

**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
РАЗРАБОТКА СИСТЕМЫ 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

Аннотация	4
Abstract	5
Глоссарий	6
Введение	8
1. Анализ предметной области	9
1.1. Особенности предметной области	9
1.2. Обзор существующих решений	11
1.2.1. Git	11
1.2.2. Mercurial	13
1.3. Формирование технического задания	16
1.3.1. Предмет разработки	16
1.3.2. Архитектура приложения	18
1.3.3. Функциональные требования к клиенту VCS	19
1.3.4. Функциональные требования к серверу VCS	20
1.3.5. Нефункциональные требования	21
2. Проектирование системы контроля версий	22
2.1. Общая архитектура	22
2.2. Репозиторий	22
2.3. База данных	23
2.4. Выбор инструментов и средств разработки	24
3. Разработка системы контроля версий	25
3.1. Реализация клиента VCS	25
3.1.1. Хранение объектов файловой системы	25
3.1.2. Структура хранения файлов в репозитории	26
3.1.3. Сравнение файлов и создание патчей	27
3.1.4. Сравнение деревьев и коммитов	28
3.1.5. Отмена изменений и откат к предыдущему состоянию	29
3.1.6. Игнорирование	30
3.1.7. Русификация и выбор языка	31
3.1.8. Шифрование и дешифрование	32

3.1.9. Управление VCS через командную строку.....	33
3.2. Реализация сервера VCS	33
3.2.1. Архитектура сервера	33
3.2.2. Хранение данных на сервере	34
3.2.3. Реализация пользовательского интерфейса	35
3.3. Интеграция клиента и сервера VCS	43
3.3.1. Отправка репозитория на сервер	43
3.3.2. Загрузка удаленного репозитория с сервера на клиент.....	45
4. Тестирование приложения	47
4.1. Тестирование клиента	47
4.2. Тестирование сервера	47
Заключение	50
Список использованных источников	51
Приложение 1	52
Приложение 2	55
Приложение 3	57
Приложение 4	58
Приложение 5	59
Приложение 6	60

Аннотация

Целью данной дипломной работы является разработка современной системы контроля версий (VCS). Система включает в себя ряд уникальных функций, таких как упрощенный набор команд для снижения порога вхождения для новых пользователей, поддержка различных языков интерфейса, в том числе русского, и шифрование данных репозитория для обеспечения безопасности.

Система контроля версий состоит из двух основных частей:

- Клиентская часть - предоставляет пользователям возможность взаимодействовать с VCS через набор команд, локально, у себя на компьютере. Разработана на Python с использованием библиотек `difflib` (для сравнения последовательностей и генерации патчей) и `hashlib` (для получения hash значений)

- Серверная часть - центральное место для хранения всех репозиториев всех пользователей. Сервер управляет хранением, извлечением и обновлением данных репозиториев. Разработана на Python с помощью веб-фреймворка Django и библиотеки Bootstrap для создания современного дизайна. В качестве БД была использована SQLite.

Благодаря правильной архитектуре и низкой степени связанности между компонентами, части системы могут функционировать независимо друг от друга, в автономном режиме.

Abstract

The goal of this thesis is to develop a modern version control system (VCS). The system includes a number of unique features, such as a simplified set of commands to lower the entry barrier for new users, support for various interface languages, including Russian, and repository data encryption for security purposes.

The version control system consists of two main parts:

- Client Side: Provides users with the ability to interact with the VCS through a set of commands locally on their computer. Developed in Python using the difflib library (for sequence comparison and patch generation) and hashlib library (for generating hash values).

- Server Side: The central place for storing all repositories of all users. The server manages the storage, retrieval, and updating of repository data. Developed in Python using the Django web framework and Bootstrap library for modern design. SQLite was used as the database.

Thanks to proper architecture and low coupling between components, the parts of the system can function independently of each other in an autonomous mode.

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, комментариев 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, изменения 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 (<https://git-scm.com/>) – 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.

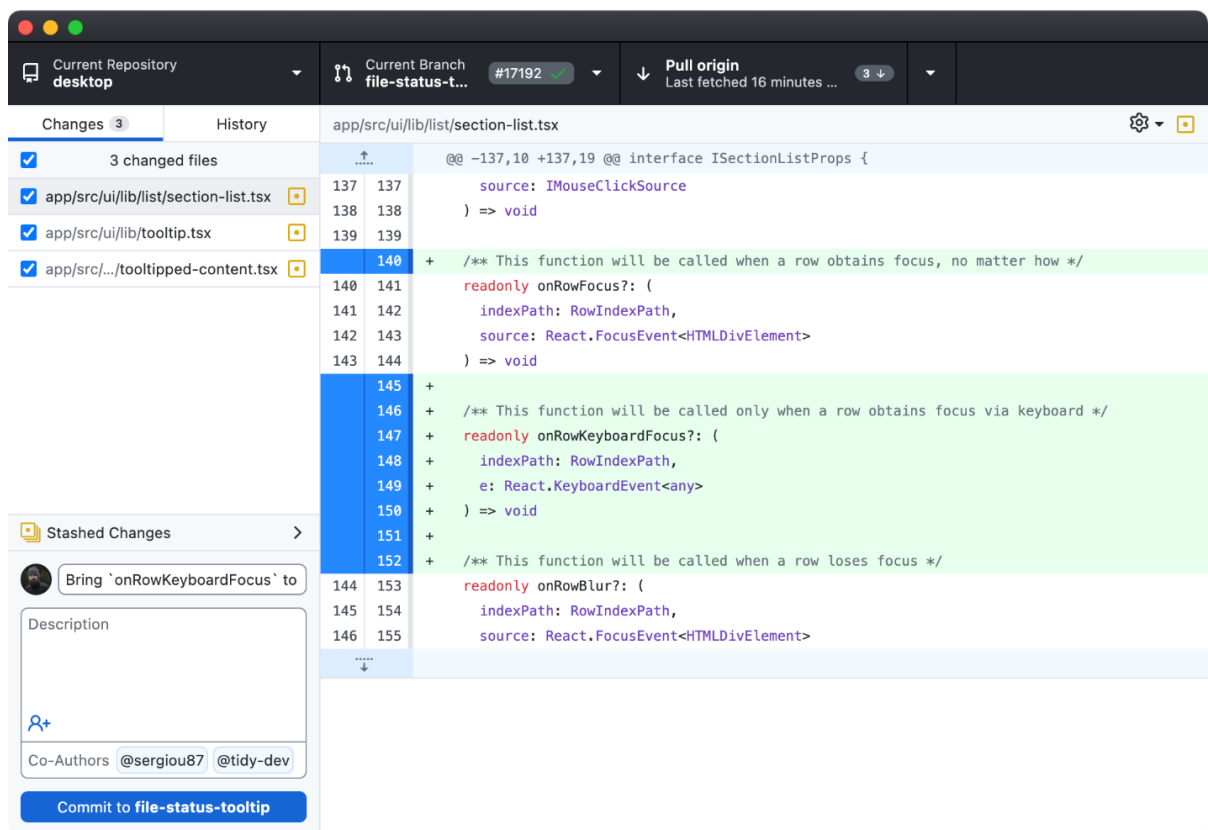


Рисунок 1. Скриншот приложения Git

Плюсы Git:

- Высокая скорость работы. Благодаря локальному хранению файлов Git обеспечивает быстрое выполнение операций отслеживания изменений, создания коммитов, слияний и прч.
- 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.

- 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:

- 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.

- Сложный пользовательский интерфейс. Несмотря на широкие возможности Git, его пользовательский интерфейс может показаться неинтуитивным для некоторых пользователей из-за разрозненности команд и параметров.

- 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.

- 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.

- 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.

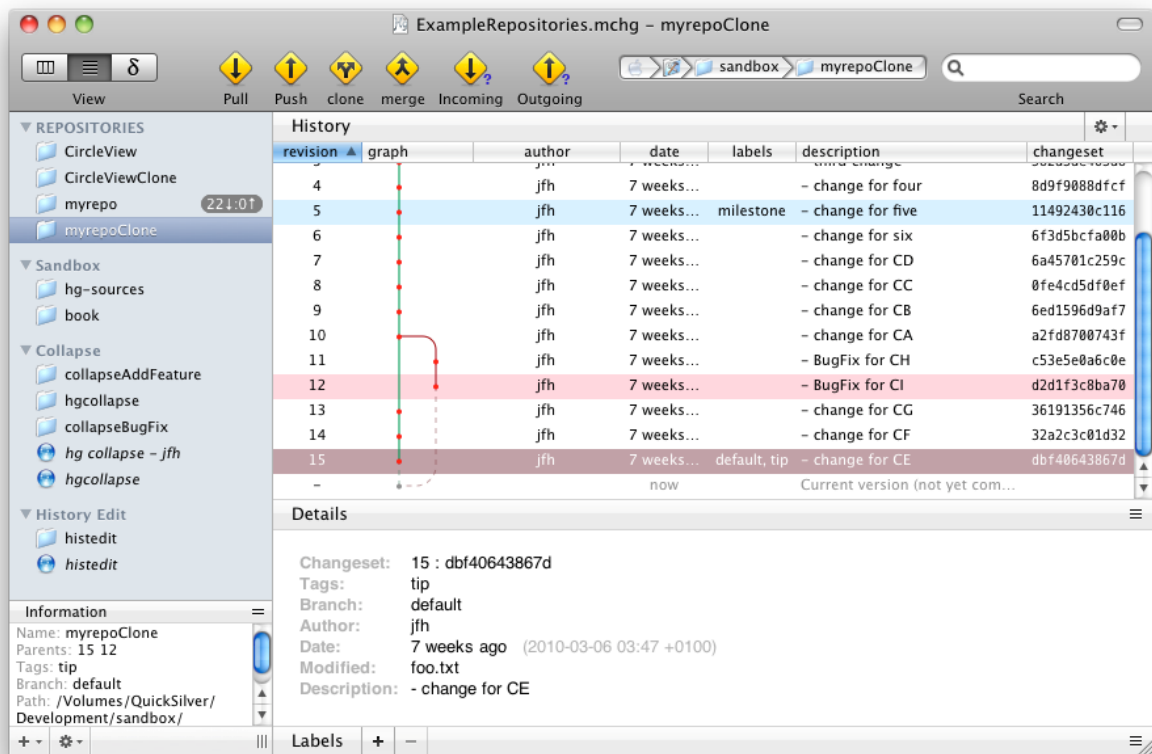


Рисунок 2. Скриншот приложения Mercurial

Плюсы Mercurial:

- Простота в освоении. Mercurial обладает более простой и интуитивно понятной концепцией, особенно для новичков. Его команды и синтаксис часто кажутся более понятными и предсказуемыми.

- Полноценная поддержка Windows. Mercurial имеет полноценную поддержку для операционной системы Windows, включая интеграцию с

командной строкой и графические интерфейсы.

- 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.

- 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:

- Меньшее сообщество и экосистема. Mercurial менее распространен и менее популярен, чем Git, что может привести к ограниченной поддержке сторонних инструментов, библиотек и расширений.

- Большой 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.

- 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 репозитория. 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 течение времени.

Назначения 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. Хранение данных в облачных хранилищах. Возможность сохранения данных репозитория на Яндекс Диске.

5. 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 [6]. 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.

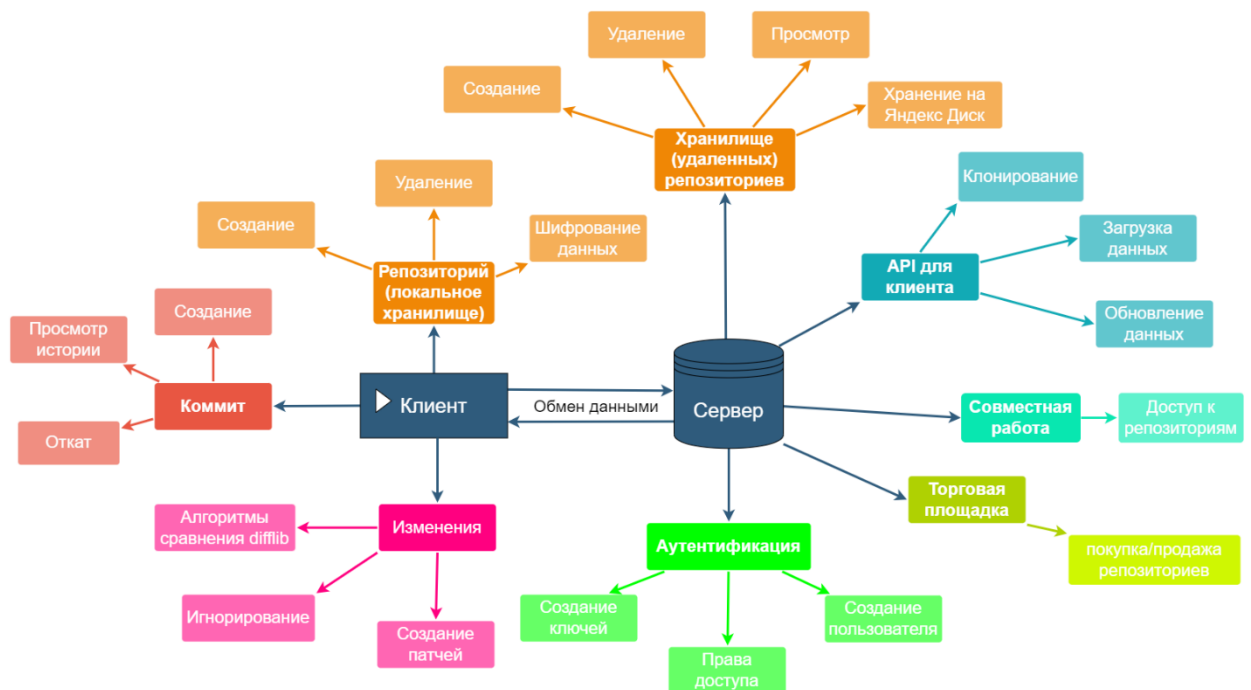


Рисунок 3. Архитектура приложения

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 репозитория, обслуживает запросы клиентов и обеспечивает доступ к данным проекта через сеть. Сервер должен быть реализован в виде веб-сервиса [7], 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 репозитория. 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 [7] 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 [8]. 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 репозиториях, 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.](#)

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.

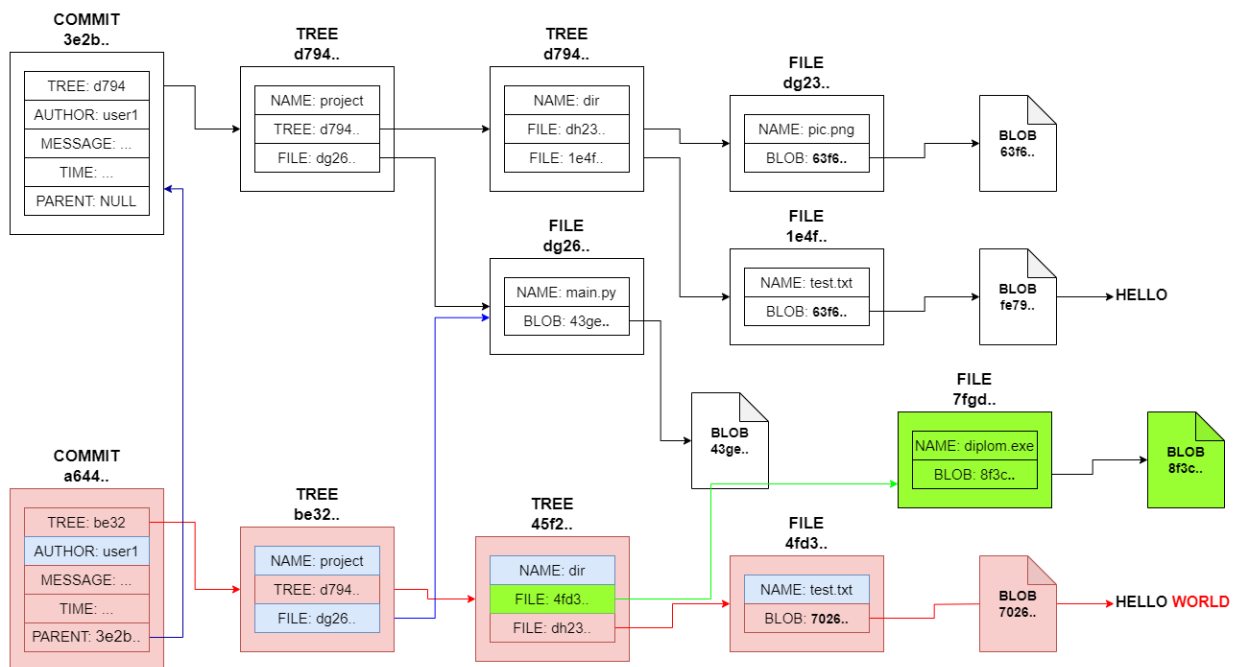
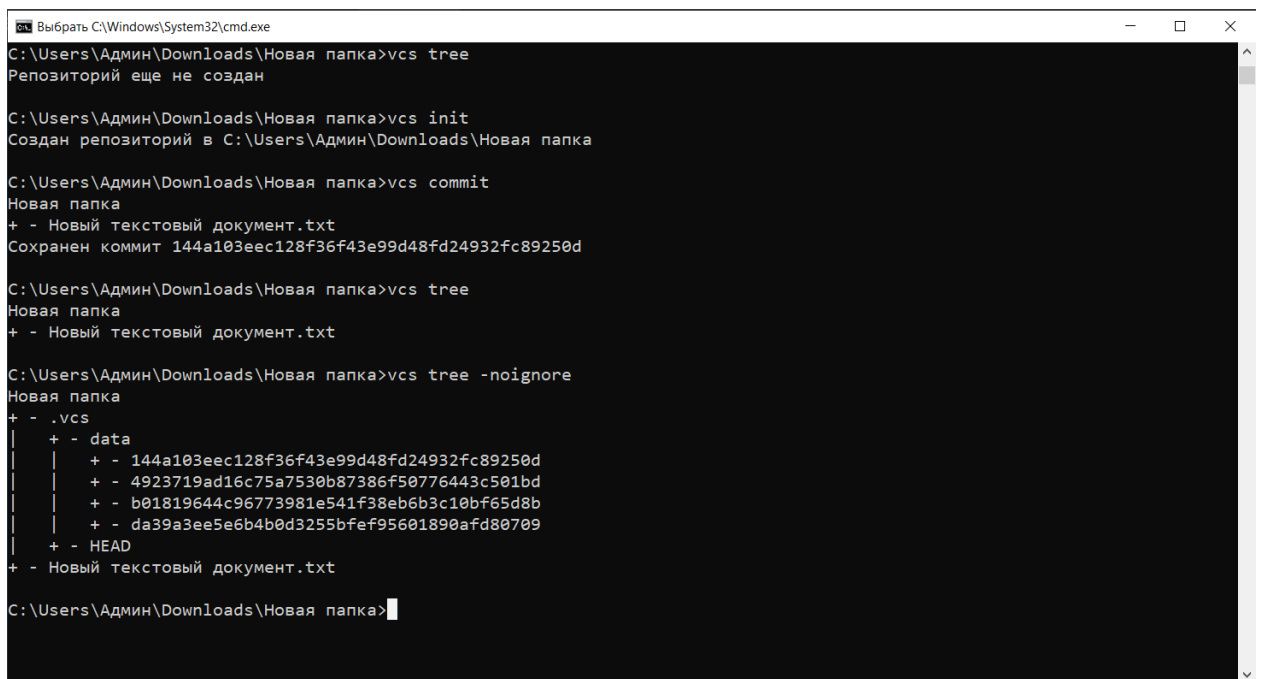


Рисунок 4. Схема репозитория

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.



```
Выбрав C:\Windows\System32\cmd.exe
C:\Users\Админ\Downloads\Новая папка>vcs tree
Репозиторий еще не создан

C:\Users\Админ\Downloads\Новая папка>vcs init
Создан репозиторий в C:\Users\Админ\Downloads\Новая папка

C:\Users\Админ\Downloads\Новая папка>vcs commit
Новая папка
+ - Новый текстовый документ.txt
Сохранен коммит 144a103eec128f36f43e99d48fd24932fc89250d

C:\Users\Админ\Downloads\Новая папка>vcs tree
Новая папка
+ - Новый текстовый документ.txt

C:\Users\Админ\Downloads\Новая папка>vcs tree -noignore
Новая папка
+ - .vcs
|   + - data
|   |   + - 144a103eec128f36f43e99d48fd24932fc89250d
|   |   + - 4923719ad16c75a7530b87386f50776443c501bd
|   |   + - b01819644c96773981e541f38eb6b3c10bf65d8b
|   |   + - da39a3ee5e6b4b0d3255bfe95601890afd80709
|   + - HEAD
+ - Новый текстовый документ.txt

C:\Users\Админ\Downloads\Новая папка>
```

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.

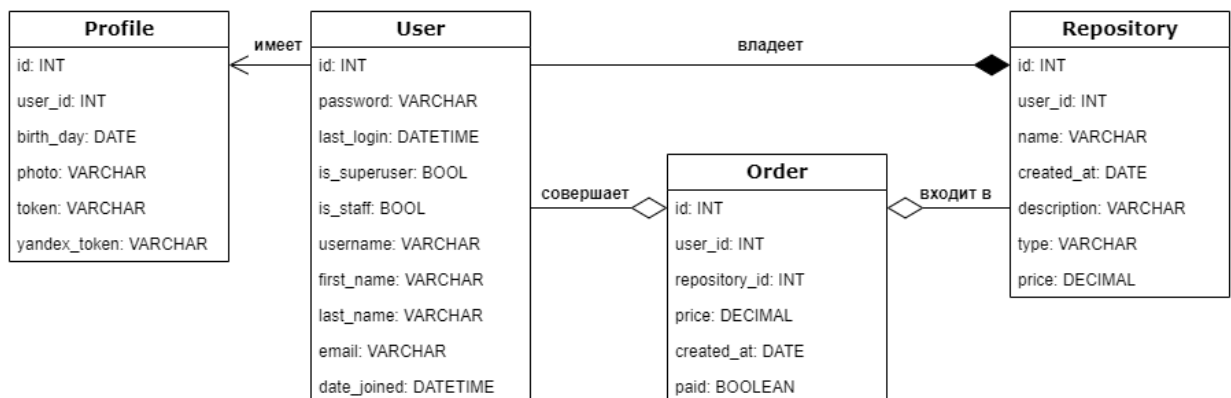


Рисунок 6. Структура БД

2.4. Выбор инструментов и средств разработки

В качестве средств для разработки были выбраны:

1. Python – основной язык программирования [1].
2. Django – фреймворк для создания веб-приложений [8].
3. SQLite – БД для хранения данных [5].
4. Diffliб – библиотека для сравнения последовательностей данных и нахождения различий между ними [1].
5. Hashlib – библиотека для генерации hash-значений данных [1].
6. Pycharm – интегрированная среда разработки (<https://www.jetbrains.com/ru-ru/pycharm/>).
7. Bootstrap – библиотека, предоставляющая набор инструментов для верстки стильных и адаптивных компонентов пользовательского интерфейса (<https://getbootstrap.com/>).
8. Dropzone.js – JavaScript библиотека, предоставляющая интерфейс для загрузки файлов на сайт с помощью drag and drop (перетаскивания) (<https://www.dropzone.dev/>).
9. Pytest [9] и Postman (<https://www.postman.com/>) – инструменты 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.

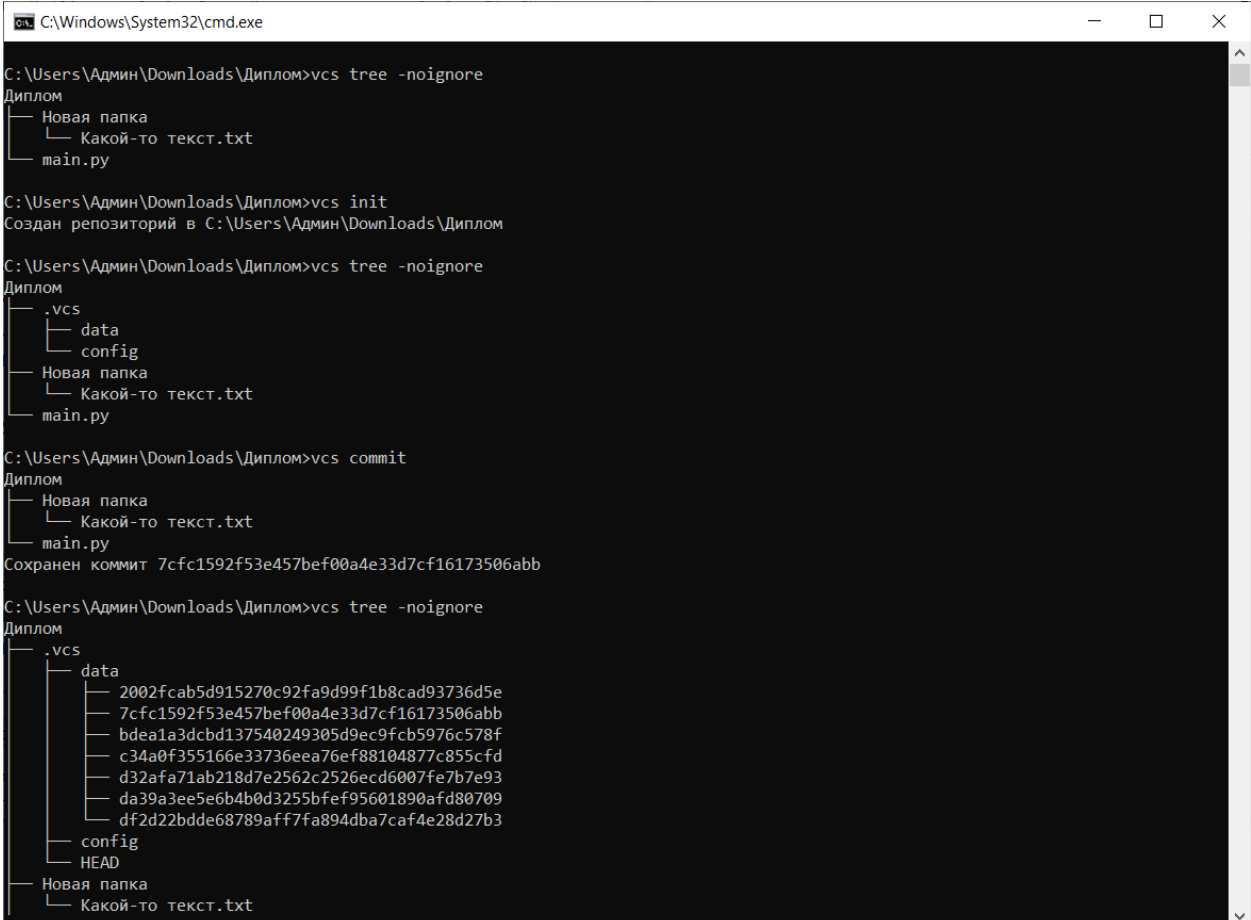
Все перечисленные классы (Blob, File, Tree, Commit) имеют методы save() и load() для сохранения в виде файла и обратной загрузки. При сохранении каждый объект записывает бинарную информацию в файл репозитория. В качестве имени файла используется его hash значение. Таким образом, если есть несколько файлов 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 хеширования SHA-1, который возвращает строку из 40 шестнадцатеричных символов (160 бит), что дает 2^{160} 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



```
C:\Windows\System32\cmd.exe

C:\Users\Админ\Downloads\Диплом>vcs tree -noignore
Диплом
├── Новая папка
│   └── Какой-то текст.txt
└── main.py

C:\Users\Админ\Downloads\Диплом>vcs init
Создан репозиторий в C:\Users\Админ\Downloads\Диплом

C:\Users\Админ\Downloads\Диплом>vcs tree -noignore
Диплом
├── .vcs
│   ├── data
│   └── config
├── Новая папка
│   └── Какой-то текст.txt
└── main.py

C:\Users\Админ\Downloads\Диплом>vcs commit
Диплом
├── Новая папка
│   └── Какой-то текст.txt
└── main.py
Сохранен коммит 7cfc1592f53e457bef00a4e33d7cf16173506abb

C:\Users\Админ\Downloads\Диплом>vcs tree -noignore
Диплом
├── .vcs
│   ├── data
│   │   ├── 2002fcab5d915270c92fa9d99f1b8cad93736d5e
│   │   ├── 7cfc1592f53e457bef00a4e33d7cf16173506abb
│   │   ├── bdea1a3dcdbd137540249305d9ec9fcb5976c578f
│   │   ├── c34a0f355166e33736eea76ef88104877c855cfd
│   │   ├── d32afa71ab218d7e2562c2526ecd6007fe7b7e93
│   │   ├── da39a3ee5e6b4b0d3255bfef95601890afd80709
│   │   └── df2d22bde68789aff7fa894dba7caf4e28d27b3
│   └── config
│       └── HEAD
├── Новая папка
│   └── Какой-то текст.txt
```

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 для того, 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 об 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 хешами. 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. Это 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. Библиотека `difflib` 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.

```
C:\Windows\System32\cmd.exe
C:\Users\Админ\Downloads\Диплом>vcs tree -noignore
Диплом
├── .vcs
│   ├── data
│   │   ├── 2002fcab5d915270c92fa9d99f1b8cad93736d5e
│   │   ├── 7cfc1592f53e457bef00a4e33d7cf16173506abb
│   │   ├── bdea1a3dcbd137540249305d9ec9fcb5976c578f
│   │   ├── c34a0f355166e33736eea76ef88104877c855cfd
│   │   ├── d32afa71ab218d7e2562c2526ecd6007fe7b7e93
│   │   ├── da39a3ee5e6b4b0d3255bfe95601890afd80709
│   │   └── df2d22bde68789aff7fa894dba7caf4e28d27b3
│   ├── config
│   └── HEAD
├── Новая папка
│   └── Какой-то текст.txt
├── main.py
└── main2.py

C:\Users\Админ\Downloads\Диплом>vcs tree -noignore
Диплом
├── .vcs
│   ├── data
│   │   ├── 2002fcab5d915270c92fa9d99f1b8cad93736d5e
│   │   ├── 7cfc1592f53e457bef00a4e33d7cf16173506abb
│   │   ├── bdea1a3dcbd137540249305d9ec9fcb5976c578f
│   │   ├── c34a0f355166e33736eea76ef88104877c855cfd
│   │   ├── d32afa71ab218d7e2562c2526ecd6007fe7b7e93
│   │   ├── da39a3ee5e6b4b0d3255bfe95601890afd80709
│   │   └── df2d22bde68789aff7fa894dba7caf4e28d27b3
│   ├── config
│   └── HEAD
├── Новая папка
│   └── Какой-то текст.txt
├── main2.py
├── новый файл.bmp
└── main.py

C:\Users\Админ\Downloads\Диплом>vcs status
? Диплом\Новая папка\Какой-то текст.txt
? Диплом\main.py > Диплом\main2.py
+ Диплом\новый файл.bmp

C:\Users\Админ\Downloads\Диплом>
```

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 посмотрим 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 проходит по дереву старого коммита и восстанавливает удаленные и измененные файлы и папки.

```
C:\Windows\System32\cmd.exe

C:\Users\Админ\Downloads\Диплом>vcs tree -noignore
Диплом
- .vcs
- data
  - 2002fcab5d915270c92fa9d99f1b8cad93736d5e
  - 7cfc1592f53e457bef00a4e33d7cf16173506abb
  - bdea1a3dcbd137540249305d9ec9fcb5976c578f
  - c34a0f355166e33736eea76ef88104877c855cfd
  - d32afa71ab218d7e2562c2526ecd6007fe7b7e93
  - da39a3ee5e6b4b0d3255bfe95601890afd80709
  - df2d22bdde68789aff7fa894dba7caf4e28d27b3
- config
- HEAD
- Новая папка
  - Какой-то текст.txt
- main.py
- новый файл.bmp

C:\Users\Админ\Downloads\Диплом>vcs status
? Диплом\Новая папка\Какой-то текст.txt
? Диплом\main.py > Диплом\main2.py
+ Диплом\новый файл.bmp

C:\Users\Админ\Downloads\Диплом>vcs hist
История коммитов:
7cfc1592f53e457bef00a4e33d7cf16173506abb

C:\Users\Админ\Downloads\Диплом>vcs rollback 7cfc1592f53e457bef00a4e33d7cf16173506abb
Сохранен коммит 244f0924eb4ed237fdd11e77484d4329c3ec9b9c
? Диплом\Новая папка\Какой-то текст.txt
? Диплом\main2.py > Диплом\main.py
- Диплом\новый файл.bmp

C:\Users\Админ\Downloads\Диплом>vcs tree
Диплом
- Новая папка
  - Какой-то текст.txt
- main.py

C:\Users\Админ\Downloads\Диплом>
```

Рисунок 9. Пример отката изменений

3.1.6. Игнорирование

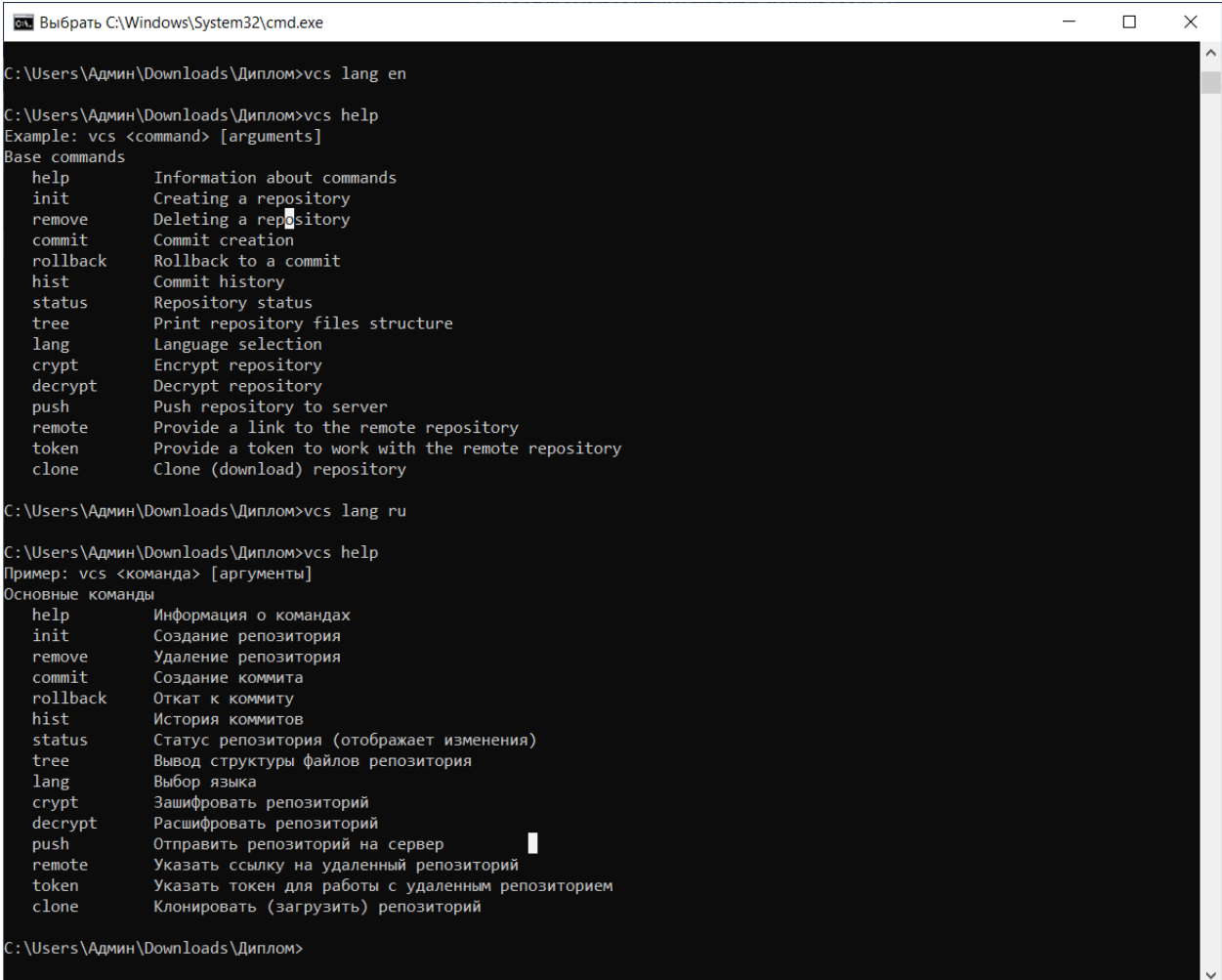
Одной из важных и полезных фишек является игнорирование файлов и папок, чтобы не отслеживать их историю. Это могут быть файлы БД, файлы с ключами и токенами доступа, которые в принципе не должны появиться в открытом доступе. Также желательно игнорирование файлов библиотек, фреймворков, служебных файлов, виртуальных сред и так далее. Я не стал придумывать новый способ игнорирования, а просто перенял старый. То есть пользователь может добавить файл `.gitignore` в любую папку проекта, и файлы и папки, перечисленные в этом файле или совпадающие с указанными там шаблонами, будут проигнорированы. Я использовал библиотеку `gitignore_parser`, которая проверяет, нужно ли игнорировать файл, и немного изменил одну из ее функций, чтобы данная библиотека поддерживала названия файлов и папок на русском языке.

Так как файлов .gitignore может быть несколько с разными уровнями вложенности, необходимо собирать все .gitignore файлы, которые находятся по пути к файлу или папке, чтобы проверить, игнорировать ее или нет.

3.1.7. Русификация и выбор языка

Для того чтобы пользователи без знания английского могли использовать VCS и не испытывать языковых сложностей, я решил добавить русификацию системы и возможность выбора языка.

На рисунке 10 можно увидеть пример переключения языка. Переведена не только справка, но и все возможные подсказки и информационные сообщения, отображающиеся у пользователя. При желании можно добавить в систему и другие языки, достаточно изменить файл с переводами.

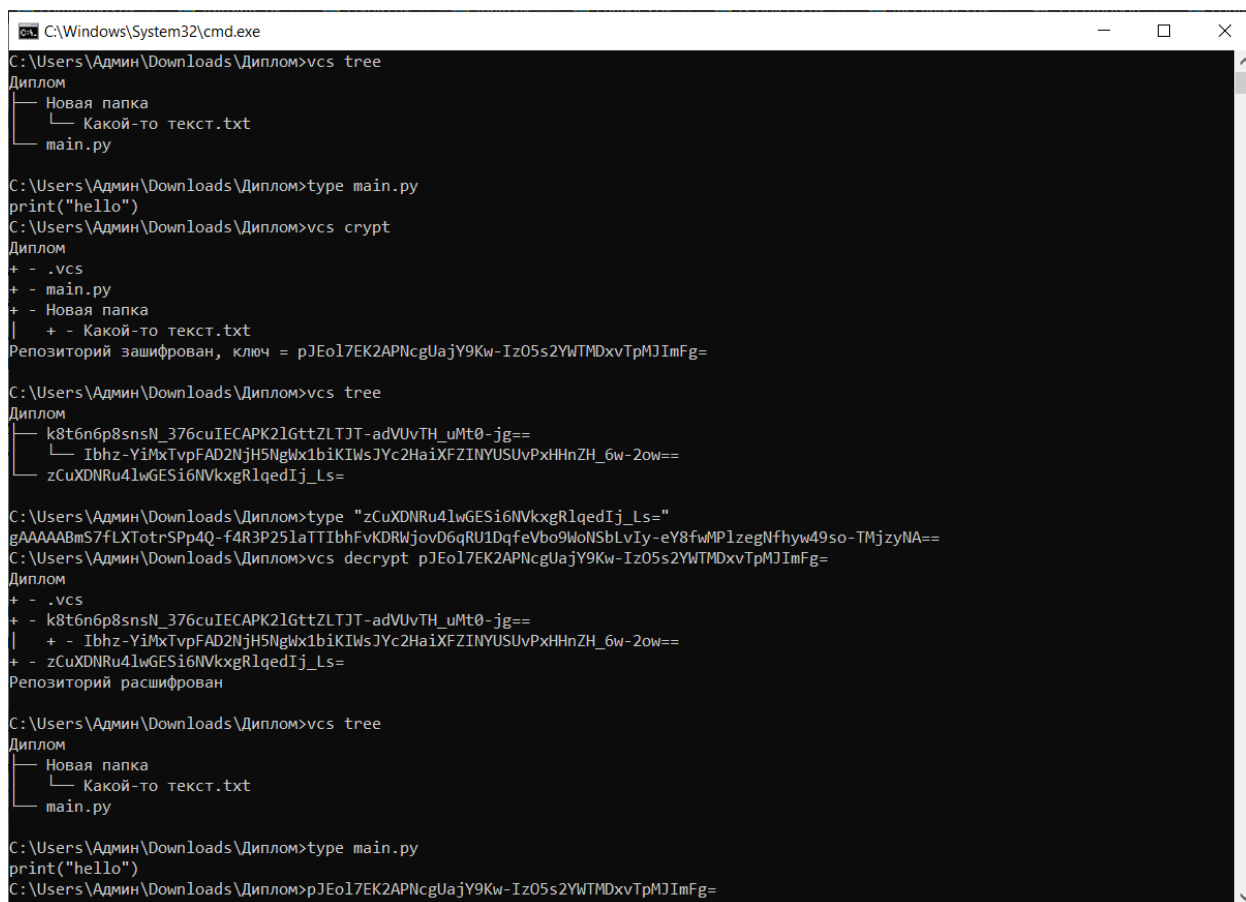


```
Выбрать C:\Windows\System32\cmd.exe
C:\Users\Админ\Downloads\Диплом>vcs lang en
C:\Users\Админ\Downloads\Диплом>vcs help
Example: vcs <command> [arguments]
Base commands
help      Information about commands
init      Creating a repository
remove    Deleting a repository
commit    Commit creation
rollback  Rollback to a commit
hist      Commit history
status    Repository status
tree      Print repository files structure
lang      Language selection
crypt     Encrypt repository
decrypt   Decrypt repository
push      Push repository to server
remote    Provide a link to the remote repository
token     Provide a token to work with the remote repository
clone     Clone (download) repository
C:\Users\Админ\Downloads\Диплом>vcs lang ru
C:\Users\Админ\Downloads\Диплом>vcs help
Пример: vcs <команда> [аргументы]
Основные команды
help      Информация о командах
init      Создание репозитория
remove    Удаление репозитория
commit    Создание коммита
rollback  Откат к коммиту
hist      История коммитов
status    Статус репозитория (отображает изменения)
tree      Вывод структуры файлов репозитория
lang      Выбор языка
crypt     Зашифровать репозиторий
decrypt   Расшифровать репозиторий
push      Отправить репозиторий на сервер
remote    Указать ссылку на удаленный репозиторий
token     Указать токен для работы с удаленным репозиторием
clone     Клонировать (загрузить) репозиторий
C:\Users\Админ\Downloads\Диплом>
```

Рисунок 10. Выбор языка

3.1.8. Шифрование и дешифрование

Для обеспечения безопасности данных пользователей я решил создать возможность шифрования файлов всего репозитория и дешифрования их при наличии ключа. В качестве алгоритма шифрования я выбрал AES (Advanced Encryption Standard) – симметричный алгоритм шифрования, который стал стандартом для шифрования данных. У данного алгоритма достаточно высокий уровень безопасности и хорошее быстродействие. Как видно на рисунке 11, помимо шифрования содержимого файлов, шифруются их имена (более коротким ключом, чтобы не превысить ограничение на длину имен файлов), что обеспечивает неплохую защиту от несанкционированного доступа. Шифрование осуществляется командой `vcs crypt`, после чего выводится сгенерированный ключ, который нужно указать при дешифровании командой `vcs decrypt`.



```
C:\Windows\System32\cmd.exe
C:\Users\Админ\Downloads\Диплом>vcs tree
Диплом
├── Новая папка
│   └── Какой-то текст.txt
└── main.py

C:\Users\Админ\Downloads\Диплом>type main.py
print("hello")

C:\Users\Админ\Downloads\Диплом>vcs crypt
Диплом
+ - .vcs
+ - main.py
+ - Новая папка
|   + - Какой-то текст.txt
Репозиторий зашифрован, ключ = pJEo17EK2APNcgUajY9Kw-Iz05s2YWTMDxvTpMJImFg=

C:\Users\Админ\Downloads\Диплом>vcs tree
Диплом
├── k8t6n6p8snsN_376cuIECAPK2lGttZLTJT-adVUvTH_uMt0-jg==
│   └── Ibhz-YiMxTvpFAD2NjH5NgWx1biKIWsJYc2HaiXFZINyUSUvPxHnZH_6w-2ow==
└── zCuXDNru4lwGESi6NVkxgRlqedIj_Ls=

C:\Users\Админ\Downloads\Диплом>type "zCuXDNru4lwGESi6NVkxgRlqedIj_Ls="
gAAAAABmS7fLXTotrSPp4Q-f4R3P25laTTIbhFvKDRWjovD6qRU1DqfeVbo9WoNSbLvIy-eY8fwMP1zegNfhyw49so-TMjzyNA==

C:\Users\Админ\Downloads\Диплом>vcs decrypt pJEo17EK2APNcgUajY9Kw-Iz05s2YWTMDxvTpMJImFg=
Диплом
+ - .vcs
+ - k8t6n6p8snsN_376cuIECAPK2lGttZLTJT-adVUvTH_uMt0-jg==
|   + - Ibhz-YiMxTvpFAD2NjH5NgWx1biKIWsJYc2HaiXFZINyUSUvPxHnZH_6w-2ow==
+ - zCuXDNru4lwGESi6NVkxgRlqedIj_Ls=
Репозиторий расшифрован

C:\Users\Админ\Downloads\Диплом>vcs tree
Диплом
├── Новая папка
│   └── Какой-то текст.txt
└── main.py

C:\Users\Админ\Downloads\Диплом>type main.py
print("hello")

C:\Users\Админ\Downloads\Диплом>pJEo17EK2APNcgUajY9Kw-Iz05s2YWTMDxvTpMJImFg=
```

Рисунок 11. Шифрование данных репозитория

3.1.9. Управление VCS через командную строку

Чтобы была возможность управлять клиентом VCS через командную строку Windows без запуска каких-либо скриптов, проделаны следующие шаги:

- Создан файл `vcs.bat`, который запускает `main.py` файл моего проекта, который, в свою очередь, обрабатывает аргументы командной строки (команды VCS и различные параметры) и выполняет необходимые действия.
- Добавлен путь к папке, в которой находится `vcs.bat`, в переменную среды `Path`.

Таким образом, при вводе команды `vcs <command>` в командную строку запускается файл `vcs.bat`, который вызывает `main.py` и передает `<command>` в качестве аргумента.

3.2. Реализация сервера VCS

3.2.1. Архитектура сервера

В качестве веб-фреймворка, я выбрал Django. Этот фреймворк обладает множеством встроенных функций, которые позволяют быстро разрабатывать веб-приложения, например, в Django есть встроенная система аутентификации и авторизации, которую я успешно использую в своем сервере. Помимо этого, есть множество других полезных функций, для работы с учетными записями пользователей и обеспечения безопасности от XSS, CSRF атак и SQL-инъекций.

Построение архитектуры приложения на Django требует продуманного подхода, чтобы обеспечить масштабируемость, гибкость и простоту изменения существующих модулей. Основным аспектом является разделение проекта на приложения, что позволяет легко управлять кодом и повторно использовать компоненты в других проектах, а также уменьшить зависимость одного модуля от другого.

В своем проекте я создал два приложения – `account` (все, что связано с аккаунтами, аутентификацией, авторизацией, изменением профиля, выходом)

и repository (все, что связано с репозиториями и данными в них).

Внутри приложений используется архитектура MTV (Model-Template-View), которая способствует организации кода и логики веб-приложений. Она обеспечивает четкое разделение обязанностей, что способствует упрощению разработки, тестирования и поддержки кода.

Модели (models.py) представляют данные и логику, связанную с данными, работой с БД, созданием таблиц и т.д.

Шаблоны (templates) отвечают за представление данных. Они определяют, как информация отображается пользователю. В моем случае это набор html файлов.

Представления (views.py) обрабатывают запросы и возвращают ответы. Они связывают модели и шаблоны, обеспечивая логику, необходимую для обработки запросов и отображения данных.

3.2.2. Хранение данных на сервере

Для хранения данных о пользователях я использовал встроенную модель User, которая хранит имя пользователя, пароль (в хешированном виде), почту, роль и прочие основные сведения. Но было необходимо также хранить токен, который будет использоваться для отправки запросов с клиента, для аутентификации, фото профиля и личную информацию. Для хранения этих данных я создал модель Profile, которая связана с User отношением один к одному, и создается вместе с созданием пользователя. Структура БД представлена на рисунке 6.

Для хранения имени, описания и владельца репозитория я создал модель Repository. Но данные самих репозиториях, файлы и папки проблематично хранить обычной БД, так как у этих файлов может быть большой объем и доступ к большому количеству таких данных может быть медленным. Также, моделирование структуры файловой системы внутри БД достаточно сложная задача. Исходя из этого, я принял решение хранить данные репозиториях непосредственно на сервере, в папке media проекта Django. Для удобства

навигации все репозитории хранятся по следующему пути: `media/files/<username>/<repository_name>`, где `username` – имя пользователя (уникально среди пользователей) и `repository_name` – имя репозитория (уникально среди репозиторий данного пользователя). На самом сервере для доступа к репозиторию используется аналогичный URL адрес: `<hostname>/<username>/<repository_name>`, `hostname` – имя сервера.

3.2.3. Реализация пользовательского интерфейса

Для создания структуры веб-страниц (расположение элементов, формы ввода, кнопки и прочее) использован HTML. Для придания современного и стильного вида, я использовал готовые классы и шаблоны Bootstrap, а также добавил немного собственных стилей с помощью CSS.

Для динамических элементов (выпадающие списки, раскрывающиеся папки, смена темы) был использован JavaScript. Для создания интерфейса загрузки файлов и самой отправки файлов на сервер была использована JS библиотека Dropzone.js. Также для корректной загрузки целых папок с файлами и подкаталогами пришлось написать скрипт на JavaScript, который сохраняет относительный путь каждого файла и отправляет отдельным параметром в POST запросе. Без этого не получить относительный путь файла (и не создать правильную файловую структуру проекта), так как браузер и сам Django очищают все, кроме имени файла, перед отправкой, для обеспечения безопасности. Листинг формы для загрузки файлов через браузер в Приложении 4.

Далее будет идти перечисление страниц, которые есть на сервере, их скриншоты и краткое описание.

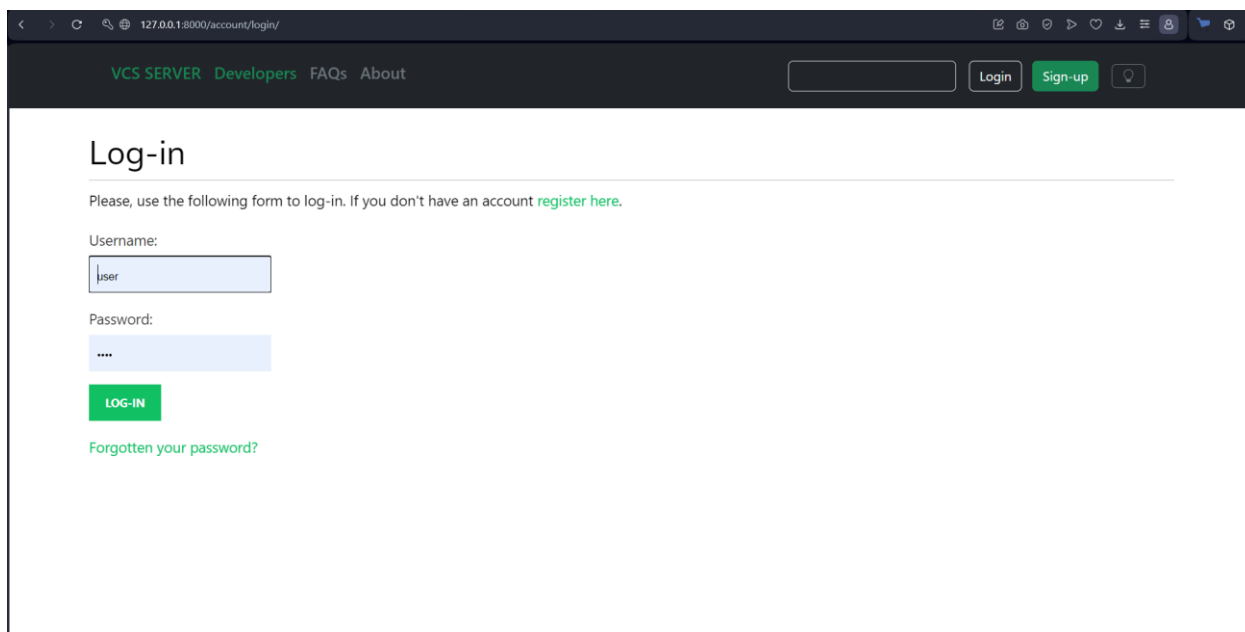


Рисунок 12. Страница входа

Как уже написано выше, для страницы входа и выхода использована встроенная механика аутентификации и авторизации. То есть мне не нужно было создавать для них представления (views.py) в Django, я лишь создал нужные формы (forms.py) и шаблоны (templates). На рисунке 12 показана страница входа. Пользователь может входить не только по имени, но и по почте, поскольку она уникальна.

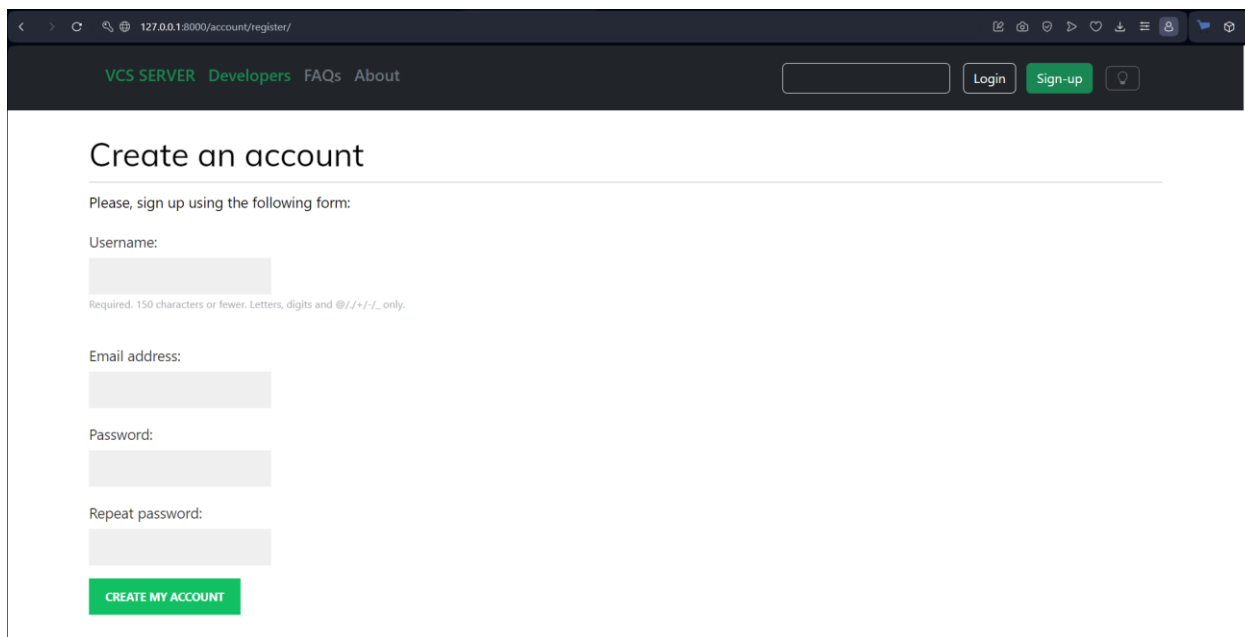


Рисунок 13. Страница регистрации

На рисунке 13 представлена страница регистрации на сайте. На рисунке 14 – страница, отображающаяся после выхода.

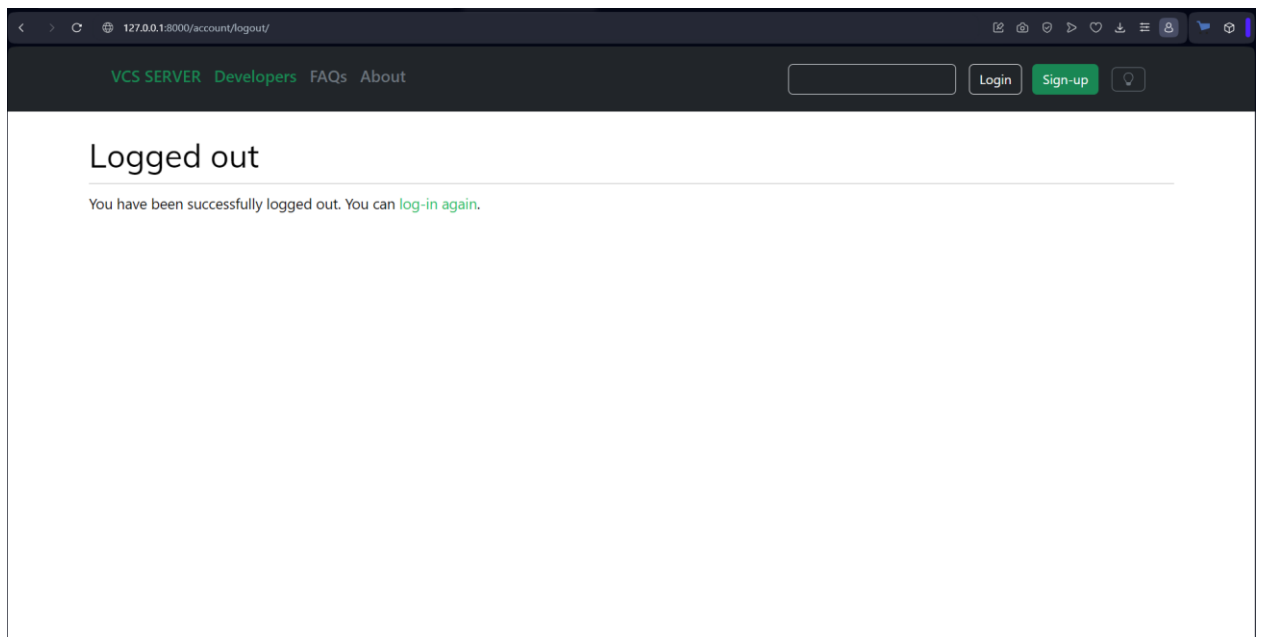


Рисунок 14. Страница выхода

На рисунке 15 показана страница редактирования профиля. Здесь пользователь может изменить свои сведения, посмотреть токен для загрузки локальных репозитиев на сервер и войти через Яндекс OAuth.

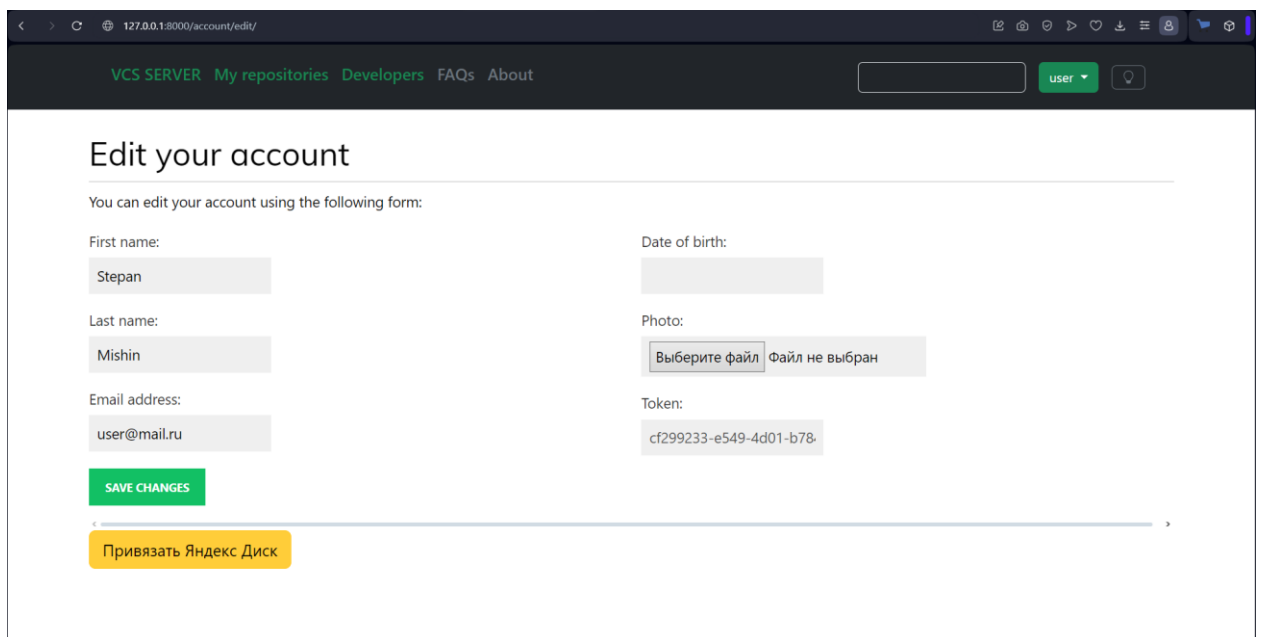


Рисунок 15. Страница редактирования профиля

На рисунке 16 показана страница изменения пароля.

127.0.0.1:8000/account/password_change/

VCS SERVER My repositories Developers FAQs About

user

Change your password

Use the form below to change your password.

Old password:

New password:

Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

New password confirmation:

CHANGE

Рисунок 16. Страница изменения пароля

На рисунке 17 показана страница сброса пароля для авторизованного пользователя.

127.0.0.1:8000/account/password_reset/

VCS SERVER Developers FAQs About

Login Sign-up

Forgotten your password?

Enter your e-mail address to obtain a new password.

Email:

SEND E-MAIL

Рисунок 17. Страница сброса пароля

На рисунке 17 показана страница сброса пароля для неавторизованного пользователя. При вводе почты ссылка с уникальным токеном для сброса пароля отправляется на почту пользователя. Пример показан на рисунке 18.

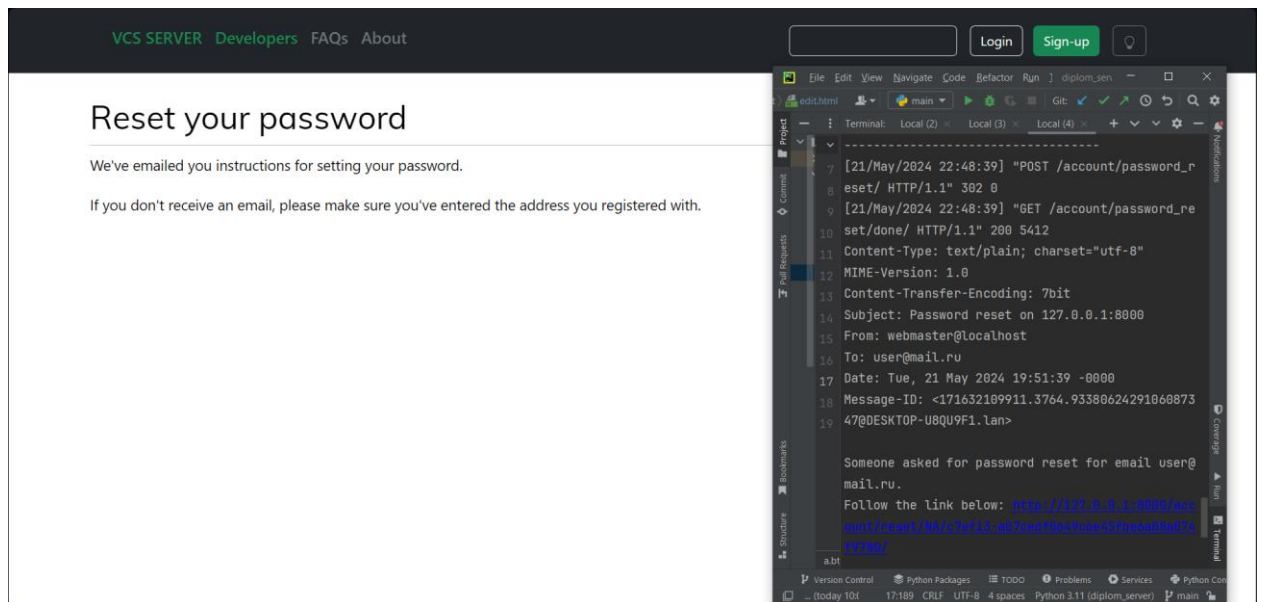


Рисунок 18. Сброса пароля с помощью почты

На рисунке 19 показана главная страница – список всех репозиториев всех пользователей.

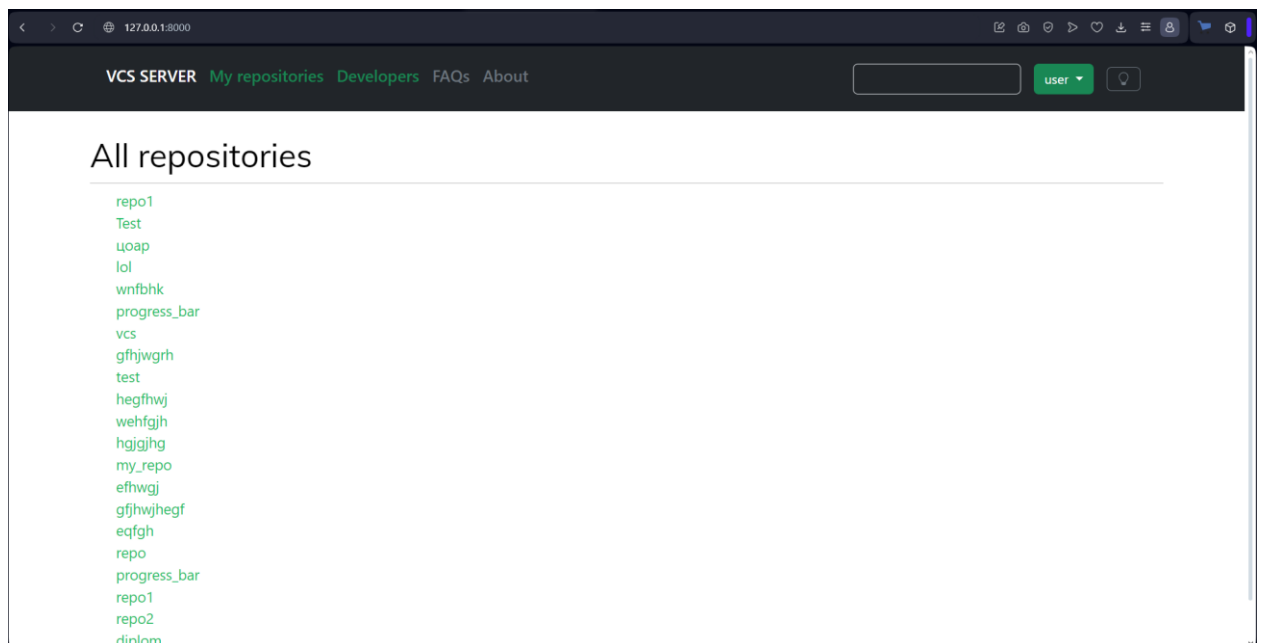


Рисунок 19. Страница со списком всех репозиториев

На рисунке 20 показана страница со списком всех пользователей.

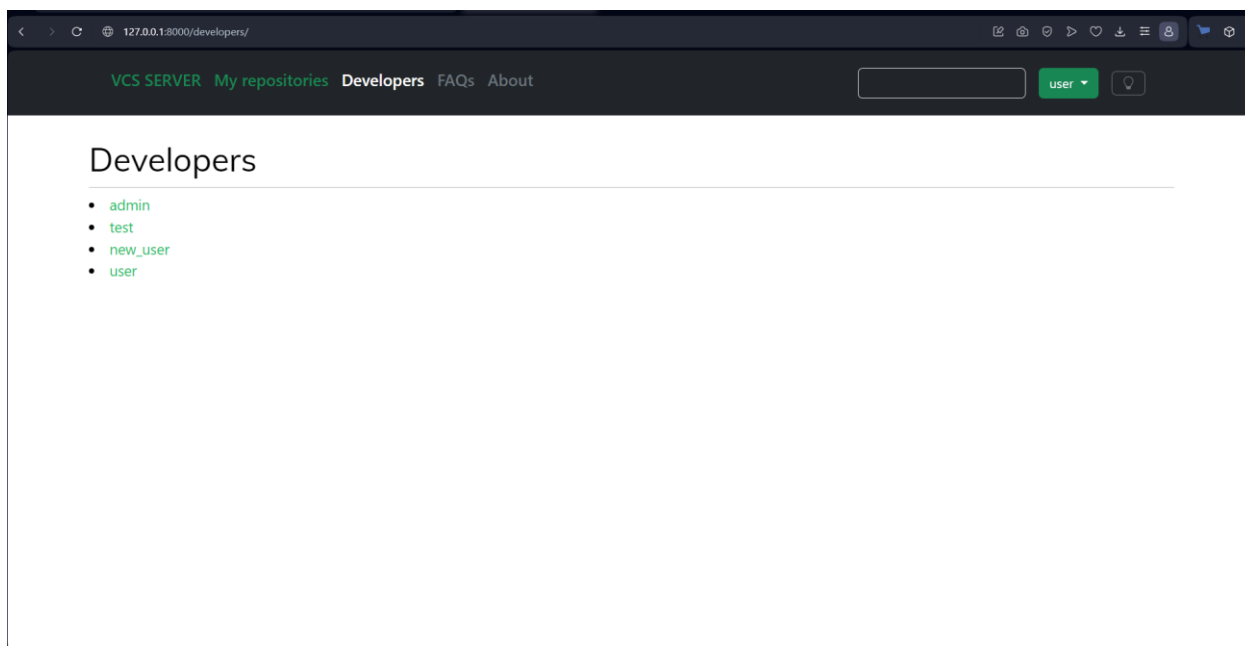


Рисунок 20. Страница со списком всех пользователей

На рисунке 21 показана страница пользователя со списком его репозиторий и кнопкой создания нового. Как видно, URL адрес совпадает с именем пользователя.

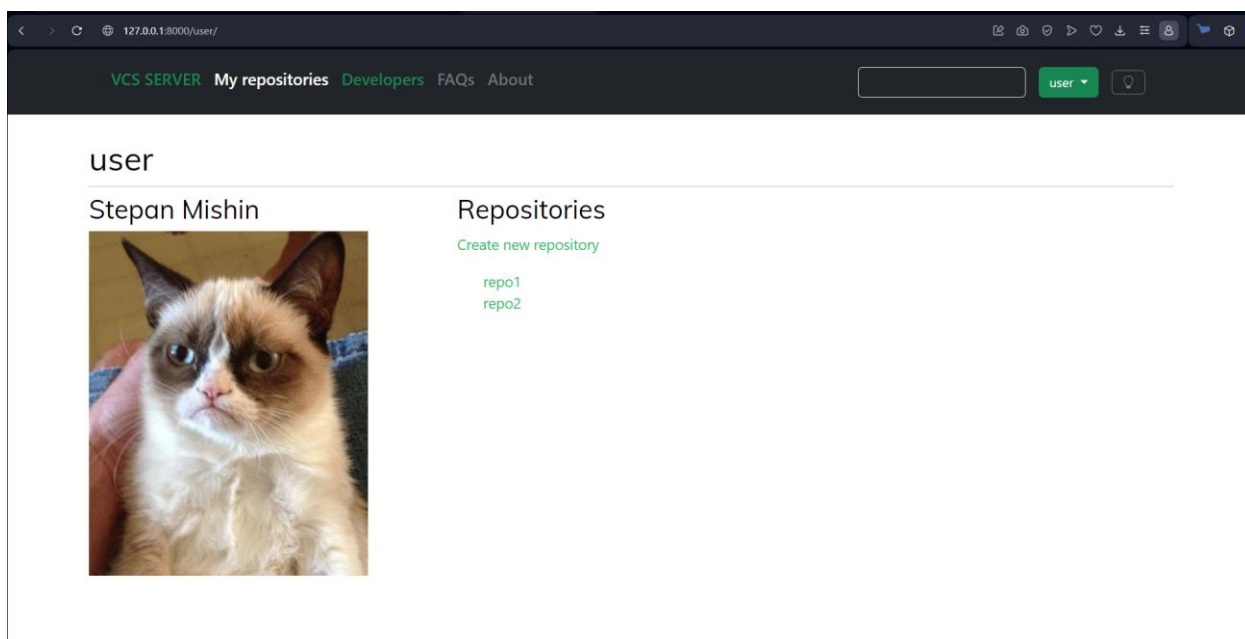


Рисунок 21. Страница пользователя

На рисунке 22 показана страница создания нового репозитория. Файлы можно загрузить двумя способами: кнопка «Выбрать файлы» открывает стандартный файловый менеджер и позволяет загрузить каталог с файлами и подкаталогами (рисунок 23). Также можно просто перетащить нужные файлы в поле drag and drop, созданное с помощью Dropzone.js, и они будут загружены

на сервер.

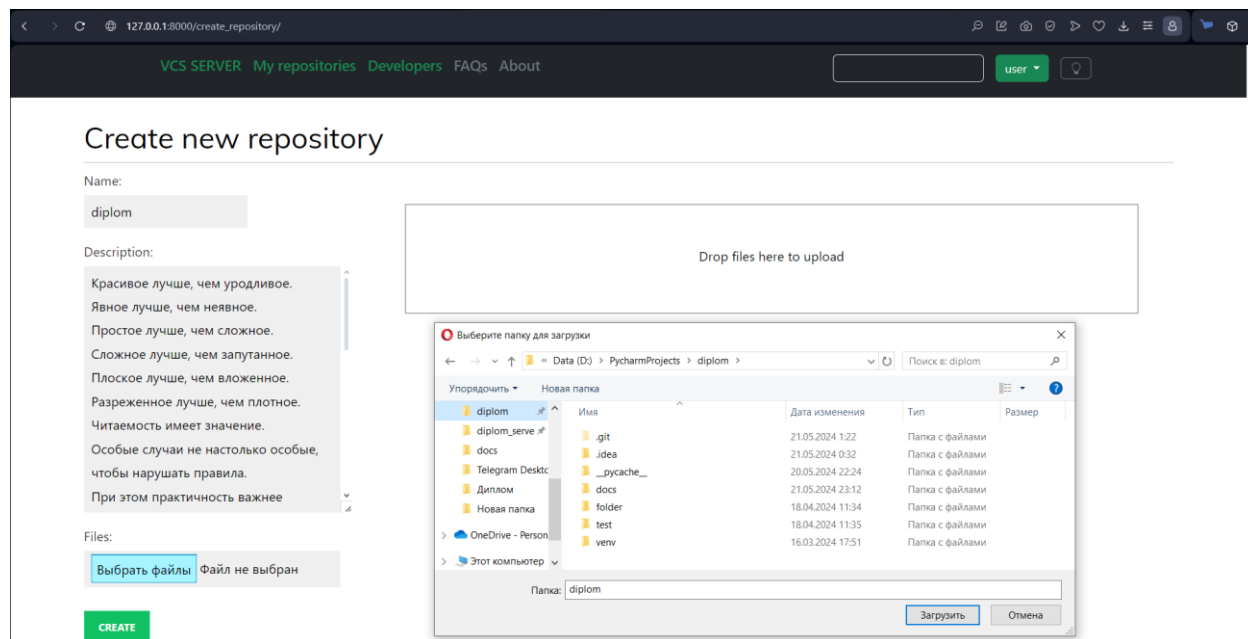


Рисунок 22. Страница создания репозитория

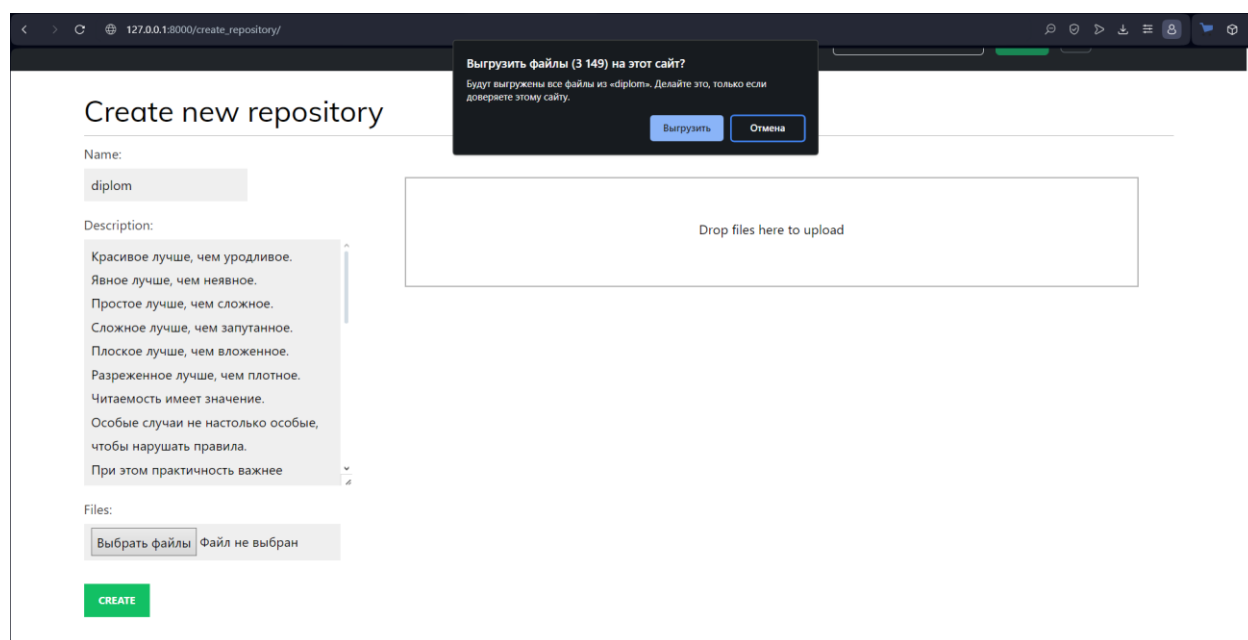


Рисунок 23. Загрузка файлов через файловый менеджер

На рисунке 24 показана страница нового репозитория. Слева отображается файловая структура проекта, справа – описание. URL адрес имеет вид `<hostname>/<username>/<repository_name>`.

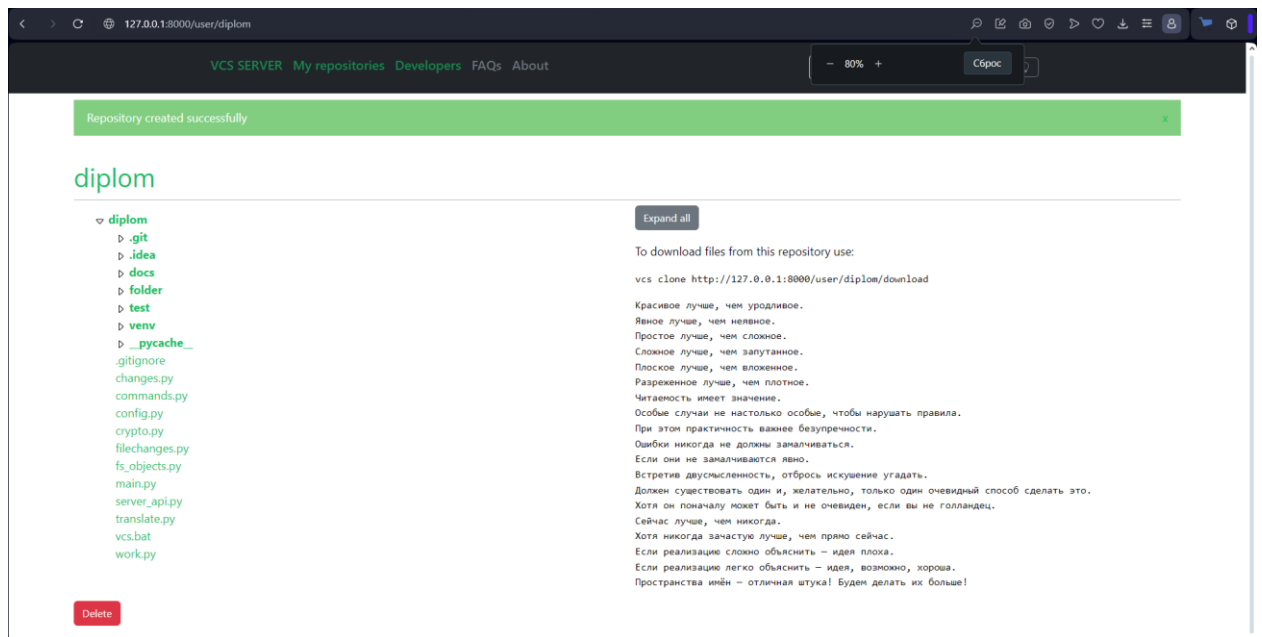


Рисунок 24. Страница репозитория

На рисунке 25 показана одна из папок нового репозитория. Сверху отображается относительный путь к данной папке из корневой. При нажатии на кнопку «Expand all» раскрывается содержимое всех папок, которые изначально свернуты. URL адрес имеет вид `<hostname>/<username>/<repository_name>/<folder_path>`, где `folder_path` – относительный путь к репозиторию.

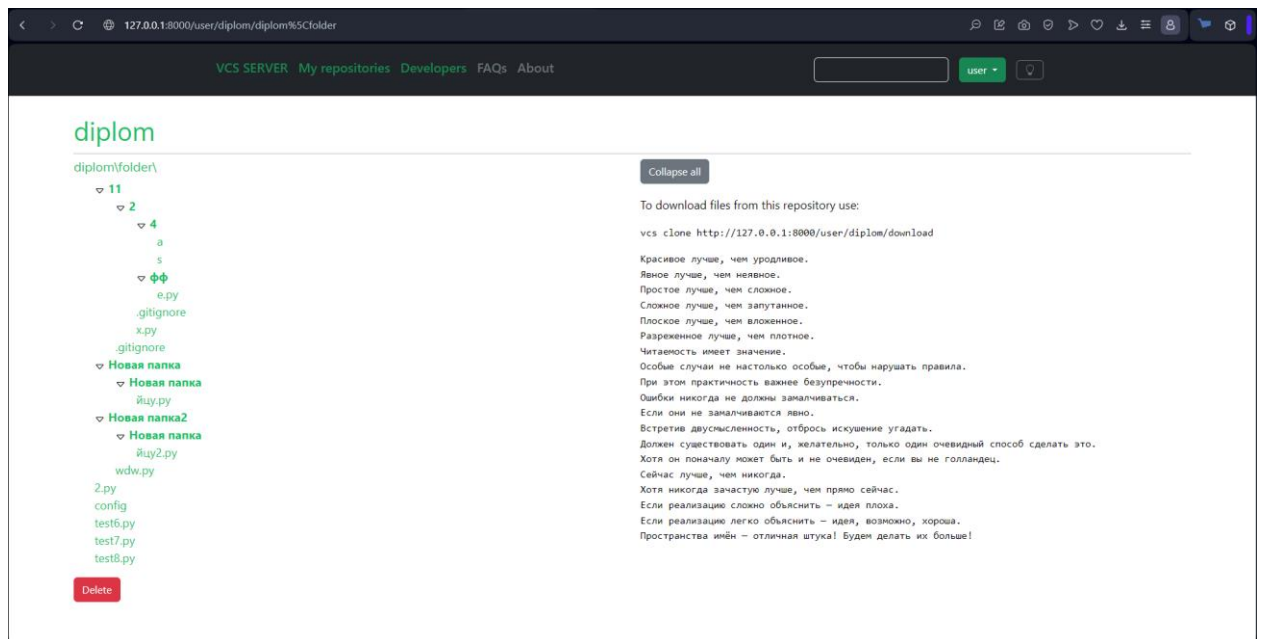


Рисунок 25. Страница папки внутри репозитория

На рисунке 26 показан один из файлов репозитория. Сверху также отображается относительный путь к файлу, размер и время изменения. Если

файл является текстовым или картинкой – то отображается его содержимое. Для подсветки синтаксиса используется библиотека Prism.js

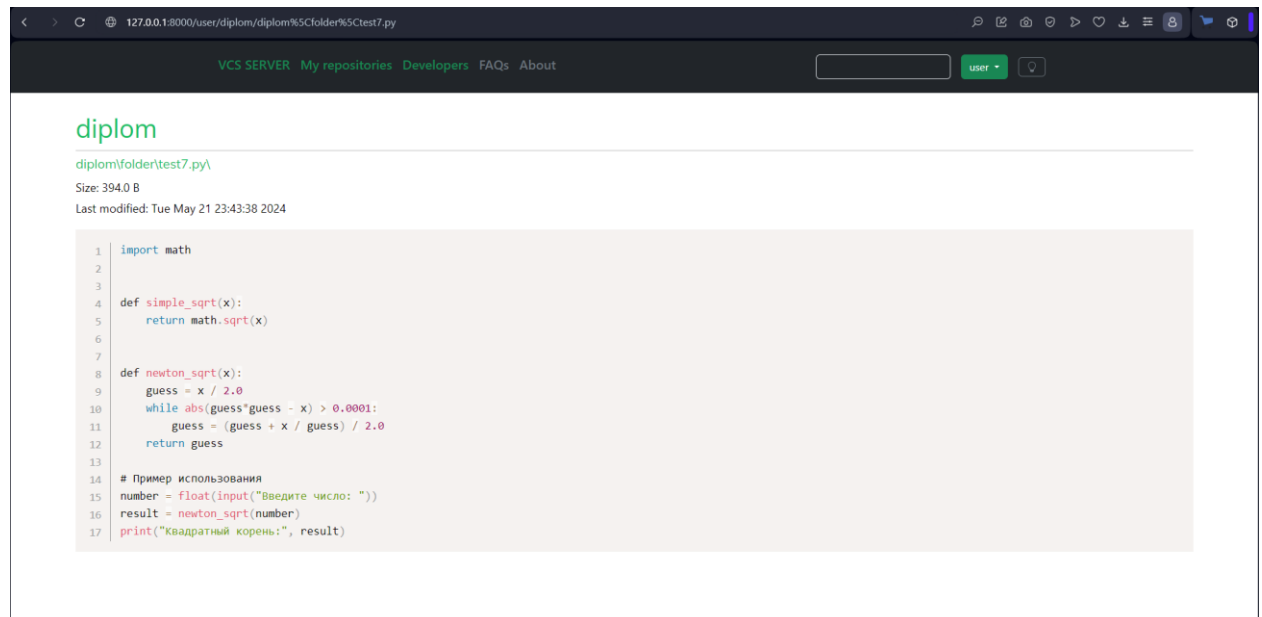


Рисунок 26. Страница файла внутри репозитория

3.3. Интеграция клиента и сервера VCS

3.3.1. Отправка репозитория на сервер

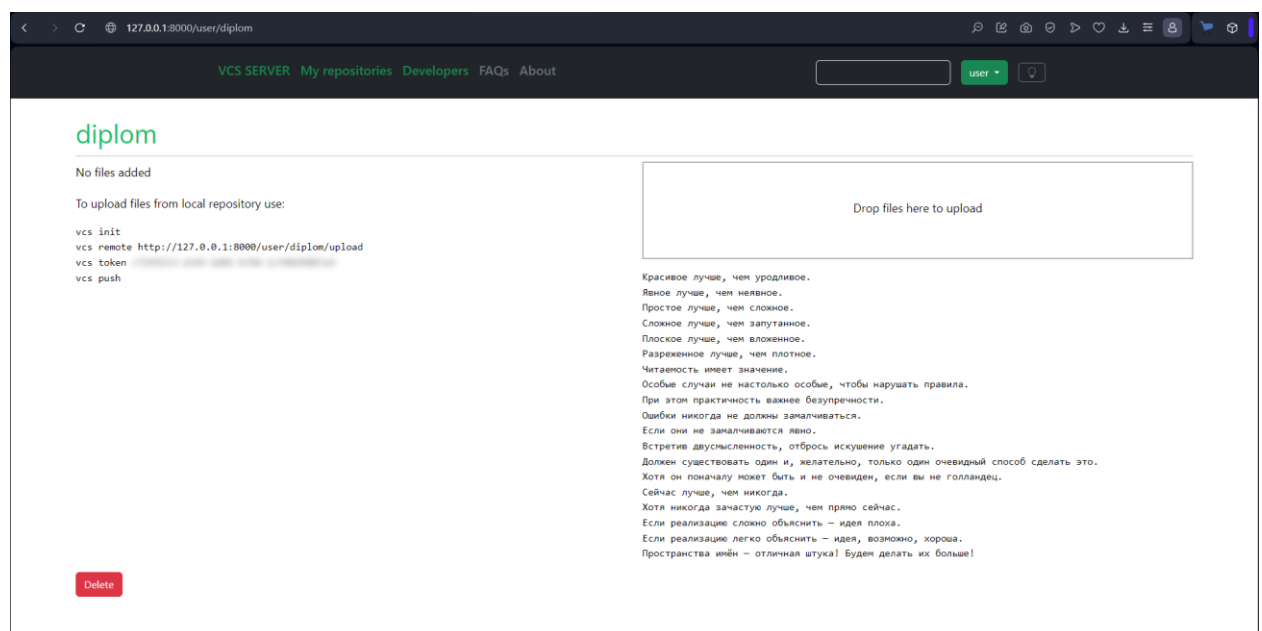


Рисунок 27. Страница репозитория без файлов

При создании репозитория, если не загрузить файлы на сервер, будет отображена страница на рисунке 27. На ней будут указаны команды, скопировав которые, можно отправить локальный репозиторий на удаленный

`vcs init` – инициализирует репозиторий (если он еще не был создан)

`vcs remote <hostname>/<username>/<repository_name>/upload` – указывает, куда загружать репозиторий (какому пользователю и в какой репозиторий). Эта информация сохраняется в `config` файл, после чего можно ей не пользоваться при отправке текущего локального репозитория в выбранный удаленный репозиторий.

`vcs push` – отправляет файлы на сервер.

A screenshot of a Windows Command Prompt window titled "C:\Windows\System32\cmd.exe - vcs push". The window has standard Windows window controls (minimize, maximize, close) in the top right corner. The background is black, and the text is white. The output shows the following sequence of commands and responses:

```
Microsoft Windows [Version 10.0.19045.4412]  
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.  
  
D:\PycharmProjects\diplo>vcs init  
Создан репозиторий в D:\PycharmProjects\diplo  
  
D:\PycharmProjects\diplo>vcs remote http://127.0.0.1:8000/user/diplo/upload  
Пользователь: user  
Репозиторий: diplo  
  
D:\PycharmProjects\diplo>vcs token cf299233-e549-4d01-b784-1c990d9087a9  
  
D:\PycharmProjects\diplo>vcs push  
Progress: |██████████|-----| 51.4% Complete Elapsed: 0:00:06 Remaining: 0:00:05
```

После отправки репозитория возвращаемся на сервер и обновляем страницу репозитория. Как видим на рисунке 29, отображаются загруженные файлы и папки. Файлы отправляются по одному на сервер. Листинг кода

отправки файлов на сервер в Приложении 5.

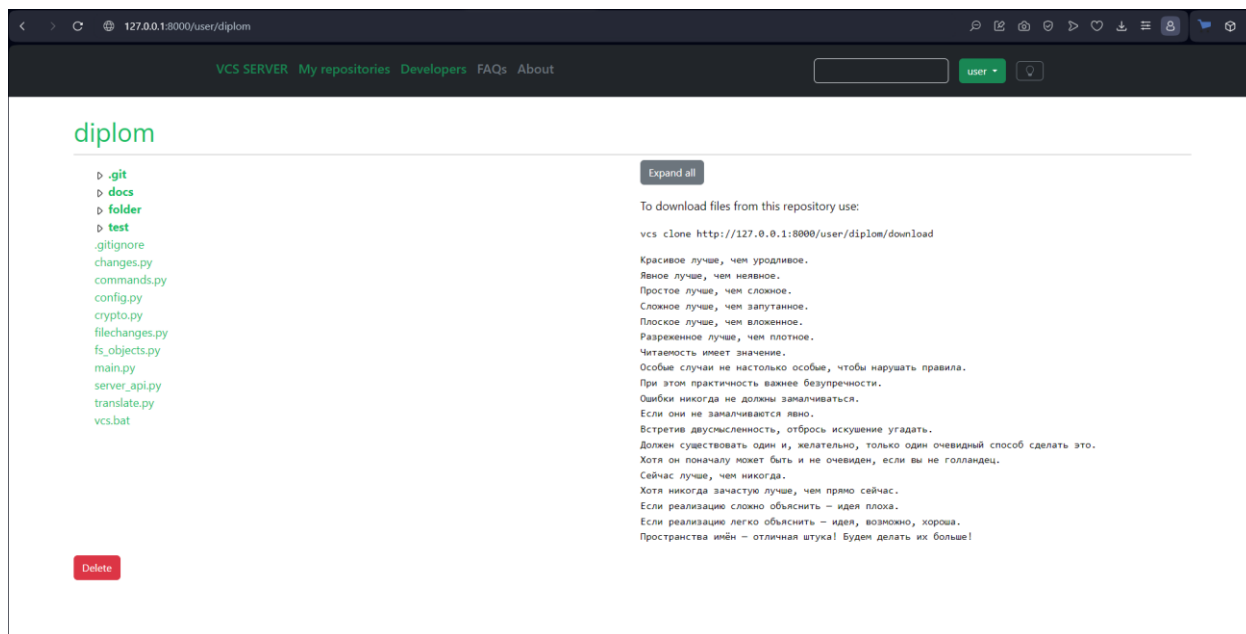


Рисунок 29. Страница репозитория

3.3.2. Загрузка удаленного репозитория с сервера на клиент

После отправки репозитория на сервер у нас или у другого пользователя может возникнуть необходимость загрузки удаленного репозитория к себе локально (клонирование). Как видим на рисунке 29, справа над описанием есть подсказка, как клонировать репозиторий с помощью команды `vcs clone <hostname>/<username>/<repository_name>/download`. Открываем командную строку в том месте, куда нужно клонировать репозиторий, копируем и выполняем команду. Результат показан на рисунке 30. Файлы репозитория архивируются на сервере, и на клиент отправляется архив, который распаковывается. Листинг кода архивирования и отправки файлов сервером в Приложении 6.

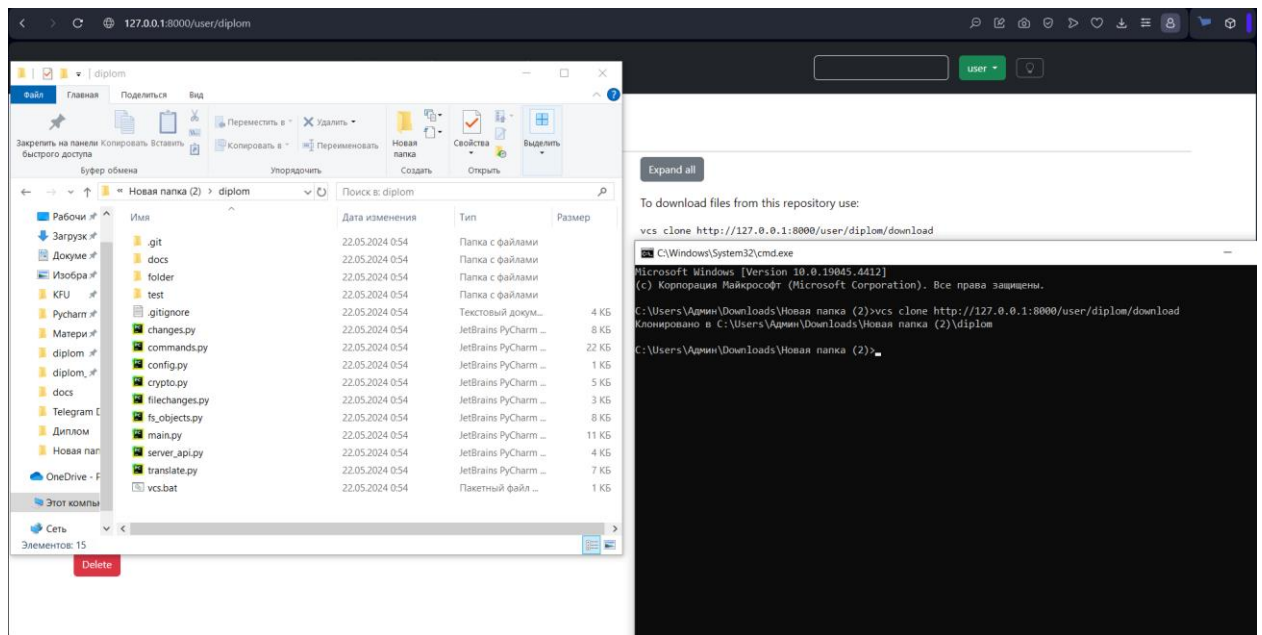


Рисунок 30. Загрузка удаленного репозитория на клиент

4. Тестирование приложения

4.1. Тестирование клиента

Было проведено модульное тестирование клиентской части с помощью фреймворка Pytest [9]. Проверены целостность файлов и папок после отката к предыдущим коммитам. Также я проверил правильность создания патчей после нескольких коммитов с изменениями. Были проверены функции для шифрования и дешифрования файлов и целостность данных после выполнения данных действий.

Часть тестов была 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.

При ручном тестировании я проверил следующие аспекты:

1. Установка и настройка клиента VCS
2. Создание нового репозитория
3. Создание нескольких репозиториях в разных каталогах
4. Удаление репозитория
5. Создание коммита
6. Игнорирование файлов
7. Вывод файловой структуры проекта
8. Просмотр истории коммитов
9. Откат к предыдущему коммиту
10. Шифрование/дешифрование репозитория
11. Отправка файлов репозитория на сервер
12. Загрузка файлов репозитория с сервера на клиент

4.2. Тестирование сервера

Тестирование серверной части было выполнено в ручном режиме с использованием Postman. Были проведены тесты каждой страницы в

следующих популярных браузерах: Google Chrome, Opera, Mozilla Firefox, Microsoft Edge. Тесты показали, что веб-сервис достаточно адаптивен и хорошо выглядит во всех перечисленных браузерах.

Проведены следующие тесты:

1. Регистрация пользователя
2. Выход из аккаунта
3. Аутентификация, вход в аккаунт
4. Изменение пароля
5. Сброс пароля
6. Создание репозитория
7. Удаление репозитория
8. Загрузка файлов в репозиторий через браузер
9. Переключение тем на сайте
10. Отображение содержимого текстовых файлов и подсветка их синтаксиса
11. Отображение изображений у графических файлов
12. Вход через Яндекс OAuth

С помощью Postman я протестировал API, которое сервер предоставляет клиенту для отправки и загрузки файлов, чтобы убедиться, что сервер возвращает корректные ответы и статусы. Пример теста на рисунке 31.

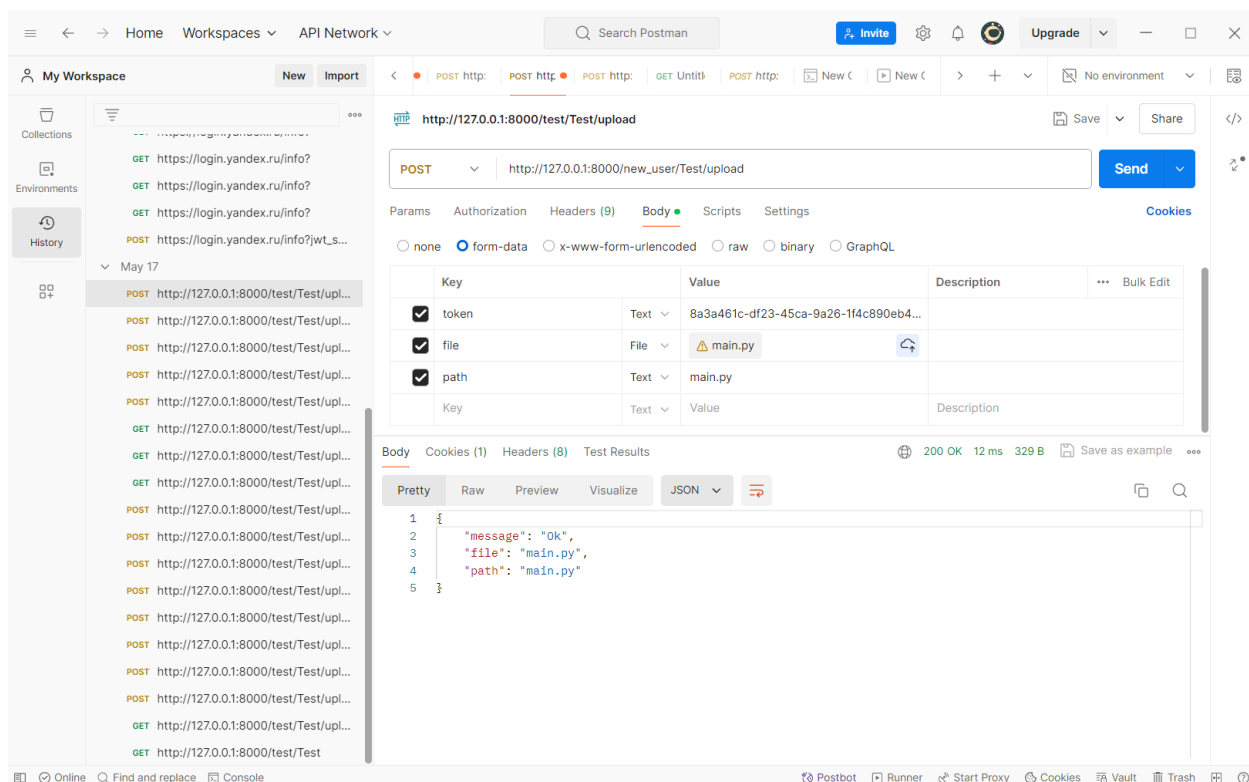


Рисунок 31. Тестирование API с помощью Postman

Заключение

В ходе выполнения дипломной работы была разработана современная система контроля версий (VCS), которая сочетает в себе простоту использования, безопасность данных и расширенные функциональные возможности. Основной целью проекта было создание VCS, доступной как для новичков, так и для опытных пользователей, обеспечивающей удобство, гибкость и надежность.

Реализованные функции, такие как упрощенный набор команд, поддержка нескольких языков, шифрование данных репозитория, делают систему уникальной и привлекательной для широкого круга пользователей. Интеграция с облачным сервисом Яндекс Диск предоставляет пользователям возможность хранить данные в удобном для них месте. Создание торговой площадки открывает новые возможности для монетизации проектов.

Результаты проведенных тестов как ручных, так и автоматических, подтвердили корректность и надежность разработанной системы. Использование современных инструментов и подходов к разработке позволило достичь хорошей производительности и безопасности системы.

В заключении можно отметить, что разработанная система контроля версий удовлетворяет современным требованиям и ожиданиям пользователей, предлагая инновационные решения и улучшенные функциональные возможности. Она имеет потенциал для дальнейшего развития и масштабирования, что открывает перспективы для использования в различных проектах и командах. Выполнение данной дипломной работы позволило не только глубже изучить концепции и методы контроля версий, но и применить полученные знания на практике, создав продукт, который может внести значительный вклад в область программного обеспечения.

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. 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. Коржов В. Многоуровневые системы клиент-сервер – Открытые системы, 1997.
7. Web API с помощью Django REST framework. [Электронный ресурс] – URL: <https://habr.com/ru/articles/160117/> (Дата обращения: 19.02.2024)
8. Меле А. Django 4 в примерах. ДМК Пресс, 2023. – 800 с.
9. Full pytest documentation. [Электронный ресурс] – URL: <https://docs.pytest.org/en/latest/contents.html> (Дата обращения: 09.04.2024)

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()

```

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)
return changes_list

```


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)

```

Листинг кода формы для загрузки файлов через браузер

```

<!-- Форма для загрузки файлов -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/dropzone/5.9.3/dropzone.min.css">
<script src="https://cdnjs.cloudflare.com/ajax/libs/dropzone/5.9.3/dropzone.min.js"></script>

<form action="{% url 'file_upload_view' temp_repository_name %}" class="dropzone" id="my-dropzone"
enctype="multipart/form-data">
  {% csrf_token %}
  <input type="text" id="full_paths" name="full_paths" hidden />
</form>

<script>
  Dropzone.options.myDropzone = {
    paramName: "file",
    init: function () {
      var myDropzone = this;

      // Обработка события "sending" для добавления информации о пути файла в FormData
      this.on("sending", function(file, xhr, formData) {
        var full_paths = {};
        if (file.webkitRelativePath) {
          full_paths[file.name] = file.webkitRelativePath;
        } else {
          full_paths[file.name] = file.fullPath || file.name; // Обработать случаи без webkitRelativePath
        }
        console.log(full_paths);
        document.querySelector("#full_paths").value = JSON.stringify(full_paths);
      });
    }
  };
</script>

```

Листинг кода отправки файлов на сервер

```

import datetime
import os.path
import sys

import requests
import zipfile
import io
import os

import commands
from config import BASE_PATH
from translate import phrase

def send_file(filepath, url, token):
    # Открытие файла в режиме чтения двоичных данных
    with open(filepath, 'rb') as f:
        files = {'file': f}
        # data = {'token': token, 'path': os.path.relpath(filepath, BASE_PATH.parent)}
        data = {'token': token, 'path': os.path.relpath(filepath, BASE_PATH)}
        # Отправка POST-запроса с файлом и данными формы
        response = requests.post(url, files=files, data=data)

    if response.status_code != 200:
        raise Exception(f'Failed: {response.status_code} {response.text}')

def send_files(url, token, path=BASE_PATH):
    tree = commands.iter_folder(path=BASE_PATH, print_content=False)
    filenames = list(tree.get_filenames())
    total_tasks = len(filenames)
    start_time = datetime.datetime.now()
    for i in range(total_tasks):
        full_path = os.path.join(BASE_PATH.parent, filenames[i])
        try:
            send_file(full_path, url, token)
        except Exception as e:
            print(e)
    return
    print_progress_bar(i + 1, total_tasks, start_time, prefix='Progress:', suffix='Complete', length=50)

```

Листинг кода архивирования и отправки файлов сервером

```

@csrf_exempt
def api_download(request, username, repository_name):
    if request.method == 'GET':
        user = get_object_or_404(User, username=username)
        repository = get_object_or_404(Repository, user=user, name=repository_name)
        # token = request.GET.get('token')
        # if user.profile.token != token:
        #     return JsonResponse({'message': 'Invalid token'}, status=401)
        full_path = os.path.join(settings.MEDIA_ROOT, 'files', username, repository_name)
        if not os.path.exists(full_path):
            return JsonResponse({'message': 'Folder not exists'}, status=404)

        # Create a zip file in memory
        zip_buffer = io.BytesIO()
        with zipfile.ZipFile(zip_buffer, 'w', zipfile.ZIP_DEFLATED) as zip_file:
            for root, dirs, files in os.walk(full_path):
                for file in files:
                    file_path = os.path.join(root, file)
                    arcname = os.path.relpath(file_path, full_path)
                    zip_file.write(file_path, arcname)
        zip_buffer.seek(0)

        response = HttpResponse(zip_buffer, content_type='application/zip')
        response['Content-Disposition'] = f'attachment; filename={os.path.basename(full_path)}.zip'
        return response
    return JsonResponse({'message': 'Method not allowed'}, status=405)

```