[Введение в Git](https://selectel.ru/blog/courses/introduction-to-git/)

# Введение в Git: настройка и основные команды.

Как установить и настроить Git в различных ОС, создать новые и клонировать существующие репозитории, а также базовые концепции ведения веток.

## Введение.

Git — один из видов систем контроля версий (или СКВ). Такие системы записывают изменения в набор файлов, а позже позволяют вернуться к определенной версии.  
Вам может пригодиться СКВ, если вы, например, программист, системный администратор, дизайнер (или в целом работаете с массивом изменяющихся файлов) и хотите сохранить каждую версию проекта. Вы сможете вернуться к любому из сохраненных состояний, просмотреть изменения и увидеть их авторов. Так гораздо проще исправлять возникающие проблемы.

В целом СКВ можно разделить таким образом:

* Локальные — все файлы хранятся только в вашей операционной системе, например, разложены по папкам с версиями.
* Централизованные — проект хранится на сервере, а ваша рабочая версия включает только текущий набор файлов.
* Распределенные — копии проекта (и вся информация о версиях) располагаются не только на сервере, но и на нескольких клиентских машинах, чтобы обеспечить устойчивость к отказу сервера.

Очевидно, что Git — не единственная система контроля версий, однако по многим параметрам самая удобная и популярная на сегодняшний день. Благодаря распределенной структуре репозитории Git хранятся на всех клиентских компьютерах, что защищает от потерь данных и позволяет полноценно управлять версиями проекта оффлайн.

Главная отличительная черта Git состоит в подходе к обработке данных. Каждый раз при сохранении данных проекта (коммите) система фиксирует состояние файла (делает снимок) и создает ссылку на этот снимок. Последующие изменения отражаются через ссылки на более ранние версии файла. Нет необходимости снова сохранять файл целиком. К тому же, основываясь на контрольных hash-суммах, система снимков обеспечивает целостность всей истории изменений. На практике это означает, что невозможно (либо крайне трудно) полностью удалить данные из рабочего каталога и утратить к ним любой доступ. В большинстве случаев данные можно восстановить из ранней версии проекта.

Таким образом, систему контроля версий в Git проще всего представлять как поток снимков (сохраненных состояний проекта).

## Принципы работы с Git.

У проектных файлов в Git есть 3 базовых состояния

* Измененные (modified) — файлы в процессе рабочего редактирования.
* Индексированные (staged) — та часть измененных файлов, которая уже подготовлена к фиксации после редактирования.
* Зафиксированные (committed) — файлы, уже сохраненные в локальном репозитории.

У Git есть рабочий каталог, где хранятся метаданные и локальная база рабочего проекта. Именно эта часть копируется, когда вы **клонируете** проект (репозиторий) с сервера.

Чаще всего работа с Git устроена примерно так:

1. Вы вносите правки в файлы рабочей копии проекта.
2. Индексируете их, подготавливая к коммиту (здесь Git создает снимки новых правок).
3. Делаете коммит, и индексированные правки наконец сохраняются в вашем каталоге Git.

## Установка Git.

Создать свой проект и начать пользоваться Git в нем достаточно просто. Мы будем рассматривать работу в командной строке терминала, потому что там реализован полный набор команд. Вероятно, в будущем вам будет проще воспользоваться встроенными инструментами в крупном приложении (например, в Visual Studio, если вы программист).

Однако командная строка все равно удобна для тонкой настройки и «нестандартных» действий, поэтому полезно представлять себе, как управлять проектом через нее.

Сначала потребуется установить Git на свой компьютер.

### Установка в Linux.

Для дистрибутивов, похожих на Fedora, RHEL или CentOS, выполните команду dnf:

**> sudo dnf install git-all**

На Ubuntu и других Debian-подобных систем введите apt:

**> sudo apt install git**

Более подробные сведения можно получить по ссылке: <https://git-scm.com/download/linux>.

### Установка на Mac.

Один из способов установки — воспользоваться Xcode Command Line Tools. В терминале нужно ввести:

**> git --version**

И следовать дальнейшим инструкциям.

Если вы пользуетесь Homebrew, запустите команду:

**$ brew install git**

Подробности доступны по ссылке: <https://git-scm.com/download/mac>.

### Установка в Windows.

Новейшая сборка доступна на официальном сайте Git по ссылке: <https://git-scm.com/download/win> (загрузка запустится автоматически).

## Настройка Git.

Настроить рабочую среду нужно только один раз — после обновлений параметры не сбросятся. Если понадобится, в любое время можно изменить ваши настройки.

Самый удобный способ изменения конфигурации — встроенная утилита **git config**. Настройки Git имеют три уровня:

1. Параметры из файла **[path]/etc/gitconfig** (системные) могут работать для всех пользователей системы и репозиториев. Они редактируются командой **git config —system**.
2. Параметры из файла **~/.gitconfig** или **~/.config/git/config** (глобальные) применяются к одному пользователю, если запустить команду **git config —global**.
3. Локальные параметры из файла **config** в рабочем каталоге **.git/config** сохраняют только для выбранного репозитория. Ему соответствует команда **git config —loca**l.

Если запускать ***git config*** без параметров, будет использоваться локальный уровень, никакие из более глобальных настроек не изменятся.

Всю используемую конфигурацию можно просмотреть так:

**> git config --list --show-origin**

Представимся Git, чтобы в рабочих коммитах сохранялось ваше авторство:

**> git config --global user.name "Danil Z"**

**> git config --global user.email danilz@danilz.com**

Также можно выбрать и текстовый редактор, введя команду **git config —global core.editor**. Например, чтобы выбрать Emacs, выполните:  
> git config --global core.editor emacs

В Windows нужно указывать полный путь к файлу. К примеру, для установки Notepad++ нужно запустить подобную команду:  
> git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

Стоит отметить, что на практике текстовый редактор в Git может и не пригодиться, особенно если вы активно используете стороннее ПО — например, в Visual Studio все текстовые заметки для Git можно писать в отдельном окне. Текстовые редакторы в командной строке отличаются своеобразным управлением, которое потребует от вас отдельного изучения.

Общий список текущих настроек просматривается с помощью команды **git config —list**. Проверить, что записано в любой из доступных настроек, можно командой с ключом  **git config <key>**:  
> git config user.email

### Выбор ветки по умолчанию.

Итак, наконец можно создать репозиторий в выбранном каталоге командой **git init**. Основная ветка автоматически будет названа *master*. Изменить это (в нашем случае задав ветку *main*) можно так:

> git config --global init.defaultBranch main

## Работа в репозитории.

Как правило, есть два варианта начать работу с репозиторием Git:

1. Можно выбрать локальный каталог и создать новый репозиторий в нем.
2. Можно клонировать существующий репозиторий с локального компьютера или сервера. Обычно проекты клонируются именно с сервера.

Если у вас на компьютере уже есть рабочий проект, но еще не назначен контроль версий, то нужно сначала перейти в каталог проекта.

Linux:

> cd /home/user/SomeConsoleApp

macOS:

> cd /Users/user/SomeConsoleApp

Windows:

> cd C:/Users/user/SomeConsoleApp

Инициализируем репозиторий:

> git init

Команда создаст каталог с именем **.git**, в котором будут храниться структурные файлы репозитория.

И, наконец, нужно добавить под контроль версий все существующие файлы командой **git add .** (точка в конце важна!). Можно добавлять и по одному файлу, с помощью **git add <имя файла>**.

Заодно создадим начальный коммит командой**git commit**:

> git add readme.md

> git commit -m 'Initial project version'

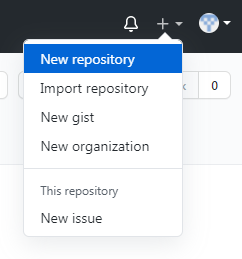
Команду **git add**можно гибко настраивать с помощью дополнительных параметров (флагов), которые подробно описаны в официальной документации: <https://git-scm.com/docs/git-add>. К примеру, команда **git add —force** добавит даже игнорируемые файлы, а **git add —update** позволит обновить отслеживаемые файлы.

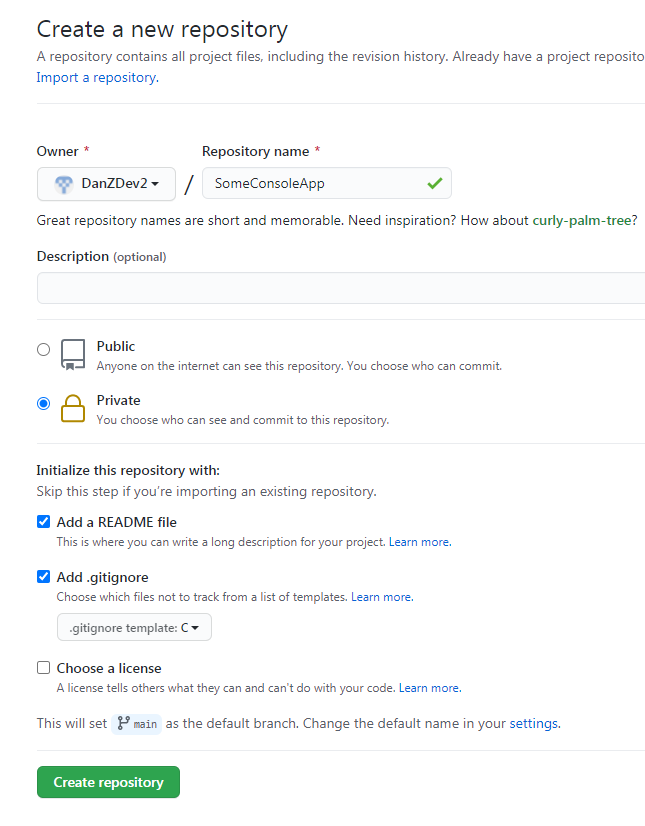
В этом репозитории вы можете продолжать работать и дальше, со временем обновляя его и отправляя рабочие версии на сервер.

### Клонирование существующего репозитория.

Когда вы работаете в команде, разрабатываемые проекты часто размещают на сервере. Это один из самых распространенных сценариев. Вам нужно получить копию проекта последней версии на свой компьютер, чтобы далее вносить в него свой вклад.

В качестве примера мы будем рассматривать проект, который создадим на ресурсе https://github.com/ . После регистрации на сайте и подтверждения по e-mail нужно создать новый репозиторий, как показано на скриншотах.





Видно, что можно выбрать тип репозитория:

* публичный (public) – доступ открыт для любого пользователя, однако права на редактирование выдает владелец проекта;
* приватный/скрытый (private) — проект виден только владельцу, другие участники добавляются вручную.

Для нашего примера создадим приватный репозиторий под названием [*SomeConsoleApp*](https://github.com/DanZDev2/SomeConsoleApp)и будем работать с ним далее.

Самые удобные способы клонирования проекта — через протоколы HTTP и SSH, прочесть обо всех более развёрнуто можно по ссылке: <https://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols>.

Для наших целей воспользуемся протоколом https и следующей командой:

> git clone https://github.com/DanZDev2/SomeConsoleApp SomeConsoleApp

На вашем компьютере в каталоге, куда вы перешли в командной строке, должен появиться каталог SomeConsoleApp, внутри него — каталог .git и все скачанные файлы репозитория последней версии.

После получения проекта обычно начинается более рутинный рабочий процесс — правки, добавление функционала и т. д. Далее в какой-то момент вы захотите сохранить прогресс в новой версии проекта.

Правила и периодичность обновления могут быть почти любыми, но хорошим тоном обычно считается сохранять рабочую (или промежуточно завершенную) версию. Важное требование для команд разработчиков — возможность сборки проекта, иначе другие участники команды будут вынуждены тратить время на борьбу с ошибками компиляции.

### Сохранение снимков и просмотр статуса проекта.

Как упоминалось ранее, часть файлов в рабочем каталоге может и не находиться под контролем версий. За отслеживаемыми файлами «наблюдает» Git, они были как минимум в прошлом снимке состояния проекта. Неотслеживаемыми могут быть, например, вспомогательные файлы в рабочем проекте, если они не зафиксированы в прошлой версии проекта и не готовы к коммиту. Их можно выделить в отдельную категорию для Git, о чем будет рассказано далее.

Сразу после клонирования все файлы проекта будут отслеживаемыми. Отредактировав их и привнеся что-то новое, вы индексируете (*stage*) и фиксируете (*commit*) правки, и так для каждой версии проекта.

При этом нужно внимательно следить, чтобы вспомогательные файлы, особенно объемные, оставались вне контроля версий. Если по недосмотру добавить их в коммит и отправить на сервер — вероятнее всего, ваши правки придется частично откатывать.

Проверить состояние файлов в рабочем каталоге можно командой **git status**. После клонирования консоль выведет примерно такую информацию:

On branch master

Your branch is up to date with ‘origin/master’.

nothing to commit, working tree clean

Теперь отредактируем файлы (в этом примере было консольное демо-приложение, созданное с помощью Visual Studio) и сравним статус:

>git status

On branch master

Your branch is up to date with ‘origin/master’.

Untracked files:

(use “git add <file>...” to include in what will be committed)

Program.cs

SomeConsoleApp.csproj

SomeConsoleApp.sln

nothing added to commit but untracked files present (use “git add” to track)

Теперь зафиксируем изменения. В коммит войдут только те файлы, которые вы изменили и добавили командой **git add**. Остальные будут лишь дополнительными файлами в каталоге проекта.

Стандартный способ — команда **git commit**, которую мы уже видели раньше. Без дополнительных аргументов она откроет встроенный текстовый редактор, поэтому для простоты рекомендуется добавить аргумент **-m**и вписать комментарий в кавычках:

> git commit -m "Task 2: basic project template added"

Для удаления ненужных файлов из репозитория можно использовать команду **git rm <file-name>**. Файл также пропадет из рабочего каталога. Выполнить коммит необходимо и в этом случае; до тех пор структура проекта не изменится.

### Файл .gitignore

Как упоминалось ранее, в рабочий каталог могут попадать файлы, которые вам бы не хотелось отправлять на сервер. Это и документы с вашими экспериментами или образцами, и автоматически генерируемые части проекта, актуальные только на вашем компьютере. Git может полностью игнорировать их, если создать в рабочем каталоге файл с названием **.gitignore** и внести в него все имена ненужных файлов и папок.

Открывать файл можно в любом текстовом редакторе. Обычно удобнее не перечислять абсолютно все имена (которые к тому же всегда известны), а воспользоваться подобными инструкциями:

/bin

/obj

\*.pdb

\*.exe

Если прописать такое содержимое файла .gitignore, то репозиторий git будет полностью игнорировать папки /bin и /obj, а также любые файлы с расширениями .pdb и .exe, хранящиеся в вашем рабочем каталоге.

Рекомендуется создавать .gitignore до первой отправки вашего проекта в удаленный репозиторий, чтобы на сервер не попало никаких лишних файлов и каталогов. Разумеется, важно проверить, чтобы в .gitignore не были упомянуты критичные для проекта файлы, иначе у других участников команды возникнут проблемы после следующего обновления.

### Управление удаленными репозиториями.

Просмотреть список текущих онлайн-репозиториев можно командой **git remote**. Добавить другие — с помощью команды **git remote add <shortname> <url>**, например:

>git remote add myDemo https://github.com/DanZDev2/DemoApp

>git remote

myDemo

origin

### Отправка изменений в удаленный репозиторий (Push).

На вашем компьютере есть проект со внесенными изменениями, но вы хотите поделиться новой версией со всей командой.

Команда для отправки изменений на сервер такова: **git push <remote-name> <branch-name>**. Если ваша ветка называется *master*, то команда для отправки коммитов станет такой:

> git push origin master

Она сработает, если у вас есть права на запись на том сервере, откуда вы клонировали проект. Также предполагается, что другие участники команды за это время не обновляли репозиторий.

Следует к тому же помнить, что в разработке для промежуточных правок часто используется не главная ветка (master), а одна из параллельных (например, Dev). Работая в команде, этому обязательно нужно уделять пристальное внимание.

### Получение изменений из репозитория (Pull).

Самый простой и быстрый способ получить изменения с сервера — выполнить команду git pull, которая извлечет (*fetch*) данные с сервера и попытается встроить/объединить (*merge*) их с вашей локальной версией проекта.

На этом этапе могут возникать конфликты версий, когда несколько человек поработали над одними и теми же файлами в проекте и сохранили свои изменения. Избежать этого можно, если изолировать части проекта, поручив работу над одной частью только одному человеку. Разумеется, на практике это не всегда выполнимо, поэтому в Git есть инструменты для разрешения конфликтов версий. Они будут рассмотрены далее.

### Создание веток и переключение между ними.

Создадим две дополнительные ветки Dev и Test (например, одна может пригодиться для процесса разработки, а другая — для запуска в тестирование). Введем команду **git branch <branch-name>** дважды с разными аргументами:

>git branch Dev

>git branch Test

Ветки созданы, но мы по-прежнему работаем в *master*. Для переключения на другую нужно выполнить **git checkout <branch-name>**:

>git checkout Dev

Switched to branch ‘Dev’

Your branch is up to date with ‘origin/Dev’.

Внесем некоторые изменения в файл *README.md* и зафиксируем их, чтобы они отразились в ветке *Dev*:

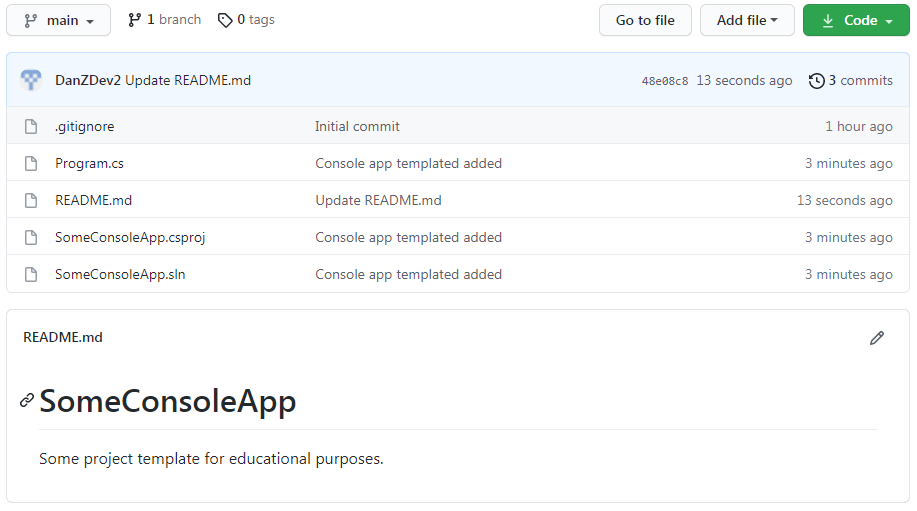
>git add .

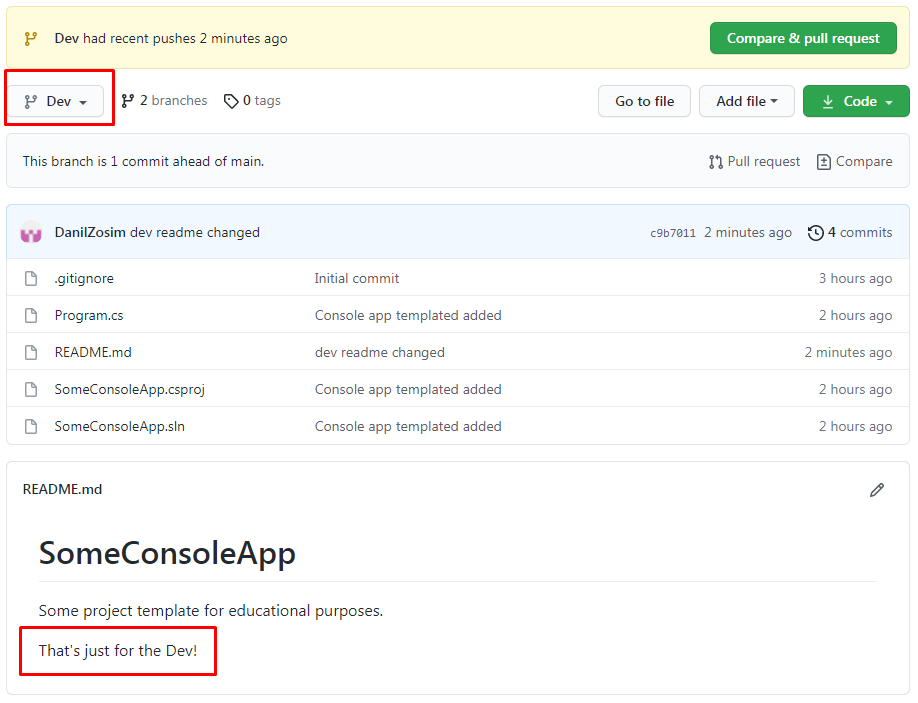
>git commit -m “dev readme changed”

[Dev #####] dev readme changed

1 file changed, 2 insertions(+)

Если теперь отправить их на сервер, то можно убедиться в появившемся отличии веток:





Для переключения обратно на ветку *master* нужно снова ввести команду **git checkout master**. Она не изменялась, а значит, после редактирования проекта ветки разойдутся. Это нормальная ситуация для проектов в Git. Важно только понимать, для каких целей используется каждая из веток, и не забывать вовремя переключаться между ними.

### Слияние веток (merge).

Работа над проектами часто ведется в несколько этапов, им могут соответствовать ветки (в нашем примере Dev → Test → master). Отдельные ветки могут создаваться для срочного исправления багов, быстрого добавления временных функций, для делегирования части работы другому отделу и т. д. Предположим, что нужно применить изменения из ветки *Dev*, внеся их в *master*. Перейдем в *master* и выполним команду **git merge <source-branch>**:

>git merge Dev

Updating #####..#####

Fast-forward

README.md | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

Изменения успешно перенесены. В наших упрощенных условиях команда завершилась без ошибок, не найдя конфликтов в файлах. Если же над общими участками какого-либо файла успели поработать несколько человек, с этим нужно разбираться вручную. При возникновении ошибок Git помечает общие части файлов из разных веток и сообщает о конфликте.

Для разрешения конфликтов есть консольная утилита git mergetool. Однако если файл проекта объемный, а общих частей много, пользоваться ей не слишком удобно. Общая рекомендация для таких случаев — пользоваться сторонними инструментами, как и в случае с текстовым редактором для Git.

Когда спорные участки всех файлов приведены к итоговому состоянию, нужно повторить стандартную процедуру: создать коммит и отправить их командой push в нужную ветку в репозитории.

Дальнейшая работа с проектом из репозитория Git, как правило, повторяется по алгоритму:

* pull (забрать изменения с сервера);
* внести правки, добавить что-то важное в проекте;  
  add (добавить изменённые файлы к коммиту);
* commit (сохранить состояние проекта с комментариями);
* push (отправить изменения на сервер).
* merge (при необходимости внедрить изменения из другой ветки проекта).

## Заключение.

Мы рассмотрели, как устанавливать и настраивать Git в различных ОС, создавать новые и клонировать существующие репозитории, получать и отправлять новые версии проекта, а также ознакомились с базовыми концепциями ведения веток.

Этой информации обычно хватает для повседневных задач, связанных с хранением рабочих проектов.

# Как установить Git.

В этой инструкции покажем, как установить Git на Windows, и поможем выбрать правильные параметры при установке.

## Введение.

В этой инструкции покажем, как установить Git на Windows, и поможем выбрать правильные параметры при установке. Затем создадим репозиторий и зафиксируем в нем изменения. Все это поможет вам сделать первые шаги в освоении Git.

## Что такое Git и зачем он нужен.

Git — это [одна из самых популярных](https://habr.com/ru/post/233935/) систем контроля версий (VCS). Такие системы помогают разработчикам хранить и версионировать исходный код приложений, настройки систем и другие текстовые файлы. И хотя ничего не мешает использовать VCS в других областях, чаще всего они применяются именно в IT.

Каждое состояние файлов в Git можно зафиксировать (сделать коммит), причем это навсегда останется в истории репозитория. Поэтому можно в любой момент посмотреть историю изменений файлов, сравнить различные версии и отменить отдельные изменения.

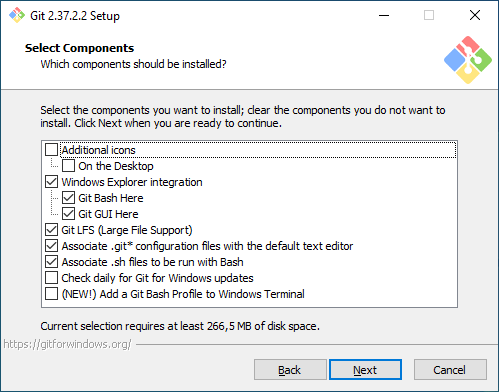
Также Git упрощает ведение параллельной разработки несколькими членами команды. Для этого используется ветвление. Условно можно сказать, что в Git-репозитории есть одна основная ветка, в которой хранится текущая стабильная версия исходного кода. Когда разработчик хочет изменить этот код, он «откалывает» себе отдельную ветку от основной и работает в ней. Когда работа закончена, он «вливает» изменения в основную ветку, чтобы его доработками смогли воспользоваться другие члены команды.

На самом деле все это описание довольно грубое, и по работе с Git можно написать не одну статью. На официальном сайте Git есть [бесплатная электронная книга](https://git-scm.com/book/), в том числе она переведена на русский язык. А в этой статье мы сосредоточимся на установке Git в Windows и его первоначальной настройке.

## Установка Git в Windows.

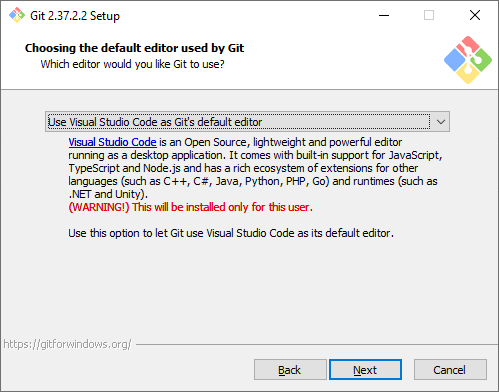
Переходим на [официальный сайт Git](https://git-scm.com/download/win), в раздел загрузок. Мы увидим несколько вариантов установки: разные разрядности, портативная версия и даже установка из исходников. Мы выберем Standalone-версию, для этого проще всего нажать ссылку Click here to download, она всегда ведет на самую актуальную версию. Запускаем скачанный файл.

**Выбор компонентов**. Первый экран — выбор компонентов для установки. Если вам нужны дополнительные иконки на рабочем столе, или если вы хотите, чтобы Git ежедневно проверял наличие новой версии, — отметьте соответствующие опции. Остальные параметры лучше оставить по умолчанию.

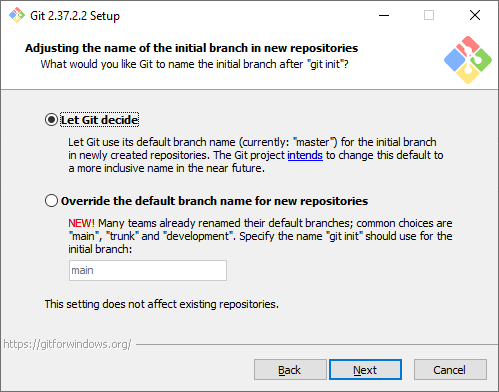


**Текстовый редактор по умолчанию**. Необходимо выбрать редактор, который будет использовать Git — например, когда вы будете писать сообщение для коммита. Это не обязательно должен быть редактор, в котором вы планируете писать исходный код.

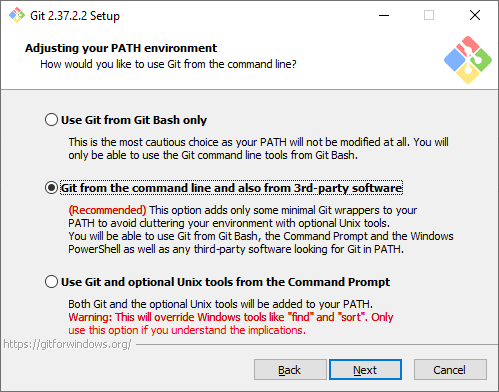
По умолчанию в установщике выбран Vim — консольный текстовый редактор, который для многих может показаться сложным в освоении. Если вы не знакомы с Vim и при этом хотите именно консольный редактор — выберите nano. Если у вас уже установлен какой-нибудь текстовый редактор — выбирайте его. Мы для примера будем использовать VSCode.



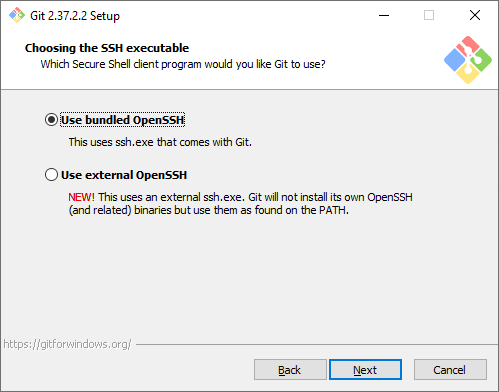
**Название первой ветки**. Тут нужно выбрать, как Git будет называть первую ветку в каждом репозитории. Раньше такая ветка всегда называлась master, но со временем это стало напоминать о временах рабства, и многие проекты и компании стали переименовывать ветки в своих репозиториях. Поэтому разработчики Git добавили эту опцию, чтобы название первой ветки можно было изменить. Мы будем придерживаться старого поведения и оставим название master.



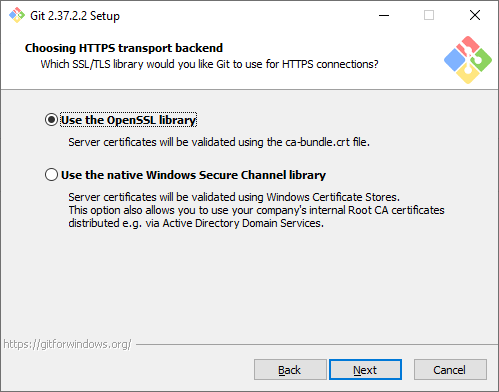
**Способ использования Git**. Первая опция сделает Git доступным только из командной строки Git Bash. Это не очень удобно, потому что не позволит пользоваться Git-ом из других оболочек или интегрировать его с редактором кода. Вторая опция самая оптимальная (ее мы и выберем) — она позволяет работать с Git-ом из разных оболочек и интегрировать его с другими приложениями. Третья опция кроме установки Git также «перезапишет» некоторые системные команды Windows аналогами из Unix, и эту опцию нужно выбирать только если вы точно понимаете, что делаете.



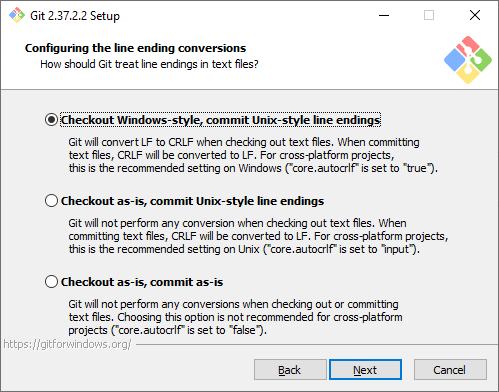
**Выбор SSH-клиента.**Изначально Git поставлялся со встроенным SSH-клиентом, но недавно появилась опция, где можно использовать внешний клиент. Если у вас уже что-то установлено на компьютере — можете выбрать вторую опцию. Мы же остановимся на первой, так как предварительно ничего не устанавливали.



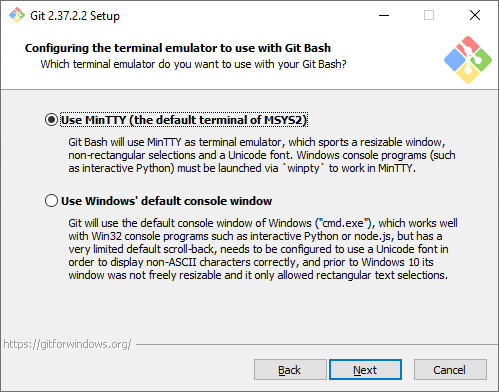
**Выбор SSL/TLS библиотеки**. По умолчанию Git будет использовать свою OpenSSL библиотеку с заранее определенным списком корневых сертификатов. Обычно этого достаточно, но если вам нужно работать со внутренними репозиториям внутри компании, которые используют самоподписанные сертификаты, выберите вторую опцию. Тогда Git будет использовать библиотеку и сертификаты из вашей операционной системы.



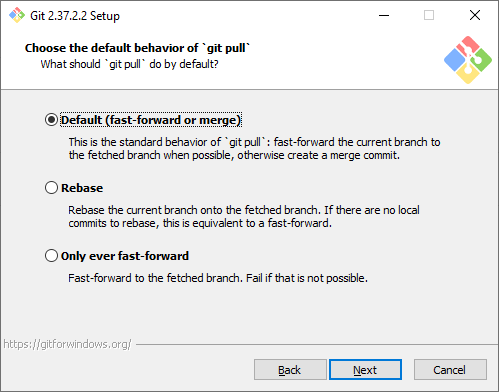
**Символы перевода строки**. Существует два основных способа формирования конца строки в файлах — CRLF и LF. Первый используется в Windows, второй — в Unix-like системах. Первая опция позволяет извлекать файлы из репозитория в Windows-стиле, при этом отправлять файлы в репозиторий в Unix-стиле. Мы рекомендуем использовать этот вариант, потому что он лучше всего подходит для кросс-платформенной команды, когда над одним кодом могут работать разработчики на разных ОС.



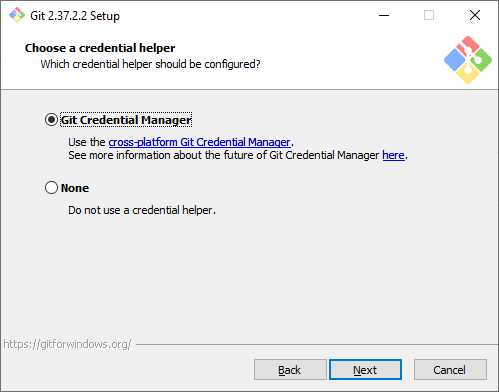
**Эмулятор терминала**. Эмулятор, который будет использоваться в командной строке Git Bash. MinTTY — удобный вариант, поэтому он выбран по умолчанию. Встроенный эмулятор CMD не очень удобен, у него есть некоторые ограничения, поэтому выбирайте его, только если делаете это осознанно.



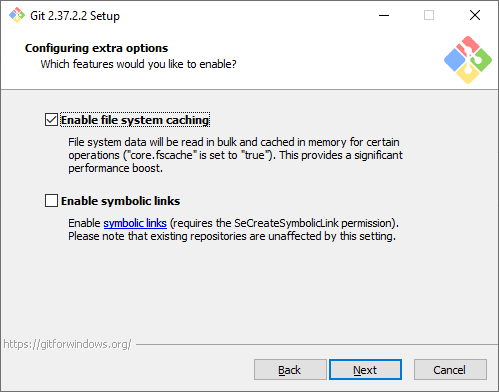
**Стратегия git pull**. Первая опция будет пытаться обновить историю коммитов без создания коммитов слияния. Это самый оптимальный и часто используемый вариант, оставим его.



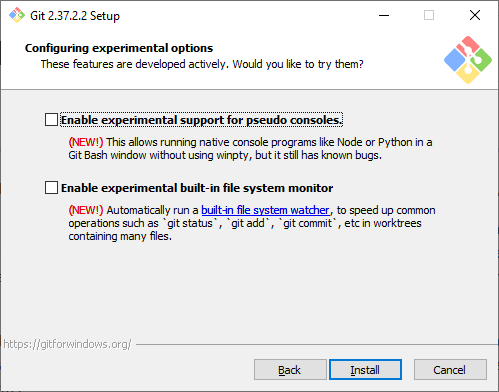
**Credential Manager**. Установка этого параметра позволит Git запоминать логины и пароли для подключения к удаленным репозиториям (например, GitHub, GitLab или корпоративное хранилище) и не вводить их постоянно.



**Дополнительные настройки**. Кэширование позволит ускорить работу Git, эту опцию рекомендуем оставить. А вот символические ссылки нам не нужны.



**Экспериментальные настройки**. Эти опции еще не переведены в стабильную стадию, поэтому их использование рекомендуется, только если вы точно понимаете, что делаете. Мы не будем ничего отмечать.



Git установлен и готов к работе.

## Установка в различные дистрибутивы Linux.

Также коротко покажем, как можно установить Git в различные дистрибутивы Linux. Как правило, самостоятельно скачивать ничего не нужно, достаточно воспользоваться встроенным в дистрибутив пакетным менеджером.

**Debian**

pt-get install git

**Ubuntu**

add-apt-repository ppa:git-core/ppa # apt update; apt install git

**Fedora 21**

yum install git

**Fedora 22+**

dnf install git

**Gentoo**

emerge --ask --verbose dev-vcs/git

**Arch Linux**

man -S git

**OpenSUSE**

ypper install git

**Mageia**

rpmi git

**FreeBSD**

pkg install git

**OpenBSD**

g\_add git

**RHEL, CentOS, Oracle Linux и др.**

Как правило, пакетный установит довольно старую версию Git, поэтому рекомендуется собирать Git из [исходных кодов](https://www.kernel.org/pub/software/scm/git/), или воспользоваться сторонним репозиторием [IUS Community](https://ius.io/).

## Первоначальная настройка и создание репозитория.

Перед началом работы с Git нужно указать свое имя и email, которые в дальнейшем будут записываться в историю изменений при каждом коммите. В будущем это позволит понять, кто именно внес те или иные изменения.

Откроем любое из приложений — Git Bash или Git CMD. Первое — это командная строка в стиле Linux, второе — командная строка в стиле Windows. Выбирайте то, что вам ближе. Мы выберем Git Bash и выполним две команды:

git config --global user.email "git-user@selectel.ru"

git config --global user.name "Selectel Git User"

Теперь Git полностью готов к работе. Давайте создадим репозиторий и зафиксируем в нем первое изменение (сделаем коммит). Для начала создадим каталог для будущего репозитория и сразу перейдем в него:

mkdir first-repo && cd first-repo

Создаем новый репозиторий в этом каталоге:

git init

Увидим ответ:

Initialized empty Git repository in C:/Users/git\_user/first-repo/.git/.

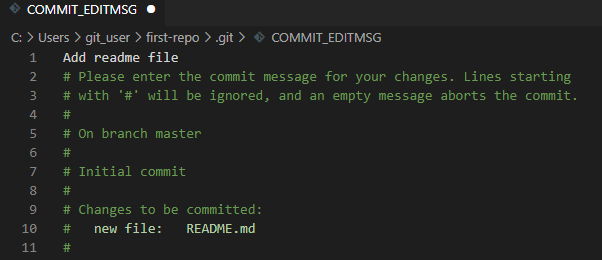
Это означает, что в директории создан новый репозиторий. Далее создадим текстовый файл, назовем его README.md, и напишем в нем любой текст. Но сам по себе этот файл не попадет в следующий коммит. Мы должны проиндексировать изменения, то есть явно сказать Git-у, что этот файл нужно учитывать в следующем коммите:

git add README.md

Далее введем команду:

git commit

Откроется текстовый редактор, который мы выбирали на этапе установки Git. Тут нам нужно ввести комментарий для коммита, то есть кратко описать изменение, которое мы сделали. Мы напишем такой комментарий:



Обратите внимание, что Git автоматически добавил небольшую подсказку в это окно. При этом она не войдет в коммит, потому что в начале строки стоит символ решетки, и Git проигнорирует ее. Но она может быть полезна для дополнительной проверки: мы видим название текущей ветки и список файлов, которые войдут в коммит.

Сохраним файл и закроем редактор. Увидим примерно следующее сообщение:

[master (root-commit) 2b8f7a5] Add readme file

1 file changed, 3 insertions(+)

create mode 100644 README.md

Мы успешно сделали первый коммит.

## Работа с Git в визуальном интерфейсе.

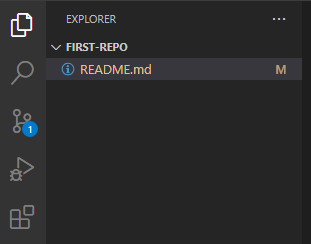
Сам по себе Git — это утилита командной строки. Но не всем может быть удобно запоминать и писать команды в терминале, поэтому часто разработчики пользуются графическим интерфейсом. Есть несколько вариантов:

**Встроенный GUI**. В базовой установке Git есть две простые утилиты: gitk и git gui. Но у них довольно старый интерфейс и пользоваться ими не всегда удобно.

**Отдельные графические утилиты**. Они могут быть понятны и красивы, но неудобны тем, что код нужно писать в одной программе, а для работы с Git нужно переключаться в другую. Примеры таких программ: GitKraken, Sourcetree, GitAtomic. Большой список таких клиентов [есть на официальном сайте Git](https://git-scm.com/downloads/guis).

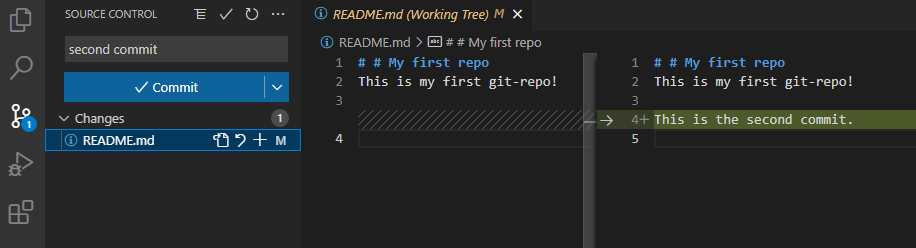
**Встроенные в IDE или текстовый редактор**. В большинстве популярных редакторов кода или IDE уже есть поддержка Git. Как правило, ничего дополнительно настраивать не нужно. Мы рассмотрим именно такой вариант на примере редактора VSCode.

Откроем директорию с репозиторием в редакторе VSCode. Внесите любое изменение в файл README.md и сохраните изменения. Обратите внимание, что в левой части редактора кое-что изменилось:



* Файл README.md подсветился желтым цветом, а рядом с ним появилась буква M (означает Modified — изменен).
* На панели Source Code появилась цифра 1, означающая, что есть одно изменение, которое можно зафиксировать.

Перейдем на панель Source Code. Слева находится список файлов, которые были изменены. Если кликнем на файл, то увидим какие именно изменения мы внесли: в этом случае добавили новую строчку This is the second commit.



Теперь давайте зафиксируем наши изменения. Рядом с названием файла нажмем на «плюс», чтобы проиндексировать его. Это аналогично команде git add, которую мы выполняли ранее. Затем в поле Message внесем комментарий и нажмем кнопку Commit. Это аналогично команде git commit.

Поздравляем, вы сделали уже два коммита в свой репозиторий!

## Заключение.

Итак, мы рассмотрели процесс установки Git под Windows, рассказали об основных параметрах установки и последующей настройки. Увидели, как репозиторий и внести в него первый коммит. Познакомились с работой в командной строке и с помощью графического интерфейса.

# Работа с ветками в Git (git branch).

Инструкция о том, как работать с ветками в Git. Расскажем, как закоммитить изменения и запушить в новую ветку, как удалить ветку или изменить ее — и это не все.

## Введение.

Ветвление стало неотъемлемой частью командной разработки, потому что оно дает возможность работать над разными версиями исходного кода. Основной идеей ветвления является отклонение от основного кода и продолжение работы независимо от него. Также это удобно в тестировании отдельного функционала, потому что позволяет работать над новой частью кода, не беспокоясь о поломке чего-то в рабочей версии. В этой инструкции расскажем о том, как работать с ветками в Git.

**Основные понятия: о ветке Git и master.**

Под веткой принято понимать независимую последовательность коммитов в хронологическом порядке. Однако конкретно в Git реализация ветки выполнена как указатель на последний коммит в рассматриваемой ветке. После создания ветки уже новый указатель ссылается на текущий коммит.

Имя основной ветки Git-проекта по умолчанию — master (однако зачастую бывает main, например, в GitHub), она появляется сразу при инициализации репозитория. Эта ветка ничем не отличается от остальных и также ее можно переименовать, но по договоренности master принято считать главной веткой в проекте.

## Что делает git branch.

Команда git branch — главный инструмент для работы с ветвлением. С ее помощью можно добавлять новые ветки, перечислять и переименовывать существующие и удалять их.

### Способы создания веток и переключения между ними.

Чтобы в Git добавить ветку мы используем:

$ git branch <name of new branch>

После данной операции ветка уже была создана, но вы по-прежнему находитесь в прежней ветке. Если вы планируете переместиться на другую ветку, в том числе только что созданную, необходимо написать **checkout**:

$ git checkout <name of branch>

Для того чтобы определить, где сейчас находится разработчик, Git использует специальный указатель **HEAD**, ссылающийся на текущую локальную ветку. В результате checkout **HEAD** переместится на иную ветку.

### Как с помощью git branch создать ветку и перейти в нее.

Чаще всего при создании новой ветки git пользователю необходимо сразу же переключиться на нее. В таком случае стоит использовать:

$ git checkout branch <name of new branch>

Это будет равносильно:

$ git branch <name of new branch>

$ git checkout <name of new branch>

И также мы получим тот же результат при использовании git checkout с ключом -b:

$ git checkout -b <name of branch>

Если пользователю нужно получить список определенного множества веток, то тогда можно воспользоваться ключами. Одними из самых распространенных будут:

**-r** — при использовании этого ключа мы получим список удаленных веток,

**-a** — используя этот параметр, в выводе будут удаленные и локальные ветки.

### О команде git checkout.

При выполнении этой команды Git требуется осуществить определенный порядок действий, чтобы переходить на ветку, которую мы указали. Для этого программа выполняет следующий алгоритм:

**Проверка, что указанная нами ветка существует в проекте**  
Этот этап необходим, так как в ином случае программа не сможет переключиться на ветвь, которая не определена. Для большего понимания нужно вспомнить, что такое ветка в git. Учитываем, что фактически задание ветки — это запись коммита, на который она ссылается. Внутри Git наличие конкретной ветки проверяется наличием одноименного файла в конкретной директории.

**Переключение указателя HEAD на новую ветку**  
Необходимо сместить указатель, чтобы Git понимал, где сейчас идет работа.

**Изменение рабочей версии таким образом, чтобы новая ветка ей полностью соответствовала**  
Сама концепция работы ветвления заключается в том, что в разных ветках находятся разные версии кода, над которыми работа ведется отдельно друг от друга. Тогда необходимо изменить рабочую копию. Git берет последний коммит и восстанавливает все изменения.

После завершения всех перечисленных выше действий можно считать, что мы полностью переключились. Также с помощью checkout можно извлечь отдельный файл (или папку) из другой ветки и получить его, предварительно перейдя в ту ветку, куда вы собираетесь перенести файл. Для этого выполняем:

$ git checkout <name of new branch> -- <path to folder>

### Основы ветвления и слияния.

Ветвление позволяет разделять рабочий процесс, оптимизировать тестирование и написание нового кода. Однако после того, как разработчик убедился, что написанный им кусок кода готов и его можно отправить к остальной части итоговой версии, удобно переместить его в основную ветку. Такой подход дает возможность получить к концу разработки проекта целый продукт в одном месте.  
Для этого в Git предусмотрено слияние — перенос изменений с одной ветки на другую. Однако сливаемая ветка (под этим определением мы подразумеваем ветку, у которой берем изменения для «вливания» их в другую ветвь) никак не меняется и остается в прежнем состоянии. Такие преобразования мы получаем, применив **git merge**:

$ git merge <name of merged branch>

Операция может привести к появлению конфликтов при попытке слить ветки. Это вызвано тем, что изменения удаляют или переписывают информацию в существующих файлах. При попытке некорректного слияния Git останавливает выполнение команды, чтобы вы могли разрешить конфликт.

Также стоит упомянуть о существовании ключей, предназначенных специально для работы с конфликтами:

**—abort** — прерывает слияние и возвращает все к началу

**—continue**— продолжает слияние после разрешения конфликта

Решить конфликт можно двумя способами:

Вручную разрешить файловый конфликт. Для этого нужно самим изменить файлы, с которыми возникли проблемы. Мы получим файлы такими, какими и представляли их при попытке слияния.

Выбрать более подходящий файл, а от второго отказаться.

### Управление ветками с помощью git branch.

Эта команда может немного больше, чем просто в git создавать ветки из текущей. Если запустить ее без параметров:

$ git branch

При выполнении этой строки мы получим список существующих веток, где символом \* будет отмечена ветка, где вы сейчас находитесь. Это может выглядеть так:

first\_branch

\* master

second\_branch

С помощью параметра -v можно получить последний сохраненный коммит в каждой ветке.

$ git branch -v

first\_branch 8fa301b Fix math

\* master 225cc2d Merge branch 'first\_branch'

second\_branch c56ee12 Refactor code style

Так же существуют опции **—merged** и —**no-merged**, с помощью которых можно отфильтровать полученную последовательность веток. То есть мы получим список ответвлений, которые уже были слиты, или, наоборот, ветки, которые еще не прошли через слияние с другими. Выведем ветки, которые уже были слиты с текущей:

$ git branch --merged

first\_branch

\* master

### Как закоммитить изменения в новую ветку.

После создания новой ветки, перехода в нее и совершения всех запланированных преобразований, нужно сделать коммит в эту же ветку, чтобы сохранить все изменения. Команды для выполнения этих действий ничем не отличаются от команд для создания коммитов в ветке мастер.

$ git add

$ git commit -m '<information about commit>'

После выполнения последовательности этих команд мы закоммитили изменения в нужной версии программы.

### Как запушить в новую ветку.

Если мы хотим запушить нашу ветку, то для этого нужно написать:

$ git push origin <name of branch>

Теперь ветка запушена. Если до этого мы уже пушили ее, то произойдет отправка новых коммитов.  
В отличии от команды git checkout, при выполнении пуша нет проверки на существование указанной ветки. Это будет значить, что при написании несуществующей ветки git создаст ее автоматически.

### Как переименовать ветку.

В процессе разработки могут возникнуть ситуации, когда человек хочет по-другому называть уже созданную ветку. Это может быть связано с разными причинами (например, разрабатываемый в данной версии функционал не соответствует названию). Чтобы переименовать ветку применяем:

$ git branch -m <new name of branch>

Однако здесь нужно быть аккуратными, чтобы не перегрузить проект ненужными ветками. Если запушить переименованную ветку, то на сервере появится ветка с новым именем, но и ветка со старым названием тоже останется. Чтобы избежать такой проблемы, необходимо удалить ветку локально и на сервере.

### Как удалить ветку.

Удаление веток не такой простой процесс, как может показаться. Можно случайно удалить несохраненные изменения в исходном коде, что приведет к нежелательным последствиям. Поэтому здесь нужно действовать осторожно. С операцией удаления над ветками справляется уже привычная команда git branch с параметром -d:

$ git branch -d <name of branch>

Для корректного удаления нужно помнить несколько правил, чтобы не получить ошибки:

**Нельзя удалить ветку, в которой вы находитесь.**Git выкинет ошибку и не произведет удаление. Следовательно, нужно перейти на другую ветку.

**Git не позволит удалить ветку, у которой есть несохраненные изменения.**Так мы избегаем ситуации, когда часть написанного кода будет безвозвратно утеряна. Если же мы уверены, что изменения в этой версии не нужны и их можно смело удалять, то вместо флага **-d** используем **-D**:

$ git branch -D <name of branch>

Соблюдая все условия, нам удастся удалить указанную ветвь.

### Работа с ветками на практике.

В инструментах для разработки на языках часто есть встроенный функционал, позволяющий работать напрямую с Git. Например, в таких средах разработки как IntelliJ IDEA, PyCharm, PhpStorm, CLine, Rider очень удобно и понятно, как правильно оперировать с разными ветками. Для примера разберем работу в одной из таких сред.

**Как работать с ветками в PhpStorm**

интерфейс для работы с ветками

Справа в нижнем углу расположены вкладки для работы с Git, где и происходит вся настройка ветвления. На этой панели расположено название текущей ветки. При желании создать новую нужно нажать на этот пункт и выбрать **New Branch**. Для смены ветки — выбрать из списка искомую и кликнуть на **Checkout**. Для удаления и переименования предусмотрены кнопки **Delete** и **Rename** соответственно.

Работа в специальном приложении почти ничем не отличается от работы в консоли, поэтому все полученные знания можно применять независимо от выбранного способа.

## Получение информации о состоянии веток.

### Как просмотреть состояния файлов ветки.

Отметим, что при переходе на другую версию, незакоммиченные изменения перенесутся на ветку, куда мы перейдем. Поэтому перед переключением необходимо убедиться, что изменения в текущей ветки уже закоммичены. Для этого подходит git status:

$ git status

Выполнение этой операции позволит просмотреть файлы, расположенные в ветке, где мы находимся. Как раз с помощью нее можно отслеживать незакоммиченные изменения, чтобы случайно не перенести их в другое место. Пустой вывод этой команды показывает то, что в ветке не присутствуют измененные файлы и мы можем без опасений продолжать с ней работу. А иначе необходимо закоммитить все нужные исправления.

### Как просмотреть истории коммитов ветки.

Неоднократно в процессе разработки нужно посмотреть на журнал изменений: для отслеживания развития проекта или для определения коммита, к которому следует вернуться. В таких ситуациях выручает команда **git log**:

$ git log <keys> --<path>

У данной команды есть множество ключей, используя которые можно получить более конкретную информацию:

**-<number>** (равноценно **-n=<number>**) — показывает последние n коммитов,

**-pretty=<value>** (доступные такие значения, как **oneline**, **short**, **medium**, **full** и другие) \*\*\*\*— форматированный вывод истории,

**-p** — выводятся изменения, содержащиеся в коммите,

**-graph** — представляет дерево взаимосвязей коммитов в виде ASCII-графа — такой метод использования позволяет получить графическое представление ветвей прямо в консоли,

**-all** — на выходе мы получаем историю всех коммитов для всех существующих веток,

**-decorate** — показывает, на что ссылаются указатели.

Если нам нужно посмотреть историю для конкретной ветви, то поможет выполнение:

$ git log <keys> <name of parent's branch>..<name of branch>

Структура веток в Git представлена в виде графа. Когда мы получаем коммиты определенной ветки, передвигаясь «вверх» по графу, мы должны остановиться в тот момент, когда дойдем до коммита, который будет меньше указателя родителя ветки. При выполнении этого условия когда ветка, чья история коммитов нас интересует, добирается до своего родителя, вывод прекращается, и мы получаем корректный ответ.

### Как просмотреть различия между коммитами.

Достаточно часто в ходе разработки какого-либо продукта у разработчика может возникнуть потребность посмотреть разницу между двумя коммитами, прежде чем заливать что-то. Для этого существует **git diff**:

$ git diff <keys> <path to file> <path to file>

Для этой операции также предусмотрены несколько ключей:

**-diff-filter=<mark>** — с помощью этого параметра, изменяя значения меток, можно задать, обновления между какими файлами мы хотим увидеть. Рассмотрим некоторые возможные значения меток:

**-D** — покажет удаленные файлы,

**-M** — мы получим файлы, модифицированные после последнего коммита.

**-word-diff=color** — повышает читабельность полученной информации: слова подсвечиваются зеленым цветом, если они были добавлены, и красным — если были удалены.

## Заключение.

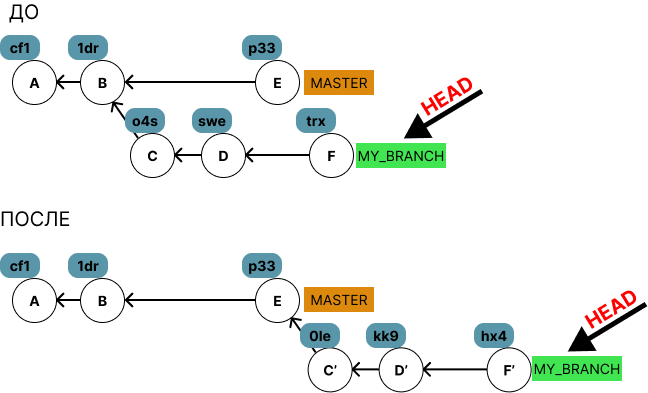
Git обладает множеством преимуществ по сравнению с другими системами контроля версий как раз из-за легковесной работы с ветвлением. Такая гибкость помогает максимально оптимизировать процесс разработки. А само ветвление сильно упрощает разработку проекта. Ветки обеспечивают безопасный совместный доступ к коду для разных людей. Ведь именно они дают возможность пластично и изящно работать над созданием нового продукта.

# Git rebase — перебазирование коммитов и веток.

Знакомим с git rebase: рассказываем о преимуществах команды и показываем, в каких случаях ее стоит использовать и почему.

## Введение.

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.



Пример работы git rebase

В этой инструкции мы познакомим вас поближе с командой git rebase, расскажем о преимуществах и тонкостях работы с ней, покажем, в каких случаях ее стоит использовать и почему.

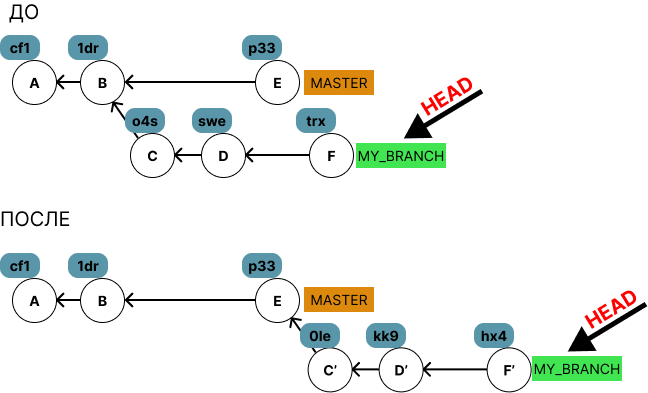
## Git rebase — что это.

Из [документации](https://git-scm.com/docs/git-rebase) — 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.

## Как работает git rebase.



Пример работы перебазирования — git rebase master

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**.

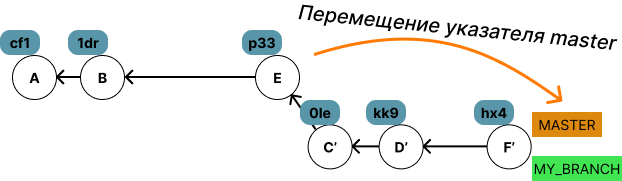
### Линейная история — реинтеграция тематической ветки после выполнения git rebase master.

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.



Слияние перебазированной ветки

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.



Линейная история мастер-ветки

### 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)

Поставлено две задачи:

Поменять местами коммиты qk01ru3 и 4kq5jn2;

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.

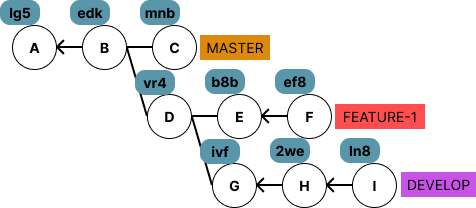
## Git rebase --onto

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.



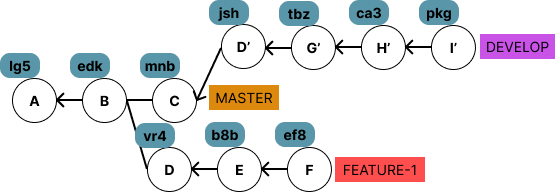
Ветки master, feature-1, develop до rebase

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:

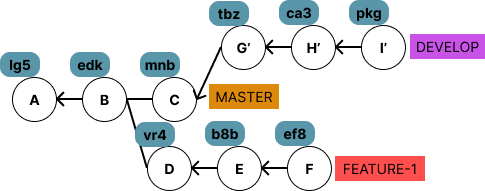


Ветки master, feature-1, develop после стандартного rebase

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.



Ветки master, feature-1, develop после rebase с опцией —onto

## 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” |

## Заключение.

В этой инструкции мы рассмотрели, как сделать rebase ветки в git, узнали про возможность более продвинутой манипуляции с коммитами в режиме interactive и onto, а также тонкости, которые следует соблюдать при работе с rebase.

# Что такое git push и как его использовать.

В инструкции рассказываем о наиболее частых сценариях использования git push.

## Введение.

Команда Git push позволяет отправлять локальную ветку на удаленный репозиторий. Она помогает разработчикам синхронизироваться в команде, а именно отправляет проделанные изменения. Если программист работает один, то пуш позволяет хранить код в облаке, например github, gitlab и не только, избавляя от риска потери данных на компьютере.

Дополнительно для синхронизации еще используют git pull для получения изменений с сервера и git remote, чтобы получить список удаленных подключений к репозиторию.

В этой инструкции мы расскажем, как запушить в удаленный git репозиторий. В статье под «пушем» будем подразумевать git push.

## Отправка изменений в чистый репозиторий.

Перед пушем надо связать локальный и удаленный репозитории. Делается это при помощи команды:

git remote add <repository\_name> link

Вместо **repository\_name** нужно дать имя удаленному репозиторию. Далее в инструкции вместо этого параметра мы будем использовать origin, так как чаще всего используют это имя.

Вместо **link**— ссылка на удаленный репозиторий, она может выглядеть по-разному в зависимости от того используется ssh или https.

Для ssh, который обязателен для [github](https://docs.github.com/en/github-ae@latest/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent)) и [gitlab](https://docs.gitlab.com/ee/user/ssh.html), потребуются сделать дополнительные манипуляции для создания ssh-ключа. Соответствующие инструкции есть на этих ресурсах.

## Отправка изменений.

Перед пушем надо зафиксировать текущие изменения, то есть сделать git commit.

Далее для отправки в терминале пишем:

git push origin <branch>

Вместо **branch** — имя ветки, которую надо отправить. Чаще всего используется master или main:

git push origin master

Такое каждый раз писать слишком громоздко, для этого придумали git push по умолчанию. Для этого единожды набираем предыдущую команду с флагом -u:

git push -u origin master

После этого можно писать более коротко, так как git запомнил, что пушить надо на сервер origin ветку под именем master:

git push

Таким образом, git позволяет запушить ветку в удаленный репозиторий. Чтобы через git добавить ветку в удаленный репозиторий, надо запушить существующую локальную ветку.

## Дополнительные опции публикации.

### Отправка ветки на сервер в ветку с другим именем.

Для того чтобы сделать git push в другую ветку, есть специальный синтаксис, где после имени ветки через двоеточие пишется имя удаленной ветки:

git push origin branch:server\_branch

где **branch** – имя локальной ветки, **server\_branch** – имя удаленной ветки на сервере.

### Отправка всех веток на сервер.

Вместо имени ветки пишем  флаг --all:

git push origin --all

После этого все зафиксированные изменения в ветках отправятся в удаленный репозиторий.

### Отправка текущей ветки.

Удобный способ отправить текущую ветку с тем же именем на сервере.

git push origin HEAD

HEAD указывает на текущую ветку (current branch). Тем самым, не надо запоминать имя ветки, на которой вы находитесь.

## Принудительная публикация (git push --force …)

При отправке может произойти ошибка публикации:

To github.com:example/test.git

! [rejected] master -> master (fetch first)

error: не удалось отправить некоторые ссылки в «github.com:example/test.git»

Это так называемый git push rejected, он означает что пуш был отклонен. Правильнее всего — сделать то, что написано в подсказке к ошибке. Надо получить и смержить изменения, затем снова отправить.

Эта ошибка происходит, так как git проверяет, что новый коммит основан на предыдущих коммитах. Пока вы вносили изменения, кто-то мог запушить изменения того же, над чем вы работали. Поэтому git не может выполнить автоматическое слияние, ваш коммит был раньше и он не базируется на обновленных коммитах в удаленном репозиториие.

Флагом **--force** или сокращенной его версией **-f**отключается проверка коммитов и при необходимости выполняется перезапись истории.

git push --force

Нужно быть аккуратными с этой командой, так как она стирает работу других людей. Эта команда оправдана лишь изредка, например, если вы почти сразу внесли изменения коммита с помощью **git commit --amend** и запушили до того, как кто-то сделал **git pull**.

## Принудительная публикация с параметром (git push --force-with-lease …)

Это более безопасный, но так же нерекомендуемый вариант вариант принудительного пушинга. Он не перезапишет работу в удаленной ветке, если в нее были добавлены коммиты от других людей.

git push --force-with-lease

Принудительная публикация с этим параметром чревата появлением git push rejected у других людей.

## Незапушенные коммиты.

Самый простой способ узнать про них при помощи команды:

git status

Вывод будет содержать имя текущей ветки и то, насколько она опережает версию сервера. Пример вывода:

On branch master

Your branch is ahead of ‘origin/master’ by 1 commit.

(use “git push” to publish your local commits)

Для более подробной информации можно использовать:

git log

Будет выведена история коммитов:

commit 0fcd9558b013f642a8c3b4a59a16a66de39c99bd (HEAD -> master)

Author: Pavel <mail@examle.com>

Date: Sun Mar 27 18:57:14 2022 +0300

Local commit

commit 289c650767d2c7c2e58486e27b8b3933c6442078 (origin/master, origin/HEAD)

Author: Pavel <mail@example.com>

Date: Fri Mar 25 19:41:47 2022 +0300

Pushed commit

В скобках пишется где и какой коммит расположен.

HEAD -> master означает что текущая ветка (current branch) — это master и это последний локальный коммит в ней.

В нижнем коммите в скобках **origin/master** означает, что это последний опубликованный коммит на сервере в ветке master, а origin/HEAD, что коммит расположен на ветке, которая совпадает с локальной веткой HEAD.

## Как пушить теги.

Для создания тегов используют git tag, а для их отправки:

git push origin <tag\_name>

Вместо ***tag\_name*** — имя тега, который надо удалить на сервере.

Также можно сделать отправку всех тегов:

git push --tags

Мы не рекомендуем выбирать этот способ, так как могут отправиться теги, которые были удалены на сервере.

## Удаление ветки или тега на сервере.

Чтобы удалить ветку на удаленном репозитории, используем уже привычную команду с флагом **--delete**:

git push --delete origin <remote\_branch\_or\_tag\_name>

**remote\_branch\_or\_tag\_name** — имя ветки или тега на сервере.

Для удаление локальной ветки:

git branch -d <branch-name>

А для удаления локального тега:

git tag -d <tag\_name>

## Продвинутые возможности.

### Удаление локальных данных (prune)

Если на сервере была удалена ветка, то локально она все еще существует. Чтобы удалить все локальные ветки, которых нет на сервере:

git remote prune origin

### Проверить, удастся ли пушинг (dry run option)

С этой опцией пушинг фактически не произойдет, но покажет вывод словно он произошел.

git push --dry-run

Так можно получить ошибку, которая могла бы произойти, и поправить ее заранее.

### Атомарный пушинг (atomic option)

Если отправить несколько веток на сервер, некоторые могут быть приняты, а другие — нет. Иногда это не то поведение, которое ожидалось. Например, пушинг делает некоторая проверяющая система и ожидаемое поведение такое: либо все будут приняты, либо все будут отклонены. В таких случаях надо делать пушинг атомарно:

git push --atomic origin branch1 branch2 …

## Заключение.

В этой инструкции мы рассмотрели наиболее частые сценарии использования git push. В этой команде много опций, некоторые из них даже редко встречаются. Со всеми ними можно ознакомиться в документации:

git push -help

# Руководство по git stash

Инструкция о том, как откладывать изменения, сделанные в рабочей копии, чтобы можно было применить их позже.

## Введение.

В процессе написания кода может возникнуть ситуация, когда нужно срочно переключиться на другую ветку (использовать git checkout) — или, например, внести правки, которые относятся к другой задаче. Однако новые изменения еще не готовы к тому, чтобы их коммитить и добавлять в репозиторий, — при этом терять их нельзя. В таком случае, как и во многих других, полезной окажется команда git stash.

## Откладывание кода.

### Git stash

git stash

**Git stash** перемещает текущие изменения (так называемые local changes) в локальную директорию, которая выполняет роль специального хранилища, то есть скрывает эти изменения, сохраняя их отдельно, с опцией вернуть позже, когда это понадобится. Таким образом, файлы рабочей копии возвращаются к своему исходному состоянию. Внесенные изменения помещаются в стек, после чего их можно легко оттуда извлечь. Важно отметить, что в рабочей копии исчезнут все измененные файлы, — независимо от того, добавлены они в индекс или нет.

После выполнения этой строки все несохраненные изменения будут сохранены, но не закоммичены. Эти отложенные участки будут сохранены в локальном репозитории и при выполнении команды **git push** не будут переданы на сервер. Такие изменения иногда называют «прятанья».

### Git stash save

Команда **git stash save** исполняет ту же функцию, но имеет другие опции.

Git stash save

Когда необходимо добавить комментарий, мы прописываем его вместе с основной командой:

git stash save "<message>"

Важно отметить, что **git stash** не спрячет изменения, не добавленные в репозиторий. Если необходимо сохранить и их, нужно использовать ту же команду с флагом:

git stash save --include-untracked

Или просто:

git stash save -u

### Git stash push

Сейчас команда **git stash save**считается неактуальной, ее более полезная замена — **git stash push**. Вызвано это прежде всего тем, что, выполняя **git stash push**, можно сохранять определенные спецификации пути. Это, к сожалению, не поддерживается **git stash save**. **Git stash save** все еще работает, но в будущем следует ожидать постепенного перехода на **git stash push** из-за доступности нового функционала.

## Применение отложенных изменений.

Теперь мы знаем, как «спрятать» ненужные в текущий момент изменения, но не знаем, как их извлечь и вернуть в рабочую версию проекта (или, еще можно сказать, сделать «git unstash»). Для решения такой проблемы понадобится **git stash pop:**

git stash pop

После выполнения в хранилище очищаются изменения и возвращаются к рабочей копии. Если же необходимо, чтобы изменения остались (то есть не удалились из хранилища), но при этом вернулись в проект, мы применим **git stash apply**:

git stash apply

Этим полезно пользоваться, когда одни и те же части кода нужны для корректной работы нескольких веток — или в том случае, если необходимо удалить ветку, в которой изначально были сделаны эти изменения. Тогда можно автоматически перенести желаемые участки кода.

git branch --delete <name of branch>

## Необычное откладывание.

Еще одну вариацию использования **git stash**составляет флаг **—keep-index**:

git stash --keep-index

Git не просто спрячет изменения, уже добавленные в индекс, но и оставит эти «прятанья» в индексе.

## Откладывание неотслеживаемых или игнорируемых файлов.

При работе с git stash нельзя забывать, что простое выполнение этой команды не дает скрывать изменения в некоторых типах файлов, то есть не убирает в хранилище появившиеся правки кода. К таким типам относятся **неотслеживаемые** и **игнорируемые**.

Все «прятанья», создаваемые git stash, делятся на две категории: **индексированные** и **неиндексированные**. Индексированными называются исправления, которые уже были добавлены в раздел проиндексированных файлов, а неиндексированными — исправления, которые отслеживаются сейчас.

**Проиндексированными**называются файлы, о чьих преобразованиях уже знает Git, но само изменение еще не записано в репозиторий.

Поэтому логично, что новые файлы не смогут быть отложены, если они все еще не проиндексированы. Точно так же, как и игнорируемые (помеченные явным образом как файлы, которые необходимо игнорировать).

При необходимости спрятать неотслеживаемые файлы поможет уже известный параметр -u (или же -include-untracked):

git stash -u

Перенос правок для игнорируемых файлов осуществляется с помощью флага -all:

git stash -all

Или короче:

git stash –a

## Управление несколькими наборами отложенных изменений.

При повторном использовании git stash мы можем понять, что фактически мы создаем коммит, но сохраняем его по отдельности. Используя следующее выражение, мы можем посмотреть на список всех спрятанных изменений:

git stash list

Полученный список будет отсортирован по времени, когда были отложены изменения: внизу списка будут самые старые изменения. Также каждое изменение имеет свой идентификатор с номером, например:

stash@{0}: WIP on master: 08f13aa add processing file

stash@{1}: WIP on master: c33b610 refactor code

Описание «прятаний» по умолчанию начинается с WIP (что указывает на незавершенную работу) перед названием ветки (в частности коммита), где их отложили.

После выполнения git stash pop будет возвращен последний из наборов откладываний (с индексом 0). Если нужно выбрать конкретный набор из всех ранее отложенных участков, то в фигурных скобках нужно явно прописать его идентификатор. Например:

git stash pop stash@{1}

## Просмотр различий между наборами отложенных изменений.

Команда **git stash show**выводит информацию по последнему сделанному изменению:

git stash show

Если нужно посмотреть информацию о другом изменении, то необходимо явно указать индекс:

git stash show stash@{1}

Очень полезной функцией может стать сравнение полученных наборов друг с другом. Для такой ситуации понадобится параметр --**patch**:

git stash show --patch

Или что будет равносильно:

git stash show –p

## Частичное откладывание изменений.

Иногда бывает удобно разделить существующие в коде незакоммиченные изменения на несколько откладываний. Тогда в разных откладываниях могут храниться изменения в разных файлах или даже в одном. Тогда необходимо написать флаг -p (или --**patch**):

git stash -p

Теперь откладывание для каждого из участков будет выполняться отдельно, при этом запрашивая подтверждение:

-? - узнать все возможные варианты для работы (ниже будут перечислены самые распространенные);

-y - отложить этот участок;

-/ - использовать для поиска регулярное выражение;

-n - не откладывать этот участок кода;

-q - отложить все выбранные участки и выйти;

-s - разделить текущее изменение на несколько откладываний.

Прервать работу откладывания поможет комбинация клавиш CTRL+C.

## Создание ветки из отложенных изменений.

При работе с разными ветками и откладыванием изменений (или другими операциями) могут возникнуть конфликты. Одной из возможностью избежать этого и ничего не сломать будет создание новой ветки и применение изменений в ней:

git stash branch <name of new branch>

Теперь будет создана новая ветка, к которой уже применятся сделанные изменения. Однако последние правки будут удалены из списка всех спрятанных изменений. Как и в предыдущих случаях, если нужно работать не с последним набором, то можно указать индекс.

## Удаление отложенных изменений.

Теперь нам нужно научиться удалять отложенные изменения:

git stash drop

Так мы удалим последний набор. Аналогично другим командам для удаления определенного набора указываем индекс:

git stash drop stash@{1}

Иногда нужно очищать хранилище от всех наборов, то есть отменять все созданные изменения. В таких случаях мы пишем:

git stash clear

## Разрешение конфликтов.

При работе с возвращением в исходный код спрятанных изменений могут возникнуть конфликты. Если это произошло, то после выполнения **git stash pop**появится сообщение о возникновении конфликта.

Чтобы посмотреть файлы, в которых возникли конфликты, нужно воспользоваться **git status**. Они будут находиться в секции **Unmerged paths**. Также все новые изменения отслеживаемых и проиндексированных файлов будут находиться в разделе **Changes to be committed**.

Разрешить такой конфликт необходимо вручную. Однако в отличие от конфликтов, например, при слиянии веток (выполнение git merge), нет потребности сразу коммитить изменения, так как код еще не закончен и не готов для полноценного коммита. А после этого можно добавить его в индекс и сделать привычный коммит.

Можно сделать вывод, что конфликты при работе **git stash pop** допустимо рассматривать в качестве незафиксированных изменений.

## Принцип работы команды git stash

Разберемся, как устроена работа команды и получим более подробное **git stash**-описание.

Мы уже знаем, что наборы откладываний на самом деле являются коммитами (из этого следует, что команда **git log** корректно отработает, выводя изменения). Ссылка в **.git/refs/stash** ведет на последнюю группу откладываний. Чтобы найти более ранние наборы, существует журнал ссылок, который так и называется — **stash**. Поэтому если нужно просмотреть «прятанья», мы пишем **stash@{n},** то есть обращаемся к записи с номером n.

В зависимости от откладываемых элементов git stash создает от двух до трех коммитов. Конкретно рассмотрим, какие это будут коммиты.

* В коммите **stash@{0}** будут содержаться файлы в исходном коде на момент запуска команды.
* Первым предком**stash@{0}** будет являться уже существующий коммит в той ветке, куда ведет HEAD.
* Второй предок **stash@{0}**— новый коммит, который является индексом.
* Третий коммит будет создан только в том случае, если в проекте содержатся неотслеживаемые Git-файлы или были указаны параметры -u или -a. Это будет коммит с файлами, которые содержатся в исходном коде и не видны Git’у.

Теперь поподробнее разберем, как **git stash**зашифровывает рабочий каталог и работает с проиндексированными файлами.

* Выше мы разбирали, что каталог может содержать исправления в разных типах файлов. Такие файлы могут быть только частично отмечены в репозитории.
* **Git stash** кодирует преобразования в отслеживаемых участках под видом ориентированного ациклического графа (это можно проверить, если нарисовать дерево, описанное выше). Целью одного коммита будет хранение неиндексированных, а второго — уже индексированных изменений, которые были отмечены как проиндексированные файлы.
* Помним, что флаг **-u** зашифровывает все правки неотслеживаемых файлов в виде другого коммита, в то время как параметр **-a**, наоборот, включает изменения в игнорируемых файлах в тот же (не создавая при этом дополнительный коммит).

Осталось разобраться, что происходит, когда мы реализуем git stash pop. Спрятанные части кода из всех перечисленных выше пунктов возвращаются сразу и в рабочую копию, и, что важно, также к разделу проиндексированных файлов Git. После этого коммит, который мы извлекли, будет удален из stash. Остальные ссылки в журнале просто сдвигаются (чтобы обеспечить далее корректную работу). Коммиты, которые мы извлекли, не очищаются сразу, но отмечаются как мусор и впоследствии удаляются.

## Заключение.

Команда **git stash**открывает новые возможности перед разработчиками в написании кода. С помощью откладываемых изменений можно удобно и оперативно работать в нескольких ветках, не рискуя потерять важные участки кода. Если использовать специальные команды и параметры к ним, можно эффективно управлять сделанными отложенными изменениями. Все рассматриваемые операции применимы также и к таким типам файлов, как неотслеживаемые и игнорируемые, что позволяет комфортно работать со всем разрабатываемым проектом в целом.

Конечно, даже такой обширный функционал не может полностью обезопасить от конфликтов или ошибок. Но если научиться умело пользоваться git stash, можно избежать некоторых сложностей в процессе разработки мобильных приложений, веб-приложений и других продуктов.

# Команда git remote add origin для работы с удаленными репозиториями.

Как создать удаленный репозиторий, подключиться к нему и какие команды Git использовать.

## Введение.

В этой инструкции мы научим вас создавать удаленный репозиторий, подключаться к нему различными методами и покажем, какие команды Git использовать.

## Немного теории.

Совместной разработка в Git предполагает, что у разработчиков, работающих над каким-то проектом, есть своя копия репозитория, соответственно и своя локальная история в зависимости от модели ветвления. Так как разработчикам обычно нужно поделиться серией коммитов, а не каким-то одним, в Git есть возможность делиться целыми ветками между репозиториями.

Удаленный репозиторий – это репозиторий, размещенный в локальной или интернет сети. Удаленный репозиторий используется для того, чтобы делиться и обмениваться кодом между разработчиками в рамках сети. Его также можно использовать, если вы разрабатываете проект на нескольких устройствах.

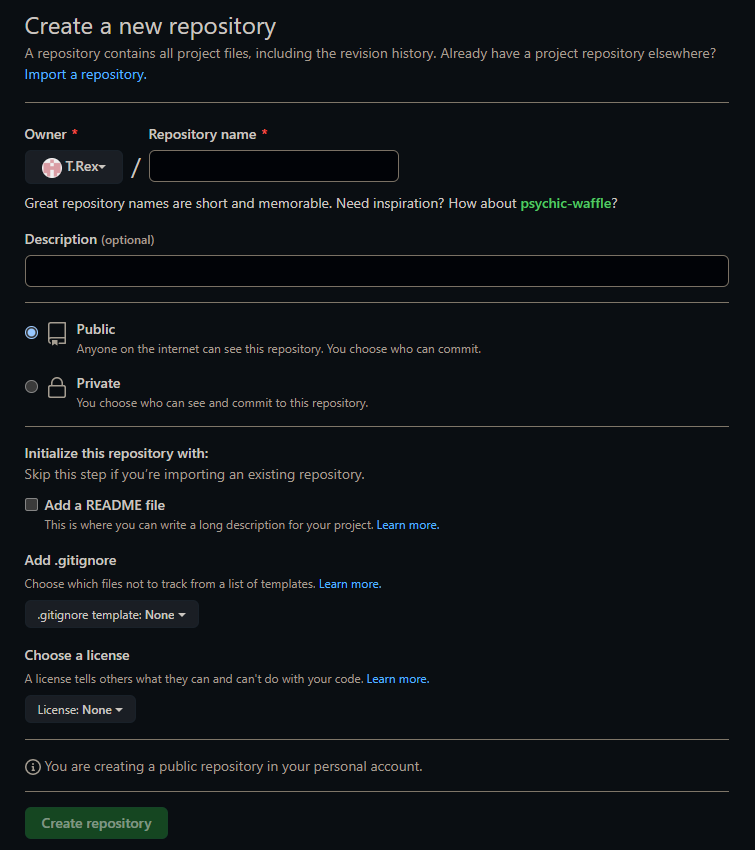
Ветки слежения – это ссылки на определенное положение веток удаленного репозитория. Перемещение веток слежения происходит автоматически при выполнении команд, которые будут показаны дальше.

Имена веток слежения имеют вид **<remote>/<branch>**. Если один из разработчиков произвел отправку на удаленный сервер ветки develop, когда у второго разработчика есть своя локальная ветка develop, то удаленная ветка будет называться **origin/develop**и соответственно быть веткой слежения. При разработке orgin по умолчанию называется удаленный сервер.

## Создание удаленного репозитория.

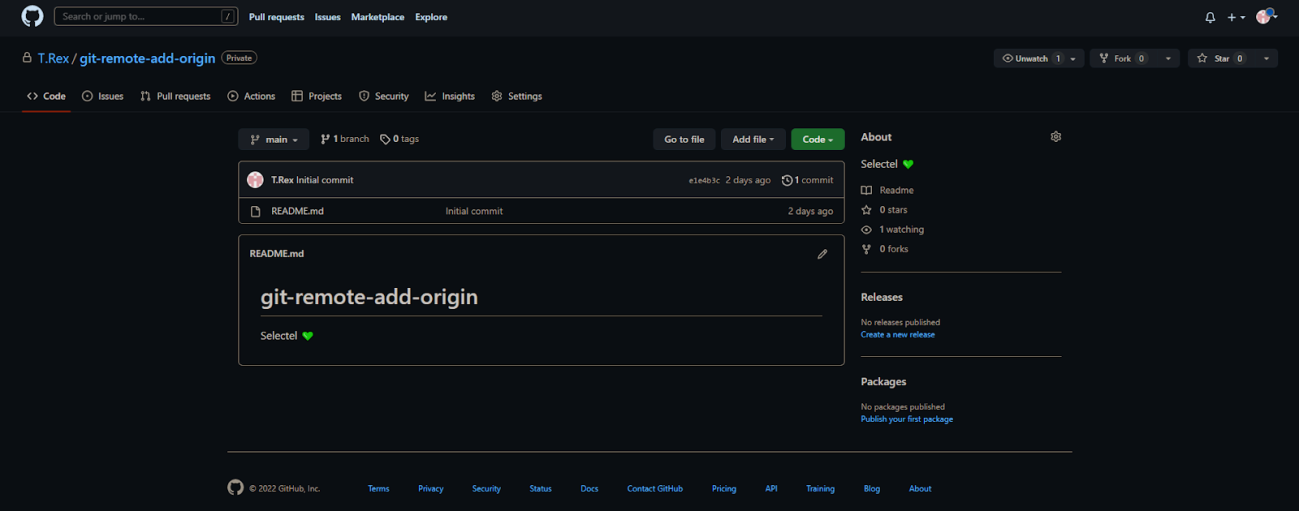
Создадим новый репозиторий на GitHub. Для этого на сайте GitHub кликните на значок плюса справа сверху или перейдите по [ссылке](https://github.com/new).

Перед нами откроется следующая форма:



* Owner (владелец) – выбрать персональный аккаунт, который будет иметь полный доступ к данному репозиторию.
* Repository name (название репозитория) – при названии репозитория принято использовать PascalCase (составные слова с заглавной буквы) и kebab-case (составные слова через дефис).
* Description (описание) – описание проекта.
* Public – кто угодно может просматривать репозиторий, но коммитить смогут только те аккаунты, которым вы предоставите доступ.
* Private – никто, кроме вас, не сможет коммитить или просматривать данный репозиторий без права доступа.
* Add a README file – создает readme.md файл, который нужен для добавления длинного описания.

После нажатия **Create repository**нас перекинет на страницу нашего нового репозитория.



Пример созданного репозитория на GitHub

Сейчас в созданном github repo нет никаких файлов. Чтобы их выгрузить из локального репозитория, нужно подключиться к удаленному.

## Настройка подключения к удаленному репозиторию.

### Добавление удаленного репозитория. Команда git remote add origin

Чтобы добавить удаленный репозиторий и присвоить ему название, которое используется для удобства как сокращение адреса, просто выполните команду

git remote add origin <URL>

По умолчанию удаленный репозиторий называется origin, о чем было сказано ранее, но можно присваивать любое название.

Ссылочные объекты у подключаемого удаленного репозитория будут скопированы.

### Подключение по HTTPS

git remote add origin https://github.com/ИМЯПОЛЬЗОВАТЕЛЯ/РЕПОЗИТОРИЙ.git

С помощью https будет проще работать на нескольких устройствах нежели с перемещением ssh ключей между ними. А порт 443, который используется данным протоколом, обычно открыт. Но для работы с этим способом подключения может понадобиться введения логина и пароля, например, при клонировании приватного репозитория.

### Подключение по SSH

git remote add origin git@github.com:ИМЯПОЛЬЗОВАТЕЛЯ/РЕПОЗИТОРИЙ.git

Чтобы подключить репозиторий по ssh, нужно произвести генерацию пары SSH-ключей:

В Git Bash прописываете одну из двух команд, вставив свою почту от GitHub аккаунта:

ssh-keygen -t ed25519 -C "ПОЧТА\_GitHub\_АККАУНТА"

ssh-keygen -t rsa -b 4096 -C "ПОЧТА\_GitHub\_АККАУНТА"

В директории С:/Users/имяпользователя/.ssh/ появятся файлы приватного и публичного (.pub) ключей. Публичный ключ необходимо прописать в настройках GitHub аккаунта, в разделе **SSH Keys > Add new.**

SSH является более безопасным протоколом, чем HTTPS, но и к тому же позволяет не прописывать каждый раз пароль, вместо этого сравнивается публичный ключ с хранящимся у вас.

Для изменения адреса подключенного удаленного репозитория используется:

git remote set-url <remote\_name> <remote\_url>

Первое значение (**<remote\_name>**) — название удаленного репозитория, второе значение (**<remote\_url>**) — новый url.

## Основы работы с удаленным репозиторием Git

### Просмотр удаленного репозитория.

Для того чтобы просмотреть список названий удаленных репозиториев, которые были добавлены, используйте команду:

git remote

Указав ключ -v, можно просмотреть, какие адреса в добавленных удаленных репозиториях используются для чтения и записи.

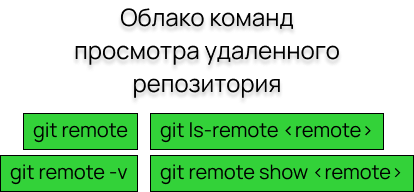
Полный список удаленных ссылок можно получить с помощью команды:

git ls-remote <remote>

Либо для получения удаленных веток и дополнительной информации используйте команду:

git remote show <remote>

**<remote>** — название удаленного репозитория, которое ему дали после подключения. Можно использовать и url интересующего удаленного репозитория.



### Изменение удаленного репозитория.

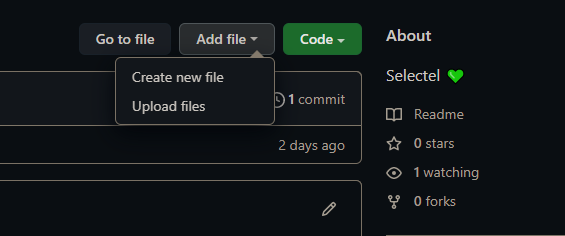
Чтобы отправить изменения (имя ветки) **<branch-name>** на сервер **<remote-name>**, задействуйте команду:

git push <remote-name> <branch-name>

Если ветка, которую вы хотите отправить, не отслеживается, то добавьте флаг **—set-upstream**. На примере ветки develop это будет выглядеть вот так:

git push --set-upstream origin develop

В GitHub же есть кнопка (**Add file**) для загрузки или создания файлов прямо в удаленном репозитории.



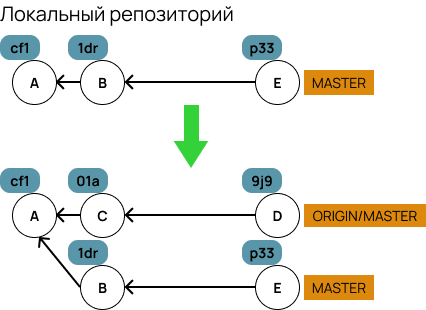
Так же возможно переименование удаленного репозитория с прежнего имени **<old-name>** на новое **<new-name>**.

git remote rename <old-name> <new-name>

### Получение изменений из удаленного репозитория.

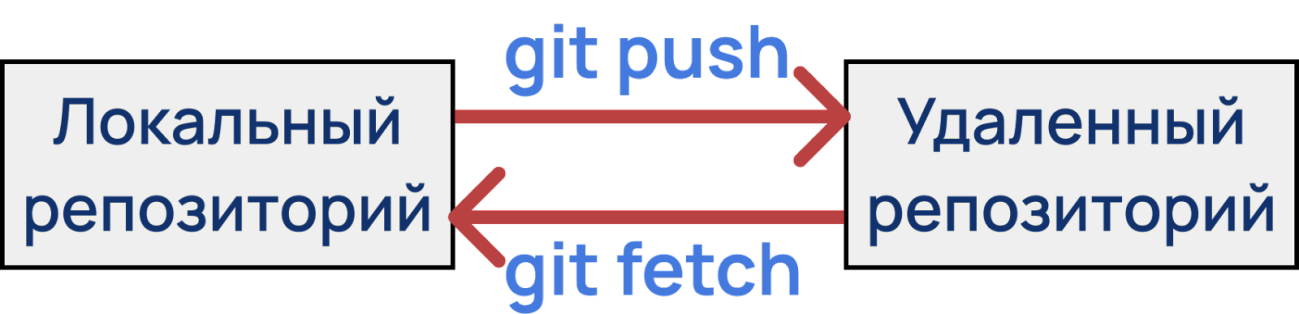
Для связи с указанным удаленным репозиторием используется следующая команда, которая подтянет новые изменения на локальный репозиторий:

git fetch [remote-name]



Локальный репозиторий после выполнения git fetch

После выполнения команды появляются ссылки на все ветки из указанного удаленного репозитория. Данная команда забирает данные в наш локальный репозиторий, но не объединяет или перебазирует их с коммитами в локальном репозитории, связанного с удаленным. Поэтому используйте **merge** или **rebase**, когда закончите работать над изменениями.



Если ветка настроена на отслеживание удаленной ветки, то можно использовать более удобную команду:

git pull

Данная команда заменяет прописывание двух команд:

git fetch; git merge

Есть и универсальнее способ: клонирование репозитория. Клонирование копирует все файлы и коммиты удаленного репозитория, после чего уже можно использовать предыдущую команду **-- git fetch**, если понадобятся новые изменения из удаленного репозитория.

git clone [url]

Клонирование может быть по HTTPS, либо по SSH. Рассмотрим каждый вариант.

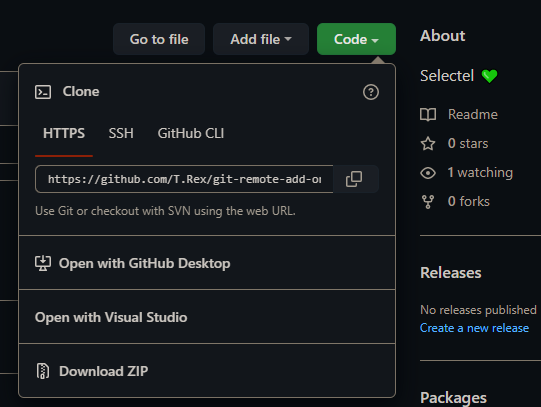
* 1. По **HTTPS**
* Может быть использовано независимо от видимости репозитория.
* Понадобятся имя пользователя и пароль аккаунта GitHub. В качестве пароля используется персональный токен доступа. Сгенерировать персональный токен доступа можно в настройках профиля GitHub (Settings > Developer Settings > Personal access tokens).
* Команда будет иметь вид:

git clone https://github.com/ИМЯПОЛЬЗОВАТЕЛЯ/РЕПОЗИТОРИЙ.git

* 1. По **SSH**
* Доступ к гиту предоставляется по защищенному протоколу.
* Необходимо сгенерировать пару SSH-ключей и добавить публичный ключ в аккаунт GitHub.
* Команда будет иметь вид:

git clone git@github.com:ИМЯПОЛЬЗОВАТЕЛЯ/РЕПОЗИТОРИЙ.git

В GitHub есть зеленая кнопка (Code), кликнув на которую можно будет скопировать url-адрес для выбранного способа клонирования или скачать архивом данный репозиторий.



Для клонирования определенной ветки репозитория, допишите флаг **—single-branch**:

git clone --single-branch [url]

После работы git clone отслеживание удаленный ветки ставится автоматически.

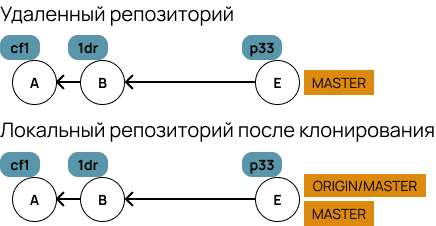
Если же при клонировании вы хотите другое название (не origin) удаленному репозиторию, то выполните:

git clone -o ticgit

Тогда по умолчанию ветка слежения будет иметь вид ticgit/master (tcgit — пример названия удаленного репозитория).

Работа git clone пошагово выглядит так:

1. Создается новая директория.
2. В ней создается репозиторий посредством выполнения git init.
3. Выполняется команда git remote add origin, которая добавляет по URL (назван origin) удаленный репозиторий.
4. Подхватываются новые изменения данного удаленного репозитория через git fetch.
5. С помощью git checkout извлекается последний коммит в рабочую директорию.



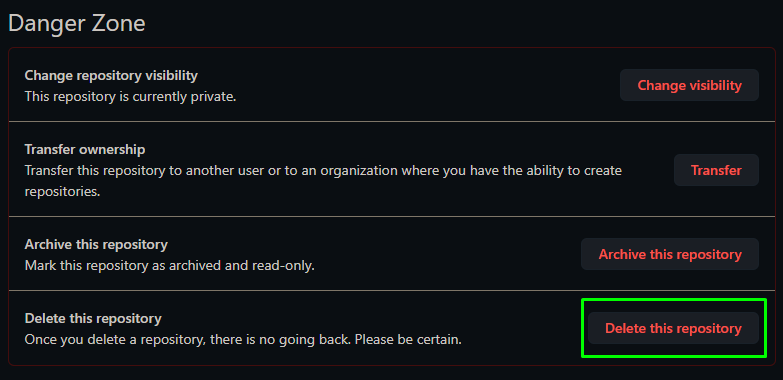
### Удаление удаленного репозитория.

Если интересует убрать подключение к удаленному репозиторию, то используйте команду

git remote rm <name>

Где <name> — название репозитория. Вместе с тем удалятся настройки и отслеживаемые ветки.

Чтобы удалить сам удаленный репозиторий, это нужно сделать на сервере, на котором он размещен. На GitHub перейдите в раздел настроек (“Settings”), проскрольте до самого низа, где будет поле Danger Zone. Нажмите на **Delete this repository**, в открывшемся новом поле введите \***Имя пользователя\*/\*Названия репозитория**\* для подтверждения удаления. В нашем случае это T.Rex/git-remote-add-origin.



Поле с возможностью удаления репозитория на GitHub

## Заключение.

Мы создали удаленный репозиторий, узнали про подключение удаленного репозитория разными способами, рассмотрели команды clone, fetch, push, pull для взаимодействия с ним.

# Команда git fetch — чем отличается от git pull

Для чего нужна команда git fetch, как работает и чем отличается git pull от git fetch.

## Введение.

В этой инструкции мы расскажем, для чего нужна команда git fetch, как работает и чем отличается git pull от git fetch.

## Git fetch, что делает данная команда.

Git fetch — команда в распределенной системе контроля версий Git, которая используется для скачивания изменений из удаленного или локального репозитория в свой локальный репозиторий.

git fetch [<опции>] [<репозиторий> | <группа репозиториев>]

Польза git fetch в том, что эта команда привносит актуальность локальному репозиторию без изменения его дерева или текущей ветки. Таким образом, следить за изменениями, сделанными другими участниками, становится проще, быстрее можно начать работу с этими изменениями.

## Что произойдет при выполнении команды git fetch

Данная команда извлекает все новые объекты из удаленного репозитория и хранит их в локальном репозитории, обновляя ветки, теги и другие метаданные локального репозитория. Таким образом, отражается состояние удаленного репозитория.

Перед дальнейшими шагами после выполнения команды хорошей практикой считается просмотреть изменения, чтобы убедиться, что они не приведут к дальнейшим конфликтам. Вот некоторые команды, которые можно использовать после команды git fetch:

Перейти к списку удаленных веток, которые были обновлены во время операции git fetch:

git branch -r

Сравнить свою локальную ветку с обновленной удаленной веткой и увидеть различия:

git diff

Просмотреть историю коммитов обновленной удаленной ветки и понять, какими были внесенные изменения:

git log

Переключиться на обновленную удаленную ветку и начать работать с внедренными изменениями:

git checkout

Включить изменения из удаленного репозитория в локальную копию проекта путем слияния. Считается самым частым действием после операции git fetch:

git merge

Наложить локальные коммиты поверх обновленной удаленной ветки. Такой вариант должен использоваться с осторожностью, так как в результате может привести к конфликтам, если удаленная ветка сильно изменилась:

git rebase

Важно помнить, что git fetch только обновляет удаленные ссылки и ветки, но не производит слияния с локальной веткой. Для внедрения изменений используется одна из двух последних приведенных в пример команд.

## Разница между git pull и git fetch

Cуществуют четыре команды, которые скачивают код из удаленного репозитория:**git clone, git remote update, git fetch, git pull**. Последние 3 команды очень похожи друг на друга.

**Git clone**, или же клонирование репозитория, полностью копирует запрашиваемый репозиторий, происходит его фиксация и установка ветки по умолчанию.

Между **git remote update** и **git fetch** отличие лишь в том, что первая из этих команд обновляет все удаленные подключения, а **git fetch** — только один репозиторий или группу репозиториев. Но есть опция, нивелирующая разницу между этими командами. Мы рассмотрим ее ниже в параграфе про опции.

**Git pull** — связка последующих команд **git fetch** и **git merge**.

git pull [<опции>] [<репозиторий>]

Такой способ ускоряет внедрение новых изменений в локальный репозиторий, но и является менее управляемым. **Git pull** следует применять, если у вас нет цели посмотреть изменения перед слиянием или вы не хотите медлить перед интеграцией изменений в локальную ветку.

К **git pull** применяются опции как от **git fetch**, так и от **git merge.** Если же вы хотите произвести rebase вместо merge, то используйте соответствующую опцию:

git pull --rebase [<репозиторий>]

## Полезные опции и настройки для команды git fetch

### Настройки.

Все настройки выполняются в**.git/config**, либо командой**git config --edit**.

Чтобы использовать git fetch, разработчик должен настроить подключение к удаленному репозиторию. Это можно сделать командой **git remote**, которая позволяет добавлять, удалять и управлять удаленными репозиториями. Когда удаленный репозиторий будет готов, команда **git fetch** может быть использована для получения последних изменений из удаленного репозитория.

Если необходимо обновить только одну ветку, например, master, не задействуя остальные доступные ветки в удаленном репозитории, то можно внести следующую настройку в конфигурации удаленного подключения к этому репозиторию:

[remote “origin]

fetch = +refs/heads/master:refs/remotes/origin/master

Или просто введя в консоль:

git fetch origin master

После набора **git fetch** с названием репозитория данная настройка будет вызываться по умолчанию. Если же такая настройка необходима однократно, то при наборе команды можно сразу указать соответствие веток. Итак, для получения данных, например, из ветки super-branch удаленного репозитория в локальную origin/super-branch, прописывается такая команда:

git fetch origin master:refs/remotes/origin/super-branch

Команда выполнится для ветки super-branch локального репозитория, получив данные из удаленного репозитория.

Если нужен не один репозиторий, а несколько, то группа репозиториев настраивается через **remotes.<group>**следующим образом:

[remotes]

group1 = first-remote second-remote commit

group2 = only-one-remote

Есть возможность включить автоматическую подписку на теги во время git fetch. То есть Git будет скачивать все новые теги из удаленного репозитория и создавать локальные теги, которые будут указывать на те же коммиты, как и в удаленном репозитории. Можно включить эту опцию глобально, добавив **--global**к команде **git config --edit**:

tags = true

### Опции.

Есть ряд опций, доступных с командой git fetch, которые позволяют разработчикам указать точные изменения. Например, можно использовать опцию **--prune**, чтобы удалить отслеживаемые ветки, которые больше не существуют в удаленном репозитории. Это помогает содержать локальный репозиторий в чистоте и уменьшает беспорядок в списке веток.

git fetch --prune

Другая опция **--all**, которая позволяет разработчикам получить изменения со всех настроенных удаленных репозиториев в локальный репозиторий. Она может быть полезна в ситуациях, когда в проекте есть несколько репозиториев и разработчикам нужно подхватывать изменения из них всех. Получается аналог команды **git remote update**.

git fetch --all

Следующая опция продемонстрирует работу **git fetch** без дальнейшего обновления каких-либо локальных веток. Это может быть полезно для теста и просмотра изменений, которые будут получены в результате выполнения**git fetch**:

git fetch --dry-run

### Пример: git fetch origin

Пусть у нас будет настроен удаленный репозиторий на GitHub (на GitLab будет тоже самое), который мы назвали локально «origin». На нем существуют ветки master и feature-branch.

Для начала перейдем к локальному репозиторию, который планируем обновить:

cd ~/my-best-project

Затем используем git fetch, чтобы получить последние изменения из удаленного репозитория:

git fetch origin

После того как операция git fetch завершится, можно проверить обновленные удаленные ветки:

git branch -r

> origin/master

> origin/feature-branch

Теперь сравним локальную ветку с обновленной удаленной веткой, чтобы увидеть различия:

git diff origin/master

Также можно посмотреть историю коммитов обновленной удаленной ветки:

git log origin/master

Переключимся на обновленную удаленную ветку:

git checkout origin/master

Cольем изменения из обновленной удаленной ветки в нашу локальную ветку:

git merge origin/master

Выбор дальнейших команд после выполнения команды**git fetch** зависят от специфики использования данной команды и структуры репозитория.

## Заключение.

Ветки с удаленным отслеживанием: git fetch работает с ветками с удаленным отслеживанием, которые являются ссылками на состояние удаленных веток в локальном репозитории.

Сетевое взаимодействие: git fetch взаимодействует с удаленным репозиторием по протоколу Git и получает информацию об удаленных ветках и их последнем состоянии. Эта информация хранится в локальном репозитории в виде веток удаленного отслеживания.

Получение изменений: команда git fetch извлекает информацию о последних изменениях, внесенных в удаленный репозиторий, и соответствующим образом обновляет ветки удаленного отслеживания. Сюда входят последние хэши коммитов, имена веток и сообщения коммитов.

Слияние изменений: после того, как git fetch извлечет изменения из удаленного репозитория, можно использовать команду git merge или git rebase, чтобы объединить изменения с локальной веткой.

Опции git fetch: можно использовать различные опции команды git fetch для управления ее поведением. Например, указать другой удаленный репозиторий, ограничить количество веток или даже удалить ветки, которые были удалены из удаленного репозитория.