

# Tools SS 2022

## Introduction to Git

Florian Beenen



08.06.2022



# Contents I

## 1. Principles

- 1.1. What is Git
- 1.2. Commits
- 1.3. References
- 1.4. Creating New Commits

## 2. Commands

- 2.1. Initializing a Repository
- 2.2. Inspecting the Repository
- 2.3. Creating New Commits
- 2.4. Branches
- 2.5. Emergency Help

## 3. Work with Remotes

- 3.1. Setting up Remotes
- 3.2. Synchronizing the Repository
- 3.3. Automated GitLab Jobs

## 4. Exercise



# Principles



# What is Git

- ▶ Software to keep track of **file versions**.
- ▶ Other related software: Subversion (SVN) or Mercurial.
- ▶ Git can be used for managing...
  - ▶ simple programming exercises (for lectures PCA, PAD, HPI, DHD, ...).
  - ▶ Bachelor/Master projects.
  - ▶ text files /  $\text{\LaTeX}$  documents.
  - ▶ personal notes.
  - ▶ ...
- ▶ Git can deal with any kind of file – it works best for text-based files where partial changes occur over time.





# Why use Git?

Git is useful because...

- ▶ you can **document** changes to files.
  - ▶ Who did what, why and when?
  - ▶ Time management: What did I do today?
  - ▶ If you have an abusive boss: Prove that you worked on your assigned project 😊.
- ▶ you can try out new stuff without breaking/affecting existing project versions.
- ▶ you cannot *accidentally* lose data when using `sudo rm -rf` too aggressively.
- ▶ you can track errors in e. g. a program back in time to find the root cause.
- ▶ it enables you to work together with multiple people on the same project.
- ▶ you can use Git as a foundation to do awesome automation tricks (e. g. automated building, testing, deploying, ...).





























# Git Terms

- ▶ A project that is managed with Git is called **Repository**.
  - ▶ A Repository is saved on your computer.
  - ▶ You can share a Repository with other people/computers to collaborate.
- ▶ Data points where file versions are saved are called **Commits**.
  - ▶ You can access old Commits.
  - ▶ You can compare two different Commits and find out what has changed.



# Git Commits

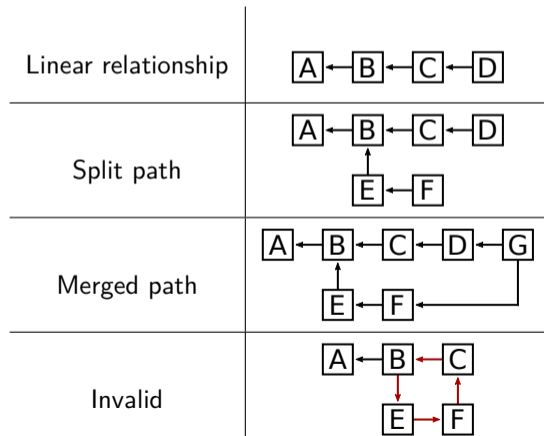
- ▶ Every Git Commit has some meta data assigned:
  - ▶ Name of the user who issued the Commit.
  - ▶ Time when the Commit was issued.
  - ▶ Description of the Commit: Has to be created by the user in a **meaningful** way.
  - ▶ A reference to the previous Commit(s).
- ▶ Git Commits are identified by a name which is formed by a SHA-1 hash (e. g. `64efe35c4535b472d2fe38a6f696adc56dc38ff4`).
  - ▶ Meant to be unique for all Commits on the world.
  - ▶ Collision attacks are possible (but unlikely).
  - ▶ The future is now: <https://www.linux-magazin.de/news/git-2-29-unterstuetzt-sha-256/> (SHA-256 in Git).

30 Nov, 2020 6 commits			
	Erste Folien zu SSH gemacht Florian Beenen authored 1 month ago		51517415  
	CI Fix3? Florian Beenen authored 1 month ago		f37355ec  
	Fix CI and Deploy Florian Beenen authored 1 month ago		43b66d1d  
	CI fix2 Florian Beenen authored 1 month ago		bc060381  
	CI fix? Florian Beenen authored 1 month ago		29dd0372  
	Erste Folien fuer Homeoffice Vortrag Florian Beenen authored 1 month ago		cdd5e1af  
29 Nov, 2020 1 commit			
	Initial commit Florian Beenen authored 1 month ago		ed6accfd  



# Commit Graph

- ▶ Every Commit has at least one predecessor (except for the first Commit).
- ▶ Commits cannot have circular dependency (graph is acyclic and directed).

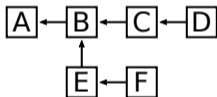






# References

- ▶ If one Commit has multiple children, it is a **merged Commit**.
- ▶ From one Commit only the parent Commits are available:

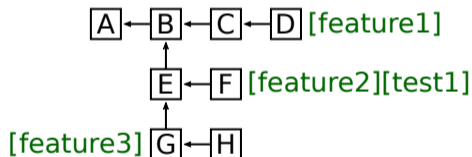


- ▶ From Commit E the history is  $A \leftarrow B \leftarrow E$ .
- ▶ All other Commits are not reachable.
- ▶ The Repository therefore needs to store where the “newest” Commits are.



# Branch

- ▶ A **Branch** is the most common form for a reference to a Commit.
- ▶ A Branch gives a name to a Commit.
- ▶ Multiple Branches may point to the same Commit.
- ▶ Branches do not necessarily need to point to the newest Commits.

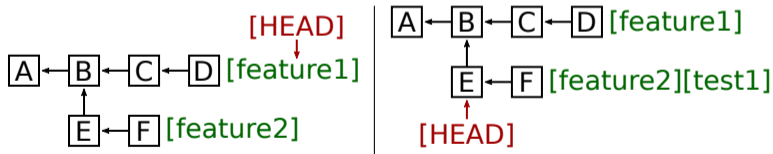


- ▶ In this graph, Commit H is not referenced by any Branch and therefore inaccessible.
- ▶ If you know the name of the Commit (the hash) you can still create a Branch for it.
- ▶ Inaccessible Commits will be purged by the garbage collector eventually.



# Special Reference: HEAD

- ▶ There is a special reference called **HEAD**.
- ▶ It is used to point to the current “active” Branch.
- ▶ The active Branch is the one you are currently working on (➡ where you create Commits).
- ▶ Normally, **HEAD** is a reference to a Branch (not a Commit).
- ▶ You can (temporarily) set **HEAD** to a Commit by specifying its hash: **Detached HEAD**.





# Project State

- ▶ There are three places where file changes are recorded:
  - ▶ The latest Commit – the Branch that is referenced by `HEAD`.
  - ▶ The current working directory: You may have edited some files (this has nothing to do with Git).
  - ▶ The **Index**: A staging area where files are collected that should be included in the next Commit.
- ▶ The Repository is regarded “clean” if all three data sets are identical.
- ▶ What you have to do in order to create a new Commit:
  1. Edit your files – do the actual work on the project.
  2. Mark files that you want to commit. Add them to the Index.
  3. After all files are marked / the Index is updated correctly, issue a Commit and specify the corresponding message.



# The Index

- ▶ Why do we need an Index?
- ▶ This sounds like unnecessary bureaucracy – we could just directly put all changed files into one Commit automatically!
- ▶ Advantages of separate Index:
  - ▶ Split file changes in different Commits.
  - ▶ Optionally, do not commit file changes in specific files at all (you tried some improvised hack on your machine ☺).
  - ▶ Splitting is useful to separate changes in a logical way:
    - ▶ I fixed a security bug in the server application. ➡ One Commit.
    - ▶ I re-wrote a function somewhere else. ➡ Another Commit.
  - ▶ Commit messages are more meaningful, it is easier to understand what has happened when looking at the Commit graph.



# Commands



# A Summary of Git Usage



<https://xkcd.com/1597/>



# Fundamentals

- ▶ To use Git you must install Git to your system.
- ▶ Install with `sudo apt install git` on Debian-like Linux distributions.
- ▶ It is a nightmare to use Git with Windows! I recommend to use *Windows Subsystem for Linux* (WSL) and then do everything “the Linux way”.
- ▶ You manage everything Git-related with the `git` command.
- ▶ You can get help here: <https://git-scm.com/docs>.
- ▶ You can also search the man page: `man git <subcommand>`.
- ▶ When using Git for the first time:
  - ▶ Set your (real) name: `git config --global user.name <first name> <last name>`.
  - ▶ Set your mail address: `git config --global user.email <e-mail address>`.
  - ▶ This data is included in every Commit! Please set it to something meaningful.
  - ▶ Enable colorful output: `git config --global color.ui auto`.
  - ▶ Set the preferred text editor: `git config --global core.editor <editor name>` (e.g. `nano`, `vim` or `gedit`).





## Create a Git Repository

- ▶ Create a fresh Repository with `git init`.
- ▶ If a Repository already existed, nothing is overwritten: It is safe to run `git init` on an existing Repository.
- ▶ The command will create the hidden directory `.git` inside your current working directory.
- ▶ The `.git` directory contains everything that Git needs to operate (all the Commits, meta data, Branches, ...).
- ▶ You should not delete / move / edit the `.git` directory.

```
florian@Florian-Laptop:/tmp/git-test$ git init
Initialized empty Git repository in /tmp/git-test/.git/
florian@Florian-Laptop:/tmp/git-test$ git init
Reinitialized existing Git repository in /tmp/git-test/.git/
florian@Florian-Laptop:/tmp/git-test$ ls -al
total 0
drwxr-xr-x 1 florian florian 512 Jan 18 17:37 .
drwxrwxrwt 1 root    root    512 Jan 18 17:37 ..
drwxr-xr-x 1 florian florian 512 Jan 18 17:37 .git
florian@Florian-Laptop:/tmp/git-test$
```



# Clone a Git Repository

- ▶ If you already have a Git Repository you can download / clone it.
- ▶ Use `git clone <url>`.
- ▶ Example: `git clone https://sus.ziti.uni-heidelberg.de/Lehre/WS1718_Tools/GIT/uebung.git`.
- ▶ Creates a new directory with the name of the cloned Repository.
- ▶ Give it a custom name with `git clone <url> <local-repo-name>`.
- ▶ You can either clone with HTTP(s) or with SSH (➡ preferred).
- ▶ Cloning and syncing with remote Repositories is discussed later!

```
florian@Florian-Laptop:/tmp$ git clone https://sus.ziti.uni-heidelberg.de/Lehre/WS1718_Tools/GIT/uebung.git
Cloning into 'uebung'...
florian@Florian-Laptop:/tmp$ ls -l uebung/
total 4
-rw-r--r-- 1 florian florian 61 Jan 18 17:46 generate_and_plot.sh
-rw-r--r-- 1 florian florian 512 Jan 18 17:46 generate_data.py
-rw-r--r-- 1 florian florian 415 Jan 18 17:46 plot_data.plt
florian@Florian-Laptop:/tmp$
```



# Check Repository State

- ▶ Inspect the working set with `git status`.
  - ▶ Lists files on the Index (planned for next Commit).
  - ▶ Lists changed files that are not on the Index.
  - ▶ Lists deleted files that are still present in the Repository.
- ▶ Check the current Branch: `git branch`.
  - ▶ There is always the `master` Branch.
  - ▶ You can see which Branch is currently active.
- ▶ Check Commit history with `git log`.
  - ▶ See all Commits of the current Branch.
  - ▶ Check out where the HEAD reference is pointing.

```
florian@Florian-Laptop:/tmp/uebung$ git log
commit c034a3c5fb068f02ef62edbf994baac5c589bac0 (HEAD -> master, origin/master, origin/HEAD)
Author: Michael Krieger <michael.krieger@ziti.uni-heidelberg.de>
Date:   Wed Oct 26 18:10:58 2016 +0200

    make the data points black and the fit curve red
```



# Inspecting a Commit

- ▶ Use `git show <Commit>` to inspect a specific Commit.
  - ▶ Can be given a Commit hash, a Branch name or another reference.
  - ▶ Shows the files that have changed.
  - ▶ Shows individual lines that have been added or deleted.
  - ▶ Show changes relative to previous Commit.
  - ▶ Changes are displayed properly only for text-based files (e. g. program sources,  $\text{\LaTeX}$  files, etc.)
- ▶ Differences between Commits: `git diff <Commit1> <Commit2>`.
  - ▶ Useful to manually compare two Commits.
  - ▶ Also get information on every changed line.

```
florian@florian-Laptop:~/tmp/uebung$ git diff c034a3c5fb068f02ef02edb94baac5c589bac0 0c0afe4eb89c4273bdc2c11d2327c4b5276bb8e7
diff --git a/generate_and_plot.sh b/generate_and_plot.sh
deleted file mode 100644
index f0e501..0000000
--- a/generate_and_plot.sh
+++ /dev/null
@@ -1,2 +0,0 @@
python generate_data.py > sine_wave.dat
gnuplot plot_data.plt
diff --git a/generate_data.py b/generate_data.py
index f39d2d2..bf492c3 100644
--- a/generate_data.py
+++ b/generate_data.py
@@ -1,17 +1,14 @@
from math import sin, pi
from random import gauss, uniform
from random import gauss
from numpy import linspace

def noisy_sine_wave(x, period, rel_phase, noise=0.1):
    return sin((x / period + rel_phase) * 2 * pi) + gauss(mu=0, sigma=noise)
def noisy_sine_wave(x, period, phase, noise=0.1):
    return sin((x + phase) / period * 2 * pi) + gauss(mu=0, sigma=noise)
```



# Working with the Index

- ▶ In order to create a new Commit, you need to populate the Index.
- ▶ Add new files to the Index with `git add <file>`.
  - ▶ You can use *Glob Expressions* like `git add *.tex` to add all `.tex` files in the current directory to the Index.
  - ▶ You can use `git add -u` to add all changed files to the Index.
    - ▶ If you edited a file, it will now be on the Index.
    - ▶ If you deleted a file manually, the file deletion will be on the Index.
    - ▶ If you created a new file that is not in the Repository, it will **not** be on the Index. Perform `git add <file>` separately.
- ▶ To check which files are on the Index use `git status`.
- ▶ To remove a file from the Index again use `git reset <file>`.
- ▶ To delete a file from disk and put the deletion on the Index use `git rm <file>`.
  - ▶ If you only want to delete the file from the Repository but not from disk use `git rm --cached <file>`.



# Ignoring Files

- ▶ You can configure your environment that you can perform `git add *` and not add any garbage.
- ▶ You may want to exclude certain directories or file types from ever getting in the Repository.
  - ▶ Binary files that you can generate automatically (e. g. compiled programs, FPGA bitfiles, compiled  $\text{\LaTeX}$  files, ...).
  - ▶ Log files that are useless.
  - ▶ Local configuration files that are bound to your computer but are useless for other people (e. g. Eclipse workspace config, ...)
- ▶ You can write a `.gitignore` file (the dot is important).
  - ▶ List of files and directories that are ignored by Git.
  - ▶ Can contain *Glob Expressions*.
    - ▶ Ignore every `.txt` file: `*.txt`.
    - ▶ Ignore log files only in Repository root: `/*.log`.
    - ▶ Ignore all directories named "build": `build/`.
- ▶ Check with `git status` that it does not report useless "untracked files". Populate your `.gitignore` until `git status` does not report any false positives any more.



# Ignoring Files (2)

```
Florian@Florian-Laptop:~/Vorlesungen/Tools/git$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .gitignore

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:   .gitignore
    modified:   git-einfuehrung/00-folien.tex

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    git-einfuehrung/build/
    git-einfuehrung/img/git-clone.png
    git-einfuehrung/img/git-diff.png
    git-einfuehrung/img/git-init.png
    git-einfuehrung/img/git-log.png
    git-einfuehrung/img/git-xkcd.png
    homeoffice-tools/build/
    homeoffice-tools/routing-tabelle.xlsx
    linux/build/

Florian@Florian-Laptop:~/Vorlesungen/Tools/git$ echo "build/" >> .gitignore
Florian@Florian-Laptop:~/Vorlesungen/Tools/git$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .gitignore

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:   .gitignore
    modified:   git-einfuehrung/00-folien.tex

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    git-einfuehrung/img/git-clone.png
    git-einfuehrung/img/git-diff.png
    git-einfuehrung/img/git-init.png
    git-einfuehrung/img/git-log.png
    git-einfuehrung/img/git-xkcd.png
    homeoffice-tools/routing-tabelle.xlsx

Florian@Florian-Laptop:~/Vorlesungen/Tools/git$ _
```



# Creating a Commit

- ▶ When you are satisfied with the status of the Index, you can create the Commit.
- ▶ Commit the Index with `git commit`.
- ▶ The configured text editor will open and you need to enter a Commit message.
- ▶ Make it a useful message that describes what exactly you did (e. g. **not** "Changes of today").
- ▶ You can also directly specify the message without text editor:  
`git commit -m "my message"`.

```
florian@Florian-Laptop:/tmp/uebung$ git commit -m "Fixed buffer overflow in client listener."  
[master 621b6ee] Fixed buffer overflow in client listener.  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 myfile.txt
```





# Branches

- ▶ Inspect current Branch with `git branch`.
- ▶ Create new Branch with `git branch <name> [<commit>]`.
  - ▶ Creates the Branch with the given name.
  - ▶ Optionally creates the Branch based on the given Commit hash.
- ▶ Change the Branch with `git checkout <branch>`.
  - ▶ Potentially changes the content of files managed by Git.
  - ▶ Will not touch untracked files.

```
florian@Florian-Laptop:/tmp/uebung$ git branch
* master
florian@Florian-Laptop:/tmp/uebung$ git branch testbranch 621b6ee172d026bb16794e518afe671b8f957ab4
florian@Florian-Laptop:/tmp/uebung$ git branch
* master
  testbranch
florian@Florian-Laptop:/tmp/uebung$ git checkout testbranch
Switched to branch 'testbranch'
florian@Florian-Laptop:/tmp/uebung$ git branch
  master
* testbranch
florian@Florian-Laptop:/tmp/uebung$
```



# Deleting Branches

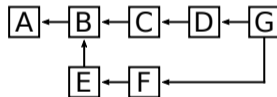
- ▶ Delete a Branch with `git branch -d <branch>`.
- ▶ If Git complains, there are Commits on that Branch that are not otherwise referenced.
- ▶ If you delete the Branch, you will lose data!
- ▶ Either merge the Branch somewhere else before deleting.
- ▶ Or forcefully delete the Branch with `git branch -D <branch>`. You have been warned.

```
florian@Florian-Laptop:/tmp/uebung$ git branch -d testbranch
error: The branch 'testbranch' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testbranch'.
florian@Florian-Laptop:/tmp/uebung$
```



# Merging Branches

- ▶ You normally keep your “golden” working version on the `master` Branch.
- ▶ It is good practice to never publish stuff on `master` that is obviously broken/work in process.
- ▶ Develop your stuff on an extra Branch. Test it extensively. **Then** merge it to `master` and delete your development Branch.
- ▶ To merge a Branch:
  1. Switch to the **target** Branch.
  2. Run `git merge <other-branch>`.
  3. Potentially resolve conflicts.
    - ▶ `git status` will tell you what is going on.
    - ▶ Manually go through the files and decide what shall be used.
    - ▶ Commit your changes.



```
Florian@Florian-Laptop:/tmp/uebung$ git merge testbranch
CONFLICT (modify/delete): myfile.txt deleted in testbranch and modified in HEAD. Version HEAD of myfile.txt left in tree.
Automatic merge failed; fix conflicts and then commit the result.
```



## Merging Branches (2)

- ▶ If you can merge directly, no conflicts exist.
- ▶ This is the case if the set of files modified on the different Branches is disjoint.
- ▶ Special case: The target Branch did not diverge: *fast-forward* (no separate merge Commit exists).
- ▶ Otherwise, Git does not know what to do. You need to inspect all offending files.

### Branch `master`

```
florian@Florian-Laptop: /tmp/uebung
GNU nano 4.8
var=12
onlyOnMaster=123
othervar="Hello"
thirdVar=1
```

### Branch `test`

```
florian@Florian-Laptop: /tmp/uebung
GNU nano 4.8
var=123
othervar="Hello"
thirdVar=1
onlyOnOtherBranch=32
```

### Merge

```
florian@Florian-Laptop: /tmp/uebung
GNU nano 4.8
<<<<<< HEAD
var=12
onlyOnMaster=123
*****
var=123
>>>>>> test
othervar="Hello"
thirdVar=1
onlyOnOtherBranch=32
```

- ▶ When you are done, `git add` the changed files.
- ▶ Commit your changes.



# Emergency Help

- ▶ Sometimes you screw up your Repository.
  - ▶ Accidentally have garbage in a Commit (add large binary data to Repository).
  - ▶ Deleted important files.
  - ▶ A screwed-up merge.
- ▶ Direct help (**Warning: These commands can lose you data**):
  - ▶ `git checkout -- <file>`: Restore/Reset a file to the version from the Repository.
  - ▶ `git reset --soft <Commit>`: Sets HEAD to the given Commit but does not touch anything.
  - ▶ `git reset [--mixed] <Commit>`: Resets the Index but does not touch the working directory.
  - ▶ `git reset --hard <Commit>`: Resets Index and working directory to given Commit (this may cost you unsaved data).
- ▶ There is a tremendously helpful website on how to “un-screw” your Git:  
<https://sethrobertson.github.io/GitFixUm/fixup.html>



# Work with Remotes



# Remote Repositories

- ▶ Until now we only worked on our local machine.
- ▶ Git becomes very useful when sharing the Repository.
  - ▶ Other people can work on the Repository.
  - ▶ You have a graphical overview with GitHub/GitLab.
  - ▶ You can use external services for automation.
- ▶ I personally recommend to use GitLab:
  - ▶ It is free.
  - ▶ You have unlimited private projects (GitHub has this as well now).
  - ~~▶ You have free quota on automated jobs (CI-pipeline).~~
  - ▶ The UI is way more intuitive than GitHub (for me at least).
- ▶ ZITI runs its own GitLab instance<sup>1</sup> here  
<https://edu.ziti.uni-heidelberg.de>. Just log in with your UniID.



<sup>1</sup> courtesy of CSG@ZITI



# Register SSH Key

- ▶ In order to synchronize with the remote Repository, it is best to register your SSH public key with GitLab.
- ▶ Go to your profile picture → *Settings* → SSH Keys.
- ▶ Obtain your SSH key (`cat ~/.ssh/id_ed25519.pub`).
- ▶ Paste it in the text box.

User Settings > SSH Keys

### 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 `~/.ssh/id_ed25519.pub` or `~/.ssh/id_rsa.pub` and begins with `ssh-ed25519` or `ssh-rsa`. Do not paste your private SSH key, as that can compromise your identity.

```
ssh-ed25519  
AAAAC3h2uClZD11NTESAAAAAmOY4A/7w13+80Bfymj8C2V+rMkK3hQIqys7S3ZL+  
florian@Florian-Laptop
```

Title:

Expires at:

Give your individual key a title.





# Clone a Repository

- ▶ You can create a new Repository via the web UI.
- ▶ You can clone it via the *Clone* button, use SSH.
- ▶ Get your terminal and enter `git clone <ssh-url>`.
- ▶ Example: `git clone git@edu.ziti.uni-heidelberg.de:myuser/testproject.git`.

The screenshot shows the GitLab web interface for a repository named 'testproject'. At the top, a blue banner states 'Project 'testproject' was successfully created.' Below this, the repository name 'testproject' is displayed with a 'Clone' button. A dropdown menu is open, showing two options: 'Clone with SSH' with the URL 'git@gitlab.ziti.uni-heidelberg.de:myuser/testproject.git' and 'Clone with HTTPS' with the URL 'https://gitlab.ziti.uni-heidelberg.de/myuser/testproject.git'. Below the clone options, there is an 'Initial commit' section showing a commit by Florian Beenen. At the bottom, there are several action buttons like 'Add README', 'Add CHANGELOG', etc., and a table with repository files.

Name	Last commit	Last update
README.md	Initial commit	6 minutes ago



# Inspecting the Remote

- ▶ Your local Git Repository that you cloned from GitLab is now linked to GitLab.
- ▶ This “link” is called a **Remote**.
- ▶ Check out your Remotes by executing `git remote -v`.
- ▶ By default the Remote is called `origin`.

```
florian@Florian-Laptop:/tmp/testproject$ git remote -v
origin  git@gitlab.ziti.uni-heidelberg.de: [REDACTED]/testproject.git (fetch)
origin  git@gitlab.ziti.uni-heidelberg.de: [REDACTED]/testproject.git (push)
florian@Florian-Laptop:/tmp/testproject$
```



# Synchronizing with the Remote

- ▶ When you issue Commits locally, you need to send them to the Remote as well.
  - ▶ `git push` will send your local Commits.
  - ▶ If you get errors, the Remote may have Commits that you don't know about (e. g. by someone else). You need to load these Commits first.
- ▶ When the Remote is updated you need to download the new Commits.
  - ▶ Load the Commits with `git fetch`.
  - ▶ Merge them with your local Branch.
  - ▶ Use `git pull` to perform `git fetch` and `git merge` simultaneously.
  - ▶ You will get conflicts if *fast-forward* is not possible.

```
florian@ec1:~/Documents/git/openshmem$ git pull
remote: Enumerating objects: 88, done.
remote: Counting objects: 100% (88/88), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 74 (delta 60), reused 74 (delta 60), pack-reused 0
Unpacking objects: 100% (74/74), done.
From https://github.com/openshmem-org/oss-s-ucx
 4d913bc..95b5993 master    -> origin/master
Updating 4d913bc..95b5993
Fast-forward
 include/shmem/api.h          | 96 ++++++-----
 include/shmem/generics.h     | 15 +++++-----
 src/collectives/shcoll/src/Makefile.am | 3 +-
 src/lock.c                   | 5 +++-
 src/shmemc/osh_common.in     | 7 +++++-
 src/shmemc/oshrun.py.in      | 46 +++++-----
 6 files changed, 73 insertions(+), 99 deletions(-)
florian@ec1:~/Documents/git/openshmem$
```



# Evacuation Plan

In case of fire



1. `git commit`



2. `git push`



3. leave building






<https://repository-images.githubusercontent.com/43623432/e3756280-e50c-11e9-877f-24272543fd9c>



# Evacuation in Large Companies

In case of fire



-  1. `git commit`
-  2. `git push`
-  3. `JIRA issue number required`  
`! [remote rejected]`  
`error: failed to push refs...`
-  4. `try to fix`
-  5. `burn alive`

[https://www.reddit.com/r/ProgrammerHumor/comments/9xfzaj/our\\_git\\_repo\\_requires\\_every\\_commit\\_to\\_have\\_a/](https://www.reddit.com/r/ProgrammerHumor/comments/9xfzaj/our_git_repo_requires_every_commit_to_have_a/)



# Automated GitLab Jobs

- ▶ You can trigger automated actions in GitLab:
  - ▶ When a Commit to `master` is made.
  - ▶ When a Commit is made that updates `.tex` files.
  - ▶ On a regular basis, e. g. every Wednesday on 09.00h.
  - ▶ ...
- ▶ Automation is based on Docker:
  - ▶ You create or supply a base image: Basically a throw-away Linux instance with specific configuration.
  - ▶ Docker container ensures that the automated job has all the software it needs in the correct version to create reliable output.
  - ▶ Get Docker containers at: <https://hub.docker.com/>
- ▶ Configuration is done in `.gitlab-ci.yml` which must be located in the Repository's root directory.



## Example: Automated LaTeX Compilation

- ▶ If you have  $\text{\LaTeX}$  in your Repository, you can automatically compile a PDF.
- ▶ You may use my Docker image `fbeenen/texlive`. Has the following stuff installed:  
`git unzip curl openssh-client texlive-full`.
- ▶ YAML files are sensitive with regard to whitespace. Do not mix tabs and spaces. Use the correct amount of indentation.
- ▶ Minimalistic `.gitlab-ci.yml`:

```
1 image: fbeenen/texlive
2
3 build:
4   script:
5     - latexmk -pdf raytrace.tex
6   artifacts:
7     paths:
8     - "*.pdf"
```



## Example: More Sophisticated LaTeX Compilation

```
1 image: fbeenen/texlive
2 variables:
3   PDF_PREFIX: "MySuperfile"
4 build:
5   before_script:
6     - apt-get update && apt-get -y install wget
7   script:
8     - cd latex
9     - source autobuild.sh
10    - buildPDF mainfile.tex
11    - mv *.pdf ../
12    - cd ../praesentation/
13    - buildPresentation myPresentation.tex
14    - mv *.pdf ../
15  only:
16    changes:
17      - "**/*.tex"
18  artifacts:
19    paths:
20      - "${PDF_PREFIX}*.pdf"
```





# Artifacts

- ▶ Your `.gitlab-ci.yml` should specify some output products: *Artifacts*.
- ▶ These serve two purposes:
  - ▶ They are used to deliver files to the next stage of the pipeline.
  - ▶ They can be downloaded by the user.
- ▶ Multiple pipeline stages can be chained: For software e. g. “Build”, “Test”, “Deploy”.





# Project with Multiple Pipeline Stages

- Project
- Repository
- Issues 11
- Merge Requests 1
- CI / CD**
- Pipelines**
- Jobs
- Schedules
- Charts
- Operations
- Wiki
- Snippets
- Settings

Status	Pipeline	Commit	Stages	Duration	Created
passed	#716 by latest	master -> a7988f13 Merge branch '19-ideen-mo...	✓ ✓ ✓	00:02:45	1 month ago
passed	#715 by latest	19-ideen-mox -> 551da516 Added option to import Grid...	✓ ✓	00:02:31	1 month ago
passed	#667 by latest	19-ideen-mox -> 7a262307 Merge branch '5-symbols-fo...	✓ ✓	00:02:14	2 months ago
passed	#666 by latest	master -> 7a262307 Merge branch '5-symbols-fo...	✓ ✓ ✓	00:03:01	2 months ago
failed	#665 by latest	5-symbols-f... -> 4478dfb5 NetgroupSidebar now has a ...	✗ ⏪	00:00:19	2 months ago
passed	#663 by latest	autobuild-f... -> 48e83b6e Fix Autobuild for usage with ...	✓ ✓	00:02:24	2 months ago



# Exercise



# Create Simple Repository

## A First Repository

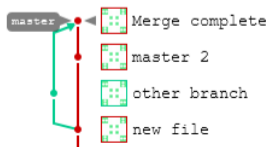
- ▶ Log onto a Linux machine (e. g. `physik1.kip.uni-heidelberg.de`) and start a Terminal.
- ▶ Create a new Git Repository in an empty directory.
- ▶ Create at least three text files and fill them with a few lines.
- ▶ Commit all files to the Repository.
- ▶ Add a new Branch and switch to the new Branch.
- ▶ Create another new file with some content.
- ▶ Commit the new file to the newly created Branch.
- ▶ Switch back to the `master` Branch.
- ▶ Delete a file and commit the deletion to the Repository.
- ▶ Merge your new Branch to the `master` Branch (there should be no conflicts)!



# Merging by Hand

## Merging Conflicts

- ▶ Select one file of your Repository and change it on the `master` Branch.
- ▶ Switch to your other Branch and change something else in the **same** file.
- ▶ Make sure, both changes are committed to the respective Branch.
- ▶ Now try to merge your additional Branch to the `master` Branch. This should yield a conflict.
- ▶ Resolve the conflict by editing the conflicting file.
- ▶ Commit the changes to the `master` Branch.
- ▶ Verify with `git log` that the Commits from the other Branch are now visible on `master`.





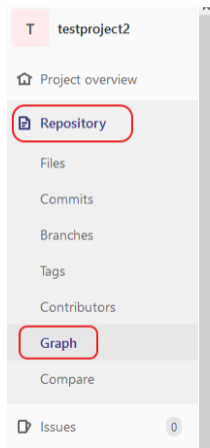
# Working with GitLab

## Import the Project in GitLab

- ▶ Log into GitLab and create an empty project (do not initialize it with a `Readme.md`).
- ▶ Register your SSH key with GitLab.
- ▶ Get the SSH URL of the newly created project (via the “Clone” button).
- ▶ Add the GitLab project as Remote to your existing local Git Repository via `git remote add origin <url>`.
- ▶ Push your local Repository to the Remote via `git push`.
- ▶ Ensure that everything works (e. g. by inspecting the Commit graph under “Repository” → “Graph”).



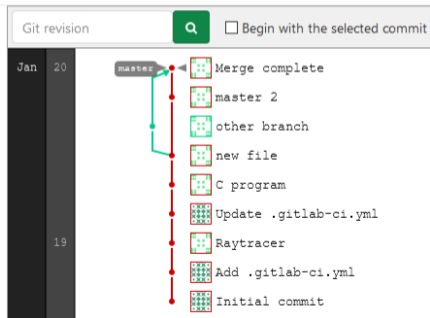
# Working with GitLab (2)



Florian > testproject2 > Graph

master


You can move around the graph by using the arrow keys.





# Automation

## GitLab CI

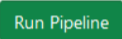
- ▶ We want to automatically build C program files.
- ▶ Add a new `.gitlab-ci.yml`.
- ▶ You can use a wizard for this: Click on .
- ▶ You can use the C++ template if you don't want to write the file from scratch. Delete everything from the template that you don't need.
- ▶ Make the `.gitlab-ci.yml` compile a file called `program.c`. The generated binary shall be called `myTool`.
- ▶ Ensure that the compiled binary file is included in the Artifacts.
- ▶ Write a small C program that does something useful (e. g. print out the first 10 square numbers) and commit it to the Repository.
- ▶ Also ensure that your `.gitlab-ci.yml` is located at the Repository root and committed to the Repository.





# Automation (2)

## GitLab CI

- ▶ ...
- ▶ Run the pipeline via “CI/CD” → 
- ▶ Debug your script by inspecting the log output until the pipeline does not fail anymore.
- ▶ If everything works you can download your executable in a ZIP directory from the Artifacts.
- ▶ Obtain the executable and run it.



#244283840

latest



P master → 7c49efe3



Merge complete



00:00:44

26 minutes ago

