Шпаргалка: Git

Кратко: основные команды

Настроить Git

- `git version` — запросить версию Git;
- `git config --global user.name "твоё имя"` — прописать в настройках программы своё имя;
- `git config --global user.email your_email@yandex.com` — прописать в настройках программы свою почту;

Если перейти из командной строки в репозиторий и использовать `git config` с ключом `--local`, можно изменить свои данные для конкретного проекта.

- `git config --list` — вывести список всех настроек.

Команду `git config --list` можно вызвать из любого места.

Создать репозиторий

- `mkdir yandex` — создать папку yandex в текущей директории;
- `cd yandex` — перейти в папку yandex;
- `git init` — инициализировать Git в текущей директории.

Теперь yandex — это локальный репозиторий.

Изменить статус файла

- `git status` — отобразить состояние текущего репозитория;
- `git add main.py` — добавить файл main.py в индекс;
- `git add --all` — добавить в индекс все файлы из вложенных каталогов;
- `git add .` — то же, что и `git add --all`;
- `git commit -m "My first commit"` — сделать коммит с комментарием My first commit;

- `git commit --amend -m "First commit: new files added"` — добавить изменения к предыдущему коммиту с комментарием First commit: new files added;
- Чтобы игнорировать файл, добавь его название в файл `.gitignore`.
- `git mv converter.py data_converter.py` — переименовать файл `converter.py` в `data_converter.py`;
- `git rm data_converter.py` — удалить файл `data_converter.py` и перестать его отслеживать;

История коммитов

- `git log` — посмотреть информацию о коммитах;
- `git show c952d96` — посмотреть изменения в конкретном коммите. Семь символов после show — это контрольная сумма нужного коммита;
- `git show HEAD` — посмотреть последний коммит;
- `git reset 7639878` — откатить изменения до коммита с контрольной суммой `7639878`;
- `git reset HEAD` — откатить изменения всех файлов до предыдущего коммита;
- `git reset HEAD main.py` — откатить изменения в файле `main.py` до предыдущего коммита;

Ветвление

- `git branch` — просмотр текущей ветки;
- `git branch develop` — создать ветку с именем `develop`;

Используй `git branch` для создания веток, только если в основной ветке были коммиты.

- `git checkout develop` — перейти в ветку `develop`;
- `git checkout -b название_ветки` — создать ветку и сразу в неё перейти. Можно использовать, даже если в основной ветке не было коммитов;
- `git merge <название ветки, которую присоединяют>` — объединить эту ветку с текущей. Обрати внимание: коммиты переходят в ту ветку, на которой ты находишься, когда мёрджишь.

- `git push -u origin main` — запушить изменения из ветки `main` на GitHub.

Подробно

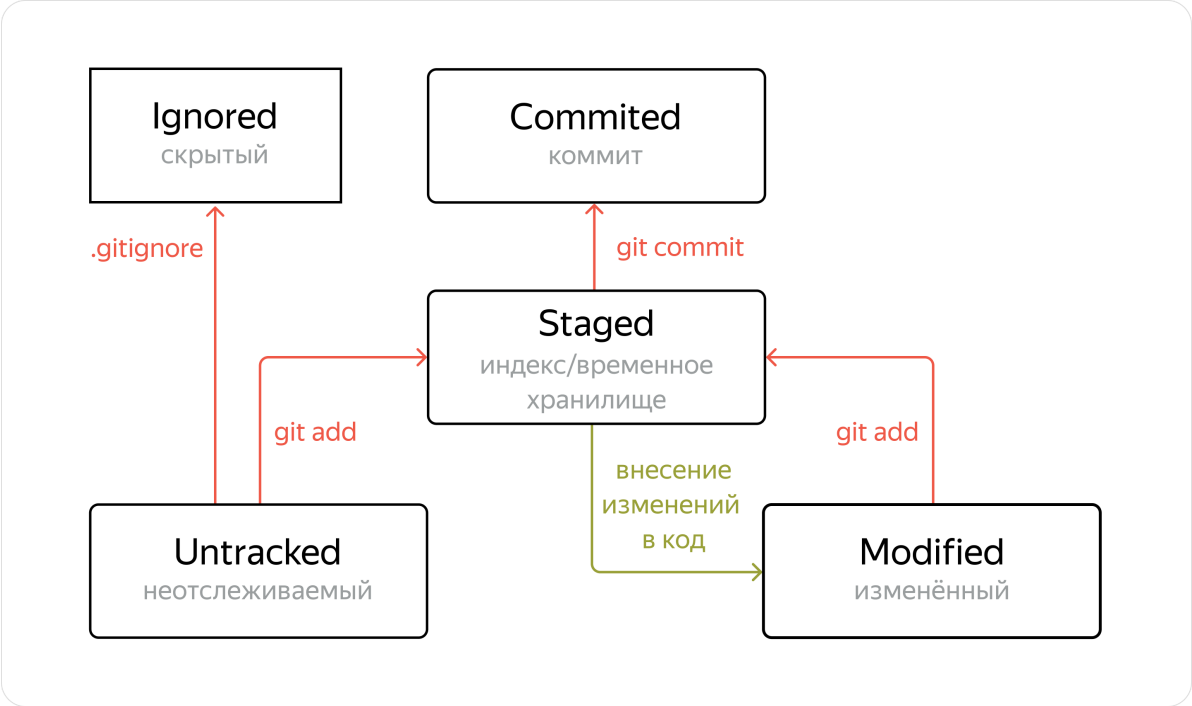
Это место, где хранятся данные, история их изменений и другая служебная информация. Проще говоря — там система собирает все сохранённые версии твоего проекта.

Обычно под каждый проект создают свой репозиторий.

Статусы файлов в Git

Версионный контроль в Git предполагает, что любой файл репозитория находится в одном из пяти состояний:

1. **Неотслеживаемый** (англ. untracked).
2. Добавленный в индекс, **индексируемый** (англ. staged, «выдвинутый на плацдарм»).
3. **Изменённый** (англ. modified).
4. **Сохранённый**, на жаргоне разработчиков «коммит» или «закоммиченный» (англ. committed, «брошенный в бой»).
5. **Игнорируемый**.



Неотслеживаемый

Когда в инициализированной папке появляется файл, он становится «неотслеживаемым». Ты можешь делать с файлом что-угодно — Git не будет учитывать изменения.

Индексируемый

Чтобы Git обратил внимание на этот файл, его нужно **добавить в индекс**. Так файл перейдёт в состояние «добавленный».

Изменённый

Если в добавленный файл внести изменения, он станет «изменённым».

Сохранённый

Когда ты закончишь редактировать файл, его нужно будет сохранить. Тогда ты сможешь вернуться к этой версии, если дальше что-то пойдёт не так. Сохранение называют **коммитом.**

Игнорируемый

Когда ты пишешь автотест или разрабатываешь проект, на твоём устройстве появляются **локальные** файлы. Это конфигурационные файлы, файлы с паролями и временные файлы.

Такие файлы не нужно отслеживать и коммитить. Их скрывают, или, иначе говоря, **игнорируют**. Скрытый файл находится в слепой зоне Git. Такой файл нельзя добавить в индекс и закоммитить. Git его просто не увидит.

Как узнать статус файла

Чтобы узнать состояние файлов в репозитории, используют команду `git status` (англ. *status*, «состояние»).

Например, выполнили в папке yandex команду `git status` и получили такое сообщение:

```
# ввели команду для отображения состояния репозитория
$ git status
# вот что Git сообщает в ответ
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  app.py
  main.py
nothing added to commit but untracked files present (use "git add" to track)
```

Git сообщает, что файлы `app.py` и `main.py` — в статусе неотслеживаемые. Система видит эти файлы, но они пока не добавлены во временное хранилище. Git не получил команду следить за их состоянием.

Как добавить файлы в индекс

Команда `git add` (англ. *add*, «добавить») добавляет файлы в индекс **Staging Area** (англ. «плацдарм», «место временного сосредоточения», «временное хранилище»).

После этой команды указывают имя файла, который предстоит отслеживать: `git add название_файла`.

```
$ git add main.py
# добавили в индекс файл main.py
```

```
$ git add --all
# добавили все файлы

$ git add .
# добавили все файлы
```

Файлы добавлены. Вот, как изменится репозиторий:

```
$ git status
# вызвали команду для отображения изменений

On branch main
No commits yet
Changes to be committed: # файлы ожидают коммита
  (use "git rm --cached <file>..." to unstage)
# добавились файлы в индекс
  new file:   app.py
  new file:   main.py
```

Теперь `app.py` и `main.py` добавлены в индекс и готовы к коммиту. У файлов в директории yandex статус **new file** (англ. «новый файл»).

Как игнорировать файл

В Git нет специальной команды для игнорирования файлов. Для этого в корне проекта создают скрытый файл `.gitignore`.

> 👉 Все файлы, названия которых начинается с точки, — скрытые.

Файл `.gitignore` — текстовый. Если внести в него название файла, Git не будет его отслеживать.

В `.gitignore` можно оставлять комментарии. Комментарий всегда начинают с символа `#`:

```
# игнорировать файл file_name.txt
file_name.txt

# игнорировать все файлы с расширением .doc
*.doc
```

Если файл находится не в корне проекта, нужно указать путь до него.

```
# игнорировать файл release.properties в директории target
target/release.properties
```

Как скрыть файл, если он уже индексируется

Если добавить файл в `.gitignore` после индексации, Git всё равно будет его отслеживать.

Чтобы скрыть отслеживаемый файл, его нужно вручную убрать из индекса. С этим поможет команда `git rm --cached <имя файла>`. Она отменяет добавление файла в индекс.

```
# создали файл example.txt
$ touch example.txt

# добавили файл в индекс
$ git add example.txt

# убрали его из индекса
$ git rm --cached example.txt
```

Как сохранить изменения

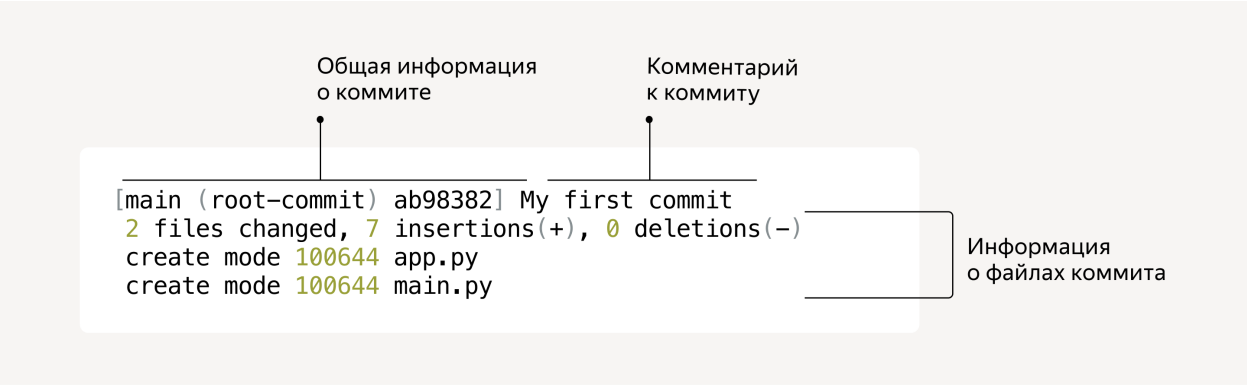
Когда все нужные файлы добавлены в индекс, можно делать коммит — зафиксировать все изменения в сохранённой версии и оставить комментарий.

Сделать это можно специальной командой `git commit` с ключом `-m` (от англ. *message*, «послание»). После ключа указывают комментарий в кавычках:

```
$ git commit -m "My first commit"
# сделали первый коммит
# текст комментария — "My first commit", в переводе "Мой первый коммит"
# комментарии лучше писать латиницей, чтобы они корректно отображались в командной строке
```

Команда выполнит коммит и выведет сообщение:

```
[main (root-commit) ab98382] My first commit
 2 files changed, 7 insertions(+), 0 deletions(-)
 create mode 100644 app.py
 create mode 100644 main.py
```



В первой строке ты видишь комментарий к коммиту `My first commit`. В квадратных скобках перед ним — информация о коммите:

- **main** — название ветви, в которой сделан коммит.
- **root-commit** (англ. «корневой коммит») — означает, что этот коммит самый первый. В этой папке такое сообщение ты больше не увидишь;
- **контрольная сумма** — первые семь символов уникального хеша, присвоенного коммиту. Здесь это `ab98382`.

Ниже информация о файлах коммита:

- изменено два файла — `2 files changed`;
- в общем счёте в них добавлено семь строк и ни одна пока не удалена: `7 insertions(+), 0 deletions(-)`;
- список зафиксированных файлов и тип действия с ними — создание (`create mode 100644 app.py` и т. д.).

Изменение последнего коммита

Если нужно добавить в репозиторий новый файл, можно не делать новый коммит. Достаточно добавить изменения к последнему коммиту.

Для этого есть опция `--amend` (англ. *amend*, «исправить»): `git commit --amend -m "Текст комментария"`.

Эта команда добавляет в последний коммит файлы и обновляет комментарий:

```
$ git commit -m "First commit: change main.py"
# сделали первый коммит, в кавычках написали комментраий

$ git add --all
# создали новые файлы и добавили их в индекс Git

$ git commit --amend -m "First commit: new files added"
# добавили эти файлы к предыдущему коммиту
```

Хеширование в Git

Хеширование — это преобразование данных в строку определённой длины, состоящую из цифр и букв. Выполняет его хеш-функция — специальный математический алгоритм.

Так данные получают уникальный идентификатор — **хеш-сумму**. Обычно её называют просто **хеш**.

Git хеширует данные с помощью алгоритма **SHA-1** (англ. Secure Hash Algorithm, «безопасный hash-алгоритм»).

Чтобы отслеживать файлы и папки, Git высчитывает хеш размером в 20 байт. У Git она состоит из 40 шестнадцатеричных символов — цифр и букв от a до f.

```
32d7ffadb1e3049d40c6fae4843bafe38d47346a # Пример хеш в Git
```

Когда ты что-то создаёшь или меняешь, Git делает снимок нового состояния. А затем присваивает ему уникальный номер — хеш. Одной операции соответствует один хеш.

Как вызвать историю коммитов

Часто нужно посмотреть историю сохранённых версий.

Для этого используют команду `git log` (англ. log, «регистрация»). Она выводит все коммиты и сообщает данные каждого из них:

```
# команда для просмотра истории коммитов
$ git log

commit c952d9626e27a4d6249faf368c7d22655476365c (HEAD -> main)
Author: Stas Basov <stasbasov@yandex.ru>
Date:   Fri Oct 11 16:00:04 2019 +0300

    added readme.txt

commit a22f3328b28ab901c12a4e7a5ce8fc543a6ed991
Author: Stas Basov <stasbasov@yandex.ru>
Date:   Fri Oct 11 15:58:36 2019 +0300

    added new file

# первая строка: присвоенный коммиту хеш
# вторая строка: имя и почта автора коммита. Необязательно это автор файла. Коммитер
# — тот, кто отдал команду git commit.
# третья строка: дата коммита
# и последняя строка: сообщение коммита
```

По умолчанию коммиты перечисляются в обратном порядке: от последнего к самому первому.

Уточнение изменений: `git show`

Случается, что в какую-то из версий финального файла добавили что-то не то. В Git можно посмотреть историю действий и вычислить этот коммит.

Чтобы просмотреть изменения в коммитах, понадобится команда `git show` (англ. show, «показать»). Последний коммит просматривается с указателем `HEAD`:

```
# эта команды выведет на экран последний коммит
$ git show HEAD

# результат:
diff --git a/readme.txt b/readme.txt
new file mode 100644
index 0000000..49861b8
--- /dev/null
+++ b/readme.txt
# в каком файле произвели изменения
@@ -0,0 +1 @@
# номер измененной строки
+Я изучаю Git. Он нужен, чтобы сохранять свой прогресс во время разработки
\ No newline at end of file
# добавленная строка
```

Изменения в любом из коммитов выводятся по **контрольной сумме** его хеша. Это первые семь символов после надписи `commit`:

```
# для просмотра всех коммитов
$ git log

commit c952d9626e27a4d6249faf368c7d22655476365c (HEAD -> main)
# чтобы посмотреть какие именно были изменения в коммите, скопируй первые
# семь символов после слова commit
```

```
# эта команда служит для просмотра изменений в коммите
$ git show c952d96
```

Отказ от изменений: `git reset`

Чтобы откатить проект до определённой версии, используют команду `git reset` (англ. reset, «сброс в исходное состояние»).

Чтобы вернуться к конкретному коммиту, к команде добавляют **контрольную сумму.** Это семь первых символов хеша нужного коммита.

```
$ git log
# посмотрели список коммитов

commit 76398788bf9c9aba93e4903ead47f1ee6d99976c (HEAD -> feature)
Author: Stas Basov <stasbasov@yandex.ru>
Date:   Thu Oct 25 17:13:01 2018 +0300
    Del all file
# первый коммит с контрольной суммой 7639878

commit 97a25f73849d758dca110bf4a70a29d6f42373ae (main)
Author: Stas Basov <stasbasov@yandex.ru>
Date:   Thu Oct 25 17:02:36 2018 +0300
    First commit: change main.py
# второй коммит

$ git reset 7639878
# откатились до первого коммита, ввели его контрольную сумму
```

Можно откатиться на один коммит назад в определённом файле. Для этого нужно указать его имя через `HEAD`:

```
$ git reset HEAD main.py
# откатили изменения до предыдушего коммита в файле main.py
```

Указатель `HEAD` означает текущий коммит.

Если не написать имя, сбрасываются все изменения последнего коммита:

```
$ git reset HEAD
# откатили изменения во всех файлах до предыдущего коммита
```

Ветвление

Это параллельная разработка по нескольким линиям.

Код можно дорабатывать по-разному: добавлять функционал, исправлять баги, тестировать, оптимизировать архитектуру. Чтобы одна задача не блокировала другую, используют ветвление. Оно позволяет разделить версии проекта и работать над ними по отдельности.

После инициализации репозитория создаётся основная линия разработки — ветка проекта. Это действие автоматически запускается командой `git init`.

Чтобы посмотреть, в какой ты сейчас ветке, понадобится команда `git branch`:

```
# команда для просмотра ветки
$ git branch

# перейди в файл, где будет указана твоя ветка. Чтобы выйти, нажми q
* main
(END)
```

Как создать новую ветку

Обычно в `main` находится стабильная версия проекта. Чтобы не испортить её в процессе доработки, создают вторую ветку. И уже там экспериментируют: дописывают или изменяют рабочий код.

Вторую ветку можно назвать как угодно. При выборе имени отталкиваются от задачи. Но в проектах, которые ты будешь сдавать на курсе, называй её `develop` (англ. «разработка»).

Новые ветки создают командой `git branch` (англ. branch, «ветка») с указанием названия ветки.

```
# создали ветку и назвали её develop
$ git branch develop
```

Как называть дополнительные ветки

Выбор имени зависит от подхода, который использует команда проекта. Наиболее популярны два: Trunk-based development и Git Flow.

Trunk-based development

В этом подходе разработчики активно работают с основной веткой. Когда нужно добавить фичу или исправить дефект, разработчик создаёт новую ветку, делает в ней необходимые изменения и сразу сливает её с основной веткой.

В этом подходе названия веток не так важны. Потому что сами ветки существуют недолго: от создания до слияния проходит мало времени. Имена дают по общему шаблону: `feature|fix/задача_описание`. Например, `feature/TASK-456_add-user-data` или `fix/TASK-684_unexpected-error`.

Git Flow

Самый популярный подход для ведения веток. Здесь одновременно существует много веток, и у каждой из них — своя задача.

Побочные ветки часто сливают между собой: далеко не все изменения сразу попадают в основную ветку.

Как правило, ветки называют так:

- `main` — основная ветка, в которой хранятся стабильные версии проекта;
- `develop` — ветка разработки, которую ведут от `main`. Может называться чуть иначе — например `dev`. Она дублирует код основной ветки. В ней разрабатывают новые фичи и исправляют найденные ошибки;
- `feature/задача_описание` — ветки, которые ведут от `develop`. В них разрабатывают новый функционал. Когда изменения готовы, ветку сливают с `develop` и удаляют;

- `hotfix/задача_описание` — ветки, которые ведут от `main`. В них разработчики исправляют ошибки и баги. Их сливают сразу в `main`, а потом удаляют;
- `release/номер` — ветки, которые ведут от `develop`. В них собирают все обновления и исправления проекта, а затем выпускают в релизе. Эти ветки сливают в `main`.

> 📌 В проектах, которые ты будешь сдавать на курсе, называй её `develop` (англ. «разработка»). Это общепринятое имя, и ревьюеры будут ожидать его.

Как переключиться с одной ветки на другую

Чтобы перейти в одну ветку из другой, нужна команда `git checkout название_ветки` (англ. checkout, в знач. «контроль»):

```
$ git checkout develop
# ввели команду

Switched to branch 'develop'
# получили сообщение о том, что переключились на ветку develop
```

Можно создать ветку и тут же переключиться на неё:

```
$ git checkout -b название_ветки
```

Слияние веток

Слияние веток — это зачисление в текущую ветку коммитов из одной или нескольких других веток.

Допустим, в ветке `main` хранится основной класс приложения `application.py`. *А* в ветке `develop` для него создаются тесты в новом файле `application_test.py`. Ветки нужно слить между собой, чтобы прогресс добавился в основную линию.

Если файл `application.py` менялся в ветке `develop`, при слиянии файл обновится в `main` $.

Для слияния веток используют команду `git merge` (англ. merge, «слияние»). Само это действие часто называют «смёрджить»:

```
$ git merge [название ветки, которую присоединяют]
```

Коммиты переходят в ту ветку, на которой ты находишься, когда мёрджишь. Обычно слияние происходит в пользу основной линии разработки. Поэтому перед мёрджем переключись в `main`:

```
$ git branch
 * develop
 main
# смотрим, в какой ветке находимся; это develop

$ git checkout main
# переключились в ветку main

$ git merge develop
# скопировали коммиты из ветки develop в ветку main
```

Конфликты

Если несколько коллег работают над разными частями кода в одном файле, при слиянии веток могут появиться конфликты.

Чтобы помочь пользователям справиться с этим, Git оставляет подсказки. Когда ветки сливаются с конфликтом, в консоли появляется сообщение:

```
$ git merge develop
# слили ветку develop в main

CONFLICT (content): Merge conflict in [название файла]
Automatic merge failed; fix conflicts and then commit the result.
```

Система объясняет, в каком файле произошли конфликты. Нужно их разрешить, а затем сделать коммит с изменениями.

Когда видишь сообщение о конфликтах, открывай проблемный файл в редакторе кода. Прямо в коде этого файла Git напишет замечание — в той строке, где текст отличается.

Нудно внести изменения в файл и сделать коммит. Тогда ветки сольются без проблем.

Удаление ветки

Когда ветка больше не нужна, её можно удалить командой `git branch -d имя ветки`.

```
$ git branch -d develop
# удалили ветку develop
```

Pull Request

Чтобы твой код добавили к основному коду проекта, нужно сделать pull request.

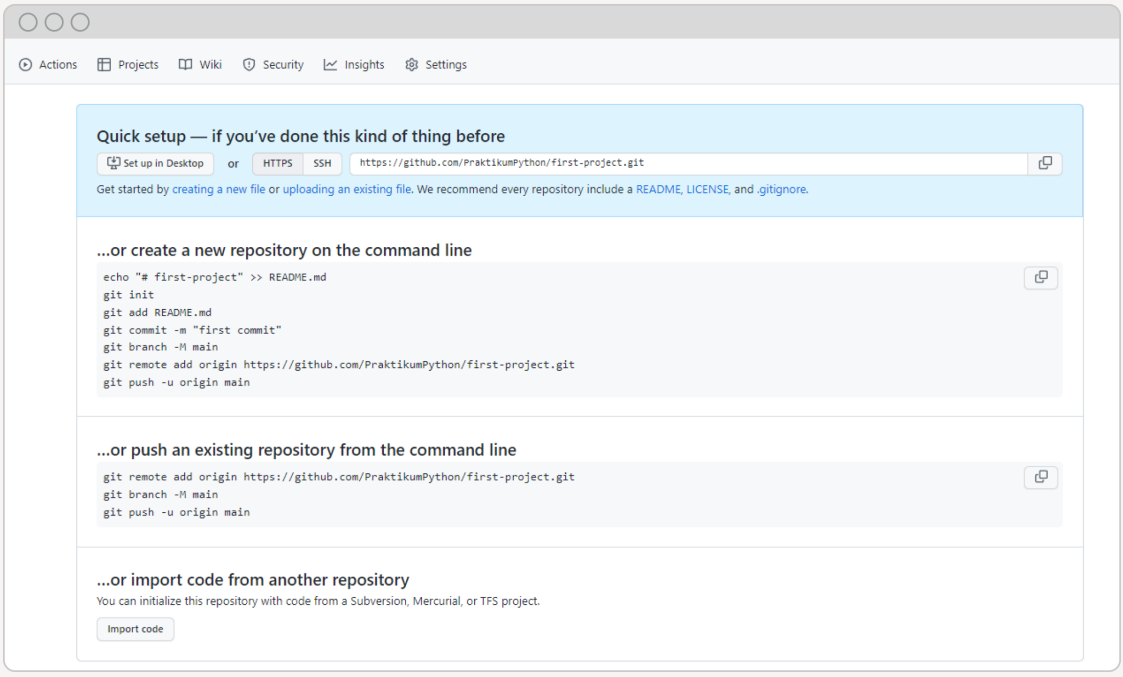
Как создать репозиторий на GitHub

1. Зайди в свой профиль.
2. Перейди во вкладку Repositories, нажми New.
3. Откроется окно Create a new repository. В поле Repository name введи имя проекта.
4. Нажми кнопку Create repository.

Как клонировать репозиторий

Чтобы с репозиторием можно было работать в IDE, его нужно склонировать на компьютер.

После создания репозитория, откроется такая вкладка:



Обрати внимание на блок Quick setup со ссылкой на твой проект.

Выбери HTTPS или SSH. Это зависит от того, какой способ ты хочешь использовать — копировать с через **https-ссылку** или с помощью **ssh-ключа**. Второй вариант сработает, только если ключ сгенерирован и привязан к твоему аккаунту GitHub.

Скопируй ссылку или путь, они понадобятся для клонирования.

Чтобы клонировать репозиторий с проектом:

1. Открой командную строку.
2. Используй команду `git clone`.

```
$ git clone https://github.com/PractikumPython/first-project.git
# скопировали репозиторий в текущую директорию с помощью ссылки
```

У команды `git clone` такая структура: `git clone — [адрес, откуда копирушь]—[путь до папки, куда копируешь]`. Если находишься в папке, куда клонируешь, не указывай путь.

После выполнения команды на компьютере появится папка с названием репозитория.

Как отправить изменения в GitHub

Нужно загрузить изменения на удалённый репозиторий. Для этого есть команда `git push` (англ. push, в значении «от себя»).

Готовый репозиторий скопирован с сервера, поэтому для первой публикации изменений нужно использовать команду `git push` с ключом `-u` и двумя аргументами:

- Первый аргумент — это имя сервера, с которого скопирован репозиторий `origin` (англ. «источник»).
- Второй аргумент — это имя основной ветки — `main`.

Ключ `-u` связывает локальную ветку с веткой удалённого репозитория. Этот ключ нужно использовать, если публикуешь новые ветки.

Опубликовать новую ветку нужно так:

```
$ git push -u origin main
```

Ты увидишь добавленные файлы в своём репозитории на GitHub.

Как сделать пул-реквест

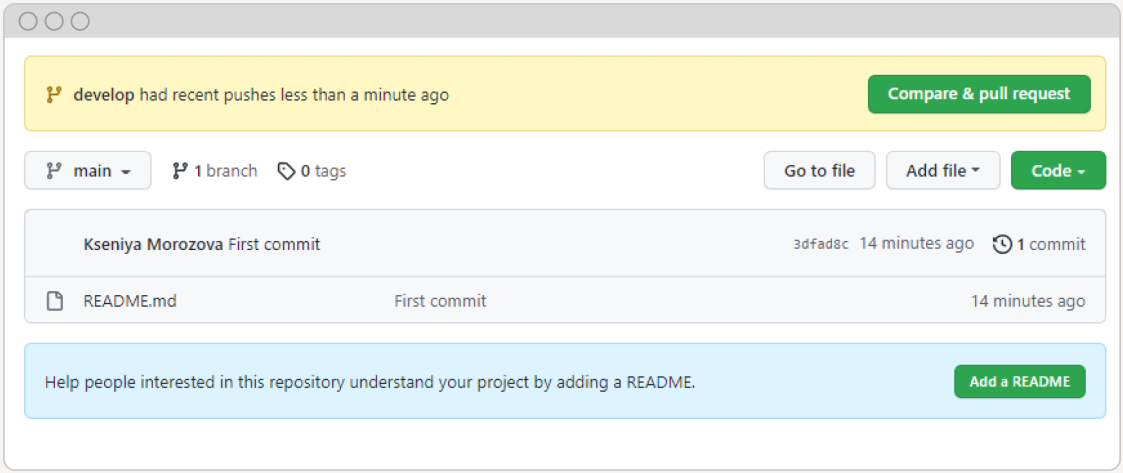
Ты не можешь просто взять и залить свой код в стабильную версию проекта, он должен пройти проверку. Для этого и нужен пул-реквест (pull request): это запрос на то, чтобы твой код объединили с основным.

Чтобы сделать пулл-ревест:

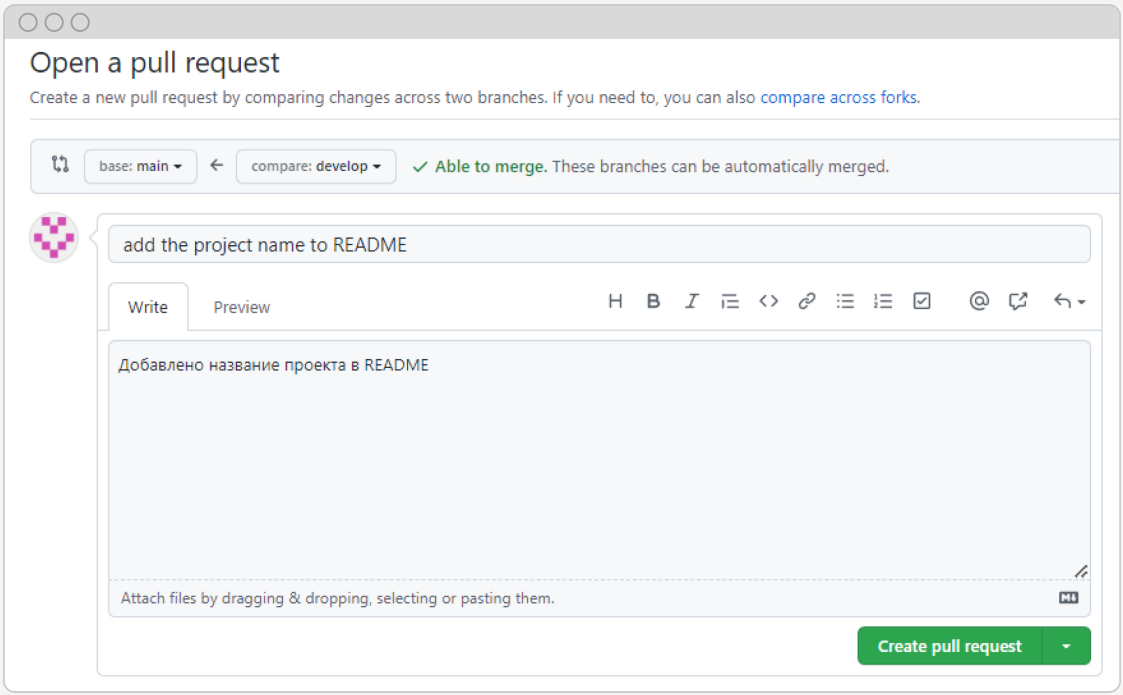
1. Создай новую ветку, например, `develop` и переключись на неё.
2. Внеси нужные изменения в файлы.
3. Убедись, что файлы индексируются. Сделай коммит с комментарием.
4. Запушь изменение в ветке develop на GitHub:

```
$ git push -u origin develop
```

5. Перейди в репозиторий на GitHub. Там ты увидишь кнопку Compare & pull request:



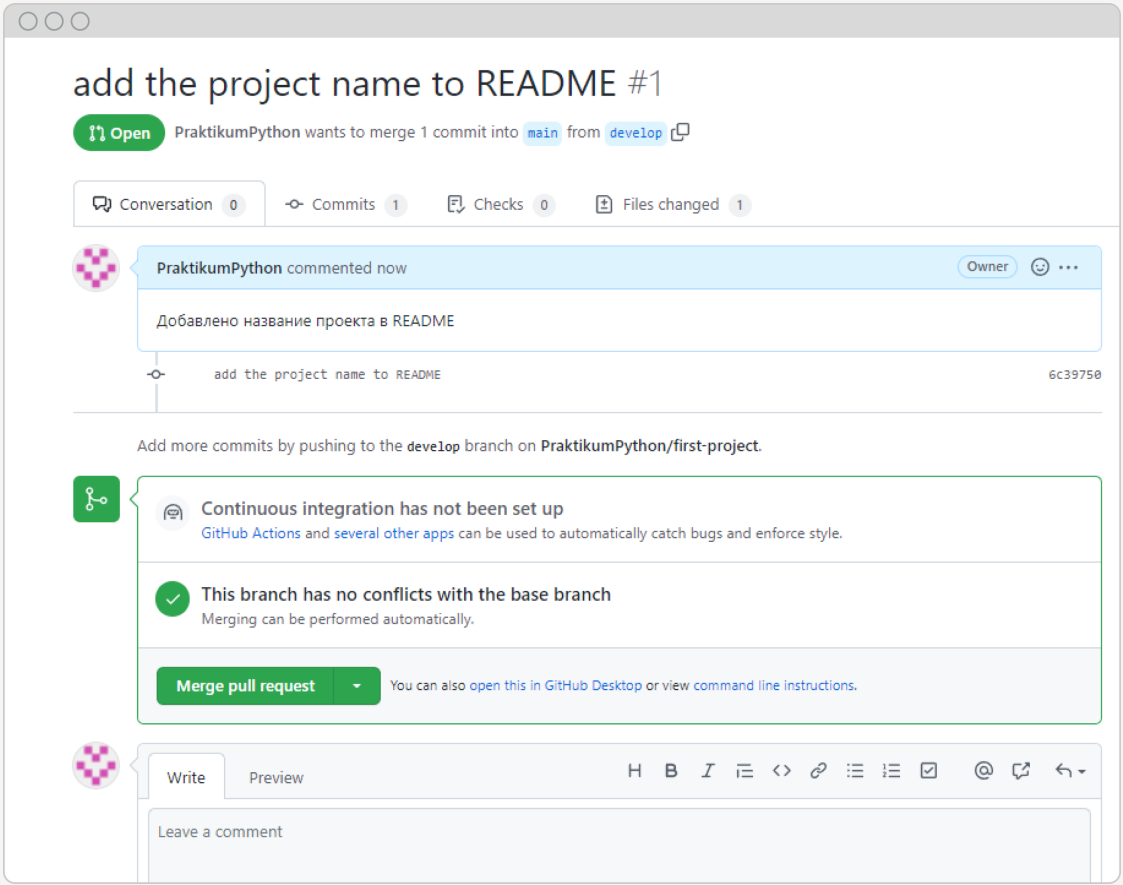
6. Заполни название и описание пул-реквеста:



7. В конце нажми зелёную кнопку Create pull request.

Этот пул-реквест получит владелец репозитория: твой наставник, тимлид или коллега.

Во вкладке Pull requests можно обсудить изменения:



Если есть замечания, их нужно исправить: сделать коммит и запушить изменения. Если замечаний нет, коллеги объединят твой код с кодом основной ветки. Для этого они нажмут Merge pull request или попросят тебя сделать это.