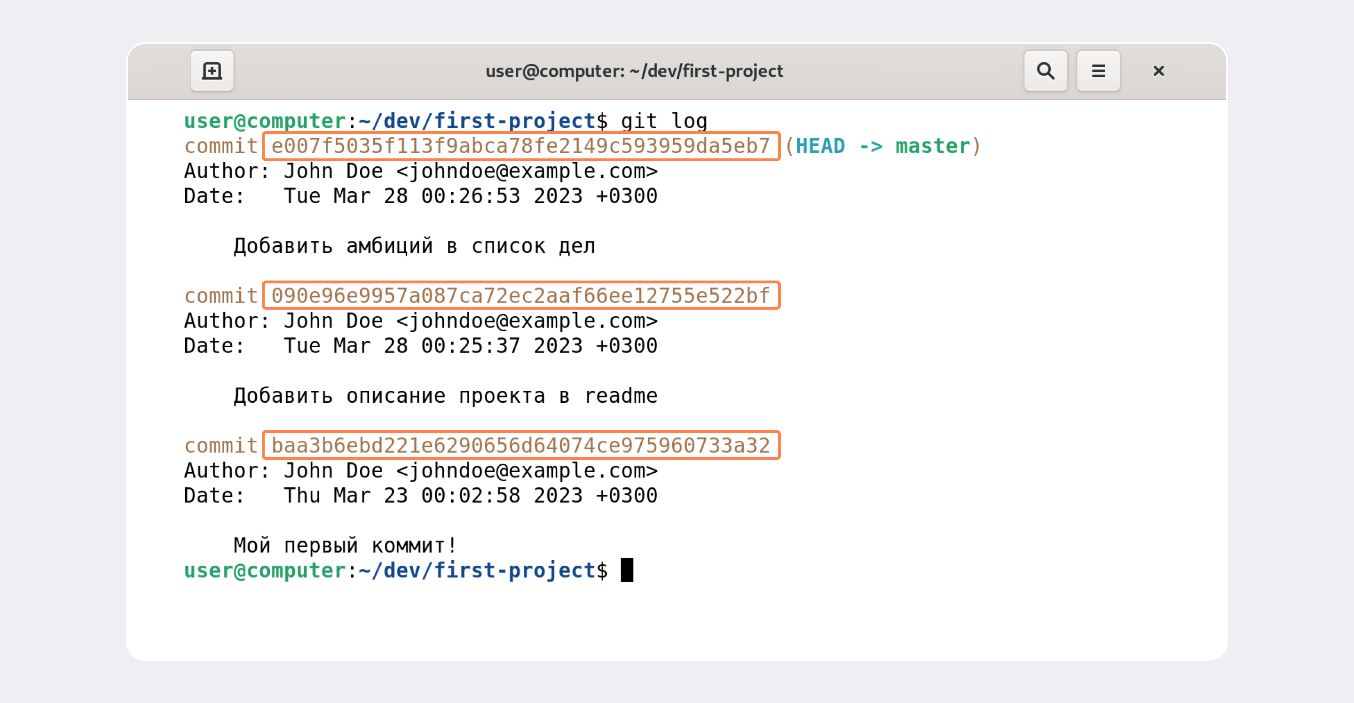
**Хеш — идентификатор коммита**

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.



В этом уроке разберём, что такое хеши и зачем они нужны.

**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.

Обычно хеш — это короткая (4040 символов в случае SHA-1) строка, которая состоит из цифр 0—90—9 и латинских букв A—F*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.

Начало формы

Выберите верные утверждения.

Правильный ответ

Таблица соответствия хеш → информация о коммите хранится в папке .git.

Git хранит всю свою служебную информацию в этой папке, в том числе и таблицу, в которой можно найти коммит по хешу.

Неправильный ответ

Основной идентификатор коммита — это имя автора + дата.

Основной идентификатор — хеш коммита, строка из 4040 символов.

Тоже правильный ответ

Основной идентификатор коммита — это его хеш.

Эта странная, но важная строка служит главным идентификатором коммита.



Если посчитать хеш одного и того же файла и одним и тем же алгоритмом, но на двух разных компьютерах, то результат может получиться разным.

Нет, результат будет тот же: он зависит только от содержимого файла (набора данных).

Правильный ответ

Если посчитать хеш одного и того же файла (одним и тем же алгоритмом) на двух разных компьютерах, то результат будет гарантированно одинаковым.

Хеш-алгоритм выдаст одно и то же значение для одного набора данных — это не зависит от того, где и как он был запущен.

Конец формы

Отлично! Теперь вы знаете, что такое хеш коммита и для чего он нужен. Подведём итоги:

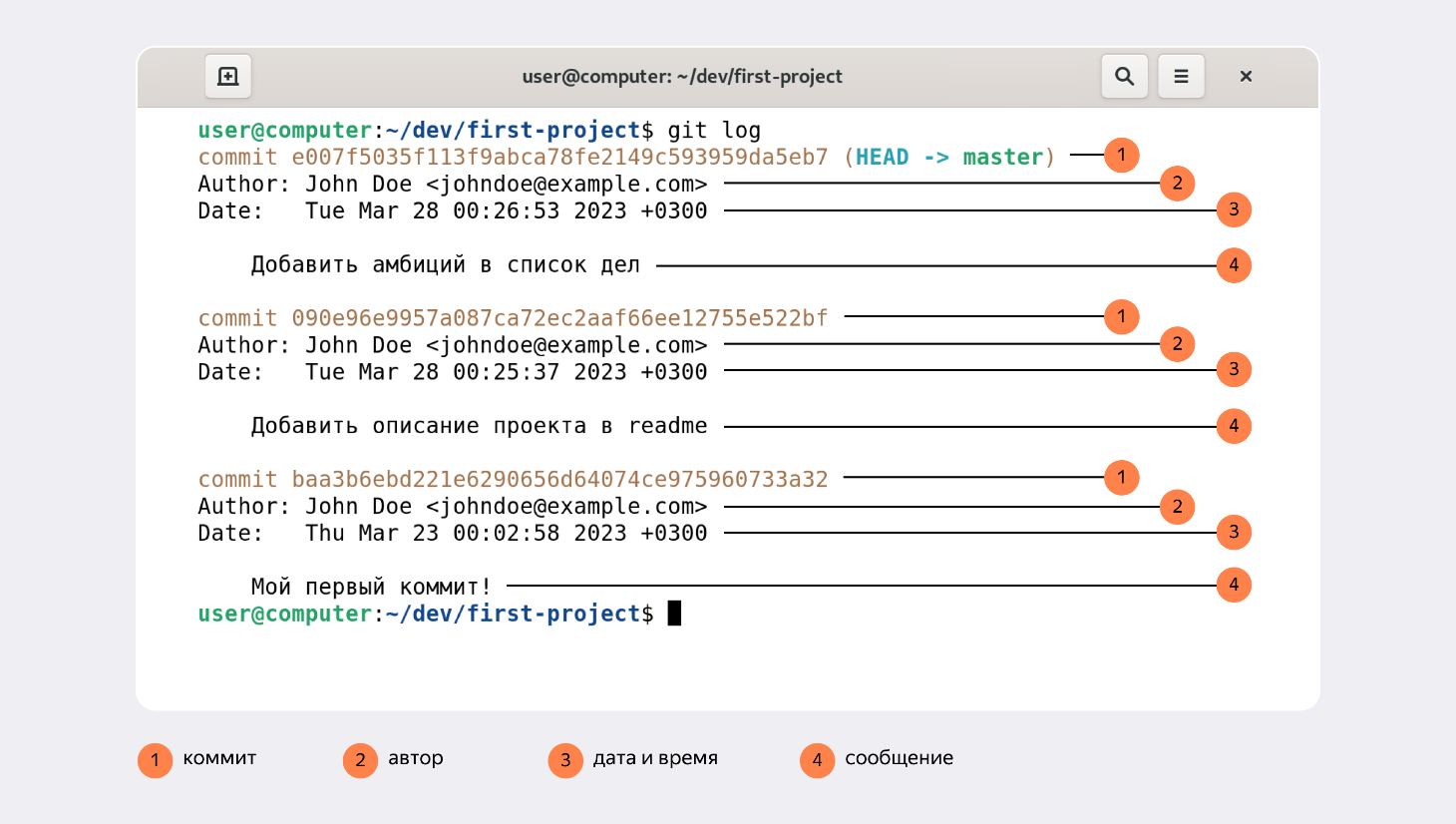
* Git преобразует информацию о коммитах с помощью алгоритма SHA-1 и для каждого из них рассчитывает уникальный идентификатор — хеш.
* Хеш — основной идентификатор коммита и позволяет узнать его автора, дату и содержимое закоммиченных файлов.
* Все хеши, а также таблицу соответствий хеш → информация о коммите Git хранит в папке .git.

**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.

Начало формы

Исходный код самого Git тоже хранится в Git-репозитории (тут уместно вспомнить про курицу и яйцо). Вот так выглядит описание самого первого коммита в репозитории Git. Изучите его.

Скопировать код

commit e83c5163316f89bfbde7d9ab23ca2e25604af290

Author: Linus Torvalds <torvalds@linux-foundation.org>

Date: Thu Apr 7 15:13:13 2005 -0700

Initial revision of "git", the information manager from hell

Выберите верные утверждения.

Правильный ответ

Автор проекта Git — Linus Torvalds

Это тот же Линус, который создал Linux.



Автор проекта Git — John Doe

Было бы здорово, но нет: имя автора указано в поле Author.



Git появился в июле $2005$ года

Первый коммит сделан 77 апреля 20052005 года.

Правильный ответ

Git появился в апреле 20052005 года

Об этом сообщает поле Date.



Git появился в 20082008 году

В 20082008 году появился GitHub, а Git — за три года до этого.

Тоже правильный ответ

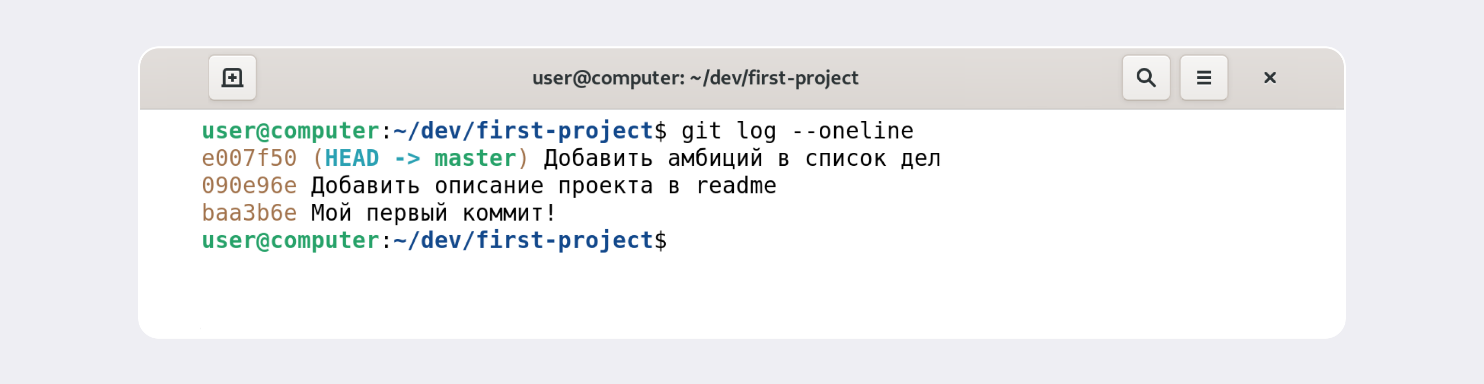
Автор описал Git как the information manager from hell

Буквально так и написано!

Конец формы

**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.

Начало формы

Сколько символов хеша выведет команда git log --oneline?



Меньше, чем нужно

git log --oneline выведет достаточное количество символов.



Правильный ответ

Столько, сколько нужно

Этой команде можно верить: она самостоятельно подберёт оптимальное количество символов.



Больше, чем нужно

Команда автоматически подберёт оптимальную длину хеша.



Ровно семь

git log --oneline может вывести и семь символов, но это происходит далеко не всегда.

Конец формы

Здорово! Самое важное в этом уроке:

* Можно вызвать не только полный лог, но и сокращённый — это делается командой git log --oneline.
* В сокращённом логе выводятся сокращённые хеши — их можно использовать точно так же, как и полные.

**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.

Скопировать код

BASH

$ 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.

Скопировать код

BASH

$ 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.

Начало формы

Выберите верные утверждения о HEAD.



Это команда Git, которая покажет подробную информацию о последнем (самом новом) коммите.

HEAD не команда, а файл, который указывает на последний коммит.

Правильный ответ

Это файл в папке .git, в котором записана ссылка (или ссылка на ссылку) на последний коммит.

HEAD — это файл в .git, в файле — ссылка, по ссылке — хеш. Совсем как в сказке: иголка — в яйце, яйцо — в утке, утка — в зайце.

Правильный ответ

Это синоним хеша последнего коммита — его можно передавать командам Git в качестве параметра.

HEAD указывает на последний коммит. Если передать его в качестве параметра, Git поймёт вас.



Это основная ветка по умолчанию в репозиториях, которые созданы на GitHub после 20202020 года.

Такая ветка называется main.



Это ссылка на самый первый (корневой) коммит в репозитории.

HEAD ссылается на самый новый, а не на первый коммит.

Конец формы

Супер! Папка .git содержит много непонятных файлов — об одном из них мы рассказали в этом уроке. Подытожим:

* В числе прочих файлов в папке .git есть служебный файл HEAD. Он указывает на самый свежий коммит.
* Вместо хеша последнего коммита можно написать слово HEAD — Git вас поймёт.

**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.

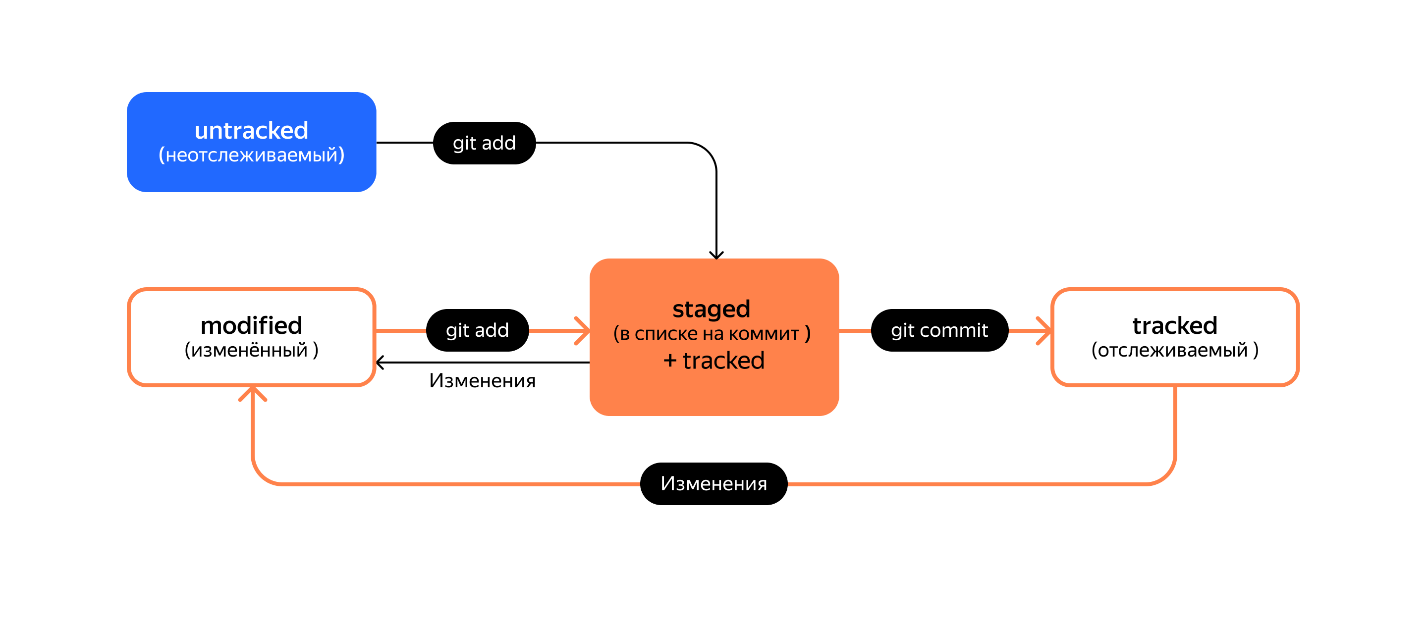
**Про staged и modified**

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 4−74−7 много-много раз.

Выглядит довольно запутанно! Но на практике разобраться с этим будет проще, чем кажется. Вы сможете наглядно проследить типичный жизненный цикл файла в Git в следующем уроке.

Начало формы

Выберите верные утверждения.



Файл может быть одновременно в состояниях tracked и untracked.

Это противоположные, несовместимые состояния.

Правильный ответ

Файл может быть одновременно в staged и modified.

Да, так может быть, если файл уже добавили в список на коммит через git add, а после этого изменили.



Файл может быть одновременно untracked и modified.

Состояние untracked означает, что изменения в файле не отслеживаются. А modified применяется только к отслеживаемым файлам.



Файл может быть одновременно untracked и staged.

Файл, попавший в staged, автоматически становится tracked.

Правильный ответ

Файл может быть tracked и staged.

Да, файл считается tracked, если он staged.

Конец формы

Ура! Это был сложный урок, зато теперь вы знаете о состояниях и жизненном цикле файлов в Git чуть больше. Важное:

* 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.

Скопировать код

BASH

$ 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.

Скопировать код

BASH

$ 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.

Скопировать код

BASH

$ 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.

Скопировать код

BASH

$ 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.

Скопировать код

BASH

$ 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.

Скопировать код

BASH

# 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.

Скопировать код

BASH

$ 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.

Скопировать код

BASH

# 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.

Теперь проверьте, как вы усвоили материал урока! Изучите скриншот с выводами команды git status для репозитория quiz-project.



Выполните задания ниже.

Начало формы

Выберите одно или несколько состояний, в которых находится файл A*A* в репозитории quiz-project.

Неправильный ответ

untracked

Файл не указан в списке Untracked files.

Тоже правильный ответ

tracked

Файл есть в папке, но не указан ни в staged, ни в untracked. Это значит, что его когда-то закоммитили и с тех пор не меняли.



staged

Файла нет в списке Changes to be committed.



modified

Файла нет в списке Changes not staged for commit.

Конец формы

Начало формы

Выберите одно или несколько состояний, в которых находится файл B*B* в репозитории quiz-project.



untracked

Файлы в staged одновременно находятся в tracked, а значит, не могут быть untracked. Также этого файла нет в списке Untracked files.

Тоже правильный ответ

tracked

Все файлы из staged являются tracked.

Тоже правильный ответ

staged

Файл указан в списке Changes to be committed.

Правильный ответ

modified

Файл указан в списке Changes not staged for commit.

Конец формы

Начало формы

Выберите одно или несколько состояний, в которых находится файл C*C* в репозитории quiz-project.

Правильный ответ

untracked

Да, файл в списке Untracked files.



tracked

Файл указан в списке Untracked files, а значит, не может быть tracked.



staged

В списке Changes to be committed файла нет.



modified

Файла нет в списке Changes not staged for commit.

Конец формы

Замечательно! Подытожим то, о чём рассказали в уроке:

* Команда git status всегда подскажет, что происходит с файлом: например, он добавлен в список «на коммит» или ещё вообще не отслеживается, или изменён.
* git status показывает явно следующие состояния файлов: untracked, staged и modified.
* git status подсказывает, какие команды можно выполнить, чтобы поменять состояние файла.

# Оформление сообщений к коммитам

То, как написаны сообщения коммитов, тоже может подчиняться определённым правилам. Иногда эти правила продиктованы культурой команды, а иногда техническими ограничениями.

Например, в выводе команды git log --oneline умещается максимум 7272 первых символа сообщения, поэтому многие правила включают пункт: «Сообщение не должно быть длиннее 7272 символов».

В этом уроке рассмотрим несколько популярных подходов к оформлению сообщений коммитов. Но сначала разберём, почему такие сообщения важны и зачем соблюдать правила их оформления.

### Зачем вообще писать сообщения

У каждого коммита в Git есть сообщение — то, что передаётся после параметра -m. Например: git commit -m "Добавить урок про оформление сообщений коммитов".

Сообщения коммитов можно сравнить с надписями на коробках в кладовке. Если надписей нет, то нужную коробку будет сложно найти: придётся заглянуть в каждую, чтобы понять, что там. А если надписи есть, то нужная найдётся сразу.

Как и надпись на коробке, сообщение коммита должно помочь определить, что внутри. Например, надпись на коробке «всякое разное» не очень полезная. Сообщение коммита «небольшие исправления» тоже: непонятно, что было исправлено в таком коммите и зачем.

Есть общие рекомендации по тому, как правильно составить сообщение. Оно должно быть:

* относительно коротким, чтобы его было легко прочитать;
* информативным.

Вот пример полезного сообщения в репозитории новостного сайта: Исправление опечатки в заголовке главной страницы на хорватском. Такое сообщение даёт много информации:

* Исправление опечатки значит, что исправлена ошибка, которая была допущена при наборе. Такое исправление не меняет смысл. То есть, например, главному редактору не нужно перепроверять этот заголовок.
* На хорватском говорит о том, что переводчикам на другие языки этот коммит можно смело пропускать.
* В заголовке главной страницы указывает, где произошли изменения. Если, например, кто-то зайдёт на сайт и ему не понравится новый заголовок, он легко найдёт по истории (git log) автора этого коммита и спросит у него, почему заголовок теперь такой.

Пример плохого сообщения для того же коммита: Исправлена опечатка. Это сообщение даёт мало информации. В такой коммит придётся «заглядывать» — разбираться, что именно поменялось и зачем.

## Стили оформления

Все люди разные и у всех есть предпочтения — в том числе, как формулировать сообщения коммитов. Кто-то использует инфинитивы: Исправить сообщение об ошибке E123, кто-то — глаголы в прошедшем времени: Исправил…, кто-то — существительные: Исправление….

Без единообразия коммитов нет и эффективной работы в Git. Это может показаться мелочью, но когда коммиты с сообщениями в разных стилях идут друг за другом, их может быть сложно читать.

Чтобы упростить работу, команды или даже целые компании часто договариваются об определённом стиле (то есть о правилах) оформления сообщений коммитов.

Например, правила могут быть такие:

* длина сообщения от 3030 до 7272 символов;
* первое слово — глагол в инфинитиве («исправить», «дополнить», «добавить» и другие);
* и так далее.

Есть много подходов к оформлению сообщений коммитов, но мы расскажем о нескольких популярных. Их используют как отдельные команды, так и целые проекты.

### 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 239239-я задача в проекте **LGS** (сокращение от англ. ***l***o***g***istic***s*** — «логистика»).

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

Скопировать код

BASH

$ 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.

Скопировать код

BASH

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.

Скопировать код

BASH

$ 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.

**Практическая работа №2. Дополняем шпаргалку**

Ура, практика! Работа над самой удобной шпаргалкой по Git ещё не закончена — в этот раз вы дополните её.

Мир IT не стоит на месте: в нём постоянно происходят перемены. Например, в коммерческой разработке готовый код редко остается без изменений: обновляются фреймворки, добавляются и совершенствуются функциональности, непрерывно оптимизируется работа всего проекта.

🔑 Кстати, фреймворки и библиотеки значительно облегчают разработку проекта. К ним относятся, например, **Flask** — микрофреймворк для создания веб-приложений на языке Python; **React** — библиотека с открытым исходным кодом для разработки пользовательских интерфейсов на JavaScript; **Express** — фреймворк для среды разработки Node.js, и так далее.

Подробнее об этих и прочих популярных технологиях, вы можете узнать на наших курсах по программированию — [«Python-разработчик плюс»](https://practicum.yandex.ru/python-developer-plus/?from=catalog), [«Фулстек-разработчик»](https://practicum.yandex.ru/fullstack-developer/?from=catalog) и других. Вводная часть этих курсов бесплатная

Подобно всем реальным проектам, ваша шпаргалка тоже требует обновления — ведь с момента прошлой практической работы вы узнали много нового. Дополните её:

1. Добавьте в шпаргалку основную информацию из уроков о хеше, логе, HEAD, статусах файлов и оформлении сообщений к коммитам.
2. Информацию из каждого урока заливайте новым коммитом. Так вам будет проще отслеживать, как меняется шпаргалка и увеличивается объём ваших знаний.

💡 Пишите понятные и информативные сообщения к коммитам. Используйте ключевые слова из названий уроков. Например: Добавить раздел про HEAD, Дополнить инфу про git log и так далее.

1. Особое внимание уделите материалу о статусах и жизненном цикле файлов в Git. Схему изменения статусов можно описать словами. Например, modified+git add = staged.

А можно усложнить задачу! Попробуйте использовать формат описания схем [Mermaid](https://github.blog/2022-02-14-include-diagrams-markdown-files-mermaid/). Принцип такой: вы описываете схему в специальном текстовом формате, а GitHub превращает описание в полноценную схему с блоками и стрелками.

Подсказка: как сделать mermaid-схему

Чтобы получить mermaid-схему в README.md, нужно добавить блок кода типа mermaid.

Скопировать код

MARKDOWN

HEAD -- это голова.

Коммит -- это всему голова.

Статусы файлов:

<тут пустая строка!>

```mermaid

%% описание схемы

```

<и тут пустая строка!>

* Блоки кода в маркдауне начинаются и заканчиваются тремя символами ```. После первых трёх ``` можно указать, какой именно код будет внутри блока. Например: ```mermaid , ```bash, ```python, ```javascript и так далее. Если ничего не указать, GitHub будет считать весь код простым текстом.

💡 Перед блоком и после него нужны пустые строки, иначе GitHub может не понять, что это блок кода.

* Два символа %% обозначают в mermaid строку-комментарий.
* Чтобы сделать схему, нужно указать формат: graph LR. **Graph** — это простейший тип схем; для шпаргалки его будет достаточно.
* Чтобы добавить элементы и связи (стрелки), используют строки вида A --> B. Такая строка создаст квадратные блоки А и B и соединит их стрелкой.   
  Дополнительно можно указывать текст на стрелке. Например, так: A -- "text" --> B.

Ниже вы найдёте заготовку для схемы статусов файлов.

Скопировать код

```mermaid

graph LR;

untracked -- "git add" --> staged;

staged -- "???" --> tracked/comitted;

%% стрелка без текста для примера:

A --> B;

```

Попробуйте описать схему самостоятельно и посмотреть, что получится!

После выполнения задания скопируйте в поле ниже вывод git log --oneline.

Ваш ответ правильный

9672083 (HEAD -> main) Дополнить инфу про git a69ad7d (origin/main) Add readme file

Поздравляем: ваш проект-помощник растёт вместе с вашими навыками!

На этом закончим. В следующей практической работе вас ждёт новая ситуация. Статус ваших знаний по Git — modified!