**Handy Git Operations**

**Git stash**

Let’s say you’re working on experimental code on a fresh branch and realise that you forgot to add sometimes 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 synchronised 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.



Running this command will store your work temporarily for late 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***:

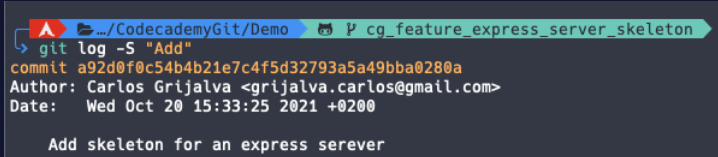


From here, you can continue your work and commit when ready

**Git log**

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

* **Git log --oneline:** shows the list of commits in one line format
* **Git log –S “keyword”:** displays a list of commits where the number of occurrences of the keyword changes within at least one file via addition, deletion, or modification. In the screenshot below, we use *git log –S “Add”* to find any commits where the number of occurrences of “Add” within a file changes



* **Git log --graph:** displays a visual representation of how 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 *--online* in order to shorten the description (like this: **git log --oneline --graph**)

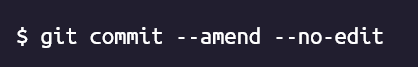
**Git commit amend**

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 realise you missed a few semicolons in one of your functions. Technically, we could create a new commit, but ideally, you would want to keep all commits specific and clean. To avoid creating a new one, you could create your changes, stage them with *git add* and then type in the command: ***git commit –amend***to update your previous commit.

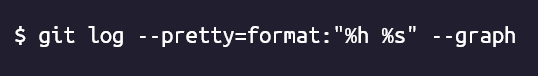
Note: although it seems like *--amend* is simply updating the commit, what Git actually does is replace the whole previous commit. For this reason, when you execute the command *git commit –amend*,your terminal editor asks you to update the commit message.

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



**Git alias commands**

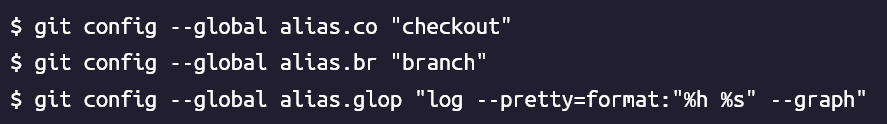
When grouping commands together, you can end up writing very long lines of Git commands in the terminal, such as:



Fortunately, Git offers a helpful feature that can make the 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.

Here are some examples:



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

instead of 

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