Geekbrains

**Разработка веб-приложения для управления задачами и планирования проектов**

Программа: Разработчик – Программист

Специализация: Разработчик – Веб-разработка на Java

ФИО: Агазаде Азер Вугар оглы

Москва

2025

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc188565745)

[ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 7](#_Toc188565746)

[1.1 Обзор существующих решений 7](#_Toc188565747)

[1.2 Требования к системе 9](#_Toc188565748)

[1.3 Выводы по анализу предметной области 11](#_Toc188565749)

[ГЛАВА 2. ПРОЕКТИРОВАНИЕ СИСТЕМЫ 14](#_Toc188565750)

[2.1 Архитектура приложения 14](#_Toc188565751)

[2.2 Проектирование базы данных 16](#_Toc188565752)

[2.3 Диаграмма классов 18](#_Toc188565753)

[2.4 Диаграмма последовательности 21](#_Toc188565754)

[2.5 Конфигурация приложения 22](#_Toc188565755)

[2.5.1 Файл `application.yml` и его роли 22](#_Toc188565756)

[2.5.2 Конфигурация CI/CD пайплайна 25](#_Toc188565757)

[2.6 Выводы по проектированию 26](#_Toc188565758)

[ГЛАВА 3. РЕАЛИЗАЦИЯ СИСТЕМЫ 29](#_Toc188565759)

[3.1 Разработка модулей приложения: Backend и REST API 29](#_Toc188565760)

[3.2 Тестирование системы 32](#_Toc188565761)

[3.3 Инструменты для обеспечения качества кода 36](#_Toc188565762)

[ЗАКЛЮЧЕНИЕ 39](#_Toc188565763)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 43](#_Toc188565764)

[ПРИЛОЖЕНИЯ 46](#_Toc188565765)

# ВВЕДЕНИЕ

Управление задачами и проектами является неотъемлемой частью эффективной организации работы в современном обществе. В условиях возрастающей цифровизации и ускорения темпов деятельности многие компании и отдельные пользователи сталкиваются с необходимостью структурирования своих процессов. Это особенно актуально для малых и средних команд, которые не всегда могут позволить себе дорогостоящие программные решения или сложные системы управления, такие как Jira, Asana или Microsoft Project. Возникает потребность в создании удобных, интуитивно понятных и доступных инструментов, которые обеспечивают контроль над выполнением задач, упрощают взаимодействие внутри команды и повышают общую продуктивность. Система управления задачами, предлагаемая в рамках данного исследования, представляет собой веб-приложение, которое сочетает в себе основные функции для организации работы над проектами. Основной акцент сделан на простоту использования, минимализм интерфейса и адаптивность под индивидуальные потребности пользователей. Это приложение ориентировано на тех, кто ищет эффективное и функциональное решение для управления задачами, не требующее больших затрат времени на обучение или сложной настройки.

**Актуальность исследования** объясняется растущей потребностью в автоматизации процессов управления задачами. Традиционные методы планирования, такие как использование бумажных блокнотов или электронных таблиц, часто оказываются недостаточными при работе над проектами, требующими командного взаимодействия. Такие подходы не обеспечивают должного уровня контроля, прозрачности и гибкости. Существующие на рынке программные решения либо избыточны по функционалу для малых и средних команд, либо экономически недоступны, что создает значительный барьер для их внедрения.

Разработка предлагаемого приложения направлена на решение следующих проблем:

* Упрощение управления проектами и задачами благодаря интуитивно понятному пользовательскому интерфейсу.
* Обеспечение прозрачности выполнения задач с использованием таких инструментов, как метрики статусов задач и фильтры.
* Интеграция данных с внешними системами за счёт возможностей экспорта и импорта.
* Масштабируемость и гибкость, позволяющие адаптировать приложение под уникальные требования конкретного пользователя или команды.

**Цель исследования** – разработка, тестирование и внедрение системы управления задачами, которая будет ориентирована на индивидуальных пользователей, а также на малые и средние команды. Приложение должно стать простым в использовании инструментом для управления проектами, сохраняя при этом гибкость и функциональность.

Для достижения поставленной цели решаются следующие задачи:

1. Проведение анализа существующих систем управления задачами и выявление их недостатков.
2. Формирование технических и функциональных требований к системе на основе изучения пользовательских потребностей.
3. Проектирование архитектуры приложения с применением современных методологий разработки.
4. Реализация веб-приложения с использованием технологий Spring Boot, Hibernate, H2, Swagger и др.
5. Разработка функций экспорта и импорта данных для повышения удобства взаимодействия с системой.
6. Автоматизация процессов тестирования и деплоя через CI/CD пайплайн.
7. Проведение тестирования приложения для обеспечения его надёжности и соответствия требованиям пользователей.

**Объект исследования** – процесс автоматизации управления задачами и проектами.

**Предмет исследования** – программное обеспечение для управления задачами, включающее функционал добавления, редактирования, удаления задач и проектов, а также мониторинга их выполнения.

**Методы исследования** включают:

* Анализ существующих решений на рынке и изучение пользовательских предпочтений.
* Использование объектно-ориентированного проектирования для создания модели приложения.
* Разработка программного обеспечения с использованием современных языков программирования и фреймворков.
* Проведение юнит, интеграционных и функциональных тестов для проверки качества системы.
* Автоматизация разработки и развертывания приложения через CI/CD пайплайн с использованием инструментов, таких как GitHub Actions.

**Практическая значимость** разработки заключается в её применимости как для образовательных целей, так и для реального использования в малых и средних командах. Простота настройки и лёгкость масштабирования делают систему пригодной для адаптации под уникальные требования пользователей. Разработанное приложение может стать основой для дальнейшего совершенствования, включая добавление новых функций, таких как уведомления, интеграция с популярными мессенджерами, а также использование облачных решений для работы в распределённых командах.

Таким образом, разработка и внедрение системы управления задачами является важным и востребованным шагом, который позволяет значительно повысить эффективность работы над проектами и удовлетворить потребности пользователей в простом, функциональном и доступном инструменте.

# ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## 1.1 Обзор существующих решений

На сегодняшний день рынок систем управления задачами и проектами представлен множеством решений, которые удовлетворяют различные потребности пользователей – от индивидуальных фрилансеров до крупных корпораций. Наиболее популярные из них, такие как Trello, Jira и Asana, обладают широким функционалом и успешно применяются для организации работы в самых разных сферах. Однако, каждое из этих решений имеет свои особенности, которые важно учитывать при разработке новой системы.

**Trello** – это облачная система управления задачами, основанная на использовании канбан-досок. Приложение позволяет пользователям организовывать задачи с помощью карточек, которые перемещаются между столбцами, отражающими этапы выполнения работы.

*Преимущества*:

* Интуитивно понятный интерфейс, подходящий для пользователей без опыта работы с подобными системами.
* Гибкость в настройке досок и карточек, что делает инструмент универсальным для применения в разных сферах.
* Наличие бесплатного тарифного плана с базовыми функциями.

*Недостатки*:

* Ограниченный функционал для управления большими проектами и командами.
* Отсутствие встроенных инструментов для создания сложных зависимостей между задачами.
* Подходит в первую очередь для визуального контроля процессов, но менее удобен для детального анализа данных.

**Jira** – мощное решение для управления проектами, которое активно используется в ИТ-компаниях для разработки программного обеспечения. Оно поддерживает методологии Scrum и Kanban, а также предоставляет инструменты для управления багами и трекинга прогресса.

*Преимущества*:

* Богатый функционал, включая поддержку гибких методологий, настройку воркфлоу и генерацию аналитических отчетов.
* Интеграция с другими продуктами Atlassian, такими как Confluence.
* Эффективное решение для управления большими проектами в крупных командах.

*Недостатки*:

* Сложный интерфейс, требующий обучения и адаптации пользователей.
* Высокая стоимость лицензий для крупных команд.
* Переизбыток функций для небольших проектов или индивидуальных пользователей.

**Asana** – это универсальная платформа для управления задачами, которая сочетает простоту интерфейса с гибкостью в настройке. Она позволяет создавать проекты, добавлять задачи, назначать исполнителей и отслеживать прогресс.

*Преимущества*:

* Чистый и удобный интерфейс, обеспечивающий комфортное использование.
* Поддержка функциональности календаря, таймлайна и списка задач, что делает инструмент гибким для организации работы.
* Интеграция с популярными внешними инструментами, такими как Slack, Google Drive и Zoom.

*Недостатки*:

* Ограниченные возможности в бесплатной версии.
* Недостаточная детализация для управления крупными проектами по сравнению с Jira.
* Некоторые пользователи отмечают сложность в адаптации для нетехнических команд.

## 1.2 Требования к системе

Создание эффективной системы управления задачами предполагает тщательное определение функциональных и нефункциональных требований. Эти требования служат основой для разработки, определяя ключевые аспекты функциональности, производительности и безопасности.

Функциональные требования направлены на обеспечение основного набора возможностей системы. Важной частью системы является возможность управления проектами, включая создание новых проектов, их редактирование и удаление. Каждый проект должен содержать название, описание и привязку к ответственному пользователю. Также система должна предоставлять удобный интерфейс для отображения списка проектов с возможностью сортировки и фильтрации по различным параметрам.

Управление задачами – еще один ключевой элемент системы. Пользователи должны иметь возможность добавлять задачи с указанием заголовка, описания, статуса (например, TODO, IN\_PROGRESS, DONE) и даты завершения. Задачи должны быть связаны с конкретными проектами, а их статусы должны обновляться по мере выполнения. Важно предусмотреть возможность поиска задач по ключевым словам, дате завершения или статусу, а также отображение задач в виде списка с удобной сортировкой.

Для упрощения интеграции с другими системами предусмотрены функции экспорта и импорта данных. Система должна поддерживать экспорт информации о проектах и задачах в формате JSON, что позволяет сохранять резервные копии или интегрировать данные с другими приложениями. Импорт данных в том же формате обеспечит восстановление или перенос данных между версиями системы.

Наряду с функциональностью важно учитывать нефункциональные требования. Система должна быть производительной, обеспечивая быстрый отклик на пользовательские действия даже при большом объеме данных. Время выполнения операций, таких как создание задачи или изменение статуса, должно быть минимальным, чтобы избежать задержек в рабочем процессе.

Удобство использования является еще одним важным аспектом. Пользовательский интерфейс должен быть интуитивно понятным, а навигация – максимально простой. Это минимизирует время, необходимое для освоения системы, и позволяет сосредоточиться на выполнении задач. Также важно предусмотреть адаптивность интерфейса для корректного отображения на устройствах с различными размерами экранов.

Вопросы безопасности требуют особого внимания. Система должна защищать данные от несанкционированного доступа, используя современные механизмы аутентификации и авторизации. Необходимо обеспечить целостность данных, исключая их повреждение или утрату. Для передачи данных следует применять шифрование, что особенно актуально при экспорте и импорте информации.

Таким образом, учет всех перечисленных требований позволит создать систему, которая будет не только функциональной, но и удобной, надежной и безопасной, отвечая современным стандартам качества программного обеспечения.

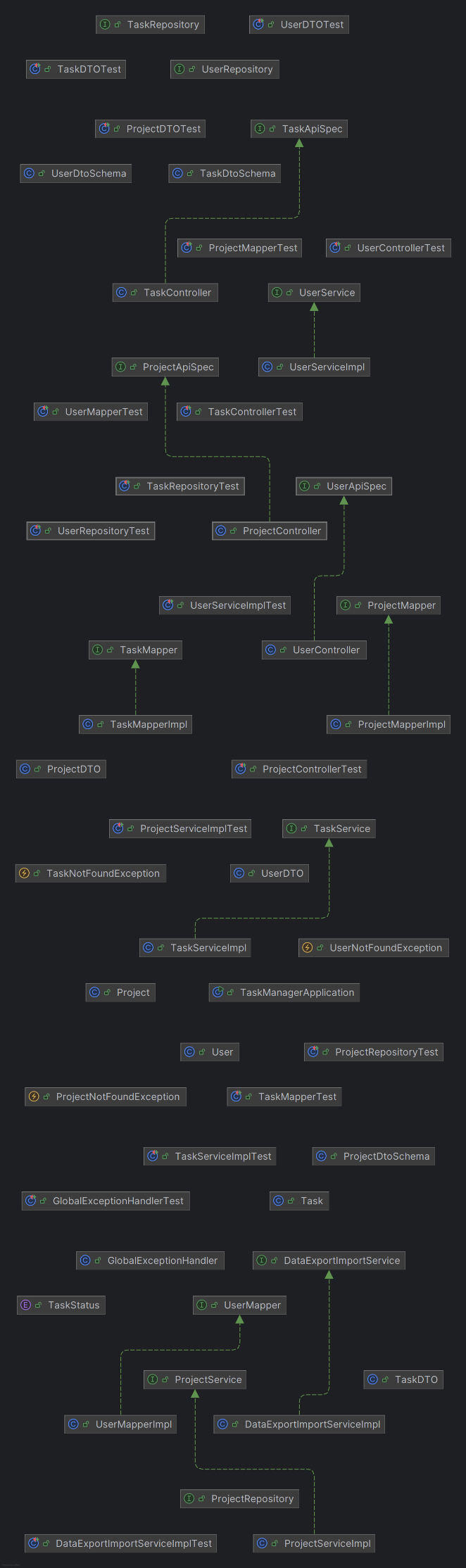
## 1.3 Выводы по анализу предметной области

Анализ предметной области показал, что существующие системы управления задачами, такие как Trello, Jira и Asana, обладают широкими возможностями, однако не всегда могут полностью удовлетворить потребности малых и средних команд. Каждая из них имеет как достоинства, так и недостатки, которые важно учитывать при выборе или разработке новой системы.

Изучение требований к системе выявило ключевые аспекты, которые должны быть реализованы для достижения высококачественного продукта. Функциональные требования, такие как управление проектами, задачами и поддержка экспорта/импорта данных, обеспечивают основные возможности системы. В то же время нефункциональные аспекты, включая производительность, удобство использования и безопасность, играют важную роль в повышении эффективности работы пользователей и надежности системы.

Особое внимание необходимо уделить интуитивному интерфейсу, который позволит минимизировать время на обучение работе с системой, и механизмам защиты данных, обеспечивающим их целостность и конфиденциальность. Эти аспекты особенно важны в современных условиях, когда требования к удобству и безопасности программного обеспечения становятся все более строгими.

Таким образом, проведенный анализ подтвердил актуальность разработки новой системы управления задачами, которая сочетает в себе простоту использования, адаптивность и функциональность. Создание такого решения позволит удовлетворить потребности целевой аудитории и устранить существующие ограничения аналогичных продуктов.



# ГЛАВА 2. ПРОЕКТИРОВАНИЕ СИСТЕМЫ

## 2.1 Архитектура приложения

Архитектура разрабатываемой системы управления задачами базируется на модели **Model-View-Controller (MVC)**, которая обеспечивает разделение приложения на логические слои. Это способствует упрощению разработки, тестирования и поддержки системы. Каждая часть архитектуры выполняет определённую функцию, обеспечивая взаимодействие между компонентами.

**Модель (Model)**

Модель отвечает за управление данными, реализацию бизнес-логики и взаимодействие с базой данных. В проекте реализованы три основные сущности: **Project**, **Task** и **User**, которые отражают ключевые элементы системы управления задачами:

* **Project** представляет проект, содержащий общую информацию, связанную с задачами, такими как название и описание.
* **Task** описывает задачи, связанные с проектами и пользователями. Она включает название, описание, статус (например, TODO, IN\_PROGRESS, DONE), а также сроки выполнения.
* **User** отвечает за представление пользователей системы, их идентификацию и привязку к проектам.

Для взаимодействия с базой данных используется **Spring Data JPA**. Этот инструмент предоставляет мощные средства для работы с базами данных, минимизируя необходимость написания SQL-запросов вручную. Репозитории, созданные на основе Spring Data JPA, позволяют реализовать основные операции (создание, чтение, обновление и удаление) с минимальными усилиями.

**Контроллеры (Controller)**

Контроллеры представляют собой посредников между моделью и запросами от пользователя. В проекте контроллеры обрабатывают HTTP-запросы и обеспечивают передачу данных в виде JSON-ответов. Используемый подход позволяет интегрировать приложение с различными клиентскими интерфейсами, такими как веб-браузеры или мобильные приложения.

Каждая из основных сущностей (проект, задача и пользователь) имеет свой отдельный контроллер, что способствует модульности и упрощает поддержку. Например:

* **ProjectController** предоставляет эндпоинты для управления проектами: создание, обновление, удаление и просмотр.
* **TaskController** отвечает за операции, связанные с задачами, включая их фильтрацию по статусу или дате.
* **UserController** позволяет управлять пользователями и их привязкой к проектам.

**Технологии, используемые в приложении**

1. **Spring Boot**

Spring Boot является основным фреймворком, используемым для разработки серверной части. Он обеспечивает лёгкость настройки, интеграцию с другими библиотеками и быстрое развертывание приложения.

2. **H2 Database**

Для хранения данных в процессе разработки и тестирования используется встроенная база данных H2. Она позволяет быстро и удобно работать с данными без необходимости настройки внешнего сервера базы данных.

3. **Swagger**

Swagger (SpringDoc OpenAPI) используется для автоматической генерации документации API. Это облегчает взаимодействие с системой и позволяет тестировать запросы через удобный веб-интерфейс.

4. **Lombok**

Lombok применяется для упрощения разработки. С его помощью генерируются стандартные методы, такие как `getters`, `setters`, `equals`, `hashCode`, а также шаблонные конструкторы. Это позволяет сократить количество шаблонного кода и сосредоточиться на логике приложения.

## 2.2 Проектирование базы данных

База данных для системы управления задачами спроектирована с учетом ключевых сущностей приложения: **Project**, **Task** и **User**. Для управления структурой базы данных используется инструмент Liquibase, который позволяет описывать изменения в схеме базы данных с помощью файлов changelog.

**Описание сущностей**

1. User (Пользователь):

* Хранит данные о пользователях системы.
* Основные атрибуты:
  + `id`: уникальный идентификатор пользователя.
  + `username`: имя пользователя.

2. Project (Проект):

* Содержит информацию о проектах, созданных пользователями.
* Основные атрибуты:
  + `id`: уникальный идентификатор проекта.
  + `name`: название проекта.
  + `description`: описание проекта.
  + `user\_id`: идентификатор пользователя, который создал проект.

3. Task (Задача):

* Представляет отдельные задачи, входящие в проекты.
* Основные атрибуты:
  + `id`: уникальный идентификатор задачи.
  + `title`: название задачи.
  + `description`: описание задачи.
  + `status`: статус задачи (например, TODO, IN\_PROGRESS, DONE).
  + `due\_date`: дата завершения задачи.
  + `project\_id`: идентификатор проекта, к которому относится задача.
  + `user\_id`: идентификатор пользователя, ответственного за задачу.

**Реализация с использованием Liquibase**

Для создания и управления схемой базы данных используется набор файлов changelog, описывающих последовательность изменений. Это обеспечивает простоту обновления и версионирования базы данных. Файл `db.changelog-master.yaml` включает ссылки на другие файлы changelog для логической организации изменений.

1. Создание схемы базы данных: файл `create-task-manager-schema.yaml`.
2. Создание таблиц:

* Таблица пользователей (users): файл ` 01\_create\_users\_table.yaml`.
* Таблица проектов (projects): файл ` 02\_create\_projects\_table.yaml `.
* Таблица задач (tasks): файл ` 03\_create\_tasks\_table.yaml `.

1. Добавление внешних ключей: файл `add-foreign-keys.yaml`.

**Реляционная модель базы данных**

На основании описанных выше требований и взаимосвязей между сущностями была построена реляционная модель базы данных:

Users:  
Первичный ключ: `id`.

Projects:  
Первичный ключ: `id`.  
Внешний ключ: `user\_id` → `users.id`.

Tasks:  
Первичный ключ: `id`.  
Внешние ключи:  
 `project\_id` → `projects.id`.  
 `user\_id` → `users.id`.

Это решение обеспечивает целостность данных, простоту работы с ними и поддержку ключевых операций, таких как создание, обновление, удаление и поиск.

## 2.3 Диаграмма классов

Диаграмма классов – это фундаментальная часть проектирования системы, которая описывает основные сущности приложения, их свойства, методы и связи между ними. В контексте проекта "Task Manager" эта диаграмма отражает архитектуру системы, показывая, как взаимодействуют основные компоненты, такие как сущности, контроллеры, сервисы и репозитории.

Основными классами в системе являются:

1. Сущности:

* User – представляет пользователя системы. Содержит поля `id` и `username`, а также список проектов, связанных с данным пользователем. Связь с классом `Project` реализована как "один ко многим" (One-to-Many).
* Project – представляет проект пользователя. Содержит поля `id`, `name`, `description`, список задач и связь с пользователем. Связь с классом `Task` также "один ко многим" (One-to-Many), что позволяет одному проекту содержать несколько задач.
* Task – представляет задачу в рамках проекта. Поля включают `id`, `title`, `description`, `status` (перечисление `TaskStatus`) и `dueDate`. Связи реализованы как "многие к одному" (Many-to-One) с классами `Project` и `User`.

1. DTO (Data Transfer Objects):

UserDTO, ProjectDTO, TaskDTO используются для передачи данных между слоями приложения. DTO помогают изолировать внутреннюю структуру сущностей от внешнего представления, обеспечивая безопасность и удобство работы.

1. Слои приложения:

* Контроллеры: классы `UserController`, `ProjectController` и `TaskController` принимают HTTP-запросы и вызывают соответствующие методы сервисов. Они являются точками входа для взаимодействия с системой через REST API.
* Сервисы: классы `UserServiceImpl`, `ProjectServiceImpl` и `TaskServiceImpl` реализуют бизнес-логику приложения. Например, в сервисе проектов происходит создание, обновление, удаление и подсчет метрик задач для проекта.
* Репозитории: интерфейсы `UserRepository`, `ProjectRepository` и `TaskRepository` предоставляют методы для взаимодействия с базой данных. Репозитории используют Spring Data JPA для реализации CRUD-операций.

4. Дополнительные компоненты:

* Мапперы (`UserMapper`, `ProjectMapper`, `TaskMapper`) обеспечивают преобразование сущностей в DTO и наоборот. Это необходимо для минимизации утечек деталей реализации в контроллеры и для работы с REST API.
* Перечисления (`TaskStatus`) определяют фиксированный набор возможных статусов задачи (`TODO`, `IN\_PROGRESS`, `DONE`).

Взаимосвязи между классами системы четко структурированы. Например, связь между сущностями `User`, `Project` и `Task` позволяет отслеживать принадлежность задач к проектам и их привязку к конкретным пользователям.

Кроме того, ключевой особенностью проектирования является использование библиотек, таких как Lombok, для сокращения объема шаблонного кода, а также применение валидаторов для проверки данных. Например, аннотации `@NotNull` и `@NotBlank` обеспечивают валидацию пользовательского ввода.

## 2.4 Диаграмма последовательности

Диаграмма последовательности демонстрирует процесс выполнения операций в приложении, начиная с поступления HTTP-запроса в контроллер и заканчивая взаимодействием с базой данных через репозитории. Рассмотрим основные сценарии:

Пример 1: Создание проекта

1. Клиент отправляет POST-запрос на `/api/projects` с JSON-объектом, содержащим данные нового проекта.
2. Контроллер `ProjectController` принимает запрос и вызывает метод `createProject` у сервиса `ProjectServiceImpl`.
3. В `ProjectServiceImpl` создается объект проекта, который передается в репозиторий `ProjectRepository` для сохранения.
4. Репозиторий сохраняет объект в базе данных и возвращает сохраненный экземпляр.
5. Сервис возвращает сохраненный проект обратно в контроллер.
6. Контроллер преобразует объект в DTO через `ProjectMapper` и отправляет его клиенту.

Пример 2: Получение задач по статусу

1. Клиент отправляет GET-запрос на `/api/tasks/filter/status?status=TODO`.
2. Контроллер `TaskController` вызывает метод `getTasksByStatus` у сервиса `TaskServiceImpl`.
3. Сервис запрашивает задачи с указанным статусом у репозитория `TaskRepository`.
4. Репозиторий выполняет SQL-запрос к базе данных и возвращает список задач.
5. Сервис преобразует сущности в DTO через `TaskMapper` и возвращает их контроллеру.
6. Контроллер отправляет результат клиенту в формате JSON.

**Основные преимущества подхода**

* Модульность: четкое разделение слоев (контроллер, сервис, репозиторий) упрощает поддержку и тестирование системы.
* Повторное использование кода: мапперы и DTO позволяют использовать универсальные компоненты для работы с данными.
* Масштабируемость: продуманная архитектура позволяет легко добавлять новые сущности и функционал без изменения существующего кода.

Эти UML-диаграммы подчеркивают, как спроектирована система, обеспечивая наглядное представление её работы. Диаграмма классов помогает понять статическую структуру приложения, а диаграмма последовательности раскрывает динамику выполнения операций.

## 2.5 Конфигурация приложения

### 2.5.1 Файл `application.yml` и его роли

Файл `application.yml` служит основным инструментом конфигурации приложения в Spring Boot. Он предоставляет возможность централизованного управления настройками, что делает приложение гибким, легко настраиваемым и удобным в поддержке.

Для проекта "Task Manager" файл `application.yml` выполняет следующие роли:

1. Настройка сервера:

- Указание порта, на котором будет работать сервер:

server:

port: 5555

Эта настройка позволяет задать уникальный порт для работы приложения, избегая конфликтов с другими сервисами.

1. Конфигурация источника данных:

- Для хранения данных используется база данных H2 в памяти:

spring:

datasource:

url: jdbc:h2:mem:taskmanagerdb

driver-class-name: org.h2.Driver

username: ${H2\_DB\_USERNAME}

password: ${H2\_DB\_PASSWORD}

Переменные окружения `H2\_DB\_USERNAME` и `H2\_DB\_PASSWORD` обеспечивают безопасность конфиденциальных данных, таких как логин и пароль.

1. Настройки Hibernate:

- Автоматическое создание и обновление схемы базы данных:

jpa:

hibernate:

ddl-auto: update

show-sql: true

Это упрощает процесс разработки, так как схема базы данных автоматически синхронизируется с сущностями.

1. Консоль H2:

- Включение веб-консоли H2 для визуального доступа к базе данных:

h2:

console:

enabled: true

Консоль позволяет разработчику проверять состояние базы данных, что особенно полезно на этапе отладки.

5. Интеграция Swagger:

- Конфигурация Swagger UI для документирования API:

springdoc:

swagger-ui:

enabled: true

path: /swagger-ui.html

api-docs:

path: /v3/api-docs

Swagger предоставляет удобный интерфейс для тестирования REST API, что облегчает взаимодействие с клиентами и сторонними разработчиками.

### 2.5.2 Конфигурация CI/CD пайплайна

Автоматизация процессов сборки, тестирования и развертывания приложения является ключевым аспектом современного DevOps. Для этого в проекте используется GitHub Actions с конфигурацией CI/CD пайплайна. Файл .github/workflows/ci-pipeline.yml.

1. Триггеры выполнения:

Пайплайн запускается при создании pull request или пуше в ветку `main`. Это гарантирует, что каждая новая функция или исправление проходят проверку перед интеграцией.

1. Основные этапы пайплайна:

Сборка проекта:

* Код проекта извлекается из репозитория с помощью шага `Checkout code`.
* С помощью `actions/setup-java` настраивается JDK версии 22.
* Команда `mvn clean install` выполняет сборку проекта.

Тестирование:

* Шаг `Run tests` запускает тесты, чтобы гарантировать работоспособность приложения.

Сборка JAR-файла:

* Команда `mvn package` создает JAR-файл приложения.

Сохранение артефакта:

* Готовый JAR-файл загружается как артефакт, доступный для дальнейшего использования (например, развертывания).

1. Преимущества конфигурации CI/CD:

* Автоматизация процессов: разработчики могут сосредоточиться на написании кода, а не на ручных процессах сборки и тестирования.
* Стабильность кода: каждая новая функция проходит тестирование перед интеграцией в основную ветку, что снижает вероятность ошибок.
* Удобство развертывания: готовые JAR-файлы можно использовать для автоматизированного или ручного деплоя.

Конфигурация приложения объединяет использование Spring Boot и GitHub Actions для обеспечения гибкости, надежности и автоматизации процессов. Файл `application.yml` упрощает управление параметрами приложения, а CI/CD пайплайн гарантирует стабильность и качество кода, повышая производительность команды разработчиков.

## 2.6 Выводы по проектированию

Проектирование системы управления задачами включало комплексный подход к разработке архитектуры, базы данных, диаграмм UML и конфигурации приложения. На основании проведенной работы можно выделить следующие ключевые аспекты:

1. Архитектура приложения:

* Использование модели MVC (Model-View-Controller) обеспечивает четкое разделение логики, представления и взаимодействия с данными, что улучшает модульность и упрощает сопровождение системы.
* Выбранные технологии – Spring Boot, Lombok, H2, Swagger и другие – способствуют ускорению разработки и тестирования, обеспечивая функциональность, производительность и удобство использования.

1. Проектирование базы данных:

* Разработанная структура базы данных с использованием сущностей Project, Task и User охватывает все ключевые аспекты управления задачами и проектами.
* Реляционная модель базы данных поддерживает целостность данных за счет связей между таблицами, что упрощает выполнение сложных запросов и аналитики.

1. Диаграммы UML:

* Диаграмма классов позволила визуализировать структуру объектов системы, их атрибуты и связи, что упростило реализацию кода.
* Диаграмма последовательности помогла формализовать взаимодействие между компонентами, обеспечив ясность процессов, таких как создание, обновление и удаление задач или проектов.

1. Конфигурация приложения:

* Использование файла `application.yml` упростило настройку параметров приложения, таких как подключение к базе данных, конфигурация сервера и интеграция Swagger UI.
* Реализация CI/CD пайплайна на GitHub Actions автоматизировала процессы сборки, тестирования и развертывания приложения, что повысило надежность и снизило вероятность ошибок при разработке.

1. Гибкость и масштабируемость:

* Проектирование было выполнено с учетом возможности масштабирования, что делает систему пригодной для использования как индивидуальными пользователями, так и малым и средним бизнесом.
* Упор на модульность и четкую организацию кода облегчит дальнейшее развитие проекта, добавление новых функций и адаптацию под изменяющиеся требования.

Таким образом, процесс проектирования заложил основу для успешной реализации системы управления задачами, отвечающей современным требованиям качества, удобства использования и надежности. Все принятые решения были направлены на достижение целей проекта и упрощение его сопровождения в будущем.

# ГЛАВА 3. РЕАЛИЗАЦИЯ СИСТЕМЫ

## 3.1 Разработка модулей приложения: Backend и REST API

Разработка backend-части приложения основывалась на архитектурном подходе REST, что позволило обеспечить простоту взаимодействия между клиентами и сервером. Основная задача backend — предоставление API для выполнения CRUD-операций (создание, чтение, обновление и удаление) над сущностями Project, Task и User.

Основные компоненты реализации REST API:

* 1. Контроллеры:

Контроллеры отвечают за обработку HTTP-запросов и маршрутизацию. Каждый контроллер реализует спецификацию API, определенную интерфейсами `ProjectApiSpec`, `TaskApiSpec` и `UserApiSpec`.

**ProjectController** управляет операциями с проектами, включая:

* Получение списка всех проектов.
* Получение проекта по ID.
* Создание нового проекта.
* Обновление проекта.
* Удаление проекта.
* Получение метрик задач, связанных с проектом.

**TaskController** отвечает за задачи:

* Получение всех задач.
* Фильтрация задач по статусу или сроку выполнения.
* Поиск задач по ключевым словам.
* Добавление, обновление и удаление задач.

**UserController** управляет пользователями:

* Получение всех пользователей.
* Создание, обновление и удаление пользователей.
* Экспорт и импорт данных о пользователях.

Все контроллеры снабжены аннотациями OpenAPI для генерации спецификаций Swagger, что упрощает документирование API.

* 1. Сервисный слой:

Вся бизнес-логика приложения сосредоточена в сервисах. Этот слой инкапсулирует операции над сущностями, предоставляя интерфейсы для работы с данными.

**ProjectServiceImpl** управляет созданием, обновлением и удалением проектов, а также расчетом метрик задач.

**TaskServiceImpl** обрабатывает запросы, связанные с задачами, включая поиск, фильтрацию и изменение статусов.

**UserServiceImpl** выполняет CRUD-операции над пользователями.

* 1. Репозитории:

Используются интерфейсы `JpaRepository`, предоставляющие готовую реализацию для базовых операций. Также добавлены пользовательские запросы через аннотации `@Query` для сложных операций:

* Поиск задач по ключевым словам.
* Фильтрация задач по сроку выполнения.
  1. Мапперы:

Для преобразования между сущностями и DTO использовались мапперы, созданные с использованием библиотеки MapStruct. Это упрощает передачу данных между слоями приложения, улучшая читаемость и уменьшая вероятность ошибок.

**Обеспечение безопасности и валидации:**

1. Валидация данных:

Использовались аннотации Bean Validation (`@NotNull`, `@NotBlank`, `@Future`) для проверки корректности данных на уровне DTO.

1. Глобальная обработка исключений:

С помощью класса `GlobalExceptionHandler` реализована централизованная обработка ошибок, таких как:

* Ошибки при поиске несуществующих записей (например, `ProjectNotFoundException`).
* Ошибки при некорректных данных.

**Преимущества реализованного API:**

1. Стандартизация:

Все методы API реализованы согласно REST-принципам, обеспечивая четкое и предсказуемое взаимодействие с клиентами.

1. Документированность:

Интеграция Swagger UI позволяет разработчикам и пользователям легко ознакомиться с доступными эндпоинтами.

1. Расширяемость:

Архитектура API позволяет легко добавлять новые функции, не затрагивая существующую реализацию.

1. Гибкость:

Использование DTO обеспечивает возможность модификации формата данных, передаваемых клиентам, без изменения основной бизнес-логики.

Таким образом, backend-часть системы была спроектирована и реализована с упором на масштабируемость, читаемость кода и удобство использования для клиентов API.

## 3.2 Тестирование системы

Тестирование является важной частью разработки, обеспечивающей надежность, стабильность и соответствие системы требованиям. В процессе тестирования системы были разработаны юнит- и интеграционные тесты, охватывающие ключевые компоненты.

**Юнит-тесты: Описание покрытых компонентов**

Юнит-тесты направлены на проверку отдельных модулей, таких как контроллеры, сервисы, репозитории и мапперы. Основная цель — убедиться, что каждая часть системы работает в соответствии с ожидаемым поведением.

1. Контроллеры

Для каждого контроллера были разработаны тесты, покрывающие их основные методы. Тесты выполнялись с использованием mock-объектов для изоляции от других частей системы.

**ProjectControllerTest**:

* Тестирование CRUD-операций с проектами:
* Создание, обновление, удаление проектов.
* Получение списка проектов и информации о проекте по ID.
* Проверка метрик задач проекта.
* Тестирование функционала экспорта/импорта данных проектов.
* Использовались mock-объекты для сервисов `ProjectServiceImpl`, мапперов `ProjectMapper`, а также модуля экспорта/импорта данных.

**TaskControllerTest**:

* Тесты для методов:
* Создание, обновление и удаление задач.
* Получение списка задач, поиск по ключевым словам, фильтрация по статусу и дате.
* Проверка интеграции с `TaskServiceImpl` и `TaskMapper`.

**UserControllerTest**:

* Тесты на создание, удаление и обновление пользователей.
* Проверка методов импорта и экспорта данных пользователей.

1. Сервисы

Тестирование сервисного слоя направлено на проверку бизнес-логики.

**ProjectServiceImplTest**:

* Проверка работы методов CRUD.
* Обработка исключений (например, `ProjectNotFoundException` при попытке работы с несуществующим проектом).
* Тестирование вычисления метрик задач проекта.

**TaskServiceImplTest**:

* Проверка методов получения задач по статусу, сроку выполнения и ключевым словам.
* Валидация поведения методов при некорректных данных (например, отсутствие задачи с заданным ID).

**UserServiceImplTest**:

* Тестирование CRUD-операций с пользователями.
* Проверка обработки ошибок, таких как `UserNotFoundException`.

1. Репозитории

Для репозиториев с использованием Spring Data JPA были разработаны тесты с аннотацией `@DataJpaTest`, чтобы проверить корректность работы запросов.

**ProjectRepositoryTest**:

* Проверка сохранения, поиска и удаления проектов.

**TaskRepositoryTest**:

* Проверка методов `findByStatus`, `findByDueDate` и `searchByKeyword`.

**UserRepositoryTest**:

* Проверка базовых операций (сохранение, поиск, удаление) для пользователей.

1. Мапперы

Использование библиотеки MapStruct обеспечило автоматическое преобразование между сущностями и DTO. Тесты мапперов покрывали:

* Преобразование DTO в сущности и обратно.
* Проверку обновления существующих объектов с использованием DTO.
* Проверку обработки null-значений и пустых коллекций.

**Интеграционные тесты**

Интеграционные тесты проверяли взаимодействие компонентов системы, таких как контроллеры, сервисы и репозитории. Они обеспечивали более глубокую проверку, включая работу с базой данных.

Подход к тестированию:

* Использование mock-объектов для замены взаимодействий между слоями.

Проверка корректности взаимодействия слоев системы:

* Отправка HTTP-запросов в контроллеры.
* Проверка обработки запросов на уровне сервисов и репозиториев.
* Тестирование полного жизненного цикла данных (создание → обновление → удаление).

Проверяемые сценарии:

* Обработка ошибок в цепочке контроллер → сервис → репозиторий.
* Генерация исключений (например, при работе с несуществующими ID).
* Корректность изменений в базе данных после выполнения операций.

**Инструменты и методологии**

1. Mocking:

* Использовались библиотеки Mockito для изоляции тестируемых модулей.
* Mock-объекты заменяли зависимости, такие как сервисы и репозитории.

1. JUnit 5:

* Основной фреймворк для написания тестов.
* Использовались аннотации `@Test`, `@BeforeEach`, `@Mock` и `@InjectMocks`.

1. H2 Database:

* Для локального хранения данных использовалась встроенная база данных H2.

Тестирование системы позволило выявить и устранить большинство ошибок на этапе разработки, повысив стабильность и надежность системы перед её развертыванием.

## 3.3 Инструменты для обеспечения качества кода

Обеспечение качества кода в проекте стало ключевым аспектом разработки, так как это позволило достичь высокой надежности, стабильности и поддерживаемости системы. В рамках проекта применялись следующие подходы и инструменты: CI/CD процессы, код ревью и автоматизированное тестирование.

**Использование CI/CD**

Для организации непрерывной интеграции и доставки (CI/CD) был настроен пайплайн на основе GitHub Actions. Пайплайн автоматизировал следующие этапы разработки:

* Сборка проекта: выполнялась команда `mvn clean install`, которая обеспечивала сборку проекта с учетом всех зависимостей.
* Запуск тестов: Тестирование проекта на каждом этапе изменений с помощью команды `mvn test` гарантировало, что новый код не нарушает работу существующего функционала.
* Создание артефактов: после успешного прохождения тестов создавался JAR-файл, который готовился для дальнейшего использования.
* Управление триггерами запуска: пайплайн запускался автоматически при создании Pull Request или внесении изменений в основную ветку, что позволяло оперативно проверять корректность изменений.

Данная автоматизация процессов минимизировала человеческий фактор и ускорила цикл разработки, позволяя разработчикам сосредоточиться на улучшении функционала.

**Код ревью**

Код ревью являлось обязательным этапом перед объединением изменений в основную ветку. Все изменения, предложенные разработчиками, проверялись на предмет:

* Читаемости кода: Использование понятных имен переменных, методов и классов.
* Соответствия архитектурным принципам: Следование паттернам проектирования и правилам разделения ответственности.
* Качества тестов: Проверка полноты покрытия тестами новых модулей и сценариев.

Процесс код ревью способствовал повышению уровня профессиональной ответственности команды, снижению числа ошибок в коде и обеспечению долгосрочной поддерживаемости проекта.

**Автоматизированное тестирование**

Автоматизированное тестирование включало написание юнит- и интеграционных тестов, что позволило протестировать отдельные модули и взаимодействие между ними.

* Юнит-тесты: Проверяли изолированную работу контроллеров, сервисов, репозиториев и мапперов. Для их реализации использовалась библиотека Mockito, которая обеспечивала создание mock-объектов и изоляцию тестируемых компонентов.
* Интеграционные тесты: Проверяли взаимодействие компонентов системы, включая контроллеры, сервисы и репозитории. Использование встроенной базы данных H2 позволяло воспроизводить сценарии работы с реальными данными. Проверялись сценарии полного жизненного цикла данных – создание, обновление, удаление и поиск.

**Результаты применения инструментов**

Совместное использование CI/CD, код ревью и автоматизированного тестирования позволило сократить число ошибок, ускорить процесс разработки и обеспечить соответствие системы функциональным и нефункциональным требованиям. Такой подход значительно повысил общее качество проекта и упростил его дальнейшую поддержку.

# ЗАКЛЮЧЕНИЕ

Для достижения поставленных целей и выполнения задач, определенных на этапе планирования, была проведена комплексная работа по проектированию, разработке и внедрению программного обеспечения. Итоги проделанной работы показывают, что все намеченные цели были достигнуты, а система, созданная в рамках проекта, соответствует заявленным требованиям и обладает широкими возможностями для дальнейшего развития.

Основная цель разработки заключалась в создании системы управления проектами, задачами и пользователями, которая была бы удобной в использовании, функциональной и надежной. Эта система должна обеспечивать автоматизацию рутинных процессов, минимизацию ошибок и повышение общей эффективности управления данными. В процессе реализации данной цели было выполнено множество задач, включая разработку архитектуры приложения, создание и тестирование отдельных компонентов, а также интеграцию всех частей в единую систему.

Важным аспектом работы стало проектирование удобного и интуитивно понятного пользовательского интерфейса. Это позволило пользователям без труда работать с системой, используя основные функции: создание, редактирование и удаление проектов, задач и пользователей, поиск данных по ключевым словам, фильтрацию по статусу и дате выполнения. Были учтены требования гибкости и масштабируемости, что дало возможность интеграции системы с внешними сервисами и дальнейшего её расширения.

Разработка функционала для работы с данными включала в себя реализацию операций импорта и экспорта, поддержку REST API и работу с базой данных H2. Это делает систему пригодной для интеграции с другими приложениями и внешними системами, а также для адаптации под специфические задачи, которые могут возникнуть у пользователей. Реализованные функции, такие как метрики задач и анализ данных, обеспечивают дополнительную ценность системы для конечных пользователей.

Особое внимание в процессе разработки уделялось качеству программного обеспечения. Для этого были разработаны и внедрены юнит- и интеграционные тесты, которые покрыли основные компоненты системы, включая контроллеры, сервисы, репозитории и мапперы. Это позволило обеспечить стабильность и надежность работы приложения. Автоматизация тестирования с использованием современных библиотек, таких как Mockito и JUnit 5, значительно сократила время на поиск и устранение ошибок, улучшив общую производительность команды разработчиков.

Проектирование архитектуры приложения основывалось на принципах модульности и разделения ответственности. Это позволило создать код, который легко поддается поддержке, расширению и модификации. Например, слой контроллеров отвечает исключительно за обработку запросов, слой сервисов реализует бизнес-логику, а репозитории занимаются взаимодействием с базой данных. Такой подход способствует прозрачности работы приложения и упрощает внесение изменений в случае необходимости.

Значимость результата данного проекта трудно переоценить. Разработанная система решает множество практических задач, связанных с управлением информацией. Она обеспечивает удобство и точность обработки данных, автоматизирует рутинные операции и минимизирует человеческий фактор. Эти преимущества делают систему полезным инструментом для управления проектами, задачами и ресурсами в различных сферах деятельности.

Несмотря на достигнутые результаты, существует множество направлений для дальнейшего развития системы. Одним из наиболее очевидных путей является расширение функциональности. Например, можно внедрить систему напоминаний, которая будет уведомлять пользователей о приближающихся сроках выполнения задач. Это поможет лучше организовать рабочий процесс и своевременно реагировать на изменения. Также возможна интеграция с внешними сервисами, такими как Google Calendar или мессенджеры, что сделает систему еще более удобной и универсальной.

Другим важным направлением является внедрение аналитических инструментов. Например, можно реализовать функции генерации отчетов и визуализации данных. Пользователи смогут видеть динамику выполнения задач, распределение ресурсов и другие важные показатели, что повысит информированность и упростит принятие управленческих решений. Такие улучшения особенно актуальны для компаний и организаций, где управление проектами играет ключевую роль.

Оптимизация производительности системы также представляет собой перспективное направление развития. В случае увеличения объема данных или нагрузки на приложение можно внедрить механизмы кэширования, которые ускорят обработку информации. Переход на распределенные базы данных или использование облачных решений позволят системе справляться с большими объемами данных, сохраняя высокую скорость работы. Такие меры обеспечат масштабируемость системы и сделают её пригодной для использования в условиях роста бизнеса.

Для обеспечения надежной работы приложения в долгосрочной перспективе необходимо также уделить внимание вопросам безопасности. Например, можно внедрить систему аутентификации и авторизации пользователей, а также реализовать защиту данных от несанкционированного доступа. Это особенно важно в условиях работы с конфиденциальной информацией, такой как данные пользователей или проекты компании.

Возможности интеграции с новыми технологиями также открывают широкие перспективы для развития системы. Например, использование машинного обучения может помочь автоматизировать анализ данных или прогнозирование выполнения задач. Такие функции могут быть полезны для крупных компаний, где необходимо обрабатывать большие объемы информации и принимать решения на основе аналитических данных.

Еще одним направлением развития является улучшение пользовательского интерфейса. Можно добавить больше настроек для кастомизации интерфейса под потребности конкретных пользователей. Например, предоставление возможности создавать собственные фильтры или настраивать уведомления сделает систему более гибкой и удобной. Также можно разработать мобильное приложение, что обеспечит доступ к системе с любого устройства.

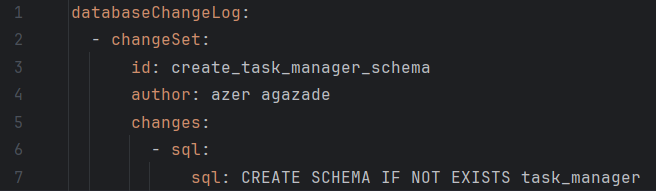
Подводя итог, можно сказать, что проект представляет собой прочную основу для дальнейшего развития. Достигнутые результаты подтверждают высокую эффективность выбранного подхода к разработке, а также его применимость в реальных условиях. Благодаря гибкости, масштабируемости и возможности интеграции с внешними системами, созданное приложение способно удовлетворить потребности самых разных пользователей. Реализация предложенных направлений улучшений позволит не только повысить функциональность системы, но и сделать её еще более востребованной и полезной в современном мире.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Craig Walls. Spring in Action. Manning Publications, 6th Edition, 2022. Руководство по Spring Framework, содержащее подробное описание ключевых возможностей Spring, включая Spring Boot.
2. Christian Bauer, Gavin King, и Gary Gregory. Java Persistence with Hibernate. Manning Publications, 2nd Edition, 2015. Основной источник знаний о работе с Hibernate и JPA.
3. Apache Maven Project. Maven: The Complete Reference. Документация Maven, доступная на официальном сайте: <https://maven.apache.org/>
4. Joshua Bloch. Effective Java. Addison-Wesley, 3rd Edition, 2018. Книга, описывающая лучшие практики программирования на Java, включая работу с объектами, потоками, и коллекциями.
5. H2 Database Engine. Official Documentation. Документация H2 базы данных, доступная на: <http://h2database.com/html/main.html>
6. Mockito Team. Mockito Documentation. Руководство по библиотеке Mockito, доступное на: <https://site.mockito.org/>
7. JUnit 5 Team. JUnit 5 User Guide. Руководство по тестированию с использованием JUnit 5, доступное на: <https://junit.org/junit5/docs/current/user-guide/>
8. Mark Lutz. Programming Python. O’Reilly Media, 4th Edition, 2010. Дополнительный материал по программированию, который применялся для анализа структур данных и взаимодействия с внешними системами.
9. Baeldung. Spring Tutorials. Серия статей по Spring Framework и Spring Boot, доступных на: <https://www.baeldung.com/>
10. Oracle. Java SE Documentation. Официальная документация Java SE, доступная на: <https://docs.oracle.com/en/java/javase/>
11. JetBrains. IntelliJ IDEA User Guide. Руководство пользователя для интегрированной среды разработки IntelliJ IDEA, доступное на: <https://www.jetbrains.com/idea/documentation/>
12. Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2nd Edition, 2018. Книга о рефакторинге и улучшении существующего кода для повышения его читаемости и поддержки.
13. Документация GitHub Actions. Официальное руководство по настройке и использованию CI/CD пайплайнов: <https://docs.github.com/en/actions>
14. Spring Boot Reference Documentation. Полное руководство по работе с Spring Boot, доступное на: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>
15. JSON.org. Основная информация о формате JSON, доступная на: <https://www.json.org/>
16. Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 1st Edition, 2003. Книга о концепциях проектирования, применимых в сложных бизнес-доменах.
17. Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 1st Edition, 2008. Руководство по написанию читаемого, поддерживаемого и эффективного кода.
18. Martin Kleppmann. Designing Data-Intensive Applications. O’Reilly Media, 1st Edition, 2017. Книга о проектировании надежных, масштабируемых систем работы с данными.
19. Spring Security Reference Documentation. Официальная документация Spring Security для интеграции с Spring Boot: <https://docs.spring.io/spring-security/reference/>
20. Vlad Mihalcea. High-Performance Java Persistence. Amazon Digital Services LLC, 1st Edition, 2016. Руководство по оптимизации производительности JPA и Hibernate.
21. Docker Documentation. Документация по настройке и использованию Docker: <https://docs.docker.com/>
22. RESTful Web Services Documentation. Стандартные практики и принципы построения RESTful API: <https://restfulapi.net/>
23. Postman API Platform Documentation. Руководство по тестированию API с использованием Postman: <https://www.postman.com/api-documentation/>
24. OpenAPI Initiative. OpenAPI Specification. Документация для проектирования API: https://swagger.io/specification/
25. Linus Torvalds. Git Documentation. Официальное руководство по использованию Git: <https://git-scm.com/doc>
26. Bruce Eckel. Thinking in Java. Prentice Hall, 4th Edition, 2006. Классическое руководство для глубокого понимания основ и тонкостей языка программирования Java.
27. Fowler, Beck, Brant, et al. Patterns of Enterprise Application Architecture. Addison-Wesley, 1st Edition, 2002. Книга о шаблонах проектирования для корпоративных приложений, включая архитектурные решения и их реализацию.

# ПРИЛОЖЕНИЯ

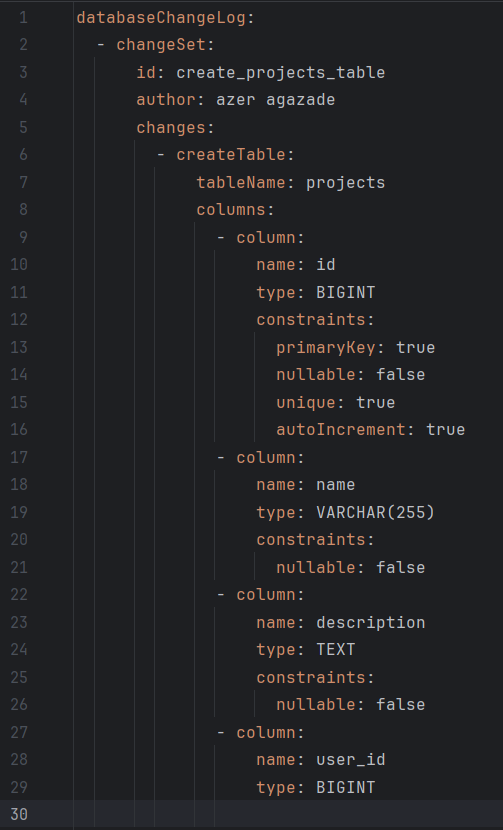
Файл create-task-manager-schema.yaml:



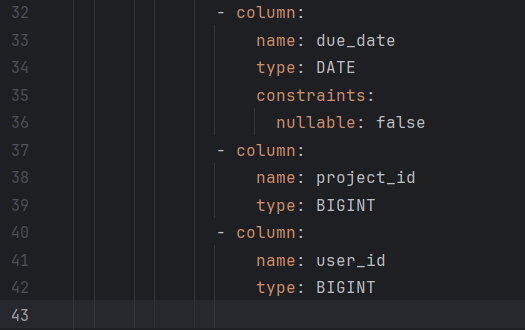
Файл 01\_create\_users\_table.yaml:



Файл 02\_create\_projects\_table.yaml:

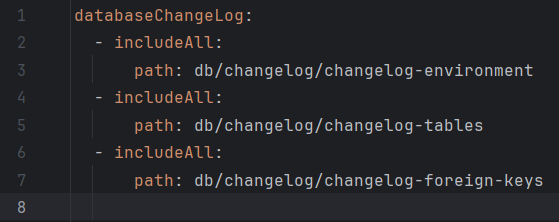


Файл 03\_create\_tasks\_table.yaml: 

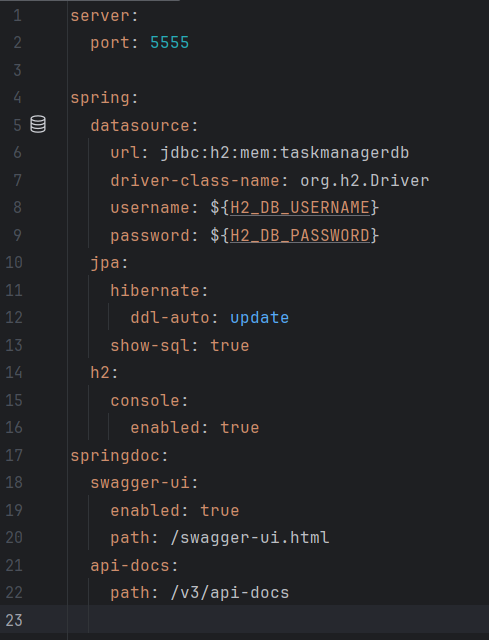


Файл add-foreign-keys.yaml: 

Файл db.changelog-master.yaml:



Файл application.yml:



Файл .github/workflows/ci-pipeline.yml:

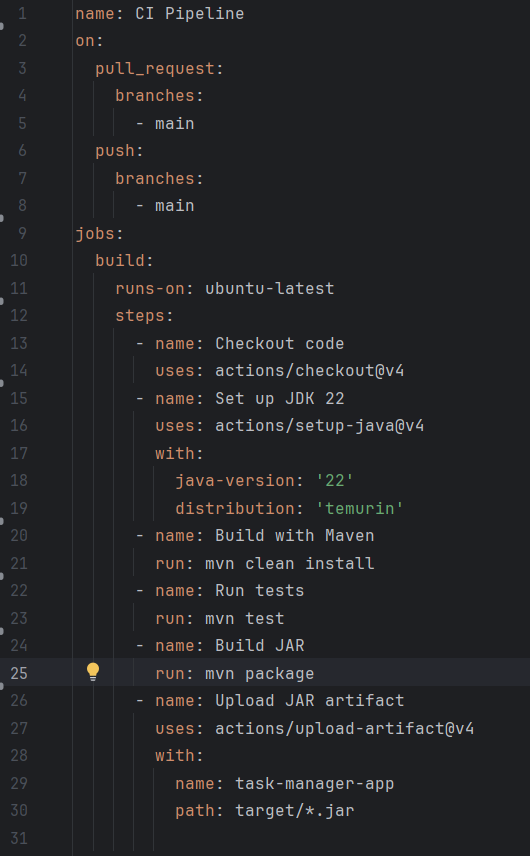
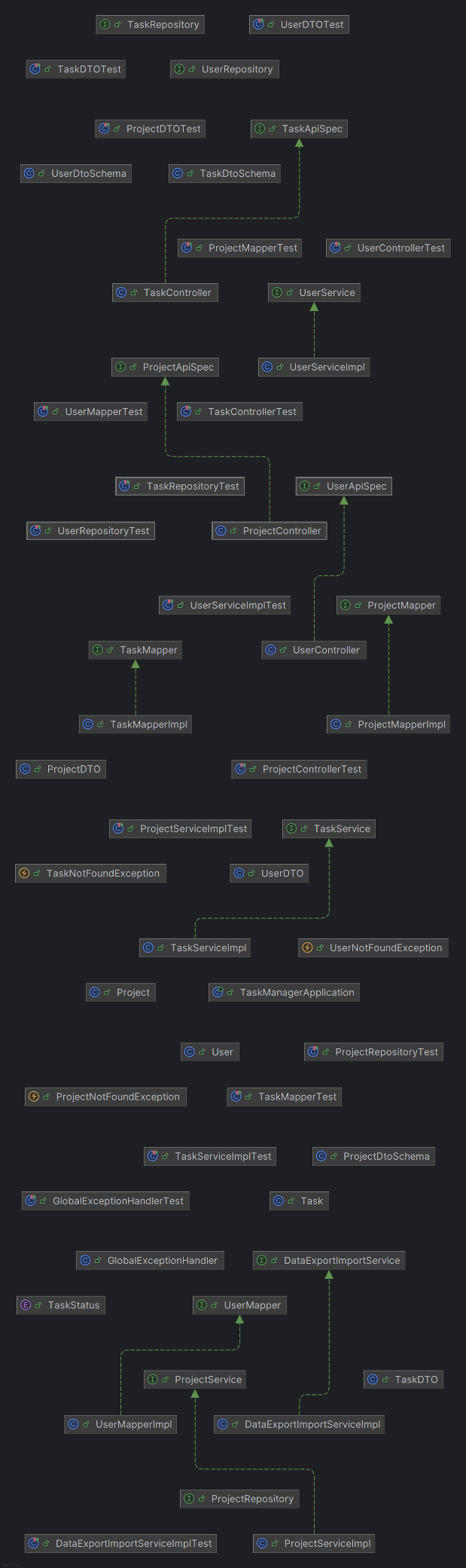


Диаграмма классов:



Ссылка на проект на гитхабе:

<https://github.com/AgazadeAV/task-manager>