# Основы

Git — это набор консольных утилит, которые отслеживают и фиксируют изменения в файлах (чаще всего речь идет об исходном коде программ, но вы можете использовать его для любых файлов на ваш вкус). С его помощью вы можете откатиться на более старую версию вашего проекта, сравнивать, анализировать, сливать изменения и многое другое. Этот процесс называется контролем версий. Существуют различные системы для контроля версий. Вы, возможно, о них слышали: SVN, Mercurial, Perforce, CVS, Bitkeeper и другие.

Git является распределенным, то есть не зависит от одного центрального сервера, на котором хранятся файлы. Вместо этого он работает полностью локально, сохраняя данные в папках на жестком диске, которые называются репозиторием. Тем не менее, вы можете хранить копию репозитория онлайн, это сильно облегчает работу над одним проектом для нескольких людей. Для этого используются сайты вроде github и bitbucket.

# 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.
- OS X — проще всего воспользоваться homebrew. После его установки запустите в терминале:

```
brew install git
```

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

# Настройка

Итак, мы установили git, теперь нужно добавить немного настроек. Есть довольно много опций, с которыми можно играть, но мы 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.

# Создание нового репозитория

Как мы отметили ранее, git хранит свои файлы и историю прямо в папке проекта. Чтобы создать новый репозиторий, нам нужно открыть терминал, зайти в папку нашего 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 наш 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.

# Коммит(фиксация изменений)

Коммит представляет собой состояние репозитория в определенный момент времени. Это похоже на снапшот, к которому мы можем вернуться и увидеть состояние объектов на определенный момент времени.
Чтобы зафиксировать изменения, нам нужно хотя бы одно изменение в области подготовки (мы только что создали его при помощи git add), после которого мы может коммитить:

```
$ git commit -m "Initial commit."
```

Эта команда создаст новый коммит со всеми изменениями из области подготовки (добавление файла hello.txt). Ключ -m и сообщение «Initial commit.» — это созданное пользователем описание всех изменений, включенных в коммит. Считается хорошей практикой делать коммиты часто и всегда писать содержательные комментарии.

# 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. Подключение к удаленному репозиторию

Чтобы загрузить что-нибудь в удаленный репозиторий, сначала нужно к нему подключиться. 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. 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.

# 2. 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
```

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

## 3. Клонирование репозитория

Сейчас другие пользователи GitHub могут просматривать ваш репозиторий. Они могут скачать из него данные и получить полностью работоспособную копию вашего проекта при помощи команды clone.

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

Новый локальный репозиторий создается автоматически с GitHub в качестве удаленного репозитория.

## 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.

# 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
```

## 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
```

# Дополнительно

В последней части этого руководства мы расскажем о некоторых дополнительных трюках, 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.