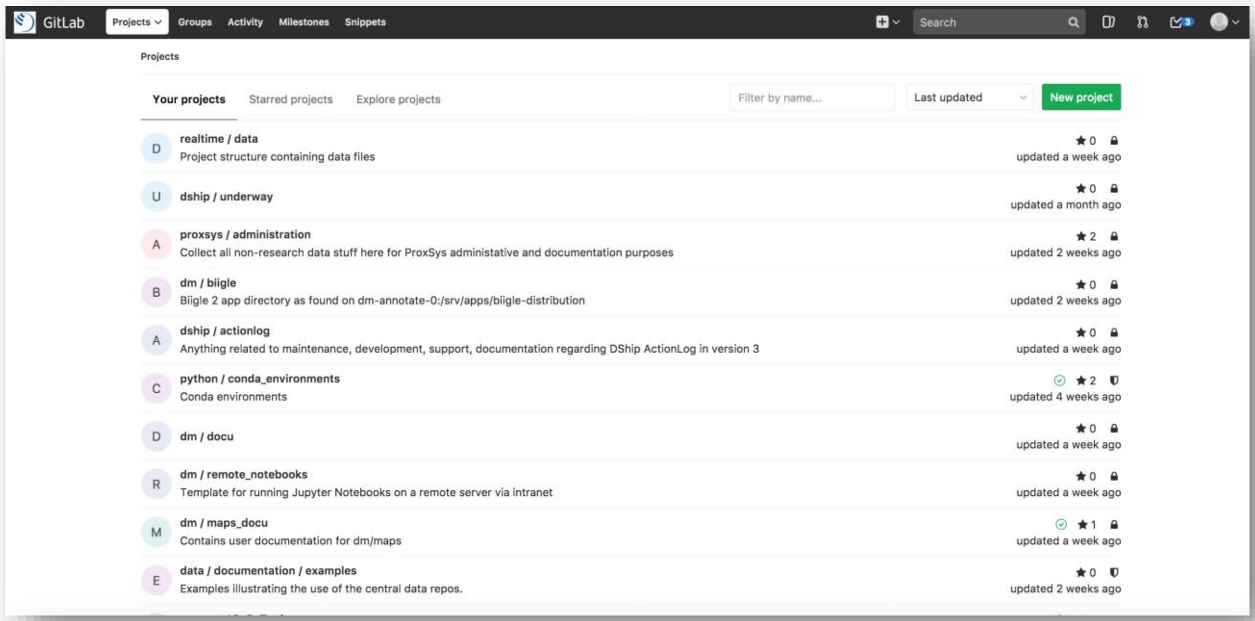
GEOMAR
Helmholtz-Zentrum für Ozeanforschung Kiel

Tutorial

Git - Version Control System



D	realtime / data Project structure containing data files	★ 0 updated a week ago
U	dship / underway	★ 0 updated a month ago
A	proxsys / administration Collect all non-research data stuff here for ProxSys administative and documentation purposes	★ 2 updated 2 weeks ago
B	dm / biigle Biigle 2 app directory as found on dm-annotate-0:/srv/apps/biigle-distribution	★ 0 updated 2 weeks ago
A	dship / actionlog Anything related to maintenance, development, support, documentation regarding DShip ActionLog in version 3	★ 0 updated a week ago
C	python / conda_environments Conda environments	★ 2 updated 4 weeks ago
D	dm / docu	★ 0 updated a week ago
R	dm / remote_notebooks Template for running Jupyter Notebooks on a remote server via intranet	★ 0 updated a week ago
M	dm / maps_docu Contains user documentation for dm/maps	★ 1 updated a week ago
E	data / documentation / examples Examples illustrating the use of the central data repos.	★ 0 updated 2 weeks ago

Contact

Data Management

Adress: GEOMAR Helmholtz Centre for Ocean Research Kiel
Wischhofstr. 1-3
24148 Kiel | Germany

Phone: 0431 / 600 2294
E-Mail: datamanagement@geomar.de

CONTENT

Introduction

Collaborate on code and research data with the Version Control System Git. Manage Git repositories with fine grained access controls and keep your code and your research data secure. Perform code and research data reviews and enhance collaboration with support even for large files (Git-LFS). Each project can track issues, utilize individual wiki and many more features out-of-the-box.
In this tutorial you will find information about how to install git, the general understanding of Git and Git (Large file storage) as well as instructions for the basic usage of the versioning system. We highly recommend to work with the command line when using Git, that is why this tutorial is focused on those workflows, but we also listed some common GUI-Clients for those who are not used to work with the command line. GUI-Clients make common tasks and features of git much easier, but as soon as you need to do some more special tasks (git has a lot of special features!), the GUI-Client will no longer work. So we recommend to get experienced with the command line.

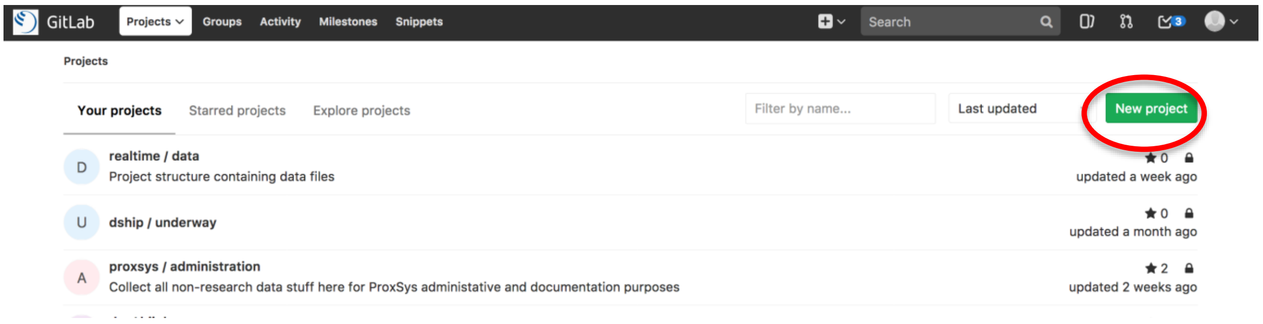
1. Installing Git and get access to GEOMAR GitLab Server

When you use Git for the first time, you need to install and set up a few things.
Install Git on Linux, Mac or Windows: https://git-scm.com/downloads

1.1 Have access to GEOMAR GitLab Server and create a new Project

To use the GEOMAR GitLab Server to share repositores you need an account on https://git.geomar.de. Please use your login credentials as for data management's https://portal.geomar.de and start right away with your first project in your personal name space. If your login fails, you need to reset/initialise your Password. Go to https://portal.geomar.de/web/data-management/home and follow the instructions there or contact the GEOMAR data management team.

- Login to **Geomar's Gitlab server** https://git.geomar.de
- In your dashboard, click the green "**New project**" button or use the plus icon in the upper right corner of the navigation bar to create a new project, e.g. planets.

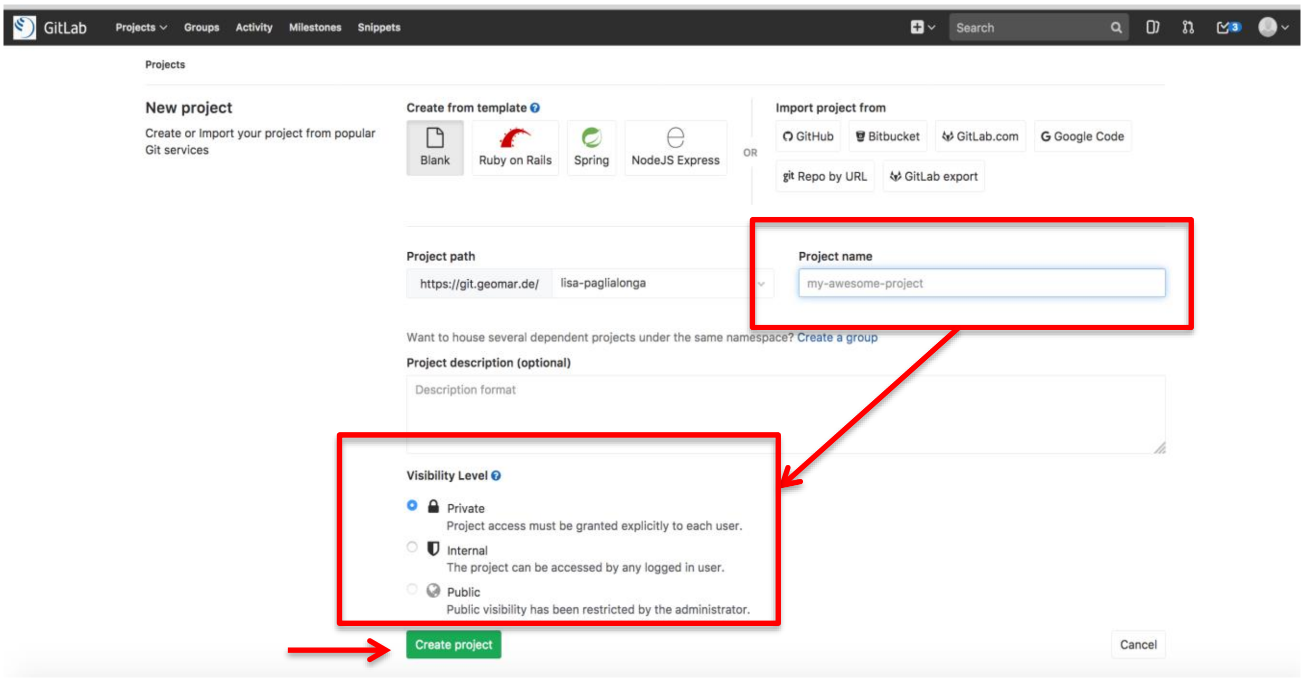


realtime / data
Project structure containing data files
★ 0 updated a week ago

dship / underway
★ 0 updated a month ago

proxsys / administration
Collect all non-research data stuff here for ProxSys administative and documentation purposes
★ 2 updated 2 weeks ago

- This opens the "**New project**" page.


Projects

New project

Create or Import your project from popular Git services

Create from template: Blank, Ruby on Rails, Spring, NodeJS Express

OR

Import project from: GitHub, Bitbucket, GitLab.com, Google Code, Repo by URL, GitLab export

Project path: https://git.geomar.de/ lisa-paglialonga

Project name: my-awesome-project

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

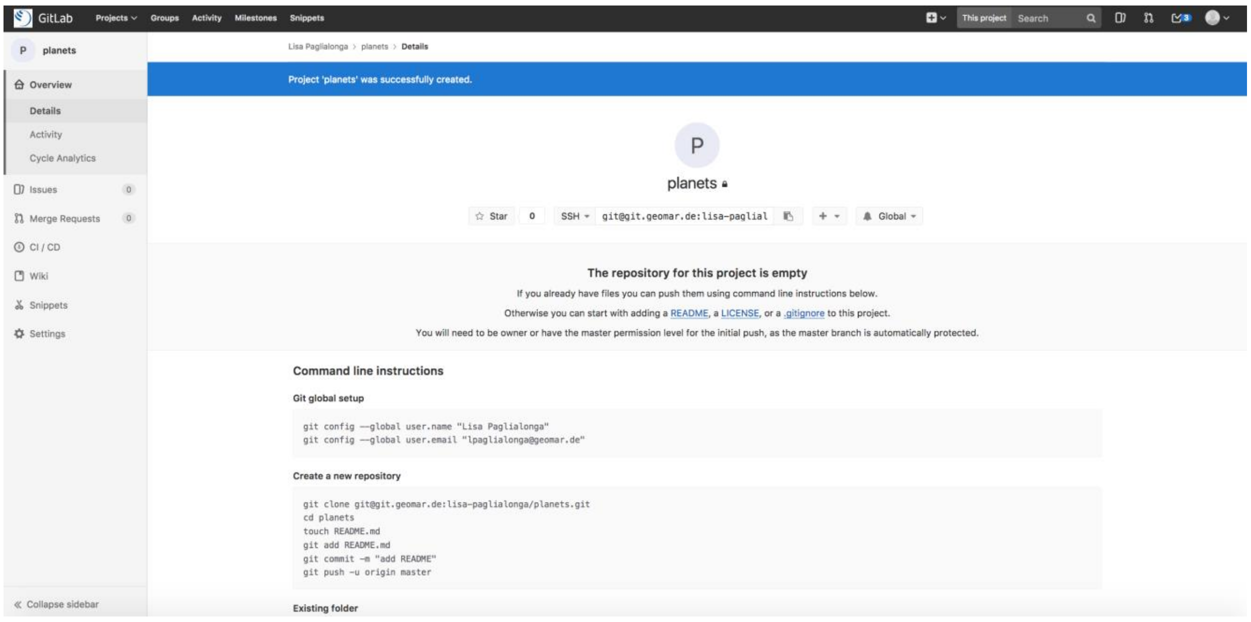
Project description (optional)

Description format

Visibility Level

- Private
  Project access must be granted explicitly to each user.
- Internal
  The project can be accessed by any logged in user.
- Public
  Public visibility has been restricted by the administrator.

- Enter the name of your project in the Project name field. If you have a project in a different repository, you can import it by clicking on "**Import project from**" button. The "**Project description (optional)**" field enables you to enter a description for your project's dashboard, which will help others understand what your project is about. Changing the Visibility Level modifies the project's viewing and access rights for users.
- Click **"Create project"**.
- As soon as the project is created, Gitlab displays a page with a URL and some information on how to configure your local repository:


Lisa Paglialonga > planets > Details

Project 'planets' was successfully created.

planets

SSH git@git.geomar.de:lisa-paglial

The repository for this project is empty

If you already have files you can push them using command line instructions below.

Otherwise you can start with adding a README, a LICENSE, or a .gitignore to this project.

You will need to be owner or have the master permission level for the initial push, as the master branch is automatically protected.

Command line instructions

Git global setup

```
git config —global user.name "Lisa Paglialonga"
git config —global user.email "lpaglialonga@geomar.de"
```

Create a new repository

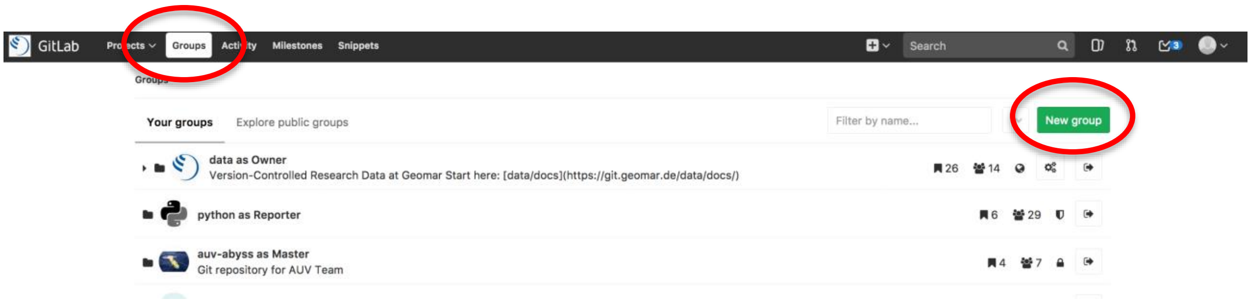
```
git clone git@git.geomar.de:lisa-paglialonga/planets.git
cd planets
touch README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

Existing folder

Your projects in GitLab can be organized in two different ways: under your own namespace for single projects, such as your-name/project-1 or under **groups**. If you organise your projects under a group, it works like a folder. You can manage your group members' permissions and access to the projects.

To create a group:

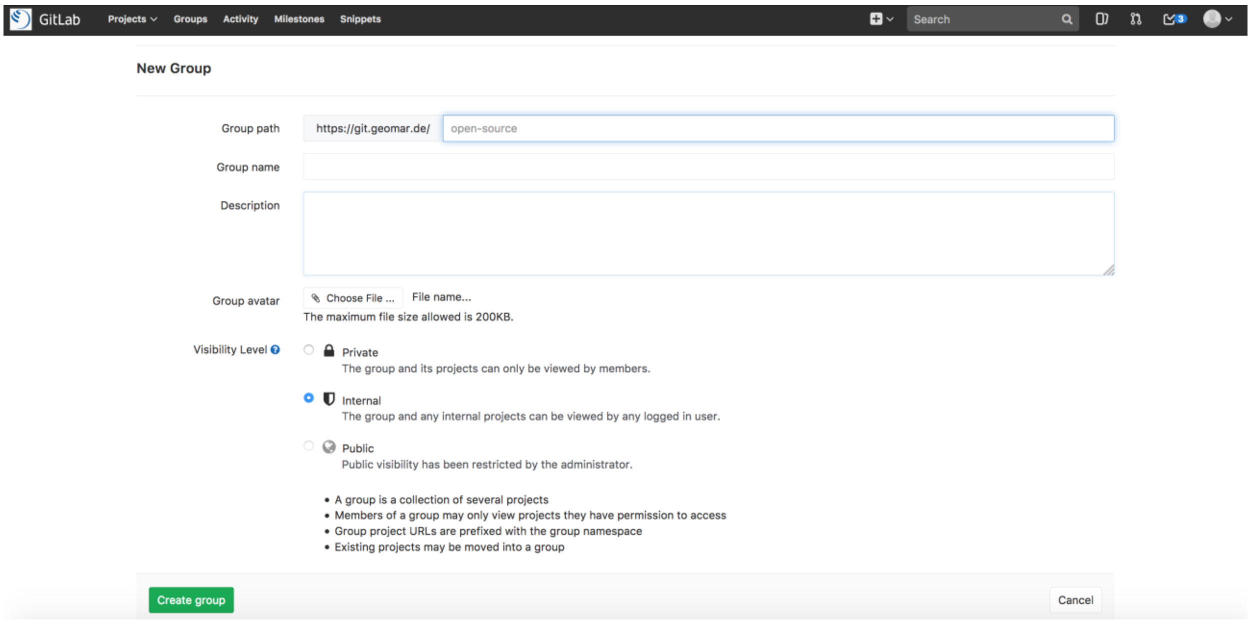
- Once in your groups dashboard, click on **New group**.



data as Owner Version-Controlled Research Data at Geomar Start here: [data/docs](https://git.geomar.de/data/docs/)	26	14
python as Reporter	6	29
auv-abyss as Master Git repository for AUV Team	4	7

Fill out the information needed:

1. Set the "Group path" which will be the namespace under which your projects will be hosted.
2. The "Group name" will populate with the path. Optionally, you can change it. This is the name that will display in the group views.
3. Optionally, you can add a description so that others can briefly understand what this group is about.
4. Optionally, choose an avatar for your project.
5. Choose the visibility level.



New Group

Group path: https://git.geomar.de/ open-source

Group name

Description

Group avatar: Choose File ... File name...
The maximum file size allowed is 200KB.

Visibility Level
- Private
  The group and its projects can only be viewed by members.
- [✓ Internal]
  The group and any internal projects can be viewed by any logged in user.
- Public
  Public visibility has been restricted by the administrator.

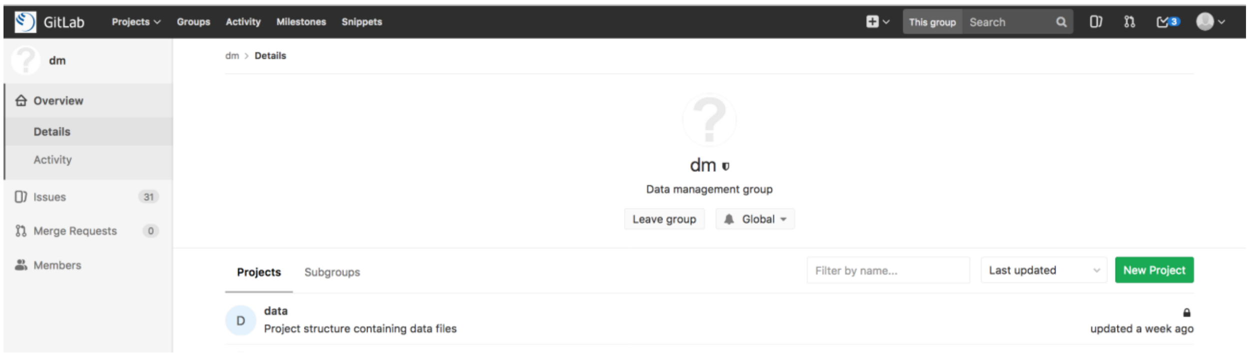
- A group is a collection of several projects
- Members of a group may only view projects they have permission to access
- Group project URLs are prefixed with the group namespace
- Existing projects may be moved into a group

- Finally, click the **Create group** button.

Add a new project to a group

There are two different ways to add a new project to a group:

- Select a group and then click on the New project button.



You can then continue on creating a project.

- While you are creating a project, select a group namespace you've already created from the dropdown menu.

1.2 Generating a SSH key for GEOMAR GitLab Server

Git is a distributed version control system, which means you can work locally but you can also share or "push" your changes to other servers. Before you can push your changes to a GitLab Server you need a secure communication channel for sharing information.

The SSH protocol provides this security and allows you to authenticate to the GitLab remote server without supplying your username or password each time.

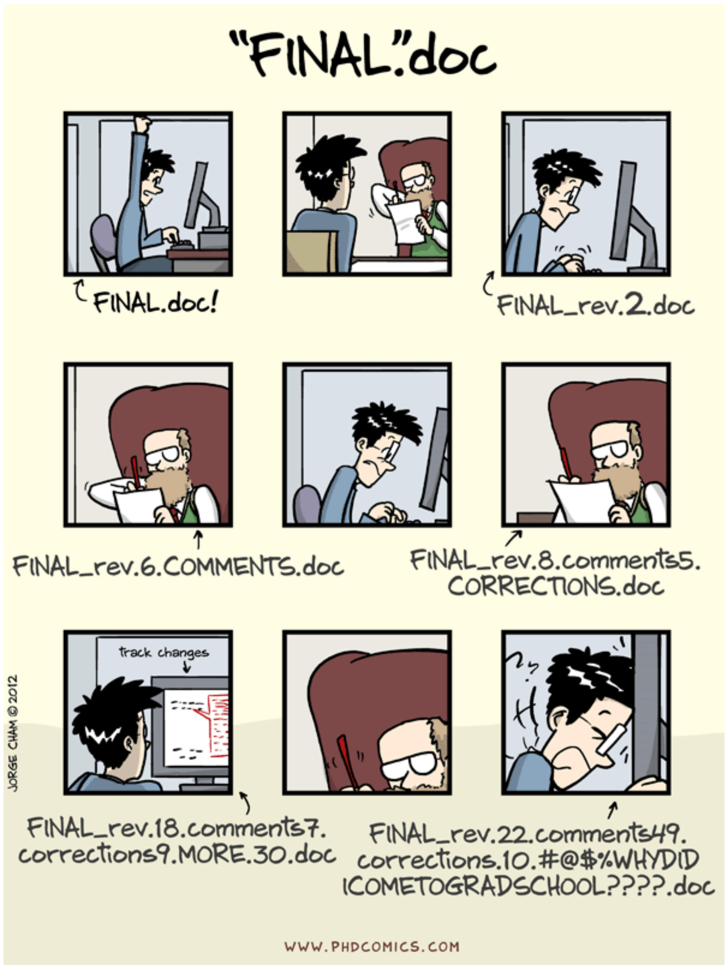
For a more detailed explanation of how the SSH protocol works, we advise you to read this nice tutorial by DigitalOcean: https://www.digitalocean.com/community/tutorials/understanding-the-ssh-encryption-and-connection-process

Please follow the instructions on https://git.geomar.de/help/ssh/README.md - generating-a-new-ssh-key-pair to add a key allowing for password-less work.

2. Understanding Git

2.1 Version Control with Git

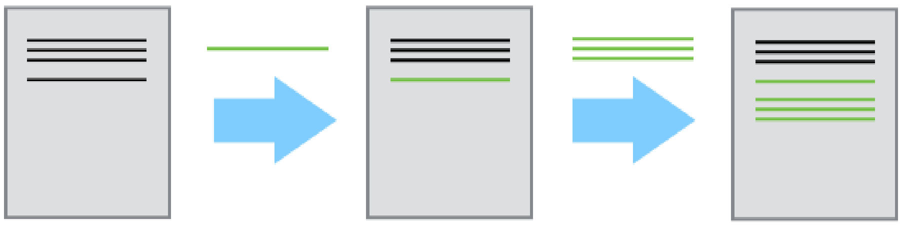
Version control can be used to keep track of what one person did and when. Even if you aren't collaborating with other people, automated version control is much better than this situation:


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.corrections9.MORE.30.doc
FINAL_rev.22.comments49.corrections.10.#@$%WHYDIDICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

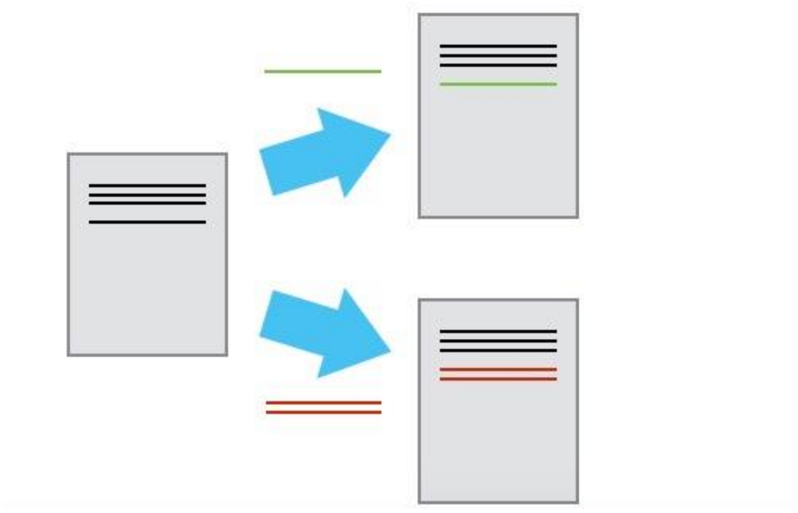
Piled Higher and Deeper" by Jorge Cham
(Source: http://www.phdcomics.com)

We've all been in this situation before: it seems ridiculous to have multiple nearly-identical versions of the same document. Some word processors let us deal with this a little better, such as Microsoft Word's "Track Changes" or Google Docs' version history.

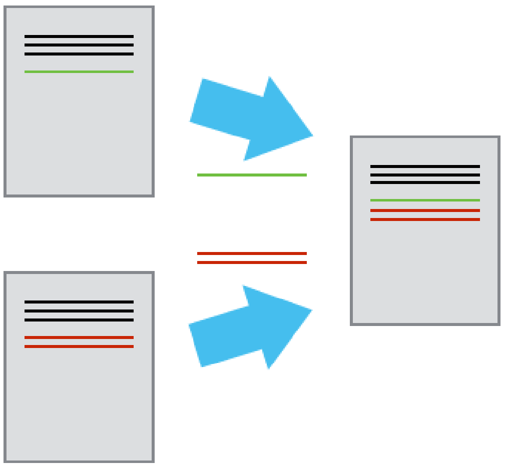
Version control systems start with a base version of the document and then save just the changes you made at each step of the way. You can think of it as a tape: if you rewind the tape and start at the base document, then you can play back each change and end up with your latest version.



Once you think of changes as separate from the document itself, you can then think about "playing back" different sets of changes onto the base document and getting different versions of the document. For example, two users can make independent sets of changes based on the same document.



If there aren’t conflicts, you can even play two sets of changes onto the same base document.



A version control system is a tool that keeps track of these changes and helps to version and merge files. It allows you to decide which changes make up the next version, called a commit, and keeps useful metadata about them. **The complete history of commits for a particular project and their metadata make up a repository.** Repositories can be kept in sync across different computers facilitating collaboration among different people.

2.2 The long history of Version Control Systems

Automated version control systems are nothing new. Tools like RCS, CVS, or Subversion have been around since the early 1980s and are used by many large companies. However, many of these are now becoming considered as legacy systems due to various limitations in their capabilities. In particular, the more modern systems, such as Git and Mercurial are distributed, meaning that they do not need a centralized server to host the repository. These modern systems also include powerful merging tools that make it possible for multiple authors to work within the same files concurrently.

3. Getting started - Setup

3.1 Setup on command line

Below are a few examples of configurations you will set as you get started with Git: your name and email address, to colorize your output, your preferred text editor, and your global settings (i.e. for every project). On a command line, Git commands are written as git verb, where verb is what we actually want to do.

So here is how you set up Git:

- `git config --global user.name ""` (enter your name between the marks)
- `git config --global user.email ""` (enter your e-mail adress between the marks)
- `git config --global color.ui "auto"`

This user name and email will be associated with your subsequent Git activity, which means that any changes pushed to GitHub (http://github.com/), BitBucket (https://bitbucket.org/), GitLab (http://gitlab.com/) or another Git host server will include this information. The commands we just ran above only need to be run once: the flag --global tells Git to use the settings for every project, in your user account, on this computer.

You also have to **set up your favorite text editor**, following this list to choose one:

Editor Configuration command:

- Atom: `git config --global core.editor "atom --wait"`
- nano: `git config --global core.editor "nano -w"`
- Text Wrangler (Mac): `git config --global core.editor "edit -w"`
- Sublime Text (Mac): `git config --global core.editor "subl -n -w"`
- Sublime Text (Win, 32-bit install): `git config --global core.editor "'c:/program files (x86)/sublime text 3/sublime_text.exe' -w"`
- Sublime Text (Win, 64-bit install): `git config --global core.editor "'c:/program files/sublime text 3/sublime_text.exe' -w"`
- Notepad++ (Win, 32-bit install): `git config --global core.editor "'c:/program files (x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"`
- Notepad++ (Win, 64-bit install): `git config --global core.editor "'c:/program files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"`
- Kate (Linux): `git config --global core.editor "kate"`
- Gedit (Linux): `git config --global core.editor "gedit -s -w"`
- Scratch (Linux): `git config --global core.editor "scratch-text-editor"`
- emacs: `git config --global core.editor "emacs"`
- vim: `git config --global core.editor "vim`

Note: vim is the default editor for many programs, if you haven’t used vim before and wish to exit a session, type Esc then :q! and Enter.

You can check your settings at any time: `git config –list`

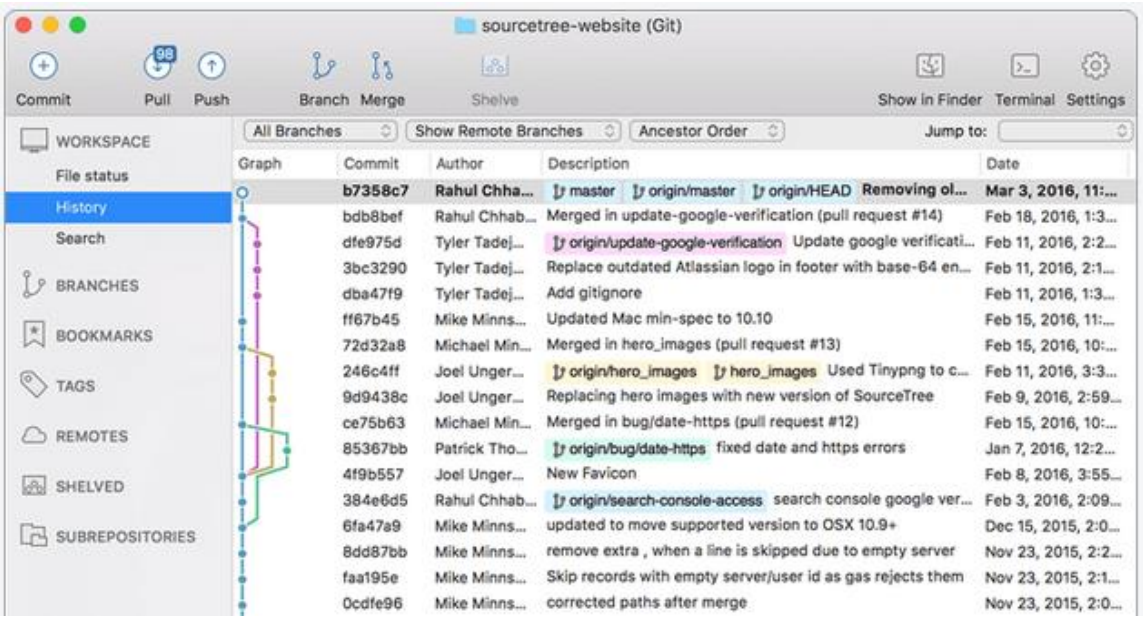
3.2 GUI-Clients for Git

We highly recommend to work with the command line when using Git, that is why this tutorial is focused on those workflows, but we also listed some common GUI-Clients for those who are not used to work with the command line. GUI-Clients make common tasks and features of Git much easier, but as soon as you need to do some more special tasks (git has a lot of special features!), the GUI-Client will no longer work. So we recommend to get experienced with the command line.

Here the most common open-source Git-Clients are listed. A good overview of all GUI-Clients gives this website.

3.2.1 Source-Tree

- For Mac & Windows
- Extensive tool, but also needs training time- so it is not suitable for beginners
- For users who want to switch from the command line it is the best choice
- It offers many advanced features (Git Large File Support, Submodules...)
- Free download: https://www.sourcetreeapp.com/
- Source-Tree Tutorial

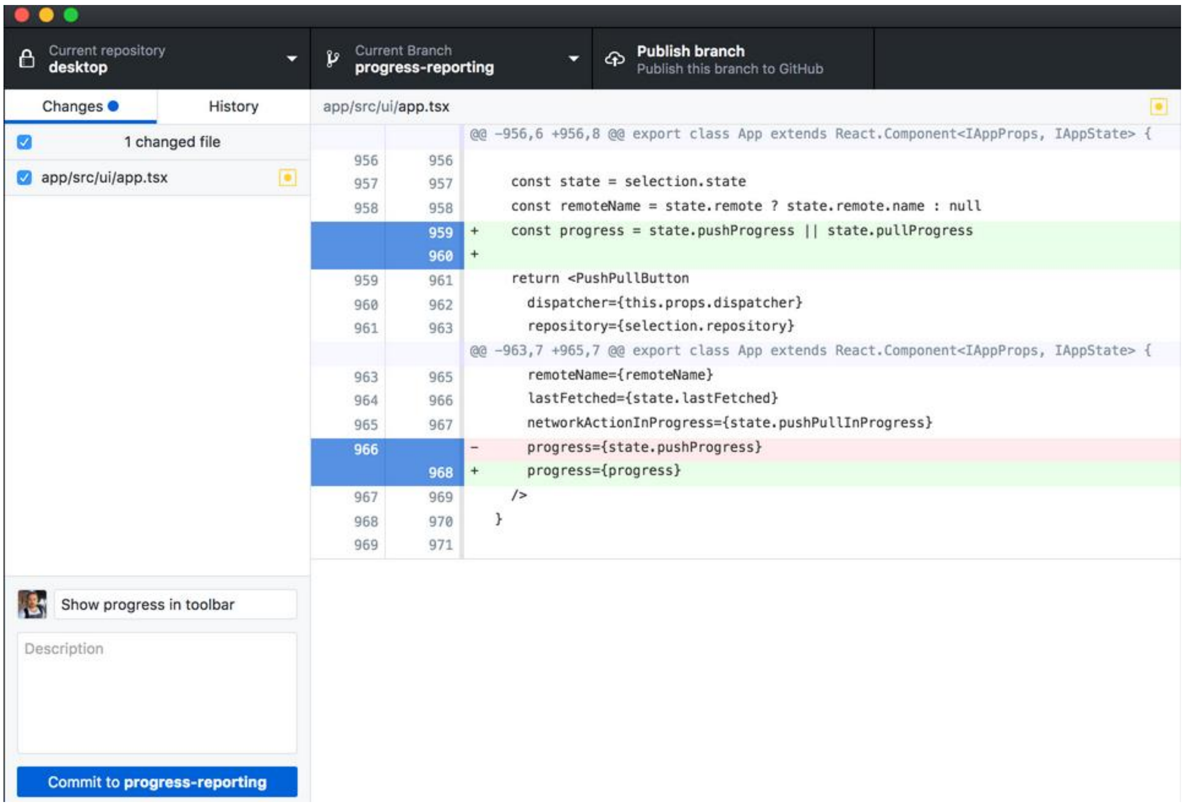


Graph	Commit	Author	Description	Date
	b7358c7	Rahul Chha...	master origin/master origin/HEAD Removing ol...	Mar 3, 2016, 11:...
	bdb8bef	Rahul Chhab...	Merged in update-google-verification (pull request #14)	Feb 18, 2016, 1:3...
	dfe975d	Tyler Tadej...	origin/update-google-verification Update google verificati...	Feb 11, 2016, 2:2...
	3bc3290	Tyler Tadej...	Replace outdated Atlassian logo in footer with base-64 en...	Feb 11, 2016, 2:1...
	dba47f9	Tyler Tadej...	Add gitignore	Feb 11, 2016, 1:3...
	ff67b45	Mike Minns...	Updated Mac min-spec to 10.10	Feb 15, 2016, 11:...
	72d32a8	Michael Min...	Merged in hero_images (pull request #13)	Feb 15, 2016, 10:...
	246c4ff	Joel Unger...	origin/hero_images hero_images Used Tinypng to c...	Feb 11, 2016, 3:3...
	9d9438c	Joel Unger...	Replacing hero images with new version of SourceTree	Feb 9, 2016, 2:59...
	ce75b63	Michael Min...	Merged in bug/date-https (pull request #12)	Feb 15, 2016, 10:...
	85367bb	Patrick Tho...	origin/bug/date-https fixed date and https errors	Jan 7, 2016, 12:2...
	4f9b557	Joel Unger...	New Favicon	Feb 8, 2016, 3:55...
	384e6d5	Rahul Chhab...	origin/search-console-access search console google ver...	Feb 3, 2016, 2:09...
	6fa47a9	Mike Minns...	updated to move supported version to OSX 10.9+	Dec 15, 2015, 2:0...
	8dd87bb	Mike Minns...	remove extra , when a line is skipped due to empty server	Nov 23, 2015, 2:2...
	faa195e	Mike Minns...	Skip records with empty server/user id as gas rejects them	Nov 23, 2015, 2:1...
	0cdfe96	Mike Minns...	corrected paths after merge	Nov 23, 2015, 2:0...

(Screenshot: SourceTree)

3.2.2 GitHub Desktop

- For Mac & Windows
- GitHub Desktop is the perfect tool for beginners
- It's minimalized on the key features of git.
- You can create repositories, copy or create a new branch, fill the created branch with commits and merge again or create a pull request.
- You get all the necessary information and you can also compare the branch with an integrated Diff-Tool.
- Free download: https://desktop.github.com/
- GitHub Desktop Tutorial



app/src/ui/app.tsx

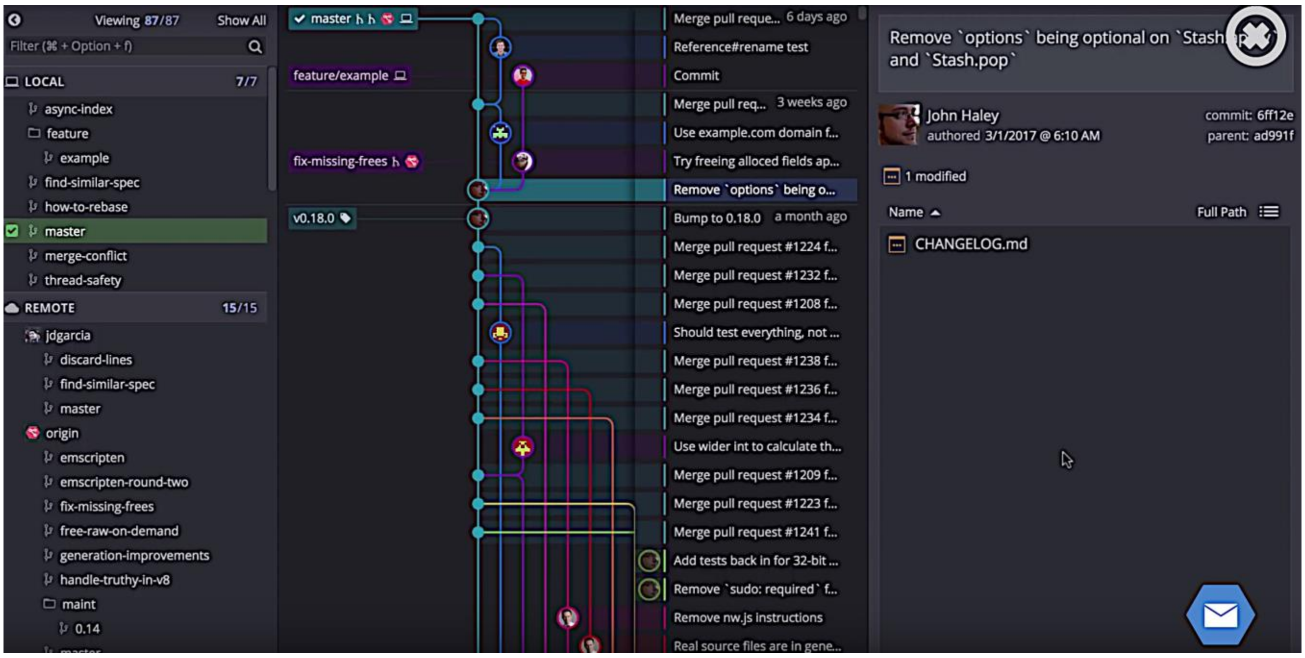
```
@@ -956,6 +956,8 @@ export class App extends React.Component<IAppProps, IAppState> {

    const state = selection.state
    const remoteName = state.remote ? state.remote.name : null
+   const progress = state.pushProgress || state.pullProgress
+
    return <PushPullButton
      dispatcher={this.props.dispatcher}
      repository={selection.repository}
@@ -963,7 +965,7 @@ export class App extends React.Component<IAppProps, IAppState> {
      remoteName={remoteName}
      lastFetched={state.lastFetched}
      networkActionInProgress={state.pushPullInProgress}
-     progress={state.pushProgress}
+     progress={progress}
    />
  }
```

(Screenshot: GitHub Desktop)

3.2.3 GitKraken

- For Linux, Mac, Windows
- Repository Management
- GitLab, GitHub and Bitbucket integration
- Drag & Drop for simplifing commands like merge, push, rebase and more
- Git Large File Support (GitLFS) and features like submodules
- Git Hooks Support and Git Flow Support to manage branches efficently
- Free download: https://www.gitkraken.com/download
- GitKraken Tutorial



Merge pull reque... 6 days ago
Reference#rename test
Commit
Merge pull req... 3 weeks ago
Use example.com domain f...
Try freeing allocated fields ap...
Remove `options` being o...
Bump to 0.18.0 a month ago
Merge pull request #1224 f...
Merge pull request #1232 f...
Merge pull request #1208 f...
Should test everything, not ...
Merge pull request #1238 f...
Merge pull request #1236 f...
Merge pull request #1234 f...
Use wider int to calculate th...
Merge pull request #1209 f...
Merge pull request #1223 f...
Merge pull request #1241 f...
Add tests back in for 32-bit ...
Remove `sudo: required` f...
Remove nw.js instructions
Real source files are in gene...

Remove `options` being optional on `Stash.apply` and `Stash.pop`

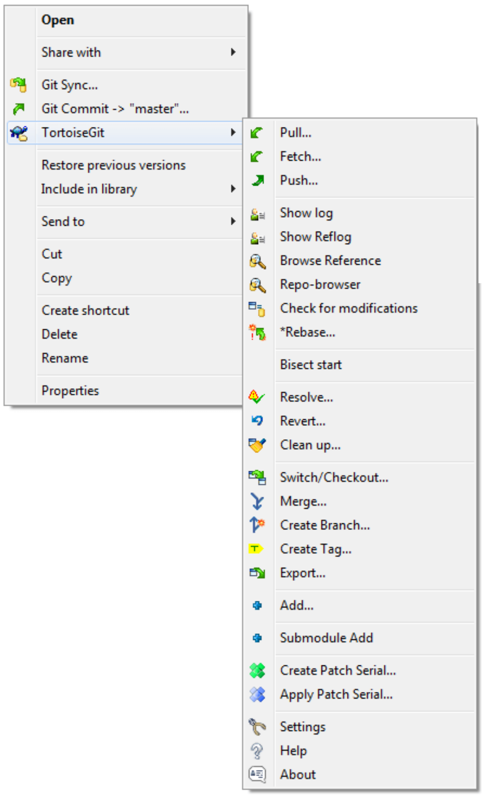
John Haley
authored 3/1/2017 @ 6:10 AM
commit: 6ff12e
parent: ad991f

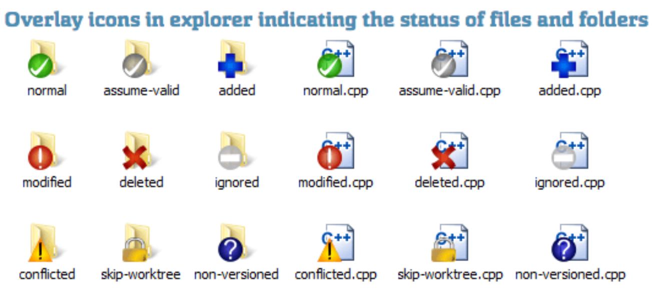
1 modified
Name
Full Path
CHANGELOG.md

(Screenshot: GitKraken)

3.2.4 TortoiseGit

- For Windows
- Windows Explorer integration (as extension): All Git commands are available from the explorer context menu. TortoiseGit adds its own submenu. TortoiseGit provides icon overlays that indicate the status of Git working trees and files.
- Supports you by regular tasks, such as committing, showing logs, diffing two versions, creating branches and tags...
- Free download: https://tortoisegit.org/
- TortoiseGit Tutorial





(Screenshots: TortoiseGit)

4. Getting started with first repository

Once Git is configured, you can start using it. Let's create a directory (here e.g. planets) on your local computer for your work and then move into that directory.

- `mkdir planets` (creating a repository called planets)
- `cd planets` (switch into this directory)

Then you tell Git to make planets a git repository, a place where Git can store versions of your files:

- `git init` (tell Git to make your directory a git repository)

If you use the command `ls` to show the directory's contents, it appears that nothing has changed:

- `ls`

But if we add the -a flag to show everything, we can see that Git has created a hidden directory within planets called **.git**:

- `ls -a`

Git stores information about the project in this special sub-directory. If you ever delete it, you will lose the project's history.

Tracking Changes

You can check that everything is set up correctly by asking Git to tell you the status of your project:

- `git status`

The first two statements tell us where you are (branch master and initial commit). The "untracked files"-message means that there is a file in the directory that Git isn't keeping track of. You can tell Git to track a file (here e.g. mars.txt) by using **`git add`.**

- `git add mars.txt`

Git now knows that it's supposed to keep track of mars.txt, but it hasn't recorded these changes as a commit yet. For that you need to run one more command **`git commit`**. As especially in collaborative projects, it is a good idea to provide detailed commit messages, so add a message to the command.

- `git commit -m "Add the file mars.txt to repository"`

When you run **`git commit`**, Git takes everything you have told to save by using **`git add`** and stores a copy permanently inside the special .git directory. This permanent copy is called a commit (or revision) and its short identifier is **e.g. f22b25e** (your commit has another identifier).

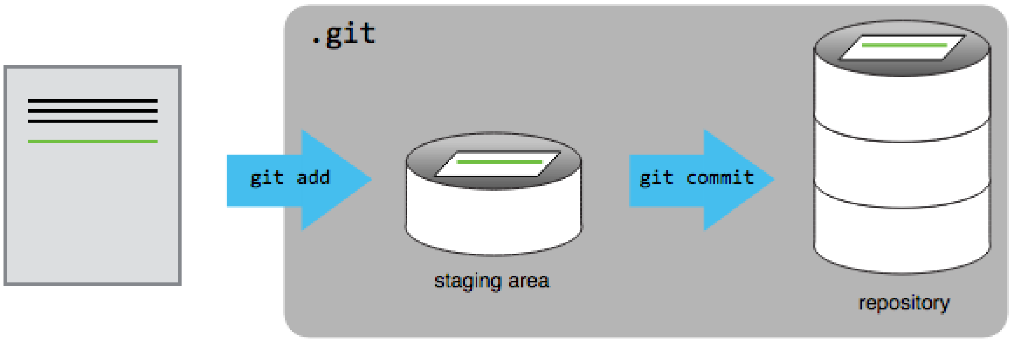
Run **git status** now:

- `git status`

...it tells us everything is up to date. The command **`git status`** gives you also information about if a file it already knows has been modified. The output will be like the following:

```
On branch master
Changes not staged for commit: (use "git add ..." to update what will be committed)
(use "git checkout -- ..." to discard changes in working directory)
modified:   mars.txt
no changes added to commit (use "git add" and/or "git commit -a"
```

The last line is the key phrase: “no changes added to commit”. You have changed this file, but you haven’t told Git you want to track those changes (which you do with **`git add`**) nor have you saved them (which you do with **`git commit`**)



If you think of Git as taking snapshots of changes over the life of a project, **`git add`** specifies what will go in a snapshot (putting things in the staging area), and **`git commit`** then actually takes the snapshot, and makes a permanent record of it (as a commit).

If you want to know what you’ve done recently, you can ask Git to show you the project’s history using **`git log`**:

- `git log`

`git log` lists all commits made to a repository in reverse chronological order. The listing for each commit includes the commit’s full identifier (which starts with the same characters as the short identifier printed by the **`git commit`** command earlier), the commit’s author, when it was created, and the log message Git was given when the commit was created.

It is good practice to always review your changes before saving them. You do this using **`git diff`**. This shows you the differences between the current state of the file and the most recently saved version:

- `git diff`

– The first line tells you that Git is producing output similar to the Unix diff command comparing the old and new versions of the file.
– The second line tells exactly which versions of the file Git is comparing; df0654a and 315bf3a are unique computer-generated labels for those versions.
– The remaining lines are the most interesting, they show you the actual differences and the lines on which they occur. In particular, the + marker in the first column shows where you added a line.

If you have files that you do not want Git to track, like backup files created the editor or intermediate files created during data analysis, you have to tell Git to ignore them. Putting these files under version control would be a waste of disk space. What’s worse, having them all listed could distract you from changes that actually matter. You do this by creating a file in the root directory of your project called **.gitignore**. Write in this file these patterns which should be ignored by Git (e.g. *.dat results/ --> These patterns tell Git to ignore any file whose name ends in .dat and everything in the results directory. If any of these files were already being tracked, Git would continue to track them.):

- `nano .gitignore`

Everyone you're sharing your repository with will probably want to ignore the same things that you're ignoring. So add and commit **.gitignore**.

If you added a file (e.g. mars.txt) to the staging area by mistake and you want to reverse it, you have to use the command **git reset** :

- `git reset mars.txt` (get file back from staging area to working directory)

If you want to delete a file (e.g. mars.txt) from the working directory and add the deletion to the staging area use the command **git rm**:

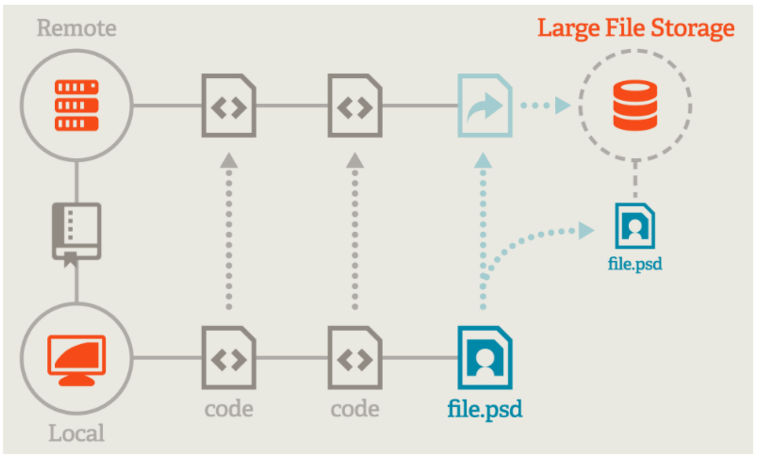
- `git rm mars.txt`
- `git commit -m "delete mars.txt"`

But if you want to remove the file only from the Git repository and not from the filesystem, use:

- `git rm --cached mars.txt`

5. Managing research data (and large binary files) with Git LFS (Large File Storage)

Managing large files such as audio, video and graphics files has always been one of the shortcomings of Git. The general recommendation is to not have Git repositories larger than 1GB to preserve performance. For projects containing large files, particularly large files that are modified regularly, the git cloning process can take a huge amount of time, as every version of every file has to be downloaded by the client. Git LFS (Large File Storage) is a Git extension developed by Atlassian, GitHub, and a few other open source contributors, that reduces the impact of large files in your repository by downloading the relevant versions of them lazily. Specifically, large files are downloaded during the checkout process rather than during cloning or fetching. Git LFS replaces large files such as audio samples, videos, datasets, and graphics with text pointers inside Git, while storing the file contents on a remote server like GitLab. (Source: https://www.atlassian.com/git/tutorials/git-lfs)



Features

- **Large file versioning**: Version large files — even those as large as a couple GB in size — with Git.
- **More repository space**: Host more in your Git repositories. External file storage makes it easy to keep your repository at a manageable size.
- **Faster cloning and fetching**: Download less data. This means faster cloning and fetching from repositories that deal with large files.
- **Same Git workflow**: Work like you always do on Git — no need for additional commands, secondary storage systems, or toolsets.
- **Same access controls and permissions**: Keep the same access controls and permissions for large files as the rest of your Git repository when working with a remote host like GitHub. (Source: https://git-lfs.github.com/)

5.1 Installing Git LFS

Download git lfs at https://git-lfs.github.com/ and install the Git command line extension. You only have to set up Git LFS once.

- `git lfs install`

5.2 Create a new repository

Setting up a git lfs repository works the same as before with the command git init. So you tell Git to make your directory called e.g planets a git repository, a place where Git can store versions of our files:

- `mkdir planets` (creating a repository called planets)
- `cd planets` (switch into this directory)
- `git init` (tell Git to make your directory a git repository)

5.3 Cloning a repository

Cloning the repository works the same as before. Git automatically detects the LFS-tracked files and clones them via HTTP. If you performed the git clone command with a SSH URL, you have to enter your GitLab credentials for HTTP authentication: e.g. `git clone git@gitlab.example.com:group/project.git`

If you already cloned the repository and you want to get the latest LFS object that are on the remote repository, e.g. from branch master: `git clone git@git.geomar.de:<your-username>/planets.git` (Make sure to use the URL for your repository rather than the dummy-one provided here).

If you already cloned the repository and you want to get the latest LFS object that are on the remote repository, e.g. from branch master:

- `git lfs fetch master`

5.4 Tracking files with Git LFS

Now you are going to decide that e.g. *.csv files are large files that you want git-lfs to track. Tracking means that in subsequent commits, these files will now be LFS files.

Note: This does NOT mean that versions of the files in previous commits will be converted. That involves a process commonly known as "rewriting history" and is described in the migration chapter 5.6.

You do this by setting a track pattern, using the **`git lfs track`** command:

- `git lfs track "*.csv"`

You can also manage subset of files (e.g. images/*.png)

- `git lfs track "images/*.png"`

...or even entire directories (e.g. images)

- `git lfs track "images/"`

To see which file types are managed by git lfs, just run git lfs track:

- `git lfs track`

Next, you need to add **.gitattributes** to your git repository. **`git lfs track`** stores the tracked files patterns in **.gitattributes**. When the repo is cloned, the track files patterns are preserved. Since the bin files were added before Git LFS was tracking e.g. .csv files, they need to be added again to the Git index: Make sure that *.gitattributes is tracked by git. Otherwise Git LFS will not be working properly for people cloning the project.

- `git add .gitattributes "*.csv"`

If you know which files you want to track, you have to add and commit them to the repository with **`git add`** and **`git commit`**:

- `git add .`
- `git commit -m "Add new large assets"`

If you push your changes to your git remote (`git push`), git lfs intercepts the files and sents them to the git lfs server. **It creates a small pointer file in your repository, which is used for linking to the acutal files on the git lfs server**.

- `git push`

If you want to know which specific files git lfs is tracking, just run:

- `git lfs ls-files` (if you haven't commited your files this list is currently empty. It is because technically the file isn't an lfs object until after you commit it)

5.5 Controlling when to download large files

By default, git clone for a lfs-enabled repository will download the latest version of the files tracked with lfs. This is OK for projects with only a few large files, but for datasets with gigabytes or even terrabytes of data, this is unacceptable.

Fortunately, there are number of ways to **prevent git lfs from automatically downloading tracked files.**

5.5.1 git lfs commands and configurations

Git lfs extends git by adding special versions of git commands, e.g. `git lfs clone` and `git lfs pull` instead of `git clone` and `git pull`. For the most part, the commands are interchangeable, but in some cases (such as the ones described below), the lfs versions provide access to lfs specific options. Using `git lfs clone` and `git lfs pull` makes sens in most cases, since they provide a smarter mechanism for downloading lfs tracked files (e.g. by starting multiple parrallel downloads).

5.5.2 Skip download globally on the client

If you never want to automatically download tracked lfs files, you can pass this as an option when initialising git lfs: `git lfs install --skip-smudge`. In git lfs terms, **'smudge'** is the process of getting the actual data for the pointers that are stored locally. By installing git lfs with the --skip-smudge option, you are setting the filter `smudge = git-lfs smudge --skip -- %f` in the global **.gitconfig** file in your home directory.

To revert this setting, either set the smudge filter by hand: `git config --global filter.lfs.smudge "git-lfs smudge -- %f"` or by re-installing git lfs and enforcing default options: `git lfs install --force` (**carefull, this will also overwrite other lfs configs you might have set!**).

5.5.3 Skip download for each operation

Instead of setting skip-smudge globally, you specify which files to download and which files to ignore for each `git lfs clone`, `git lfs pull` and `git lfs fetch`-operation individually. Your git lfs installation needs to be fairly recent for this to work, also make sure to use the `git lfs-` variety of the commands. Files are excluded by passing a pattern to the -X or --exclude option. Some examples:

- `GIT_LFS_SKIP_SMUDGE=1 git clone git@git.geomar.de:dm/git_workflow_nb.git` **(This will download only pointer files, substituting the original data files! In order to download the original contents you then have to explicitely pull them→ see chapter 5.5.5.)**
- `git lfs clone git@git.geomar.de:dm/git_workflow_nb.git --branch lfs_test --exclude=*` (by passing -X*, you are telling git lfs to exclude all files from the downloads)
- `git lfs clone git@git.geomar.de:dm/git_workflow_nb.git --branch lfs_test -X *.zip` (exclude all Zip-Files)
- `git lfs clone git@git.geomar.de:dm/git_workflow_nb.git --branch lfs_test --exclude='Test_Files/*'` (exclude all files in the subfolder Test_Files)

5.5.4 Configure repository

In the methods above, the filtering is done by each git user themselve. For git repositories with large amounts of data, it is preferable to configure defaults in the repository itself, preventing new or forgetfull users from downloading the whole repository by mistake.

You can do this by using the command `git config -f .lfsconfig lfs.fetchexclude 'Test_Files/*` which creates the following `.lfsconfig` file (you could of course also create it with your favourite text editor):

```
[lfs]
        fetchexclude = Test_Files/*
```

Don't forget to add your new .lfsconfig to git: `git add .lfsconfig`!

The `lfs.fetchexclude`-option let you specify a pattern of files to exclude by default. The example above excludes all files in the Test_Files folder, i.e. clients will not automatically download lfs tracked files from this folder on clone or pull. This means that the command `git clone git@git.geomar.de:dm/git_workflow_nb.git --branch lfs_test` will download only the placeholders for the large files tracked by git lfs in the Test_File directory.

5.5.5 Downloading skipped files

Now you know how to prevent git lfs from downloading large files, but what if you actually want some of these files? The `-X` option has a counterpart: with `git pull -I` or `git pull --include` you can specify the file you want to have locally:

- `git lfs pull -I "*.csv" -X ""` (download all previously excluded .csv files)
- `git lfs pull -I "Test_Files/" -X ""` (download all previously excluded files in Test_Files folder)
- `git lfs pull --include="Test_Files/" --exclude=""` (same as above)

Make sure to include an empty -X `""`/`--exclue=""` statement, this is needed to temporarily override the `fetchexclude` setting from the `.lfsconfig`.

Finally, running `git lfs fetch --all` will download all files tracked by lfs, regardless of other configurations.

5.6 Migrating existing repository data to LFS

While multiple tools exist, the current recommendation is to use git-lfs-migrate.

- Install Java 1.8 or later
- Download the latest binaries from here: https://github.com/bozaro/git-lfs-migrate/releases/tag/0.2.5
- Do a mirror clone of the repository to rewrite: `git clone --mirror git@github.com:bozaro/git-lfs-migrate.git`

Rewrite e.g. all *.mp4 video files in the repository:

- `java -jar git-lfs-migrate.jar \`
- `-s git-lfs-migrate.git \`

- `-d git-lfs-migrate-converted.git \`
- `-g git@github.com:bozaro/git-lfs-migrate-converted.git \`
- `"*.mp4"`

Push the converted repository as a new repository:

- e.g `git push --mirror git@github.com:bozaro/git-lfs-migrate-converted.git`

6. Using Git to share code and data

Version control really comes into its own when you begin to collaborate with other people. Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like GitHub, BitBucket or GitLab to hold those master copies.

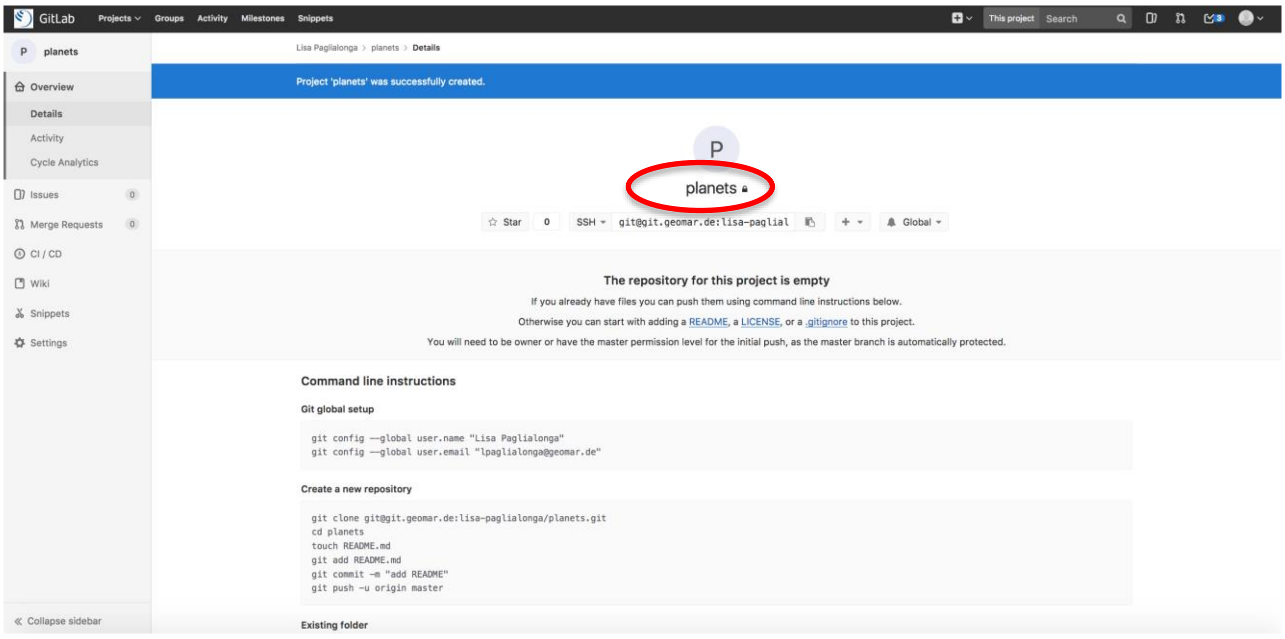
Start by sharing the changes you've made to your current project with the "world".

- Login to **Geomar's Gitlab server** https://git.geomar.de
- In your dashboard, click the green "**New project**" button (see description in Chapter 1.1)

You can now connect your local repository with the Git Lab repository. You do this by making the Gitlab repository a remote for the local repository. The website of the repository on Gitlab includes the string we need to identify: They provide a url to the remote repository.

Copy the URL that is displayed, go into the local planets repository, and run this command:

- `cd planets`
- `git remote add origin git@git.geomar.de:<your-username>/planets.git`
  (Make sure to use the URL for your repository rather than the dummy-one provided here.)



Project 'planets' was successfully created.

The repository for this project is empty

If you already have files you can push them using command line instructions below.

Otherwise you can start with adding a README, a LICENSE, or a .gitignore to this project.

You will need to be owner or have the master permission level for the initial push, as the master branch is automatically protected.

Command line instructions

Git global setup

```
git config —global user.name "Lisa Paglialonga"
git config —global user.email "lpaglialonga@geomar.de"
```

Create a new repository

```
git clone git@git.geomar.de:lisa-paglialonga/planets.git
cd planets
touch README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

Existing folder

You can check that the command has worked by running git remote -v:

- `git remote -v`

The name origin is a local nickname for your remote repository: you could use something else if you want to, but origin is by far the most common choice. Once the nickname origin is set up, the following command will push the changes from our local repository to the repository on GitLab:

- `git push -u origin master`
  (The -u-option associates the local branch master with origin/master which is your nickname for master in the remote repository. This way, you can later just use `git push` and Git knows where to push.)

You can also **clone** an existing repository on Gitlab, which means downloading a project with entire history from the remote repository.

- `git clone git@git.geomar.de:<your-username>/planets.git`
  (Make sure to use the URL for your repository rather than the dummy-one provided here.)

You can pull changes from the remote repository to the local one as well:

- `git pull`

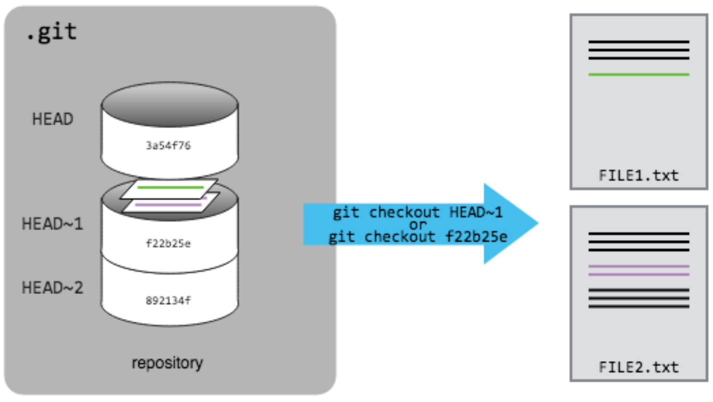
6. Optionally, choose an avatar for your project.

7. Using Git to explore history aka provenance

- You can refer to the most recent commit of the working directory by using the identifier HEAD. If you want to see what you changed at different steps, you can use `git diff`, but with the notation HEAD~1, HEAD~2, and so on, to refer to old commits: e.g. **`git diff HEAD~1 mars.txt`**

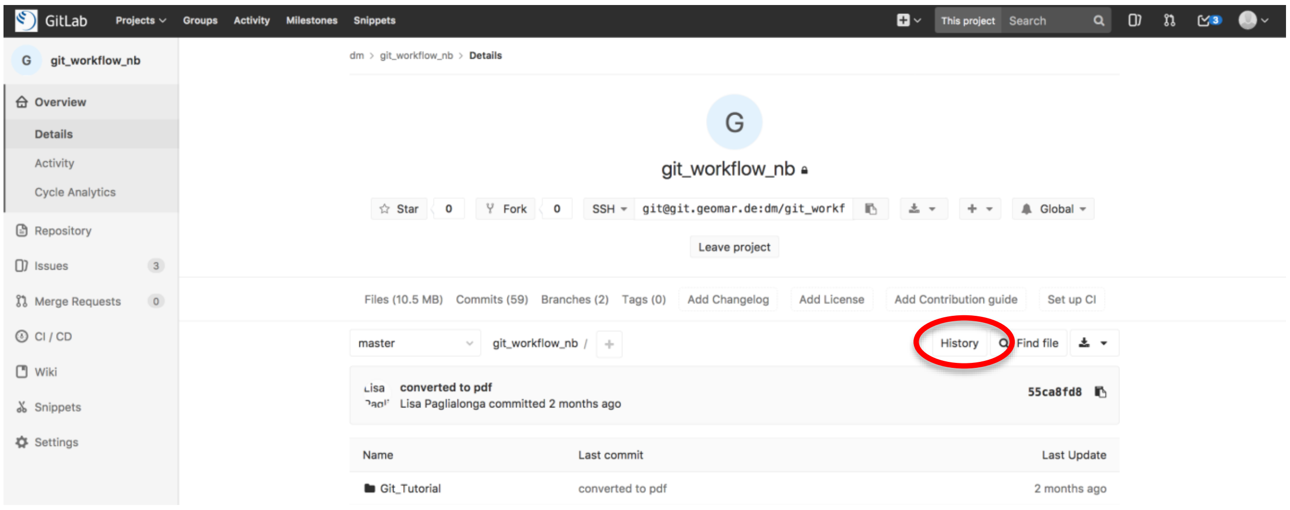
- We can also refer to commits using those long strings of digits and letters that **`git log`** displays. These are unique IDs for the changes, and “unique” really does mean unique: every change to any set of files on any computer has an unique 40-character identifier. But typing out random 40-character strings is annoying, so Git let you use just the first few characters: e.g. **`git diff f22b25e mars.txt`**

- **`git checkout`** checks out (i.e., restores) an old version of a file. In this case, we’re telling Git that we want to recover the version of the file recorded in HEAD, which is the last saved commit. If we want to go back even further, we can use a commit identifier instead: e.g. **`git checkout f22b25e mars.txt`**



History on GitLab

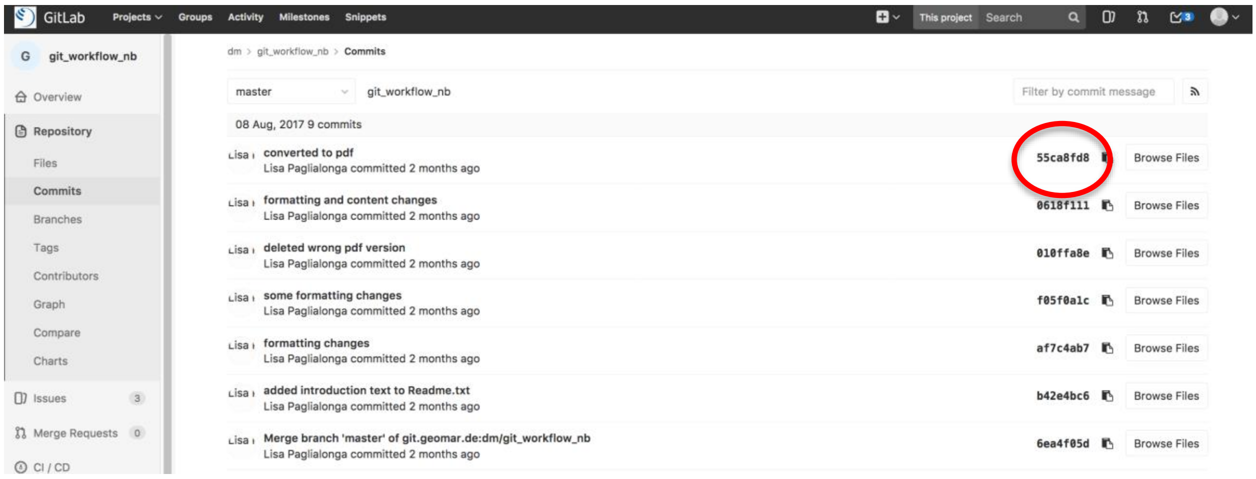
You can also explore the history of your project on GitLab. To do this click on **History** above your project contents:



Name	Last commit	Last Update
Git_Tutorial	converted to pdf	2 months ago

converted to pdf
Lisa Paglialonga committed 2 months ago
55ca8fd8

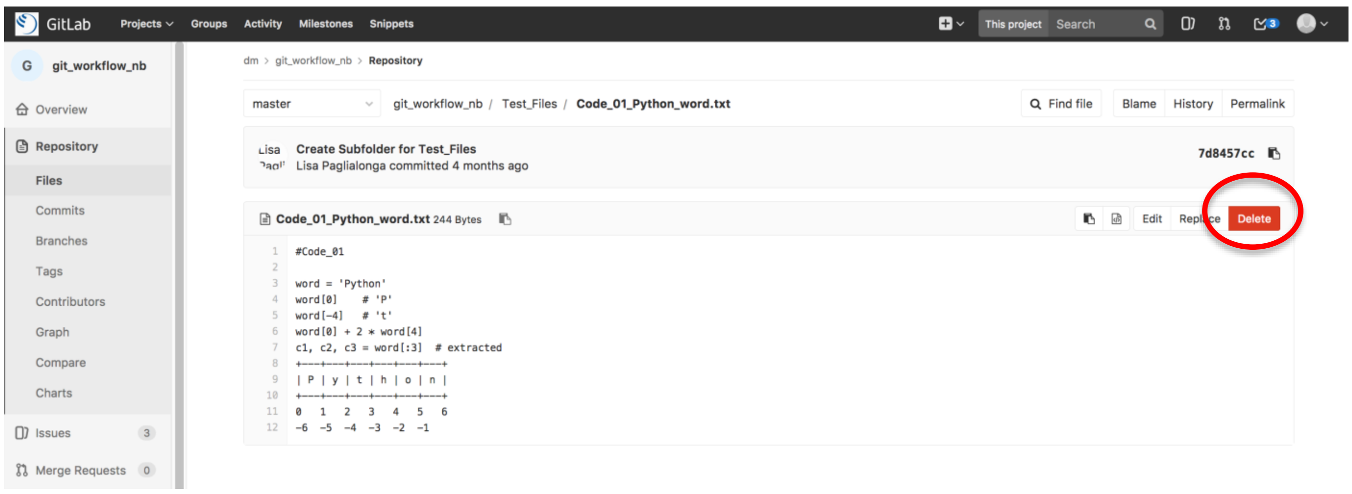
In the History view all commits of your project are listed. If you want to have a look to a certain version of your project, you have to click on the appropriate message or ID of the versions commit on the right side:



08 Aug, 2017 9 commits

converted to pdf Lisa Paglialonga committed 2 months ago	55ca8fd8	Browse Files
formatting and content changes Lisa Paglialonga committed 2 months ago	0618f111	Browse Files
deleted wrong pdf version Lisa Paglialonga committed 2 months ago	010ffa8e	Browse Files
some formatting changes Lisa Paglialonga committed 2 months ago	f05f0a1c	Browse Files
formatting changes Lisa Paglialonga committed 2 months ago	af7c4ab7	Browse Files
added introduction text to Readme.txt Lisa Paglialonga committed 2 months ago	b42e4bc6	Browse Files
Merge branch 'master' of git.geomar.de:dm/git_workflow_nb Lisa Paglialonga committed 2 months ago	6ea4f05d	Browse Files

If your file is already on GitLab and you want to delete it, you now can directly delete it from the web GUI!



Create Subfolder for Test_Files
Lisa Paglialonga committed 4 months ago
7d8457cc

Code_01_Python_word.txt 244 Bytes

```
#Code_01

word = 'Python'
word[0]    # 'P'
word[-4]   # 't'
word[0] + 2 * word[4]
c1, c2, c3 = word[:3]  # extracted
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

8. Advanced usage of Git - branching and merging

To understand the basic workflow of branching and merging in Git, have a look at the example of the following link: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

The most important commands are the following:

- `git branch "enter branch name"` (create a branch)
- `git checkout "branch name"` (switches to this new branch)
- `git checkout master` (switches back to the master branch)
- `git merge "branch name"` (merges branch into the current branch)
- `git branch -d "branch name"` (deletes the branch)

9. Advanced usage of Git - keep a Git repository as a subdirectory

9.1 Submodules

It often happens that while working on one project, you need to use another project from within it. Perhaps it's a library that a third party developed or that you're developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other. Git addresses this issue using submodules. Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

Let's start by adding an existing Git repository as a submodule of the repository that you're working on. To add a new submodule you use the **git submodule add** command with the absolute or relative URL of the project you would like to start tracking. In this example, we'll add a library called "Earth".

```
git submodule add git@git.geomar.de:<your-username>/Earth
```

(Make sure to use the URL for your repository rather than the dummy-one provided here)

By default, submodules will add the subproject into a directory named the same as the repository, in this case "Earth". You can add a different path at the end of the command if you want it to go elsewhere.

If you run **`git status`** at this point, you'll notice a few things:

- First you should notice the new **.gitmodules** file. This is a configuration file that stores the mapping between the project's URL and the local subdirectory you've pulled it into.
- The other listing in the git status output is the project folder entry. If you run `git diff` on that, you see something interesting: Although Earth is a subdirectory in your working directory, Git sees it as a submodule and doesn't track its contents when you're not in that directory. Instead, Git sees it as a particular commit from that repository.

Notice: If you have multiple submodules, you'll have multiple entries in this file. It's important to note that this file is version-controlled with your other files, like your .gitignore file. It's pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.

You have to **commit** and **push** these changes:

- `git commit -m "adding a subdirectory to my project"`
- `git push origin`

9.2 Cloning a Project with Submodules

Here we'll clone a project with a submodule in it. When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet. The Earth directory is there, but empty. You must run two commands: **`git submodule init`** to initialize your local configuration file, and **`git submodule update`** to fetch all the data from that project and check out the appropriate commit listed in your superproject:

Now your Earth subdirectory is at the exact state it was in when you committed earlier.

There is another way to do this which is a little simpler, however. If you pass --recursive to the `git clone` command, it will automatically initialise and update each submodule in the repository.

- `git clone --recursive git@git.geomar.de:<your-username>/Earth`
  (Make sure to use the URL for your repository rather than the dummy-one provided here.)

To get more informations about "Working on a Project with Submodules" see the following tutorial https://git-scm.com/book/en/v2/Git-Tools-Submodules.

(Adopted from: https://git-scm.com/book/en/v2/Git-Tools-Submodules)

Some chapters from this tutorial are adopted from: http://vulcanus.geomar.de/%7Ewrath/edu_git-intro_v1.0/01-basics/index.html

If you have other questions or comments please contact the data management team:

Phone:	0431 / 600 2294
E-Mail:	datamanagement@geomar.de
Location:	Eastshore / Building 1/ Entrance 2/ Room 110 - 112
Adress:	GEOMAR Helmholtz Centre for Ocean Research Kiel Wischhofstr. 1-3 24148 Kiel | Germany