

Работа с системами контроля версий на примере Git Hub

Введение

Система контроля версий (СКВ) — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. Для примеров будем использовать исходные коды программ, но на самом деле под версионный контроль можно поместить файлы практически любого типа.

Если вы графический или веб-дизайнер и хотели бы хранить каждую версию изображения или макета — а этого вам наверняка хочется — то пользоваться системой контроля версий будет очень мудрым решением. СКВ даёт возможность возвращать отдельные файлы к прежнему виду, возвращать к прежнему состоянию весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внёс в код какую-то ошибку, и многое другое. Вообще, если, пользуясь СКВ, вы всё испортите или потеряете файлы, всё можно будет легко восстановить.

Локальные системы контроля версий

Многие предпочитают контролировать версии, просто копируя файлы в другой каталог (как правило, добавляя текущую дату к названию каталога). Такой подход очень распространён, потому что прост, но он и чаще даёт сбои. Очень легко забыть, что ты не в том каталоге, и случайно изменить не тот файл, либо скопировать файлы не туда, куда хотел, и затереть нужные файлы. Чтобы решить эту проблему, программисты уже давно разработали локальные СКВ с простой базой данных, в которой хранятся все изменения нужных файлов (см. рисунок 1).

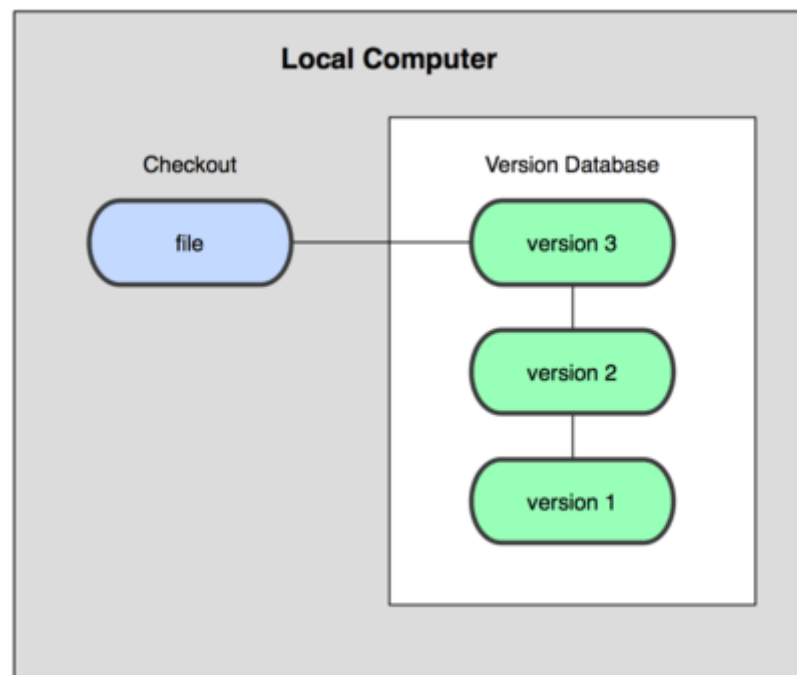


Рис. 1 Схема локальной СКВ.

Одной из наиболее популярных СКВ такого типа является rcs, которая до сих пор устанавливается на многие компьютеры. Даже в современной операционной системе Mac OS X утилита rcs устанавливается вместе с Developer Tools. Эта утилита основана на работе с наборами патчей между парами версий (патч — файл, описывающий различие между файлами), которые хранятся в специальном формате на диске. Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи.

Централизованные системы контроля версий

Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы контроля версий (ЦСКВ). В таких системах, например, CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий (см. рисунок 2).

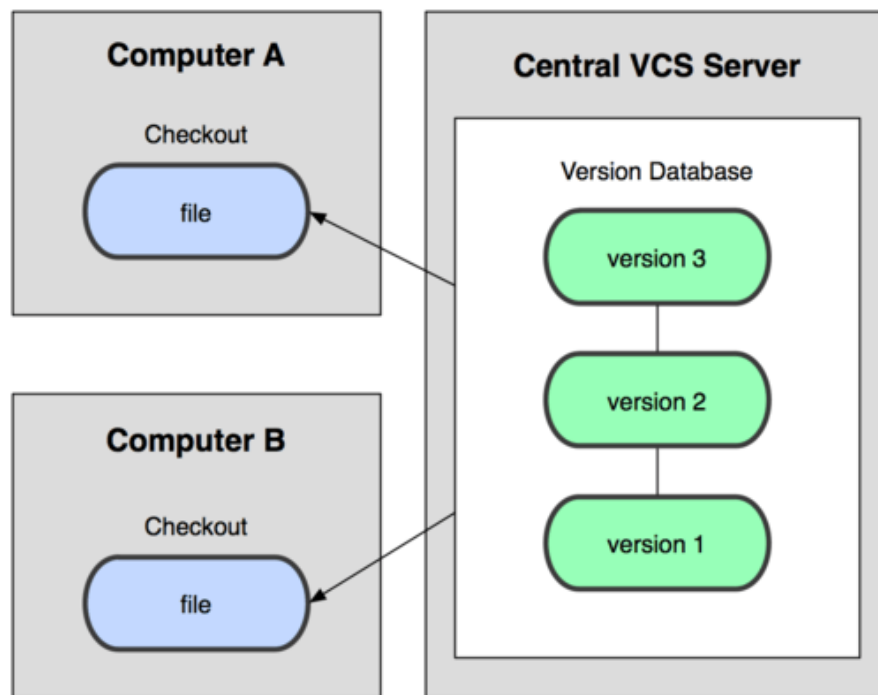


Рис. 2 Схема централизованного контроля версий.

Такой подход имеет множество преимуществ, особенно над локальными СКВ. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСКВ намного легче, чем локальные базы на каждом клиенте.

Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей. Локальные системы контроля версий подвержены той же проблеме: если вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределённые системы контроля версий

И в этой ситуации в игру вступают распределённые системы контроля версий (РСКВ). В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных (см. рисунок 3).

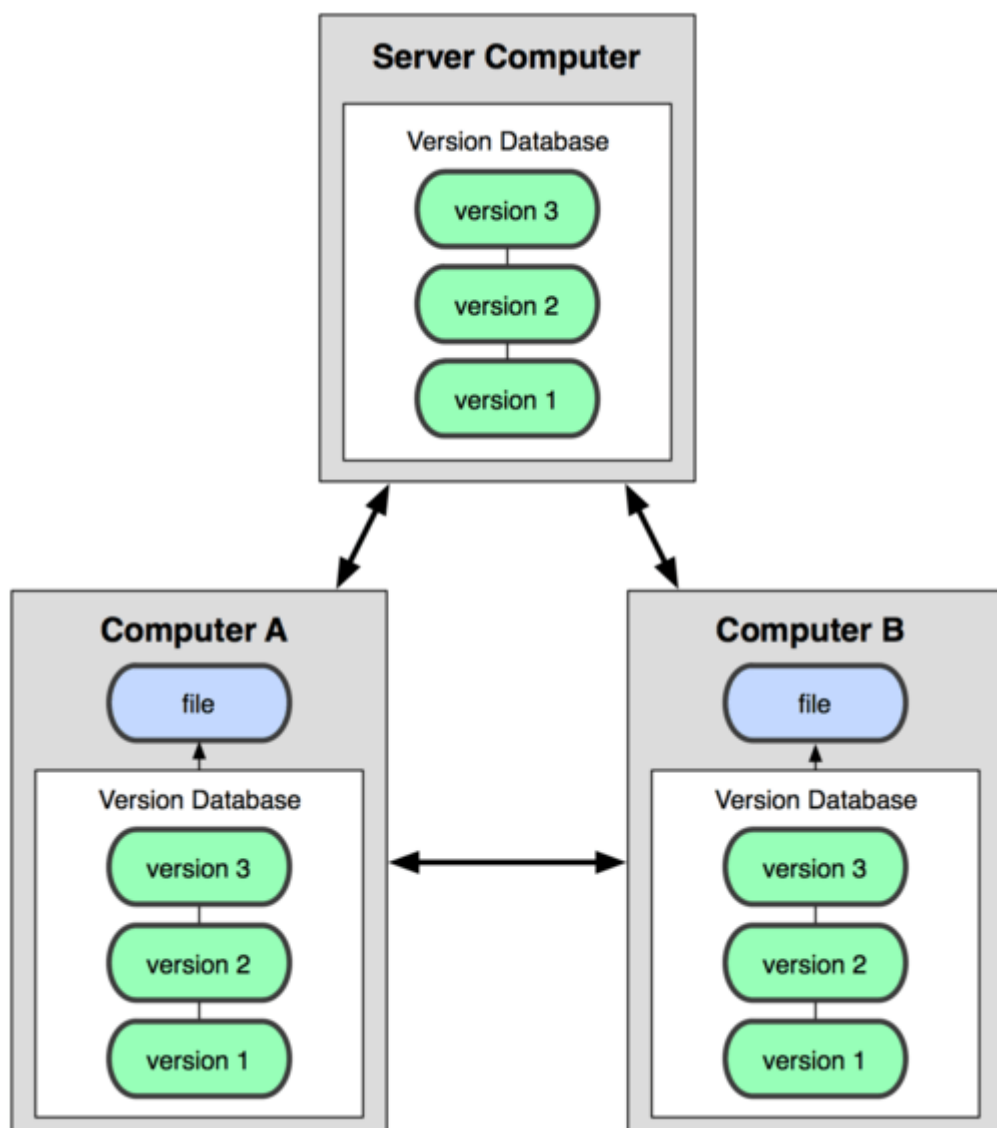


Рис. 3 Схема распределённой системы контроля версий.

Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями, таким образом, можно одновременно работать

по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

Краткая история Git

Как и многие замечательные вещи, Git начинался с, в некотором роде, разрушения во имя созидания и жарких споров. Ядро Linux — действительно очень большой открытый проект. Большую часть существования ядра Linux (1991-2002) изменения к нему распространялись в виде патчей и заархивированных файлов. В 2002 году проект перешёл на проприетарную РСКВ BitKeeper.

В 2005 году отношения между сообществом разработчиков ядра Linux и компанией, разрабатывавшей BitKeeper, испортились, и право бесплатного пользования продуктом было отменено. Это подтолкнуло разработчиков Linux (и в частности Линуса Торвальдса, создателя Linux) разработать собственную систему, основываясь на опыте, полученном за время использования BitKeeper. Основные требования к новой системе были следующими:

- Скорость
- Простота дизайна
- Поддержка нелинейной разработки (тысячи параллельных веток)
- Полная распределённость
- Возможность эффективной работы с такими большими проектами, как ядро Linux (как по скорости, так и по размеру данных)

С момента рождения в 2005 году Git развивался и эволюционировал, становясь проще и удобнее в использовании, сохраняя при этом свои первоначальные качества. Он невероятно быстр, очень эффективен для больших проектов, а также обладает превосходной системой ветвления для нелинейной разработки.

Основы Git

Так что же такое Git в двух словах? Эту часть важно усвоить, поскольку если вы поймёте, что такое Git, и каковы принципы его работы, вам будет гораздо проще пользоваться им эффективно. Изучая Git, постарайтесь освободиться от всего, что вы знали о других СКВ или постоянно пользуетесь, таких как Subversion или Perforce. В Git'e совсем не такие понятия об информации и работе с ней как в других системах, хотя пользовательский интерфейс очень похож. Знание этих различий защитит вас от путаницы при использовании Git'a.

Слепки вместо патчей

Главное отличие Git'a от любых других СКВ (например, Subversion и ей подобных) — это то, как Git смотрит на свои данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar и другие) относятся к хранимым данным как к набору файлов и изменений, сделанных для каждого из этих файлов во времени, как показано на рисунке 4.

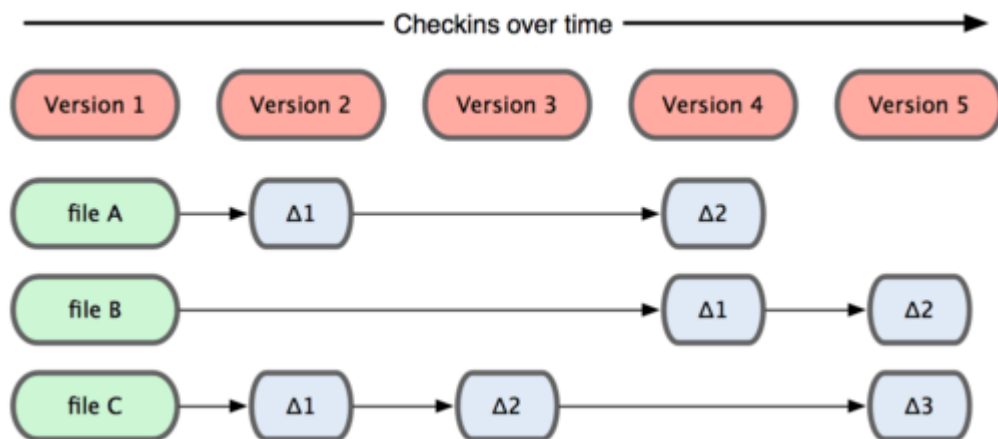


Рис. 4 Другие системы хранят данные как изменения к базовой версии для каждого файла.

Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает

ссылку на ранее сохранённый файл. То, как Git подходит к хранению данных, похоже на рисунок 5.

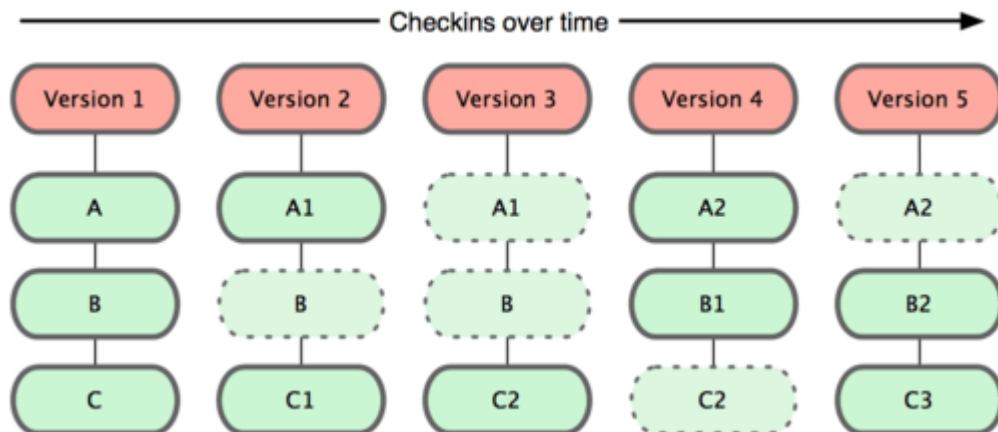


Рис. 5 Git хранит данные как слепки состояний проекта во времени.

Это важное отличие Git'a от практически всех других систем контроля версий. Из-за него Git вынужден пересмотреть практически все аспекты контроля версий, которые другие системы переняли от своих предшественниц. Git больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто СКВ.

Почти все операции — локальные

Для совершения большинства операций в Git'e необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. Если вы пользовались централизованными системами, где практически на каждую операцию накладывается сетевая задержка, вы, возможно, подумаете, что боги наделили Git неземной силой. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными.

К примеру, чтобы показать историю проекта, Git'у не нужно скачивать её с сервера, он просто читает её прямо из вашего локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, Git может взять файл месячной давности и вычислить разницу на

месте, вместо того чтобы запрашивать разницу у СКВ-сервера или качать с него старую версию файла и делать локальное сравнение.

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к Сети или VPN. Если вы в самолёте или в поезде и хотите немного поработать, можно спокойно делать коммиты, а затем отправить их, как только станет доступна сеть. Если вы пришли домой, а VPN-клиент не работает, всё равно можно продолжать работать. Во многих других системах это невозможно или же крайне неудобно. Например, используя Perforce, вы мало что можете сделать без соединения с сервером. Работая с Subversion и CVS, вы можете редактировать файлы, но сохранить изменения в вашу базу данных нельзя (потому что она отключена от репозитория). Вроде ничего серьёзного, но потом вы удивитесь, насколько это меняет дело.

Git следит за целостностью данных

Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git'a и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

Механизм, используемый Git'ом для вычисления контрольных сумм, называется SHA-1 хешем. Это строка из 40 шестнадцатеричных символов (0-9 и a-f), вычисляемая в Git'e на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

B6DABFD326E06D8896CBE83B3BD52645CCE69C28

Работая с Git'ом, вы будете встречать эти хеши повсюду, поскольку он их очень широко использует. Фактически, в своей базе данных Git сохраняет всё не по именам файлов, а по хешам их содержимого.

Чаще всего данные в Git только добавляются

Практически все действия, которые вы совершаете в Git'e, только добавляют данные в базу. Очень сложно заставить систему удалить данные или сделать что-то неотменяемое. Можно, как и в любой другой СКВ, потерять данные, которые вы ещё не сохранили, но как только они зафиксированы, их очень сложно потерять, особенно если вы регулярно отправляете изменения в другой репозиторий.

Поэтому пользоваться Git'ом — удовольствие, потому что можно экспериментировать, не боясь что-то серьёзно поломать

Три состояния

Теперь внимание. Это самое важное, что нужно помнить про Git, если вы хотите, чтобы дальше изучение шло гладко. В Git'e файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. "Зафиксированный" значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проектах, использующих Git, есть три части: каталог Git'a (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area).

Каталог Git'a — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git'a, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git'a и помещаются на диск для того, чтобы вы их просматривали и редактировали.

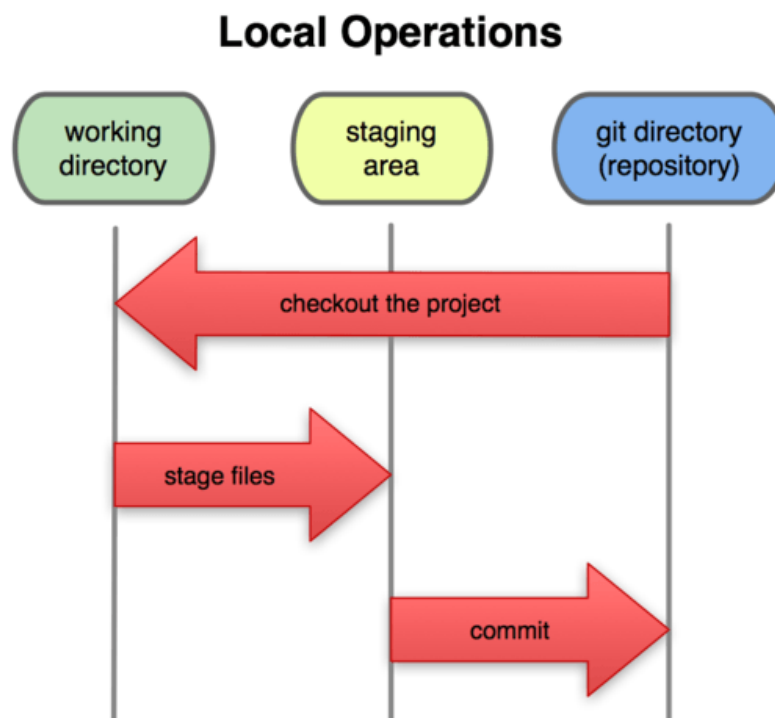


Рис. 6 Рабочий каталог, область подготовленных файлов, каталог Git'a.

Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git'a, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

Стандартный рабочий процесс с использованием Git'a выглядит примерно так:

1. Вы вносите изменения в файлы в своём рабочем каталоге.
2. Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
3. Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог Git'a на постоянное хранение.

Если рабочая версия файла совпадает с версией в каталоге Git'a, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым.

Индивидуальные задания

Создание учетной записи на github.com

Зайдите на github.com

Зарегистрируйтесь на нем и подтвердите регистрацию на своей почте.

Join GitHub · GitHub

← → ↻ https://github.com/join

Join GitHub

The best way to design, build, and ship software.

Step 1:
Set up your account

Step 2:
Choose your subscription

Step 3:
Tailor your experience

Create your personal account

There were problems creating your account.

Username *

Login can't be blank

Email can't be blank

Password can't be blank

very account

You'll love GitHub

- Unlimited public repositories
- Unlimited private repositories
- ✓ Limitless collaboration
- ✓ Frictionless development
- ✓ Open source community

В процессе регистрации могут появиться вопросы: чем занимаетесь, какие ЯП знаете, какой аккаунт хотите. Все шаги можно пропустить, выбираете бесплатный аккаунт. **Перед тем, как создавать репозиторий, перейдите в Settings и изменить название главной ветки с «main» на «master» и нажмите Update.**

Search or jump to...

Pull requests Issues Marketplace Explore

idmtest
Your personal account

Go to your personal profile

Signed in as idmtest

Set status

Your profile

Your repositories

Your codespaces

Your projects

Your stars

Your gists

Upgrade

Feature preview

Help

Settings

Sign out

Account settings

Profile

Account

Appearance New

Account security

Billing & plans

Security log

Security & analysis

Emails

Notifications

SSH and GPG keys

Repositories

Repository default branch

Choose the default branch for your new personal repositories. You might want to change the default name due to different workflows, or because your integrations still require "master" as the default branch name. You can always change the default branch name on individual repositories. [Learn more about default branches.](#)

main Update

Repositories

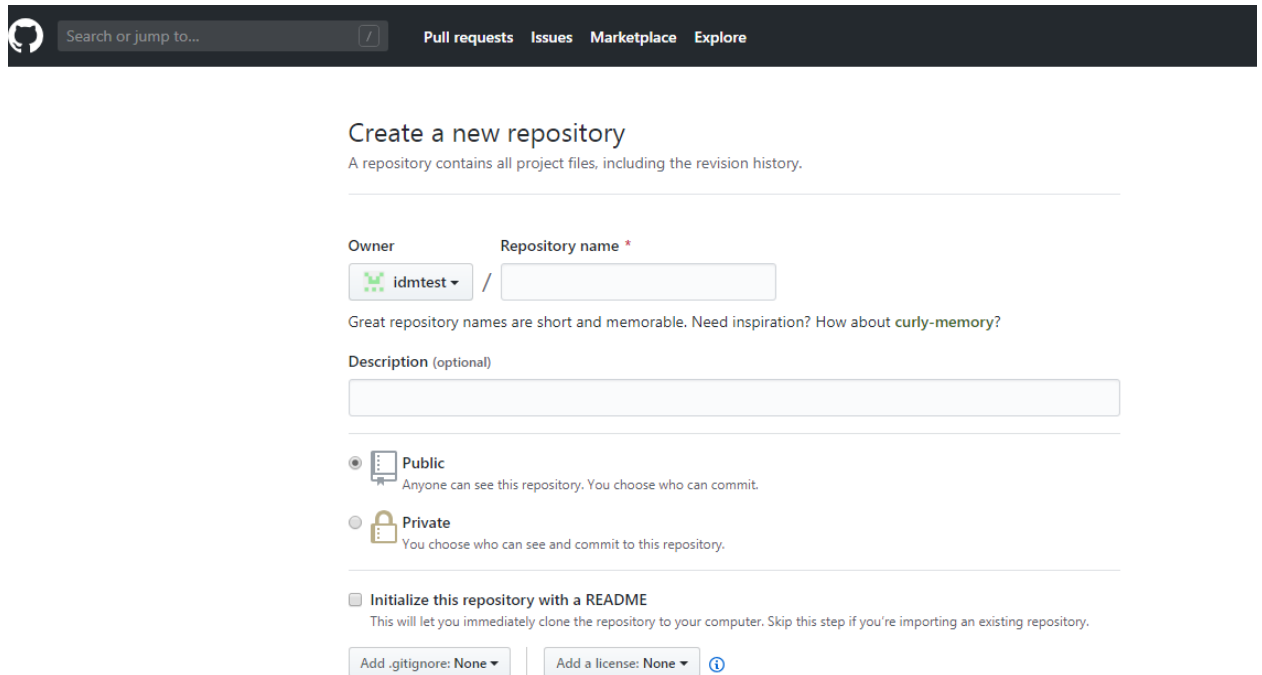
Repositories Deleted repositories

idmtest

idmtest/firstApp 3 KB 0 collaborators

idmtest/idmtest.github.io 2 KB 0 collaborators


Далее создайте новый репозиторий



Search or jump to... 71 [Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)


Create a new repository


A repository contains all project files, including the revision history.

Owner:  idmtest /

Great repository names are short and memorable. Need inspiration? How about curly-memory?

Description (optional):

☒  **Public**
Anyone can see this repository. You choose who can commit.

☐  **Private**
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

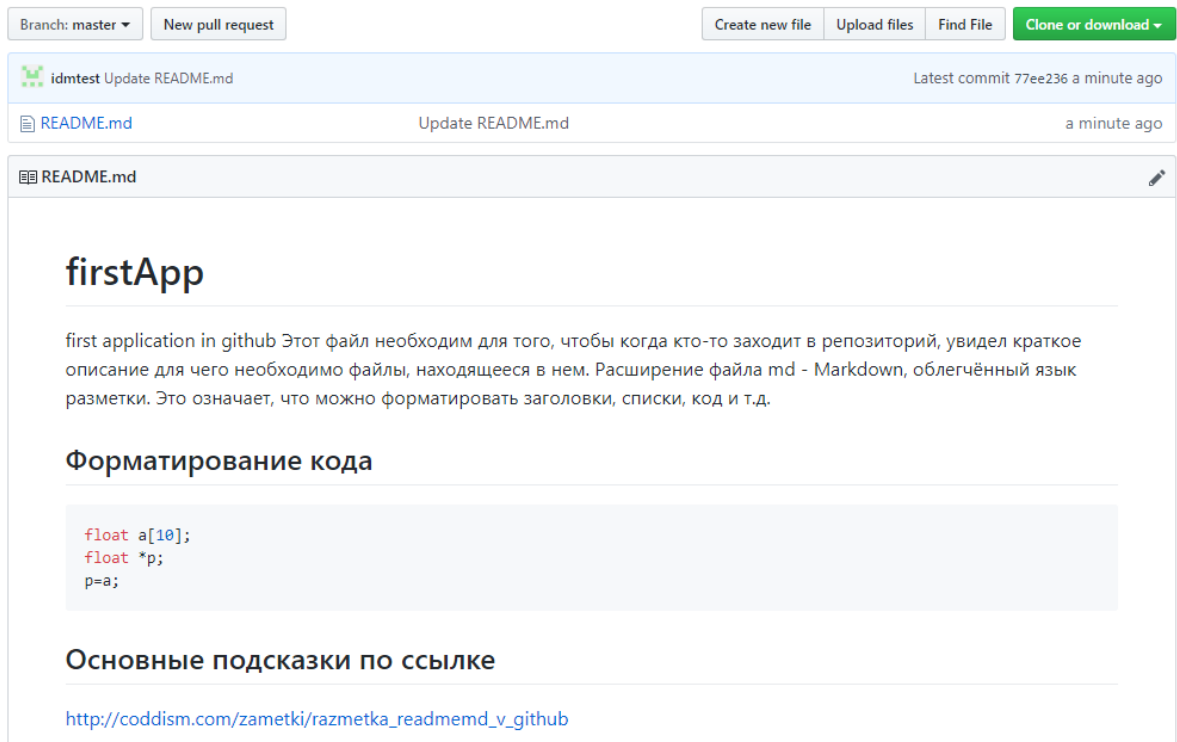
[?](#)

Перед вами появится окно создания нового репозитория.

Придумайте имя репозитория, можете вставить описание и выбрать тип. С недавнего времени github в бесплатной версии стал поддерживать скрытые репозитории, но на лабораторных работах мы будем использовать публичные.

Также можете установить галочку напротив предложения создать файл README – считается хорошим тоном, если кто-то заходит в ваш репозиторий и первым делом ему будет открываться описание вашего проекта.

Итак, мы создали собственный репозиторий, в примере это <https://github.com/idmtest/firstApp>, но у вас будет собственный.

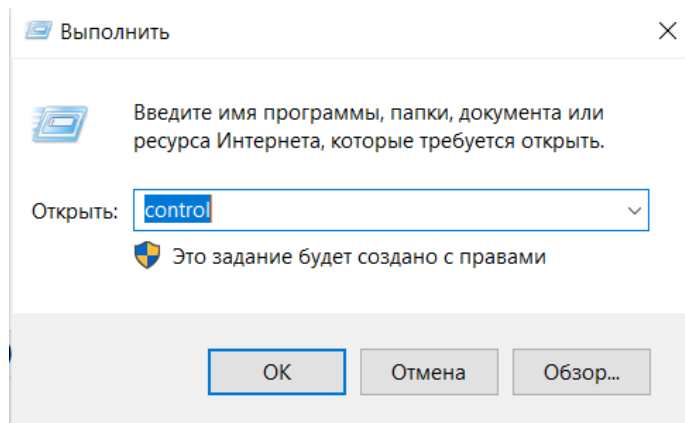


Установка git

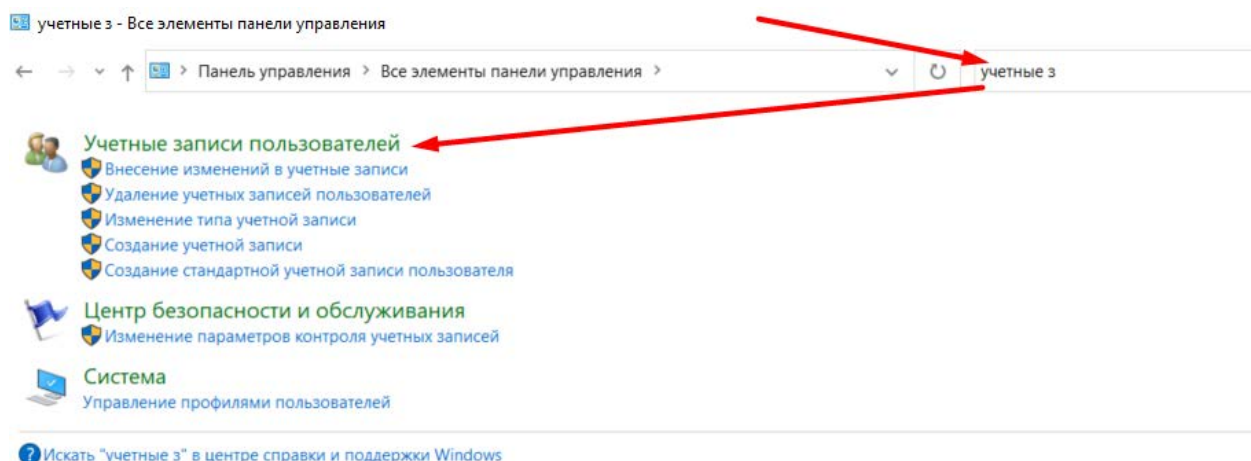
Если у вас не установлен git, скачайте и установите необходимую версию <https://git-scm.com/downloads>. Проверить это просто. Откройте командную строку и напишите git, если появилось кучу команд, то все ок. Если появилась ошибка, то git нужно установить.

Удаление предыдущей записи git на ПК

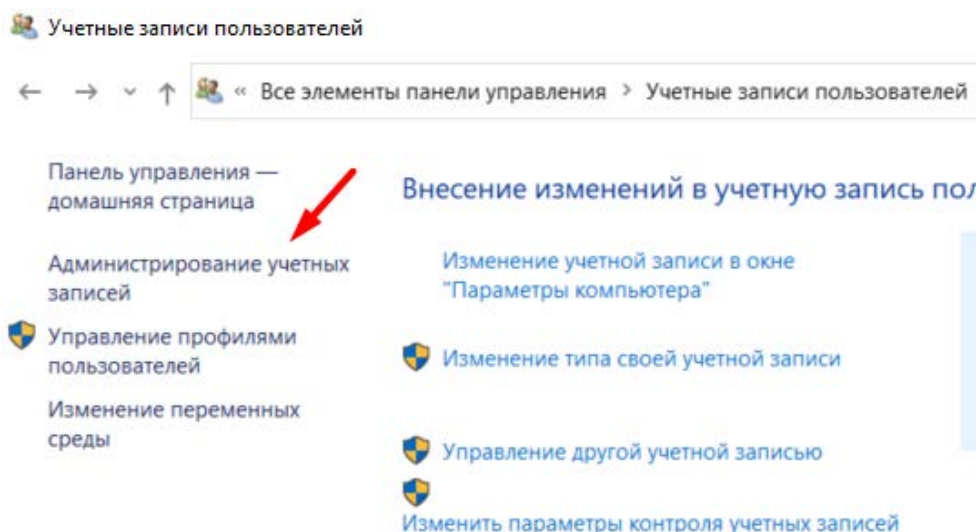
Если вы работаете не на своем ПК и там уже был залогированный Git, то лучше удалить предыдущую запись. Проще это сделать через локальные записи, для этого вам необходимо нажать Win+R и вписать в появившемся окошке «Выполнить» команду control.



Далее начать печатать в строке поиска «Учетные записи» и выбрать «Учетные записи пользователей»



Выбрать «Администрирование учетных записей»



Далее выбрать «Учетные записи Windows» и в «Общие учетные записи» найти поле с github, развернуть его и нажать удалить.

Можно создать несколько таких записей, но лучше ограничиться одной.

Администрирование учетных данных

Просмотр и удаление сохраненных сведений о входе для веб-сайтов, подключенных приложений и сетей.



Учетные данные для Интернета



Учетные данные Windows

Архивация учетных данных Восстановление учетных данных

Учетные данные Windows

Добавить учетные данные Windows

Учетные данные на основе сертификата

Добавить учетные данные на основе сертификата

Нет сертификатов.

Общие учетные данные

Добавить общие учетные данные

GitHub - <https://api.github.com/Sodi1>

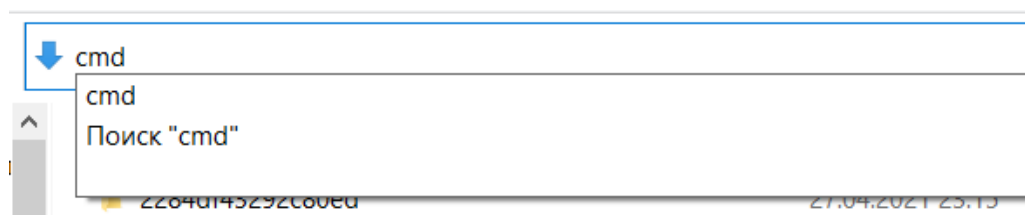
Изменено: 07.09.2021

Создание локального репозитория

Создадим теперь локальную папку, которая будет связана с нашим облачным git хранилищем. В общем случае облачный git может представлять также локальный сервер, объединенный в единую информационную сеть со всеми ПК.

Создайте на диске папку со своей фамилией, лучше в «Документы» или «Загрузка», так будет меньше конфликтов с правом доступа. Создайте в ней файл test.txt и напишите в нем свою фамилию. ***Примечание:** названия лучше писать латиницей, с кириллицей тоже будет работать, но отображаться будет непонятно.

Откройте командную строку «cmd» и перейдите с помощью команды «cd» в созданный вами каталог. Проще всего это сделать, выделив путь в проводнике, написать вместо него cmd и нажать enter.



***Примечание:** При работе с git можно использовать PowerShell, GitBash, Терминал и даже различные IDE, но мы будем работать с обычной командной строкой.

Есть довольно много опций и параметров, с которыми можно играть, но мы настроим самые важные: ваше имя пользователя и адрес электронной почты (лучше указать почту, на которую был зарегистрирован gitHub). Запустите команды (со своими данными):

```
git config --global user.name "My Name"
```

```
git config --global user.email myEmail@example.com
```

ОБЯЗАТЕЛЬНО выполните эти команды, иначе все ваши изменения могут помечаться почтой и именем того, кто раньше представился

Перейдите в созданную вами папку в командной строке используя команду *cd*.

Теперь необходимо проинициализировать эту папку как git репозиторий

Введите в командной строке:

```
git init
```

Командная строка вернет вам сообщение, что проинициализирована пустая git директория

Теперь используя проводник windows создайте в своей папке любой текстовый файл и пропишите в него ваше имя.

Введите команду:

```
git status
```

и вам отобразится сообщение, что есть новый файл, но он не отслеживается. Это значит, что файл новый и система еще не знает, нужно ли следить за изменениями в файле или его можно просто игнорировать. Для того, чтобы начать отслеживать новый файл, нужно его специальным образом объявить.


```
D:\idm_test>git init
Reinitialized existing Git repository in D:/idm_test/.git/

D:\idm_test>cd ..

D:\>cd idm_test

D:\idm_test>git init
Initialized empty Git repository in D:/idm_test/.git/

D:\idm_test>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        test.txt

nothing added to commit but untracked files present (use "git add" to track)
D:\idm_test>
```

В git есть концепция области подготовленных файлов. Можно представить ее как холст, на который наносят изменения, которые нужны в коммите. Сперва он пустой, но затем мы добавляем на него файлы (или части файлов, или даже одиночные строчки) командой `add` и, наконец, коммитим все нужное в репозиторий (создаем слепок нужного нам состояния) командой `commit`.

Фиксация изменений в области заготовленных файлов

Если хотим добавить только один файл, можно написать:

```
git add test.txt
```

Если измененных файлов много, то можно сделать так:

```
git add .
```

Выполнив нужную команду снова посмотрим статус репозитория

```
D:\idm_test>git add .

D:\idm_test>git status
On branch master

No commits yet

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

        new file:   test.txt

D:\idm_test>
```

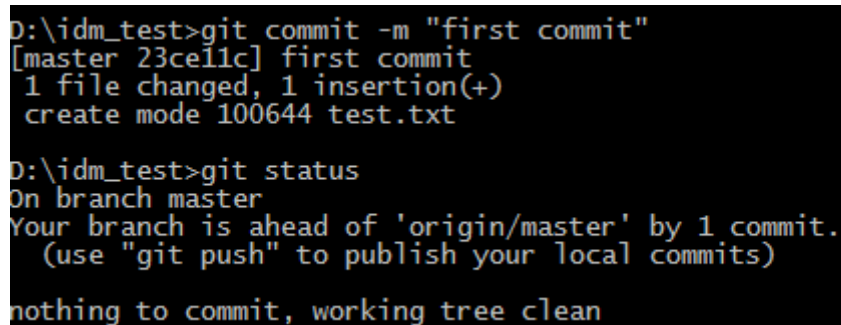
Файл готов к коммиту. Сообщение о состоянии также говорит нам о том, какие изменения относительно файла были проведены в области подготовки — в

данном случае это новый файл, но файлы могут быть модифицированы или удалены.

Теперь закомитим наши файлы. Коммит представляет собой состояние репозитория в определенный момент времени как некий опечаток во времени. Коммит отмечается hash суммой и к которому мы можем в любой момент времени вернуться.

Чтобы зафиксировать изменения, нам нужно хотя бы одно изменение в области подготовки (мы как раз создали новый файл):

```
git commit -m "first commit"
```



```
D:\idm_test>git commit -m "first commit"
[master 23cellc] first commit
1 file changed, 1 insertion(+)
create mode 100644 test.txt

D:\idm_test>git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Команда создаст новый коммит со всеми изменениями из области подготовки. Считается хорошей практикой делать коммиты часто и всегда писать содержательные комментарии. Вызовете `git status` и посмотрите, что поменялось.

Отправка коминат на сервер

Перед тем как перейти к более продвинутым вещам, рассмотрим, как связать наш локальный репозиторий с облачным репозиторием, который мы создали на github.com

Чтобы связать наш локальный репозиторий с репозиторием на GitHub, выполним следующую команду в терминале. Обратите внимание, что нужно обязательно изменить URI репозитория на свой.

```
git remote add origin https://github.com/idmtest/firstApp.git
```

Проект может иметь несколько удаленных репозиториев одновременно. Чтобы их различать, мы дадим им разные имена. Обычно главный репозиторий называется `origin`.

Можно вызвать команду, для просмотра, к какому проекту мы подключены

git remote -v

```
D:\idm_test>git remote -v
origin https://github.com/idmtest/firstApp.git (fetch)
origin https://github.com/idmtest/firstApp.git (push)
```

Если вдруг у вас на этом месте показывается несколько репозиториев, то скорее всего верхние команды привели к запутыванию веток, это можно разрешить следующей командой. Также эта команда может пригодиться, если у вас возникли ошибки, связанные с unrelated-histories.

git pull origin master --allow-unrelated-histories

Запустите *git remote -v* снова

Если репозиторий указан неверно, его можно удалить командой (но только если он указан неверно)

git remote rm origin

После необходимо снова подключить верный репозиторий

Запросим изменения с сервера

В нашем случае у нас есть файл и на сервере и локальном репозитории, поэтому нам вначале надо получить все данные с нашего репозитория, а потом уже их заливать на сервер нашими локальными данными.

Если вы сделали изменения в вашем репозитории (а мы их сделали, создав файл readme), другие пользователи могут скачать изменения при помощи команды *pull*.

git pull origin master

```
D:\idm_test>git pull origin master
remote: Enumerating objects: 18, done.
remote: Counting objects: 100% (18/18), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 18 (delta 4), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (18/18), done.
From https://github.com/idmtest/firstApp
 * branch                master      -> FETCH_HEAD
 * [new branch]          master      -> origin/master
```

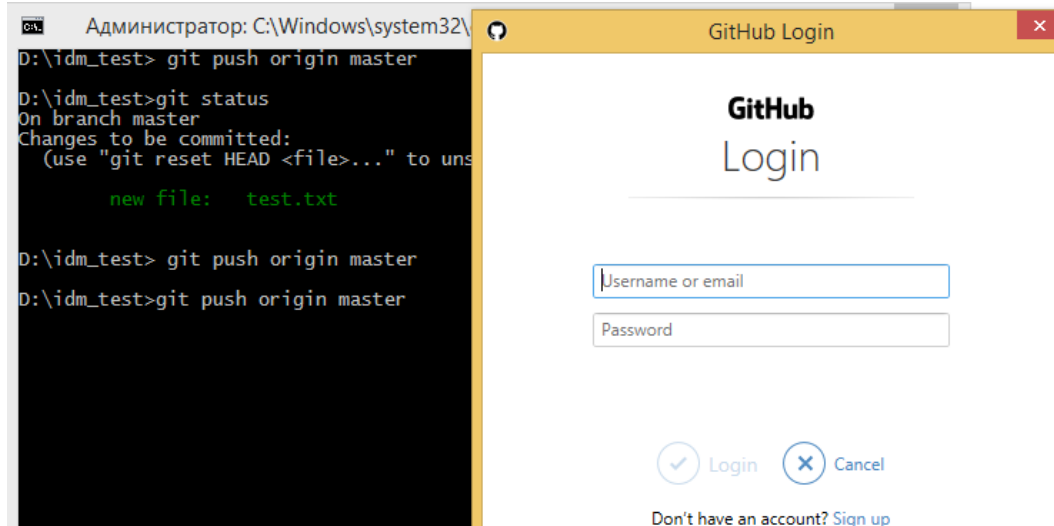
Теперь перешлем локальный коммит на сервер

Теперь отправим коммит на сервер, команда, предназначенная для этого — *push*. Она принимает два параметра: имя удаленного репозитория (мы назвали

наш origin) и ветку, в которую необходимо внести изменения (master — это ветка по умолчанию для всех репозиториях).

git push origin master

У вас должно появиться окно, где нужно ввести авторизоваться на github.com



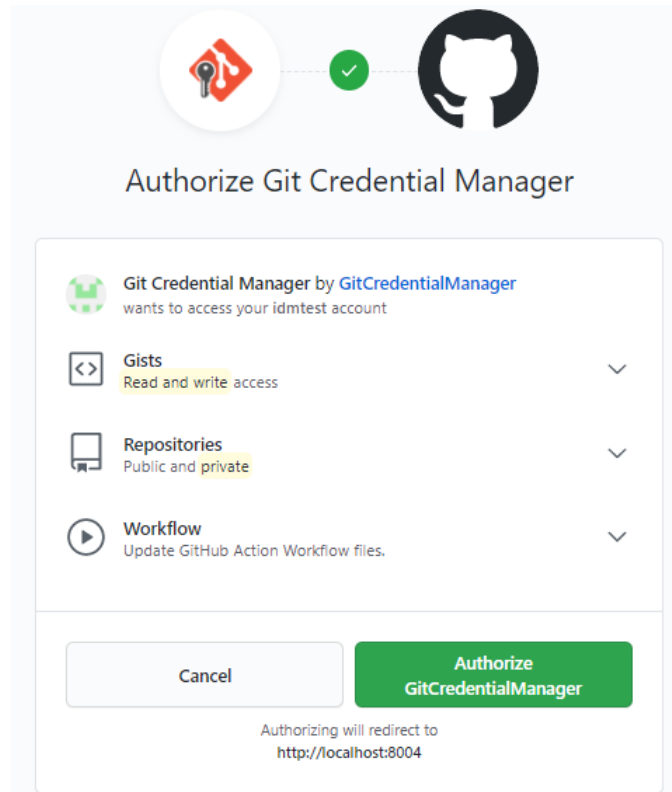
Если появляется ошибка, связанная с unrelated-histories, то воспользуйтесь командой, которая была описана выше.

Окно может выглядеть и по-другому, где предлагается ввести персональный токен или зайти через браузер. Выбираем вариант использования браузера — надо нажать Sign in with your browser



Откроется страница в браузере по умолчанию (если вы использовали для регистрации этот же браузер и не выходили из github.com, то все ок. Если это другой браузер, то залогиньтесь на github под своим аккантом)

На странице нажмите «Authorize GitCredentialManager»



Если все правильно, то изменения отправятся на сервер

```
D:\idm_test>git push origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 269 bytes | 134.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/idmtest/firstApp.git
77ee236..23ce11c master -> master
```

Если вы получаете ошибку 403, скорее всего git не понимает адрес удаленного репозитория, либо неверно введены логин пароль, можно попробовать перелогиниться, вызвав команды

```
git config --global credential.github.com.interactive always
```

теперь снова пытаемся сделать push и вводим новые логин/пароль

```
git config --global credential.github.com.interactive auto
```

Наш комит будет иметь hash 23ce11c1117d7350b49ba3cbb9e2384a706d7249, по которому можно будет вернуться к нему в случае необходимости

Посмотрим наши изменения с использованием команды

```
git log
```

```

D:\idm_test>git log
commit 23ce11c1117d7350b49ba3cbb9e2384a706d7249 (HEAD -> master, origin/master)
Author: idm_test <idm_test@mail.ru>
Date:   Wed Apr 17 21:44:12 2019 +0300

    first commit

commit 77ee2366bbfc613692071e9855511df46e14481a
Author: idmtest <49284736+idmtest@users.noreply.github.com>
Date:   Thu Apr 4 19:18:44 2019 +0200

    Update README.md

commit 2aa1809c27b0291463568ce15c908b67d796bee4
Author: idmtest <49284736+idmtest@users.noreply.github.com>
Date:   Thu Apr 4 19:09:25 2019 +0200

    Update README.md

```

Видны коммиты по изменениям в файле Readme.md и в файле с именем (first commit), чтобы выйти из просмотра лога надо нажать q.

Посмотрим изменения на github.com

7 commits

1 branch

0 releases

1 contributor

Branch: master ▾

New pull request

Create new file

Upload files

Find File

Clone or download ▾

idmtest first commit

Latest commit 23ce11c 15 minutes ago

README.md

Update README.md

13 days ago

test.txt

first commit

15 minutes ago

Если нажать на «commit», то увидим все наши изменения

Branch: master ▾	
Commits on Apr 17, 2019	
first commit idmtest committed 16 minutes ago	23ce11c <>
Commits on Apr 4, 2019	
Update README.md idmtest committed 13 days ago	Verified 77ee236 <>
Update README.md idmtest committed 13 days ago	Verified 2aa1809 <>
Update README.md idmtest committed 13 days ago	Verified 765e571 <>

Создание новой ветки

Во время разработки новой функциональности считается хорошей практикой работать с копией оригинального проекта, которую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения

до тех пор, пока вы не решаете слить изменения вместе. Это происходит по набору причин:

- Уже рабочая, стабильная версия кода сохраняется.
- Различные новые функции могут разрабатываться параллельно разными программистами.
- Разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.
- В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.

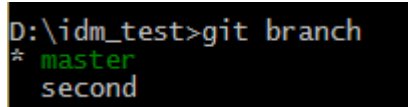
Обычно в проекте создается минимум 2 ветки (branch) – master и ветка для разработки. Ветка master создается сам, а дополнительные ветки создаются по желанию пользователя (локально), заливаются они в удаленный репозиторий уже с учетом доступных прав.

Создадим новую ветку second:

git branch second

Можно посмотреть в какой ветки находимся, используя команду

git branch



```
D:\idm_test>git branch
* master
  second
```

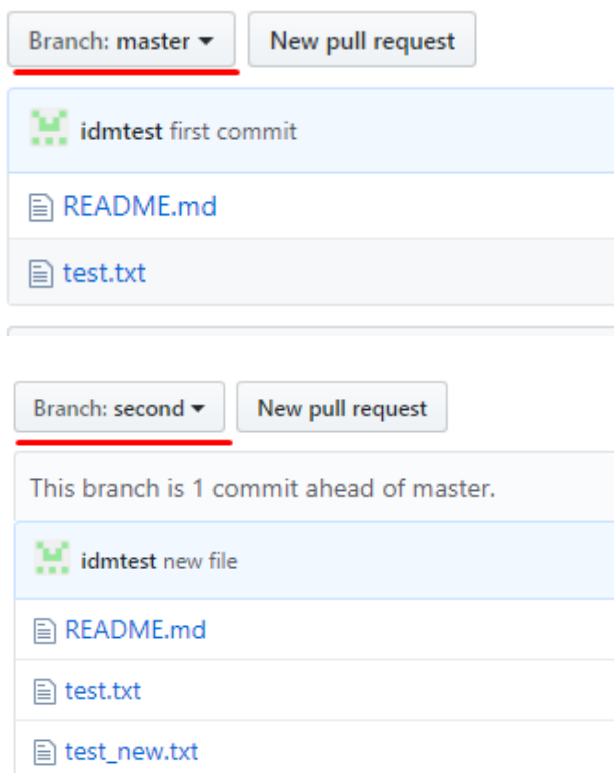
Переключимся на другую ветку, используя команду

git checkout second

Создайте новый файл в нашем локальном репозитории и напишите в нем свою фамилию, добавьте в область подготовленных файлов, закомитьте и отправьте на сервер, только учтите, что у вас теперь ветка не master, а second

```
D:\idm_test>git push origin second
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 301 bytes | 150.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'second' on GitHub by visiting:
remote:   https://github.com/idmtest/firstApp/pull/new/second
remote:
To https://github.com/idmtest/firstApp.git
 * [new branch]      second -> second
```

Зайдите на github и посмотрите, что в ветке master нет вашего файла, а в ветке second он присутствует



Слияние веток

Переключитесь снова на ветку master

Если через проводник windows в папку с проектом, то файла test_new.txt не будет виден, потому что мы переключились обратно на ветку master, в которой такого файла не существует. Чтобы он появился, нужно воспользоваться merge для объединения веток (применения изменений из ветки second к основной версии проекта).

git merge second

Если все прошло без ошибок, можно удалить ветку

git branch -d second

Давайте немного усложним это задание, предположив, что в двух ветках могут быть одинаковые файлы и над ними работают разные разработчики

Создайте ветку с названием newdev

Переключитесь на нее

Добавьте в файл с именем отчество

Зафиксируем изменения

Закоммитьте изменения

Попробуйте переключиться обратно на ветку master и посмотреть файл: в нем только имя, отчество в другой ветке

Объединим ветки

git merge newdev

Откройте файл снова, там есть и имя, и отчество

Переключитесь на ветку newdev и удалите в отчестве несколько букв

Зафиксируйте, закоммите

Переключимся на ветку master, добавьте к отчеству несколько букв

Зафиксируйте, закоммите

Объединим ветки

git merge newdev

Теперь ничего не получится, т.к. есть изменения в обеих ветках

Просмотр изменений и разрешение конфликтов

Нужно разрешить конфликты

```
D:\idm_test>git merge newdev
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Наберите команду, для просмотра изменений

git diff

```
D:\idm_test>git diff
diff --cc test.txt
index 36bc2fb,10b259b..0000000
--- a/test.txt
+++ b/test.txt
@@@ -1,2 -1,2 +1,6 @@@
    Ilya
-   Ivanovichii
-   Ivano
++<<<<<< HEAD
++Ivanovichii
++=====
++Ivano
++>>>>>> newdev
```

Иногда конфликты исправляются автоматически, но обычно с этим приходится разбираться вручную — решать, какой код остается, а какой нужно удалить.

Система не смогла разрешить конфликт автоматически, значит, это придется сделать разработчикам. Приложение отметило строки, содержащие конфликт:

```
<<<<<< HEAD
```

```
Ivanovichii
```

```
=====
```

```
Ivano
```

```
>>>>>> newdev
```

Над разделителем ===== мы видим последний (HEAD) коммит, а под ним — конфликтующий. Таким образом, мы можем увидеть, чем они отличаются и решать, какая версия лучше. Или вовсе написать новую. В этой ситуации мы так и поступим, перепишем все, удалив разделители (HEAD, ==, <<<, >>>), и дадим git понять, что закончили.

Процесс может быть довольно утомительным и может быть очень сложным в больших проектах. Многие разработчики предпочитают использовать для разрешения конфликтов клиенты с графическим интерфейсом.

Выбираем нужную строку (какую строку оставить, может вообще все хотим оставить), удаляем разделители, сохраняем файл, фиксируем изменения, коммитим, отправляем на сервер

```
D:\idm_test>git commit -m "test"
[master 3a0cb6f] test
D:\idm_test>git commit -m "test"
D:\idm_test>git push origin master
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 880 bytes | 176.00 KiB/s, done.
Total 9 (delta 0), reused 0 (delta 0)
To https://github.com/idmtest/firstApp.git
   375316a..3a0cb6f  master -> master
```

Удалите ветку newdev

Удаление веток на сервере

Если у вас есть несмерженная ветка, но вы хотите ее удалить, необходимо использовать команду:

git branch -D second

```
D:\idm_test>git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

D:\idm_test>git branch
* master
  newdev

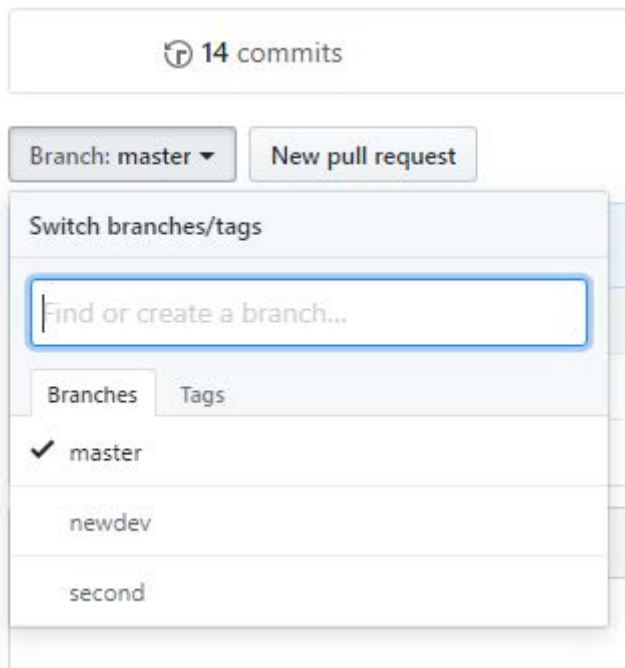
D:\idm_test>git branch -d
fatal: branch name required

D:\idm_test>git branch -d newdev
Deleted branch newdev (was a76b0f1).

D:\idm_test>git branch
* master

D:\idm_test>
```

Кстати, если посмотреть на github, то наши ветки там остались:



Давайте удалим их из github

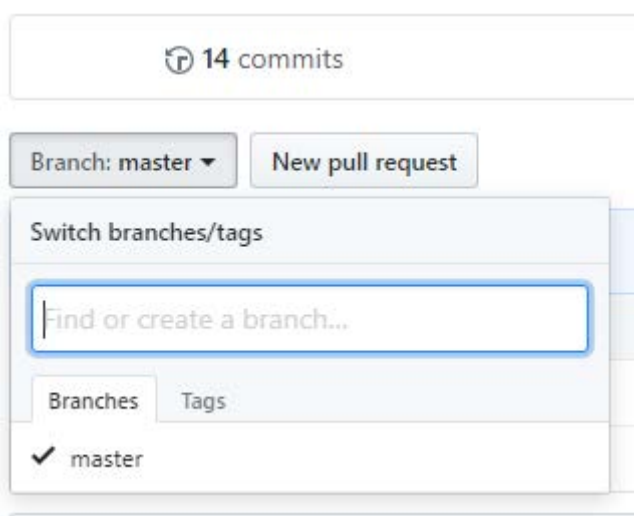
git push origin --delete newdev

git push origin --delete second

```
D:\idm_test>git push origin --delete newdev
To https://github.com/idmtest/firstApp.git
- [deleted]          newdev

D:\idm_test>git push origin --delete second
To https://github.com/idmtest/firstApp.git
- [deleted]          second

D:\idm_test>
```



Все ветки удалены

Возврат к предыдущему состоянию

Гит позволяет вернуть выбранный файл к состоянию на момент определенного коммита. Это делается уже знакомой нам командой `checkout`, которую мы ранее использовали для переключения между ветками. Но она также может быть использована для переключения между коммитами.

Чтобы посмотреть все комиты, можно использовать команду

git log

```
D:\idm_test>git log
commit 3a0cb6f3ad12a96dfe2c2cc0e21b4442d60eecff
Merge: 239d053 a76b0f1
Author: idm_test <idm_test@mail.ru>
Date: Wed Apr 17 23:18:10 2019 +0300

    test

commit 239d053b7a91fbce893f4ecd1ae724bea32ec479
Author: idm_test <idm_test@mail.ru>
Date: Wed Apr 17 22:58:59 2019 +0300

    ret11

commit a76b0f1427436d69af329836cabf2201387ba9c5
Author: idm_test <idm_test@mail.ru>
Date: Wed Apr 17 22:58:22 2019 +0300

    ret1

commit 375316a2e84cc37140051a0596b5ee591af7e6dd
Author: idm_test <idm_test@mail.ru>
Date: Wed Apr 17 22:57:05 2019 +0300

    ret

commit 5329c95b83b37a15c8a1c0ced9b3a865e9bd41dd
Author: idm_test <idm_test@mail.ru>
Date: Wed Apr 17 22:53:41 2019 +0300
```

Или можно использовать `github` или любой клиент

Commits on Apr 17, 2019

test idmtest committed 21 hours ago	3a0cb6f	<>
ret11 idmtest committed 22 hours ago	239d053	<>
ret1 idmtest committed 22 hours ago	a76b0f1	<>
ret idmtest committed 22 hours ago	375316a	<>
change file idmtest committed 22 hours ago	5329c95	<>
branch newdev and add in text idmtest committed 22 hours ago	792318a	<>
new file idmtest committed 22 hours ago	c2e30e3	<>
first commit idmtest committed 23 hours ago	23ce11c	<>

Выберите любой коммит, на который хотите откатиться, достаточно указать его первые несколько символов (перед выполнением посмотрите свои файлы, чтобы запомнить, что там находится)

git checkout [hash commit]

```
D:\idm_test>git checkout 5329c95
Note: checking out '5329c95'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 5329c95 change file
```

Если посмотреть файлы теперь, то увидите, что они поменялись — мы вернулись назад

Чтобы это произошло создалась псевдо-ветка начинающаяся на этом коммите, посмотрите ветки и вы увидите:

```
D:\idm_test>git branch
* (HEAD detached at 5329c95)
master
```

Есть несколько способов решения проблемы, обычно можно поменять изменения в последнем коммите используя:

git revert HEAD (лучше не пробуйте)

Можно также изменить коммит по hash:

git revert [hash commit] (можже не пробуйте)

По идеи нужно создавать новую ветку и продолжать в ней работать, а потом смерзиться

При отмене старых коммитов нужно быть готовым к тому, что возникнут конфликты. Такое случается, если файл был изменен еще одним, более новым коммитом. И теперь git не может найти строчки, состояние которых нужно откатить, так как они больше не существуют.

Если вы абсолютно уверены, что коммит, к которому вы откатились единственно правильный, а все последующие неверные, их можно удалить

git reset --hard HEAD – удаляет все, что не закоммичено

git reset --hard HEAD~n – удаляет n коммитов

Если вы работаете с кем-то над одним проектом, то правильнее переключиться на работающий коммит, проверить что все в нем работает создать от него новую ветку и провести смерживание. Либо просто перенести изменения в файлы, которые сейчас лежать в HEAD.

Все команды, которые позволяют поменять HEAD git, при этом удаляя ненужные коммиты, могут являться опасными, в плане потери времени из-за путаницы изменений файлов.

Откатитесь обратно на master ветку

Исправление коммита

Возможно ситуация, что вы закрепили файлы (поменяйте что-то в файле и закрепите их, вызовите git status), но еще не коммитили, хотите убрать файлы из области закрепления, вызовите команду:

git reset HEAD

Ваши файлы останутся такими же, но уйдут из области закрепления (вызовите git status) и снова будет показано, что есть измененные файлы

```

D:\idm_test>git status
On branch master
Your branch and 'origin/master' have diverged,
and have 2 and 1 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")

D:\idm_test>git add .

D:\idm_test>git status
On branch master
Your branch and 'origin/master' have diverged,
and have 2 and 1 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   test.txt

D:\idm_test>git reset HEAD
Unstaged changes after reset:
M       test.txt

D:\idm_test>git status
On branch master
Your branch and 'origin/master' have diverged,
and have 2 and 1 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")

```

Теперь снова добавим в область закрепления и закомитим, но еще не отправляем на сервер, можно вызвать ту же команду и тогда все, что осталось незакомиченным будет удалено

```

D:\idm_test>git reset HEAD

D:\idm_test>git status
On branch master
Your branch and 'origin/master' have diverged,
and have 2 and 1 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)

nothing to commit, working tree clean

D:\idm_test>

```

Если хотите изменить комментарий можно вызвать команду:

`git commit --amend`

Откроется консоль в редакторе Vi, чтобы начать что-то менять, надо нажать клавишу Insert, когда изменения в комментарий внесены, необходимо снова нажать Insert. Чтобы сохранить изменения, нажмите Esc, потом нажмите двоеточие (появится внизу экрана), напишите w! и нажмите Enter. Для выхода снова нажмите двоеточие, введите q! и нажмите Enter.

```
D:\idm_test>git commit -m "change comment"
[master 9524f05] change comment
1 file changed, 1 insertion(+), 1 deletion(-)

D:\idm_test>git reset HEAD

D:\idm_test>git add .

D:\idm_test>git commit --amend
[master 23112e9] change comment new
Date: Thu Apr 18 21:43:38 2019 +0300
1 file changed, 1 insertion(+), 1 deletion(-)

D:\idm_test>git log
commit 23112e9f9feaaaf880dce17caaa32615b55132db (HEAD -> master)
Author: idm_test <idm_test@mail.ru>
Date: Thu Apr 18 21:43:38 2019 +0300

    change comment new

commit 56dd348d6534834acc235b5ef4ffe3c5005483be
Author: idm_test <idm_test@mail.ru>
Date: Thu Apr 18 21:41:28 2019 +0300

    change comment
```

Отправка только нужных файлов на сервер

В большинстве проектов есть файлы или целые директории, в которые мы не хотим (и, скорее всего, не захотим) коммитить. Мы можем удостовериться, что они случайно не попадут в `git add -A` при помощи файла «.gitignore»

Создайте вручную файл под названием «.gitignore» и сохраните его в директорию проекта.

Внутри файла перечислите названия файлов/папок, которые нужно игнорировать, каждый с новой строки.

Файл «.gitignore» должен быть добавлен, закоммичен и отправлен на сервер, как любой другой файл в проекте.

Впишите в «.gitignore» файл, который не хотите отправлять на сервер.

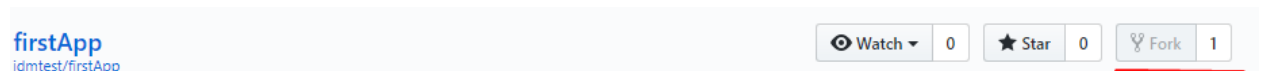
Зафиксируйте изменения, закоммите, отправьте на сервер

Все изменения можно проводить непосредственно на github, там также можно делать коммиты, только фиксация изменений происходит при сохранении файла

Совместная работа с git

Выберете себе напарника, скинте друг другу адреса своих репозиториев

Переходите по полученным ссылкам на репозиторий и нажимаете fork



Теперь это репозиторий добавляется к вам в проекты github

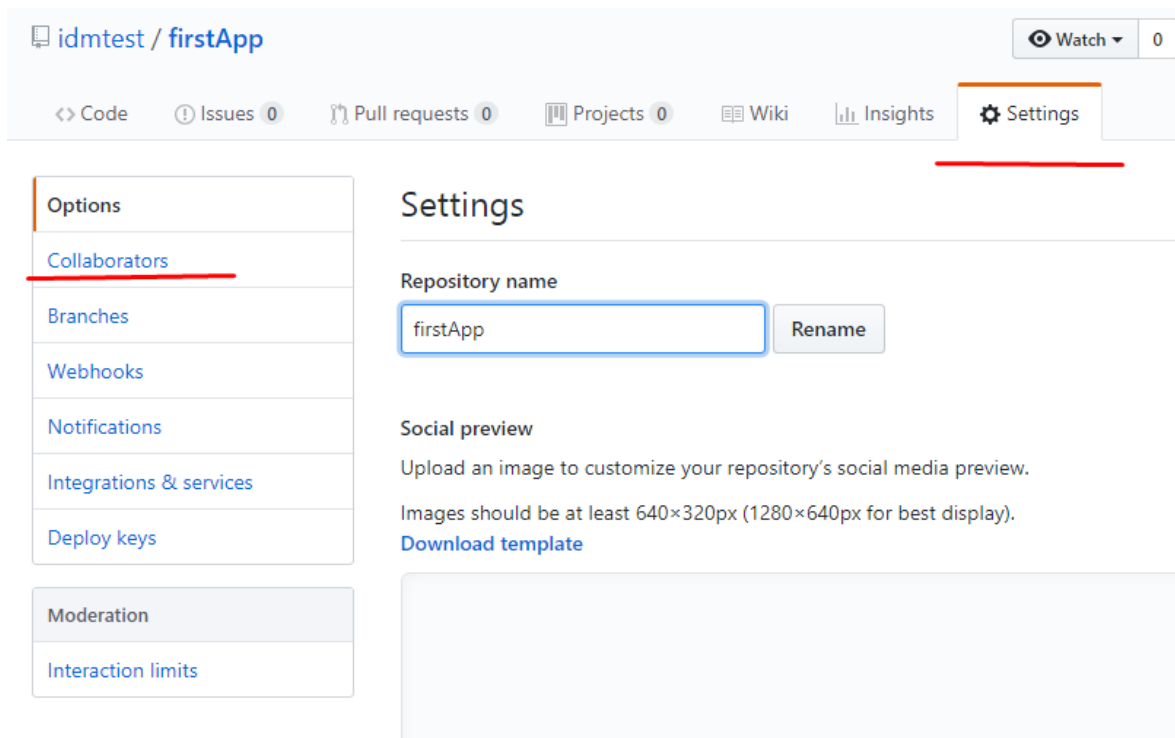
Далее через консоль клонируем этот репозиторий на локальный компьютер, предварительно создайте папку, в которую будете клонировать и в консоли перейдите в нее

git clone «адрес репозитория»

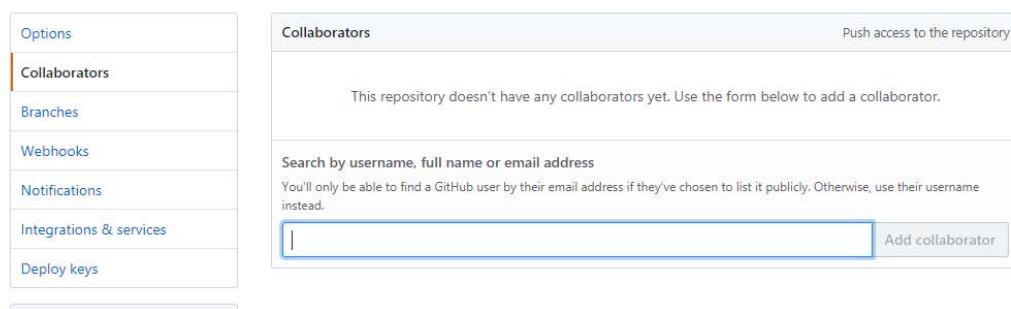
Таким образом вы создали скопировали проект и работаете с ним, но запись в проект вашего напарника не происходит

Необходимо дать права доступа для записи в ваш репозиторий

Для этого перейдите на github в раздел Settings, раздел Collaboration



Найдите в строке поиска того, кому хотите дать доступ в проект



Пусть вам также дадут доступ к репозиторию

Создайте новую папку и перейдите в нее в консоли

Сделайте clone проекта, к которому вам дали доступ

git clone «адрес репозитория»

Попробуйте теперь что-то поменять, закоммитеть и отправить на сервер.

Поработайте над одним файлом

Добавьте новый файл, отправьте его на сервер, чтобы, когда все разработчики сделают pull себе он появился локально у них

Сделайте push нескольких новых файлов, а также измененных добавленных файлов одним из разработчиков

Попробуйте получить результат, когда есть конфликты, которые необходимо разрешить

Шпаргалка Git

Инициализация репозитория

`git init` (от англ. *initialize*, «инициализировать») — инициализируй репозиторий.

Синхронизация локального и удалённого репозитория

`git remote add origin https://github.com/idmtest/firstApp.git` (от англ. *remote*, «удалённый» + *add*, «добавить») — привяжи локальный репозиторий к удалённому с URL <https://github.com/idmtest/firstApp.git>;

`git remote -v` (от англ. *verbose*, «подробный») — проверь, что репозитории действительно связались;

`git push -u origin main` (от англ. *push*, «толкать») — в первый раз загрузи все коммиты из локального репозитория в удалённый с названием `origin`.

💡 Ваша ветка может называться `master`, а не `main`. Подправьте команду, если это необходимо.

`git push` (от англ. *push*, «толкать») — загрузи коммиты в удалённый репозиторий после того, как он был привязан с помощью флага `-u`.

Подготовка файла к коммиту

`git add todo.txt` (от англ. *add*, «добавить») — подготовь файл `todo.txt` к коммиту;

`git add --all` (от англ. *add*, «добавить» + *all*, «всё») — подготовь к коммиту сразу все файлы, в которых были изменения, и все новые файлы;

`git add .` — подготовь к коммиту текущую папку и все файлы в ней.

Создание и публикация коммита

`git commit -m "Комментарий к коммиту."` (от англ. *commit*, «совершать», фиксировать» + *message*, «сообщение») — сделай коммит и оставь комментарий, чтобы было проще понять, какие изменения сделаны;

`git push` (от англ. *push*, «толкать») — добавь изменения в удалённый репозиторий.

Просмотр информации о коммитах

`git log` (от англ. *log*, «журнал [записей]») — выведи подробную историю коммитов;

`git log --oneline` (от англ. *log*, «журнал [записей]» + *oneline*, «одной строкой») — покажи краткую информацию о коммитах: сокращённый хеш и сообщение.

Просмотр состояния файлов

`git status` (от англ. *status*, «статус», «состояние») — покажи текущее состояние репозитория.

Добавление изменений в последний коммит

`git commit --amend --no-edit` (от англ. *amend*, «исправить») — добавь изменения к последнему коммиту и оставь сообщение прежним;
`git commit --amend -m "Новое сообщение"` — измени сообщение к последнему коммиту на Новое сообщение.

💡 Выйти из редактора Vim: нажать Esc, ввести `:qa!`, нажать Enter.

«Откат» файлов и коммитов

`git restore --staged hello.txt` (от англ. *restore*, «восстановить») — переведи файл `hello.txt` из состояния `staged` обратно в `untracked` или `modified`;
`git restore hello.txt` — верни файл `hello.txt` к последней версии, которая была сохранена через `git commit` или `git add`;
`git reset --hard b576d89` (от англ. *reset*, «сброс», «обнуление» + *hard*, «суровый») — удали все незакоммиченные изменения из `staging` и «рабочей зоны» вплоть до указанного коммита.

Просмотр изменений

`git diff` (от англ. *difference*, «отличие», «разница») — покажи изменения в «рабочей зоне», то есть в `modified`-файлах;
`git diff a9928ab 11bada1` — выведи разницу между двумя коммитами;
`git diff --staged` — покажи изменения, которые добавлены в `staged`-файлах.

Что еще почитать:

https://learngitbranching.js.org/?locale=ru_RU – красивые визуализации при работе с Git