

Гайд о том, как пользоваться git

*“Люди делятся на три типа: кто не делает бэкапы, кто делает бэкапы и кто делает бэкапы бэкапов.”*

*Ли Си Цын, искусство написания кода в ночь перед сдачей.*

*128 г. д.н.э*

**Версия от 21.11.2022**

**Автор:** Штанов Евгений Юрьевич

Есть вопросы? Для связи со мной:

[https://t.me/Klavishnik0\\_o](https://t.me/Klavishnik0_o)

[https://vk.com/klavishnik0\\_o](https://vk.com/klavishnik0_o)

shtanov.klavishnik@yandex.ru

# GIT

## Для кого и для чего

Итак, GIT. Что это такое и почему это нужно использовать?

GIT - это система контроля версий. Основные её достоинства - простота, “дешевизна” и повсеместное использование - иметь навыки в данной системе необходимо любому разработчику, чтобы получить работу.

*Что такое «система контроля версий» и почему это важно? Система контроля версий - это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться к предыдущему состоянию.*

Проще говоря - если вы внесли нежелательные изменения в структуру проекта (написали лишнюю строку или удалили файл), вы всегда можете откатиться к прошлому состоянию проекта.

Также данная система контроля версий позволяет **хранить ваши проекты на внешних бесплатных хранилищах (репозиториях), доступных везде.**

**САМОЕ ГЛАВНОЕ ДЛЯ ВАС!**

ИСПОЛЬЗУЯ СИСТЕМУ ВЕРСИЙ МОЖНО ВСЕГДА ВЕРНУТЬСЯ К СОСТОЯНИЮ, КОГДА ВСЕ РАБОТАЛО!

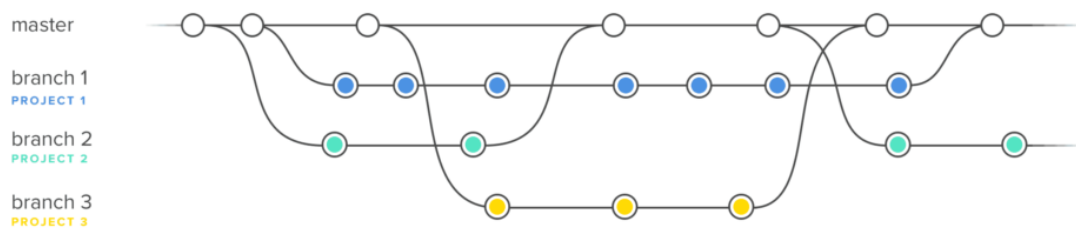
## Как это работает

GIT позволяет вести разработку проектов. Место, где хранится код вашего проекта называется **репозиторием**. Он может быть локальным - храниться только на вашем компьютере. Может быть внешним - есть множество сайтов, предоставляющих возможность вести репозитории (самые известные <https://github.com/> и <https://bitbucket.org/>). Внешние репозитории могут быть частными (private) и публичными (public). Частные репозитории доступны только для владельца и допущенных пользователей, публичные - всем.

В репозитории хранится ваша информация. Она может быть любой, но будем считать, что вы там храните код. Также там ведется история изменений вашего кода. Создается она не автоматически, а за счет так называемых **коммитов (commit)**. Коммит - это зафиксирование состояние вашего репозитория - такая контрольная точка, как сохранение в компьютерной игре - вы всегда можете создать несколько сохранений, откатиться (загрузиться) на предыдущее.

GIT позволяет вести одновременную разработку проекта за счет системы **веток (branch)**. Что это такое? Это когда вы от одного коммита (состояния кода) ведете несколько изменений кода параллельно. Т.е. вы можете одновременно реализовывать библиотеку в одной ветке и main в другой и они никак не влияют друг на друга. В конце вы можете объединить это вместе в один коммит .

Звучит сложно, но лучше всего это описывает такая картинка:



# Переход к практике

## Регистрация и создание внешнего репозитория

Если вы намерены “поднять” свой сервер git или использовать его локально, то данный шаг можно пропустить

В качестве примера будем использовать <https://github.com/>  
Регистрируемся там.

**ВАЖНО!** С недавних пор данный ресурс отключил аутентификацию по логину и паролю. Теперь есть аутентификация по токену (их можно создать несколько штук [например для разных систем] он используется как пароль, но его можно отозвать в любой момент) либо по ssh ключу. В качестве примера будем использовать вторую

Профиль на Гитхабе и все проекты в нём — ваше публичное портфолио разработчика, поэтому нужно завести профиль, если у вас его еще нет.

1. Зайдите на сайт <https://github.com> и нажмите кнопку Sign up.
2. Введите имя пользователя (понадобится в дальнейшей работе), адрес электронной почты (такой же, как при настройке Git) и пароль.
3. На почту придет код активации — введите на сайте.
4. Появится окно с выбором тарифного плана. Если вы пользуетесь Гитхабом для учёбы, то укажите, что профиль нужен только для вас и вы студент.
5. Опросы и выбор интересов можно пропустить. На этом всё — вы зарегистрировались и у вас есть собственный профиль.

## Устанавливаем ssh ключи

Заходим на сервер и терминале пишем следующее:

```
ssh-keygen -t ed25519 -C "your\_email@example.com"
```

С помощью данной команды мы создаем пару ssh ключей (про асимметричное шифрование, RSA можно почитать отдельно или попытать вашего преподавателя).

После выполнения данной команды у нас откроется интерактивное окошко.

- 1) Первый запрос будет на имя файлов с ключами, которые создаст данная утилита. Предлагаю назвать samos
- 2) Ввод парольной фразы и её подтверждение пропустим.

На экране будет примерно такой вывод

```
[klavishnik@unix:~]$ ssh-keygen -t ed25519 -C "your_email@example.com"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/users/klavishnik/.ssh/id_ed25519): samos
samos already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in samos
Your public key has been saved in samos.pub
The key fingerprint is:
SHA256:7qa4WI54DmQxq22bD7k0egI36h+a75wF07mBbMPb5/U your_email@example.com
The key's randomart image is:
+--[ED25519 256]--+
|
| o
| +
| + . S
| B.+ + .
|+OBB.. ..
|+*X@Bo..o
|+X@XBo..o.E
+-----[SHA256]-----+
```

После этого делаем

```
ls -la ~/.ssh/
```

У нас появились два файла

```
-rw----- 1 klavishnik users 419 Nov 17 15:24 samos
-rw-r--r-- 1 klavishnik users 104 Nov 17 15:24 samos.pub
```

**samos** - это закрытый ключ. Его передавать нельзя, храним только на сервере.

**samos.pub** - это открытый ключ. Его можно передавать, поскольку раскрытие такого рода ключей на безопасность не повлияет.

**ВАЖНО!** Обычно файлы создаются по адресу ~/.ssh/

У нас на сервере они создаются в корне, т.е. по адресу ~/

Теперь мы должны записать наш **закрытый ключ (samos)** в локальный менеджер ssh ключей.

Проверим, запущена ли утилита:

```
eval "$(ssh-agent -s)"
```

Если в ответ терминал покажет надпись **«Agent pid»** и число — значит, всё ок, агент запущен.

Теперь добавим наш ключ командой.

```
ssh-add ~/samos
```

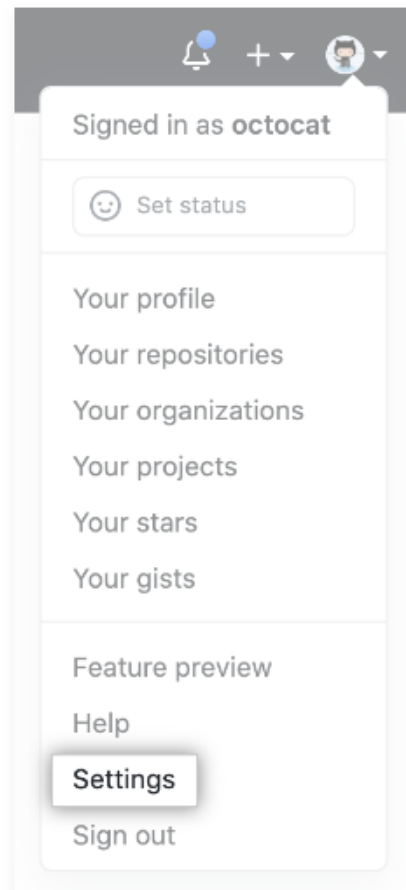
Теперь добавим **открытый ключ (samos.pub)** в профиль на гитхабе

Сделаем

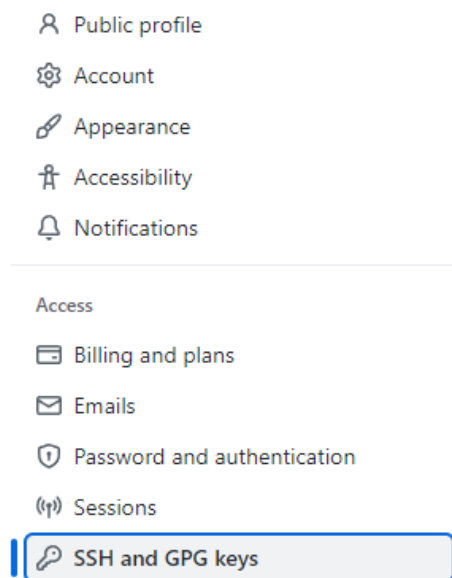
`cat samos.pub`

Все что выведется на экран - копируем правой кнопкой мыши.

Переходим на сайт гитхаба и справа сверху нажимаем лкм на свой аватар и выбираем пункт Settings



Далее переходим в раздел с SSH ключами



Нажимаем **new ssh key**

В поле внутри вставляем наш ключ

A screenshot of the 'Add SSH key' form in GitHub. The form has three main sections: 'Title' with a text input field, 'Key type' with a dropdown menu set to 'Authentication Key', and 'Key' with a large text area. The 'Key' field contains a list of supported key types: 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', and 'sk-ssh-ed25519@openssh.com'. At the bottom of the form is a green button labeled 'Add SSH key'.

Поле title заполняем по собственному усмотрению, например samos.

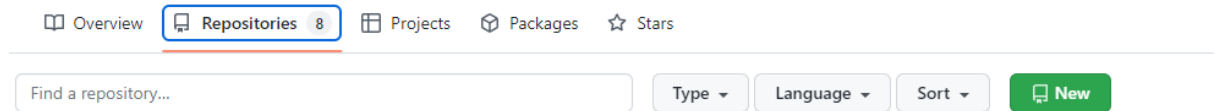
Нажимаем **add ssh key** и сохраняем.

У нас появится возможность заливать на репозитории гитхаба нашу информацию.

## Создаем внешний репозиторий

Переходим на страницу с личным профилем (нажимаем на аватар и первая строчка в списке).

Жмем на вкладку Repositories



И на кнопку NEW справа

### Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

 Klavishnik ▾

Repository name \*

/ test\_git ✓

Great repository names are short and memorable. Need inspiration? How about [cuddly-funicular?](#)

Description (optional)

☒  Public

Anyone on the internet can see this repository. You choose who can commit.

☐  Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

☐ Add a README file

This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: None ▾

Choose a license

A license tells others what they can and can't do with your code. [Learn more.](#)

License: None ▾

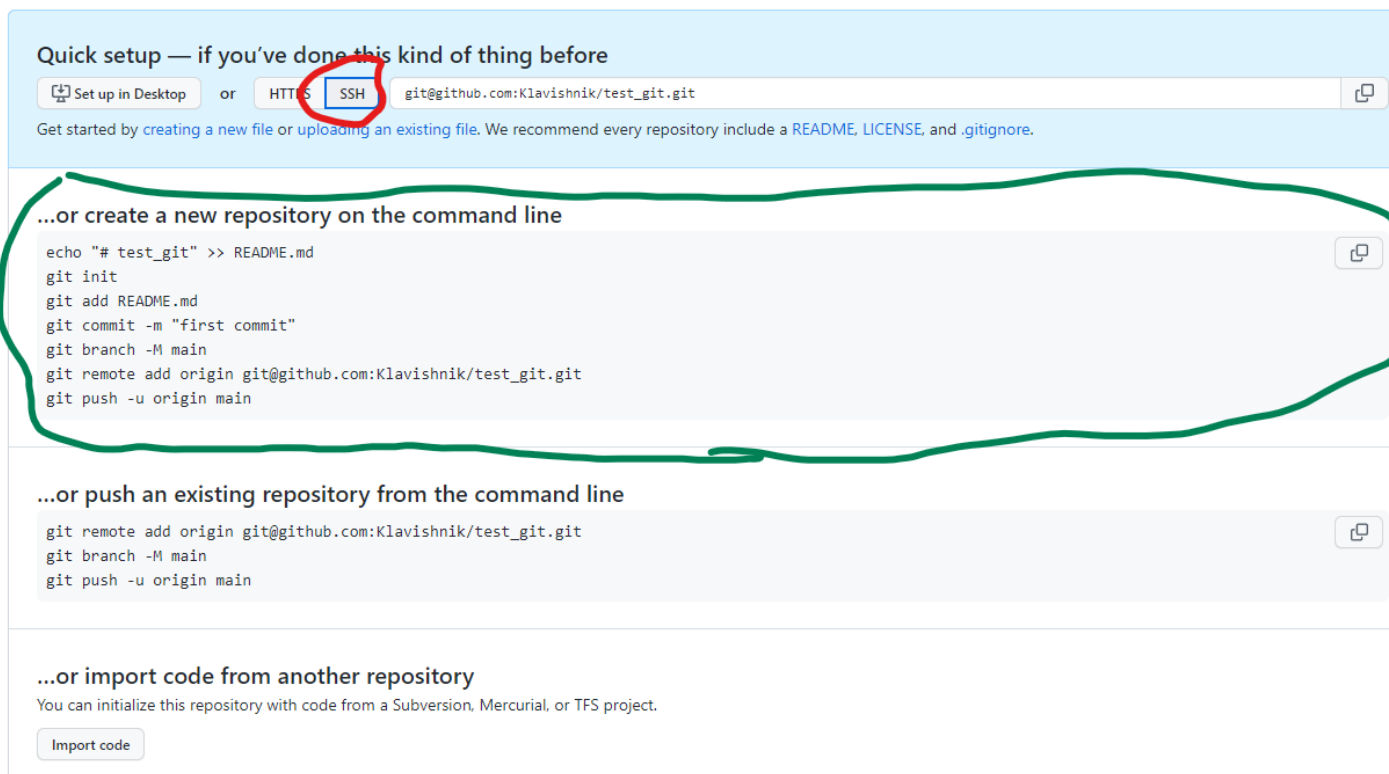
 You are creating a public repository in your personal account.

Create repository

Заполняем информацию. Из обязательного только название. Нажимаем Create repository

Внешние репозитории могут быть частными (private) и публичными (public). Частные репозитории доступны только для владельца и допущенных пользователей, публичные - всем.

Попадаем на такую вкладку. Выбираем SSH (выделено красным)



Далее возвращаемся на сервер.  
Сначала сделаем команду

```
[klavishnik@unix:~]$ git --version
git version 2.36.2
```

Если выдало версию, то git установлен, можно использовать.

Создаем папку с нашим проектом, например test\_git  
mkdir test\_git

И вставляем команды, которые даны выше (выделено зеленым).  
Далее будет объяснения, что происходит.



## Инициализация репозитория и первый коммит

Сначала необходимо добавить ваши данные в систему (почта нужна для аутентификации и а имя будет отображаться в коммитах)

```
git config --global user.email "ВАША ПОЧТА с гит"  
git config --global user.name "Ваш логин на гит"
```

Проверим

```
git config --global --list
```

```
[klavishnik@unix:~/test_git]$ git config --list  
user.email=shtanov.klavishnik@yandex.ru  
user.name=Klavishnik  
core.repositoryformatversion=0  
core.filemode=true  
core.bare=false  
core.logallrefupdates=true  
remote.origin.url=git@github.com:Klavishnik/test_git.git  
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*  
branch.master.remote=origin  
branch.master.merge=refs/heads/master
```

Перейдем в нашу папку с будущим проектом

```
cd test_git
```

Создадим файл README.md с одной строкой test\_git

```
echo "# test_git" >> README.md
```

**ВАЖНО** - .md - это расширение файлов языка разметки Markdown. README.md - это стандартное название файла заметки, которое будет отображаться внутри вашего репозитория, если зайти на веб версию гитхаба (покажу далее).

Сделаем инициализацию репозитория командой

```
git init
```

Посмотрим состояние репозитория командой `git status`

```
[klavishnik@unix:~/test_git]$ git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
```

ГИТ подсказывает, что был добавлен новый файл, но чтобы его отслеживала система контроля версий, нужно его добавить в отслеживаемые. Сделаем это.

Добавим файлы, отслеживаемые системой контроля версий.  
Можно сделать так, указав конкретный файл

```
git add README.md
```

Или так, добавив всю папку  

```
git add .
```

Посмотрим, что теперь пишет git status

```
[klavishnik@unix:~/test_git]$ git status
On branch master

No commits yet

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

У нас есть изменения, которые нужно закоммитить.

Для этого пишем следующее

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

Поздравляю! Мы сделали первый локальный коммит. Т.е. сделали контрольную точку и зафиксировали состояние проекта. При коммите необходимо писать краткое описание коммита. Это и делается через флаг -m. Данный комментарий потом можно будет увидеть в истории коммитов. Нужно это для того, чтобы быстро найти коммит, в котором вы сделали определенные изменения.

Можно не использовать флаг -m, а написать git commit. Тогда у вас откроется стандартный для системы текстовый редактор (nano или vim) и в нем нужно будет дать описание коммита

```
[klavishnik@unix:~/test_git]$ git commit -m "first commit"
[master (root-commit) 0272c9d] first commit
1 file changed, 1 insertion(+)
create mode 100644 README.md
```

Ну видно, что один файл изменился, какой это файлик был

```
[klavishnik@unix:~/test_git]$ git status
On branch master
nothing to commit, working tree clean
```

Новых веток пока создавать не будем.

Попробуем залить наши изменения на внешний репозиторий  
Для этого сначала обновим список внешних репозиториях на этой локальной машине

```
git remote add origin git@github.com:USERNAME/GIT_NAME.git
```

где

**USERNAME** - имя пользователя на github

**GIT\_NAME** - название репозитория

И зальем изменения

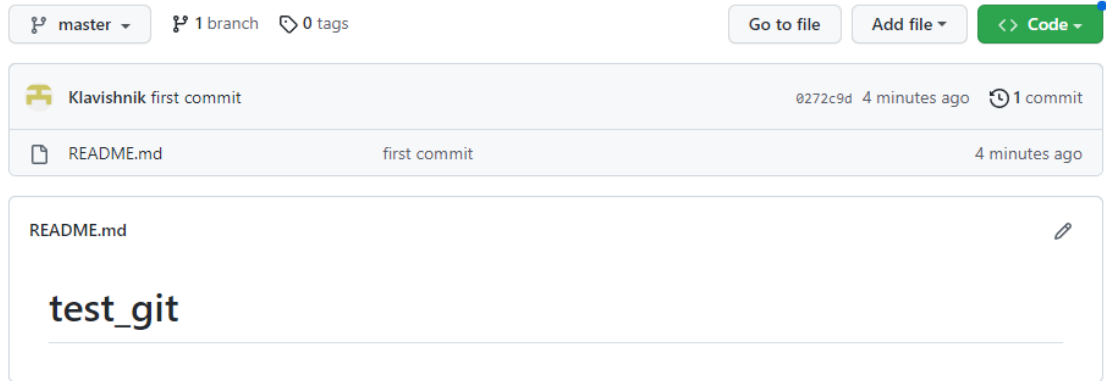
```
git push -u origin master
```

```
[klavishnik@unix:~/test_git]$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 227 bytes | 227.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:Klavishnik/test_git.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

Примечание: если вы выполняли все указания гитхаба при создании репозитория, ваша основная ветка будет называться main вместо master, соответственно здесь и далее необходимо будет заменять master на main.

Видно, что на репозиторий все улетело.

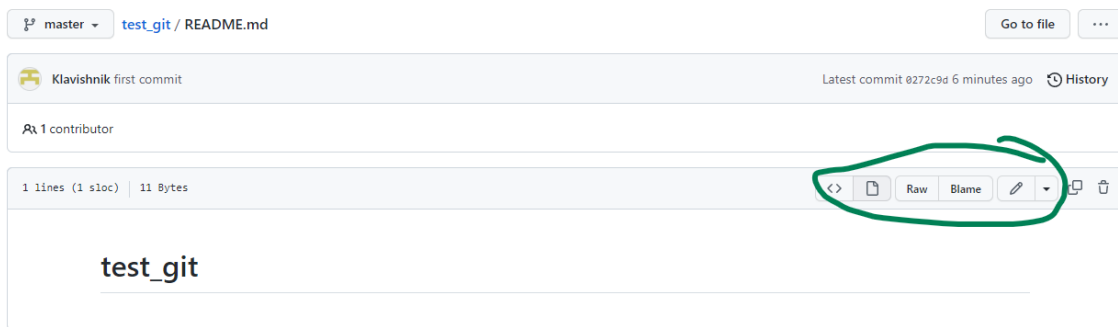
Пойдем на репозиторий, посмотрим что произошло



Видно, что у нас в списке файлов появился один файлик README.md

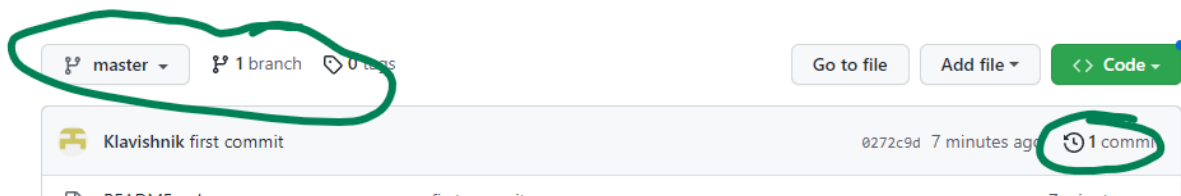
По традиции в данном файлике должно содержаться описание проекта / инструкция по использованию. Гитхаб ищет данный (именно с этим именем) файл в корне репозитория и сразу отображает его в веб версии репозитория.

Если щелкнуть на сам файл README.md (рядом с ним еще ярлык файла), то мы сможем посмотреть исходный код файла. Сделаем это:

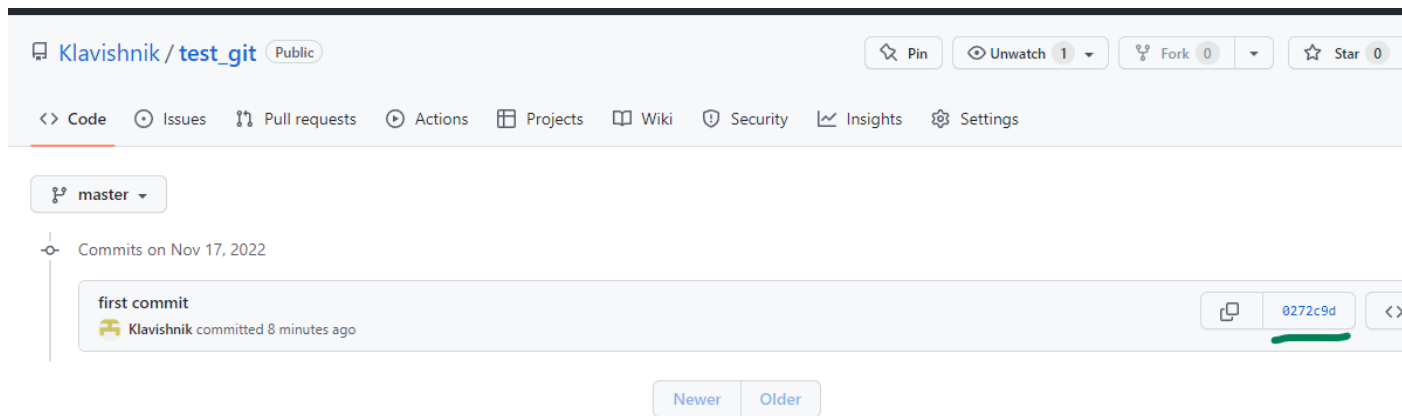


Если нажать на RAW, то можно увидеть исходное содержимое файла в “сыром” виде. Если нажать на карандаш - то можно править файл прямо на репозитории. О синхронизации изменений в следующих главах.

Вернемся обратно



Здесь можно смотреть все ветки и историю коммитов.



Это история изменений репозитория на данной ветке. Можно посмотреть подробнее, если щелкнуть на коммит. Также показывается сообщение, с которым мы коммитили код. (first commit)

Справа есть хэш коммита, он нужен, чтобы сразу можно было на него переключиться.

Как все это сделать в терминале? Как посмотреть историю коммитов?

`git log`

```
[klavishnik@unix:~/test_git]$ git log
commit 0272c9dd7fe3f0b298186bfc05067e136d106c (HEAD -> master, origin/master)
Author: Klavishnik <shtanov.klavishnik@yandex.ru>
Date: Thu Nov 17 15:56:17 2022 +0300

    first commit
```

Как посмотреть ветки?

`git branch`

```
[klavishnik@unix:~/test_git]$ git branch
* master
```

Звездочка укажет на ту, на которой мы сейчас находимся

Как посмотреть все наши действия с репозиторием?

`git reflog`

```
[klavishnik@unix:~/test_git]$ git reflog
0272c9d (HEAD -> master, origin/master) HEAD@{0}: commit (initial): first commit
```

Здесь будут указываться изменения репозитория, осуществляемые вами во всех ветках. Например, добавление коммита, переход на другую ветку, обновление репозитория и так далее.

## Коммитим, пушим и откатываемся

### КОММИТЫ

Итак, попробуем покоммитить наш проект. Создадим в нашем проекте файл с именем 1.c и и заполним в нем пару строчек, например, подключим библиотеки.

```
[klavishnik@unix:~/test_git]$ cat 1.c
#include <stdio.h>
#include <stdlib.h>
```

git status

```
[klavishnik@unix:~/test_git]$ git status
On branch master
Your branch is up to date with 'origin/master'.

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

nothing added to commit but untracked files present (use "git add" to track)
```

Добавим файл 1.c в индекс

```
git add 1.c
```

Посмотрим, что вышло

```
git status
```

```
[klavishnik@unix:~/test_git]$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  new file:   1.c
```

Коммитим

```
git commit -m "add 1.c file"
```

```
[klavishnik@unix:~/test_git]$ git commit -m "add 1.c file"
[master 17ce88a] add 1.c file
1 file changed, 2 insertions(+)
create mode 100644 1.c
```

```
git status
```

```
[klavishnik@unix:~/test_git]$ 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
```

Здесь написано, что коммитить нечего - все изменения репозитория зафиксированы, но наш локальный репозиторий опережает внешний репозиторий на один коммит.

Проверим  
Тут два коммита

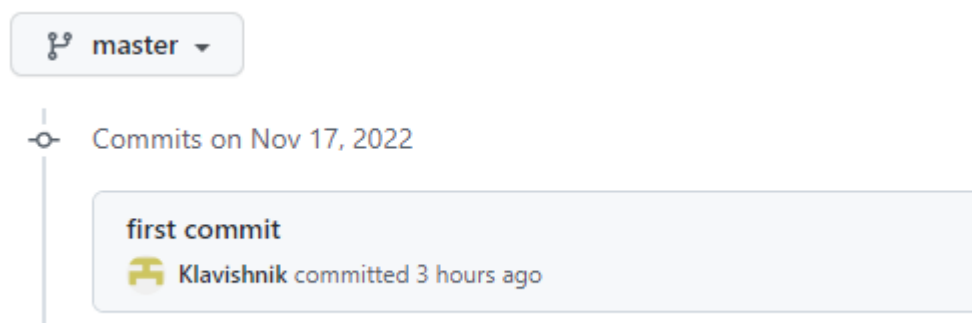
```
[klavishnik@unix:~/test_git]$ git log
commit 17ce88a94ed98ca9598f12e9f18f177d6c14a20c (HEAD -> master)
Author: Klavishnik <shtanov.klavishnik@yandex.ru>
Date: Thu Nov 17 19:23:44 2022 +0300

    add 1.c file

commit 0272c9dd7fe3f0b298186bfcfb05067e136d106c (origin/master)
Author: Klavishnik <shtanov.klavishnik@yandex.ru>
Date: Thu Nov 17 15:56:17 2022 +0300

    first commit
```

А тут один



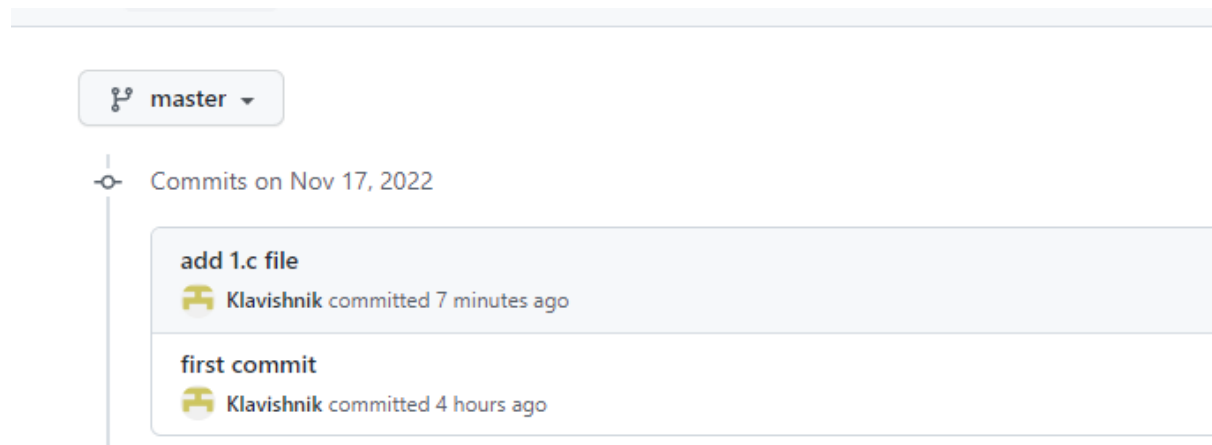
На сервере ничего нового

Запушим на сервер

git push

```
[klavishnik@unix:~/test_git]$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 24 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 303 bytes | 303.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:Klavishnik/test_git.git
0272c9d..17ce88a master -> master
```





Залилось на удаленный репозиторий и коммит наш отобразился в истории.

```
git status
```

```
[klavishnik@unix:~/test_git]$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Все обновлено, в том числе и на сервере.

## Откаты (состояния репозитория)

Сохраняться мы научились, а как загружаться?

А вот это правильный вопрос (с) потому что ответов на него даже больше, чем два.

Итак, рассмотрим две основные ситуации:

- 1) Нам нужно отменить все изменения в репозитории (вернуться к состоянию последнего коммита)

Допустим вы что-то делали, оно перестало работать и вы хотите вернуться к последнему состоянию (сделать загрузку быстрого [последнего] сохранения).

Допустим я чуть подправлю файл 1.c

```
[klavishnik@unix:~/test_git]$ cat 1.c
#include <stdio.h>
#include <stdlib.h>

int main()
```

И хочу отменить эти изменения (т.е. удалить последнюю строку)  
Состояние репозитория у меня не закоммичено

```
[klavishnik@unix:~/test_git]$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   1.c

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

Для этого делаем

git restore .                    для отката всей папки  
git restore file\_name        для отката конкретного файла

```
[klavishnik@unix:~/test_git]$ git restore .

[klavishnik@unix:~/test_git]$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Естественно в истории коммитов ничего отображаться не будет.

## 2) Нам нужно вернуться к определенному коммиту

Если вы долго что-то делали, успели несколько раз зафиксировать состояние репозитория и поняли, что ~~ваша жизнь не имеет смысла~~ разработка зашла немного не туда и вам нужно “загрузиться” на пару “уровней раньше”, то нужно понять следующее:

Даже если вы ошиблись - **удалять коммиты / ветки не нужно!**

История изменений в гит - **священна!**

Если вы ошиблись, то алгоритм следующий:

- 1) откатываемся на конкретный коммит, где все нормально;
- 2) от него ведем разработку в новой ветви.

Система git устроена таким образом, что создание новой ветки обходится достаточно “дешево” для системы, поэтому механизм ветвления можно использовать часто (в рамках разумного).

Повторю, что ветви идут параллельно друг другу и изменения в них не отражаются в других ветвях. Чтобы объединить ветки, нужно сделать специальную операцию - **запрос на вытягивание (pull request)**, случайно сделать его не выйдет.

Итак, что мы делаем.

Во-первых, я немного поменяю файл 1.c и зафиксирую изменения и отправлю их::

```
[klavishnik@unix:~/test_git]$ git commit -m "update"
[master 1696e25] update
1 file changed, 5 insertions(+)

[klavishnik@unix:~/test_git]$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 24 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 328 bytes | 164.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:Klavishnik/test_git.git
17ce88a..1696e25 master -> master

[klavishnik@unix:~/test_git]$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

А теперь я хочу откатиться на 2 коммит. Как это сделать?

`git reflog`

```
[klavishnik@unix:~/test_git]$ git reflog
1696e25 (HEAD -> master, origin/master) HEAD@{0}: commit: update
17ce88a HEAD@{1}: commit: add 1.c file
0272c9d HEAD@{2}: commit (initial): first commit
```

Перейдем на коммит HEAD@{1}

Для этого используем команду

`git checkout commit`

Вместо commit подставляем хэш (цифры из желтого первого столбца)

```
[klavishnik@unix:~/test_git]$ git checkout 17ce88a
Note: switching to '17ce88a'.

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 switching back to a branch.

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

    git switch -c <new-branch-name>

Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 17ce88a add 1.c file

[klavishnik@unix:~/test_git]$ git status
HEAD detached at 17ce88a
nothing to commit, working tree clean
```

Переключились. Проверим

`cat 1.c`

```
[klavishnik@unix:~/test_git]$ cat 1.c
#include <stdio.h>
#include <stdlib.h>
```

Да, все верно.

А что теперь? А теперь создаем новую ветку и продолжаем вести разработку от этого коммита:

Создадим новую ветку с названием func\_main

`git branch func_main`

Проверим  
git branch

```
[klavishnik@unix:~/test_git]$ git branch
* (HEAD detached at 17ce88a)
  func_main
  master
```

Переключаемся на новую ветку  
git checkout func\_main

```
[klavishnik@unix:~/test_git]$ git checkout func_main
Switched to branch 'func_main'
```

Делаем изменения в 1.c

Коммитим и загружаем на внешний репозиторий

```
git add 1.c
git commit -m "added func"
```


```
git push origin func_main
```


```
[klavishnik@unix:~/test_git]$ git push origin func_main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 24 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 317 bytes | 317.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'func_main' on GitHub by visiting:
remote:   https://github.com/Klavishnik/test_git/pull/new/func_main
remote:
To github.com:Klavishnik/test_git.git
 * [new branch]      func_main -> func_main
```










Видно, что на репозитории появилась новая ветка func\_main

Посмотрим, что у нас на веб версии репозитория


## Ветка master


 master ▾










 Commits on Nov 17, 2022

<b>update</b>  Klavishnik committed 12 minutes ago	 1696e25 
<b>add 1.c file</b>  Klavishnik committed 38 minutes ago	 17ce88a 
<b>first commit</b>  Klavishnik committed 4 hours ago	 0272c9d 

## Ветка func\_main (новая)

 func\_main ▾

 Commits on Nov 17, 2022

<b>added func</b>  Klavishnik committed 2 minutes ago	 156bf41 
<b>add 1.c file</b>  Klavishnik committed 38 minutes ago	 17ce88a 
<b>first commit</b>  Klavishnik committed 4 hours ago	 0272c9d 

[Newer](#) [Older](#)

Обратите внимание на цифры справа - это хэш коммита. Видно, что первые два (снизу) совпадают, а третий отличается - собственно, мы откатились до второго коммита и сделали третий уже новым.

# Вывод истории в виде графа

Для наглядного отображения истории вашего репозитория, его можно вывести в виде графа.

Выполним:

```
git log --all --decorate --oneline --graph
```

```
[klavishnik@unix:~/test_git]$ git log --all --decorate --oneline --graph
* 156bf41 (HEAD -> func_main, origin/func_main) added func
| * 1696e25 (origin/master, master) update
|/
* 17ce88a add 1.c file
* 0272c9d first commit
```



Слева можно видеть, как у нас расположились ветки (|) и коммиты (\*)

А вот так красивее

```
git log --graph --abbrev-commit --decorate --date=relative --all
```

```
[klavishnik@unix:~/test_git]$ git log --graph --abbrev-commit --decorate --date=relative --all
* commit 156bf41 (HEAD -> func_main, origin/func_main)
| Author: Klavishnik <shtanov.klavishnik@yandex.ru>
| Date: 9 minutes ago
|
| added func
|
| * commit 1696e25 (origin/master, master)
|/ Author: Klavishnik <shtanov.klavishnik@yandex.ru>
| Date: 20 minutes ago
|
| update
|
* commit 17ce88a
| Author: Klavishnik <shtanov.klavishnik@yandex.ru>
| Date: 45 minutes ago
|
| add 1.c file
|
* commit 0272c9d
| Author: Klavishnik <shtanov.klavishnik@yandex.ru>
| Date: 4 hours ago
|
| first commit
```

# Работаем с существующим внешним репозиторием из нескольких клиентов

## Клонируем существующий репозиторий (git clone)

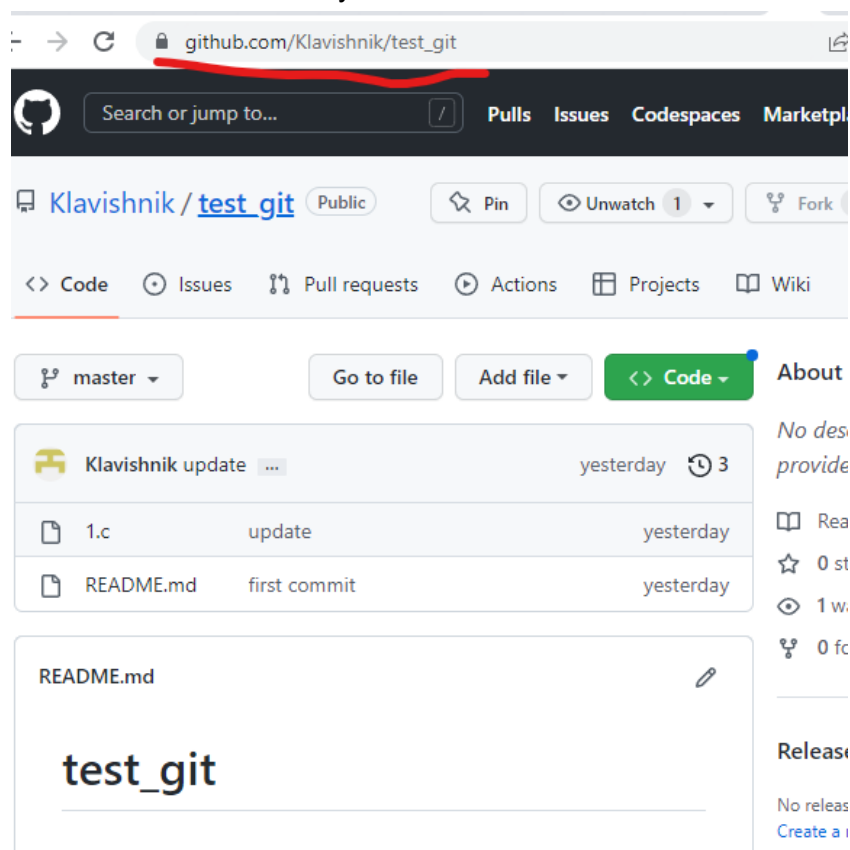
Итак, у нас есть существующий репозиторий, который мы, например, вели из дома с нашей домашней машины.

Мы пришли в универ и нужно скачать наши изыскания, чтобы показать преподавателю. Что делать?

Для этого есть команда

```
git clone <название репозитория>
```

Название можно взять тут





Выполним

```
user@Shtanov-Home-PC:~$ git clone https://github.com/Klavishnik/test_git
Cloning into 'test_git'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 12 (delta 1), reused 11 (delta 0), pack-reused 0
Unpacking objects: 100% (12/12), 994 bytes | 994.00 KiB/s, done.
user@Shtanov-Home-PC:~$ ls -l
total 8
drwxr-xr-x 3 user user 4096 Nov 18 23:55 test_git
```

Видно, что все скачалось.

Если хотим скачать другую ветку, то можно использовать такие флаги:

```
git clone -b [название ветки] <ссылка до репозитория>
```

Пример

```
git clone -b func_main https://github.com/Klavishnik/test_git.git
```

```
user@Shtanov-Home-PC:~$ git clone -b func_main https://github.com/Klavishnik/test_git.git
Cloning into 'test_git'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 15 (delta 1), reused 14 (delta 0), pack-reused 0
Unpacking objects: 100% (15/15), 1.23 KiB | 1.23 MiB/s, done.
user@Shtanov-Home-PC:~$ cd test_git/
user@Shtanov-Home-PC:~/test_git$ git branch
* func_main
```

Собственно, у нас есть локальная копия репозитория. Её можно менять, можно коммитить, но только **локально**. Ввиду того, что скачали репозиторий по **ссылке https** при попытке **запустить изменения** появится **ошибка авторизации!**

```
user@Shtanov-Home-PC:~/test_git$ touch 3.c
user@Shtanov-Home-PC:~/test_git$ git add .
user@Shtanov-Home-PC:~/test_git$ git commit -m "test"
[master 2f95e81] test
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 3.c
user@Shtanov-Home-PC:~/test_git$ git push
Username for 'https://github.com': █
```

## Клонируем и пушим в существующий репозиторий

Для работы с внешним репозиторием с другого компьютера нужно пройти этап авторизации.

Чтобы пройти авторизацию с другого устройства, необходимо по аналогии с первым компьютером обеспечить наличие пары ключей и загрузить публичный ключ в настройки своего профиля на гитхабе. Здесь возможно два принципиально различных подхода:

1. Создать абсолютно новую пару ключей для нового устройства (см. “Устанавливаем ssh ключи”);
2. Копировать уже имеющиеся ключи с первого устройства на новое.

У обоих подходов есть свои плюсы и минусы, однако 1-й является более общепринятым так как обеспечивает гораздо большую степень безопасности. По этой причине, для подключения нового устройства рекомендуется повторить шаги, описанные в “Устанавливаем ssh ключи”.

Однако, если по какой-то причине вы решили воспользоваться 2-м способом, вы можете сделать это, например, при помощи команды scp:

```
scp username@name_server1:path/to/file path/local/pc
```

Выполнять эту команду нужно на машине, КУДА вы хотите скачать файл.

server1 - имя сервера откуда вы хотите файлы скопировать

Через двоеточие указывается путь до файла и его название

Пример

```
scp user@samos:~/samos ~/.ssh
```

Потребуется ввод пароля

После получения ключа добавляем его по аналогии с предыдущим разделом

Проверим, запущена ли утилита:

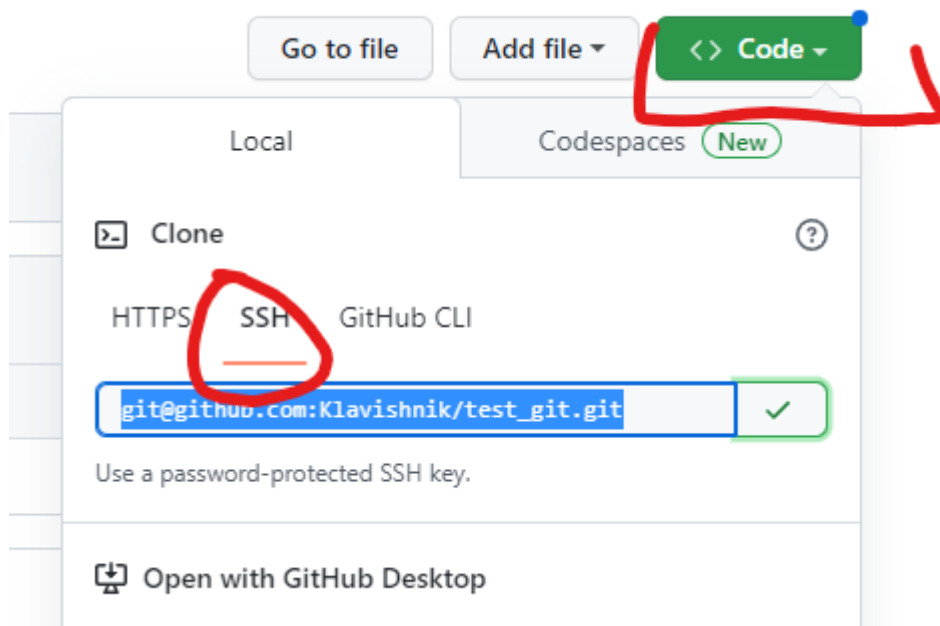
```
eval "$(ssh-agent -s)"
```

Если в ответ терминал покажет надпись «**Agent pid**» и число — значит, всё ок, агент запущен.

Теперь добавим наш ключ командой.

```
ssh-add ~/samos
```

После добавления ключа нового устройства скачаем репозиторий через ssh



Скачаем master ветку

```
git clone git@github.com:Klavishnik/test\_git.git
```

```
user@Shtanov-Home-PC:~$ git clone git@github.com:Klavishnik/test_git.git
Cloning into 'test_git'...
Warning: Permanently added the ECDSA host key for IP address '140.82.121.4' to the list of known hosts.
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 15 (delta 1), reused 14 (delta 0), pack-reused 0
Receiving objects: 100% (15/15), done.
Resolving deltas: 100% (1/1), done.
user@Shtanov-Home-PC:~$
```

Делаем git status

```
user@Shtanov-Home-PC:~/test_git$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
user@Shtanov-Home-PC:~/test_git$
```

Мы на ветке master и она синхронизирована с веткой на удаленном репозитории

Попробуем переключиться на ветку func\_main и что-нибудь в неё залить  
Сначала найдем эту ветку  
Посмотрим все ветки

```
git branch -a
```

```
user@Shtanov-Home-PC:~/test_git$ git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/func_main
remotes/origin/master
```

Напоминаю origin - название нашего внешнего репозитория (remotes намекает), которое мы дали при синхронизации

Переключаемся на ветку

```
git checkout func_main
```


```
user@Shtanov-Home-PC:~/test_git$ git checkout func_main
Branch 'func_main' set up to track remote branch 'func_main' from 'origin'.
Switched to a new branch 'func_main'
```


Ветка func\_main обновлена до состояния внешнего репозитория (не отличается от него). Мы успешно переключились на эту ветку.


Поменяем что-нибудь

```
user@Shtanov-Home-PC:~/test_git$ ls -la
total 20
drwxr-xr-x 3 user user 4096 Nov 19 00:23 .
drwxr-xr-x 6 user user 4096 Nov 19 00:21 ..
drwxr-xr-x 8 user user 4096 Nov 19 00:23 .git
-rw-r--r-- 1 user user  56 Nov 19 00:23 1.c
-rw-r--r-- 1 user user  11 Nov 19 00:21 README.md
user@Shtanov-Home-PC:~/test_git$ echo 3 >> 3
user@Shtanov-Home-PC:~/test_git$ git add .
user@Shtanov-Home-PC:~/test_git$ git commit -m "new file"
[func_main 1c0b7c2] new file
 1 file changed, 1 insertion(+)
 create mode 100644 3
user@Shtanov-Home-PC:~/test_git$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 300 bytes | 300.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:Klavishnik/test_git.git
   156bf41..1c0b7c2  func_main -> func_main
user@Shtanov-Home-PC:~/test_git$
```

Я добавил новый файл 3 и успешно запустил его в внешний репозиторий


 func\_main ▾


 2 branches



 0 tags




Go to file

Add file ▾

 Code ▾

This branch is [2 commits ahead](#), [2 commits behind](#) master.  [Contribute](#) ▾

 Klavishnik new file 1c0b7c2 1 minute ago  4 commits

 1.c	added func	yesterday
 3	new file	1 minute ago
 README.md	first commit	yesterday

Вот он появился

# Синхронизация изменений репозитория на разных клиентах.

## Синхронизация при работе в одиночку

Допустим ваш репозиторий живет на двух машинах. На обеих он уже клонирован. Например, на домашней машине вы залили изменения на гитхаб, пошли в универ и там хотите обновить папку проекта. Вопрос как?

```
git status
```

Покажет, что все обновлено. Но это не совсем верно.

```
user@Shtanov-Home-PC:~/test_git$ git status
On branch func_main
Your branch is up to date with 'origin/func_main'.

nothing to commit, working tree clean
```

Синхронизируемся с сервером командой

```
git fetch
```

И посмотрим вывод `git status`

```
user@Shtanov-Home-PC:~/test_git$ git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 309 bytes | 309.00 KiB/s, done.
From github.com:Klavishnik/test_git
   1c0b7c2..dd0368d  func_main -> origin/func_main
user@Shtanov-Home-PC:~/test_git$ git status
On branch func_main
Your branch is behind 'origin/func_main' by 1 commit, and can be fast-forwarded.
(use "git pull" to update your local branch)
```

Система подсказывает, что наш репозиторий “опаздвает” на один коммит и сама подсказывает, что нужно выкачать изменения командой `git pull`

Делаем

```
git pull
```

```
user@Shtanov-Home-PC:~/test_git$ git pull
Updating 1c0b7c2..dd0368d
Fast-forward
 4 | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 4
user@Shtanov-Home-PC:~/test_git$ git status
On branch func_main
Your branch is up to date with 'origin/func_main'.

nothing to commit, working tree clean
```

## Синхронизация при работе в команде

Рассмотрим ситуацию, когда ваш коллега запушил в эту же ветку что-то пока вы работали и вас об этом не предупредил.

Перейдем обратно на сервер

```
[klavishnik@unix:~/test_git]$ git status
On branch func_main
nothing to commit, working tree clean
```

Мы на той же ветке

```
[klavishnik@unix:~/test_git]$ ls -la
total 8
drwxr-xr-x  3 klavishnik users  46 Nov 17 19:59 .
drwx----- 12 klavishnik users 302 Nov 19 00:12 ..
-rw-r--r--  1 klavishnik users  56 Nov 17 19:59 1.c
drwxr-xr-x  8 klavishnik users 184 Nov 19 00:26 .git
-rw-r--r--  1 klavishnik users  11 Nov 17 15:52 README.md
```

Нового файла нет

Попробуем создать новый файл 4

```
echo 4 > 4
```

```
git add .
```

```
git commit -m "file 4"
```

```
git push -u origin func_main
```

```
[klavishnik@unix:~/test_git]$ echo 4 > 4
```

```
[klavishnik@unix:~/test_git]$ git add .
```

```
[klavishnik@unix:~/test_git]$ git commit -m "file 4"
```

```
[func_main e9f6b92] file 4
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 4
```

```
[klavishnik@unix:~/test_git]$ git push -u origin func_main
```

```
To github.com:Klavishnik/test_git.git
```

```
! [rejected]        func_main -> func_main (fetch first)
```

```
error: failed to push some refs to 'github.com:Klavishnik/test_git.git'
```

```
hint: Updates were rejected because the remote contains work that you do
```

```
hint: not have locally. This is usually caused by another repository pushing
```

```
hint: to the same ref. You may want to first integrate the remote changes
```

```
hint: (e.g., 'git pull ...') before pushing again.
```

```
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Конфликт

git подсказывает, что причина в том, что на удаленном репозитории есть изменения, которых нет на локальной машине. Это могло произойти, например, когда ваш коллега одновременно запустил в эту же ветку что-то пока вы работали.

**!Поэтому, прежде, чем залить что-то на репозиторий, нужно выкачать последние изменения!**

Нужно посмотреть, не приведет это к конфликтам

Делаем

git fetch

Так получим последнюю информацию о изменениях

И осмотримся, что там на репозитории натворили без нас

git branch -a

Переключимся на внешнюю ветку. **На ней мы находимся в режиме чтения и лучше ничего не менять**

git checkout remotes/origin/func\_main

git branch

```
[klavishnik@unix:~/test_git]$ git branch -a
* func_main
  master
  remotes/origin/func_main
  remotes/origin/master

[klavishnik@unix:~/test_git]$ git checkout remotes/origin/func_main
Note: switching to 'remotes/origin/func_main'.

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 switching back to a branch.

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

    git switch -c <new-branch-name>

Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 1c0b7c2 new file

[klavishnik@unix:~/test_git]$ git branch
* (HEAD detached at origin/func_main)
  func_main
```



Смотрим, что изменилось

```
[klavishnik@unix:~/test_git]$ ls -la
total 12
drwxr-xr-x  3 klavishnik users  55 Nov 19 00:39 .
drwx----- 12 klavishnik users 302 Nov 19 00:39 ..
-rw-r--r--  1 klavishnik users  56 Nov 17 19:59 1.c
-rw-r--r--  1 klavishnik users   2 Nov 19 00:39 3
drwxr-xr-x  8 klavishnik users 184 Nov 19 00:39 .git
-rw-r--r--  1 klavishnik users  11 Nov 17 15:52 README.md
```

Ну вроде ничего критичного. Эти изменения наш код не затрагивают, поэтому возвращаемся обратно на нашу ветку

git checkout func\_main

```
[klavishnik@unix:~/test_git]$ git checkout func_main
Previous HEAD position was 1c0b7c2 new file
Switched to branch 'func_main'

[klavishnik@unix:~/test_git]$ git branch -a
* func_main
  master
  remotes/origin/func_main
  remotes/origin/master
```

И делаем

git pull

Так мы скачаем все изменения и объединит с нашими локальными

git pull origin func\_main

```
[klavishnik@unix:~/test_git]$ git pull origin func_main
From github.com:Klavishnik/test_git
* branch          func_main -> FETCH_HEAD
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

Ошибка

Это произошло, потому что мы поменяли нашу папку и скачали изменения из репозитория. Они расходятся и нужно их синхронизировать.

Мы уже посмотрели, что изменения внешнего репа не влияет на локалку, поэтому сделаем merge - сольем все изменения вместе.

git config pull.rebase true - таким способом мы сделали настройку для ВСЕХ таких будущих конфликтов. Т.е. при git pull у нас все будет объединяться

```
git pull origin func_main
```

```
[klavishnik@unix:~/test_git]$ git config pull.rebase true

[klavishnik@unix:~/test_git]$ git pull origin func_main
From github.com:Klavishnik/test_git
* branch          func_main -> FETCH_HEAD
Successfully rebased and updated refs/heads/func_main.
```

Вот что из этого вышло

```
[klavishnik@unix:~/test_git]$ ls -la
total 16
drwxr-xr-x  3 klavishnik users  64 Nov 19 00:51 .
drwx----- 12 klavishnik users 302 Nov 19 00:43 ..
-rw-r--r--  1 klavishnik users  56 Nov 17 19:59 1.c
-rw-r--r--  1 klavishnik users   2 Nov 19 00:51 3
-rw-r--r--  1 klavishnik users   2 Nov 19 00:51 4
drwxr-xr-x  8 klavishnik users 219 Nov 19 00:51 .git
-rw-r--r--  1 klavishnik users  11 Nov 17 15:52 README.md
```

Зальем наши изменения на репозиторий

```
git push -u origin func_main
```

```
[klavishnik@unix:~/test_git]$ git push -u origin func_main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 24 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 329 bytes | 329.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:Klavishnik/test_git.git
   1c0b7c2..dd0368d  func_main -> func_main
branch 'func_main' set up to track 'origin/func_main'.
```

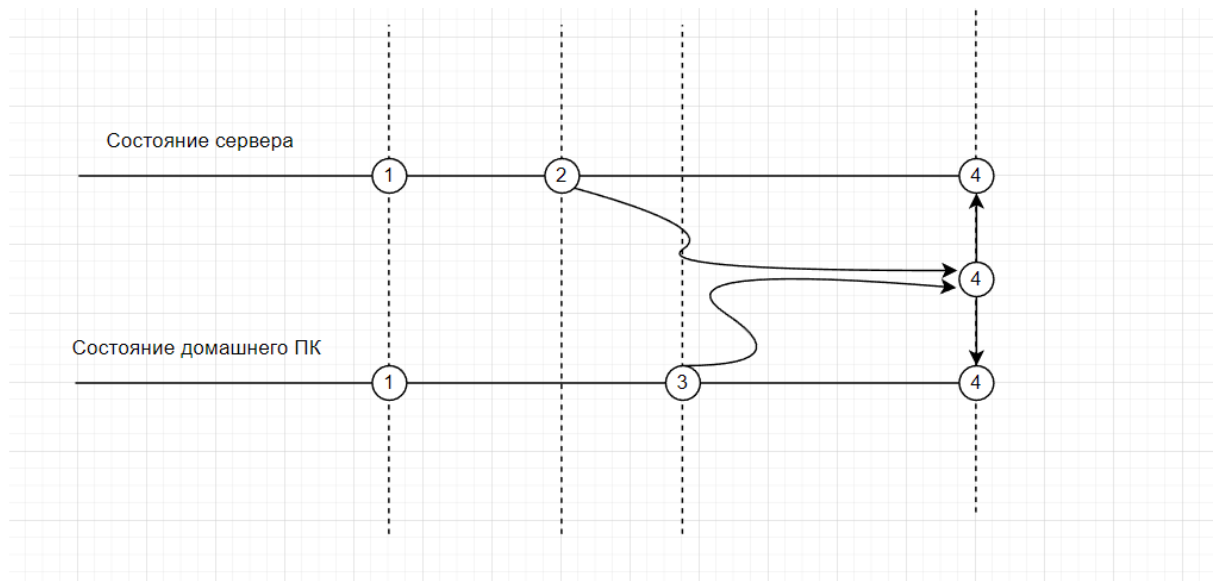
func\_main ▾
2 branches
0 tags
Go to file
Add file ▾
<> Code ▾

This branch is [3 commits ahead](#), [2 commits behind](#) master.
Contribute ▾

Klavishnik file 4 <span>dd0368d 3 minutes ago</span> <span>5 commits</span>	
1.c	added func yesterday
3	new file 30 minutes ago
4	file 4 3 minutes ago
README.md	first commit yesterday

Наши ветки синхронизированы

А теперь еще раз, что же у нас произошло, но в картинках



Итак, мы клонировали репозиторий с сервера в состоянии (1).

Потом начали менять наши файлы локально и пришли в состояние (3).

В это время на сервер были загружены данные (2)

Мы объединили состояние (2) и (3) в состояние (4) и отправили на сервер.

# Markdown

## Markdown -

чтобы ваш README.md (и описание репозитория на веб морде) выглядело примерно так

4. memory\_leak

heap\_example - отдельный пример неправильного выделения памяти под динамический массив

### Как собрать?

в каждой папке есть **Makefile** который принимает параметры в формате `make name`, где `name` :

- *static* - позволяет прогнать код статическим анализатором clang и поместить отчет в папку **html-dir**;
- *orig* - собирает обычный бинарь без всяких проверок и инструментации. Имя скомпилированного файла - **bin\_clear**;
- *asan/msan* - сборка бинаря с инструментацией ASAN or MSAN (смотря в какой папке собираете). Имя скомпилированного файла **bin\_asan** или **bin\_msan**;
- *clean* - удаляет все файлы, полученные при сборке;
- *all* - выполнит стадии *static*, *asan/(msan)*, *orig*. Данная стадия запускается при выполнении `make` без аргументов

Примеры:

*build all:*

```
cd asan/heap_buffer_overflow/  
make
```

*clean:*

```
make clean
```

### Как использовать?

1. Перейдите в директорию с нужным вам примером и выполните: `./bin_asan`

или для папки с msan `./bin_msan`

Вы увидите вывод санитайзера с описанием ошибки

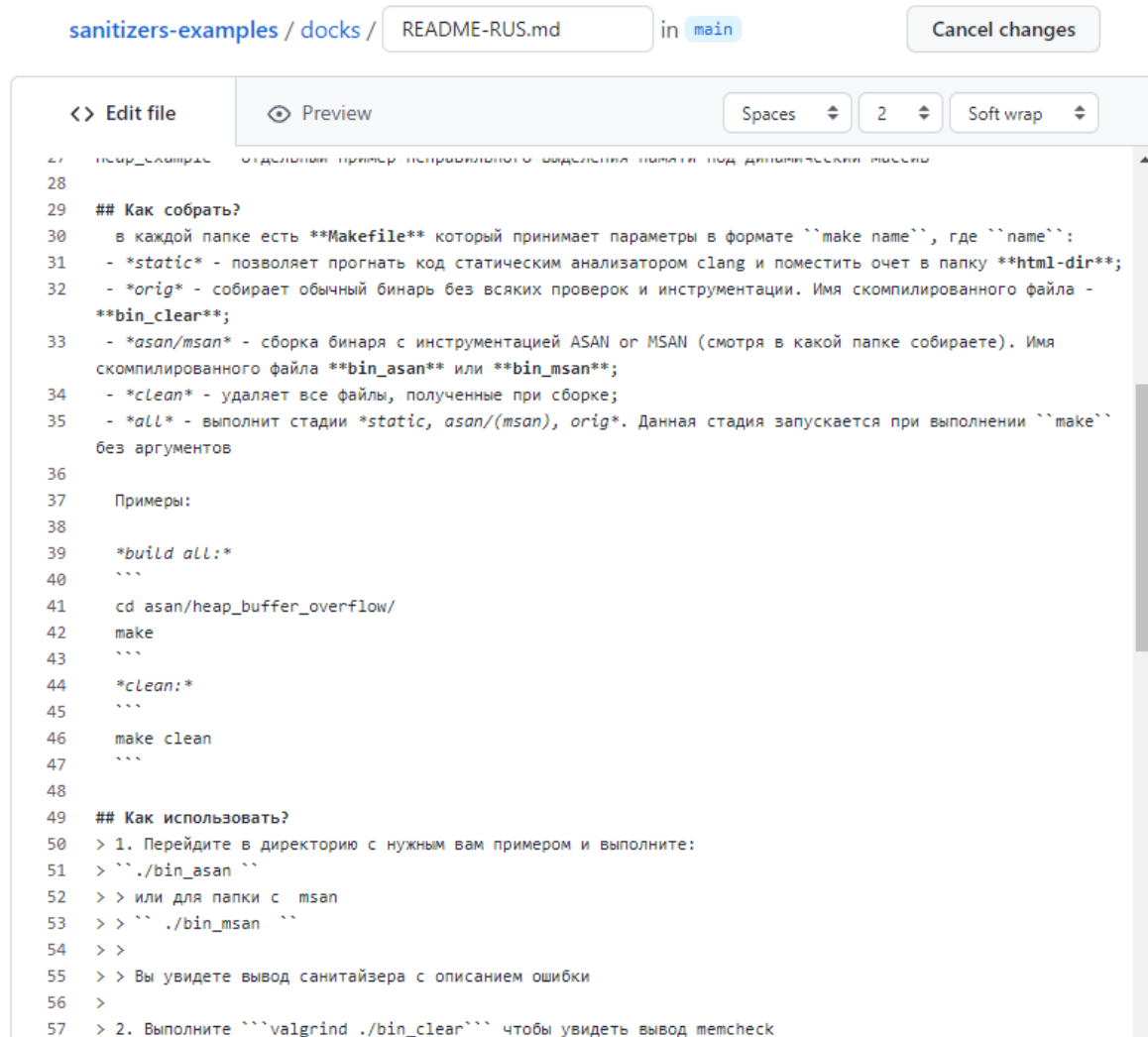
2. Выполните `valgrind ./bin_clear` чтобы увидеть вывод memcheck

3. После сборки (выполнения команды `make`) перейдите в **html\_dir** - в ней есть файл `.html` с отчетом статического анализатора. Его можно посмотреть через веб-браузер.

### Шаблон

можно использовать язык разметки **markdown**

В сыром виде выглядит примерно вот так



```
sanitizers-examples / docks / README-RUS.md in main Cancel changes

<> Edit file Preview Spaces 2 Soft wrap

27  #sanitizer_examples - отдельный пример, настроив который вы можете писать под различными процессорами
28
29  ## Как собрать?
30  в каждой папке есть **Makefile** который принимает параметры в формате make name, где name:
31  - *static* - позволяет прогнать код статическим анализатором clang и поместить отчет в папку **html-dir**;
32  - *orig* - собирает обычный бинарь без всяких проверок и инструментации. Имя скомпилированного файла -
**bin_clear**;
33  - *asan/msan* - сборка бинаря с инструментацией ASAN or MSAN (смотря в какой папке собираете). Имя
скомпилированного файла **bin_asan** или **bin_msan**;
34  - *clean* - удаляет все файлы, полученные при сборке;
35  - *all* - выполнит стадии *static, asan/(msan), orig*. Данная стадия запускается при выполнении make
без аргументов
36
37  Примеры:
38
39  *build all:*
40  ```
41  cd asan/heap_buffer_overflow/
42  make
43  ```
44  *clean:*
45  ```
46  make clean
47  ```
48
49  ## Как использовать?
50  > 1. Перейдите в директорию с нужным вам примером и выполните:
51  > ./bin_asan
52  > > или для папки с msan
53  > > ./bin_msan
54  > >
55  > > Вы увидите вывод санитайзера с описанием ошибки
56  >
57  > 2. Выполните valgrind ./bin_clear чтобы увидеть вывод memcheck
```

Как сделать также?

Смотрите здесь

<https://www.markdownguide.org/basic-syntax/>

Работает по аналогии с HTML, даже проще

## Ссылки

<https://git-scm.com/book/ru/v2>

Официальная книга - руководство по git от создателей и полностью на русском!

<https://www.atlassian.com/ru/git/tutorials/> Примеры

**А здесь интерактивный симулятор данной системы**

<https://learngitbranching.js.org>

<https://git-school.github.io/visualizing-git/>

### Markdown Синтаксис

<https://www.markdownguide.org/basic-syntax/>

## Советы

Есть GUI версии гит в том числе под Windows

Искать тут <https://git-scm.com/downloads/guis/>

От себя советую SourceTree (для Bitbucket) или GitHub Desktop (для github)

## Выводы

Ну пока собственно и все

Для самообразования нужно посмотреть систему pull request'ов и merge ветвей, но это оставляю на самостоятельное изучение