1. Введение

Git - программа, позволяющая контролировать изменения файлов проекта как для одиночной, так и совместной разработки кода.

Базовые возможности Git:

- Возврат к любой версии кода из прошлого
- Просмотр истории изменений и восстановление любых данных
- Совместная работа разработчиков без боязни потерять данные или затереть чужую работу

2. Установка и настройка

Установка:

```
# Mac
# https://brew.sh/
brew install git

# Ubuntu
sudo apt update  # на всякий случай смотрим новые версии
sudo apt install git-all

# Windows
установка Ubuntu on Windows и затем git
```

После установки git нужно зайти в терминал и проверить, что он работает:

```
git --version

git version 2.28.0
# Ваша версия может отличаться, но важно, чтобы она была не ниже 2.23.0
```

Если у вас установилась более старая версия git, и вы работаете в Ubuntu или Ubuntu on Windows, то попробуйте выполнить следующие команды:

```
sudo apt install software-properties-common
sudo add-apt-repository ppa:git-core/ppa
sudo apt update
sudo apt install git
```

После установки git нужно настроить. Для своей работы ему важно знать ваше имя и емейл. Эти данные подставляются в историю изменений. Только так можно узнать, кто и что сделал в проекте:

```
# Выполняется из любой директории
git config --global user.name "<имя фамилия>"
git config --global user.email "<ваш емейл>"
```

Установка редактора:

Рекомендуется ставить VSCode

Аккаунт на GitHub:

Для работы понадобится создать аккаунт на GitHub - это бесплатный (для одиночного использования) сервис, в котором хранят свои проекты большинство компаний и разработчиков

После создания аккаунта нужно добавить ssh-ключи на github.com. Ключи позволяют работать репозиториям с Github без необходимости постоянно вводить логин и пароль при синхронизации локального и удаленного репозитория (находящегося на Github).

Выполняется в два этапа. Сначала нужно сгенерировать ssh-ключи, а затем один из них (публичный) добавить в настройки Github.

```
# Создание ssh-ключей
ssh-keygen -t ed25519 -C "your_email@example.com"
# Дальше будет несколько вопросов. На все вопросы нужно нажимать Enter.

# Запуск агента ssh, который следит за ключами
eval "$(ssh-agent -s)"

# Добавления нового ssh-ключа в агент
ssh-add ~/.ssh/id_ed25519
```

Когда ssh-ключи созданы и добавлены в систему, можно приступать к интеграции с Github.

1. Выведите содержимое файла *~/.ssh/id_ed25519.pub* и скопируйте его:

   ```
   cat ~/.ssh/id_ed25519.pub
   ```

2. Добавьте ssh-ключ в аккаунт Github. При добавлении вас попросят назвать ключ. Напишите что-нибудь в стиле *home*.

3. Рабочий процесс

Git может отслеживать файлы проекта только в том случае, когда они помещены под контроль версий. Для этого нужно зайти в директорию проекта и выполнить команду инициализации `git init`. Проект может быть как новый, так и уже существующий. Процесс инициализации от этого не поменяется.

```
# Создаем новый проект
mkdir hexlet-git

# Переходим в созданную директорию
cd hexlet-git

# Выполняем инициализацию
git init

Initialized empty Git repository in /private/tmp/hexlet-git/.git/
```

Команда `git init` создает репозиторий — директорию *.git*, которая содержит все необходимые для работы git файлы.

С помощью команды `git status` можно посмотреть статус репозитория:

```
git status

On branch main
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

В этом выводе указано, что репозиторий пустой (No commits yet) и в него нечего добавить, так как нет новых или изменённых файлов. Добавим несколько файлов:

```
# Создаем файл README.md со строкой текста
echo 'Hello, Hexlet!' > README.md
echo 'Haskell Curry' > PEOPLE.md
```

Теперь снова смотрим на статус:

```
git status

# Часть вывода убрана
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    PEOPLE.md
    README.md
```

Git увидел, что в проекте появились новые файлы, о которых ему ничего не известно. Они помечаются как **неотслеживаемые** (untracked files). Git не следит за изменениями в таких файлах, так как они не добавлены в репозиторий. Добавление в репозиторий происходит в два шага. Первым шагом выполняется команда подготовки файлов git add <путь до файла>:

```
# Для каждого нового или измененного файла
git add README.md
```

Смотрим что произошло:

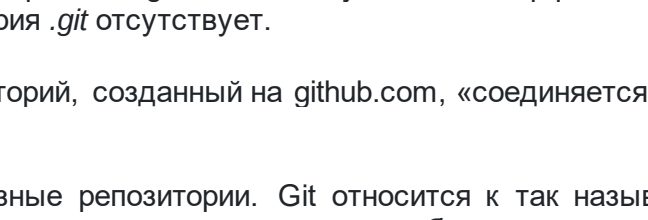
```
git status

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

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

Файл *README.md* теперь находится в состоянии "подготовлен к коммиту" или, другими словами, файлы попадают в индекс. Под коммитом понимается окончательное добавление в репозиторий, когда git запоминает файл навсегда и следит за всеми последующими изменениями.

Коммит — это операция, которая берёт все подготовленные изменения (они могут включать любое количество файлов) и отправляет их в репозиторий как единое целое.

```
git commit -m 'add README.md'

[main (root-commit) 3c5d976] add README.md
1 file changed, 1 insertion(+)
create mode 100644 README.md
```

Флаг -m означает message, то есть описание коммита. Коммит можно выполнять и без него, но тогда откроется редактор, в котором нужно будет ввести описание коммита. Рекомендуется делать осмысленные описания — это хороший тон.



Почему нельзя добавлять все изменённые файлы сразу в коммит? Такой процесс создан для удобства программистов. Дело в том, что во время разработки может меняться и добавляться много файлов. Но это не значит, что мы хотим добавить все эти изменения в один коммит.

Со смысловой точки зрения, коммит — это какое-то логически завершённое изменение внутри проекта. Его размер бывает очень маленьким, например, исправлением опечатки в одном файле, а иногда и большим, например, при внедрении новой функциональности. Главное в коммите — его **атомарность**, то есть он должен выполнять ровно одну задачу.

Теперь файл README.md находится внутри репозитория. Убедиться в этом можно, запустив команду git status:

```
git status

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

git status не выводит файлы, которые добавлены в репозиторий и не содержат изменений. При этом сам файл *README.md* находится внутри директории *hexlet-git*.

4. Интеграция с GitHub

Добавим наш репозиторий на github.com. Сохранённый репозиторий в любой момент можно извлечь и продолжить работу в нём с последнего добавленного туда коммита. Это полезно на случай, если мы случайно удалим или изменим локальный репозиторий так, что с ним станет невозможно работать.

1. Создайте репозиторий на Гитхабе. Назовите его hexlet-git. Важно, чтобы репозиторий создавался пустым, поэтому не нужно отмечать галочки, добавляющие файлы.

2. На странице репозитория вы увидите готовые команды для подключения созданного репозитория на Гитхабе к уже существующему репозиторию у вас на компьютере.

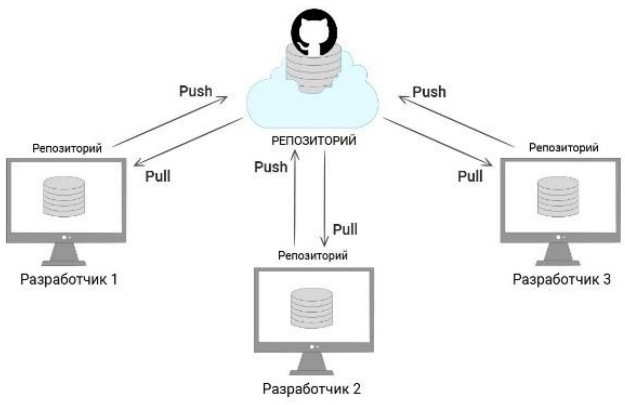
Выполните эти шаги:

```
# Подробнее эти команды мы разберём позже
git remote add origin git@github.com:<ИМЯ НА ГИТХАБЕ>/hexlet-git.git
git branch -M main
git push -u origin main
```

Обновите страницу с репозиторием на github.com. Изучите её интерфейс и содержимое репозитория. Обратите внимание на то, что директория *.git* отсутствует.

После этой команды репозиторий, созданный на github.com, «соединяется» с локальным репозиторием *hexlet-git*.

В действительности это разные репозитории. Git относится к так называемым распределённым системам контроля версий. У git нет какого-то центрального места, где бы лежал один главный репозиторий, а разработчики работали бы с ним со своих компьютеров. В git у каждого разработчика и даже на Github находится свой собственный полноценный репозиторий. Эти репозитории git связывает между собой общей историей и возможностью обмениваться изменениями. В примере выше именно команда git push отправляет изменения во вновь созданный репозиторий.



Прямо сейчас, после выполнения команд выше, локальный и удалённый репозиторий идентичны. Но в процессе работы они всё время будут расходиться, и программисты должны не забывать синхронизировать изменения: заливать в репозиторий новые коммиты и забирать оттуда коммиты, сделанные другими разработчиками.

Теперь не важно, какие изменения делаются в локальном репозитории, на Github все коммиты попадут только после команды `git push`. Не забывайте делать её, бывает такое, что разработчик случайно удаляет локальный репозиторий, забыв запушить изменения.

Далее мы попробуем скачать репозиторий с Гитхаба так, как будто у нас нет локальной копии. Для этого удалите директорию проекта *hexlet-git* с вашего компьютера.

Клонирование

Репозиторий, созданный на Github, - публичный. Любой человек может **клонировать** его к себе на компьютер и начать работать с ним так, как будто это его личный репозиторий. Единственное ограничение - он не сможет запушить изменения, так как Github не даёт напрямую менять чужие репозитории.

Клонирование - базовая операция при работе с удалёнными репозиториями. Проекты, над которыми работают программисты, всегда находятся в системах, подобных Github. Для работы с ними нужно клонировать репозиторий к себе на компьютер. Делается это с помощью команды `git clone`. Полную команду для клонирования можно получить на странице репозитория. Для этого нажмите большую кнопку Code, перейдите на вкладку SSH и скопируйте содержимое.

```
git clone git@github.com:<ИМЯ НА ГИТХАБЕ>/hexlet-git.git
cd hexlet-git
ls -la

# Если эта операция проходит первый раз
# То, вероятно, вы увидите такое подобное сообщение
The authenticity of host github.com cannot be established. RSA key fingerprint is SHA256: xxxxxxxxxx Are you sure you want to continue connecting (yes/no/[fingerprint])? yes Warning: Permanently added github.com (RSA) to the list of known hosts.
# Наберите yes и нажмите Enter
```

Мы получили точную копию репозитория, который был у нас до удаления директории *hexlet-git*.

Получение изменений с Github

Разработчики не только отправляют изменения на Гитхаб, но и забирают изменения оттуда. Чаще всего это изменения, сделанные другими разработчиками проекта, но необязательно. Бывает такое, что один разработчик работает над одним проектом с разных компьютеров, на каждом из которых своя собственная копия репозитория (git работает только так). В таком случае, перед началом работы нужно всегда выполнять команду `git pull --rebase`, которая скачивает из внешнего репозитория новые коммиты и добавляет их в локальный репозиторий.

Обычно, в статьях пишут, что достаточно вызывать *git pull*, но это может приводить к созданию ненужных merge-коммитов, ухудшающих историю изменений. Правильная работа с *git pull* требует знания таких вещей, как ветвление и *git rebase*.

Итог

Мы создали репозиторий с несколькими коммитами. Этот репозиторий добавлен на Гитхаб и может быть склонирован для дальнейшей разработки. Какую пользу из git мы можем извлечь к текущему моменту? У нас есть запасная копия (бекап) кода на сайте Github. Как минимум, не страшно потерять код. Теперь его легко восстановить при случае и поделиться с другими.

Отдельно стоит сказать, что Гитхаб это хоть и самая популярная, но не единственная площадка для хостинга репозиториев. Кроме него особенно известны Bitbucket и Gitlab. Последний можно даже поставить к себе на сервер и "хостить" репозитории внутри своей компании, что многие и делают по соображениям безопасности или экономии.

5. Рабочая директория (Working Directory)

После клонирования *hexlet-git* внутри мы можем увидеть директорию *.git* и добавленные нами файлы. Что произойдет если попробовать удалить один из файлов?

```
rm PEOPLE.md
git status

On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:    PEOPLE.md

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

Git сообщает о том, что файл был удалён, и предлагает команды для восстановления или коммита изменений.

Откуда он знает, что файл был удалён?

Внутри директории проекта мы видим файлы проекта с одной стороны и директорию *.git* с другой. Репозиторием является именно директория *.git*. Она хранит всю информацию о том, какие были изменения, а также сами изменения. А вот всё, что находится снаружи, это так называемая рабочая директория (working directory). Эти файлы (и директории, если они есть) извлекаются из *.git* в момент клонирования. Каждый раз, когда мы производим изменения в рабочей директории, git сравнивает изменённые файлы с файлами внутри *.git*, то есть их состоянием на момент последнего коммита. Если есть изменения относительно последней зафиксированной версии, то git сообщает нам об этом в выводе `git status`.

В этом очень легко убедиться, если последовать совету git в выводе выше и восстановить удалённый файл:

```
git restore PEOPLE.md
git status

On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

# Сам файл вернулся на место таким, каким он был на момент последнего коммита
```

Можно удалить все файлы в рабочей директории и потом без проблем их восстановить. Так мы достигаем важной цели — делаем возможным быстрое восстановление последней версии кода, если изменения, которые мы делали, нас больше не устраивают. Или мы можем закоммитить их, если это нужно:

```
rm PEOPLE.md
# Любое изменение обязательно добавлять в индекс
git add PEOPLE.md
git commit -m 'remove PEOPLE.md'

[main e15afd2] remove PEOPLE.md
1 file changed, 1 deletion(-)
delete mode 100644 PEOPLE.md
# Теперь этот файл пропал из рабочей директории
```

Обратите внимание на важную деталь. Независимо от того, удаляем мы файл, добавляем или меняем, процедура выполнения коммита не меняется. После изменений всегда делается `git add`, который подготавливает изменение к коммиту (а не добавляет файл!), и после этого выполняется коммит.

Кстати, у git есть команда `git rm`, которая объединяет в себе удаление и подготовку к коммиту:

```
git rm PEOPLE.md
# равносильно rm + git add
```

6. Анализ сделанных изменений

Во время разработки программистам часто приходится останавливаться и анализировать изменения, которые они сделали с последнего коммита. Потребность смотреть изменения становится очевидной, если представить себе, что такое работа над реальным проектом

Анализировать изменения важно даже в небольших проектах. Прямо сейчас во время разработки этого курса изменилось несколько файлов и `git status` выглядит так:

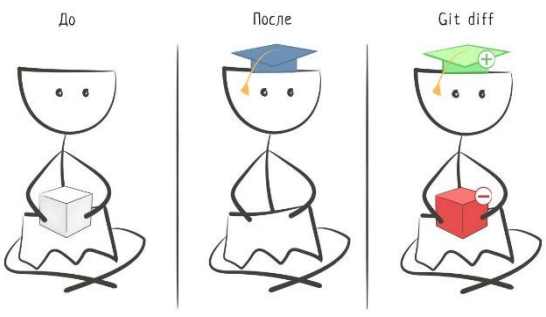
```
git status

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   300-working-directory/README.md

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:   100-intro/README.md
    modified:   250-github/README.md
    modified:   300-working-directory/README.md
    modified:   300-working-directory/spec.yml
    modified:   350-changes/README.md
```

Воспроизведем подобную ситуацию в нашем проекте. Выполним следующий код в репозитории *hexlet-git*:

```
echo 'new line' >> INFO.md
echo 'Hello, Hexlet! How are you?' > README.md

git status

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:   INFO.md
    modified:   README.md

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

Изменились оба файла. В один мы добавили строчку, в другом заменили. Как теперь посмотреть эти изменения? Для этого в git есть команда `git diff`, которая показывает разницу между тем, что было и что стало:

```
git diff

diff --git a/INFO.md b/INFO.md
index d5225f8..40f51f1 100644
--- a/INFO.md
+++ b/INFO.md
@@ -1 +1,2 @@

 git is awesome!
+new line

diff --git a/README.md b/README.md
index ffe7ece..00fd294 100644
--- a/README.md
+++ b/README.md
@@ -1 +1 @@

-Hello, Hexlet!
+Hello, Hexlet! How are you?
```



Вывод команды поначалу может смутить. Здесь довольно много служебных данных, за которыми уже идут изменения. Вывод `git diff` содержит именно те строки, которые изменились (и иногда строки вокруг измененных для удобства анализа), а не файлы целиком. Слева от них ставится знак "-", если строка была удалена, и "+" для добавленных строк.

Сама команда не просто выводит на экран разницу между файлами, но и запускает пейджер — специальную программу, которая позволяет перемещаться по выводу и искать внутри него нужные данные. Для перемещения вниз по дифу нужно нажать `f`, для перемещения наверх — `b` или `u`. Для выхода из режима просмотра нажмите `q`.

По умолчанию `git diff` показывает изменения только для тех модифицированных файлов, которые ещё не были добавлены в индекс. Подразумевается, что добавленные в индекс файлы смотреть не нужно, ведь мы их уже подготовили к коммиту. В реальности же часто хочется и, более того, нужно увидеть эти изменения. Для этого нужно запустить команду вывода дифа с флагом `--staged`:

```
# Выведет все изменения сделанные в рабочей директории
# которые были добавлены в индекс
git diff –staged
```

`git diff` - команда, которую нужно обязательно запускать перед каждым коммитом. Она позволяет проанализировать добавляемые изменения и исправить возможные ошибки. Иногда программисты по ошибке добавляют в коммит то, что туда не должно попасть.

7. Анализ истории изменений (коммитов)

Git Log

Самая простая аналитика выполняется командой git log. Она показывает список всех выполненных коммитов, отсортированных по дате добавления (сверху самые последние):

```
git log

# Ниже неполный вывод истории проекта

commit 5120bea3e5528c29f8d1da43731cbe895892eb6d
Author: tirion <tirion@got.com>
Date:   Thu Sep 17 18:04:19 2020 -0400

    add new content

commit e6f625cf8433c8b1f1aaed58cd2b437ec8a60f27
Author: tirion <tirion@got.com>
Date:   Thu Sep 17 16:14:09 2020 -0400

    add INFO.md

commit 273f81cf2117044f1973ea80ce1067a94bea3f80
Author: tirion <tirion@got.com>
Date:   Thu Sep 17 16:08:39 2020 -0400

    remove NEW.md

# Этот вывод показывается через пейджер
```

Из этого вывода мы можем узнать кто, когда и какие коммиты делал. Если коммиты оформлены хорошо, то по их описанию уже многое понятно.

У команды git log есть полезный флаг -p, который сразу выводит диф для каждого коммита:

```
git log -p
# Тут все коммиты с полным дифом
# Мотать вперед f, мотать назад b
# Выйти из режима просмотра — q
```

Git Show

У каждого коммита есть идентификатор (хеш), уникальный набор символов. С помощью хеша можно посмотреть все изменения, сделанные в рамках одного коммита:

```
git show 5120bea3e5528c29f8d1da43731cbe895892eb6d
# Тут выводится диф между этим коммитом и предыдущим

diff --git a/INFO.md b/INFO.md
index d5225f8..40f51f1 100644
--- a/INFO.md
+++ b/INFO.md
@@ -1 +1,2 @@
 git is awesome!
+new line
diff --git a/README.md b/README.md
index ffe7ece..00fd294 100644
--- a/README.md
+++ b/README.md
@@ -1 +1 @@
-Hello, Hexlet!
+Hello, Hexlet! How are you?
# То есть то, что было изменено этим коммитом
```

Хеши коммитов в git очень длинные, и ими бывает неудобно пользоваться. Поэтому разработчики git добавили возможность указывать только часть хеша. Достаточно взять первые 8 символов и подставить их в ту команду, которая работает с коммитами:

```
git show 5120bea3
```

Чаще всего вам не придётся высчитывать их самим, большая часть команд git выводит хеш коммита в сокращенном варианте, облегчая его использование. Такое упрощение хорошо работает, потому что даже первые 8 символов будут всегда уникальными.

Git Blame

А что, если мы не знаем коммита, но нам интересно, кто последним менял конкретную строку в файле? Для этого подойдет команда git blame <путь до файла>. Эта команда выводит файл и рядом с каждой строкой показывает того, кто её менял и в каком коммите.

```
git blame INFO.md

e6f625cf (tirion 2020-09-17 16:14:09 -0400 1) git is awesome!
5120bea3 (tirion 2020-09-17 18:04:19 -0400 2) new line
```

Важно помнить, что изменение строчки — не то же самое, что её написание. Вполне возможно, что программист исправил небольшую опечатку, а саму строку написал кто-то до него. В любом случае, имея такой вывод, уже легко пойти дальше и изучить конкретный коммит.

Git Grep

Команда git grep ищет совпадение с указанной строкой во всех файлах проекта. Это очень удобная команда для быстрого анализа из терминала. Она удобнее обычного grep, так как знает про игнорирование и не смотрит в директорию *.git*, а ещё умеет искать по истории:

```
git grep line

INFO.md:new line

# Флаг i позволяет искать без учета регистра
git grep -i hexlet

README.md:Hello, Hexlet! How are you?

# Поиск в конкретном коммите
git grep Hexlet 5120bea3

# Поиск по всей истории
# rev-list возвращает список хешей коммитов
git grep Hexlet $(git rev-list --all)
```

GitHub

В простых ситуациях анализировать проект можно прямо на Гитхабе. Он позволяет просматривать историю коммитов, изменения в конкретном коммите и многое другое.

8. Отмена изменений в рабочей директории

Одна из ключевых возможностей Git — "откат" любых сделанных изменений буквально одной командой. Такое практически невозможно сделать без использования системы контроля версий. Только если помнить все изменения наизусть. Здесь мы поговорим про откат изменений, которые сделаны в рабочей директории, но ещё не попали в коммит.

Важно! Откат незакоммиченных изменений безвозвратен. Не существует никакой физической возможности получить эти изменения обратно, поэтому будьте крайне осторожны.

Неотслеживаемые файлы

Самая простая ситуация. Вы добавили новые файлы в репозиторий (или сгенерировали их как-то) и поняли, что они вам не нужны. В этом случае можно выполнить очистку:

```
mkdir one
touch two

git status

On branch main
Your branch is up to date with 'origin/main'.

# Пустые директории в git не добавляются в принципе.
# Физически директория one находится в рабочей директории,
# но её нет в git, и он её игнорирует
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    two

# Выполняем очистку
# -f – force, -d – directory
git clean -fd

Removing one/
Removing two
```

Изменённые файлы в рабочей директории

Для отмены изменений в таких файлах используется команда `git restore`. Причём git сам напоминает об этом при проверке статуса:

```
echo 'new text' > INFO.md
git status

On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  # Ниже написано, как отменить изменение
  (use "git restore <file>..." to discard changes in working directory)
    modified:   INFO.md

# Отменяем
git restore INFO.md
```

Изменённые подготовленные к коммиту

С файлами, подготовленными к коммиту, можно поступить по-разному. Первый вариант — отменить изменения совсем, второй — отменить только индексацию, не изменяя файлы в рабочей директории. Второе полезно в том случае, если изменения нам нужны, но мы не хотим их коммитить сейчас.

```
echo 'new text' > INFO.md
git add INFO.md
git status

On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   INFO.md
```

И здесь снова помогает Git. При выводе статуса он показывает нужную нам команду для перевода изменений в рабочую директорию:

```
git restore --staged INFO.md
git status

On branch main
Your branch is up to date with 'origin/main'.

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:   INFO.md
```

Теперь, если нужно, можно выполнить `git restore` и окончательно отменить изменения в выбранных файлах.

9. Отмена коммитов

В git практически всегда есть способ восстановить или изменить любые коммиты. На крайний случай спасет повторный `git clone`.

Что делать, если коммит уже сделан, но он нас по каким-то причинам не устраивает? Ситуаций может быть много:

- Забыли добавить в коммит нужные файлы
- Изменения нужно «откатить», чтобы доработать
- Изменения больше не актуальны, и их нужно удалить
- Изменения были сделаны по ошибке, и их нужно отменить

Git по большей части система «только вперёд». Правильный подход при работе с гитом — создание нового, а не изменение старого. Все ситуации, описанные выше, можно решить новым коммитом, изменяющим код в нужном направлении. Это не только удобно, но и безопасно. Изменение истории коммитов — операция опасная и чревата проблемами при синхронизации с удалёнными репозиториями.

Git revert

Самая простая ситуация — отмена изменений. Фактически она сводится к созданию ещё одного коммита, который выполняет изменения противоположные тому коммиту, который отменяется. Руками создавать подобный коммит довольно сложно, поэтому в git добавили команду, автоматизирующую откат. Эта команда называется `git revert`:

git revert B


```
# Этой команде нужен идентификатор коммита
# Это коммит, которым мы удалили файл PEOPLE.md
git revert aa600a43cb164408e4ad87d216bc679d097f1a6c
# После этой команды откроется редактор, ожидающий ввода описания коммита
# Обычно сообщение revert не меняют, поэтому достаточно просто закрыть редактор
[main 65a8ef7] Revert "remove PEOPLE.md"
 1 file changed, 1 insertion(+)
 create mode 100644 PEOPLE.md
# В проект вернулся файл PEOPLE.md

git log -p

commit 65a8ef7fd56c7356dcee35c2d05b4400f4467ca8
Author: tirion <tirion@got.com>
Date:   Sat Sep 26 15:32:46 2020 -0400

    Revert "remove PEOPLE.md"

    This reverts commit aa600a43cb164408e4ad87d216bc679d097f1a6c.

diff --git a/PEOPLE.md b/PEOPLE.md
new file mode 100644
index 0000000..4b34ba8
--- /dev/null
+++ b/PEOPLE.md
@@ -0,0 +1 @@
+Haskell Curry
```

Команда *revert* может "отменять" не только последний коммит, но и любой другой коммит из истории проекта.

Git reset

Иногда удалить нужно только что сделанный по ошибке коммит. Конечно, и в этом случае подходит `git revert`, но так загрязняется история. Если этот коммит сделан был только сейчас и ещё не отправлялся на Github, то лучше сделать так, как будто бы этого коммита не существовало в принципе.

Git позволяет удалять коммиты. Это опасная операция, которую нужно делать только в том случае, если речь идет про новые коммиты, которых нет ни у кого, кроме вас.

Если коммит был отправлен во внешний репозиторий, например, на Github, то менять историю ни в коем случае нельзя, это сломает работу у тех, кто работает с вами над проектом.

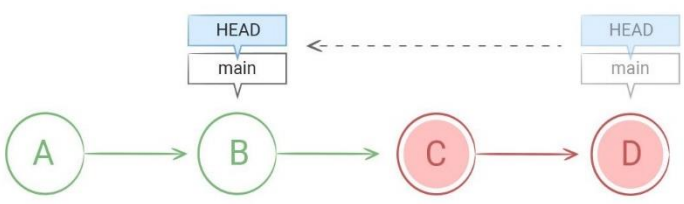
Для удаления коммита используется команда `git reset`.

```
# добавляем новый коммит, который мы сразу же удалим
echo 'test' >> INFO.md
git add INFO.md
git commit -m 'update INFO.md'

[main 17a77cb] update INFO.md
 1 file changed, 1 insertion(+)
 # Важно, что мы не делаем git push

git reset --hard HEAD~

HEAD is now at 65a8ef7 Revert "remove PEOPLE.md"

# Если посмотреть git log, то последнего коммита там больше нет
```

`git reset` — мощная команда, имеющая множество различных флагов и способов работы. С её помощью удаляются или отменяются (без удаления) коммиты, восстанавливаются файлы из истории и так далее. Работа с ней относится к продвинутому использованию git, здесь же мы затрагиваем только самую базу.

Флаг `--hard` означает полное удаление. Без него `git reset` отменит коммит, но не удалит его, а поместит все изменения этого коммита в рабочую директорию, так что с ними можно будет продолжить работать. `HEAD~` означает "один коммит от последнего коммита". Если бы мы хотели удалить два последних коммита, то могли бы написать `HEAD~2`.

git reset --hard HEAD~2



HEAD (голова) — так обозначается последний сделанный коммит.

Если не указывать флаг `--hard`, то по умолчанию подразумевается флаг `--mixed`. В таком варианте *reset* отправляет изменения последнего коммита в рабочую директорию. Затем их можно исправить или отменить и выполнить новый коммит.

```
echo 'no code no pain' > README.md
git add README.md
git commit -m 'update README.md'

[main f85e3a6] update README.md
 1 file changed, 1 insertion(+)

# Теперь откатываем последний коммит
git reset HEAD~

Unstaged changes after reset:
M   README.md

git status

On branch main
Your branch is up to date with 'origin/main'.

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:   README.md
```

Последнего коммита больше не существует, но изменения, сделанные в нём, не пропали. Они находятся в рабочей директории для дальнейшей доработки.

10. Изменение последнего коммита

Крайне часто разработчики делают коммит и сразу же понимают, что забыли добавить часть файлов через git add. Оставшуюся часть изменений можно дослать следующим коммитом либо, если изменения ещё не были отправлены во внешнюю систему, можно добавить изменения в текущий коммит. Для этого во время коммита добавляется флаг --amend:

```
echo 'experiment with amend' >> INFO.md
echo 'experiment with amend' >> README.md
git add INFO.md
# Забыли сделать подготовку README.md к коммиту
git commit -m 'add content to INFO.md and README.md'

[main 256de25] add content to INFO.md and README.md
 1 file changed, 1 insertion(+)

git status

On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

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:   README.md

# Увидели, что забыли добавить файл. Добавляем

git add README.md
git commit --amend
# После этой команды откроется редактор, ожидающий ввода описания коммита
# Здесь можно поменять сообщение или выйти из редактора, оставив старое

[main d96151a] add content to INFO.md and README.md
 Date: Sat Sep 26 16:02:07 2020 -0400
 2 files changed, 2 insertions(+)

git status

On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

В реальности --amend не добавляет изменения в существующий коммит, этот флаг приводит к откату коммита (через reset) и выполнению нового коммита с новыми данными. Поэтому мы и видим ровно один коммит, хотя команда git commit выполнялась два раза (первый раз — когда сделали ошибочный коммит).

Для того, чтобы не открывался редактор для ввода описания коммита к команде git commit --amend можно добавить опцию --no-edit. В этом случае описание коммита не изменится.

11. Индекс

Индекс в Git — это специальная промежуточная область, в которой хранятся изменения файлов на пути от рабочей директории до репозитория. При выполнении коммита в него попадают только те изменения, которые были добавлены в индекс.

Понятие индекса в Git появилось не случайно. Даже когда разработчик работает над одной задачей, по пути он натыкается на разные места в коде, которые либо плохо оформлены, либо содержат ошибки, либо должны быть исправлены в соответствии с какими-то новыми требованиями. И в большинстве ситуаций совершенно нормально исправлять эти недочеты, что все и делают. В итоге в рабочей директории появляется множество разных исправлений, которые частично относятся к выполняемой задаче, а частично содержат множественные исправления, напрямую не связанные с основными изменениями. В чём здесь проблема?

Если делать ровно один коммит, включающий в себя и основную задачу, и дополнительные исправления, то появляется несколько неприятных побочных эффектов. Во-первых, сложнее смотреть историю. Коммит начинает содержать совершенно несвязанные изменения, которые отвлекают во время ревью (проверки чужого кода).

```
# Обычно в таких коммитах встречается условие И в описании
# Показатель того, что в рамках одного коммита сделано несколько изменений
git commit -m 'add new feature and fix something'
```

Во-вторых, что вероятно даже важнее, откат коммита по любым причинам приведет к тому, что откатятся правки, которые всё равно нужно будет делать.

Именно здесь помогает индекс. Его наличие позволяет меньше переживать на тему того, как сформируется коммит.

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

```
git add somefile
git commit -m 'add somefile'
```

Если речь идет про один-два файла, которые нужно закоммитить прямо сейчас, то можно сделать проще. Команда git commit принимает на вход аргументы — пути до файлов. Она автоматически добавляет эти файлы в индекс и затем в коммит. Данный подход работает только с уже отслеживаемыми файлами.

```
echo 'new data' >> INFO.md
# Не нужно явно вызывать git add
git commit INFO.md -m 'update INFO.md'
```

Иногда бывает наоборот — мы исправили много файлов и хотим добавить их в коммит сразу все. Тогда поможет точка:

```
# Добавляет абсолютно все изменения рабочей директории в индекс
git add .
```

Команда выше очень опасна. С ее помощью крайне легко закоммитить много лишнего, особенно если не помнить про необходимость перед коммитом смотреть git diff --staged.

Ну и совсем страшная, но полезная команда — это коммит с одновременным добавлением всего в индекс:

```
# Флаг -a автоматически добавляет все изменения рабочей директории в индекс
git commit -am 'do something'
```

С другой стороны, нередко разные изменения делаются в одних и тех же файлах. То есть изменения в этих файлах по-хорошему должны находиться в разных коммитах. И даже такое можно сделать с помощью Git. Для этого подходит команда git add -i, которая показывает измененные куски файлов и спрашивает, что с ними сделать. С помощью этой команды можно очень точно выбрать то, что должно попасть в коммит, а что нет. Ее использование обычно показывает хороший уровень владения Git.

12. Перемещение по истории

Git позволяет не только просматривать историю, но и перемещаться по ней, загружая в рабочую директорию состояние кода на момент выполнения любого коммита.

```
# Показывает сокращенный вывод
git log --oneline

fc74e2d update README.md
65a8ef7 Revert "remove PEOPLE.md"
5120bea add new content
e6f625c add INFO.md
273f81c remove NEW.md
aa600a4 remove PEOPLE.md
fe9893b add NEW.md
3ce3c02 add PEOPLE.md
3c5d976 add README.md
```

Переключимся на момент, когда был выполнен коммит с сообщением *add INFO.md*. Для этого используется команда git checkout <хеш коммита>:

```
git checkout e6f625c

Note: switching to 'e6f625c'.

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.

Or undo this operation with:

git switch -
```

Выполните команду выше (хеш вашего коммита может отличаться) и изучите рабочую директорию. Вы увидите, что пропала часть изменений из-за возврата в прошлое. Сами изменения никуда не делись, и мы снова можем вернуться на последний коммит следующей командой:

```
# Что такое main, мы поговорим позже
git checkout main
```

Переключившись в нужный коммит, можно не только изучить содержимое репозитория, но и забрать какие-то изменения, которые были удалены, но снова понадобились для работы. Для этого достаточно их скопировать, переключиться на последний коммит и вставить в нужный файл.

Где я

Переключение по коммитам отражается только на содержимом рабочей директории. Больше нигде не видно, где мы находимся. Из-за этого немало программистов, забыв, где они находятся, начинают работать и очень удивляются, когда не получается выполнить коммит.

Самый простой способ узнать место нахождения — вызвать команду git branch. В обычной ситуации, когда мы находимся на последнем коммите, git покажет такой вывод:

```
git branch

# О том, что такое main, мы поговорим позже
* main
```

Но если прямо сейчас загружен коммит из прошлого, то вывод станет таким:

```
* (HEAD detached at e6f625c)
  main
```

Такой способ проверки текущего местоположения требует постоянного внимания. Нужно не забывать его использовать и, конечно же, все забывают это делать. Гораздо надежнее и удобнее вывести текущее местоположение прямо в командной строке. Например, так:

```
# Если на последнем коммите
hexlet-git git:(main)

# Если на коммите из прошлого
hexlet-git git:(e6f625c)
```

Именно так делают большинство профессиональных разработчиков. Как добиться такого вывода? Ответ на этот вопрос зависит от используемого командного интерпретатора. В Bash вывод местоположения происходит благодаря редактированию переменной окружения $PS1.

13. Понимание Git

Давайте выведем коммиты нашего проекта *hexlet-git* в специальном виде, который активируется опцией --graph:

```
git log --graph

# Неполный вывод, чтобы не отвлекаться от сути
* commit e7bb5e51f96e572084f6c04ba3312e32ce6b8c0f (HEAD -> main, origin/main, origin/HEAD)
|
|   update README.md
|
* commit 65a8ef7fd56c7356dcee35c2d05b4400f4467ca8
|
|   Revert "remove PEOPLE.md"
|
|   This reverts commit aa600a43cb164408e4ad87d216bc679d097f1a6c.
|
* commit 5120bea3e5528c29f8d1da43731cbe895892eb6d
|
|   add new content
|
* commit e6f625cf8433c8b1f1aaed58cd2b437ec8a60f27
|
|   add INFO.md
|
* commit 273f81cf2117044f1973ea80ce1067a94bea3f80
|
|   remove NEW.md
```

Обратите внимание на полоску слева. Она отражает связи между коммитами. Каждый новый коммит базируется на коде предыдущего коммита. С точки зрения информатики коммиты выстраиваются в так называемый односвязный список. В таком списке каждый элемент ссылается на предыдущий. Последний элемент при этом называется головой списка (head по-английски).

В Git элементы списка — это сами коммиты. И так же, как в односвязном списке, новый коммит (как элемент) имеет ссылку на старый (предыдущий), а предыдущий — на свой предыдущий, и так далее до первого коммита, который никуда не ссылается, так как он первый. Понятие «голова списка» (HEAD) в git присутствует явно и активно используется для разных операций. Например, удаление последнего коммита выглядит так:

```
# HEAD~1 означает: взять голову и удалить, начиная от нее, один коммит
# То есть только последний коммит
git reset --hard HEAD~1
```

Сам список коммитов тоже имеет название, вы его уже видели: это — *main*. В терминологии Git такой список называется веткой (branch). Именно поэтому команда для показа текущего местоположения в истории называется git branch:

```
git branch

* main
```

Строго говоря, ветка в Git — это просто подвижный указатель на один из коммитов. При каждом новом коммите указатель сдвигается вперёд автоматически.

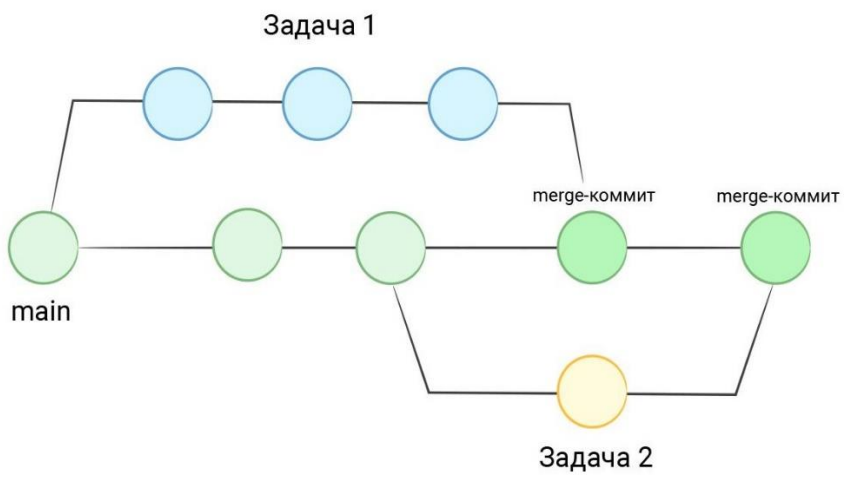
Основная «работа» Git — формирование односвязного списка, состоящего из коммитов. Это — ключевое знание для понимания того, что такое Git. Подавляющее большинство команд Git — это всего лишь небольшие программы, которые умеют ходить по этому списку и извлекать нужную информацию. Фактически всё сводится к блужданию по этому списку вперед-назад и его изменению, включая добавление новых коммитов.

Ветки

Если смотреть дальше, то мы увидим, что цепочка коммитов в Git это не просто односвязный список — это множество односвязных списков (направленный ациклический граф), переплетенных вместе.

Представьте себе, что в один момент времени два разных человека должны делать какие-то длинные задачи, требующие нескольких дней разработки или даже больше. Main-ветка в таком случае должна оставаться рабочей, то есть коммитить промежуточные изменения в нее нельзя, так как они могут сломать её. Но коммитить всё равно надо, так как просто небезопасно копить изменения в рабочей директории, не отправляя их в Git. Что делать в такой ситуации?

Гит позволяет *отпочковываться* от основного списка, формируя «ветки». То есть создается отдельный список коммитов, который идет мимо main. В конце разработки все коммиты из такой ветки вливаются обратно в main.



14. Игнорирование файлов (Gitignore)

В процессе работы над любым проектом в директории с кодом создаются файлы, которые не являются частью исходного кода. Все эти файлы можно условно разделить на несколько групп:

Инструментарий

- Служебные файлы, добавляемые операционной системой (.DS_Store в Mac)
- Конфигурационные и временные файлы редакторов (например, .idea, .vscode)

Временные файлы

- Логи. В них содержится полезная информация для отладки, которая собирается во время запуска и работы приложения
- Кеши. Файлы, которые нужны для ускорения разных процессов

Артефакты

- Результаты сборки проекта. Например, после компиляции или сборки фронтенда
- Устанавливаемые во время разработки зависимости (например, node_modules, vendor)
- Результаты выполнения тестов (например, информация о покрытии кода тестами)

Всё это в обычной ситуации не должно попадать в репозиторий. Как правило, эти файлы не несут никакой пользы с точки зрения исходного кода. Они создаются либо автоматически (кеши, логи), либо по запросу (например, скачиваются зависимости или собирается проект). Главная проблема с этими файлами в их постоянном изменении при, как правило, очень больших размерах. Если добавлять их в репозиторий, то практически в каждом коммите, кроме изменений исходного кода, будет и пачка изменений в этих файлах. Читать историю таких коммитов крайне сложно.

Git позволяет гибко настраивать игнорирование определенных файлов и директорий. Делается это с помощью файла .gitignore, который нужно создать в корне проекта. В этот файл добавляются файлы и директории, которые надо игнорировать. Например:

```
# В этом файле можно оставлять комментарии
# Имя файла .gitignore
# Файл нужно создать самостоятельно

# Каждая строчка — это шаблон, по которому происходит игнорирование

# Игнорируется файл в любой директории проекта
access.log

# Игнорируется директория в любой директории проекта
node_modules/

# Игнорируется каталог в корне рабочей директории
/coverage/

# Игнорируются все файлы с расширением sqlite3 в директории db,
# но не игнорируются такие же файлы внутри любого вложенного каталога в db
# например, /db/something/lala.sqlite3
/db/*.sqlite3

# игнорировать все .txt файлы в каталоге doc/
# на всех уровнях вложенности
doc/**/*.txt
```

Git поддерживает игнорирование файлов, но сам его не настраивает. Для игнорирования файлов и директорий, нужно создать файл .gitignore в корне проекта и добавить его в репозиторий.

```
touch .gitignore
# добавляем в файл правила игнорирования по примеру выше
git add .gitignore
git commit -m 'update gitignore'
```

Как только *.gitignore* создан и в него добавлен какой-то файл или директория, игнорирование заработает автоматически. Все новые файлы, попадающие под игнорирование, не отобразятся в выводе команды git status.

Иногда бывает такое, что программист случайно уже добавил в репозиторий файл, который нужно проигнорировать. В этой ситуации недостаточно обновить правила игнорирования. Дополнительно придется удалить файл или директорию из git с помощью git rm и закоммитить.

15. Stash

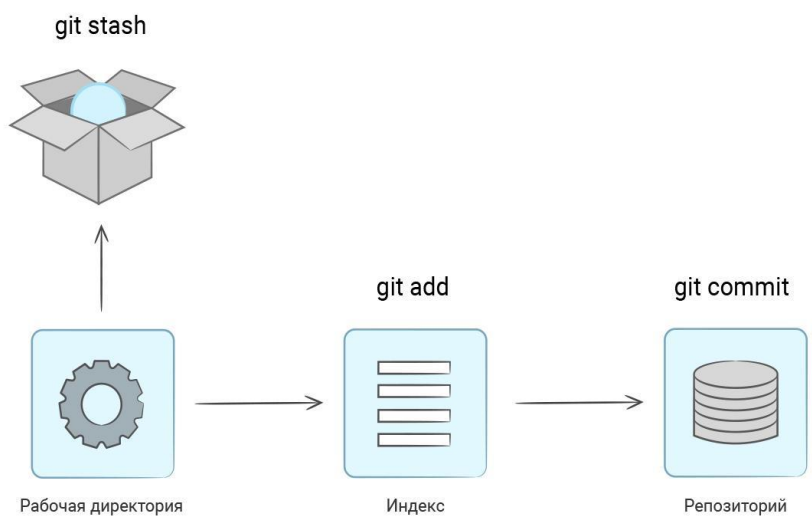
Представьте себе ситуацию. Вы работаете над какой-то важной задачей и исправили довольно много файлов. В этот момент появляется срочная задача — сделать какое-то изменение в исходном коде, не связанное с тем, над чем вы сейчас работаете. Ваши изменения ещё не готовы, и они не должны попасть в репозиторий. Что делать?

В самом простом случае, если ваши изменения не пересекаются с изменениями по срочной задаче, вы можете внести исправления, добавить их в индекс, закоммитить и запушить. Но обычно это неудобно и не всегда возможно. А если изменения нужно делать в тех файлах, с которыми вы работаете прямо сейчас?

Подобная ситуация встречается регулярно и, к счастью, она легко решается. В git существует набор команд, позволяющий «прятать» изменения в рабочей директории и восстанавливать их при необходимости.

```
touch FILE.md
git add FILE.md
git status

On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   FILE.md

# Прячем файлы. После этой команды пропадут все изменённые файлы
# независимо от того, добавлены они в индекс или нет
git stash

Saved working directory and index state WIP on main: e7bb5e5 update README.md

git status

On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

git stash не удаляет файлы, они попадают в специальное место внутри директории *.git* «на сохранение». Эта команда не трогает новые файлы, так как они ещё не являются частью репозитория.



После выполнения всех нужных изменений на чистой рабочей директории можно вернуть спрятанные изменения с помощью команды git stash pop:

```
# Восстанавливаем
git stash pop

On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   FILE.md

Dropped refs/stash@{0} (b896d4a0126ef4409ede63857e5d996953fe75c5)

# Проверяем
git status

On branch main
Your branch is up to date with 'origin/main'.

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

Файлы вернулись в том виде, в котором они попали в стеш (stash).

Stash в Git работает по принципу стека. Он позволяет сохранить внутрь любое количество изменений и восстановить их в обратном порядке:

```
git stash

# изменяем файлы
git stash

# Вернутся последние изменения
git stash pop

# Вернутся предпоследние изменения
git stash pop
```

16. Открытые проекты (Open Source)

Open Source Software (OSS; разг. опенсорс) — это программное обеспечение (ПО), код которого открыт (его можно посмотреть), и на него почти всегда можно влиять.

Программисты каждый день пользуются программным обеспечением с открытым исходным кодом. К такому ПО относится как прикладное, например, VSCode или Git, так и огромное число библиотек в их коде, а также практически все современные языки программирования. Та скорость, с которой могут разрабатываться современные проекты, во многом возможна как раз благодаря программам и библиотекам с открытым исходным кодом.

- Linux (> 12 миллионов строк кода)
- Chromium (> 16 000 000)
- Git
- jQuery

Откуда вообще берутся опенсорс-проекты и почему они так популярны? Почти всегда такой код появляется как побочный продукт разработки других проектов. Например, в процессе разработки Linux появилась необходимость в удобной программе для управления версиями. Так появился Git. Но почему его код был открыт? Ответ достаточно простой. Удачные проекты привлекают множество разработчиков, которые помогают им развиваться. Они пишут отчёты об ошибках, присылают исправления и даже становятся полноправными разработчиками. И всё это бесплатно. Разработчики удовлетворяют своё эго, радуются тому, что создают нечто новое и вообще помогают этой вселенной.

Как побочный эффект, такие разработчики гораздо легче находят работу и, в целом, имеют более прокачанные навыки кодирования, чем те, кто не работает с открытыми проектами.

Если вы вернетесь в прошлое и посмотрите, сколько опенсорс-проектов создавалось тогда и сколько там было задействовано людей, то вы увидите огромный разрыв с современным положением дел. Сложный процесс включения в разработку, сложный процесс принятия изменений — вот неполный перечень проблем, ожидавших тех, кто желал стать участником этого движения. Люди выступали с докладами на конференциях, где час (Карл!!!) описывали правила принятия людей и кода от них в проект.

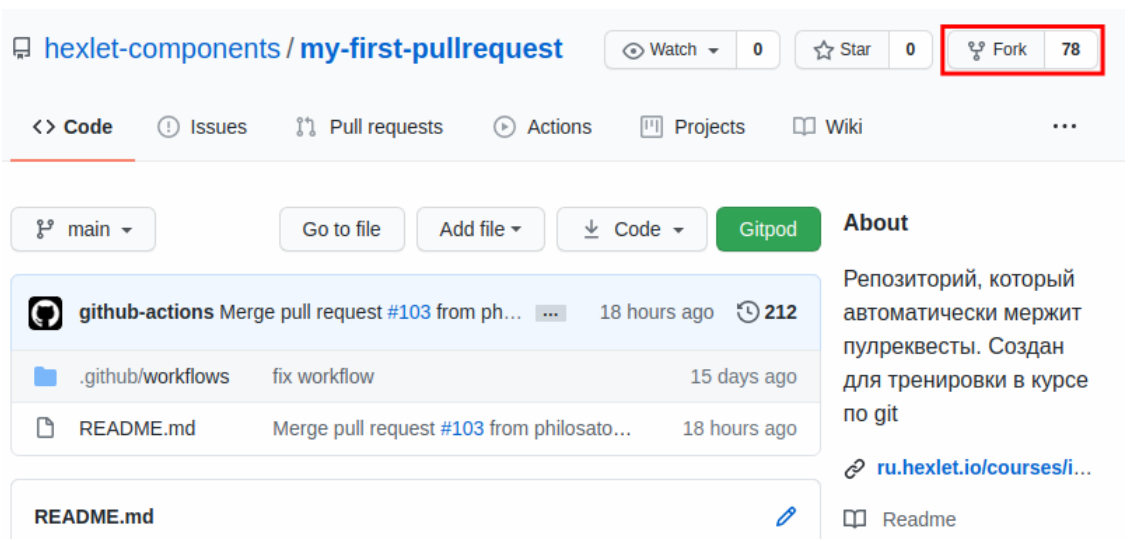
Сейчас начать делать опенсорс стало делом пары минут, а чтобы начать где-то участвовать, не нужно прилагать практически никаких усилий. А случилось это благодаря двум составляющим: Git и GitHub.

Один из важнейших механизмов GitHub — запрос на включение изменений (pull request; разг. пулреквест). Именно он позволяет легко и непринуждённо вливаться в разработку любых проектов. Допустим, работая с определённой библиотекой, мы заметили ошибку в коде или документации.

Пройдёмся по шагам, которые необходимо выполнить для исправления этой ошибки. Наша конечная цель в том, чтобы разработчики библиотеки приняли наш код.

Клонирование

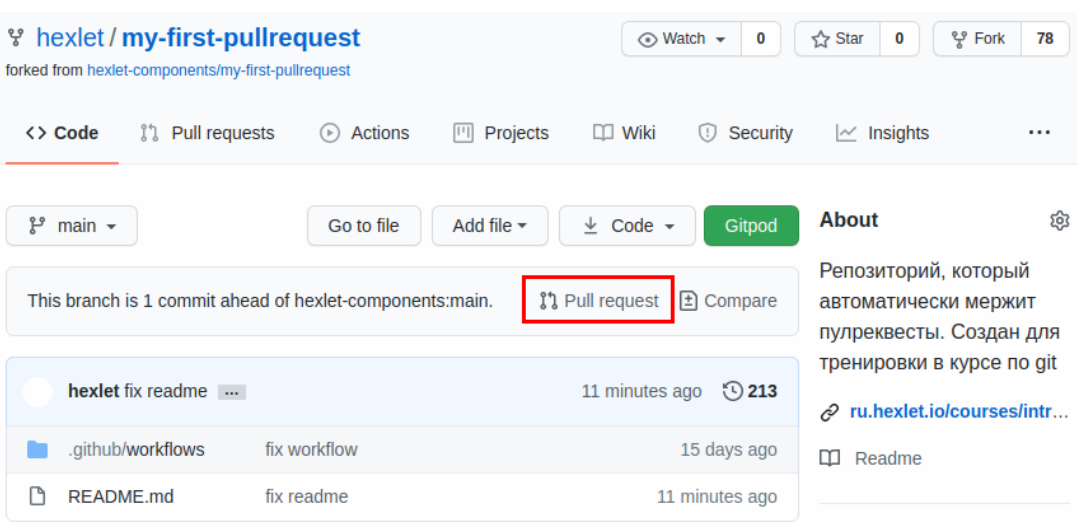
Первым шагом необходимо создать копию репозитория в своём аккаунте. Делается это буквально одной кнопкой «Fork» на странице репозитория.



После этого действия в вашем аккаунте окажется репозиторий с таким же именем. GitHub знает, что это копия оригинального репозитория, и помечает его особым образом. Дальше всё как обычно. Мы клонируем репозиторий на компьютер и производим необходимые изменения. Хорошей практикой считается делать изменения в отдельной ветке (обычно созданной от ветки main).

Запрос на включение изменений кода

После того, как изменения залиты на GitHub, в его интерфейсе произойдут изменения. На странице склонированного репозитория появится кнопка «pull request».



Если её нажать, то откроется страница, на которой можно указать название пулреквеста и его описание. После отправки пулреквеста в исходном (оригинальном) репозитории на странице «Pull requests» отобразится ваш запрос. Теперь остаётся ждать, когда разработчики библиотеки либо его примут, либо отклонят, либо зададут уточняющие вопросы (тогда с пулреквестом, возможно, потребуется ещё поработать).

Исправления прямо на Гитхабе

В более простых ситуациях, когда достаточно исправить текст или опечатку, Github позволяет сделать пулреквест прямо из своего интерфейса. Для этого достаточно открыть любой файл проекта и нажать на иконку редактирования. После завершения редактирования Github сам предложит создать пулреквест с этим изменением.