Git Crash Course for TDDC88 Group 2

David Berntsson

September 18, 2025

When working in a project, it's important to have version control system (Git) in case things break and you want to roll back to an earlier stage of development, as well as an online backup (GitHub/GitLab) in case you lose your local copy of the project. Furthermore, if you're working in a team, a version control system is vital to make the collaboration as seamless as possible.

Before we get started with git, we first need an ssh key. This allows us to inform our GitLab account that our computer can be trusted when cloning and working on a project. To generate an ssh key, the following terminal command should work on all operating systems:

```
ssh-keygen
```

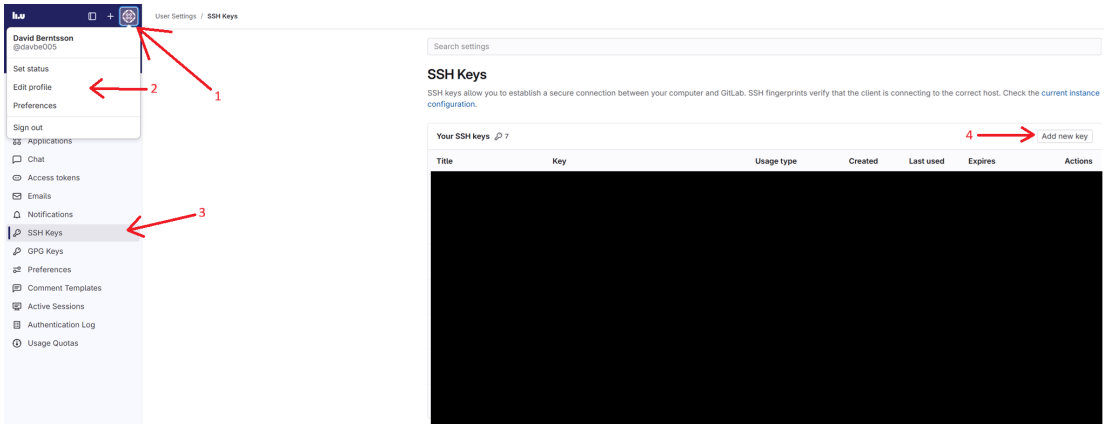
This will give you a few prompts such as password and save location, you can simply press enter for all of these prompts and it will generate the key at the location it suggested.

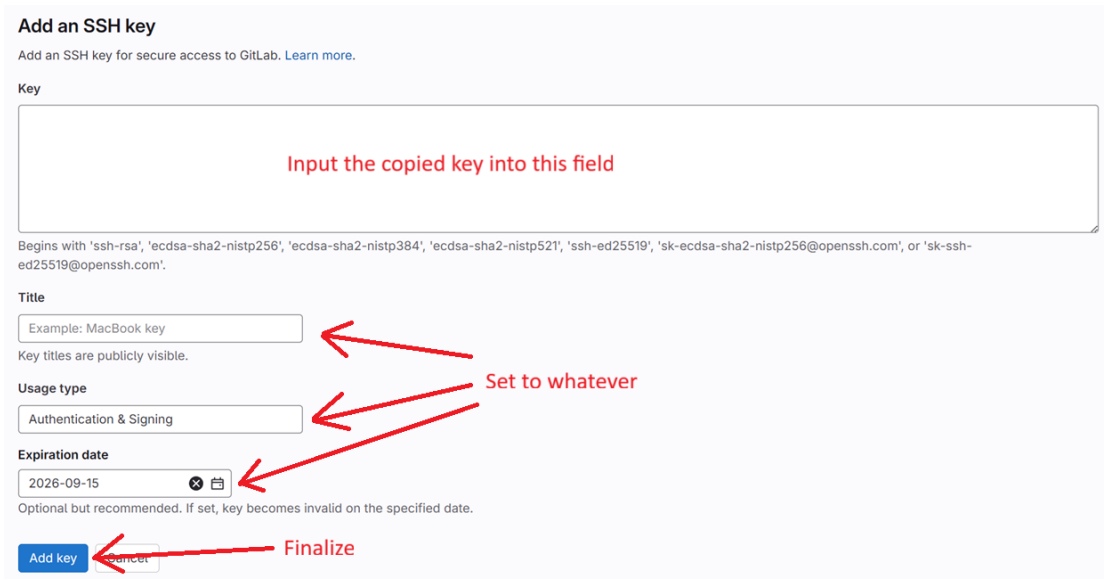
Next, you need the public part of the key you just generated. The file we're looking for while be the same as the one that was generated, but with '.pub' added at the end. You can either open the file with any text editor or with the following command:

```
cat path/to/.ssh/<key>.pub
```

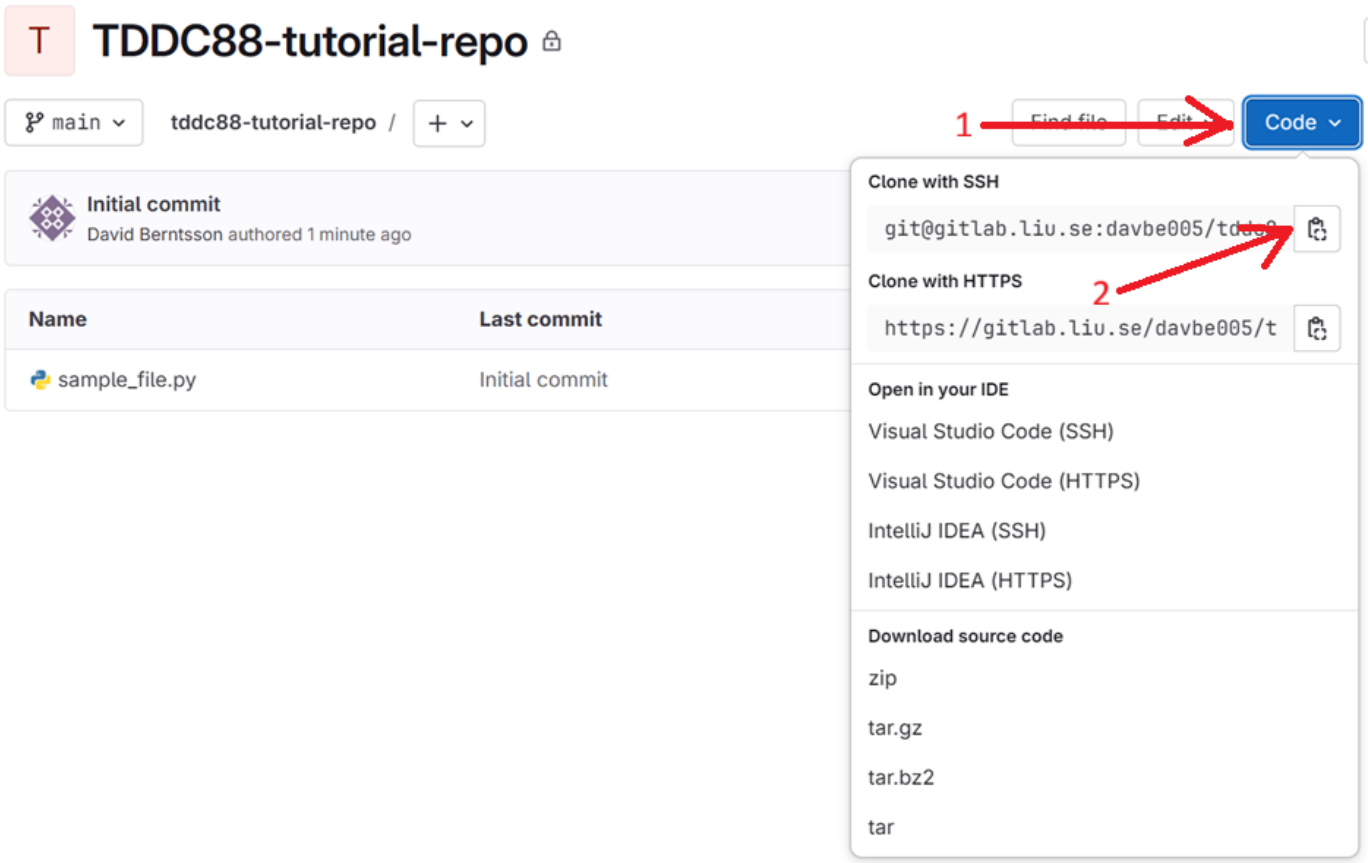
Where the path is replaced by the actual path you generated or chose. After you have the key, simply copy it from the terminal or text editor.

Next, log in to gitlab and perform the following actions to add an ssh key:





Once this is done, you can now clone the repo through ssh. Navigate to the tutorial repo (https://gitlab.liu.se/davbe005/tddc88-tutorial-repo) and click the following buttons:



Next, go back to the terminal and type the following command (replacing `<url>` with what you copied from the repo):

```
git clone <url>
```

Now you have a local copy of the repo. In git, you can use branches to work on different features simultaneously. Think of it like different miniprojects that you can later merge together to combine all the features. To see all the branches in the project (both remote and local), type the following:

```
git branch -a
```

As you can see, you are currently in the main branch, which should have some commits. To see a log of the commits, you can type the following command:

```
git log
```

You can use the up and down arrows or page up and page down buttons to scroll through the log, and exit by pressing Q.

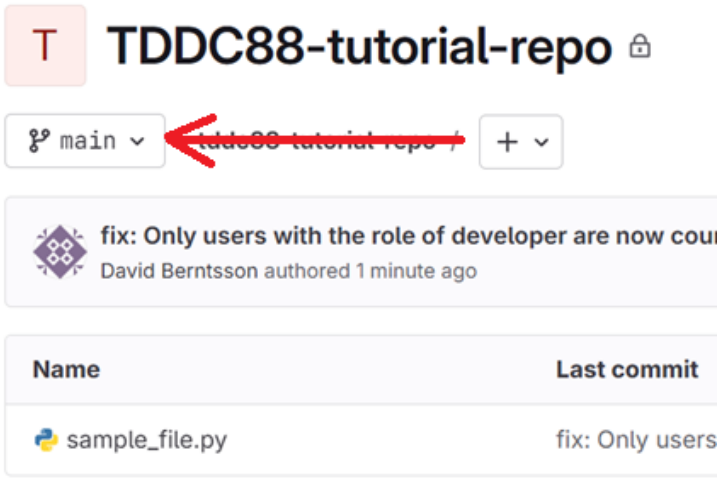
To create a new branch and switch to a new branch, you can do the following:

```
git checkout -b <branch>

or

git branch <branch>
git switch <branch>
```

You have now created a local branch based off of main, if you run `git branch -a` you'll see that the branch you are currently on is the one you just created and if you run `git log` you'll see that it has the same commits as the main branch. To also create (and connect it to the local version) this branch on the remote repository, run the following command:

```
git push -u origin HEAD
```

If you go back to the online repository, and click the following button, you should see the branch you have created:



Now, make some changes in the sample_file.py such as adding your name to the list of users or fixing something you think looks like a bug or adding another feature. Once you are done, you need to add the changed files (one in this case, unless you've created new ones) to the commit. Run the following command:

```
git status
```

All the files that you haven't added to the commit yet are red and the ones you've added are green, to add a specific file, you can run the following command:

```
git add path/to/file.extension
```

If you want to add all the red files, either of the following commands usually work:

```
git add *

or

git add -A
```

The asterisk in `git add *` is generally referred to as a wildcard, meaning it will add any file that matches the statement, where the '*' can be replaced by anything. For example, if you have a folder with several files, you can run

```
git add folder/*
```

and it will add all the files in that folder. Furthermore, if you want to add all files of a specific type, you can also use the wildcard.

```
git add *.py
```

will add all files that end in the .py file extension. These can of course be combined and

```
git add folder/*.py
```

will add all files that are in the folder and ends with .py.

After adding the file or files, running `git status` again should show the files as green which means that they will be part of the commit. To commit the files, run the following command:

```
git commit -m "<message>"
```

The -m flag means that you will add a message to the commit (this is standard practice and if you try to run the command without it, you might be prompted with a text editor to add a commit message anyway). The message itself should describe what the commit is about. When I worked at Axis, we used the following commit semantics:

Figure 1: Image taken from https://gist.github.com/joshbuchea/6f47e86d2510bce28f8e7f42ae84c716

I think it brings a lot of structure to the commits and forces the developers to think about creating an informative message so we will also be using this structure when creating our commits. Furthermore, the title of the commit should be no more than 50 characters. Furthermore, any lines below the title should also not be too long, roughly around 72 characters. These aren't really hard rules but a general guideline that most developers follow. To create a commit message with multiple line in the terminal, simply omit the second ””’.

For example, a commit message could look like the following:

```
git commit -m "feat:  Add user profile picture upload
Users can now upload profile pictures via the settings page.
Supported formats are PNG, JPEG, and WebP."
```

To push the added commit to the remote repository, simply run the following command:

```
git push
```

If you go back to the website, and navigate to your branch, you should see that it has the commit you just pushed and that the files have been modified.

Now, while working on a feature in a branch, the main branch might have received new commits. This means that the branch you're currently working on is no longer built on top of the main branch. I.e. they have different commit histories. To obtain the new commits in the main branch, you can do the follow command while you're in your feature branch:

```
git rebase main
```

Note, that this will rebase your branch on to your local copy of the main branch, to have the new commits from the online repo, you must first switch to the main branch and do the following command:

```
git pull.
```

This is the opposite of `git push`. It will pull the commits from the remote branch and merge the changes into your local copy of the branch.

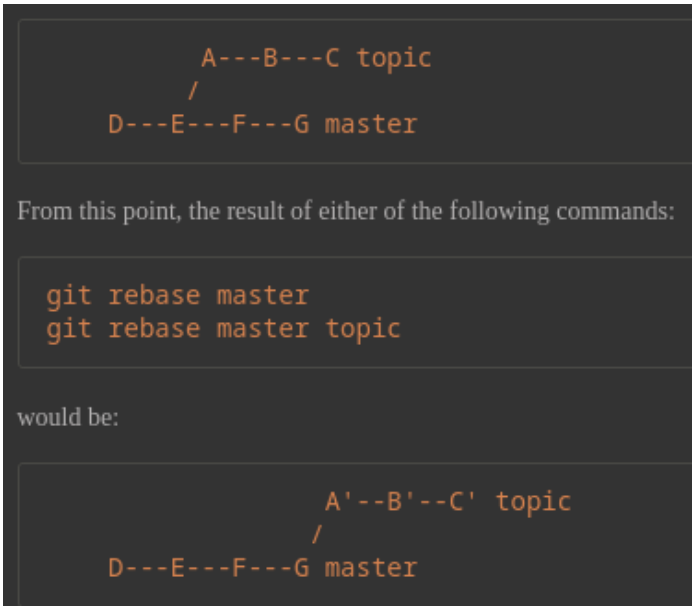
Merge Conflicts

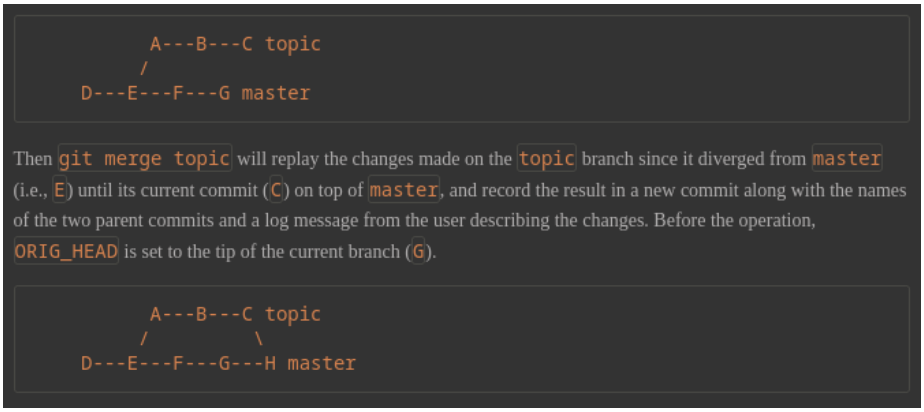
A major fear of people working with Git is dealing with merge conflicts. A merge conflict usually occurs because two different commits from two different branches have modified the same line in a file. Git is unable to resolve this on its own, and therefore it leads to a merge conflict that the developer performing the merge or rebase will have to resolve. Depending on how we work in this course, some of us might have to deal with merge conflicts, and some might not. If any of you encounter a merge conflict, you can either try to resolve it on your own, or you can get in touch with one of the lead developer who hopefully has experience dealing with them. If you're resolving it on your own, an editor such as VSCode can be used to deal with a merge conflict in an easier manner than doing in a simple text editor such as notepad on Windows. But remember to always be careful when solving a merge conflict because either one of the

modifications should be kept or both in some cases. During the education, I will show you how a merge conflict works, and how to resolve it.

Merge vs Rebase

Durign the merge conflict section I mentioned that you can either merge or rebase, what is the difference? The following two figures from the git documentation show how each functionalities work:





Generally, you wanna do rebases because it keeps the commit history clean. Instead of branching paths back and forth for all feature branches being merged into the main branch, you instead get a singular line of commits. Then when you want to bring a feature branch into main, you do it through a pull request (or merge request as it's called on GitLab) which will handle things automatically for you. It's done this way since merges into main first need to be approved by appropriate people otherwise new hires could for example perform disastrous things to the main branch.

Squashing commits

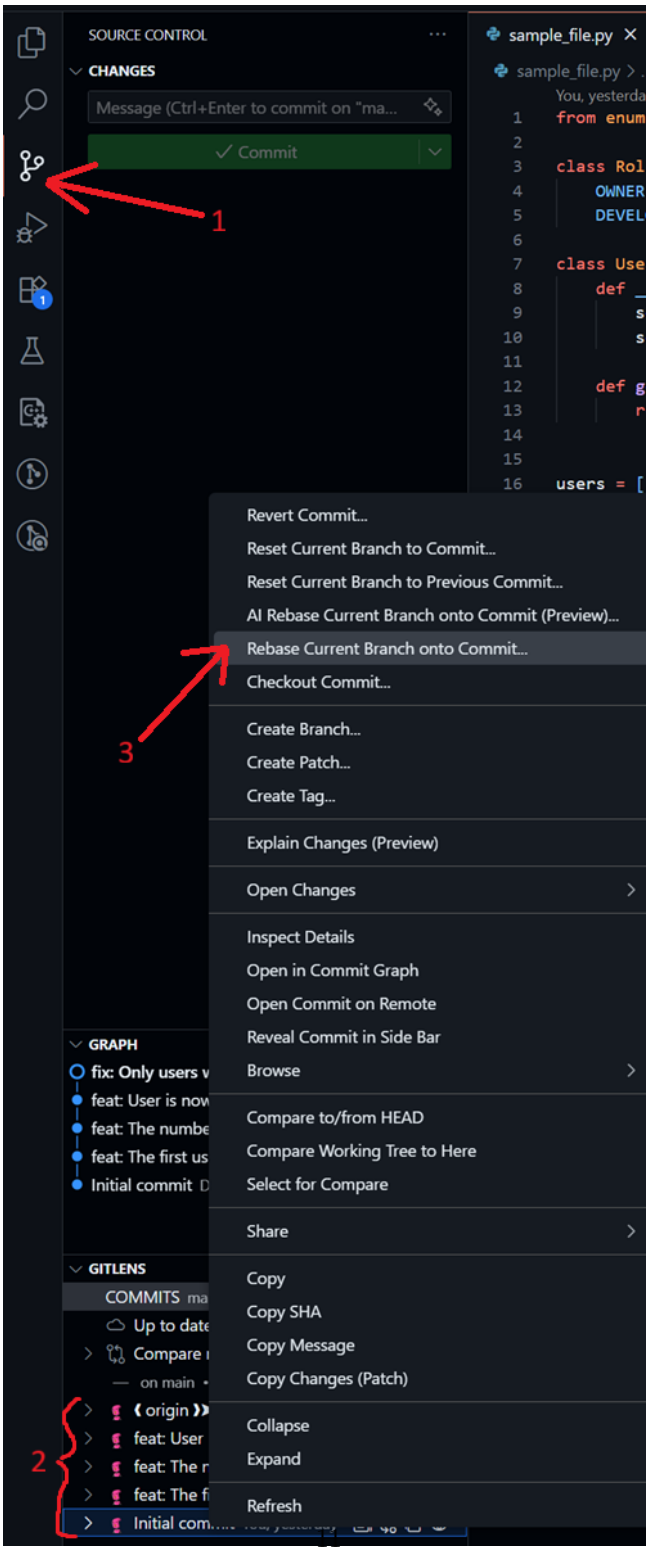
Sometimes, you might implement a feature, then later on you find a bug and you fix it in a separate commit. The history could look like the following:

```
e0184a feat:  add feature A
50c9e8 fix:  fix some bugs with feature D
32096c feat:  add feature D
ae4f28 refactor:  refactor some functions in B
6dcd4c feat:  add feature C
```

In this case, you would want to squash commit 50c9e8 into 32096c so it looks like feature D never needed any bugfixes. This is useful because someone else looking at your commit history doesn't care about fixes you had to implement for features you also implemented. Imagine you're part of a large project and it has thousands of commits. If 30% of those commits are fixes that could've easily been squashed into the feature they fixed, then you would have a lot of commits that are uninteresting to anyone looking through the commit history and the commit history would be easier to navigate if those 30% of commits were instead squashed. I will show you an example of squashing a commit into another. Generally, you run the following command:

```
git rebase -i <commit>
```

Where ¡commit¿ is the non-inclusive endpoint for all the commits you want to include in your interactive rebase, if you also want the ¡commit¿ itself you can do ¡commit¿ 1 and it will be included in the interactive rebase. Performing the command will bring up a text editor where you can choose what to do with each commit, and it shows you the option you can choose from. This is, however, a rather cumbersome process. If you're using VSCode, you can use gitlens to perform an interactive rebase and it will provide a simple to use UI where you can easily choose between the options and reorder commits. To perform an interactive rebase with gitlens, click the following buttons after downloading it from the extensions tab:


SOURCE CONTROL
CHANGES
Message (Ctrl+Enter to commit on "ma...
Commit
1
sample_file.py
from enum
class Rol
OWNER
DEVEL
class Use
def
users = [
Revert Commit...
Reset Current Branch to Commit...
Reset Current Branch to Previous Commit...
AI Rebase Current Branch onto Commit (Preview)...
Rebase Current Branch onto Commit...
Checkout Commit...
3
Create Branch...
Create Patch...
Create Tag...
Explain Changes (Preview)
Open Changes
Inspect Details
Open in Commit Graph
Open Commit on Remote
Reveal Commit in Side Bar
Browse
Compare to/from HEAD
Compare Working Tree to Here
Select for Compare
Share
Copy
Copy SHA
Copy Message
Copy Changes (Patch)
Collapse
Expand
Refresh
GRAPH
fix: Only users
feat: User is now
feat: The numbe
feat: The first us
Initial commit
GITLENS
COMMITS
Up to date
on main
origin
feat: User
feat: The
feat: The
Initial com
2