Geekbrains

Разработка системы управления задачами с использованием Java Spring, MySQL и Angular

Программа:
Разработчик — Веб-разработка на Java
Харламенко Всеволод Вадимович

г. Белгород
2025

Содержание

Введение

Что из себя представляет проект

Проект представляет собой систему управления задачами, разработанную с использованием технологий Java Spring для серверной части, MySQL для базы данных и Angular для фронтенда.

Система позволяет администратору назначать задачи пользователям, отслеживать их выполнение, устанавливать статусы задач и добавлять комментарии к ним.

Пользователи могут просматривать свои задачи, изменять их статусы и добавлять комментарии по ходу выполнения.

Система включает функционал регистрации и авторизации, что позволяет пользователям и администраторам управлять своими учетными записями и взаимодействовать с системой.

Обоснование темы проекта

Эффективное управление задачами является неотъемлемой частью успешной работы любой команды, особенно в организациях с большими объемами проектов. Существующие решения часто не предоставляют гибкости или нужной персонализации.

Разработка собственной системы управления задачами решает эту проблему, позволяя обеспечить оптимизацию рабочего процесса и улучшить взаимодействие между пользователями. Система, включающая возможность назначения задач, изменения их статусов и добавления комментариев, дает возможность оперативно контролировать и отслеживать выполнение работы.

Цель проекта

Целью проекта является разработка системы управления задачами, которая позволит администраторам эффективно распределять задачи между пользователями, отслеживать их выполнение, изменять статусы задач и добавлять к ним комментарии.

Система будет способствовать повышению прозрачности выполнения задач и улучшению координации между членами команды.

План работы

1. Анализ современных решений для управления задачами.
2. Проектирование архитектуры приложения, включая систему назначения задач, управления статусами и комментариями.
3. Реализация функционала регистрации пользователей и администраторов, а также системы управления учетными записями.
4. Разработка и интеграция интерфейса для взаимодействия пользователей с системой.
5. Тестирование функционала системы управления задачами.
6. Подведение итогов и подготовка рекомендаций для дальнейшего улучшения системы.

Какую проблему будет решать проект

Проект решает проблему эффективного распределения задач в команде, позволяя отслеживать их выполнение и статусы в реальном времени.

Возможность добавлять комментарии к задачам повышает коммуникацию внутри команды и помогает быстрее реагировать на изменения.

Полезный опыт для решения задачи

У меня был небольшой опыт работы с фреймворком Spring и управлением базами данных с использованием MySQL.

Также я самостоятельно изучал Angular в процессе обучения на курсе. В результате я решил применить свои знания и навыки для создания одного большого проекта — системы управления задачами (Task Management System), используя Spring для серверной части, MySQL для работы с базой данных и Angular для frontend.

Инструменты, используемые в проекте

- **IntelliJ IDEA** — для разработки серверной части на Java.
- **Visual Studio Code** — для разработки фронтенда.
- **MySQL, DBeaver** — для работы с базой данных.
- **Postman** — для тестирования API.
- **Git, GitHub** — для управления версиями и командной работы.

Основные технологии, планируемые для использования

- **Backend**: Java (Spring Boot, Spring Security, Hibernate);
- **Frontend**: HTML, SCSS, TypeScript, Angular.
- **База данных**: MySQL
- **Тестирование**: Postman для тестирования API.

Состав команды

Проект выполнялся индивидуально. В процессе работы я исполнял все роли:

- **Проектировщик**: разработка архитектуры приложения и базы данных.
- **Дизайнер**: создание пользовательского интерфейса и визуального стиля.
- **Разработчик**: реализация серверной и клиентской частей системы.
- **Тестировщик**: проверка корректности работы системы и тестирование функционала.

Глава 1. Основы разработки систем управления задачами

Система управления задачами: особенности, структура и функции

Общие принципы систем управления задачами

Системы управления задачами (Task Management Systems) предназначены для упрощения планирования и выполнения задач. Они помогают пользователям, командам и организациям создавать рабочие процессы, отслеживать выполнение задач и эффективно распределять ресурсы. Главной целью таких систем является повышение продуктивности и обеспечение прозрачности работы.

Типичные функциональные возможности систем управления задачами:

- **Создание задач** с возможностью добавления подробных описаний, сроков выполнения и приоритетов.
- **Распределение задач** между пользователями или группами пользователей.
- **Отслеживание прогресса** выполнения задач через различные статусы (например, "Новая", "В работе", "Завершена").
- **Управление сроками** — возможность установки дедлайнов и напоминаний.
- **Добавление комментариев**, что упрощает взаимодействие между участниками процесса.
- **Отчеты и аналитика** по выполнению задач и проектам.

Популярные системы управления задачами

Существует множество решений для управления задачами, и каждая из них предлагает свой набор уникальных функций и подходов. Рассмотрим несколько популярных систем, которые используются в различных областях:

- **Trello** (фото пример)
    - **Описание**: Облачная программа для управления проектами небольших групп, разработанная Fog Creek Software.

      Trello использует парадигму для управления проектами, известную как канбан, метод, который первоначально был популяризирован Toyota в 1980-х для управления цепочками поставок. Trello использует freemium-бизнес-модель, платные услуги были запущены в 2013 году.

      В 2017 году куплен Atlassian за 425 миллионов долларов США.

    - **Преимущества**: Простота в использовании, возможность быстро создавать и управлять задачами через drag-and-drop интерфейс.

      Подходит для небольших команд и личных проектов.

    - **Недостатки**: Недостаток сложных функций для крупных команд и предприятий, ограниченные возможности для отчетности и аналитики.
    - **Пример использования**: Подходит для личных проектов, небольших стартапов, а также для ведения отдельных рабочих процессов, например, для разработки контента.
- **Asana** (фото пример)
    - **Описание**: Asana — это мощная система для управления проектами, которая включает функции для отслеживания задач, проектов и рабочих процессов.

      Она предлагает различные виды представления данных (списки, доски, календари) и мощные инструменты для анализа и отчетности.

    - **Преимущества**: Гибкость, возможность настраивать рабочие процессы, интеграция с другими сервисами (например, Slack, Google Drive).
    - **Недостатки**:

      1) *Ограничения бесплатной версии*: Бесплатная версия Asana имеет ограниченный набор функций и подходит в основном для небольших команд.

      2) *Сложность некоторых функций*: Несмотря на интуитивный интерфейс, некоторые функции могут быть сложными для понимания и освоения, особенно для новых пользователей.

      3) *Отсутствие автономного приложения*: Asana является веб-приложением, и для работы с ней требуется подключение к интернету, что может быть неудобным в условиях ограниченного доступа к сети.

    - **Пример использования**: Идеальна для крупных команд и организаций, где требуется отслеживание множества задач в разных проектах и отделах.
- **Jira** (фото пример)
    - **Описание**: Jira — одна из самых популярных систем для управления проектами в области разработки программного обеспечения, созданная компанией Atlassian.

      Она предоставляет мощные инструменты для отслеживания задач, багов, выполнения спринтов и управления проектами с использованием методологии Scrum.

    - **Преимущества**: Поддержка гибких рабочих процессов, интеграция с другими инструментами разработки (например, Bitbucket), высокие возможности настройки.

    - **Недостатки**: Дорогая для малого бизнеса, высокая сложность интерфейса, большое количество опций, которые могут перегружать пользователей.
    - **Пример использования**: Особенно полезна для команд разработчиков, работающих по методологии Agile и Scrum.
- **Monday.com** (фото пример)
    - **Описание**: Monday.com — это универсальная платформа для управления проектами и задачами, которая позволяет настроить различные рабочие процессы для разных типов пользователей.
    - **Преимущества**: Интуитивно понятный интерфейс, поддержка различных видов представления данных (доски, диаграммы, календари), гибкость настройки рабочих процессов.
    - **Недостатки**: Платная модель подписки может быть дорогой для малого бизнеса.
    - **Пример использования**: Подходит для широкого спектра задач — от управления проектами и задачами до отслеживания времени и создания отчетности.

Сравнение популярных систем

Для того чтобы понять, почему важно разрабатывать собственную систему управления задачами, стоит выделить основные преимущества и недостатки существующих решений.

Сравним несколько популярных платформ по ключевым аспектам:

Критерий	Trello	Asana	Jira	Monday.com
Простота использования	Высокая	Средняя	Низкая	Средняя
Гибкость настройки	Низкая	Средняя	Высокая	Средняя
Подходит для крупных команд	Нет	Да	Да	Да
Поддержка отчетности	Ограниченная	Хорошая	Отличная	Хорошая
Стоимость	Бесплатно (ограничено)	Платно	Платно	Платно

Проблемы существующих решений

Несмотря на большое количество доступных инструментов для управления задачами, все они имеют свои ограничения:

- Простота использования: Некоторые решения, такие как Jira, могут быть сложными для новых пользователей, особенно в малых командах. Это может затруднить внедрение и обучение.

- Низкая гибкость: Решения вроде Trello не всегда позволяют настроить сложные рабочие процессы, что ограничивает их использование в крупных проектах с несколькими этапами.

- Высокая стоимость: Для небольших команд или стартапов стоимость профессиональных версий таких платформ, как Asana и Jira, может быть слишком высокой.

- Отсутствие персонализации: Многие решения не позволяют полностью настроить систему под специфические нужды пользователей или компаний.

Преимущества создания собственной системы

Разработка собственной системы управления задачами, которая будет отвечать специфическим требованиям проекта или организации, позволяет:

- Учитывать особенности и уникальные рабочие процессы компании.
- Получить полную гибкость в настройке интерфейса и функционала.
- Избежать излишних затрат на дорогие платные решения.
- Разработать систему, ориентированную на нужды конкретных пользователей, с возможностью дальнейшей кастомизации и масштабирования.

Таким образом, текущий рынок предлагает разнообразие систем для управления задачами, однако часто они не могут удовлетворить все требования конкретных пользователей. Разработка собственной системы может решить эти проблемы, обеспечив более точную настройку под нужды проекта.

Анализ требований к системе управления задачами

Для разработки эффективной системы управления задачами крайне важно четко определить как функциональные, так и нефункциональные требования. Они помогают создать систему, которая будет удобной, эффективной и подходящей для использования в реальных условиях.

1. Определение ключевых функциональных и нефункциональных требований

Функциональные требования

Функциональные требования определяют, что именно должна делать система. В контексте системы управления задачами они включают:

- **Создание и редактирование задач:** Система должна позволять пользователю создавать задачи с заглавием, подробными описаниями, сроками выполнения и приоритетами. Также должна быть возможность редактировать эти задачи в процессе выполнения.
    - Пример: Пользователь может создать задачу "Разработать дизайн главной страницы" с датой завершения 30 января 2025г. и установить приоритет "Высокий".

- **Управление задачами и их состояниями**: В системе должна быть возможность отслеживать статус задач: "В процессе", "Завершена". Пользователь должен иметь возможность перемещать задачи между этими статусами.
    - Пример: После того как дизайнер завершил задачу, он изменит статус на "Завершена".

- **Распределение задач между пользователями:** Система должна позволять назначать задачи конкретным пользователям или группам пользователей.
    - Пример: Руководитель проекта назначает задачу "Проверка функционала" разработчику, указав его в поле "Ответственный".

- **Отчеты и аналитика:** Система должна предоставлять функционал для создания отчетов о выполнении задач, времени, затраченном на их выполнение, и эффективности работы команды.
    - Пример: Руководитель может создать отчет о выполнении задач за месяц, чтобы проанализировать продуктивность команды.

Нефункциональные требования

Нефункциональные требования определяют, как система должна работать, какие качества она должна иметь, чтобы быть удобной, безопасной и эффективной в использовании:

- **Производительность**: Система должна быть способна обрабатывать большое количество задач и пользователей без значительных задержек. Особенно это важно для организаций с большим количеством сотрудников.
    - Пример: При загрузке списка задач с сотнями записей не должно быть заметных задержек.

- **Удобство интерфейса**: Интерфейс системы должен быть интуитивно понятным и доступным для пользователей с разным уровнем технической подготовки.
    - Пример: Пользователь без опыта работы с такими системами должен легко разобраться, как создать и назначить задачу.

- **Безопасность**: Система должна обеспечивать защиту данных пользователей, предотвращать несанкционированный доступ и поддерживать надежную аутентификацию и авторизацию.

- **Масштабируемость**: Система должна быть масштабируемой, чтобы с увеличением количества пользователей и задач не возникало проблем с производительностью.
    - Пример: Система должна сохранять свою производительность при добавлении новых сотрудников и создании десятков тысяч задач.

2. Разработка функциональной модели системы

Функциональная модель описывает основные действия и возможности, которые система должна предоставлять своим пользователям. В моем случае модель будет включать:

Авторизация и аутентификация:

- Пользователи должны иметь возможность безопасно входить в систему, используя свои учетные данные (логин и пароль).

Управление задачами:

- Пользователи могут редактировать статус задачи.
- Каждый пользователь может видеть только те задачи, к которым у него есть доступ, в зависимости от его роли.

Распределение задач и управление приоритетами:

- Задачи могут быть назначены одному или нескольким пользователям.
- Задачи могут иметь разные приоритеты: низкий, средний, высокий и критический.

Уведомления и напоминания:

- Уведомления о назначении задачи, изменении статуса, приближающихся дедлайнах.
- Напоминания о задачах, срок выполнения которых подходит.

Панель управления для администраторов:

- Администратор может управлять пользовательскими задачами, изменять приоритет задачи и т.д.

3. Выбор основных пользователей системы и их ролей

Для эффективного управления системой необходимо выделить несколько типов пользователей, каждый из которых будет выполнять определенные функции.

Роли пользователей:

1. **Администратор**
    - Основная роль, которая имеет полный доступ ко всем функциям системы.
    - Может создавать и редактировать задачи, назначать пользователей на задачи, изменять настройки системы.
2. **Исполнитель**
    - Пользователи, которые непосредственно работают с задачами, выполняют их в установленный срок.
    - Могут изменять статус задач, добавлять комментарии.
3. **Читатель (планирую реализовать в своем проекте)**
    - Пользователь с ограниченными правами. Может только просматривать задачи и их статусы, но не может редактировать их.
    - Подходит для ролей, где пользователи должны отслеживать прогресс, но не управлять задачами.

Пример сценариев использования для каждой роли:

- **Администратор**: Создает новую задачу для пользователя.
- **Исполнитель**: Получает задачу от администратора, изменяет статус задачи после выполнения, добавляет комментарии о выполнении работы.

Глава 2. Подготовка к разработке системы управления задачами

Задачей проекта является разработка системы Task Management, которая предназначена для индивидуального использования. Она должна позволять пользователю управлять своими задачами, отслеживать их выполнение, оставлять комментарии и упрощать планирование.

На данном этапе система еще находится в стадии разработки, но уже включает основные функции:

- Регистрация и авторизация пользователей.
- Управление задачами: создание, редактирование и удаление.
- Добавление комментариев к задачам.
- Базовый анализ задач.

Также система предусматривает наличие отдельной учетной записи администратора. С помощью аккаунта администратора возможно:

- Создание задач для пользователей.
- Контроль над выполнением задач.

Пользователи (сотрудники), в свою очередь, имеют доступ к задачам, назначенным им администратором, с ограниченным набором действий:

- Изменение статуса задачи (например, "В процессе", "Завершена").
- Добавление комментариев к задаче для уточнения деталей или отчета о выполнении.

Таким образом, система обеспечивает удобное взаимодействие между администратором (например админ - это тимлид команды) и пользователями, позволяя эффективно организовать процесс управления задачами.

1. Серверная часть (Backend)

Для разработки серверной части приложения использовался фреймворк **Spring** (v3.4.1), который предоставил удобный способ создания RESTful веб-сервисов и организации структуры проекта. Выбор этой технологии был обусловлен ее широкими возможностями, поддержкой модульности и масштабируемости.

Инструменты и зависимости:

В процессе разработки использовались следующие основные инструменты и зависимости:

1. **Spring Boot Starter Web**:
    - Предоставляет встроенный сервер (Tomcat), а также набор классов и аннотаций для создания REST API.

2. **Spring Boot Starter Data JPA**:
    - Используется для упрощенной работы с базами данных, реализуя слой репозиториев и предоставляя удобный механизм для выполнения CRUD-операций.

3. **Spring Boot Starter Security**:
    - Обеспечивает реализацию аутентификации и авторизации, что гарантирует защиту ресурсов приложения.

4. **JWT (JSON Web Token)**:
    - Используется для реализации безопасной аутентификации.
    - Зависимости:

      jjwt-api, jjwt-impl, jjwt-jackson — для работы с токенами JWT.

5. **Lombok**:
    - Упростил разработку, автоматизируя генерацию кода (геттеры, сеттеры, конструкторы, toString() и т. д.).

6. **H2 Database** (runtime):
    - Встроенная база данных для тестирования и разработки.

7. **MySQL Connector**:
    - Используется для подключения приложения к реальной базе данных MySQL.

8. **Apache Commons Lang**:
    - Библиотека для работы со строками, коллекциями, числовыми значениями и другими объектами.

9. **Spring Boot Starter Test**:
    - Набор инструментов для модульного и интеграционного тестирования приложения.

10. **Annotations от JetBrains**:
    - Предоставляют удобные аннотации для повышения читаемости и проверки кода (например, @Nullable, @NotNull).

Организация кода:

- Все слои приложения (контроллеры, сервисы, репозитории) структурированы согласно принципам MVC + Service Layer.
- Для управления базами данных использовался **Hibernate**, который является частью Spring Data JPA, упрощая работу с объектно-реляционным отображением (ORM).
- Реализована защита API с использованием Spring Security и токенов JWT для аутентификации пользователей.

Сборка и запуск:

Для управления проектом использовалась система **Maven**, где файл `pom.xml` включал все необходимые зависимости.

Плагин `spring-boot-maven-plugin` для сборки исполняемого JAR-файла.

2. Фронтенд часть (Frontend)

Для разработки клиентской части приложения использовался **Angular** (версия 19.1.0). Выбор Angular был сделан из-за его мощных возможностей для построения модульной архитектуры, удобного двустороннего связывания данных (two-way binding) и встроенной поддержки TypeScript.

Структура проекта

Проект организован модульно, с разделением на следующие основные области:

- ➔ **auth** — отвечает за аутентификацию пользователей.
  - ◆ **components**:
    - `login` — компонент для входа пользователей в систему.
    - `signup` — компонент для регистрации новых пользователей.
  - ◆ **services**:
    - `auth` — компонент, который содержит логику работы с запросами аутентификации, такими как вход и регистрация пользователей, отправляемыми на сервер
    - `storage` — управляет сохранением данных в LocalStorage (токенов).
- ➔ **modules/admin** — модуль для функционала администратора.
  - ◆ **components**:
    - `admin-dashboard` — компонент панели управления для администратора.
    - `post-task` — компонент для добавления новых задач.
    - `update-task` — компонент для редактирования существующих задач.
    - `view-task-details` — компонент для просмотра подробной информации о задачах.

- **services**:
  - `admin.service.ts` — предоставляет сервисы для работы с задачами: создание, обновление и получение данных с сервера.

➔ **modules/employee** — модуль для функционала сотрудников.

- **components**:
  - `employee-dashboard` — компонент панели управления для сотрудника.
  - `view-task-details` — компонент для просмотра деталей задач.

➔ **shared** — модуль с общими компонентами, утилитами и сервисами, которые могут быть использованы в нескольких частях приложения.

Зависимости проекта

1. **@angular/core и другие Angular пакеты**:
   - Основные библиотеки для работы с Angular.
   - **@angular/router** — используется для маршрутизации, что позволяет переключаться между страницами приложения без перезагрузки.
   - **@angular/forms** — предоставляет реактивные формы для работы с пользовательским вводом.

2. **@angular/material**:
   - UI-библиотека компонентов, таких как кнопки, карточки, диалоговые окна, формы и другие элементы интерфейса.

3. **RxJS**:
   - Используется для работы с реактивным программированием. Основные операторы RxJS применяются для обработки потоков данных, таких как запросы HTTP.

4. **Express и cors**:
    - Используются для поддержки серверного рендеринга (SSR) и настройки CORS для взаимодействия с API.

5. **localstorage-polyfill**:
    - Библиотека для работы с LocalStorage в окружениях, где он может быть недоступен.

Основные функции

- **Аутентификация (Auth)**:
    - Пользователь может войти в систему или зарегистрироваться.
    - После успешной аутентификации данные токена сохраняются в LocalStorage.

- **Административный функционал (Admin Module)**:
    - Администратор может:
        - Создавать новые задачи (через компонент `post-task`).
        - Обновлять существующие задачи (через компонент `update-task`).
        - Просматривать список и детали задач.

- **Функционал для сотрудников (Employee Module)**:
    - Сотрудники видят список задач, назначенных им, могут просматривать их детали, оставлять комментарии и изменять их статус.

- **Маршрутизация**:
    - Приложение использует модуль Angular Router для маршрутов, таких как `/login`, `/signup`, `/admin-dashboard`, `/employee-dashboard`

Фронтенд реализован с использованием Angular, что обеспечивает удобную работу с данными, маршрутизацию и масштабируемость приложения. Использование модульной структуры делает проект легко расширяемым.

3. Дизайн пользовательского интерфейса и удобство использования

Интерфейс был разработан с целью создания максимально интуитивно понятной и удобной среды для работы. Основной акцент сделан на простоте использования и четкой организации информации.

Панель входа в аккаунт:

Контейнер с заголовком:

1. Используется `<mat-card>` с заголовком `Логин`.
2. Заголовок задается через `<mat-card-title>`

Форма входа:

- Форма реализована с использованием Angular `Reactive Forms` через `[formGroup]` и управляется объектом `loginForm`.
- Поля формы:
    - **Email**:
        - Поле ввода (`<input>`) с указанием `formControlName="email"`.
        - Добавлена валидация, и если email некорректный или не введён, показывается сообщение через `<mat-error>`.
    - **Пароль**:
        - Поле с маскировкой пароля, которая может быть отключена кнопкой "показать/скрыть пароль" (используется `mat-icon`).
        - Включена валидация на обязательность.
        - Минимум 8 символов для пароля
- **Кнопка отправки**:
    - Используется кнопка с `mat-raised-button` и типом `submit`.
    - Заблокирована, если форма (`loginForm`) некорректна.

Ссылка на регистрацию:

- В нижней части добавлен текст "Нет аккаунта? Зарегистрируйтесь" и кнопка с маршрутом (`routerLink="/register"`) для перехода на страницу регистрации.

Панель регистрации

Основные элементы:

1. **Контейнер с заголовком**:
   - `<mat-card>` с заголовком `Регистрация`.
2. **Форма регистрации**:
   - Также реализована через `Reactive Forms` (`signupForm`).
   - Поля формы:
     - **Ваше имя**:
       - Поле ввода для имени.
     - **Email**:
       - Поле с аналогичной логикой проверки email.
     - **Пароль**:
       - Поле ввода пароля, с возможностью показать/скрыть его.
       - Добавлена обязательная проверка.
     - **Подтверждение пароля**:
       - Поле для повторного ввода пароля. Логика сравнения пароля и его подтверждения реализованы в коде компонента.
   - **Кнопка регистрации**:
     - Используется кнопка для отправки формы `signUp()`, которая блокируется при некорректных данных.

- **Валидация**:
  - Валидация пароля и подтверждение пароля реализованы в signup.component.ts файле

Идеи для улучшения

1. **Планирую улучшить UX**:
    - Добавить индикатор сложности пароля в панели регистрации.
2. **Мобильная адаптация**:
    - Обеспечить корректное отображение на мобильных устройствах.
3. **Логика подсказок**:
    - Добавить подсказки, например, формат email или требуемую длину пароля.
4. **Загрузка**:
    - Включить индикатор загрузки на кнопках при выполнении входа/регистрации.

4. Составление плана тестирования системы управления задачами

Цель тестирования API — убедиться в корректной работе всех доступных методов взаимодействия с системой управления задачами. Тестирование проводится с помощью Postman для проверки функционала, надежности и соответствия спецификациям.

Область тестирования

Тестированию подлежат следующие ключевые функции API:

1. **Аутентификация**:
    - Регистрация пользователей.
    - Авторизация (получение токена).
2. **Работа с задачами**:
    - Создание задачи.
    - Обновление информации о задаче.
    - Удаление задачи.
    - Получение списка задач.
3. **Права доступа**:
    - Проверка доступа к ресурсам для пользователей с разными ролями.

Типы тестирования

1. **Функциональное тестирование**:
    - Проверка каждого эндпоинта на соответствие спецификации.
    - Валидация обязательных и необязательных параметров.
2. **Тестирование ошибок**:
    - Проверка обработки некорректных запросов (например, запросы с недостающими данными или неверными параметрами).
    - Обработка запросов с отсутствующим или истекшим токеном авторизации.

Инструменты тестирования

- **Postman**:
    - Ручное выполнение тестов.

Примеры тест-кейсов для API

1. **Авторизация**
    - Отправка запроса с корректными данными (логин и пароль) → ожидается получение токена.
    - Запрос с неверными данными (например, неправильный пароль) → ожидается ошибка 401 Unauthorized.
2. **Создание задачи**
    - Отправка запроса с корректными данными (название, описание, дедлайн) → задача должна быть создана, ответ 201 Created.
    - Пропуск обязательного поля (например, "название") → ожидается ошибка 400 Bad Request.
3. **Обновление задачи**
    - Запрос на обновление существующей задачи → изменения должны примениться, ответ 200 OK.
    - Попытка обновить несуществующую задачу → ошибка 404 Not Found.
4. **Получение списка задач**
    - Запрос без параметров → возвращается полный список задач.
    - Неверный параметр фильтрации → ошибка 400 Bad Request.
5. **Удаление задачи**
    - Удаление существующей задачи → задача должна быть удалена, ответ 204 No Content.
    - Удаление задачи, которая уже была удалена или не существует → ошибка 404 Not Found.
6. **Безопасность**
    - Запрос к защищенному эндпоинту без токена → ошибка 401 Unauthorized.
    - Использование токена истекшей сессии → ошибка 401 Unauthorized.

Глава 3. Реализация системы управления задачами

1. Архитектура проекта

Мой проект построен на основе многослойной архитектуры MVC с дополнительным Service-слоем, обеспечивающим четкое разделение ответственности:

- **Controller**: Обработка входящих HTTP-запросов и передача данных в сервисы.
- **Service**: Бизнес-логика приложения, где происходит обработка данных.
- **Repository**: Работа с базой данных через Spring Data JPA.
- **Entity**: Модели данных, представляющие сущности базы данных.
- **DTO**: Классы для передачи данных между слоями.
- **Utils**: Утилитарные классы для общих задач (JWT-утилита).
- **Configs**: Конфигурация приложения, включая безопасность и фильтры.

2. Структура проекта

1. **`configs`**
    - **`JwtAuthFilter`**: Фильтр для проверки JWT токенов на каждом входящем запросе.
    - **`WebConfig`**: Конфигурация веб-приложения (CORS).
    - **`WebSecurityConfig`**: Настройка Spring Security для защиты эндпоинтов, аутентификации и авторизации.

2. **`controllers`**
    - **`admin/AdminController`**: Контроллер для управления задачами и пользователями, доступный только для пользователей с ролью `ADMIN`.
    - **`auth/AuthController`**: Контроллер для аутентификации и регистрации пользователей (+ создается автоматически аккаунт админа).
    - **`employee/EmployeeController`**: Контроллер для сотрудников, предоставляющий доступ к задачам и комментариям.
3. **`dtos`** DTO используются для передачи данных между слоями:
    - **`AuthRequest`** / **`AuthResponse`**: Для входа в систему и обработки JWT токена.
    - **`SignUpRequest`**: Данные, необходимые для регистрации пользователя.
    - **`TaskDTO`** и **`CommentDTO`**: Для создания, обновления и передачи данных задач и комментариев.
    - **`UserDTO`**: Для представления информации о пользователях.
4. **`entities`**
    - **`User`**: Модель пользователя с полями `id`, `name, email`, `password`, `userRole`
    - **`Task`**: Модель задачи с полями `id`, `title`, `description`, `status`, `priority`, `dueDate`, и `user`.
    - **`Comment`**: Модель комментария с полями `id`, `content`, `createdAt`, и связью с задачей и пользователем.

5. **enums**
    - **TaskStatus**: Перечисление для статусов задач (PENDING, IN_PROGRESS, COMPLETED, DEFERRED, CANCELED).
    - **UserRole**: Перечисление ролей пользователей (ADMIN, EMPLOYEE).
6. **repository**
    - **UserRepository**: Для работы с данными пользователей.
    - **TaskRepository**: Для работы с данными задач.
    - **CommentRepository**: Для работы с данными комментариев.
7. **services**
    - **admin.AdminService и AdminServiceImpl**: Бизнес-логика для администрирования задач и пользователей.
    - **auth.AuthService и AuthServiceImpl**: Логика для входа в систему и регистрации.
    - **employee.EmployeeService и EmployeeServiceImpl**: Логика для управления задачами сотрудников и их комментариями.
    - **jwt.UserService и UserServiceImpl**: Утилиты для поиска пользователей в процессе авторизации.
8. **utils**
    - **JwtUtil**: Класс для работы с JWT (генерация, валидация, извлечение информации из токена).

3. Основной функционал

1. **Аутентификация и авторизация**
    - Аутентификация с использованием JWT.
    - Роли пользователей (`ADMIN`, `EMPLOYEE`) используются для ограничения доступа к эндпоинтам.
2. **Управление пользователями**
    - Регистрация через `AuthController` с использованием DTO.
    - Администратор может управлять пользовательскими задачами через `AdminController (метод postTask())`.
3. **Управление задачами**
    - CRUD-операции для задач через `AdminController` и `EmployeeController`.
    - Задачи имеют статус, приоритет и связаны с пользователями.
4. **Комментарии**
    - Пользователи могут оставлять комментарии к задачам через `EmployeeController`.
    - Комментарии сохраняются в базе данных и связаны с задачей.
5. **Защита API**
    - Spring Security проверяет JWT токены на всех защищенных эндпоинтах.
    - Только администратор имеет доступ к административным операциям.

4. Преимущества архитектуры

- **Модульность**: Каждая часть системы четко разделена на слои, что упрощает сопровождение и развитие.
- **Безопасность**: Использование Spring Security и JWT обеспечивает надежную защиту API.
- **Масштабируемость**: Благодаря разделению бизнес-логики в сервисах, новые функции могут быть добавлены без нарушения текущей логики.

Создание проекта с помощью Spring Initializr

1. Перешел на сайт Spring Initializr (https://start.spring.io/):

- Выбрал:
  - **Project**: Maven.
  - **Language**: Java.
  - **Spring Boot version**: 3.4.2
- Указал параметры проекта:
  - **Group**: com.finalproject.
  - **Artifact**: TaskManagement.
  - **Name**: Task Management Application.
- Выбрал основные зависимости:
  - **Spring Web**: Для создания REST-контроллеров.
  - **Spring Data JPA**: Для работы с базой данных через ORM.
  - **Spring Security**: Для настройки аутентификации и авторизации.
  - **Lombok**: Для сокращения шаблонного кода (например, `@Getter`, `@Setter`, `@Builder`).
  - **MySQL Driver**: Для подключения к MySQL.

Скачал и распаковал проект, затем открыл его в IDE (IntelliJ IDEA).

2. Создание структуры пакетов

После создания проекта я начал структурировать его по пакетам. Это стандартный и удобный подход для создания многослойного приложения:

- **configs**: Для конфигурации Spring Security, фильтров, и других общих настроек.
- **controllers**: Для REST-контроллеров, которые обрабатывают HTTP-запросы.
- **services**: Для бизнес-логики.
- **repository**: Для взаимодействия с базой данных.
- **entity**: Для создания сущностей JPA.
- **dto**: Для передачи данных между слоями.
- **utils**: Для вспомогательных классов.

3. Настройка *application.properties*

```
spring.application.name=TaskManagement

# Настройки подключения к базе данных MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/MyDataBase

spring.datasource.username=root

spring.datasource.password=password

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Настройка JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true
```

Пояснение:

- `spring.datasource.url`: URL подключения к базе данных (MySQL).
- `spring.datasource.username` и `spring.datasource.password`: Данные для доступа к базе.
- `spring.jpa.hibernate.ddl-auto=update`: Автоматическое создание или обновление схемы базы данных на основе сущностей.
- `spring.jpa.show-sql=true`: Включение отображения SQL-запросов в логах для отладки.

4. Настройка MySQL

Т.к. у меня это локальная база данных MySQL, то для корректного запуска приложения необходимо сначала создать и настроить эту БД с именем **MyDataBase.**

```
CREATE DATABASE MyDataBase;
```

5. Подключение драйвера MySQL

В `pom.xml` добавил зависимость для MySQL:

```
<dependency>
   <groupId>com.mysql</groupId>
   <artifactId>mysql-connector-j</artifactId>
   <scope>runtime</scope>
</dependency>
```

Эта зависимость позволяет приложению взаимодействовать с MySQL через JDBC.

6. Использование H2 для тестирования

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
</dependency>
```

7. Проверка подключения

1. Приложение успешно подключается к базе данных:
    - В логах Spring Boot увидел сообщения о создании схемы таблиц.
    - Т.к. используется `spring.jpa.hibernate.ddl-auto=update`, таблицы будут автоматически созданы на основе классов-сущностей.

Postman: Создание коллекции для ручного тестирования API

1. Установка и настройка Postman

- Скачал и установил Postman с официального сайта (https://www.postman.com/).

2. Создание коллекции

Коллекция в Postman — это удобный способ организовать запросы, связанные с одним проектом.

Шаги:

1. Открыл Postman.
2. На главной странице нажал **Create Collection**.
3. Имя коллекции `Task Management API`.
4. Описание коллекции "Коллекция для тестирования API системы управления задачами".

3. Добавление запросов в коллекцию

Шаги:

1. Нажал + для создания нового запроса.
2. Указал метод запроса (POST) в выпадающем списке.
3. Ввел URL эндпоинта, который я буду тестировать `http://localhost:8080/api/auth/login`
4. Перешел во вкладку **Body**:
    - Выбрал **raw**.
    - Установил тип данных как JSON.

```
{
    "email":"AnotherUser01@gmail.com",
    "password":"AnotherUser01"
}
```

5. Нажал **Save** и выбрал коллекцию для сохранения запроса.

4. Пример структуры коллекции

- **Auth**
  - POST `/api/auth/login` — Вход в систему.
  - POST `/api/auth/signup` — Регистрация пользователя.
- **Tasks**
  - GET `/api/admin/tasks` — Получить список задач.
  - POST `/api/admin/task` — Создать новую задачу.
  - PUT `/api/admin/task/{id}` — Обновить задачу.
  - DELETE `/api/admin/task/{id}` — Удалить задачу.
- **Comments**
  - GET `/api/admin/task/{taskId}/comments` — Получить комментарии к задаче.
  - POST `/api/admin/task/comment` — Добавить комментарий к задаче.
- **Admin**
  - GET `/api/admin/users` — Получить список пользователей.

5. Тестирование с помощью Postman

1. Выбрал запрос в коллекции и нажал **Send**.
2. Проверил:
   - **Response Status** (200 OK, 401 Unauthorized).
   - **Response Body** — данные, возвращенные API в формате JSON.

Пример ответа для запроса логина:


```
{"jwt":
"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJBbm90aGVyVXNlcjAxQGdtYWlsLmNvbSIsImlhdCI6MTczNzk2NzUyNSwiZ
XhwIjoxNzM4MDUzOTI1fQ._GmHUqA4Puy1dMR39meZhQbxpwI5lS-r8Bx1wJV6EWs",

    "userId": 13,

    "userRole": "EMPLOYEE"

}
```

7. Преимущества коллекции

- Можно запускать запросы последовательно и проверять ответы.
- Коллекции легко экспортировать/импортировать, чтобы делиться с командой.
- Возможность автоматизировать тестирование с помощью **Postman Runner** или интеграции с CI/CD.

8. Экспорт коллекции

Если нужно передать коллекцию другому разработчику:

1. Нажмите на три точки рядом с коллекцией.
2. Выберите **Export**.
3. Сохраните файл `.json` на компьютер.
4. Коллега может импортировать коллекцию через **Import** в Postman.

Реализация frontend-части с использованием Angular

1. Создание проекта и настройка Angular

1. **Инициализация проекта:**

   Проект был создан с использованием Angular CLI:

   ```
   ng new TASK_ANGULAR_V19
   ```

   Выбрана модульная структура для разделения логики на независимые модули.

2. **Установка необходимых пакетов:**

   Подключены основные библиотеки и зависимости:
   `npm install @angular/material @angular/forms @angular/router @angular/common @angular/common/http @angular/core`

   Настроены маршруты с использованием Angular Router.

2. Структура файлов и модулей

`components`

Директория содержит компоненты, отвечающие за отображение и обработку пользовательского интерфейса:

- **`login`**:
  - Файлы:
    - `login.component.ts`: отвечает за логику авторизации.
    - `login.component.html`: шаблон формы авторизации.
    - `login.component.scss`: стили компонента.

    - Основная функциональность:
        - Обработка ввода пользователя (email, пароль).
        - Вызов сервиса `auth.service.ts` для отправки данных на сервер.
        - Обработка токена после успешного входа.
- **`signup`**:
    - Аналогичная структура для регистрации пользователя.
    - Отправляет данные на API через `auth.service.ts`.

`modules`

Модули организуют приложение по ролям (`admin`, `employee`) и позволяют изолировать их логику.

- **`admin`**:
    - **Компоненты:**
        - `admin-dashboard`: отвечает за отображение панели администратора.
        - `post-task`, `post-task`, `update-task`, `view-task-details`: компоненты для управления задачами.
    - **Сервисы:**
        - `admin.service.ts`: обработка API-запросов для функций администратора.
- **`employee`**:
    - **Компоненты:**
        - `employee-dashboard`: панель сотрудника с задачами.
        - `view-task-details`: просмотр деталей задач.
    - **Сервисы:**
        - `employee.service.ts`: запросы на получение и обновление информации о задачах сотрудника.

`services`

Сервисы обеспечивают взаимодействие с backend API:

- **`auth`**:
    - Логика авторизации:

```
login(loginDTO: any): Observable<any> {

return this.http.post('http://localhost:8080/api/auth/login', loginDTO);

}
```

- Отправка данных на сервер для получения токена.

```
signup(signupDTO: any): Observable<any> {

return this.http.post('http://localhost:8080/api/auth/signup', signupDTO);

}
```

- Регистрация пользователя.

Хранение токена в `localStorage` через `storage.service.ts`

- **`storage`**:
    - Управление данными в локальном хранилище:

```
static saveToken(token: string): void
```

- Сохранить токен.

```
static getToken(): string | null
```

- Получить токен.

- **`admin.service.ts` и `employee.service.ts`**:
    - Обработка специфичных для ролей API-запросов, например:
        - Создание задач (`postTask`).
        - Обновление задач (`updateTask`).
        - Получение списка задач для сотрудника.

`shared`

Модуль содержит общие компоненты, утилиты и директивы:

- `shared.module.ts`: подключение Angular Material компонентов и общих сервисов.

3. Реализация ключевых функций

Форма авторизации (Login)

HTML (`login.component.html`):

```
<div class="container">
    <mat-card>
        <mat-card-title>Логин</mat-card-title>
        <mat-card-content class="form-container">
            <form [formGroup]="loginForm">
                <mat-form-field appearance="outline">
                    <mat-label>Email</mat-label>
                    <input matInput required formControlName="email">
<mat-error *ngIf="loginForm.get('email')?.invalid && loginForm.get('email')?.touched">
                        Пожалуйста, введите корректный email.
                    </mat-error>
                </mat-form-field>

                <mat-form-field appearance="outline">
                    <mat-label>Пароль</mat-label>
                    <input matInput type="{{ hidePassword ? 'password' : 'text' }}"
                        required formControlName="password">
                    <button mat-icon-button matSuffix (click)="togglePasswordVisibility()">
                        <mat-icon>
                            {{hidePassword ? 'visibility_off' : 'visibility' }}
                        </mat-icon>
                    </button>
  <mat-error *ngIf="loginForm.get('password')?.invalid && loginForm.get('password')?.touched">
                        Пожалуйста, введите корректный пароль.
                    </mat-error>
                </mat-form-field>
                     <button class="login-btn" mat-raised-button color="primary" type="submit"
(click)="login()" [disabled]="loginForm.invalid">
                    Войти
                </button>
            </form>
        </mat-card-content>
        <mat-card-actions class="centered-actions">
            <span>Нет аккаунта? Зарегистрируйтесь :)</span>
        </mat-card-actions>
        <button mat-button class="mat-btn" routerLink="/register">
            Зарегистрироваться
        </button>
    </mat-card>
</div>
```

Логика (`login.component.ts`):

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { SharedModule } from '../../../shared/shared.module';
import { AuthService } from '../../services/auth/auth.service';
import { MatSnackBar } from '@angular/material/snack-bar';
import { catchError, of } from 'rxjs';
import { StorageService } from '../../services/storage/storage.service';
import { Router } from '@angular/router';
@Component({
  selector: 'app-login',
  imports: [SharedModule],
  templateUrl: './login.component.html',
  styleUrl: './login.component.scss'
})
export class LoginComponent implements OnInit{
    loginForm!:FormGroup;
    hidePassword:boolean = true;
    constructor(
      private fb:FormBuilder,
      private service: AuthService,
      private snackbar: MatSnackBar,
      private router: Router
    ){}
    ngOnInit(){
      this.loginForm = this.fb.group({
        email:[null,[Validators.required, Validators.email]],
        password:[null,[Validators.required]],
      })
    }
    togglePasswordVisibility(){
      this.hidePassword = !this.hidePassword
    }
    login() {
      console.log(this.loginForm.value);
      this.service.login(this.loginForm.value).pipe(
        catchError(() => {
          this.snackbar.open('Неверные учетные данные', 'Закрыть', {
            duration: 3000
          });
          return of(null);
        })
      ).subscribe({
        next: (response) => {
          console.log(response);
```

```
          if (response && response.userId) {
            const user = {
              id: response.userId,
              role: response.userRole
            };
            console.log(response.jwt);
            StorageService.saveUser(user);
            StorageService.saveToken(response.jwt);
             this.snackbar.open("Вы успешно вошли в аккаунт!", "Закрыть", {duration:
5000});
            if(StorageService.isAdminLoggedIn()) {
              this.router.navigateByUrl("/admin/dashboard");
            } else if (StorageService.isEmployeeLoggedIn()) {
              this.router.navigateByUrl("/employee/dashboard");
            } else {
                  this.snackbar.open("Не удалось определить роль пользователя.",
"Закрыть", {
                duration: 3000
              });
            }
          }
        }
      });
    }
  }
```

Admin Dashboard

```
import { Component } from '@angular/core';
import { AdminService } from '../../services/admin.service';
import { SharedModule } from '../../../../shared/shared.module';
import { FormBuilder, FormGroup } from '@angular/forms';
import { MatSnackBar } from '@angular/material/snack-bar';
@Component({
  selector: 'app-admin-dashboard',
  imports: [SharedModule],
  templateUrl: './admin-dashboard.component.html',
  styleUrl: './admin-dashboard.component.scss'
})
export class AdminDashboardComponent {
  listOfTasks:any = [];
  searchTaskForm!:FormGroup;

  constructor(private service: AdminService,
    private fb:FormBuilder,
    private snackbar:MatSnackBar
  ) {}
  ngOnInit() {
    this.searchTaskForm = this.fb.group({title:[null]})
    this.getTasks();
  };
  getTasks() {
    this.service.getTasks().subscribe((response)=> {
      console.log(response);
      this.listOfTasks = response;
    })
  }
  submitForm() {
    console.log(this.searchTaskForm.value);
    const title = this.searchTaskForm.get('title')!.value;
    this.listOfTasks = [];
    if (title.length === 0) {this.getTasks();}
this.service.searchTasks(title).subscribe((res)=>{this.listOfTasks = res;})
  }
  deleteTask(id:number) {
    this.service.deleteTask(id).subscribe((response)=>{
      this.getTasks;
      window.location.reload();
      this.snackbar.open('Вы удалили задачу!', 'Закрыть', {duration:5000});
    })
  }
}
```

Сервис (`admin.service.ts`):

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { StorageService } from '../../../auth/services/storage/storage.service';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AdminService {
  constructor(private http: HttpClient) { }
  getUsers(): Observable<any> {
    return this.http.get('http://localhost:8080/api/admin/users', httpOptions);
  }
  postTask(taskDto: any): Observable<any> {
          return   this.http.post('http://localhost:8080/api/admin/task',   taskDto,
httpOptions);
  }
  searchTasks(title: string): Observable<any> {
      return this.http.get('http://localhost:8080/api/admin/tasks/search/' + title,
httpOptions);
  }
  deleteTask(id: number): Observable<any> {
          return   this.http.delete('http://localhost:8080/api/admin/task/'   +   id,
httpOptions);
  }
  updateTask(id: number, taskDTO: any): Observable<any> {
      return this.http.put('http://localhost:8080/api/admin/task/' + id, taskDTO,
httpOptions);
  }
  getTaskById(id: number): Observable<any> {
           return   this.http.get('http://localhost:8080/api/admin/task/'   +   id,
httpOptions);
  }
  getCommentsById(id: number): Observable<any> {
          return   this.http.get('http://localhost:8080/api/admin/task/'   +   id   +
'/comments', httpOptions);
  }
  createComment(taskId: number, content: string): Observable<any> {
    const params_ = {
      taskId: taskId,
      postedBy:StorageService.getUserId() ?? ''
    };
     return this.http.post('http://localhost:8080/api/admin/task/comment', content,
{
      params: params_,
      headers: new HttpHeaders({
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${StorageService.getToken()}`
      }),
      withCredentials: true
    });
  }
```

```
  getTasks(): Observable<any> {
    console.log('Current Token:', StorageService.getToken());
    console.log('httpOptions:', httpOptions);
    return this.http.get('http://localhost:8080/api/admin/tasks', httpOptions);
  }
}

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json',
    'Authorization': `Bearer ${StorageService.getToken()}`
  }),
  withCredentials: true
};
```

4. Тестирование приложения

- Все компоненты были протестированы через встроенный `ng serve`.
- Интерфейс и функциональность тестировались в браузере Chrome.
- Для тестирования взаимодействия с API использовался Postman.

Заключение

Итоги разработки приложения Task Management System

В процессе разработки приложения **Task Management System** были успешно реализованы ключевые части системы.

Backend был построен с использованием **Java Spring**, что позволило создать RESTful API для обработки всех основных операций с задачами, таких как создание, редактирование, удаление и просмотр. Для хранения данных о пользователях, задачах и их статусах была интегрирована база данных **MySQL**.

Для обеспечения безопасности был реализован механизм аутентификации и авторизации с помощью **Spring Security**, что позволило контролировать доступ к системе и защищать пользовательские данные.

На **frontend** использовался **Angular**, что позволило создать динамичный и отзывчивый интерфейс для пользователя.

Тестирование ключевых функций системы включало в себя как тестирование на стороне backend (с использованием Postman), так и проверку пользовательского интерфейса, что обеспечило высокий уровень надежности и функциональности приложения.

Планы на будущее

Приложение будет развиваться дальше. В планах интеграция системы уведомлений для напоминаний о сроках выполнения задач. Будет реализована интеграция с календарем, мультиязычность и расширенная система управления правами доступа, чтобы администраторы могли назначать роли и разграничивать доступ.

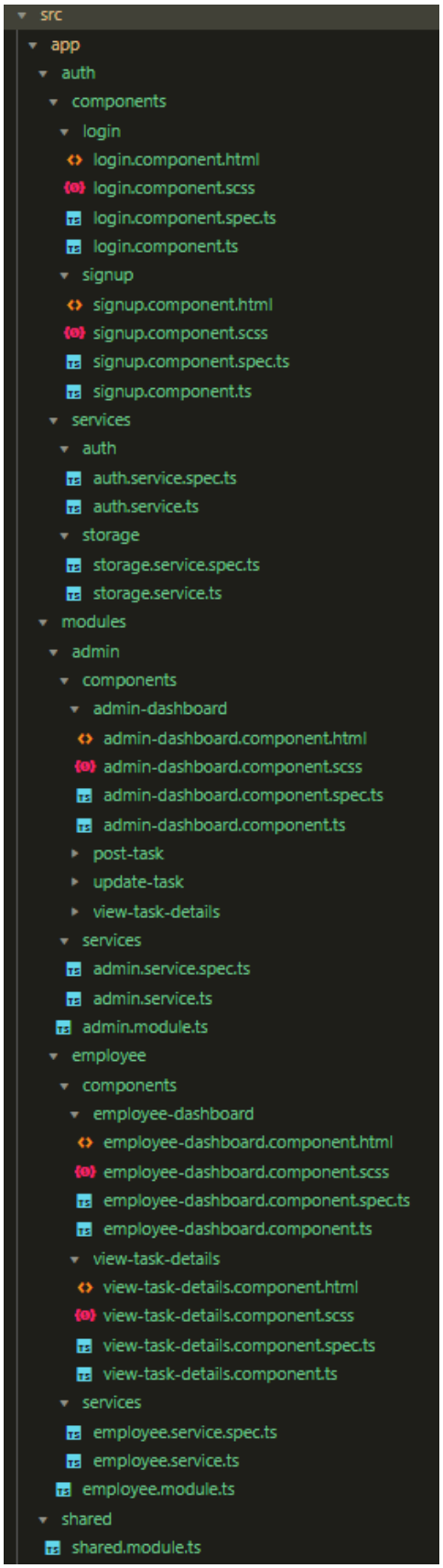
Я очень рад результатам, которых удалось достичь, и с удовольствием продолжу дальнейшую разработку проекта, добавляя новые возможности.

Список используемой литературы

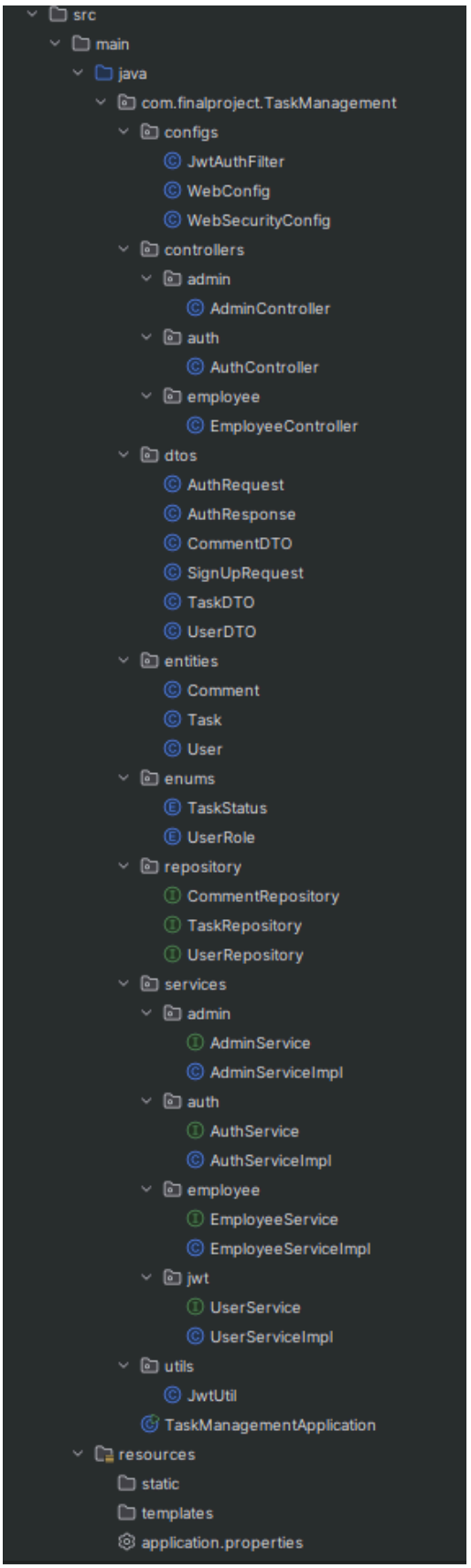
1. **"Spring in Action"** (6-е издание) — Крейг Уоллс
2. **"Pro Angular"** — Адам Фримен
3. **"MySQL 8.0 Reference Manual"** — MySQL Documentation
4. **Wikipedia** - https://www.wikipedia.org/

Приложения

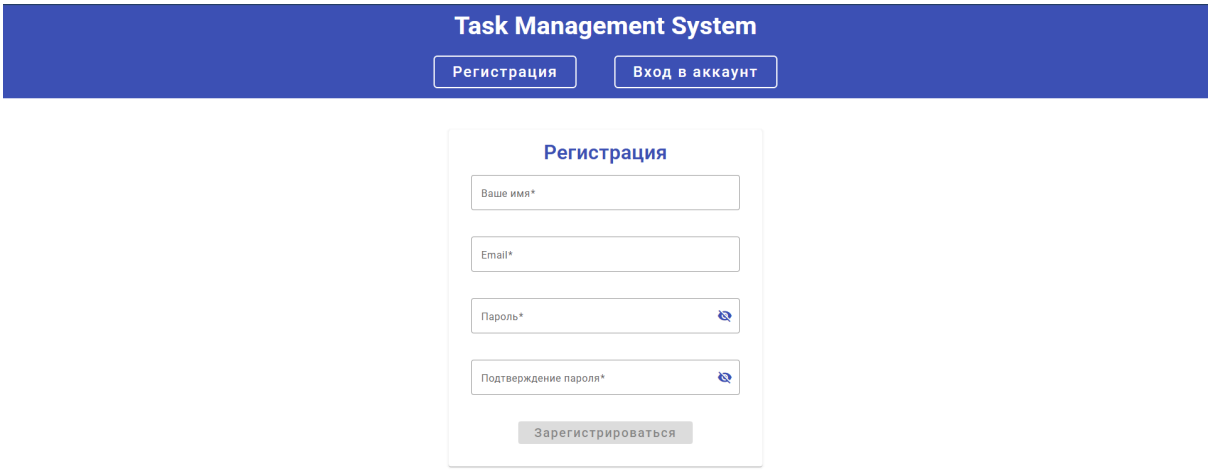
Структура frontend



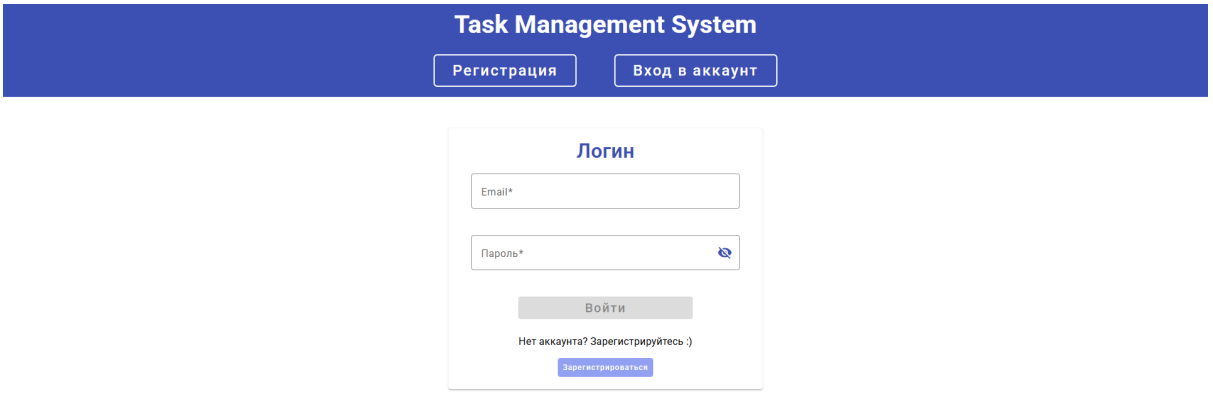
Структура backend



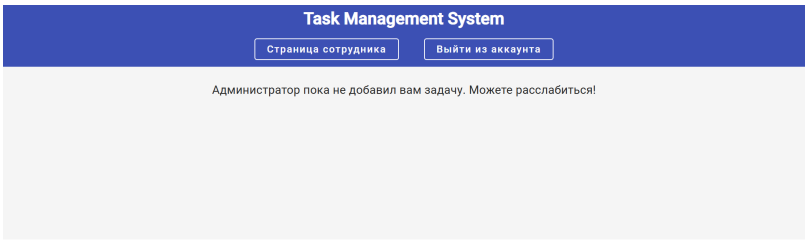
Форма регистрации



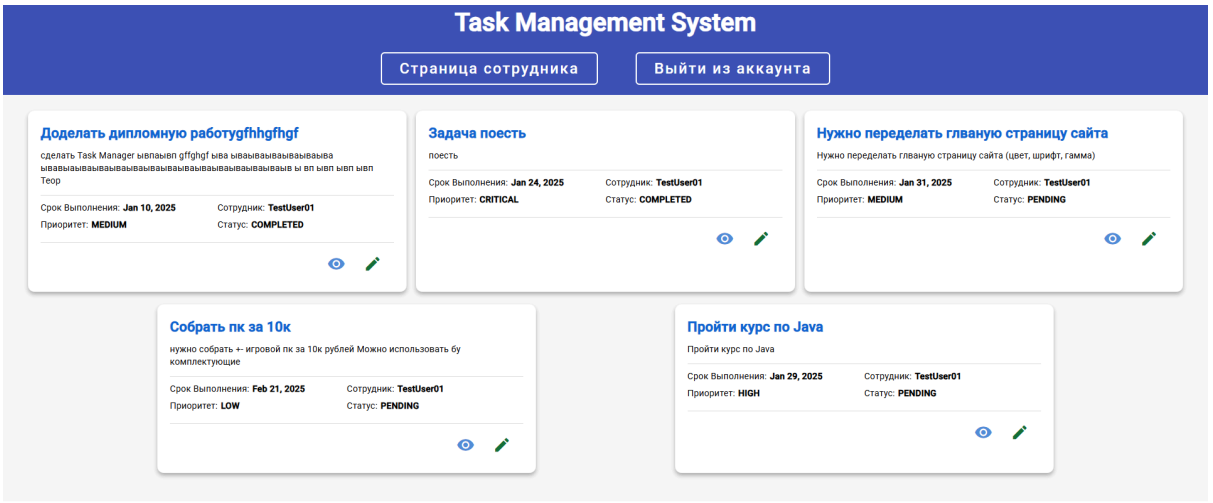
Форма логин



Сотрудник без задач



Сотрудник с задачами



Главная страница админа