**СИСТЕМА КОНТРОЛЯ ВЕРСИЙ GIT**

***Система контроля версий*** - программа, которая позволяет хранить разные версии одного и того же документа, легко переключаться между ранними и поздними вариантами, вносить и отслеживать изменения.

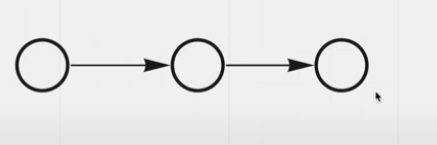
***Github*** – своего рода соцсеть для разработчиков, в которой можно делиться проектами, комментить их, ставить лайки и так далее.

***Репозиторий*** - это рабочая директория с вашим проектом. По сути, это та же папка с HTML, CSS, JavaScript и прочими файлами, что хранится у вас на компьютере, но находится на сервере GitHub. Поэтому вы можете работать с проектом удалённо на любой машине, не переживая, что какие-то из ваших файлов потеряются — все данные будут в репозитории при условии, что вы их туда отправите.

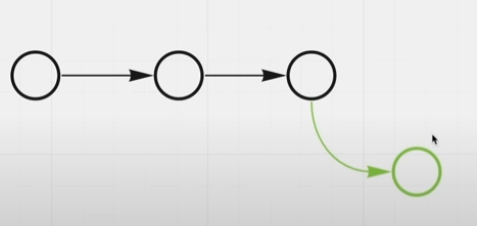
Если над проектом трудится команда разработчиков, как правило, создаётся общий репозиторий, в котором находится рабочая версия проекта (назовём его *мастер-репозиторий*). При этом каждый пользователь клонирует себе в профиль оригинальный репозиторий и работает именно с копией. Такая копия называется ***форком***. Так как **форк** — ваша персональная версия мастер-репозитория, в нём вы можете пробовать разные решения, менять код и не бояться что-то сломать в основной версии проекта.

***Создать коммит (commit)*** значит зафиксировать изменения любых файлов, входящих в репозиторий. По сути, ***коммит – это сохранение, к которому можно откатиться в случае ошибок при дальнейшей работе.***

***Ветка в Github*** – это копия репозитория. Как минимум, репозиторий должен содержать основную ветку, идя по которой разработчик движется в определённом векторе разработки, составляя попутно коммиты. Представим, что сохранения – это кружки. Сохранения основной ветки – чёрные кружки:



Что если разработчик придумал какую-нибудь фичу, которая не входила в планы или, допустим, сильно отличается от задуманного? Для этого он может создать новую ветку, по которой будет вести работу и не мешать разработке в основной ветке. Кстати, новая ветка при создании является копией последнего сохранения (т.е. третьего чёрного кружка):



***Запрос на извлечение (pull request)*** означает, что вы сообщаете другим, что вы передали изменения, внесённые вами в ветке, в главный репозиторий. Соавторы хранилища могут принять или отклонить запрос на извлечение. После его открытия вы можете обсудить и проанализировать свою работу с соавторами.

Допустим, мы дали возможность поработать над какой-либо частью проекта другу. Для этого он создаёт новую ветку и работает в ней, а потом создаёт пулл-реквест, чтобы внедрить эти изменения в основную ветку.

***Статусы файлов***

Статус *Untracked* означает, что Git видит файл, которого не было в предыдущем снимке состояния (коммите);

Статус *Unmodified* означает, что гит видит файлы, которые не были изменены после последнего коммита

Статус *Modified* означает, что гит видит файлы, в которые были внесены изменения с момента последнего коммита

*Stage (или cache)* – это хранилище для файлов с изменениями, информация о которых попадет в единый коммит. Stage является элементом архитектуры трех деревьев, на базе которой построен git, более подробно смотрите здесь. Для добавления файла README.md в stage необходимо воспользоваться командой git add.

**ОСНОВНЫЕ КОМАНДЫ**

***git status*** – отслеживание изменений в репозитории

***git log*** – показ логов репозитория

***git add имя\_файла.формат*** - команда, которая говорит Git, на какие файлы нужно смотреть и записывать их изменения, а все остальные файлы будут игнорироваться (после этой команды можно ввести git status, чтобы посмотреть, какие произошли изменения)

git add \* - добавить все файлы в список отслеживаемых

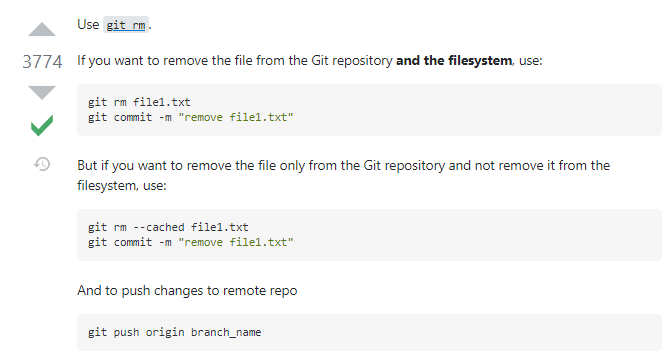
***git commit -m "тут ваше сообщение о коммите"*** – сделать коммит внесённых изменений (подтвердить изменения)

Отмена коммитов: <https://ru.hexlet.io/courses/intro_to_git/lessons/commits-cancelation/theory_unit>

***git push origin main*** – пуш коммита на main ветку (можно просто **git push**)

|  |
| --- |
| ***git fetch*** - собирает все коммиты из целевой ветки, которых нет в текущей ветке, и сохраняет их в локальном репозитории. Это действие (fetch) не влияет на локальные ветки и текущие изменения, просто изменения с удаленного сервера скачиваются в директорию локального репозитария. |
| ***git merge –*** сливает коммиты в локальную копию репозитория |
| ***git pull*** – шорткат команд ***git fetch*** и ***git merge***, который сразу обновляет локальный репозиторий с сервера |

Удаление файла/папки из репозитория:



***git rm –r имя\_файла.расширение\_или\_имя\_папки***

***git clone -b название\_ветки ссылка\_на\_репозиторий/репозиторий.git –*** клонировать определённую ветку удалённого репозитория себе на локалку

Пример: git clone -b Artem [git@gitlab.digital-sector.ru:testing/vitz\_test.git](mailto:git@gitlab.digital-sector.ru:testing/vitz_test.git)

**Работа с ветками**

***git branch*** – показывает список веток

***git branch some\_branch*** – создаёт новую ветку, где «some\_branch» - имя, которое будет присвоено новой ветке, но пользователь остаётся в текущей ветке

***git checkout some\_branch*** – переключение на ветку «some\_branch»

***git checkout –b some\_branch*** – **создание новой ветки и переключение на неё**

***git branch –d some\_branch*** – безопасное удаление ветки «some\_branch», т.е. ветка не удалится, если изменения, сделанные в данной ветке, не были смерджены в какую-либо другую ветку

***git branch –D some\_branch*** – принудительное удаление ветки «some\_branch»

***git push origin --delete имя\_ветки*** - удаление ветки в удалённом репозитории

|  |  |
| --- | --- |
| ***git reset --hard origin/branch\_to\_overwrite***  ***git pull*** | ***Спуллить изменения в локальный репозиторий, перезаписав локальные изменения*** |

***Мерджи (слияния веток)***

Слить одну ветку в другую:

***git checkout имя\_ветки\_в\_которую\_будем\_делать\_слияние***

***git merge имя\_ветки\_которую\_будем\_сливать***

**Работа с SSH-ключами**

*С какими протоколами работает Git и для чего нужны SSH-ключи?*

Git умеет работать с четырьмя сетевыми протоколами для передачи данных: локальный, SSH, "свой" протокол Git и HTTP[S].

Базовым протоколом является **Локальный протокол,** при использовании которого удалённым репозиторием считается просто каталог на диске. Не подходит для удалённого доступа через сеть, не считая частных случаев с использованием сетевых файловых систем (NFS, CIFS, etc.)

Наверное, наиболее часто используемый транспортный протокол — это **SSH.** Причина в том, что доступ по SSH, как правило, уже настроен в большинстве окружений. Кроме того, SSH — единственный из сетевых протоколов, предоставляющий доступ и на чтение, и на запись. Два других сетевых протокола (HTTP[S] и Git) в большинстве случаев дают доступ только на чтение, поэтому даже если они вам доступны, вам всё равно понадобится SSH для записи. К тому же SSH — протокол с аутентификацией и шифрованием трафика "из коробки". Недостаток SSH в том, что, используя его, вы не можете обеспечить анонимный доступ к репозиторию.

Другой вариант — "свой" **Git-протокол.** Вместе с Git'ом поставляется специальный демон, который слушает порт 9418 и предоставляет сервис, схожий с протоколом ssh, но абсолютно без аутентификации. Чтобы использовать Git-протокол для репозитория, вы должны создать файл git-daemon-export-ok, иначе демон не будет работать с этим репозиторием, но следует помнить, что в протоколе отсутствуют средства безопасности. Соответственно, любой репозиторий в Git'е может быть либо доступен для клонирования всем, либо не доступен никому. Как следствие, обычно вы не можете отправлять изменения по этому протоколу. Технически открыть доступ на запись можно, но из-за отсутствия авторизации в этом случае кто угодно, зная URL вашего проекта, сможет его изменить. Короче, это редко используемая возможность.

И последний вариант — **HTTP[S].** Прелесть протоколов HTTP и HTTPS в простоте их настройки. По сути, всё, что необходимо сделать — поместить голый репозиторий внутрь каталога с HTTP документами, установить перехватчик post-update и всё. Обратной стороной использования протокола HTTP является его относительно низкая эффективность для клиента. Обычно клонирование или извлечение изменений из репозитория при использовании HTTP гораздо продолжительнее, а объем данных и нагрузка на сеть намного больше, чем у любого другого имеющегося сетевого протокола.

**Итого: доступ к удалённому репозиторию по SSH — самый распространённый вариант настройки удалённого доступа, быстрый, удобный и безопасный.** Настроив авторизацию в SSH по ключам, Вы будете избавлены от необходимости вводить пароли для доступа к репозиторию, сохраняя, однако, приемлемый уровень безопасности.

*Генерация SSH-ключа:*

*Переходи в директорию с ключами:*

$ cd ~/.ssh

$ ls

authorized\_keys2 id\_dsa known\_hosts

config id\_dsa.pub

*Генерим ключ:*

$ ssh-keygen -o

Generating public/private rsa key pair.

Enter file in which to save the key (/home/schacon/.ssh/id\_rsa):

Created directory '/home/schacon/.ssh'.

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/schacon/.ssh/id\_rsa.

Your public key has been saved in /home/schacon/.ssh/id\_rsa.pub.

The key fingerprint is:

d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local

*Получаем содержимое ключа, которое потом копируем:*

$ cat ~/.ssh/id\_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU

GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3

Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA

t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En

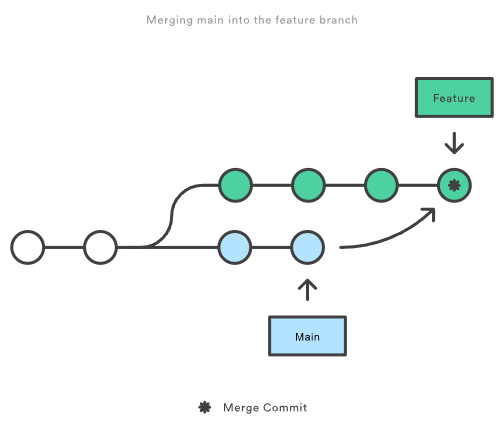
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx

NrRFi9wrf+M7Q== schacon@mylaptop.local

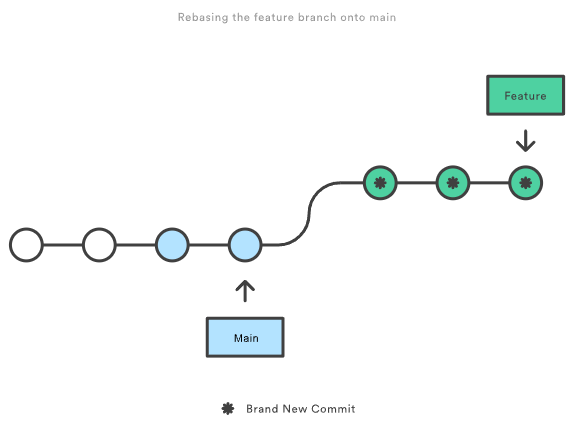
<https://git-scm.com/book/ru/v2/Git-%D0%BD%D0%B0-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%B5-%D0%93%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F-%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D0%B3%D0%BE-SSH-%D0%BA%D0%BB%D1%8E%D1%87%D0%B0>

**Отличия git merge от git rebase**

Git merge сохраняет всю историю коммитов. Когда вы выполняете слияние двух веток, Git создает новый коммит, который включает в себя различия между этими ветками. Этот новый коммит становится историческим стопом, объединяющим эти изменения. История ветки остается неизменной, а само действие слияния оставляет явный след, который можно увидеть в истории коммитов:



В отличие от merge, git rebase переписывает историю коммитов. Вместо создания нового коммита для объединения изменений, как это делает merge, rebase перемещает или "переносит" все изменения в вашей текущей ветке на целевую ветку. Это делает историю проекта более линейной, и это может упростить процесс чтения и понимания истории коммитов:

****