Learn Git in 30 Minutes

June 2nd 2016 **Quick Learn**

Git has exploded in popularity in recent years. The version control system is used by huge open source projects like Linux with thousands of contributors, teams of various sizes, solo developers and even students.

Beginners are often terrified by all the cryptic commands and arguments that git requires. But you don't need to know all of that to get started. You can begin by mastering a handful of the most often used ones, and then slowly build from there. And this is exactly what we will teach you today. Let's begin!

The basics

Git is a collection of command line utilities that track and record changes in files (most often source code, but you can track anything you wish). With it you can restore old versions of your project, compare, analyze, merge changes and more. This process is referred to as **version control**. There are a number of version control systems that do this job. You may have heard some of them - SVN, Mercurial, Perforce, CVS, Bitkeeper and more.

Git is decentralized, which means that it doesn't depend on a central server to keep old versions of your files. Instead it works fully locally by storing this data as a folder on your hard drive, which we call a **repository**. However you can store a copy of your repository online, which makes it easy for multiple people to collaborate and work on the same code. This is what websites like GitHub and BitBucket are used for.

1. Installing Git

Installing Git on your machine is straightforward:

- Linux - Simply open up a new terminal and install git via your distribution's package manager. For Ubuntu the command is: `sudo apt-get install git`
- Windows - we recommend git for windows as it offers both a GUI client and a BASH comand line emulator.
- OS X - The easiest way is to install homebrew, and then just run `brew install git` from your terminal.

If you are an absolute beginner, then a graphical git client is a must. We recommend GitHub Desktop and Sourcetree, but there are many other good and free ones online. Getting to know the basic git commands is still important even if you use a GUI app, so for the remaining of this lesson, this will be our only focus.

2. Configuring Git

Now that we've installed git on our computer, we will need to add some quick configurations. There are a lot of options that can be fiddled with, but we are going to set up the most important ones: our username and email. Open a terminal and run these commands:

```
$ git config --global user.name "My Name"
$ git config --global user.email myEmail@example.com
```

Every action we do in Git will now have a stamp with our name and address on it. This way users always know who did what and everything is way more organized.

3. Creating a new repository - `git init`

As we mentioned earlier, git stores its files and history directly as a folder in your project. To set up a new repository, we need to open a terminal, navigate to our project directory and run `git init`. This will enable Git for this particular folder and create a hidden *.git*directory where the repository history and configuration will be stored.

Create a folder on your Desktop called *git_exercise*, open a new terminal and enter the following:

```
$ cd Desktop/git_exercise/
$ git init
```

The command line should respond with something along the lines of:

```
Initialized empty Git repository in /home/user/Desktop/git_exercise/.git/
```

This means that our repo has been successfully created but is still empty. Now create a simple text file called *hello.txt* and save it in the *git_exercise* folder.

4. Checking the status - `git status`

Git status is another must-know command that returns information about the current state of the repository: is everything up to date, what's new, what's changed, and so on. Running `git status` in our newly created repo should return the following:

```
$ git status

On branch master

Initial commit

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

    hello.txt
```

The returned message states that *hello.txt* is untracked. This means that the file is new and Git doesn't know yet if it should keep track of the changes happening to that file or just ignore it. To acknowledge the new file, we need to stage it.

5. Staging - `git add`

Git has the concept of a *"staging area"*. You can think of this like a blank canvas, which holds the changes which you would like to commit. It starts out empty, but you can add files to it (or even single lines and parts of files) with the `git add` command, and finally commit everything (create a snapshot) with `git commit`.

In our case we have only one file so let's add that:

```
$ git add hello.txt
```

If we want to add everything in the directory, we can use:

```
$ git add -A
```

Checking the status again should return a different response from before.

```
$ git status

On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached ..." to unstage)

    new file:   hello.txt
```

Our file is ready to be commited. The status message also tells us what has changed about the files in the staging area - in this case its *new file*, but it can be *modified* or *deleted*, depending on what has happened to a file since the last `git add`.

6. Commiting - `git commit`

A commit represents the state of our repository at a given point in time. It's like a snapshot, which we can go back to and see how thing were when we took it.

To create a new commit we need to have at least one change added to the staging area (we just did that with `git add`) and run the following:

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

This will create a new commit with all the changes from the staging area (adding hello.txt). The `-m "Initial commmit"` part is a custom user-written description that summarizes the changes done in that commit. It is considered a good practice to commit often and always write meaningful commit messages.

Remote repositories

Right now our commit is local - it exist only in the *.git* folder. Although a local repository is useful by itself, in most cases we will want to share our work and deploy it to a server or a repository hosting service.

1. Connecting to a remote repository - `git remote add`

In order to upload something to a remote repo, we first have to establish a connection with it. For the sake of this tutorial our repository's address will be *https://github.com/tutorialzine/awesome-project*. We advise you to go ahead and create your own empty repository at GitHub, BitBucket or any other service. The registration and setup may take a while, but all services offer good step-by-step guides to help you.

To link our local repository with the one on GitHub, we execute the following line in the terminal:

```
# This is only an example. Replace the URI with your own repository address.
$ git remote add origin https://github.com/tutorialzine/awesome-project.git
```

A project may have many remote repositories at the same time. To be able to tell them apart we give them different names. Traditionally the main remote repository in git is called *origin*.

2. Uploading to a server - `git push`

Now it's time to transfer our local commits to the server. This process is called a **push**, and is done every time we want to update the remote repository.

The Git command to do this is `git push` and takes two parameters - the name of the remote repo (we called ours *origin*) and the branch to push to (*master* is the default branch for every repo).

```
$ 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
```

Depending on the service you're using, you will need to authenticate yourself for the push to go through. If everything was done correctly, when you go in your web browser to the remote repository created earlier, *hello.txt* should be available there.

3. Cloning a repository - `git clone`

At this point, people can see and browse through your remote repository on Github. They can download it locally and have a fully working copy of your project with the `git clone` command:

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

A new local respository is automatically created, with the github version configured as a remote.

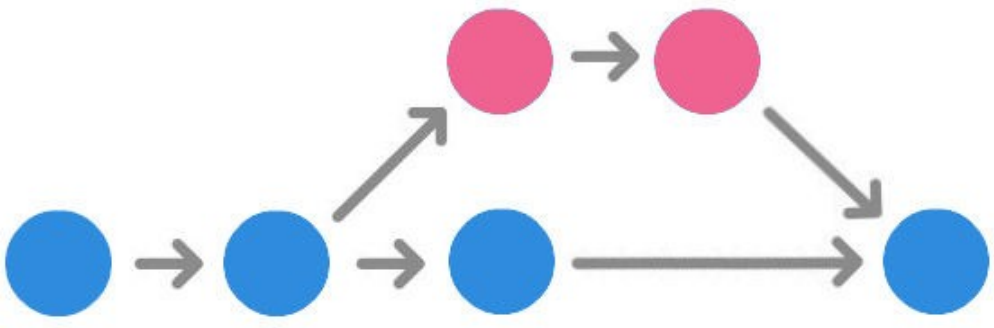
4. Getting changes from a server - `git pull`

If you make updates to your repository, people can download your changes with a single command - **pull**:

```
$ git pull origin master

From https://github.com/tutorialzine/awesome-project
 * branch             master     -> FETCH_HEAD
Already up-to-date.
```

Since nobody else has commited since we cloned, there weren't any changes to download.



Branches

When developing a new feature, it is considered a good practice to work on a copy of the original project, called a *branch*. Branches have their own history and isolate their changes from one another, until you decide to merge them back together. This is done for a couple of reasons:

- An already working, stable version of the code won't be broken.
- Many features can be safely developed at once by different people.
- Developers can work on their own branch, without the risk of their codebase changing due to someone else's work.
- When unsure what's best, multiple versions of the same feature can be developed on separate branches and then compared.

1. Creating new branches - `git branch`

The default branch of every repository is called ***master***. To create additional branches use the `git branch <name>` command:

```
$ git branch amazing_new_feature
```

This just creates the new branch, which at this point is exactly the same as our *master*.

2. Switching branches - `git checkout`

Now, when we run `git branch`, we will see there are two options available:

```
$ git branch
  amazing_new_feature
* master
```

Master is the current branch and is marked with an asterisk. However, we want to work on our new amazing features, so we need to switch to the other branch. This is done with the `git checkout` command, expecting one parameter - the branch to switch to.

```
$ git checkout amazing_new_feature
```

3. Merging branches - `git merge`

Our "amazing new feature" is going to be just another text file called *feature.txt*. We will create it, add it, and commit it.

```
$ git add feature.txt
$ git commit -m "New feature complete."
```

The new feature is complete, we can go back to the master branch.

```
$ git checkout master
```

Now, if we open our project in the file browser, we'll notice that *feature.txt* has disappeared. That's because we are back in the master branch, and here *feature.txt* was never created. To bring it in, we need to `git merge` the two branches together, applying the changes done in *amazing_new_feature* to the main version of the project.

```
git merge amazing_new_feature
```

The master branch is now up to date. The awesome_new_feature branch is no longer needed and can be removed.

```
git branch -d amazing_new_feature
```

Advanced

In the last section of this tutorial, we are going to take a look at some more advanced techniques that are very likely to come in handy.

1. Checking difference between commits

Every commit has it's unique id in the form of a string of numbers and symbols. To see a list of all commits and their ids we can use `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
```

As you can see the ids are really long, but when working with them it's not necessary to copy the whole thing - the first several symbols are usually enough.
To see what was new in a commit we can run `git show [commit]` :

```
$ 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?
```

To see the difference between any two commits we can use `git diff` with the [commit-from]..[commit-to] syntax:

```
$ 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?
```

We've compared the first commit to the last one, so we see all the changes that have ever been made. Usually it's easier to do this using the `git difftool` command which brings up a graphical client showing all differences side-to-side.

2.Reverting a file to a previous version

Git allows us to return any selected file back to the way it was in a certain commit. This is done via the familiar `git checkout` command, which we used earlier to switch branches, but can also be used to swtich between commits (it's pretty common in Git for one command to be used for multiple seemingly unrelated tasks).

In the following example we will take *hello.txt* and reverse everything we've done to it since the initial commit. To do so we have to supply the id of the commit we want to go back to, as well as the full path to our file.

```
$ git checkout 09bd8cc1 hello.txt
```

3. Fixing a commit

If you notice that you've made a typo in your commit message, or you've forgotten to add a file and you see right after you commit, you can easily fix this with `git commit --amend`. This will add everything from the last commit back to the staging area, and attempt to make a new commit. This gives you a chance to fix your commit message or add more files to the staging area.

For more complex fixes that aren't in the last commit (or if you've pushed your changes already), you've got to use `git revert`. This will take all the changes that a commit has introduced, reverse them, and create a new commit that is the exact opposite.

The newest commit can be accessed by the HEAD alias.

```
$ git revert HEAD
```

For other commits it's best to use an id.

```
$ git revert b10cc123
```

When reverting older commits, keep in mind that merge conflicts are very likely to appear. This happens when a file has been altered by another more recent commit, and now Git cannot find the right lines to revert, since they aren't there anymore.

4. Resolving Merge Conflicts

Apart from the scenario depicted in the previous point, conflicts regularly appear when merging branches or pulling someone else's work. Sometimes conflicts are handled automatically by git, but other times the person dealing with them has to decide (and usually handpick) what code stays and what is removed.

Let's look at an example where we're trying to merge two branches called john_branch and tim_branch. Both John and Tim are writing in the same file a function that displays all the elements in an array.

John is using a for loop:

```
// Use a for loop to console.log contents.
for(var i=0; i<arr.length; i++) {
    console.log(arr[i]);
}
```

Tim prefers forEach:

```
// Use forEach to console.log contents.
arr.forEach(function(item) {
    console.log(item);
});
```

They both commit their code on their respective branch. Now if they try to merge the two branches they will see the following error message:

```
$ 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.
```

Git wasn't able to merge the branches automatically, so now it's up to the devs to manually resolve the conflict. If they open the file where the conflict resides, they'll see that Git has inserted a marker on the conflicting lines.

```
<<<<<<< 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.
```

Above the ===== we have the current HEAD commit, and below the conflicting one. This way we can clearly see the differences, and decide which is the better version, or write a new one all together. In this situation we go for the latter and rewrite the whole thing, removing the markers to let Git know we're done.

```
// Not using for loop or forEach.
// Use Array.toString() to console.log contents.
console.log(arr.toString());
```

When everything is set, a merge commit has to be done to finish the process.

```
$ git add -A
$ git commit -m "Array printing conflict resolved."
```

As you can see this process is quite tiresome and can get extremely hard to deal with in large projects. Most developers prefer to resolve conflicts with the help of a GUI client, which makes things much easier. To run a graphical client use `git mergetool`.

5. Setting up .gitignore

In most projects there are files or entire folders that we don't want to ever commit. We can make sure that they aren't accidentally included in our `git add -A` by creating a *.gitignore* file:

1. Manually create a text file called .gitignore and save it in your project's directory.
2. Inside, list the names of files/directories to be ignored, each on a new line.
3. The .gitignore itself has to be added, committed and pushed, as it is just like any other file in the project.

Good examples for files to be ignored are:

- log files
- task runner builds
- the node_modules folder in node.js projects
- folders created by IDEs like Netbeans and IntelliJ
- personal developer notes

A .gitignore banning all of the above will look like this:

```
*.log
build/
node_modules/
.idea/
my_notes.txt
```

The slash at the end of some of the lines signals that this is a folder and we are ignoring everything inside it recursively. The asterisk serves it's usual function as a wild card.

Conclusion

This ends our tutorial on git! We tried our best to bring you only the most important information you need, presented as short and concise as possible.

Git is quite complex and has a lot more features and tricks to offer. If you wish to find out more, here are some learning resources we recommend:

- The official Git docs, including a whole book and video lessons - here.
- Getting git right - Atlassian's collection of tutorials and articles - here.
- A list of GUI clients - here.
- Git cheat sheet (PDF) - here.
- Online tool for generating .gitignore files - here.