Git — это система контроля версий, которая помогает отслеживать изменения в проекте. Этот инструмент можно использовать как для индивидуальной, так и для командной работы.

Git позволяет сохранять изменения локально и при необходимости возвращаться к предыдущим версиям проекта. Также можно создать удалённую копию на хостинг-платформе, которая работает с Git, и поделиться результатом с другими.

**Инициализируем репозиторий**

Сделать папку репозиторием — git init

Чтобы Git начал отслеживать изменения в проекте, папку с файлами этого проекта нужно сделать **Git-репозиторием** (от англ. *repository* — «хранилище»). Для этого следует переместиться в неё и ввести команду git init (от англ. **init***ialize* — «инициализировать»).

Например, создайте папку first-project и сделайте её Git-репозиторием: перейдите в неё с помощью команды cd и выполните git init.

$ cd ~/dev/first-project *# перешли в нужную папку*

$ git init *# создали репозиторий*

Также git init выведет сообщение вида Initialized empty Git repository in <\*ваша папка с проектом\*>/.git/ (англ. «инициализирован пустой Git-репозиторий в <\*ваша папка\*>/.git/»). В подпапке .git Git будет хранить всю служебную информацию.

### «Разгитить» папку, если что-то пошло не так, — rm -rf .git

Если вы случайно сделали Git-репозиторием не ту папку, её можно «разгитить». Для этого нужно удалить скрытую подпапку .git.

$ cd <папка с репозиторием> # перешли в папку

$ rm -rf .git # удалили подпапку .git

Разберём подробнее, что такое -rf:

* ключ -r (от англ. recursive — «рекурсивно») позволяет удалять папки вместе с их содержимым;
* ключ -f (от англ. force — «заставить») избавит вас от вопросов вроде «Вы точно хотите удалить этот файл? А этот? И этот тоже?».

Будьте осторожны: в подпапке .git хранится история изменений. Если удалить .git, то вся история проекта будет стёрта без возможности восстановления — останется только последняя версия файлов.

Проверить состояние репозитория — git status

После инициализации репозитория first-project запустите команду git status (от англ. *status* — «статус», «состояние») — она показывает текущее состояние репозитория.

Команда git status выведет:

название текущей ветки: On branch master или On branch main;

сообщение о том, что в репозитории ещё нет коммитов: No commits yet;

сообщение, которое говорит: «чтобы что-нибудь закоммитить (то есть зафиксировать), нужно сначала это создать» — nothing to commit (create/copy files and use "git add" to track).

**Добавляем файлы в репозиторий**

Подготовить файлы к сохранению — git add

$git add --all - сделать все файлы в папке отслеживаемыми в репозитории

$git add 'имя файла.расширение' - сделать конкретный файл отслеживаемым в репозитории

$git add . - добавить всю текущую папку в репозиторий (точка позволяет обратиться к текущей папке)

Добавим в репозиторий два файла. Например, файл todo.txt, в котором будет список дел, и readme.txt для информации о проекте.

**Почему именно текстовые файлы?**

Чаще всего в Git хранят исходный код программ. Но с точки зрения системы контроля версий любой код — это всего лишь текстовый файл. Какой-то специальной логики для хранения программ в ней нет.

Так что Git вполне может быть повседневным инструментом как разработчиков, которые пишут на разных языках программирования, так и администраторов и DevOps, которые, например, работают над файлами конфигураций, или технических писателей, которые создают документацию, и так далее.

Чтобы сделать наши примеры универсальными, мы будем использовать именно текстовые файлы.

Создайте файлы todo.txt и readme.txt в папке first-project и запустите git status, чтобы посмотреть, что изменилось.

$ touch todo.txt

$ touch readme.txt

*# создали файлы todo.txt и readme.txt*

$ git status *# проверили статус*

Git сообщит, что в папке first-project есть untracked files (от англ. *track* — «следить», *untracked* — «неотслеженный», «неотслеживаемый») — ещё не отслеживаемые файлы readme.txt и todo.txt.

Состояние untracked значит, что Git ещё не хранит информацию о версиях файла и не может отследить, как он изменялся.

Сейчас в first-project два файла. Мы хотим отслеживать состояние обоих, поэтому можем использовать команду git add --all (от англ. add — «добавить» + от англ. all — «всё»). Ключ, или флаг, --all позволяет подготовить к сохранению все файлы в репозитории.

$ git add --all *# подготовили к сохранению все файлы в репозитории*

$ git status *# проверили статус*

Добавлять файлы можно и по одному, без ключа --all.

$ git add todo.txt

$ git add readme.txt

$ git status

Также можно добавить текущую папку целиком — в этом случае все файлы в ней тоже будут добавлены. Обратиться к текущей папке в Bash позволяет точка (.).

$ git add . *# добавить всю текущую папку*

$ git status

Файлы, которые отмечены зелёным, теперь отслеживаются и готовы к сохранению. Но сохранения пока не произошло, потому что команда git add только запоминает текущее содержимое (контент) файла.

💡 **Чем отличается запоминание от сохранения?**

Команда git add не сохраняет содержимое файлов в репозитории. Само сохранение, или фиксацию состояния файлов, называют **коммитом** (от англ. *commit* — «совершать», «фиксировать»). «Сделать коммит» значит сохранить текущую версию файла.

Если провести аналогию, команду git add можно сравнить с добавлением товаров в корзину в интернет-магазине, а коммит — с оформлением и оплатой заказа.

Если сейчас отредактировать любой из «зелёных» файлов в папке first-project, он перейдёт в состояние modified (англ. «изменённый») и будет и в «зелёном», и в «красном» списках.

Например, откройте файл todo.txt в любом редакторе (подойдёт даже блокнот) и напишите в нём: 1. Пройти пару уроков по Git..

Сохраните изменения, а затем снова вызовите команду git status в консоли.

Файл todo.txt теперь есть и в «зелёном», и в «красном» списках:

* зелёным отмечена пустая версия файла — в таком виде он был во время последнего запуска команды git add;
* красным отмечена версия с текстом 1. Пройти пару уроков по Git..

Чтобы запомнить новое состояние файла, нужно снова ввести команду git add и передать в качестве параметра имя изменённого файла или ключ --all.

$ git add todo.txt

*# или*

$ git add –all

Теперь файл todo.txt снова готов к сохранению! Будет сохранена последняя добавленная версия с текстом 1. Пройти пару уроков по Git..

# Делаем первый коммит

Коммит — это одна из основных сущностей в Git (и в других системах контроля версий). Коммит гарантирует, что изменения будут сохранены в истории и при необходимости к ним можно будет «откатиться». Это как если бы вы могли выполнить операцию Ctrl+Z для целой папки (репозитория).

Выполнить коммит — git commit

Сделать коммит можно командой git commit c ключом -m (от англ. message — «сообщение»), который присваивает коммиту сообщение.

Обычно в таком сообщении поясняется, в чём именно состояли изменения. Это как заметки на полях: благодаря им проще читать и понимать текст. Сообщение коммита выполняет те же функции — улучшает понимание и упрощает навигацию. Оно пишется после ключа -m в кавычках.

Например, перейдите в папку first-project и выполните коммит со следующим комментарием.

$ git commit -m ‘Мой первый коммит!’

После нажатия Enter текущая версия файлов будет сохранена в репозитории с сообщением Мой первый коммит!. **Коммит** (по названию команды git commit) — это по сути список файлов с их контентом.

*Команда git commit выведет информацию о коммите.*

* *[master (root-commit) baa3b6e] значит:* 
  + *коммит был в ветке master;*
  + *root-commit — это самый первый, или «корневой» (англ. root), коммит в ветке, у следующих коммитов такой надписи не будет;*
  + *baa3b6e — сокращённый идентификатор коммита (подробнее об этом мы ещё расскажем).*
* *2 files changed, 1 insertion(+) значит:* 
  + *изменились два файла (readme.txt и todo.txt);*
  + *одна строка была добавлена (1. Пройти пару уроков по Git.).*
* *Строки вида create mode 100644 readme.txt — это более подробная информация о новых (добавленных в Git) файлах.* 
  + *create (англ. «создать») говорит, что файл был создан. Если бы файл был удалён, на этом месте было бы слово delete (англ. «удалить»).*
  + *mode 100644 сообщает, что это обычный файл. Также возможны варианты 100755 для исполняемых файлов (например, что-нибудь.exe) и 120000 для файлов-ссылок в Linux. Файлы-ссылки не содержат данных сами по себе, а только ссылаются на другие файлы — как «ярлыки» в Windows.*

💡 Обратите внимание: после того как вы сделали первый коммит, команда git status перестала выводить сообщение No commits yet (анг. «ещё нет коммитов»).

### **Ещё раз о разнице между git add и git commit**

Сначала команда git add сообщает Git, какие именно файлы нужно сохранить и какую их версию. Затем с помощью команды git commit происходит само сохранение.

Сначала вы просите друзей встать в ряд — это команда git add. И только после того, как все заняли свои места, поправили волосы и улыбнулись, вы нажимаете кнопку и делаете снимок — это команда git commit. Сам получившийся снимок и будет коммитом. В нашем случае на этой фотографии с обратной стороны ещё есть подпись «Мой первый коммит!».

# Просматриваем историю коммитов

Просмотреть историю коммитов — git log

это понадобится для отслеживания того, что происходит в репозитории.

В самостоятельном задании прошлого урока вы сделали три коммита в ваш репозиторий. Чтобы увидеть их все, введите команду git log (от англ. *log* — «журнал [записей]»).

Обратите внимание, что по умолчанию git log выводит коммиты в обратном хронологическом порядке — последние коммиты оказываются первыми сверху. В этом можно убедиться, если посмотреть на дату и время их создания.

Если после выполнения команды вы видите, что в репозитории есть только один коммит или их нет вообще, вернитесь к прошлому уроку и убедитесь, что git add и git commit были вызваны в нужном порядке.

# Знакомство с GitHub

До этого момента вы использовали Git локально: сейчас проект first-project хранится только на вашем компьютере. Но одно из ключевых преимуществ Git — удобство командной работы над файлами. Чтобы поделиться репозиторием — например, с коллегами, — нужно завести его удалённую версию.

Процесс командной работы может выглядеть так: вы работаете над файлами проекта, например пишете код, на своём компьютере и сохраняете в локальном репозитории. Как только накапливается достаточно правок, чтобы поделиться ими с остальными, вы передаёте их на удалённый репозиторий. Там ваши коллеги смогут посмотреть то, что получилось, и даже скачать себе на компьютер.

Есть несколько платформ для такой командной работы. Самая популярная — GitHub. В нескольких следующих уроках покажем, как с ней работать.

### **Что такое GitHub**

[GitHub](https://github.com/) — платформа для хранения IT-проектов и совместной работы над ними с использованием Git. По сути, это сайт, куда можно загрузить файлы своего проекта для обмена с другими людьми.

С английского языка слово **hub** переводится как «узловая станция». Здесь можно завести аккаунт и вместе со своей командой работать над любыми задачами. Можно создавать проекты разных типов:

* приватный — только для вас;
* командный — только для членов команды;
* публичный — будет виден всем.

Также можно присоединиться к чужому open source проекту и работать над ним вместе с другими людьми со всего мира.

А ещё GitHub — это социальная сеть для разработчиков.

### **Git и платформы для удалённой работы**

Git и GitHub — это два разных проекта, которые развиваются независимо друг от друга.

Git: - консольный инструмент для работы с локальными и удалёнными репозиториями; - проект с открытым исходным кодом.

GitHub: - платформа для размещения удалённых репозиториев; - принадлежит компании Microsoft.

Кроме GitHub, есть и другие платформы для командной работы. Например, GitLab и Bitbucket, которые тоже позволяют работать с Git. У каждой из этих платформ свои особенности и дополнительная функциональность: - GitLab можно развернуть в виде сервера в приватной сети; - Bitbucket — продукт компании Atlassian, поэтому он легко интегрируется с другими инструментами этой компании, такими как Jira. В этом курсе вы будете взаимодействовать с GitHub. Но в целом эти платформы похожи, и если вы изучите одну из них, то переход на другую не будет проблемой.

💡 **Можно ли делать сложные проекты без GitHub?**

Такие платформы, как GitHub, Bitbucket и другие, значительно упрощают процесс командной работы. Но при этом Git может использоваться и без них для создания даже больших проектов.

Например, ядро Linux — самой популярной операционной системы для серверов, телефонов и суперкомпьютеров — разрабатывают с помощью **патчей** (от англ. *patch* — «заплата», «лоскут»). Это файлы, которые содержат отличия исходной версии от последующих.

Такие патчи рассматривает и объединяет в основную версию ядра лично Линус Торвальдс — создатель Linux и Git. Это происходит без использования средств платформ вроде GitHub.

**Регистрация на GitHub**

Вы познакомились с платформой GitHub — пришло время зарегистрироваться на ней. Поехали!

Инструкция по регистрации

1.В правом верхнем углу главной страницы GitHub нажмите на Sign up (англ. «зарегистрироваться»).

2. На экране будут последовательно появляться поля для ввода.

2.1. Введите адрес электронной почты (англ. Enter your email).

2.2. Придумайте пароль (англ. Create a password).

2.3. Введите имя пользователя (англ. Enter a username).

3. Платформа спросит, хотите ли вы получать на почту рассылку с обновлениями и новостями (англ. Would you like to receive product updates and announcements via email?). Введите y, если хотите получать рассылку, или n, если не хотите.

4. Нажмите кнопку Continue (англ. «продолжить»).

5. GitHub предложит вам пройти капчу. Сделайте это.

6. После прохождения капчи нажмите Create account (англ. «создать аккаунт»).

7. Введите короткий код, который будет отправлен на указанный вами почтовый адрес.

Поздравляем! Вы успешно зарегистрировались на крупнейшем веб-хостинге проектов GitHub. Теперь у вас есть возможность работать бок о бок с миллионами профессионалов по всему миру, обмениваться идеями и развиваться.

**Создаём удалённый репозиторий**

В прошлом уроке вы завели аккаунт на GitHub. Теперь разберём, как создать удалённый репозиторий, чтобы в будущем связать его с локальным.

Инструкция по созданию репозитория на GitHub

1. Зайдите в свой профиль по ссылке https://github.com/username, где username — имя, которое вы указали при регистрации.

Эта страница — презентация вас и ваших проектов. Её видят другие пользователи. Надпись You don't have any public repositories yet (англ. «у вас пока нет публичных репозиториев») сообщает, что пока у вас нет проектов.

2. Создайте репозиторий. Для этого перейдите на вкладку Repositories (англ. «репозитории»), а затем нажмите на зелёную кнопку New (англ. «новый») справа.

3. Открылось окно создания нового репозитория. Назовите его first-project. Название удалённого репозитория необязательно должно совпадать с именем папки проекта у вас на компьютере. Но чтобы не путаться, будем называть их одинаково.

Другие поля вам пока не понадобятся. Смело нажимайте на зелёную кнопку Create repository (англ. «создать репозиторий») внизу.

Готово! Удалённый репозиторий создан. Страница с ним открывается автоматически.

Осталось связать удалённый репозиторий с локальным, который уже есть на вашем компьютере. GitHub предоставляет для этого инструкцию (пункт …or push an existing repository from the command line).

Но прежде, чтобы упростить работу с GitHub и сделать её более безопасной, вы научитесь генерировать SSH-ключи (от англ. Secure Shell — «безопасная оболочка»). Об этом в следующем уроке.

**Что такое SSH. Генерируем SSH-ключ**

Представьте, что у вас есть ключ от двери, за которой хранится важный документ. Чтобы получить доступ к этому документу, вам нужно вставить ключ в замочную скважину и повернуть его. Поскольку ключ есть только у вас, ваш документ надёжно защищён от посторонних глаз.

Чтобы получить доступ к репозиторию на GitHub, вам тоже нужно предоставить ключ, который подтверждает вашу личность и права на чтение или изменение данных. Без этого ключа доступ будет ограничен. Об этом и пойдёт речь в уроке.

**Что такое SSH**

Когда компьютеры обмениваются данными в сети, они следуют сетевым протоколам (англ. network protocols) — правилам обмена данными между компьютерами.

Один из наиболее распространённых сетевых протоколов — SSH (от англ. Secure Shell Protocol).

Он обеспечивает безопасный обмен данными в сети. С помощью этого протокола можно получать данные с удалённого компьютера или отправлять их на него. Трафик шифруется, поэтому протокол безопасен.

SSH использует пару ключей для обеспечения безопасности — публичный и приватный:

**Приватный ключ (англ. private key)** хранится только на вашем компьютере и не должен передаваться кому-либо ещё. Он используется для расшифровки данных.

**Публичный ключ (англ. public key)** доступен всем и используется для шифрования данных. Они могут быть расшифрованы парным приватным ключом.

Только вы можете расшифровать данные с помощью приватного ключа, но любой владелец публичного ключа может их для вас зашифровать. Эти два ключа связаны и образуют SSH-пару. В будущем вы наверняка будете использовать их для взаимодействия с GitHub и другими удалёнными серверами.

**Проверка наличия SSH-ключа**

Прежде чем генерировать SSH-ключи, убедитесь, что у вас их ещё нет. По умолчанию директория с SSH-ключами находится в домашней директории пользователя. Перейдите в неё.

$ cd ~ # перешли в домашнюю директорию

Обычно SSH-ключи находятся в директории .ssh/. Проверить наличие этой директории и файлов в ней можно с помощью следующей команды.

$ ls -la .ssh/ # вывели список созданных ключей

Если папка пустая или её нет, всё в порядке.

Если есть файлы с похожими названиями, SSH-ключи уже создавались:

id\_dsa.pub;

id\_ecdsa.pub;

id\_ed25519.pub;

id\_rsa.pub.

Если вы не создавали эти файлы, удалите их все.

**Инструкция по генерации SSH-ключа**

1. Для генерации SSH-пары можно использовать программу ssh-keygen. Откройте терминал и введите следующую команду.

$ ssh-keygen -t ed25519 -C "электронная почта, к которой привязан ваш аккаунт на GitHub"

Используйте электронную почту, к которой привязан ваш GitHub-аккаунт.

Если вы видите сообщение об ошибке, то, скорее всего, ваша система не поддерживает алгоритм шифрования ed25519. Ничего страшного: используйте другой алгоритм.

$ ssh-keygen -t rsa -b 4096 -C "электронная почта, к которой привязан ваш аккаунт на GitHub"

После ввода отобразится такое сообщение.

> Generating public/private rsa key pair. # сгенерированы публичный и приватный ключи

2. Укажите место хранения ключей. Простой вариант — сделать домашний каталог пользователя путём по умолчанию. Для этого нажмите Enter.

> Enter a file in which to save the key (C:\Users\<имя\_пользователя>\.ssh\):[Press enter]

Теперь в указанной директории появится пара ключей.

3. Программа запросит кодовую фразу (англ. passphrase) для доступа к SSH-ключу. Вы можете оставить поле пустым. Для этого нажмите Enter, а затем ещё раз Enter для подтверждения.

> Enter passphrase (empty for no passphrase): [Type a passphrase]

> Enter same passphrase again: [Type passphrase again]

💡 Быть или не быть кодовой фразе — вот в чём вопрос

Как бы странно ни звучало, кодовая фраза — это «пароль от ключа». Представьте, что SSH-ключ лежит в шкатулке. А на самой шкатулке — кодовый замок, который открывается кодовой фразой.

Многие пользователи Git не используют кодовую фразу для защиты своего SSH-ключа. Если такой фразы нет, то её не нужно вводить всякий раз при взаимодействии с удалённым репозиторием.

С другой стороны, применение кодовой фразы усиливает безопасность ключей. Если вы используете эту фразу, ключ будет надёжно защищён в случае несанкционированного доступа к вашему компьютеру.

4. Готово! Теперь осталось проверить, что ключи действительно сгенерировались. Для этого вызовите эту команду.

ls -a ~/.ssh

На экране должны появиться два файла — один с расширением .pub, другой — без. Файл в .pub — публичный, им можно делиться с веб-сайтами или коллегами. Файл без расширения .pub — приватный. Ни в коем случае не передавайте его никому!

Вся последовательность действий в консоли показана на скриншоте ниже.



**Привязываем SSH-ключ к GitHub**

В прошлом уроке вы сгенерировали SSH-ключ, но он пока не привязан к аккаунту на GitHub. Исправим это.

**Инструкция по связыванию SSH-ключа и GitHub-аккаунта**

1. После выполнения команды **ssh-keygen** из предыдущего урока в директории **~/.ssh** будет создано два файла — **id\_ed25519 и id\_ed25519.pub (или id\_rsa и id\_rsa.pub** — в зависимости от того, какой алгоритм вы использовали):

**id\_ed25519/id\_rsa — приватный ключ** (файл без .pub в конце). Ни в коем случае не копируйте его и не делитесь им.

**id\_ed25519.pub/id\_rsa.pub — публичный ключ** (на это указывает расширение .pub).

Скопируйте содержимое файла с публичным ключом в буфер обмена.

*# скопировать содержимое ключа в буфер обмена:*

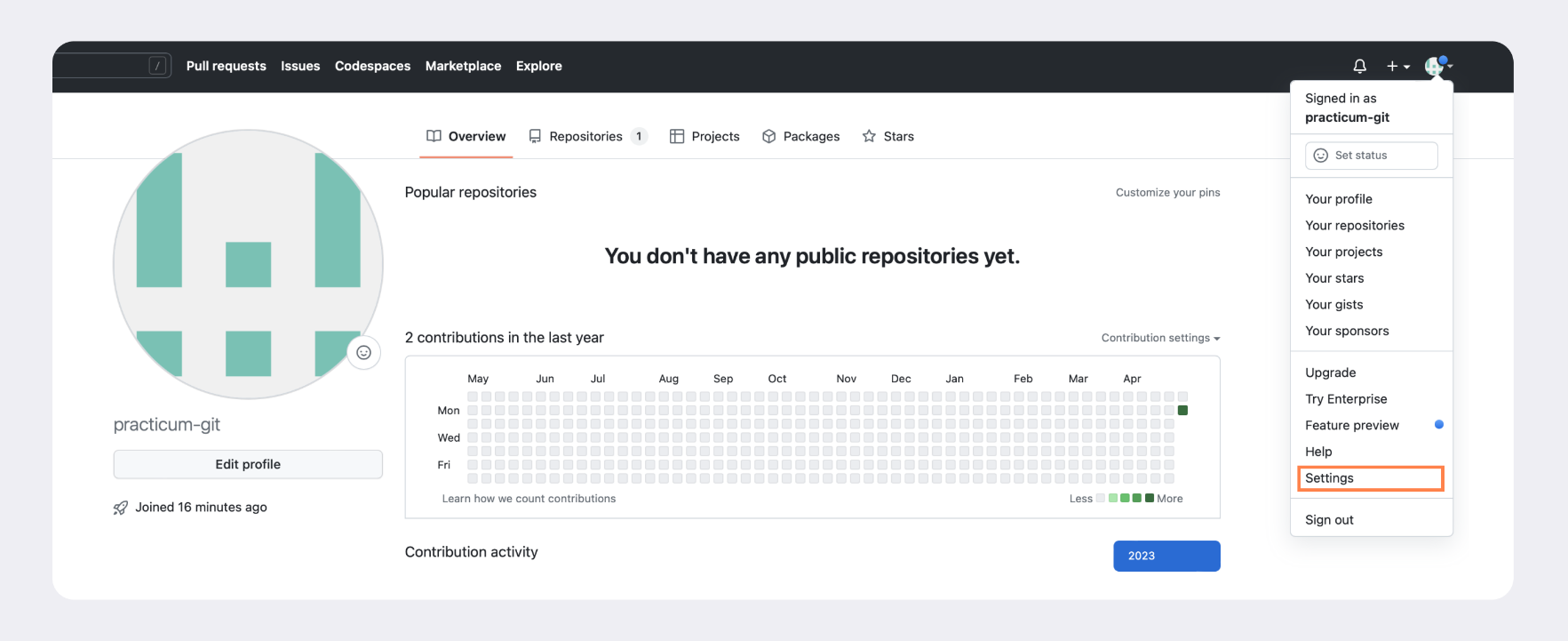
$ clip < ~/.ssh/id\_rsa.pub

*# для ed25519:*

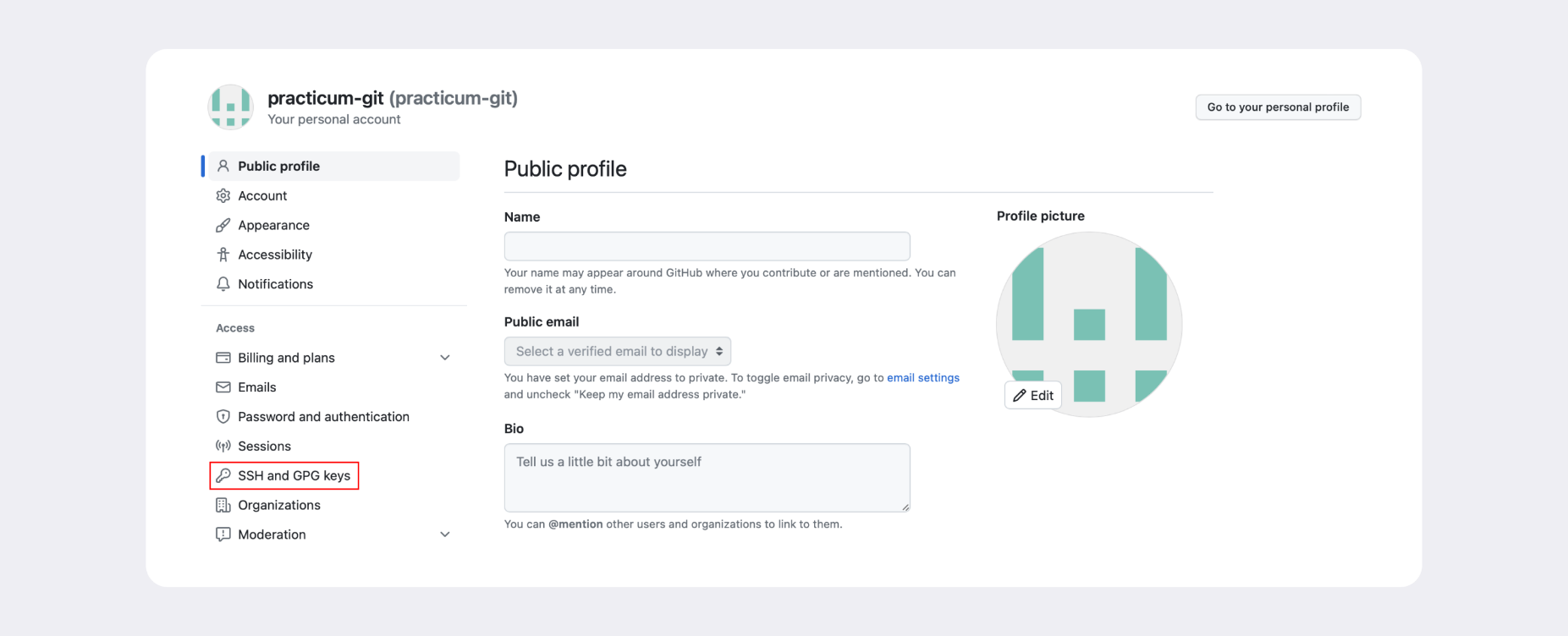
$ clip < ~/.ssh/id\_ed25519.pub

Если **clip** не сработает, выведите содержимое файла с помощью **cat ~/.ssh/id\_rsa.pub или cat ~/.ssh/id\_ed25519.pub** и скопируйте вывод в буфер обмена из консоли.

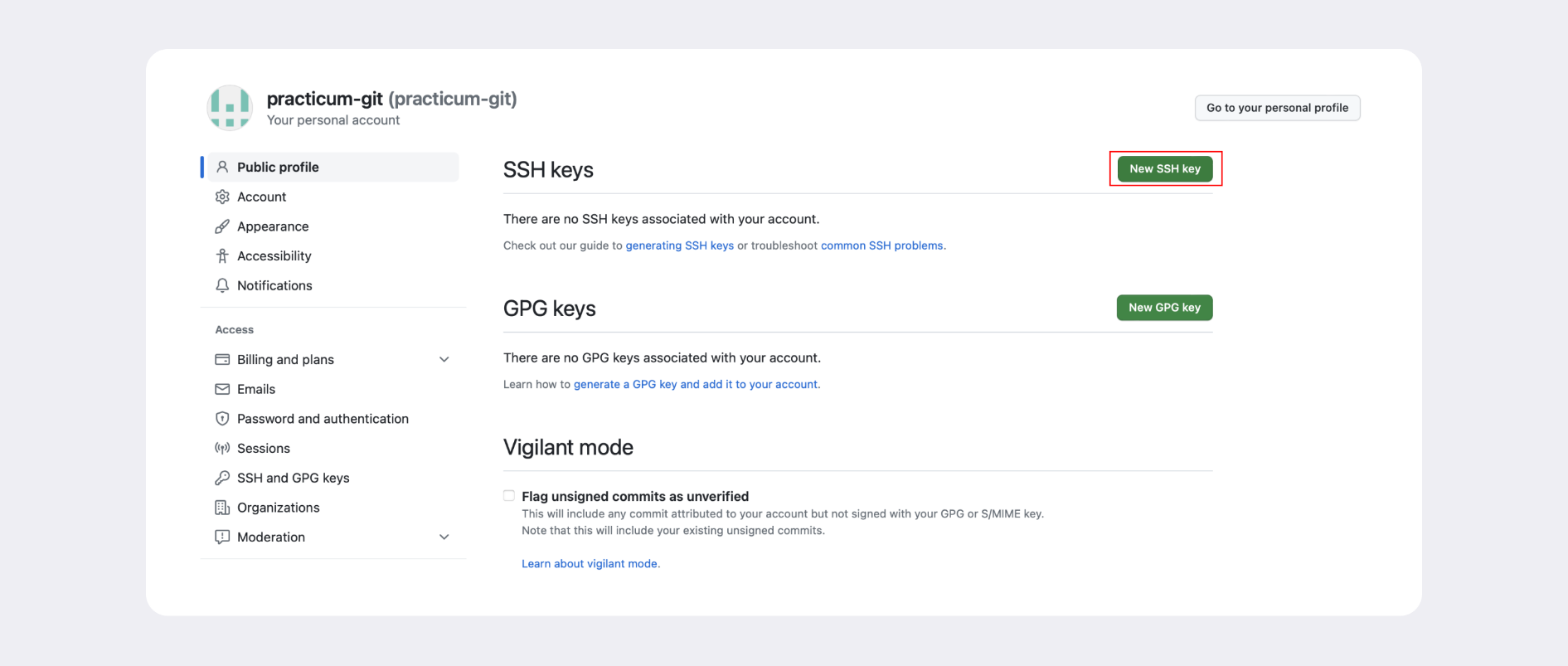
1. Перейдите на **GitHub** и выберите пункт **Settings** (англ. «настройки») в меню аккаунта.



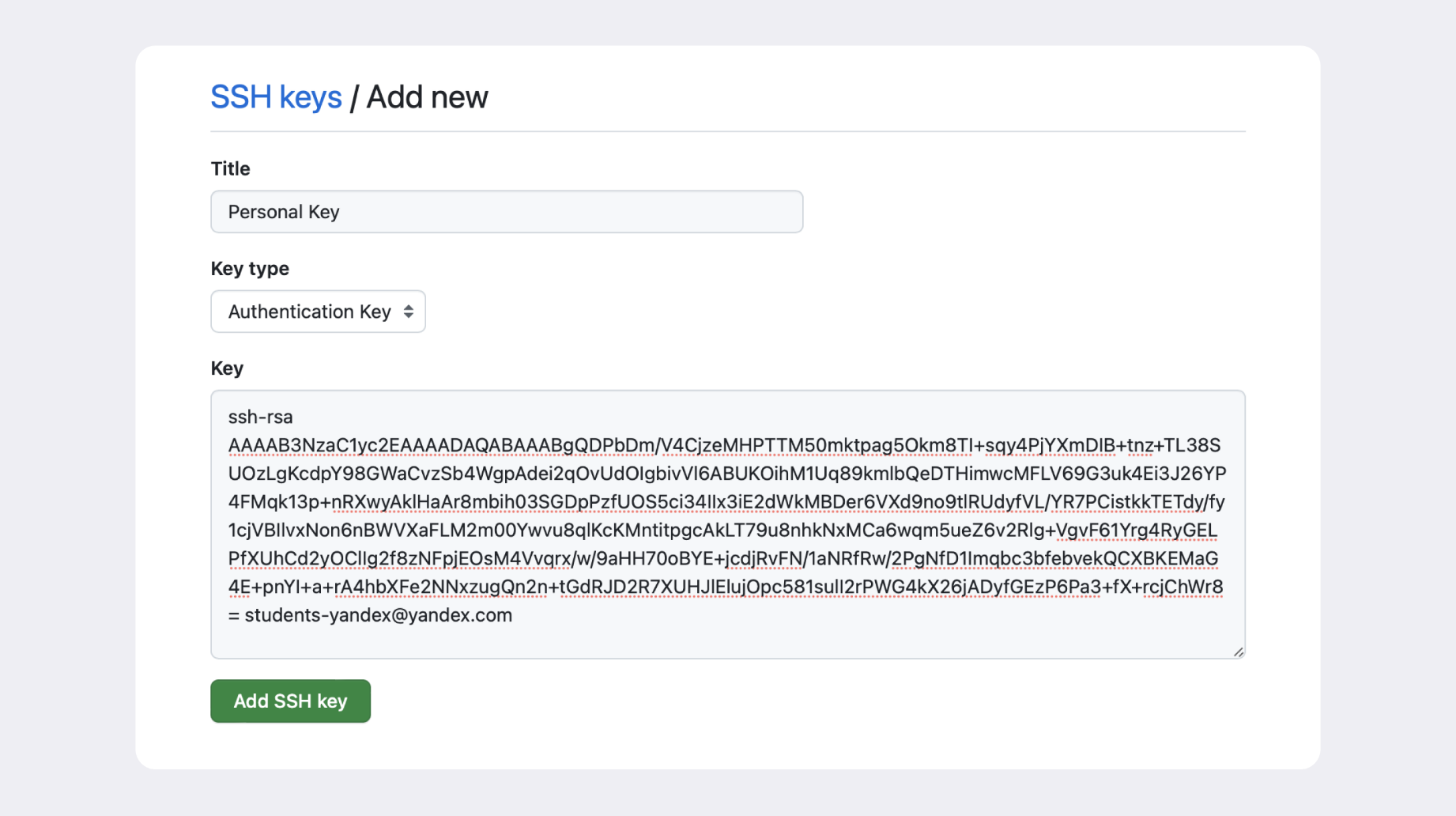
1. В меню слева нажмите на пункт **SSH** **and GPG keys.**



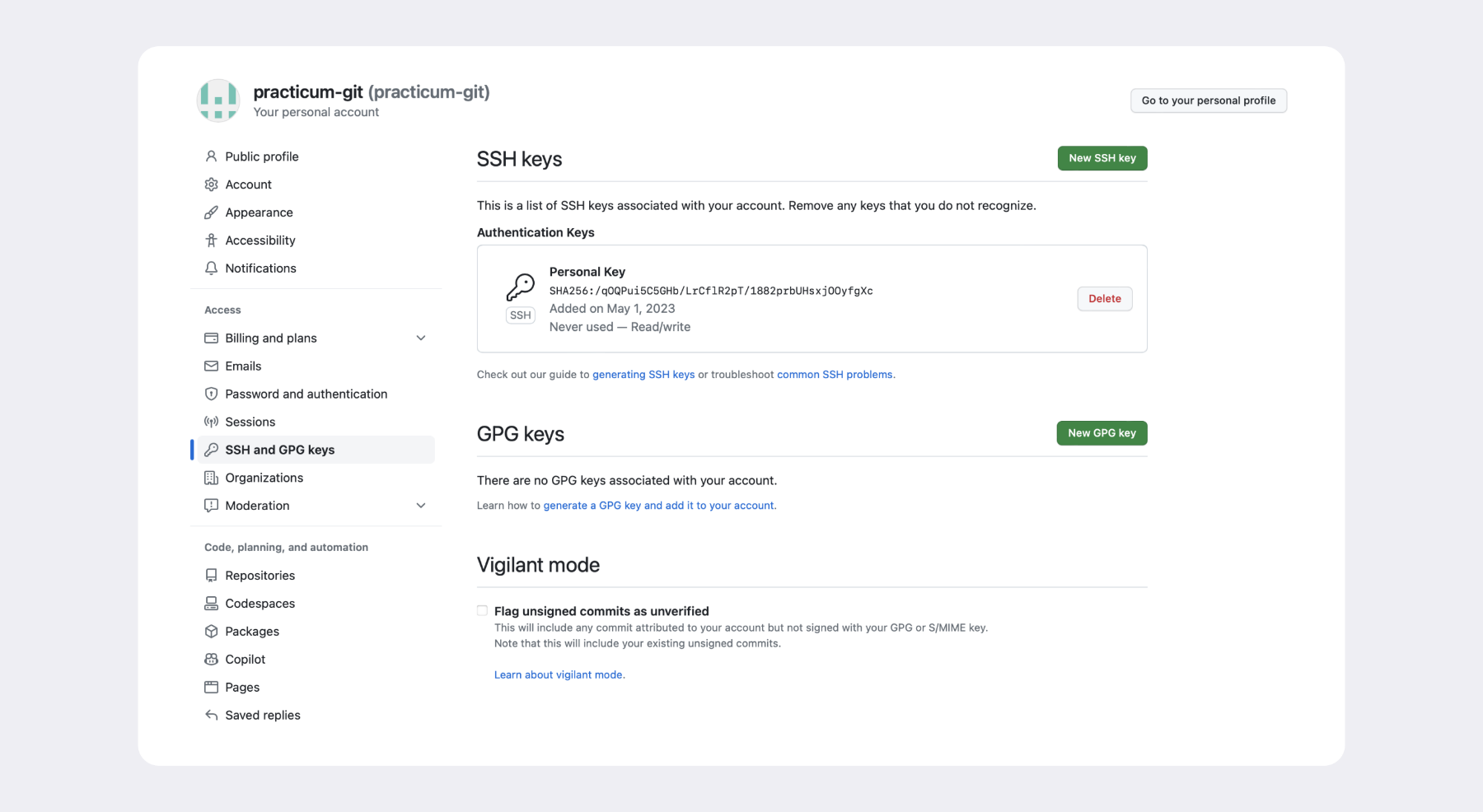
1. В открывшейся вкладке выберите **New SSH key** (англ. «новый SSH-ключ»).



1. В поле **Title** (англ. «заголовок») напишите название ключа. Например, **Personal key** (англ. «личный ключ»).
2. В поле **Key type** (англ. «тип ключа») должно быть **Authentication Key** (англ. «ключ аутентификации»).
3. В поле **Key** скопируйте ваш ключ из буфера обмена.



1. Нажмите на кнопку **Add SSH key** (англ. «добавить SSH-ключ»).



1. Проверьте правильность ключа с помощью следующей команды:

$ ssh -T [git@github.com](mailto:git@github.com)

Если это первый раз, когда вы используете Git, чтобы поделиться проектом на GitHub, появится похожее предупреждение:

The authenticity of host 'github.com (140.82.121.4)' can't be established. ED25519 key fingerprint is SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvCOqU. This key is not known by any other names. Are you sure you want to continue connecting (yes/no/[fingerprint])?

Это предупреждение сообщает, что вы никогда не соединялись с сервером GitHub. Поэтому Git не может гарантировать, что сервер является тем, за кого он себя выдаёт.

Для подтверждения подлинности сервер генерирует и публикует ключи SHA256. Вы можете проверить ключи GitHub по этой ссылке. Если ключ в предупреждении совпадает с тем, что вы видите на сайте, значит, сервер является действительным. Введите yes, чтобы продолжить. Вы увидите приветствие на экране.

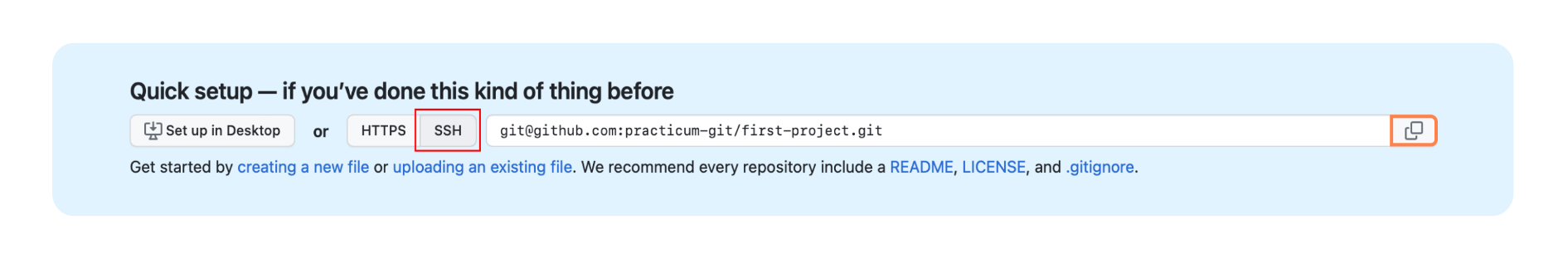
Hi %ВАШ\_АККАУНТ%! You've successfully authenticated, but GitHub does not provide shell access.

**Связываем локальный и удалённый репозитории**

Сейчас у вас есть локальный репозиторий first-project, который хранится на вашем компьютере, и удалённый репозиторий на GitHub. Вы сгенерировали SSH-ключ для безопасной работы и теперь готовы связать удалённый репозиторий с локальным.

**Привязать удалённый репозиторий к локальному — git remote add**

Перейдите на страницу удалённого репозитория, выберите тип SSH и скопируйте URL. Кнопка справа позволит сделать это мгновенно.



Откройте консоль, перейдите в каталог локального репозитория и введите команду **git remote add** (от англ. remote — «удалённый» и add — «добавить»).

$ cd ~/dev/first-project

$ git remote add origin git@github.com:%ИМЯ\_АККАУНТА%/first-project.git

Команде необходимо передать два параметра**: имя удалённого репозитория и его URL**. В качестве имени используйте слово **origin**. А **URL вы скопировали со страницы удалённого репозитория.**

💡 *Как выполнить вставку в командную строку?*

*В командную строку нельзя вставить текст из буфера обмена с помощью привычного сочетания Ctrl+V. На Windows (в Git Bash) и Linux для этого используется сочетание Ctrl+Shift+V, а на macOS — Cmd+V.*

*Также можно нажать правую кнопку мыши и выбрать пункт Paste (англ. «вставить») в выпадающем меню.*

**origin** (англ. «источник») — стандартный псевдоним, с помощью которого можно обращаться к главному удалённому репозиторию (обычно такой репозиторий один). Это значительно упрощает работу.

Убедиться, что репозитории связаны, — git remote -v

Отлично: вы связали локальный репозиторий с удалённым. Осталось убедиться, что всё работает, с помощью следующей команды:

$ git remote -v

origin git@github.com:%ИМЯ\_АККАУНТА%/%ИМЯ-ПРОЕКТА%.git (fetch)

origin git@github.com:%ИМЯ\_АККАУНТА%/%ИМЯ-ПРОЕКТА%.git (push)

В выводе вы должны увидеть две строчки, аналогичные тем, что показаны выше.

Флаг **-v** — короткая форма флага **--verbose** (англ. «подробный»). Он позволяет показать больше информации в выводе.

**Синхронизируем локальный и удалённый репозитории**

Вы зарегистрировались на GitHub, сгенерировали SSH-ключ и привязали локальный репозиторий к удалённому. Теперь разберём, как выкладывать свои правки на удалённый репозиторий. Но сначала немного о ветках.

Мы упоминали, что каждый коммит сохраняет актуальное состояние файлов. Сами же коммиты хранятся в ветках (англ. branch).

Если коммит — это снимок состояния файлов, то ветка — временна́я шкала, на которой расположены эти снимки. Ветка всегда начинается от одного из коммитов.

В репозитории может существовать сразу несколько веток — параллельных историй изменений. Также они могут соединяться друг с другом.

Самая первая ветка в репозитории появляется автоматически и называется **main** (англ. «основная») или **master.** Её имя нужно указывать при отправке коммитов на удалённый репозиторий или при получении их из него.

💡***main*** *или* ***master****?*

*Раньше основная ветка в репозиториях, созданных на GitHub, называлась master, но с 11 октября 2020 года (после волны протестов движения Black Lives Matter) её переименовали в main.*

*Во всех репозиториях, созданных раньше этой даты, название основной ветки не поменялось. Поэтому в проектах, которые начали именно с master, и в руководствах по работе с Git вы по-прежнему можете встретить имя master.*

**Отправить изменения на удалённый репозиторий — git push**

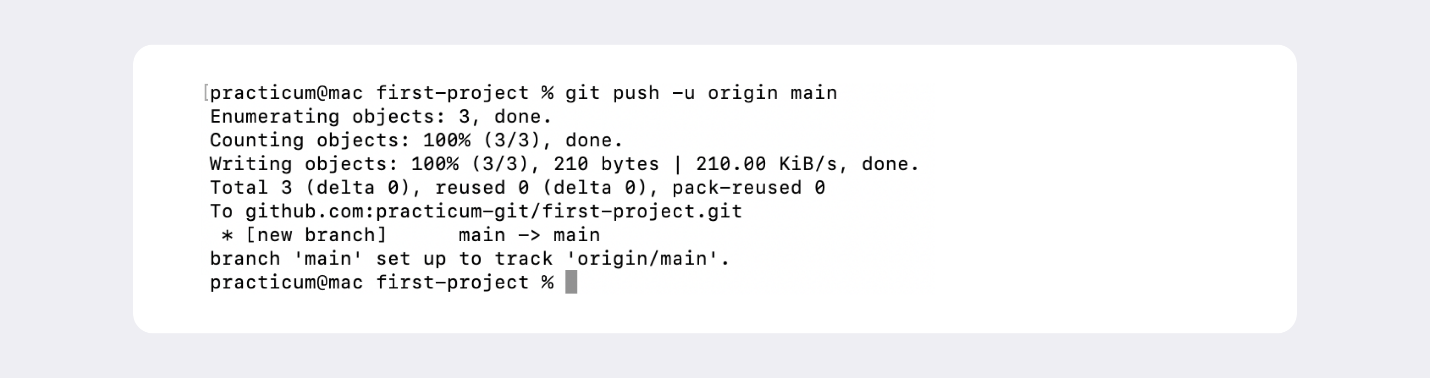
Вы уже прошли весь «цикл коммита»: подготовили файлы с помощью **git add**, закоммитили их с комментарием командой **git commit -m**. Осталось загрузить содержимое локального репозитория на GitHub. За это отвечает команда **git push** (от англ. push — «толкать»).

В первый раз эту команду нужно вызвать с флагом **-u** и параметрами **origin** (имя удалённого репозитория) и **main** или **master** (название текущей ветки). Флаг **-u** свяжет локальную ветку с одноимённой удалённой. Как вы связывали локальный и удалённый репозитории в предыдущем уроке, так же и здесь нужно дополнительно связать ветки.

$ git push -u origin main *# Если команда приведёт к ошибке, попробуйте*

*# заменить main на master.*

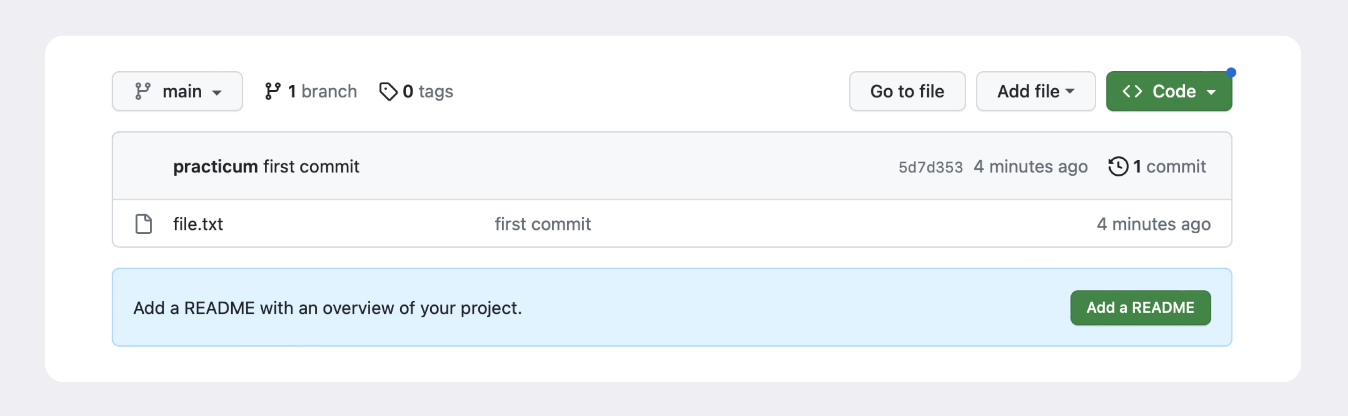
Появится такой экран:

****

При взаимодействии с удалёнными репозиториями Git выводит в консоль отладочную информацию: количество объектов (файлов), которые отправляются на сервер, информацию о прогрессе сжатия и записи и так далее.

Если вы указывали кодовую фразу при настройке SSH-ключей, её нужно будет ввести.

Зайдите в репозиторий first-project на GitHub. Вы увидите, что в репозитории появились файлы с изменениями.

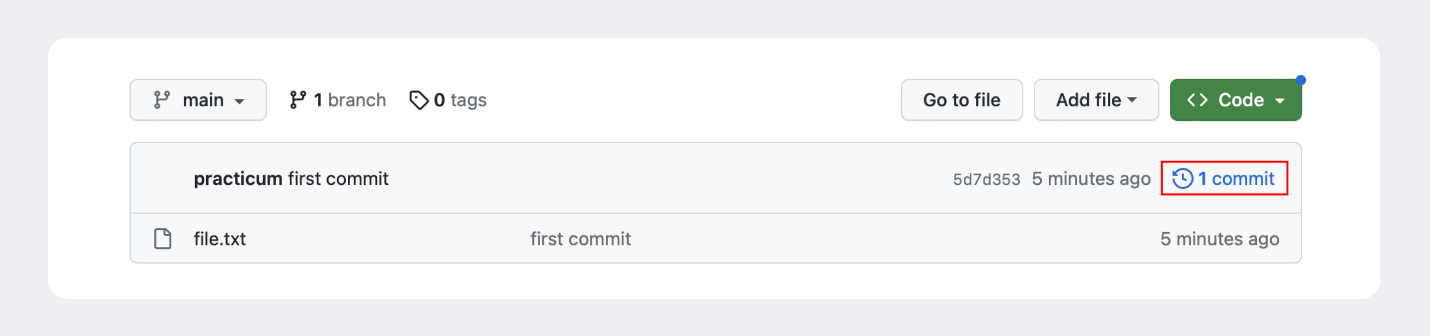


В дальнейшем при работе с удалённым репозиторием флаг **-u** можно опустить и писать просто **git push.**

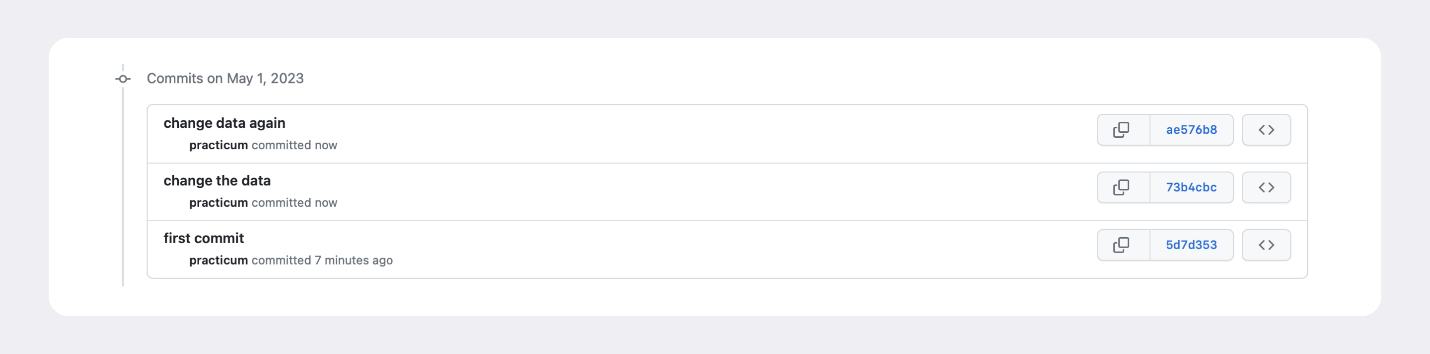
**Работа с графическим интерфейсом GitHub**

GitHub предоставляет удобный интерфейс для работы с репозиторием.

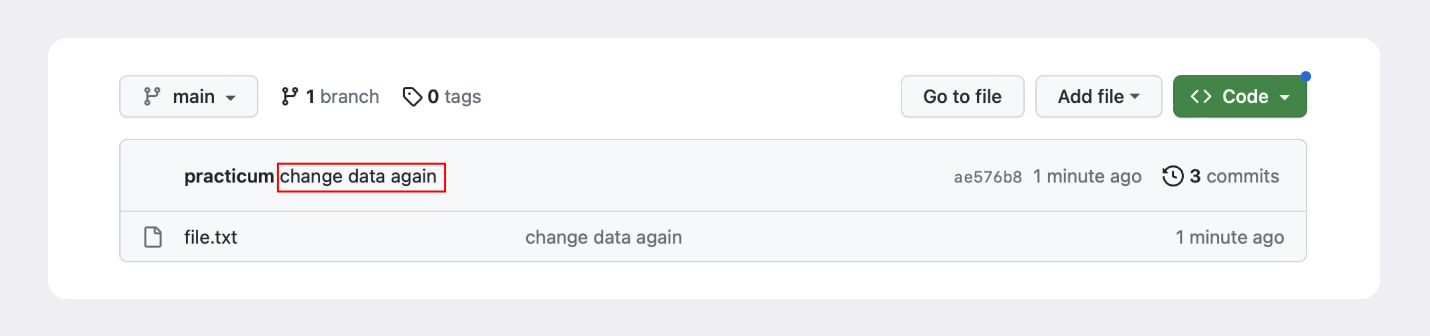
Например, нажмите на кнопку **commit** в правой части страницы, чтобы просмотреть все коммиты в репозитории.



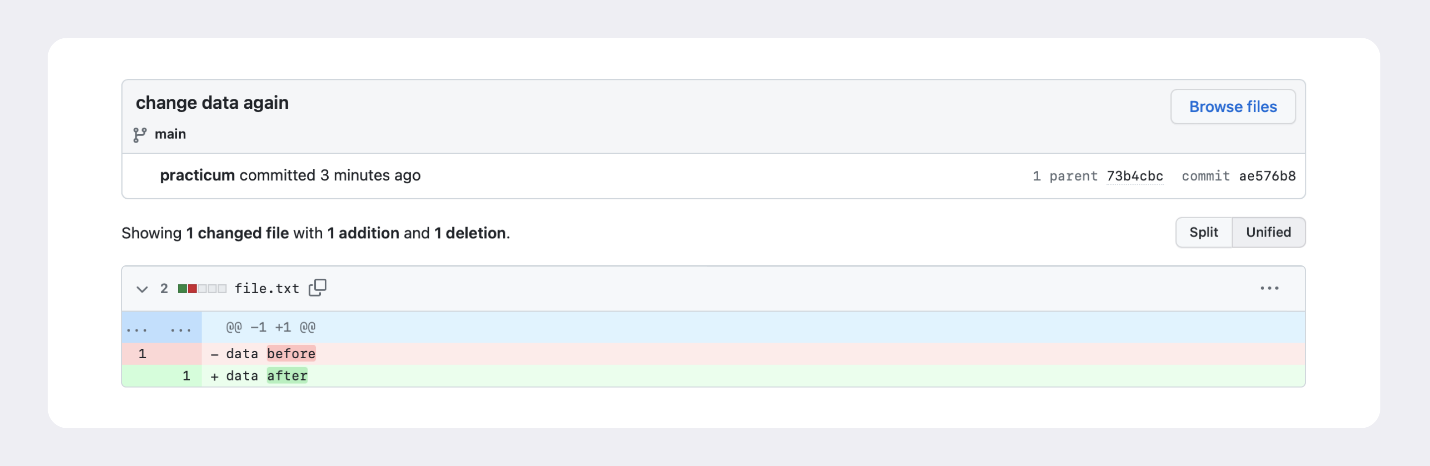
Откроется окно с коммитами и их авторами.



Сообщение коммита в репозитории тоже является ссылкой.



Перейдите по ссылке, кликните на текст последнего коммита над репозиторием — так вы сможете увидеть все изменения, которые были внесены в репозиторий в этом коммите.



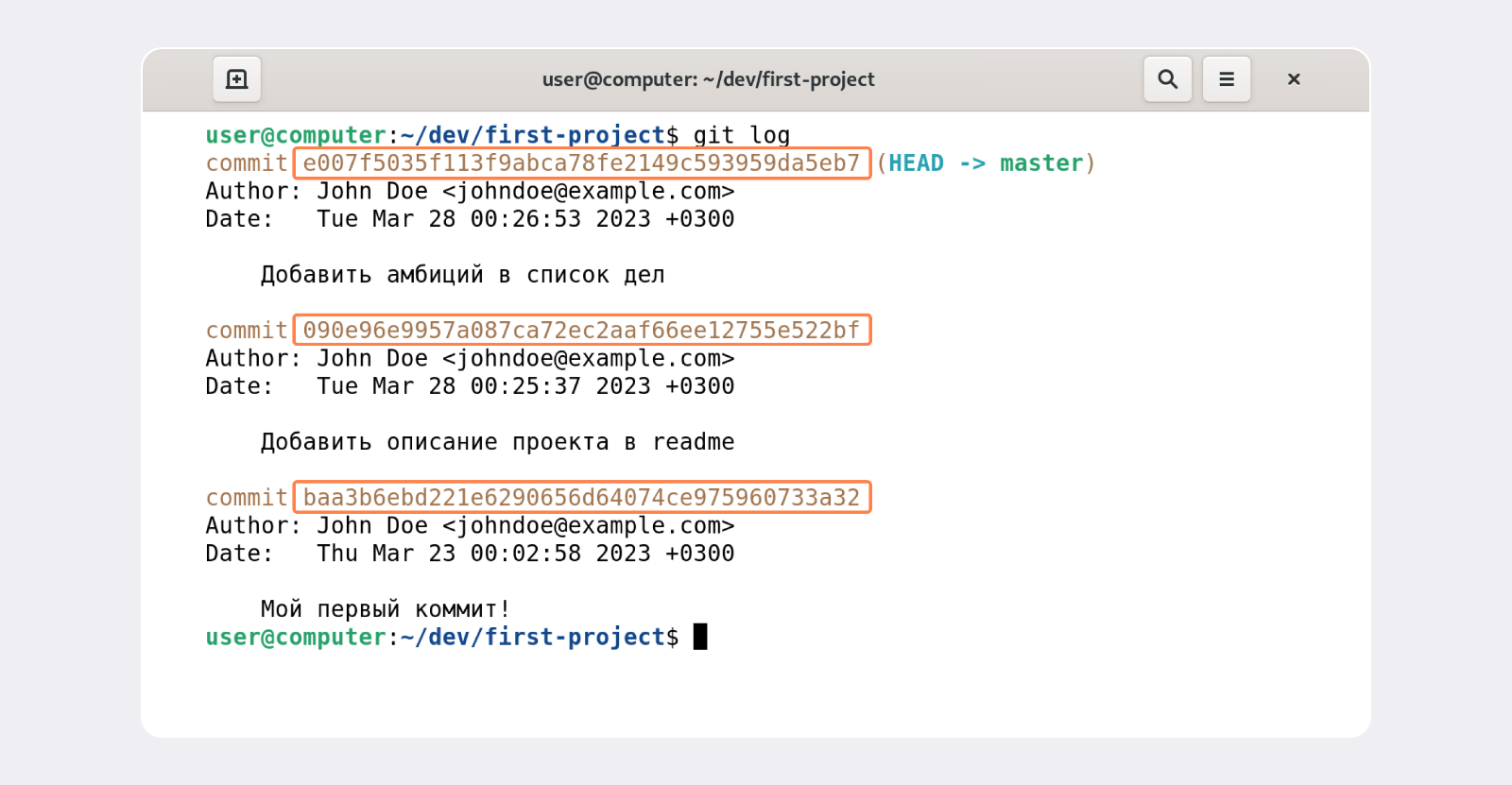
Копилка ваших знаний о Git постепенно пополняется! Вот о чём мы рассказали:

* Коммиты хранятся в ветках. Начальная ветка создаётся автоматически и называется **main** или **master.**
* За отправку изменений на удалённый репозиторий отвечает команда **git push.**
* Интерфейс GitHub позволяет удобно просмотреть все коммиты в репозитории, а также изменения в этих коммитах.

**5. Навигация по камитам. Статусы файлов.**

**Хэш – идентификатор коммита.**

В процессе работы с Git вам будет часто встречаться понятие **«хеш коммита»**. Эти странные строчки с бессмысленным (на первый взгляд) набором букв и цифр вы могли видеть, когда вызывали команду **git log** и выводили историю коммитов.



### **Что такое хеш. Хеширование коммитов**

**Хеширование** (от англ. hash, «рубить», «крошить», «мешанина») — это способ преобразовать набор данных и получить их «отпечаток» (англ. fingerprint).

Информация о коммите — это набор данных: когда был сделан коммит, содержимое файлов в репозитории на момент коммита и ссылка на предыдущий, или **родительский** (англ. parent), коммит.

Git хеширует (преобразует) информацию о коммите с помощью алгоритма **SHA-1** (от англ. ***S***ecure ***H***ash ***A***lgorithm — «безопасный алгоритм хеширования») и получает для каждого коммита свой уникальный **хеш** — результат хеширования.

Git преобразует информацию о коммитах с помощью алгоритма SHA-1 и для каждого из них рассчитывает уникальный идентификатор — хеш.

Обычно хеш — это короткая (4040 символов в случае SHA-1) строка, которая состоит из цифр 0-9 и латинских букв *A*-*F* (неважно, заглавных или строчных). Она обладает следующими важными свойствами:

* если хеш получить дважды для одного и того же набора входных данных, то результат будет гарантированно одинаковый;
* если хоть что-то в исходных данных поменяется (хотя бы один символ), то хеш тоже изменится (причём сильно).

Чтобы убедиться в этом, можно поэкспериментировать с SHA-1 [на этом сайте](https://emn178.github.io/online-tools/sha1.html) — попробуйте ввести в поле **input** (англ. «ввод») разные символы, слова или предложения и понаблюдайте, как меняется хеш в поле **output** (англ. «вывод»).

### **Хеш — основной идентификатор коммита**

Git хранит таблицу соответствий хеш → информация о коммите. Если вы знаете хеш, вы можете узнать всё остальное: автора и дату коммита и содержимое закоммиченных файлов. Можно сказать, что хеш — основной идентификатор коммита.

При работе с Git хеши будут встречаться вам регулярно. Их можно будет передавать в качестве параметра разным Git-командам, чтобы указать, с каким коммитом нужно произвести то или иное действие.

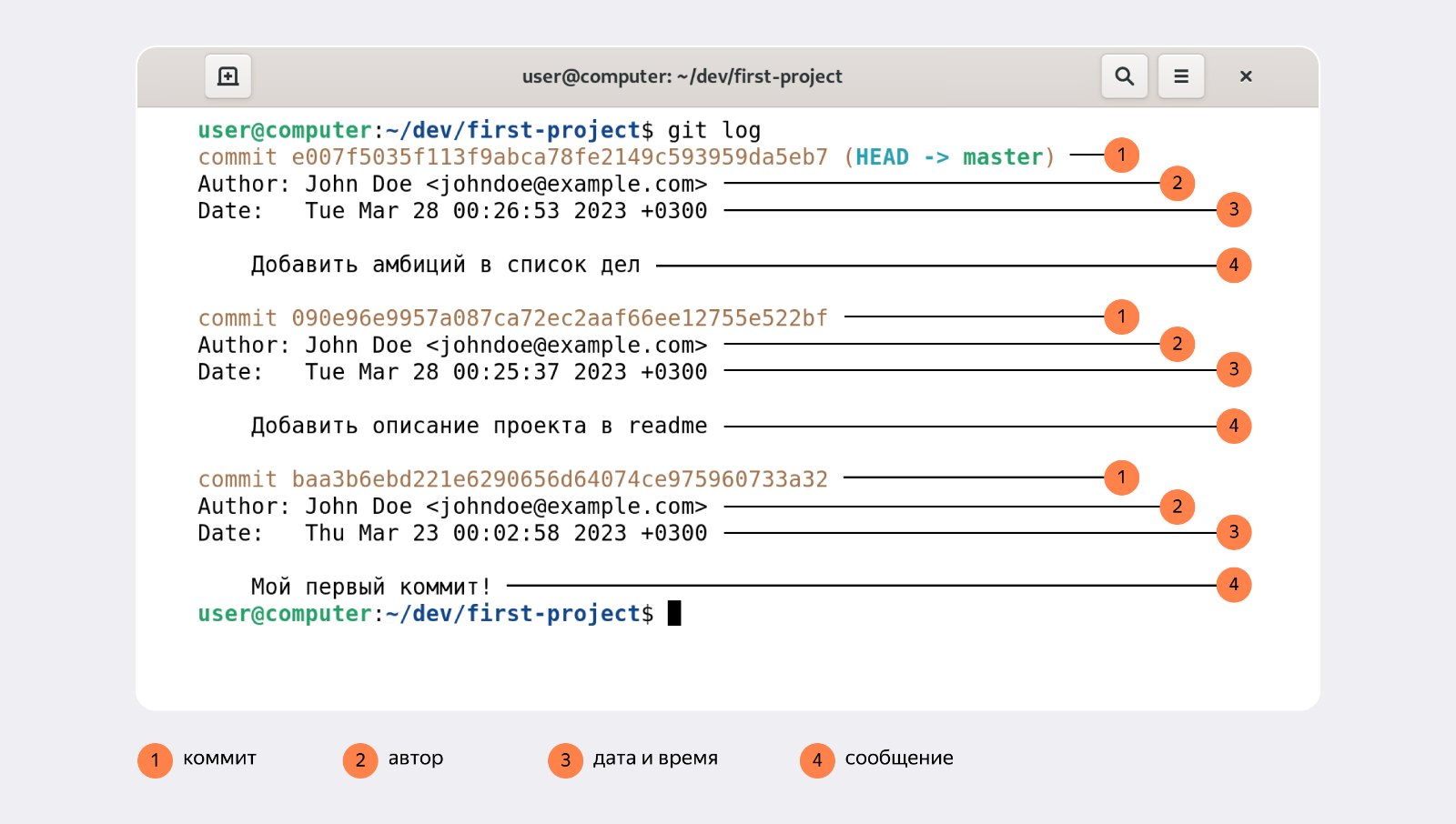
Все хеши и таблицу хеш → информация о коммите Git сохраняет в служебные файлы. Они находятся в скрытой папке .git в репозитории проекта.

### **Исследуем лог**

В этом уроке рассмотрим подробнее, из каких элементов состоит описание коммита, а также как вывести сокращённый **лог** (от англ. log — «журнал [записей]»). Сокращённый лог полезен, если нужно быстро найти нужный коммит среди сотни других.

### **Элементы описания коммита**

После вызова **git log** появляется список коммитов.

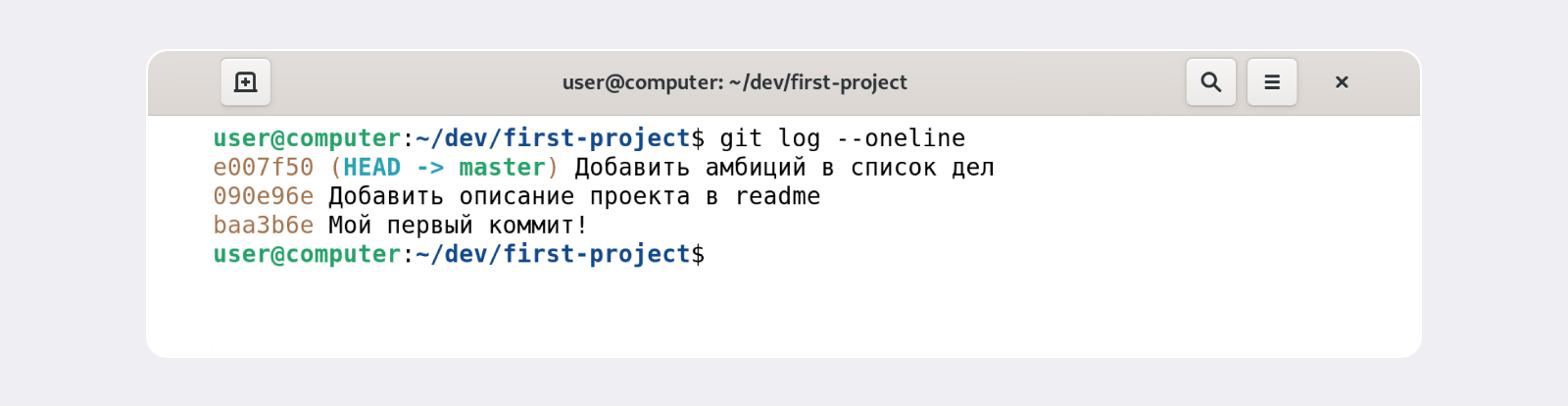


Разберём элементы, из которых состоит описание:

* строка из цифр и латинских букв после слова **commit** — это хеш коммита;
* **Author** — имя автора и его электронная почта;
* **Date** — дата и время создания коммита;
* в конце находится сообщение коммита.

**Получить сокращённый лог — git log --oneline**

Получить сокращённый лог можно с помощью команды git log с флагом --oneline (англ. «одной строкой»). В терминале появятся только первые несколько символов хеша каждого коммита и их комментарии.



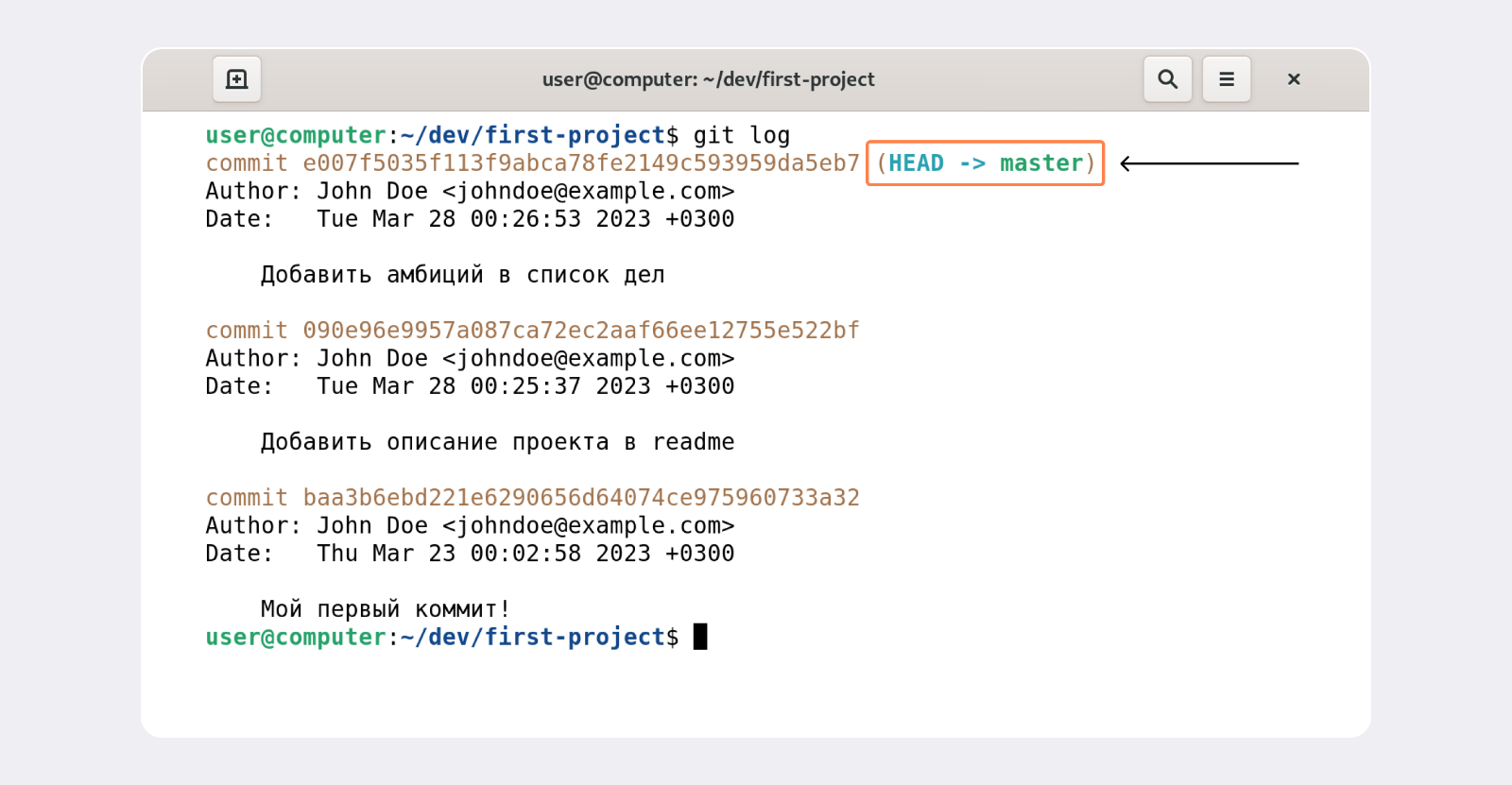
Сокращённый лог полезен, если в репозитории уже много коммитов — например, сотни или тысячи. В этом случае можно быстро найти нужный по описанию.

Сокращённый хеш (то есть первые несколько символов полного) можно использовать точно так же, как и полный. Для этого команда **git log --oneline** автоматически подбирает такую длину сокращённых хешей, чтобы они были уникальными в пределах репозитория и Git всегда мог понять, о каком коммите идёт речь.

💡 Обратите внимание: если выход из просмотра логов не произошёл автоматически, нажмите клавишу **Q** (от англ. ***Q****uit —* «выйти») в английской раскладке клавиатуры.

**HEAD — всему голова**

При вызове команды git log вы также могли заметить надпись (HEAD -> master) после хеша одного из коммитов. В этом уроке расскажем, что она означает.



**Файл HEAD**

Файл **HEAD** (англ. «голова», «головной») — один из служебных файлов папки **.git**. Он указывает на коммит, который сделан последним (то есть на самый новый).

В этом можно убедиться с помощью терминала. Перейдите в папку **.git** командой **cd.** Посмотрите содержимое файла HEAD командой cat.

$ pwd *# посмотрели, где мы*

/Users/user/dev/first-project

$ cd .git/

$ ls *# посмотрели, какие есть файлы*

COMMIT\_EDITMSG ORIG\_HEAD description index logs/ refs/

HEAD config hooks/ info/ objects/

$ cat HEAD *# команда cat показывает содержимое файла*

ref: refs/heads/master *# в файле вот такая ссылка*

Внутри **HEAD** — ссылка на служебный файл: **refs/heads/master** (или **refs/heads/main** в зависимости от названия ветки). Если заглянуть в этот файл, можно увидеть хеш последнего коммита.

**cat refs/heads/master *# взяли ссылку из файла HEAD***

***# внутри хеш***

**e007f5035f113f9abca78fe2149c593959da5eb7**

**$ git log**

***# сверяем с хешем последнего коммита***

**commit e007f5035f113f9abca78fe2149c593959da5eb7**

**Author: John Doe <johndoe@example.com>**

**Date: Tue Mar 28 00:26:53 2023 +0300**

**Добавить амбиций в список дел**

**... *# другие коммиты***

Когда вы делаете коммит, Git обновляет **refs/heads/master** — записывает в него хеш последнего коммита. Получается, что **HEAD** тоже обновляется, так как ссылается на **refs/heads/master.**

При работе с Git указатель **HEAD** используется довольно часто. Мы уже упоминали, что многие команды Git принимают в качестве параметра хеш коммита. Если нужно передать последний коммит, то вместо его хеша можно просто написать слово **HEAD** — Git поймёт, что вы имели в виду последний коммит.

**Статусы файлов в Git**

До появления Git системы контроля версий выделяли только два статуса у файлов: **«уже закоммичен»** и **«ещё не закоммичен».** Например, в Subversion (самой популярной VCS до эпохи Git) не нужно было выполнять команду — аналог **git add**, а можно было просто сделать коммит **(svn commit**). Эта команда по умолчанию добавляла в коммит все новые и изменённые файлы.

Такое поведение интуитивно более понятно. Зато Git даёт больше контроля за состоянием файлов. Хотя сначала это может показаться сложным, со временем вы оцените удобство более явного подхода.

В этом уроке разберём подробнее, в каких состояниях (или статусах) могут находиться файлы в репозитории. А ещё проследим типичный жизненный цикл файла в Git.

**Статусы untracked/tracked, staged и modified**

Одна из ключевых задач Git — отслеживать изменения файлов в репозитории. Для этого каждый файл помечается каким-либо статусом. Рассмотрим основные.

* **untracked** (англ. «неотслеживаемый»)   
  Мы говорили, что новые файлы в Git-репозитории помечаются как untracked, то есть неотслеживаемые. Git «видит», что такой файл существует, но не следит за изменениями в нём. У untracked-файла нет предыдущих версий, зафиксированных в коммитах или через команду git add.
* **staged** (англ. «подготовленный»)

После выполнения команды **git add** файл попадает в **staging area** (от англ. stage — «сцена», «этап [процесса]» и area — «область»), то есть в список файлов, которые войдут в коммит. В этот момент файл находится в состоянии **staged.**

В одном из предыдущих уроков мы сравнили коммит с фотографией. Можно развить эту аналогию и сказать, что команда **git add** добавляет персонажей (текущее содержимое файла или нескольких файлов) на **сцену** (англ. stage) для общей фотографии, а **git commit** делает снимок всей сцены целиком.

💡 **Staging area, index и cache**

**Staging area** также называют **index** (англ. «каталог») или **cache** (англ. «кеш»), а состояние файла **staged** иногда называют **indexed** или **cached.**

Все три варианта могут встречаться в документации и в качестве флагов команд Git. А также в интернете — например, в вопросах и ответах [на сайте Stack Overflow](https://stackoverflow.com/).

* **tracked** (англ. «отслеживаемый»)   
  Состояние tracked — это противоположность untracked. Оно довольно широкое по смыслу: в него попадают файлы, которые уже были зафиксированы с помощью git commit, а также файлы, которые были добавлены в staging area командой git add. То есть все файлы, в которых Git так или иначе отслеживает изменения.
* **modified** (англ. «изменённый»)   
  Состояние modified означает, что Git сравнил содержимое файла с последней сохранённой версией и нашёл отличия. Например, файл был закоммичен и после этого изменён.

💡 Для файлов в состояниях **staged** и **modified** обычно не указывают, что они также **tracked**, потому что это состояние подразумевается.

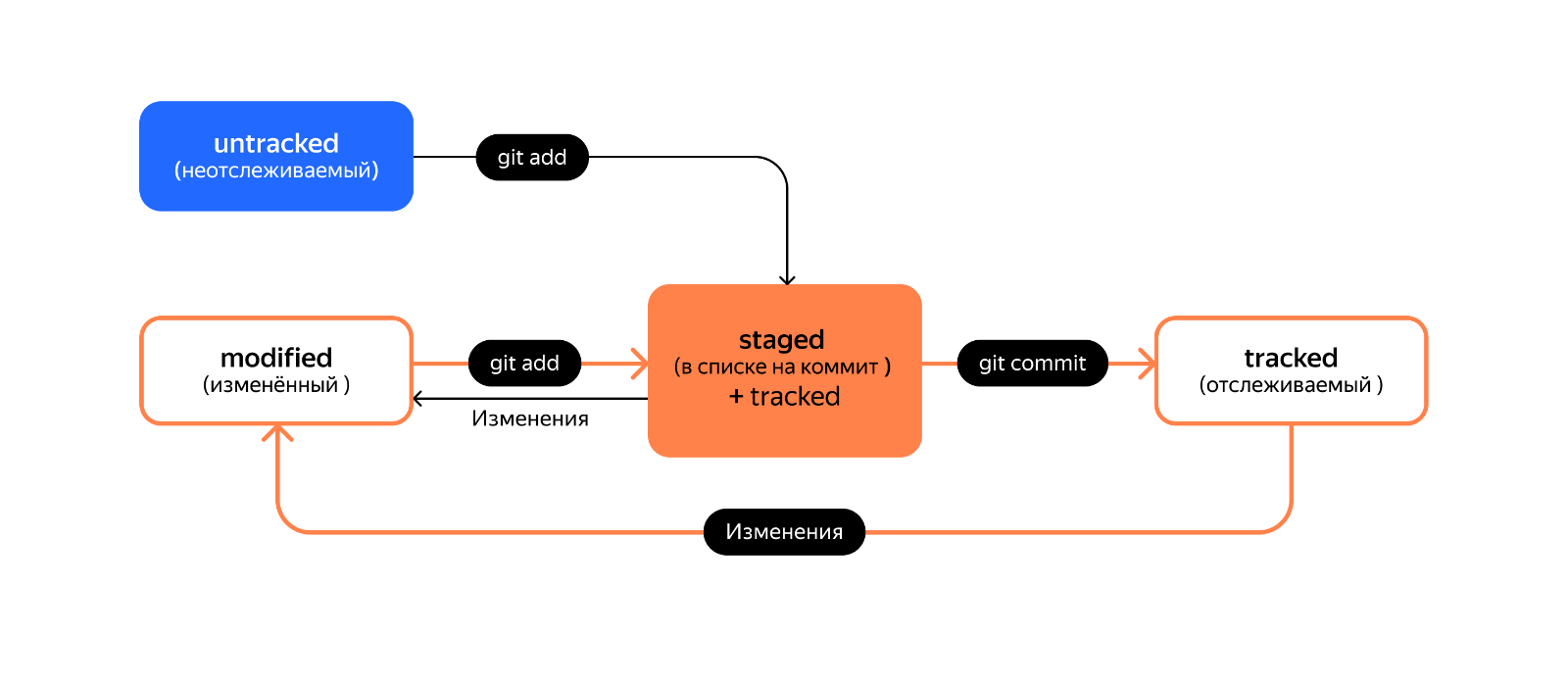
**Про staged и modified**

Команда **git add** добавляет в **staging area** только текущее содержимое файла. Если вы, например, сделаете **git add file.txt**, а затем измените **file.txt**, то новое содержимое файла не будет находиться в **staging**.

Git сообщит об этом с помощью статуса **modified:** файл изменён относительно той версии, которая уже в **staging.** Чтобы добавить в **staging** последнюю версию, нужно выполнить **git add file.txt** ещё раз.

**Типичный жизненный цикл файла в Git**

Может показаться, что файлы в репозитории попадают в разные состояния хаотично. На практике это не так, и у большинства файлов вполне предсказуемый путь.



1. Файл только что создали. Git ещё не отслеживает содержимое этого файла. Состояние: **untracked.**
2. Файл добавили в **staging area** с помощью **git add**. Состояние: **staged (+ tracked).** 
   * Возможно, изменили файл ещё раз. Состояния: **staged, modified (+ tracked)**. Обратите внимание: **staged** и **modified** у одного файла, но у разных его версий.
   * Ещё раз выполнили **git add**. Состояние: **staged (+ tracked).**
3. Сделали коммит с помощью **git commit.** Состояние: **tracked.**
4. Изменили файл. Состояние: **modified (+ tracked).**
5. Снова добавили в **staging area** с помощью **git add.** Состояния: **staged (+ tracked).**
6. Сделали коммит. Состояния: **tracked.**
7. Повторили пункты 4−7много-много раз.

* Статусом **untracked** помечается файл, о существовании которого Git знает, но не следит за изменениями в нём. Этот статус — противоположность **tracked,** в который попадают все файлы, отслеживаемые Git.
* Файл переходит в статус **staged** после выполнения **git add.**
* Статус **modified** означает, что файл был изменён.
* Большинство файлов в проектах «шагает» по следующему циклу: «изменён» → «добавлен в список на коммит» → «закоммичен» → «изменён» → и так далее.

**Как читать git status**

Частая ошибка при использовании Git — закоммитить лишнее или, наоборот, забыть добавить важный файл в коммит. Этого легко избежать, если не забывать проверять статусы файлов с помощью команды **git status**. Как читать её вывод, покажем в этом уроке.

**Какие состояния показывает git status**

Большинство файлов в типичном проекте будут находиться в состоянии **tracked** (то есть закоммичены и не изменены после коммита). Вы не увидите это состояние в выводе команды **git status** - иначе она бы каждый раз выводила список вообще всех файлов проекта.

В итоге **git status** показывает только следующие состояния файлов:

* **staged (Changes to be committed в выводе git status);**
* **modified (Changes not staged for commit);**
* **untracked (Untracked files).**

**Подготавливаем репозиторий**

Чтобы попрактиковаться, инициализируйте новый репозиторий **~/dev/git-status-lesson.** Создайте в нём файл **README.md** и закоммитьте его.

$ cd ~/dev

$ mkdir git-status-lesson

$ cd git-status-lesson

$ git init

*# тут Git выведет что-нибудь, но мы это пропустим*

$ touch README.md

$ git add README.md

$ git commit -m 'Добавить README'

~~~~*# по традиции первым создадим и закоммитим файл README.md*

Дальше вы будете добавлять в репозиторий файлы и смотреть на их статусы.

**Типичные варианты вывода git status**

Рассмотрим четыре примера состояний, в которых может находиться ваш репозиторий.

1. **Нет ни staged-, ни modified-, ни untracked-файлов.**

Если ничего не менять в **git-status-lesson** после первого коммита, то в нём не должно быть ни изменённых файлов (**modified**), ни новых (**untracked**), ни добавленных в список на коммит (**staged**). Вызовите команду **git status**. Её вывод будет примерно таким.

$ git status

On branch master

nothing to commit, working tree clean

Это означает, что в репозитории нет новых или изменённых файлов. Последняя строка **nothing to commit, working tree clean** буквально переводится как **«нечего коммитить, рабочая директория чиста».**

Первая строка **On branch master** сообщает, что текущая ветка — **master.**

1. **Найдены неотслеживаемые файлы.**

Создайте в папке **~/dev/git-status-lesson** файл **fileA.txt**. Теперь в репозитории есть новый файл в состоянии **untracked.** Снова вызовите команду **git status**. Результат будет таким.

$ touch fileA.txt

$ git status

On branch master

Untracked files: *# найдены неотслеживаемые файлы*

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

fileA.txt

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

Файл **fileA.txt** отображается в секции **неотслеживаемых** файлов **- Untracked files.** Это значит, что он не был добавлен в репозиторий через **git add.**

💡 Обратите внимание: в самом выводе **git status** есть подсказка, какую команду использовать, чтобы добавить файл в список на коммит: **Use** **git add <file>** **to include in what will be committed** (англ. «используйте **git add <file>,** чтобы добавить в список на коммит»).

Добавьте **fileA.txt в staging area** с помощью **git add** и снова запросите **git status.**

$ git add fileA.txt

$ git status

On branch master

Changes to be committed: *# новая секция*

(use "git restore --staged <file>..." to unstage)

new file: fileA.txt

💡 В этот раз **git status** подсказывает, что существует команда **git restore**. Мы познакомим вас с ней в одном из будущих уроков.

Теперь **fileA.txt** находится в секции **Changes to be committed** (англ. **«изменения, которые попадут в коммит»).** Если сейчас выполнить коммит, то в репозитории будет зафиксирована текущая версия этого файла. Закоммитьте его.

$ git commit -m 'Добавить файл fileA.txt'

*# тут будет вывод комманды commit, он нас не интересует*

$ git status

On branch master

nothing to commit, working tree clean

Вывод команды **git status** такой же, какой был после первого коммита: «Директория чиста».

1. **Найдены изменения, которые не войдут в коммит**

Теперь откройте файл **fileA.txt** и добавьте в него несколько слов — например, **Это файл A!**. Сохраните **fileA.txt** и вызовите команду **git status.** Её результат будет такой.

*# внесли в fileA.txt правки*

*# запросили статус*

$ git status

On branch master

Changes not staged for commit: *# ещё одна секция*

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: fileA.txt

Файл **fileA.txt** был изменён, но ещё не добавлен в **staging area** после этого. Так он оказался в секции **Changes not staged for commit (**англ. **«изменения, которые не подготовлены к коммиту»).** Эта секция соответствует статусу **modified.**

Подготовьте правки к коммиту с помощью **git add**.

$ git add fileA.txt

$ git status

On branch master

Changes to be committed: *# все изменения готовы к коммиту*

(use "git restore --staged <file>..." to unstage)

modified: fileA.txt

Теперь в коммит попадёт уже новая версия файла **fileA.txt.**

💡 Обратите внимание: хотя вывод команды **git status** очень похож на тот, который был после первого добавления файла **fileA.txt,** они всё же отличаются.

Когда совсем новый файл попадает в **staging area**, перед его названием указывается **new file**. Вот так: **new file: fileA.txt.**

Если файл уже однажды попадал в историю (с помощью коммита) и был изменён, после выполнения **git add** он будет записан уже так: **modified: fileA.txt.**

1. **Файл добавлен в staging area, но после этого изменён**

Вы добавили файл в **staging area**, но перед самым коммитом вспомнили важную мелочь. Например, вместо одного восклицательного знака в конце строки **Это файл A!** нужно поставить три.

Откройте текстовый редактор и добавьте нужные правки. Теперь можно выполнить коммит, но в любой непонятной ситуации сначала стоит вызвать **git status.** Он покажет следующее.

*# изменили fileA.txt*

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: fileA.txt

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: fileA.txt

Файл попал и в **staged (Changes to be committed),** и в **modified (Changes not staged for commit)**. В **staging area** находится версия файла с одним восклицательным знаком, а в **Changes not staged for commit** — уже изменённая версия, с тремя.

Чтобы закоммитить самую свежую версию файла, нужно снова выполнить **git add** перед коммитом.

**6. Работа над ошибками в коммитах**

**Стили оформления:**

* ***Корпоративный.***

Во многих компаниях применяется **Jira** — **система для организации проектов и задач**. У каждой задачи в **Jira** есть идентификатор из нескольких заглавных латинских букв и номера. Например, **LGS-239** значит, что это **239-я задача в проекте LGS** (сокращение от англ. logistics — «логистика»).

В корпоративном стиле в начале сообщения обычно указывают **Jira-ID**, а после — текст сообщения.

$ git commit -m "LGS-239: Дополнить список пасхалок новыми числами"

Какие-то команды могут договариваться, с какой части речи начинать сообщение и какой длины оно должно быть, какие-то — нет. Но требование о наличии Jira-ID обычно строгое: оно позволяет автоматически связывать коммиты с задачами и проектами.

* ***Conventional Commits***

Стандарт **Conventional Commits** (англ. «соглашение о коммитах») отличается качественной документацией и подробной проработкой. Он подходит для репозиториев с исходным кодом программ. Использовать его для других типов проектов (например, для перевода книги) было бы неудобно.

**Conventional Commits** предлагает такой формат коммита:

*<type>: <сообщение>.*

Первая часть **type** - это тип изменений. Таких типов достаточно много. Вот два примера:

* **feat** (сокращение от англ. feature) — для новой функциональности;
* **fix** (от англ. «исправить», «устранить») — для исправленных ошибок.

Более подробный список можно увидеть [на сайте с описанием этого стиля](https://www.conventionalcommits.org/ru/v1.0.0-beta.4/#%D1%81%D0%BF%D0%B5%D1%86%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F).

Например, сообщение может быть таким:

git commit -m "feat: добавить подсчёт суммы заказов за неделю"

* ***GitHub-стиль***

GitHub можно использовать не только для хранения файлов проекта, но и для ведения списка **задач** (англ. *issue*) этого проекта. Если коммит «закрывает» или «решает» какую-то задачу, то в его сообщении удобно указывать ссылку на неё. Для этого в любом месте сообщения нужно указать #<номер задачи>. Например, вот так:

$ git commit -m "Исправить #334, добавить график температуры"

В таком случае GitHub свяжет коммит и задачу.

💡 **Инфинитив и императив**

Для сообщений на русском языке часто рекомендуют использовать инфинитивы. Например: Добавить тесты для PipkaService, Исправить ошибку #123 и так далее.

Для сообщений на английском рекомендуется использовать **повелительное наклонение** (англ. *imperative*). Например: Use library mega\_lib\_300, Fix exit button и так далее.

Эти рекомендации сложились исторически, и им следуют многие проекты.