Git and GitHub

Course Material • Hamburg Coding School • 25.7.2020

Outline

Git & GitHub

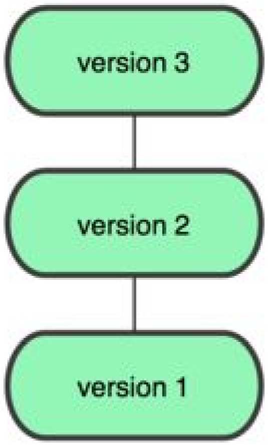
Introduction to Git

What is git?

- Git is a tool for version control.
- It is used for collaborative work on the same code base.
- Example:

If the team that programs the game Super Mario works together, one developer wants to edit the shape of the clouds, and another wants to change the color of the tubes. Git makes it easy that they can both work on the same code simultaneously.

- All changes are saved in so-called **commits**.
- Many commits make up the **version history**, or **commit history**.



Commit history (source: https://git-scm.com/book/en/v1/Getting-Started-About-Version-Control)

- Is git a language or a software? Git is a software.
- Git runs on the command line.
- There are applications with a graphical user interface, but for this workshop, we will use the command line.

Installing Git

Check if git is already installed:

```
$ git --version
git version 2.17.2 (Apple Git-113)
```

On Mac:

We need **homebrew**.
Check if you have it installed:

```
$ brew --version
Homebrew 1.8.3
```

If this gives an error, you need to install homebrew: https://brew.sh/

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)
"
```

Then install git with homebrew like this:

```
$ brew install git
```

Verify if the installation was successful:

```
$ git --version
git version 2.17.2 (Apple Git-113)
```

On Windows:

Download and install git: https://gitforwindows.org/

Open a command prompt and verify if git was successfully installed:

```
$ git --version
git version 2.9.2
```

Command Line Basics

For this, you can use the Terminal (on Mac), or the Command Prompt (on Windows), or use the Terminal in Visual Studio Code.

`cd <directory>`	Navigate into a directory
`cd ..`	Navigate up to the parent directory
`mkdir <directory>`	Create a new directory
`touch <file>`	Create a new file (Mac/Linux)
`type nul > file.txt`	Create a new file (Windows)
`ls`	List all files and directories in the current directory
`rm <file or folder>`	Delete a file or folder (Mac/Linux)
`rmdir <folder>`	Delete a folder (Windows)
`del <file>`	Delete a file (Windows)
`rm -r <folder>`	Delete a folder and all contents (Mac/Linux)
`rmdir /S <folder>`	Delete a folder and all contents (Windows)

Tip:
If you want to know what a command does, type it in: https://explainshell.com. It will give you an explanation and a breakdown of all commands and parameters.

Git Setup

If you use git for the first time, you need to set up a few things before you start:

- Define the user name
- Define your email

This is needed so that git has information about the author of the changes.

You do this with the following commands:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

If you want to see the user name and email you defined, you can use:

```
$ git config --list --show-origin
```

You can also do these things by editing this file: `~/.gitconfig`

(Your **.gitconfig** file is at your home folder. It is a hidden file, so you need to configure your computer to show hidden files.)

More information:
To read more about git configuration and where global and local (project specific) configuration is stored, go to:
https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

Git Commands

`git init`	Create a new git project / initialize git
`git status`	What's the status in this git project? Green: it is staged for commit (git knows this change) Red: it has not been staged and would not be committed *(explained below in more detail)*
`git diff`	See what changed
`git add <filename>`	Add a change to git for the next commit. This command stages the changed file for the next commit.
`git add .`	Add all files that you have been working on.
`git commit`	Make a git commit

This will open **vim**. Write the commit message and close with **`:x`**.
If you don't want to use vim, you can use the command:

```
git commit -m "Your commit message"
```

Best Practice:
Choose a simple commit message that describes precisely what has been changed.

`git log`	See the history of commits
`git --help`	Get help (read the manual)

Official Documentation:
You can look up all git commands here: https://git-scm.com/docs

Staging Changes

Before doing a commit, you have to collect all your changes that you want to commit on the ***stage***. You can say: you need to *stage* them.

You do that with the **`git add`** command.

Files that are staged are marked **green**.

If you then use the **`git commit`** command, all changes that were on the stage will be put into that commit.

Git Status

Git knows four different states of files:

1. Untracked
2. Unmodified
3. Modified
4. Staged

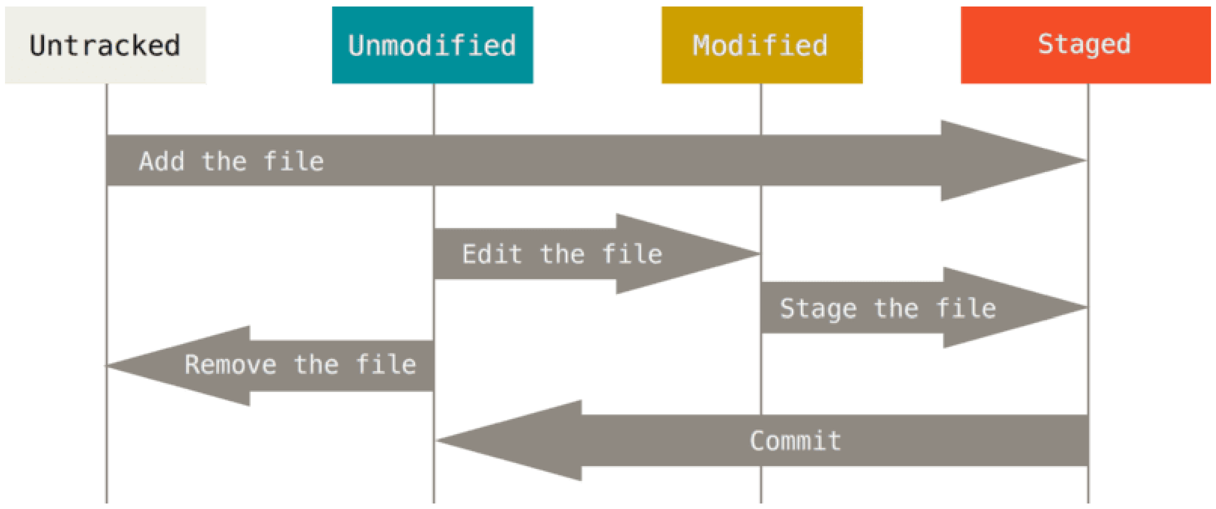
If you create a new file locally, it is first ***untracked***. Git knows that it is there, but it won't do anything with it.

If you **add** a file with the **`git add`** command, the file gets ***staged***. You can also say, you put it *on the stage*. It means that git knows the file, has collected all changes and will include them in the next commit.

If you **commit** your changes with the **`git commit`** command, everything on the stage will be put in a new commit. This will be added on top of your commit history. All changes that you committed now have the status ***unmodified***.

If you now change a file that has been added or committed before, git will know that there are changes that are not on the stage, i.e. the file is ***modified***.

You can always check the status of your files with **`git status`**.



The File Status Lifecycle
(source: https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository)

Undoing Changes

Resetting Modified Files:

If you want to undo local changes, i.e. revert changes of ***modified*** files, you can do that with the command:

```
git checkout -- <file>
```

If you want to discard all changes in all modified files, you can use:

```
git checkout -- .
```

The dot is a shortcut for all files.

Removing files from the stage:

If you have ***staged*** files, but you don’t want to commit them, you can remove them from the stage with:

```
git reset HEAD <file>
```

Or if you want to remove all files from the stage:

```
git reset HEAD .
```

Undoing a commit:

You can undo the last commit by resetting it:

```
git reset --soft HEAD~1
```

The **`--soft`** flag has the effect that all changes of your last commit are not deleted, but they are now uncommitted modified files.

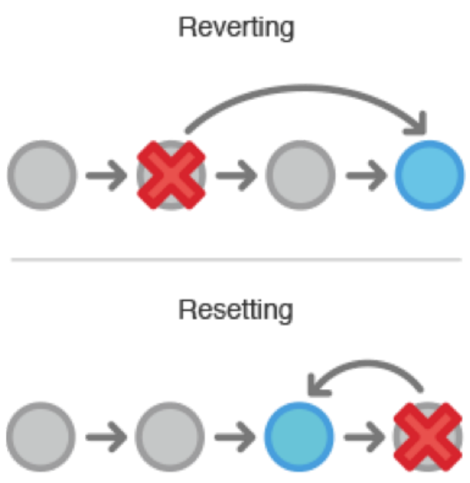
If you are sure that you want to delete all changes of the last commit and never see them again, you can use the **--hard** flag instead:

```
git reset --hard HEAD~1
```

The tilde **~** sign followed by a **1** means: take the HEAD (your current position in the git history) and reset one commit from there.

Reverting a Commit (Optional Knowledge):

Instead of resetting a commit, you can also revert a commit.
The difference is: if you reset a commit, you remove it; if you revert a commit, you add an additional commit to the history that undoes the changes.



Reverting vs. Resetting
(Source: https://alexdiliberto.com/talks/all-things-git/)

Reset

You use this when:

- You want to undo the last commit(s).

Revert

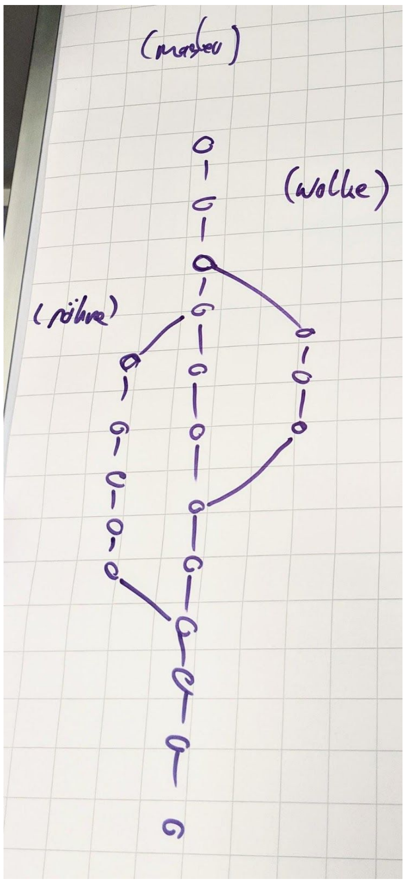
You use this when:

- You want to undo a commit that is some commits back in the commit history.
- It is important that you see in the point in the commit history when the revert has been done (may be important in large projects with many developers).

Branches

If you want to collaborate on the same code base, you use branches, so everyone can commit changes independently.

A branch is a string of commits that is independent of the other commits.



Example:

In the Super Mario development team, we have a current status of the published game. That's on the ***master*** branch.

The developer who wants to change the cloud, creates a branch called ***cloud*** and makes his commits there.

The developer who wants to change the tube, creates a branch called ***tube*** and makes his commits there.

When they are done, each developer merges his changes into the master branch.

This results in a commit history with multiple branches.

The **master** branch is usually the default branch.

If you want to make changes independently from the other developers, you create a branch. When you are done, these can be merged back to the master branch.

Git Commands for Branches

`git branch`	Shows all existing branches, marks the one you are on.
`git branch newbranch`	Creates a new branch called "newbranch".
`git checkout newbranch`	Switches to the branch called "newbranch".
`git checkout -b newbranch`	Shorthand for the two above: creates a new branch called "newbranch" and then switches to it.
`git branch -d oldbranch`	Deletes the branch "oldbranch".

Remember:
Always make sure you are in the branch you want to be in, with: `git branch`

Merging a branch into master

First, you need to be on the master branch.	`git checkout master`
Then, you merge your branch into master.	`git merge branchname`
You don't need the merged branch anymore, so you can delete it.	`git branch -d branchname`

Best Practice:
Always delete branches that you don't need anymore. The less branches you have, the better: it is much less confusing. Make it a habit to delete old branches before you create a new one.

Note:

You cannot delete a branch you are on. If you want to delete a branch, you need to be on a different branch.

If the branch is not fully merged yet, you get an error message. To delete the branch anyway, type:

```
git branch -D branchname
```

Best Practice:

Give meaningful but short branch names. A branch name should describe the feature or bugfix you are working on.

You usually use minus "-" between words:

```
user-login
fix-404-error
```

A common branch naming strategy is to use "feature/" or "bugfix/" as a prefix:

```
feature/user-login
bugfix/404-error
```

Optional Knowledge:
If your team uses a ticket system, e.g. Jira, you can use the project name and ticket number to identify the task you are working on.
In this example, we use a project called WEBSITE:

```
feature/WEBSITE-347-user-login
bugfix/WEBSITE-348-404-error
```

Branching Strategies (Optional Knowledge)

You can use git differently, depending on your team size.
There are different branching strategies:

If you work alone:

It doesn't make so much sense to use a lot of branches. You can mainly use the master branch. It makes sense for larger changes to create branches, so you can continue working on the master branch, even though you are not finished with your work on the other branch.

If you work in a team:

You need multiple branches. The master branch is usually the one where the published version of your website or app is.

An example for a popular branching strategy for larger teams with continuous delivery:

https://nvie.com/posts/a-successful-git-branching-model/

Git Commit History

It is a good idea to have a look at your git commit history on a regular basis.

Showing the git history:

```
git log
```

This will look something like:

```
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

If the log is getting too long, git will show a colon `:` at the end. This means: you are in scroll mode. You can scroll down or up with your arrow keys.

💡 Exit the scroll mode by typing **q**

To see only one line per commit, you can use:

```
$ git log --pretty=oneline
```

Output:

```
ca82a6dff817ec66f44342007202690a93763949 changed version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

For a super pretty graph with colors, you can use this command:

```
git log --graph --pretty=format:'%Cred%h%Creset
-%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset'
--abbrev-commit
```

Output:

```
* 2d3acf9 - (HEAD, master) ignore errors from SIGCHLD on trap (11 days ago) <Scott Chacon>
* 5e3ee11 - Merge branch 'master' of git://github.com/dustin/grit (2 weeks ago) <Scott
Chacon>
|\
| * 420eac9 - Added a method for getting the current branch. (3 weeks ago) <Scott Chacon>
* | 30e367c - timeout code and tests (3 weeks ago) <Scott Chacon>
* | 5a09431 - add timeout protection to grit (3 weeks ago) <Scott Chacon>
* | e1193f8 - support for heads with slashes in them (3 weeks ago) <Scott Chacon>
|/
* d6016bc - require time for xmlschema (7 weeks ago) <Scott Chacon>
* 11d191e - Merge branch 'defunkt' into local (9 weeks ago) <Scott Chacon>
```

Git Aliases (optional knowledge)

In git, you can save long commands under short names, creating your own shortcuts. These are called ***aliases***.

In the example above, the command for making a pretty graph log is very long, so we want to create an alias for it. We do it like this:

```
$ git config --global alias.lg "log --color --graph
--pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"
```

What is happening here is: we save an alias called “**`lg`**” in the global git config file (see the **`alias.lg`** there?).

Now we can use this on the command line like:

```
$ git lg
```

It will output our pretty graph log just as the long command.

To create your own git alias, you can use it like this:

```
$ git config --global alias.lg "..."
```

In the double quotes you put your git command, just without “git” (everything after “git”).

The git log color coding example is from here:
https://coderwall.com/p/euwpig/a-better-git-log

GitHub

While git is a software, **GitHub** is a platform to host your git repositories.

You can make it private (visible only for you) or public (visible for anyone, used for open source).

Git has a user interface that displays diffs and commits neatly.

https://github.com/

Remote Repositories

What is a repository?

A git **repository** is a project, or the root folder of your project.
If you used **`git init`** to initialize it, you created a repository.

Now, git is made so that many people can work on the same code base. This is realized by having a copy of the repository somewhere on a server, where everyone can access it. This copy is called: **remote**.

What is a remote?
Remote is your server where you store your git repository. With that, you synchronize your local repository that is on your laptop.

In our case, we will use **GitHub** as remote.

Clone a Repository

Cloning means creating a copy of a remote repository.

```
$ git clone <url of a remote repository>
```

Create a remote copy of your repository

1. Create an empty new repository on GitHub.
2. Upload your local repository to this new GitHub repository:

```
$ git remote add origin <url to your GitHub repository>
```

Check what your remote is

Sometimes you want to check if you have a remote, or what your remote is.
Remotes have a name, also called an ***alias***, and a URL (the link to the remote).

To see if you have a remote, and what its alias is, use:

```
$ git remote show
```

In our case, this shows **`origin`**. This is the name or alias for our remote server.
That's the default name that every remote repository gets.
You usually leave it at that. Just if you want to add more remotes, you would use different names.

To see what the URL of the remote is, use:

```
$ git remote show origin
```

This will show something like this:

```
* remote origin
  Fetch URL: git@github.com:hamburgcodingschool/website.git
  Push  URL: git@github.com:hamburgcodingschool/website.git
  HEAD branch: master
```

Push and Pull

To push your local, committed changes to your remote repository, use:

```
$ git push
```

If your branch is new, you need to add something to upload it:

```
$ git push -u origin <branch name>
```

To see if there are changes that someone else pushed to remote:

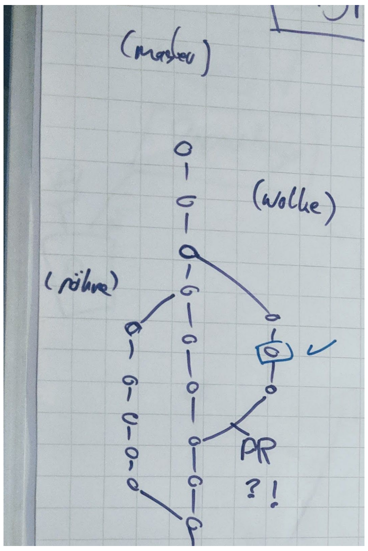
```
$ git fetch
```

To pull these changes to your local repository:

```
$ git pull
```

Pull Requests

GitHub has a feature called **Pull Requests.**
A pull request is a request to merge a branch back to master (or back into the original branch where you branched it from).



The easiest way to create a Pull Request is to do so from the GitHub website.

For example, if you pushed something to a new branch, there will be a yellow button for creating a Pull Request on the main page of your GitHub repository.

Pull Requests are a good way to have your team mates look at your code before it is merged and do a **code review**.

Pull Request Workflow:

On GitHub:

1. Click on "Create pull request".
2. Check your changes below. Correct mistakes, if you find any, and push them again.
3. Give the pull request a meaningful title, and some description if it is a larger change.
4. Create the pull request.
5. Choose someone to give you a code review.
6. If they request changes, change your code and push again.
7. If they approve, click "Squash and merge." (If there are conflicts, solve them directly in GitHub if possible).
8. After merging, click "Delete branch" to delete the merged branch.

Locally:

1. Go to the master branch	**`git checkout master`**
2. Get the new commits of the merged branch	**`git pull`**
3. Delete the old branch	**`git branch -d oldBranch`**

Solving Conflicts

There might be a situation where both your team mate and you changed the same file, and now you want to merge both changes into master.

Git cannot know which version has preference, because only humans may do that, so it tells you: there is a conflict.

On the command line, this looks something like this:

```
$ git checkout master

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.
```

In the file that has the conflict (here: `index.html`), it marks the conflict something like this:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Both changes have changed the same line. Now you need to erase everything except the change that you want to keep. If you prefer the second version, it should now look like this:

```
<div id="footer">
 please contact us at support@github.com
</div>
```

Then you add the file:	**`git add index.html`**
And then you do:	**`git commit`**

This is how git conflicts are solved. ✌

More Information:
To read more about solving conflicts in git, refer to:
https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

GitHub Pages

You can use GitHub for publishing your website.

GitHub Pages is some sort of web space. It works with GitHub.

Instead of FTP and using a web server, we can use GitHub Pages.

For that, the repository needs to be public, or else you need to have a Pro account.

You can make a repository public on **Settings** > **Danger zone**.

Rebase (optional knowledge)

Sometimes you have the following situation:

Your branch has changes, but master also has changes.
You don't want to merge master into your branch, but you want your branch to build on the newest commits on master.

With **`rebase`** you can take your branch and put it on the latest commits on master.

```
$ git rebase master
```

This means that you "rewrite history": the branch you are currently on will be "plucked out" and planted on top of the master branch.

Watch out: after a **`git rebase master`** you have to do a **`git push -f`** so that the remote is the same as yours locally.

Rebasing - Expert Mode 🤓 (optional knowledge)

There is also an interactive mode for rebasing. With that, you can make even more things with it, e.g. rewrite the commit history.

```
$ git rebase HEAD~3 -i
```

This opens a vim editor, that looks something like this:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
# ...
```

There you can, for example, squash multiple commits to one.
You do this by changing the prefix that is currently pick.

`pick`	will keep that commit with that commit message.
`amend`	will let you change the commit or commit message.
`squash`	will put the changes in that commit into the one with pick.
`drop`	lets the commit disappear, as if it was never there.

💡 Close the vim editor with **`:wq!`**

If there are conflicts, you need to solve them manually.
Once you did that, you add the files with **`git add`**, and then do:

```
$ git rebase --continue
```

It is possible that you need to solve the same conflicts multiple times, because git takes each single commit and puts them into its new place.

Rebasing is a difficult thing. It can be a valuable skill knowing how to use rebase, but if you have little experience, it is ok to not know the details. It is totally ok to first try that together with a mentor or teacher.

Ask us anytime if you need help!

More Information:
Rebasing: https://git-scm.com/book/en/v2/Git-Branching-Rebasing
Rewriting history: https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History

Glossary

`version control`	A system that you can use to save versions of your files
`commit`	A version of your project
`version history / commit history`	The list of all versions of your project
`repository`	Your project
`vim`	A text editor on the command line
`HEAD`	Your current position in the git commit history / branches.
`remote`	A remote repository that you can access via the internet.
`alias`	1. A custom shortcut for a command. 2. A short name for the remote.

General Commands for the Command Line

`cd <directory>`	Navigate into a directory
`cd ..`	Navigate up to the parent directory
`mkdir <directory>`	Create a new directory
`touch <file>`	Create a new file (Mac/Linux)
`type nul > file.txt`	Create a new file (Windows)
`ls`	List all files and directories
`rm <file or folder>`	Delete a file or folder (Mac/Linux)
`rmdir <folder>`	Delete a folder (Windows)
`del <file>`	Delete a file (Windows)

`rm -r <folder>`	Delete a folder and all contents (Mac/Linux)
`rmdir /S <folder>`	Delete a folder and all contents (Windows)

Git Commands

`git --version`	Tells you which version of the git software is currently installed
`git init`	Create a new git project / initialize git
`git status`	What's the status in this git project? Green: it is staged for commit (git knows this change) Red: it has not been staged and would not be committed
`git diff`	See what changed
`git add <filename>`	Add a change to git for the next commit. This command stages the changed file for the next commit.
`git add .`	Add all files that you have been working on.
`git commit`	Make a git commit. This will open **vim**. Write the commit message and close with `:x`.
`git commit -m "Your commit message"`	Make a git commit and include the commit message.
`git log`	See the history of commits
`git --help`	Get help (read the manual)
`git checkout -- <file>`	Revert changes in a modified file
`git checkout -- .`	Revert changes in all modified files
`git reset HEAD <file>`	Removes a file from the stage

`git reset HEAD .`	Removes all files from the stage
`git reset --soft HEAD~1`	Undo last commit by resetting it. The **`--soft`** flag has the effect that all changes of your last commit are not deleted, but they are now uncommitted modified files. The tilde **~** sign followed by a **1** means: take the HEAD (your current position in the git history) and reset one commit from there
`git reset --hard HEAD~1`	Deletes all changes of the last commit to never see them again.
`git branch`	Shows all existing branches, and marks the one you are on.
`git branch newbranch`	Creates a new branch called “newbranch”
`git checkout newbranch`	Switches to the branch called “newbranch”
`git checkout -b newbranch`	Shorthand for the two above: creates a new branch called newbranch and then switches to it
`git branch -d oldbranch`	Deletes the branch “oldbranch”
`git clone <url of a remote repository>`	Creates a copy on your computer of a remote repository
`git remote add origin <url to your GitHub repository>`	Creates a remote copy of your repository
`git remote show origin`	Shows the URL of the remote repository
`git push`	Pushes your local, commited changes to your remote repository
`git push -u origin <branch name>`	To push a branch for the first time to the remote
`git fetch`	Fetches all information about the remote repository.

`git pull`	Pull all new changes from the remote repository to your local repository.
`git merge master`	Merge the branch “master” into the branch you are currently on
`git rebase master`	Rewrite history so that the branch you are currently on will be “plucked out” and planted on top of the master branch.

Official Documentation:
You can look up all git commands here: https://git-scm.com/docs

Useful Links

The Git Book: https://git-scm.com/book/en/v2

Git Documentation: https://git-scm.com/docs

Explain Shell Commands: https://explainshell.com

Online book with cheat sheets: https://www.git-tower.com/learn/git/

Slides of an introduction to git: https://alexdiliberto.com/talks/all-things-git/

GitHub: https://github.com/

GitHub and Git Tutorial: https://guides.github.com/introduction/git-handbook/

Excellent git tutorial and cheat sheet: https://duzun.me/tips/git

Adding a remote: https://help.github.com/articles/adding-a-remote/

An example for a popular branching strategy for larger teams with continuous delivery:

https://nvie.com/posts/a-successful-git-branching-model/