Git для любителей cherry-pick в VS Code

**Составитель**: М.К.Арабов (ЗАО Витакор)

Казань - 2024

**Введение**

Система контроля версий - неотъемлемый инструмент в арсенале современного разработчика. Она позволяет эффективно управлять изменениями в коде, отслеживать их и сотрудничать с другими участниками проекта. В этой книге мы рассмотрим одну из наиболее популярных платформ для работы с системой контроля версий - GitHub.

GitHub не только предоставляет возможность хранить git-репозитории с открытым кодом, но и служит площадкой для совместной работы над проектами, обмена опытом и обучения. Мы изучим основные принципы работы с GitHub, функционал платформы и способы эффективного использования ее возможностей.

Присоединяйтесь к нам в путешествии по миру GitHub, где каждый разработчик может найти вдохновение, обменяться знаниями и создать нечто удивительное.

Надеюсь, это введение соответствует содержанию книги. Если у вас есть какие-либо дополнения или изменения, пожалуйста, дайте знать!

**Что такое GitHub и для чего именно он нужен?**

GitHub это система управления проектами и версиями кода, а также платформа социальных сетей, созданная для разработчиков. Но для чего используется GitHub? Ну, среди прочего, это позволяет вам работать совместно с другими людьми по всему миру, планировать свои проекты и отслеживать свою работу.

GitHub неразрывно связан с системой контроля версий Git, которую разработчики устанавливают на персональный компьютер. На базе Git есть платформа GitHub, где хранятся git-репозитории с открытым кодом. Чтобы разобраться, как всё устроено в GitHub, нужно понимать, зачем нужна система контроля версий Git (сокращённо VCS от англ. Version Control System).

В целом GitHub — это социальная сеть для разработчиков, в которой можно найти проекты с открытым кодом от других разработчиков, практиковаться в написании кода и хранить свое портфолио.

**Что такое система контроля версий?**

Когда разработчики создают новый проект, они всегда продолжают вносить обновления в код. Даже после запуска проектов им всё равно нужно обновлять версии, исправлять ошибки, добавлять новые функции и т. д.

Система контроля версий помогает отслеживать изменения, внесённые в базу кода. Более того, он записывает, кто внёс изменения и может восстановить стёртый или изменённый код.

**Что такое Git**

Чтобы команда могла совместно работать над одним файлом, нужен сервис, где будет храниться папка с файлами — репозиторий. Сервер видит, какие изменения вносит каждый член команды, и контролирует, нет ли конфликтующих между собой частей кода. Такие VCS называют централизованными. Это удобнее, чем постоянно делать резервные версии. Но если сервер будет неисправен, все наработки пропадут.

Поэтому удобнее воспользоваться такими сервисами, как Git, — с распределёнными VCS. Файлы синхронизируются между ПК и центральным репозиторием. Каждый программист получает весь репозиторий, а не конкретные файлы. Можно как отправить изменения на сервер, так и скачать те, что внесли другие программисты.

Ревьюер — обычно коллега-программист — одобряет запрос на добавление кода, и тогда этот код становится частью репозитория. Если есть замечания, ревьюер пишет комментарии, и разработчик их учитывает.

В случае с распределёнными VCS актуальные файлы можно взять у любого из членов команды. И даже если что-то вышло из строя у одного из разработчиков, это не страшно.

**В чём разница между Git и GitHub**

С помощью Git программисты и разработчики ориентируются в коде и отслеживают изменения. Git помогает вернуть файлы в исходное состояние и видеть изменения, внесённые в определённый период. Разработчик выполняет разные команды (например, commit, push), а все изменения синхронизируются с центральным репозиторием.

Git — это система контроля версий, а GitHub — онлайн-сервис, по сути социальная сеть. Одна из основных целей GitHub — быть единым местом для проектов с исходным кодом. Предполагается, что пользователь делится чем-то полезным, а другие люди смогут участвовать в разработке.

С другими словами если Git — это сердце GitHub, то Hub — это его душа. Концентратор в GitHub — это то, что превращает командную строку, такую ​​как Git, в крупнейшую социальную сеть для разработчиков.

Помимо участия в определённом проекте, GitHub позволяет пользователям общаться с единомышленниками. Вы можете следить за людьми и смотреть, что они делают или с кем они общаются.

Ещё один вариант — использовать GitHub как хранилище проектов для портфолио: легко дать на них ссылку.

**Как создать проект на GitHub**

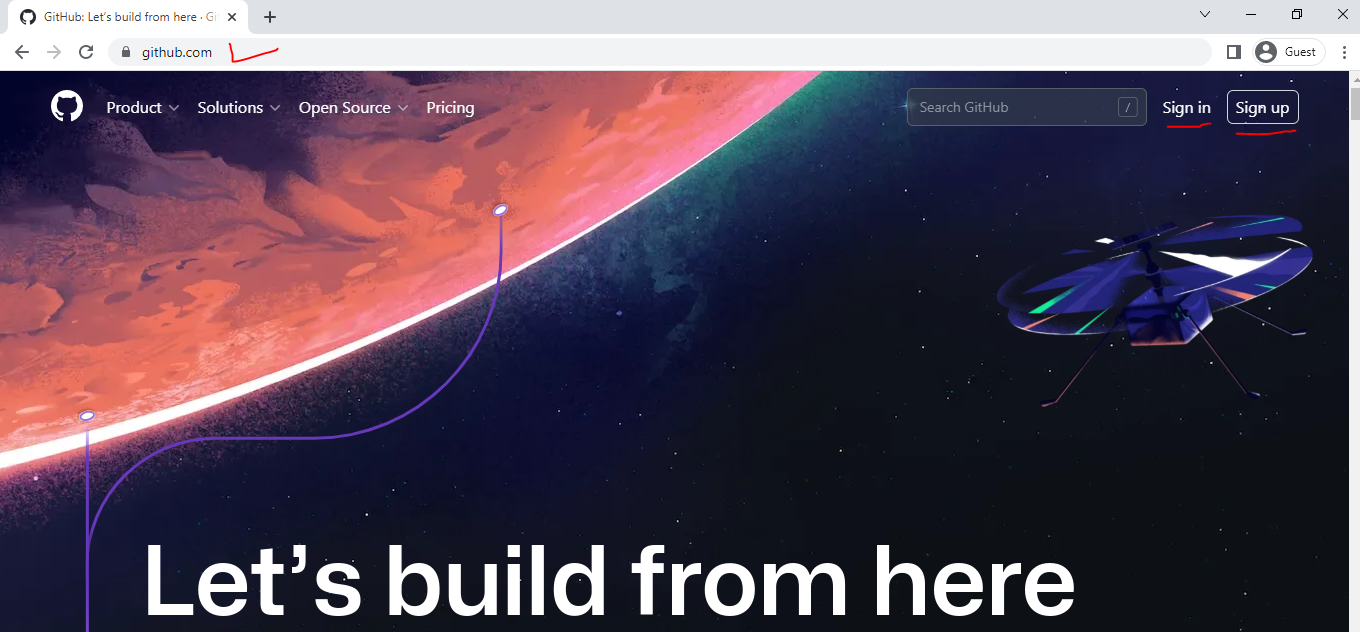
Чтобы выложить на GitHub проект, нужна регистрация на сайте сервиса. Первоначально необходимо указать адрес электронной почты, который станет логином для учётной записи. Сюда будут приходить все уведомления, и с помощью электронной почты можно будет восстановить доступ к GitHub в случае необходимости.

**Создание учетной записи GitHub**

С помощью учетной записи GitHub вы получите полную поддержку GitHub  для управления кодом и совместной работы с другими пользователями в проектах разработки.

Если у вас еще нет учетной записи GitHub, создайте ее.

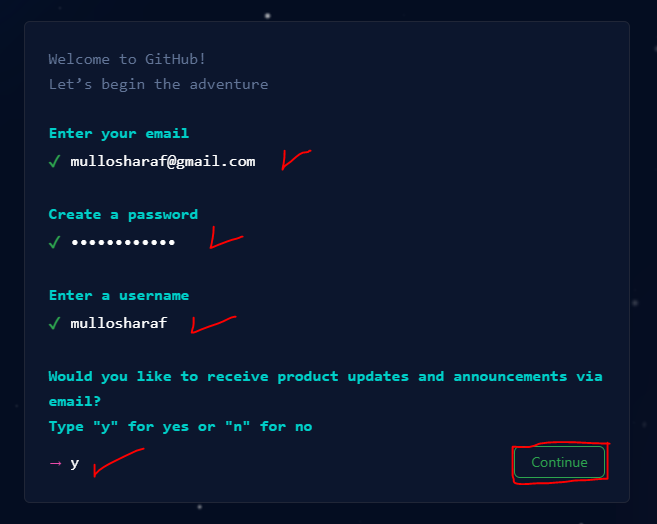
1. Откройте https://github.com в веб-браузере.



1. Нажмите кнопку «Войти(Sign in)» и выберите «Создать новую учетную запись(Create new account)».
2. Введите свой адрес электронной почты и нажмите кнопку «Continue».



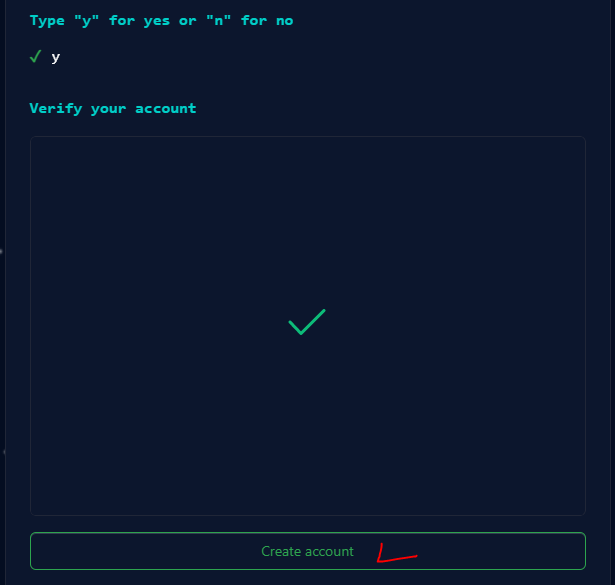
1. **Создайте пароль для новой учетной** записи GitHub и **введите имя пользователя**. Затем выберите, хотите ли вы получать обновления и объявления по электронной почте, а затем нажмите кнопку «Continue».



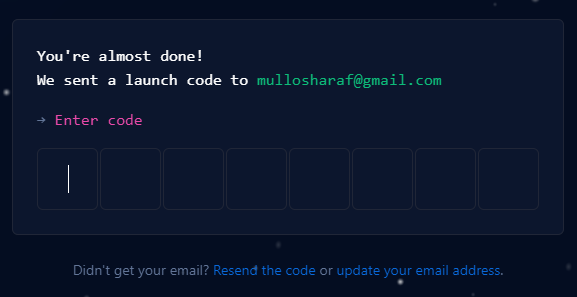
В результате откроется следующее окно



1. В этом окне нужно решить головоломку. Для этого нажмите «Start puzzle» и следуйте инструкциям.
2. После решение головоломки нажмите кнопку **«Create an account**».

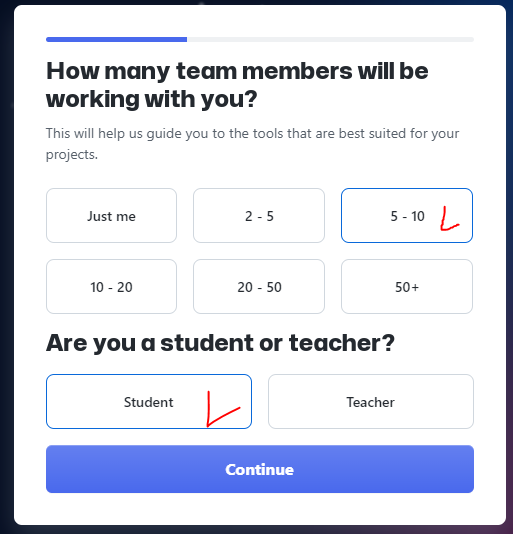


1. Затем GitHub отправляет код запуска на ваш адрес электронной почты. Введите код запуска в диалоговом окне **ввода кода** и нажмите клавишу **ВВОД**.

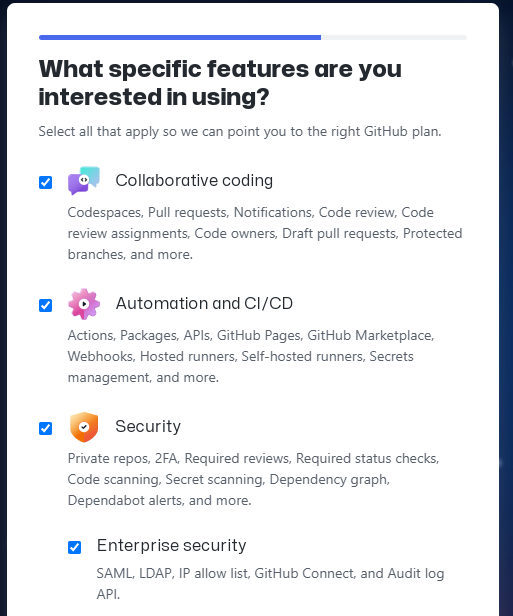


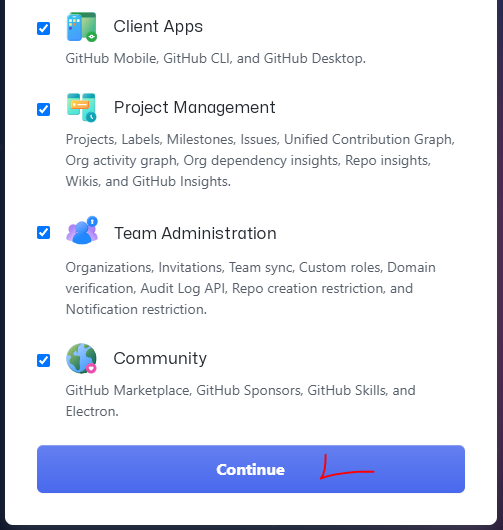
1. GitHub задает некоторые вопросы, которые помогут вам адаптировать свой опыт. Выберите ответы, которые применяются к вам в следующих диалогах:

* **Сколько участников команды будет работать с вами?**
* **Какие специальные функции вы хотите использовать?**



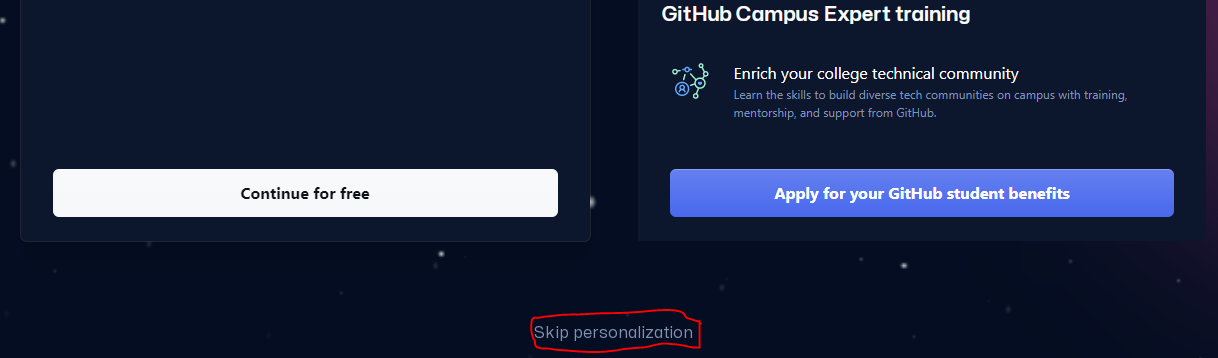
Здесь, например, я выбрал 5-10 команд и представился студентам. Далее нажмите кнопку «Continue».



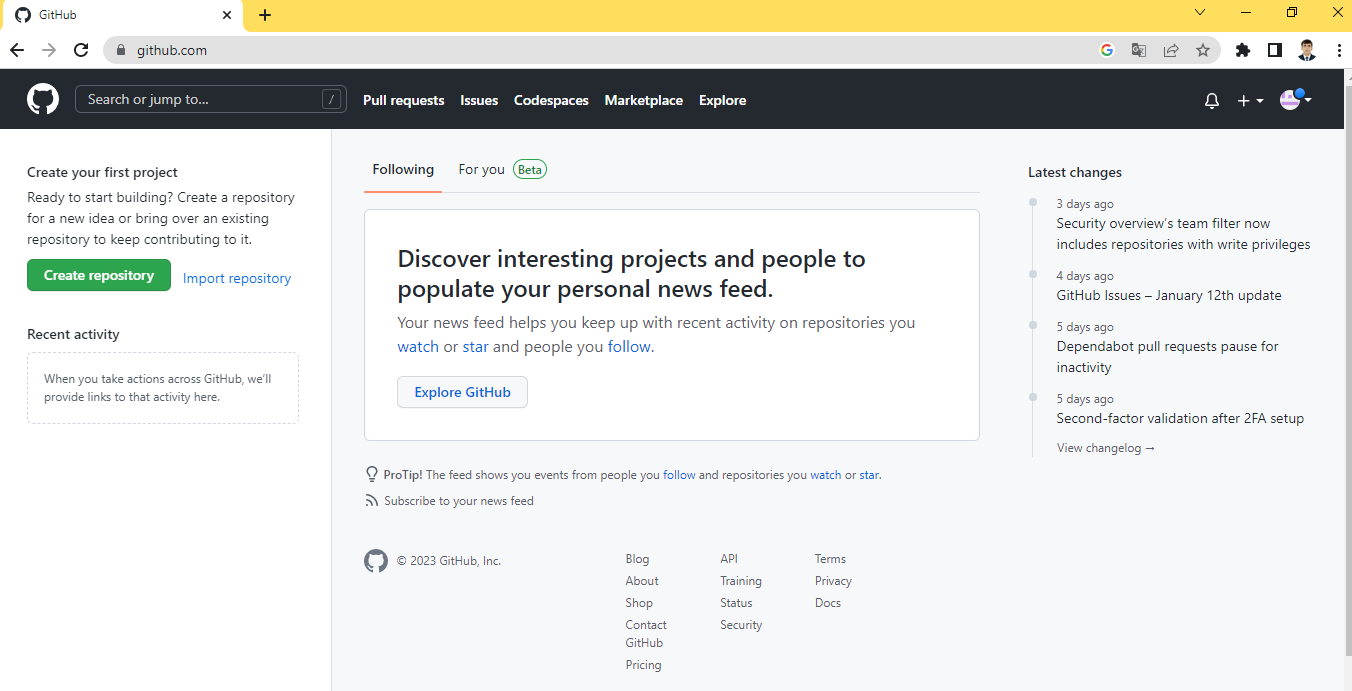


В этом окне выберите необходимые пункты и нажмите кнопку «Continue».

1. На экране **совместной работы и отправки команд** можно выбрать, хотите ли вы использовать бесплатную учетную запись или учетную запись команды. Чтобы выбрать **бесплатную** учетную запись, нажмите кнопку «Skip personalization».



1. GitHub открывает персонализированную страницу в браузере.



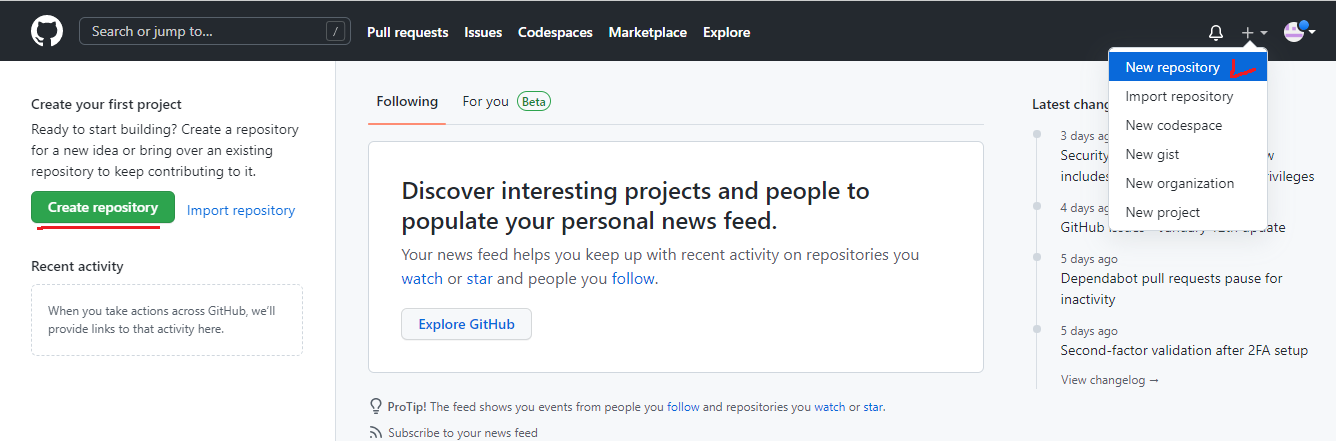
Поздравляем! Вы успешно создали учетную запись GitHub.

**Создание репозитория**

Для того чтобы загрузить проект на GitHub, нужно начать с репозитория. Репозиторий используют для организации одного проекта. В нём содержатся файлы с кодом, необходимые для разработки проекта. Есть приватные и публичные репозитории. К приватным можно подключить ограниченное число человек, публичные легко находятся в поиске другими пользователями.

Репозитории могут включать файл README с информацией о проекте. Для загрузки на GitHub своего репозитория добавлять файл README не нужно. В этом случае нужен именно пустой репозиторий, куда будет залит существующий код. С пустым репозиторием на GitHub отобразит подсказку, как создать или клонировать репозиторий.

Чтобы загрузить или создать репозиторий, воспользуйтесь выпадающим меню в правом верхнем углу на любой из страниц GitHub и выберите New repository.



Чуть позже рассмотрим создание репозитория из Visual Studio Code.

**Что такое репозиторий простыми словами**

Репозиторий — это хранилище всех версий кода. Он бывает трех видов:

* Локальный — расположен на одном компьютере, и работать с ним может только один человек.
* Централизованный — расположен на сервере, куда имеют доступ сразу несколько программистов.
* Распределенный — самый удобный вариант с облачным хранилищем. Главный репозиторий хранится в облаке, а его локальные копии — у разработчиков на компьютерах. Когда программист вносит правки в локальную версию, ее можно синхронизировать с удаленной. Получается, что в облаке всегда актуальный код.

Для работы с распределенными репозиториями нужен удобный сервис. Самые популярные — GitHub, GitLab и Bitbucket. У них понятный интерфейс, в котором можно управлять проектом, добавлять новые объекты и искать общедоступные репозитории.

На первом этапе рассмотрим работу с локальным репозиторием

**Что такое Гит?**

Git — это *распределенная система контроля версий* (DVCS). Давайте немного разберемся и посмотрим, что это значит.

### **Контроль версий**

Система *контроля версий* (VCS) — это набор инструментов, отслеживающих историю набора файлов. Это означает, что вы можете указать вашей системе VCS (в нашем случае Git) сохранять состояние ваших файлов в любой момент. Затем вы можете продолжить редактирование файлов и сохранить это состояние. Сохранение состояния аналогично созданию резервной копии вашего рабочего каталога. При использовании Git мы называем это сохранение состояния *фиксацией* .

Когда вы делаете коммит в Git, вы добавляете сообщение о коммите, которое на высоком уровне объясняет, какие изменения вы внесли в этот коммит. Git может показать вам историю всех коммитов и их сообщений. Это обеспечивает полезную историю проделанной вами работы и действительно может помочь точно определить, когда в систему проникла ошибка.

Помимо отображения журнала внесенных вами изменений, Git также позволяет сравнивать файлы между различными коммитами. Как я упоминал ранее, Git также позволит вам без особых усилий вернуть любой файл (или все файлы) в более раннюю фиксацию.

### **Распределенный контроль версий**

Хорошо, это *система контроля версий* . Какова *распределенная* часть? Вероятно, проще всего ответить на этот вопрос, начав с небольшой истории. Ранние системы контроля версий работали, сохраняя все эти коммиты локально на жестком диске. Эта коллекция коммитов называется *репозиторием* . Это решило проблему «мне нужно вернуться туда, где я был», но не очень хорошо масштабировалось для команды, работающей над той же кодовой базой.

Когда начали работать более крупные группы (и сетевое взаимодействие стало более распространенным), системы контроля версий изменились и теперь хранят репозиторий на центральном сервере, который используется многими разработчиками. Хотя это решило множество проблем, оно также создало новые, например, блокировку файлов.

Следуя примеру некоторых других продуктов, Git отказался от этой модели. У Git нет центрального сервера с окончательной версией репозитория. У всех пользователей есть полная копия репозитория. Это означает, что иногда бывает сложно вернуть всех разработчиков на одну страницу, но это также означает, что разработчики могут большую часть времени работать в автономном режиме, подключаясь к другим репозиториям только тогда, когда им нужно поделиться своей работой.

Последний абзац на первый взгляд может показаться немного запутанным, потому что многие разработчики используют GitHub в качестве *центрального репозитория* , из которого каждый должен брать данные. Это правда, но Git этого не навязывает. В некоторых случаях просто удобно иметь центральное место для обмена кодом. Полный репозиторий по-прежнему хранится во всех локальных репозиториях, даже если вы используете GitHub.

**Понимание того, как работает Git**

Git работает, позволяя нам делать снимок текущего состояния всех файлов в папке нашего проекта. Каждый раз, когда мы сохраняем один из этих снимков, мы делаем коммит в Git. Затем цикл повторяется, и Git создает новые снимки, показывая, как наш проект выглядел в любой момент.

Git был создан в 2005 году Линусом Торвальдсом, создателем ядра Linux. Git - это проект с открытым исходным кодом, лицензированный под GNU General Public License (GPL) v2. Изначально он был создан для упрощения разработки ядра из-за отсутствия подходящей альтернативы.

Общий рабочий процесс для создания коммита в Git и сохранения различных снимков проходит через следующие этапы:

* Изменение файлов (**Change**): Внесите необходимые изменения в файлы вашего проекта.
* Добавление изменений в индекс (**Stage**): Используйте команду git add, чтобы добавить изменения в индекс. Например, git add файл\_1 файл\_2 добавит файлы файл\_1 и файл\_2 в индекс.
* Создание коммита (**Commit**): Используйте команду git commit -m "Ваше сообщение коммита", чтобы создать коммит с добавленными изменениями. Ваше сообщение должно четко описывать сделанные изменения.

Как упоминается на третьем этапе, Git использует специальную базу данных, называемую репозиторием. Эта база данных хранится внутри директории вашего проекта в папке с именем .git.

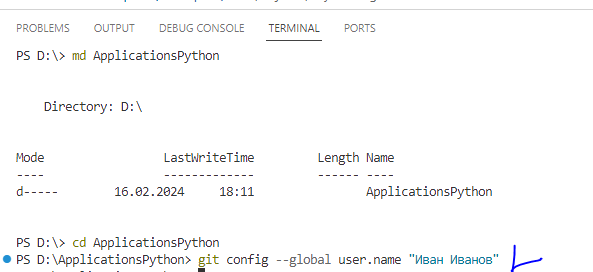
## Основное использование

Теперь, когда мы поговорили о том, что такое Git в целом, давайте рассмотрим пример и посмотрим на него в действии. Мы начнем с работы с Git только на нашей локальной машине. Как только мы это освоим, мы добавим GitHub и объясним, как с ним можно взаимодействовать.

### **Создание нового репо**

Чтобы работать с Git, вам сначала нужно сообщить ему, кто вы. Вы можете установить свое имя пользователя с помощью git config команды:

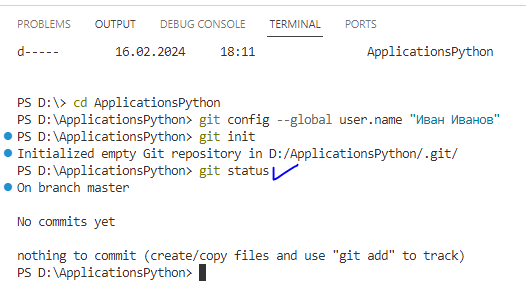
git config --global user.name "Иван Иванов"



Как только это будет настроено, вам понадобится репозиторий для работы. Создать репозиторий очень просто. Используйте git init команду в каталоге:

git init

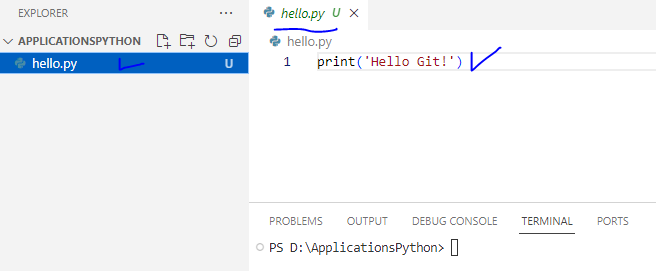
Если у вас есть репозиторий, вы можете спросить об этом Git. Команда Git, которую вы будете использовать чаще всего, — git status. Попробуйте это сейчас:



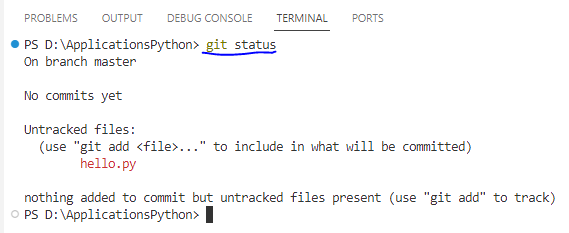
Это покажет вам пару бит информации: в какой ветке вы находитесь master(о ветках мы поговорим позже) и что вам *нечего коммитить* . Эта последняя часть означает, что в этом каталоге нет файлов, о которых Git не знает. Это хорошо, поскольку мы только что создали каталог.

### **Добавление нового файла**

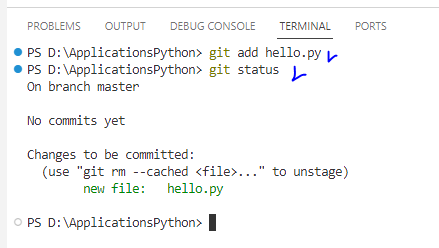
Теперь создайте файл, о котором Git не знает. В вашем любимом редакторе создайте файл hello.py, содержащий только оператор печати.



Теперь, если вы запустите git status еще раз, вы увидите другой результат:



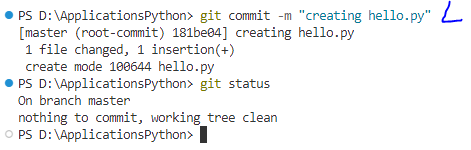
Теперь Git видит новый файл и сообщает, что он *не отслеживается*. Это всего лишь способ Git сказать, что файл не является частью репозитория и не находится под контролем версий. Мы можем это исправить, добавив файл в Git. Используйте git addкоманду, чтобы это произошло:



Теперь Git знает об этом hello.py и перечисляет его в списке *изменений, которые необходимо зафиксировать* . Добавление файла в Git перемещает его в *промежуточную область* (обсуждается ниже) и означает, что мы можем зафиксировать его в репозитории.

### **Внесение изменений**

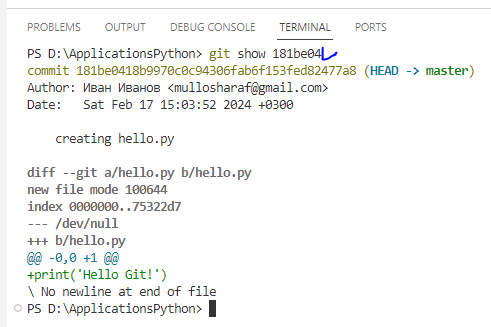
Когда вы *фиксируете* изменения, вы указываете Git сделать снимок этого состояния в репозитории. Сделайте это сейчас, используя git commit команду. Этот –m параметр указывает Git использовать следующее сообщение о фиксации. Если вы не используете -m, Git откроет вам редактор для создания сообщения о фиксации. В общем, вы хотите, чтобы ваши сообщения о коммите отражали то, что изменилось в коммите:



Вы можете видеть, что команда фиксации вернула кучу информации, большая часть которой не так уж и полезна, но она сообщает вам, что изменился только один файл (что имеет смысл, поскольку мы добавили один файл). Он также сообщает вам *SHA* коммита (181be04). Немного позже мы поговорим о SHA.

Повторный запуск git status команды показывает, что у нас есть *чистый* рабочий каталог, а это означает, что все изменения сохраняются в Git.

Для просмотра дополнительной информации о коммите можно использовать команду git show. Например, выполнение git show 181be04 позволит вам просмотреть информацию о конкретном коммите. В выводе будут отображены данные об авторе, дате, сообщении коммита и изменениях, внесенных в этом коммите. Эту команду можно выполнить в терминале Git для получения подробностей о коммите с указанным идентификатором.



**Область подготовки к коммиту (The Staging Area)**

В отличие от многих систем контроля версий, в Git есть область подготовки (часто называемая индексом). Область подготовки - это то, как Git отслеживает изменения, которые вы хотите включить в свой следующий коммит. Когда мы выполнили git add выше, мы сказали Git, что хотим переместить новый файл hello.py в область подготовки. Это изменение отразилось в git status. Файл перешел из раздела "untracked" в раздел "to be committed" вывода.

Обратите внимание, что область подготовки отражает точное содержимое файла на момент выполнения git add. Если вы его измените снова, файл будет отображаться как в подготовленной, так и в неподготовленной частях вывода статуса.

На любом этапе работы с файлом в Git (предполагая, что он уже был зафиксирован один раз), может быть три версии файла, с которыми вы можете работать:

* версия на вашем жестком диске, которую вы редактируете
* другая версия, которую Git сохраняет в вашей области подготовки
* последняя версия, зафиксированная в репозитории

Все три могут быть разными версиями файла. Перемещение изменений в область подготовки и затем их фиксация приводит все эти версии обратно в синхронизацию.

Когда я начинал работать с Git, мне казалась область подготовки немного запутанной и немного раздражающей. Казалось, что это добавляет лишние шаги к процессу без каких-либо польз. Когда вы только учитесь Git, это действительно так. Позже будут ситуации, когда вы начнете действительно ценить наличие этой функциональности. К сожалению, эти ситуации выходят за рамки этого руководства.

**.gitignore**

Команда статуса очень удобна, и вы будете часто ее использовать. *Но иногда вы обнаружите, что в неотслеживаемом* разделе отображается множество файлов, которые вы хотите, чтобы Git просто не видел. Вот тут-то и .gitignore появляется файл.

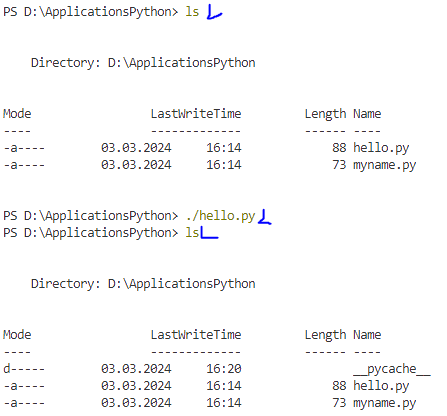
Давайте рассмотрим пример. Создайте новый файл Python в том же каталоге с именем myname.py.

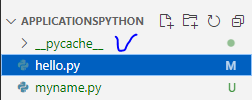
| # myname.py  def get\_name():      return "Алсу Витакор" |
| --- |

Затем измените файл hello.py, включив его, myname и вызовите его функцию:

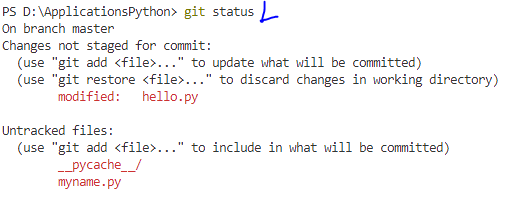
| # hello.py  import myname  name = myname.get\_name()  print("Привет {}".format(name)) |
| --- |

Когда вы импортируете локальный модуль, Python скомпилирует его в байт-код и оставит этот файл в вашей файловой системе. В Python 2 останется файл с именем myname.pyc, но мы предполагаем, что вы используете Python 3. В этом случае он создаст \_\_pycache\_\_каталог и сохранит там файл pyc. Вот что показано ниже:





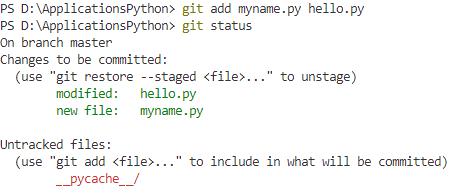
Теперь, если вы запустите git status, вы увидите этот каталог в неотслеживаемом разделе. Также обратите внимание, что ваш новый myname.py файл не отслеживается, а внесенные вами изменения hello.py находятся в новом разделе под названием «Изменения, не подготовленные для фиксации». Это просто означает, что эти изменения еще не добавлены в промежуточную область. Давайте попробуем:



Прежде чем мы перейдем к файлу gitignore, давайте немного наведем порядок, который мы натворили. Сначала мы добавим файлы myname.py и hello.py, как мы это делали ранее:

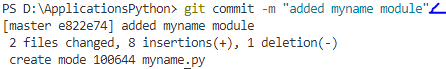
git add myname.py hello.py

git status

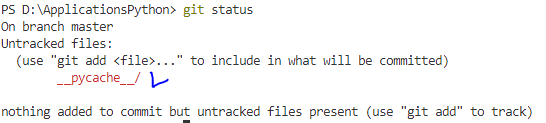


Давайте зафиксируем эти изменения и завершим очистку:

git commit -m "added myname module"



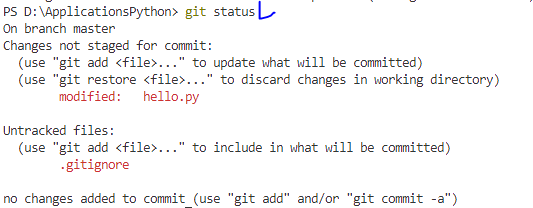
Теперь, когда мы запускаем status, мы видим только этот \_\_pycache\_\_каталог:



Чтобы все \_\_pycache\_\_каталоги (и их содержимое) игнорировались, мы добавим файл .gitignore в наш репозиторий. Это так просто, как кажется. Отредактируйте файл (запомните точку перед именем!) в вашем любимом редакторе.



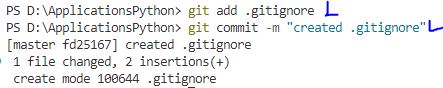
Теперь, когда мы запускаем git status, мы больше не видим \_\_pycache\_\_каталог. Однако мы видим новое .gitignore! Взглянем:



Этот файл представляет собой обычный текстовый файл, и его можно добавить в репозиторий, как и любой другой файл. Сделайте это сейчас:

git add .gitignore

git commit -m "created .gitignore"



Другая распространенная запись в файле .gitignore — это каталог, в котором вы храните свои виртуальные среды. Вы можете узнать больше о virtualenvs здесь[2], но каталог virtualenv обычно называется env или venv. Вы можете добавить их в свой файл .gitignore. Если файлы или каталоги присутствуют, они будут игнорироваться. Если их нет, то ничего не произойдет.

Также возможно хранить глобальный файл .gitignore в вашем домашнем каталоге. Это очень удобно, если ваш редактор предпочитает оставлять временные или резервные файлы в локальном каталоге.

Вот пример простого файла .gitignore для Python:



Более полный пример можно найти здесь[3], а если вы хотите создать свой собственный, здесь git help gitignoreприведены все необходимые детали.

**Что НЕ следует добавлять в репозиторий Git.**

Когда вы впервые начинаете работать с любым инструментом контроля версий, особенно с Git, у вас может возникнуть соблазн поместить все в репозиторий. Это вообще ошибка. Существуют ограничения Git, а также проблемы безопасности, которые вынуждают вас ограничивать типы информации, добавляемой в репозиторий.

Начнем с основного эмпирического правила, касающегося всех систем контроля версий.

***В систему контроля версий помещайте только исходные файлы, а не генерируемые файлы.***

В этом контексте исходный файл — это любой файл, который вы создаете, обычно печатая его в редакторе. Сгенерированный файл — это то , что создает компьютер, обычно путем обработки исходного файла. Например, hello.py это исходный файл, а hello.pyc это сгенерированный файл.

Есть две причины не включать сгенерированные файлы в репозиторий. Во-первых, это пустая трата времени и места. Сгенерированные файлы можно воссоздать в любое время, и, возможно, их потребуется создать в другой форме. Если кто-то использует Jython или IronPython, пока вы используете интерпретатор Cython, файлы .pyc могут сильно отличаться. Фиксация одного конкретного варианта файлов может вызвать конфликт.

Вторая причина отказа от хранения сгенерированных файлов заключается в том, что эти файлы часто больше исходных исходных файлов. Размещение их в репозитории означает, что теперь всем необходимо загружать и хранить эти сгенерированные файлы, даже если они их не используют.

Этот второй пункт приводит к еще одному общему правилу относительно репозиториев Git: фиксируйте двоичные файлы с осторожностью и категорически избегайте фиксации больших файлов. Это правило во многом связано с тем, как работает Git.

Git не хранит полную копию каждой версии каждого зафиксированного вами файла. Скорее, он использует сложный алгоритм, основанный на различиях между последующими версиями файла, чтобы значительно уменьшить необходимый объем памяти. Двоичные файлы (например, файлы JPG или MP3) на самом деле не имеют хороших инструментов сравнения, поэтому Git часто просто нужно сохранять весь файл каждый раз, когда он фиксируется.

Когда вы работаете с Git, и особенно когда вы работаете с GitHub, никогда не помещайте конфиденциальную информацию в репозиторий, особенно ту, которой вы можете поделиться публично. Это настолько важно, что я повторю это еще раз:

**Внимание:** *никогда не размещайте конфиденциальную информацию в общедоступном репозитории на GitHub. Пароли, ключи API и подобные элементы не следует хранить в репозитории. В конце концов кто-нибудь их****найдет****.*

**Что такое SHA**

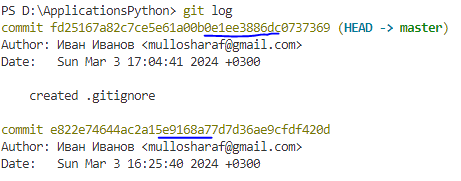
**На память.** *SHA (Secure Hash Algorithm) - это семейство криптографических хеш-функций, разработанных для вычисления уникального хеш-кода фиксированной длины из произвольных данных. Они широко используются в криптографии для обеспечения целостности данных, аутентификации сообщений и других целей безопасности. Наиболее известными версиями SHA являются SHA-1, SHA-256, SHA-384 и SHA-512. Каждая версия имеет свою длину хеша и уровень безопасности. Например, SHA-256 создает хеш длиной 256 бит и считается более безопасным, чем SHA-1, который создает хеш длиной 160 бит и считается устаревшим из-за уязвимостей.*

Когда Git сохраняет вещи (файлы, каталоги, коммиты и т. д.) в вашем репозитории, он делает это сложным образом, используя хэш-функцию. Нам здесь не нужно вдаваться в детали, но хэш-функция принимает объект и создает уникальный идентификатор для этого объекта, который значительно короче (20 байтов, в нашем случае). Этот идентификатор называется «SHA» в Git. Нельзя гарантировать его уникальность, но для большинства практических приложений это подходит.

Git использует свой алгоритм хеширования для индексации всего в вашем репозитории. У каждого файла есть SHA, отражающий содержимое этого файла. Каждый каталог, в свою очередь, также имеет свой хеш. Если файл в этом каталоге изменяется, то меняется и SHA каталога.

Каждый коммит содержит SHA верхнего уровня каталога в вашем репозитории вместе с другой информацией[4]. Именно так одно 20-байтовое число описывает полное состояние вашего репозитория.

Вы можете заметить, что иногда Git использует полное значение из 20 символов для отображения SHA:

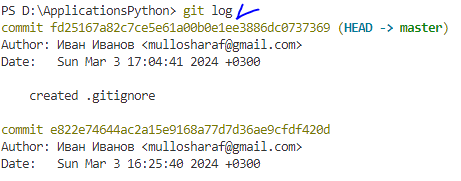


Обычно он показывает вам полную строку символов, но вам не всегда нужно ее использовать. Правило Git заключается в том, что вам нужно указать достаточно символов, чтобы гарантировать уникальность SHA в вашем репозитории. В общем, семи символов более чем достаточно.

Каждый раз, когда вы фиксируете изменения в репозитории, Git создает новый SHA, описывающий это состояние. В следующих разделах мы рассмотрим, чем полезны SHA.

**Журнал Git**

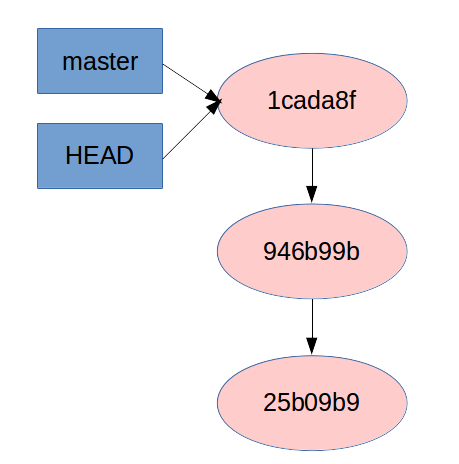
Еще одна очень часто используемая команда Git — git log. Журнал Git показывает историю коммитов, которые вы сделали на данный момент:



Как вы можете видеть в листинге выше, все сообщения о коммитах для нашего репозитория показаны по порядку. Начало каждого коммита отмечается словом «коммит», за которым следует SHA этого коммита. git log дает вам историю каждого из SHA.

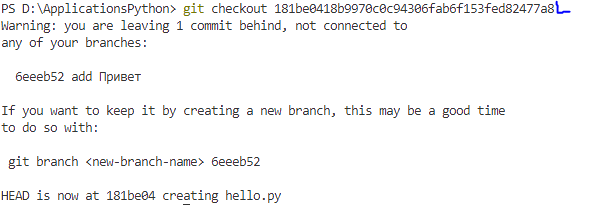
**Возвращение во времени: проверка конкретной версии вашего кода**

Поскольку Git запоминает каждый сделанный вами коммит с помощью SHA, вы можете указать Git перейти к любому из этих коммитов и просмотреть репозиторий в том виде, в котором он существовал на тот момент. На диаграмме ниже показано, что думает Git о нашем репозитории:



Не беспокойтесь о том, что master и HEAD означают на диаграмме. Мы объясним это чуть позже.

Чтобы изменить положение нашей истории, мы воспользуемся командой git checkout, чтобы сообщить Git, какой SHA мы хотим просмотреть. Давайте попробуем это:



Хорошо, здесь много информации, которая может быть запутанной. Давайте начнем с определения некоторых терминов. Начнем с HEAD.

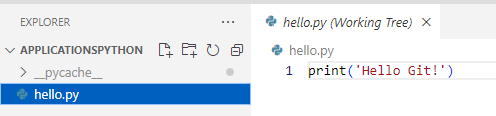
HEAD - это имя Git для любого SHA, на который вы смотрите в данный момент. Это НЕ означает то, что находится на вашем файловой системе или в области подготовки изменений. Это означает то, что Git считает, что вы выбрали для просмотра. Так что, если вы изменили файл, версия на вашей файловой системе отличается от версии в HEAD (и да, HEAD пишется ЗАГЛАВНЫМИ БУКВАМИ).

Далее у нас есть ветка (branch). Самый простой способ думать о ветке - это метка на SHA. У нее есть еще несколько свойств, которые могут быть полезны, но пока думайте о ветке как о метке на SHA.

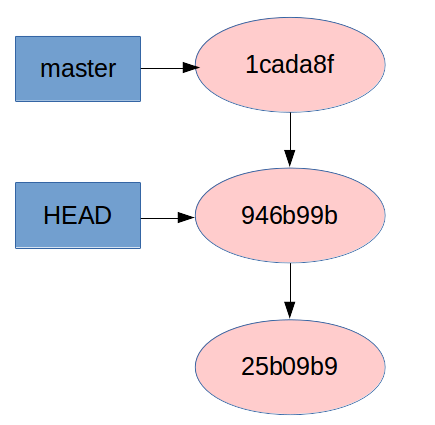
**Примечание:** *Те из вас, кто работал с другими системами контроля версий (я смотрю на вас, Subversion), будут иметь совершенно другое представление о том, что такое ветка. Git обрабатывает ветки по-другому, и это хорошо.*

Когда мы объединяем всю эту информацию, мы видим, что "открепленный HEAD" просто означает, что ваш HEAD указывает на SHA, к которому не привязана ветка (или метка). Git довольно красноречиво говорит вам, как исправить эту ситуацию. Бывают моменты, когда вам захочется это исправить, и бывают моменты, когда можно работать в таком "открепленном HEAD" состоянии без проблем.

Вернемся к нашему демо. Если вы посмотрите на текущее состояние системы, вы увидите, что файл .gitignore больше не присутствует.



Мы вернулись к состоянию системы до того, как внесли эти изменения. Ниже приведена диаграмма нашего репозитория в этом состоянии. Обратите внимание, как указатели HEAD и master указывают на разные SHA:



Хорошо. Теперь, как вернуться к предыдущему состоянию? Есть два способа, один из которых вам уже должен быть знаком: git checkout 1cada8f. Это вернет вас к SHA, на котором вы были, когда начали перемещаться.

**Примечание**: *Одна странность, по крайней мере в моей версии Git, заключается в том, что он все равно выдает предупреждение о "открепленном HEAD", даже если вы вернулись к SHA, связанному с веткой.*

Другой, более распространенный способ вернуться - это переключиться на ветку, на которой вы были. Git всегда начинает с ветки под названием master. Мы узнаем, как создавать другие ветки позже, но пока что остановимся на master.

Чтобы вернуться к предыдущему состоянию, просто выполните



Это вернет вас к последнему зафиксированному SHA в ветке master, который в нашем случае имеет сообщение о коммите "created .gitignore". Другими словами, git checkout master говорит Git сделать HEAD указывающим на SHA, помеченный меткой или веткой master.



Обратите внимание, что существует несколько способов указать конкретный коммит. Вероятно, самым простым для понимания является SHA. Другие методы используют различные символы и имена для указания способа достижения определенного коммита из известного места, например, HEAD. Я не буду углубляться в эти детали в этом руководстве, но если вам нужны дополнительные сведения, вы можете найти их здесь[5].

**Основы ветвления.**

Давайте поговорим немного больше о ветвях. Ветви предоставляют вам способ держать отдельные потоки разработки отдельно. Хотя это может быть полезно, когда вы работаете в одиночку, это почти необходимо, когда вы работаете в команде.

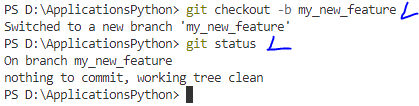
В Git существует несколько типов стандартных веток, которые часто используются в репозиториях. Некоторые из наиболее распространенных веток в Git включают:

* **master**: Ранее использовалась по умолчанию в Git для основной ветки разработки. Однако, начиная с версии Git 2.28, рекомендуется использовать название main вместо master для основной ветки.
* **main**: Это новое стандартное название для основной ветки, которое стало более распространенным и признанным в сообществе Git. Основная идея заключается в том, что main является более нейтральным термином по сравнению с master.
* **develop**: Часто используется для разработки и интеграции новых функций перед объединением их с основной веткой.
* **feature branches**: Ветки, создаваемые для работы над конкретной функцией или задачей. Обычно создаются от основной ветки (например, main или develop) и затем сливаются обратно после завершения работы.
* **release branches**: Ветки, создаваемые для подготовки к выпуску новой версии программного продукта. Здесь могут происходить исправления ошибок и финальные подготовки к релизу.

Это лишь несколько примеров типов веток, которые могут использоваться в Git. В зависимости от методологии разработки (например, Gitflow) или особенностей проекта, могут быть использованы и другие типы веток.

Представьте, что я работаю в небольшой команде и мне нужно добавить функцию в проект. Пока я работаю над этим, я не хочу добавлять свои изменения в master, так как они все еще не работают правильно и могут нарушить работу моих коллег.

Я мог бы просто подождать, пока не закончу все изменения, но это не очень безопасно и не всегда практично. Поэтому, вместо работы в master, я создам новую ветку:



Команда git checkout -b my\_new\_feature используется для создания новой ветки с именем my\_new\_feature и одновременного переключения на эту ветку. При выполнении этой команды Git создаст новую ветку с указанным именем и автоматически переключит вас на эту ветку, чтобы вы могли продолжить работу в новом контексте, не затрагивая основную ветку (обычно master).

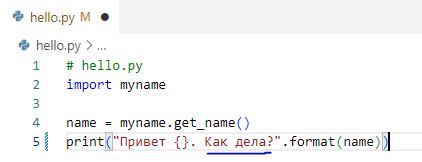
**Примечание.** *Название ветки должно быть описательным и отражать цель или содержание изменений, которые вы планируете внести в этой ветке. Например, если вы добавляете новую функцию, название ветки может быть "add-new-feature". Важно выбирать осмысленные и информативные названия веток для облегчения работы с ними в будущем.*

Как вы можете видеть выше, выполнение команды git status в нашей ветке показывает нам, что имя ветки изменилось. Давайте посмотрим на журнал (лог):

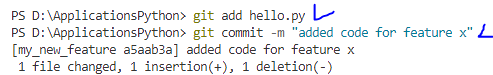


Как я надеюсь, вы ожидали, журнал выглядит точно так же. При создании новой ветки новая ветка начнется с того места, где вы находились. В данном случае мы были в верхней точке master, fd25167a82c7ce5e61a00b0e1ee3886dc0737369, поэтому именно там начинается новая ветка.

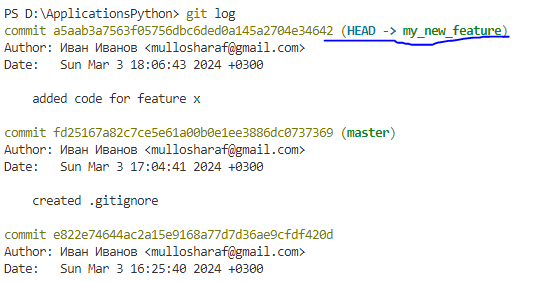
Теперь давайте поработаем над этой функцией. Внесите изменение в файл hello.py и зафиксируйте его.



Я покажу вам команды для проверки, но перестану показывать вывод команд для тех вещей, которые вы уже видели:



Теперь, если вы выполните команду git log, вы увидите, что наш новый коммит присутствует. В моем случае, у него SHA a5aab3a7563f05756dbc6ded0a145a2704e34642, но в вашем репозитории вероятно будет другой SHA.

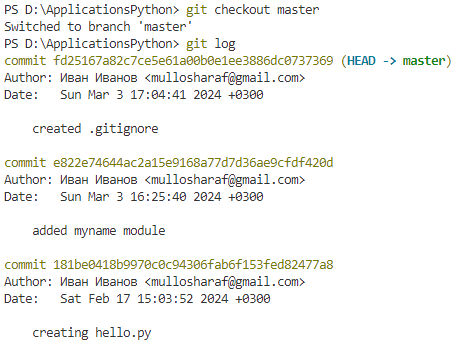


Переключитесь обратно на ветку master и посмотрите на журнал:

git checkout master

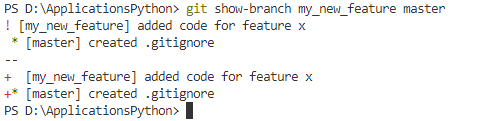
git log

Присутствует ли новый коммит "added code for feature x"?



Git имеет встроенный способ сравнения состояния двух веток, чтобы вам не пришлось так усердно работать. Это команда show-branch. Вот как она выглядит:

git show-branch my\_new\_feature master



График, который он создает, изначально немного запутан, поэтому давайте детально разберемся. Сначала вы вызываете команду, указав ей названия двух веток. В нашем случае это были my\_new\_feature и master.

Первые две строки вывода - это ключ к расшифровке остального текста. Первый непробельный символ на каждой строке - это либо \* или !, за которыми следует имя ветки, а затем сообщение коммита для самого последнего коммита в этой ветке. Символ \* используется для указания того, что ветка в настоящее время выбрана для работы, в то время как ! используется для всех остальных веток. Этот символ находится в столбце, соответствующем коммитам в таблице ниже.

Третья строка - это разделитель.

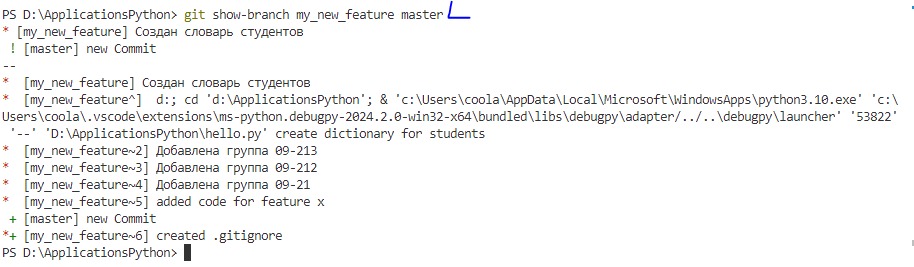
Начиная с четвертой строки, перечислены коммиты, которые есть в одной ветке, но отсутствуют в другой. В нашем текущем случае это довольно просто. Есть один коммит в my\_new\_feature, который отсутствует в master. Вы можете увидеть это на четвертой строке. Обратите внимание, как эта строка начинается с \* в первом столбце. Это указывает на то, в какой ветке находится этот коммит.

Наконец, последняя строка вывода показывает первый общий коммит для двух веток.

Этот пример довольно простой. Чтобы сделать более интересный пример, я добавил несколько дополнительных коммитов в my\_new\_feature и несколько в master.

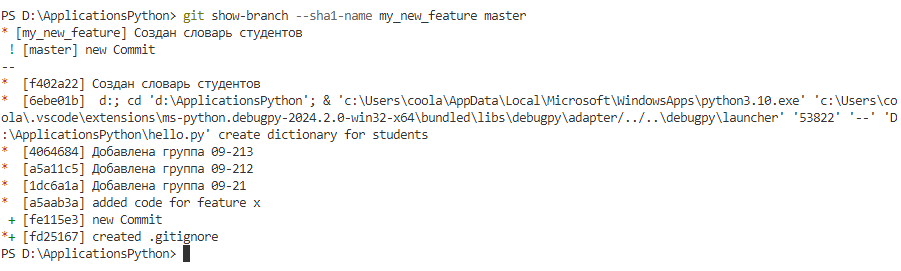
Это делает вывод более интересным

git show-branch my\_new\_feature master



Теперь вы видите, что в каждой ветке есть разные коммиты. Обратите внимание, что текст [my\_new\_feature~2] - это один из методов выбора коммитов, о котором я упоминал ранее. Если вы предпочитаете видеть SHA, вы можете попросить их отобразить, добавив опцию --sha1-name к команде:

git show-branch --sha1-name my\_new\_feature master



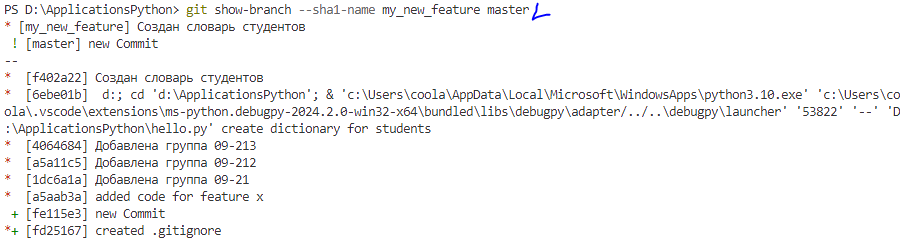
Когда вы наконец завершили работу над функцией и готовы передать ее остальной части команды, что вы делаете?

Существует три основных способа переноса коммитов из одной ветки в другую: слияние (merging), перебазирование (rebasing) и cherry-picking. Мы рассмотрим каждый из них по очереди в следующих разделах.

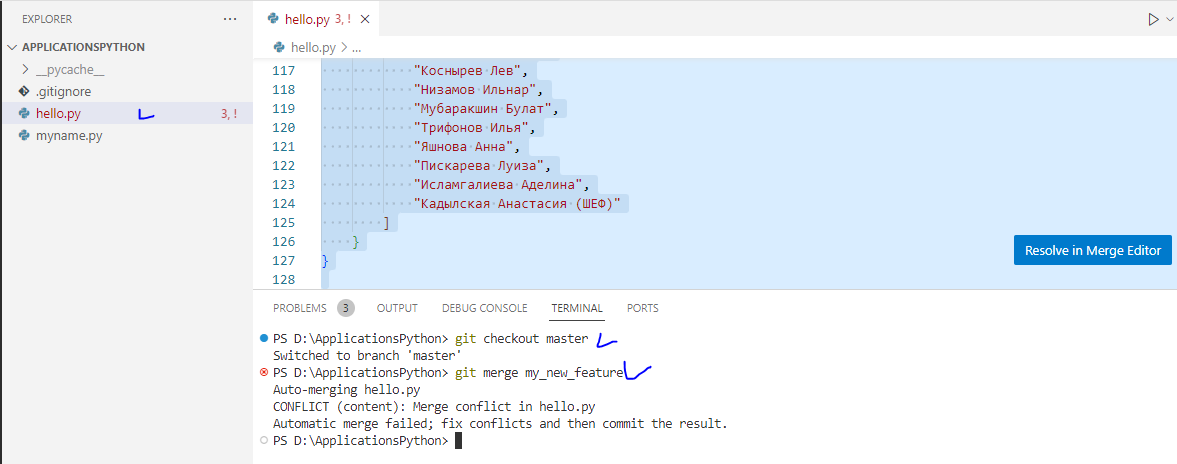
**Слияние (Merging).**

Слияние - самый простой из трех способов для понимания и использования. При выполнении слияния Git создаст новый коммит, который объединяет верхние SHA двух веток, если это необходимо. Если все коммиты в другой ветке находятся впереди (основаны на) вершины текущей ветки, Git просто выполнит быстрое слияние (fast-forward merge) и поместит эти новые коммиты в эту ветку.

Давайте вернемся к моменту, когда вывод show-branch выглядел так:



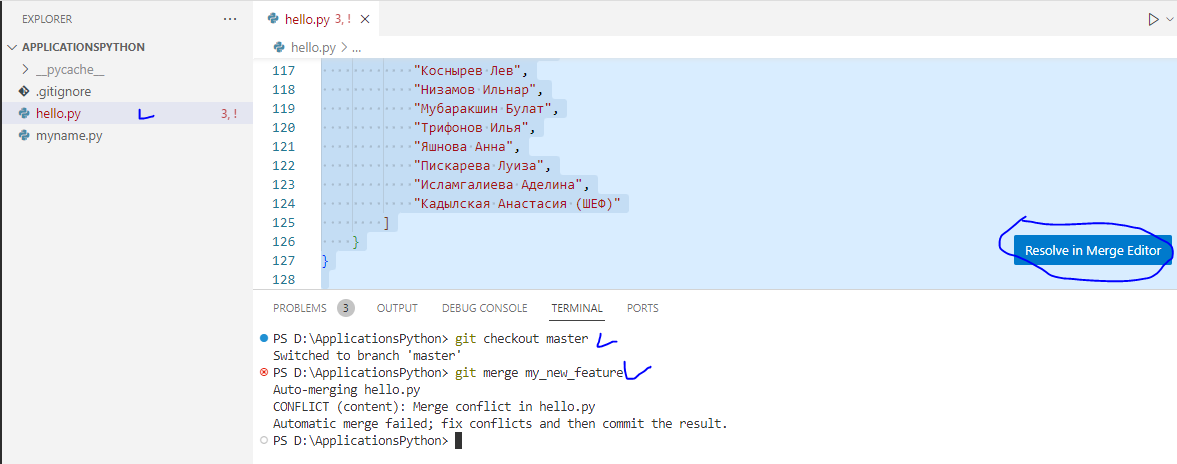
Теперь мы хотим, чтобы все коммиты my\_new\_feature был на master. Проверьте мастер и запустите git merge там команду:



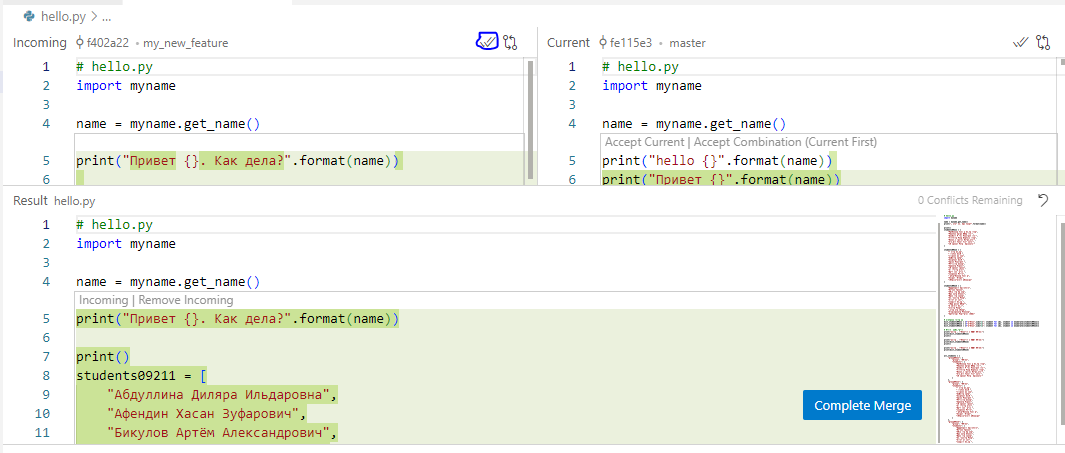
Похоже, что у нас возник конфликт слияния при объединении ветки my\_new\_feature в ветку master. Чтобы разрешить этот конфликт, вам нужно вручную отредактировать конфликтующий файл(ы), а затем зафиксировать изменения. Вот общие шаги по разрешению конфликта слияния в Git:

1. Откройте конфликтующий файл(ы) в вашем редакторе кода.
2. Найдите маркеры конфликта <<<<<<<, ======= и >>>>>>> в файле. Эти маркеры указывают на конфликтующие изменения из разных веток.
3. Отредактируйте конфликтующие участки файла для разрешения различий между конфликтующими ветками.
4. Удалите маркеры конфликта и убедитесь, что файл отражает желаемое окончательное состояние.
5. Сохраните изменения после разрешения конфликтов.
6. Определите разрешенные файлы с помощью git add <разрешенный\_файл> или git add ., чтобы добавить все изменения.
7. Зафиксируйте разрешенные изменения с помощью git commit -m "Разрешение конфликта слияния путем включения обоих изменений".
8. После разрешения конфликта вы можете продолжить слияние, запустив git merge --continue.

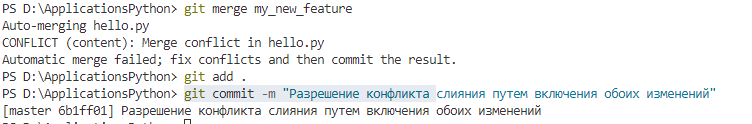
После того, как вы разрешите конфликты и зафиксируете изменения, вы сможете успешно завершить слияние.

Для решения этой проблемы в Visual Studio Code нажмите кнопку «Resolve in Merge Editor». Выберите все необходимые изменения. 

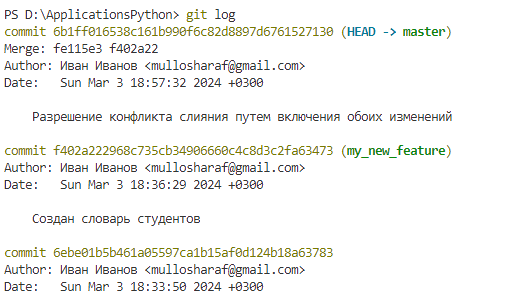
После этоно нажмите кнопку "Complete Merge".



После зафиксирования изменений с помощью git commit -m "Разрешение конфликта слияния путем включения обоих изменений", процесс завершается и изменения сохраняются в истории репозитория.



Поскольку мы находились на ветке master, мы сделали слияние ветки my\_new\_feature к нам. Вы можете увидеть, что это было слияние прямого продвижения и какие файлы были изменены. Давайте теперь посмотрим на лог:



Если бы мы внесли изменения в master перед слиянием, Git создал бы новый коммит, который объединял бы изменения из двух веток.

Одним из хороших качеств Git является способность понимать общих предков различных веток и автоматически объединять изменения. Если один и тот же участок кода был изменен в обеих ветках, Git не может определить, что делать. В таком случае он останавливает слияние на полпути и дает вам инструкции о том, как исправить проблему. Это называется конфликтом слияния.

**Перебазирование.**

Перебазирование похоже на слияние, но ведет себя немного иначе. При слиянии, если в обеих ветках есть изменения, создается новый коммит слияния. В случае перебазирования Git берет коммиты из одной ветки и поочередно воспроизводит их поверх другой ветки.

**Выборочное копирование (Cherry-Picking).**

Выборочное копирование (Cherry-Picking) - это еще один метод перемещения коммитов из одной ветки в другую. В отличие от слияния и перебазирования, при выборочном копировании вы явно указываете, какие именно коммиты вы имеете в виду. Самым простым способом сделать это является указание одного SHA:

Для перемещения коммита с ветки my\_new\_feature на ветку master вы можете использовать команду git cherry-pick:

* 1. Сначала переключитесь на ветку master:
  2. Затем определите идентификатор (хэш) коммита на ветке feature\_branch, который вы хотите переместить. Вы можете найти этот хэш, выполнив команду git log на ветке my\_new\_feature.
  3. После того как вы найдете хэш нужного коммита, используйте команду git cherry-pick <commit> для перемещения этого коммита на ветку master. Например:

git cherry-pick 6b1ff016538c161b990f6c82d8897d6761527130

где 6b1ff016538c161b990f6c82d8897d6761527130- хэш коммита на ветке my\_new\_feature.

Эта команда говорит Git взять изменения, внесенные в коммите 4a4f449, и применить их к текущей ветке.

* 1. Разрешите любые конфликты, которые могут возникнуть при применении коммита с помощью инструментов слияния Git.
  2. После успешного применения коммита на ветку master, закоммитьте изменения с помощью git commit, если это необходимо.

Таким образом, коммит будет перемещен с ветки feature\_branch на ветку master с использованием git cherry-pick.

Эта функция может быть очень удобной, когда вам нужно конкретное изменение, но не вся ветка, на которой было сделано это изменение.

Быстрый совет о ветвлении: Я не могу завершить эту тему, не порекомендовав отличный ресурс для изучения ветвления в Git. Learn Git Branching предлагает набор упражнений с графическими представлениями коммитов и веток, чтобы ясно объяснить разницу между слиянием, перебазированием и выборочным копированием. Я настоятельно рекомендую потратить время на выполнение этих упражнений [6].

**Работа с удаленными репозиториями**

Все команды, о которых мы говорили до этого момента, работают только с вашим локальным репозиторием. Они не выполняют никакой коммуникации с сервером или по сети. Оказывается, есть всего четыре основные команды Git, которые фактически общаются с удаленными репозиториями:

* clone (клонирование)
* fetch (забор)
* pull (получение)
* push (отправка)

Вот и все. Все остальное выполняется на вашем локальном компьютере. (Хорошо, чтобы быть полностью точным, есть и другие команды, которые общаются с удаленными репозиториями, но они не входят в базовую категорию.)

Давайте рассмотрим каждую из этих команд по очереди.

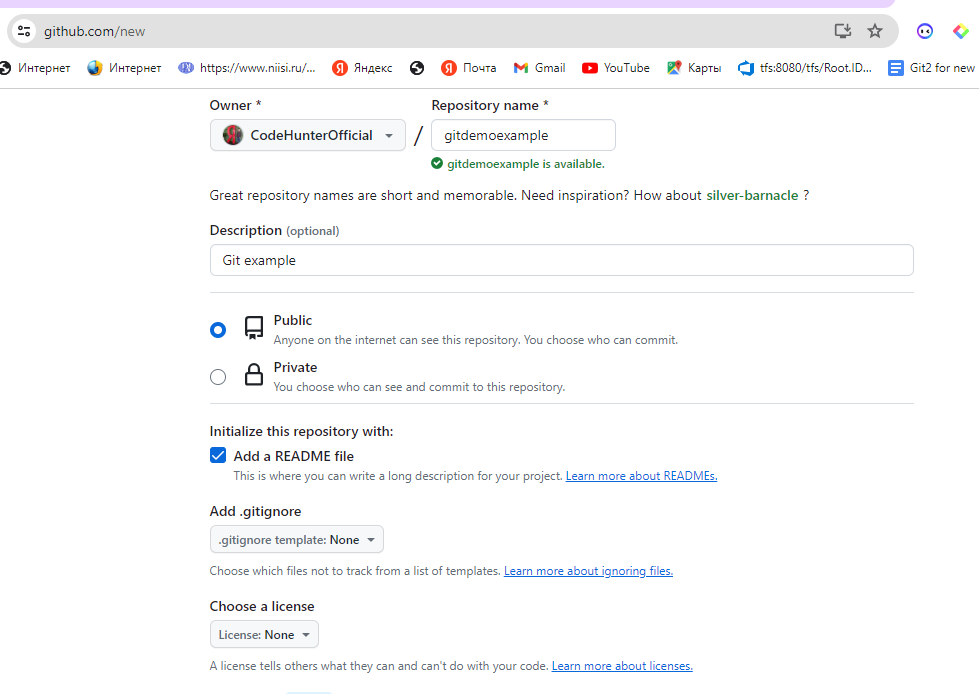
**Push (отправка)**

Как вы, вероятно, догадались, команда git push является противоположностью команды git pull. Ну, почти противоположность. Push отправляет информацию о ветке, которую вы пушите, и спрашивает удаленный репозиторий, не хотел бы ли он обновить свою версию этой ветки в соответствии с вашей.

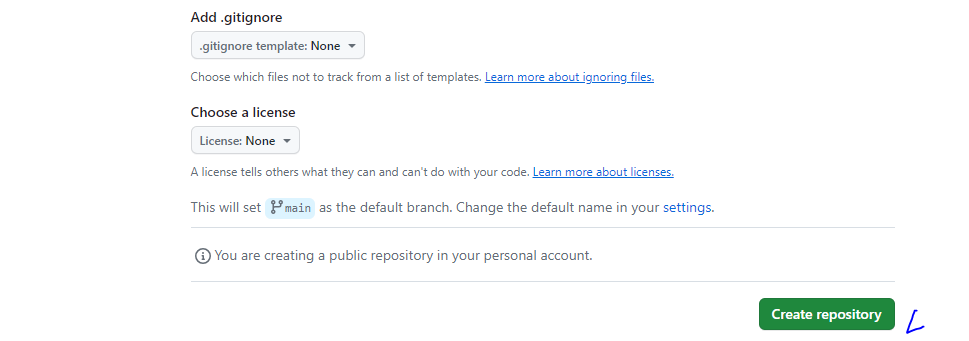
Обычно это означает, что вы отправляете свои новые изменения на сервер. Здесь есть много деталей и сложностей, касающихся того, что такое коммит прямого продвижения.

Чтобы создать удаленный репозиторий на GitHub из локального репозитория, вам следует выполнить следующие шаги:

* 1. Перейдите на свою учетную запись на GitHub по ссылке: <https://github.com/CodeHunterOfficial>
  2. На странице вашего профиля нажмите кнопку "New" (Новый) или "Create repository" (Создать репозиторий) для создания нового репозитория.
  3. Заполните необходимую информацию о новом репозитории, такую как название, описание, публичный или приватный доступ и другие настройки по вашему усмотрению.

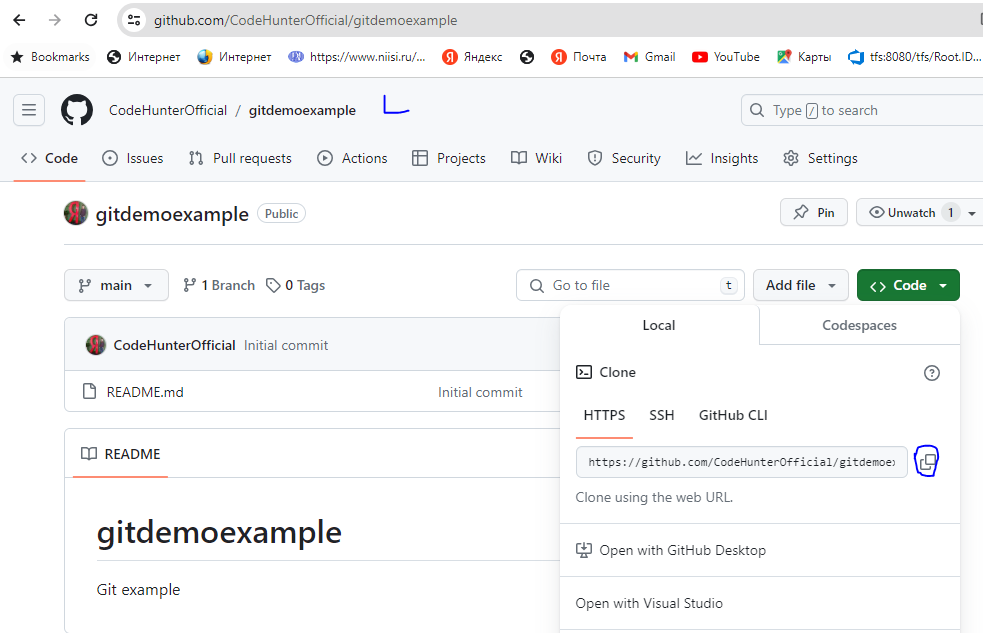


* 1. Нажмите кнопку "Create repository" (Создать репозиторий) для завершения создания удаленного репозитория на GitHub.



* 1. После создания удаленного репозитория, вам будет предоставлен URL этого репозитория. Скопируйте этот URL, так как он понадобится вам для связывания вашего локального репозитория с удаленным.

<https://github.com/CodeHunterOfficial/gitdemoexample>



* 1. После этого вы можете добавить удаленный репозиторий к вашему локальному репозиторию с помощью команды:

| git remote add origin <URL\_удаленного\_репозитория> |
| --- |

Здесь <URL\_удаленного\_репозитория> - это URL вашего созданного удаленного репозитория на GitHub. В нашем случае:

git remote add origin <https://github.com/CodeHunterOfficial/gitdemoexample.git>

Затем вы можете отправить свои изменения на удаленный сервер с помощью команды push:

git push -u origin <название\_ветки>

Здесь <название\_ветки> - это название ветки, которую вы хотите отправить на удаленный сервер. В нашем случае:

git push -u origin master

После выполнения этих шагов, ваш локальный репозиторий будет связан с созданным удаленным репозиторием на GitHub, и вы сможете отправлять изменения на удаленный сервер с помощью команды push.

**Clone (Клонирование)**

Команда Git clone используется, когда у вас есть адрес известного репозитория, и вы хотите создать локальную копию. Для этого примера давайте использовать небольшой репозиторий, который у меня есть на моей учетной записи GitHub, github-playground.

Страница GitHub для этого репозитория находится здесь. На этой странице вы найдете кнопку "Clone or Download", которая дает вам URI, который нужно использовать с командой git clone. Если вы скопируете его, вы сможете клонировать репозиторий следующим образом:

git clone <https://github.com/CodeHunterOfficial/gitdemoexample.git>

Теперь у вас есть полный репозиторий этого проекта на вашем локальном компьютере. Это включает в себя все коммиты и все ветки, которые когда-либо были созданы в нем. (Примечание: Этот репозиторий использовался моими друзьями, когда они учились Git. Я скопировал или создал его форк от кого-то другого.)

Если вы хотите попробовать другие команды для работы с удаленными репозиториями, вам следует создать новый репозиторий на GitHub и следовать тем же шагам. Вы можете создать форк репозитория github-playground на свою учетную запись и использовать его. Создание форка на GitHub выполняется путем нажатия кнопки "Fork" в пользовательском интерфейсе.

Git clone - это команда в Git, которая используется для создания локальной копии удаленного репозитория. При выполнении git clone происходит загрузка всех файлов, истории коммитов и веток с удаленного репозитория на ваш компьютер.

Git clone обычно используется для:

* Создания локальной копии удаленного репозитория для работы над проектом локально.
* Получения проекта из удаленного репозитория для совместной работы или разработки.

**Примеры использования git clone:**

Клонирование удаленного репозитория на локальную машину:

git clone https://github.com/username/repository.git

Клонирование удаленного репозитория в определенную папку на локальном компьютере:

git clone https://github.com/username/repository.git my-project

Клонирование удаленного репозитория с указанием имени новой папки:

git clone https://github.com/username/repository.git new-folder-name

Надеюсь, эти примеры помогут вам понять, как использовать команду git clone и в каких ситуациях она может быть полезна!

**Fetch (забор)**

Для того чтобы четко объяснить команду fetch, нам нужно вернуться назад и поговорить о том, как Git управляет отношениями между вашим локальным репозиторием и удаленным репозиторием. Этот следующий кусок информации является фоновым, и хотя вы не будете использовать его ежедневно, он поможет лучше понять разницу между fetch и pull.

Когда вы клонируете новый репозиторий, Git не просто копирует одну версию файлов в этом проекте. Он копирует весь репозиторий и использует его для создания нового репозитория на вашем локальном компьютере.

Git не создает для вас локальные ветки, за исключением master. Однако он отслеживает ветки, которые были на сервере. Для этого Git создает набор веток, которые все начинаются с remotes/origin/<branch\_name>.

Почти никогда вы не будете проверять эти ветки remotes/origin, но полезно знать, что они существуют. Помните, что каждая ветка, которая существовала на удаленном сервере в момент клонирования репозитория, будет иметь ветку в remotes/origin.

Когда вы создаете новую ветку, и имя совпадает с существующей веткой на сервере, Git помечает вашу локальную ветку как отслеживающую ветку, связанную с удаленной веткой. Мы увидим, как это полезно при переходе к команде pull.

Теперь, когда вы знаете о ветках remotes/origin, понимание git fetch будет довольно простым. Все, что делает fetch, это обновляет все ветки remotes/origin. Он изменяет только ветки, хранящиеся в remotes/origin, а не какие-либо из ваших локальных веток.

Git fetch - это команда в Git, которая извлекает изменения из удаленного репозитория, но не применяет их к текущей ветке. Она обновляет ваш локальный репозиторий, чтобы отслеживать изменения, которые произошли в удаленном репозитории.

Git fetch обычно используется для:

Просмотра изменений, которые произошли в удаленном репозитории.

Сравнения локальных изменений с изменениями на удаленном репозитории.

**Примеры использования git fetch:**

Извлечение изменений из удаленного репозитория:

git fetch origin

Просмотр изменений, которые были извлечены, но еще не слиты:

git diff origin/main

Обновление локальной ветки до последнего состояния из удаленного репозитория:

git fetch origin

git merge origin/main

**Pull (получение)**

Команда Git pull просто сочетает в себе две другие команды. Сначала она выполняет git fetch для обновления веток remotes/origin. Затем, если ветка, на которой вы находитесь, отслеживает удаленную ветку, то выполняется git merge соответствующей удаленной ветки remotes/origin в вашу ветку.

Например, предположим, что вы находитесь на ветке my\_new\_feature, и ваш коллега только что добавил некоторый код в нее на сервере. Если вы выполните git pull, Git обновит ВСЕ ветки remotes/origin, а затем выполнит git merge remotes/origin/my\_new\_feature, что позволит получить новый коммит в вашу текущую ветку!

Конечно, здесь есть некоторые ограничения. Git не позволит вам даже попытаться выполнить git pull, если у вас есть измененные файлы на локальной системе. Это может привести к слишком большому беспорядку.

Если у вас есть коммиты на вашей локальной ветке, и на удаленном сервере также есть новые коммиты (т.е. "ветки разошлись"), то часть git merge при выполнении pull создаст коммит слияния, как мы обсуждали ранее.

Те из вас, кто внимательно читал, заметят, что вы также можете заставить Git выполнить перебазирование вместо слияния, выполнив git pull -r.

Git pull - это команда в Git, которая извлекает изменения из удаленного репозитория и автоматически сливает их с текущей веткой. Это комбинация двух действий: git fetch для извлечения изменений и git merge для слияния изменений с локальной веткой.

Git pull обычно используется для:

* 1. Получения последних изменений из удаленного репозитория и обновления локальной ветки.
  2. Слияния изменений из удаленной ветки с текущей локальной веткой.

**Примеры использования git pull:**

Получение и автоматическое слияние изменений из удаленной ветки:

git pull origin main

Получение изменений из удаленного репозитория и слияние их с текущей веткой:

git pull

Получение изменений из удаленной ветки и слияние их с текущей веткой, разрешая возможные конфликты:

git pull origin feature-branch

**Другие полезные коммпанды git**

**Отмена коммитов и изменений**

git clean

Команда "git clean" в Git используется для удаления файлов и директорий, которые не отслеживаются в репозитории. Это позволяет очистить рабочий каталог от ненужных файлов. Важно помнить, что удаление происходит без возможности восстановления, поэтому нужно быть осторожным при использовании этой команды.

**Примечание**. *Неотслеживаемые файлы в Git - это файлы, которые находятся в рабочем каталоге, но не добавлены в индекс (staging area) и не находятся под контролем версий Git. Это могут быть файлы, которые были созданы после клонирования репозитория или файлы, которые были проигнорированы с помощью файла .gitignore.*

*Когда файлы не отслеживаются Git, они не включаются в коммиты и не участвуют в управлении версиями. Команда "git clean" используется для удаления этих неотслеживаемых файлов из рабочего каталога.*

*Если вы хотите начать отслеживать неотслеживаемые файлы, вы можете использовать команду "git add" для добавления их в индекс, а затем сделать коммит, чтобы включить их в историю версий вашего репозитория.*

Вот несколько примеров использования команды "git clean":

* 1. Удалить все неотслеживаемые файлы и директории в рабочем каталоге:

git clean –f

* 1. Удалить все неотслеживаемые файлы включая игнорируемые файлы (например, в .gitignore):

git clean –fX

* 1. Просмотреть, какие файлы будут удалены без фактического удаления:

git clean –n

* 1. Удалить определенный файл или директорию (например, "example.txt"):

**git revert**

Можно сказать, что команда git revert представляет собой типичную команду отмены. Однако принцип ее действия отличается от привычной отмены изменений. Вместо удаления коммита из истории проекта эта команда отменяет внесенные в нем изменения и добавляет новый коммит с полученным содержимым. В результате история в Git не теряется, что важно для обеспечения целостной истории версий и надежной совместной работы.

Отмена с помощью команды revert необходима, когда нужно обратить изменения, внесенные в некоем коммите из истории проекта. Это может быть полезно, если баг появился в проекте из-за конкретного коммита. Вам не потребуется вручную переходить к этому коммиту, исправлять его и выполнять коммит нового снимка состояния — команда git revert сделает это автоматически.

Примеры использования команды "git revert":

1. Допустим, у вас есть коммит с идентификатором "abc123", который вы хотите отменить:

git revert abc123

1. Если вы хотите отменить последний коммит:

git revert HEAD

1. Чтобы отменить несколько коммитов, вы можете указать диапазон коммитов:

git revert abc123..def456

После выполнения команды "git revert" Git создаст новый коммит, который отменит изменения указанного коммита, и вы сможете продолжить работу с обновленной историей изменений. Таким образом, команда "git revert" позволяет безопасно отменить изменения в Git, сохраняя при этом целостность истории изменений вашего проекта.

**Распространенные опции**

-e

--edit

Это параметр по умолчанию, указывать его необязательно. Он открывает настроенный системный редактор, в котором вам будет предложено изменить комментарий к коммиту перед выполнением отмены

--no-edit

Пример: git revert HEAD --no-edit

Действие этого параметра обратно действию -e. Редактор во время выполнения отмены не откроется.

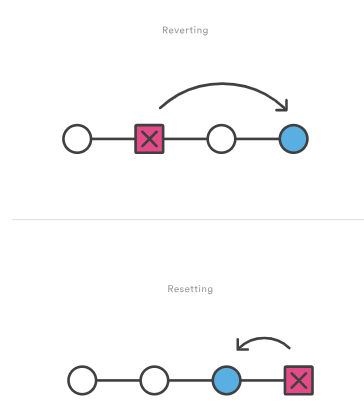
-n

--no-commit

При передаче этого параметра команда git revert не будет создавать новый коммит, обратный целевому по своему содержимому. Вместо этого обращенные изменения будут добавлены в раздел проиндексированных файлов и рабочий каталог.

**Разница между командами git reset и git revert**

Важно понимать, что команда git revert отменяет одиночный коммит, но не возвращает проект в предшествовавшее состояние с удалением всех последующих коммитов. Такой результат в Git дает команда reset, а не revert.



У команды revert есть два серьезных преимущества перед командой reset. Во-первых, она не изменяет историю проекта и, соответственно, безопасна для коммитов, которые уже опубликованы в общем репозитории. Подробнее о том, почему изменять общую историю опасно, см. на странице команды [git reset](https://www.atlassian.com/ru/git/tutorials/undoing-changes/git-reset).

Во-вторых, целью для команды git revert можно выбрать любой отдельный коммит в истории проекта, в то время как действие команды git reset может отменить только коммит, предшествующий текущему. Если бы возникла необходимость отменить старый коммит с помощью git reset, пришлось бы удалить все коммиты после целевого, затем удалить его и выполнить все последующие коммиты заново.

**git reset**

Команда git reset – это мощный способ отменить операцию. Существует три возможных аргумента:

--mixed

Пример: git reset --mixed HEAD~1

Значение по умолчанию. Команда git reset --mixed аналогична git reset. Вы переключите HEAD на последний коммит, и все изменения, добавленные после него, будут доступны в качестве неотслеживаемых (untracked) изменений в вашем рабочем каталоге.

--soft

Пример: git reset –soft HEAD~1

HEAD переключается на последний коммит, однако, изменения, добавленные после этой фиксации, остаются с пометкой staged.

--hard

Пример: git reset  -- hard HEAD~1

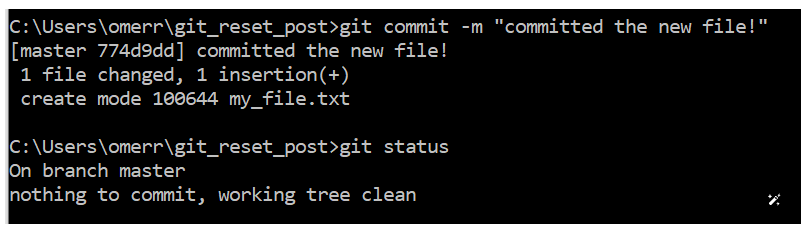
Используйте команду git reset --hard только тогда, когда вы знаете, что делаете. Вы переключите HEAD на последний коммит и уничтожите изменения, сделанные после него. Это действие не может быть отменено.

Пример. Пусть мы добавили новый файл myfile.txt и выпольныли комманду git add myfile.txt. Теперь выполним git status

Как показывает git status, наш файл теперь в стейджинге и готов к коммиту. Да, он еще не является частью никакого коммита. Другими словами, сейчас он находится в рабочей директории, а также в индексе, но не в репозитории.



Если мы теперь выполним git commit, мы создадим коммит на основе состояния индекса. Таким образом новый коммит (в примере — commit 3) будет включать файл, который мы чуть ранее добавили в стейджинг.



Рабочая директория находится в точно таком же состоянии, как индекс и репозиторий.

При выполнении git commit текущая ветка master начинает указывать на только что созданный объект commit.



**Внутренняя работа git reset**

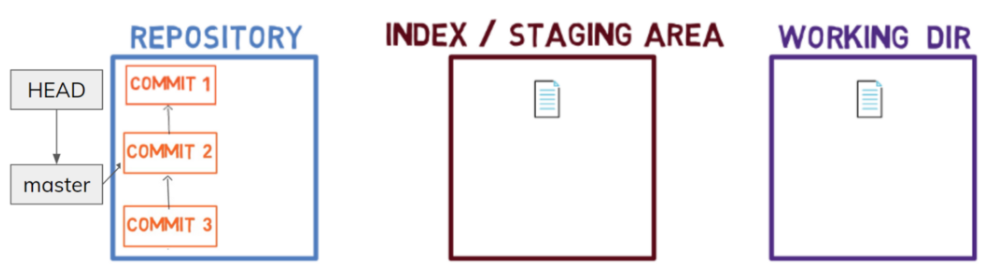
Мне нравится представлять git reset как команду, которая поворачивает вспять описанный выше процесс (внесение изменений в рабочей директории, добавление их в индекс, а затем сохранение в репозиторий).

Как уже сказали у git reset есть три режима: --soft, --mixed и --hard. Я рассматриваю их как три стадии:

Стадия 1. Обновление HEAD — git reset —soft

Прежде всего, git reset меняет то, на что указывает HEAD. Если мы выполним git reset --hard HEAD~1, HEAD будет указывать не на master, а на HEAD~1. Если использовать флаг --soft, git reset на этом и остановится.

Если вернуться к нашему примеру, HEAD будет указывать на commit 2, и таким образом new\_file.txt не будет частью дерева текущего коммита. Но он будет частью индекса и рабочей директории.



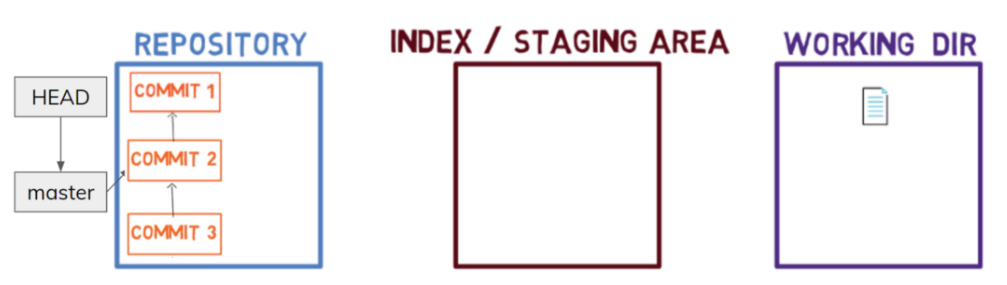
Если посмотреть git status, мы увидим, что этот файл определенно в стейджинге, но не закоммичен.

Иными словами, мы вернули процесс на стадию, где мы уже применили git add, но еще не применяли git commit.

Стадия 2. Обновление индекса — git reset —mixed

Если мы используем git reset --mixed HEAD~1, git не остановится на обновлении того, на что указывает HEAD. Помимо этого обновится еще и индекс (до состояния уже обновленного HEAD).

В нашем примере это значит, что индекс будет в том же виде, что и commit 2:

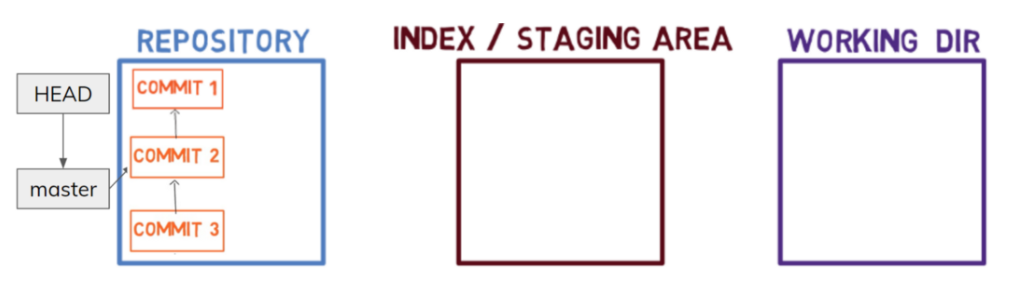


Таким образом мы вернули процесс на стадию до выполнения команды git add. Новосозданный файл является частью рабочей директории, но не индекса и не репозитория.

Стадия 3. Обновление рабочей директории — git reset —hard

Если использовать git reset  -- hard HEAD~1, то после перевода указателя HEAD (на что бы он ни указывал раньше) на HEAD~1, а также обновления индекса до (уже обновленного) HEAD, git пойдет еще дальше и обновит рабочую директорию до состояния индекса.

Применительно к нашему примеру это означает, что рабочая директория будет приведена к состоянию индекса, который уже приведен в состояние commit 2:



Собственно, мы вернули весь процесс на этап до создания файла my\_file.txt.

**git amend**

Git amend - это команда в Git, которая позволяет внести изменения в последний коммит. Она используется, когда вы хотите внести небольшие изменения в последний коммит без создания нового коммита.

Вот несколько примеров использования git amend:

* 1. Добавление недостающего файла в последний коммит:

| git add забытый\_файл  git commit --amend |
| --- |

* 1. Изменение сообщения последнего коммита:

| git commit --amend -m "Новое сообщение коммита" |
| --- |

* 1. Изменение содержимого последнего коммита:

| // Внесите необходимые изменения  git add измененные\_файлы  git commit --amend |
| --- |

**Предупреждение***. Используйте git amend осторожно, так как это изменяет историю коммитов, и это может привести к проблемам при работе в команде или с общим репозиторием.*

**git stash**

Git stash - это механизм в Git, который позволяет временно сохранить незакоммиченные изменения в отдельном стеке, чтобы потом можно было вернуться к ним. Это полезно, когда вы хотите временно переключиться на другую задачу или ветку, но не хотите коммитить незавершенные изменения.

Примеры использования git stash:

* 1. Сохранение изменений перед переключением веток:

| git stash  git checkout another-branch  git stash pop |
| --- |

* 1. Сохранение изменений перед обновлением репозитория:

| git stash  git pull  git stash pop |
| --- |

* 1. Сохранение изменений перед исправлением конфликтов при слиянии веток:

| git stash  # Решение конфликтов  git stash pop |
| --- |

Команда git stash apply в Git применяет последний сохраненный stash, не удаляя его из стека. Это отличается от git stash pop, который применяет последний stash и удаляет его из стека. git stash apply полезен, когда вы хотите применить сохраненные изменения, но оставить stash в стеке для возможности применения его позже или несколько раз.

Пример использования git stash apply:

| git stash apply |
| --- |

Эта команда применит последний stash, но не удалит его из стека. Если нужно удалить stash после применения, можно воспользоваться командой git stash drop.

Команда `git stash save`используется для временного сохранения изменений, которые вы внесли в рабочую директорию, но еще не зафиксировали и не готовы коммитить. Когда вы выполняете `git stash save`, Git сохраняет ваши изменения и очищает рабочую директорию до последнего коммита. Позже вы можете применить эти изменения обратно с помощью `git stash apply` или `git stash pop`.

Пример работы с командами git stash save:

Сохранение изменений в стэше:

git stash save "Мой временный стэш"

Просмотр списка сохраненных стэшей:

git stash list

Применение последнего сохраненного стэша:

git stash apply

Применение конкретного сохраненного стэша (например, №1):

git stash apply stash@{1}

Применение и удаление последнего сохраненного стэша:

git stash pop

Удаление конкретного сохраненного стэша (например, №1):

git stash drop stash@{1}

**git diff**

Git diff - это команда в системе контроля версий Git, которая используется для отображения различий между двумя состояниями репозитория. Это позволяет увидеть, какие изменения были внесены в файлы с момента последнего коммита или между двумя ветками.

**Примеры использования git diff:**

Просмотр изменений в рабочем каталоге по сравнению с последним коммитом:

git diff

Просмотр изменений между двумя коммитами:

git diff <commit\_id1> <commit\_id2>

Например,. git diff 4339579 7ff9ebd

Просмотр изменений в конкретном файле:

git diff <file\_name>

Просмотр изменений между текущей веткой и другой веткой:

git diff <branch\_name>

Используя git diff, разработчики могут легко отслеживать и анализировать изменения в своем коде, что помогает в решении конфликтов слияния, отладке и обзоре кода.

**git reflog**

git reflog - это команда в Git, которая позволяет просматривать историю ссылок (references) в вашем репозитории, включая информацию о перемещении указателей веток и HEAD. Она полезна для восстановления потерянных коммитов или веток, а также для отслеживания истории изменения указателей.

**Примеры использования git reflog:**

Просмотр истории перемещения HEAD:

| git reflog HEAD |
| --- |

Восстановление удаленной ветки:

| git reflog  git checkout -b <branch-name> <commit-hash> |
| --- |

Восстановление потерянного коммита:

| git reflog  git cherry-pick <commit-hash> |
| --- |

Потерянный коммит в Git - это коммит, на который нет никаких указателей (веток или тегов), из-за чего он становится недоступным из текущей истории репозитория. Это может произойти, например, после слияния веток, перезаписи истории с помощью команды rebase или после удаления ветки.

В случае потери коммита, его все еще можно восстановить, используя команду git reflog, которая позволяет просматривать историю перемещения указателей и восстанавливать "потерянные" коммиты по их хэшу.

Пример восстановления удаленной ветки:

* 1. Используйте git reflog для просмотра истории перемещения указателей:

| git reflog |
| --- |

* 1. Найдите хэш коммита, на котором была удалена ветка.
  2. Создайте новую ветку на найденном коммите:

| git checkout -b <branch-name> <commit-hash> |
| --- |

Таким образом, потерянный коммит - это коммит, на который нет прямых ссылок, но который все еще существует в истории репозитория и может быть восстановлен с помощью соответствующих инструментов Git.

git reflog особенно полезен в случаях, когда произошли нежелательные изменения с указателями веток или HEAD, и нужно восстановить предыдущее состояние.

**Работа с ветками в Git (git branch)**

Работа с ветками в Git - это важный аспект при управлении версиями вашего кода. Команда git branch используется для создания, просмотра и управления ветками в Git. Вот несколько примеров использования git branch:

Просмотр списка веток:

| git branch |
| --- |

Создание ветки:

| git branch <branch-name> |
| --- |

Переключение на созданную ветку:

| git checkout <branch-name>  или  git switch <branch-name> |
| --- |

Удаление ветки:

| git branch -d <branch-name> |
| --- |

Если ветка не была слита, используйте -D для принудительного удаления.

Переименование ветки:

| git branch -m <old-branch-name> <new-branch-name> |
| --- |

**git blame**

Git blame - это команда в системе контроля версий Git, которая позволяет узнать, кто и когда внес изменения в определенные строки кода в репозитории. Это полезный инструмент для отслеживания истории изменений и определения ответственного за определенный участок кода.

**Примеры использования git blame:**

Чтобы узнать, кто внес определенные изменения в файл script.js:

git blame script.js

Для выяснения, кто последний изменил строки с 10 по 20 в файле styles.css:

git blame -L 10,20 styles.css

Для просмотра истории изменений конкретной строки кода в файле index.html:

git blame -L 42,42 index.html

Используя git blame, разработчики могут более эффективно сотрудничать, отслеживать изменения и разбираться в коде проекта.

**git tag**

Git tag - это механизм в системе контроля версий Git, который позволяет помечать определенные коммиты как определенные версии или моменты в истории проекта. Теги используются для фиксации определенного состояния репозитория на определенный момент времени, например, для обозначения релизов или важных точек в истории проекта.

**Примеры использования git tag:**

1. Создание аннотированного тега для текущего коммита с сообщением:

git tag -a v1.0 -m "Release version 1.0"

1. Просмотр списка всех тегов в репозитории:

git tag

1. Просмотр информации о конкретном теге:

git show v1.0

1. Публикация тегов на удаленном репозитории:

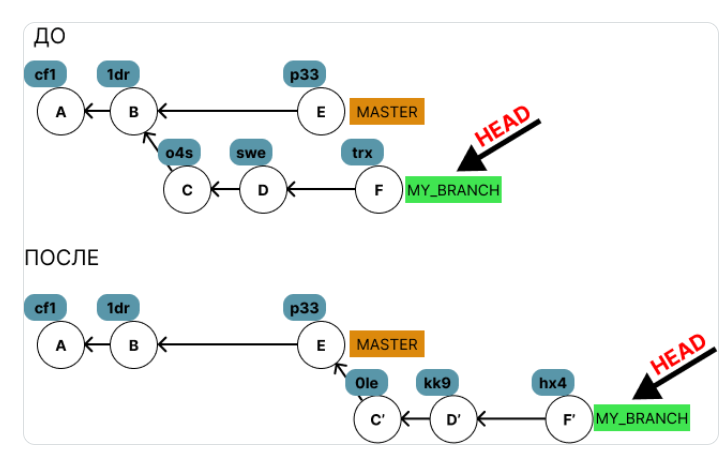
git push origin v1.0

Используя git tag, разработчики могут легко организовывать и отслеживать версии проекта, облегчая работу с релизами и обновлениями.

**git rebase**

Git rebase — перебазирование коммитов и веток

Rebase (перебазирование) — один из способов в git, позволяющий объединить изменения двух веток. У этого способа есть преимущество перед merge (слияние) — он позволяет переписать историю ветки, придав тот истории тот вид, который нам нужен.



**Git rebase — что это**

Из документации [7]— это наложение коммитов поверх другого базового коммита. Под базовым понимается тот коммит, к которому применяются коммиты выбранной ветки.

git rebase [<upstream> [<branch>]]

Первый аргумент обязательный (upstream) — это базовый коммит, к которому применятся коммиты выбранной ветки. Второй аргумент можно не задавать, если HEAD указывает на ветку, которая будет нами перебазирована.

Чтобы понимать процесс работы перебазирования, обратимся к рисунку 1.

У нас есть две ветки — master и my\_branch. Мы находимся на ветке my\_branch (HEAD указывает на ветку my\_branch). Выполняем команду:

git rebase master

После этого git удалит и последовательно переместит коммиты C, D, F из ветки my\_branch в ветку master — сначала C, затем D и F. Новые коммиты C’, D’, F’ полностью идентичны удаленным, меняется только хеш.

Сначала для ветки my\_branch базовым коммитом был B, но после стал коммит E. Это и есть процесс под названием перебазирование.

**Как использовать git rebase**

Перебазирование в git используется для придания линейности истории ветки, чтобы удобно отслеживать изменения, или для обновления ветки разработки последними изменениями из основной ветки. Также есть и другие варианты использования — с помощью интерактивного режима и параметра **—onto**.

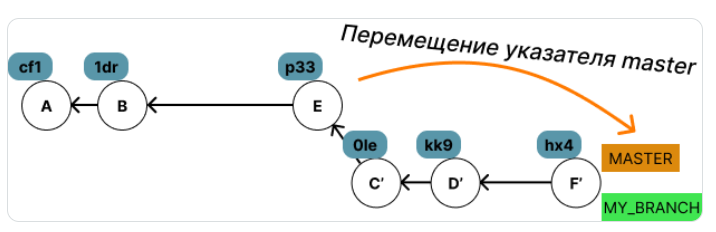
Линейная история — реинтеграция тематической ветки после выполнения git rebase master

После того как мы использовали команду git rebase, можно перемотать ветку master командой git merge:

git checkout master

git merge my\_branch

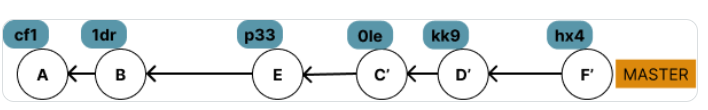
К команде слияния можно добавить флаг либо **—ff** (fast-forward merge), чтобы не создавать коммит слияния, или добавить **—no-ff** — для его создания. Создание коммита слияния помогает определить, когда ветки объединились, и какие коммиты тематической ветки были сделаны.



Когда работа с тематической веткой закончена, удаляем ее:

git branch -d my\_branch

Это приведет к законченному виду истории, когда мы внедрили изменения из тематической ветки в основную часть проекта.



**Конфликты**

Так как git rebase последовательно переприменяет коммиты, то могут возникнуть конфликты слияния (merge conflicts). Первая причина появления конфликта  — объединение коммитов, содержащих изменения в одних и тех же файлах. Вторая причина — несколько человек изменяют одинаковый файл на одной расшаренной ветке. Чтобы узнать, в каких файлах есть конфликтующие изменения, проверим статус.

git status

Нам будет предложено решить конфликтные коммиты, затем пометить их решенными:

git add/rm <conflicted\_files>

Дальше нужно продолжить перебазирование:

git rebase --continue

Или еще откатить изменения — вернуться в состояние до использования команды rebase.

git rebase --abort

Есть и третий вариант с перезапуском шага и перезагрузкой процесса перебазирования:

git rebase --skip

Но будьте аккуратны, skip пропустит (удалит) конфликтный коммит.

git rebase interactive

Интерактивный режим rebase используется для перезаписи истории посредством изменения самих коммитов, а также информации в них. Переход в интерактивный режим перебазирования делается при помощи флага -i или —interactive.

git rebase [-i | --interactive]

Выполнение этой команды создаст список коммитов в хронологическом порядке добавления, чтобы пользователь мог по своему желанию отредактировать их перед последующим перебазированием. Дальше мы рассмотрим подробнее, как это происходит.

Как пользоваться интерактивным режимом

Например, вот созданная ветка master с пятью коммитами:

| 2hqsibn selected new method in script.js (HEAD -> master)  4kq5jn2 changes to the script.j  qk01ru3 resolved conflict  fmjgyu6 added new files  z2zgn0c initial commit (origin/master) |
| --- |

**Замечание**. Когда вы выполняете команду git rebase -i <commit-hash> для выполнения интерактивного rebase, Git открывает текстовый редактор, в котором вы можете выбрать действия для каждого коммита, с которым вы работаете в процессе ребейза.

Если Git не может найти стандартный текстовый редактор (например, Emacs), он выдает ошибку. В таком случае, установка переменной окружения GIT\_EDITOR на значение nano (или другой текстовый редактор, который у вас установлен) позволяет указать Git использовать этот редактор вместо стандартного

В Windows PowerShell: $env:GIT\_EDITOR = "nano"

В Git Bash: $ export GIT\_EDITOR=nano

Помимо редактора nano, вы можете установить и использовать другие текстовые редакторы для работы с Git. Некоторые из распространенных текстовых редакторов, которые можно использовать в Git, включают:

Vim: Vim - мощный текстовый редактор с широкими возможностями. Для установки переменной GIT\_EDITOR на значение vim, вы можете использовать:

В Windows PowerShell: $env:GIT\_EDITOR = "vim"

В Git Bash: $ export GIT\_EDITOR=vim

Notepad++: Notepad++ - легкий и расширяемый текстовый редактор для Windows. Для использования Notepad++ в качестве редактора для Git, установите переменную GIT\_EDITOR следующим образом:

В Windows PowerShell: $env:GIT\_EDITOR = "notepad++"

В Git Bash: $ export GIT\_EDITOR=notepad++

Sublime Text: Sublime Text - популярный текстовый редактор с обширными возможностями и плагинами. Для использования Sublime Text в Git, установите переменную GIT\_EDITOR так:

В Windows PowerShell: $env:GIT\_EDITOR = "subl"

В Git Bash: $ export GIT\_EDITOR=subl

Выбор редактора зависит от ваших предпочтений и установленных программ на вашем компьютере.

**Поставлено две задачи:**

* 1. Поменять местами коммиты qk01ru3 и 4kq5jn2;
  2. исправить ошибку в комментарии четвертого коммита (4kq5jn2).

Для этого мы включаем интерактивный режим rebase. В нем можно указать определенное количество коммитов для изменения. Чтобы это сделать, необходимо передать в аргумент коммит, предшествующий тому, который мы будем изменять (в данном случае это fmjgyu6), либо задать “HEAD~[x]”, где вместо [x] — нужное нам число коммитов.

git rebase -i HEAD~3

Откроется текстовый редактор по умолчанию, где также будет приведено описание команд, используемых в интерактивном режиме rebase:

| pick qk01ru3 resolved conflict  pick 4kq5jn2 changes to the sсript.j  pick 2hqsibn selected new method in script.js  # Rebase fmjgyu6..2hqsibn onto fmjgyu6  #  # Commands:  # p, pick <commit> = use commit  # r, reword <commit> = use commit, but edit the commit message  # e, edit <commit> = use commit, but stop for amending  # s, squash <commit> = use commit, but meld into previous commit  # f, fixup <commit> = like "squash", but discard this commit's log message  # x, exec <command> = run command (the rest of the line) using shell  # b, break = stop here (continue rebase later with 'git rebase --continue')  # d, drop <commit> = remove commit |
| --- |

Обратите еще раз внимание, что коммиты в интерактивном режиме расположены в другом порядке, от более старого к новому. Приступим к решению задач.

Для того чтобы поменять коммиты местами, мы просто меняем местами их строчки.

Чтобы исправить ошибку в комментарии, мы меняем команду “pick” в 4kq5jn2 на “reword” и в следующем окне переписываем комментарий на “changes to the script.js”.

В итоге имеем следующий порядок коммитов:

| reword 4kq5jn2 changes to the sсript.js  pick qk01ru3 resolved conflict  pick 2hqsibn selected new method in script.js |
| --- |

Не забываем сохранять изменения, как вы обычно делаете при сохранении в вашем текстовом редакторе. Выходим из интерактивного режима закрытием текстового редактора.

Теперь наша ветка имеет вид:

| vh9xwf3 selected new method in script.js (HEAD -> master)  s70zmpx resolved conflict  b0jemdh changes to the script.js  fmjgyu6 added new files  z2zgn0c initial commit (origin/master) |
| --- |

Имейте в виду, что хеши коммитов, с которыми работал rebase, станут иными.

**Какой режим выбрать: стандартный или интерактивный**

Интерактивный rebase — продвинутая версия обычного rebase, которая дает возможность большего взаимодействия с коммитами. Если нужно разделить, объединить, удалить коммиты, изменить их описание и порядок, интерактивный режим справится на отлично. Если же такой потребности нет, быстрее будет использовать стандартный режим rebase, в котором от пользователя не требуются лишние действия до возникновения конфликтов.

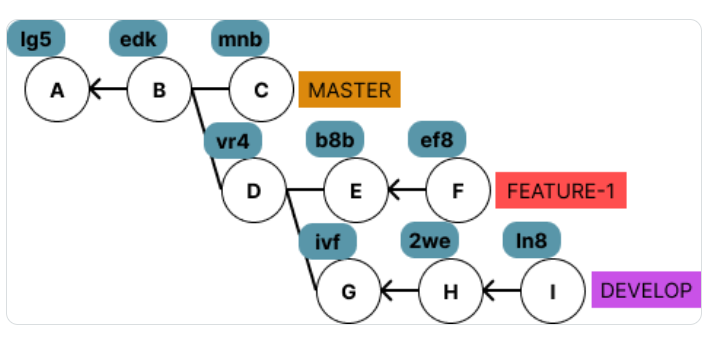
git rebase —onto

Onto относится к возможностям, раскрывающих rebase во всей красе.

git rebase --onto <newbase> [<upstream> [<branch>]]

Эта опция позволяет указать коммит, с которого будут перебазироваться коммиты (первый аргумент), иначе это называется новая база. Третий аргумент (branch) можно не указывать, если HEAD указывает на ветку, которая будет нами перебазирована.

**Пример**. У нас есть три ветки: master, feature-1, develop.

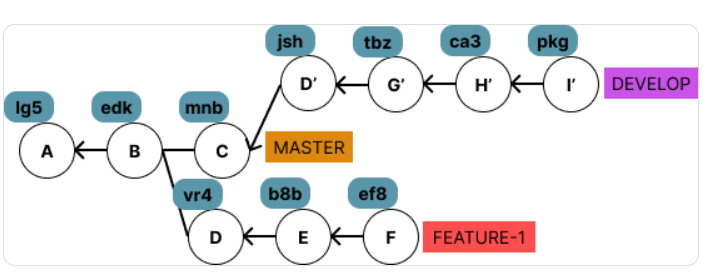


Нам нужно интегрировать изменения из ветки develop в master. Воспользуемся обычным rebase:

git checkout develop

git rebase master

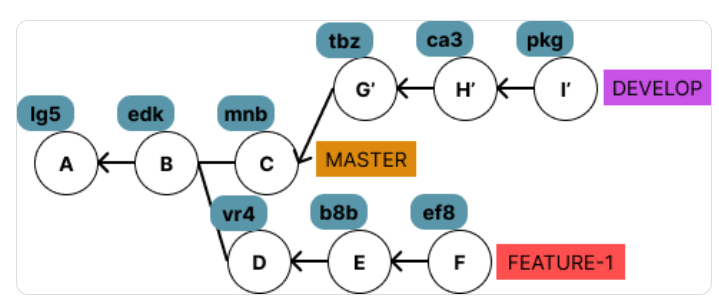
Тогда получим вот такой результат с двумя одинаковыми коммитами D:



Это произошло потому, что перебазирование переприменило коммиты D, G, H, I, так как по отношению к ветке master коммиты ветки develop начинаются с коммита D как связывающего коммита. Теперь решим задачу с интеграцией изменений из develop немного по-другому. Чтобы избежать таких случаев как с коммитом D, воспользуемся новой командой:

git rebase --onto master feature-1 develop

Результат проиллюстрирован на рисунке ниже.



**Rebase удаленного репозитория**

При работе с удаленным репозиторием в тематической ветке, например на GitHub, следует быть осторожным, используя rebase. Как вы уже знаете, rebase перезаписывает историю, в процессе изменяются хеши коммитов, а это может привести к конфликтам в работе с веткой у других членов команды. Поэтому, если вы работаете над тематической веткой не одни, стоит прибегнуть к нескольким правилам для предотвращения возможных проблем.

Синхронизировать изменения. Перед тем как вы будете заливать свой код на тот же GitHub, выполните git pull изменений, чтобы избежать конфликтных ситуаций.

Не перебазировать давно созданные ветки. Количество шагов в rebase равно количеству коммитов на перебазируемой ветке, если не указаны иные опции. Поэтому с увеличением количества неперебазированных коммитов, растет и вероятность появления конфликта.

Чтобы внедрить изменения в мастер, стоит создать свою локальную ветку и перебазировать ее поверх origin/master. Тогда останется лишь сделать перемотку или бесконфликтное слияние для владельца.

Не проводить rebase уже отправленных коммитов в публичный репозиторий. Коллегам придется выполнить слияние, что приведет к путанице.

**Pull rebase**

Чтобы запушить свою ветку, когда git не знает, как объединить ветки, используется режим force:

git push origin <branch> --force

С этим режимом будут скопированы родительские коммиты feature на origin, указатель перемещается, как он установлен на локальном репозитории. Важно указать идентификатор ветки в <branch>, иначе запушатся все локальные ветки ориджина.

А чтобы извлечь изменения из удаленного репозитория, вместо обычного pull можно использовать режим rebase:

git pull --rebase origin <branch>

Локальные merge коммиты не образуются, а история будет выглядеть линейно.

**Дополнительные опции перебазирования**

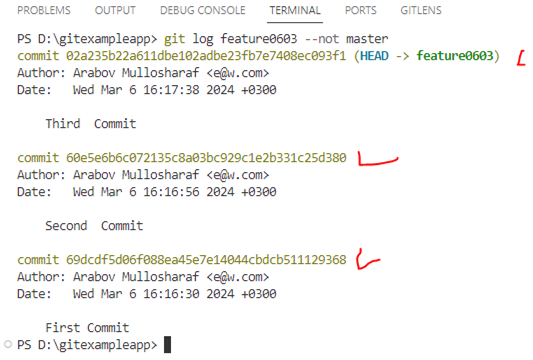
Ниже приведена таблица некоторых опций, которые могут быть полезны для работы с rebase.

| Опции | Пояснение |
| --- | --- |
| -s <strategy>—strategy=<strategy> | Использовать стратегию слияния вместо дефолтного “ort”, что изменит поведение rebase. [Подробнее](https://git-scm.com/docs/merge-strategies) в документации. |
| -X <strategy-option>—strategy-option=<strategy-option> | Эта опция для применения более одной стратегии в порядке, заданном пользователем. |
| -x <cmd>—exec <cmd> | Выполнение одной или более shell-команд после каждого шага rebase в интерактивном режиме. Если выполнение команды неудачно, перебазирование остановится. |
| —no-keep-empty | Не оставлять пустые коммиты. То есть убирать те коммиты, которые ничего не меняют по отношению к родителю. |
| —allow-empty-message | Позволяет перебазировать пустые коммиты с пустым сообщением. |
| —autosquash | В интерактивном режиме берет коммиты, которые начинаются с fixup! или squash! и ставит соответствующую команду, чтобы объединить коммит с предыдущим.Чтобы это значение всегда работало по умолчанию, можно прописать следующее:“git config —global rebase.autosquash true” |

**Конкретный пример**

Предположим, что в нашей ветке feature0603 у нас есть 3 коммита.

git log feature0603 --not master



Требуется изменить описание «Second Commit» на «My Second Commit».

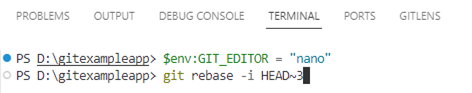
**Решение.**

1) Установим переменную окружения GIT\_EDITOR на значение nano.

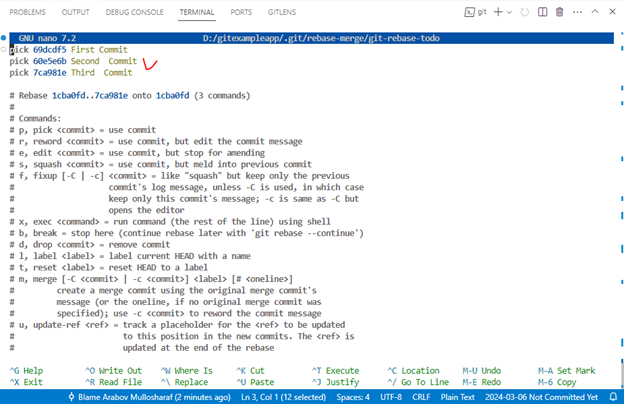
$env:GIT\_EDITOR = "nano"



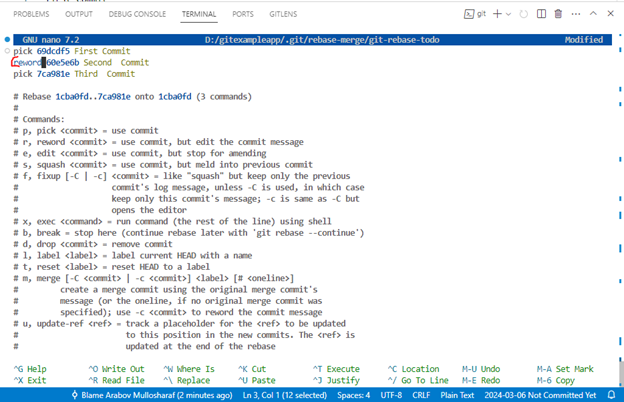
2) Введите команду git rebase -i HEAD~N (или git rebase -i SHA\_Commit^ для конкретного коммита), где N - количество коммитов, которые вы хотите изменить (в вашем случае, вероятно, N=3).



3) В результате откроется интерактивный режим перебазирования.



4) В редакторе измените pick на reword для нашего коммита, то есть для коммита, описание которого вы хотите изменить. В нашем случае «Second Commit».



5) Сохраните изменения и закройте редактор.

4.1. Нажмите Ctrl + O для сохранения изменений.

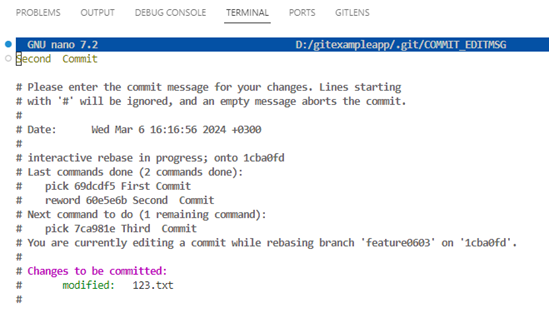


Файл todo для операции rebase в Git содержит инструкции о том, как обрабатывать каждый коммит в процессе перебазирования. В конце файла указано "Имя файла для записи", где вы должны указать путь к файлу, в который будет записан результат выполнения ребейза.

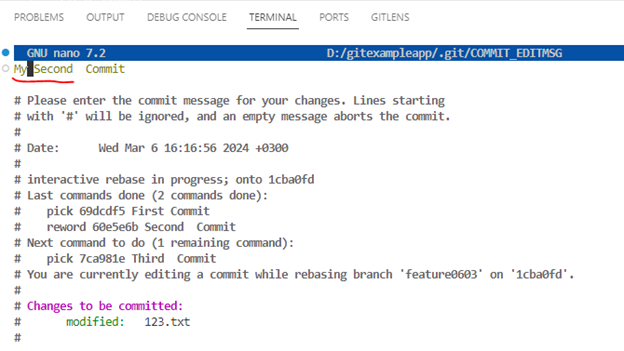
Если вы завершите редактирование файла todo для операции rebase в Git, но не укажете имя файла для записи и просто сохраните изменения, то Git сохранит результаты ребейза в исходном файле, который был открыт для редактирования. Это означает, что изменения будут сохранены в файле D:/gitexampleapp/.git/rebase-merge/git-rebase-todo, который был указан в начале файла.

4.2. Нажмите Enter.

4.3. Затем нажмите Ctrl + X для выхода из редактора. После выход из редактора.



4.4. Введите новое описание коммита, если требуется.



После завершения редактирования сообщения коммита, вы можете сохранить изменения, нажав Ctrl + O, затем нажмите клавишу Enter, и выйти из редактора, нажав Ctrl + X.

5) Проверим изменения:



**Литература**

1. [Introduction to Git and GitHub for Python Developers](https://realpython.com/python-git-github-intro/#aside-what-is-a-sha)
2. [Python Virtual Environments: A Primer](https://realpython.com/python-virtual-environments-a-primer/)
3. [Python.gitignore](https://github.com/github/gitignore/blob/main/Python.gitignore)
4. [How is git commit sha1 formed · GitHub](https://gist.github.com/masak/2415865)
5. [Git - Revision Selection](https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection)
6. [Добро пожаловать в LearnGitBranching!](https://learngitbranching.js.org/?locale=ru_RU)
7. [git-rebase Documentation](https://git-scm.com/docs/git-rebase)
8. <https://www.atlassian.com/ru/git/tutorials>
9. [Git - Documentation](https://git-scm.com/doc)
10. [GitHowTo](https://githowto.com/ru)