GitLab on Windows

Last Modified: February 22, 2023

DISCLAIMER: You do **not** have to use Git in the ways described here. You do not have to use Git on Windows, you do not have to use Git Bash, you do not have to use GitHub Desktop. This is *for reference* to help you, in case you do not have experience and/or formed preferences yet. If you do follow this, and you get stuck, send a message to Sara.

In general, for feedback or concerns regarding this document, please e-mail Sara. There are so many git tutorials out there, google is your friend. Big recommendation of the Pro Git book (which Git will recommend to you as well, after downloading). I try to give a step-by-step tutorial here for the most basic things that would be useful for our specific needs with git. If there is something unclear or missing, please drop me a line so that I can improve it.

Git Tree structure

The basic structure of our Git contains a *group*, *subgroups* and *projects*. The following is a simple example for easier understanding:

- Group: PVA
  - Subgroup: Teaching
    - Subgroup: WS-2022-2023
      - Project: ws2223-pva1
      - Project: ws2223-denken
    - Subgroup: SS-2022
      - Project: ss22-appliedmachinelearning

The projects is where the real fun is, i.e. all the relevant documents, source code files, presentations, etc. In the following we refer to projects and repositories interchangeably.

The location (path) of a project within groups and subgroups is reflected in the URL, e.g.:
`https://gitlab.hrz.tu-chemnitz.de/pva1/teaching/ws-2022-2023/ws2223-pva1`.

IMPORTANT NOTE: I am in the process of organizing Git currently. If you need to add something urgently, please get in touch with me (Sara) first.

Terminology

If you want to download a repository to your computer with all its files and the possibility to change them, delete them or add new files - you want to *clone* the repo.

If you already have cloned the repo and you want to locally update the files because someone else made changes - you want to *pull*.

If you have made some changes and want to update the remote repo you need to *add*, *commit* and *push*. If you are curious what that means, keep on reading the list in this paragraph, otherwise skip forward.

- According to Git, files can be *tracked* and *untracked*, i.e. Git either knows about them from before or not, respectively. A tracked file can be *unmodified*, *modified* and *staged*.
- A file that is in the same state in the remote and your personal local repository is *unmodified*.

- Once you edit the file, Git recognizes that and the file becomes *modified*.
- If a file is *staged*, that means that you are ready to update the remote file with your changes. In order to stage a modified file, you need to *add* it.
- On the other hand, if you create a new file which did not exist in the repository, Git will recognize it as an *untracked* file, and in order to stage it, you need to *add* it.
- After you have *added* all of the changes you've made to the staging area, you can *commit* them. With this you record the current version of all files in your project and you are able to revert back to them if needed.
- Done with all changes, staging, commiting, and its finally time to let the world know (i.e. the rest of us in the professorship, we *are* your world) what is the current state of affairs, and you finally *push* your changes to the remote repository. Now everyone else can *pull* them to their local projects.

After we go through the Git installation on Windows, you will see an example of all these commands (and more) in action. You can find a testing repository here, which contains a file called "main_test.txt". It will be used as a running example in this document, and you are also encouraged to use it to test if everything works on your computer.

Downloading Git

Download Git for Windows here and install it.

Git with Git Bash

Search for Git Bash and open it. If you are familiar with Unix, you probably already recognized Bash and you know where we're going at with this. You will see a terminal as in Figure 1.

Figure 1: Git Bash

In the following, I will show you how to navigate around your computer using this terminal, clone a repository, add and modify a file.

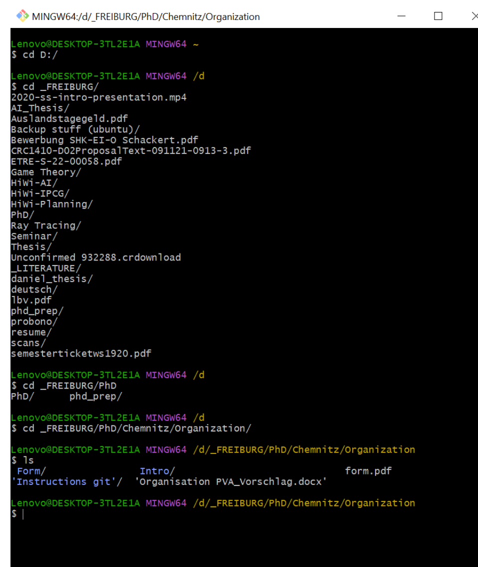
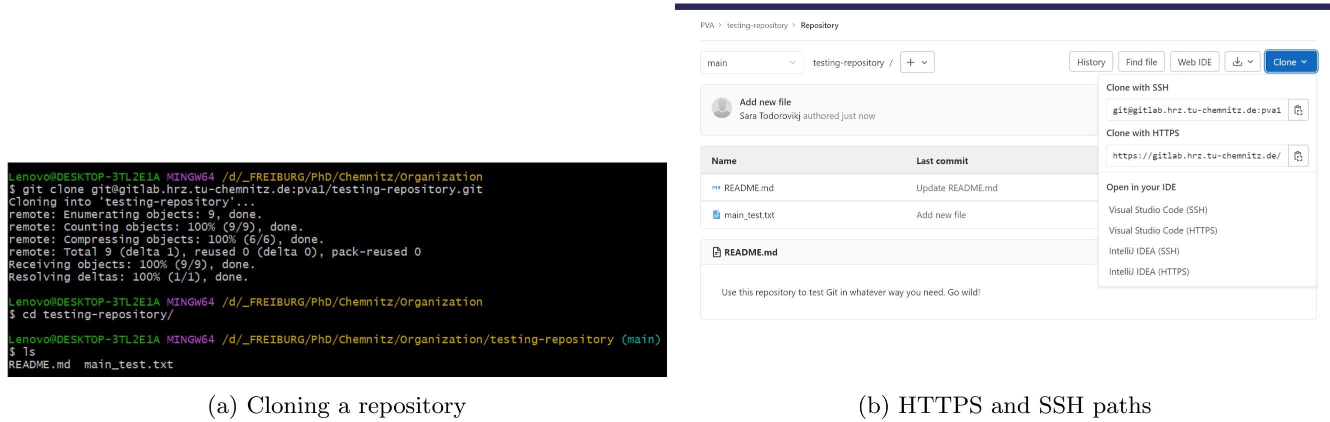


Figure 2: `cd` and `ls`

In order to enter a directory use the command `cd` (*c*hange *d*irectory). To list all files in a directory use the command `ls` (*l*i*s*t). In Figure 2, you can see me entering my D:/ drive. Then I want to navigate to "_FREIBURG/PHD/Chemnitz/Organization". In the second step, when writing out the path, I pressed Tab in order to list all possibilities to continue what I am typing[1]. If there is only one possibility, pressing Tab means auto-complete. Once I entered the Organization folder, I can see all directories and files in it with `ls`. To create a new folder, use the command `mkdir` (*make dir*ectory) (not illustrated here, but you get it).

[1] Me exposing my chaotic D:/ drive like this is a very vulnerable move from my side. Please respect and appreciate that, thank you.

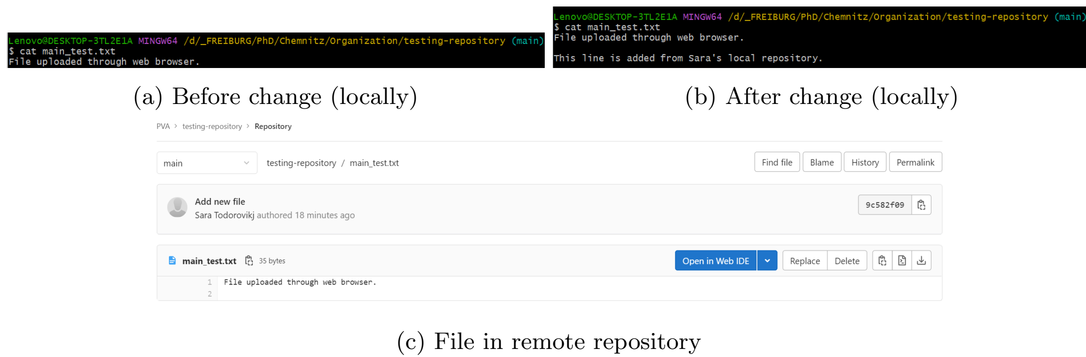
CLONING TIME!![2] Let's clone our testing repository here and enter it. In Figure 3a, you can see how that's done. We use the command `git clone` followed by the URL. Depending on the transfer protocol, you will either use `git clone https://gitlab.hrz.tu-chemnitz.de/pva1/testing-repository.git` (https) or `git clone git@gitlab.hrz.tu-chemnitz.de:pva1/testing-repository.git` (ssh). In the screenshot I used ssh. Both paths can be found on the website, as you can see in Figure 3b. The difference between them is out of the scope for this document, but if you are curious - this is a nice article. Main difference is that with ssh you do not have to constantly enter your credentials, in contrast to https. You will need an SSH Key, but we get to that later. Anyway, back to Fig. 3a. After we cloned the repository, we have a new directory named after the repo. We enter it and list the files in it.



(a) Cloning a repository (b) HTTPS and SSH paths

Figure 3: Cloning, transfer protocols, ...

We can preview the text file in the terminal using the command `cat`[3]. In Fig. 4a we display the file as it is in the remote repository. Then we modify it and we display it again, as seen in Fig. 4b. The rest of the world still doesn't know about these changes, and they only have access to the first version as saved remotely (Fig. 4c).



(a) Before change (locally) (b) After change (locally)

(c) File in remote repository

Figure 4: Modifying a file

[2]... was what Keith Campbell, Ian Wilmut et al. said in 1996. #Dolly
[3]Meow.

But, obviously we believe that sharing is caring, so let's share our beautiful modifications with the rest.

I would like to take your attention to Fig. 5. The first useful command that we see here is `git status`, which gives us the status of the repo, unsurprisingly. It tells us that there are some changes which are not *staged* (remember staging from above?) and that there are no changes added to commit. We want to get rid of this scary, red color, so we use the command `git add` and stage the modified file. Now, we call the status again, and we see an approving green color, yay. You can add multiple files to the staging area before you commit. Next, we will commit our modifications using `git commit -m`, where `-m` is for message. So, everything that is in the staging area will be commited, and we will add a cute, but informative message that describes what kind of changes our commit is making. Then we call the status once again, and it tells us that there is nothing to commit and our working tree is clean, but that we are ahead of the remote repository by 1 commit. You can make multiple commits before pushing. But we are done, so let's publish our changes, so that we are all on the same page, using `git push`. The last status call tells us that we are up to date, and everything is great. Now the file in the remote repository matches our local copy (Fig. 6).

```
Lenovo@DESKTOP-3TL2E1A MINGW64 /d/_FREIBURG/PhD/Chemnitz/Organization/testing-repository (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   main_test.txt

no changes added to commit (use "git add" and/or "git commit -a")

Lenovo@DESKTOP-3TL2E1A MINGW64 /d/_FREIBURG/PhD/Chemnitz/Organization/testing-repository (main)
$ git add main_test.txt

Lenovo@DESKTOP-3TL2E1A MINGW64 /d/_FREIBURG/PhD/Chemnitz/Organization/testing-repository (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   main_test.txt


Lenovo@DESKTOP-3TL2E1A MINGW64 /d/_FREIBURG/PhD/Chemnitz/Organization/testing-repository (main)
$ git commit -m "add sara's modified file"
[main 8b023b6] add sara's modified file
 1 file changed, 2 insertions(+)

Lenovo@DESKTOP-3TL2E1A MINGW64 /d/_FREIBURG/PhD/Chemnitz/Organization/testing-repository (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

Lenovo@DESKTOP-3TL2E1A MINGW64 /d/_FREIBURG/PhD/Chemnitz/Organization/testing-repository (main)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 373 bytes | 186.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To gitlab.hrz.tu-chemnitz.de:pva1/testing-repository.git
   9c582f0..8b023b6  main -> main

Lenovo@DESKTOP-3TL2E1A MINGW64 /d/_FREIBURG/PhD/Chemnitz/Organization/testing-repository (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Figure 5: Git status, add, commit and push

Figure 6: Updated remote file

The principle is the same when you add a new file instead of modifying an already existing one. Add, commit and push.

Note: In general, before you start working on files in repositories, **PLEASE** pull first (using `git pull`), then change and add-commit-push. This is definitely the place where the trait "avoiding conflict" **must** be present.

What is a conflict in this scenario and why do we want to avoid it?
Imagine a file that contains the line "The name of the last person editing this file is: NAME". You and me both have this version of the file in our local repository. Now I work on the file and I change the line it to "The name of the last person editing this file is: Sara". I add, commit and push my changes.
A few hours later you open the repository, you forget to pull, so when you open the file, you see "The name of the last person editing this file is: NAME". You then change it to "The name of the last person editing this file is: MY-NAME". You try to add, commit and push and then git starts complaining about a conflict! Oh no! It knows that "NAME" should be changed to "MY-NAME". But in fact in the remote repository there is no "NAME" anymore... Poor Git.

Now, imagine a big-scale conflict when you and your colleague are working on the same paragraph in the paper and nothing can be combined, sounds stressful, no? Therefore, please **PULL BEFORE MAKING CHANGES!**. If you experience a merge conflict contact Sara or Daniel.

SSH Keys

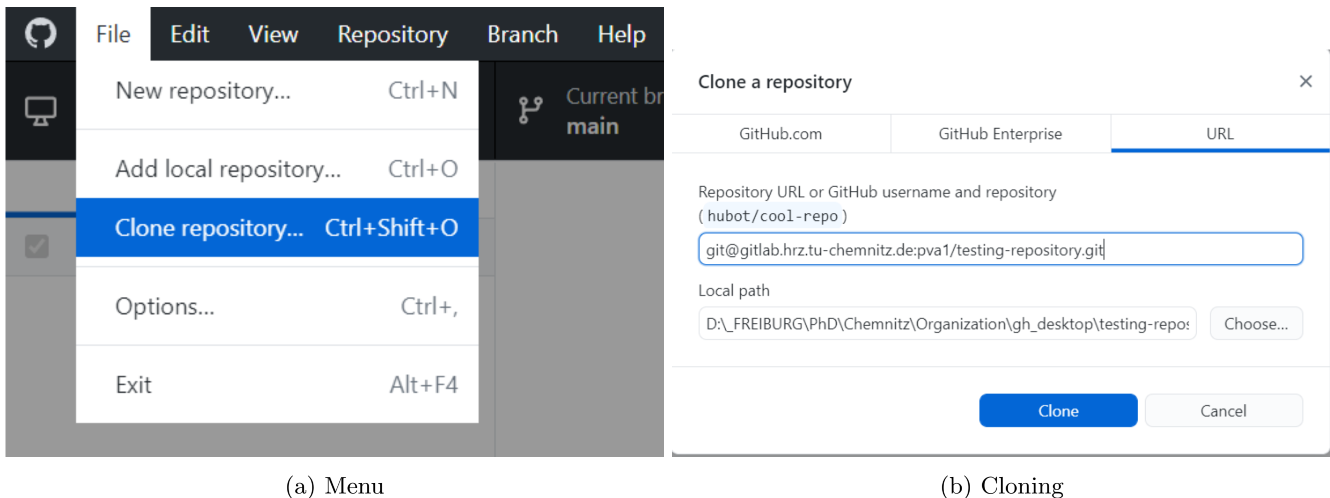
You only need to do this once. Since I have already done this a long time ago, I cannot currently create step-by-step instructions with screenshots. Follow this beautiful tutorial. If unsure, come to me we can do it together.

GitHub Desktop for GitLab

Don't want to deal with terminals? Here is an alternative. Might be a bit "tricky" (GitHub vs. GitLab, long story), but I am providing all necessary steps to make it work.

There *are* other options except for GitHub Desktop, and if you have any kind of preference, please go for it. Download GitHub Desktop here and install it.

Then let's clone the repository. Go to "File > Clone repository..." (Fig. 7a) and you will see a window pop up like in Fig. 7b. Choose "URL", add the URL in the first space (I am using the ssh protocol here again), and then choose where you want it saved locally.



(a) Menu (b) Cloning

Figure 7: Cloning a repo in GitHub Desktop

If you use https, at this point you will get an authentication error message. You will need to enter a username and password. Your username is whatever follows after the "@" when you go to your account page. For reference, see my username in Fig. 8 being "satod–tu-chemnitz.de".

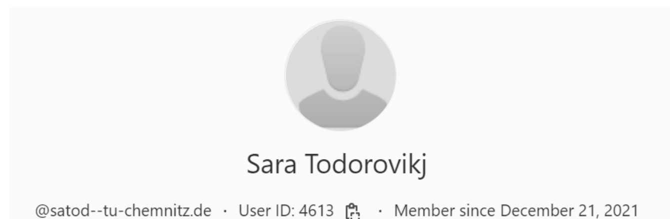
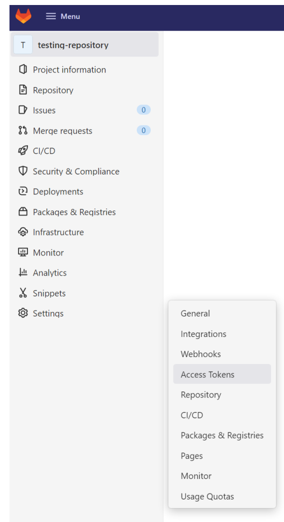


Figure 8: Username

Now, the password. When you have the project opened in your browser, go to "Settings > Access Tokens". Then fill out the form for adding a project access token (name, expiration date, choose at least **maintainer** as a role and select **api**). Then, you will get a random sequence of characters as shown on top of Fig. 9c. That is your password. Et, voila, your repository is cloned.



(a) Username

PVA > testing-repository > Access Tokens

Search page

Project Access Tokens

Generate project access tokens scoped to this project for your applications that need access to the GitLab API.
You can also use project access tokens with Git to authenticate over HTTP(S). Learn more.

Add a project access token

Enter the name of your application, and we'll return a unique project access token.

Token name

testing-token

For example, the application using the token or the purpose of the token. Do not give sensitive information for the name of the token, as it will be visible to all project members.

Expiration date

2022-10-19

Select a role

Maintainer

Select scopes

Scopes set the permission levels granted to the token. Learn more.

api
Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.

read_api
Grants read access to the API, including all groups and projects, the container registry, and the package registry.

read_repository
Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.

write_repository
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).

read_registry
Grants read-only access to container registry images on private projects.

write_registry
Grants write access to container registry images on private projects.

Create project access token

Active project access tokens (0)

This project has no active access tokens.

(b) Cloning

Your new access token has been created.

Search page

Project Access Tokens

Generate project access tokens scoped to this project for your applications that need access to the GitLab API.
You can also use project access tokens with Git to authenticate over HTTP(S). Learn more.

Your new project access token

--LF5vPMnGugcwGk8umE

Make sure you save it - you won't be able to access it again.

Add a project access token

Enter the name of your application, and we'll return a unique project access token.

Token name

For example, the application using the token or the purpose of the token. Do not give sensitive information for the name of the token, as it will be visible to all project members.

Expiration date

2022-10-19

Select a role

Guest

Select scopes

Scopes set the permission levels granted to the token. Learn more.

api
Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.

read_api
Grants read access to the API, including all groups and projects, the container registry, and the package registry.

read_repository
Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.

write_repository
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).

read_registry
Grants read-only access to container registry images on private projects.

write_registry
Grants write access to container registry images on private projects.

Create project access token

Active project access tokens (1)

Token name	Scopes	Created	Last Used	Expires	Role	
testing-token	api	Sep 19, 2022	Never	in 4 weeks	Maintainer	Revoke

(c) Cloning

Figure 9: Password

In Fig. 10 you can see the top left corner of GitHub Desktop after cloning the repo. It tells us that we are currently in "testing-repository", that we are on the branch *main* (branches irrelevant now, just stick with main), and pressing "fetch origin" is basically the equivalent of running `git pull`. It also tells us that there are 0 changed files in the repo, so we are all on the same page. For now.

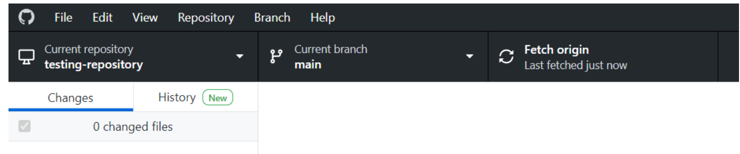
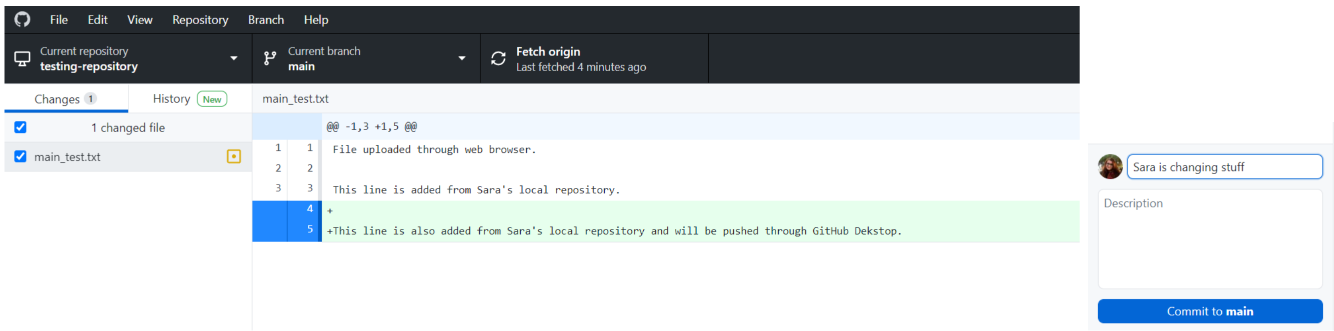
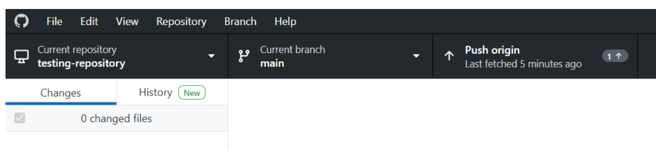


Figure 10: GitHub Desktop

Oh oh, we made a little change and GitHub Desktop found out (Fig. 11a) and is showing us what exactly the change is. By having the checkmark next to the name of the file in the left column, your file is staged (i.e. `git add`). Let's look at the bottom left corner of GitHub Desktop, as shown in Fig. 11b. We can add our little commit message in the top field, and then we press "Commit to main", so we executed `git commit`. What is left to do now is to push. And, oh, what is that in the top left corner? The "Fetch origin" button changed to "Push origin" (Fig. 11c)! Let's press it. And the remote repository is now aware of our changes, as we can see in Fig. 11d.



(a) Changes in file (b) Commiting



(c) Cloning (d) Cloning

Figure 11: Pushing a file with GitHub Desktop

And that's all folks! Happy pushing!