Git Commands

Git Commits

A commit to a repository is a snapshot of the current state of the project's root directory. Since this explanation doesn't really tell anything, we need to delineate the underlying concept.

Let's say you're working with a bunch of papers. You've written ten texts about animals on separate sheets of paper and you want to note what texts they are and when you wrote them. You take out another sheet of paper, call it a "commit," and write on this commit paper: "I've written text #1. It's about birds. I've written text #2. It's about dogs..." Then you create a copy of each text. The last thing you do is you gather those ten copies, pin the commit paper on top of them, and lay them in a drawer.

The next day you rewrite the original texts, then get the copies from your drawer and compare the texts. This is basically what Git does. You create files and write code in them. When you're ready, you *commit* your files to a repository: you create copies of files and lay them in a drawer (a repository). (Our inner nerd wants to specify that Git doesn't actually push copies of files to the repository; Git creates a light representation of the project files for performance benefits.)

Each day you write that commit message and add new texts. Git creates a history of your commits, so you can trace back to the very beginning of the project development to see what files have been changed or added, who added or changed them and when.

Tracking Files with Git

Now we can answer the question, "Why does Git need to track files?" Before we commit any files to a local repository, Git wants to know what those files are. Git only knows what to commit when it's tracking files.

We've explained three basic Git concepts you need to know, but we've also moved far away from explaining Git commands. Nevertheless, it's crucial to grasp Git's basic concepts to understand how Git commands work. Now that we've explained the meaning of Git concepts, we can get back to the commands.

Let's remind you what output you'll see after you run "git status" for the first time:

```
$ git status
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

Since there are no files in the root directory yet, Git shows that there's nothing to commit, If you have any changes "git status" will show. Our safe deposit box (repository) is empty. To do anything further, we need to populate the root folder with at least one file. or example we added example_file.txt to the root directory. Now we can move on to the next step.

When you run "git status" once more (assuming you've added a file to the project's root directory), you'll get a different output:

```
$ git status
On branch master
Initial commit
Untracked files:(use "git add <file>..." to include in what will be
committed)
      example_file.txt
nothing added to commit but untracked files present (use "git add" to
track)
```

Note the "Untracked files" message with the file "example_file.txt". Git conveniently informs us that we've added a new file to the project. But that isn't enough for Git. As Git tells us, we need to track "example_file.txt". In other words, we need to add "example_file.txt" to the staging area. The

following section will uncover the basic Git commands for working with the staging area.

Here are the basic Git commands you've learned so far:

Staging Files with Git

Before we cover simple Git commands used for staging files, we need to explain what the staging area is.

Let's say you want to move some of your valuable effects to a lock box, but you don't know yet what things you'll put there. For now, you just gather things into a basket. You can take things out of the basket if you decide that they aren't valuable enough to store in a lock box, and you can add things to the basket as you wish. With Git, this basket is the *staging area*. When you move files to the staging area in Git, you actually *gather and prepare* files for Git before committing them to the local repository.

To let Git track files for a commit, we need to run the following in the terminal:

```
$ git add example_file.txt
```

That's it; you've added a file to the staging area with the "add" command. Don't forget to pass a filename to this command so Git knows which file to track.

But what has this "add" command actually done? Let's view an updated status (we promised that you'll often run "git status", didn't we?):

```
$ git status
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
   new file: example_file.txt
```

The status has changed! Git knows that there's a newly created file in your basket (the staging area), and is ready to commit the file. We'll get to committing files in the next section. For now, we want to talk more about the "git add" command.

What if you create or change several files? With a basket as your staging area, you have to put things into the basket one by one. Committing files to the repository individually isn't convenient. What Git can do is provide alternatives to the "**git add <file-name>**" command.

Let's assume you've added another three files to the root directory: my-file.ts, another-file.js, and new_file.rb. Now you want to add all of them to the staging area. Instead of adding these files separately, we can add them all together:

```
$ git add my-file.ts another-file.js new_file.rb
```

All you need to do is type file names separated by spaces following the "add" command. When you run "git status" once more to see what has changed, Git will output a new message listing all the files you've added:

```
$ git status
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
   new file: another_file.js
   new file: my_file.ts
   new file: my_new_file.txt
   new file: new_file.rb
```

Adding several files to the staging area in one go is much more convenient! But hold on a second. What if the project grows enormously and you have to add more than three files? How can we add a dozen files (or dozens of files) in one go? Git accepts the challenge and offers the following solution:

```
$ git add .
```

Remember when we told you that you can take things out of your imaginary basket? Git can also take things out of its basket by removing files from the staging area. To remove files from the staging area, use the following command:

```
$ git rm --cached my-file.ts
```

In our example, we specified the command "rm", which stands for remove. The "--cached" option indicates files in the staging area. Finally, we pass a file that we want to unstage. Git will output the following message for us:

```
$ rm 'my_file.ts'
```

Let's run "git status" again:

```
$ git status
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
   new file: another-file.js
   new file: my_new_file.txt
   new file: new_file.rb
```

```
Untracked files:
(use "git add <file>..." to include in what will be committed)
   my-file.ts
```

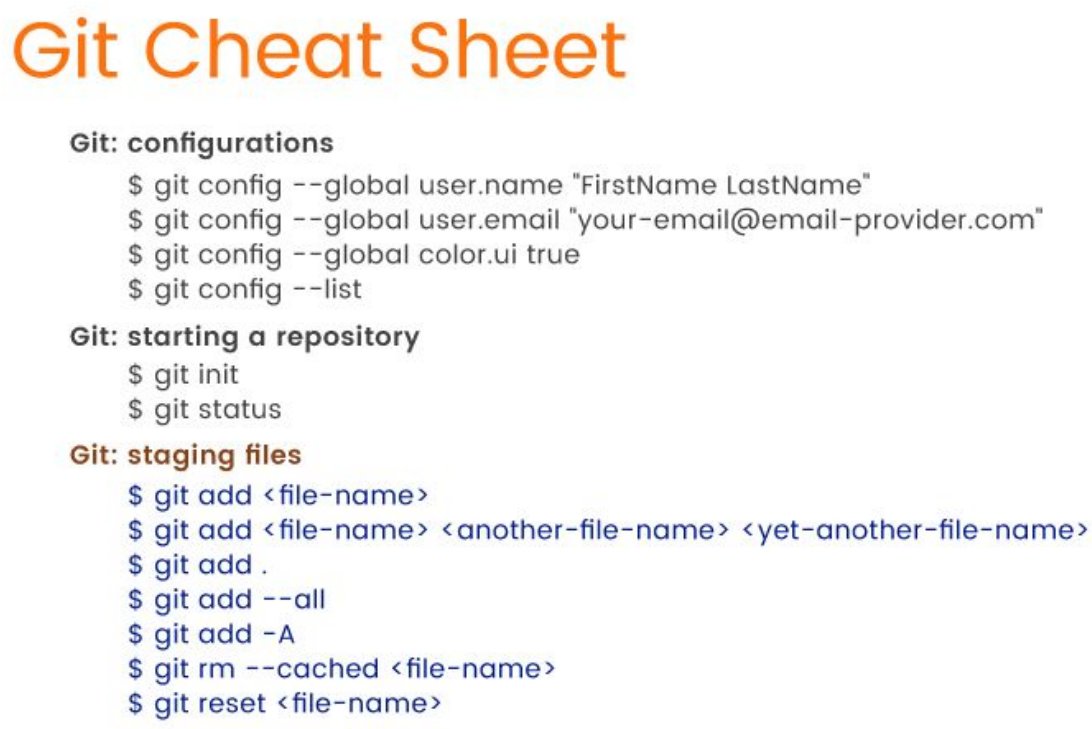
Git is no longer tracking my-file.ts. In this simple way, you can untrack files if necessary. As an alternative to "rm --cached <filename>", you can use the "reset" command:

```
$ git reset another-file.js
```

You can consider "reset" as the opposite of "add".

We've provided enough Git commands to add and remove files to and from the staging area. Now it's time to get familiar with committing files to the local repository.

Here are the most common Git commands you've learned so far:



Committing Changes to Git

Let's start with a quick overview of committing to the Git repository. By now, you should have at least one file tracked by Git (we have three). As we mentioned, tracked files aren't located in the repository yet. We have to commit them: we need to carry our basket with stuff to the lock box. There are several useful Git commands to do (almost) the same: move (commit) files from the staging area (an imaginary basket) to the repository (a lock box).

There's nothing difficult about committing to a repository. Just run the following command:

```
$ git commit -m "Add three files"
```

To commit to a repository, use the "commit" command. Next, pass the "commit" command the "-m" option, which stands for "message". Lastly, type in your commit message. We wrote "Add three files" for our example, but it's recommended that you write more meaningful messages like "Add admin panel" or "Update admin panel". Note that we didn't use the past tense! A commit message must tell what your commit *does* – adds or removes files, updates app features, and so on.

Once we've run "git commit -m 'Add three files'", we get the following output:

```
[master (root-commit) abfbdeb] Add three files
3 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 another_file.js
create mode 100644 my_new_file.txt
create mode 100644 new_file.rb
```

The message tells us that there have been three files added to the current branch, which in our example is the master or the main branch. The "create

mode 100644" message tells us that these files are regular non-executable files. The "0 insertions(+)" and "0 deletions(-)" messages mean we haven't added any new code or removed any code from the files. We actually don't need this information; it only confirms that the commit was successful.

So, what have we done so far? We added files to a project directory in the first section. Then we added files to the staging area, and now we've committed them. The basic Git flow looks like this:

- Create a new file in the root directory or in a subdirectory, or update an existing file.
- Add files to the staging area by using the "git add" command and passing necessary options.
- Commit files to the local repository using the "git commit -m <message>" command.
- Repeat.

That's enough to get the idea of Git's flow. But let's get back to committing files. We should mention a great alternative to the standard "git commit -m 'Does something'" command.

When working on a project, chances are you'll modify some files and commit them many times. Git's flow doesn't really change for adding modified files to a new commit. In other words, every time you make changes you'll need to add a modified file to the staging area and then commit. But this standard flow is tedious. And why should you have to ask Git to track a file that was tracked before?

The question is how can we add modified files to the staging area and commit them at the same time. Git provides the following super command:

```
$ git commit -a -m "Do something once more"
```

Note the "-a" option, which stands for "add". Git will react to this command like this: "I'll just commit the files immediately. Don't forget to write a

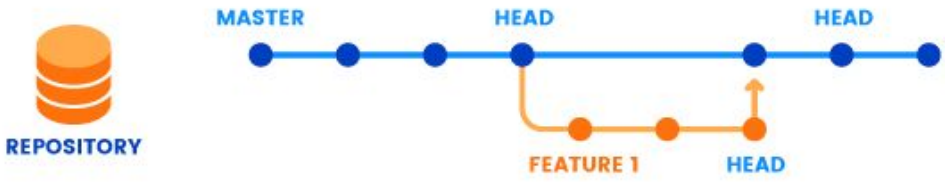
commit message, though!" As we can see, this little trick lets us avoid running two commands.

There will be times when you'll regret committing to a repository. Let's say you've modified ten files, but committed only nine. How can you add that remaining file to the last commit? And how can you modify a file if you've already committed it? There are two ways out. First, you can undo the commit:

```
$ git reset --soft HEAD^
```

As you may recall, the "reset" command is the opposite of the "add" command. This time, "reset" tells Git to undo the commit. What follows "reset" is the "--soft" option. The "--soft" option means that the commit is canceled and moved before HEAD. You can now add another file to the staging area and commit, or you can amend files and commit them.

To understand what that "HEAD" thing represents, recall that we work in branches. Currently we're in the master branch, and HEAD *points* to this master branch. When we switch to a different branch later, HEAD will point to that different branch. HEAD is just a pointer to a branch:



What you see in the image is that each dot represents a separate commit, and the latest commit is at the top of the branch (HEAD). In the command "git reset --soft HEAD^" the last character "^" represents the last commit. We can read "git reset --soft HEAD^" as "Undo the last commit in the current branch and move HEAD back by one commit."

Instead of resetting the HEAD and undoing the last commit, we can rectify a commit by using the "--amend" option when committing to a repository. Just add the remaining file to the staging area and then commit:

```
$ git add file-i-forgot-to-add.html
$ git commit --amend -m "Add the remaining file"
```

The "--amend" option lets you *amend the last commit* by adding a new file (or multiple files). Using the "--amend" option, you can also overwrite the message of your last commit.

Think of this command in this way: you took out the top stack of papers from the drawer and "amended" them by simply *unstapling* the bunch of papers, adding another paper, file-i-forgot-to-add.html, on top and rewriting the message on the "commit" paper.

By this time, you've done some work with Git on your computer. You've created files, added them to the staging area, and committed them. But these actions only concern your local repository. When working in a team, you'll also use a *remote* repository. What are the basic Git commands to work with remote repositories? The answer is in the following section.

As a summary, so far you've learned the following Git commands:

Git Cheat Sheet

Git: configurations

```
$ git config --global user.name "FirstName LastName"
$ git config --global user.email "your-email@email-provider.com"
$ git config --global color.ui true
$ git config --list
```

Git: starting a repository

```
$ git init
$ git status
```

Git: staging files

```
$ git add <file-name>
$ git add <file-name> <another-file-name> <yet-another-file-name>
$ git add .
$ git add --all
$ git add -A
$ git rm --cached <file-name>
$ git reset <file-name>
```

Git: committing to a repository

```
$ git commit -m "Add three files"
$ git reset --soft HEAD^
$ git commit --amend -m <enter your message>
```