**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное автономное образовательное учреждение**

**высшего образования**

**«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА АНАЛИЗА ДАННЫХ И

ТЕХНОЛОГИЙ ПРОГРАММИРОВАНИЯ

Направление: 09.03.03 – Прикладная информатика

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

**СИСТЕМА КОНТРОЛЯ ВЕРСИЙ**

**Работа завершена:**

Студент 4 курса

группы 09-051

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.    \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Мишин С.С.

**Работа допущена к защите:**

Научный руководитель

ст. преподаватель

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.    \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сафина Л.И.

Заведующий кафедрой

к.ф.-м.н.

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.    \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Бандеров В.В.

Казань-2024

Содержание

[Глоссарий 3](#_Toc167128233)

[Введение 5](#_Toc167128234)

[1. Анализ предметной области 6](#_Toc167128235)

[1.1. Особенности предметной области 6](#_Toc167128236)

[1.2. Обзор существующих решений 8](#_Toc167128237)

[1.2.1. Git 8](#_Toc167128238)

[1.2.2. Mercurial 10](#_Toc167128239)

[1.3. Формирование технического задания 13](#_Toc167128240)

[1.3.1. Предмет разработки 13](#_Toc167128241)

[1.3.2. Архитектура приложения 15](#_Toc167128242)

[1.3.2.1 Клиент VCS 15](#_Toc167128243)

[1.3.2.2 Сервер VCS 16](#_Toc167128244)

[1.3.3. Функциональные требования к клиенту VCS 16](#_Toc167128245)

[1.3.4. Функциональные требования к серверу VCS 17](#_Toc167128246)

[1.3.5. Нефункциональные требования 18](#_Toc167128247)

[2. Проектирование системы контроля версий 19](#_Toc167128248)

[2.1. Общая архитектура 19](#_Toc167128249)

[2.2. Репозиторий 19](#_Toc167128250)

[2.3. База данных 20](#_Toc167128251)

[2.4. Выбор инструментов и средств разработки 21](#_Toc167128252)

[3. Разработка веб-приложения 23](#_Toc167128253)

[3.1. Реализация системы хранения данных 23](#_Toc167128254)

[3.2. Реализация пользовательского интерфейса 23](#_Toc167128255)

[3.3. Реализация функциональной части приложения 28](#_Toc167128256)

[4.1. Тестирование пользовательского интерфейса 29](#_Toc167128257)

[4.2. Тестирование функциональной части приложения 29](#_Toc167128258)

[Заключение 33](#_Toc167128259)

[Список использованных источников 34](#_Toc167128260)

[Приложение 1 36](#_Toc167128261)

[Приложение 2 36](#_Toc167128262)

# Глоссарий

Blob – (binary large object) – бинарный объект, представляющий содержимое файла.

Hash – уникальный идентификатор объекта. Вычисляется на основе содержимого объекта, и он служит для обеспечения целостности данных и проверки изменений объекта.

Hash-функция – это функция, которая принимает входные данные произвольной длины и преобразует их в некоторое фиксированное значение фиксированной длины, известное как hash-значение или hash-сумма. Это значение обычно представляется в виде последовательности чисел и/или букв, которые выглядят как случайная строка.

IDE – (integrated development environment) интегрированная среда разработки.

VCS – (version control system) система контроля версий.

.gitignore – файл, в котором описаны все файлы и папки, которые VCS должна игнорировать (не хранить их изменения).

Ветка (branch) – перемещаемый указатель на один из коммитов. Ветка указывает на последний коммит в своей цепочке коммитов, который называется «головным коммитом» (head commit). Новая ветка создает копию текущей ветки, в которой вы находитесь, и дает вам возможность работать над изменениями в изолированном пространстве без влияния на другие ветки.

Дерево – объект, который представляет собой папку или каталог файловой системы. Хранит имя каталога и ссылки на содержимое данного каталога.

Диффы изменений (diffs) – сравнения между различными версиями файла. Они показывают, какие конкретные строки или символы были добавлены, удалены или изменены между двумя версиями файла.

Коммит – snapshot (копия хранимых данных) проекта в определенный момент времени. Коммит хранит в себе ссылку на дерево проекта, автора коммита, время создания, комментарий и ссылку на предыдущий коммит.

Репозиторий – центральное хранилище для всех файлов, данных и истории изменений проекта. В репозитории хранятся все версии файлов, а также информация об изменениях, коммитах, ветках и тегах. Может быть как локальным (на устройстве клиента), так и удаленным (на удаленном от клиента сервере).

Файл – любой объект файловой системы (кроме каталогов). Хранит имя и ссылку на Blob.

# Введение

В современном мире разработка программного обеспечения становится все более сложной и коллаборативной задачей, требующей эффективного управления изменениями в исходном коде. Системы контроля версий (VCS) [3, 4] играют ключевую роль в этом процессе, предоставляя механизмы отслеживания, управления и совместной работы над версиями программного продукта.

Тема разработки новой системы контроля версий становится важной, учитывая постоянное развитие технологий и появление новых требований к процессам разработки. Перспективы улучшения эффективности совместной работы, безопасности данных и интеграции с современными средами разработки делают актуальной не только оптимизацию уже существующих VCS, но и создание новых, инновационных систем.

Цель: Создать систему контроля версий.

Задачи:

1. Изучить требования к системе контроля версий
2. Проанализировать существующие системы контроля версий
3. Составить техническое задание
4. Спроектировать части будущей системы контроля версий
5. Реализовать спроектированную систему
6. Протестировать систему.

# 1. Анализ предметной области

# 1.1. Особенности предметной области

Системы контроля версий являются инструментом, помогающим разработчикам отслеживать изменения в версиях своих файлов и организовывать совместную работу нескольких разработчиков над одним проектом. Предметная область, связанная с VCS задает определенные требования к данным системам.

Основными особенностями предметной области, которые следует учитывать при разработке VCS, являются:

1. Хранение данных файлов и папок (storing). Одной из основных проблем, с которой сталкиваешься при создании VCS – это принцип хранения данных и состояний.
2. Отслеживание изменений (tracking). Отслеживаются изменения файлов и папок, изменение их содержимого, имени, или расположения в файловой системе. Существуют различные способы сравнения содержимого файлов при помощи hash-функций и diff-алгоритмов для выявления отличий.
3. История изменений (history). Просмотр всех версий файлов и самого проекта. Это возможно при перемещении к нужному коммиту. Необходима функция для просмотра истории коммитов.
4. Откат (reverting). Возможность откатить изменения до предыдущего состояния. Если изменения в коде привели к проблемам или ошибкам, разработчики могут использовать откат, чтобы вернуться к предыдущему рабочему состоянию проекта.
5. Игнорирование определенных файлов (ignoring). Не все файлы проекта обычно нужно отслеживать. Некоторые файлы слишком большие и не изменяются (например, библиотеки, файлы IDE), некоторые могут содержать конфиденциальные данные (пароли, переменные среды и т.д.). Такие файлы можно игнорировать путем добавления их в .gitignore файл.
6. Работа в распределенном режиме (distributing). Это означает, что каждый клиент имеет полную копию репозитория. Это обеспечивает возможность работы независимо от сетевого подключения к центральному серверу.
7. Работа с удаленными репозиториями (remote repo). Возможность синхронизировать локальный репозиторий с удаленными репозиториями, такими как GitHub (<https://github.com/>), GitLab (<https://gitlab.com/>) или Bitbucket (<https://bitbucket.org/>). Это позволяет разработчикам делиться своим кодом, работать в команде и делать резервные копии своего кода.
8. Ветвление (branching). Ветвление позволяет разработчикам работать над различными версиями проекта параллельно, а затем объединять их изменения обратно в основную ветку.
9. Слияние (merging). Это процесс объединения изменений из одной ветки с другой. После того, как изменения в одной ветке были завершены и протестированы, их можно объединить с другой веткой, чтобы интегрировать эти изменения в основной поток разработки.
10. Отслеживание проблем и задач (tasks). Некоторые VCS интегрированы с системами управления задачами, такими как GitHub Issues (<https://github.com/features/issues>)  или  Jira  (<https://www.atlassian.com/ru/software/jira>). Это позволяет разработчикам связывать изменения в коде с определенными проблемами или задачами, что упрощает отслеживание прогресса и взаимодействие в команде.
11. Аудит и безопасность (security). VCS обеспечивают аудит изменений, что позволяет организациям следить за тем, кто и когда вносил изменения в код или другие файлы проекта. Также они могут предоставлять механизмы контроля доступа, чтобы управлять правами доступа к репозиториям и файлам.

# 1.2. Обзор существующих решений

## 1.2.1. Git

Git – на данный момент самая популярная VCS из существующих. Она обладает децентрализованностью (т.е. каждый клиент копирует полностью историю репозитория, что обеспечивает большую надежность и гибкость при работе в распределенных командах, независимо от подключения к сети). Также Git имеет большой набор инструментов для командной работы, такие как работа с ветками и слияниями, управление конфликтами, рецензии кода.

Рассмотрим особенности этой VCS. Часть пользовательского интерфейса представлена на рисунке 1.

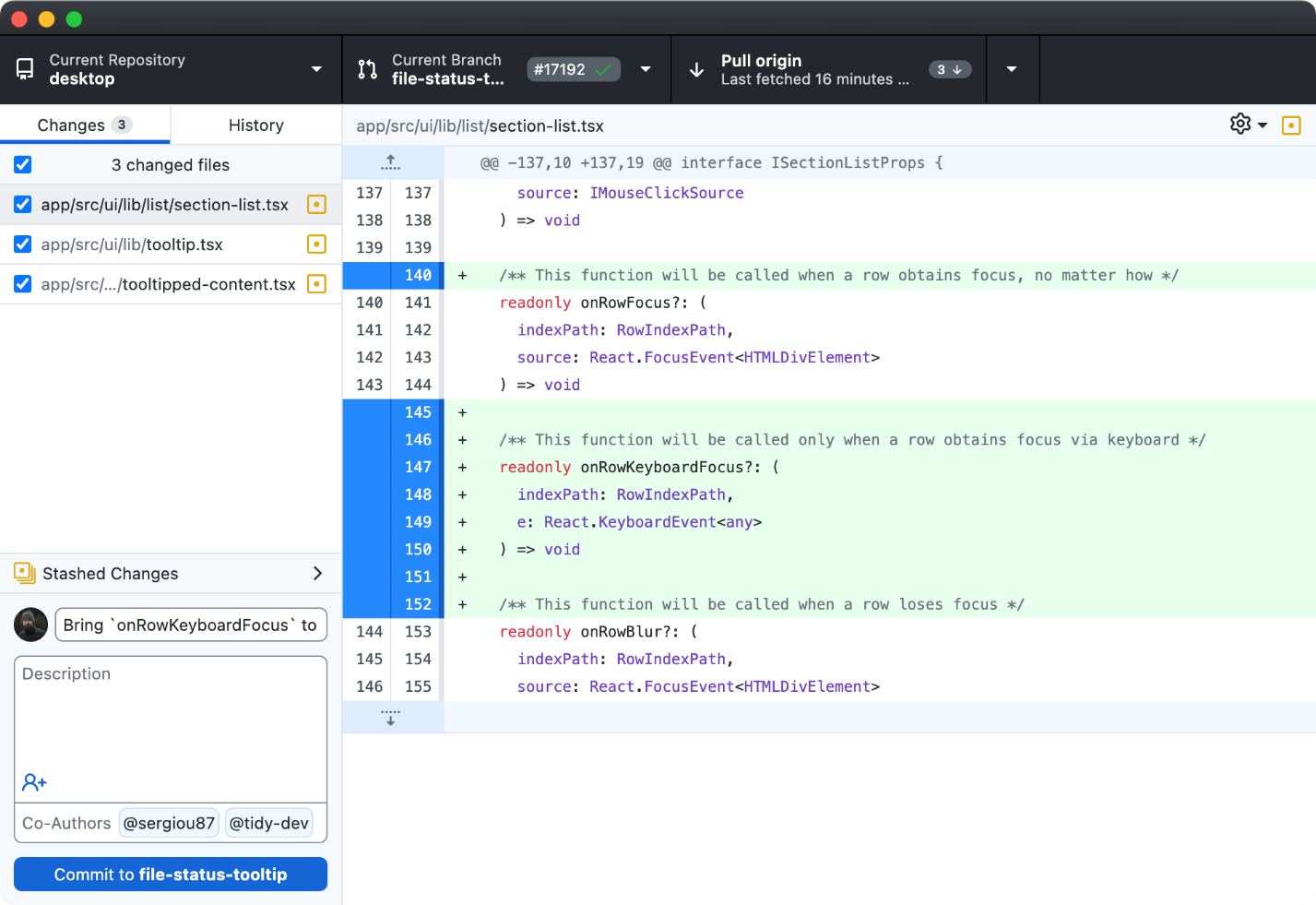


Рисунок 1. Скриншот приложения Git

Плюсы Git:

1. Высокая скорость работы. Благодаря локальному хранению файлов Git обеспечивает быстрое выполнение операций отслеживания изменений, создания коммитов, слияний и прч.
2. Большое сообщество и поддержка. Git имеет огромное сообщество пользователей и разработчиков, что обеспечивает широкий выбор инструментов, библиотек и ресурсов для работы с ним. Также сам Git постоянно развивается, обновляется и улучшается.
3. Гибкость и настраиваемость. Git предоставляет разработчикам широкие возможности для настройки рабочего процесса, использования плагинов и интеграции с другими инструментами разработки, например, с IDE.

Минусы Git:

1. Сложность изучения для новичков. Для некоторых новичков Git может показаться сложным в освоении из-за большого количества команд и концепций, таких как ветвление, слияние и ребейз.
2. Сложный пользовательский интерфейс. Несмотря на широкие возможности Git, его пользовательский интерфейс может показаться неинтуитивным для некоторых пользователей из-за разрозненности команд и параметров.
3. Проблемы с большими файлами и объемом данных. Git может столкнуться с проблемами производительности при работе с очень большими файлами или репозиториями с большим объемом данных. В таких случаях могут потребоваться оптимизации или альтернативные решения.
4. Сложности при переписывании истории. Переписывание истории в Git (например, с помощью команды rebase) может быть опасным и может привести к потере данных или нежелательным результатам, особенно если это делается в общем репозитории.
5. Зависимость от знания и понимания командной строки. Хотя существуют графические интерфейсы для Git, для эффективной работы с ним часто требуется знание командной строки, что может быть непривычно для новичков или тех, кто предпочитает графический интерфейс.

## 1.2.2. Mercurial

Mercurial (<https://www.mercurial-scm.org/>) – тоже одна из самых популярных распределенных VCS. Она использует другую модель хранения данных, нежели Git. В Mercurial каждая ревизия (commit) хранится как отдельный изменяемый объект, в то время как в Git каждая ревизия хранится как набор изменений относительно предыдущей ревизии. Также у Mercurial свои подходы к обработке ветвления и слияний.

Рассмотрим особенности этой VCS. Часть пользовательского интерфейса представлена на рисунке 2.

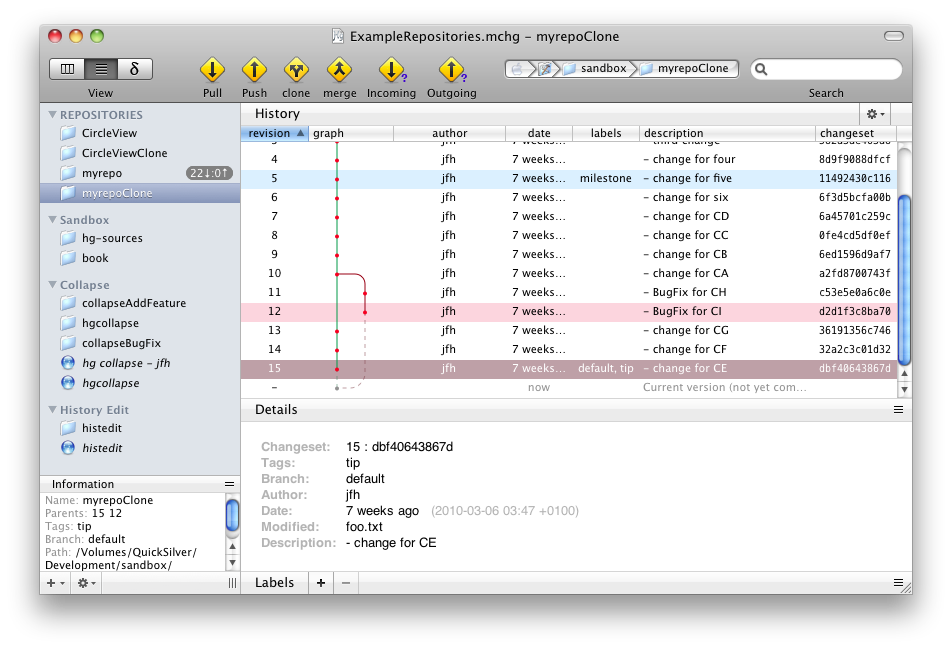


Рисунок 2. Скриншот приложения Mercurial

Плюсы Mercurial:

1. Простота в освоении. Mercurial обладает более простой и интуитивно понятной концепцией, особенно для новичков. Его команды и синтаксис часто кажутся более понятными и предсказуемыми.
2. Полноценная поддержка Windows. Mercurial имеет полноценную поддержку для операционной системы Windows, включая интеграцию с командной строкой и графические интерфейсы.
3. Простая установка и настройка. Установка и настройка Mercurial проще, чем у Git, что делает его более привлекательным для пользователей, которым нужно быстро начать работу с системой контроля версий.
4. Хорошая производительность на некоторых операциях. В некоторых случаях Mercurial может быть быстрее или эффективнее Git при выполнении таких задач, как слияния веток или выполнение некоторых запросов.

Минусы Mercurial:

1. Меньшее сообщество и экосистема. Mercurial менее распространен и менее популярен, чем Git, что может привести к ограниченной поддержке сторонних инструментов, библиотек и расширений.
2. Больший объем данных при работе с репозиториями. Mercurial может потреблять больше оперативной памяти и требовать больше места на диске для хранения данных по сравнению с Git, особенно при работе с большими репозиториями.
3. Ограниченные возможности для совместной работы над проектами. В Git существует более широкий выбор инструментов для совместной работы над проектами, таких как форки, pull request и рецензии кода.

Исходя из обзора рынка существующих VCS, можно сделать следующие выводы:

1. Монополия Git. Несмотря на то, что существуют разные системы контроля версий, их количество не очень велико. И Git на данный момент является доминирующей VCS, которую используют большинство разработчиков.
2. Пользовательский опыт и удобство использования. Важным аспектом при выборе VCS является ее удобство использования. Пользователи предпочитают системы с интуитивно понятным интерфейсом, простыми и понятными командами, а также широкой поддержкой сообщества и ресурсов для обучения.
3. Производительность и масштабируемость. В современном мире разработки ПО данные требования становятся все более важными. Компании и проекты ищут системы, которые могут эффективно обрабатывать большие объемы данных и обеспечивать высокую производительность при работе с репозиториями.
4. Интеграция и совместимость. Важным аспектом при выборе системы контроля версий является ее интеграция с другими инструментами и сервисами, такими как среды разработки, системы сборки, непрерывной интеграции и хостинг-провайдеры репозиториев. Пользователи и организации часто выбирают системы, которые легко интегрируются с их существующими рабочими процессами и инфраструктурой.

В целом, рынок систем контроля версий продолжает развиваться и адаптироваться к изменяющимся потребностям разработчиков и организаций. На данный момент, Git доминирует над остальными VCS за счет широкого набора инструментов для опытных разработчиков и большого сообщества, но является сложным для новичков, имеет проблемы при редактировании истории коммитов и работе с большими файлами.

В своей системе контроля версий я хочу объединить преимущества вышеперечисленных VCS. Я хочу сделать ее доступней для новичков, не перегружая большим количеством действий и команд, но также сохранить гибкость настроек для опытных пользователей и совместимость с существующими инструментами разработки.

# 1.3. Формирование технического задания

## 1.3.1. Предмет разработки

Предметом разработки является система контроля версий для управления изменениями в исходном коде и других ресурсах проекта в течении времени.

Назначение приложения:

1. Отслеживание изменений. VCS позволяют записывать и хранить изменения, внесенные в файлы проекта. Это включает добавление новых файлов, удаление существующих файлов и внесение изменений в содержимое файлов.
2. История версий. Системы контроля версий сохраняют историю изменений, позволяя пользователям просматривать предыдущие версии файлов и анализировать историю разработки.
3. Откат к предыдущим версиям. VCS позволяют легко откатываться к предыдущим версиям проекта в случае необходимости. Это полезно при обнаружении ошибок или нежелательных изменений.
4. Коллаборация. Системы контроля версий позволяют нескольким разработчикам работать над одним проектом одновременно. Они могут синхронизировать свои изменения, обмениваться кодом и отслеживать работу других участников команды.
5. Аудит и управление изменениями. Системы контроля версий обеспечивают аудит изменений и могут предоставлять информацию о том, кто, когда и что изменил в проекте.

Отличия от конкурентов:

1. Простота в использовании. Меньший набор основных команд, которые легко запомнить новичкам и наличие справочной информации по командам.
2. Русификация и возможность смены языка. Устраняет языковой барьер для пользователей, которые не обладают знаниями английского. Возможность переключать язык во время использования VCS.
3. Шифрование данных репозитория. Возможность одной командой зашифровать все данные текущего проекта алгоритмом шифрования с высоким уровнем криптостойкости.
4. Автокоммит. Возможность включения автоматического создания коммитов, которые создаются при наличии изменений в проекте.
5. Хранение данных в облачных хранилищах. Возможность сохранения данных репозитория на Яндекс Диске.
6. Функционал торговой площадки. Возможность для пользователей монетизировать свои проекты и программы, продавая их, и возможность приобрести их для других пользователей.

Цель разработки системы контроля версий заключается в предоставлении инструмента, который поможет разработчикам легко и эффективно управлять разработкой программного обеспечения и других проектов, обеспечивая эффективное управление изменениями, коллаборацию и сохранность истории разработки, повышать качество кода, улучшать взаимодействие внутри команды разработчиков и лучше организовывать процесс разработки.

## 1.3.2. Архитектура приложения

Система контроля версий будет состоять из двух главных компонентов – клиент и сервер. Вместе клиент и сервер образуют архитектурную модель распределенной системы контроля версий (DVCS), где каждый разработчик имеет локальную копию репозитория на своем компьютере (клиенте), и изменения синхронизируются между клиентами через центральный сервер.

Таким образом, клиент и сервер могут рассматриваться как части архитектуры системы контроля версий, выполняющие различные роли и обеспечивающие взаимодействие между пользователями и репозиториями.

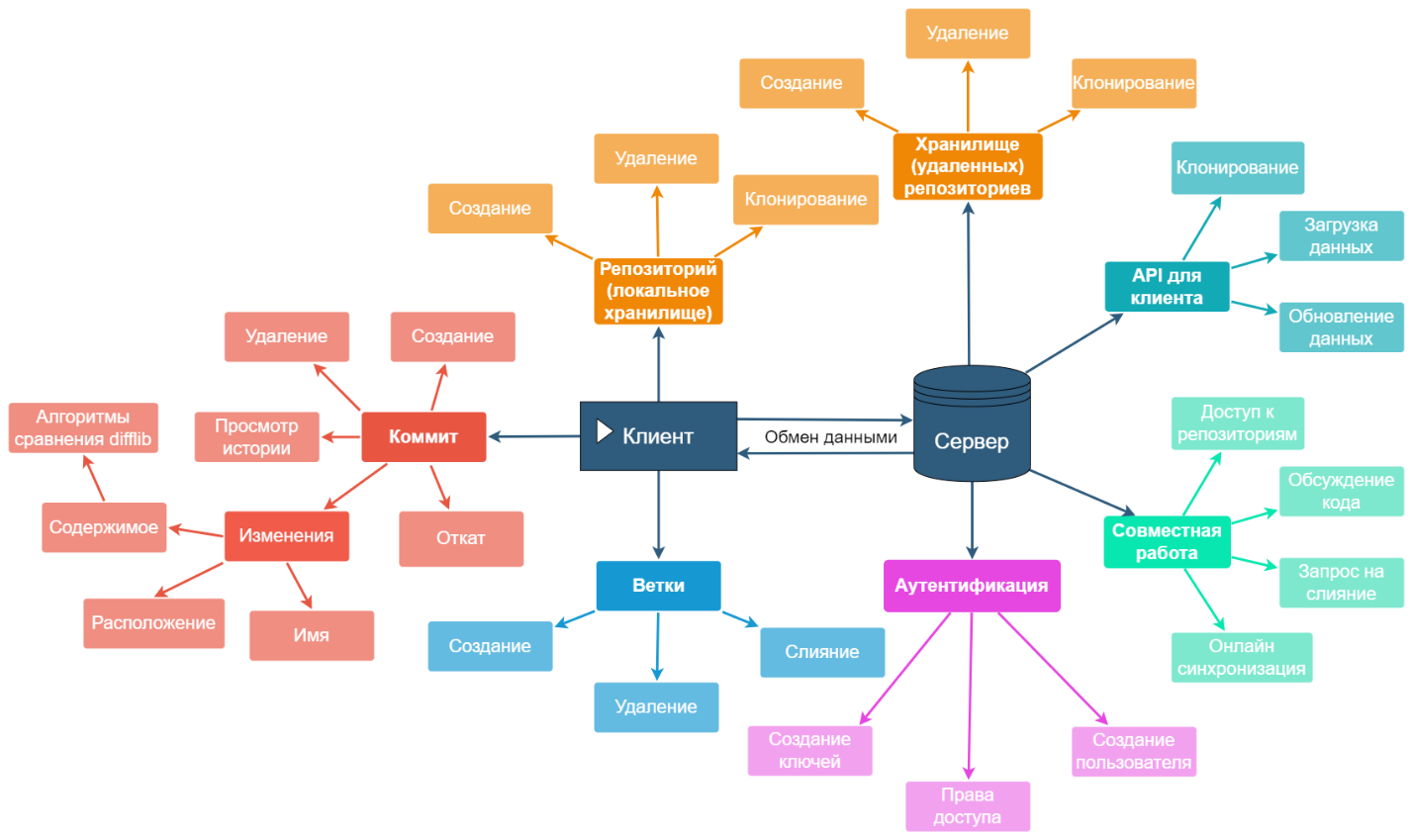


Рисунок 3. Архитектура приложения

### 1.3.2.1 Клиент VCS

Это программное обеспечение, которое устанавливается и используется локально на компьютере пользователя для создания и работы с локальным или удаленным репозиторием. Клиент позволяет пользователям выполнять такие операции, как создание коммитов, просмотр изменений, загрузка и отправка изменений на удаленный репозиторий, а также просмотр истории и другие действия, связанные с управлением версиями файлов. Клиент будет представлен консольным приложением.

### 1.3.2.2 Сервер VCS

Это программное обеспечение, которое управляет хранением и изменением репозиториев, обслуживает запросы клиентов и обеспечивает доступ к данным проекта через сеть. Сервер должен быть реализован в виде веб-сервиса, что сделает возможным его использование на любой платформе. Данный сервис должен иметь инструменты аутентификации и авторизации пользователей, API для взаимодействия с клиентами, возможность просмотра данных репозитория, папок и файлов.

## 1.3.3. Функциональные требования к клиенту VCS

1. Создание репозитория. Пользователи могут создавать новые репозитории для хранения и управления исходным кодом и другими ресурсами проекта в любом месте файловой системы.
2. Игнорирование файлов. Пользователи могут создавать .gitignore файлы в которых указывать игнорируемые файлы и папки, которые не будут отслеживаться VCS.
3. Коммит изменений. Пользователи могут фиксировать изменения в репозитории, создавая коммиты с описанием внесенных изменений.
4. История изменений. Пользователи могут просматривать историю изменений репозитория, включая список коммитов, диффы изменений и метаданные о каждом коммите.
5. Откат к предыдущим версиям. Пользователи могут откатываться к предыдущим версиям файлов или к предыдущим коммитам для восстановления предыдущего состояния проекта.
6. Загрузка репозитория на сервер. Пользователи могут загружать репозиторий на удаленный сервер для хранения данных на нем и обмена информацией с другими пользователями.
7. Клонирование репозитория. Пользователи могут клонировать удаленный репозиторий для создания локальной копии проекта на своем компьютере.
8. Удаление репозитория. Пользователи должны иметь возможность удалять репозиторий локально.
9. Русификация и выбор языка. Пользователи должны иметь выбрать удобный для них язык или переключиться на него во время работы программы.
10. Шифрование данных. Пользователи должны иметь возможность зашифровать данные репозитория и получить ключ для дешифрования.

## 1.3.4. Функциональные требования к серверу VCS

1. Хранилище репозиториев. Сервер содержит хранилище, где хранятся репозитории проектов. Каждый репозиторий содержит файлы, история изменений, и другие метаданные проекта.
2. Просмотр данных репозитория, папок и файлов. Пользователи могут открыть любую папку репозитория и посмотреть любые файлы внутри него.
3. Удаление репозитория и его данных.
4. Работа с клиентскими запросами. Сервер предоставляет протоколы и API для взаимодействия с клиентскими приложениями. Это включает в себя загрузку данных репозитория с клиента на сервер и клонирование репозитория с сервера на клиент.
5. Совместная работа. Пользователи могут совместно работать над проектом, обмениваясь изменениями.
6. Аутентификация и авторизация. Аутентификация пользователей при доступе к репозиториям. Использование механизмов авторизации, таких как пароли, и токены доступа.

## 1.3.5. Нефункциональные требования

Для приложения определены следующие нефункциональные требования:

1. Удобство использования. VCS должна быть простой в использовании, иметь интуитивно понятный интерфейс для разработчиков и обладать минимумом команд для выполнения основных функций. Пользователи должны легко осваивать приложение.
2. Производительность. Система должна быть быстрой и эффективной в обработке запросов пользователей, таких как коммиты и загрузка изменений.
3. Масштабируемость. Система должна масштабироваться для поддержки большого числа пользователей и репозиториев, а также обеспечивать возможность расширения ее возможностей в будущем.
4. Надежность и отказоустойчивость. Система должна быть надежной и стабильной, минимизируя возможность потери данных и сбоев в работе.
5. Безопасность. Система должна обеспечивать защиту данных и конфиденциальность информации пользователей, а также предоставлять механизмы аутентификации и авторизации для контроля доступа к репозиториям.
6. Доступность. Приложение должно быть доступным для пользователей в любое время, с минимальными периодами недоступности.
7. Совместимость и адаптивность. Клиент должен быть совместим с различными операционными системами (Windows, MacOS, Unix), а сервер должен быть совместим с различными браузерами и иметь адаптивный интерфейс.
8. Документация и поддержка. Система должна иметь подробную документацию, руководство пользователя и техническую поддержку для помощи пользователям в работе с ней и решения возможных проблем.
9. Хранение и резервное копирование данных. Система должна обеспечивать надежное хранение данных и регулярное создание резервных копий для защиты от потери информации.

# 2. Проектирование системы контроля версий

# 2.1. Общая архитектура

Про общую архитектуру VCS уже было сказано в [главе 1.3.2.](#_1.3.2._Архитектура_приложения)

Она будет состоять из двух компонентов – клиента и сервера. Здесь подробнее рассмотрим особенности этих компонентов.

# 2.2. Репозиторий

Основополагающий компонент. Это хранилище наших данных. Именно здесь мы храним коммиты, информацию об изменениях, файлы и прочие элементы. Схема репозитория представлена на рисунке 4.

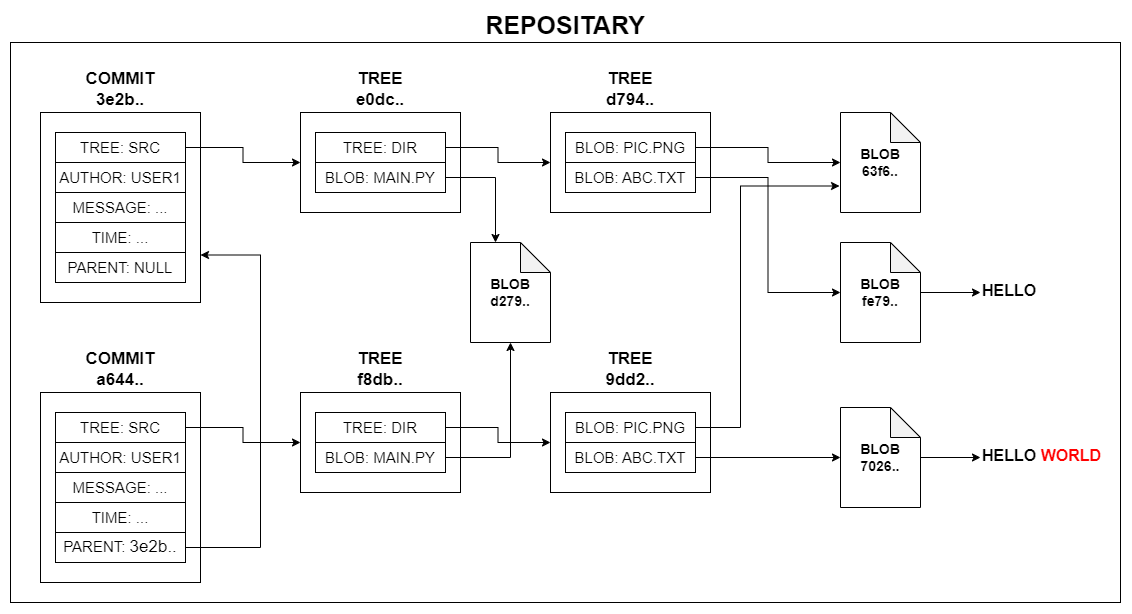


Рисунок 4. Схема репозитория

Репозиторий хранит в себе все созданные коммиты. Каждый коммит хранит hash дерева проекта, имя автора, комментарий, время создания и hash родительского коммита (кроме первого коммита, у него хранится null). Дерево проекта представляет корневую папку проекта. Как и остальные деревья, оно может содержать имя папки и список содержимого (точнее список из hash-значений). Элементами этого списка могут быть как другие деревья, так и файлы. Объект файл содержит имя файла и hash его Blob’а (содержимого). В Blob хранится только содержимое файла. У каждого из этих объектов есть также такое поле как hash – он же является именем для данного объекта в файловой системе. Все перечисленные объекты хранятся в папке data, которая находится в репозитории (папка .vcs). Также в репозитории хранится файл HEAD. Этот файл содержит hash последнего коммита.

На рисунке 5 представлен пример создания репозитория с одной папкой и одним файлом. В папке data было создано 4 файла – дерево для папки проекта «Новая папка», файл для «Новый текстовый документ.txt», Blob для хранения содержимого файла и файл самого коммита.

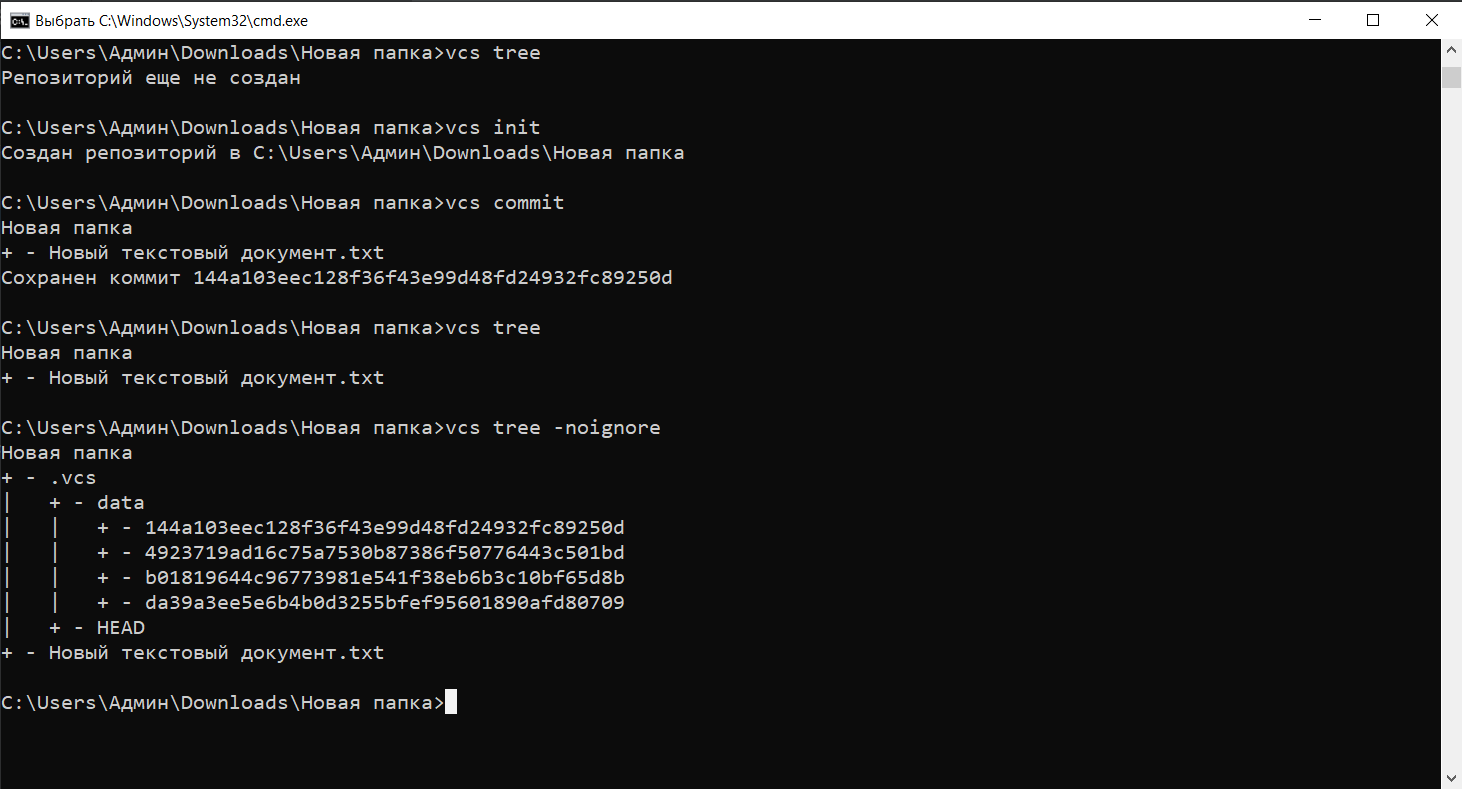


Рисунок 5. Пример репозитория

# 2.3. База данных

База данных потребуется для хранения данных о пользователях VCS и репозиториях, хранящихся на сервере. В качестве БД я выбрал SQLite. Это удобная и компактная БД, которая не требует отдельного сервера и настройки [5]. Информация в ней хранится в виде файла. Такая БД очень удобна для тестирования и использования в небольших проектах и программах, которые часто выполняют прямые операции чтения/записи на диск.

Структура БД отображена на рисунке 6.

В ней находится 3 таблицы: User, Profile, Repository. Таблицы User и Profile хранят информацию о пользователе, между ними связь один к одному. Таблица Repository соответственно хранит информацию о репозитории и имеет внешний ключ – id пользователя.

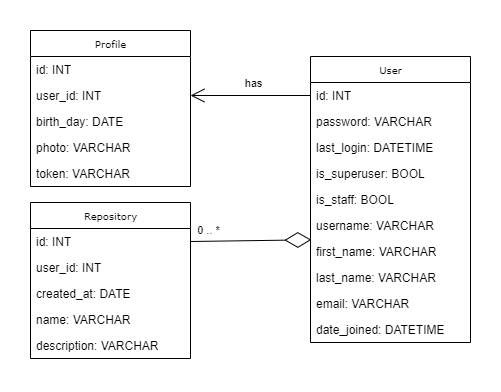


Рисунок 6. Структура БД

# 2.4. Выбор инструментов и средств разработки

В качестве средств для разработки были выбраны:

1. Python - основной язык программирования.
2. Django – фреймворк для создания веб-приложений.
3. SQLite – БД для хранения данных.
4. Difflib – библиотека для сравнения последовательностей данных и нахождения различий между ними.
5. Hashlib – библиотека для генерации hash-значений данных.
6. Pycharm - интегрированная среда разработки.
7. Bootstrap – библиотека, предоставляющая набор инструментов для верстки стильных и адаптивных компонентов пользовательского интерфейса
8. Dropzone.js – Java Script библиотека, предоставляющая вам интерфейс для загрузки файлов на сайт с помощью drag and drop (перетавскивания).
9. Pytest и Selenium – для тестирования системы.

# 3. Разработка системы контроля версий

# 3.1. Реализация клиента VCS

### 3.1.1. Хранение объектов файловой системы

Разработку клиента я начал с поиска и создания инструментов, которые отслеживают информацию об изменениях файлов и папок и хранят эти данные. За основу я взял модель хранения данных в Git репозиториях.

Для понимания, что файл был изменен можно использовать hash функции, которые вычисляют уникальный hash объекта, исходя из его содержимого. Для получения hash файла я складывал значение hash Blob’a (объект, хранящий содержимое файла) с именем файла. Таким образом, если файл будет переименован или изменено содержимое, у него будет новый hash.

Для хранения информации о каталоге был создан класс Tree, который хранит список файлов (объектов File) и подкаталогов (объектов Tree). Hash каталога вычисляется как сумма hash значений всех вложенных в него объектов + имя каталога.

Чтобы сохранить текущее состояние проекта создается объект Commit - снимок проекта в определенный момент времени. Commit хранит ссылку на каталог проекта (Tree), ссылку на предыдущий коммит (если он был), а также имя автора, сообщение и время создания. При сравнении коммитов сравниваются только объекты Tree, без учета времени и предыдущих коммитов.

Все перечисленные классы (Blob, File, Tree, Commit) имеют методы save() и load() для сохранения в виде файла и обратной загрузки. При сохранении каждый объект записывает бинарную информацию в файл репозитория. В качестве имени файла используется его hash значение. Таким образом если есть несколько файлов с одинаковым содержимым – у них будут одинаковые Blob объекты и в файловой системе будет храниться только один файл Blob’a, на который будут указывать несколько объектов File.

Проблема выбранного способа хранения заключается в том, что может возникнуть ситуация, когда у двух разных файлов будет одинаковый hash и они будут сохранены как один файл. Это связано с ограниченным количеством hash значений которые можно использовать (поскольку длина hash строки ограничена). В моей программе используется алгоритм хэширования SHA-1, который возвращает строку из 40 шестнадцатеричных символов (160 бит) что дает различных комбинаций. Вероятность описанной ситуации довольно мала, но есть. Можно увеличить размер hash до 256 или 512 бит, но это может повлечь к проблемам, связанным с ограничением на длину имен файлов и длину полного пути к файлу.

Листинг кода получения hash значений и классов, представляющих объекты файловой системы в Приложении 1.

### 3.1.2. Структура хранения файлов в репозитории

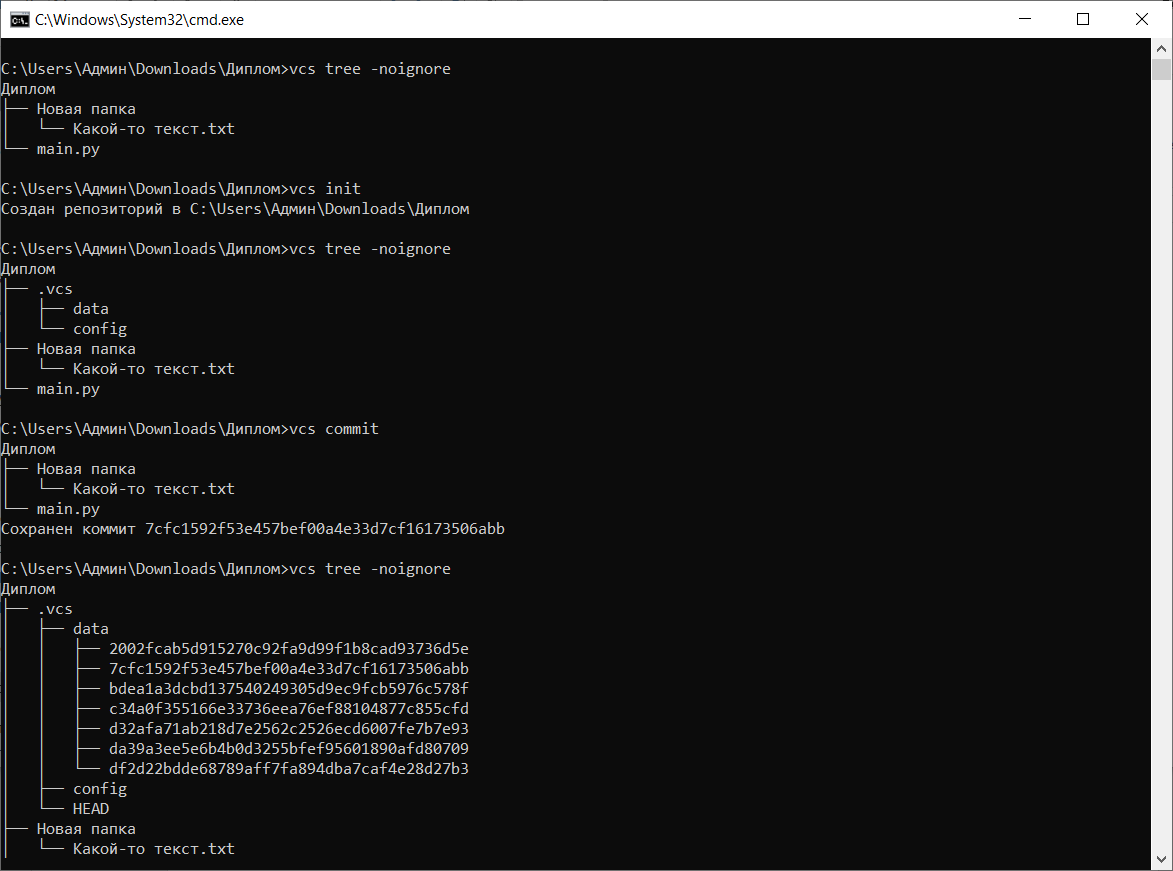


Рисунок 7. Пример хранения данных репозитория

На рисунке 7 можно увидеть, как через консоль создаются коммит в папке «Диплом» в которой изначально хранится одна папка «Новая папка» и два текстовых файла.

Сначала выводится файловая структура корневой папки с помощью команды vcs tree –noignore. Флаг –noignore нужен чтобы не игнорировались файлы и папки. После, командой vcs init инициализируется репозиторий. Создается папка репозитория .vcs, в этой папке находится папка data, куда будут сохраняться все объекты файловой системы и изменения. Также создается файл config, в котором хранится информация о используемом языке и в дальнейшем будет храниться информации о пользователе (имя, почта, название удаленного репозитория, токен), после первой отправки файлов на сервер VCS. Коммит создается командой vcs commit, после чего все объекты (Blob, File, Tree, Commit) сохраняются в папке .vcs/data. На рисунке 7 видно как было создано 7 файлов: 2 файла объектов Tree (папки «Диплом», «Новая папка»), 2 файла объекта File («Какой-то текст.txt», «main.py»), 2 файла объекта Blob (содержимое файлов) и 1 файл объекта Commit. Имена файлов совпадают с их хэшами. Также был создан файл HEAD, в котором теперь будет храниться hash текущего коммита (для проверки изменений и проч.)

### 3.1.3. Сравнение файлов и создание патчей

После того, как по hash значениям мы поняли, что в файл был изменен, нужно получить список этих изменений (разница между старой и новой версиями файлов). Для получения этих изменений и работы с ними я использовал библиотеку difflib. Она хорошо подходит для сравнения последовательностей символов и генерации патчей (процесс определения и записи изменений, внесенных в файл, по сравнению с его предыдущей версией. Патч содержит информацию о различиях между двумя версиями файла и может быть применен к исходной версии файла для преобразования его в новую версию.) Эта библиотека основывается на алгоритме Мэйерса и используется для нахождения наименьшего количества операций (вставка, удаление, замена), необходимых для преобразования одной последовательности в другую. Алгоритм Мэйерса работает за время O(ND), где N — длина последовательностей, а D — размер минимального набора операций для преобразования одной последовательности в другую.

Используя данные инструменты, мы решаем сразу две проблемы – определение различий между файлами и уменьшение размера хранимых файлов (без использования данного алгоритма пришлось бы хранить обе версии файла, а с ним мы храним изначальную версию файла и список изменений).

### 3.1.4. Сравнение деревьев и коммитов

В начале создается текущий коммит (снимок) проекта. Алгоритм проходится по всем объектам файловой системы, создает объекты классов Blob, File, Tree описанных ранее. После создания коммита происходит его сравнение с предыдущим коммитом. Это происходит, как уже говорилось выше, путем сравнения деревьев коммитов (корневой каталог проекта). Сравниваются их hash значения и имена. При наличии изменений алгоритм рекурсивно сравнивает подкаталоги и файлы, сохраняя список изменений и возвращая его.

На рисунке 8 можно посмотреть пример получения изменений в репозитории. Был добавлен файл «новый файл.bmp», переименован «main.py» в «main2.py» и изменен «Какой-то текст.txt». После выполнения команды мы видим список изменений между текущим состоянием проекта и предыдущим.

Листинг кода, осуществляющего сравнение коммитов в Приложении 2.

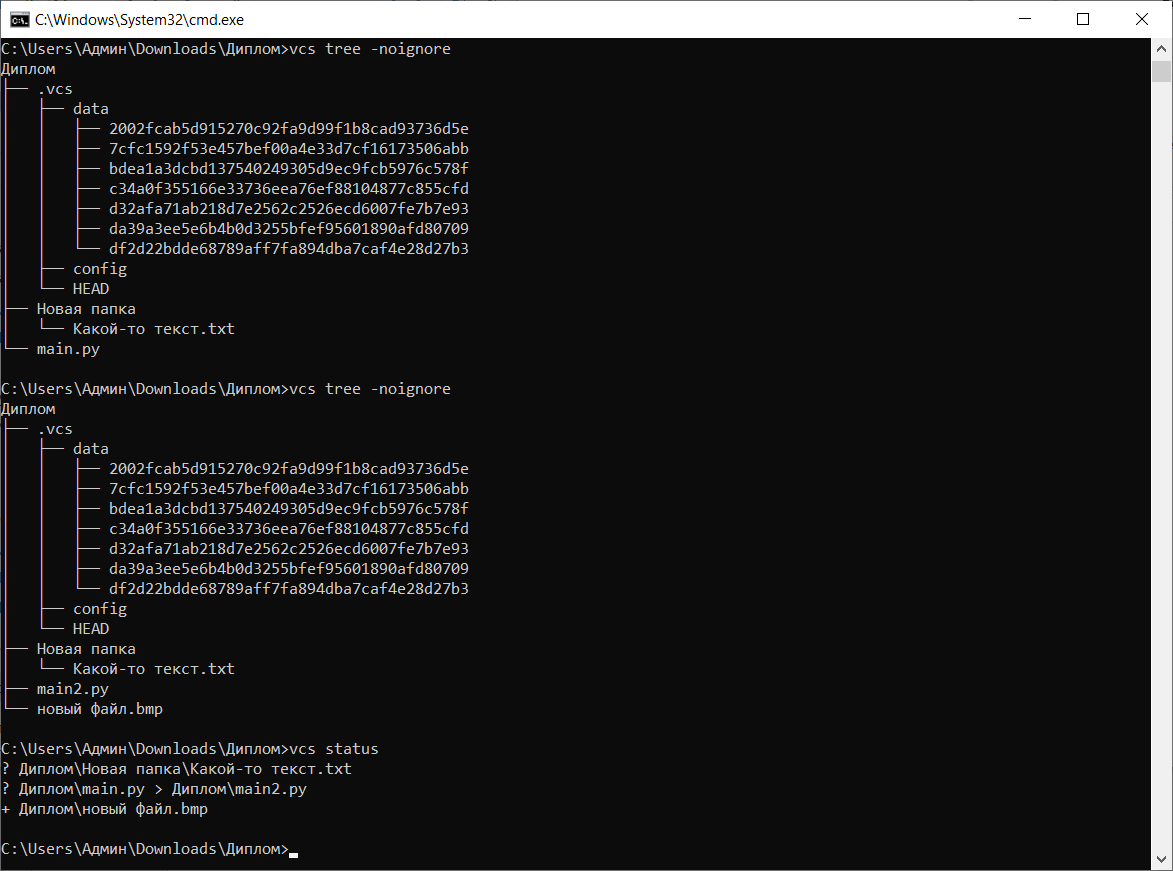


Рисунок 8. Пример изменения файлов репозитория

### 3.1.5. Отмена изменений и откат к предыдущему состоянию

Представим ситуацию, что мы допустили ошибку и хотим вернуться к предыдущей версии нашего проекта. Для начала просмотрим историю изменений с помощью команды vcs hist. Копируем hash последнего коммита и возвращаемся к нему командой vcs rollback <hash>. После этого проверяем, что все файлы вернулись в прежнее состояние.

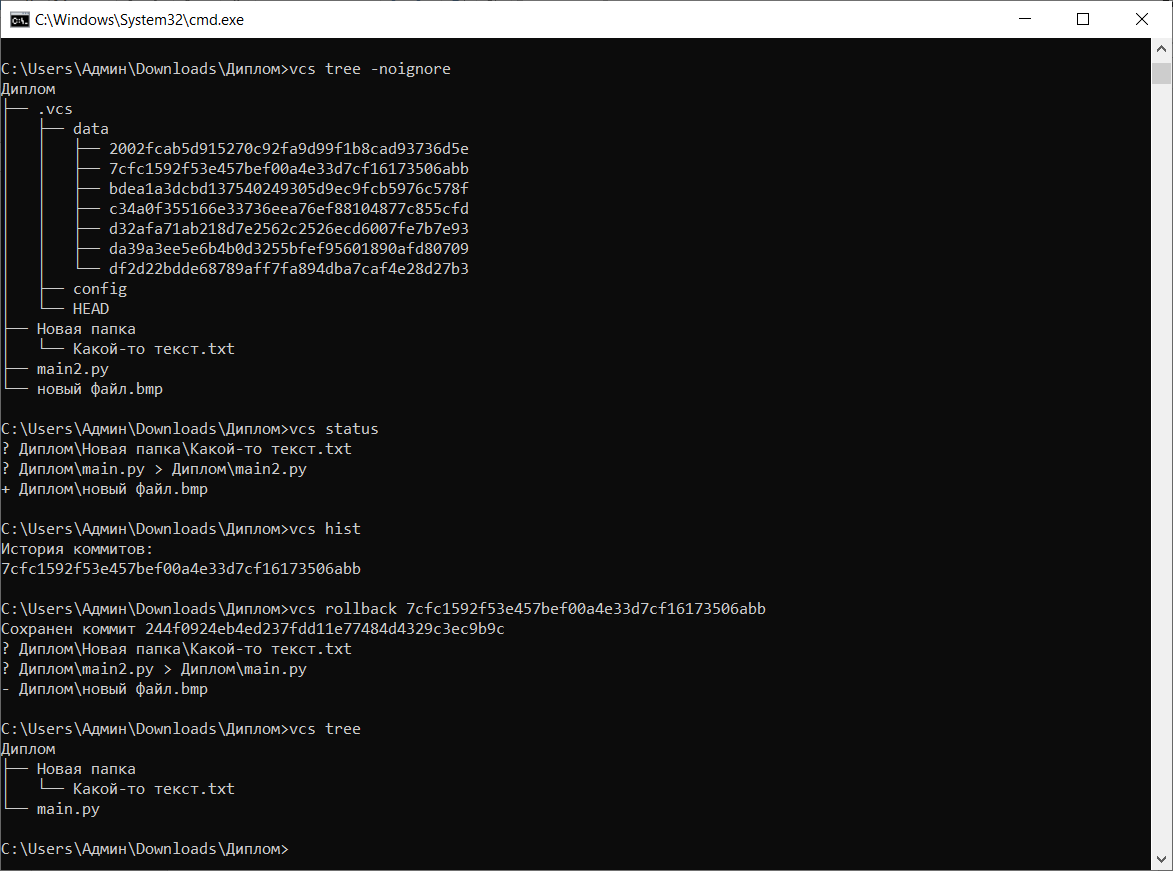
Листинг кода, который возвращает в состояние предыдущего коммита в Приложении 3. Данная функция получает список изменений между коммитами, вызывая функцию из Приложения 2, удаляет новые и измененные файлы и папки после нового коммита, после чего проходится по дереву старого коммита и восстанавливает удаленные и измененные файлы и папки.

Рисунок 9. Пример отката изменений

Эта функция также используется для возвращения к предыдущему коммиту (состоянию)

В Django для создания таблиц в базе данных SQLite используется механизм миграций (migrations). Миграции позволяют вам определить структуру вашей базы данных с использованием моделей Django и затем автоматически создать или изменить таблицы базы данных в соответствии с вашими моделями. Сначала для каждой таблицы необходимо создать модель, после чего можно применить миграции и эти таблицы будут созданы в БД. Листинг кода создания таблиц в Приложении 1.

# 3.1. Реализация системы хранения данных

В Django для создания таблиц в базе данных SQLite используется механизм миграций (migrations). Миграции позволяют вам определить структуру вашей базы данных с использованием моделей Django и затем автоматически создать или изменить таблицы базы данных в соответствии с вашими моделями. Сначала для каждой таблицы необходимо создать модель, после чего можно применить миграции и эти таблицы будут созданы в БД. Листинг кода создания таблиц в Приложении 1.

# 3.2. Реализация пользовательского интерфейса

Для разработки графического интерфейса были использованы HTML, CSS, JavaScript и библиотека MDBootstrap. С помощью HTML создается структура страницы, далее добавляются стили с помощью CSS и классов из MDBootstrap, а с помощью JavaScript создаются динамические элементы и события.

При открытии веб-приложения отобразится страница входа с полями для ввода данных, кнопка войти и ссылка для регистрация нового аккаунта (рисунок 6.).

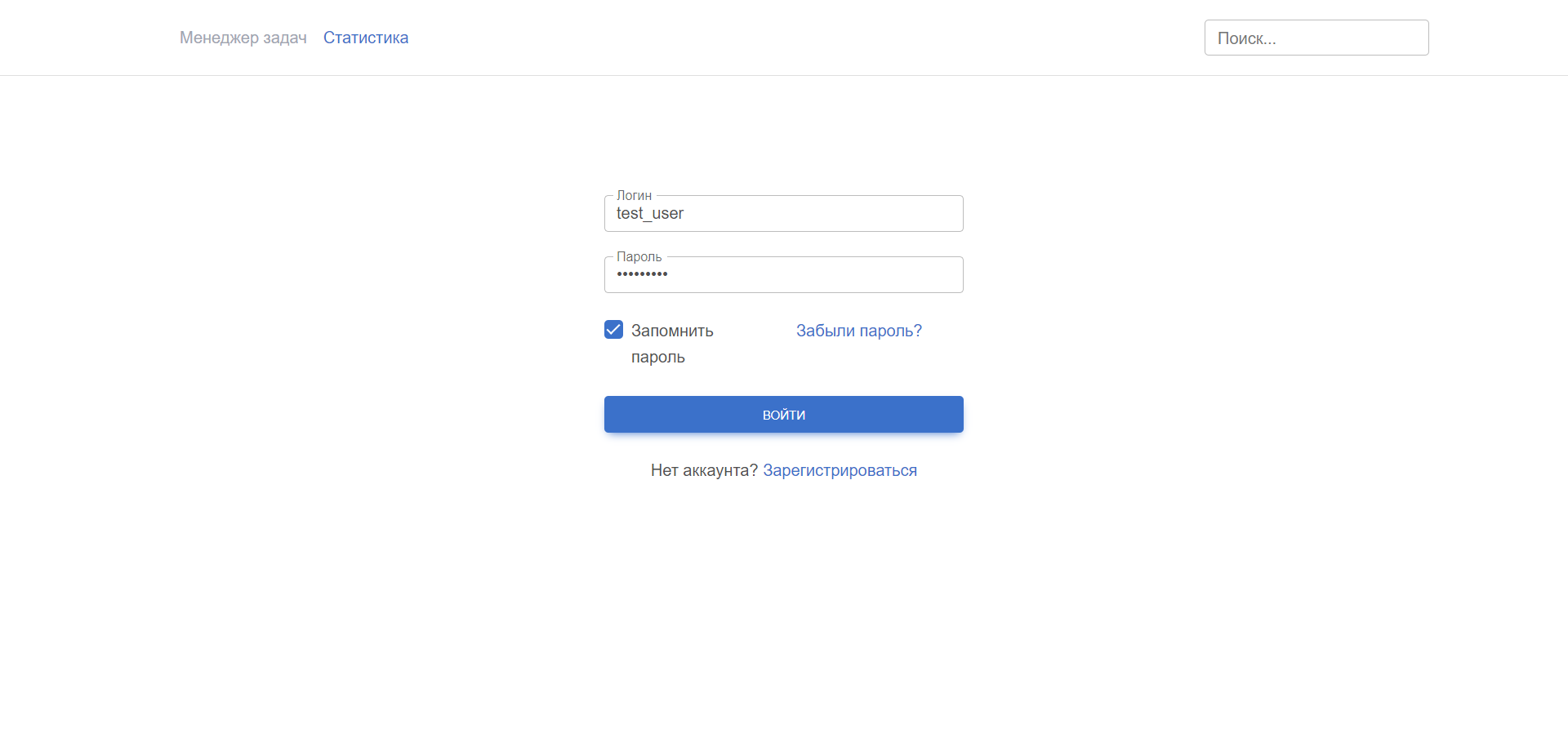


Рисунок 6. Страница входа

Чтобы попасть на главную страницу, нужно пройти авторизацию или регистрацию, страница которой выглядит следующим образом (рис 7). На ней находятся поля для данных аккаунта и кнопка регистрации.

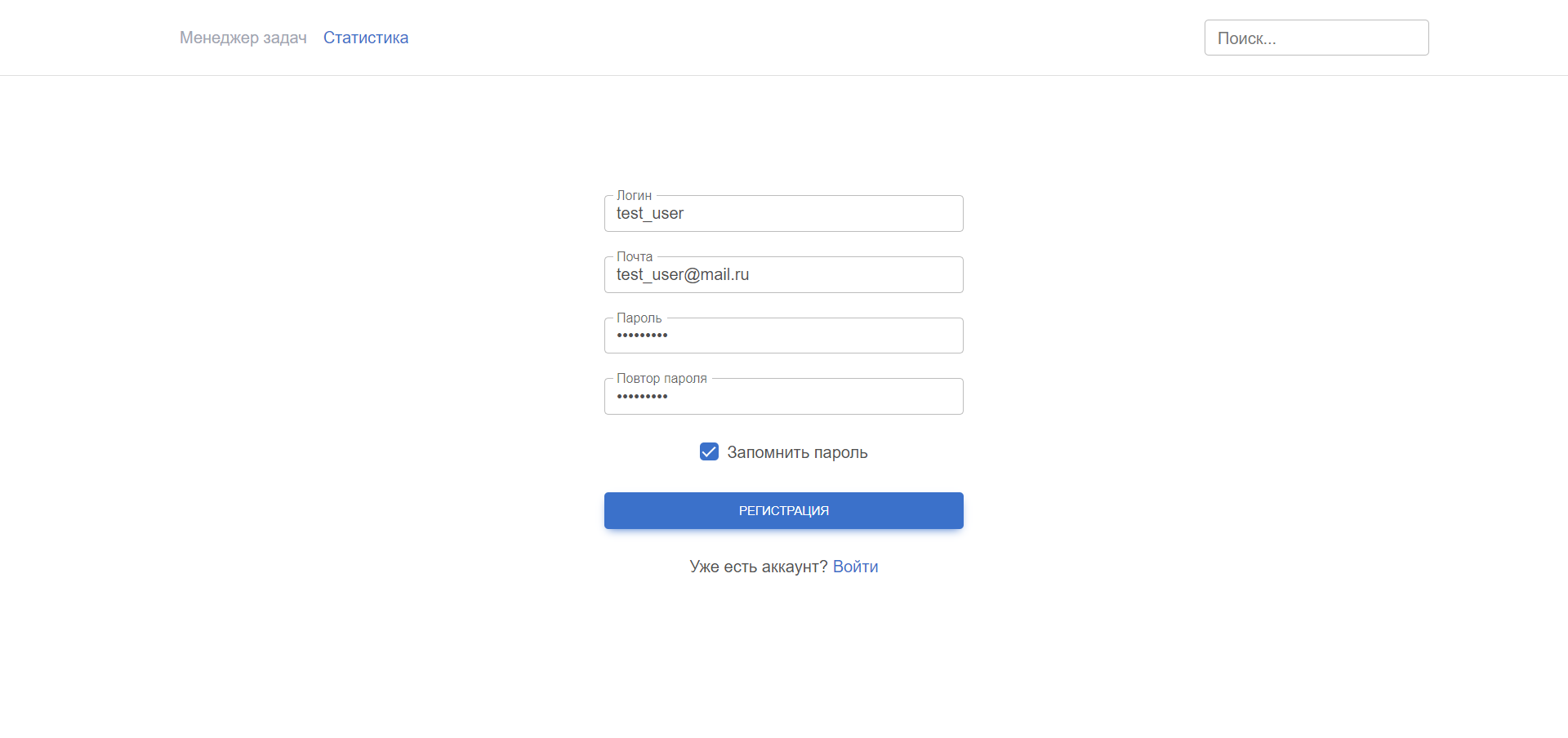


Рисунок 7. Страница регистрации

На рисунке 8 – страница личного профиля. После регистрации пользователь попадает сюда. Здесь он может просмотреть свои личные данные, а также загрузить свое фото профиля.

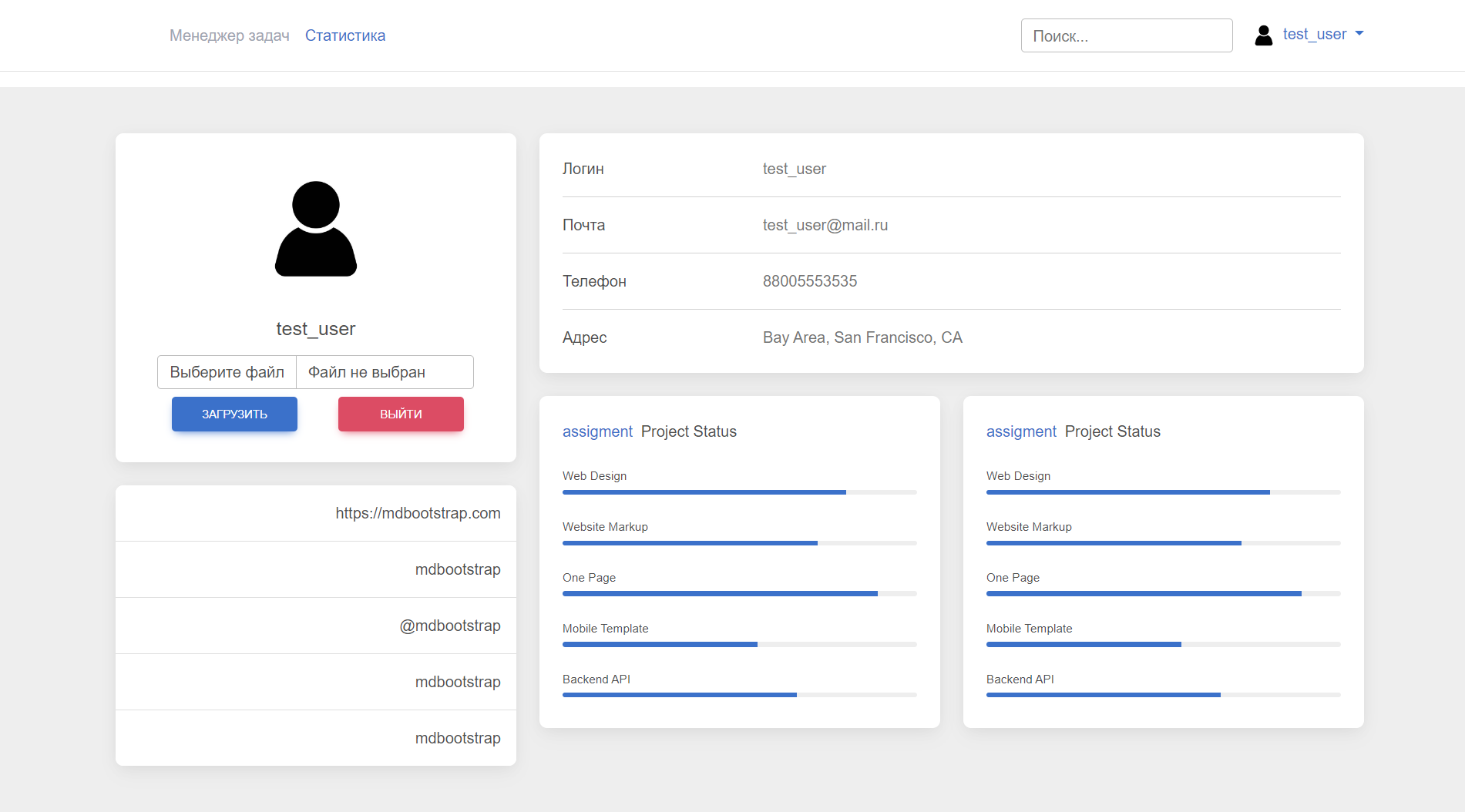


Рисунок 8. Личный профиль

Нажав на логин в верхнем правом углу, откроется выпадающий список, и пользователь может перейти на страницу настроек, где можно изменить личные данные и настроить оповещения в Telegram.

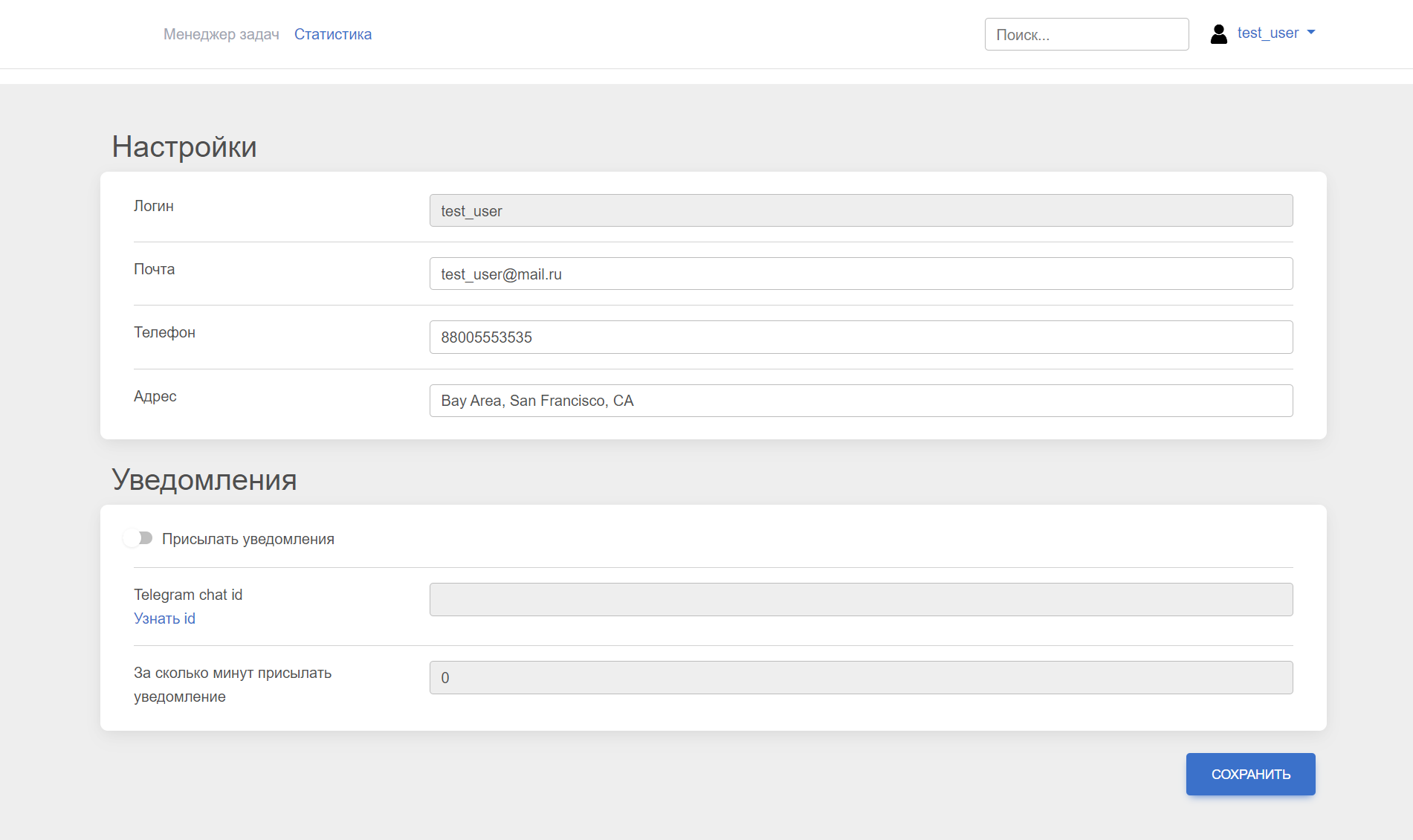


Рисунок 9. Страница настроек.

На главную страницу можно попасть, нажав в верхнем левом углу «Менеджер задач». Здесь можно добавлять, редактировать, удалять задачи и категории задач.

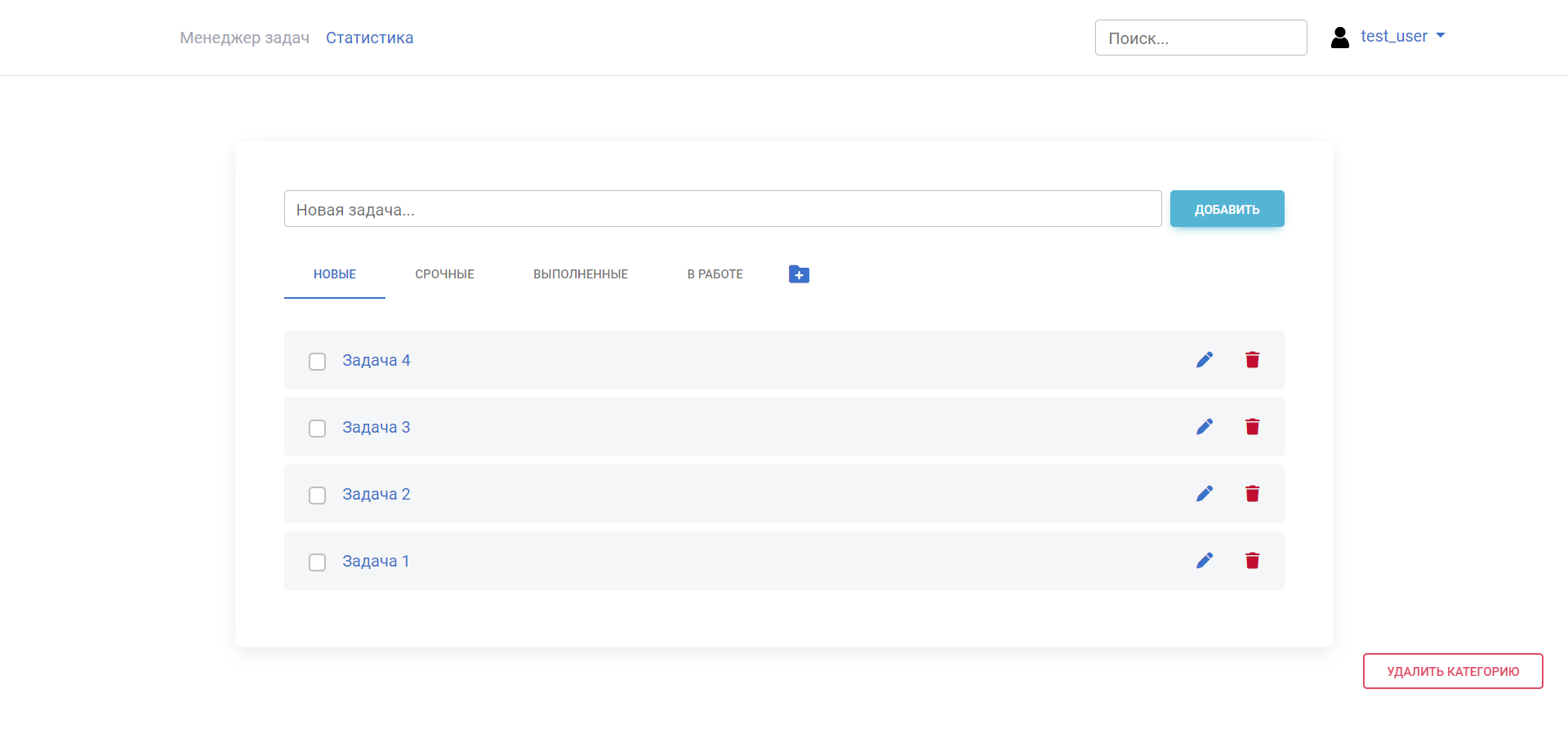


Рисунок 10. Главная страница

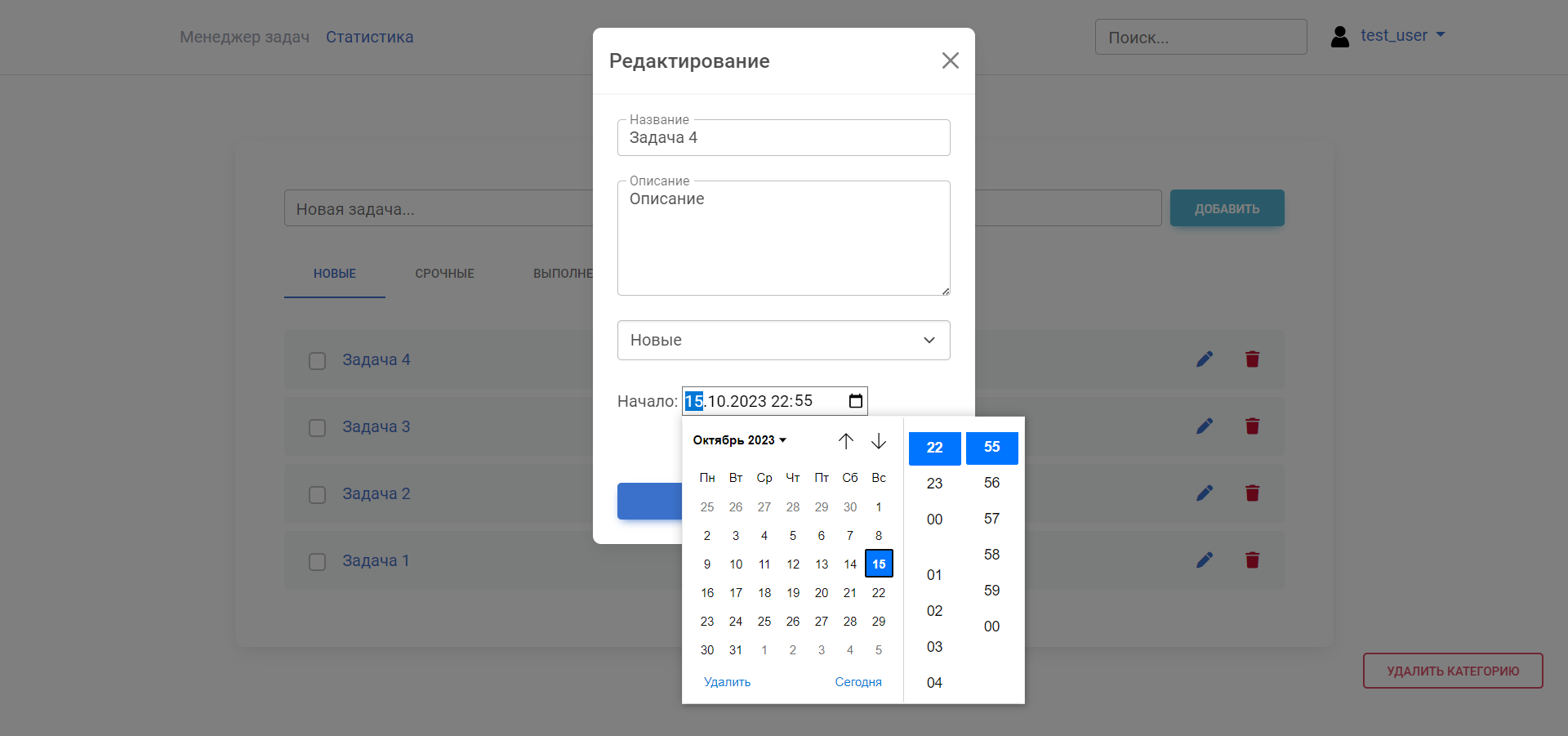
При нажатии на задачу открывается меню ее редактирования (рис 11).

Рисунок 11. Редактирование задачи

При нажатии на иконку папки с плюсом рядом с категориями открывается меню создания новой категории (рис 12).

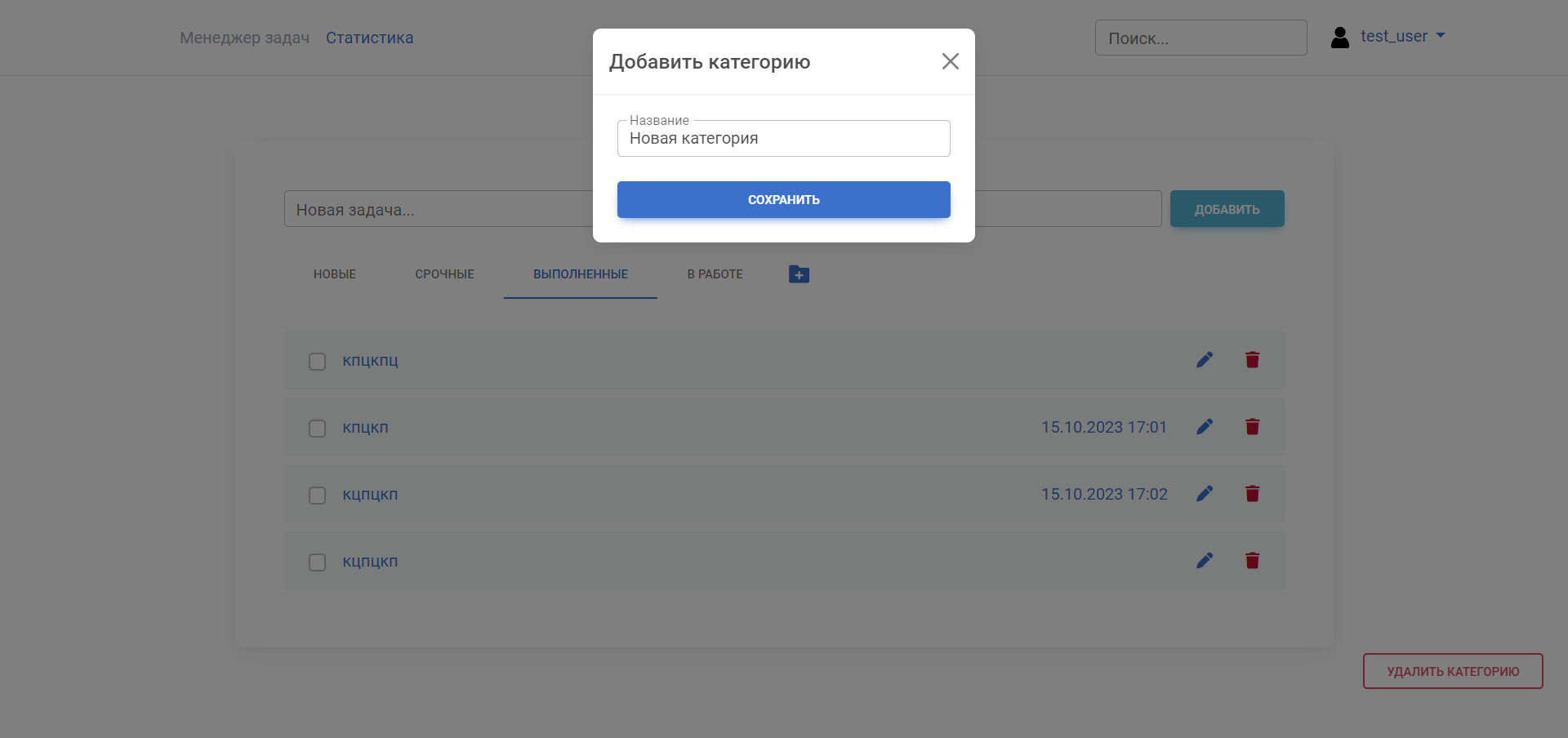


Рисунок 12. Добавление новой категории

Чтобы просмотреть статистику задач, нужно нажать по ссылке «Статистика» в левом верхнем углу (рис 13).

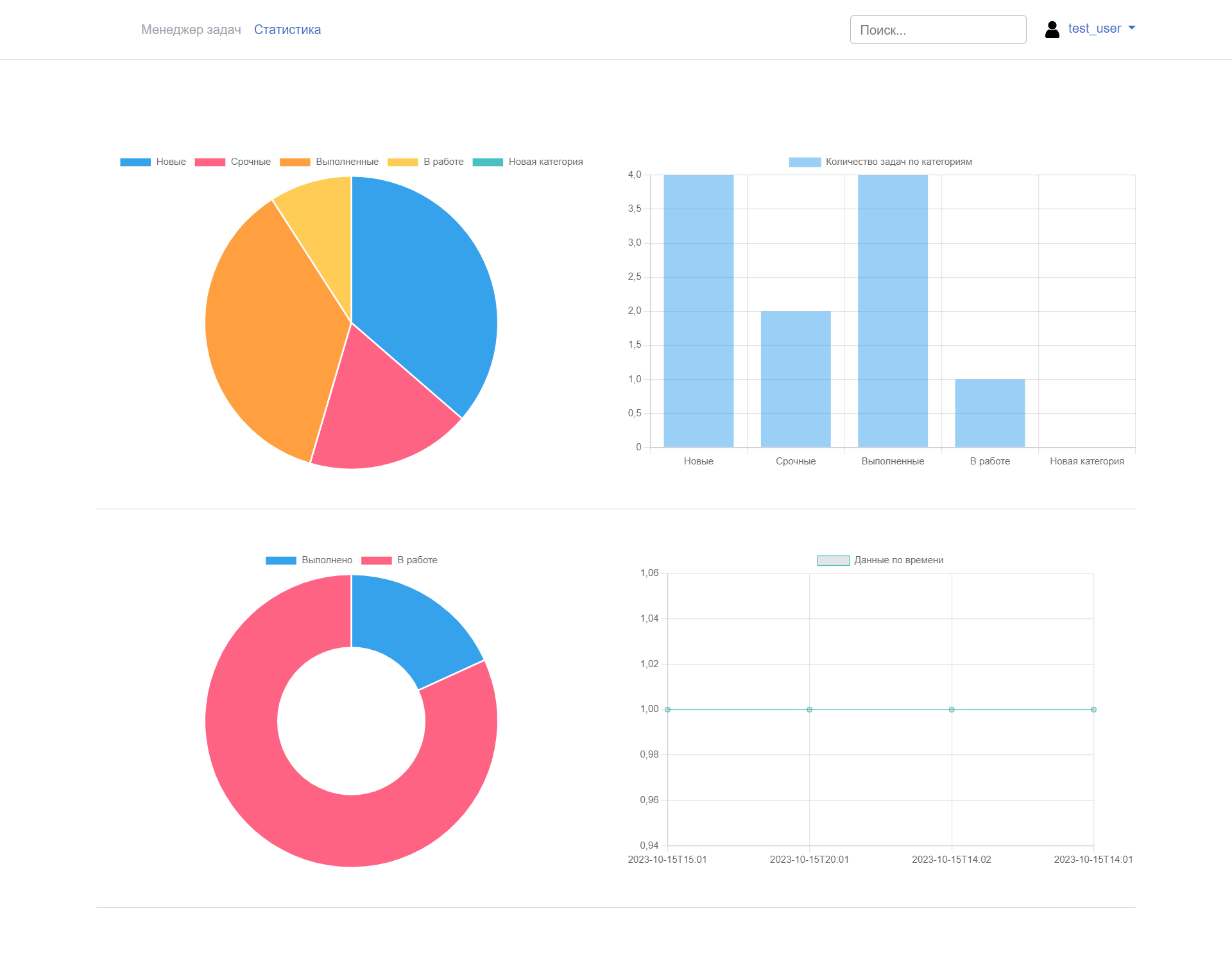


Рисунок 13. Страница статистики

# 3.3. Реализация функциональной части приложения

Для разработки функциональной части приложения были использованы язык программирования Python, фреймворк для создания веб-приложений Django, Chart.js для создания инфографики, Aiogram для отправки уведомлений в Telegram, Celery для создания фоновых задач. Листинг кода создания таблиц можно посмотреть в Приложении 2.

4. Тестирование приложения

# 4.1. Тестирование пользовательского интерфейса

Тестирование пользовательского интерфейса было проведено вручную. Были проведены тесты в различных браузерах с разными размерами экрана, которые показали, что веб-сервис достаточно адаптивен и хорошо выглядит на любых устройствах и платформах.

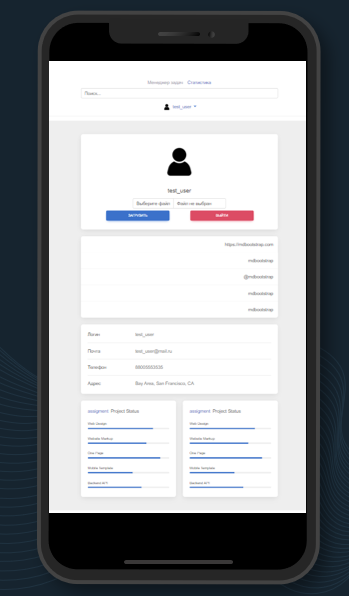
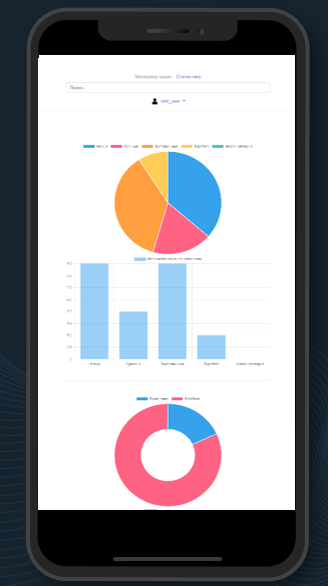
 

Рисунок 14. Веб-приложение в мобильной версии.

# 4.2. Тестирование функциональной части приложения

Тестирование функциональной части приложения было выполнено в автоматизированном режиме с помощью Selenium.

Тесты:

1. Регистрация
2. Вход
3. Изменение личных данных
4. Включение уведомлений
5. Создание задач
6. Редактирование задач
7. Удаление задач
8. Создание категории
9. Удаление категории
10. Просмотр статистики

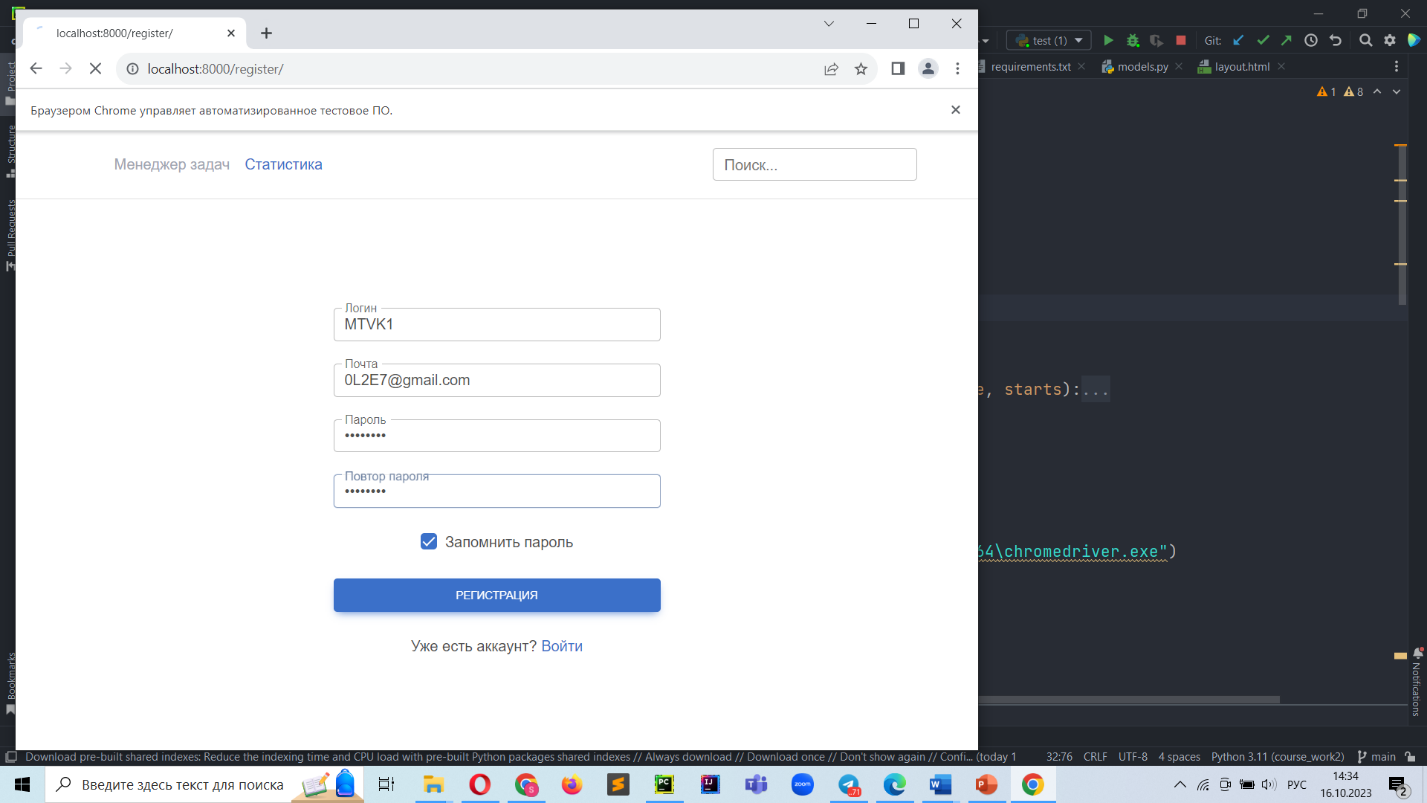


Рисунок 15. Тест регистрации

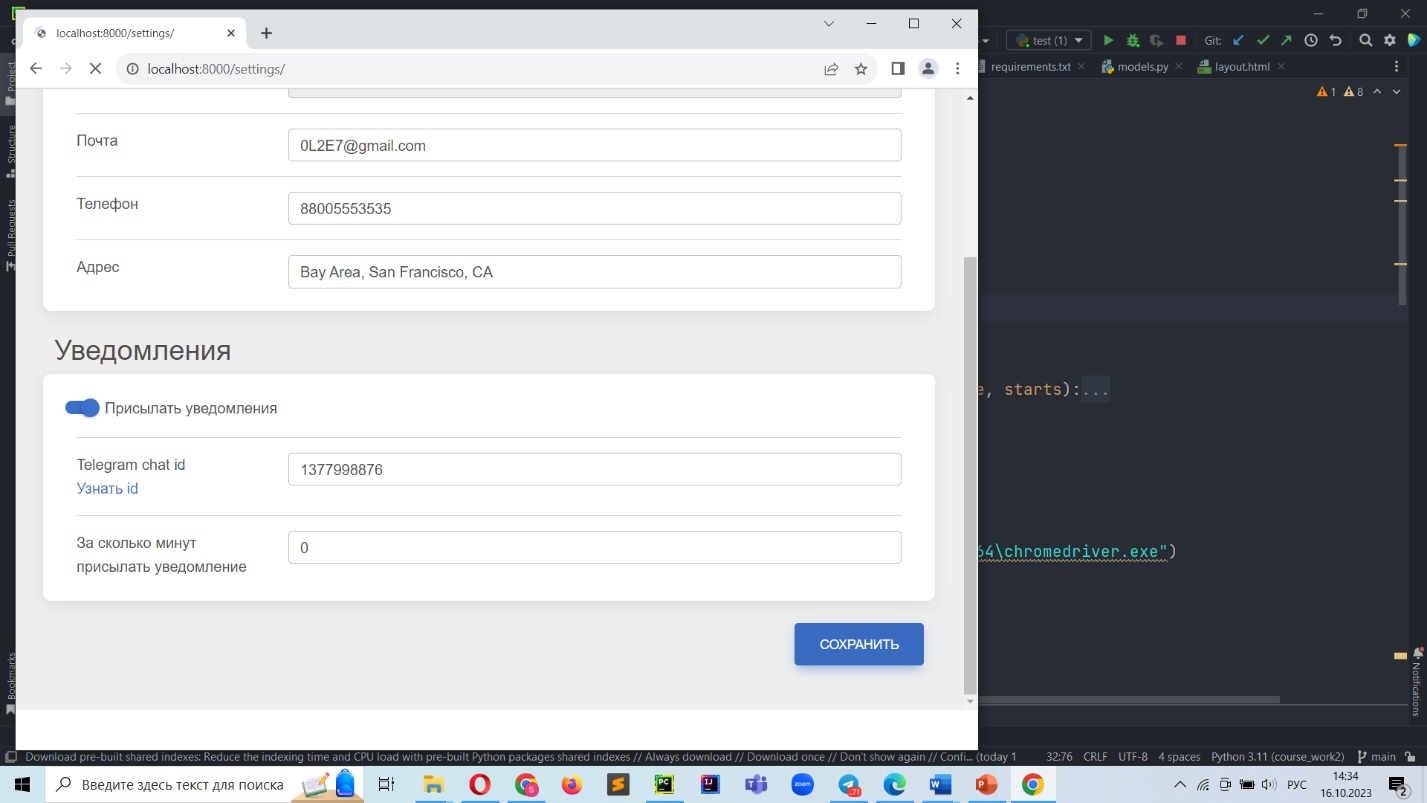


Рисунок 16. Тест включения уведомлений

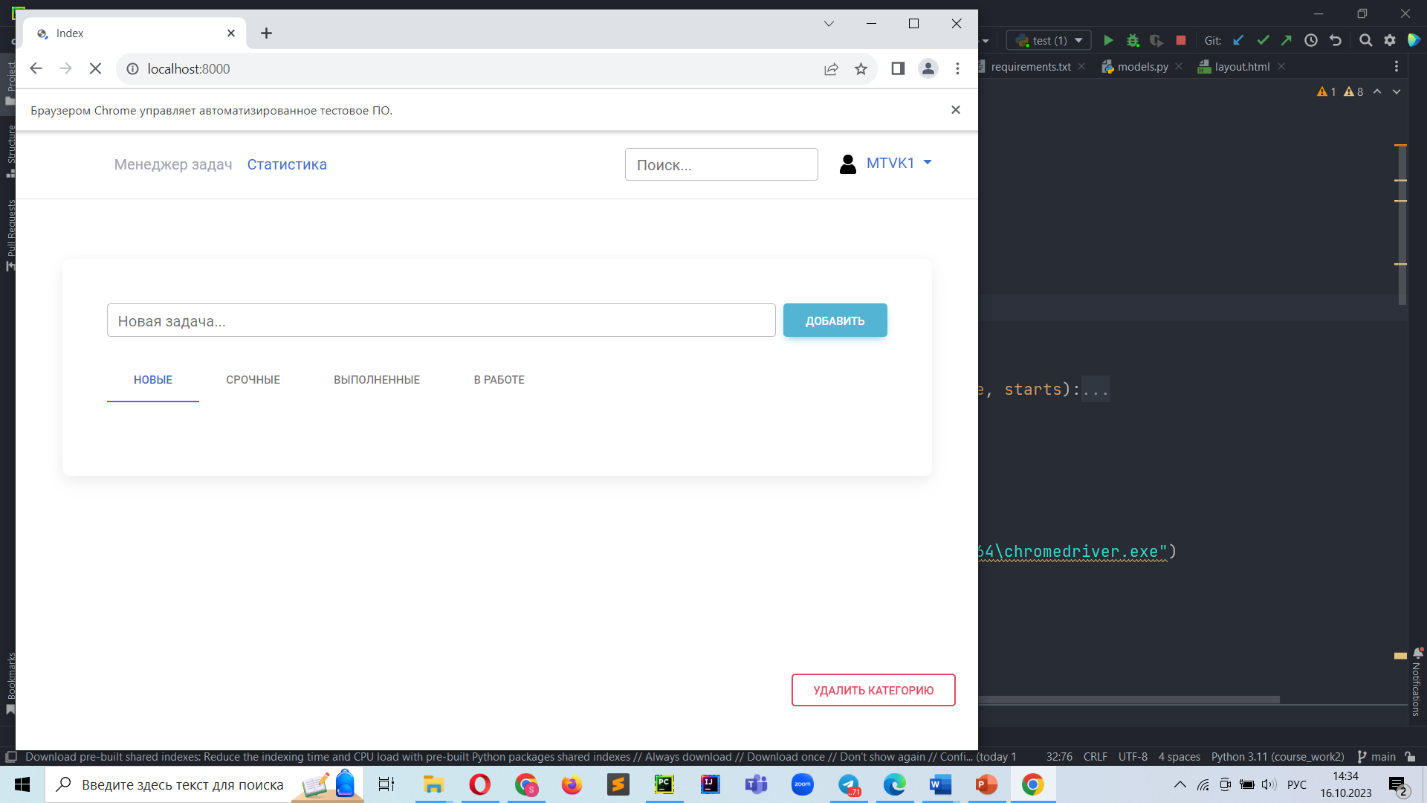
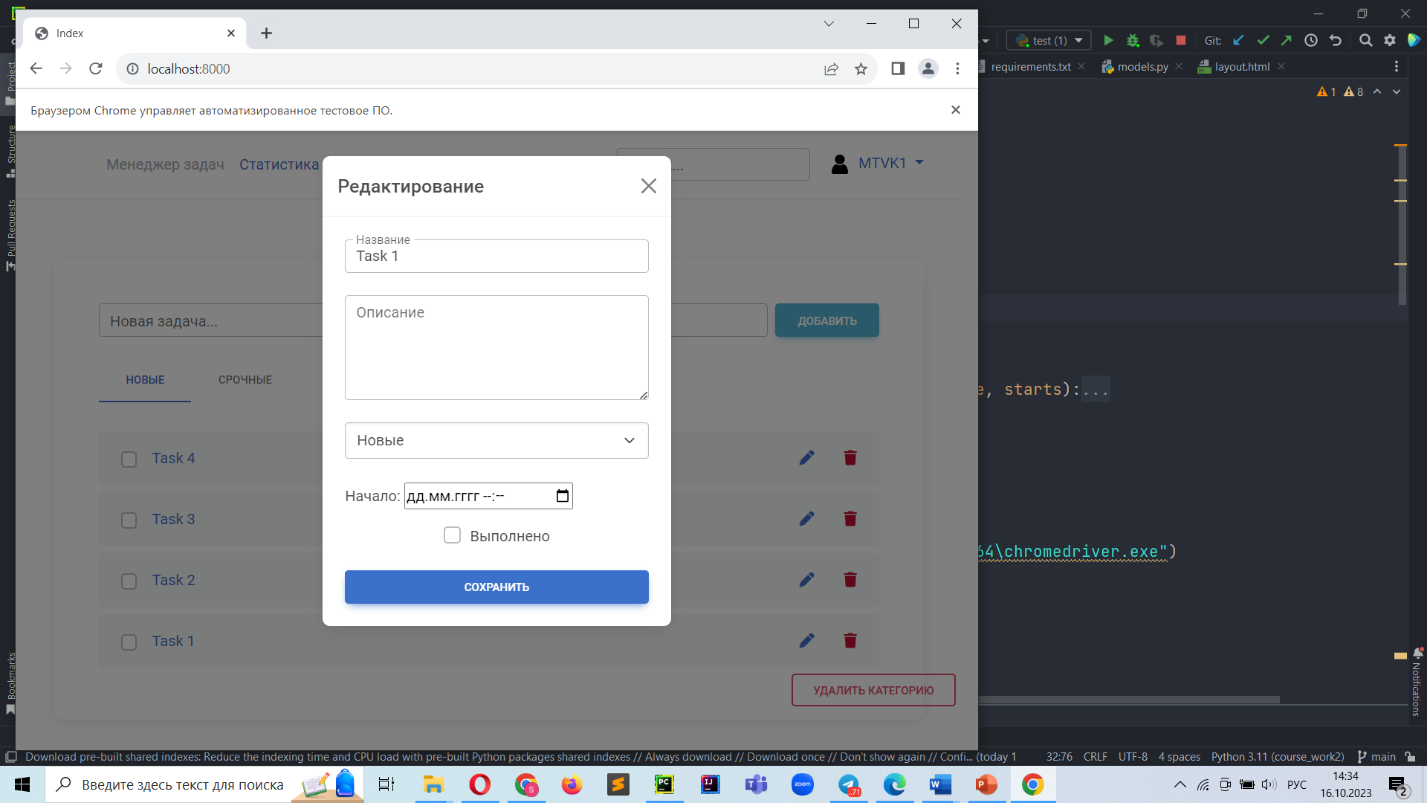
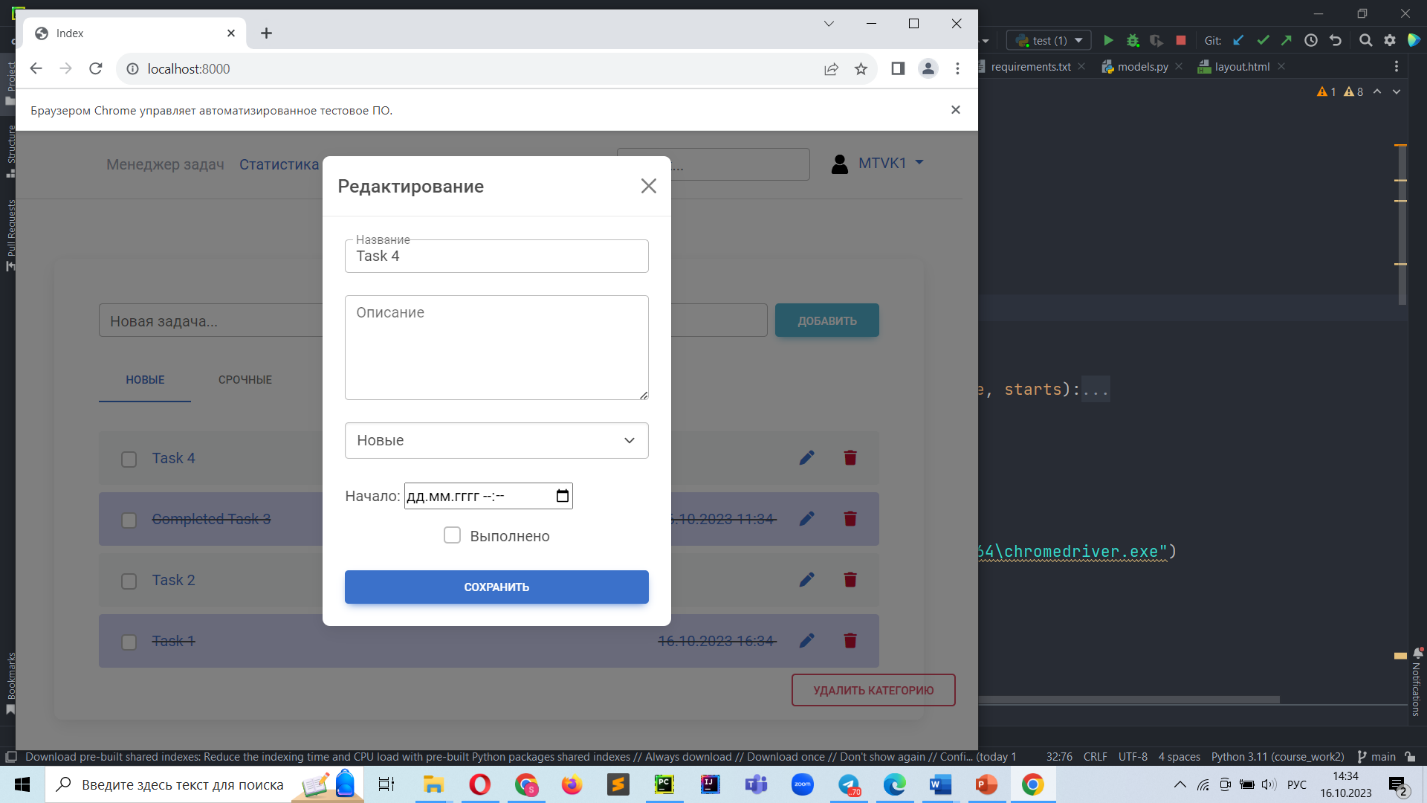


Рисунок 17. Тест главной страницы

Рисунок 18. Тест добавления задач

Рисунок 19. Тест редактирования задач

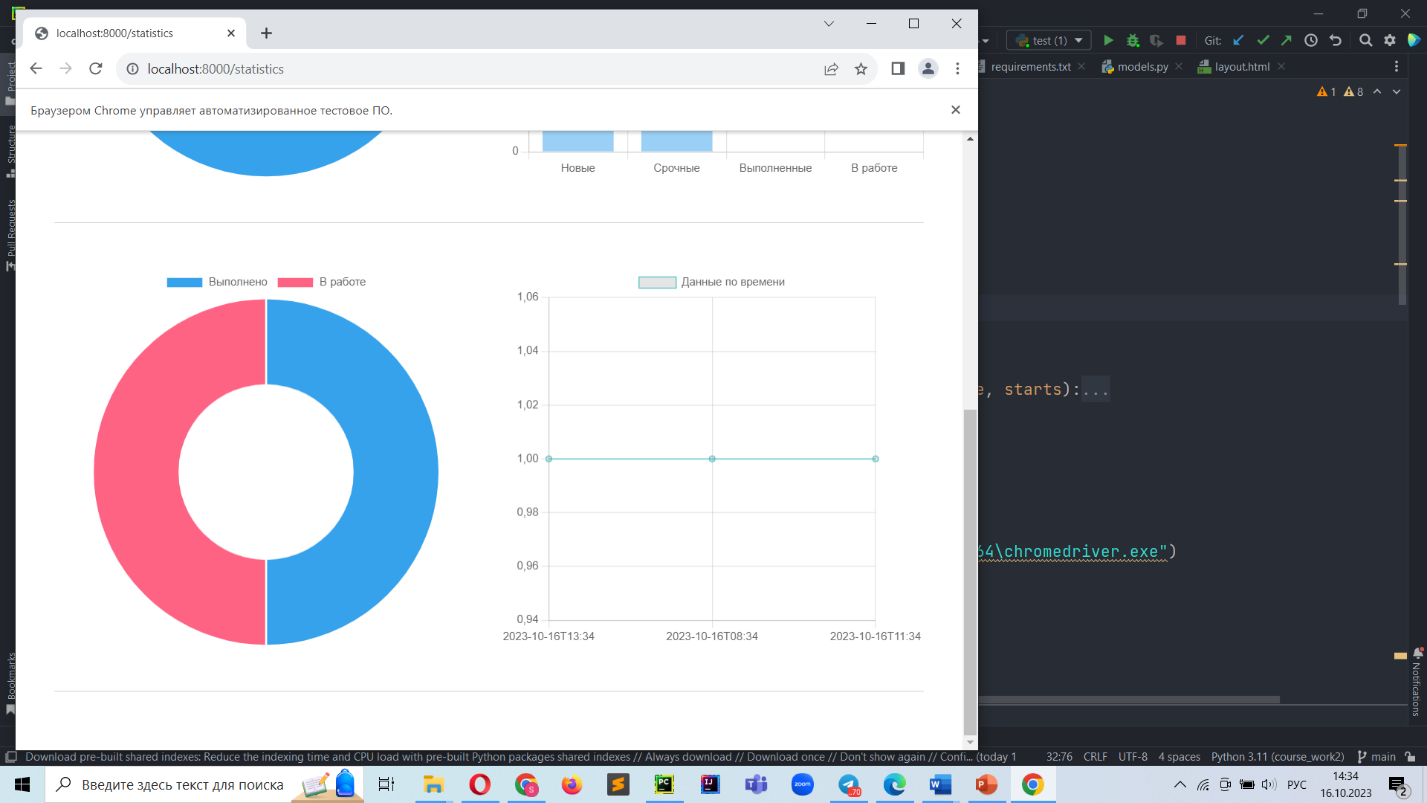


Рисунок 20. Тест просмотра статистики

Также часть тестов произведена в ручном режиме, чтобы полностью охватить те кейсы, которые не были проверены при автоматизированном тестировании. Такой метод тестирования позволяет проверить функциональность и удобство использования программного продукта с точки зрения конечного пользователя.

# Заключение

В мире, насыщенном информацией и быстрым темпом жизни, эффективное управление временем и задачами становится неотъемлемой частью нашего успеха и благополучия. В ходе исследования и разработки данной курсовой работы, мы глубоко погрузились в мир современных инструментов планирования задач, а также рассмотрели их плюсы и минусы.

В ходе курсовой работы было разработано веб-приложение для управления списком дел и задач, которое позволяет пользователям экономить и организовывать свое время.

Приложение имеет понятный и удобный и функциональный интерфейс, совместимо с разными версиями браузеров, обеспечивает безопасность и конфиденциальность данных пользователей.

Для разработки приложения были использованы следующие технологии: язык программирования Python, фреймворк Django для создания веб-приложений, Sqlite для хранения данных, MDBootstrap для стильного дизайна, Chart.js для создания красочных графиков и диаграмм, Aiogram для уведомлений в Telegram, Celery для управления фоновыми задачами (отправление уведомлений), Selenium для автоматизированного тестирования.

Приложение было протестировано на разных устройствах с разными размерами экранов и получило положительные отзывы от потенциальных пользователей. Разработанное приложение может быть использовано людьми для личного использования с целью организации времени и задач, оно может также быть использовано в рабочем пространстве и для управления проектами. Многие офисные сотрудники и фрилансеры могут воспользоваться таким приложением для повышения эффективности работы. Планировщик задач может быть использован для управления проектами в различных областях, включая информационные технологии, строительство, маркетинг, образование и другие. Он поможет распределить задачи, следить за сроками и оценивать прогресс.

# Список использованных источников

1. Официальная документация Python. [Электронный ресурс] – URL: <https://docs.python.org/> (Дата обращения: 05.01.2024)
2. Официальная документация Django. [Электронный ресурс] – URL: <https://docs.djangoproject.com/> (Дата обращения: 10.03.2024)
3. Git - О системе контроля версий. [Электронный ресурс] – URL: <https://git-scm.com/book/ru/v2/> (Дата обращения: 12.01.2024)
4. 17. Страуб Б., Чакон С. Pro Git. 2nd edition – Apress, 2014. – 440 с.
5. Руководство по SQLite. [Электронный ресурс] – URL: <https://metanit.com/sql/sqlite/> (Дата обращения: 05.04.2024)
6. Руководство по использованию SQLite в Django. [Электронный ресурс] /docs.djangoproject.com - Django - Режим доступа: https://docs.djangoproject.com/en/4.0/topics/db/sqlite3/ - Дата обращения: 15.02.2023.
7. "Создание графического интерфейса с использованием HTML, CSS и JavaScript". [Электронный ресурс] /w3schools.com - W3Schools - Режим доступа: https://www.w3schools.com/ - Дата обращения: 20.03.2023.
8. Интеграция уведомлений в веб-приложение с использованием aiogram. [Электронный ресурс] /docs.aiogram.dev - Aiogram - Режим доступа: https://docs.aiogram.dev/ - Дата обращения: 25.04.2023.
9. Создание графиков и диаграмм с использованием Chart.js. [Электронный ресурс] /chartjs.org - Chart.js - Режим доступа: https://www.chartjs.org/ - Дата обращения: 30.04.2023.
10. Документация Celery. [Электронный ресурс] /docs.celeryproject.org - Celery - Режим доступа: https://docs.celeryproject.org/ - Дата обращения: 15.04.2023.
11. Документация Selenium. [Электронный ресурс] /selenium.dev - Selenium - Режим доступа: https://www.selenium.dev/ - Дата обращения: 10.04.2023.

# Приложение 1

Листинг кода классов, представляющих объекты файловой системы

**import** hashlib  
**import** os  
**import** pickle  
**from** pathlib **import** Path  
**from** typing **import** Union  
  
**from** config **import** DATA\_FOLDER  
  
**'''  
Классы описывающие объекты файловой системы  
'''  
  
def** get\_sha1\_hash(data: bytes) -> str:  
 *# Создаем объект хэша SHA-1* sha1\_hash = hashlib.sha1()  
 *# Обновляем объект хэша с данными* sha1\_hash.update(data)  
 *# Получаем вычисленный хэш в шестнадцатеричном формате* hex\_digest = sha1\_hash.hexdigest()  
 **return** hex\_digest  
  
  
**class** Blob:  
 *'''  
 Класс для хранения двоичных данных файлов  
 Блобов может быть меньше чем файлов, если в проекте есть файлы с одинаковым содержимым  
 В этом случае их хэш будет одинаков, и файл блоба будет общий  
 '''* **def** \_\_init\_\_(self, content):  
 self.content = content  
 self.hash = get\_sha1\_hash(content)  
  
 **def** save(self, path):  
 *'''  
 Сохраняем объект в бинарном файле  
 '''* **if** self.hash **in** os.listdir(DATA\_FOLDER):  
 *# такой объект уже есть в предыдущих коммитах, не сохраняем* **return** *# сохраняем объект* **with** open(os.path.join(path, self.hash), **'wb'**) **as** file:  
 pickle.dump(self, file)  
  
  
**class** File:  
 *'''  
 Класс для хранения данных о файле  
 '''* **def** \_\_init\_\_(self, name: Path):  
 self.name = name  
 **if** os.path.exists(name) **and** os.path.isfile(name):  
 **with** open(self.name, mode=**'rb'**) **as** file:  
 file\_content = file.read()  
 self.blob = Blob(file\_content)  
 self.name = self.name.name  
 self.hash = get\_sha1\_hash((str(self.name) + self.blob.hash).encode(**'utf-8'**))  
 **else**:  
 **raise** FileNotFoundError(**f"File '{**name**}' does not exist."**)  
  
 **def** save(self, path):  
 *'''  
 Сохраняем файл в бинарном файле, заменяя блоб его хэшем  
 Блоб сохраняем как отдельный файл  
 '''* **if** self.hash **in** os.listdir(DATA\_FOLDER):  
 *# такой объект уже есть в предыдущих коммитах, не сохраняем* **return** *# сохраняем объект* self.blob.save(path)  
 self.blob = self.blob.hash  
 **with** open(os.path.join(path, self.hash), **'wb'**) **as** file:  
 pickle.dump(self, file)  
  
 **def** load(self):  
 *'''  
 Загружаем блоб из файловой системы* **:return***:  
 '''* blob\_path = os.path.join(DATA\_FOLDER, self.blob)  
 **if not** os.path.exists(blob\_path):  
 **raise** Exception(**f'Нет такого элемента {**self.blob**}'**)  
 self.blob = load(blob\_path)  
  
  
**class** Tree:  
 **def** \_\_init\_\_(self, name: Path):  
 self.name = name  
 **if** os.path.exists(name) **and** os.path.isdir(name):  
 self.children = []  
 self.\_hash = **None** self.name = self.name.name  
 **else**:  
 **raise** FileNotFoundError(**f"Directory '{**name**}' does not exist."**)  
  
 **def** add\_child(self, child: Union[File, **'Tree'**]):  
 self.children.append(child)  
  
 @property  
 **def** hash(self):  
 data = str(self.name)  
 **for** child **in** self.children:  
 data += child.hash  
 **return** get\_sha1\_hash(data.encode(**'utf-8'**))  
  
 **def** print\_tree(self):  
 **def** iterate\_tree(node, level=0):  
 print(**'│ '** \* (level - 1), **'│── ' if** level > 0 **else ''**, str(node.name), sep=**''**)  
 **if** type(node) == Tree:  
 **for** child **in** node.children:  
 iterate\_tree(child, level + 1)  
  
 iterate\_tree(self)  
  
 **def** get\_filenames(self, node=**None**, level=0, prev\_path=**''**):  
 **if** node **is None**:  
 node = self  
 **if** isinstance(node, Tree):  
 prev\_path = os.path.join(prev\_path, node.name)  
 **for** child **in** node.children:  
 **yield from** self.get\_filenames(child, level + 1, prev\_path)  
 **elif** isinstance(node, File):  
 **yield** os.path.join(prev\_path, node.name)  
  
 **def** save(self, path):  
 *'''  
 Сохраняем дерево в бинарном файле, элементы дерева заменяем хэшами  
 и сохраняем как отдельные файлы.  
 Сохраняем только если такого хэша еще не было в пред коммитах* **:param** *path: путь к папке куда сохранять* **:return***:  
 '''* hash = self.hash *# хэш меняется после изменения self.children* **if** hash **in** os.listdir(DATA\_FOLDER):  
 *# такой объект уже есть в предыдущих коммитах, не сохраняем* **return** *# сохраняем дерево* **for** i **in** range(len(self.children)):  
 child\_hash = self.children[i].hash  
 self.children[i].save(path)  
 self.children[i] = child\_hash  
 **with** open(os.path.join(path, hash), **'wb'**) **as** file:  
 pickle.dump(self, file)  
  
 **def** load(self):  
 *'''  
 Загружаем компоненты дерева из файловой системы (деревья, файлы и тд)* **:return***:  
 '''* **for** i **in** range(len(self.children)):  
 child\_path = os.path.join(DATA\_FOLDER, self.children[i])  
 **if not** os.path.exists(child\_path):  
 **raise** Exception(**f'Нет такого элемента {**self.children[i]**}'**)  
 self.children[i] = load(child\_path)  
 self.children[i].load()  
  
  
**class** Commit:  
 **def** \_\_init\_\_(self, tree: Tree, parent\_hash: str = **None**):  
 self.tree = tree  
 self.parent\_hash = parent\_hash  
 **if** self.parent\_hash **is None**:  
 self.hash = get\_sha1\_hash(self.tree.hash.encode(**'utf-8'**))  
 **else**:  
 self.hash = get\_sha1\_hash((self.tree.hash + self.parent\_hash).encode(**'utf-8'**))  
 *# self.author  
 # self.message  
 # self.time* **def** save(self, path):  
 *'''  
 Сохраняем объект в бинарном файле, заменяем дерево хэшем,  
 дерево сохраняем отдельным файлом  
 '''* tree\_hash = self.tree.hash  
 self.tree.save(path)  
 self.tree = tree\_hash  
 **with** open(os.path.join(path, self.hash), **'wb'**) **as** file:  
 pickle.dump(self, file)  
  
 **def** load(self):  
 *'''  
 Загружаем компоненты коммита из файловой системы (деревья, файлы и тд)* **:return***:  
 '''  
 # Ищем дерево* tree\_path = os.path.join(DATA\_FOLDER, self.tree)  
 **if not** os.path.exists(tree\_path):  
 **raise** Exception(**f'Нет такого дерева {**self.tree**}'**)  
  
 self.tree = load(tree\_path)  
 self.tree.load()  
  
  
**def** load(filename):  
 *'''  
 Загружаем объект из бинарного файла  
 '''* **with** open(filename, **'rb'**) **as** f:  
 loaded\_obj = pickle.load(f)  
 **return** loaded\_obj

# Приложение 2

Листинг кода получения списка изменений с предыдущим коммитом

**def** status(path=BASE\_PATH, old\_commit\_hash=**None**, new\_commit\_hash=**None**, gitignore=**True**):  
 *'''  
 Возвращает список кортежей изменений между old\_commit\_hash и new\_commit\_hash.  
 ('+', <filename>) добавление файла  
 ('-', <filename>) удаление файла  
 ('?', <filename>) изменение файла  
 ('?', <filename>, '>', > <new\_filename>) переименование файла* **:param** *path: путь по которому проверяется статус* **:param** *old\_commit\_hash: из какого коммита нужно сделать new\_commit\_hash* **:param** *new\_commit\_hash: коммит в который должны перейти  
 '''* changes\_list = []  
  
 **if** new\_commit\_hash **is None**: *# создаем новый коммит* new\_commit = make\_commit(path, print\_content=**False**, gitignore=gitignore)  
 **else**:  
 new\_commit = load(os.path.join(DATA\_FOLDER, new\_commit\_hash))  
 new\_commit.load()  
  
 **if** old\_commit\_hash **is None**: *# берем последний коммит* **if not** VCS\_FOLDER.exists():  
 print(**f"{**phrase[**'Репозиторий еще не создан'**][lang]**}"**)  
 **return** changes\_list  
 **elif not** HEAD\_PATH.exists():  
 print(**f"{**phrase[**'Коммитов еще не было'**][lang]**}"**)  
 **return** changes\_list  
 **else**:  
 **with** open(HEAD\_PATH, **'r'**) **as** file:  
 last\_commit\_hash = file.read()  
 prev\_commit = load(os.path.join(DATA\_FOLDER, last\_commit\_hash))  
 prev\_commit.load()  
 **else**:  
 *# prev\_commit = make\_commit(path, print\_content=False)* prev\_commit = load(os.path.join(DATA\_FOLDER, old\_commit\_hash))  
 prev\_commit.load()  
  
 *# сравниваем prev\_commit и new\_commit  
 # делаю у коммитов одинаковых родителей, чтобы если нет изменений, совпадали хэши* new\_commit = Commit(new\_commit.tree, prev\_commit.parent\_hash)  
 **if** new\_commit.hash == prev\_commit.hash:  
 print(**f"{**phrase[**'Изменений нет'**][lang]**}"**)  
 **return** changes\_list  
 **else**:  
 *# дерево проекта отличается* prev\_tree = prev\_commit.tree  
 new\_tree = new\_commit.tree  
 prev\_child\_hash = **''**.join([child.hash **for** child **in** prev\_tree.children])  
 new\_child\_hash = **''**.join([child.hash **for** child **in** new\_tree.children])  
 **if** prev\_child\_hash == new\_child\_hash:  
 changes\_list.append((**'?'**, prev\_tree.name, **'>'**, new\_tree.name))  
 **else**:  
 **if** prev\_tree.name != new\_tree.name:  
 changes\_list.append((**'?'**, prev\_tree.name, **'>'**, new\_tree.name))  
  
 **def** walk\_tree(new\_tree, prev\_tree, find\_deleted=**False**):  
 **for** new\_child **in** new\_tree.children:  
 **if** type(new\_child) == Tree:  
 new\_child\_hash = **''**.join([child.hash **for** child **in** new\_child.children])  
 **else**:  
 new\_child\_hash = new\_child.blob.hash  
  
 found = content\_changed = **False  
 for** prev\_child **in** prev\_tree.children:  
 **if** new\_child.name == prev\_child.name:  
 **if** new\_child.hash == prev\_child.hash:  
 found = **True** content\_changed = **False** *# удаляем найденный объект, чтобы он не использовался при поиске* prev\_tree.children.remove(prev\_child)  
 **else**:  
 found = **True** content\_changed = **True  
 break  
 else**:  
 **if** type(prev\_child) == Tree:  
 prev\_child\_hash = **''**.join([child.hash **for** child **in** prev\_child.children])  
 **else**:  
 prev\_child\_hash = prev\_child.blob.hash  
  
 **if** new\_child\_hash == prev\_child\_hash:  
 **if not** find\_deleted:  
 changes\_list.append((**'?'**, os.path.join(\*tree\_stack, prev\_child.name), **'>'**,  
 os.path.join(\*tree\_stack, new\_child.name)))  
 found = **True** content\_changed = **False** prev\_tree.children.remove(prev\_child)  
 **break** *# предыдущая версия была найдена* **if** found **and** content\_changed:  
 *# папка изменилась* **if** type(new\_child) == Tree:  
 tree\_stack.append(new\_child.name)  
 walk\_tree(new\_child, prev\_child, find\_deleted)  
 tree\_stack.pop()  
 prev\_tree.children.remove(prev\_child)  
 *# файл изменился* **else**:  
 **if not** find\_deleted:  
 changes\_list.append((**'?'**, os.path.join(\*tree\_stack, new\_child.name)))  
 prev\_tree.children.remove(prev\_child)  
 **elif not** found:  
 **if not** find\_deleted:  
 **if** type(new\_child) == Tree:  
 changes\_list.append((**'+'**, os.path.join(\*tree\_stack, new\_child.name)))  
 **else**:  
 changes\_list.append((**'+'**, os.path.join(\*tree\_stack, new\_child.name)))  
 **else**:  
 **if** type(new\_child) == Tree:  
 changes\_list.append((**'-'**, os.path.join(\*tree\_stack, new\_child.name)))  
 **else**:  
 changes\_list.append((**'-'**, os.path.join(\*tree\_stack, new\_child.name)))  
  
 tree\_stack = [new\_tree.name]  
 prev\_tree\_copy = copy.deepcopy(prev\_tree)  
 walk\_tree(new\_tree, prev\_tree\_copy)  
 walk\_tree(prev\_tree, new\_tree, find\_deleted=**True**)  
 *#* ***TODO: объект перемещен. Также если объкт переименован и немного изменен, будет создаваться новый файл*** *#* ***и удаляться старый, хотя по сути это переименование* return** changes\_list

# Приложение 3

Листинг кода отката к предыдущему коммиту

**def** restore\_commit(commit\_hash, path=BASE\_PATH):  
 *'''  
 Восстанавливает файлы из коммита* **:param** *path: куда восстановить коммит  
 '''  
 # переходим на уровень вверх если корневая папка (иначе имя проекта будет дублироваться)* **if** path == BASE\_PATH:  
 path = path.parent  
 *# удаляем все файлы которых в коммите не было, и те которые были изменены* changes\_list = status(path, new\_commit\_hash=commit\_hash)  
 **for** change **in** changes\_list:  
 **if** change[0] **in '-?'**:  
 p = os.path.join(path, change[1])  
 **try**:  
 **if** os.path.isfile(p):  
 os.remove(p)  
 **elif** os.path.isdir(p):  
 shutil.rmtree(p)  
 **else**:  
 print(**f"{**phrase[**'Не найдено для удаления'**][lang]**}:"**, p)  
 **except** FileNotFoundError **as** e:  
 print(e)  
 **for** change **in** changes\_list:  
 print(\*change)  
  
 commit = load\_commit(commit\_hash)  
 **try**:  
 os.mkdir(os.path.join(path, commit.tree.name))  
 **except** FileExistsError **as** e:  
 **pass** *# добавляем папку проекта к пути* tree\_stack = [commit.tree.name]  
  
 *# проходимся по дереву коммита и восстанавливаем файлы и папки* **def** walk\_tree(tree, tree\_stack):  
 **for** child **in** tree.children:  
 **if** isinstance(child, File):  
 **with** open(os.path.join(path, \*tree\_stack, child.name), **'wb'**) **as** f:  
 f.write(child.blob.content)  
 **elif** isinstance(child, Tree):  
 **try**:  
 os.mkdir(os.path.join(path, \*tree\_stack, child.name))  
 **except** FileExistsError **as** e:  
 **pass** tree\_stack.append(child.name)  
 walk\_tree(child, tree\_stack)  
 tree\_stack.pop()  
  
 walk\_tree(commit.tree, tree\_stack)  
  
 *# обновляем хэш коммита в HEAD* **with** open(HEAD\_PATH, **'w'**) **as** file:  
 file.write(commit\_hash)