Git

Manual

José Costa

GitHub profile:
https://github.com/Ze1598

1. Introduction

This Manual was created with the objective of providing quick and easy access to the most common Git commands and operations.

It starts with the basics of what is Git and how start a project (a repository) with this tool, and expands other topics such as backtracking, creating and utilizing branches and, last but not least, how to work cooperatively in the same project.

Regarding the content presented throughout the document, it was heavily based on what I learnt in Codeacademy's "Learn Git" course.

2. Table of Contents

3. Figure's Index

4. Hello Git

Git is a software that allows you to keep track of changes made to a project over time. Git works by recording the changes you make to a project, storing those changes, then allowing you to reference them as needed.

4.1. git init

Now that we have started working on the screenplay, let's turn the current working directory into a Git project. We do this with:

```
git init
```

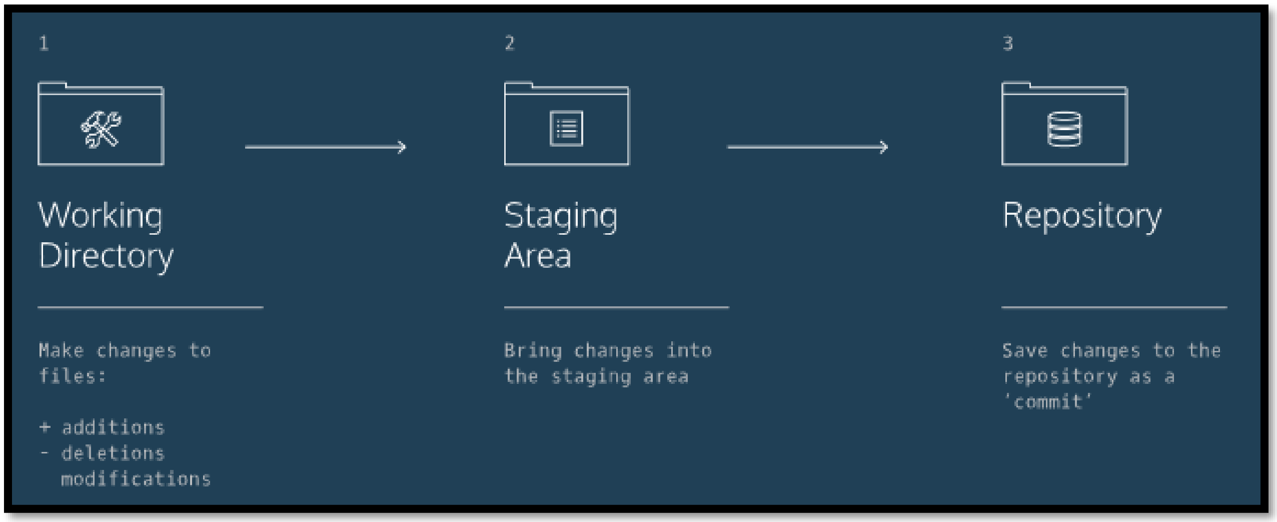
The word `init` means initialize. The command sets up all the tools Git needs to begin tracking changes made to the project.

4.2. Git Workflow

We have a started the Git project. A Git project can be thought of as having three parts:

- A Working Directory: where you'll be doing all the work: creating, editing, deleting and organizing files
- A Staging Area: where you'll list changes you make to the working directory
- A Repository: where Git permanently stores those changes as different versions of the project

The Git workflow consists of editing files in the working directory, adding files to the staging area, and saving changes to a Git repository. In Git, we save changes with a commit, which will be explained further ahead.



[1]

Figure 1: Git workflow

4.3. git status

As you develop the files for the directory, you will be changing the contents of the working directory. You can check the status of those changes with:

```
git status
```

In the output, notice the file in red under untracked files. Untracked means that Git sees the file but has not started tracking changes yet.

4.4. git add

In order for Git to start tracking untracked files, the files need to be added to the staging area.

We can add a file to the staging area with:

```
git add file_name
```

Where `file_name` is the name of the file to be added.

4.5. git diff

Good work! Now you know how to add a file to the staging area.

[1] Source: https://www.codecademy.com/learn/learn-git

Imagine that we type another line in the file we just added. Since the file is tracked, we can check the differences between the working directory and the staging area with:

```
git diff file_name
```

Once again, *`file_name`* is the actual name of the file.

Notice the output:

- Changes to the file are marked with a `+` and are indicated in green.
- IMPORTANT: press `q` on your keyboard to exit `diff` mode.

To update the already existing file in the staging area, simply use once again the `git add` *`file_name`* command.

4.6. git commit

A commit is the last step in our Git workflow. A commit permanently stores changes from the staging area inside the repository.

`git commit` is the command we'll do next. However, one more bit of code is needed for a commit: the option `-m` followed by a message. Here's an example:

```
git commit -m "First commit"
```

Standard Conventions for Commit Messages:

- Must be in quotation marks
- Written in the present tense
- Should be brief (50 characters or less) when using `-m`

4.7. git log

Often with Git, you'll need to refer back to an earlier version of a project. Commits are stored chronologically in the repository and can be viewed with `git log`.

In the output, notice:

- A 40-character code, called a SHA, that uniquely identifies the commit. This part is written in a different color (in Git Bash it uses a green-yellow-ish color).

- The commit author
- The date and time of the commit
- The commit message

4.8. Generalizations

You have now been introduced to the fundamental Git workflow. You learned a lot! Let's take a moment to generalize:

Git is the industry-standard version control system for web developers

Use Git commands to help keep track of changes made to a project:

- `git init` creates a new Git repository
- `git status` inspects the contents of the working directory and staging area
- `git add` adds files from the working directory to the staging area
- `git diff` shows the difference between the working directory and the staging area
- `git commit` permanently stores file changes from the staging area in the repository
- `git log` shows a list of all previous commits

5. Backtracking Intro

When working on a Git project, sometimes we make changes that we want to get rid of. Git offers a few eraser-like features that allow us to undo mistakes during project creation

5.1. head commit

In Git, the commit you are currently on is known as the `HEAD` commit. In many cases, the most recently made commit is the `HEAD` commit.

To see the `HEAD` commit, enter:

```
git show HEAD
```

The output of this command will display everything the `git log` command displays for the `HEAD` commit, plus all the file changes that were committed.

5.2. git checkout

What if you decide to change a line in a file, but then decide you wanted to discard that change?

You could rewrite the line how it was originally, but what if you forgot the exact wording? The command `git checkout HEAD file_name` will restore the file in your working directory to look exactly as it did when you last made a commit.

Here, `file_name` again is the actual name of the file. If the file is named *changes.txt*, the command would be `git checkout HEAD changes.txt`.

5.3. more git add

Let’s say the repository we are working on contains five files. In Git, it's common to change many files, add those files to the staging area, and commit them to a repository in a single commit.

For example, you modify two files. After this, you can add the changed files to the staging area with:

```
git add filename_1 filename_2
```

5.4. git reset

Great! The files you've added to the staging area belong in the same commit.

What if, before you commit, you accidentally delete an important line from a file *file1.txt*? Unthinkingly, you add *file1.txt* to the staging area. Now you don't want to include it in the commit.

We can unstage that file from the staging area using

```
git reset HEAD file1.txt
```

This command resets the file in the staging area to be the same as the `HEAD` commit. It does not discard file changes from the working directory, it just removes them from the staging area.

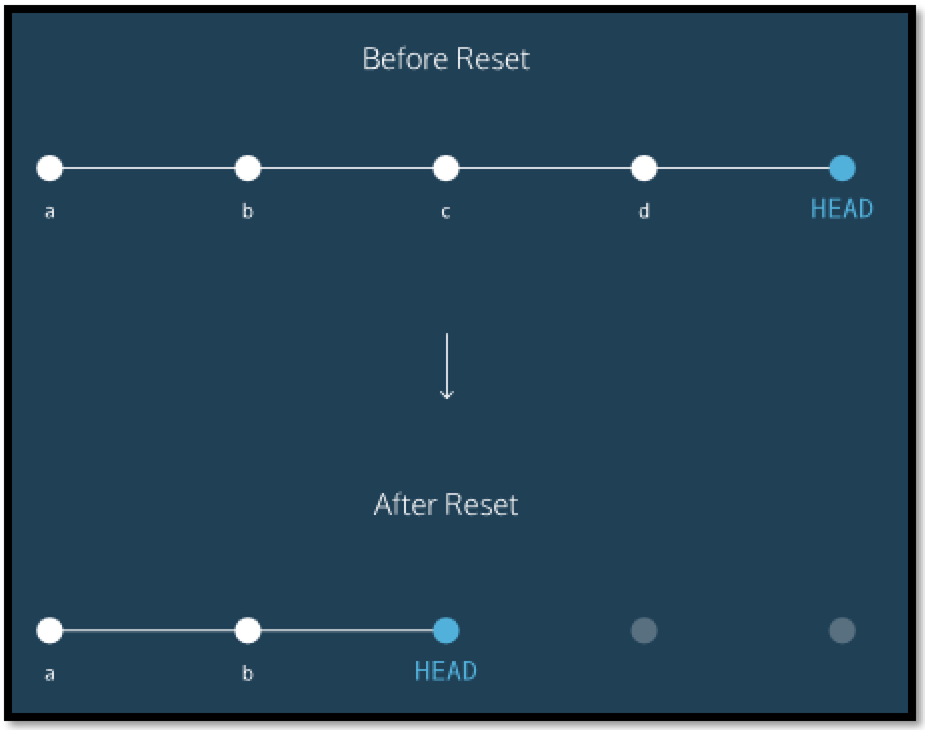
However, what about cases when you simply want to go back to the state of a previous commit? You can do this with: `git reset commit_SHA`.

This command works by using the first 7 characters of the SHA of a previous commit. For example, if the SHA of the previous commit is *5d692065cf51a2f50ea8e7b19b5a7ae512f633ba*, use: `git reset 5d69206`.

`HEAD` is now set to that previous commit.

To better understand `git reset commit_SHA`, look at the diagram below. Each circle represents a commit.

(Continued in the next page)



[2]

Figure 2: git reset graphical representation

Before reset:

- `HEAD` is at the most recent commit

After resetting:

- `HEAD` goes to a previously made commit of your choice
- The gray commits are no longer part of your project
- You have, in essence, rewound the project's history

5.5. Generalizations

Congratulations! You've learned three different ways to backtrack in Git. You can use these skills to undo changes made to your Git project.

Let's take a moment to review the new commands:

- `git checkout HEAD` *`filename`*: Discards changes in the working directory.

[2] Source: https://www.codecademy.com/learn/learn-git

- `git reset HEAD filename`: Unstages file changes in the staging area.
- `git reset commit_SHA`: Resets to a previous commit in your commit history.

Additionally, you learned a way to add multiple files to the staging area with a single command: `git add filename_1 filename_2`.

6. git branch

Up to this point, you've worked in a single Git branch called `master`. Git allows us to create branches to experiment with versions of a project.

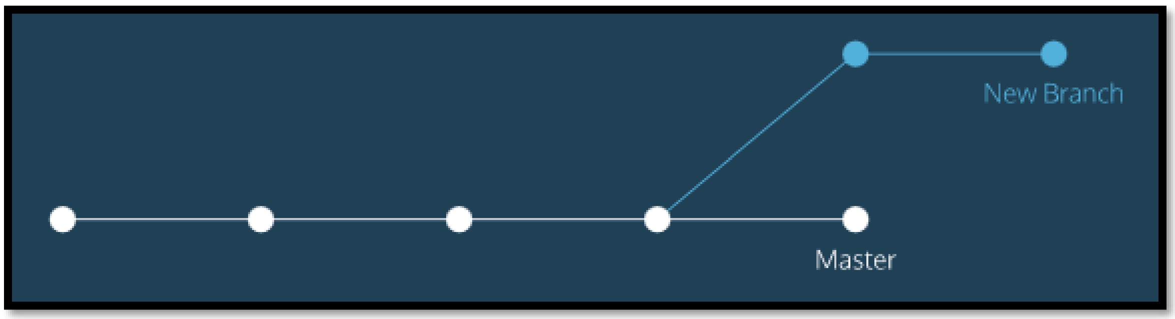
Imagine you want to create a version of a story with a happy ending. You can create a new branch and make the happy ending changes to that branch only. It will have no effect on the master branch until you're ready to merge the happy ending to the master branch.

In this chapter, we'll be using Git branching to develop multiple versions of a resumé.

You can use the command `git branch` to check which branch you're currently working on.

6.1. branching overview

The diagram below illustrates branching.



[3]

Figure 3: Branching graphical representation

The circles are commits and together form the Git project's commit history.

New Branch is a different version of the Git project. It contains commits from Master but also has commits that Master does not have.

6.2. git branch

Right now, the Git project has only one branch: `master`.

To create a new branch, use `git branch *new_branch*`.

[3] Source: https://www.codecademy.com/learn/learn-git

`new_branch` would be the name of the new branch you create, like photos or blurb. Be sure to name your branch something that describes the purpose of the branch. Also, branch names can't contain whitespaces: `new-branch` and `new_branch` are valid, but `new branch` is not.

6.3. git checkout

When a new branch is first created, the both `master` and that branch are identical: they share the same exact commit history. You can switch to the new branch with `git checkout branch_name`.

Here, `branch_name` is the name of the branch. If the branch's name is `skill`, then use `git checkout skill`.

Once you switch branch, you're able to make commits on that branch, having no impact on `master`. You can continue your workflow, while `master` stays intact!

6.4. Commit on a New Branch

All the commands you do on `master` also work on the other branches you create.

For example, to add files to the staging area, use `git add filename`.

And to commit, use `git commit -m "Commit message"`.

6.5. git merge

What if you wanted to include all the changes made to the created branch on the master branch? We can easily accomplish this by merging the branch into master with `git merge branch_name`.

For example, if I wanted to merge the `skills` branch to `master`, I would enter `git merge skills`.

In a moment, you'll merge branches. Keep in mind:

- Your goal is to update `master` with changes you made to `skills`.
- `skills` is the giver branch, since it provides the changes.
- `master` is the receiver branch, since it accepts those changes.

Notice the output: The merge is a "fast forward" because Git recognizes that `skills` contains the most recent commit. Git fast forwards `master` to be up to date with `skills`.

6.6. Merge Conflict

The merge was successful because `master` had not changed since we made a commit on `skills`. Git knew to simply update `master` with changes on `skills`.

What would happen if you made a commit on `master` before you merged the two branches? Furthermore, what if the commit you made on `master` altered the same exact text you worked on in `skills`? When you switch back to `master` and ask Git to merge the two branches, Git doesn't know which changes you want to keep. This is called a merge conflict.

Let's say you decide you'd like to merge the changes from `skills` into `master`.

Here's where the trouble begins!

From the terminal, enter the command `git merge skills`. This will try to merge `skills` into `master`.

In the output, notice the lines:

```
CONFLICT (content): Merge conflict in resumé.txt

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

We must fix the merge conflict.

In the code editor, look at the file in question. Git uses markings to indicate the `HEAD` (`master`) version of the file and the `skills` version of the file, like this:

```
<<<<<<< HEAD

master version of line

=======

skills version of line

>>>>>>> skills
```

Git asks us which version of the file to keep: the version on `master` or the version on `skills`. You decide you want the `skills` version.

From the code editor:

- Delete the content of the line as it appears in the `master` branch
- Delete all of Git's special markings including the words `HEAD` and `skills`. If any of Git's markings remain, for example, `>>>>>>>` and `=======`, the conflict remains.

Then, from the terminal add the file to the staging area once more and commit.

6.7. delete branch

In Git, branches are usually a means to an end. You create them to work on a new project feature, but the end goal is to merge that feature into the master branch. After the branch has been integrated into master, it has served its purpose and can be deleted.

The command `git branch -d` ***`branch_name`*** will delete ***`branch_name`*** from your Git project.

6.8. Generalizations

Let's take a moment to review the main concepts and commands from the lesson before moving on.

Git branching allows users to experiment with different versions of a project by checking out separate branches to work on.

The following commands are useful in the Git branch workflow.

- `git branch`: Lists all of a Git project's branches.
- `git branch` ***`branch_name`***: Creates a new branch.
- `git checkout` ***`branch_name`***: Used to switch from one branch to another.
- `git merge` ***`branch_name`***: Used to join file changes from one branch to another.
- `git branch -d` ***`branch_name`***: Deletes the branch specified.

7. Cooperative Work

So far, we've learned how to work on Git as a single user. However, Git offers a suite of collaboration tools to make working with others on a project easier.

Imagine that you're a science teacher, developing some quizzes with Sally, another teacher in the school. You are using Git to manage the project.

In order to collaborate, you and Sally need:

- A complete replica of the project on your own computers
- A way to keep track of and review each other's work
- Access to a definitive project version

You can accomplish all of this by using remotes. A `remote` is a shared Git repository that allows multiple collaborators to work on the same Git project from different locations. Collaborators work on the project independently, and merge changes together when they are ready to do so.

7.1. git clone

Sally has created the remote repository, `science-quizzes`, in the directory `curriculum`, which teachers on the school's shared network have access to. In order to get your own replica of `science-quizzes`, you'll need to clone it with: `git clone` *`remote_location clone_name`*.

In this command:

- *`remote_location`* tells Git where to go to find the remote. This could be a web address or a filepath, such as: *`/Users/teachers/Documents/some-remote`*
- *`clone_name`* is the name you give to the directory in which Git will clone the repository.

7.2. git remote -v

Nice work! We have a clone of Sally's `remote` on our computer. One thing that Git does behind the scenes when you clone `science-quizzes` is give the remote address the name `origin`, so that you can refer to it more conveniently. In this case, Sally's remote is `origin`.

You can see a list of a Git project's remotes with the command `git remote -v`.

- Git lists the name of the remote, `origin`, as well as its location.
- Git automatically names this remote `origin`, because it refers to the remote repository of origin. However, it is possible to safely change its name.
- The remote is listed twice: once for (`fetch`) and once for (`push`).

7.3. git fetch

After you cloned `science-quizzes`, you had to run off to teach a class. Now that you're back at your computer, there's a problem: what if, while you were teaching, Sally changed the `science-quizzes` Git project in some way. If so, your clone will no longer be up-to-date.

An easy way to see if changes have been made to the remote and bring the changes down to your local copy is with `git fetch`.

This command will not merge changes from the `remote` into your local repository. It brings those changes onto what's called a remote branch.

7.4. git merge

Even though Sally's new commits have been fetched to your local copy of the Git project, those commits are on the `origin/master` branch. Your local `master` branch has not been updated yet, so you can't view or make changes to any of the work she has added.

We'll use the `git merge` command to integrate `origin/master` into your local `master` branch. The command: `git merge origin/master` will accomplish this for us.

7.5. Git workflow

Now that you've merged `origin/master` into your local `master` branch, you're ready to contribute some work of your own. The workflow for Git collaborations typically follows this order:

1. Fetch and merge changes from the `remote`
2. Create a branch to work on a new project feature
3. Develop the feature on your branch and commit your work

4. Fetch and merge from the `remote` again (in case new commits were made while you were working)
5. Push your branch up to the `remote` for review

Steps 1 and 4 are a safeguard against `merge` conflicts, which occur when two branches contain file changes that cannot be merged with the `git merge` command. Step 5 involves `git push`.

7.6. git push

Now it's time to share our work with Sally.

The command `git push origin` *`your_branch_name`* will push your branch up to the remote, `origin`. From there, Sally can review your branch and merge your work into the `master` branch, making it part of the definitive project version.

7.7. generalizations

Let's review:

- A remote is a Git repository that lives outside your Git project folder. Remotes can live on the web, on a shared network or even in a separate folder on your local computer.
- The Git Collaborative Workflow are steps that enable smooth project development when multiple collaborators are working on the same Git project.

We also learned the following commands

- `git clone`: Creates a local copy of a remote.
- `git remote -v`: Lists a Git project's remotes.
- `git fetch`: Fetches work from the remote into the local copy.
- `git merge origin/master`: Merges origin/master into your local branch.
- `git push origin` *`branch_name`*: Pushes a local branch to the `origin` remote.

Git projects are usually managed on Github, a website that hosts Git projects for millions of users. With Github you can access your projects from anywhere in the world by using the basic workflow you learned here.

8. Finishing Notes

This Manual was only created after completing Codeacademy’s course on the matter. Thus, in order to create this document, I had to first learn various things regarding Git, from the simple `git init` command used to create a new repository, to more advanced topics such as working cooperatively in the same project.

This means that, after reaching this paragraph hopefully you’ll have acquired important knowledge about this precious tool for programmers and even for people who don’t code at all.