# GitLab HowTo

Informatik, Universität Rostock 10.2021

## Contents

# 1 GIT?

## 1.1 What is GIT?

GIT is a decentral Version Control System for the collaborative work of one or more persons. It stores changes of your files in a history. If you use it correctly, you may trace changes to text files line by line and assign the change to a specific user. If needed, you may restore an older version of your file. GIT stores only the changed lines. Therefore it needs a small amount of memory to store a long history of many files. Commit your changes as small and often as possible. So you will be able to pinpoint a changing, that messes with your code by boing step by step back in the history of changes.

You should put in a GIT repository:

- Source code
- Makefiles
- Configuration needed to build your project.
- Needed files that can't be generated or loaded from the internet or other repositories.
- Documentation with needed graphics.
- Protocoll that documentates a process that can not be reproduced automatically.

## 1.2 What <u>not</u> to store in in GIT

As it comes to binary files (like images, videos, ZIP files, PDF files, JAR packets) things get complicated. This files are not stored in text lines. So GIT has to store the whole file on every change. Even if you simply change one pixel in a big image. For that reason binary files can grow your GIT history quickly.

The following files should not stored in GIT. Especially if the file size is > 5 MB. (Exception: This file is needed to create your output files and there is no other way to create this file.)

- Videos (E.g: `*.avi` ; `*.mp4` ; `*.wmv`)
- Images (E.g: `*.jpg` ; `*.png` ; `*.tiff` ; `*.bmp` ; `*.xcf` ; `*.psd`)
- Presentations (E.g: `*.ppt` ; `*.pptx` ; `*.pdf`)
- Audio files (E.g: `*.wav` ; `*.mp3` ; `*.flac` ; `*.ogg`)
- Executables, libraries, installer (E.g: `*.exe` ; `*.dll` ; `*.lib` ; `*.msi` ; `*.o` ; `*.so` ; `*.class` ; `*.jar`)
- Binary files (E.g: `*.bin` ; `*.dat` ; `*.zip` ; `*.gz` ; `*.bz`)
- Files you can generate with your sources
- Files which are generated during the building process (E.g: `*.o` ; `*.class` ; `*.aux` ; `*.idx`)
- Debug or program output files (E.g: `*.log`, `*.dat`)
- Folders and files from another version management system (E.g: `.svn`)
- Backups, temporary files, local workspace configuration (E.g: `*.swp` ; `*~` ; `tmp` )

Your repository should stay < 300 MB, including history. If you are unsure whether your file belongs in the GIT repository, ask your supervisor.

## 1.3 Configure files managed by GIT

In every folder of your GIT repository you may add a `.gitignore` file. This is a text file, listing all folders or files that should be ignored by GIT when using `git add`. In the list you may use wildcards (*). „Best practice“ is to create the `.gitignore` file only in the main directory of your project.

### 1.3.1 Example: C Project

The project has this folder structure:

- `Makefile`
- `.gitignore`
- `a.out`
- `guessMyNumber.c`
- `guessMyNumber.o`
- `myLibrary.a`
- `myLibrary.c`
- `myLibrary.o`
- `myLibrary.so`

The files `*.c` are source code. This files should be in the repository.

The `*.o` files are files the compiler needs during the build process. The file `a.out` is the generated executable program. The files `myLibrary.a` and `myLibrary.so` are the libraries generated from `myLibrary.c`. These files should not be in the repository.

The `.gitignore` might be like the following:

```
*.a
*.so
*.o
*.out
```

### 1.3.2 Example: Tex Document

- `.gitignore`
- `myPaper.tex`
- `myPaper.pdf`
- `myPaper.aux`
- `myPaper.idx`
- `myPaper.toc`
- `myPaper.out`
- `myPaper.log`
- `chapter1.tex`
- `chapter1.aux`
- `.chapter1.aux.swp`
- `chapter2.tex`
- `chapter2.tex~`
- `images/`
  - `logo.png`
  - `interestingGraphic.pdf`

The `*.tex` files are the TeX source code. The folder `images` contains images needed to create the final document. These files should be in the repository.

`myPaper.pdf` is the generated document. The `*.log`, `*.out`, `*.aux`, `*.toc`, `*.idx` files are generated and needed during the building process. These files should not be in the repository.

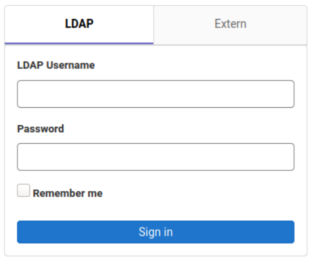
The `*.swp` files are created and used by the text editor (e.g: `chapter1.swp`). The editor stores a backup for each edited file (e.g: `chapter2.tex~`). The backup files and the `*.swp` files should not be in the repository.

The `.gitignore` might be like the following:

```
*.swp
*~
*.aux
*.idx
*.toc
*.out
*.log
```

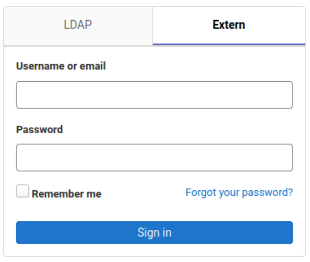
# 2 Login

## 2.1 Login with an account at informatik.uni-rostock.de



1. Navigate with your browser to:
   `https://git.informatik.uni-rostock.de/users/sign_in`
2. Select: `LDAP`
3. Enter the name of your informatik account.
4. Enter the password of your informatik account.
5. Click on `Sign In`.

## 2.2 Login with an external account



1. Navigate with your browser to:
   `https://git.informatik.uni-rostock.de/users/sign_in`
2. Select: `Extern`
3. Enter your e-mail address.
4. Enter your password.
5. Click on `Sign In`.

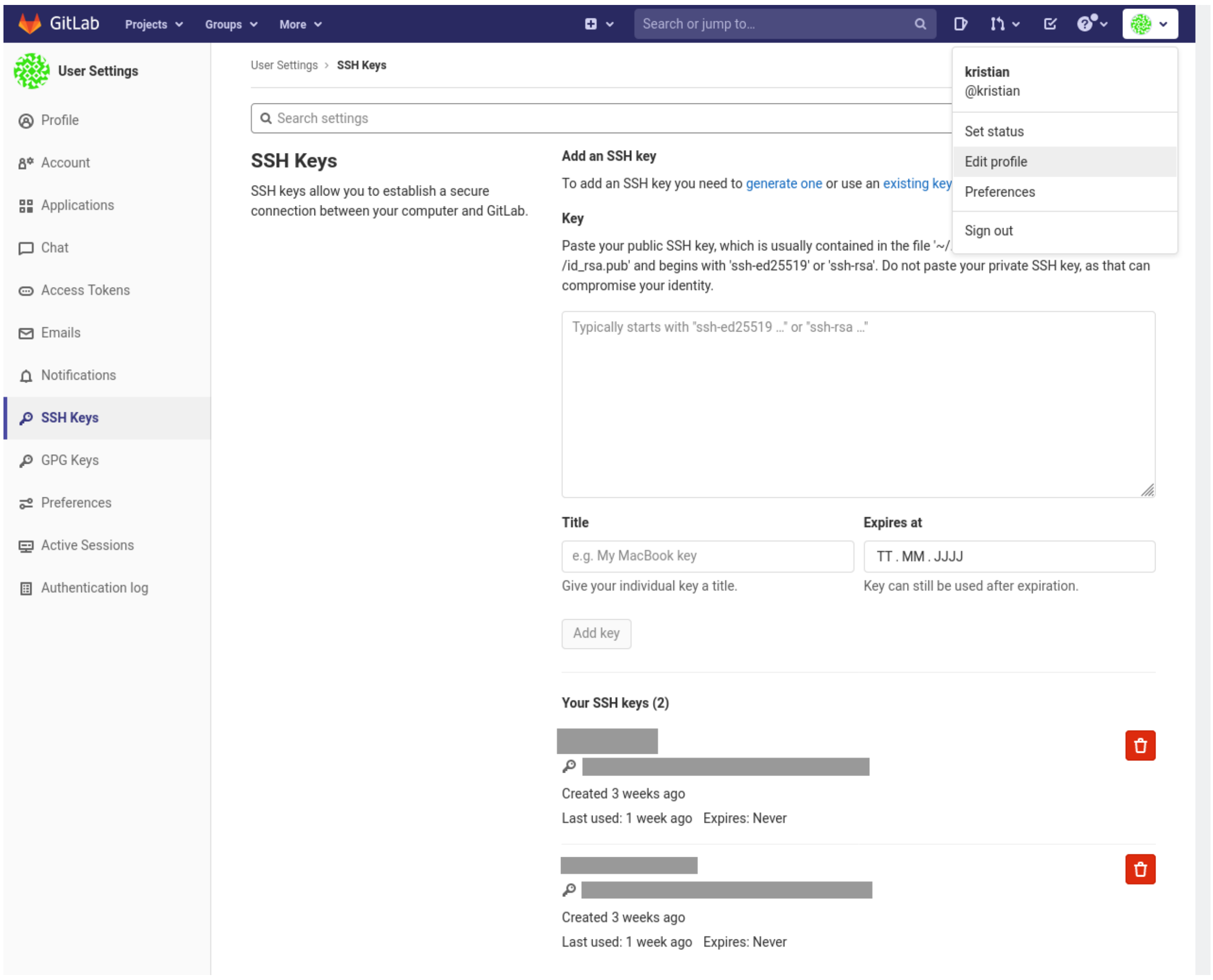
# 3 Set up SSH keys

## 3.1 Introducion

SSH keys are used to authenticate yourself to the GIT server. This are pairs of public and private keys. The public key is send to the GIT server. The private key is kept secret. To be able to lock a single computer or user (e.g. if your computer account was hacked) there should be a pair for every computer and user.

## 3.2 Open the overview page

- Click on your symbol on the top right side.
- Click on `Edit profile` in the appearing menu.
- Click on `SSH Keys` in the menu on the left side.



SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab.

Add an SSH key

To add an SSH key you need to generate one or use an existing key

Key

Paste your public SSH key, which is usually contained in the file '~/. /id_rsa.pub' and begins with 'ssh-ed25519' or 'ssh-rsa'. Do not paste your private SSH key, as that can compromise your identity.

Typically starts with "ssh-ed25519 ..." or "ssh-rsa ..."

Title: e.g. My MacBook key
Give your individual key a title.

Expires at: TT . MM . JJJJ
Key can still be used after expiration.

Add key

Your SSH keys (2)

[redacted]
[redacted]
Created 3 weeks ago
Last used: 1 week ago Expires: Never

[redacted]
[redacted]
Created 3 weeks ago
Last used: 1 week ago Expires: Never

## 3.3 Create an SSH key

This step is needed if you don't have a SSK key on your computer or you want to create a new one for GIT. Otherwise you can skip this step.

- Start a commandline on your computer.
- Run the following command:

  ```
  ssh-keygen -t ecdsa -b 521
  ```

  The Output looks like:

  ```
  [bob@earth ~]$ ssh-keygen -t ecdsa -b 521
  Generating public/private ecdsa key pair.
  Enter file in which to save the key (/home/bob/.ssh/id_ecdsa):
  Enter passphrase (empty for no passphrase):
  Enter same passphrase again:
  Your identification has been saved in /home/bob/.ssh/id_ecdsa
  Your public key has been saved in /home/bob/.ssh/id_ecdsa.pub
  The key fingerprint is:
  SHA256:5vKu...9Jk0 bob@earth
  The key's randomart image is:
  +---[ECDSA 521]---+
  |              .. |
  |             ..  |
  |            o   .|
  |     o . . +..o  |
  |  . = . S +o.+ . |
  | ....+ +   .E o  |
  | +.*..+ o  o + . |
  |. Booo.+ o. o +  |
  | o..oooo+ o=++   |
  +----[SHA256]-----+
  ```

- On the request „`Enter file in which to save the key`“ you may press enter **[Enter]** and accept the default value or enter your own filename.
- On the request „`Enter passphrase`“ you may secure your SSH keys by a password. You have to repeat your input in the line „`Enter same passphrase again`“.

  As you enter the passphrase it will not shown. You will need the passphrase on every transfear of data from or to the GIT server.

## 3.4 Add an SSH key

- You find your SSH keys in your home folder in the folder `.ssh`. The files we need have the extension `.pub`. E.g: `id_ecdsa.pub` or `id_rsa.pub` If you have generated a key in the previous step you will find the name of the needed file in the line: „`Your public key has been saved in ...`“
- Open your public key in a text editor. The content is a single line and should look like this:

  ```
  ecdsa-sha2-nistp521 AAAA...FHw== user@computer
  ```

- Copy your key in the field `Key`.
- Give your key a `Title` so you will remember to what account/computer it belongs.
- Click on `Add key`.

### 3.5 Remove an SSH key

- Find the key you want to delete in the `Your SSH keys` list.
- Click on the trash symbol 🗑, on the right side of the key.
- Confirm the question `Are you sure you want to delete this SSH key?` with a click on `Delete`.

# 4 Create new project

## 4.1 Using the command line

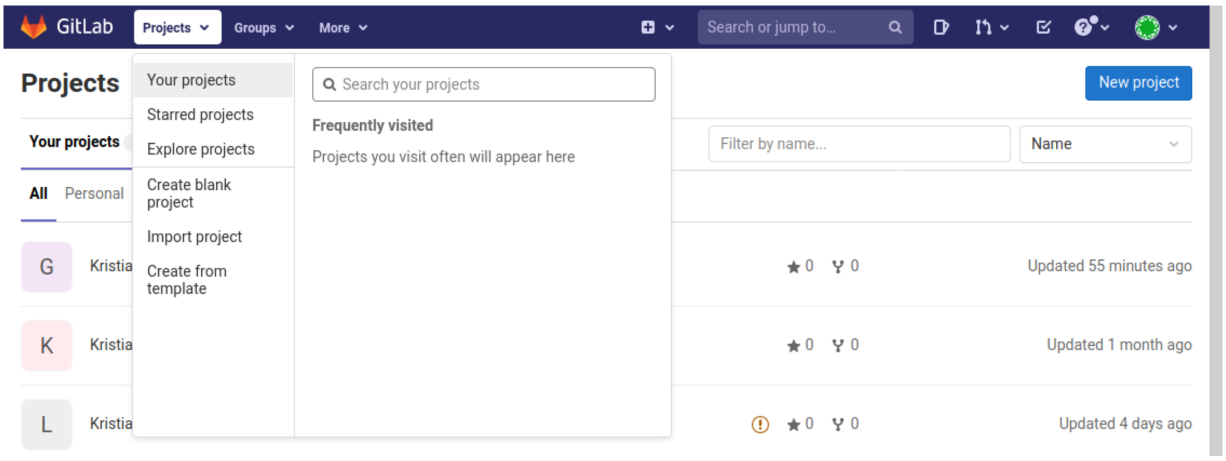
- **You have no local project?**
  Use all steps (1) til (6)
- **You have a local project but it is not managed by GIT?**
  Use steps (2), (4), (5) and (6)
- **You have a local GIT project?**
  Use steps (2) and (6)

The example uses the account **myAccount** and the project name **myProject**. The parts printed in **green** have to be replaced by values according to your account / project.

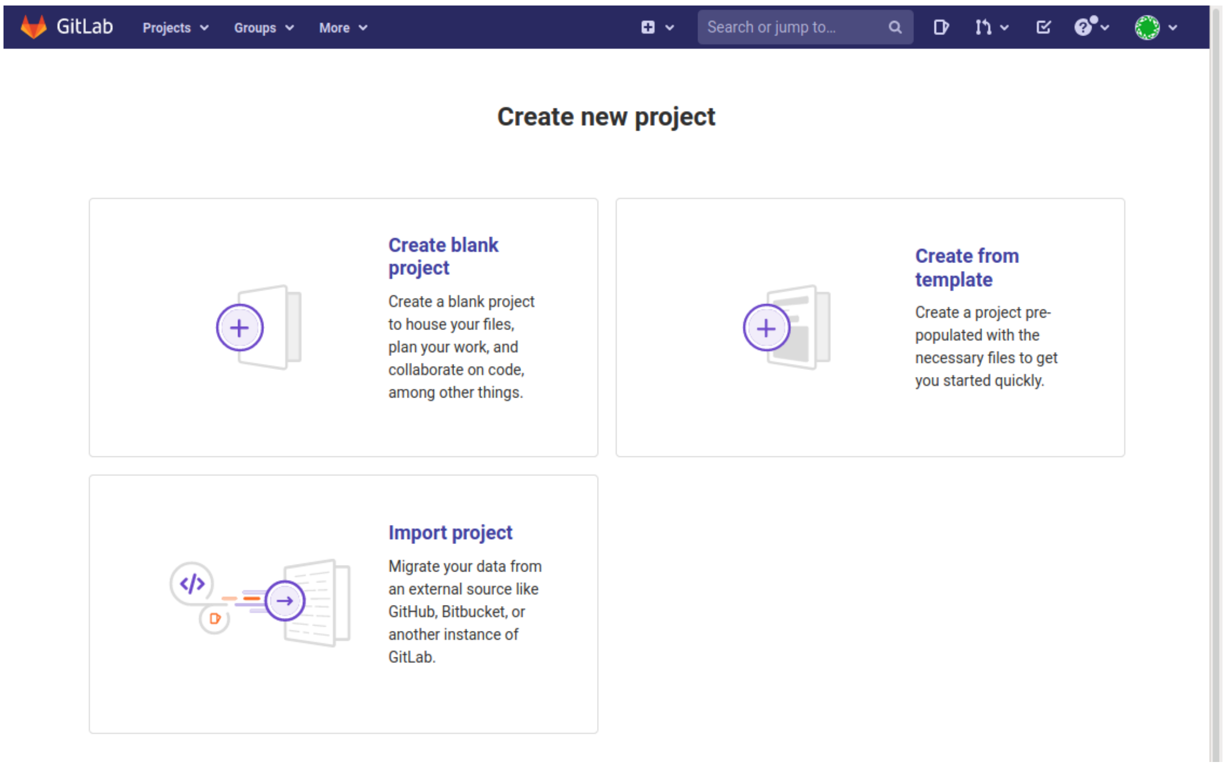
1. Create a folder for the new project:
   ```
   mkdir myProject
   ```
2. Enter the folder for the new project:
   ```
   cd myProject
   ```
3. Create some files:
   ```
   echo 'My Project' > README.md
   ```
4. Initialize GIT.
   ```
   git init
   ```
5. Add the files to the new GIT Repository:
   ```
   git add .
   ```
   ```
   git commit -m ¨Initial commit.¨
   ```
6. Send new project to server.
   ```
   git push -set-upstream git@git.informatik.uni-rostock.de:myAccount/myProject master
   ```

## 4.2 With GITLab examples

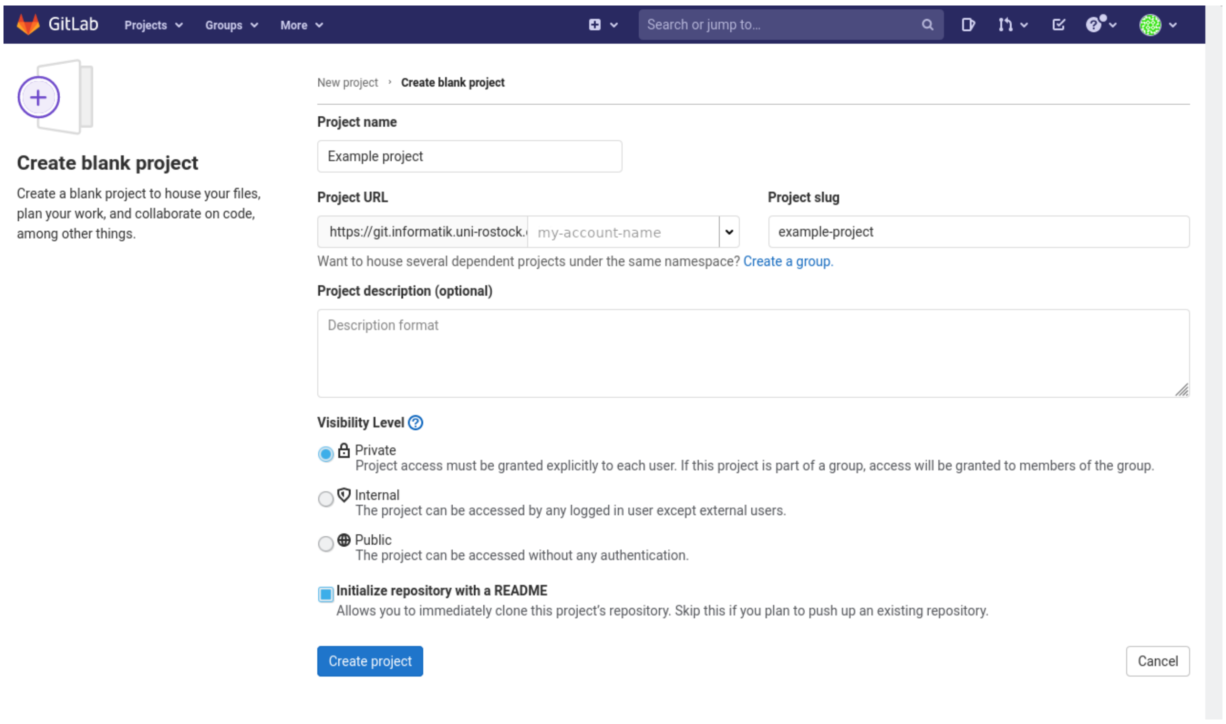
- Click on `Projects` on the top left side.
  A menu appears.



- Click on `your projects` in the appearing menu.
  The list of your projects appears.
- Click on `new project` on the right side.
  The selection page for a new project appears.


GitLab
Projects
Groups
More
Search or jump to...
Create new project
Create blank project
Create a blank project to house your files, plan your work, and collaborate on code, among other things.
Create from template
Create a project pre-populated with the necessary files to get you started quickly.
Import project
Migrate your data from an external source like GitHub, Bitbucket, or another instance of GitLab.

- Click on `Create blank project`.



**Create blank project**

Create a blank project to house your files, plan your work, and collaborate on code, among other things.

Project name: Example project

Project URL: https://git.informatik.uni-rostock. my-account-name

Project slug: example-project

Want to house several dependent projects under the same namespace? Create a group.

Project description (optional): Description format

Visibility Level

- [✓] Private — Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.
- [ ] Internal — The project can be accessed by any logged in user except external users.
- [ ] Public — The project can be accessed without any authentication.

[✓] Initialize repository with a README — Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Create project | Cancel

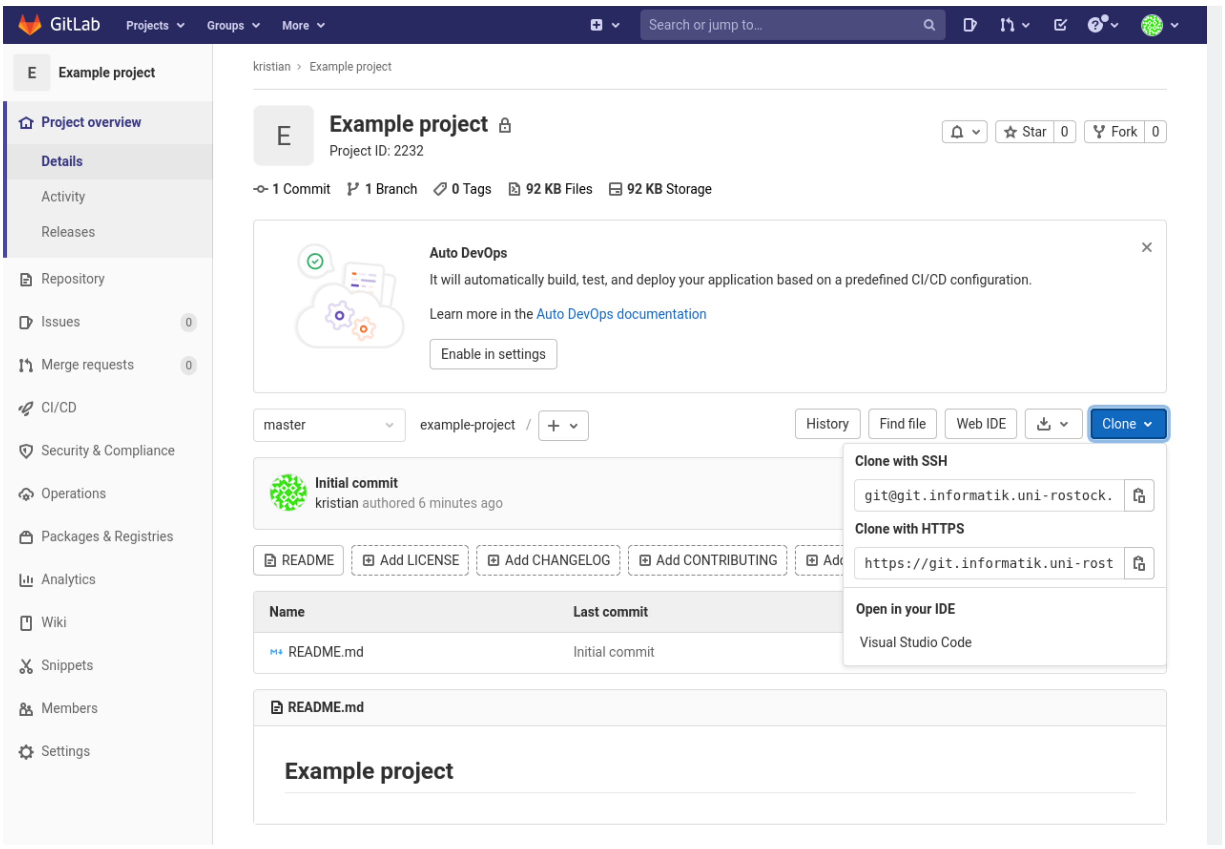
- Enter a name for your project in the box `Project name`.
- You may add a short description for your project in the box `Project description`.
- Activate the checkbox `Initialize repository with a README`, so you have a non-empty repository and may check it out to your computer.
- Click on `Create project`.

  The overview of your new project appears.

  Next step is cloning your repository. (Chapter 5)

# 5 Clone a repository

- Open your project on https://git.informatik.uni-rostock.de.
- Click on `Clone` and copy the link below `Clone with SSH`.



It will automatically build, test, and deploy your application based on a predefined CI/CD configuration.

Learn more in the Auto DevOps documentation

- Open a terminal.
- Navigate to the folder where you want to store your repository.

```
cd my/favorite/repo/storage/directory
```

- In the command below replace **git@git.informatik.uni-rostock.de:kristian/example-project.git** with the link you copied before. Execute the command.

```
git clone git@git.informatik.uni-rostock.de:example-user/example-project.git
```

If you are asked for a login then your SSH-key is missing in GITLab. Please follow the actions in Chapter 3 „Set up SSH keys“.

# 6 Working with the repository

## 6.1 Workflow

During the work with a GIT repository usually one follows a workflow like this.

1. Load the newest changes from server. `fetch / pull`
2. If needed: change or crate a branch. `checkout / branch`
3. Add, change or remove files. `add / remove`
4. Commit changes: Add a meaningfull comment. `commit`
5. Send newest commits to server. `push`

## 6.2 Inspecting changes

- `git status`
  Shows the local changes compared to the newest commit in the current branch.
  Example:

```
[bob@earth ~/myProject] (master)$ git status
On branch master
Your branch is up to date with ´origin/master´.

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.c

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

- `git log`
  Shows a list with the commits.
  Example:

```
[bob@earth ~/myProject] (master)$ git log
commit 1c5e59568e86377b620614f9e48250b4181fe9ab
Author:  MySelf <my-self@my-server.de>
Date:  Tue May 4 17:10:13 2021 +0200

        Added Ackermann function.

commit 899efba3143f3c24fa9168c803f2caf5d4fe3974
Author:  MySelf <my-self@my-server.de>
Date:  Tue May 4 13:13:46 2021 +0200

        First commit.
```

## 6.3 Update the working directory.

- `git fetch`
  Loads the newest version of the repository from server but leaves the working directory as it is.
- `git pull`
  Loads the newest version of the repository from server and updates the working directory.
- `git checkout` **NAME**
  Updates your working copy according to **NAME**.
  - If **NAME** is a name of a file in the repository, then this file will be replaced by the version in the newest commit.
  - If **NAME** is the ID of a commit, then all files will be replaced by their version in this commit. In the example from `git log` from above:

    ```
    git checkout 899efba3143f3c24fa9168c803f2caf5d4fe3974
    ```
    Returns to the state of the commit with the title `First commit.`
  - If `NAME` is a name of a branch, then all files will be replaced by their version in this branch. All following commits will be added to this branch.

## 6.4 Working branch

You should use branches, if you develop a new feature or plan to do big changes. If you are done with your changes and tested it successfully, you may merge it with your master branch.

- `git branch`
  Shows the current and the available branches on this computer.
- `git branch` **myNewFeature**
  Creates a new branch with the name **myNewFeature**. It contains a copy of all data from the current branch.
- `git checkout` **myOtherNewFeature**
  Changes to the branch with the name **myOtherNewFeature**.
- `git merge` **myNewFeature**
  Merges the branch **myNewFeature** in the current branch.

Example: The current branch is `master`. The new file `example.c` is to be created and the file `main.c` needs to be edited.

```
[bob@earth ~/myProject] (master)$ git pull
[bob@earth ~/myProject] (master)$ git branch example
[bob@earth ~/myProject] (example)$ vim example.c
[bob@earth ~/myProject] (example)$ git add example.c
[bob@earth ~/myProject] (example)$ vim main.c
[bob@earth ~/myProject] (example)$ git add main.c
[bob@earth ~/myProject] (example)$ git commit -m ¨Added new example function.¨
[bob@earth ~/myProject] (example)$ git push

[bob@earth ~/myProject] (example)$ git checkout master
[bob@earth ~/myProject] (master)$ git pull
[bob@earth ~/myProject] (master)$ git merge example
[bob@earth ~/myProject] (master)$ git push
```

## 6.5 Changing things

- `git add FILE_NAME`
  Adds the new or changed file **FILE_NAME** to the next commit.
- `git add .`
  Adds all new or changed files in the current folder and all subfolder to the next commit.
- `git rm FILE_NAME`
  Removes the file **FILE_NAME** from the working copy and marks it as deleted for the next commit.
- `git commit`
  Saves the marked changes as a commit. It shows a text editor. In this editor you have to enter a comment for this commit.

## 6.6 Sending changes to the server

After you have committed your changes, you should send your commits to the server:

```
git push
```

## 6.7 Removing a file out of the history

- Be sure that there is no open merge request.
- This command removes the file from your local history:
  ```
  git filter-branch -force \
       -index-filter ¨git rm -cached -ignore-unmatch FILE_NAME¨ \
       -prune-empty -tag-name-filter cat - -all
  ```
- Add the file to your `.gitignore` file.
  ```
  echo ¨FILE_NAME¨ >> .gitignore
  ```
  ```
  git commit -m ¨Add FILE_NAME to .gitignore¨
  ```
- This command sends the changed History to the server.
  ```
  git push origin -force -all
  ```
- Tell your collaborators to rebase, not merge, any branches they created off of your old (tainted) repository history. One merge commit could reintroduce some or all of the tainted history that you just went to the trouble of purging.
  Details on topic „rebase“: https://git-scm.com/book/en/Git-Branching-Rebasing
- Additional information on the topic „remove data from the history“ you will find in:
  https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/removing-sensitive-data-from-a-repository

# 7 Contact

If you have questions or suggestions please send a mail to: stg-cs@uni-rostock.de