[**Git для всех**](https://stepik.org/course/126023)

**Система контроля версий**

Перед тем, как говорить про какую-либо конкретную систему контроля версий необходимо понимать, что вообще такое системы контроля версий, какими они бывают и зачем нужны. Да, сейчас будет теоретический блок, но куда же без него? Начинаем с азов)

Система контроля версий (СКВ, VCS, Version Control System, а Википедия дает еще и термин "Система управления версиями") – это программное обеспечение, позволяющее хранить все изменения, которые мы когда-либо вносили в проект. VCS используется для облегчения работы с изменяющейся информацией: позволяет хранить несколько версий одного и того же документа и при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое. VCS еще называют машиной времени для программистов (и не только). Давайте разбираться почему)).

Допустим, разработчику надо написать довольно большую программу с большим количеством функций, фич, какими-то особенностями. Логично предполагаем, что данную работу он будет делать по шагам:

1) разработает структуру программы

2) реализует небольшими порциями код программы

3) скооперирует программу целиком

4) запускает и проверяет на работоспособность

Если программа выполняет то, что разработчик задумал – это уже определенная часть работы, которую он сделал. Дальше он будет добавлять какие-то новые функции, тесты, и таким образом, постепенно придет к той программе, которую нужно было сделать изначально.

Кажется, что все просто, но на самом деле нет.

Путь написания программы очень тернист. Разрабатывая ту или иную функцию, программисты сталкиваются с разными архитектурными ограничениями, которые стопорят дальнейшую разработку. И в таких случаях возвращение на несколько шагов назад к предыдущему результату (версии) – до внесения изменений, которые есть сейчас – реально поможет увидеть, что добавлялось/менялось и в итоге вызвало проблему.

Выполняя небольшие кусочки работы, мы можем их фиксировать, создавая тем самым историю. С помощью VCS мы можем вернуться в любой кусочек в прошлых версиях. Кроме того, если мы хотим создать альтернативную реальность своей разработки – то тоже можем начать делать это с определенного момента в истории нашего кода. Можно провести аналогию с компьютерной игрой – прежде чем начать играть на новом уровне, надо сначала сохраниться на данном уровне. То же самое нам предоставляет и система контроля версий.

Если разработка обширная, то справиться в одиночку разработчику трудно. Надо подключать коллег, и система контроля версий также позволяет включать в этот процесс других программистов. Не важно, где создается код – на нашем ноутбуке или компьютере коллеги в другой стране. Важно, когда он создается и когда фиксируется – для этих моментов система будет создавать точки в истории нашего проекта.

**Виды систем контроля версий**

Итак, теперь вы знаете, что **VCS** – **Version Control System** – это cистема контроля версий. Неотъемлемыми теоретическими терминами при изучении Git являются еще

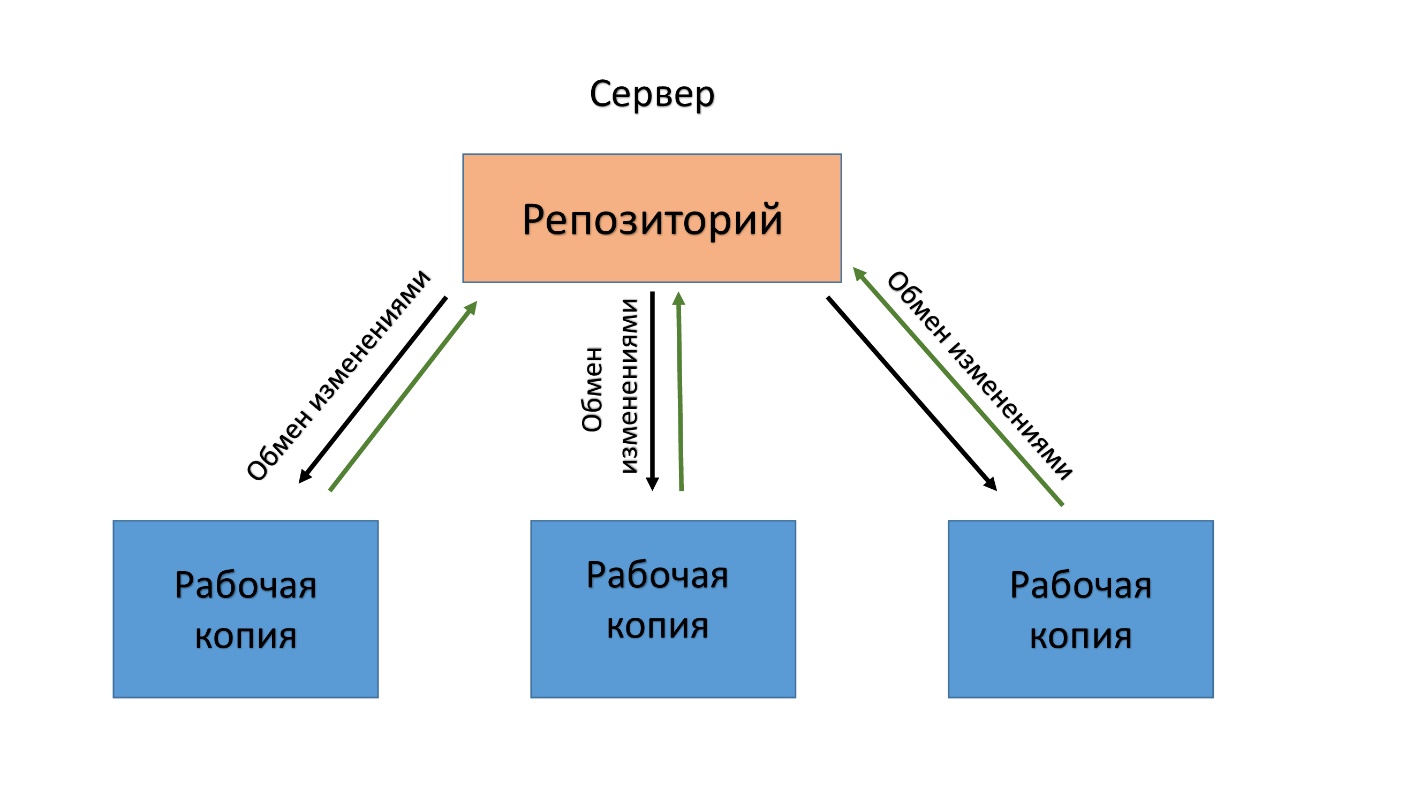
1)**репозиторий**(хранилище с кодом, если кратко; если не кратко – смотрим в теорию)

2) **коммит**(имеет 2 значения: фиксация изменений (снимок) или конкретная версия репозитория)

3) **ветка** (**брэнч, бранч, бранча**– кому как нравится – альтернативная реальность кода, история которого начинается от конкретного коммита).

Системы контроля версий бывают централизованные и распределенные.

1. Централизованные системы – это более старый вид контроля версий, который использовался в 70-е гг, но иногда применяется и в настоящее время. Здесь у нас есть один-единственный сервер (репозиторий), содержащий всю историю проекта, все версии изменений. Подключаясь к этому серверу, разработчики получают из репозитория код, работают с ним и отправляют его в центральный репозиторий на сервер. То есть, обмен изменениями происходит только с сервером.



Работать с Git – это работать в терминале, поэтому для комфортного прохождения данного курса надо знать, что такое терминал и как в нем работать.

**Терминал** – это программа, которая эмулирует поведение "железного" терминала, состоящего из клавиатуры и монитора. Это окружение, где мы можем вводить команды и получать на них ответ.

Запуская терминал, мы видим окошко с командной строкой внутри. Внутри себя терминалзапускает командную оболочку. **Командная оболочка и терминал — не одно и то же. Командная оболочка** – это устройство (программа), через которую мы управляем операционной системой и установленными программами, используя командную строку. В командной строке мы передаем системе команду, которую та должна выполнить.

**Командная строка** – это то место, куда мы будем вводить свои команды – приглашение терминала для ввода.

Оболочка занимается только передачей и приемом данных, но все эти данные должен еще кто-то обрабатывать, выполнять команды, интерпретировать их синтаксис. Этим занимается терминал. Командных оболочек достаточно много, это bash, shell и другие, но чаще всего применяется bash.

Git Bash — это приложение для сред Microsoft Windows, эмулирующее работу командной строки Git. Оно идет в составе установочного пакета для Windows, оно представляет собой пакет, который устанавливает в операционную систему Windows оболочку Bash, некоторые утилиты Bash и систему Git.

После того, как вы скачаете и установите Git на Windows, найдите входящий в состав пакета файл .exe и откройте его, чтобы запустить Git Bash.

**Работа с локальным репозиторием**

Существует два способа: инициализировать новый или клонировать уже существующий.

При инициализации мы привязываем наш проект с системой Git.

В первом случае мы придерживаемся следующего алгоритма действий:

1) создаем папку будущего проекта, переходим в нее и инициализируем репозиторий в ней с помощью команды git init .. Точка после пробела в конце команды обязательна.

2) репозиторий готов к работе: можем добавлять в него свои файлы, редактировать их, удалять и т.д.

3) после этого делаем коммит изменений, который говорит, что теперь новые файлы готовы к отправке в удаленный репозиторий

4) связываем локальный и удаленный репозитории с помощью определенной команды, которая дает Git'у знать, где находится удаленный сервер (куда отправлять файлы)

5) отправляем данные в удаленный репозиторий используем команду git push.

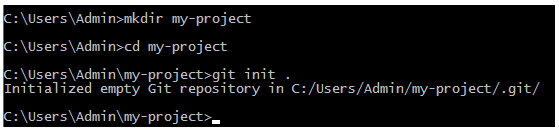
Если мы хотим использовать Git для уже имеющегося у нас проекта, делаем все то же самое, открыв нужный каталог.

Во втором случае мы хотим скопировать репозиторий из удаленного хранилища (например, GitHub). Мы будем клонировать себе его копию, выполним команду git clone с указанием ссылки на удаленный репозиторий.

Мы разберем 1-й вариант — будем инициализировать новый репозиторий.

Для начала создадим в терминале папку "my-project" и зайдем в нее с помощью команды cd my-project

Пока эта папка еще не является локальным гит-репозиторием: гит за ней не следит и вообще ее не видит. Для того, чтобы подключить гит, нужно создать базу данных для него с помощью команды с точкой через пробел в конце git init . Это создаст на базе текущей папки my-project новый подкаталог .git (репозиторий Git), в котором и будет содержаться вся конфигурация Git и история проекта. Точка после команды говорит о том, что именно в этой папке, где мы сейчас находимся (my-project), и будет база данных git:



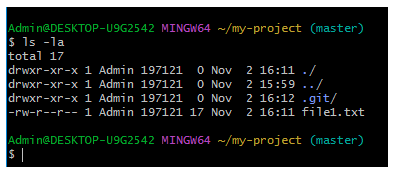
Итак, запоминаем шаги создания папки .git внутри папки нашего проекта:

mkdir my-project

cd my-project

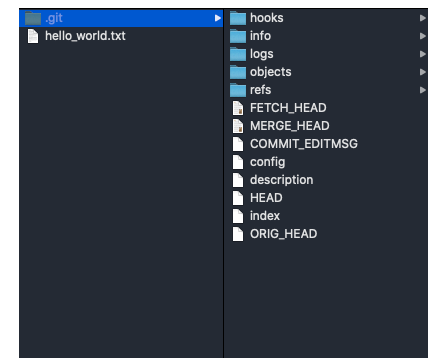
git init .

Посмотрим, что у нас создалось с помощью команды ls -la (данная команда не работает в командной строке Windows, там эквивалентом является команда DIR; но отлично работает в командной строке Git Bash, достаточно в проводнике Windows в локальном репозитории нажать правую кнопку мыши и в меню выбрать Git Bash Here):



Фиолетовым подсвечивается наша директория .git

Зайдем в эту директорию с помощью команды ls .git и рассмотрим, какие файлы и папки она содержит.



В папке hooks гит хранит набор скриптов, которые могут автоматически запускаться во время выполнения git команд.

В папку info гит заботливо поместит файл exclude, в котором можно указывать любые файлы, и гит не станет добавлять их в свою историю. Это почти то же самое что и .gitingnore (рассмотрим чуть позже), за тем исключением, что exclude не сохраняется в истории проекта, и мы не сможем им поделиться с другими.

В папке logs гит будет хранить историю проекта для всех веток в этом проекте.

В папке objects гит соберет blob-объекты, каждый из которых проиндексирован уникальным SHA. Не заморачиваемся, просто доверяем гиту))

В папку refs гит закинет копию ссылок на объекты коммитов в локальных и удаленных ветках.

FETCH\_HEAD  и MERGE\_HEAD хранят в себе ссылки в виде SHA на ветки, которые участвовали соответственно в git fetch или git merge. Эти команды изучим в других главах.

В файле COMMIT\_EDITMSG содержится последнее введенное нами сообщение коммита (да, к коммиту надо добавлять сообщение)

Файл config содержит настройки Git репозитория, как раз здесь хранятся сконфигурированные нами в п.2.2 параметры: email и имя пользователя.

Файл description содержит в себе название проекта и его описание. Он предназначен для GitWeb, специального веб-интерфейса, написанного для просмотра Git репозитория через веб-браузер.

Файл HEAD содержит ссылку на текущую ветку, в которой мы работаем.

Файл index является промежуточной областью с файлами, которые уже упакованы Git. Это хранилище для файлов с изменениями, информация о которых потом попадет в единый коммит.

В файл ORIG\_HEAD будут попадать ветка, с которой проводилось слияние.

Отлично, запомним команды (вернитесь выше, чтобы вспомнить, что они обозначают):

ls -la

ls .git

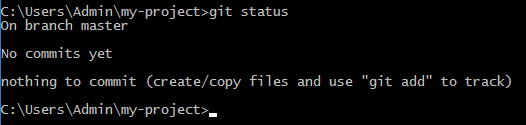
Итак, наш проект myproject находится в локальном репозитории, с помощью команды git init мы инициализировали в этом проекте репозиторий Git. Теперь наш проект (наша папка) myproject будет отслеживаться гитом, а может и не отслеживаться, в любом случае он является рабочим каталогом (working directory), т.к. в нем находится папка .git.

Папка .git – это база данных Git. Именно в нее будут сохранятся все наши изменения, это и есть наш локальный репозиторий.

Задав команду git status , мы увидим, что в этой папке ничего не было сделано:

No commits yet

nothing to commit   – все пусто.



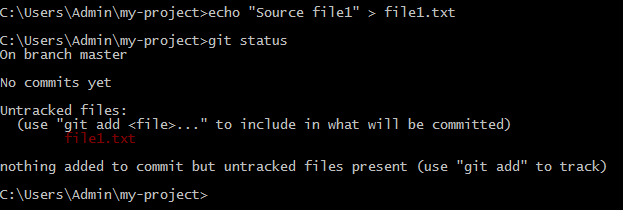
git status – данная команда показывает, **что** сейчас происходит. Команда git status показывает, на каком этапе мы сейчас находимся, показывая статус всех файлов: количество untracked, deleted, new и прочих файлов, количество коммитов, на которое отличается локальная версия репозитория от удаленного и т.д.

Создадим файл file1.txt с каким-то текстом, например, "Source file1":

echo "Source file1" > file1.txt

Команда "echo "содержимое файла" > название\_файла" создает файл с указанным содержимым.

Теперь зададим команду git status и увидим, что наша директория отслеживается гитом:



Гит увидел новый для него файл, подсветил его красным цветом и определил, что он – Untracked. В данном состоянии Git просто знает об этом файле в проекте,  но еще не отслеживает его. Чтобы отслеживать файл, мы должны его зафиксировать (сделать коммит). А чтобы зафиксировать, нам надо подготовить его к фиксации — отправить его в промежуточную область — stage (staging area, индекс, стадия подготовленных файлов).

Для добавления файла в stageиспользуется команда git add, а сам процесс добавления файлов в промежуточную область называется индексацией. Индекс — промежуточное место между нашим прошлым коммитом и следующим. Мы можем добавлять (git add) или удалять файлы из индекса (git reset), а чтобы просмотреть индекс, используется команда git status.

Вообще понятие индекса в Git появилось неслучайно. Ведь когда разработчик работает пусть даже над одной задачей, то выполняя ее, он вместе с тем исправляет и разные недочеты в коде. В итоге в рабочей директории появляется много разных исправлений, которые частично относятся к выполняемой задаче, а частично содержат исправления, напрямую не связанные с основными изменениями. И если делать ровно один коммит, включающий в себя и основную задачу, и дополнительные исправления, то будем иметь неприятные побочные эффекты:

- сложнее смотреть историю, ведь коммит начинает содержать совершенно несвязанные изменения, которые отвлекают во время ревью;

- откат коммита по любым причинам приведет к тому, что откатятся правки, которые все равно нужно будет делать.

Именно здесь помогает индекс: он позволяет меньше переживать на тему того, как сформируется коммит.

С помощью индексации гиту не приходится отслеживать все файлы в проекте, т.к. некоторые файлы (например, файлы классов, файлы журналов, файлы результатов, временные файлы данных) создаются динамически, и отслеживать их версии просто бессмысленно. А вот созданные нами артефакты проекта — файлы исходного кода, файлы данных, файлы конфигурации и другие — содержат бизнес-логику приложения. Они должны отслеживаться гитом, поэтому мы их обязательно добавляем в промежуточную область.

Стандартный способ работы с индексом — это добавление или изменение файлов и последующий коммит.

Если мы добавим наш файл в индекс с помощью git add , то с этого момента Git начинает его отслеживать.

Останется только закоммитить внесенные изменения (= создали файл file1.txt с текстом "Source file1")  командой: git commit и оставить небольшое описание в кавычках (одинарных или двойных — не важно, важно оставить комментарий):

git commit -m 'my first commit, this is Version 1.0'

таким образом мы сделаем наш первый коммит)

Команда git commit **делает коммит**— **снимок текущего состояния изменений, добавленных в индекс**. Такие коммиты (подтвержденные снимки состояния) можно рассматривать как «безопасные» версии проекта — Git не будет их менять, пока мы явным образом не попросим об этом. Каждый коммит имеет:

- хэш (уникальный id), например, 549892a096...2184cf57f7

- сообщение, например "Added files" (commit message говорит, ЧТО делает коммит, а не КАК делает)

- cписок измененных файлов, например, index.txt

- изменения по каждому файлу

Если посмотрим статус, то увидим, что никаких изменений больше нет, а наши файлы соответствуют последнему снимку:

https://ucarecdn.com/16cfa856-3112-402c-9400-269fe0906c60/

Итак, на данном этапе мы поняли, что сделать коммит — это значит сделать изменения, собрать эти изменения командой git addи указать коммит-сообщение после ключа "-m" (расшифровывается как "message"). Кстати, старайтесь сразу писать правильные комментарии после  ключа -m: по вашему комментарию к коммиту должно было понять, какие изменения были сделаны. Это также должно помочь новым участникам проекта лучше понять смысл коммитов: надо знать, зачем человек его сделал и какие последствия это вызовет.

Команда git commit берет все данные, добавленные в индекс с помощью git add, и сохраняет их снимок во внутренней базе данных.

Важно! Все файлы, для которых мы не выполнили git add после момента редактирования, не войдут в этот коммит. В нем будет только их старая версия (если таковая имеется).

Остается только запушить наши результаты командой git push на удаленный сервер, чтобы другие разработчики тоже имели к ним доступ.

После этого урока вы должны знать следующие команды Git:

git init . – создает новый репозиторий

git status – отображает список измененных, добавленных и удаленных файлов

git add  – добавляет указанные файлы в индекс

git reset – отменяет действие команды git add на файл

git commit– фиксирует добавленные в индекс изменения

git commit -m – фиксирует добавленные в индекс изменения с определенным сообщением

git push – отправляет изменения на удаленный репозиторий

**Статусы (состояния) файла в Git**

**untracked** — неотслеживаемый гитом файл

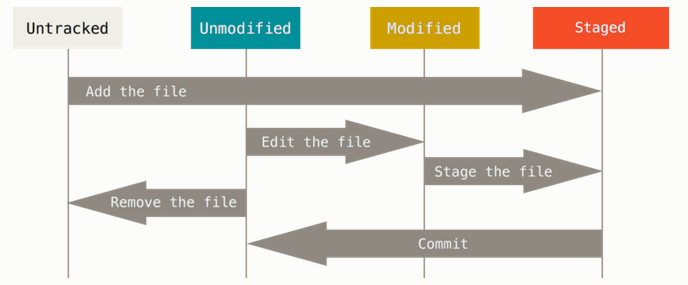
**unmodified** — неизмененный отслеживаемый гитом файл

**modified** — измененный отслеживаемый гитом файл

**staged** — подготовленный к коммиту отслеживаемый гитом файл

Каждый файл в рабочем каталоге находится в одном из двух состояний: отслеживаемый гитом (он уже о нем знает) и неотслеживаемый. Отслеживаемые файлы — это те файлы, для которых **был сделан коммит**(он зафиксирован, для него есть последний снимок состояния в проекте). Они могут быть неизмененными, измененными или подготовленными к коммиту. Неотслеживаемые файлы — это любые файлы в рабочем каталоге, которым еще **не делался коммит** и еще не подготовлены к коммиту.

Таким образом, когда мы впервые клонируем репозиторий, все файлы (и папки) будут проходить следующие состояния:



Состояние Untracked имеют все неотслеживаемые файлы и директории. Отслеживаемые файлы, как уже было сказано выше, могут находится в 3 состояниях: неизмененный отслеживаемый (Unmodified), измененный отслеживаемый (Modified) и подготовленный для фиксации в коммит (Staged).

Когда мы с помощью команды git add добавляем неотслеживаемый файл, то он переходит в состояние Staged и начинает отслеживаться гитом:

Untracked     >    Staged

Если у нас файл уже отслеживается гитом (в состоянии Unmodified) и мы его изменяем — добавляем строчки, удаляем что-то и т.д. — то он переходит в состояние "измененный отслеживаемый":

Unmodified   >   Modified

В этом состоянии Git продолжает осуществлять контроль над всеми сделанными в этом файле изменениями, просто файл не будет добавлен в следующий коммит. Статус Modified показывает, что файл имеет историю в системе Git и был изменен относительно его последнего состояния. Если мы теперь сохраним наш измененный файл, то он переходит в состояние Staged:

Modified    >    Staged

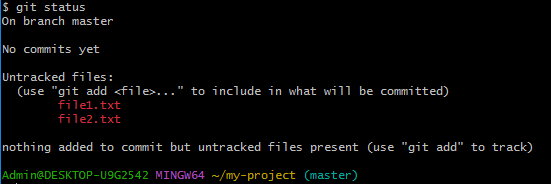
Если мы делаем коммит файла, который находится в состоянии Staged, он переходит в состояние Unmodified (неизмененный отслеживаемый):

Staged    >   Unmodified

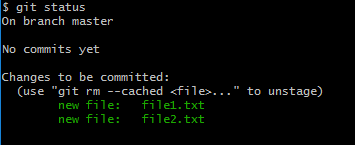
Весь цикл можно представить следующим образом: мы изменяем файл, сохраняем его в индексе и делаем коммит, а потом все сначала  
Unmodified    >   Modified   >   Staged    >   Unmodified

Разберем эти состояния на примере

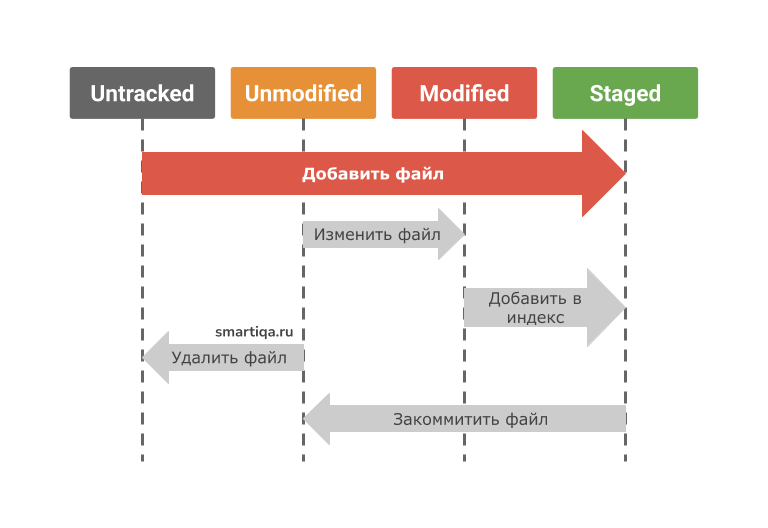
Вернемся в нашу папку myproject, в которой уже есть один файл file1.txt. Создадим еще один файл file2.txt и видим, что у нас имеются 2 неотслеживаемых файла:



Из прошлого урока мы узнали, что для того, чтобы сделать файл отслеживаемым, мы используем команду git add. С ее помощью мы добавим наши файлы file1.txt и file2.txt в индекс (команды git add \* и git add . добавляют все файлы). Если теперь мы посмотрим его статус с помощью git status, то увидим, что они подсвечиваются зеленым и для Git они **new**, т.е.файлы только начали отслеживаться гитом и пока не имеют истории:



Файлы были в статусе Untracked (только что созданы и еще не отслеживаются гитом), в выводе команды git status они приняли состояние new, значит, что они перешли в статус Staged:



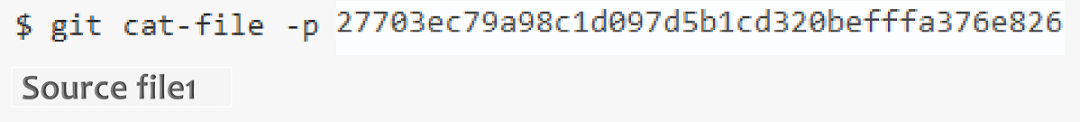
Теперь Git видит их. Разберем на примере файла file1.txt, что сделал Git, когда мы задали команду git add?

**1. Git взял файл file1.txt, создал для него новый объект, добавил его в папку оbjects каталога .git в свою базу данных и дал ему имя** — т.е. сохранил его как блоб-файл (подробнее о блоб-файлах, их именах см. 1 Глава, 1.3.Строение Git)

Все файлы ссылаются друг на друга с помощью 40-значного SHA-1 — уникального идентифицирующего хэша (контрольной хэш-суммы содержимого и заголовка объекта), его можно посмотреть в репозитории:



Эта 40-значная строка (у вас скорее всего будут другие цифры-буквы)  https://ucarecdn.com/1e6e01f5-5f14-4e1d-9ca3-0a93b26bee6a/  и есть единица хранения файла file1.txt с именем https://ucarecdn.com/1e6e01f5-5f14-4e1d-9ca3-0a93b26bee6a/. Первые 2 символа определяют подкаталог файла (27), остальные 38 — имя. Данный объект https://ucarecdn.com/98a0cbae-bc41-4b52-aea7-26a7e9ca8f40/ хранит снимок содержимого файла file1.txt.  Git создал директорию .git/objects/27 и сохранил в нее блоб-файл с именем 703ec79a98c1d097d5b1cd320befffa376e826 Получить обратно содержимое объекта можно командой cat-file   с ключом -p, который означает автоматическое определение типа содержимого и вывод содержимого на печать в удобном виде:



Простое добавление файла в Git приводит к сохранению его содержимого в папке objects. Теперь мы умеем добавлять данные в Git и извлекать их обратно))

**2. Добавил имя сохраненного blob-файла в индекс.**

Мы помним, что в папке .git/index, Git хранит список файлов, за которыми он следит. Каждая строчка состоит из имени файла и его хэша. Вот таким получилось содержимое файла file1.txt в .git/index:

https://ucarecdn.com/67e8fce1-d023-46a8-8aa7-5a4d5ef4ce72/

(Подробнее про остальные объекты в Git (дерево, таг, коммит) — можно почитать здесь: [https://uleming.github.io/gitbook/1\_Объектная\_модель\_git.html](https://uleming.github.io/gitbook/1_%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_git.html) )

Выполним следующее задание:

1. Создадим еще один файл num.txt с содержимым ffffff:

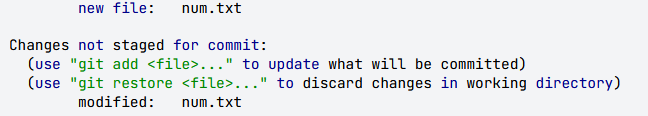
echo "ffffff" > num.txt

2. Добавим его в индекс и посмотрим статус. Появился новый файл:

https://ucarecdn.com/07be7f5b-0948-40e4-8526-392e9fd4da25/

3. Изменим содержимое файла num.txt  с ffffff на aaaaaa (команда для редактирования: nano <имя файла> откроет консольный текстовый редактор для изменения содержимого файла. Она не работает в командной строке Windows, поэтому рекомендуется использовать Git Bash. Чтобы сохранить изменения и выйти из редактора, надо нажать Ctrl+X) и посмотрим статус снова.

Будет примерно так:



4. Прежде чем читать дальше, ответьте на вопрос: какой статус появился у файла num.txt?

Думаете, modified?

Не совсем)), файл num.txt одновременно находится в двух состояниях: staged (у него ведь еще состояние new) и modified.

Сейчас разберемся, как такое возможно)

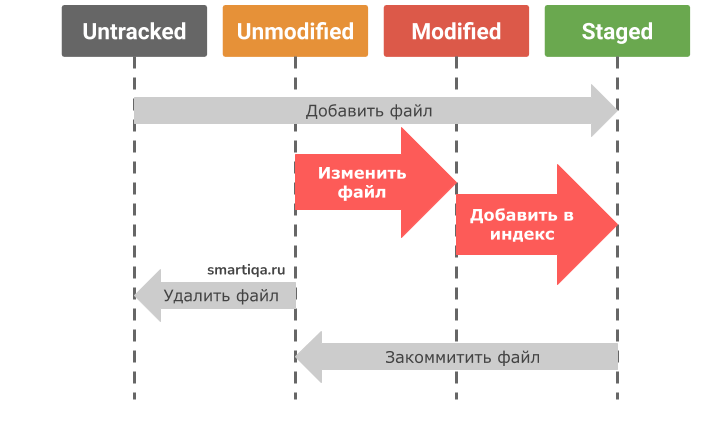
Когда мы изменили содержимое файла num.txt  с ffffff на aaaaaa, вместе с этим поменялась и хэш-сумма. Git это заметил, и предупредил нас: состояние modified не проиндексировано:

Changes not staged for commit

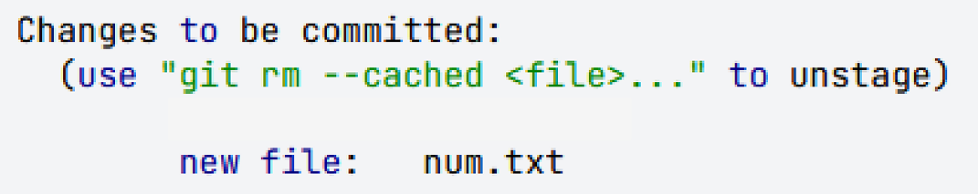
Если мы сейчас сделаем коммит, то в него попадет файл num.txt со значением ffffff, а не aaaaaa. Чтобы в коммит попали новые изменения, нам нужно заново проиндексировать файл num.txt (добавим новые изменения файла в индекс):

git add num.txt

Происходит    Unmodified  >  Modified   и  затем   Modified > Staged:



Посмотрите статус и убедитесь, что num.txt теперь тоже в staged:



Кстати, кроме файлов в Git можно добавлять и папки. Чтобы добавить новую директорию в гит, надо добавить хотя бы один файл из этой директории в Git.

**Зачем и как мы добавляем файл в .gitignore**

На любом проекте в рабочей директории всегда есть файлы и папки, которые мы не только не хотим автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. Обычно к таковым относятся автоматически генерируемые файлы: различные логи, результаты сборки программ и т.п.

Поэтому в Git можно создать специальный игнорируемый файл .gitignore, в котором мы перечислим шаблоны, соответствующие таким файлам, а также файлы, которые по какой-либо иной причине не должны попадать в коммиты. Игнорируемый файл Git помечает как файл, который необходимо игнорировать. Данный файл желательно настроить вручную до того, как начать серьезно работать, так как это защитит от случайного добавления в репозиторий файлов, которых мы там видеть не хотим.

Так выглядит файл .gitignore:



В первой строке гиту приказано игнорировать все файлы, которые заканчиваются на .rbc, в четвертой — игнорировать директорию log, в шестой — все файлы конфигураций баз данных sqlite3 и т.д.

Для большинства правил используются определенные шаблоны, например, если мы хотим ограничиться только корневым уровнем в репозитории — необходимо явно это указать поставив “/” перед шаблоном.

К шаблонам в файле .gitignore применяются следующие правила:

- пустые строки, а также строки, начинающиеся с #, игнорируются

- можно использовать стандартные glob шаблоны

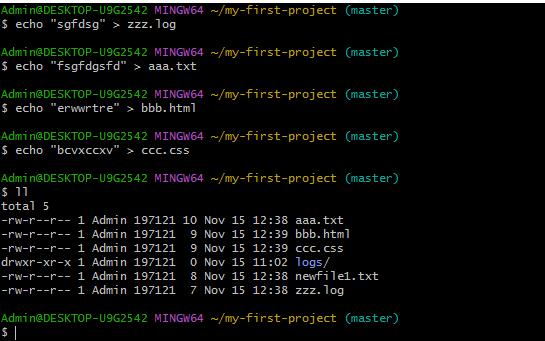
- можно заканчивать шаблон символом слэша (/) для указания каталога

- можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

Каждый игнорируемый файл указывается в отдельной строке. Git будет полностью игнорировать эти файлы, поместив пути к ним в [.gitignore](https://www.atlassian.com/git/tutorials/gitignore).

От теории —  к практике

Создадим несколько файлов и папку logs (для создания папки нужна команда mkdir имя\_папки), посмотрим их с помощью команды ll и проверим статус:



Да, точно, файлы и папка logs есть. Понимаем, что папка logs и файлы zzz.log и aaa.txt не должны попасть в коммит. Создаем здесь же файл .gitignore:

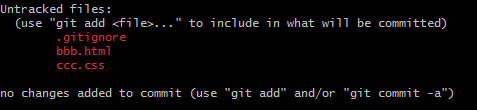
https://ucarecdn.com/88fb4969-ccb4-47fe-8d89-82ba68424f7a/

Этой командой мы создаем файл .gitignore и одновременно открываем редактор nano, в котором вручную пропишем файлы, которые должны быть проигнорированы:



Одна звездочка — это подстановочный знак, который может соответствовать как нескольким символам, так и ни одному, главное, что файлы с такими расширениями — .log и .txt — теперь будут игнорироваться гитом. Косая черта в конце шаблона означает каталог, т.е. наша папка logs тоже будет игнорироваться. Чтобы выйти из редактора, нажмите Ctrl+X, затем Ctrl+Y (для подтверждения выхода) и Enter.

Теперь если мы посмотрим статус, то увидим, что все файлы, попадающие под игнорирование, не отобразятся в выводе команды git status: у нас есть .gitignore и еще файлы, которые нужно добавить в репозиторий. Папки и файлов с расширениями .log и .txt уже нет:



Проиндексируем их и увидим, что эти файлы стали зелеными. Файл .gitignore надо добавить в репозиторий, поэтому закоммичиваем наши файлы. Теперь если мы создадим какой-нибудь файл с расширением .log, которое мы "заигнорили", Git его не видит.

**Итог:**

1. Чтобы сделать файл отслеживаемым, используем команду git add

2. Когда мы делаем файл отслеживаемым, происходит следующее:

   2.1. Создается блоб-объект для этого файла с именем – 40-символьным хэшем содержимого файла, причем его первые две буквы называют подпапку в папке .git/objects, а остальные 38 – сам файл. Такое разделение имени ускоряет поиск blob-файла среди других.

   2.2. Имя этого blob-файла записывается в индекс и с этого момента GIt считает файл подготовленным к коммиту.

3. Если мы поменяем содержимое файла, нам нужно снова добавить его в индекс.

4. Индекс — промежуточное место между нашим прошлым коммитом и следующим. Мы можем добавлять или удалять файлы из индекса, а также просмотреть индекс с помощью git status.

5. Команда git status показывает, на каком этапе мы сейчас находимся

5. Коммиты создаются с помощью команды git commit, которая делает снимок состояния проекта на текущий момент времени. Коммиты снимков состояния Git всегда выполняются в локальный репозиторий.

**Редактирование закоммиченных файлов**

В пункте 3.1. когда мы разбирали создание блоб-объекта, то редактировали еще незафиксированный (незакоммиченный) файл. А как редактировать закоммиченный файл?

После того, как вы переварили п.3.1, теперь для вас это просто цветочки)

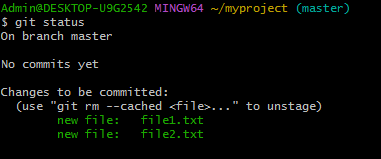
Итак, создадим в терминале папку "myproject", зайдем в нее и подключим к ней Git.

Создадим в ней файл file1.txt с текстом "Source file1"

echo "Source file1" > file1.txt

По аналогии создадим в ней еще файл file2.txt с текстом "Source file2"

Добавляем их в индекс и проверяем статус:

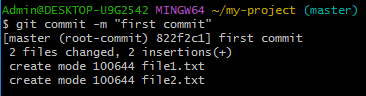


Закоммичиваем их  с помощью команды git commit -m "first commit", это наша первая версия репозитория. К команде git commit всегда нужно добавлять флаг -m (от слова message) и текст сообщения в кавычках, в противном случае вы можете увидеть такое:

https://ucarecdn.com/2efd37e8-4604-4e5c-a568-ab87f7a8e8f8/

т.е. гит очень не любит коммиты без комментариев, имейте ввиду)

После команды git commit -m "some text" мы увидим, что сделали первый коммит (первую версию проекта), в котором у нас 2 файла:

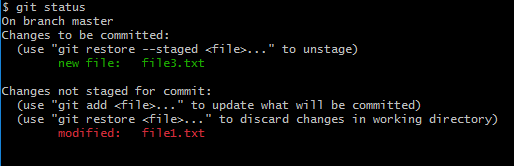


Создадим еще файл file3.txt с текстом "Source file3", добавляем его в индекс, но не закоммичиваем. Теперь у нас в репозитории 3 файла: 2 закоммиченных и один новый, отслеживаемый гитом, это пока что первая копия (версия) нашего репозитория.

Понимаем, что наш file1.txt надо подправить. Заходим в него:

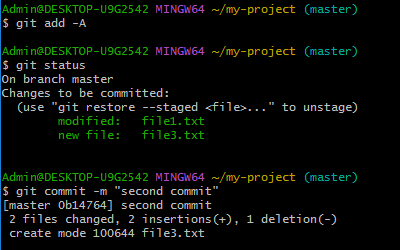
nano file1.txt

Исправляем текст Source file1 на другой или добавляем какой-нибудь код, возвращаемся обратно в терминал (чтобы сохранить изменения и выйти из редактора можно воспользоваться сочетанием клавиш Ctrl+X. Т.к. мы изменяли файл, то появится вопрос: "Save modified buffer (ANSWERING No Will DESTROY CHANGES)? ", жмем клавишу "y" и затем Enter. Изменения будут сохранены, редактор закроется) и проверяем статусы наших файлов:



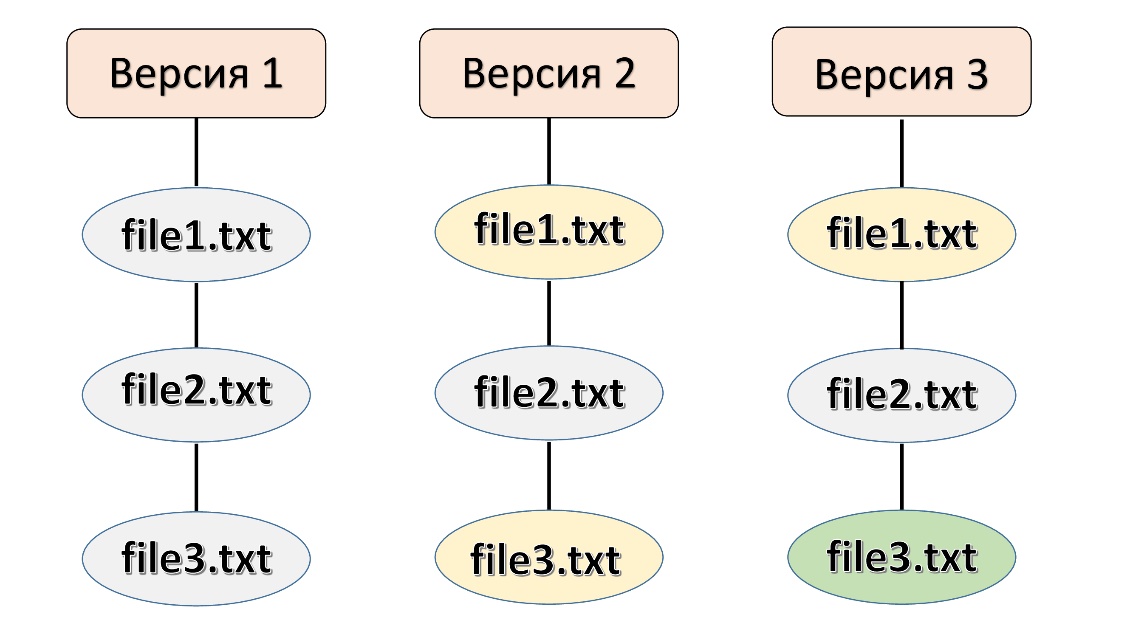
Видим измененный file1.txt (ранее закоммиченный) и отслеживаемый гитом (но еще не закоммиченный) file3.txt.

Если нам нужно добавить и эти изменения в наш репозиторий, то используем git add \* или git add -А или git add .(добавит все новые файлы), затем проверим статус, чтобы убедиться, что наши файлы в индексе. Остается только сделать им коммит, в котором подписать, что мы изменили:



Готова вторая копия нашего репозитория.

Так, в первой версии (копии) репозитория хранятся созданные нами 3 файла: file1.txt и file2.txt закоммичены, file3.txt отслеживается. Во второй хранятся эти же самые файлы, но file1.txt и file3.txt изменены (в file1.txt мы поменяли содержание, а file3.txt закоммитили), а file2.txt остался такой же, как и в первой версии. Если теперь нам что-то надо поменять в file3.txt и мы его редактируем, а потом коммитим — у нас уже создастся третья версия нашего репозитория:



Каждым новым коммитом мы создаем новую версию нашего репозитория. И на каждую версию мы можем вернуться, когда захотим.

**Работа с удаленным репозиторием**

**Создаем проект и пушим первый коммит**

Существует 2 способа создания проекта:

1. Если у нас проект с нуля: создать репозиторий на github и склонировать этот проект (репозиторий) к себе на компьютер.

2. Если у нас уже есть проект: зайти в папку проекта и связать его с уже существующим репозиторием на Github. Это называется инициализация.

Рассмотрим оба способа.

1. Создаем новый проект.

Та как мы уже создали свой аккаунт на GitHub, то давайте создадим здесь какой-нибудь проект, который будет нашим удаленным репозиторием. Сделаем себе его копию, поработаем в ней, а потом зальем все сюда!

1. На главном экране GitHub Desktop выбираем пункт «Create a New Repository on your hard drive». Или же в правом верхнем углу между колокольчиком и фотографией вашего профиля нажмите на знак "+"

2. Откроется окошко Create a new repository, где в поле Owner *—* это вы, а в поле Repository name вводим название репозитория mysuperproject1, описание пропустим, если хотим, то подключаем README файл (можно и без него) и нажимаем на кнопу Create a repository.

3. Далее Git выдаст нам подсказки, как связать удаленный репозиторий с нашим локальным компьютером. Нас интересует клонирование, чтобы можно было работать локально и потом загружать все обратно. Для этого нам нужны:

- ссылка на проект (есть у репозитория)

- терминал

- папка на локальном компьютере, куда будем клонировать (назовем ее test).

Для начала создадим папку. Откроем терминал и перейдем в нее в терминале, напишем команду git clone и вставим ссылку на репозиторий mysuperproject1. Ссылку увидим, когда нажмем на зеленую кнопку Code:

https://ucarecdn.com/37e47a18-d057-4909-ac8a-febb4903e078/

Берем ссылку по HTTPS (url-адрес) и в терминале после команды git clone пропишем эту ссылку

git clone [https://github.com/](https://github.com/vuejs/vue.git)kotBantik/mysuperproject1

После того, как в терминале нажмем на Enter, в текущей папке test создастся папка mysuperproject1 с файлом README. По умолчанию .git уже тут тоже имеется.

Возможно, такой способ больше не пройдет, тогда понадобится ссылка по SSH. Копируем ее, вставляем после команды git clone, жмем Enter и видим, что система просит от нас пароль для гита. Т.е. нам нужен ключ SSH у себя на компьютере.

Сначала убедимся, что у нас нет ключа SSH, т.к. по умолчанию пользовательские SSH ключи сохраняются в каталоге ~/.ssh нашего домашнем каталоге . Перейдем в этот каталог и посмотрим его содержимое:

cd ~/.ssh

ls

Ищем файлы с именами  id\_rsa и  id\_rsa.pub.  Если таковых нет (или даже нет каталога .ssh), то генерируем их, используя программу ssh-keygen, эту команду прописываем в терминале:

ssh-keygen

Программа начнет создавать файл ssh и попросит указать расположение файла для сохранения ключа (.ssh/id\_rsa), затем дважды ввести пароль для шифрования. Вводить пароль каждый раз при использовании ключа будет неудобно, поэтому оставляем пустым и подтверждаем. Затем подтверждаем еще раз и теперь все готово: нашей домашней директории видим файл .ssh. Если мы зайдем в нее, то увидим 2 ключа: 2 файла с именем id\_rsa и id\_rsa.pub . Файл с расширением .pub — это наш открытый ключ, а второй файл — наш приватный ключ.

Здесь нам нужен только открытый ключ — заходим в него командой cat, копируем содержимое и идем в настройки своего аккаунта на гитхабе. Ищем в них SSH Keys и в открывшееся поле Keys вставляем и жмем Add key.

Теперь снова идем в проект и делаем клонирование по SSH, а не по HTTPS. Вставляем в терминале эту ссылку после команды git clone, подтверждаем и видим, что скопировался репозиторий mysuperproject1 без каких-либо паролей).

Зайдем внутрь, создадим файл file1.txt с текстом "HelloWorld" и закинем его на удаленный репозиторий!

Видим наш красный файлик, индексируем его и закоммичиваем.

А теперь отправляем обратно на сервер:

git push origin

Т.е. дословно мы даем гиту команду отправить (запушить) данные на origin (удаленный репозиторий на сервере).

Перейдем в удаленный репозиторий на свой аккаунт и видим там наши изменения. Поздравляем себя! Мы запушили коммит не сервер!

2. Используем имеющийся проект в качестве репозитория

Допустим, у нас на компьютере уже есть проект (папка, которая и будет репозиторием) something. Заходим в папку проекта и выполняем команды в терминале:



Что мы сделали данными командами:

1) проинициализировали репозиторий — связали нашу локальную папку something с гитом

2) через git add добавили файл README

3) сделали начальный коммит ("Initial commit")

4) с помощью команды git remote add origin подвязывает наш локальный репозиторий с удаленным репозиторием: "удаленный добавляем к настоящему (вновь созданному)", который будет находиться по адресу git@github.com название\_аккаунта/название\_репозитория

5) после git remote add origin обязательно пушим наши изменения: git push origin master. Т.к. мы уже подвязали к конкретному репозиторию something наш локальный репозиторий something, то вместо названия репозитория для пуша мы просто указываем origin (оно является принятым сокращением для названия репозитория), а также прописываем ту ветку, которую будем пушить *—* master. Если мы хотим запушить текущую ветку, на которой работаем сейчас — тогда после origin пишем HEAD:

git remote add origin HEAD

Скорее всего появится сообщение типа "Sign in что-то там", это значит, что нам надо авторизоваться не только в браузере, но и в той системе гит, которая у нас на компьютере. Жмем на синюю кнопочку Sign in with your browser, в открывшемся окошке подтверждаем авторизацию (зеленая кнопка), затем обновляем нашу страничку на гитхабе.

Все, можно приступать к работе над проектом. Он у нас теперь и там и тут)

P.S. *—*с этого момента и дальше будем считать, что репозиторий = проект.

Команды этого урока

git clone [https://github.com/](https://github.com/vuejs/vue.git)kotBantik/mysuperproject1*—*клонирует удаленный репозиторий на локальную машину

git remote add origin [https://github.com/KotBantik/something.git](https://github.com/)*—*cвязывает наш локальный репозиторий с удаленным репозиторием

git push origin*—*отправляет коммиты на сервер

git remote add origin HEAD*—*отправляет на сервер текущую ветку со всеми коммитами, которые мы сделали в ней.

**Что такое git pull**

Итак, мы запушили коммит на удаленный репозиторий с помощью git push origin

origin *—*это удаленная ветка (на сервере, в Github). По умолчанию мы находимся в ветке master. Подробно работу с ветками мы рассмотрим в следующем уроке, а пока достаточно запомнить, что master (или main) *—* это то, что на нашей машине, а origin *—* в удаленном репозитории, на гитхабе. Поэтому иногда вместо команды git push origin пишут git push origin master , которая расшифровывается как: отправить на сервер ветку master.

Теперь научимся "пуллить". Pull - это скачивание данных с сервера. Процесс похож на клонирование репозитория, но здесь мы скачиваем не все коммиты, а только новые. Для этого используем команду git pull *—* стягиваем (получаем) изменения от наших коллег с удаленного репозитория. Или от себя самого, если работаем на разных машинах).

Допустим, мы начали работать в компании над новым проектом. Склонировали полностью весь репозиторий, со всеми его файлами, коммитами, индексами и тому подобным. Поработали в нем, внесли изменения в локальный репозиторий (индексация и фиксация),  отправили произведенные изменения на удаленный сервер, на GitHub с помощью git push. Теперь все наработки, которые мы сделали на домашнем компьютере, оказались с считанные секунды помещенными на сервере GitHub, под бдительным оком Git.

То же самое сделали и другие участники команды.

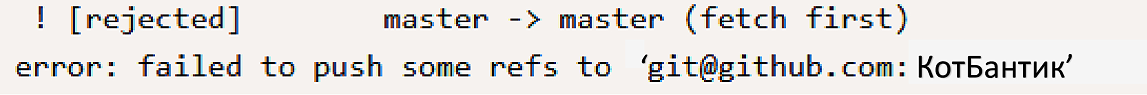
На следующий день мы приходим на работу и хотим продолжить работать*—*воспользуемся командой git pull — увидим, что нового добавили наши коллеги. И продолжим работать.

Иногда команда git pull выглядит так: git pull origin master *—* получить данные с сервера, а точнее, с ветки master. Все то же самое, что и git pull)

Имейте ввиду: git pull  !=  git clone. Команда git clone копирует весь репозиторий целиком, как есть - со всеми его файлами. Команда  git pull копирует разницу между удаленным и локальным репозиторием.

**Когда не проходит git push**

Допустим, мы сделали новый коммит, пушим его (хотим отправить на сервер), а гит в ответ выдает такое:



Т.е., коммит отклонен, наш пуш не прошел. Разбираемся, почему так бывает.

Git устроен так, что локально мы можем коммитить сколько угодно. Но прежде чем отправить свои коммиты на сервер, нужно подтянуть новые коммиты с сервера *—*те самые, которые успели сделать наши коллеги. То есть сделать git pull.

Когда мы делаем git push, Git сначала проверяет *—* нет ли на сервере новых коммитов. Если они есть, то мы видим сообщение *—* git push rejected. Значит, нам нужно сначала сделать git pull, а затем снова запушить. Если же при попытке пуша новых коммитов на сервере нет, то git push пройдет сразу и отправит наши коммиты на сервер.

Бывает так, что когда мы сделали локальный коммит, а после этого подтянули новые коммиты с сервера, у нас в терминале появляется мердж-коммит: "Merge branch 'master' of github.com:наш\_аккаунт". Не пугаемся, т.к. мердж-коммит не несет смысловой информации, кроме самого факта мерджа. Просто удаляем его, подтягивая изменения командой git pull с флажком --rebase:

git pull --rebase origin master

То есть всегда пулльтесь с флажком ребейза *—* git pull --rebase origin master. При этом наш локальный коммит окажется "поверх" нового коммита с сервера, а мердж-коммита не будет. И не забудем после этого запушить свой коммит на сервер.

**git fetch**

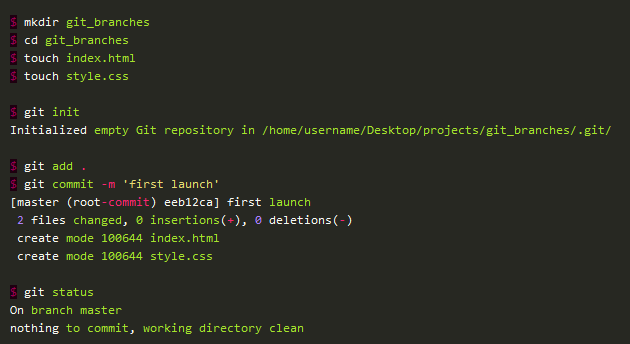
Есть еще одна команда для получения изменений с удаленного репозитория *—* git fetch, которая считается «безопасным» вариантом. Она загружает удаленное содержимое, но не обновляет рабочее состояние локального репозитория, оставляя текущую работу нетронутой.

**Важно:**

Когда мы работаем в одиночку, то удаленный репозиторий нужен только для сохранения резервной копии, туда мы будем только пушить. Но при работе в команде следует пушить коммиты почаще (коллеги быстрее получат доступ к новым изменениям) и пуллиться почаще (почаще получать свежие изменения).

**Немного о том, что такое branch в Git. И немного про алиасы.**

Допустим, мы создали папку git\_branches, внутри которой два файла *—* index.html и style.css. Затем инициализируем Git-репозиторий, добавим созданные файлы под версионный контроль Git — все это мы уже умеем:



В выводе команды git status мы видим, что находимся "On master branch". Это означает, что при инициализации Git-репозитория была автоматически создана бранча с именем main (раньше ее называли master), и на данный момент мы находимся в этой ветке. Вообще ветка main (master) в Git — это не какая-то особенная ветка, это точно такая же, как и все остальные ветки в локальном репозитории. Она существует лишь потому, что ее создает команда git init, а большинство людей просто не меняют ее название. А вот origin master — это ветка на удаленном репозитории.  
Nothing to commit, working directory clean *—*в этой строчке говорится о том, что в ветке нечего коммитить, и рабочий каталог чист.

В данном курсе main = master))

Допустим, мы хотим кардинально отредактировать эти файлы, потом проиндексировать\зафиксировать их. Но понимаем, что изменения такого рода могут нарушить работу уже существующего проекта. Поэтому, чтобы обезопасить себя (=подстраховать!), мы и создаем точную копию существующего проекта, в которую и переходим для дальнейших кардинальных модификаций). Ветка в репозитории Git *— э*то разные пути развития проекта, по сути разные последовательности коммитов.

Для этого создадим новую бранчу с любым именем (например, second), которая будет точной копией ветки master на этом этапе. Ветка создается командой git branch. Для переключения между ветками используется команда git checkout имя\_ветки\_на\_которую\_переключаемся. А можно вообще использовать команду, которая одновременно создаст ветку и переключит нас на нее:

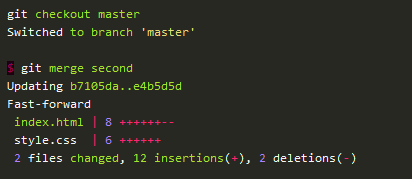
git checkout -b имя\_ветки, где "b" — расшифровывается как branch. Дадим гиту эту команду:

https://ucarecdn.com/721e7d93-97ab-4f11-86dc-bfa5da42eabb/

И гит нам подсказывает, что теперь мы переключились (Switched to) на ветку second. Вот теперь заходим в наши файлы index.html и style.css, кардинально их редактируем, проиндексируем и зафиксируем. Наша ветка second имеет 2 отредактированных файла, в то время, как на ветке master эти 2 файла еще не отредактированы. На master гит "запечатлил" тот момент, когда мы совершили переход из ветки master на ветку second. Если мы сейчас перепрыгнем на ветку master с помощью git checkout -b master то мы вернемся в последнее зафиксированное гитом состояние этой ветки (master).

Хорошо. Посмотрели еще раз наши кардинальные изменения, поняли, что они нам нравятся, хотим добавить их в первоначальную ветку master, чтобы потом продолжить развитие проекта уже с этого места. То, что мы хотим сделать — называется "слияние" веток master и second, оно проводится командой git merge имя\_ветки\_в\_которой\_делали\_изменения — "смерживание" веток. В команде слияния указываем имя той ветки, которую хотим слить с текущей, в которой я нахожусь на данный момент.

Прыгаем с помощью checkout на ветку master, т.к. **мы должны находиться в той ветке, в которую вносим изменения из другой ветки**. А затем сделаем само слияние:



Теперь ветка master имеет в себе все, что есть и в ветке second, последняя нам уже не нужна, поэтому ее можно удалить командой git branch -d имя\_ветки:

https://ucarecdn.com/acb960ab-28da-446b-a941-fe191b5ec27f/

Если посмотрим на вывод команды git hist, то увидим, что у нас осталась только одна ветка master.

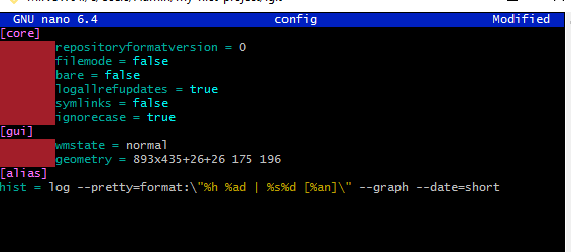
Команду git hist мы пока не трогали, значит пришло и ее время. Она используется для вывода **красивого** лога коммитов, позволяя избежать ввода очень длинной команды log. Т.е. вместо того, чтобы каждый раз писать

git log --pretty=format:" "

мы просто создадим алиас hist в файле config в папке git. Алиасы — это псевдонимы команд, они эквивалентны исходным командам. В кавычках укажем те параметры, которые хотим видеть в логе — \"хэш\_коммита   дата\_коммита  |  комментарий\_и\_имена ссылок   [имя\_автора]\":

[alias] hist = log --pretty=format:\"%h %ad | %s%d [%an]\" --graph --date=short

Зайдем в нашем каталоге git в файл config с помощью команды nano config, добавим  алиас для команды hist:



после чего выведем команду git hist:

https://ucarecdn.com/a3d9b78c-50ba-4a83-a663-e10b28390df9/

Видим, что наш лог упорядочен в соответствии с теми настройками, что мы задали в config.

Мы создали алиас в локальном файле конфигурации. Создавать и менять псевдонимы (алиасы) можно как в глобальных файлах ( глобальный файл конфигурации расположен в $HOME/.gitconfig) так и в локальных файлах конфигурации (путь к локальному файлу конфигурации начинается с активного репозитория Git в /.git/config).

В системах Linux такой файл находится в домашнем каталоге пользователя в файле /.gitconfig.

Также легко можно настроить алиасы для любой команды с помощью git config. Подробнее можно почитать в книге про Git: [https://git-scm.com/book/ru/v2/Основы-Git-псевдонимы-в-Git](https://git-scm.com/book/ru/v2/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-Git-%D0%9F%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D0%BD%D0%B8%D0%BC%D1%8B-%D0%B2-Git)

Посмотреть на пример лога, созданного при помощи алиасов можно здесь <https://skazkin.su/sozdanie-komandy-git-hist-krasivyj-log-kommitov/>

Существуют разные варианты произношения: сделать мерж/мёрж, "смержить"/"смёржить"

**Что такое "форкнуть" репозиторий**

Мы знаем, что для работы над проектом сначала создается удаленный репозиторий *—*origin (мастер-репозиторий). Это общий репозиторий с рабочей версией проекта, содержащий эталонный код. Обычно этот официальный репозиторий создает человек, который в команде занимается поддержкой проекта, и в его репозитории находится официальный проект — "золотой код". Каждый программист клонирует себе на компьютер полную копию мастер-репозитория и работает в ней.

Создать такую копию можно сразу на Github при помощи опции "Fork". Получим копию мастер-репозитория, которая называется "форк". Fork *—* это вcего-навсего копия репозитория, это тоже самое, что branch в Git. Только на гитхабе такой branch называется форком.

Допустим, мы работаем в небольшой компании разработчиком. С нами работают еще люди, в числе которых и человек, занимающийся поддержкой проекта. У нас у всех свои публичные репозитории, а в репозитории этого человека (назовем его PM'ом) как раз и находится официальный проект. Чтобы начать работу над проектом, мы должны сначала создать форк репозитория PM'а на гитхабе: заходим в свой аккаунт, переходим к репозиторию PM'а и жмем кнопку *Fork*. Указав имя и описание для репозитория, создаваемого с помощью форка, мы получаем копию серверной части проекта. Потом клонируем его на свой локальный компьютер и работаем в нем.

Команда git clone автоматически создает удаленный репозиторий origin, который указывает на наш репозиторий, созданный с помощью форка. Прежде чем писать какой бы то ни было код, мы должны создать новую ветку для функции. В ней мы и будем работать: безнаказанно менять код и не бояться что-то сломать в основной версии проекта. А потом отправим PM'у запрос на слияние нашей ветки (нашего измененного репозитория) с официальным репозиторием. Такой запрос называется pull request.

Если PM'у наши изменения понравятся, то он/она внесет их в оригинальный репозиторий *—* примет запрос и выполнит слияние.

Само слово fork в переводе означает *ответвление*. Для того, чтобы воспользоваться fork, нужно иметь свою собственную учетную запись на GitHub, что уже у нас есть)  
Давайте на практике выполним хотя бы один fork на GitHub. Понадобится Is-thirteen))

Зачем  Is-thirteen?

Практически в любом языке программирования проверка на равенство числа 13 выглядит примерно так:

if (num == 13)

  //true

else

  //false

Кажется, что все просто. Но нет, не для того программисты создали целый проект —  **Is-thirteen** — проект на GitHub, состоящий из сотен строк кода, 92 участников и имеющий свою политику изменений кода. И все для того, чтобы проверить является ли число равным 13. Не верите? Заходите: <https://github.com/jezen/is-thirteen>

В репозитории is-thirteen есть несколько разделов, например, issues, где люди могут задавать вопросы, писать комментарии и запрашивать новые функции. Типа таких:

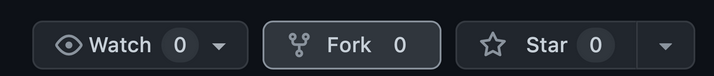
- Add support for XIII (добавьте поддержку для XIII) <https://github.com/jezen/is-thirteen/issues/82>

- But what is 13? (А что такое 13?) <https://github.com/jezen/is-thirteen/issues/730>

И много еще чего веселенького)

Имейте ввиду, если вам когда-либо понадобится такой пакет программного обеспечения как is-thirteen, просто не забудьте обратить внимание на предупреждение в файле проекта [README](https://github.com/jezen/is-thirteen/blob/master/README.md) ([https://github.com/jezen/is-thirteen/blob/master/README.md)](https://github.com/jezen/is-thirteen/blob/master/README.md): «ПОЖАЛУЙСТА ПРОЧИТАЙТЕ ИСХОДНЫЙ КОД, потому что мы торопились и многое сломали».

Так вот, заходим на GitHub под своей учетной записью и переходим по ссылке в репозиторий is-thirteen: <https://github.com/jezen/is-thirteen> . В верхнем правом углу нажимаем на кнопку Fork.



Вот, у нас теперь копия оригинального репозитория.

Приглядитесь к “главной” надписи — она изменилась с "jezen/is-thirteen" на "наше-имя/is-thirteen". Можем делать с ним все, что хотим — просто пользоваться или же вносить свои собственные изменения. Если захотим поделиться изменениями, то можем отправить пользователям Is-thirteen запрос на слияние pull request нашего репозитория is-thirteen с удаленным репозиторием jezen/is-thirteen. Если все ОК — наш запрос могут и принять)))

Мы можем удалить репозиторий наше-имя/is-thirteen: заходим в него, находим строку Settings в правом нижнем углу окна браузера, в списке. Открываем ее, сразу же копируем имя репозитория в поле Repository name и двигаемся вниз страницы, пока не находим раздел Danger zone. В этой опасной зоне находим подраздел Delete this repository и жмем на кнопку Delete. Читаем предупреждением об удалении репозитория. В окне запроса имени удаляемого репозитория вводим (или вставляем из буфера обмена) название нашего репозитория. Подтверждаем.

Мы молодцы!

Запоминаем, что Fork нужен для тех же целей, что и branch в Git. С помощью него создается точная копия оригинального репозитория, только на сервисе GitHub. В копии репозитория можно вносить свои кардинальные изменения, редактировать файлы или удалять папки. Как только все изменения будут внесены, то можно поделиться ими *—* отправить pull request.

**Ветвление в Git**

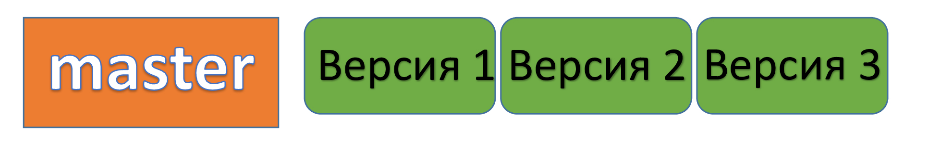
Давайте теперь подробно разбираться, что из себя представляет механизм ветвления в системе Git. Вспомним, что branch – ветка, альтернативная реальность в истории нашего кода, которая начинается с конкретного коммита.

Допустим, на проекте работает много людей и представим, что все они работают с основной веткой – master (main) – золотой веткой, нашим эталоном. Сделали первый коммит (добавили к main первую версию), потом второй (добавили вторую), потом третий и т.д. И в шестом коммите сделали что-то такое, после чего код в золотой ветке перестал работать. Мы возвращаемся назад в пятую версию, ищем там то, что мешает сделать шестую, подпортив тем самым вид нашей золотой ветки. Это не хорошо, т.к. наша золотая ветка, содержащая эталонный код, должна работать всегда, это то, что в продакшен.

Поэтому есть негласное правило – никогда не делать коммиты сразу в master, кроме первого коммита. А дальше мы делаем ветвление и программируем уже в этой копии. Работаем там, на новой ветке, и если видим, что все у нас хорошо – запрашиваем разрешение соединить нашу ветку с главной. Т.е. в master по сути делаются только merge, а не коммиты.

Изучаем, как надо работать правильно)

Например, разрабатываем сайт, нашли в нашем коде какой-то баг, который надо пофиксить.



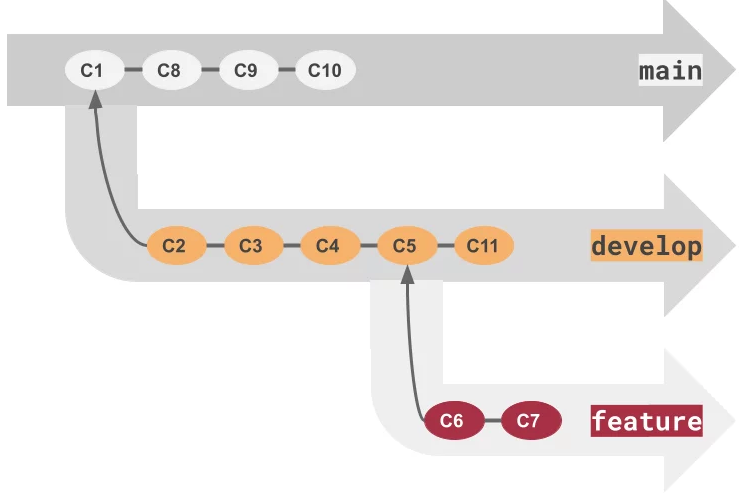
Лезть и редактировать наш master – нельзя, это эталонный код, неважно, что там и обнаружен этот баг – человек, который будет его там исправлять может случайно нарушить все и весь сайт вообще работать не будет. Поэтому этот человек делает себе отдельную ветку и называет ее fixing, например.

Ветка – стандартный и эффективный механизм для работы с нашим кодом. При ее создании в нее копируются все коммиты, которые были в master. Вот в ней и будет работать наш человек – редактировать, тестировать и т.д. Главное, что он не трогает master. Он все пофиксил в своей ветке, проверил, что все хорошо, сохранил версию и потом делает запрос на соединение с веткой master. Запрос одобряется, в золотой ветке теперь следующая версия, где все работает.



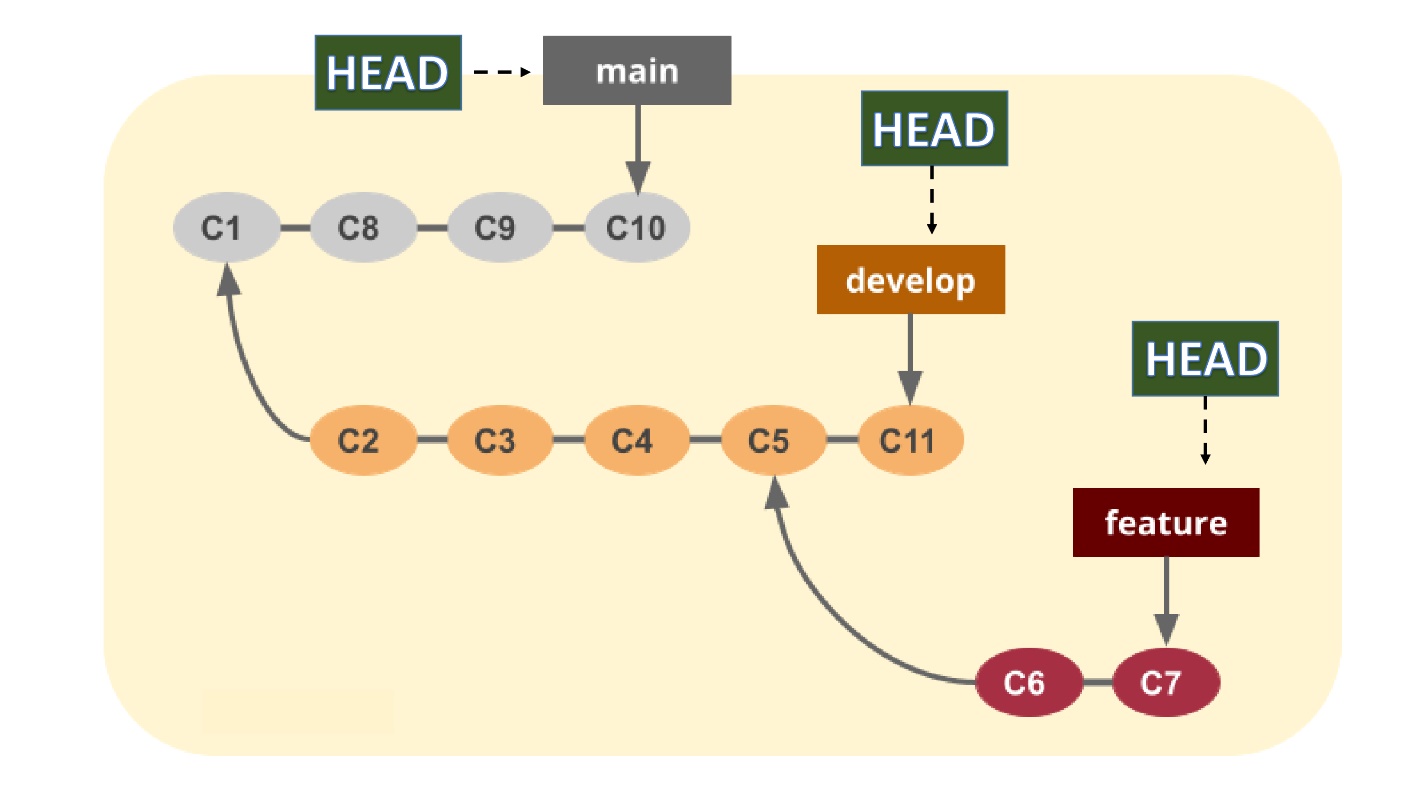
А ветку fixing теперь можно и удалить.

По логике ветка– это последовательность коммитов, которые идут друг за другом в хронологической последовательности:



Мы видим 3 ветки – main, develop и feature, каждая их которых – это поток коммитов в хронологической последовательности, причем эти потоки не пересекаются, работа в ветках идет параллельно.

С точки зрения внутренней реализации Git, ветка – это ссылка на последний коммит в этой ветке:



Ветка main – это последовательность коммитов C1, C8, C9 и C10, она указывает на коммит, который является последним в "потоке" коммитов в данной ветке – Git поставил себе указатель HEAD над коммитом C10. Для ветки developпоследним коммитом является C11, а последним коммитом ветки feature является C7.

Указатель HEAD– это ссылка на определенный коммит или ветку, некоторая метка, которую использует Git чтобы определять, в каком состоянии находится рабочая копия (рабочий каталог, т.е. все файлы репозитория, за исключением директории .git/). На какой коммит указывает HEAD– из того коммита и загружаются файлы в рабочий каталог. Кроме HEAD у гита есть еще указатели:

- ORIG\_HEAD – указывает на тот же коммит, на который указывал HEADдо передвижения назад. Он появляется, когда мы вручную передвигаем HEAD на какой-нибудь НЕ последний коммит, и позволяет вернуться на хронологически последний коммит без проблем;

- пользовательские указатели – создаваемые самими пользователями. Например, мы можем создать указатель Version-1.2.1, и он будет указывать на коммит, в котором хранится версия 1.2.1 нашего проекта. Это довольно удобно, поскольку так мы можем переключаться на коммит с той или иной версией, не запоминая его хэш.

Каждый раз, когда мы создаем новый коммит, Git автоматически перемещает указатель этой веткина последний коммит. Тем не менее, мы можем создавать свои ветки, переключаться между ними и перемещать указатель ветки между коммитами самостоятельно.

**Создание веток**

Допустим, у нас есть проект. Мы ведем разработку последовательно – у нас есть серия изменений, которые мы фиксируем с помощью коммита, и если все нам нравится – передаем эти изменения на сервер в удаленный репозиторий. Но иногда нам хочется создать альтернативную историю с какого-то конкретного коммита – свою ветку, в которой будем шкодить. Тогда с помощью git checkout -b test мы создадим ветку test, например.

Команда git checkout умеет создавать ветки и сразу переключаться на них. Это намного удобнее, чем сначала создавать ветку (git branch имя\_ветки), а потом на нее прыгать (git checkout имя\_ветки). Поэтому данный способ является более предпочтительным, так как задействует только одну команду git checkoutсо специальным ключом -b.

Итак, создаем ветку: git branch имя\_ветки переходим на нее git checkout имя\_ветки

или все сразу:

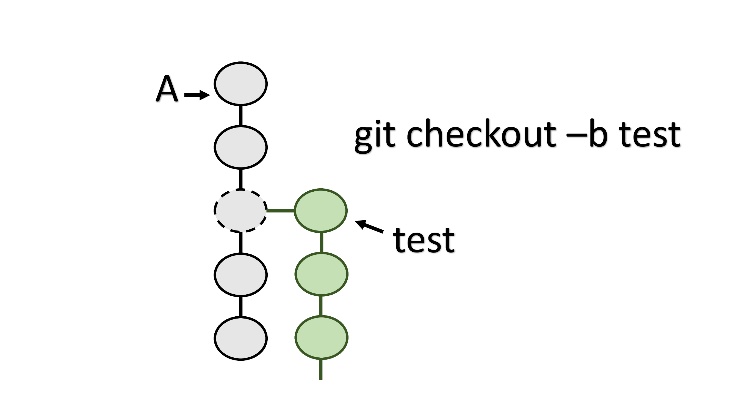
git checkout -b имя\_ветки

Внутри Git при создании ветки выполняет следующие шаги:

1) в папке .git\refs\heads он создает новый файл и дает ему имя ветки, которую мы хотим создать (при условии, что такого файла не существует). Например, если мы выполним команду git branch test, то создастся файл .git\refs\heads\test.

2) в этот файл записывается хэш текущего коммита. С него ветка и начнется.

Важно понимать, что как только мы создали новую ветку, она указывает на тот же коммит, что и основная ветка, и HEAD. Визуально это выглядит так:



Git создаст тот же самый коммит, с которого пойдет новая ветка test. Мы можем параллельно вносить изменения и в ветке А и в ветке test, причем эти изменения не будут пересекаться.

Переключиться на любую локальную ветку можно с помощью команды git checkout имя-ветки.

Когда мы ее задаем, Git сначала проверит, существует ли указанная ветка, затем переключит указатель HEADна новую ветку и меняет рабочую копию так, чтобы она соответствовала этой новой ветке. Ведь рабочие копии на разных ветках могут отличаться, поскольку ветки работают независимо друг от друга. На этом этапе Git просто смотрит на последний сделанный в данной ветке коммит, после чего восстанавливает структуру и файлы в рабочей копии:

- добавляет в рабочую копию файлы, которые есть в новом коммите, но которых нет в коммите, с которого мы переключились

- удаляет файлы, которые были в коммите, с которого мы переключились, но которых нет в новом коммите

- игнорирует (но не удаляет) файлы, которых нет ни в одном из этих двух коммитов.

Выполнив эти три условия для всех файлов в рабочей копии, Git полностью переключит нас на нужную ветку.

**Перемещение по веткам**

Итак, постепенно приходит осознание того, что работая с Git, мы постоянно создаем и перемещаемся по веткам. Когда мы работаем с большим количеством веток, можно легко забыть имя нужной, а без имени ветки переключится на нее не получится. Для таких ситуаций существует команда просмотра списка веток – уже знакомая нам git branch, но с другими ключами. Чтобы посмотреть, на какой ветке мы работаем, применяем команду git branch. Она покажет локальные ветки, которые есть в текущей рабочей копии (рабочем каталоге). git branch -r  покажет удаленные ветки на сервере, а если вообще хотим посмотреть все существующие ветки и в локальном и удаленном репозитории, то к команде git branch добавляем ключ -a (означает, что все - all): git branch -a.

Что делать, если на текущей ветке есть незакоммиченные изменения,

а нам надо переключиться с нее и поработать в другой?

Допустим, у нас 2 ветки – test и A – и мы работаем в ветке test. Делаем в ней были какие-то изменения, пока их не коммитим – а вдруг не пригодятся? И в какой-то момент нам надо переключиться с ветки test на ветку A, но коммиты на ветке test мы все еще сделать не готовы. Если мы сейчас просто переключимся на другую ветку, то Git просто проигнорирует все наши изменения на ветке test, когда мы с ветки A снова вернемся на ветку test, наши изменения останутся в том же состоянии, в котором и были до этого.

Согласитесь, такое поведение очень неудобно, ведь мы можем запутаться в структуре собственного проекта. Поэтому есть два варианта решения проблемы:

1) добавляем все изменения в коммит и меняем ветку

2) откладываем изменения, не добавляя их в коммит, воспользовавшись командой git stash. Эта команда позволяет запомнить изменения, которые были на текущей ветке и мы можем спокойно перейти на другую ветку. git stash спрячет наши изменения в некоторый внутренний стек Git.  Каждое применение git stashсоздает отдельный коммит, хэш которого и записывается в стек. Поэтому с помощью этой команды можно неограниченное количество раз откладывать изменения.

Рассмотрим подробнее варианты применения git stash.

1) Если мы не добавляли файл в индекс (а как мы помним, по умолчанию гит не берет не проиндексированные файлы), тогда нам нужно добавить сначала файл в индекс или к команде git stash добавить -u (полный вариант  --include untracked). Такая запись означает, что мы добавляем в stash все файлы, которые имеем, включая не проиндексированные.

https://ucarecdn.com/8cd6e5c1-1aa6-45c8-8dcd-590f799d89e7/

Ответом на такую команду будет сообщение о том, что рабочий каталог сохранен на этом коммите:

https://ucarecdn.com/b3325078-5853-4ad2-9b8e-edcc54cff0b7/

Таким образом, прежде чем переключиться на другую ветку, мы сохранили изменения в текущей. Поработали на другой ветке,  вернулись в ветку test и хотим наши изменения извлечь, чтобы продолжить работать с ними дальше. Просмотреть список всех отложенных изменений можно командой git stash list, а чтобы достать изменения из стека и вернуть их в рабочую копию, нужно использовать команду git stash pop или git stash apply. Первая команда забирает данные из stash и удаляет их оттуда, а вторая забирает, но не удаляет их из stash.

2) Если мы добавляли файл в индекс, то достаточно просто команды git stash

Таким образом, мы можем откладывать наши изменения перед переходом на новую ветку, а затем доставать их обратно. Кстати, если мы захотим удалить все файлы, которые находятся в stash, то используем git stash clear.

Команды урока:

git checkout -b имя-ветки – создает ветку с указанным именем и автоматически переключает на нее

git checkout имя-ветки – автоматически переключает на любую локальную ветку

git stash – откладывает изменения, не добавляя их в коммит

git stash list – показывает список всех отложенных изменений

git stash pop – забирает данные из stash и удаляет их

git stash apply – забирает данные из stash, но не удаляет их

git stash clear – удаляет все файлы, которые находятся в stash

git branch -a – показывает все существующие ветки и в локальном и удаленном репозитории

git branch -r  – показывает удаленные ветки на сервере

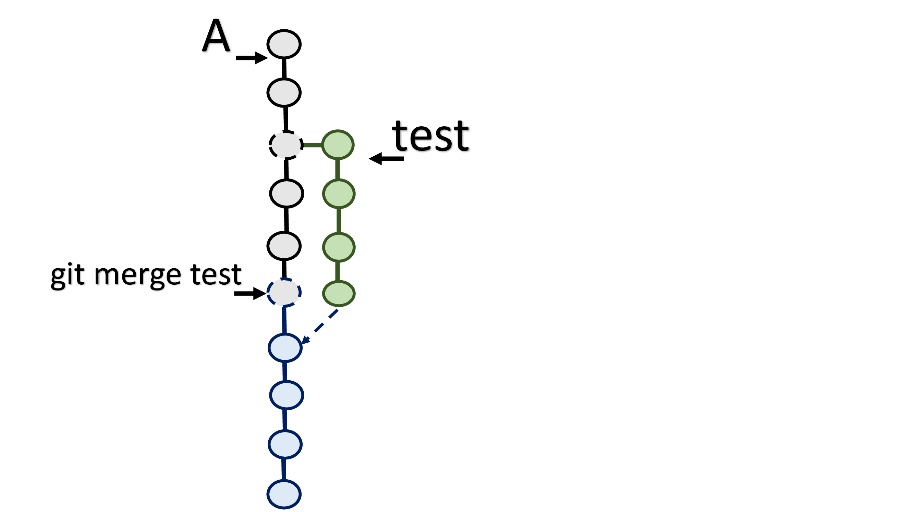
**Слияние веток**

Итак, мы поработали в ветке test, поняли, что получилась классная фича, хотим включить ее в основной продукт – главную ветку main (на рисунке она названа A). Делаем git merge test.

Важно! В merge указывает ту ветку, на которой работаем и которую хотим соединить с главной. Мы работали в test, значит ее и указываем. Команда git merge автоматически смерживает указанную ветку с main.

Но если мы не проверили, на какой ветке сидим, например, в этот момент мы находились на ветке A, а не на ветке test (но мёржим-то ветку test), то произойдет следующее. Git создаст ревизию – ссылку на объект git, по сути еще один коммит. В литературе встречается термин "ревизия", который означает то же самое, что и "коммит", хотя коммит – это фиксация одного из объектов git (блоб, дерево, теги), а ревизия является способом ссылки на объект git.

Так вот, эта ревизия будет содержать исходную ветку A и изменения, которые мы сделали в ветке test:



После создания ветку можно отправить на сервер с помощью команды git push origin имя-ветки.  Там она попадает в наш удаленный репозиторий. И потом ее снова нужно смержить, но уже с "золотой" веткой.

Еще момент. Допустим, мы хотим запушить ветку, которую мы создали у себя локально и которой еще нет в удаленном репозитории. Тогда в ответ на команду git push origin гит нам скажет в комментарии, что на гитхабе-то такой бранчи нет еще. А заодно даст подсказку, как это подправить:

git push --set-upstream origin имя\_ветки

На нашем удаленном репозитории на гитхабе создастся эта бранча. Перейдя в репозиторий на гитхабе, мы увидим, что добавилась эта новая ветка с нашим локальным репозиторием и теми файлами, которые мы в нем насоздавали.

На сервере с помощью команды git merge имя-ветки наша ветка соединится с веткой проекта master branch, причем добавлять название origin здесь уже не нужно. Процесс смерживания веток автоматически выполняется с аккаунта на гитхабе, куда мы в свой удаленный репозиторий сначала запушили ветку.

Важно! Команда  git push origin имя\_ветки – отправляет нашу ветку на сервер в удаленный репозиторий проекта на гитхаб, а уже там команда git merge имя\_ветки – сливает нашу ветку с главной веткой в удаленном репозитории.

Мы также можем забрать себе на компьютер ветку с удаленного репозитория командой:

git checkout origin/имя-ветки -b имя-ветки

После работы в локальном репозитории могут оставаться различные ненужные (=неотслеживаемые) файлы. Чтобы удалить все лишнее, используем команду git clean -f  для удаления всех untracked файлов или git clean -fd для удаления и файлов и папок.

Конфликты при слиянии веток

Когда два человека изменяют одни и те же строки в файле или один разработчик удаляет файл, который в это время изменяет другой разработчик, то при дальнейшем слиянии веток появятся конфликты. Т.е., в обеих ветках есть изменения одних и тех же строк. В таком случае, при слиянии этих веток Git не может автоматически определить, какое изменение является правильным. Конфликты затрагивают только того разработчика, который выполняет слияние, остальная часть команды о конфликте не знает. Git помечает файл как конфликтующий и останавливает процесс слияния. В этом случае ответственность за разрешение конфликта несут разработчики, а сами конфликты исправляются вручную: руками поправляются изменения там, где Git не смог это сделать автоматически, затем все собирается в один коммит и пушится на сервер.

Бывает так, что при заливке проекта в пустой репозиторий на гитхаб после ввода команды git push мы можем увидеть следующую ошибку: "fatal: the current branch has no upstream branch". Это ошибка говорит о том, что гит не может решить вопрос аутентификации, потому что мы забыли добавить параметр -u при пуше в первый раз. Поэтому действуем так:

git push -u origin имя\_ветки

В таком случае мы создаем в удаленном репозитории ветку, соответствующую нашей локальной и связываем их:  в remote/branchname будет производиться push в локальную ветку branchname, а из remote/branchname будет производиться pull в локальную ветку branchname.

Опять же, это актуально, только если мы пушим в пустой репозиторий. Если клонировали – то соответствие ветвей уже настроено. А вот как только мы создадим новую локальную ветвь и захотим ее запушить на удаленный репозиторий, нам понадобится ключ -u. Кстати, можно выбрать другое имя для создаваемой ветки на удаленном репозитории, просто вместо названия текущей ветки указать любое другое:

git push -u origin любое\_другое\_имя\_ветки

Иногда используют команду git merge --abort, чтобы откатить состояние до того, что было до запуска слияния.

Удаление веток

Для того, чтобы удалить какую-то ветку, используем команду git branch с ключом -d и прописываем название удаляемой ветки:

git branch -d имя\_ветки

Ветка будет удалена только если она полностью слита с какой-то другой веткой. В противном случае гит не даст нам ее удалить.

Бывает так, что мы создали какую-то ветку, сделали в ней коммит, но потом поняли, что нам не нравится, не хотим ее мержить. Чтобы удалить эту локальную ветку, надо перейти на master (git checkout master) и стереть ее там:

git branch -D имя\_ветки

В отличии от ключа -d ключ -D удалит ветку в любом случае, даже если в ней есть изменения, которые мы можем потерять. Останется только master.

А чтобы удалить ветку с сервера, надо выполнить пуш ветки, только с флагом --delete:

git push origin --delete имя\_ветки

Кстати, ведь мы в проекте не одни, в нем будут постоянно появляться новые ветки. Чтобы их увидеть, сначала нужно выполнить команду git fetch, которая сходит на сервер и проверит, что там есть нового. Эта команда не создает локальные ветки, она просто подтягивает информацию о них. Например, команда: git fetch origin some\_branch скачает нам ветку some\_branch из удаленного репозитория, если таковой у нас нет локально.

Команды урока

git branch – по умолчанию выводит список локальных веток. С ключами -r, -a можно вывести, соответственно, либо только удаленные ветки, либо все ветки.

git checkout – переключает пользователя на другую ветку

git branch -d имя\_ветки – удаляет ветку, которая полностью слита с какой-то другой веткой

git branch -D имя\_ветки – удаляет любую локальную ветку

git push origin --delete имя\_ветки – удаляет ветку с сервера

git fetch – стягивает информацию о ветках на сервере к нам на локальную машину.

git clean -f  – удаляет все неотслеживаемые файлы

git clean -fd – удаляет неотслеживаемые файлы и папки

git push -u имя\_ветки – создает в удаленном репозитории ветку, соответствующую локальной ветке и обе связывает.

*Поработаем с ветками*

Создадим отдельный проект – папку branch и проинициализируем ее:

mkdir branch

cd branch

git init .

Теперь это наш локальный репозиторий, в котором создадим файл index.html:

echo "some text" > branch.html

Проиндексируем его и закоммитим. Итог: у нас есть главная ветка master с начальным коммитом, мы можем проверить это командой git branch. Как мы помним, работать в master нежелательно, поэтому делаем себе ветку first и будем работать с ней:

git checkout -b first

https://ucarecdn.com/6330bcd3-4bee-4aa6-879b-90d67946cccf/

Проверим, что мы на ветке first с помощью git status, а также проверим, сколько всего у нас веток с помощью git branch:

https://ucarecdn.com/69467a01-ee9a-4f94-8880-daf0c303e2b3/

Создадим еще одну ветку second, но теперь уже с помощью git branch:

git branch second

Как вы думаете, на какой ветке сейчас мы находимся? А вот и не на second), мы все еще на first, потому что ветку second мы создали, но не переключились на нее, как это было сделано автоматически в команде git checkout -b. Поэтому переходим:

git checkout second

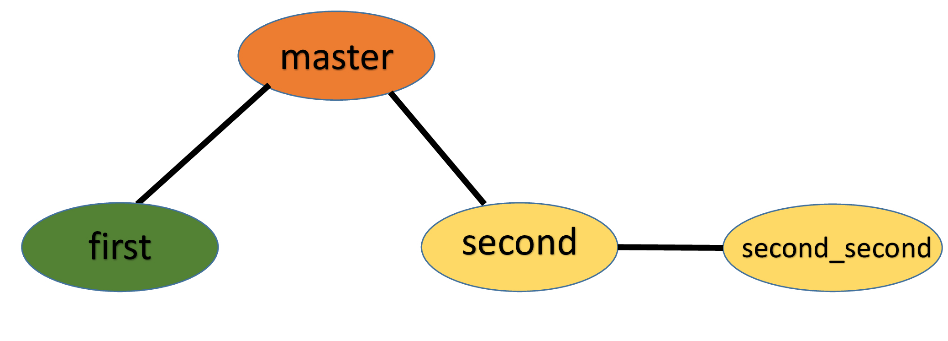
https://ucarecdn.com/9261c1e6-5cf1-43ea-8fc9-182ae60f7f60/

Отлично. Теперь ответвимся еще и от ветки second (на какой ветке находимся, от такой и "ответвляемся"):

git checkout -b second\_second

Хорошо, добавим немного изменений на каждую ветку: ff.txt в папку first, ss.html в папку second и hh.css в папку second\_second, проиндексировать и закоммитить. (Можно сделать это даже не в терминале, а зайти непосредственно в сами папки, насоздавать там эти файлы, а в терминале их проиндексировать и закоммитить, все как положено). Если сейчас мы перейдем на ветку master, то увидим, что в ней нет новых созданных файлов, т.к. они есть только в ветках first и second. Ветка second\_second наследуется от ветки second и имеет все те же файлы, что и последняя (копия ветки second в этот момент была родителем second\_second), плюс свой собственный файл hh.css. Т.е. в ветке second\_second сейчас 2 файла: родительский ss.html  и свой hh.css. А в ветке second есть только один файлик ss.html .

Ветка second\_second – самостоятельная ветка, если мы что-то будем изменять в родительской ветке second, то это никак не повлияет на ветку second\_second, потому что в second\_second попало только то, что на тот момент было в second. Дальнейшая работа в second уже не касается second\_second, в ней таки и останется 2 файлика (если, конечно, мы не создадим в ней новые)



Ветки можно переименовать! Флаг -m в команде git branch позволит это сделать. Например, если мы сейчас находимся на ветке second\_second, то задав гиту команду git branch -m new\_second, наша second\_second станет new\_second. Если мы хотим переименовать другую ветку, то после флага -m сначала пишем название изменяемой ветки, а потом через пробел ее новое название:

git branch -m first new\_first

Теперь наша first стала new\_first.

Удалим ветку new\_second с помощью git branch -d new\_second. Сможем сейчас?

Нет! Мы же ее не смержили – не влили эти изменения в second. Когда мы отделили от ветки second другую ветку, то работали в ней – создали там файл hh.css. А если есть изменения в дочерних ветках, то гит требует объединить их в родительскую ветку, ту, от которой мы и ответвились. Если мы все равно хотим ее удалить, не делая merge, то используем команду git branch -D new\_second. Теперь удалим)

Закинем все в удаленный репозиторий: заходим в аккаунт на GitHub, создаем новый репозиторий (кнопка New) app, например, делаем публичным (галочка в Public) и подтверждаем нажатием на зеленую кнопку Create repository. Возвращаемся в терминал и подвязываем текущий репозиторий branch и удаленный app:

git remote add origin [https://github.com/название\_вашего\_аккаунта/app.git](https://github.com/%D0%BD%D0%B0%D0%B7%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B2%D0%B0%D1%88%D0%B5%D0%B3%D0%BE_%D0%B0%D0%BA%D0%BA%D0%B0%D1%83%D0%BD%D1%82%D0%B0/app.git)

Теперь запушим туда наши изменения – наши ветки master, new\_first, second. Чтобы запушить какую-то одну ветку – достаточно перейти на нее и с нее дать гиту команду git push origin HEAD.

Итак, все запушилось. Идем в удаленный репозиторий и обновляем страницу. Видим ветку master, т.к. за один раз пушится только та ветка, из которой мы это сделали. В нашем случае мы сделали push из ветки master, теперь сделаем из ветки first – переходим на нее командой checkout и пушим изменения из нее:

git push origin HEAD

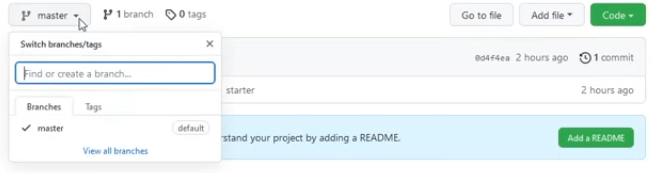
Снова обновим страницу аккаунта и видим, что теперь у нас две ветки:

https://ucarecdn.com/71ecaab7-d099-4199-ba99-d18d24cee061/

Давайте удалим ветку first прямо с удаленного репозитория? Тогда в терминале прописываем:

git push origin -d first

Этой командой мы обращаемся уже не к локальному репозиторию branch, а к удаленному app и говорим, что изменения должны быть применены к нему конкретно. Смотрим на удаленном репозитории:

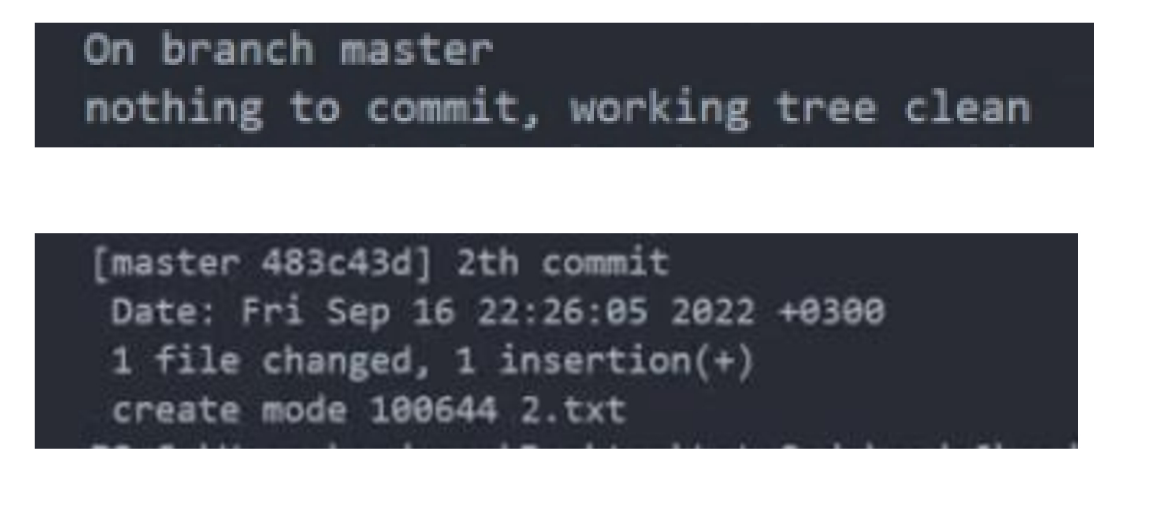


Кое-что о команде git cherry-pick

git cherry-pick позволяет брать коммит с одной ветки и повторять его на другой ветке. Создадим от ветки master другую ветку new, в которой создадим 3 файла (1.txt, 2.txt и 3.txt) и закоммитим эти изменения:



Из всех этих коммитов нам понравился второй, с сообщением "2th commit", решаем его взять из этой ветки new и добавить на ветку master. Для этого берем номер этого коммита (его хэш). Раньше мы рассматривали, как посмотреть хэш коммита – командой git log отобразим список всех коммитов. Смотрим, копируем номер нужного нам коммита и переходим на ветку master, т.к. цель – добавить коммит в нее. С master вызываем команду git cherry-pick и вставим скопированный номер коммита.



На рисунке: мы сначала проверили с помощью git status, что находимся действительно на master, затем после команды git cherry-pick номер\_коммита видим, что к master добавился наш коммит.

Как проходит pull request и merge

C помощью запросов pull мы уведомляем участников команды о том, что поработали на конкретной ветке и подготовили некий функционал, который требуется проверить, а затем выполнить слияние с главной веткой. Т.е., создавая **pull request**(пулл-реквест), мы всего лишь **просим** другого разработчика (например, человека, занимающегося поддержкой проекта) **забрать** ветку из нашего репозитория в его репозиторий. Поэтому для создания пулл-реквеста необходимо указать 4 параметра: исходный репозиторий, исходную ветку, репозиторий назначения и ветку назначения. Давайте по порядку рассмотрим практический пример использования pull-request в рабочем процессе с форками (о форках мы говорили на 4 уроке модуля 4.2).:

1) мы поработали в отдельной ветке в своем локальном репозитории

2) затем отправляем свою ветку в удаленный репозиторий на GitHub (или другой сервис) командой push

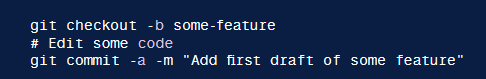
3) там же на гитхабе создаем запрос pull

4) наши модификации проверяются, обсуждаются, если нужно — возвращаются нам обратно для правки

5) человек, занимающийся поддержкой проекта, смерживает (сливает) нашу ветку в официальный репозиторий и закрывает запрос pull.

У нас на проекте есть свой публичный репозиторий — исходный, соответственно и своя ветка — исходная ветка — та, что содержит наши изменения. Когда мы предлагаем слить нашу ветку с основной базой кода, то официальный репозиторий проекта является репозиторием назначения (целевой репозиторий), а целевой веткой — главная ветка main, "золотая ветка". Кроме того, мы можем посылать пулл-реквесты нашим коллегам за пределами официального репозитория проекта. Например, если мы работаем над какой-то веткой вместе с другим разработчиком, то можем создать пулл-реквест, указав в качестве целевого репозитория не официальный репозиторий проекта, а публичный репозиторий нашего коллеги. Тогда в качестве исходной ветки и ветки назначения мы просто укажем одну и ту же ветку (в которой работаем).

Итак, мы форкаем репозиторий PM'а на гитхабе, клонируем его на свой локальный компьютер, видим там главную ветку, например она называется some-branch, создаем от нее свою ветку some-feature и работаем в ней:



Делаем сколько угодно коммитов на нашей ветке, а закончив свою задачу, помещаем ветку some-feature в собственный репозиторий на гитхабе (не в официальный репозиторий проекта) с помощью простой команды git push:

git push origin some-branch

Теперь наши изменения будут доступны человеку, занимающемуся поддержкой проекта (или любым другим участникам, которым может понадобиться доступ к этим изменениям). После добавления своей ветки в гитхаб, мы из своего же аккаунта создаем пулл-реквест, перейдя в свой репозиторий, созданный с помощью форка, и нажав на кнопку *Pull request* в верхнем правом углу. Отобразится форма, в которой наш репозиторий автоматически будет указан в качестве исходного. Нам нужно только указать исходную ветку (some-branch), целевой репозиторий и ветку назначения — главную ветку main, а также ввести заголовок и описание запроса pull. После нажатия на кнопку Сreate pull request нашему PM'у придет уведомление. Он видит все созданные другими членами команды пулл-реквесты в своем репозитории. Нажав на наш пулл-реквест, он увидит описание пулл-реквеста, историю коммитов нашей ветки и все изменения в пулл-реквесте.

Если PM считает функционал готовым к слиянию с проектом, ему достаточно нажать кнопку *Merge* (Слияние), чтобы одобрить запрос pull и выполнить слияние нашего функционала со своей веткой main.

Команда этого занятия:

***git cherry-pick 40\_символьный\_номер\_коммита*** – позволяет брать коммит с одной ветки и повторять его на другой ветке