> You prefer videos? Don't miss the Full Git Crash Course video at the top of this page!

The Main Commands

Let's create a new project, a folder which contains an "index.html" file for example, and add some basic code to it:

```
<p>Hello</p>
<p>Some text</p>
```

> Are you totally new to web development? Make sure to have a look at our How The Web Works video to easily get started!

git init

Let's write our first Git command. `git init` will turn our project into a project managed by Git. You should see the Initialized empty Git Repository info in your terminal, confirming that Git now monitors the project. But why is our project empty?

git status

`git status` provides information about the current state. We have untracked files, meaning that Git is aware of the files being located in our folder but we need to explicitly tell Git to track these files (i.e. to check if the files changed) and to save any changes in a new Commit.

git add

Tracking files either works with `git add filename` ("filename" should be the name of the file you want to track) or `git add .` which will track changes in all files inside the Repository. With `git status` we

though.

git commit

As a last step we need to commit or save the current code version in our Repository. `git commit -m` adds the latest version of the code to our Branch, `-m` should provide a meaningful Commit message. In our case we could use `git commit -m "added starting code"`.

Diving Deeper into Branches and Commits

Time for a quick summary before we dive deeper:

1) We turned our project folder into a Git Repository with `git init` 2) We told Git to track any changes in the `index.html` file with `git add .` 3) We saved the current version of our code in a commit with `git commit -m "added starting code"`

That was simple wasn't it? But what about Branches? I told you before that we automatically created our master Branch with our first Commit, but where can we see this Branch?

git branch

`git branch` lists all Branches in our Repository. Although we didn't add any name, the master Branch was automatically created with our first Commit:

```
Manuels-MBP:project LorenzM$ git branch
* master
```

We have one Branch only at the moment, the asterisk indicates that we are also currently working in this Branch.

Let's add more code to our index.html file:

With `git add .` and `git commit -m "div added"` we can add this new code as a second Commit to our Branch.

##git log

Entering `git log` in our terminal will display all Commits inside our Branch:

```
commit 114ca9893e7005fead7060bfd386225a97141d91 (HEAD -> master)
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
Date:   Wed May 30 11:05:32 2018 +0200

    div added

commit d1a1a697cd8778997af26040422515413ebdaa0a
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
```

Each Commit contains a unique ID, the author, the date and the Commit message (this `-m "your message"` part). “HEAD” just points to the latest Commit of our current Branch, more on that in the next chapter.

Scrolling down a bit will also show us our initial Commit named “starting code”.

But how does Git create these Commits? Each Commit is not an updated copy of the previous Commit. After the first Commit, Git just tracks changes for each following Commit. So with `git add .`, Git checks our files for any changes, `git commit` saves these changes in a new Commit then.

Tracking changes like that makes Git fast and efficient as creating copies over and over again would simply bloat our Git folder, definitely something we want to avoid in any project!

#The HEAD

```
<p>Hello</p>
<p>Some text</p>
<div>Another text</div>

<div>Testing the head</div>
```

I added another “div” and created a new Commit (`git add .` and `git Commit -m "head testing"`), as explained earlier, this Commit became the HEAD as it is the latest Commit in our Branch:

```
commit 245865f7eec1590dafe3ae5cd59a7abd5078da21 (HEAD -> master)
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
Date:   Wed May 30 13:31:05 2018 +0200

    head testing

commit 114ca9893e7005fead7060bfd386225a97141d91
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
```

The latest Commit might not always be the one we want to work on though, we could either just want to have a quick look at a previous Commit or we might want to go back to an earlier Commit and delete later Commits.

##git checkout

`git checkout` is the required command here as it allows us to select a specific Branch (more on that later) or a specific Commit as required in our case.

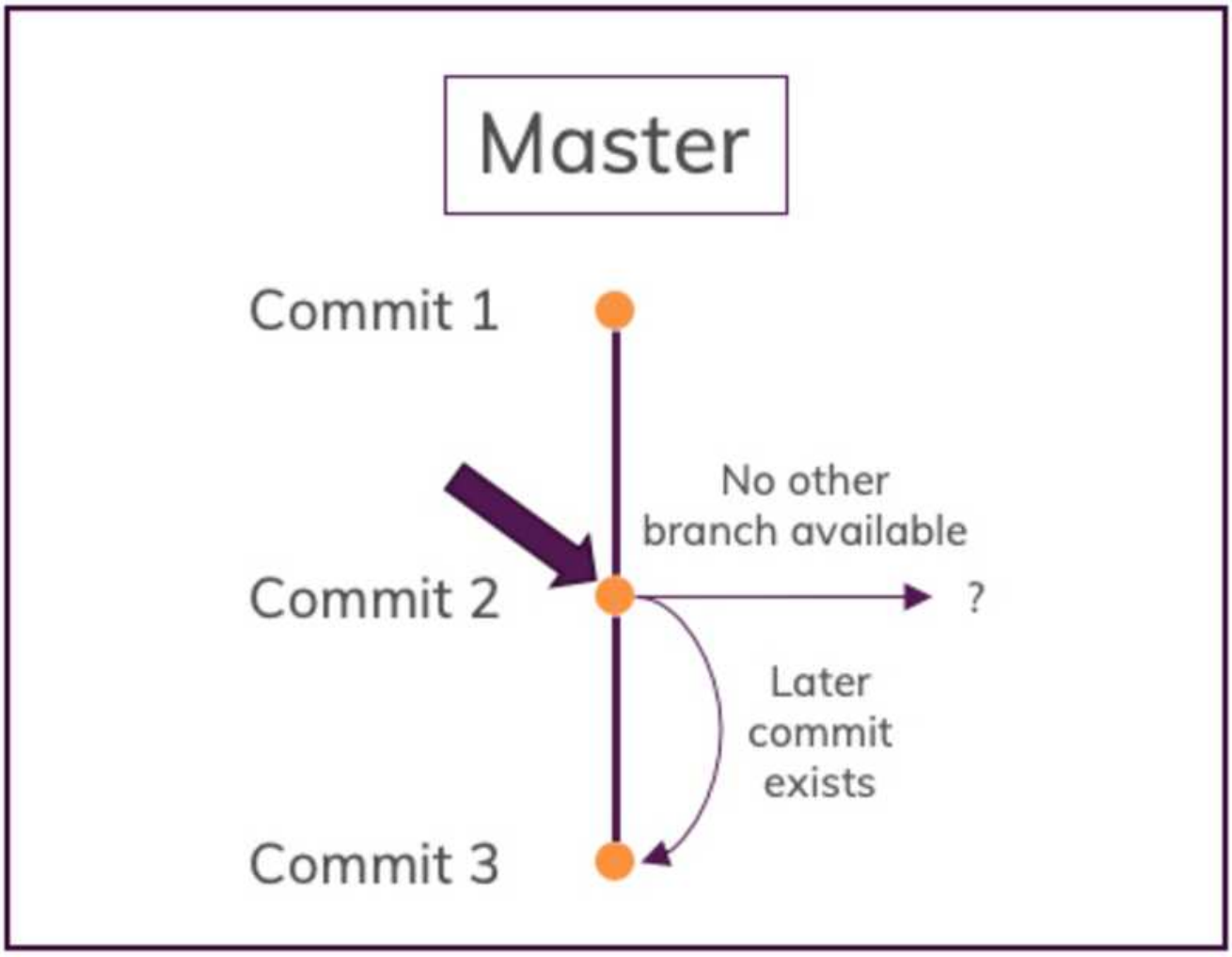
Here we can use the ID we previously had a look at with `git log`:

```
commit 114ca9893e7005fead7060bfd386225a97141d91
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
Date:   Wed May 30 11:05:32 2018 +0200
```

Let’s checkout the second Commit named “div added” with `git checkout yourcommitsID` (just copy and paste the Commit’s ID to

Our HEAD is now pointing towards this Commit, but the Commit we just checked out is no longer part of our master Branch. In simple words it's kind of in the middle of nowhere at this stage, it doesn't belong to any Branch and therefore this approach just allows us to jump back and have quick look at the previous code.

If you really want to go back to this Commit, and therefore get rid of the later Commits, just checking it out like this won't work.



We first have to go back to our master Branch with `git checkout master` (or simply `git master`), now we're back in the latest Commit of the Branch (our HEAD).

Deleting our latest Commit is easy now. Copy the ID of the Commit you want to jump back to (NOT the ID of the Commit you want to delete) and

But be careful! After deleting the Commit we cannot undo this change, so think twice before using this command.

Reverting Unstaged Changes

Let's again add some code:

```
<p>Hello</p>
<p>Some text</p>
<div>Another text</div>

<div>Something I don't need</div>
```

Turns out that we actually don't need that code and we would like to delete it - the changes were not commited though. Besides manually deleting our code (which won't work that easily if you changed a lot of code in different places), `git checkout -- .` let's us jump back to our last Commit (which doesn't include these uncomitted changes of course).

Awesome, these were the basics about Commits, time to move on to Branches now.

Working with Branches

Up to this point we only worked in the master Branch. In a real world project, you typically have multiple Branches, for example:

- The master Branch which contains your currently stable and running version of your website
- The feature Branch where you're currently working on a new feature

The final goal would be to merge both Branches as soon as the new feature is finished and ready to go live.

`git checkout -b` creates a new Branch based on the latest Commit of the Branch you're currently working in. As we are in the latest Commit in the master Branch which contains the "index.html" file, the new Branch created will therefore also include that code and all Commits inside that Branch - so we basically copy the entire Branch.

`git checkout -b new-feature` will create that new Branch named "new-feature":

```
Manuels-MBP:project LorenzM$ git checkout -b new-feature
Switched to a new branch 'new-feature'
Manuels-MBP:project LorenzM$ git branch
  master
* new-feature
```

After creating the Branch we're automatically working in the latest Commit of this Branch (you can always check that with `git branch` and `git log`) and we can add code to it, just a `styles.css` file to keep it simple.

`git add .` and `git commit -m "css added"` will create a new Commit with these changes in our "new-feature" Branch.

This also results in two different HEADS. In the "new-feature" Branch the Commit we just created is the HEAD, in the master Branch we have a different HEAD, our "div added" Commit - the second one on the screenshot:

```
commit 1b9a14add29f19bfe73505144daa4704fd0d933a (HEAD -> new-feature)
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
Date:   Wed May 30 13:56:38 2018 +0200

    css added

commit 114ca9893e7005fead7060bfd386225a97141d91 (master)
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
```

Merging Branches

This will add the changes made in the “new-feature” Branch to our master Branch and therefore our last “css added” Commit is now the HEAD of both Branches.

Deleting Git Branches

As our new feature is implemented into the master Branch, we can get rid of the “new-feature” Branch with `git branch -D new-feature`.

Solving Merge Conflicts

Merging Branches is a great feature, but sometimes it can also cause conflicts.

We currently have this code in the latest Commit in our master Branch:

```
<p>Hello</p>
<p>Some text</p>
<div>Another text</div>
```

Let’s create a new Branch with `git checkout -b new-feature` and let’s change the code as follows:

```
<p>GOODBYE</p>
<p>Some text</p>
<div>Another text</div>
```

After adding and commiting the changes, let’s switch back to the master and change the code like this:

```
<p>BYE</p>
<p>Some text</p>
<div>Another text</div>
```

If we add and commit this change too, we worked on the same code in two different Branches (the first `<p>`), so what happens if we try to

`git merge new-feature` will give us an error: "Automatic merge failed; fix conflicts and then commit the result".

The conflict can be solved by deleting the code you do not want to keep. After that, commit the changes as learned before, this resolves the conflict and merges the Branches successfully.

But again: Be careful whenever you delete code, Branches or Commits, after deleting information it is gone so you should know what you're doing here.

Useful Git Commands

Quickly need to find a Git command? Here is your little helper:

- `git init`: Initialize a Git repository in the current folder
- `git status`: Show the current status of your Git repository (the "working tree" status)
- `git add .`: Track changes of all files in your Repository
- `git commit -m "your message"`: Save updated code to a new Commit named "your message"
- `git log`: List all Commits inside your Branch
- `git checkout branch-name`: Jump to the last Commit of a Branch
- `git checkout commitid` Jump to a specific Commit of a Branch (commitid should be the ID of the Commit you want to checkout)
- `git checkout -- .`: Jump back to the last Commit and remove any untracked changes

- `git branch`: List all Branches inside your Repository
- `git checkout -b branch-name`: Create a new Branch named branch-name
- `git merge branch-name`: Merge two Branches, branch-name is the Branch you merge with the Branch you're currently working in
- `git branch -D branch-name`: Delete the Branch named branch-name