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

**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.
* 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

**Handy Git Operations**

**Git provides us with a vast number of different commands that are listed on the documentation which can be intimidating at first. We will break down a couple that are powerful for daily tasks.**

**Introduction**

Git provides us with a vast number of different commands that are listed on the [documentation](https://git-scm.com/docs) which can be intimidating at first. We will break down a couple that are powerful for daily tasks.

**Git stash**

Let’s say you’re working on experimental code on a fresh branch and realize that you forgot to add something to a previous commit in order to continue your work. In order to go to a different branch, one must always be at a clean commit point. In this case you don’t want to commit your experimental code since it’s not ready but you also don’t want to lose all the code you’ve been working on.

A good way to handle this is by using **git stash**, which allows you to get back to a clean commit point with a synchronized working tree, and avoid losing your local changes in the process. You’re “stashing” your local work temporarily in order to update a previous commit and later on retrieve your work.

The flow when using **git stash** might look something like this:

While working on a file, you find a small bug in a separate file from a previous commit that needs to be fixed before you continue.

$ git stash

Running the command above will store your work temporarily for later use in a hidden directory.

At this point, you can switch branches and do work elsewhere.

Once the bug is fixed, you want to retrieve the code you were working on previously, you can “pop” the work that was stored when you used **git stash**.

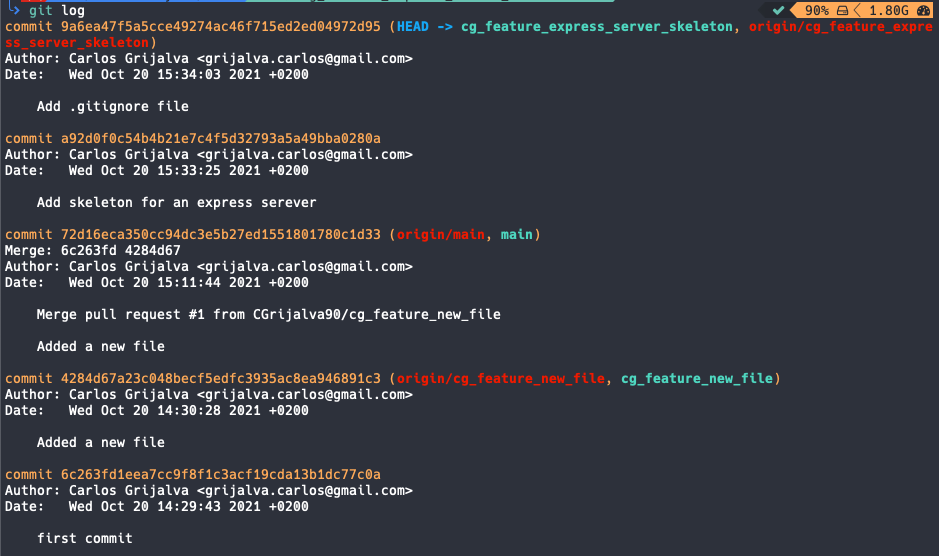
$ git stash pop

From here, you can continue your work and commit it when ready.

Coming up we have a short video demo’ing this in action.

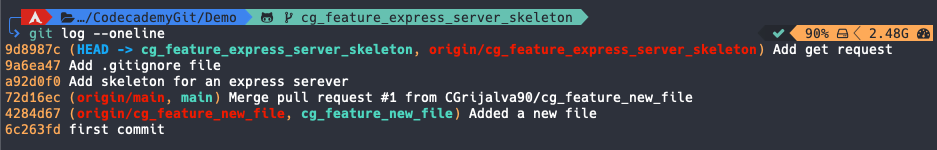
**Git log**

At this point you might be familiar with the command **git log**, which allows you to view the commit history of the branch you currently have checked out:

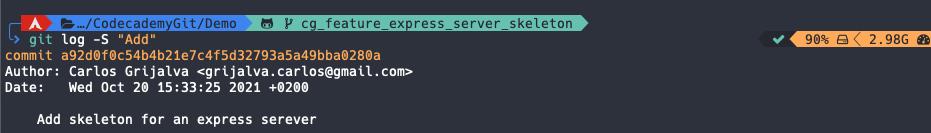


There are other ways you can use **git log** in order to view recorded changes. Here are a few examples:

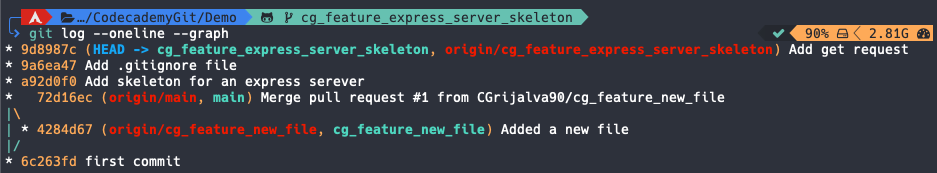
* **git log --oneline** shows the list of commits in one line format.



* **git log -S "keyword"** displays a list of commits that contain the keyword in the message. In the screenshot below, we use **git log -S "Add"** to find any commits with “Add” in the message.



* **git log --oneline --graph** - **--graph** Displays a visual representation of how the branches and commits were created in order to help you make sense of your repository history. When used alone, the description can be very lengthy, so you can combine the command with **--oneline** in order to shorten the description.

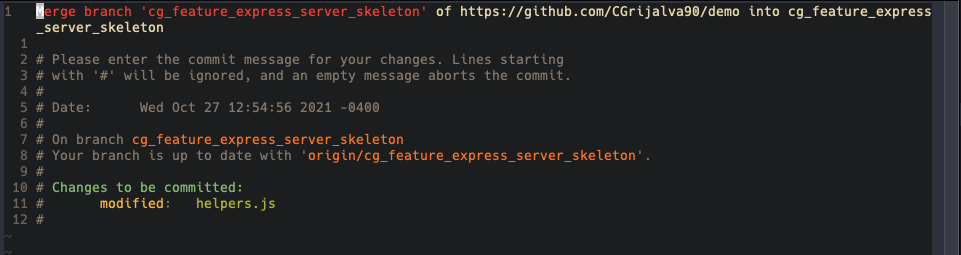


**Git commit amend**

Git’s **--amend** flag is extremely useful when updating a commit, it allows you to correct mistakes and edit commits easily instead of creating a completely new one.

Let’s say you finish working on a lengthy feature and everything seems to be working fine so you commit your work. Shortly after, you realize you missed a few semicolons in one of your functions. You could technically create a new commit, but ideally, you want to keep all commits specific, clean, and succinct. To avoid creating a new one, you could create your changes, stage them with **git add** and then type the command **git commit --amend** to update your previous commit.

It’s important to note that although it seems like **--amend** is simply updating the commit, what Git actually does is replace the whole previous commit. For this reason, when you execute the command **git commit --amend**, your terminal editor asks you to update your commit message:



However, if you want to keep the same commit message, you can simply add the flag **--no-edit**:

$ git commit --amend --no-edit

**Git alias commands**

When grouping commands together, you can end up writing very long lines of Git commands in the terminal such as:

$ git log --pretty=format:"%h %s" --graph

Fortunately, Git offers a helpful feature that can make your Git experience simpler, easier, and more familiar: aliases.

If you have a set of commands that you use regularly and want to save some time from typing them, you can easily set up an alias for each command using Git config.

Below are a couple of examples:

$ git config --global alias.co "checkout"  
$ git config --global alias.br "branch"  
$ git config --global alias.glop "log --pretty=format:"%h %s" --graph"

Once the aliases are configured, next time you want to check out to another branch you could type the command:

$ git co example\_branch

Instead of:

$ git checkout example\_branch

Using Git aliases can create a much more fluid and efficient workflow experience when using Git. By getting creative with your aliases, you’re able to wrap a sequence of Git commands into one in order to save time and effort.