ApiKeyAuth

// @Description Update an existing profile's information by profile ID

// @Tags authors

// @Accept json

// @Produce json

// @Param id path int true "Profile ID"

// @Param profile body models.Profile true "Updated profile data"

// @Success 202 {object} models.Profile

// @Failure 400 {object} errorResponse

// @Failure 404 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/profiles/{id} [put]

func UpdateProfile(c \*gin.Context) {

id := c.Param("id")

req := models.CreateProfileRequest{}

// Парсим JSON из тела запроса

if err := c.BindJSON(&req); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"message": "invalid request"})

utils.Logger.Panic("Data is bad|(profile\_handler.go|UpdateProfile|)|:", err)

return

}

// Проверка, есть ли такой id

var existing models.Profile

if err := database.DbPostgres.Where("id = ?", id).First(&existing).Error; err != nil {

if errors.Is(err, gorm.ErrRecordNotFound) {

Продолжение листинг кода А.9

c.JSON(http.StatusNotFound, gin.H{"message": "profile not found"})

return

}

c.JSON(http.StatusInternalServerError, gin.H{"message": "database error"})

utils.Logger.Error("DB error when checking profile (profile\_handler.go|UpdateProfile):", err)

return

}

var hash string

var err error

if req.Password != "" {

// Хеширование пароля

hash, err = password.Hash(req.Password)

if err != nil {

c.JSON(http.StatusBadRequest, gin.H{"message": "Problem with password hashing"})

utils.Logger.Panic("Hash wasn't working(profile\_handler.go|CreateProfile|):", err)

return

}

}

p := models.Profile{

Nickname: req.Nickname,

HashPassword: hash,

AccessLevel: req.AccessLevel,

Firstname: req.Firstname,

Lastname: req.Lastname,

}

// Обновляем профиль по ID с использованием GORM

if err := database.DbPostgres.Model(&models.Profile{}).Where("id = ?", id).Updates(p).Error; err != nil {

utils.Logger.Panic("Update isn't done(profile\_handler.go|UpdateProfile|):", err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "internal server error"})

return

}

// Использование GORM для поиска профиля по ID

var profile models.Profile

if err := database.DbPostgres.First(&profile, id).Error; err != nil {

if err == gorm.ErrRecordNotFound {

c.JSON(http.StatusNotFound, gin.H{"message": "profile not found"})

} else {

c.JSON(http.StatusInternalServerError, gin.H{"message": "internal server error"})

}

utils.Logger.Panic("Неудачный запрос|(profile\_handler.go|GetProfileById|):", err)

return

}

// Отправляем успешный ответ с обновленным профилем

c.JSON(http.StatusAccepted, profile)

}

Продолжение листинг кода А.9

// @Summary Delete a profile by ID

// @Security ApiKeyAuth

// @Description Delete a profile from the system by its ID

// @Tags authors

// @Accept json

// @Produce json

// @Param id path int true "Profile ID"

// @Success 202 {object} string

// @Failure 404 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/profiles/{id} [delete]

func DeleteProfile(c \*gin.Context) {

id := c.Param("id")

// Удаляем профиль по ID с использованием GORM

if err := database.DbPostgres.Delete(&models.Profile{}, id).Error; err != nil {

c.JSON(http.StatusNotFound, gin.H{"message": "profile wasn't deleted"})

utils.Logger.Error("Delete isn't done(profile\_handler.go|DeleteProfile|):", err)

return

}

// Отправляем успешный ответ о удалении

c.JSON(http.StatusAccepted, gin.H{"message": "profile was deleted"})

}

Листинг кода А.10 – Сервис Post

package service

import (

database "ProjectONE/internal/database/postgres"

"ProjectONE/internal/models"

"ProjectONE/pkg/utils"

"net/http"

"strconv"

"time"

"github.com/gin-gonic/gin"

"gorm.io/gorm"

)

// @Summary Get all posts

// @Security ApiKeyAuth

// @Description Retrieve a list of all posts in the system

// @Tags posts

// @Accept json

// @Produce json

// @Param page query int false "Page number (default: 1)"

// @Param limit query int false "Number of posts per page (default: 5)"

// @Success 200 {array} models.Post

// @Failure 500 {object} errorResponse

// @Router /v1/posts [get]

func GetPosts(c \*gin.Context) {

page, \_ := strconv.Atoi(c.DefaultQuery("page", "1"))

limit, \_ := strconv.Atoi(c.DefaultQuery("limit", "5"))

if page < 1 {

page = 1

Продолжение листинг кода А.10

}

if limit < 1 {

limit = 5

}

offset := (page - 1) \* limit

var posts []models.Post

if err := database.DbPostgres.Limit(limit).Offset(offset).Find(&posts).Error; err != nil {

utils.Logger.Panic("Failed to fetch posts:", err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "Failed to fetch posts"})

return

}

c.JSON(http.StatusOK, posts)

}

// @Summary Get a post by ID

// @Security ApiKeyAuth

// @Description Retrieve a post's details by its unique ID

// @Tags posts

// @Accept json

// @Produce json

// @Param id path int true "Post ID"

// @Success 200 {object} models.Post

// @Failure 404 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/posts/{id} [get]

func GetPostById(c \*gin.Context) {

id := c.Param("id")

var post models.Post

if err := database.DbPostgres.First(&post, id).Error; err != nil {

if err == gorm.ErrRecordNotFound {

c.JSON(http.StatusNotFound, gin.H{"message": "Post not found"})

} else {

c.JSON(http.StatusInternalServerError, gin.H{"message": "Internal server error"})

}

utils.Logger.Panic("Failed to fetch post by ID:", err)

return

}

c.JSON(http.StatusOK, post)

}

// @Summary Create a new post

// @Security ApiKeyAuth

// @Description Create a new post with title, description, and author information

// @Tags posts

// @Accept json

// @Produce json

// @Param post body models.Post true "New post data"

// @Success 201 {object} models.Post

// @Failure 400 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/posts [post]

func CreatePost(c \*gin.Context) {

var p models.Post

Продолжение листинг кода А.10

if err := c.ShouldBindJSON(&p); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"message": "Invalid request"})

utils.Logger.Panic("Invalid post data:", err)

return

}

if err := database.DbPostgres.Create(&p).Error; err != nil {

utils.Logger.Panic("Failed to create post:", err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "Failed to create post"})

return

}

c.JSON(http.StatusCreated, p)

}

// @Summary Update an existing post

// @Security ApiKeyAuth

// @Description Update the details of an existing post by its ID

// @Tags posts

// @Accept json

// @Produce json

// @Param id path int true "Post ID"

// @Param post body models.Post true "Updated post data"

// @Success 202 {object} models.Post

// @Failure 400 {object} errorResponse

// @Failure 404 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/posts/{id} [put]

func UpdatePost(c \*gin.Context) {

id := c.Param("id")

var p models.Post

if err := c.ShouldBindJSON(&p); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"message": "Invalid request"})

utils.Logger.Panic("Invalid post data for update:", err)

return

}

// Обновляем только нужные поля

p.DateLastModified = time.Now() // Функция для текущего времени, если есть

if err := database.DbPostgres.Model(&models.Post{}).Where("id = ?", id).Updates(p).Error; err != nil {

utils.Logger.Panic("Failed to update post:", err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "Failed to update post"})

return

}

c.JSON(http.StatusAccepted, p)

}

// @Summary Delete a post by ID

// @Security ApiKeyAuth

// @Description Delete an existing post by its unique ID

// @Tags posts

// @Accept json

// @Produce json

// @Param id path int true "Post ID"

// @Success 202 {object} string

// @Failure 404 {object} errorResponse

Продолжение листинг кода А.10

// @Failure 500 {object} errorResponse

// @Router /v1/posts/{id} [delete]

func DeletePost(c \*gin.Context) {

id := c.Param("id")

if err := database.DbPostgres.Delete(&models.Post{}, id).Error; err != nil {

utils.Logger.Panic("Failed to delete post:", err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "Failed to delete post"})

return

}

c.JSON(http.StatusAccepted, gin.H{"message": "Post was deleted"})

}

Листинг кода А.11 – Сервис Comment

package service

import (

database "ProjectONE/internal/database/postgres"

"ProjectONE/internal/models"

"ProjectONE/pkg/utils"

"net/http"

"time"

"github.com/gin-gonic/gin"

)

// @Summary Get all comments

// @Security ApiKeyAuth

// @Description Retrieve a list of all comments from the database

// @Tags comments

// @Produce json

// @Success 200 {array} models.Comment

// @Failure 500 {object} errorResponse

// @Router /v1/comments [get]

func GetComments(c \*gin.Context) {

var comments []models.Comment

result := database.DbPostgres.Find(&comments)

if result.Error != nil {

utils.Logger.Error("Failed to get comments:", result.Error)

c.JSON(http.StatusInternalServerError, gin.H{"message": "failed to get comments"})

return

}

c.JSON(http.StatusOK, comments)

}

// @Summary Get comment by ID

// @Security ApiKeyAuth

// @Description Retrieve a specific comment by its ID from the database

// @Tags comments

// @Produce json

// @Param id path int true "Comment ID"

// @Success 200 {object} models.Comment

// @Failure 404 {object} errorResponse

// @Router /v1/comments/{id} [get]

func GetCommentById(c \*gin.Context) {

id := c.Param("id")

var comment models.Comment

Продолжение листинг кода А.11

result := database.DbPostgres.First(&comment, id)

if result.Error != nil {

utils.Logger.Error("Comment not found:", result.Error)

c.JSON(http.StatusNotFound, gin.H{"message": "comment not found"})

return

}

c.JSON(http.StatusOK, comment)

}

// @Summary Create a comment

// @Security ApiKeyAuth

// @Description Add a new comment to the database

// @Tags comments

// @Accept json

// @Produce json

// @Param comment body models.Comment true "Comment Data"

// @Success 201 {object} models.Comment

// @Failure 400 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/comments [post]

func CreateComment(c \*gin.Context) {

var comment models.Comment

if err := c.ShouldBindJSON(&comment); err != nil {

utils.Logger.Error("Invalid comment data:", err)

c.JSON(http.StatusBadRequest, gin.H{"message": "invalid request"})

return

}

comment.DatePublication = time.Now()

if err := database.DbPostgres.Create(&comment).Error; err != nil {

utils.Logger.Error("Failed to create comment:", err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "failed to create comment"})

return

}

c.JSON(http.StatusCreated, comment)

}

// @Summary Update a comment

// @Security ApiKeyAuth

// @Description Update an existing comment's information by its ID

// @Tags comments

// @Accept json

// @Produce json

// @Param id path int true "Comment ID"

// @Param comment body models.Comment true "Updated Comment Data"

// @Success 202 {object} models.Comment

// @Failure 400 {object} errorResponse

// @Failure 404 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/comments/{id} [put]

func UpdateComment(c \*gin.Context) {

id := c.Param("id")

existingComment := models.Comment{}

if err := database.DbPostgres.First(&existingComment, id).Error; err != nil {

utils.Logger.Error("Comment not found:", err)

Продолжение листинг кода А.11

c.JSON(http.StatusNotFound, gin.H{"message": "comment not found"})

return

}

var updatedComment models.Comment

if err := c.ShouldBindJSON(&updatedComment); err != nil {

utils.Logger.Error("Invalid update data:", err)

c.JSON(http.StatusBadRequest, gin.H{"message": "invalid request"})

return

}

existingComment.Text = updatedComment.Text

if err := database.DbPostgres.Save(&existingComment).Error; err != nil {

utils.Logger.Error("Failed to update comment:", err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "failed to update comment"})

return

}

c.JSON(http.StatusAccepted, existingComment)

}

// @Summary Delete a comment

// @Security ApiKeyAuth

// @Description Remove a comment from the database by its ID

// @Tags comments

// @Produce json

// @Param id path int true "Comment ID"

// @Success 202 {object} string

// @Failure 404 {object} errorResponse

// @Failure 500 {object} errorResponse

// @Router /v1/comments/{id} [delete]

func DeleteComment(c \*gin.Context) {

id := c.Param("id")

var comment models.Comment

if err := database.DbPostgres.First(&comment, id).Error; err != nil {

utils.Logger.Error("Comment not found:", err)

c.JSON(http.StatusNotFound, gin.H{"message": "comment not found"})

return

}

if err := database.DbPostgres.Delete(&comment).Error; err != nil {

utils.Logger.Error("Failed to delete comment:", err)

c.JSON(http.StatusInternalServerError, gin.H{"message": "failed to delete comment"})

return

}

c.JSON(http.StatusAccepted, gin.H{"message": "comment was deleted"})

}

Листинг кода А.12 – Файл workWithData.go

package service

import (

database "ProjectONE/internal/database/postgres"

"ProjectONE/internal/models"

"ProjectONE/pkg/utils"

"encoding/json"

"os"

Продолжение листинг кода А.12

"time"

)

// DumpDataToFile выгружает все записи из таблиц в файл

func DumpDataToFile() error {

utils.Logger.Info("Начинаем выгрузку данных из БД...")

var profiles []models.Profile

var posts []models.Post

var comments []models.Comment

if err := database.DbPostgres.Find(&profiles).Error; err != nil {

return err

}

if err := database.DbPostgres.Find(&posts).Error; err != nil {

return err

}

if err := database.DbPostgres.Find(&comments).Error; err != nil {

return err

}

data := map[string]interface{}{

"timestamp": time.Now().Format(time.RFC3339),

"profiles": profiles,

"posts": posts,

"comments": comments,

}

file, err := os.Create("dump.json")

if err != nil {

return err

}

defer file.Close()

encoder := json.NewEncoder(file)

encoder.SetIndent("", " ")

if err := encoder.Encode(data); err != nil {

return err

}

utils.Logger.Info("Данные успешно выгружены в dump.json")

return nil

}

Листинг кода А.13 – Файл logger.go

package utils

import (

"os"

"time"

"github.com/sirupsen/logrus"

)

// Logger — это глобальный логгер для всего приложения

var Logger \*logrus.Logger

// InitLogger — функция инициализации логгера

func InitLogger(logFile string) {

Logger = logrus.New() // Создаем новый логгер

// Устанавливаем формат вывода (JSON или текст)

Продолжение листинг кода А.13

Logger.SetFormatter(&logrus.JSONFormatter{

TimestampFormat: time.RFC3339, // Формат времени

})

// Устанавливаем уровень логирования (Debug, Info, Warn, Error, Fatal, Panic)

Logger.SetLevel(logrus.DebugLevel)

// Настраиваем вывод логов

// Если указан logFile, логи будут записываться в файл

if logFile != "" {

file, err := os.OpenFile(logFile, os.O\_CREATE|os.O\_WRONLY|os.O\_APPEND, 0666)

if err != nil {

Logger.Fatalf("Не удалось открыть файл логов: %v", err)

}

Logger.SetOutput(file)

return

}

// Если файл не указан, выводим логи в стандартный вывод (консоль)

Logger.SetOutput(os.Stdout)

}