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 is powerful for daily tasks.

Introduction

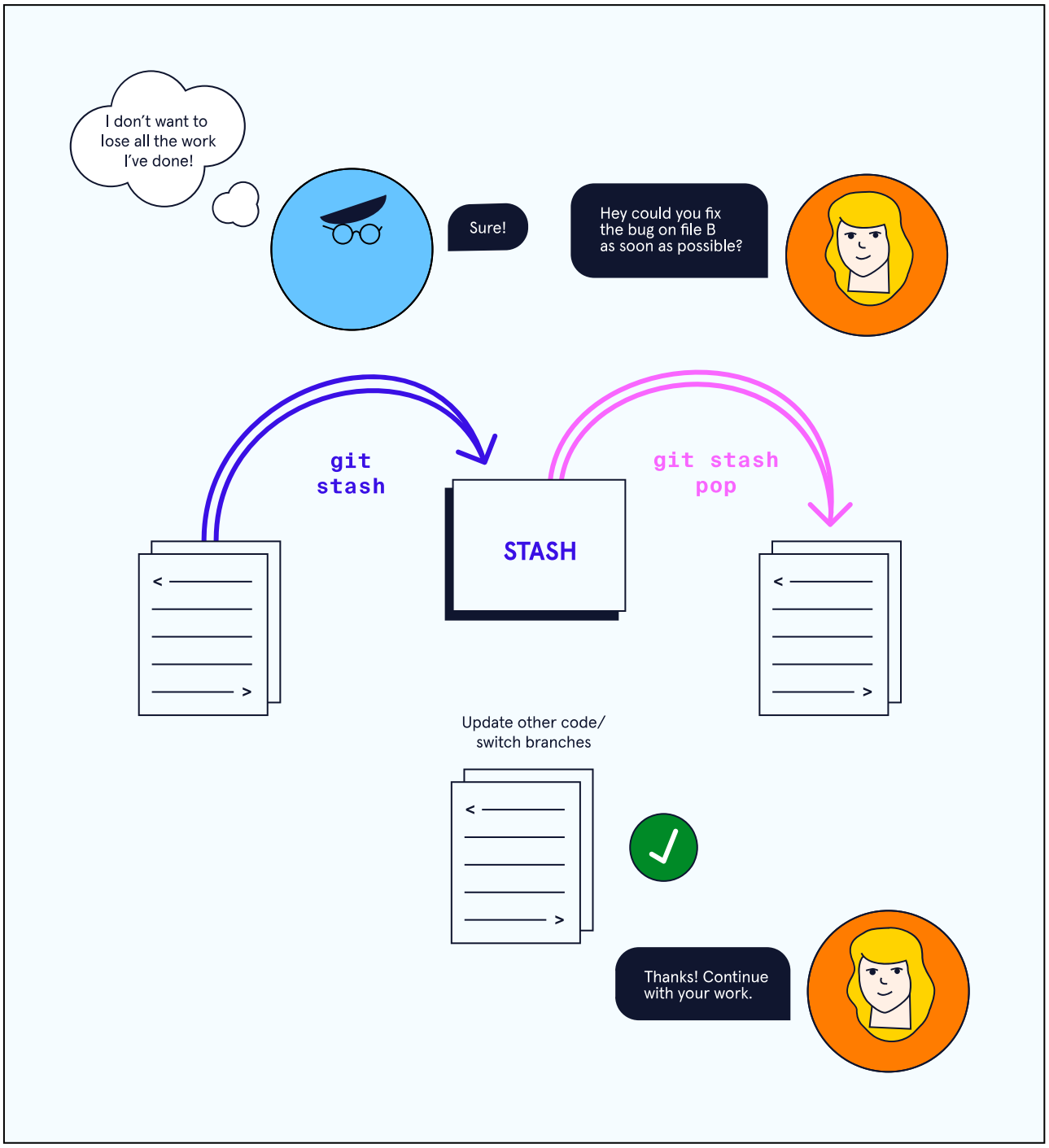
Git provides us with a vast number of different commands that are listed on the documentation which can be intimidating at first. We will break down a couple that are powerful for daily tasks.

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:


I don't want to lose all the work I've done!
Sure!
Hey could you fix the bug on file B as soon as possible?
git stash
STASH
git stash pop
Update other code/ switch branches
Thanks! Continue with your work.

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.

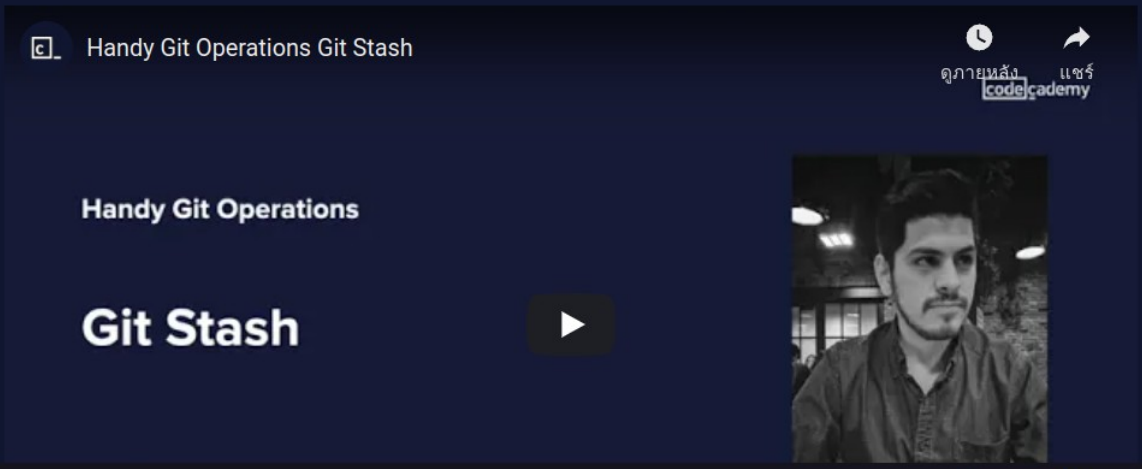
At this point, you can switch branches and do work elsewhere.

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:

```
↳ git log
commit 9a6ea47f5a5cce49274ac46f715ed2ed04972d95 (HEAD -> cg_feature_express_server_skeleton, origin/cg_feature_expre
ss_server_skeleton)
Author: Carlos Grijalva <grijalva.carlos@gmail.com>
Date:   Wed Oct 20 15:34:03 2021 +0200

    Add .gitignore file

commit a92d0f0c54b4b21e7c4f5d32793a5a49bba0280a
Author: Carlos Grijalva <grijalva.carlos@gmail.com>
Date:   Wed Oct 20 15:33:25 2021 +0200

    Add skeleton for an express serever

commit 72d16eca350cc94dc3e5b27ed1551801780c1d33 (origin/main, main)
Merge: 6c263fd 4284d67
Author: Carlos Grijalva <grijalva.carlos@gmail.com>
Date:   Wed Oct 20 15:11:44 2021 +0200

    Merge pull request #1 from CGrijalva90/cg_feature_new_file

    Added a new file

commit 4284d67a23c048becf5edfc3935ac8ea946891c3 (origin/cg_feature_new_file, cg_feature_new_file)
Author: Carlos Grijalva <grijalva.carlos@gmail.com>
Date:   Wed Oct 20 14:30:28 2021 +0200

    Added a new file

commit 6c263fd1eea7cc9f8f1c3acf19cda13b1dc77c0a
Author: Carlos Grijalva <grijalva.carlos@gmail.com>
Date:   Wed Oct 20 14:29:43 2021 +0200

    first commit
```

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.

```
../CodecademyGit/Demo  cg_feature_express_server_skeleton
↳ git log --oneline
9d8987c (HEAD -> cg_feature_express_server_skeleton, origin/cg_feature_express_server_skeleton) Add get request
9a6ea47 Add .gitignore file
a92d0f0 Add skeleton for an express serever
72d16ec (origin/main, main) Merge pull request #1 from CGrijalva90/cg_feature_new_file
4284d67 (origin/cg_feature_new_file, cg_feature_new_file) Added a new file
6c263fd first commit
```

- `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.

```
../CodecademyGit/Demo  cg_feature_express_server_skeleton
↳ git log -S "Add"
commit a92d0f0c54b4b21e7c4f5d32793a5a49bba0280a
Author: Carlos Grijalva <grijalva.carlos@gmail.com>
Date:   Wed Oct 20 15:33:25 2021 +0200

    Add skeleton for an express serever
```

- `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.

```
../CodecademyGit/Demo  cg_feature_express_server_skeleton
↳ git log --oneline --graph
* 9d8987c (HEAD -> cg_feature_express_server_skeleton, origin/cg_feature_express_server_skeleton) Add get request
* 9a6ea47 Add .gitignore file
* a92d0f0 Add skeleton for an express serever
*   72d16ec (origin/main, main) Merge pull request #1 from CGrijalva90/cg_feature_new_file
|\
| * 4284d67 (origin/cg_feature_new_file, cg_feature_new_file) Added a new file
|/
* 6c263fd first commit
```

```
| * 4284d67 (origin/cg_feature_new_file, cg_feature_new_file) Added a new file
|/
* 6c263fd first commit
```

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:

```
1  Merge branch 'cg_feature_express_server_skeleton' of https://github.com/CGrijalva90/demo into cg_feature_express
   _server_skeleton
 1
 2 # Please enter the commit message for your changes. Lines starting
 3 # with '#' will be ignored, and an empty message aborts the commit.
 4 #
 5 # Date:      Wed Oct 27 12:54:56 2021 -0400
 6 #
 7 # On branch cg_feature_express_server_skeleton
 8 # Your branch is up to date with 'origin/cg_feature_express_server_skeleton'.
 9 #
10 # Changes to be committed:
11 #       modified:   helpers.js
12 #
~
```

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.