Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «**Информатика и системы управления**»

КАФЕДРА «**Программное обеспечение ЭВМ и информационные технологии**»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

Создание информационной системы для онлайн трекера времени

Студент группы **ИУ7-63Б** _____ **Д.Ю Неумоин**
(Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ **Д.А. Кузнецов**
(Подпись, дата) (И.О. Фамилия)

***2023** г.*

РЕФЕРАТ

Объектом разработки является информационная система для онлайн трекера времени.

Цель работы – спроектировать и реализовать базу данных, содержащую данные учёта затраченного времени пользователей, разработать программное обеспечение, которое позволит работать с этой базой данных.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- проанализировать варианты модели данных и выбрать подходящий вариант для решения задачи;
- проанализировать существующие СУБД и выбрать удовлетворяющие требованиям к хранению данных;
- спроектировать базу данных, описать ее сущности и связи;
- реализовать программное обеспечение, предоставляющее интерфейс доступа к данным.

Готовое приложение позволяет создавать записи о затраченном времени, создавать проекты, цели, а также добавлять пользователей в друзья и смотреть их статистику.

Приложение реализовано на языке Golang, в качестве СУБД была выбрана PostgreSQL.

Результатом исследования является уменьшение времени ответа конечной точки за счет кеширования ответов на 15%.

КЛЮЧЕВЫЕ СЛОВА: трекер времени, базы данных, SQL, NoSQL, PostgreSQL, Redis, Docker, Golang, REST API, кеширование.

Расчетно-пояснительная записка 61 с., 8 рис., 9 табл., 17 ист., 5 прил.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире все больше людей сталкиваются с проблемой управления временем. Неумение правильно распределить свое время может привести к неприятным последствиям, как для личной жизни, так и для работы. Именно поэтому создание информационной системы для онлайн трекера времени является актуальной темой для исследования. Ее основная задача заключается в организации записи и анализа затраченного времени, что позволяет максимально эффективно использовать его в дальнейшем.

Реализация системы требует наличия базы данных и методов взаимодействия с ней.

Цель работы – спроектировать и реализовать базу данных, содержащую данные учёта затраченного времени пользователей, разработать программное обеспечение, которое позволит работать с этой базой данных.

Для достижения поставленной цели, требуется решить следующие задачи:

- проанализировать варианты модели данных и выбрать подходящий вариант для решения задачи;
- проанализировать существующие СУБД и выбрать удовлетворяющие требованиям к хранению данных;
- спроектировать базу данных, описать ее сущности и связи;
- реализовать программное обеспечение, которое позволит получить доступ к данным.

1 Аналитическая часть

1.1 Формализация задачи

В данной курсовой работе будут использоваться следующие понятия: запись времени, тег, проект, цель.

Запись времени – это временной интервал, содержащий название деятельности, которой занимался человек в течении этого интервала.

Тег – это метка, присваиваемая к объекту записи. Теги используются для категоризации, организации и быстрого поиска информации. В контексте онлайн трекера времени теги могут использоваться для классификации записей времени по проектам, клиентам или задачам.

Проект – специальная метка, относящая запись времени к определенной повторяющейся деятельности, например ”спорт” или ”курсовой проект”.

Пример структуры записи:

- ”13:00 - 14:00 | 1 час” – интервал;
- ”аналитическая часть” – название деятельности;
- ”курсовой проект” – проект;
- ”за компьютером” – тег.

Цель – временной интервал, в течении которого нужно потратить заданное количество часов на определенный проект. Пример цели – с 1.09.2023 по 9.09.2023 потратить на проект ”Курсовая работа” 10 часов.

Возможности пользователя:

- создание проекта;
- создание записи;
- создание тега;
- составление целей;
- добавление других пользователей в друзья;
- отслеживание временных затрат за определенный период пользователя и его друзей;
- отслеживание выполнения целей.

На рисунке 1 представлена диаграмма использования приложения.

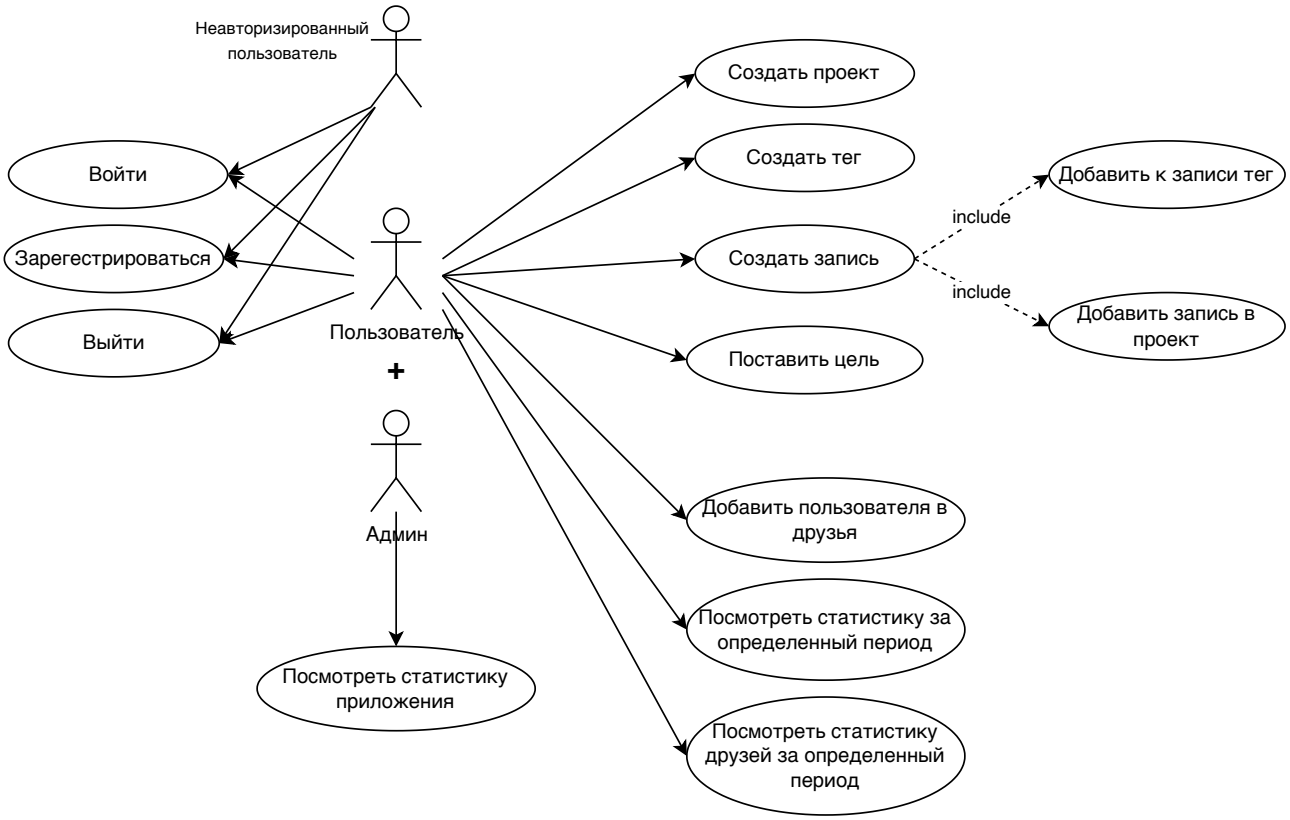

Рисунок 1 – Диаграмма использования приложения

1.2 Пользователи и данные в системе

Система, разрабатываемая в рамках курсового проекта, предполагает собой REST-API веб сервис [2].

Доступ к системе имеет разграничение по ролям: неавторизированный пользователь, авторизированный пользователь и администратор. При этом должно быть ограничение доступа к данным в зависимости от роли пользователя.

Функционал каждой роли представлен в таблице 1.

Таблица 1 – Типы пользователей и их функционал

Тип пользователя	**Функционал**
Неавторизованный пользователь	Регистрация, авторизация
Авторизованный пользователь	Создание проекта, тега, записи, цели, добавление в друзья, просмотр статистики друзей, отслеживание выполнения целей
Администратор	Весь функционал авторизированного пользователя, просмотр данных каждого пользователя без добавления в друзья, просмотр статистики приложения(сколько пользователей, проектов, целей)

Данные в приложении можно разделить на несколько типов:

- сессии;
- данные о пользователях;
- записи времени;
- проекты;
- теги;
- связи между пользователями;
- цели.

1.3 Модели данных

Модель данных – это абстрактное, самодостаточное, логическое определение объектов, операторов и прочих элементов, в совокупности составляющих абстрактную машину доступа к данным, с которой взаимодействует пользователь. Эти объекты позволяют моделировать структуру данных, а операторы — поведение данных[1].

Существует 4 основных типа моделей организации данных:

- иерархическая;
- сетевая;
- реляционная;
- нереляционная.

1.3.1 Иерархическая модель

В иерархической модели данных используется представление базы данных в виде древовидной структуры, состоящей из объектов разных уровней. Среди объектов существуют связи, любой объект может включать в себя несколько объектов более низкого уровня. Такие объекты находятся в отношении предка к потомку, при этом вероятна ситуация, когда объект-предок имеет несколько потомков, тогда как у объекта-потомка обязательно должен быть только один предок.

Основной недостаток иерархической модели данных заключается в ее ограничении на строго иерархические отношения между элементами что приводит к тому, что невозможно реализовать отношение ”многие ко многим ” .

1.3.2 Сетевая модель

Сетевая модель – это структура, у которой любой элемент может быть связан с любым другим элементом.

Сетевые базы данных, являются своеобразной модификацией иерархических баз данных. Отличаются от иерархических лишь тем, что у дочернего элемента может быть несколько предков, то есть, элементов стоящих выше него.

Достоинством сетевой модели данных является ее быстродействие в отличие от иерархической БД, гибкость в хранении данных, универсальность в сравнении с другими моделями, а также возможность доступа к данным через значения нескольких отношений.

Основным недостатком является то, что сетевая модель не поддерживает простого доступа к данным, так как для доступа к определенной записи необходимо проходить через несколько уровней связи, что затрудняет работу с данными и усложняет их поддержку и модификацию.

1.3.3 Реляционная модель

Реляционная модель данных является совокупностью данных и состоит из набора двумерных таблиц. При табличной организации отсутствует иерархия элементов. Таблицы состоят из строк – записей и столбцов – полей. На пересечении строк и столбцов находятся конкретные значения. Для каждого поля определяется множество его значений. За счет возможности просмотра строк и столбцов в любом порядке достигается гибкость выбора подмножества элементов.

Взаимодействие с реляционными базами данных осуществляется с помощью SQL (Structured Query Language)[4], стандартизированного языка запросов данных.

Работа с реляционными базами данных базируется на понятии внешних ключей, которые обеспечивают связь между таблицами. Внешний ключ является связующим элементом, который ссылается на первичный ключ другой таблицы и позволяет объединять данные из разных таблиц в один запрос.

Оптимизация работы с базами данных основывается на использовании индексов. Индексы позволяют ускорить поиск данных в таблице. Учитывая большой объем данных, применение индексов позволяет значительно ускорить процесс запроса значений.

Основным преимуществом реляционных баз данных является их интуитивная понятность и удобство использования, так как они основаны на простых

концепциях, что упрощает группировку, фильтрацию и агрегацию данных.

1.3.4 Нереляционная модель

Данные нереляционных баз данных не имеют общего формата, в них для хранения используется не система из строк и столбцов, а применяется модель, которая оптимизирована для хранения определённого типа содержимого. Например, данные могут храниться в виде документов JSON [3], графов, а также пар ключ-значение.

Преимущества модели - возможность создания сложных структур данных, удобство при работе с объектами. Недостатки - сложность реализации, неудобство при работе с большим количеством данных.

1.3.5 Выбор модели базы данных для решения задачи

Для решения задачи будет использоваться реляционная модель данных по описанным причинам:

- известна структура данных, которая редко меняется;
- структура данных не является сложной(с вложенностью, древовидной);
- наличие стандартизированного высокоуровневого языка запросов SQL.

Для хранения сессий пользователей следует использовать нереляционную модель, так как у сессий нет отношений и связей, они хранятся парами ”key-value” .

В качестве решения задачи хранения сессий зачастую используются in-memory СУБД.

In-Memory СУБД – это система, которая хранит данные в оперативной памяти компьютера. Это позволяет обеспечить высокую скорость доступа к данным и выполнения запросов, поскольку не требуется считывание и запись данных на диск.

Такие СУБД относятся к нереляционным.

1.4 Формализация данных

База данных пользователей, записей времени, проектов, тегов, целей и отношений между ними

Данные пользователей, записей времени, проектов, тегов, целей и отношений между ними должны храниться в одной базе данных, так как данные связаны между собой.

На рисунке 2 представлена ER-диаграмма в нотации Чена.

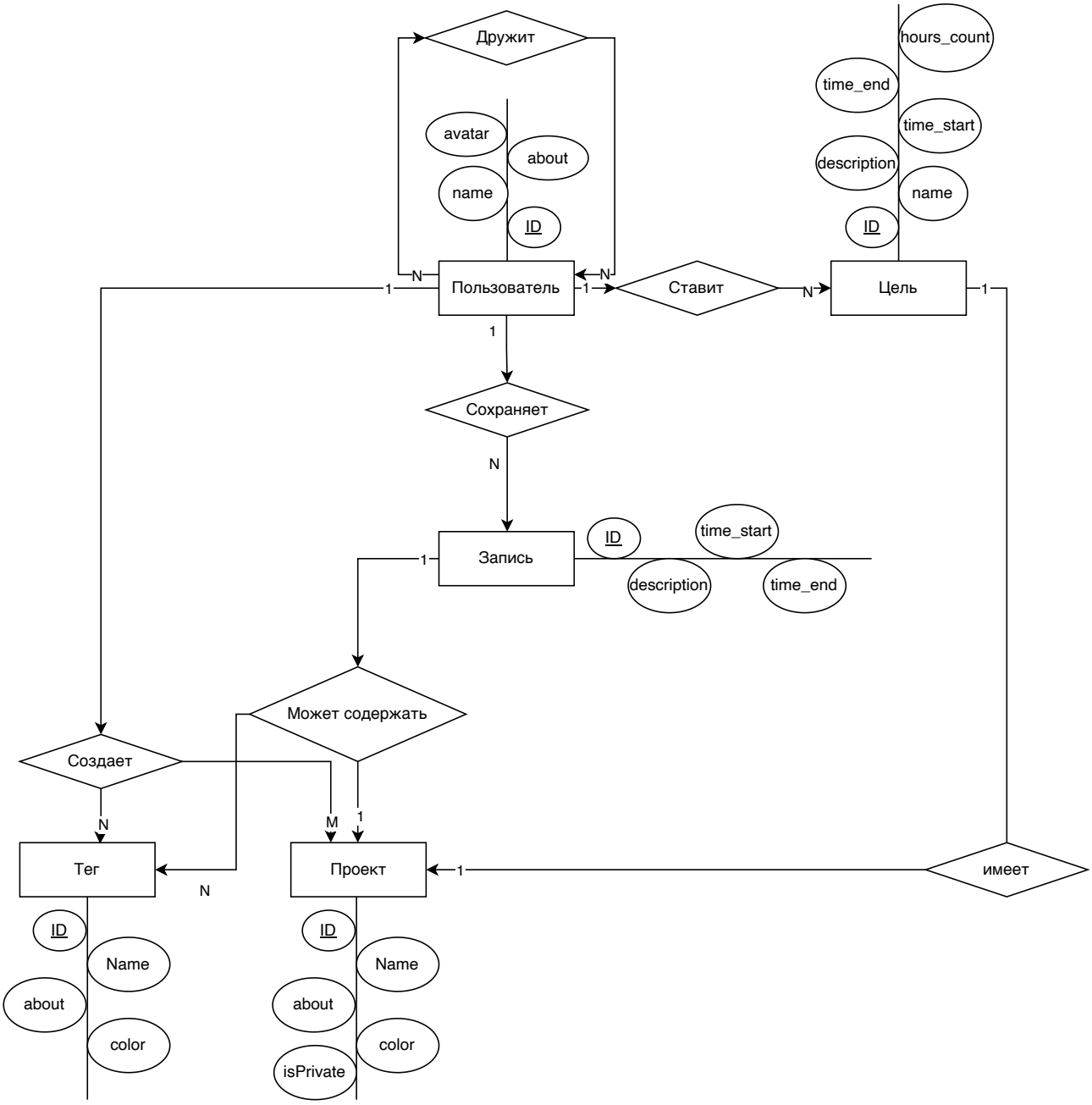

Рисунок 2 – ER-диаграмма сущностей

База данных сессий

Современные веб-приложения работают на основе HTTP протокола, который не хранит свое состояние, то есть каждый запрос обрабатывается как отдельная транзакция. Чтобы избежать необходимости повторной аутентификации пользователя при каждом запросе, необходимо самостоятельно отслеживать взаимодействие клиентов с приложением. Для этого используются сессии[5], которые хранят ”состояние” между сайтом и клиентом.

Для эффективного хранения данных сессий, необходимо использовать специальную базу данных, в которой хранятся уникальные идентификаторы пользователей и сессий. Также необходимо обеспечить быстрое получение данных и защиту от сбоев системы, например, сохранять данные на диск с возможностью восстановления.

Вывод

В данном разделе:

- формализована задача;
- определены и описаны роли доступа к данным приложения;
- определены типы данных приложения;
- приведена диаграмма использования приложения;
- проанализированы и выбраны модели данных;
- формализованны данные, используемые в системе.

2 Конструкторская часть

2.1 Проектирование базы данных

Диаграмма разрабатываемой базы данных представлена на рисунке 3.

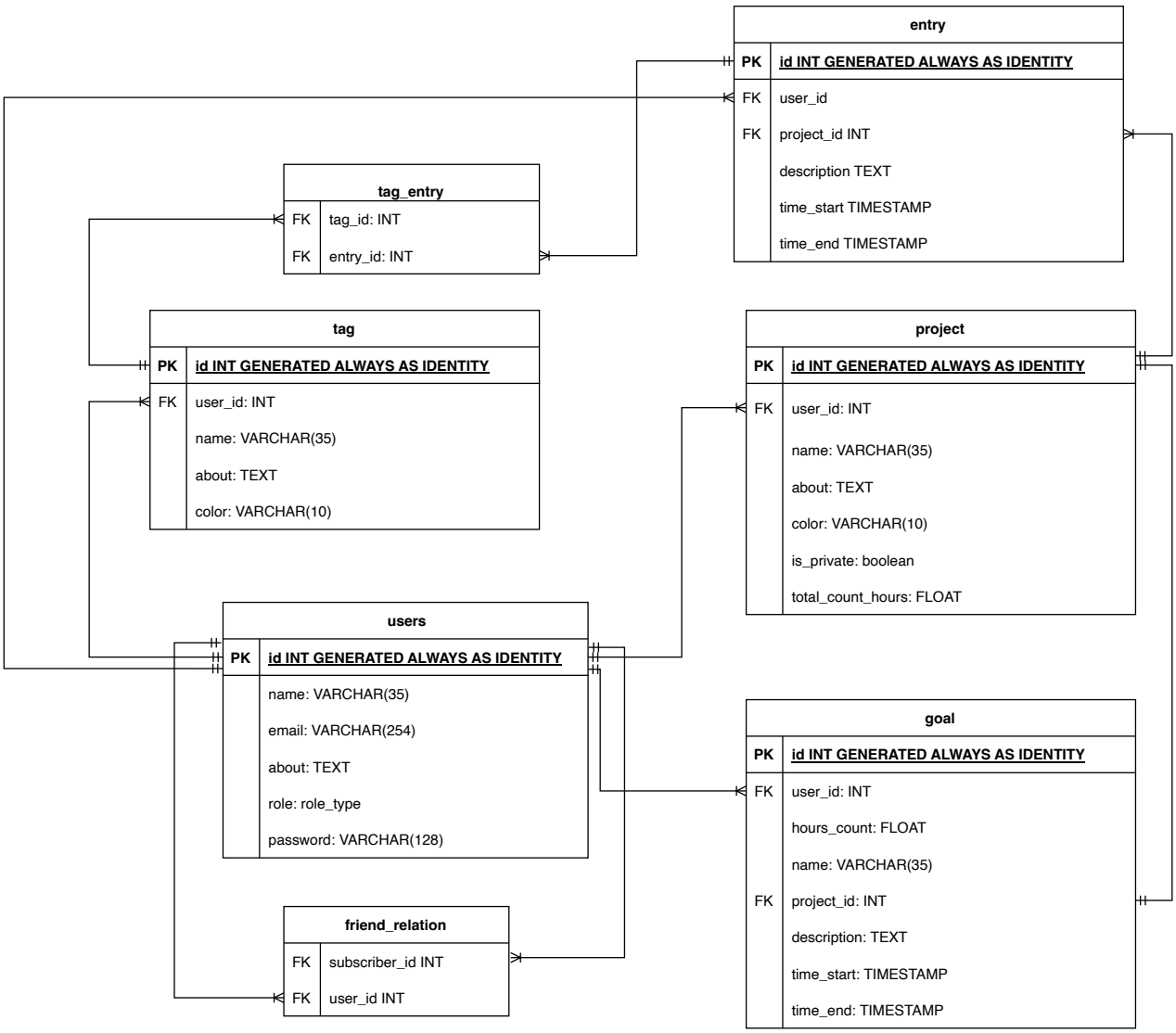

Рисунок 3 – ER-диаграмма базы данных

Таблицы

База данных должна хранить рассмотренный на рисунке 3 данные. Для этого можно выделить следующие таблицы:

- таблица пользователей users;
- таблица проектов project;
- таблица записей времени entry;
- таблица меток tag;
- таблица целей goal.

Users

Таблица users хранит информацию о пользователях.

Таблица 2 – Описание полей таблицы `users`

Поле	**Описание**
`id`	Идентификатор пользователя, уникальный
`name`	Имя пользователя в системе
`email`	Адрес электронно почты пользователя
`about`	Описание профиля
`last_name`	Фамилия пользователя
`password`	Хеш пароля пользователя

Project

Таблица project хранит информацию о проектах.

Таблица 3 – Описание полей таблицы `project`

Поле	**Описание**
`id`	Идентификатор проекта, уникальный
`user_id`	Идентификатор пользователя, создавшего проект
`name`	Название проекта
`about`	Описание проекта
`color`	Цвет проекта в интерфейсе
`is_private`	Признак приватности проекта
`total_count_hours`	Количество часов, потраченных на проект

Goal

Таблица goal хранит информацию о целях.

Таблица 4 – Описание полей таблицы `goal`

Поле	**Описание**
`id`	Идентификатор цели, уникальный
`user_id`	Идентификатор пользователя, создавшего цель
`project_id`	Идентификатор проекта, к которому прикреплена цель
`hours_count`	Количество часов, которые нужно потратить на проект
`name`	Название цели
`description`	Описание цели
`time_start`	Дата начала выполнения цели
`time_end`	Дата, окончания цели

Entry

Таблица entry хранит информацию о записях времени.

Таблица 5 – Описание полей таблицы `entry`

Поле	**Описание**
`id`	Идентификатор записи, уникальный
`user_id`	Идентификатор пользователя, создавшего запись
`project_id`	Идентификатор проекта, к которому прикреплена запись, опциональный
`description`	Описание записи
`time_start`	Дата начала интервала записи
`time_end`	Дата конца интервала

Tag

Таблица tag хранит информацию о метках пользователя.

Таблица 6 – Описание полей таблицы `tag`

Поле	**Описание**
`id`	Идентификатор метки, уникальный
`user_id`	Идентификатор пользователя, создавшего метку
`name`	Название метки
`about`	Описание метки
`color`	Цвет метки

2.2 Проектирование базы данных сессий

Для сохранения ”состояние” между сервером и клиентами требуется хранить данные, которые идентифицируют пользователя. В разделе 1.4 было опи-

сано, что для этого используются In-Memory СУБД, такие базы зачастую не имеют структуры и хранят значения парами ”key-value”. Таким образом, для хранения пользовательских сессий требуются:

- *session_token(key)* – уникальный идентификатор сессии;
- *user_id(value)* – идентификатор пользователя, которому принадлежит сессия.

2.3 Целостность данных

Для обеспечения целостности данных, а также для связывания таблиц необходимо, чтобы все строки в рамках одной таблицы имели уникальный идентификатор. В рассматриваемой базе даных, каждая таблица имеет свой первичный ключ - *id*

Внешние ключи

Для обеспечения целостности ссылочных данных необходимы внешние ключи, которые позволяют связывать данные между таблицами, обеспечивая целостность данных и поддерживая связи между ними.

Внещние ключи в рассматриваемых таблицах:

- в таблице `project` поле `user_id` ссылается на поле `id` таблицы `users`;
- в таблице `goal` поле `user_id` ссылается на поле `id` таблицы `users`, поле `project_id` ссылается на поле `id` таблицы `project`;
- в таблице `entry` поле `user_id` ссылается на поле `id` таблицы `users`, поле `project_id` ссылается на поле `id` таблицы `project`;
- в таблице `tag` поле `user_id` ссылается на поле `id` таблицы `users`.

Связи

Для хранения отношений ”дружбы” используется таблица friend_relation. Если оба пользователя являются подписчиками друг друга, то они являются друзьями.

Так как у временных записей(entry) может быть несколько меток, для хранения связей запись-метка используется таблица tag_entry.

Таблица 7 – Описание полей таблицы `friend_relation`

Поле	**Описание**
`subscriber_id`	Идентификатор пользователя, отправившего заявку в друзья
`user_id`	Идентификатор пользователя, которому приходит заявка в друзья

Таблица 8 – Описание полей таблицы `tag_entry`

Поле	**Описание**
`tag_id`	Идентификатор метки
`entry_id`	Идентификатор записи, которой принадлежит метка

2.4 Триггеры

Для обновления поля total_count_hours таблица project, нужен механизм, который при каждой вставке, изменении или удалении строк в таблице entry будет пересчитываться поле total_count_hours той строки, на которую ссылается project_id.

Данный механизм можно реализовать с помощью треггера, который будет срабатывать при обновлении таблицы entry. После того, как в таблицу добавились, изменились или удалились строки, будет срабатывать функция, которая будет пересчитывать поле total_count_hours в таблице project.

На рисунке 4 представлена схема алгоритма функции триггера update_total_count_hours().

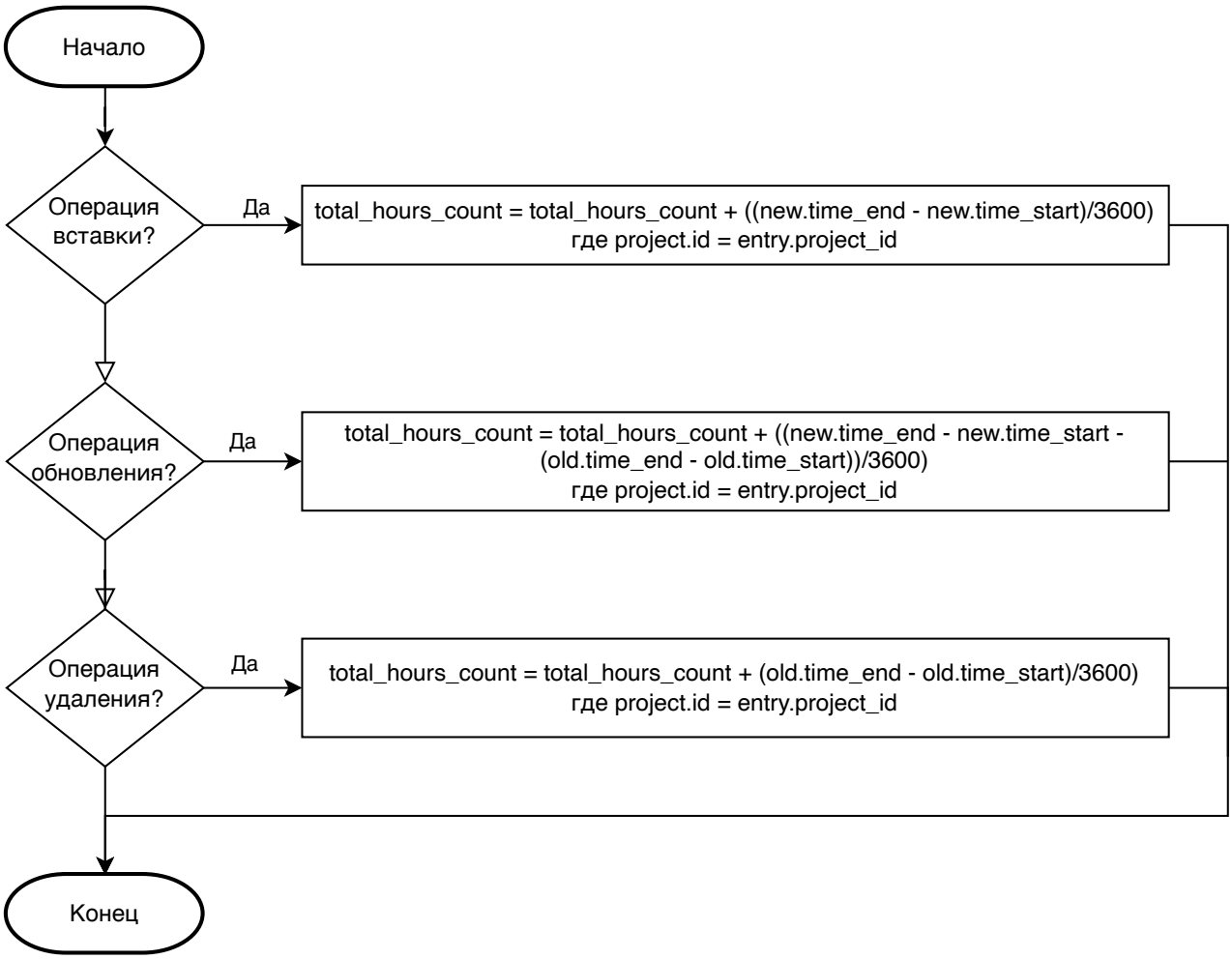

Рисунок 4 – Схема алгоритма функции триггера

2.5 Ролевая модель

База данных имеет три роли доступа к данным:

- администратор – доступны все операции над всеми таблицами, создание таблиц, ролей;
- разработчик – доступ ко всем таблицам для CRUD операций;
- гость – доступна операция SELECT на все таблицы, кроме users;

Вывод

В данном разделе:

- спроектированы сущности базы данных;
- спроектирована базы данных сессий;

- описаны ограничения целостности базы данных;
- описаны требуемые триггеры;
- описана ролевая модель на уровне базы данных.

3 Технологическая часть

3.1 Выбор СУБД

В данной работе в качестве основной СУБД была выбрана PostgreSQL [6], так как обладает следующими свойствами:

- надежность: PostgreSQL обеспечивает целостность данных и защиту от потери информации;
- гибкость: база данных поддерживает множество функция и широкий спектр языков программирования, предоставляя различные методы доступа к данным;
- поддерживает сложные структуры и широкий спектр встроенных и определяемых пользователем типов данных.

В качестве базы данных сессий был выбран Redis[7]. Это хранилище удовлетворяет всем необходимым требованиям(in-memory key-value хранилище). Кроме этого, Redis обладает простотой в использовании и интеграции с различными языками программирования.

3.2 Выбор средства реализации приложения

В данное работе приложением является Web-сервер, доступ к которому осуществляется с помощью REST API [2]. В качестве языка реализации сервера был выбран Golang [8]. Этот язык изначально был разработан для создания масштабируемых и высокопроизводительных систем, также он имеет множество различных библиотек для написания Web-приложений.

Для взаимодействия с базами данных будут использоваться встроенные в Golang драйвера.

Для реализации REST API был выбран фреймворк echo [9].

В качестве инструмента сборки и развертывания используется Docker[10], который позволяет упростить процесс интеграции приложения и базы данных. Docker-контейнеры могут быть запущены на любой платформе, которая поддерживает Docker, что упрощает перенос приложений между различными окру-

жениями.

3.3 Детали реализации

Система представляет собой Web-приложение, принимающее запросы от клиентов, обрабатывая и возвращая ответ.

Взаимодействие с серверов реализовано через REST API.

Сценарии создания объектов базы данных приведены в листингах А.1 – А.2.

Для создания ролевой модели и триггеров используются сценарий, приведенный в листингах Б.1 – Б.2.

Пример взаимодействия с СУБД PostgresSQL и Redis приведены в листингах Г.1 – Г.4 и в листингах Г.5 – Г.6 соответственно.

Описание REST API проектируемого приложения представлено в таблице 9.

Таблица 9 – Описание `REST API` реализуемого приложения

Путь	Метод	Описание
/signin	POST	Метод авторизации пользователя
/signup	POST	Метод для регистрации пользователя в системе
/logout	POST	Метод завершения сесссии пользователя
/entry{id}	GET	Метод для получения временной записи по идентификатору
/entry{id}	DELETE	Метод для удаления временной записи по идентификатору
/entry/create	POST	Метод для создания временной записи
/entry/edit	POST	Метод для редактирования временной записи
/me/entries	GET	Метод для получения временных записей аутентифицированного пользователя
/user/{user_id}/entries	Get	Метод для получения временных записей пользователя по его идентификатору
/goal{id}	GET	Метод для получения цели по идентификатору
/goal{id}	DELETE	Метод для удаления цели по идентификатору
/goal/create	POST	Метод для создания цели
/goal/edit	POST	Метод для редактирования цели

Продолжение на следующей странице

Таблица 9 – продолжение

Путь	Метод	Описание
/me/goals	GET	Метод для получения целей аутентифицированного пользователя
/user/{user_id}/goals	Get	Метод для получения целей пользователя по его идентификатору
/project{id}	GET	Метод для получения проекта по идентификатору
/project{id}	DELETE	Метод для удаления проекта по идентификатору
/project/create	POST	Метод для создания проекта
/project/edit	POST	Метод для редактирования проекта
/me/projects	GET	Метод для получения проектов аутентифицированного пользователя
/user/{user_id}/projects	Get	Метод для получения проектов пользователя по его идентификатору
/tag{id}	GET	Метод для получения метки по идентификатору
/tag{id}	DELETE	Метод для удаления метки по идентификатору
/tag/create	POST	Метод для создания метки
/tag/edit	POST	Метод для редактирования метки
/me/tags	GET	Метод для получения меток аутентифицированного пользователя
/user/{user_id}/tags	Get	Метод для получения меток пользователя по его идентификатору
Продолжение на следующей странице		

Таблица 9 – продолжение

Путь	Метод	Описание
/me	GET	Метод для получения информации об аутентифицированном пользователе
/me/edit	POST	Метод для редактирования информации об аутентифицированном пользователе
/users	GET	Метод для получения информации о всех пользователях
/user/{user_id}	Get	Метод для получения информации о пользователе по его идентификатору
/friends/subscribe/{user_id}	POST	Метод для подписки на пользователя по его идентификатору
/friends/subscribe/{user_id}	POST	Метод для отподписки на пользователя по его идентификатору
/me/friends	GET	Метод для получения всех друзей аутентифицированного пользователя
/me/subs	GET	Метод для получения всех подписчиков аутентифицированного пользователя
/user/{user_id}/subs	Get	Метод для получения всех подписчиков пользователя по его идентификатору
/user/{user_id}/friends	Get	Метод для получения всех друзей пользователя по его идентификатору
Продолжение на следующей странице		

Таблица 9 – продолжение

Путь	Метод	Описание
/admin/stat	Get	Метод для получения статистики приложения
		Конец таблицы

3.4 Тестирование

Для тестирования проекта были реализованы интеграционные тесты. В тестах проверялась интеграция приложения с базой данных. Для написания и запуска тестов использовался фреймворк языка Python pytest[11]. Он выбран в качестве основного инструмента для автоматизированного тестирования, поскольку обладает широким набором функций и простым синтаксисом для написания тестов. Кроме того, для тестирования взаимодействия с базой данных использовался фреймворк sqlalchemy ORM[12].

Конечные точки приложения можно разделить на 4 основных категории: создание, просмотр, редактирование и удаление, данных. Паттерн тестирования для каждой из категорий:

- создание – сначала происходит обращение к конечной точке на создание объекта с входными данными, затем проверяется наличие этого объекта в базе данных;
- просмотр – в базе данных создается объект, затем происходит обращение к конечной точке на получение этого объекта;
- редактирование – в базе данных создается объект, затем происходит обращение к конечной точке на изменение этого объекта, проверяется что базе данных объект изменился;
- удаление – в базе данных создается объект, затем происходит обращение к конечной точке на удаление этого объекта, проверяется что в базе данных этого объекта нет.

Фикстура (fixture) в тестовом фреймворке - это функция, которая может быть запущена перед тестом или группой тестов и предоставляет необходимые данные и/или ресурсы для тестов. В данной курсовой работе фикстуры использовались для управления окружением тестирования и подготовки данных для тестов. В частности, использовались две основные фикстуры: одна для определения базового URL(Uniform Resource Locator) для API, а другая для управле-

ния базой данных.

Фикстура api_url() возвращает базовый URL для API. Фикстура db_all_tables() отвечает за управление базой данных в рамках тестов. Эта фикстура используется для подготовки БД перед выполнением тестов и для очистки БД после их выполнения. Это позволяет выполнять тесты на ”чистой” базе данных и обеспечивает независимость тестов друг от друга.

Исходный код фикстур приведен в листинге Д.1, тестирование на примере сущности ”проект” в листингах Д.2 – Д.7.

Вывод

В данном разделе:

- определены основная СУБД и СУБД для хранения сессий;
- выбран тип приложения;
- определены средства реализации приложения;
- представлен интерфейс взаимодействия с приложением;
- приведены детали реализации разрабатываемого приложения и базы данных;
- описаны методы и технологии тестирования.

4 Исследовательская часть

4.1 Цель исследования

Целью исследования является сравнение производительности программного обеспечения с использованием кеширования в in-memory СУБД и без него.

Задачи, требуемые для достижения цели:

- реализовать кеширование ответов на уровне приложения;
- наполнить базу данных тестовыми данными;
- провести нагрузочное тестирование с кешированием и без кеширования с постоянным и линейным профилем нагрузки;
- сравнить полученные результаты и сформулировать вывод.

4.2 Описание исследования

Для оптимизации запросов в приложении было реализовано кеширование ответов. Система проверяет аргументы каждого запроса и ищет соответствующий ответ в кеше. Если ответа нет, то данные запрашиваются из базы данных, сохраняются в кеш и отдаются пользователю. Повторный запрос будет обработан быстрее, так как данные уже есть в кеше и запрос в базу данных не требуется.

В роли кеша будет использоваться in-memory СУБД redis. Для генерации нагрузки и сбора метрик будет использоваться Yandex.Tank[13].

Для анализа полученных метрик будет использоваться инструмент анализа производительности веб-приложений Overload[14].

Для тестирования была выбрана конечная точка $/user/\{user\_id\}/projects$ с параметром $user\_id$ равным идентификатору пользователя, этот параметр находится в интервале от 1 до 10.

Набор тестовых данных включал в себя 10 пользователей и 10000 проектов. Данные были сгенерированы с помощью скрипта на языке python [15] и библиотеки faker [16].

В листинге 1 представлен список выполняемых запросов для тестирования.

Листинг 1: Примеры запросов для нагрузочного тестирования

```
[Connection: close]
[Host: localhost]
[Cookie: session_token=a18271fd-1c25-4695-87a1-5a9fb749a72f]
/user/1/projects
/user/2/projects
/user/3/projects
/user/5/projects
/user/6/projects
/user/7/projects
/user/8/projects
/user/9/projects
/user/10/projects
```

Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, следующие.

- Операционная система: macOS 12.5.1.
- Память: 16 ГБ.
- Процессор: Intel® Core™ i7-1068NG7 CPU @ 2.30ГГц [17].

Исследование проводилось на ноутбуке, включенном в сеть электропитания. Во время экспериментов ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.3 Результаты исследования

Ниже представлены графики зависимости времени ответа конечной точки от количества запросов в секунду (rps) при линейной и постоянной нагрузке.

Графики включают в себя три метрики: rps, avg, и 98-й квантиль времени ответа, где rps - это количество запросов в секунду направленных на конечную точку, avg – среднее время ответа конечной точки.

Линейная нагрузка

График времени ответа конечной точки при линейной нагрузке без использования кеширования представлен на рисунке 5.

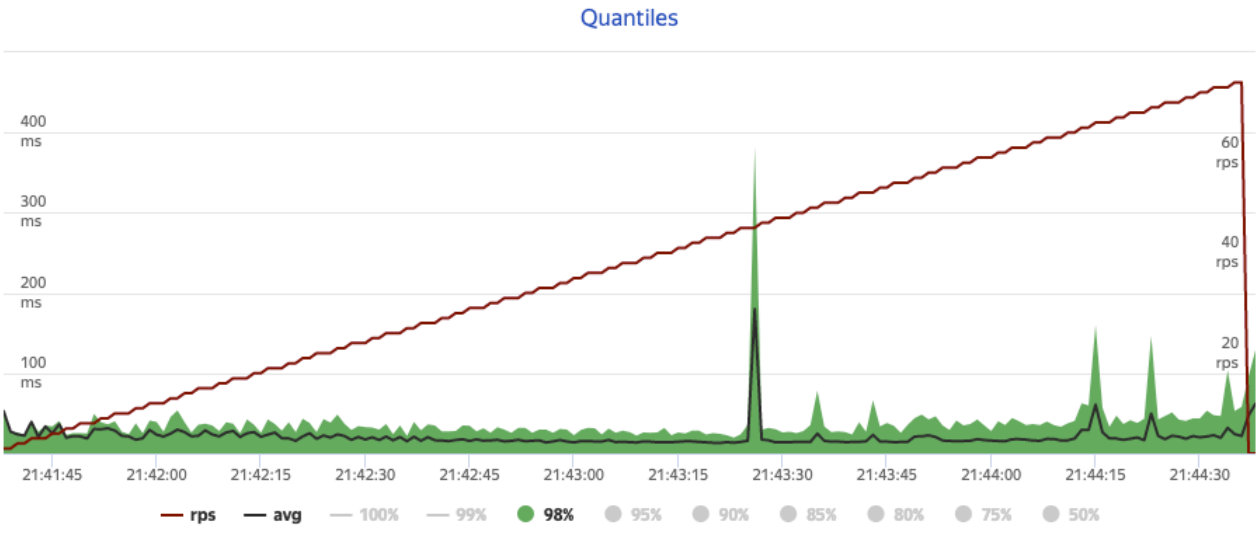

Рисунок 5 – График времени ответа конечной точки при линейной нагрузке без использования кеширования

График времени ответа конечной точки при линейной нагрузке с использованием кеширования представлен на рисунке 6.

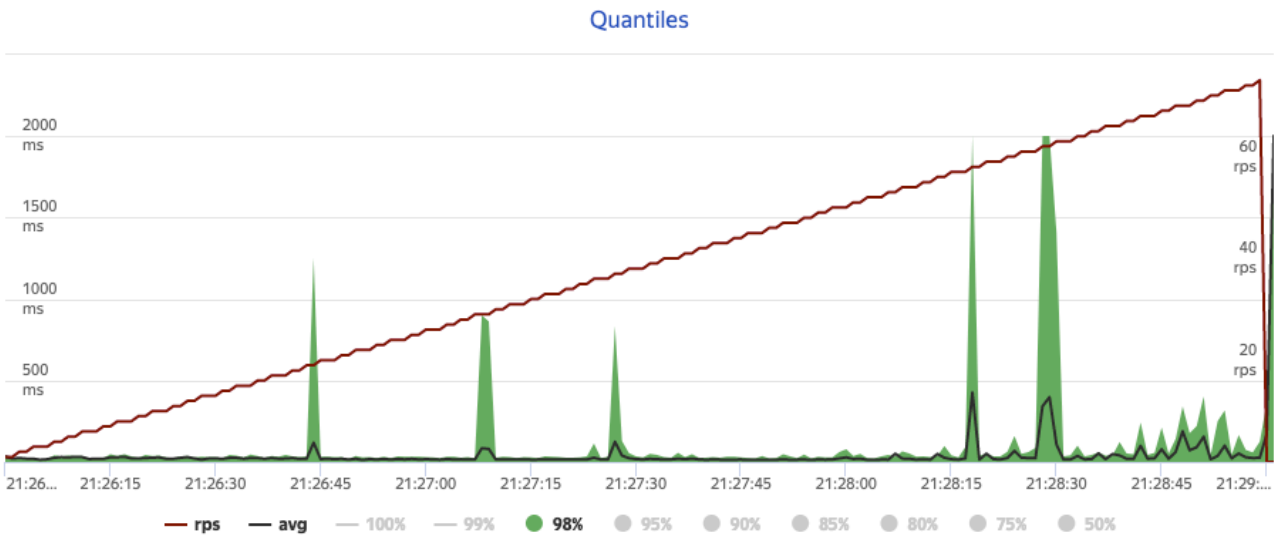

Рисунок 6 – График времени ответа конечной точки при линейной нагрузке с использованием кеширования

Исследование с помощью линейной нагрузки показало, что кеширование не приводит к росту производительности, наоборот, появляются частые ”скачки” времени ответа.

Постоянная нагрузка

График времени ответа конечной точки при постоянной нагрузке без использования кеширования представлен на рисунке 7.

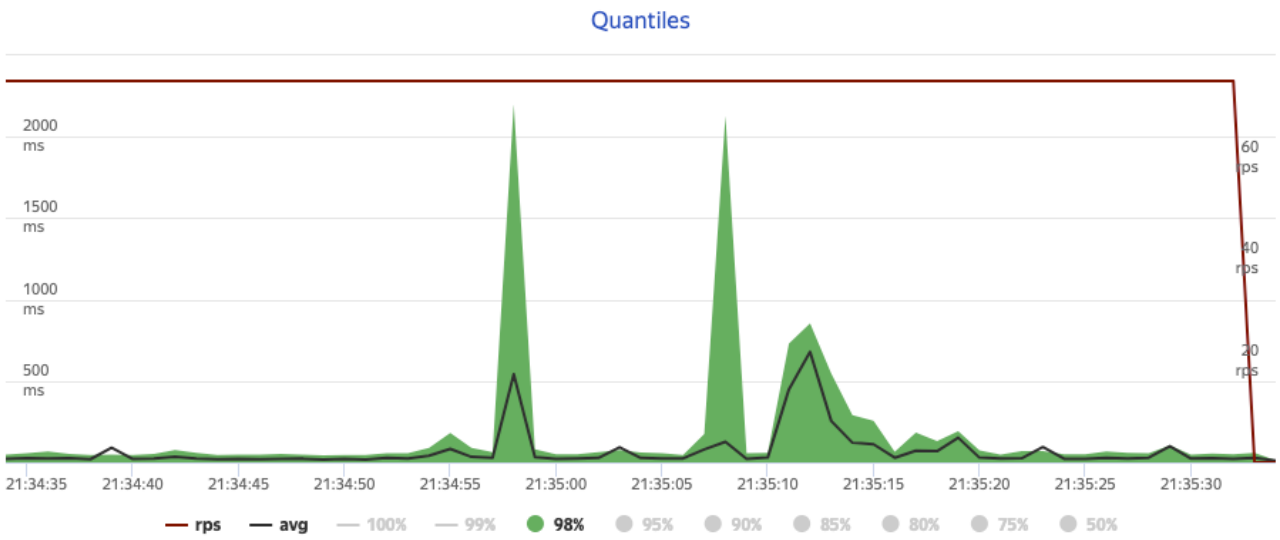

Рисунок 7 – График времени ответа конечной точки при постоянной нагрузке без использования кеширования

График времени ответа конечной точки при постоянной нагрузке с использованием кеширования представлен на рисунке 7.

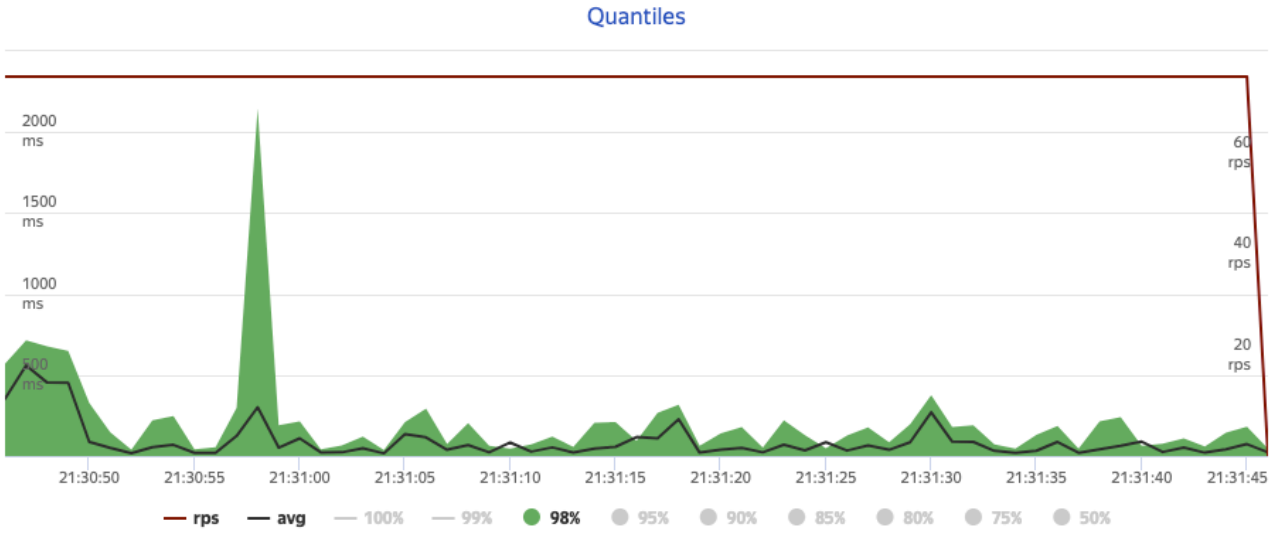

Рисунок 8 – График времени ответа конечной точки при постоянной нагрузке с использованием кеширования

Исследование с помощью постоянной нагрузки(75 запросов в секунду) показало, что кеширование работает быстрее с момента, когда все данные попали в кеш. Общий 99-й квантиль времени ответа без кеширования – 760 мс. С кешированием – 660 мс. Также на графиках видно, что ”скачков” времени

ответа без использования кеширования больше, чем с его использованием.

Вывод

В данном разделе было проведено исследование, в ходе которого было реализовано кеширование ответов приложения конечной точки $/user/\{user\_id\}/projects$.

Исследование показало, что при линейной нагрузке кеширование приводит к частым ”скачкам” времени ответа сервиса, а при постоянной нагрузке(75 запросов в секунду) наоборот – к уменьшению времени ответа на 15%.

Использование кеширования в системе может повысить производительность конечной точки при постоянной нагрузке. Однако, кеширование должно быть применено только к конечным точкам, которые имеют высокий коэффициент попадания запросов в кеш. В противном случае, применение кеширования не приведет к повышению производительности, поскольку система будет сначала проверять наличие необходимых данных в кеше, а только потом - в базе данных.

ЗАКЛЮЧЕНИЕ

При выполнении курсовой работы была достигнута цель – спроектирована и реализована база данных, содержащая данные учёта затраченного времени пользователей, разработан программный интерфейс приложения, который позволяет работать с этой базой данных.

В ходе выполнения поставленной цели были выполнены следующие задачи:

- проанализированы варианты модели данных и выбран подходящий вариант для решения задачи;
- проанализированы существующие СУБД и выбраны удовлетворяющие требованиям к хранению данных;
- спроектирована база данных, описаны ее сущности и связи;
- реализовано программное обеспечение, предоставляющее интерфейс доступа к данным.

В процессе написания курсовой работы были получены знания в области проектирования баз данных, разработки Web-приложений, кеширования данных, нагрузочного тестирования.

В рамках выполнения данной курсовой работы было разработано программное обеспечение, предоставляющее интерфейс для работы с базой данных. Для улучшения показателей производительности был реализован механизм кеширования ответов.

Исследование показало, что кеширование при постоянной нагрузке(75 запросов в секунду) уменьшает время ответа сервиса на 15%, в сравнении с вариантом без кеширования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Дейт К. Дж. Введение в системы баз данных. — 8-е изд. — М.: «Вильямс», 2006.

2. Введение в Rest API [Электронный ресурс]. – Режим доступа: https://mcs.mail.ru/blog/vvedenie-v-rest-api свободный – (15.04.2023)

3. Json [Электронный ресурс]. – Режим доступа: https://www.json.org/json-ru.html, свободный – (15.04.2023)

4. Structured Query Language (SQL) [Электронный ресурс]. – Режим доступа: https://www.techtarget.com/searchdatamanagement/definition/SQL, свободный – (15.04.2023)

5. What is a Web Session? [Электронный ресурс]. – Режим доступа: https://redisson.org/glossary/web-session.html, свободный – (15.04.2023)

6. PostgreSQL. [Электронный ресурс]. – Режим доступа: https://www.postgresql.org/, свободный – (21.05.2023)

7. Redis. [Электронный ресурс]. – Режим доступа: https://redis.io/, свободный – (21.05.2023)

8. Golang. [Электронный ресурс]. – Режим доступа: https://go.dev/, свободный – (21.05.2023)

9. Echo. High performance, extensible, minimalist Go web framework. [Электронный ресурс]. – Режим доступа: https://echo.labstack.com/, свободный – (21.05.2023)

10. Docker. [Электронный ресурс]. – Режим доступа: https://www.docker.com/, свободный – (21.05.2023)

11. Pytest. [Электронный ресурс]. – Режим доступа: https://docs.pytest.org/en/7.3.x/contents.html, свободный – (27.05.2023)

12. SQLAlchemy - The Database Toolkit for Python. [Электронный ресурс]. – Режим доступа: https://www.sqlalchemy.org, свободный – (27.05.2023)

13. Yandex Tank. [Электронный ресурс]. – Режим доступа: https://yandextank.readthedocs.io/en/latest/, свободный – (27.05.2023)

14. Overload a performance analytics service. [Электронный ресурс]. – Режим доступа: https://overload.yandex.net/login/?next=/, свободный – (27.05.2023)

15. Python. [Электронный ресурс]. – Режим доступа: https://www.python.org/, свободный – (27.05.2023)

16. Faker. [Электронный ресурс]. – Режим доступа: https://faker.readthedocs.io/en/master/, свободный – (27.05.2023)

17. Intel. [Электронный ресурс]. – Режим доступа: https://www.intel.com/content/www/us/en/support/ru-banner-inside.html, свободный – (27.05.2023)

ПРИЛОЖЕНИЕ А

Скрипты создания объектов БД

В листингах А.1 – А.2 представлены скрипты создания объектов БД.

Листинг А.1: Скрипт создания объектов БД. Часть 1.

```
CREATE TYPE role_type AS ENUM ('user', 'admin');
CREATE TABLE IF NOT EXISTS users (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    name VARCHAR(35) NOT NULL,
    email VARCHAR(254) NOT NULL UNIQUE,
    about TEXT DEFAULT '',
    role role_type DEFAULT 'user',
    password VARCHAR(128) NOT NULL
);

CREATE TABLE IF NOT EXISTS tag (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    user_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    name VARCHAR(35) NOT NULL,
    about TEXT DEFAULT '',
    color VARCHAR(10) NOT NULL
);

CREATE TABLE IF NOT EXISTS project (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    user_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    name VARCHAR(35) NOT NULL,
    about TEXT DEFAULT '',
    color VARCHAR(10) NOT NULL,
    is_private boolean NOT NULL,
    total_count_hours FLOAT DEFAULT 0
);
```

Листинг А.2: Скрипт создания объектов БД. Часть 2.

```
CREATE TABLE IF NOT EXISTS goal (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    user_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    hours_count FLOAT NOT NULL,
    name VARCHAR(35) NOT NULL,
    project_id INT NOT NULL REFERENCES project(id) ON DELETE CASCADE,
    description TEXT DEFAULT '',
    time_start TIMESTAMP NOT NULL,
    time_end TIMESTAMP NOT NULL
);

CREATE TABLE IF NOT EXISTS entry (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    user_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    project_id INT REFERENCES project(id) ON DELETE CASCADE,
    description TEXT DEFAULT '',
    time_start TIMESTAMP NOT NULL,
    time_end TIMESTAMP NOT NULL
);

CREATE TABLE IF NOT EXISTS tag_entry (
    tag_id INT NOT NULL REFERENCES tag(id) ON DELETE CASCADE,
    entry_id INT NOT NULL REFERENCES entry(id) ON DELETE CASCADE,
    PRIMARY KEY (tag_id, entry_id)
);

CREATE TABLE IF NOT EXISTS friend_relation (
    subscriber_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    user_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    PRIMARY KEY (subscriber_id, user_id)
);
```

ПРИЛОЖЕНИЕ Б

Скрипты создания ролевой модели БД

В листингах Б.1 – Б.2 представлен скрипт создания ролевой модели БД.

Листинг Б.1: Скрипт создания ролевой модели БД. Часть 1.

```
-- Администратор
CREATE ROLE administrator;
GRANT ALL PRIVILEGES ON SCHEMA public TO administrator;
GRANT ALL PRIVILEGES ON DATABASE postgres TO administrator;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO administrator;
CREATE USER admin with
    CREATEDB
    CREATEROLE
    ENCRYPTED PASSWORD 'admin'
    IN ROLE administrator LOGIN;

-- Разработчик
CREATE ROLE developer;
GRANT SELECT, INSERT, UPDATE, DELETE
    ON users, project, entry, tag, tag_entry, goal, friend_relation
    TO developer;
GRANT USAGE ON SCHEMA public TO developer;

CREATE USER dev
    WITH ENCRYPTED PASSWORD 'dev'
    IN ROLE developer LOGIN;

-- Гость
CREATE ROLE guest;
```

Листинг Б.2: Скрипт создания ролевой модели БД. Часть 2.

```
25  GRANT SELECT
26      ON project, entry, tag, tag_entry, goal, friend_relation
27      TO guest;
28  GRANT USAGE ON SCHEMA public TO guest;
29
30  CREATE USER visitor
31      WITH ENCRYPTED PASSWORD 'guest'
32      IN ROLE guest LOGIN;
```

ПРИЛОЖЕНИЕ В

Скрипты создания триггера

В листинге В.1 представлен скрипт создания триггера для обновления количества часов, затраченных на проект.

Листинг В.1: Скрипт создания триггера для обновления количества затраченных на проект часов.

```
CREATE OR REPLACE FUNCTION update_total_count_hours()
RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'INSERT') THEN
        UPDATE project
        SET total_count_hours = total_count_hours +
        (EXTRACT(EPOCH FROM (NEW.time_end - NEW.time_start))) / 3600
        WHERE id = NEW.project_id;
    ELSIF (TG_OP = 'UPDATE') THEN
        UPDATE project
        SET total_count_hours = total_count_hours +
        (EXTRACT(EPOCH FROM (NEW.time_end - NEW.time_start))
        - EXTRACT(EPOCH FROM (OLD.time_end - OLD.time_start))) / 3600
        WHERE id = NEW.project_id;
    ELSIF (TG_OP = 'DELETE') THEN
        UPDATE project
        SET total_count_hours = total_count_hours
        - (EXTRACT(EPOCH FROM (OLD.time_end - OLD.time_start))) / 3600
        WHERE id = OLD.project_id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_total_count_hours_trigger
AFTER INSERT OR UPDATE OR DELETE
    ON entry FOR EACH ROW EXECUTE FUNCTION update_total_count_hours();
```

ПРИЛОЖЕНИЕ Г

Паттерны взаимодействия с PostgresSQL и Redis

Листинг Г.1: Взаимодействие с PostgreSQL. Часть 1

```
package postgres

import (
    "fmt"
    "timetracker/internal/User/repository"
    "timetracker/models"

    "github.com/pkg/errors"
    "gorm.io/gorm"
)

type User struct {
    ID       uint64 `gorm:"column:id"`
    Name     string `gorm:"column:name"`
    Email    string `gorm:"column:email"`
    About    string `gorm:"column:about"`
    Role     string `gorm:"column:role"`
    Password string `gorm:"column:password"`
}

func (User) TableName() string {
    return "users"
}

func toPostgresUser(u *models.User) *User {
    return &User{
        ID:       u.ID,
        Name:     u.Name,
        Email:    u.Email,
        About:    u.About,
        Role:     u.Role,
```

Листинг Г.2: Взаимодействие с PostgreSQL. Часть 2

```
32              Password: u.Password,
33          }
34      }
35
36      func toModelUser(u *User) *models.User {
37          return &models.User{
38              ID:       u.ID,
39              Name:     u.Name,
40              Email:    u.Email,
41              About:    u.About,
42              Role:     u.Role,
43              Password: u.Password,
44          }
45      }
46
47      func toModelUsers(entries []*User) []*models.User {
48          out := make([]*models.User, len(entries))
49
50          for i, b := range entries {
51              out[i] = toModelUser(b)
52          }
53
54          return out
55      }
56
57      type userRepository struct {
58          db *gorm.DB
59      }
60
61      func (ur userRepository) CreateUser(user *models.User) error {
62          postgresUser := toPostgresUser(user)
63
64          tx := ur.db.Create(postgresUser)
65
66          if tx.Error != nil {
67              return errors.Wrap(tx.Error, "database error (table user)")
```

Листинг Г.3: Взаимодействие с PostgreSQL. Часть 3

```
68          }
69
70          user.ID = postgresUser.ID
71          return nil
72      }
73
74      func (ur userRepository) UpdateUser(user *models.User) error {
75          postgresUser := toPostgresUser(user)
76
77          tx := ur.db.Omit("id").Updates(postgresUser)
78
79          if tx.Error != nil {
80              return errors.Wrap(tx.Error, "database error (table user)")
81          }
82
83          return nil
84      }
85
86      func (ur userRepository) GetUser(id uint64) (*models.User, error) {
87          var user User
88
89          tx := ur.db.Where(&User{ID: id}).Take(&user)
90
91          if errors.Is(tx.Error, gorm.ErrRecordNotFound) {
92              return nil, models.ErrNotFound
93          } else if tx.Error != nil {
94              return nil, errors.Wrap(tx.Error, "database error (table user)")
95          }
96
97          return toModelUser(&user), nil
98      }
99
```

Листинг Г.4: Взаимодействие с PostgreSQL. Часть 4

```
100 func (ur userRepository) GetUserByEmail(email string)
101     (*models.User, error) {
102     var user User
103     fmt.Println("email", email)
104 
105     tx := ur.db.Where(&User{Email: email}).Take(&user)
106 
107     if errors.Is(tx.Error, gorm.ErrRecordNotFound) {
108         return nil, models.ErrNotFound
109     } else if tx.Error != nil {
110         return nil, errors.Wrap(tx.Error, "database error (table user)")
111     }
112 
113     return toModelUser(&user), nil
114 }
115 
116 func (ur userRepository) GetUsers() ([]*models.User, error) {
117     users := make([]*User, 0, 10)
118     tx := ur.db.Omit("password").Find(&users)
119 
120     if tx.Error != nil {
121         return nil, errors.Wrap(tx.Error, "database error (table users)")
122     }
123 
124     return toModelUsers(users), nil
125 }
126 
127 func (ur userRepository) GetUsersByIDs(userIDs []uint64)
128         ([]*models.User, error) {
129     users := make([]*User, 0, 10)
130     tx := ur.db.Omit("password").Find(&users, userIDs)
131 
132     if tx.Error != nil {
133         return nil, errors.Wrap(tx.Error, "database error (table users)")
134     }
```

Листинг Г.5: Взаимодействие с Redis. Часть 1.

```
package redis

import (
    "context"
    "github.com/pkg/errors"
    "github.com/redis/go-redis/v9"
    "timetracker/internal/Auth/repository"
    "timetracker/models"
)

type authRepository struct {
    db  *redis.Client
    ctx context.Context
}

func (ar authRepository) CreateCookie(cookie *models.Cookie) error {
    err := ar.db.Set(ar.ctx,
                    cookie.SessionToken,
                    cookie.UserID, cookie.MaxAge).Err()

    if err != nil {
        return errors.Wrap(err, "redis error")
    }

    return nil
}

func (ar authRepository) GetUserByCookie(value string) (string, error) {
    userIdStr, err := ar.db.Get(ar.ctx, value).Result()

    if errors.Is(err, redis.Nil) {
        return "", models.ErrNotFound
    } else if err != nil {
        return "", errors.Wrap(err, "redis error")
    }

```

Листинг Г.6: Взаимодействие с Redis. Часть 2.

```
37          return userIdStr, nil
38      }
39
40      func (ar authRepository) DeleteCookie(value string) error {
41          err := ar.db.Del(ar.ctx, value).Err()
42
43          if err != nil {
44              return errors.Wrap(err, "redis error")
45          }
46
47          return nil
48      }
49
50      func NewAuthRepository(db *redis.Client) repository.RepositoryI {
51          return &authRepository{
52              db:  db,
53              ctx: context.Background(),
54          }
55      }
```

ПРИЛОЖЕНИЕ Д

Тестирование

Листинг Д.1: Фикстуры.

```
@pytest.fixture(scope='module')
def api_url():
    return 'http://localhost:8080'

@pytest.fixture(autouse=True)
def db_all_tables():
    # setup
    engine = create_engine(
        'postgresql://test:test@localhost:13081/postgres',
        isolation_level="READ UNCOMMITTED")
    connection = engine.connect()
    session = Session(bind=connection)
    session.execute(text("truncate table users cascade;"))
    session.execute(text("truncate table project cascade;"))
    session.execute(text("truncate table tag cascade;"))
    session.execute(text("truncate table friend_relation cascade;"))
    session.execute(text("truncate table tag_entry cascade;"))
    session.execute(text("truncate table goal cascade;"))
    session.execute(text("truncate table entry cascade;"))
    session.commit()

    # return session
    yield session

    # tear down
    session.execute(text("truncate table users cascade;"))
    session.execute(text("truncate table project cascade;"))
    session.execute(text("truncate table tag cascade;"))
    session.execute(text("truncate table friend_relation cascade;"))
    session.execute(text("truncate table tag_entry cascade;"))
    session.execute(text("truncate table goal cascade;"))
    session.execute(text("truncate table entry cascade;"))
    session.commit()
```

Листинг Д.2: Тестирование на примере сущности ”проект ” . Часть 1.

```
import pytest
from confest import db_all_tables, api_url, clean_users, auth_user
import requests
import classes
from utils import compare_alch_with_dict, \
    compare_json, compare_json_arr, objects_to_dicts
import time

elem_url = "project"

def test_create(db_all_tables, api_url, auth_user):
    input_data = {
        "about": "string",
        "color": "string",
        "is_private": False,
        "name": "string"
    }

    headers = {"Cookie": auth_user["Cookie"]}
    response = requests.post(f"{api_url}/{elem_url}/create",
                             json=input_data, headers=headers)
    project = db_all_tables.query(classes.Project).first()

    excepted = input_data

    assert response.status_code == 200
    assert compare_alch_with_dict(project, excepted)

def test_edit(db_all_tables, api_url, auth_user):
    user_id = auth_user["id"]
    project_db_json = {
        "about": "string",
        "color": "string",
        "is_private": False,
        "user_id": user_id,
        "name": "string"
```

Листинг Д.3: Тестирование на примере сущности ”проект ” . Часть 2.

```
    }

    project = classes.Project(**project_db_json)
    db_all_tables.add(project)
    db_all_tables.commit()


    input_data = {
        "id": project.id,
        "about": "string",
        "color": "string",
        "is_private": False,
        "name": "new_name"
    }

    headers = {"Cookie": auth_user["Cookie"]}
    response = requests.post(f"{api_url}/{elem_url}/edit",
                             json=input_data, headers=headers)
    db_all_tables.refresh(project)
    assert response.status_code == 200

    project_db_new = db_all_tables.query(classes.Project).first()
    excepted = input_data

    assert compare_alch_with_dict(project_db_new, excepted)

@pytest.mark.negative
def test_edit_not_found(db_all_tables, api_url, auth_user):
    user_id = auth_user["id"]
    input_data = {
        "id": 1,
        "about": "string",
        "color": "string",
        "is_private": False,
        "name": "new_name"
```

Листинг Д.4: Тестирование на примере сущности ”проект ” . Часть 3.

```
72          }
73
74          headers = {"Cookie": auth_user["Cookie"]}
75          response = requests.post(f"{api_url}/{elem_url}/edit",
76                                   json=input_data, headers=headers)
77          assert response.status_code == 404
78
79      def test_delete(db_all_tables, api_url, auth_user):
80          user_id = auth_user["id"]
81          project_db_json = {
82              "about": "string",
83              "color": "string",
84              "is_private": False,
85              "user_id": user_id,
86              "name": "string"
87          }
88
89          project = classes.Project(**project_db_json)
90          db_all_tables.add(project)
91          db_all_tables.commit()
92
93          headers = {"Cookie": auth_user["Cookie"]}
94          response = requests.delete(f"{api_url}/{elem_url}/{project.id}",
95                                     headers=headers)
96          assert response.status_code == 204
97
98          count = db_all_tables.query(classes.Project).count()
99          db_all_tables.commit()
100         assert count == 0
101
102     @pytest.mark.negative
103     def test_delete_not_found(db_all_tables, api_url, auth_user):
104         headers = {"Cookie": auth_user["Cookie"]}
105         response = requests.delete(f"{api_url}/{elem_url}/1", headers=headers)
```

Листинг Д.5: Тестирование на примере сущности ”проект ” . Часть 4.

```
106         assert response.status_code == 404
107
108     @pytest.mark.negative
109     def test_create_empty(db_all_tables, api_url, auth_user):
110         input_data = {}
111
112         headers = {"Cookie": auth_user["Cookie"]}
113         response = requests.post(f"{api_url}/{elem_url}/create",
114                                  json=input_data, headers=headers)
115
116         assert response.status_code != 200
117
118
119     def test_get_by_id(db_all_tables, api_url, auth_user):
120         user_id = auth_user["id"]
121
122         project_json = {
123             "user_id": user_id,
124             "name": 'New project',
125             "about": "Some description",
126             "color": 'blue',
127             "is_private": True,
128         }
129
130         project = classes.Project(**project_json)
131         db_all_tables.add(project)
132         db_all_tables.commit()
133
134         headers = {"Cookie": auth_user["Cookie"]}
135         response = requests.get(f"{api_url}/{elem_url}/{project.id}",
136                                 headers=headers)
137         assert response.status_code == 200
138
139         body = response.json()['body']
140         assert compare_json(body, project_json)
```

Листинг Д.6: Тестирование на примере сущности ”проект ” . Часть 5.

```
141
142     @pytest.mark.negative
143     def test_get_by_id_not_found(db_all_tables, api_url, auth_user):
144         headers = {"Cookie": auth_user["Cookie"]}
145         response = requests.get(f"{api_url}/{elem_url}/1",
146                                 headers=headers)
147         assert response.status_code == 404
148
149     def test_get_my(db_all_tables, api_url, auth_user):
150         user_id = auth_user["id"]
151
152         project_json_1 = {
153             "user_id": user_id,
154             "name": 'New project',
155             "about": "Some description",
156             "color": 'blue',
157             "is_private": True,
158         }
159
160         project_1 = classes.Project(**project_json_1)
161         db_all_tables.add(project_1)
162         db_all_tables.commit()
163
164         project_json_2 = {
165             "user_id": user_id,
166             "name": 'New project 2',
167             "about": "Some description",
168             "color": 'blue',
169             "is_private": True,
170         }
171
172         project_2 = classes.Project(**project_json_2)
173         db_all_tables.add(project_2)
174         db_all_tables.commit()
175
176         headers = {"Cookie": auth_user["Cookie"]}
```

Листинг Д.7: Тестирование на примере сущности ”проект ” . Часть 6.

```
177         response = requests.get(f"{api_url}/me/{elem_url}s",
178                                 headers=headers)
179         assert response.status_code == 200
180 
181         body = response.json()['body']
182 
183         excepted_arr = objects_to_dicts([project_1, project_2])
184         assert compare_json_arr(body, excepted_arr)
185 
186     def test_get_my_empty(db_all_tables, api_url, auth_user):
187         headers = {"Cookie": auth_user["Cookie"]}
188         response = requests.get(f"{api_url}/me/projects", headers=headers)
189         assert response.status_code == 200
190 
191         body = response.json()['body']
192 
193         excepted_arr = []
194         assert compare_json_arr(body, excepted_arr)
```