Установка

Установить git на свою машину очень просто:

* Linux — нужно просто открыть терминал и установить приложение при помощи пакетного менеджера вашего дистрибутива. Для Ubuntu команда будет выглядеть следующим образом:

sudo apt-get install git

* Windows — мы рекомендуем git for windows, так как он содержит и клиент с графическим интерфейсом, и эмулятор bash. <https://git-scm.com/download/win>
* OS X — проще всего воспользоваться homebrew. После его установки запустите в терминале: <https://sourceforge.net/projects/git-osx-installer/>

brew install git

Для новичка клиент с графическим интерфейсом (например, GitHub Desktop и Sourcetree) будет полезен, но, тем не менее, знать команды очень важно.

# Настройка

После установки git, теперь нужно добавить немного настроек. Есть довольно много опций, с которыми можно играть, но настроим самые важные: наше имя пользователя и адрес электронной почты. Откройте терминал и запустите команды:

git config --global user.name "My Name"

git config --global user.email myEmail@example.com

Теперь каждое наше действие будет отмечено именем и почтой. Таким образом, пользователи всегда будут в курсе, кто отвечает за какие изменения — это вносит порядок.

Git хранит весь пакет конфигураций в файле .gitconfig, находящемся в вашем локальном каталоге. Чтобы сделать эти настройки глобальными, то есть применимыми ко всем проектам, необходимо добавить флаг –global. Если вы этого не сделаете, они будут распространяться только на текущий репозиторий.

Для того, чтобы посмотреть все настройки системы, используйте команду:

git config --list

git config

git help [команда]

git [команда] --help

Для удобства и легкости зрительного восприятия, некоторые группы команд в Гит можно выделить цветом, для этого нужно прописать в консоли:

git config --global color.ui true

git config --global color.status auto

git config --global color.branch auto

Git относительно дружелюбен и всегда подскажет разработчику, если тот запутался, например:

1. Команда git --help - выводит общую документацию по git
2. Если введем git log --help - он предоставит нам документацию по какой-то определенной команде (в данном случае это - log)
3. Если вы вдруг сделали опечатку - система подскажет вам нужную команду
4. После выполнения любой команды - отчитается о том, что вы натворили
5. Также Гит прогнозирует дальнейшие варианты развития событий и всегда направит разработчика, не знающего, куда двигаться дальше

# Создание нового репозитория

Как было отмечено ранее, git хранит свои файлы и историю прямо в папке проекта. Чтобы создать новый репозиторий, нужно открыть терминал, зайти в папку проекта и выполнить команду init. Это включит приложение в этой конкретной папке и создаст скрытую директорию .git, где будет храниться история репозитория и настройки.

Создайте на рабочем столе папку под названием git\_exercise. Для этого в окне терминала введите:

$ mkdir Desktop/git\_exercise/

$ cd Desktop/git\_exercise/

$ git init

Командная строка должна вернуть что-то вроде:

Initialized empty Git repository in /home/user/Desktop/git\_exercise/.git/

Это значит, что репозиторий был успешно создан, но пока что пуст. Теперь создайте текстовый файл под названием hello.txt и сохраните его в директории git\_exercise.

# Определение состояния

status — это еще одна важнейшая команда, которая показывает информацию о текущем состоянии репозитория: актуальна ли информация на нём, нет ли чего-то нового, что поменялось, и так далее. Запуск git status на нашем свежесозданном репозитории должен выдать:

$ git status

On branch master

Initial commit

Untracked files:

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

hello.txt

Сообщение говорит о том, что файл hello.txt неотслеживаемый. Это значит, что файл новый и система еще не знает, нужно ли следить за изменениями в файле или его можно просто игнорировать. Для того, чтобы начать отслеживать новый файл, нужно его специальным образом объявить.

# Подготовка файлов

В git есть концепция области подготовленных файлов. Можно представить ее как холст, на который наносят изменения, которые нужны в коммите. Сперва он пустой, но затем мы добавляем на него файлы (или части файлов, или даже одиночные строчки) командой add и, наконец, коммитим все нужное в репозиторий (создаем слепок нужного нам состояния) командой commit.

В нашем случае у нас только один файл, так что добавим его:

$ git add hello.txt

Если нам нужно добавить все, что находится в директории, мы можем использовать

$ git add -A

Проверим статус снова, на этот раз мы должны получить другой ответ:

$ git status

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached ..." to unstage)

new file: hello.txt

Файл готов к коммиту. Сообщение о состоянии также говорит нам о том, какие изменения относительно файла были проведены в области подготовки — в данном случае это новый файл, но файлы могут быть модифицированы или удалены.

# Фиксация изменений

## **Как сделать коммит**

Представим, что нам нужно добавить пару новых блоков в html-разметку (index.html) и стилизовать их в файле style.css. Для сохранения изменений, их необходимо закоммитить. Но сначала, мы должны обозначить эти файлы для Гита, при помощи команды git add, добавляющей (или подготавливающей) их к коммиту. Добавлять их можно по отдельности:

git add index.html

git add css/style.css

или вместе - всё сразу:

git add .

Конечно добавлять всё сразу удобнее, чем прописывать каждую позицию отдельно. Однако, тут надо быть внимательным, чтобы не добавить по ошибке ненужные элементы. Если же такое произошло изъять оттуда ошибочный файл можно при помощи команды

git reset:

git reset css/style.css

Теперь создадим непосредственно сам коммит

git commit -m 'Add some code'

Флажок -m задаст commit message - комментарий разработчика. Он необходим для описания закоммиченных изменений. И здесь работает золотое правило всех комментариев в коде: «Максимально ясно, просто и содержательно обозначь написанное!»

## **Как посмотреть коммиты**

Для просмотра все выполненных фиксаций можно воспользоваться историей коммитов. Она содержит сведения о каждом проведенном коммите проекта. Запросить ее можно при помощи команды:

git log

В ней содержится вся информация о каждом отдельном коммите, с указанием его хэша, автора, списка изменений и даты, когда они были сделаны. Отследить интересующие вас операции в списке изменений, можно по хэшу коммита, при помощи команды git show :

git show hash\_commit

Ну а если вдруг нам нужно переделать commit message и внести туда новый комментарий, можно написать следующую конструкцию:

git commit --amend -m 'Новый комментарий'

В данном случае сообщение последнего коммита перезапишется. Но злоупотреблять этим не стоит, поскольку эта операция опасная и лучше ее делать до отправки коммита на сервер.

# Удаленные репозитории

Сейчас наш коммит является локальным — существует только в директории .git на нашей файловой системе. Несмотря на то, что сам по себе локальный репозиторий полезен, в большинстве случаев мы хотим поделиться нашей работой или доставить код на сервер, где он будет выполняться.

## **1. Что такое удаленный репозиторий**

Репозиторий, хранящийся в облаке, на стороннем сервисе, специально созданном для работы с git имеет ряд преимуществ. Во-первых - это своего рода резервная копия вашего проекта, предоставляющая возможность безболезненной работы в команде. А еще в таком репозитории можно пользоваться дополнительными возможностями хостинга. К примеру -визуализацией истории или возможностью разрабатывать вашу программу непосредственно в веб-интерфейсе.

**Клонирование**

Клонирование - это когда вы копируете удаленный репозиторий к себе на локальный ПК. Это то, с чего обычно начинается любой проект. При этом вы переносите себе все файлы и папки проекта, а также всю его историю с момента его создания. Чтобы склонировать проект, сперва, необходимо узнать где он расположен и скопировать ссылку на него. В нашем руководстве мы будем использовать адрес https://github.com/tutorialzine/awesome-project, но вам посоветуем, попробовать создать свой репозиторий в GitHub, BitBucket или любом другом сервисе:

git clone https://github.com/tutorialzine/awesome-project

При клонировании в текущий каталог, там будет создана папка, в которую поместятся все проектные файлы и скрытая директория .git, с самим репозиторием, или с необходимой информацией о нем. В такой ситуации, для клонируемого репозитория, по умолчанию, будет создана папка с одноименным названием, но его можно залить и в другую директорию, например:

git clone https://github.com/tutorialzine/awesome-project new-folder

# 2. Подключение к удаленному репозиторию

Чтобы загрузить что-нибудь в удаленный репозиторий, сначала нужно к нему подключиться. Регистрация и установка может занять время, но все подобные сервисы предоставляют хорошую документацию.

Чтобы связать наш локальный репозиторий с репозиторием на GitHub, выполним следующую команду в терминале. Обратите внимание, что нужно обязательно изменить URI репозитория на свой.

# This is only an example. Replace the URI with your own repository address.

$ git remote add origin https://github.com/tutorialzine/awesome-project.git

Проект может иметь несколько удаленных репозиториев одновременно. Чтобы их различать, мы дадим им разные имена. Обычно главный репозиторий называется origin.

## **3. Отправка изменений на сервер**

Сейчас самое время переслать наш локальный коммит на сервер. Этот процесс происходит каждый раз, когда мы хотим обновить данные в удаленном репозитории.

Команда, предназначенная для этого -push. Она принимает два параметра: имя удаленного репозитория (мы назвали наш origin) и ветку, в которую необходимо внести изменения (master — это ветка по умолчанию для всех репозиториев).

$ git push origin master

Counting objects: 3, done.

Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/tutorialzine/awesome-project.git

\* [new branch] master -> master

Эта команда немного похожа на git fetch, с той лишь разницей, что при помощи fetch мы импортируем коммиты в локальную ветку, а применив push, мы экспортируем их из локальной в удаленную. Если вам необходимо настроить удаленную ветку используйте git remote. Однако пушить надо осторожно, ведь рассматриваемая команда перезаписывает безвозвратно все изменения. В большинстве случаев, ее используют, чтобы опубликовать выгружаемые локальные изменения в центральный репозиторий. А еще ее применяют для того, чтобы поделиться, внесенными в локальный репозиторий, нововведениями, с коллегами или другими удаленными участниками разработки проекта. Подытожив сказанное, можно назвать git push - командой выгрузки, а git pull и git fetch - командами загрузки или скачивания. После того как вы успешно запушили измененные данные, их необходимо внедрить или интегрировать, при помощи команды слияния git merge.

В зависимости от сервиса, который вы используете, вам может потребоваться аутентифицироваться, чтобы изменения отправились. Если все сделано правильно, то когда вы посмотрите в удаленный репозиторий при помощи браузера, вы увидите файл hello.txt

## **4. Запрос изменений с сервера**

Если вы сделали изменения в вашем удаленном репозитории, другие пользователи могут скачать изменения при помощи команды pull.

$ git pull origin master

From https://github.com/tutorialzine/awesome-project

\* branch master -> FETCH\_HEAD

Already up-to-date.

Так как новых коммитов с тех пор, как мы склонировали себе проект, не было, никаких изменений доступных для скачивания нет.

## **Как удалить локальный репозиторий**

Вам не понравился один из ваших локальных Git-репозиториев и вы хотите стереть его со своей машины. Для этого вам всего лишь надо удалить скрытую папку «.git» в корневом каталоге репозитория. Сделать это можно 3 способами:

1. Проще всего вручную удалить эту папку «.git» в корневом каталоге «Git Local Warehouse».
2. Также удалить, не устраивающий вас, репозиторий можно на github. Открываете нужный вам объект и переходите в пункт меню Настройки. Там, прокрутив ползунок вниз, вы попадете в зону опасности, где один из пунктов будет называться «удаление этого хранилища».
3. Последний метод удаления локального хранилища через командную строку, для этого в терминале необходимо ввести следующую команду:

cd repository-path/

rm -r .git

# Ветвление

Во время разработки новой функциональности считается хорошей практикой работать с копией оригинального проекта, которую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока вы не решаете слить изменения вместе. Это происходит по набору причин:

* Уже рабочая, стабильная версия кода сохраняется.
* Различные новые функции могут разрабатываться параллельно разными программистами.
* Разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.
* В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.

## **1. Создание новой ветки**

Основная ветка в каждом репозитории называется master. Чтобы создать еще одну ветку, используем команду branch <name>

$ git branch amazing\_new\_feature

Это создаст новую ветку, пока что точную копию ветки master.

## **2. Переключение между ветками**

Сейчас, если мы запустим branch, мы увидим две доступные опции:

$ git branch

amazing\_new\_feature

\* master

master — это активная ветка, она помечена звездочкой. Но мы хотим работать с нашей “новой потрясающей фичей”, так что нам понадобится переключиться на другую ветку. Для этого воспользуемся командой checkout, она принимает один параметр — имя ветки, на которую необходимо переключиться.

$ git checkout amazing\_new\_feature

В Git ветка — это отдельная линия разработки. Git checkout позволяет нам переключаться как между удаленными, так и меду локальными ветками. Это один из способов получить доступ к работе коллеги или соавтора, обеспечивающий более высокую продуктивность совместной работы. Однако тут надо помнить, что пока вы не закомитили изменения, вы не сможете переключиться на другую ветку. В такой ситуации нужно либо сделать коммит, либо отложить его, при помощи команды git stash, добавляющей текущие незакоммиченные изменения в стек изменений и сбрасывающей рабочую копию до HEAD'а репозитория.

## **3. Слияние веток**

Наша “потрясающая новая фича” будет еще одним текстовым файлом под названием feature.txt. Мы создадим его, добавим и закоммитим:

$ git add feature.txt

$ git commit -m "New feature complete.”

Изменения завершены, теперь мы можем переключиться обратно на ветку master.

$ git checkout master

Теперь, если мы откроем наш проект в файловом менеджере, мы не увидим файла feature.txt, потому что мы переключились обратно на ветку master, в которой такого файла не существует. Чтобы он появился, нужно воспользоваться merge для объединения веток (применения изменений из ветки amazing\_new\_feature к основной версии проекта).

$ git merge amazing\_new\_feature

Теперь ветка master актуальна. Ветка amazing\_new\_feature больше не нужна, и ее можно удалить.

$ git branch -d awesome\_new\_feature

Если хотите создать копию удаленного репозитория - используйте git clone. Однако если вам нужна только определенная его ветка, а не все хранилище - после git clone выполните следующую команду в соответствующем репозитории:

git checkout -b <имя ветки> origin/<имя ветки>

После этого, новая ветка создается на машине автоматически.

# Как удалять ветки в Git?

Бывают ситуации, когда после слива каких-то изменений из рабочей ветки в исходную версию проекта, ее, по правилам хорошего тона, необходимо удалить, чтобы она более не мешалась в вашем коде. Но как это сделать?

Для локально расположенных веток существует команда:

git branch -d local\_branch\_name

где флажок -d являющийся опцией команды git branch - это сокращенная версия ключевого слова --delete, предназначенного для удаления ветки, а local\_branch\_name – название ненужной нам ветки.

Однако тут есть нюанс: удалить текущую ветку, в которую вы, в данный момент просматриваете - нельзя. Если же вы все-таки попытаетесь это сделать, система отругает вас и выдаст ошибку с таким содержанием:

Error: Cannot delete branch local\_branch\_name checked out at название\_директории

Так что при удалении ветвей, обязательно переключитесь на другой branch.

# Дополнительно

В последней части этого руководства будет показано некоторые дополнительные трюки, которые могут вам помочь.

## **1. Отслеживание изменений, сделанных в коммитах**

У каждого коммита есть свой уникальный идентификатор в виде строки цифр и букв. Чтобы просмотреть список всех коммитов и их идентификаторов, можно использовать команду log:

[spoiler title='Вывод git log']

$ git log

commit ba25c0ff30e1b2f0259157b42b9f8f5d174d80d7

Author: Tutorialzine

Date: Mon May 30 17:15:28 2016 +0300

New feature complete

commit b10cc1238e355c02a044ef9f9860811ff605c9b4

Author: Tutorialzine

Date: Mon May 30 16:30:04 2016 +0300

Added content to hello.txt

commit 09bd8cc171d7084e78e4d118a2346b7487dca059

Author: Tutorialzine

Date: Sat May 28 17:52:14 2016 +0300

Initial commit

[/spoiler]

Как вы можете заметить, идентификаторы довольно длинные, но для работы с ними не обязательно копировать их целиком — первых нескольких символов будет вполне достаточно. Чтобы посмотреть, что нового появилось в коммите, мы можем воспользоваться командой show [commit]

[spoiler title='Вывод git show']

$ git show b10cc123

commit b10cc1238e355c02a044ef9f9860811ff605c9b4

Author: Tutorialzine

Date: Mon May 30 16:30:04 2016 +0300

Added content to hello.txt

diff --git a/hello.txt b/hello.txt

index e69de29..b546a21 100644

--- a/hello.txt

+++ b/hello.txt

@@ -0,0 +1 @@

+Nice weather today, isn't it?

[/spoiler]

Чтобы увидеть разницу между двумя коммитами, используется команда diff (с указанием промежутка между коммитами):

[spoiler title='Вывод git diff']

$ git diff 09bd8cc..ba25c0ff

diff --git a/feature.txt b/feature.txt

new file mode 100644

index 0000000..e69de29

diff --git a/hello.txt b/hello.txt

index e69de29..b546a21 100644

--- a/hello.txt

+++ b/hello.txt

@@ -0,0 +1 @@

+Nice weather today, isn't it?

[/spoiler]

Мы сравнили первый коммит с последним, чтобы увидеть все изменения, которые были когда-либо сделаны. Обычно проще использовать git difftool, так как эта команда запускает графический клиент, в котором наглядно сопоставляет все изменения.

## **2. Возвращение файла к предыдущему состоянию**

Гит позволяет вернуть выбранный файл к состоянию на момент определенного коммита. Это делается уже знакомой нам командой checkout, которую мы ранее использовали для переключения между ветками. Но она также может быть использована для переключения между коммитами (это довольно распространенная ситуация для Гита - использование одной команды для различных, на первый взгляд, слабо связанных задач).

В следующем примере мы возьмем файл hello.txt и откатим все изменения, совершенные над ним к первому коммиту. Чтобы сделать это, мы подставим в команду идентификатор нужного коммита, а также путь до файла:

$ git checkout 09bd8cc1 hello.txt

## **3. Исправление коммита**

Если вы опечатались в комментарии или забыли добавить файл и заметили это сразу после того, как закоммитили изменения, вы легко можете это поправить при помощи commit —amend. Эта команда добавит все из последнего коммита в область подготовленных файлов и попытается сделать новый коммит. Это дает вам возможность поправить комментарий или добавить недостающие файлы в область подготовленных файлов.

Для более сложных исправлений, например, не в последнем коммите или если вы успели отправить изменения на сервер, нужно использовать revert. Эта команда создаст коммит, отменяющий изменения, совершенные в коммите с заданным идентификатором.

Самый последний коммит может быть доступен по алиасу HEAD:

$ git revert HEAD

Для остальных будем использовать идентификаторы:

$ git revert b10cc123

При отмене старых коммитов нужно быть готовым к тому, что возникнут конфликты. Такое случается, если файл был изменен еще одним, более новым коммитом. И теперь git не может найти строчки, состояние которых нужно откатить, так как они больше не существуют.

## **4. Разрешение конфликтов при слиянии**

Помимо сценария, описанного в предыдущем пункте, конфликты регулярно возникают при слиянии ветвей или при отправке чужого кода. Иногда конфликты исправляются автоматически, но обычно с этим приходится разбираться вручную — решать, какой код остается, а какой нужно удалить.

Давайте посмотрим на примеры, где мы попытаемся слить две ветки под названием john\_branch и tim\_branch. И Тим, и Джон правят один и тот же файл: функцию, которая отображает элементы массива.

Джон использует цикл:

// Use a for loop to console.log contents.

for(var i=0; i<arr.length; i++) {

console.log(arr[i]);

}

Тим предпочитает forEach:

// Use forEach to console.log contents.

arr.forEach(function(item) {

console.log(item);

});

Они оба коммитят свой код в соответствующую ветку. Теперь, если они попытаются слить две ветки, они получат сообщение об ошибке:

$ git merge tim\_branch

Auto-merging print\_array.js

CONFLICT (content): Merge conflict in print\_array.js

Automatic merge failed; fix conflicts and then commit the result.

Система не смогла разрешить конфликт автоматически, значит, это придется сделать разработчикам. Приложение отметило строки, содержащие конфликт:

[spoiler title='Вывод']

<<<<<<< HEAD // Use a for loop to console.log contents. for(var i=0; i<arr.length; i++) { console.log(arr[i]); } ======= // Use forEach to console.log contents. arr.forEach(function(item) { console.log(item); }); >>>>>>> Tim's commit.

[/spoiler]

Над разделителем ======= мы видим последний (HEAD) коммит, а под ним - конфликтующий. Таким образом, мы можем увидеть, чем они отличаются и решать, какая версия лучше. Или вовсе написать новую. В этой ситуации мы так и поступим, перепишем все, удалив разделители, и дадим git понять, что закончили.

// Not using for loop or forEach.

// Use Array.toString() to console.log contents.

console.log(arr.toString());

Когда все готово, нужно закоммитить изменения, чтобы закончить процесс:

$ git add -A

$ git commit -m "Array printing conflict resolved."

Как вы можете заметить, процесс довольно утомительный и может быть очень сложным в больших проектах. Многие разработчики предпочитают использовать для разрешения конфликтов клиенты с графическим интерфейсом. (Для запуска нужно набрать git mergetool).

## **5. Настройка .gitignore**

В большинстве проектов есть файлы или целые директории, в которые мы не хотим (и, скорее всего, не захотим) коммитить. Мы можем удостовериться, что они случайно не попадут в git add -A при помощи файла .gitignore

1. Создайте вручную файл под названием .gitignore и сохраните его в директорию проекта.
2. Внутри файла перечислите названия файлов/папок, которые нужно игнорировать, каждый с новой строки.
3. Файл .gitignore должен быть добавлен, закоммичен и отправлен на сервер, как любой другой файл в проекте.

Вот хорошие примеры файлов, которые нужно игнорировать:

* Логи
* Артефакты систем сборки
* Папки node\_modules в проектах node.js
* Папки, созданные IDE, например, Netbeans или IntelliJ
* Разнообразные заметки разработчика.

Файл .gitignore, исключающий все перечисленное выше, будет выглядеть так:

\*.log

build/

node\_modules/

.idea/

my\_notes.txt

Символ слэша в конце некоторых линий означает директорию (и тот факт, что мы рекурсивно игнорируем все ее содержимое). Звездочка, как обычно, означает шаблон.

# Git bash и git.io

Руководствуясь часто встречающимися, при изучении системы, вопросами новичков, разберем еще несколько непонятных словосочетаний.

* Git Bash(Bourne Again Shell) — это приложение, являющееся эмулятором командной строки и предоставляющее, операционной системе, некоторые распространенные утилиты bash и собственно саму систему Git. Это терминал, используемый для взаимодействия с персональным компьютером, посредством письменных команд.
* URL-адреса хранилищ на Гитхабе могут быть довольно длинными, из-за больших имен репозиториев и файлов. Работать с такими ссылками очень не удобно. Поэтому сайт github.io создал git.io — неплохой сервис по преобразованию этих длинных и беспорядочных URL-адресов в более короткие и понятные. Сайт был создан в 2011 году и вплоть до недавнего времени отлично справлялся со своими обязанностями. Однако в начале этого года компания Гитхаб, из-за участившихся попыток хакеров использовать сайт в злонамеренных целях, остановила работу сервиса, а чем известила пользователей в [своем блоге](https://github.blog/changelog/2022-01-11-git-io-no-longer-accepts-new-urls/). Разработчики популярного ресурса рекомендуют пользоваться другими URL-cutter’ами, пока работа сервиса не будет налажена.

# Заключение.

Git довольно сложен, и в нем есть еще много функций и трюков. Если вы хотите с ними познакомиться, вот некоторые ресурсы, которые мы рекомендуем:

* Официальная документация, включающая книгу и видеоуроки – <https://git-scm.com/doc>
* “Getting git right” – Коллекция руководств и статей от Atlassian – <https://www.atlassian.com/git>
* Список клиентов с графическим интерфейсом –  <https://www.git-scm.com/downloads/guis>
* Онлайн утилита для генерации .gitignore файлов – <https://www.gitignore.io/>

# Шпаргалка по основным командам Git

### **git add**

Команда git add добавляет содержимое рабочей директории в индекс (staging area) для последующего коммита. По умолчанию git commit использует лишь этот индекс, так что вы можете использовать git add для сборки слепка вашего следующего коммита.

### **git status**

Команда git status показывает состояния файлов в рабочей директории и индексе: какие файлы изменены, но не добавлены в индекс; какие ожидают коммита в индексе. Вдобавок к этому выводятся подсказки о том, как изменить состояние файлов.

### **git diff**

Команда git diff используется для вычисления разницы между любыми двумя Git деревьями. Это может быть разница между вашей рабочей директорией и индексом (собственно git diff), разница между индексом и последним коммитом (git diff --staged), или между любыми двумя коммитами (git diff master branchB).

### **git difftool**

Команда git difftool просто запускает внешнюю утилиту сравнения для показа различий в двух деревьях, на случай если вы хотите использовать что-либо отличное от встроенного просмотрщика git diff.

### **git commit**

Команда git commit берёт все данные, добавленные в индекс с помощью git add, и сохраняет их слепок во внутренней базе данных, а затем сдвигает указатель текущей ветки на этот слепок.

### **git reset**

Команда git reset, как можно догадаться из названия, используется в основном для отмены изменений. Она изменяет указатель HEAD и, опционально, состояние индекса. Также эта команда может изменить файлы в рабочей директории при использовании параметра --hard, что может привести к потере наработок при неправильном использовании, так что убедитесь в серьёзности своих намерений прежде чем использовать его.

### **git rm**

Команда git rm используется в Git для удаления файлов из индекса и рабочей директории. Она похожа на git add с тем лишь исключением, что она удаляет, а не добавляет файлы для следующего коммита.

### **git mv**

Команда git mv — это всего лишь удобный способ переместить файл, а затем выполнить git addдля нового файла и git rm для старого.

### **git clean**

Команда git clean используется для удаления мусора из рабочей директории. Это могут быть результаты сборки проекта или файлы конфликтов слияний.

# Шпаргалка по ветвлению и слиянию

### **git branch**

Команда git branch — это своего рода “менеджер веток”. Она умеет перечислять ваши ветки, создавать новые, удалять и переименовывать их.

### **git checkout**

Команда git checkout используется для переключения веток и выгрузки их содержимого в рабочую директорию.

### **git merge**

Команда git merge используется для слияния одной или нескольких веток в текущую. Затем она устанавливает указатель текущей ветки на результирующий коммит.

### **git mergetool**

Команда git mergetool просто вызывает внешнюю программу слияний, в случае если у вас возникли проблемы слияния.

### **git log**

Команда git log используется для просмотра истории коммитов, начиная с самого свежего и уходя к истокам проекта. По умолчанию, она показывает лишь историю текущей ветки, но может быть настроена на вывод истории других, даже нескольких сразу, веток. Также её можно использовать для просмотра различий между ветками на уровне коммитов.

### **git stash**

Команда git stash используется для временного сохранения всех незакоммиченных изменений для очистки рабочей директории без необходимости коммитить незавершённую работу в новую ветку.

### **git tag**

Команда git tag используется для задания постоянной метки на какой-либо момент в истории проекта. Обычно она используется для релизов.

# Шпаргалка по совместной работе и обновлению проектов

Не так уж много команд в Git требуют сетевого подключения для своей работы, практически все команды оперируют с локальной копией проекта. Когда вы готовы поделиться своими наработками, всего несколько команд помогут вам работать с удалёнными репозиториями.

### **git fetch**

Команда git fetch связывается с удалённым репозиторием и забирает из него все изменения, которых у вас пока нет и сохраняет их локально.

### **git pull**

Команда git pull работает как комбинация команд git fetch и git merge, т.е. Git вначале забирает изменения из указанного удалённого репозитория, а затем пытается слить их с текущей веткой.

### **git push**

Команда git push используется для установления связи с удалённым репозиторием, вычисления локальных изменений отсутствующих в нём, и собственно их передачи в вышеупомянутый репозиторий. Этой команде нужно право на запись в репозиторий, поэтому она использует аутентификацию.

### **git remote**

Команда git remote служит для управления списком удалённых репозиториев. Она позволяет сохранять длинные URL репозиториев в виде понятных коротких строк, например "origin", так что вам не придётся забивать голову всякой ерундой и набирать её каждый раз для связи с сервером. Вы можете использовать несколько удалённых репозиториев для работы и git remote поможет добавлять, изменять и удалять их.

### **git archive**

Команда git archive используется для упаковки в архив указанных коммитов или всего репозитория.

### **git submodule**

Команда git submodule используется для управления вложенными репозиториями. Например, это могут быть библиотеки или другие, используемые не только в этом проекте ресурсы. У команды submodule есть несколько под-команд — add, update, sync и др. — для управления такими репозиториями.

# Шпаргалка по осмотру и сравнению

### **git show**

Команда git show отображает объект в простом и человекопонятном виде. Обычно она используется для просмотра информации о метке или коммите.

### **git shortlog**

Команда git shortlog служит для подведения итогов команды git log. Она принимает практически те же параметры, что и git log, но вместо простого листинга всех коммитов, они будут сгруппированы по автору.

### **git describe**

Команда git describe принимает на вход что угодно, что можно трактовать как коммит (ветку, тег) и выводит более-менее человекочитаемую строку, которая не изменится в будущем для данного коммита. Это может быть использовано как более удобная, но по-прежнему уникальная, замена SHA-1.

# Шпаргалка по отладке

В Git есть несколько команд, используемых для нахождения проблем в коде. Это команды для поиска места в истории, где проблема впервые проявилась и собственно виновника этой проблемы.

### **git bisect**

Команда git bisect — это чрезвычайно полезная утилита для поиска коммита в котором впервые проявился баг или проблема с помощью автоматического бинарного поиска.

### **git blame**

Команда git blame выводит перед каждой строкой файла SHA-1 коммита, последний раз менявшего эту строку и автора этого коммита. Это помогает в поисках человека, которому нужно задавать вопросы о проблемном куске кода.

### **git grep**

Команда git grep используется для поиска любой строки или регулярного выражения в любом из файлов вашего проекта, даже в более ранних его версиях.

Если вы только начинаете работать с Git, или переходите на Git с другой СКВ, то такая шпаргалка может вам очень пригодиться.

# Полезные ссылки

# 1. [Лучший курс по Git от Lynda.com. На русском](https://proglib.io/p/system-git)

# 2. [Про Git, Github и Gitflow простыми словами](https://proglib.io/p/git-github-gitflow)

# 3. <https://git-scm.com/videos>

# 4. [Начало работы с Git: подробный гайд для новичков](https://javarush.com/groups/posts/2683-nachalo-rabotih-s-git-podrobnihy-gayd-dlja-novichkov)

# 5. [Git для новичков](https://www.youtube.com/watch?v=f2TurkSBnYQ&list=PLY4rE9dstrJyTdVJpv7FibSaXB4BHPInb&index=4)

6. [Работа с git через консоль](https://htmlacademy.ru/blog/git/git-console)

7. [Git для новичков (часть 1)](https://habr.com/ru/articles/541258/), [Git для новичков (часть 2)](https://habr.com/ru/articles/542616/), [Тренажер](https://learngitbranching.js.org/?locale=ru_RU)

# 8. [Знакомство с Git и GitHub: руководство для начинающих](https://medium.com/nuances-of-programming/%D0%B7%D0%BD%D0%B0%D0%BA%D0%BE%D0%BC%D1%81%D1%82%D0%B2%D0%BE-%D1%81-git-%D0%B8-github-%D1%80%D1%83%D0%BA%D0%BE%D0%B2%D0%BE%D0%B4%D1%81%D1%82%D0%B2%D0%BE-%D0%B4%D0%BB%D1%8F-%D0%BD%D0%B0%D1%87%D0%B8%D0%BD%D0%B0%D1%8E%D1%89%D0%B8%D1%85-54ea2567d76c)

# 9. [Git: пошаговая инструкция](https://github.com/deeppavlov/dlschl/wiki/Git:-%D0%BF%D0%BE%D1%88%D0%B0%D0%B3%D0%BE%D0%B2%D0%B0%D1%8F-%D0%B8%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D0%B8%D1%8F), [Инструкция по работе с Git](https://github.com/deeppavlov/dlschl/wiki/%D0%98%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D0%B8%D1%8F-%D0%BF%D0%BE-%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%B5-%D1%81-Git)