

ТЕМА 1. Работа с репозиториями.

Репозиторий (repository) в контексте систем контроля версий (VCS), таких как Git, представляет собой хранилище данных, которое содержит все файлы, историю изменений, ветки, теги и другую информацию о проекте. В репозитории сохраняются все версии файлов, начиная с первоначальной версии, и позволяют разработчикам отслеживать, совместно работать над проектом и управлять кодом.

В контексте Git, репозиторий может быть локальным (на вашем компьютере) или удаленным (на сервере). Вот какие функции выполняет репозиторий:

- **Хранение истории изменений.** Репозиторий сохраняет историю изменений каждого файла в проекте, начиная с самого первого коммита. Это позволяет вернуться к любой предыдущей версии файла и понять, кто и когда вносил изменения.
- **Управление версиями.** Репозиторий позволяет отслеживать и управлять разными версиями файлов. Это включает создание новых версий (коммитов), слияние изменений из разных веток и тегирование для обозначения важных моментов в истории проекта.
- **Совместная работа.** Репозиторий на удаленном сервере (например, GitHub, GitLab, Bitbucket) позволяет разработчикам совместно работать над проектом. Они могут клонировать репозиторий, вносить изменения, создавать ветки, делать запросы на слияние и обмениваться изменениями.
- **Резервное копирование.** Репозиторий на удаленном сервере служит важным инструментом для резервного копирования кода и данных проекта. Это предотвращает потерю информации при сбоях или случайных событиях.
- **Отслеживание ошибок.** В репозитории также можно хранить информацию о задачах, багах и других задачах. Это позволяет связать изменения в коде с конкретными задачами и обеспечивает более структурированный процесс разработки.

Каждый разработчик имеет свой локальный репозиторий, с которым он работает. Он может клонировать удаленный репозиторий для получения начальной копии кода и затем вносить изменения в свой локальный репозиторий. После завершения работы изменения могут быть отправлены на удаленный репозиторий для совместной работы и ревью.

1.1. Работа с GitHub

GitHub - это веб-платформа для хостинга и совместной разработки проектов с использованием системы контроля версий Git. Она предоставляет инструменты для управления и хранения кода, совместной работы над проектами, отслеживания задач, управления версиями и многого другого. GitHub стал одной из самых популярных платформ для разработки программного обеспечения и сотрудничества в сообществе разработчиков.

Основные аспекты GitHub:

- **Репозитории.** GitHub позволяет пользователям создавать удаленные репозитории, в которых можно хранить и управлять исходным кодом, файлами, документацией и другой информацией. Репозитории могут быть публичными (доступны для всех) или приватными (доступ ограничен).
- **Совместная разработка.** GitHub обеспечивает инструменты для совместной работы над проектами. Пользователи могут создавать ветки, вносить изменения, комментировать код, проводить ревью изменений и создавать запросы на слияние для объединения изменений.
- **Отслеживание задач.** GitHub предоставляет систему управления задачами (Issues), где разработчики могут создавать задачи, отслеживать баги, функциональные требования и другие задачи, связанные с проектом.

- **Pull Requests.** Pull Request (PR) - это механизм запроса на слияние изменений из одной ветки в другую. Он позволяет разработчикам обсуждать и ревью изменения, а также вносить правки перед тем как изменения будут включены в основную ветку.
- **Интеграция с CI/CD.** GitHub интегрируется с системами непрерывной интеграции (CI) и непрерывной доставки (CD), что позволяет автоматизировать процесс сборки, тестирования и развертывания кода.
- **GitHub Actions.** Это встроенные инструменты для автоматизации рабочих процессов, таких как сборка, тестирование, деплоймент, уведомления и многое другое.
- **Социальные аспекты.** GitHub создает социальное сообщество разработчиков, где пользователи могут подписываться на репозитории, ставить звезды, следить за активностью других разработчиков и общаться через комментарии.

GitHub позволяет разработчикам работать над проектами независимо от их масштаба, обеспечивая удобные средства для совместной работы, управления кодом и отслеживания изменений.

1.2. Краткая инструкция как начать пользоваться GitHub

- **Создание аккаунта.**
 - Перейдите на [официальный сайт GitHub](https://github.com/) и нажмите на кнопку "Sign up" (Зарегистрироваться).
 - Введите свои данные: имя пользователя, электронную почту и пароль.
 - Пройдите процесс верификации, если это потребуется.
- **Создание репозитория.**
 - После входа в аккаунт, нажмите на кнопку "+", расположенную в верхнем правом углу, и выберите "New repository" (Новый репозиторий).
 - Введите имя для вашего репозитория, описание (опционально), выберите публичный или приватный режим доступа, и добавьте файл `.gitignore` и лицензию, если это необходимо.
 - Нажмите на кнопку "Create repository" (Создать репозиторий).

Это базовые шаги по использованию GitHub. Помните, что GitHub предоставляет множество функций, таких как работы с задачами, комментирование кода, настройка автоматизации с помощью GitHub Actions и многое другое. Которые мы разберем в следующей теме.

1.3. Отчетность по выполнению работ

Вся отчетность по текущему курсу будет Вами должна быть сделана с помощью средств GitHub.

1. Создавайте ветку в вашем репозитории. Каждая ветка должна соответствовать Теме по которой вы выполняете цикл практических и самостоятельных работ (Рисунок 1.1.)

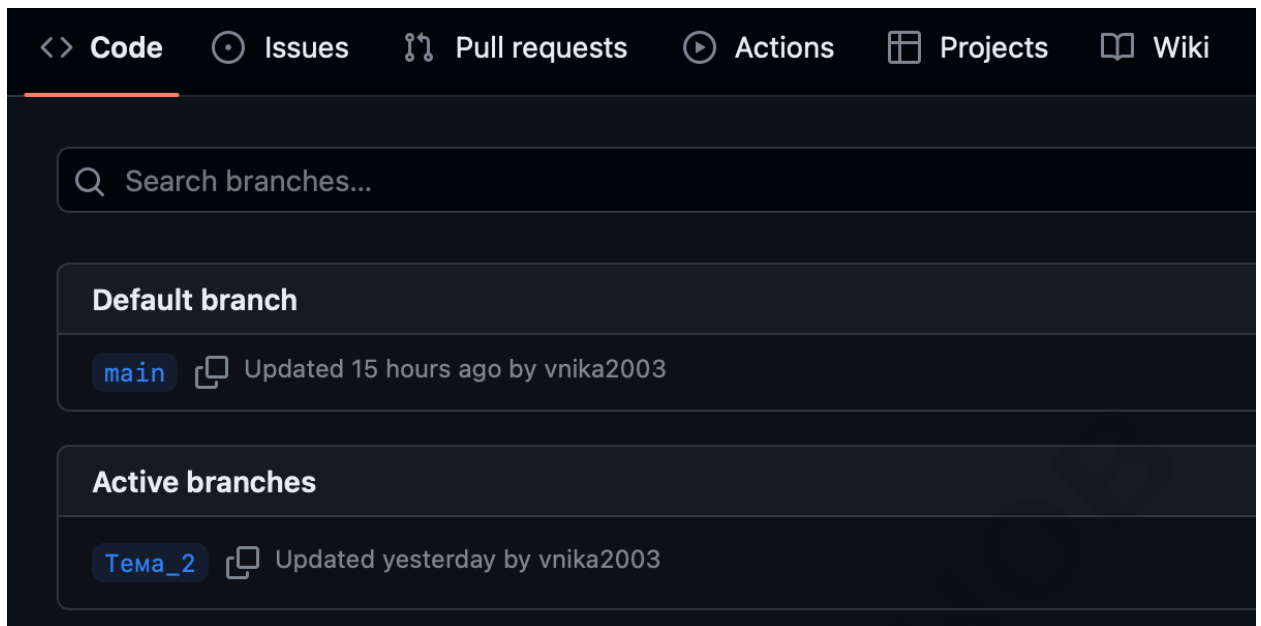


Рисунок 1.1. Создание ветки

2. В каждой ветке должны содержаться:
 - Отчет в формате *.md (Инструкция по заполнению ниже).
 - Файлы с кодом каждого **самостоятельного** задания. (Пример: тема 2 содержит 10 самостоятельных заданий, соответственно. Соответственно 10 файлов с расширением *.py должны быть в ветке Тема_2). Файлы **лабораторных** работ приветствуются, но не обязательны!
 - Папка со скринами (Нужны для оформления отчета)
- Пример представлен на рисунке 1.2.

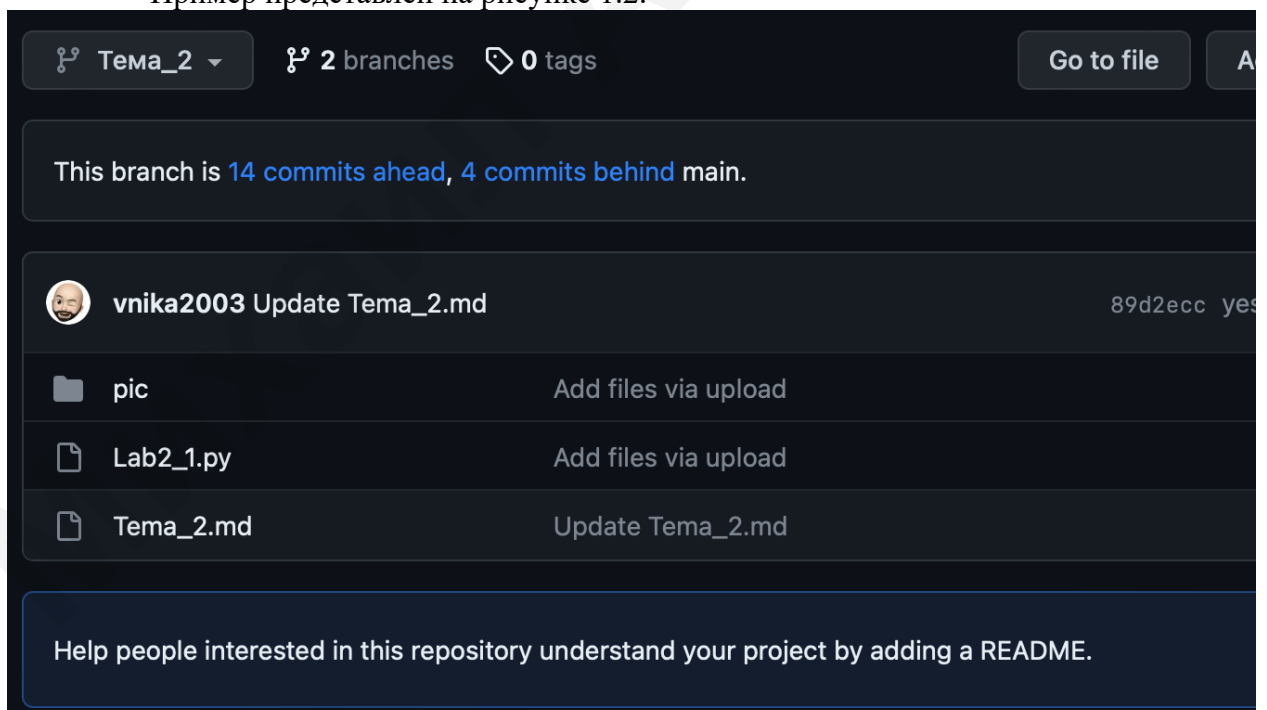


Рисунок 1.2. Краткий пример оформления ветки


3. Оформление файла **README.md**
 Отчет состоит из:
 - Названия темы, ФИО и группы студента (Рисунок 1.3.)

Preview

Code

Blame

51 lines (38 loc) · 2.1 KB

 Code 55% faster with GitHub Copilot

Тема 2. Базовые операции языка Python

Отчет по Теме #2 выполнил(а):

- Панов Михаил Александрович
- ИВТ-21-1

Рисунок 1.3. «Шапка» отчета

- Таблица с итогами. Ваша задача самостоятельно проставить + или - по выполненным работам. Знак "+" - задание выполнено; знак "-" - задание не выполнено. Данная таблица составляется с расчетом на честность студента (Рисунок 1.4.)

Задание	Лаб_раб	Сам_раб
Задание 1	+	-
Задание 2	-	-
Задание 3	-	-
Задание 4	-	-
Задание 5	-	-
Задание 6	-	-
Задание 7	-	-
Задание 8	-	-
Задание 9	-	-
Задание 10	-	-

знак "+" - задание выполнено; знак "-" - задание не выполнено;

Работу проверили:

- к.э.н., доцент Панов М.А.

Рисунок 1.4. Таблица с итогами

- Оформление задания (Рисунок 1.5.)
Укажите номер задания, код, скрин с результатами, выводы. На скрине с результатами должно быть видно текущую дату и время.
Выводы в лабораторных заданиях приветствуются.
Выводы в самостоятельных заданиях обязательны. Без выводов в самостоятельных заданиях работа не засчитывается!!!

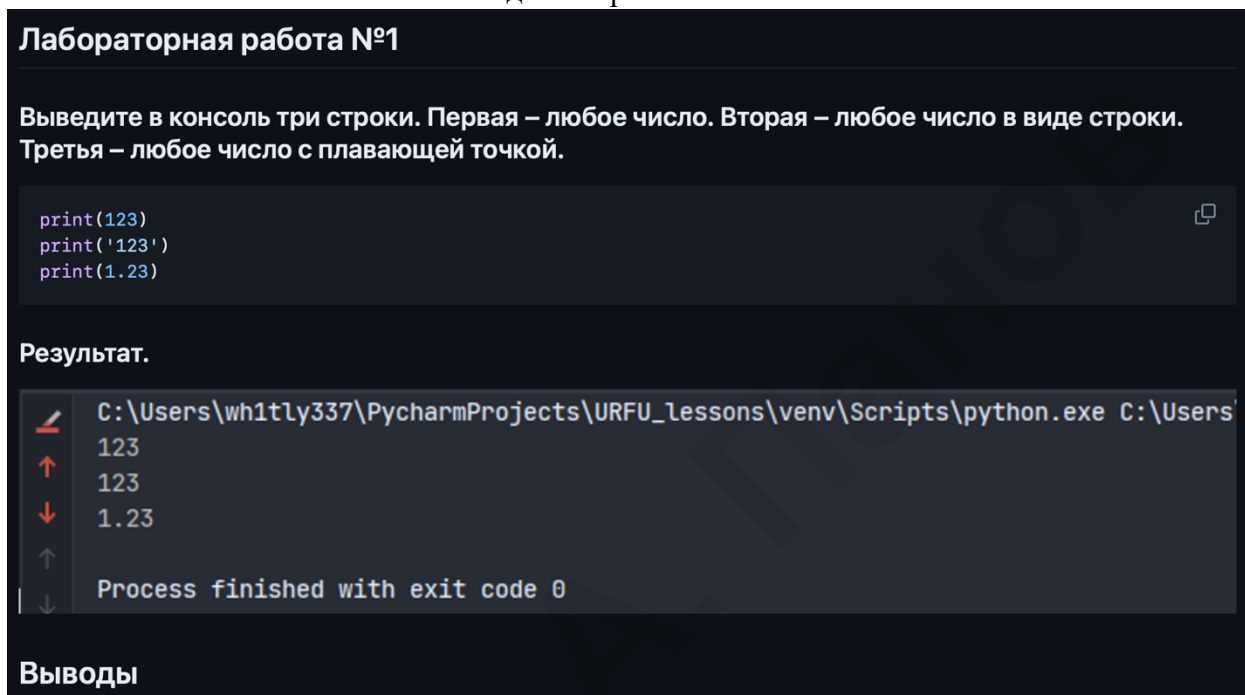


Рисунок 1.5. Оформление задания

Если Тема содержит 10 лабораторных и 10 самостоятельных работ, то в отчете должны быть 20 блоков кода, 20 скриншотов вывода в консоли (обязательно наличие видимой даты и времени) и 10 обязательных выводов по самостоятельным работам.

Ссылка на репозиторий должна быть приложена к отчету на портале.

ТЕМА 2. Работа с Git

Git - это распределенная система контроля версий, разработанная Линусом Торвальдсом. Она используется для управления изменениями в исходном коде программного обеспечения и других текстовых документах. Git позволяет не только отслеживать изменения в файлах, но и эффективно сотрудничать над проектами с другими разработчиками.

Git используется для управления версиями и сотрудничества над проектами в области разработки программного обеспечения и других текстовых документов. Вот несколько основных целей и причин, по которым Git является важным инструментом:

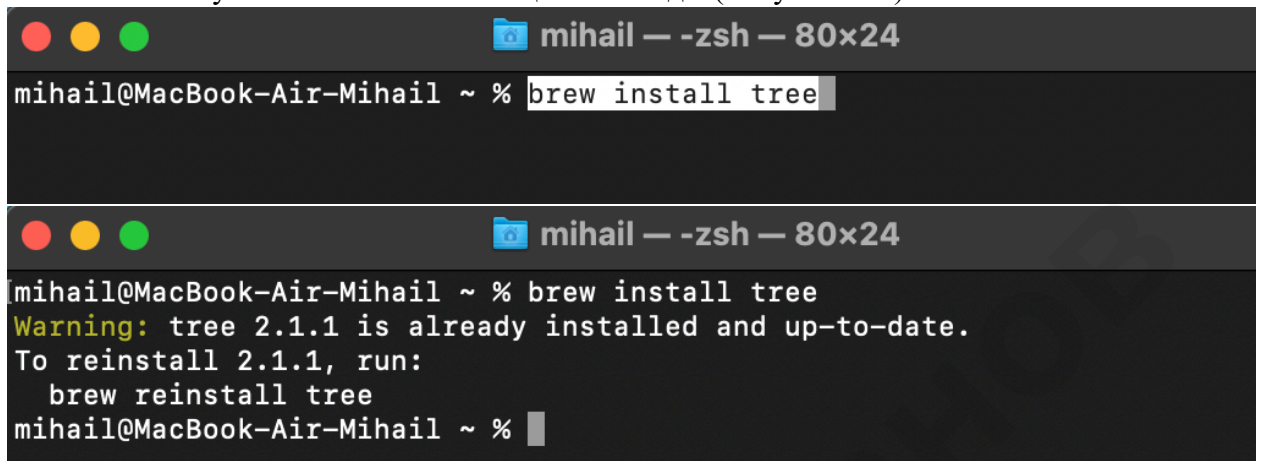
- **Управление версиями.** Git позволяет отслеживать изменения в исходном коде и других файлах с течением времени. Вы можете видеть, какие конкретные изменения были внесены, кто и когда вносил эти изменения. Это упрощает работу с историей кода и помогает быстро вернуться к предыдущим версиям, если что-то идет не так.
- **Ветвление и параллельная разработка.** Git позволяет создавать отдельные ветви, в которых можно разрабатывать новые фичи, исправления ошибок и другие изменения независимо. Это способствует параллельной разработке и предотвращает конфликты между изменениями, вносимыми разными разработчиками.
- **Сотрудничество.** Git упрощает сотрудничество между разработчиками. Он позволяет им работать над одним и тем же проектом, объединяя свои изменения через слияния. Это также способствует рецензированию кода, обсуждению изменений и внесению предложений по улучшению.
- **Откат и исправление ошибок.** Если что-то идет не так, Git позволяет быстро откатиться к предыдущим версиям кода, чтобы исправить проблемы. Это особенно полезно при обнаружении ошибок после внесения изменений.
- **Удаленное хранение и резервное копирование.** Git поддерживает удаленные репозитории, которые можно использовать для хранения кода на сервере. Популярные хостинговые платформы, такие как GitHub и GitLab, предоставляют возможность хранения и совместной работы над кодом удаленно.
- **Инструменты совместной работы.** Git интегрируется с различными инструментами для совместной разработки, автоматической сборки, тестирования и развертывания, что делает процесс разработки более автоматизированным и эффективным.
- **Отслеживание изменений в текстовых документах.** Наряду с разработкой программного обеспечения, Git может использоваться для отслеживания изменений в любых текстовых документах, таких как документация, научные работы и другие текстовые файлы.

2.1. Установка

Установка Git зависит от операционной системы, которую вы используете. Вот инструкции по установке Git на различных популярных операционных системах:

- **Windows:**
 - Перейдите на официальный сайт Git: <https://git-scm.com/download/win>
 - Скачайте исполняемый файл для вашей архитектуры (32-битная или 64-битная).
 - Запустите загруженный файл и следуйте инструкциям мастера установки.

- Во время установки выберите опции по своему усмотрению. Обычно рекомендуется использовать настройки по умолчанию.
- **MacOS:**
 - Если у вас установлен Homebrew (популярный менеджер пакетов для macOS), вы можете установить Git с помощью команды (Рисунок 2.1.):



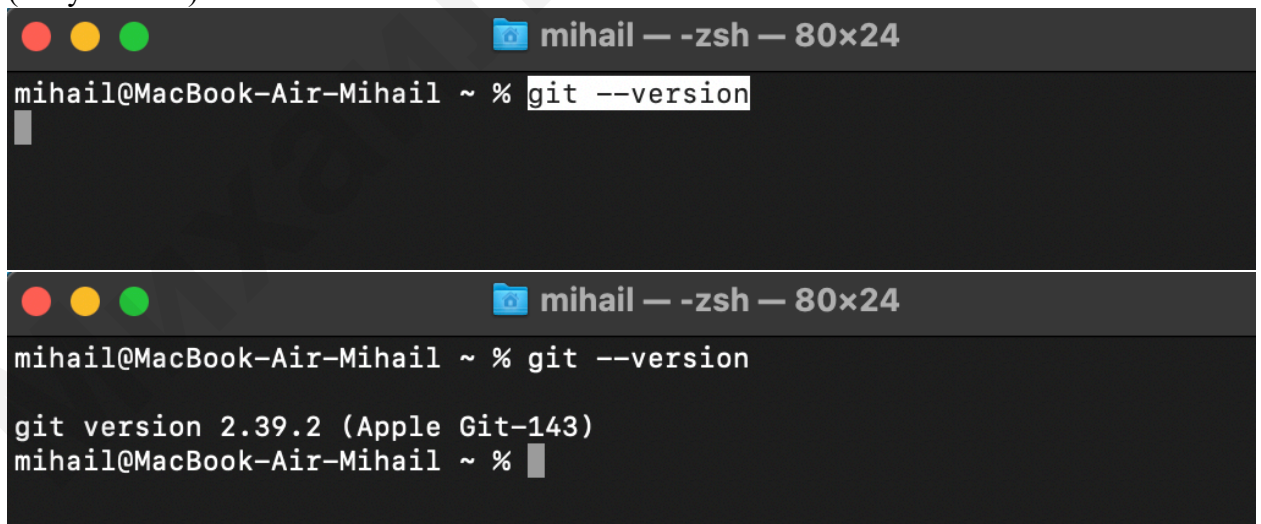
```
mi hail@MacBook-Air-Mi hail ~ % brew install tree

mi hail@MacBook-Air-Mi hail ~ % brew install tree
Warning: tree 2.1.1 is already installed and up-to-date.
To reinstall 2.1.1, run:
  brew reinstall tree
mi hail@MacBook-Air-Mi hail ~ %
```

Рисунок 2.1. Установка Git

- Или вы можете загрузить установщик Git с официального сайта: <https://git-scm.com/download/mac>
- Запустите загруженный файл и следуйте инструкциям мастера установки.
- **Linux:**
 - Для большинства дистрибутивов Linux Git можно установить через менеджера пакетов.

После установки Git, вы можете проверить его версию, введя в терминале команду (Рисунок 2.2.):



```
mi hail@MacBook-Air-Mi hail ~ % git --version

mi hail@MacBook-Air-Mi hail ~ % git --version
git version 2.39.2 (Apple Git-143)
mi hail@MacBook-Air-Mi hail ~ %
```

Рисунок 2.2. Проверка версии Git

Если команда успешно выполнена и показывает версию Git, это означает, что Git установлен и готов к использованию.

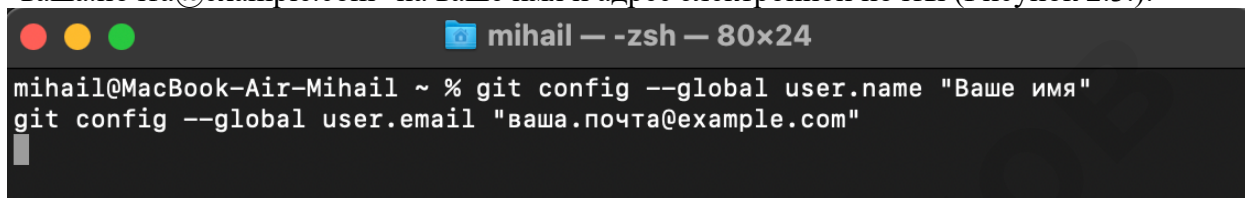
2.2.Настройка

Настройка Git включает в себя установку базовой конфигурации, такой как имя пользователя и адрес электронной почты, а также настройку опций, связанных с отображением и поведением Git. Вот как настроить Git:

– **Установка имени пользователя и адреса электронной почты:**

Одной из первых вещей, которые вы должны сделать после установки Git, это настройка вашего имени и адреса электронной почты. Эти данные будут включены в каждый коммит, который вы создаете.

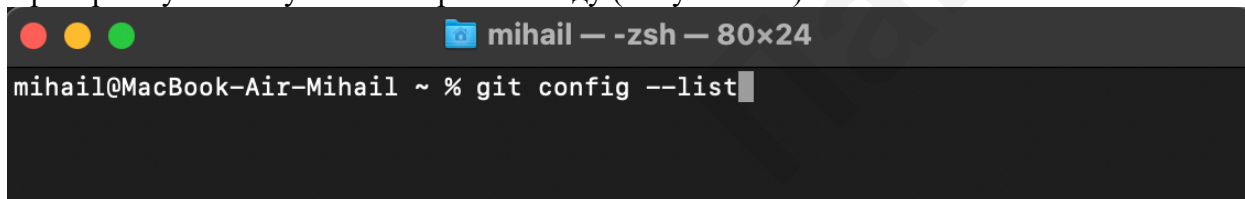
В терминале выполните следующие команды, заменив "Ваше имя" и "ваша.почта@example.com" на ваше имя и адрес электронной почты (Рисунок 2.3.):



```
mi hail — zsh — 80x24
mi hail@MacBook-Air-Mi hail ~ % git config --global user.name "Ваше имя"
git config --global user.email "ваша.почта@example.com"
```

Рисунок 2.3. Установка имени пользователя и адреса электронной почты

Проверить установку можно через команду (Рисунок 2.4.)

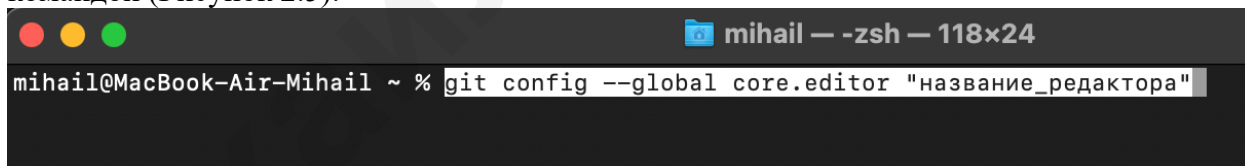


```
mi hail — zsh — 80x24
mi hail@MacBook-Air-Mi hail ~ % git config --list
```

Рисунок 2.4. Проверка установки

– **Настройка редактора для сообщений коммитов (по желанию):**

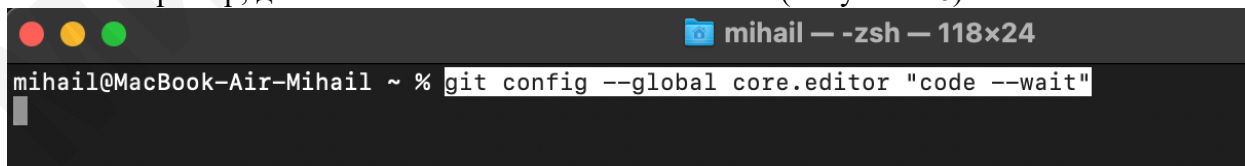
Git использует стандартный текстовый редактор для написания комментариев к коммитам. По умолчанию это может быть nano, vim или другой редактор командной строки. Если вы хотите использовать другой редактор, вы можете настроить его следующей командой (Рисунок 2.5):



```
mi hail — zsh — 118x24
mi hail@MacBook-Air-Mi hail ~ % git config --global core.editor "название_редактора"
```

Рисунок 2.5. Настройка редактора для сообщений

Например, для использования Visual Studio Code (Рисунок 2.6):



```
mi hail — zsh — 118x24
mi hail@MacBook-Air-Mi hail ~ % git config --global core.editor "code --wait"
```

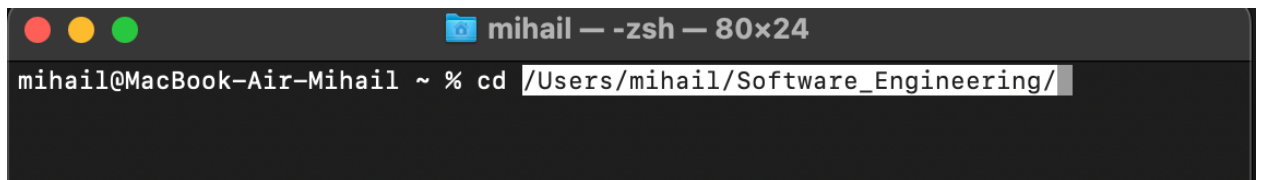
Рисунок 2.6. Настройка редактора для сообщений Visual Studio Code

2.3. Создание нового репозитория

Создание нового репозитория в Git представляет собой первоначальную инициализацию папки проекта как репозитория, в котором Git будет отслеживать историю изменений. Вот как создать новый репозиторий:

– **Инициализация репозитория локально:**

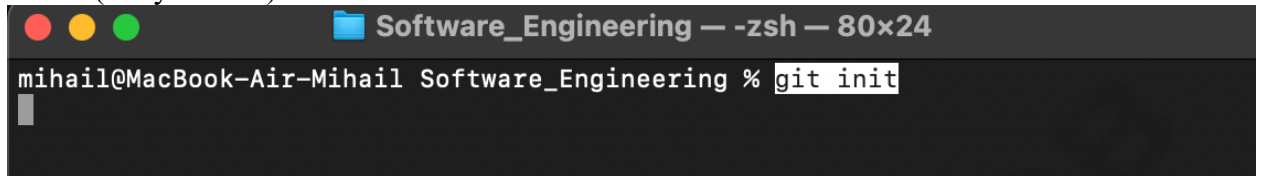
Перейдите в папку проекта с помощью терминала (Рисунок 2.7):



```
mi hail — -zsh — 80x24
mi hail@MacBook-Air-Mi hail ~ % cd /Users/mi hail/Software_Engineering/
```

Рисунок 2.7. Переход в папку проекта
инициализация репозитория:

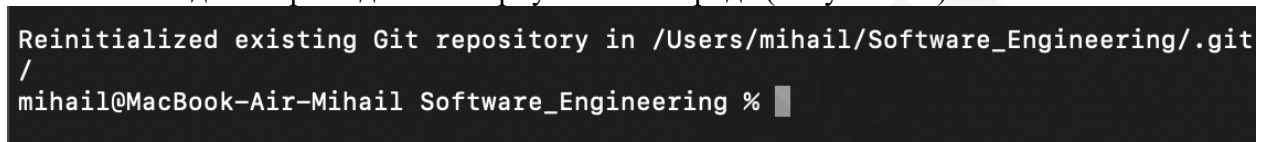
Выполните команду *Git init*, чтобы инициализировать пустой репозиторий в текущей папке (Рисунок 2.8):



```
Software_Engineering — -zsh — 80x24
mi hail@MacBook-Air-Mi hail Software_Engineering % git init
```

Рисунок 2.8. Переход в папку проекта

Командная строка должна вернуть что-то вроде (Рисунок 2.9):



```
Reinitialized existing Git repository in /Users/mi hail/Software_Engineering/.git
/
mi hail@MacBook-Air-Mi hail Software_Engineering %
```

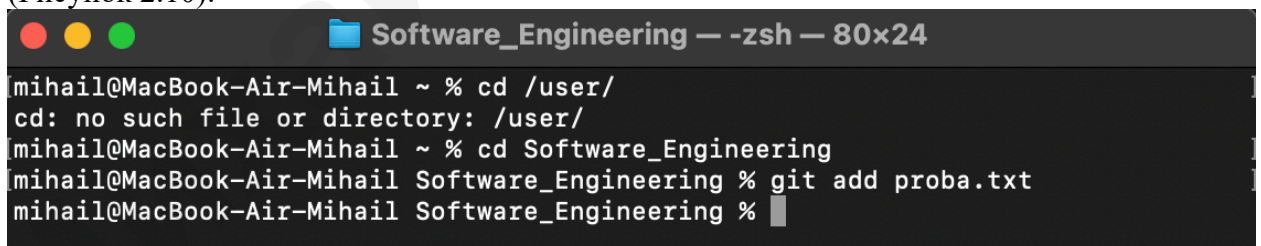
Рисунок 2.9. Результат выполнения *Git init*

2.4. Подготовка файлов

Подготовка файлов в Git означает начало отслеживания изменений в файлах, чтобы включить их в будущий коммит. Процесс включает в себя команду *git add*, которая добавляет файлы в "индекс" (или "промежуточное хранилище") перед тем, как они будут включены в следующий коммит. Вот как подготовить файлы к коммиту:

– Добавление одного файла:

Чтобы подготовить конкретный файл к коммиту, используйте команду *git add* с указанием пути к файлу или создайте ваш первый файл (*proba.txt*) в папке репозитория (Рисунок 2.10):



```
Software_Engineering — -zsh — 80x24
mi hail@MacBook-Air-Mi hail ~ % cd /user/
cd: no such file or directory: /user/
mi hail@MacBook-Air-Mi hail ~ % cd Software_Engineering
mi hail@MacBook-Air-Mi hail Software_Engineering % git add proba.txt
mi hail@MacBook-Air-Mi hail Software_Engineering %
```

Рисунок 2.10. Добавление одного файла

– Проверка статуса

После добавления файлов с помощью *git add*, вы можете использовать команду *git status*, чтобы увидеть, какие файлы были подготовлены к коммиту и какие файлы имеют неотслеживаемые изменения (Рисунок 2.11):

```

mihail@MacBook-Air-Mihail Software_Engineering % git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   Proba.txt

mihail@MacBook-Air-Mihail Software_Engineering %

```

Рисунок 2.11. Файл готов к коммиту

2.5. Фиксация изменений

Коммит в Git представляет собой фиксацию изменений в вашем проекте. Каждый коммит содержит снимок (snapshot) состояния файлов в определенный момент времени, включая все добавленные, измененные и удаленные файлы. Коммиты составляют историю изменений вашего проекта, позволяя вам отслеживать, кто, когда и какие изменения вносил.

Когда вы создаете коммит, вы также включаете описание (комментарий) к этому коммиту, объясняя, какие изменения были внесены и почему. Это помогает разработчикам и другим участникам проекта понимать, что было сделано в этом коммите без необходимости изучения кода.

– Как сделать коммит

Создайте первый коммит, фиксируя текущее состояние файлов. Используйте команду `git commit` с флагом `-m` для добавления описания коммита (Рисунок 2.12):

```

mihail@MacBook-Air-Mihail Software_Engineering % git commit -m "Первый коммит: начальная версия проекта"

```

Рисунок 2.12. Создание первого коммита

Результатом выполнения будет следующее (Рисунок 2.13):

```

[main (root-commit) 6486680] Первый коммит: начальная версия проекта
 1 file changed, 1 insertion(+)
 create mode 100644 Proba.txt
mihail@MacBook-Air-Mihail Software_Engineering %

```

Рисунок 2.13. Создание первого коммита

– Как просмотреть коммиты

Для просмотра коммитов и истории изменений в Git вы можете использовать команды, работающие с историей репозитория. Вот несколько команд, которые помогут вам просматривать коммиты:

– Просмотр истории коммитов:

Для просмотра списка коммитов в хронологическом порядке используйте команду `git log`. (Рисунок 2.14.) Эта команда покажет вам информацию о каждом коммите, включая идентификатор коммита, автора, дату и время коммита, а также описание.

```

mihail@MacBook-Air-Mihail Software_Engineering % git log
commit 6486680d93b9ec3e01ac57be71246c0426df8cad (HEAD -> main)
Author: vnika2003 <vnika2003@mail.ru>
Date:   Wed Aug 30 17:59:58 2023 +0500

```

```

    Первый коммит: начальная версия проекта
mihail@MacBook-Air-Mihail Software_Engineering %

```

Рисунок 2.14. Просмотр истории коммитов

- Ограничение количества выводимых коммитов:

По умолчанию *git log* может показывать много коммитов. Вы можете ограничить количество выводимых коммитов, используя параметр *-n*, где *n* - это количество коммитов, которое вы хотите видеть (Рисунок 2.15.)

```
mi hail@MacBook-Air-Mi hail Software_Engineering % git log -n 5

commit 6486680d93b9ec3e01ac57be71246c0426df8cad (HEAD -> main)
Author: vnika2003 <vnika2003@mail.ru>
Date:   Wed Aug 30 17:59:58 2023 +0500

Первый коммит: начальная версия проекта
mi hail@MacBook-Air-Mi hail Software_Engineering %
```

Рисунок 2.15. Ограничение количества выводимых коммитов

- Краткий вывод коммитов:

Если вы хотите более краткое представление каждого коммита, вы можете использовать параметр *--oneline*. Это выведет каждый коммит в одной строке, сокращенно отображая его идентификатор и описание (Рисунок 2.16.)

```
mi hail@MacBook-Air-Mi hail Software_Engineering % git log --oneline

6486680 (HEAD -> main) Первый коммит: начальная версия проекта
mi hail@MacBook-Air-Mi hail Software_Engineering %
```

Рисунок 2.16. Краткий вывод коммитов

- Графическое представление истории:

Для более наглядного представления истории коммитов, вы можете использовать команду *git log* с параметром *--graph*. Это покажет графическое дерево коммитов с ветвями и слияниями (Рисунок 2.17.)

```
mi hail@MacBook-Air-Mi hail Software_Engineering % git log --graph

* commit 6486680d93b9ec3e01ac57be71246c0426df8cad (HEAD -> main)
  Author: vnika2003 <vnika2003@mail.ru>
  Date:   Wed Aug 30 17:59:58 2023 +0500

Первый коммит: начальная версия проекта
mi hail@MacBook-Air-Mi hail Software_Engineering %
```

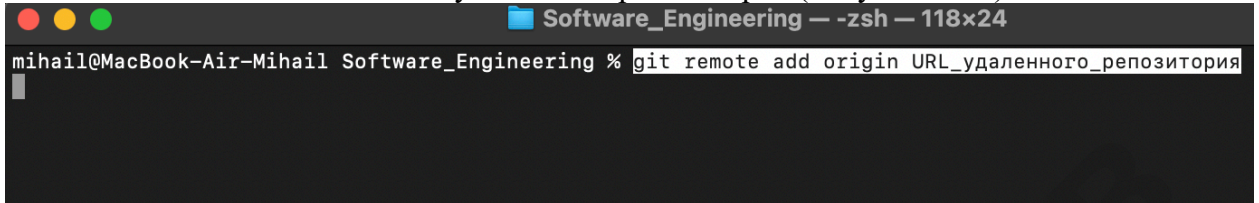
Рисунок 2.17. Графическое представление истории

2.6. Подключение к удаленному репозиторию

Для подключения к удаленному репозиторию в Git и начала работы с ним, вам нужно выполнить несколько шагов.

- Создайте учетную запись на платформе хостинга (например, GitHub, GitLab, Bitbucket). Если у вас еще нет учетной записи на хостинге репозитория (например, GitHub), создайте ее. Это понадобится для создания удаленного репозитория и совместной работы.

- Создайте новый удаленный репозиторий. Войдите в свою учетную запись на платформе хостинга и создайте новый репозиторий. Вам будут предоставлены инструкции по настройке нового репозитория.
- Связывание локального репозитория с удаленным. Перейдите в папку вашего локального проекта (если еще не находитесь там) с помощью терминала. Свяжите локальный репозиторий с удаленным с помощью команды `git remote add` и URL вашего удаленного репозитория (Рисунок 2.18.)

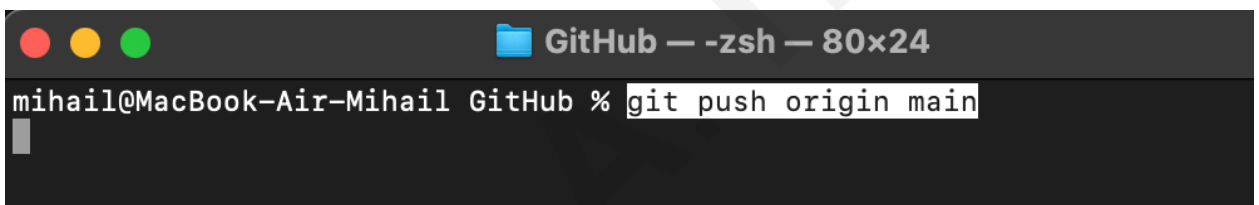


```
Software_Engineering — -zsh — 118x24
mihail@MacBook-Air-Mihail Software_Engineering % git remote add origin URL_удаленного_репозитория
```

Рисунок 2.18. Связывание локального репозитория с удаленным

Замените `URL_удаленного_репозитория` на фактический URL вашего удаленного репозитория (например, `https://github.com/ваш-логин/ваш-репозиторий.git`).

- Загрузите изменения на удаленный репозиторий. Если вы хотите загрузить текущие изменения из вашего локального репозитория на удаленный, используйте команду `git push` (Рисунок 2.19.)



```
GitHub — -zsh — 80x24
mihail@MacBook-Air-Mihail GitHub % git push origin main
```

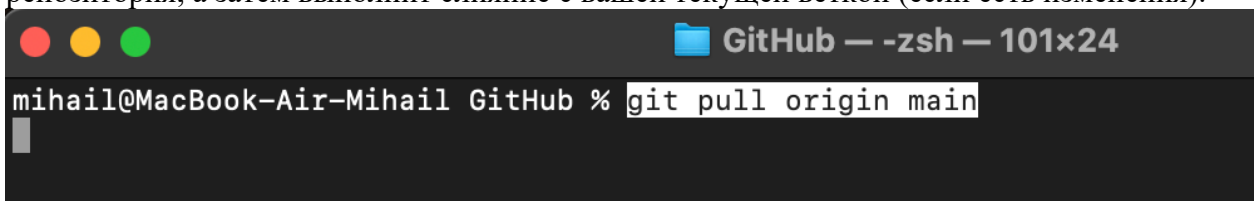
Рисунок 2.19. Загрузка изменений на удаленный репозиторий

Здесь `origin` - это имя удаленного репозитория, а `main` - это имя вашей главной ветки. Замените `main` на имя вашей ветки, если оно отличается.

Теперь вы связали локальный репозиторий с удаленным и можете работать с кодом и делиться изменениями с другими разработчиками через ваш удаленный репозиторий.

- Команда `git pull` используется для извлечения изменений из удаленного репозитория и объединения их с текущей локальной веткой. Это включает в себя два этапа: сначала команда `git fetch` получает изменения из удаленного репозитория, а затем команда `git merge` (или `git rebase`, если вы предпочитаете) объединяет изменения с вашей текущей веткой.

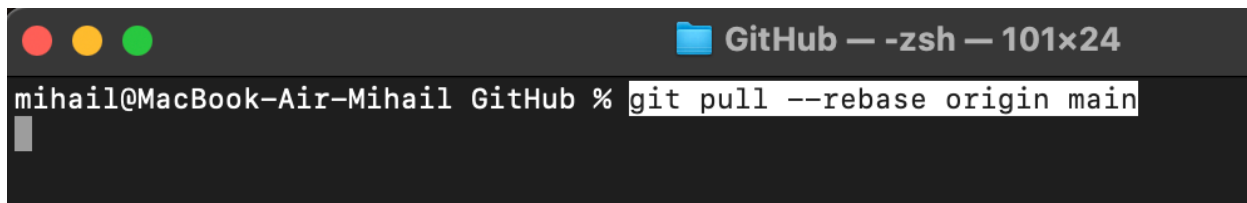
Извлечение изменений с помощью `git pull`. Вам нужно находиться в рабочей директории вашего локального репозитория и в той ветке, в которую вы хотите внести изменения. Затем выполните команду `git pull` с указанием имени удаленного репозитория и ветки. Например, для ветки `main` и удаленного репозитория с именем `origin` (Рисунок 2.20.) Эта команда выполнит сначала `git fetch` для получения изменений из удаленного репозитория, а затем выполнит слияние с вашей текущей веткой (если есть изменения).



```
GitHub — -zsh — 101x24
mihail@MacBook-Air-Mihail GitHub % git pull origin main
```

Рисунок 2.20. Извлечение изменений с помощью `git pull`

Как упоминалось ранее, команда *git pull* по умолчанию использует метод слияния (*git merge*). Если вы предпочитаете использовать перебазирование (*git rebase*), вы можете указать это явно. Это выполнит перебазирование изменений из удаленной ветки `'main'` на вашу текущую ветку (Рисунок 2.21.)

A terminal window titled "GitHub — -zsh — 101x24" with a dark background. The prompt is "mihail@MacBook-Air-Mihail GitHub %". The command "git pull --rebase origin main" is entered and highlighted in white. The cursor is at the end of the command line.

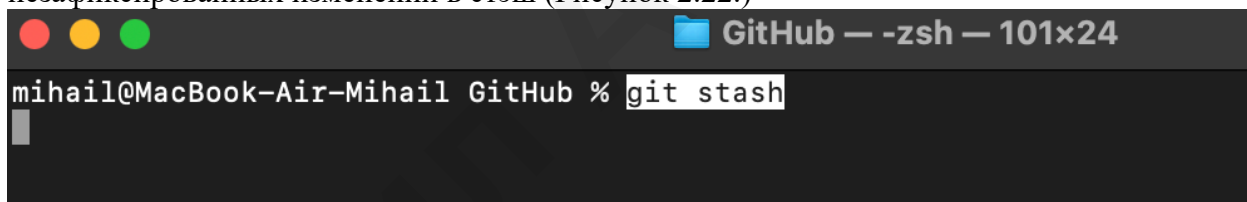
```
mihail@MacBook-Air-Mihail GitHub % git pull --rebase origin main
```

Рисунок 2.21. Указание метода слияния

Пожалуйста, обратите внимание, что перед выполнением операции *git pull* или *git pull --rebase*, рекомендуется убедиться, что у вас нет незафиксированных изменений в текущей ветке, чтобы избежать конфликтов при слиянии или перебазировании.

- Добавление изменений в стэш (stash) позволяет временно сохранить незафиксированные изменения в отдельном месте, чтобы вы могли переключиться на другую ветку, выполнить операции, и затем вернуться к сохраненным изменениям. Это полезно, когда вы хотите сохранить текущее состояние рабочей директории, но временно переключиться на другую задачу. Вот как это делается:

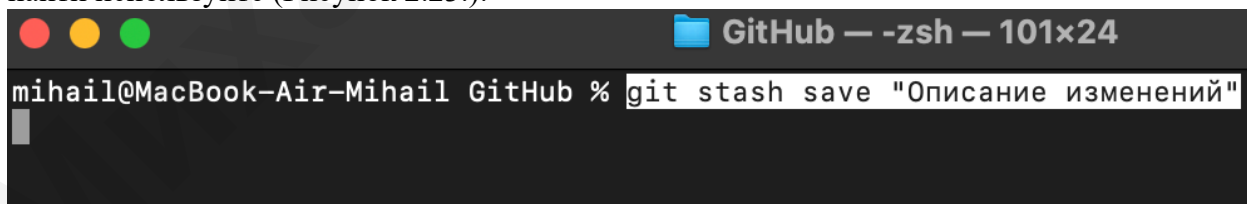
Добавление изменений в стэш. Используйте команду *git stash* для добавления всех незафиксированных изменений в стэш (Рисунок 2.22.)

A terminal window titled "GitHub — -zsh — 101x24" with a dark background. The prompt is "mihail@MacBook-Air-Mihail GitHub %". The command "git stash" is entered and highlighted in white. The cursor is at the end of the command line.

```
mihail@MacBook-Air-Mihail GitHub % git stash
```

Рисунок 2.22. Добавление изменений в стэш

Если вы хотите дать название своему стэшу, чтобы в дальнейшем было проще его найти используйте (Рисунок 2.23.):

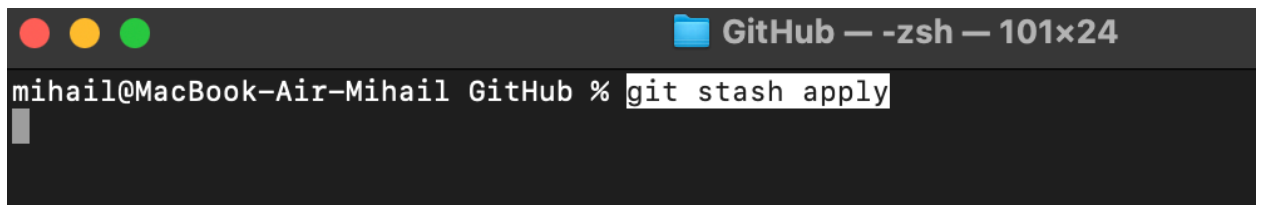
A terminal window titled "GitHub — -zsh — 101x24" with a dark background. The prompt is "mihail@MacBook-Air-Mihail GitHub %". The command "git stash save 'Описание изменений'" is entered and highlighted in white. The cursor is at the end of the command line.

```
mihail@MacBook-Air-Mihail GitHub % git stash save "Описание изменений"
```

Рисунок 2.23. Добавление изменений в стэш с названием

Переключение на другую ветку или выполнение операций. После того как вы сохранили изменения в стэше, вы можете переключиться на другую ветку, выполнить другие операции или даже перезапустить Git. Ваши незафиксированные изменения не будут мешать.

Возврат к сохраненным изменениям. Чтобы вернуть сохраненные изменения из стэша, выполните команду `'git stash apply'` (Рисунок 2.24.)

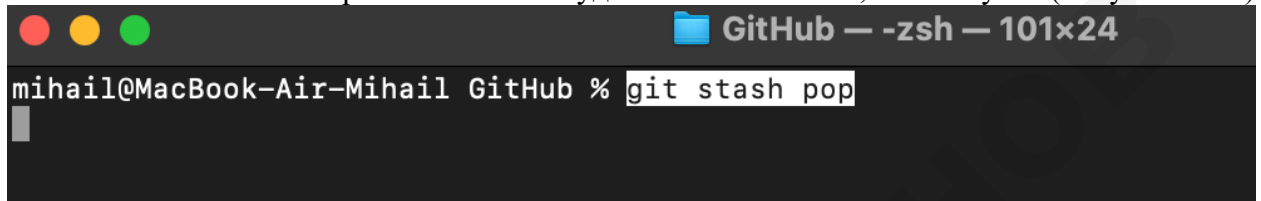
A terminal window titled "GitHub — -zsh — 101x24" with three colored window control buttons (red, yellow, green) on the left. The prompt is "mi hail@MacBook-Air-Mihail GitHub %". The command "git stash apply" is entered and highlighted in white text on a dark background. A cursor is visible at the end of the command line.

```
mi hail@MacBook-Air-Mihail GitHub % git stash apply
```

Рисунок 2.24. Возврат к сохраненным изменениям

Если у вас было несколько сохраненных стэшей, вы можете указать конкретный стэш по его индексу (например, `'stash@{1}'`).

Если вы хотите применить стэш и удалить его из стэша, используйте (Рисунок 2.25.)

A terminal window titled "GitHub — -zsh — 101x24" with three colored window control buttons (red, yellow, green) on the left. The prompt is "mi hail@MacBook-Air-Mihail GitHub %". The command "git stash pop" is entered and highlighted in white text on a dark background. A cursor is visible at the end of the command line.

```
mi hail@MacBook-Air-Mihail GitHub % git stash pop
```

Рисунок 2.25. Применение стэш и удаление его из стэша

Пожалуйста, обратите внимание что, если вы возвращаетесь к сохраненным изменениям, они могут вызвать конфликты с текущим состоянием. В этом случае вам придется разрешить конфликты вручную.

С использованием стэша вы можете более гибко управлять временными изменениями, не создавая постоянных коммитов, и безопасно переключаться между задачами.

2.7. Ветвление

Ветвление (branching) в системе контроля версий Git используется для организации и управления различными линиями разработки. Каждая ветка представляет собой отдельную линию изменений, которую можно развивать независимо от других веток. Ветвление позволяет вам работать над различными фичами, исправлениями ошибок или экспериментами параллельно, не вмешиваясь в основную линию разработки. Вот некоторые из основных применений ветвления:

- **Разработка фичей.** Вы можете создать новую ветку для работы над определенной фичей или задачей. Это позволяет вам внести и протестировать изменения, не влияя на стабильную версию вашего продукта. После завершения работы над фичей, вы можете объединить ветку с основной веткой.
- **Исправление ошибок (hotfixes).** Если обнаружены критические ошибки в выпущенной версии продукта, вы можете создать отдельную ветку для исправления этих ошибок. После тестирования и релиза, изменения могут быть включены как в текущую стабильную версию, так и в будущие версии продукта.
- **Эксперименты и исследования.** Вы можете создать временные ветки для проведения экспериментов или исследований. Это позволяет вам проверять новые идеи, изменения или подходы, не затрагивая основную линию разработки.

- Совместная разработка. Когда вы работаете в команде, каждый член команды может создавать свои собственные ветки для работы над определенными задачами. Это изолирует изменения разных разработчиков, позволяя каждому работать независимо и потом интегрировать свои изменения в общий код.
- Тестирование и обратная связь. Вы можете создавать ветки для тестирования различных сценариев и версий вашего продукта. Это позволяет вам проводить тщательное тестирование, не затрагивая основную линию разработки. Также ветвление помогает вам получать обратную связь от других разработчиков, не влияя на стабильную версию продукта.

В целом, ветвление дает вам гибкость и контроль над разработкой, позволяя разрабатывать и внедрять новые функции, исправлять ошибки и проводить эксперименты в структурированной и безопасной среде.

Создание новой ветки.

Выполните команду `git branch` с именем новой ветки, которую вы хотите создать (Рисунок 2.26.)

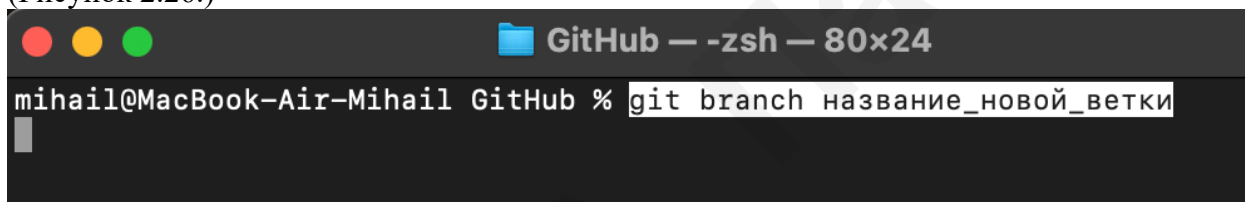
A screenshot of a terminal window titled "GitHub — -zsh — 80x24". The prompt is "mi hail@MacBook-Air-Mi hail GitHub %". The command "git branch название_новой_ветки" is entered and highlighted with a white selection box.

Рисунок 2.26. Создание новой ветки

Переключение на новую ветку.

После того как вы создали новую ветку, вы можете переключиться на нее с помощью команды `git checkout` (подходит для старых версий Git (Рисунок 2.27.) или `git switch` (более современный способ (Рисунок 2.28.):

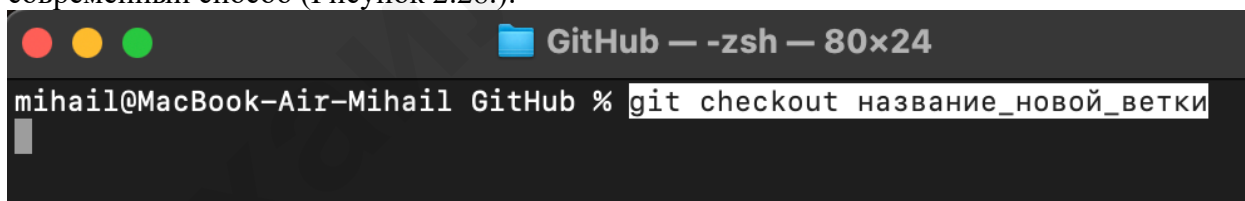
A screenshot of a terminal window titled "GitHub — -zsh — 80x24". The prompt is "mi hail@MacBook-Air-Mi hail GitHub %". The command "git checkout название_новой_ветки" is entered and highlighted with a white selection box.

Рисунок 2.27. Создание новой ветки с *checkout*

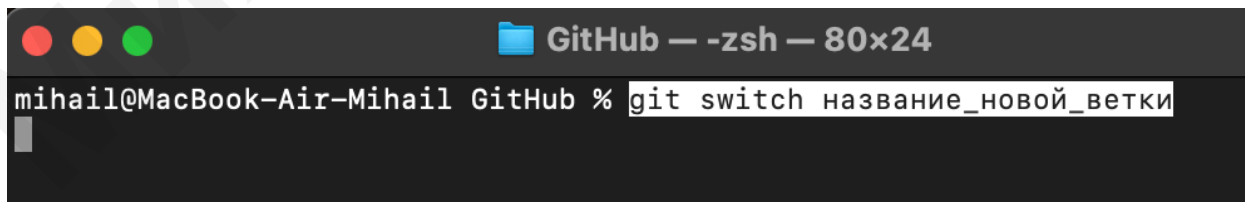
A screenshot of a terminal window titled "GitHub — -zsh — 80x24". The prompt is "mi hail@MacBook-Air-Mi hail GitHub %". The command "git switch название_новой_ветки" is entered and highlighted with a white selection box.

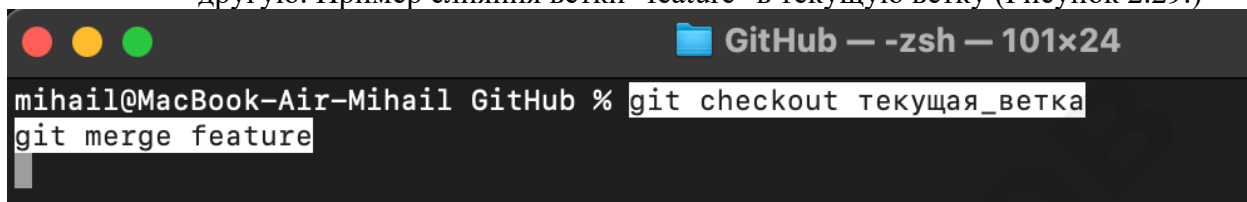
Рисунок 2.28. Создание новой ветки с *switch*

Теперь вы находитесь на новой ветке и можете вносить изменения, коммитить их и работать над функциональностью или исправлениями, специфичными для этой ветки. Создание отдельных веток помогает управлять изменениями, изолировать разные фичи и упрощает совместную работу в команде.

Слияние веток.

Слияние веток в Git позволяет объединить изменения из одной ветки в другую. Процесс слияния может происходить с использованием двух основных методов: слияние (*merge*) и перебазирование (*rebase*). Вот как это работает:

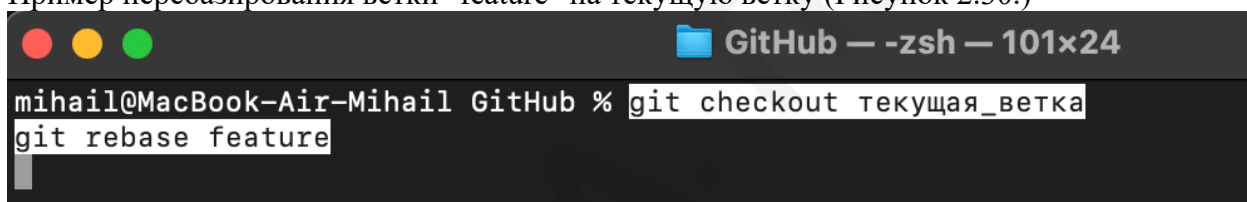
- Слияние (Merge). Слияние веток означает объединение изменений из одной ветки в другую. Процесс создает новый коммит, который представляет собой объединение изменений. Это полезно, когда вы хотите сохранить отдельные линии разработки, но также хотите включить изменения из одной ветки в другую. Пример слияния ветки "feature" в текущую ветку (Рисунок 2.29.)

A screenshot of a terminal window titled "GitHub — -zsh — 101x24". The prompt is "mihail@MacBook-Air-Mihail GitHub %". The user has entered the command "git checkout текущая_ветка" and "git merge feature".

```
mihail@MacBook-Air-Mihail GitHub % git checkout текущая_ветка
git merge feature
```

Рисунок 2.29. Пример слияния ветки

Перебазирование (Rebase). Перебазирование изменяет историю коммитов так, чтобы новые коммиты были включены непосредственно в конец целевой ветки. В результате история становится линейной, но вы перезаписываете историю коммитов в текущей ветке. Пример перебазирования ветки "feature" на текущую ветку (Рисунок 2.30.)

A screenshot of a terminal window titled "GitHub — -zsh — 101x24". The prompt is "mihail@MacBook-Air-Mihail GitHub %". The user has entered the command "git checkout текущая_ветка" and "git rebase feature".

```
mihail@MacBook-Air-Mihail GitHub % git checkout текущая_ветка
git rebase feature
```

Рисунок 2.30. Пример слияния ветки

Выбор между слиянием и перебазированием зависит от контекста и вашего рабочего процесса. Слияние создает дополнительные коммиты слияния, но сохраняет историю. Перебазирование делает историю линейной, но изменяет историю коммитов.

Важно также помнить о возможных конфликтах слияния или перебазирования. Конфликты могут возникнуть, если одни и те же файлы изменялись в обеих ветках. В этом случае Git предупредит вас о конфликтах, и вам нужно будет разрешить их вручную.

Прежде чем выполнить слияние или перебазирование, рекомендуется всегда убедиться, что вы находитесь в актуальной версии вашей целевой ветки и имеете резервную копию важных изменений.

2.8. Особенности применения «Фетч»

"Фетч" (fetch) в контексте системы контроля версий, такой как Git, представляет собой операцию, которая извлекает изменения и обновления из удаленного репозитория, но не объединяет их с вашей локальной веткой. Фетч обновляет информацию о состоянии удаленного репозитория и его ветках в вашем локальном репозитории, но не влияет на вашу текущую рабочую ветку. Рассмотрим основное назначение:

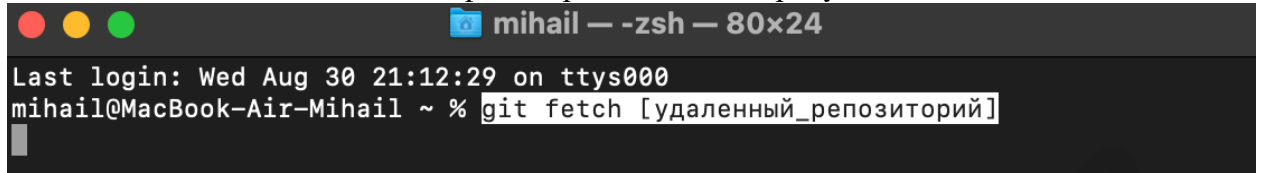
Синхронизация с удаленным репозиторием. Фетч позволяет вам убедиться, что ваш локальный репозиторий содержит актуальную информацию о состоянии удаленного репозитория. Это полезно, когда другие разработчики могли внести изменения в удаленный репозиторий, и вы хотите узнать об этих изменениях.

Просмотр изменений. Фетч также позволяет вам просматривать изменения, которые были сделаны в удаленной ветке, без их объединения в вашей текущей ветке. Это

может помочь вам принять решение о том, нужно ли вам сливать эти изменения или какие-то дополнительные действия.

Подготовка к слиянию. Фетч может использоваться в качестве предварительной операции перед слиянием (merge) или перебазируванием (rebase) вашей текущей ветки с удаленной. Это позволяет вам убедиться, что ваши изменения не будут конфликтовать с изменениями в удаленной ветке.

Команда для выполнения фетча представлена на рисунке 2.31.:



```
mi hail — -zsh — 80x24
Last login: Wed Aug 30 21:12:29 on ttys000
mi hail@MacBook-Air-Mi hail ~ % git fetch [удаленный_репозиторий]
```

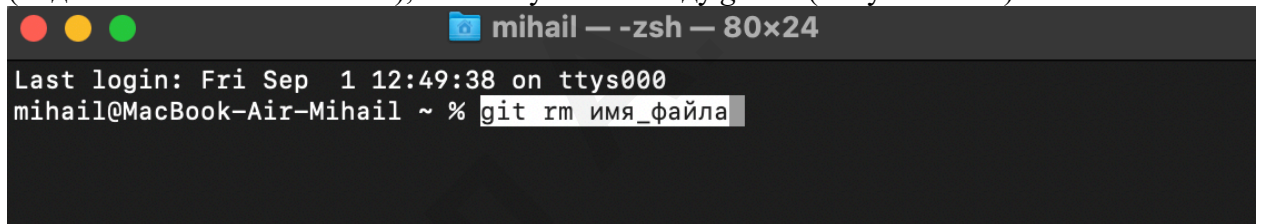
Рисунок 2.31. Команда для выполнения фетча

После выполнения фетча вы можете просмотреть изменения и, при необходимости, выполнить слияние или перебазирование с вашей текущей веткой для интеграции изменений из удаленного репозитория.

2.9. Удаление файлов, веток, локальных и удалённых репозитория

Удаление файлов, веток, локальных и удаленных репозитория в Git может потребоваться по разным причинам, таким как очистка ненужных данных или веток. Вот как выполняются эти операции:

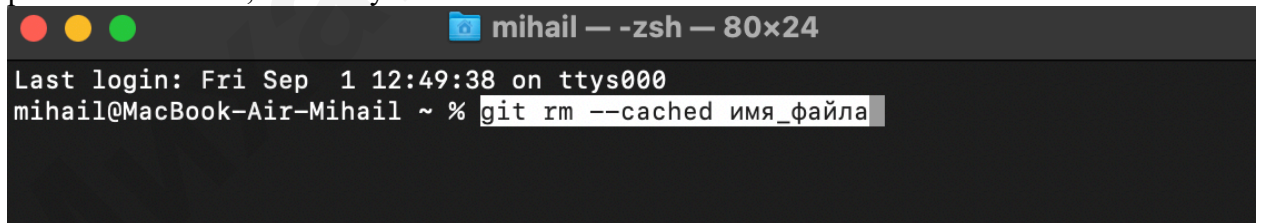
Удаление файла. Чтобы удалить файл из вашего рабочего каталога и из индекса (подготовленные изменения), используйте команду `git rm` (Рисунок 2.32.)



```
mi hail — -zsh — 80x24
Last login: Fri Sep 1 12:49:38 on ttys000
mi hail@MacBook-Air-Mi hail ~ % git rm имя_файла
```

Рисунок 2.32. Удаление файл из рабочего каталога и из индекса

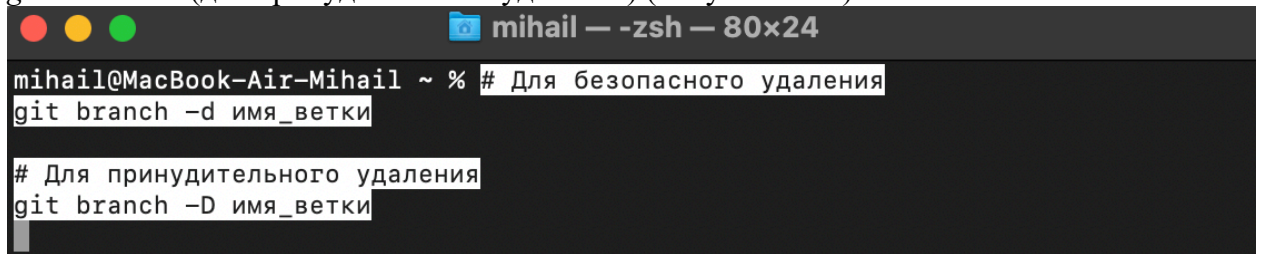
Если вы хотите удалить файл только из индекса (Рисунок 2.33.), но оставить его в рабочем каталоге, используйте:



```
mi hail — -zsh — 80x24
Last login: Fri Sep 1 12:49:38 on ttys000
mi hail@MacBook-Air-Mi hail ~ % git rm --cached имя_файла
```

Рисунок 2.33. Удаление файл из рабочего каталога и из индекса

Удаление локальной ветки. Для удаления локальной ветки используйте команду `git branch -d` (для безопасного удаления, проверяет, что ветка была полностью слита) или `git branch -D` (для принудительного удаления) (Рисунок 2.34.)

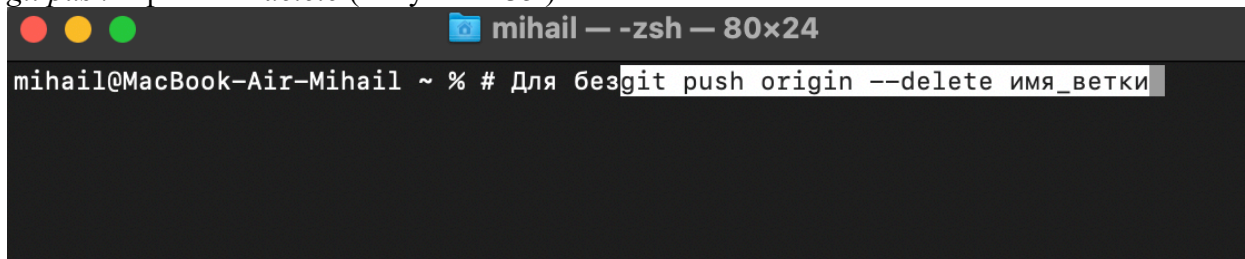


```
mi hail — -zsh — 80x24
mi hail@MacBook-Air-Mi hail ~ % # Для безопасного удаления
git branch -d имя_ветки

# Для принудительного удаления
git branch -D имя_ветки
```

Рисунок 2.34. Удаление локальной ветки

Удаление удаленной ветки. Для удаления удаленной ветки используйте команду `git push` с флагом `-delete` (Рисунок 2.35.)

A terminal window titled "mi hail — -zsh — 80x24" showing the command `git push origin --delete имя_ветки` being entered at the prompt `mi hail@MacBook-Air-Mi hail ~ %`.

```
mi hail@MacBook-Air-Mi hail ~ % # Для безgit push origin --delete имя_ветки
```

Рисунок 2.35. Удаление удаленной ветки

Удаление локального репозитория. Удаление локального репозитория означает удаление каталога, содержащего весь ваш локальный репозиторий. Просто удалите каталог с помощью команды `rm` (в Unix-подобных системах) или `rmdir` (в Windows) в терминале или проводнике.

Удаление удаленного репозитория. Удаление удаленного репозитория обычно означает удаление репозитория с хостинг-платформы, такой как GitHub или GitLab, и это обычно делается веб-интерфейсом платформы.

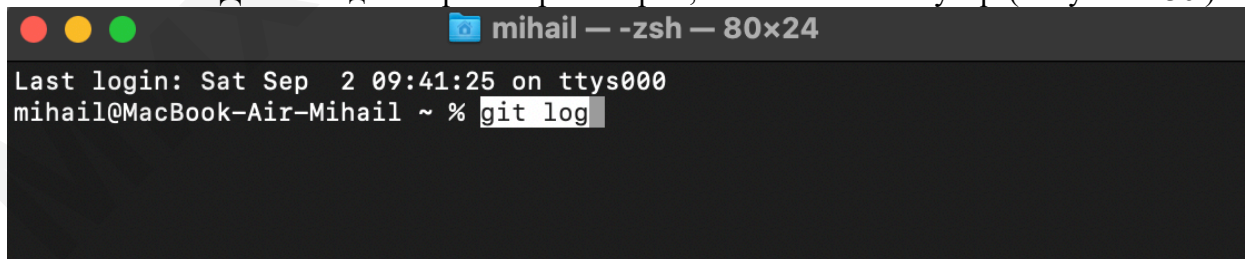
На GitHub, например, вы можете удалить удаленный репозиторий следующим образом:

- Перейдите на страницу вашего репозитория на GitHub.
- Нажмите на вкладку "Settings" (Настройки).
- Прокрутите вниз до раздела "Danger Zone" и нажмите на "Delete this repository"
- Введите имя вашего репозитория для подтверждения удаления.

2.10. Отслеживание изменений в коммитах

Для отслеживания изменений в коммитах в Git можно использовать несколько команд и инструментов. Вот некоторые из них:

git log. Команда `git log` выводит историю коммитов для текущей ветки. Она показывает информацию о каждом коммите, включая хэш коммита, автора, дату и сообщение коммита. Вы можете просматривать историю коммитов и искать интересные для вас изменения. Для выхода из просмотра истории, нажмите клавишу "q" (Рисунок 2.36.).

A terminal window titled "mi hail — -zsh — 80x24" showing the command `git log` being entered at the prompt `mi hail@MacBook-Air-Mi hail ~ %`. The output shows the last login time and the command being executed.

```
Last login: Sat Sep  2 09:41:25 on ttys000
mi hail@MacBook-Air-Mi hail ~ % git log
```

Рисунок 2.36. Команда `git log`

git diff. Команда `git diff` позволяет сравнивать изменения между коммитами или ветками. Вы можете использовать ее, чтобы увидеть, какие строки кода были добавлены, изменены или удалены.

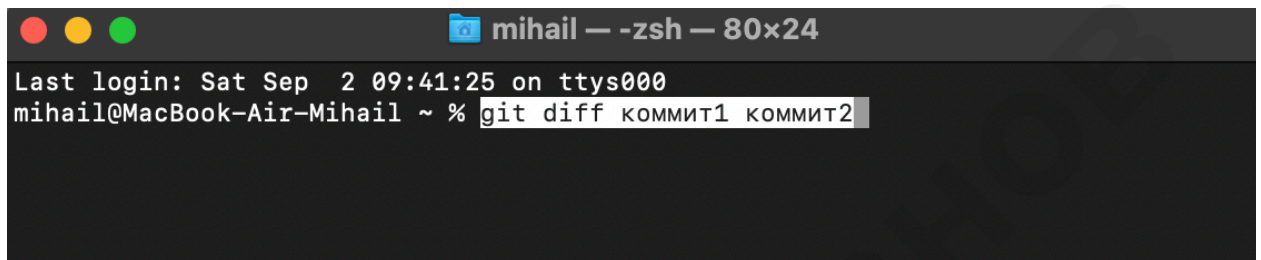
Например, чтобы посмотреть изменения между последним коммитом и текущим состоянием рабочей директории (Рисунок 2.37.).



```
mi hail — -zsh — 80x24
Last login: Sat Sep  2 09:41:25 on ttys000
mi hail@MacBook-Air-Mi hail ~ % git diff
```

Рисунок 2.37. Команда *git log*

Или чтобы посмотреть изменения между двумя коммитами (Рисунок 2.38.)



```
mi hail — -zsh — 80x24
Last login: Sat Sep  2 09:41:25 on ttys000
mi hail@MacBook-Air-Mi hail ~ % git diff КОММИТ1 КОММИТ2
```

Рисунок 2.38. Команда *git log* чтобы посмотреть изменения между двумя коммитами

GitHub и GitLab. Если вы используете хостинг-платформы Git, такие как GitHub или GitLab, вы можете просматривать изменения в коммитах прямо на веб-странице репозитория. Каждый коммит имеет страницу, на которой отображаются изменения, внесенные этим коммитом.

Графические клиенты. Существуют графические клиенты для Git, такие как Sourcetree, GitKraken и GitHub Desktop, которые предоставляют более удобный способ просматривать и анализировать изменения в коммитах. Они обычно предоставляют дружелюбный пользовательский интерфейс для визуализации истории коммитов и изменений в коде.

Важно уметь использовать команды `'git log'` и `'git diff'`, так как они предоставляют мощные средства для отслеживания изменений и анализа истории в вашем репозитории.

2.11. Возвращение файла к предыдущему (определенному) состоянию

Для возврата файла к предыдущему (определенному) состоянию в Git, вы можете использовать команду *git checkout*. Эта команда позволяет вам восстановить файл из определенного коммита.

Определите хэш коммита. Сначала вам нужно определить хэш (SHA-1) коммита, в состоянии которого вы хотите вернуть файл. Вы можете найти этот хэш, используя команду *git log*, чтобы просмотреть историю коммитов и найти нужный коммит. Найдите хэш коммита, который соответствует состоянию файла, которое вас интересует.

Используйте *git checkout*. После того как вы определили хэш коммита, используйте команду *git checkout*, чтобы восстановить файл из этого коммита. Замените `'<commit_hash>'` на фактический хэш коммита и `'<file_path>'` на путь к файлу (Рисунок 2.39.)

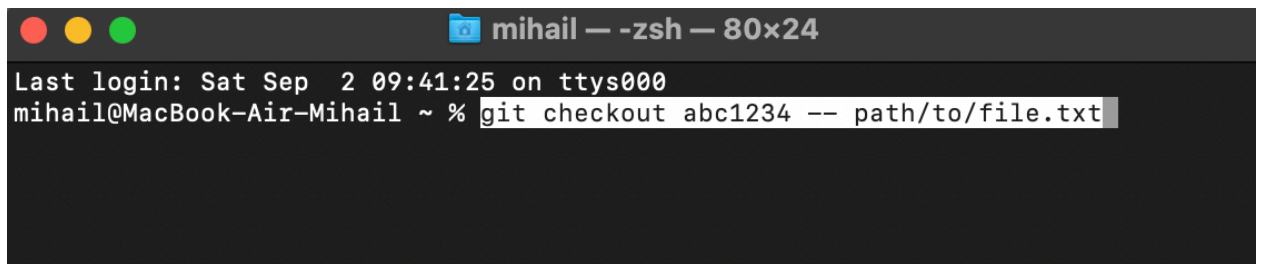
A terminal window titled 'mi hail — -zsh — 80x24'. The prompt is 'mi hail@MacBook-Air-Mi hail ~ %'. The command 'git checkout abc1234 -- path/to/file.txt' is entered and highlighted. The window also shows the last login time: 'Last login: Sat Sep 2 09:41:25 on ttys000'.

Рисунок 2.39. Команда заменит текущую версию файла на состояние из указанного коммита.

Коммит изменений. После восстановления файла к предыдущему состоянию, не забудьте зафиксировать этот измененный файл с помощью команды *git commit*, чтобы сохранить его состояние (Рисунок 2.40.). Обратите внимание, что *'git checkout'* заменит текущую версию файла, поэтому будьте осторожны и убедитесь, что вы вернули файл к правильному состоянию.

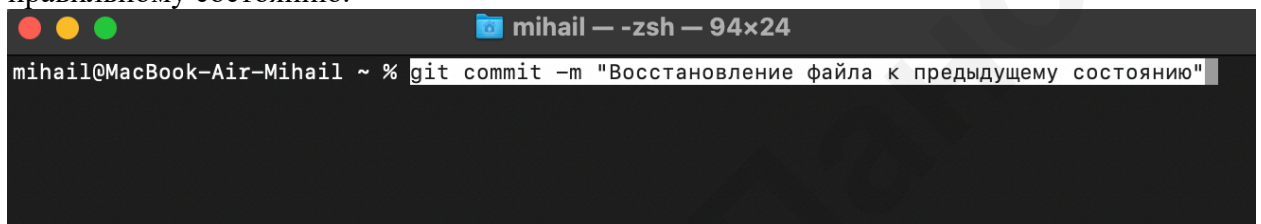
A terminal window titled 'mi hail — -zsh — 94x24'. The prompt is 'mi hail@MacBook-Air-Mi hail ~ %'. The command 'git commit -m "Восстановление файла к предыдущему состоянию"' is entered and highlighted.

Рисунок 2.40. Коммит изменений.

2.12. Возвращение к предыдущему коммиту

Для возврата к предыдущему коммиту в Git вы можете использовать команду *git reset*. В зависимости от того, какой коммит вы хотите сделать активным, есть несколько способов:

Сброс до предыдущего коммита (с потерей изменений). Если вы хотите отменить последний коммит и потерять все изменения, которые вы внесли в нем, выполните следующую команду (Рисунок 2.41.):

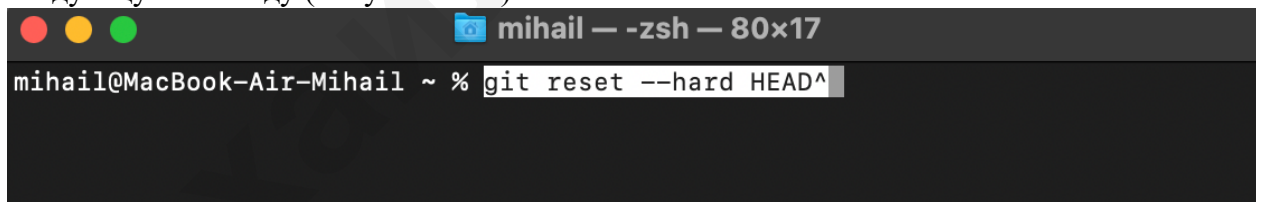
A terminal window titled 'mi hail — -zsh — 80x17'. The prompt is 'mi hail@MacBook-Air-Mi hail ~ %'. The command 'git reset --hard HEAD^' is entered and highlighted.

Рисунок 2.41. Сброс до предыдущего коммита (с потерей изменений).

Это сбросит указатель текущей ветки на коммит выше (предыдущий коммит), а также удалит все изменения, которые были внесены в последнем коммите.

Сброс до предыдущего коммита (с сохранением изменений в рабочей директории). Если вы хотите отменить последний коммит, но сохранить изменения в рабочей директории (в незафиксированном состоянии), выполните следующую команду (Рисунок 2.42.).

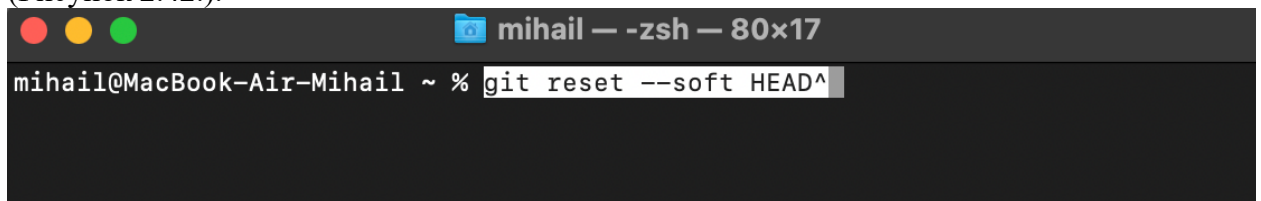
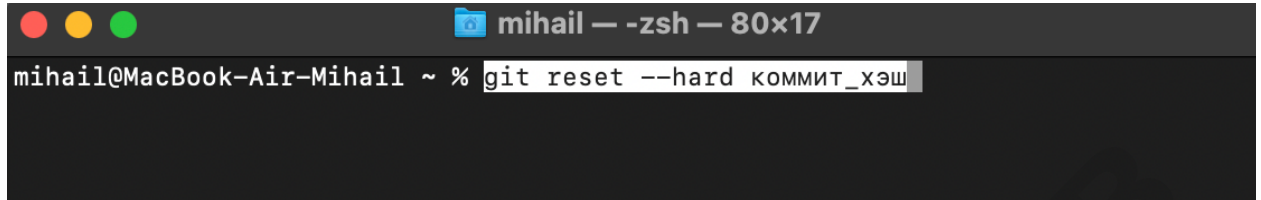
A terminal window titled 'mi hail — -zsh — 80x17'. The prompt is 'mi hail@MacBook-Air-Mi hail ~ %'. The command 'git reset --soft HEAD^' is entered and highlighted.

Рисунок 2.42. Сброс до предыдущего коммита (с сохранением изменений в рабочей директории).

Сброс до конкретного коммита. Если вы хотите вернуться к определенному коммиту (не обязательно предыдущему), вам нужно указать хэш этого коммита (Рисунок 2.43.). Это сбросит указатель текущей ветки и рабочей директории до состояния указанного коммита.

A terminal window titled 'mi hail — -zsh — 80x17' with a dark background. The prompt is 'mi hail@MacBook-Air-Mi hail ~ %'. The command 'git reset --hard коммит_хэш' is entered and highlighted in white text on a dark background.

```
mi hail@MacBook-Air-Mi hail ~ % git reset --hard коммит_хэш
```

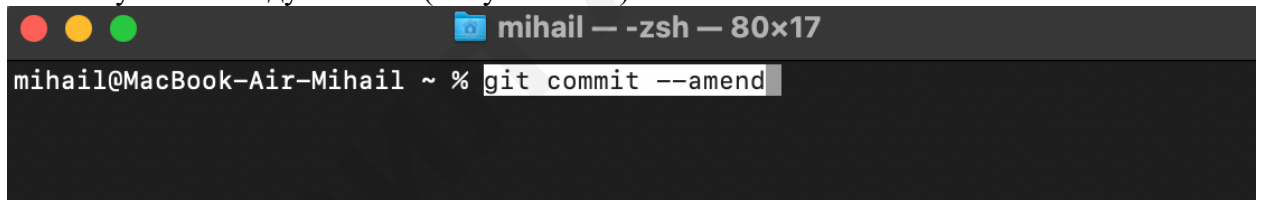
Рисунок 2.43. Сброс до конкретного коммита.

Пожалуйста, будьте осторожны при использовании команды `'git reset --hard'`, так как она может привести к потере данных. Всегда убедитесь, что вы понимаете, какие изменения будут потеряны, перед выполнением этой команды.

2.13. Исправление коммита

Исправление коммита в Git может потребоваться, если вы сделали ошибку в предыдущем коммите и хотите внести изменения или дополнения в него. Есть несколько способов это сделать, в зависимости от того, какой уровень коррекции вам нужен. Вот два основных способа:

Исправление последнего коммита. Если вы хотите внести изменения в последний коммит (например, изменить сообщение коммита или добавить пропущенные файлы), используйте команду `--amend` (Рисунок 2.44.)

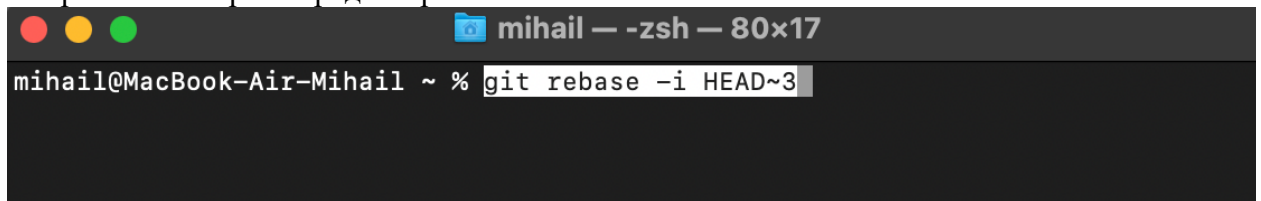
A terminal window titled 'mi hail — -zsh — 80x17' with a dark background. The prompt is 'mi hail@MacBook-Air-Mi hail ~ %'. The command 'git commit --amend' is entered and highlighted in white text on a dark background.

```
mi hail@MacBook-Air-Mi hail ~ % git commit --amend
```

Рисунок 2.44. Исправление последнего коммита.

После выполнения этой команды Git откроет текстовый редактор, где вы можете изменить сообщение коммита или добавить пропущенные файлы. После внесения изменений сохраните файл и закройте редактор. Коммит будет исправлен.

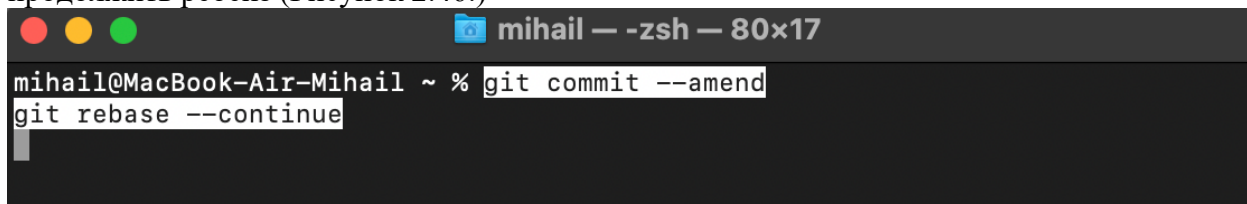
Исправление более раннего коммита. Если вы хотите изменить коммит, который не является последним, используйте команду `'git rebase -i'`. Например, если вы хотите исправить коммит два коммита назад (Рисунок 2.45.) Где `'HEAD~3'` указывает на коммит, к которому вы хотите вернуться. Git откроет текстовый редактор с списком коммитов. Замените слово `'pick'` (или `'edit'`) на `'edit'` рядом с коммитом, который вы хотите исправить. Сохраните и закройте редактор.

A terminal window titled 'mi hail — -zsh — 80x17' with a dark background. The prompt is 'mi hail@MacBook-Air-Mi hail ~ %'. The command 'git rebase -i HEAD~3' is entered and highlighted in white text on a dark background.

```
mi hail@MacBook-Air-Mi hail ~ % git rebase -i HEAD~3
```

Рисунок 2.45. Исправление более раннего коммита.

После завершения ребейза Git переключит вас на выбранный коммит. Вы можете внести изменения, которые вам нужны, сделать коммит с этими изменениями, а затем продолжить ребейз (Рисунок 2.46.)



```
mi hail — -zsh — 80x17
mi hail@MacBook-Air-Mi hail ~ % git commit --amend
git rebase --continue
```

Рисунок 2.46. Исправление более раннего коммита.

Пожалуйста, имейте в виду, что изменение коммитов может быть опасным, если другие разработчики уже скачали ваши изменения. В этом случае вам придется обсудить это с командой и, возможно, применить ребейз с использованием `--force` для отправки измененных коммитов в удаленный репозиторий.

2.14. Разрешение конфликтов при слиянии

Конфликты при слиянии (merge conflicts) возникают в Git, когда система не может автоматически объединить изменения из разных веток из-за конфликтующих изменений в одних и тех же строках кода. Разрешение конфликтов требует вмешательства разработчика. Шаги для разрешения конфликтов при слиянии:

Запустите команду слияния. Обычно конфликты возникают при выполнении операции слияния с другой веткой. (Рисунок 2.47.) При этом Git может сообщить о конфликтах и приостановить слияние.

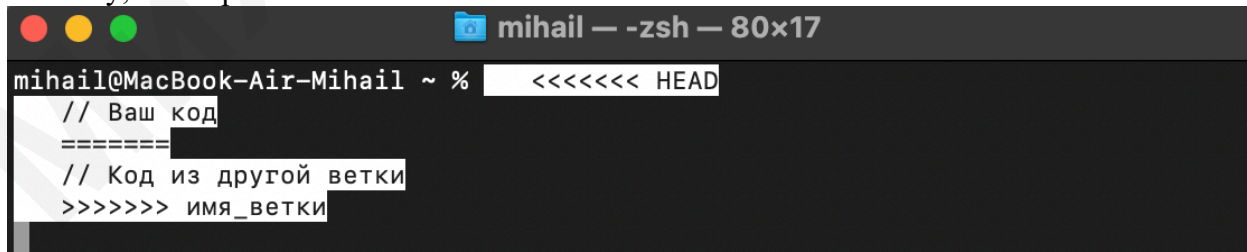


```
mi hail — -zsh — 80x17
mi hail@MacBook-Air-Mi hail ~ % git merge имя_ветки
```

Рисунок 2.47. Команда слияния.

Откройте файлы с конфликтами. Git пометит конфликтующие файлы в вашем рабочем каталоге. Вы можете использовать текстовый редактор, чтобы открыть такие файлы. В этих файлах вы увидите конфликтующие участки, помеченные специальными метками (Рисунок 2.48.).

Метка `<<<<<<< HEAD` указывает на текущую ветку, а метка `>>>>>>> имя_ветки` на ветку, с которой вы пытаетесь слить.

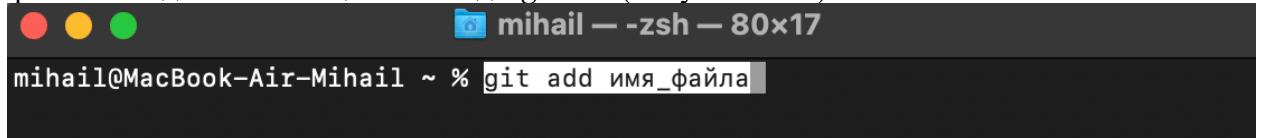


```
mi hail — -zsh — 80x17
mi hail@MacBook-Air-Mi hail ~ % <<<<<<< HEAD
// Ваш код
=====
// Код из другой ветки
>>>>>>> имя_ветки
```

Рисунок 2.48. Открытие файла с конфликтами

Разрешите конфликты. Внесите необходимые изменения в коде, чтобы устранить конфликт. Удалите метки с `<<<<<<<`, `=====`, `>>>>>>>`, и оставьте только код, который должен остаться. Решение зависит от конкретной ситуации. Вы можете сохранить изменения и закрыть файл.

Добавьте измененные файлы. После разрешения конфликта добавьте измененные файлы в индекс с помощью команды `git add` (Рисунок 2.49.)



```
mi hail — -zsh — 80x17
mi hail@MacBook-Air-Mihail ~ % git add имя_файла
```

Рисунок 2.49. Добавление изменённых файлов

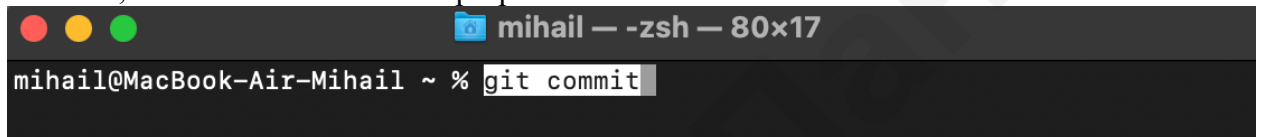
Или, если вы хотите добавить все измененные файлы `add` (Рисунок 2.50.)



```
mi hail — -zsh — 80x17
mi hail@MacBook-Air-Mihail ~ % git add .
```

Рисунок 2.50. Добавление всех изменённых файлов

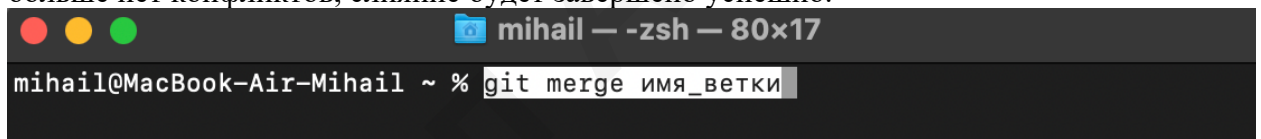
Завершите слияние. После разрешения всех конфликтов завершите слияние командой `git commit` (Рисунок 2.51.) Git автоматически создаст коммит, описывающий слияние, и включит в него ваши разрешенные изменения.



```
mi hail — -zsh — 80x17
mi hail@MacBook-Air-Mihail ~ % git commit
```

Рисунок 2.51. Завершение слияния

Продолжите операцию слияния (если нужно). Если у вас было приостановленное слияние, выполните команду слияния еще раз, чтобы продолжить (Рисунок 2.52.) Если больше нет конфликтов, слияние будет завершено успешно.



```
mi hail — -zsh — 80x17
mi hail@MacBook-Air-Mihail ~ % git merge имя_ветки
```

Рисунок 2.52. Продолжение операции слияния

Завершите операцию слияния. Если вы используете команду слияния через Git CLI, завершите операцию слияния. Например, после успешного слияния можно выполнить следующую команду (Рисунок 2.53.) для удаления сливаемой ветки, если она больше не нужна.



```
mi hail — -zsh — 80x17
mi hail@MacBook-Air-Mihail ~ % git branch -d имя_ветки
```

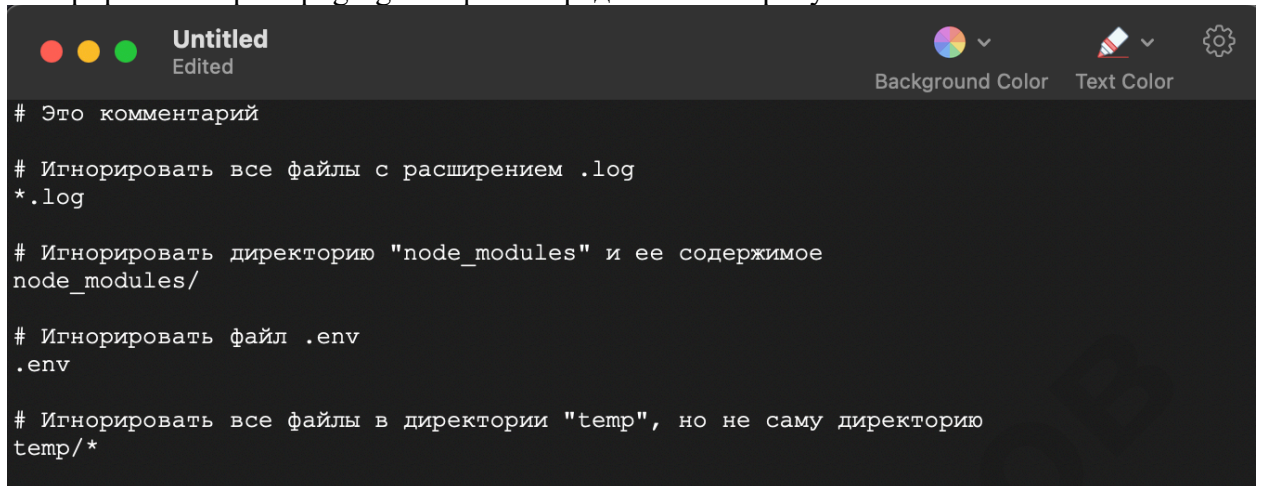
Рисунок 2.53. Завершение операции слияния

Таким образом, разрешение конфликтов при слиянии включает в себя изменение кода, чтобы устранить конфликты, и продолжение слияния после их разрешения.

2.15. Настройка. gitignore

.gitignore - это текстовый файл в корневой директории Git-репозитория, который используется для указания Git'у файлов и директорий, которые не должны быть отслеживаемыми системой контроля версий. Это позволяет исключить определенные файлы и директории из репозитория, так что они не будут автоматически добавлены в коммиты и не будут видны другим разработчикам при клонировании репозитория.

`.gitignore` содержит шаблоны имен файлов или директорий, которые Git должен игнорировать. Пример `.gitignore` файла представлен на рисунке 2.54.



```
# Это комментарий

# Игнорировать все файлы с расширением .log
*.log

# Игнорировать директорию "node_modules" и ее содержимое
node_modules/

# Игнорировать файл .env
.env

# Игнорировать все файлы в директории "temp", но не саму директорию
temp/*
```

Рисунок 2.54. Пример `.gitignore` файла

Преимущества использования `.gitignore`:

Исключение ненужных файлов. Вы можете избавиться от временных и промежуточных файлов, конфиденциальных данных, настроек среды и других файлов, которые не должны быть в репозитории.

Сокращение размера репозитория. Игнорируемые файлы не будут добавлены в репозиторий, что помогает снизить размер репозитория.

Соблюдение принципов безопасности. Некоторые файлы, такие как файлы с паролями или ключами API, могут быть конфиденциальными. `.gitignore` позволяет исключить их из репозитория.

Сохранение чистоты репозитория. При игнорировании временных и производных файлов можно поддерживать репозиторий более чистым и управляемым.

`.gitignore` файлы могут быть добавлены в репозиторий, чтобы обеспечить согласованность в игнорируемых файлах для всех участников проекта. Таким образом, `.gitignore` - это важный инструмент для настройки управления версиями в Git.

Git имеет огромную ценность в программной инженерии и является одним из наиболее популярных инструментов управления версиями. Его ценность проявляется во многих аспектах:

- **Управление версиями кода.** Git предоставляет мощные средства для отслеживания и управления версиями вашего кода. Вы можете сохранять историю изменений, переключаться между версиями, отменять изменения и даже восстанавливать код к предыдущим состояниям.
- **Совместная работа.** Git облегчает совместную работу над проектами. Множество разработчиков может работать над одним проектом, и Git помогает объединять их изменения, управлять конфликтами и поддерживать согласованность кодовой базы.
- **Ветвление и слияние.** Git позволяет создавать ветки (branches) для разработки разных функциональных возможностей или исправления различных ошибок параллельно. После тестирования изменений можно безопасно объединить их обратно в основную ветку.

- **История изменений.** Git сохраняет историю всех изменений, сделанных в коде. Это помогает разработчикам исследовать историю проекта, понимать, кто и что изменил, и отслеживать, когда и почему были внесены изменения.
- **Восстановление и восстановление данных.** Если что-то идет не так или код сломан, Git позволяет восстановить предыдущие стабильные версии, что делает его мощным инструментом для восстановления данных в случае сбоев.
- **Работа в оффлайне.** Git позволяет работать с репозиторием даже без доступа к интернету. Разработчики могут выполнять коммиты, ветвление и другие операции без подключения к удаленному серверу.
- **Эффективное управление конфликтами.** Git помогает в управлении конфликтами при слиянии изменений, что делает процесс слияния более понятным и управляемым.
- **Инструменты сотрудничества.** Сервисы хостинга, такие как GitHub, GitLab и Bitbucket, предоставляют инструменты для совместной работы над проектами, отслеживания ошибок, управления задачами и интеграции с другими инструментами разработки.
- **Открытый исходный код.** Git является свободным и открытым исходным кодом, что означает, что его исходный код доступен для изучения и улучшения сообществом разработчиков.
- **Популярность и экосистема.** Git является стандартом в индустрии разработки ПО, и вокруг него сформировалась богатая экосистема инструментов и ресурсов.

Общая ценность Git заключается в том, что он делает процесс разработки программного обеспечения более управляемым, совместимым и эффективным, что помогает разработчикам создавать более качественное и надежное программное обеспечение.