TAM DataHub

User Manual

Introduction to the common research infrastructure and workflows of the cluster initiative The Adaptive Mind

The DataHub Data Stewards
Stefan Lenze[1,2]
Julia-Katharina Pfarr[3], SFB/TRR135 - INF Project

Contact & Support
datahub@uni-marburg.de

Available Online
https://uni-marburg.de/pMoBmv

Cite as
Lenze, S. & Pfarr, J.-K. (2024) TAM DataHub User Manual: Introduction to common research infrastructure and workflows. Version 1.1. https://doi.org/10.60834/tam-datahub-4

[1] Philipps-University Marburg, Department of Physics, Neurophysics
[2] Philipps-University Marburg, University Computer Center, Research-related Applications
[3] Philipps-University Marburg, Department of Psychology, General and Biological Psychology

Welcome to the DataHub!

In this manual, you will find an overview on the DataHub, introductions to individual resources and services and workflows to help you using the DataHub. The Manual is meant to be used after you consulted the local Data Stewards and already received an introduction by e.g., a workshop to all services.

If you are new to the DataHub, don't hesitate to contact us for support via our email: datahub@uni-marburg.de.

Your DataHub Team

DataHub News

TAM GitLab

A dedicated TAM GitLab for use within the DataHub is now available (Marburg University network / VPN required). The TAM GitLab offers access to the TAM-owned data storage at the secured high performance computing infrastructure of MaRC3 and MaSC.
We encourage all DataHub users to store and manage their research data and code on the TAM GitLab!

GitLab uses Git LFS to handle large (binary) files without degrading performance or stability. Please refer to the Workflow section and the FAQ of this manual for information and streamlined instructions on using GitLab and Git LFS.

TAM DataHub Repository

The TAM DataHub (DSpace) publication platform is currently being set up for TAM and allows for various scientific output to be published with rich metadata and persistent identifiers (DOIs).

It is available at **tam-datahub.online.uni-marburg.de** and has started pilot (test) operation. Pilot users are highly welcome! Please contact your Data Stewards if you are interested.

This manual contains a service description and a user workflow documentation (work in progress).

What is the DataHub?

The DataHub is an infrastructure project of the "The Adaptive Mind" (TAM) consortium in which research groups in the field of psychology and neuroscience, based at multiple hessian universities work together. TAM is a cluster project funded by the Hessian Ministry of Higher Education, Research, Science and the Arts from 2021 to 2025.

Guided by the "FAIR" principles, the DataHub offers resources, services and support to affiliated researches on various levels to ease collaborative work:

- Central storage and compute resources: MaSC / MaRC3,
- Services that allow efficient usage of these resources: JupyterHub, TAM GitLab and the TAM DataHub Repository (DSpace)
- Support on how to use the services in a way that fits the needs of the individual project.

The goals and the functional scope of the DataHub are illustrated in the figure below. Important features and principles of the DataHub are outlined in the TAM Data and Code Policy.

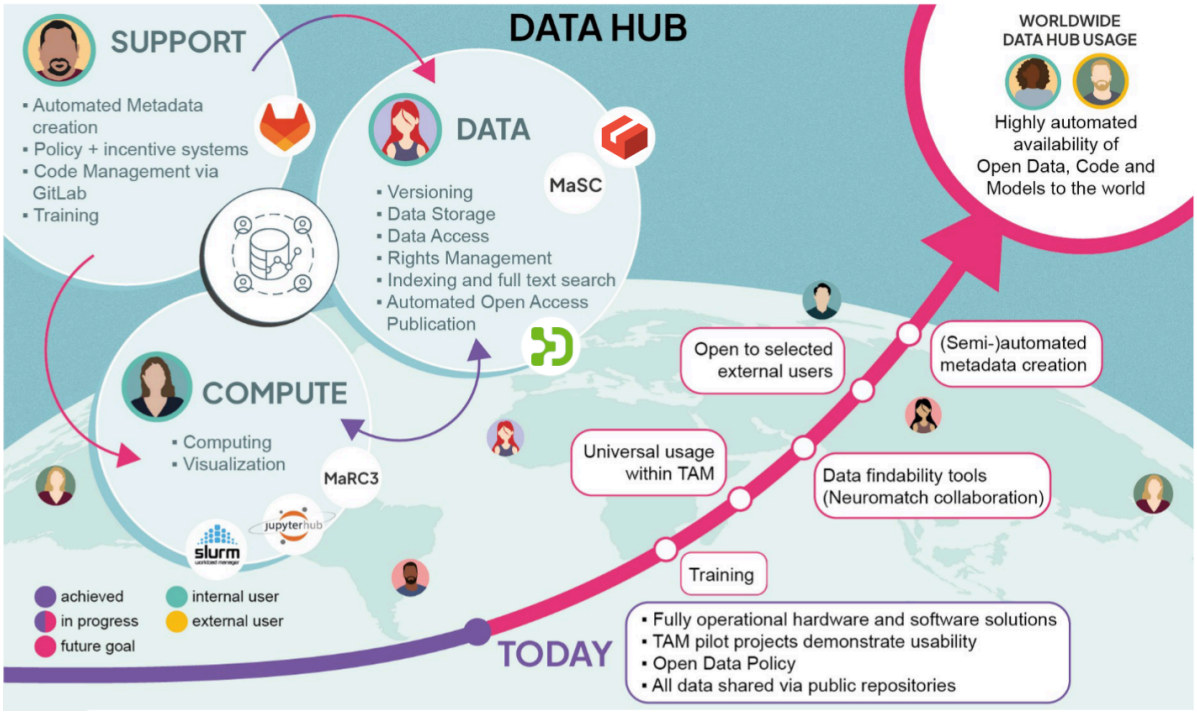


Figure 1: Functional scope and goals of the DataHub.

What is FAIR about the DataHub?

FAIR means, your data should be easily **F**indable, persistently **A**ccessible, broadly **I**nteroperable and thus **R**e-usable. Sharing "FAIR" data is a sustainable way of handling your data and allows you to get the most value out of your work for yourself, your organization and the greater community.

In practice this may be a real challenge, especially if domain-specific standards are not settled and/or persons from various fields work together. With our services and support we aim to empower you to practice and improve on FAIR data management. Therefore, we kindly ask you to use certain tools, workflows and principles of organization and description of data whenever possible.

That sounds difficult? Don't worry - you are not alone! The DataHub Team stands by to offer training, answer your questions and find solutions together with you.

Support

The DataHub Team consists of people with backgrounds in science, IT and research data management. We offer face-to-face **workshops**, recorded **tutorials**, group and individual **consulting**, and general **support** on DataHub infrastructure, tools and research data management.

You are most welcome to contact us via datahub@uni-marburg.de. You may also get in touch with the Data Stewards directly:

- Julia-Katharina Pfarr (NOWA / SFB 135)
- Stefan Lenze (TAM).

The DataHub Architecture

This section gives an overview on the DataHub regarding its structure and services, how they interact and what purposes they serve. If you are new to the DataHub or if you don't have experience working with git or the HPC environment, please contact the DataHub Team or your local Data Stewards before starting to use it.

The DataHub builds on its own storage and compute resources and cooperates with the University Computer Center in Marburg. It enables affiliated researchers to work in a common infrastructure with similar workflows and thus allows efficient collaboration.

Compute Infrastructure

The DataHub compute resources are integrated into and managed by a larger high performance computing (HPC) cluster, the Marburg Compute Cluster (MaRC3). Depending on your needs and experience, there are different ways to access and use the compute resources some of which are shown in Figure 1.

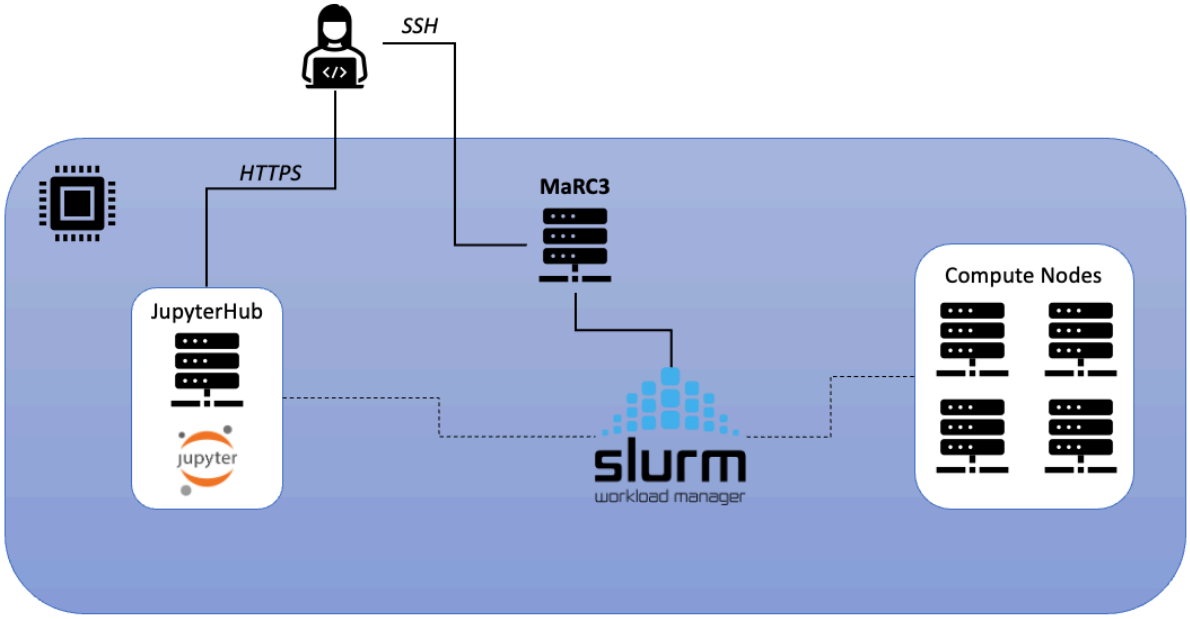


Figure 1: Ways to use the compute infrastructure: 1) Access the JupyterHub via web browser (HTTPS) to create an interactive JupyterLab session on the cluster, 2) access the cluster directly via command line interface and SSH to run jobs via the management system SLURM.

For users who like to work interactively with their data, the JupyterHub might be a good choice. It can be accessed at https://marc3.jupyter.uni-marburg.de and allows you to use JupyterLab / Jupyter Notebooks for comprehensive, multi-language data science. For more information see JupyterHub section.

Users who have elevated requirements on computational power can establish an SSH connection and interact with the Linux shell of MaRC3. You can access MaRC3 via command line using: `ssh -p 223 <username>@marc3a.hrz.uni-marburg.de`.

Compute jobs can be enqueued directly at workload management system, allowing you to be very specific on kind and amount of computing power you need. This approach allows you also to scale your processing, especially by parallelizing it. For more information see HPC section.

Storage Infrastructure

Important Note:
The **TAM GitLab** for use within the DataHub is now available (Marburg University network / VPN required). We encourage all DataHub users to store and manage their research data and code on the TAM GitLab! For publication and external findability of research output, the **TAM DataHub Repository** (DSpace) is currently being set up and test operation will start shortly.

These services replace our previous local GIN platform as further software development of the GIN project and sustainable operation was not foreseeable.

Data in the TAM GitLab are actually stored in a secured high performance computing infrastructure at the Marburg Storage Cluster (MaSC). The storage cluster contains the TAM storage resources which currently amount to about 700 TB total usable capacity. The MaSC itself is directly connected to the MaRC3 HPC which brings the advantage of fast data transport for computation. Figure 2 illustrates ways to access the common storage.

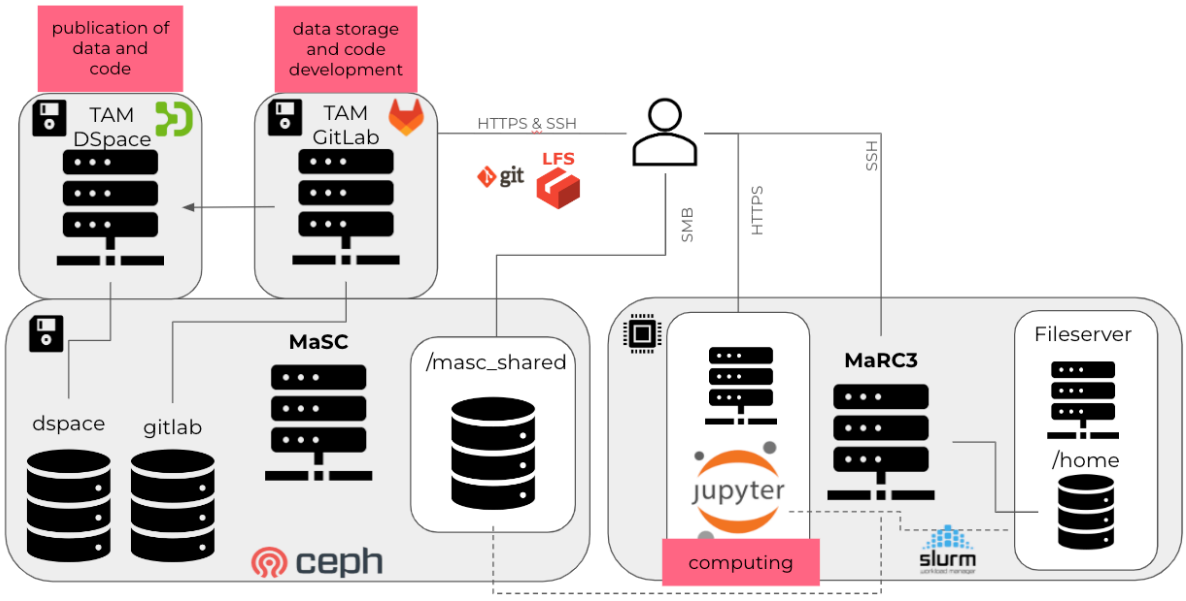


Figure 2: Routes to access the storage infrastructure.

Users may interact with TAM GitLab via web browser or command line interface. Interaction with GitLab is also possible and potentially faster, when working on MaRC3 either directly via SSH, or by using JupyterHub. Git LFS is used for efficient file storage in a git-based workflow. Using MaRC3 provides access to additional storage pools like /home or /masc_shared (see HPC section for details). Client-side software may be used to browse and interact the MaRC3 file systems (including MaSC) conveniently via SSH. In special cases, mounting /masc_shared via SMB is also possible.

Versioning and Provenance Tracking

To prevent getting lost with your own data or other person's data, we believe that version control and tracking of data provenance are the most important baselines. At the DataHub we decided to use git technology to this purpose and have set up a GitLab instance for TAM, which in conjunction with Git LFS enables efficient workflows with git. While the basic git is optimized to work collaboratively on code (or text in general), git-LFS is optimized to work also on large binary data (like videos or any arbitrary recordings).

Git helps you to track, reproduce and integrate different versions of a dataset both in a sequential way ("history") and also in a parallel fashion ("branches"). This means you can try different variants of an analysis in parallel, work with them, and keep a reusable record of how these variants evolved (provenance tracking).

Data Organization and Description

To make understanding and re-using data and code easier, we ask participating researchers to adhere to some common principles of data organization and description.

We ask affiliated researchers to use a certain template folder structures which follow the philosophy of the so called "**TONIC**" template for project repositories. Concerning actual data repositories, we strongly recommend using the **BIDS** standard whenever possible and as early as reasonable in the succession of data processing. We also ask you to provide at least basic **metadata** for each dataset that is uploaded to the shared storage.
→ Please find details and templates in the Data and Code Management section.

As TAM is much about collaborating and sharing data in a sustainable way, both internally and with a larger community, common data organization and description are very important aspects.

GitLab

What is GitLab?

GitLab is a web service for hosting / managing git-based repositories and has a multitude of additional features for software development. It is thus comparable to GitHub. In its community edition, it is free, open-source software. Features include:

- Hosting and managing git repositories
- Storing large (binary) data with Git LFS
- Online integrated development environment (IDE)
- Issue tracking
- Project management: Kanban boards & milestones
- Wikis for documentation
- Automatization pipelines: CI/CD
- Management of packages and containers
- Web hosting with GitLab pages.

We encourage all DataHub users to use the TAM GitLab!

Fundamentals

Managing Permissions

From a high-level perspective, GitLab differentiates *groups*, *users* and *projects*. This allows a mostly self-administered management of permissions:

- A group can have various sub-groups in an arbitrary hierarchical structure.
- Users can be members of these groups and may have different roles which determine the extent of permissions with respect to a given group (and potential sub-groups).
- Projects can be created by users in their own domain ("namespace") or in the namespace of any group in which they have the corresponding role / permissions (see Figure 1).

- Depending on a user's role with regard to a group or a project, the user may attribute other users with with certain roles (permissions) for that group / project.

To start a project on GitLab, you may create a repository under your account or, provided you are member of a group, in the namespace of that group (see below). If you have insufficient rights, you may ask your group-administrator to increase your permissions or create a project for you. Depending on the sensitivity of your data, you may grant general access to the project ("public"), restrict access to authenticated users of the TAM-GitLab ("internal"), or - in case of sensitive data - keep the project to yourself / your group and invite other members explicitly ("private"). As the TAM Gitlab is restricted to the University Network, "public" only means persons, who have access to this Network (including VPN).

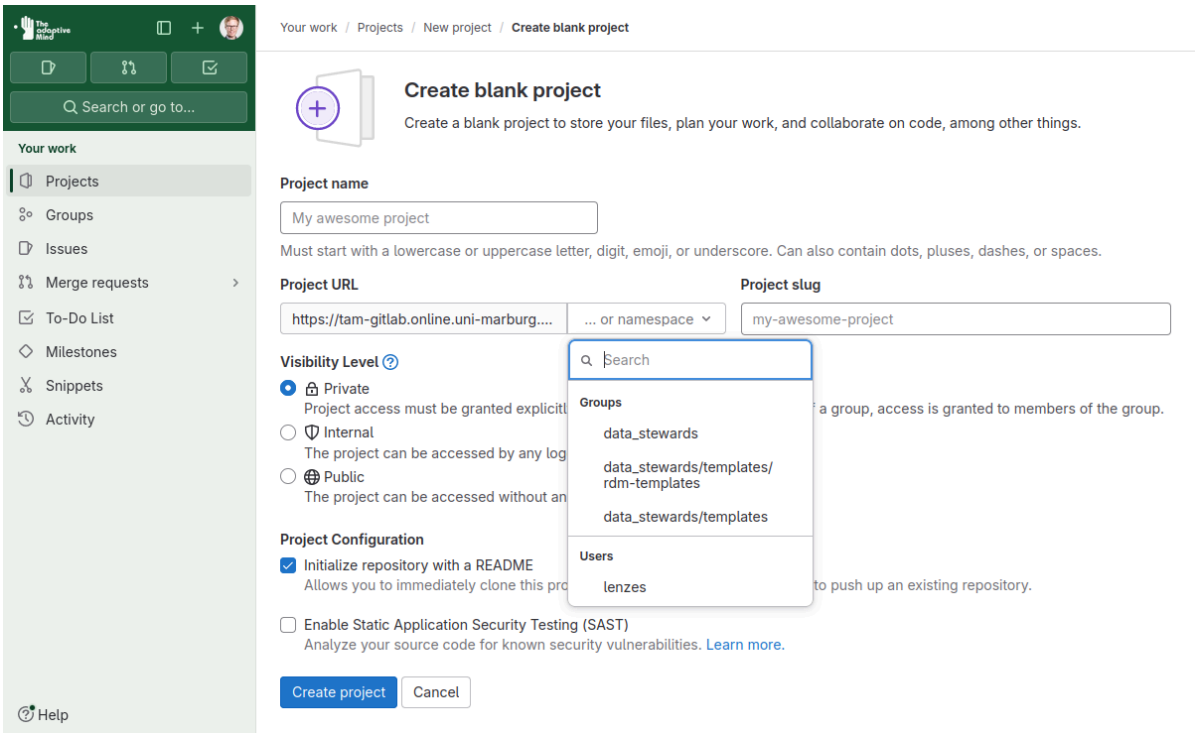


Figure 1: Creating a new project / repository in GitLab.

GitLab Projects and Repositories

Each project has a repository (a git-controlled, online file storage) at its heart (see Figure 2). You can work on the files in this repository either online in the GitLab IDE, or use git to clone the repository to your local working environment, use whatever tools you like, and commit and push changes back to the repository at GitLab.

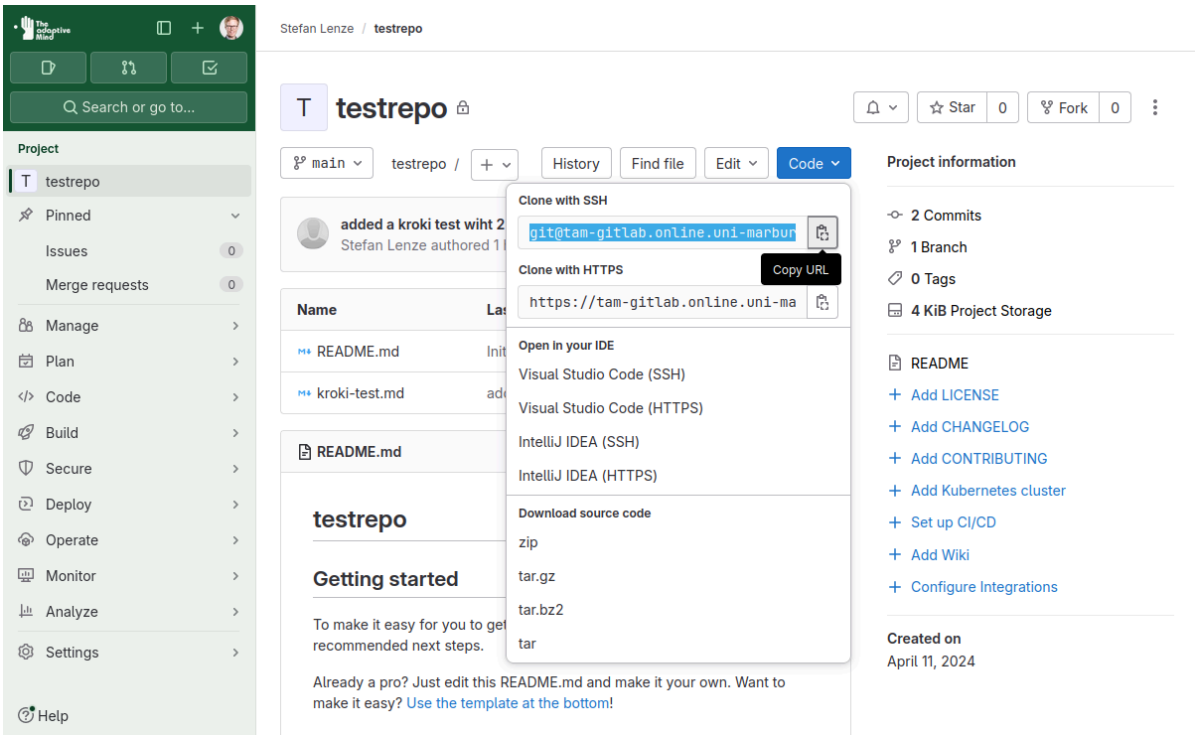


Figure 2: A repository on GitLab.

GitLab started using VS Code as Web IDE (shown below), which may add interesting features for some users.

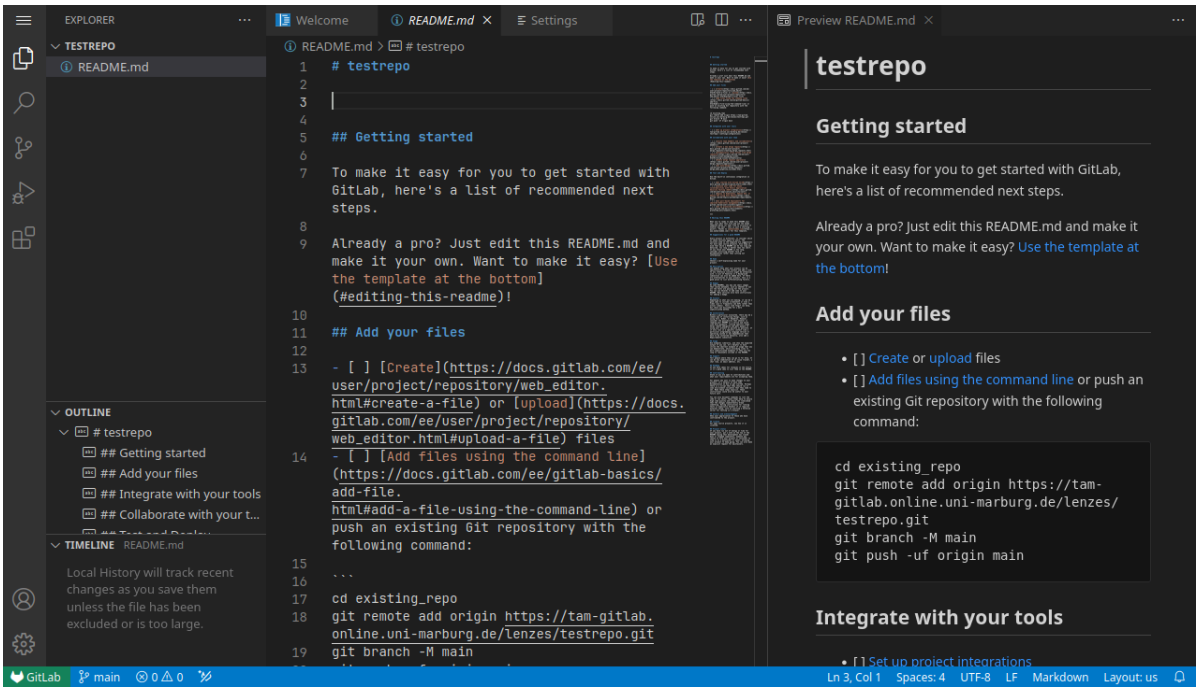


Figure 3: Editing files in the GitLab's online IDE (Visual Studio Code).

Describing the functionality of git and GitLab in detail is beyond the scope of this manual. For this, we offer specific training and online materials (see below).

Using the TAM GitLab

With the TAM GitLab, we have a single platform to store and manage project data (with Git LFS), for individual or joint code development, and for simple exchange within the DataHub. Another bonus of the TAM GitLab is that we Data Stewards are actively monitoring the platform. We can help you to improve the consistency and reusability of your projects and provide templates and other information.

The TAM GitLab builds on the TAM-owned resources in the high performance infrastructure of MaSC and MaRC which allows us to offer a substantial amount of already paid storage in an environment with elevated security.

> ⚠ **Don't confuse GitLab instances!**
>
> As there are multiple instances of GitLab, we particularly ask you to **use the TAM GitLab but NOT THE MARBURG UNIVERSITY GITLAB** for research data that is in the scope of the DataHub.
> The Marburg University GitLab also has stricter policies on available storage, project count and data sensibility!

We trust in **fair-use** of the provided resources and have not set quotas. We ask users to go along with the template group hierarchy based on collaborating universities and research groups. However, there will be cases in which it makes sense to organize overarching projects separately. In such cases, users may create top-level groups (see below) on their own.

In the Data Stewards Group we provide **templates** to organize the internal structure of your projects / repos. Please use (i.e. fork from) these templates when creating own projects. This will help us all to understand and reuse each other's work more efficiently. Whenever possible please adhere to the **BIDS** standard for organizing your data. This is a well-supported, machine-actionable standard which systematically includes relevant metadata.

Since the TAM GitLab will include sensitive data, it cannot be opened directly to the internet. Thus, using Marburg staff accounts and VPN with two-factor authentication (2FA) will still be required, and access is restricted to DataHub users (and collaborators). See DataHub Access, if you don't

have access yet. Persons who were granted access to the previous GIN platform already have access to the TAM GitLab.

If in doubt, discuss the organization of your projects and data with your Data Stewards.

FAQ, Troubleshooting and Support

- See FAQ of this manual.
- The NOWA Support Page and the NOWA School have a collection of (recorded) tutorials, workshops and online materials that also cover git and GitLab.
- If you face issues with establishing connections, see the workflow section of this manual.

JupterHub

What is JupyterHub?

The Jupyter**Hub** enables you to use the popular Jupyter**Lab** interface without bothering to install and maintain software on your local computer. Instead, the JupyterHub runs on the MaRC3 high performance computing (HPC) cluster and allows you to start private JupyterLab sessions. Thus, it provides you an easy-to-use interface to use the HPC infrastructure in an interactive way.

Jupyter**Lab** is a free and ope-source web-service, specialized on scientific computation and data analysis tasks within so called Jupyter Notebooks. A Jupyter **Notebook** is structured into cells that may contain code or text (usually Markdown). It allows you to add rich description next to your code and to interact conveniently with data, code and results. Currently there is a python kernel available by default (only python code can be run). Kernels for other languages like R can be installed from a virtual environment.

The JupyterHub is generally meant for testing, development, and exploratory data analysis. Large calculations should be translated into a sourcefile and started as a batch job from the regular scheduling system (SLURM) of the cluster (see HPC).

For Details see MaRC3 documentation (requires login, accessible from Marburg University network / VPN). The public JupyterHub service description, terms of use and privacy statement can be found here.

Access to the JupyterHub

The JupyterHub can by accessed at: https://marc3.jupyter.uni-marburg.de/ (login with Marburg university staff account name + password).

The DataHub provides access to the MaRC3 cluster, which includes permission to use the JupyterHub. If you do not yet have a staff account at Marburg University **or a MaRC3 authorization**, see the DataHub Access section of this manual.

Select a Job Profile

Once logged in, you can start a JupyterLab session (server). Depending on your needs, you can select one of three different job profiles. Your choice will determine the number of CPU cores and RAM that will be reserved for you during the runtime of your session (see figure below).

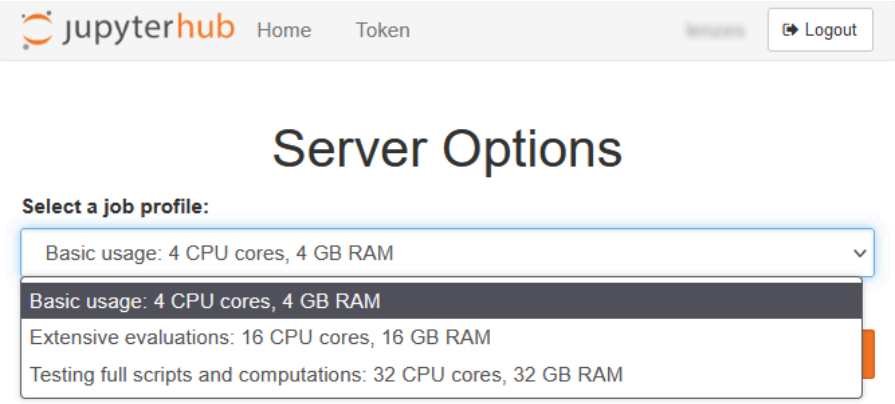


Figure 1: Selecting a performance profile to start a JupyterLab session.

In the spirit of fair use, you should only use the 16 or 32 core option when you actually need it. Otherwise, you might block resources unnecessarily.

How to use JupyterLab

After starting a JupyterLab session, you have access to the various file systems of MaRC3 and you are directed to your personal home directory. Using the launcher, you can start with a new Jupyter Notebook (*.ipynb), open a terminal session or create a new markdown or python file. An example is shown below.

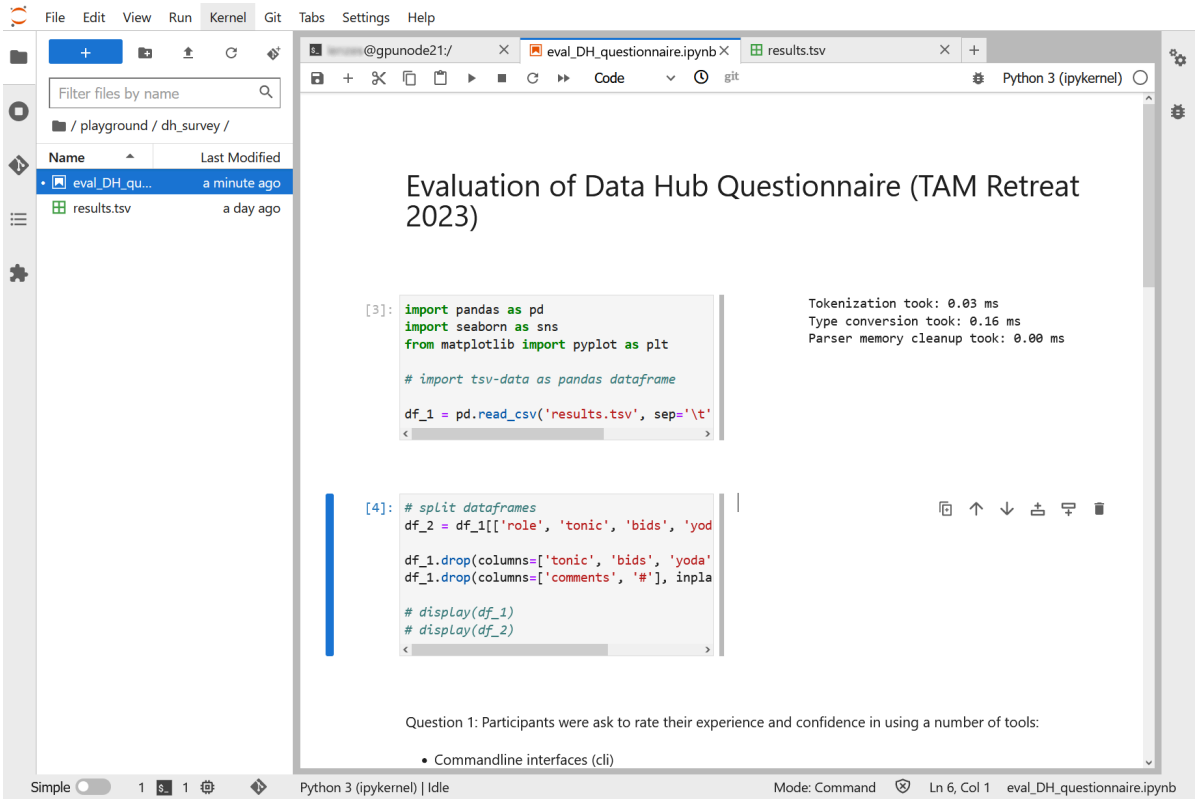


Figure 2: Working on a Jupyter Notebook at MaRC3.

> **Jupyter training**
>
> You can find a recorded Jupyter introduction (plus presentation materials) held by Dr. Christian Berger as part of the HeFDI Data Talks 2024 here.

FAQ, Troubleshooting and Support

- FAQ can be found here or in the FAQ of this manual.
- Detailed information on Jupyter products can be found on the Jupyter website.
- Using the JupyterHub actually means using the HPC infrastructure. Potential issues might also be rather general HPC topics than specific issues with JupyterHub. It could be helpful to search also the service description and FAQ on HPC.

High Performance Computing at MaRC3

Detailed and up-to-date information on the "Marburger Rechen-Cluster 3" (MaRC3) and on the Marburg storage cluster (MaSC) can be found at the HPC-User documentation (log-in required with Marburg University staff account). This page summarizes key aspects.

The below figure gives a brief overview of the hardware currently available in the cluster. Since new hardware is almost constantly being added to the cluster, the below information is prone to become outdated.

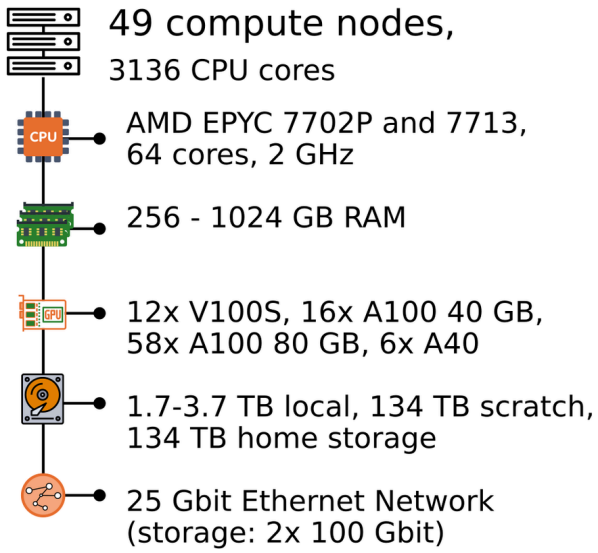


Figure 1: Overview of MaRC3a hardware (updated: June 2023)

Access

There are different ways to access the MaRC3 HPC cluster, including direct access via terminal and SSH connection or via browser and Jupyterhub. The general scheme is illustrated in figure 2.

- Access to MaRC3 generally requires a Marburg University account and a separate registration for HPC usage. For information on applying for these see the getting access section of this manual.
- Make sure you are connected to the network of Marburg university directly by cable / WLAN or via VPN.

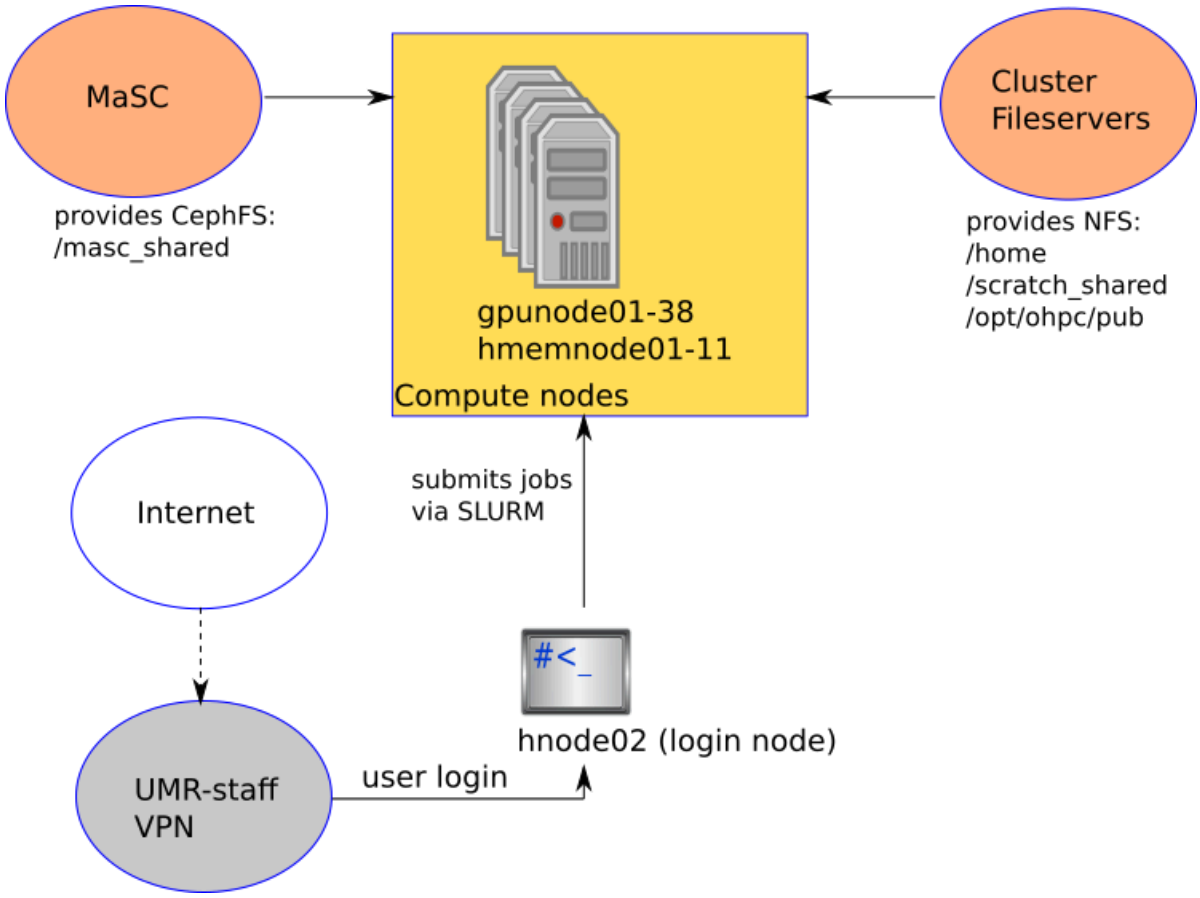


Figure 2: Schematic cluster stucture (user-side view).

In case you want to connect via terminal:

- Initially, you may need to change your user shell (the specific command line interface) to "bash" using this online form (2FA required).
- Start a terminal session and establish an SSH connection to MaRC3 by entering `ssh -p 223 USERNAME@marc3a.hrz.uni-marburg.de` (replace USERNAME by your Marburg University staff account name).
- You will be directed to a "log-in" node of the cluster where you have access to the various file systems which are available from maRC3.
- From here, you may use the SLURM management system to create computing jobs which are queued and executed by a pool of "worker" nodes.

Please do NOT use the log-in nodes for computation. Their resources are needed otherwise.

Usage of MaRC3

You can use the MaRC3 resources in different ways depending on the needs of your project and your experience.

- If you like to interact with your code and your project does not need demanding (parallel) computation, using the **JupyterHub** might be a good choice. This might be the case in (early) data exploration or for development purposes.
- If you need to interact at runtime with your code but JupyterLab is not your preferred environment, an **interactive job** is a valid choice. There are also options to enable **graphical interaction**. Ideally, this should be avoided though, as graphical sessions are generally quite slow and usually not very efficient. A notable exception is the usage of **RStudio IDE** (see MaRC3 documentation).
- If you like to take advantage of specialized hardware and parallel computation (e.g. run the same code on many datasets at the same time), scheduling batch jobs likely is a good choice.

In the background, there is a management and scheduling system (Slurm), which files the concurrent jobs of all users to queues and prioritizes them to ensure that resources are efficiently used and everyone has a fair chance to use the cluster. At the same time, this means that a job might not be executed immediately. Waiting time depends on multiple factors, including the cluster's current load, the requested resources for a job (cpu cores, ram, job duration, ...), and the user's past jobs (accounting).

As a TAM Member, you have access to a specialized so-called "owner-partition" *owner_tam* which gives you privileged access to the resources which were sponsored by TAM. For details see MaRC3 documentation.

Storage at MaRC3 and MaSC

Once you log in (via SSH or via JupyterHub), you have access to multiple file systems that serve different purposes and thus differ largely in terms of capacity, access speed and persistence. Knowing which file system to use for what is crucial for getting good job performance and avoiding loss of data!!

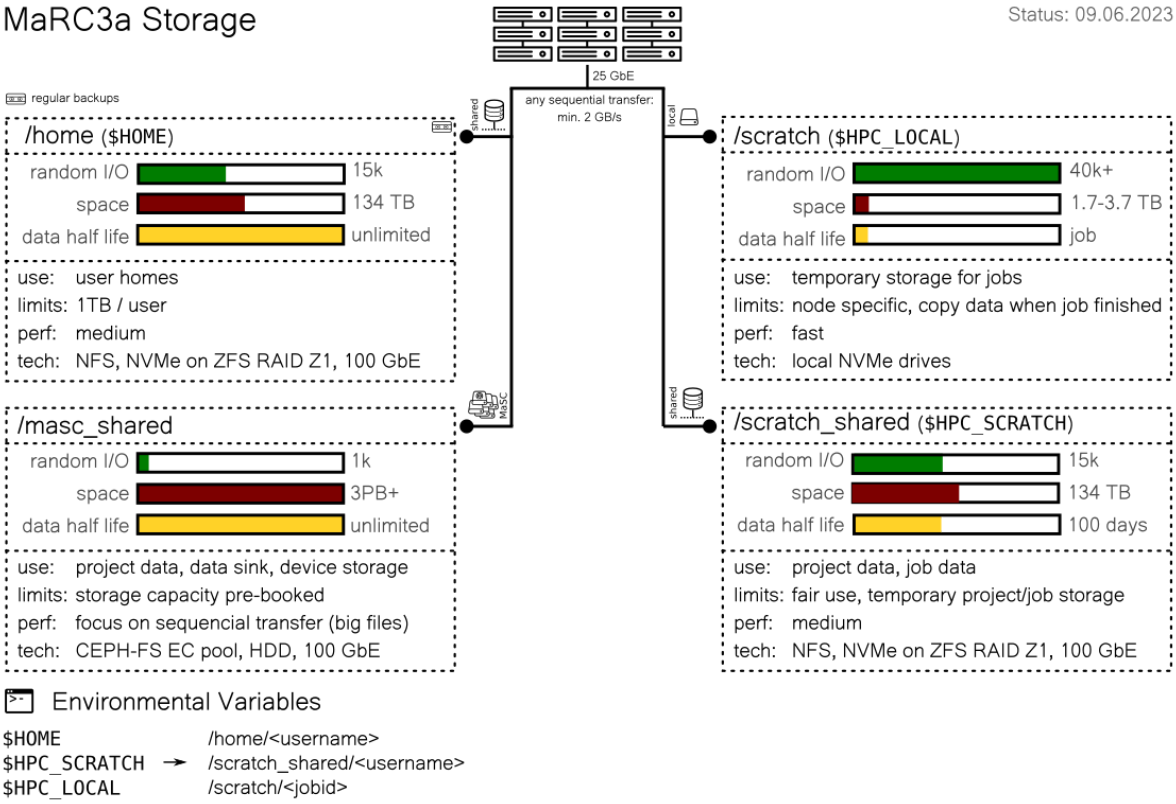


Figure 3: MaRC3a file systems (figure by Marcus Lechner; updated by René Sitt).

The general principle is as follows:

- Large datasets are stored at high capacity locations **at MaSC** which come with the disadvantage of comparatively slow access rates (**TAM GitLab** or */masc_shared*).
- Before the actual computation, data is cloned / copied to a smaller / less persistent but highly accessible (very fast) storage location **at MaRC3** like */scratch* or */scratch_shared*. This prevents slowing down your computation by limited data access rates and thus saves your time and the time of other persons who would also like to use the compute cluster.

The compute and the storage cluster, MaRC3 and MaSC are located in close proximity, thus allowing optimal data transport. Because the storages at MaSC and MaRC3 are shared resources, we kindly ask you to respect fair usage.

The Filesystem */masc_shared* is directly mounted at MaRC3 and contains dedicated storage areas for groups and research projects. Persons affiliated with TAM get access to the */masc_shared/in_tam* directory. Upon request, direct access to shared directories at MaSC via SMB (locally mountable network drive) can be granted.

Avoid storing data directly at /masc_shared

This is a deprecated, NOT recommended way to use the DataHub which undermines version control and has general data management issues (including no regular backups)!!

It is highly recommended to rather have data "permanently" stored in a central remote repository at TAM GitLab and make it available for computation on a fast "local" storage on demand.

Environments and Software

There is a system of centrally managed environment modules which provide basic standard configurations for certain use cases or enable you to use specific software. The entry point is the `module` command.

Table 1: Module commands (from MaRC3 documentation)

Command	Effect	Notes
module avail	list loadable modules	does not display dependency subtrees for modules that are not currently loaded
module list	list currently loaded modules	
module load	add to environment	synonymous to 'module add '
module unload	remove from environment	synonymous to 'module del '
	module spider	list ALL modules
module spider	list all versions of	can also be a search term (i.e. 'module spider gnu' will find 'gnu7' and 'gnu9' modules)
module purge	unload all currently loaded modules	useful in job scripts to start with a defined (empty) environment before loading needed modules

convenience functions that are not listed here

An important module is the **miniconda** module which is an open-source, cross platform and multi-language environment and package management system. Using the miniconda module (`module load miniconda`), you can create and enter user-defined (private) environments and install software within them. It also allows you to have specific software versions in parallel (e.g. different python versions). Please refer to the (Ana)conda section of the MaRC3 documentation for details.

If you need software that requires higher privileges or licenses, you can contact the MaRC3 team (see below).

FAQ, Troubleshooting and Support

- See FAQ of this manual.
- Support: Questions, problems, installation requests and suggestions can be sent to the DataHub or the MaRC3 Team.
  Please have the DataHub team in CC, if you are mailing directly to the MaRC3 Team.

TAM DataHub Repository

Contents and publication workflow reference

This article outlines the purpose, features and status of the TAM-DataHub Repository. Instructions for usage are part of the workflow section of this manual.

Purpose and Features

For data and code publication with persistent Identifiers (DOI), we are currently establishing the TAM DataHub Repository, based on the modern and widely used DSpace software. This research data repository further simplifies early sharing of data and code and is tailored to the needs of researchers in the field of psychology and neuroscience.

Our TAM DataHub Repository fills the gap between our institutional repositories like data_UMR, JLUdata and TUdatalib, general research repositories like OSF or Zenodo, and discipline-specific repositories such as OpenNeuro or ZPID. You are free to continue using such repositories in addition to our DataHub Repository and you are welcome to contact your local Data Stewards to select suitable publication platforms.

The TAM DataHub Repository has started pilot (test) operation with the following core features implemented:

- Access the repository publicly via: tam-datahub.online.uni-marburg.de
- Search the repository for relevant information via full-text search.
- Authenticate with your Uni Marburg account via Shibboleth to deposit publications.
- Upload one or multiple files or archives via web-browser and control access with submission- and file-based options (open access, restricted access, embargo, ...).
- Operation in the secured HPC infrastructure of MaSC and MaRC enables storing data with normal and elevated sensitivity requirements (not during pilot (test) phase, see below).
- Submissions up to several hundred gigabytes (and above upon request) can be accepted.
- Daily backups of all data and of the whole platform ensure availability of your archived and published research.

- Use field-specific metadata schemata to describe your text- and dataset-publications. The schemata largely base on established schemata like Dublin Core for long-term interoperability. Specific item types (e.g. Text Publications, Datasets, Persons, Projects, etc.) are available, each with corresponding metadata. The modular submission process has some basic, mandatory metadata and extensive, optional metadata to provide the level of detail that fits your needs (see figure).
- Define relations to connect related publications, data and code, archived documents and responsible persons, projects and organizations.
- Use pre-selected values and controlled vocabularies to enable consistent and meaningful annotation.
- Get feedback from the DataHub Data Stewards who act as curators and can help you to optimize annotation of your publication.
- Obtain persistent identifiers (DOI) for your publications which also makes them findable via DataCite Metadata.

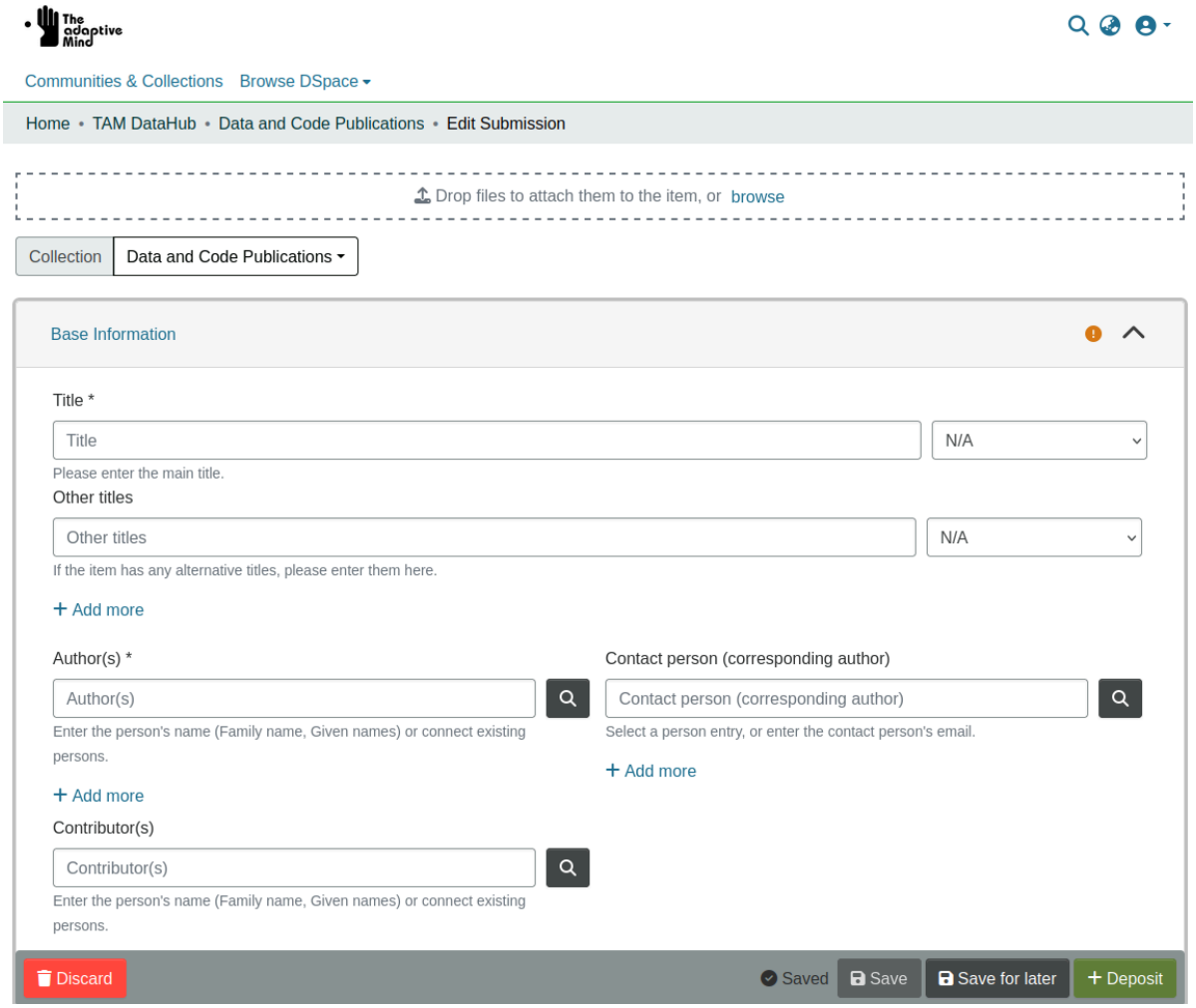


Figure: Submitting a Dataset in the TAM DataHub Repository.

Further features will be added continuously, some of which are listed below:

- Training resources
- User interface customization
- GitLab to DSpace publication pipeline
- Integration with external data sources

Pilot Operation

The TAM DataHub Repository has started pilot (test) operation. **Test users are highly welcome!** Please contact us via datahub@uni-marburg.de.

No sensitive data!

During pilot phase, **PLEASE DO NOT UPLOAD ANY SENSITIVE DATA!**

Temporary downtimes and updates

Expect temporary downtimes and updates including but not limited to:

- User Interface, especially item pages
- Improvements in metadata schemata
- Changes in the submission workflow

Manual authorization

Currently, users need to be authorized manually to publish at TAM DataHub. This should usually not take longer than two days. If you have logged in first time and are still not able to create submission within this time, please contact your Data Stewards.

DataHub Access

Please read this section carefully in order to send a valid request for DataHub access and don't miss any steps.

Uni Marburg Staff account for Non-Uni-Marburg TAM members

Account request

Researchers who want to work with the DataHub need a staff email account at Marburg University. If you are not primarily affiliated with Marburg University, an account needs to be created for you. For this, please reach out to datahub@uni-marburg.de and provide the following information about yourself and your work:

- Your email you want to receive messages regarding your account with
- First- and last name(s)
- Your birthday
- If you want communication in german or english
- Your status (PhD Student, postdoc, PI, etc.)
- The start- and end-date of the work you are doing for the projects with primary access to the DataHub services (i.e., the time of your collaboration)
- Your work address.

After applying for an account you will receive a letter at your work address with credentials that allow you activation your account. Note that these are valid only for two weeks. If you don't activate your account by this time, you need to set up a video call with the University Computer Center (HRZ) Marburg to verify your account. Due to the postal process, the procedure may take a few days.

Two-Factor Authentication after receiving your Uni-Marburg Staff Account

Additionally, you will need to **actively apply** for a 2-factor authentication (2-FA) on the website (go to Token beantragen) of the HRZ Marburg. You can choose between an App Token or a TAN token. You can choose a video call for authentication, then activation will be faster than via post mail.

After you received your 2-FA, you can use the DataHub services with your email account, password, and 2-FA.

Uni-Marburg Staff

If you already have a staff account at Marburg University but need additional permissions to use the **TAM-GitLab** or the **MaRC3 cluster** and **JupyterHub**), please also request these via datahub@uni-marburg.de.
In this case please also provide your:

- First- and last name(s)
- Birthday
- Marburg University staff account name.

Access to VPN

The University Computer Center Marburg provides instructions to set up VPN connection for various operating systems here.

Access to JupyterHub and MaRC3

The MaRC3 cluster is a powerful and valuable shared resource. Users have to adhere to the user agreement and to fair use principles. Please be aware that there might be restrictions on the usage, especially for users in / from foreign countries, due to specific regulations on supercomputers like EU dual-use directive, embargoes, etc.

By default we provide TAM DataHub users with permissions to use both the TAM GitLab and the JupyterHub / MaRC3, where TAM users have access to a privileged "owner partition". You will receive a separate email from the University Computer Center Marburg on this. You can find an introduction to MaRC3 HPC usage here.

Data and Code Management

How to Organize and Describe Data and Code?

As reasoned in the introduction and the architecture of the DataHub and in line with the TAM Policy, some conventions are made to advance towards "FAIR" data and to achieve more sustainable and reproducible science. Key aspects are: - Describing data with rich metadata, ideally using standardized schemata (like BIDS). - Applying (domain-) standards to organize data and code (like BIDS). - Aggregating or linking research products and everything needed to reproduce them into modular / self-contained units. - Using open and wide-spread file formats whenever possible.

These things are usually not easy to solve and often common principles have to be tailored / integrated to satisfy the needs of a specific project. You are very welcome to contact your local Data Stewards to find custom solutions for your project.

Brain Imaging Data Structure (BIDS)

Concerning data organization and annotation, the TAM Policy states to use the BIDS standard (see BIDS Starter Kit) whenever possible and as early as reasonable in the succession of data processing. While BIDS started out as a standard in neuro-imaging, it supports more and more modalities by adding extensions.

On the one hand, BIDS provides fixed conventions how to organize data according to processing state, data subjects, sessions and modalities. It describes how detailed metadata can be included in a way that is actionable to both humans and machines. This creates the ground on which standardized code / software can be developed and uses by everyone and reduces the need for adaptations. On the other hand, its common principles and modality agnostic files are flexible enough to include rich custom meta-data on various levels.

But also if there is not yet an extension (or just an extension proposal) for a specific modality, BIDS bases on well-founded common principles which are at least worth taking a look at, if you are designing your data organization. You can always organize and annotate your data in a "bidsy" way, even if there is no specification for your data type, yet. Your local Data Stewards are happy to give you an introduction to BIDS as it became very comprehensive over the last years and the start can be a bit overwhelming.

TONIC research folder structure

We kindly ask you to use a certain folder structure called **TONIC** for project (top-level) repository structure. A template can be forked from within the TAM GitLab. Here you'll find a template for organizing multiple projects in one repository (link) or for organizing a single project (link). The folder names should be self-explanatory but if you have any questions on how to use the template, please consult your local Data Stewards.

Data Management Workflow and Tutorials

The above mentioned structures and standards are meant to be used within the TAM GitLab during active project development. Detailed instructions and explanations on how to do that can be found in the sections DataHub Workflow and in the soon available Tutorial Videos.

Publication

For data and code publication with persistent Identifiers (DOI), we are currently establishing the TAM DataHub Repository, based on the modern and widely used DSpace software. This research data repository further simplifies early sharing of data and code and is tailored to the needs of researchers in the field of psychology and neuroscience.

Our TAM DataHub Repository will fill the gap between our institutional repositories like data_UMR, JLUdata and TUdatalib, general research repositories like OSF or Zenodo, and discipline-specific repositories such as OpenNeuro or ZPID.

You are free to continue using such repositories in addition to our DataHub Repository. Additionally, the Marburg University GitLab instance may be used to publish finalized git repos with non-sensitive data and code. You are welcome to contact your local Data Stewards to select suitable publication platforms.

Data Security and Protection

Preventing loss of data

Working with valuable data and code, it is important to have a suitable backup and recovery strategy. Using the DataHub, you have access to different storage locations or platforms, which differ in the availability and actual implementation of backups. An overview can be found below:

Table 1: Availability of backups across different storage locations used in the DataHub (updated 01.08.2024)

Platform / location	Backup	Notes
TAM GitLab	yes	application backup*
TAM DataHub Repository	yes	application backup*
MaRC: /home	yes	regular over-night backups
MaRC: /scratch_shared	no	file-system for temporary use
MaSC: /masc_shared	no	

Warning

*** An application backup is not to intended to restore individual user data!** It protects against a general system failure.

Using **git**, you already have the tools to visit earlier states of your work within a given repository / dataset. So be very careful when deleting whole repositories or modifying their version record.

In cases where backups are not available for the respective storage location, users need to care themselves for regular backups (e.g. 3-2-1 rule).

FAQ and Troubleshoot

If you should not find the answer to your questing here, you are welcome to reach out to the DataHub Team.

For general information and FAQ on research data management, you may also visit the website of the hessian federal state initiative HeFDI or browse their published materials at zenodo.

Important note:
The TAM GitLab for use within the DataHub is now available (Marburg University network / VPN required). We encourage all DataHub users to use the TAM GitLab to manage their research data and code as this facilitates exchange within the project.

This **GitLab has replaced our local GIN platform**, which was discontinued, because further software developed and sustainable operation was not foreseeable.

This FAQ also contains information on using the deprecated GIN platform and the DataLad tool which both make use of git-annex to handle large data. These information has lost most of its relevance and will be removed in future versions.

TAM DataHub Repository

The TAM DataHub Repository which bases on the DSpace publication platform is currently being set up for TAM to enable publishing data and metadata with open or restricted access and obtaining persistent identifiers (DOIs).

The service will start test operation shortly. While it is not yet available, you are very welcome to contact your local Data Stewards to select suitable publication platforms (see also RDM section of this manual).

Git

What is git and how does it work?

This will be an overview answer on a questing that potentially fills books to answer in detail. It focusses on some basic principles, why git is useful beyond software engineering and what is to

be considered when using it for managing large amounts of data. Note that we offer comprehensive materials and workshops on using git and GitLab on the NOWA Website or via the NOWA School.

Git is a tool for distributed version management. Distributed means, that every collaborator has a copy of the repository and its history. A git repository is a directory with all its contents (files & subdirectories) which is managed by git, meaning there is a special hidden .git folder that contains essentially the "history" of the repository. Changes are integrated explicitly by committing a new version of the whole repository to this history and differences between collaborators are also explicitly integrated "merged". A Git repository is not only able to store and reproduce sequential versions ("history") but also enables parallel version tracks called "branches". In a branch you could for instance check the influence of a changed parameter. With these concepts you gain a great capacity of organizing your work while always being able to revisit and reuse earlier stages.

To store files and their versions, git initially saves the files and then records only the line-by-line differences at each commit.Git can thus reconstruct any committed state of a repository and highlight the changes. This works nicely for text files like programming code, descriptive text (like .txt and .md) or tabular data (like .tsv and .csv). However, for binary files like photos or videos (.jpg / .mp4), git cannot efficiently track the changes and it takes considerable computational power to reconstruct file versions. Therefore extensions like Git LFS are used which identify files by hash-sums and always store a full copy of the file, if a change is detected. As a consequence, it is ***not* advisable** to modify binary content arbitrarily often, as the repo's size would grow rapidly.

Issue: Unprotected private keys are ignored

Error:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0755 for '/home/USER_NAME/.ssh/id_ed25519' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
Load key "/home/USER_NAME/.ssh/id_ed25519": bad permissions
git@gitlab.uni-marburg.de: Permission denied (publickey).
```

Solution:

Linux

- Open a terminal window and set write protection for private key: `chmod 600 ~/.ssh/KEYNAME`

Windows (Win10)

- Navigate to private key (typically at C:\Users\USER_NAME\.ssh\KEYNAME)
- Open file properties, select security panel and click "Advanced" button to open the file permission panel.
- Disable inheritance and convert to explicit permissions.
- Remove all permission except for the current user.
- Apply changes and quit.

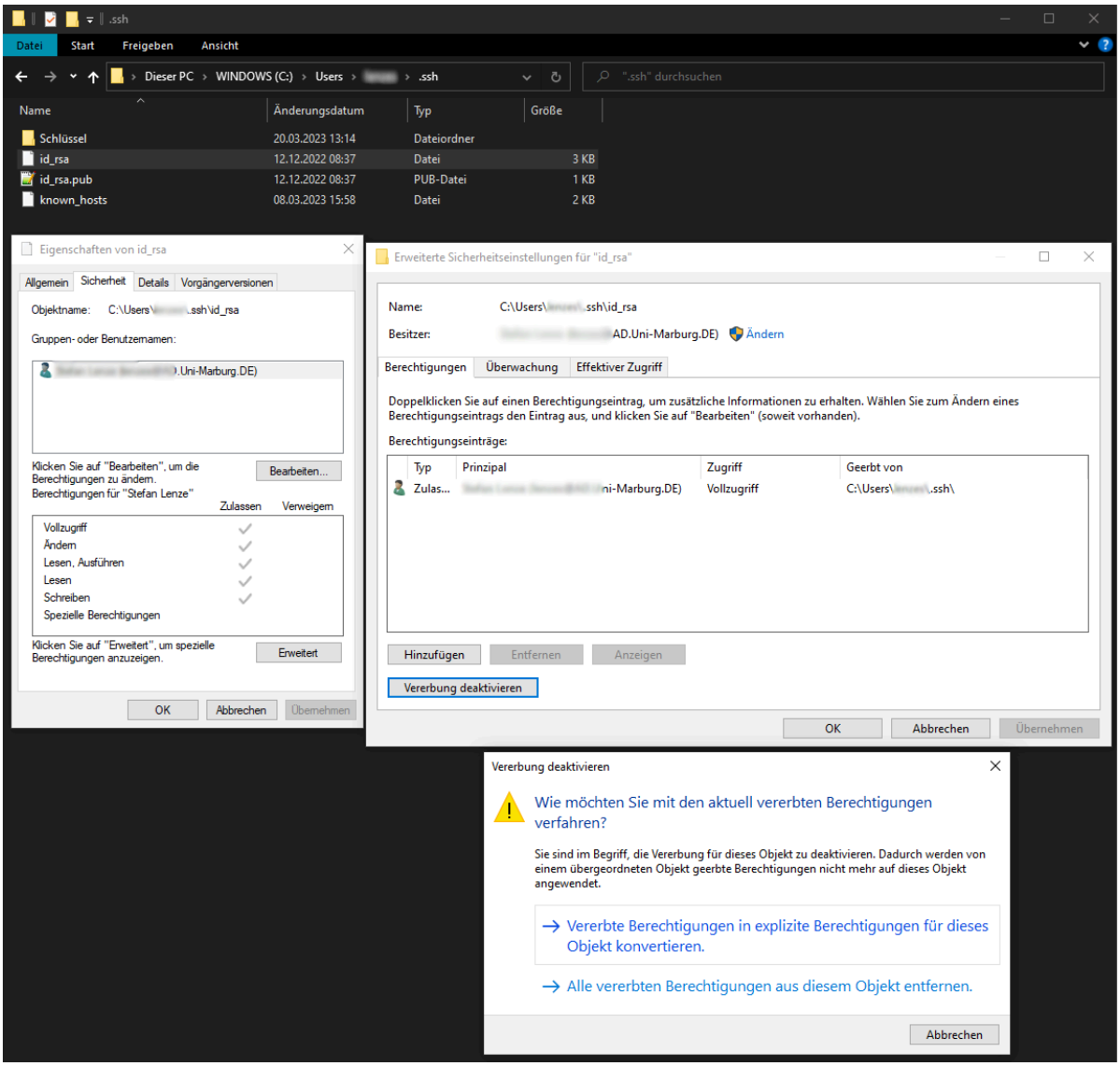


Figure: Restricting permissions for private SSH key on Windows. Workflow adapted from superuser.com.

GitLab

There are multiple instances of GitLab. Which one should I use?

We ask all DataHub users to use the TAM GitLab instance to store and manage their research data.

The TAM-GitLab builds on TAM-owned resources in the high performance infrastructure of MaSC and MaRC which allows us to offer a substantial amount of already paid storage in an environment with elevated security. As there are now multiple instances of GitLab, we particularly ask you to **use this TAM GitLab but NOT THE MARBURG UNIVERSITY GITLAB** for research data that is in the scope of the DataHub. If you already have such data on the Marburg University GitLab, please move it to our TAM GitLab.

Can I use DataLad to manage a repository on GitLab?

That is possible. As many DataLad commands will work also on pure git repositories, you are free to use DataLad or git commands. However, please be aware that these repositories and their remote origin at GitLab CANNOT USE GIT-ANNEX!

Please use Git LFS, if you store large files on GitLab.

Git LFS

What is Git LFS?

Git LFS stands for *Large File Storage* and is an open source extension to git which allows it to handle large / binary files efficiently. Without such an extension, repositories easily bloat up and get impractical to use. Git LFS solves the problem by redirecting large files to a separate storage space and lets Git handle only references to the LFS files which keeps the repo small.
You can define individual files, filetypes or directories which shall be managed this way.

How to install Git LFS?

You can install the latest Git LFS client as described on git-lfs.com by downloading a current installer or via the PackageCloud repository.

Depending on your operating system, you may also use their native package managing tools. Please be aware that you may install older Git LFS versions this way!

- Mac OS: `brew install git-lfs`

- Ubuntu / Debian: `sudo apt install git-lfs`.

On MaRC3 / JupyterHub, Git LFS has already been installedand enabled as a default module JupyterHub servers have it activated by default.
Be aware that if you purge all modules, you might need to enable the git lfs module again: `module load git-lfs`. Otherwise, your files will not be forwarded to the LFS store and may blow up your repo!

To check successful installation / activation, you can use `git lfs --version`.

Afterwards Git LFS has to be initialized once for your user: `git lfs install`. This also applies to first usage on MaRC3 / JupyterHub.

How to track large files with Git LFS?

For each git repository, you can now define which files shall be handled by Git LFS for instance:

- files: `git lfs track "07_disseminations/myManuscript.pdf"`
- directories: `git lfs track "03_data/001_myExperiment/*"`
- file types `git lfs track "*.pdf" "*.png"`

To summarize these rules, you can use `git lfs track` or inspect the .gitattributes file in your repo's root. With `git lfs ls-files` you can list all lfs files of a repo. The tracking patterns are written to the *.gitattributes* file in the root of each git repo and you can also inspect or edit the file directly. Negative patterns like in .gitignore are not supported by Git LFS(docs).

It is important to think of the filetypes and paths beforehand committing and pushing large files. Otherwise they will be tracked by git and will require specialized tools to reshape the commit history and migrate them them to LFS.

Limitations

- Currently, Git LFS does not allow to track files, based on their size (e.g. >1MB).
- On Windows, git before 2.34.0 does not handle files in the working tree larger than 4 gigabytes. Newer versions of git, as well as Unix versions, are unaffected.
- If not all contribution persons have lfs installed, untouched objects might be shown as modified.

How to manage previously committed files with Git LFS?

You may have existing repos with binary files which are managed by git. For better performance and stability you want to use Git LFS not only for new binary files, but also for the old ones. Alternatively, you may have unintentionally committed files to the git repo which should have gone to LFS store. This might happen for instance, if you don't enable the git-lfs module on the MaRC3 cluster. Potentially, you might have also already committed an additional copy of those files to the LFS store while one version of the files remains in the git history.

Git LFS comes with a migration tool which essentially rewrites your git history and enables you to migrate files between the git-repo and the LFS-store as if they would have always been there. It also updates the Git LFS tracking pattern in the .gitattributes file if necessary.

Say, you wanted to track all jpg and png files with git lfs and replace the existing files by git lfs pointers:

- Migrate locally (change file pattern to your needs):
  `git lfs migrate import --include="*.jpg, *.png" --everything`
- If you use a remote repository on GitLab, overwrite it with your local changes. Be aware that this may cause conflicts, if collaborating persons have already pulled the to be modified commits.
  `git push --all --force`

How to use the Git LFS locking feature?

With a relatively new feature of Git LFS, you can prevent others from concurrently modifying files you intend to edit. The feature works not exclusively on LFS files. - Defining files as lockable will remove write permissions from these files in the working directory: `git lfs track "*.jpg" --lockable` - To edit those files, you need to lock them (for others): `git lfs lock FILENAME.ext`. - To get a list of currently locked files and corresponding users, use `git lfs locks`. - Other users will be blocked from locking an already locked file. - If you have locked and modified a file, you must first commit changes before you can unlock it again. Use `git lfs unlock` for this.

Note that despite using this feature, you may still produce merge conflicts on lockable files. To prevent this: - Pull changes before you lock files to edit them.
- If you have committed a change of a locked file, push it before you unlock the file again.

See GitHub documentation

How to solve merge conflicts on LFS files?

Merge conflicts arise when conflicting versions of files from differnet sources need to be integrated. For binary files, these can not be solved by a line by line integration but only by telling git with which file you want to proceed. It is thus recommendable to avoid merge conflicts as shown below by communicating with collaborators (and using the previously described locking feature).

Creating a merge conflict

- Both 'a' and 'b' have a repo with an LFS file, a jpg image for instance.
- They each modify the file differently and commit their changes.
- 'b' pushes to origin and afterwards 'a' pulls from there, so that 'a' has to solve a conflict between the local and the remote commit from 'b'.
- In 'a's working directory, Git LFS replaces the file by an lfs pointer file in which the differences (SHA and file size) are shown.

Solving the conflict

- The pointer file is manually modified to point one of the versions for example to 'our' version (modification by 'a') by deleting SHA and file size of the remote version (modification from 'b') plus all git markers and saving the file. The merge commit is pushed to origin. GitLab recognizes the changed SHA reference and now shows a preview of the modification by 'a'.
- The working directory of 'a' still holds the pointer instead of the jpg image. With `git lfs pull`, the pointer is again replaced by the referenced file in the working directory.
- From the perspective of 'b', a simple `git pull` will update the working directory with the version from 'a' without showing the pointer files.

How are git-annex and Git LFS related?

Both tools serve the same general purpose in enabling git to manage large files. Git-annex is generally considered the more versatile but complex tool while Git LFS is often considered to be easier to use but lacks some features that git-annex has.

Git-annex handles specified large files by storing them as "annex-objects" and replacing them with symlinks in the working tree. If you need to modify them, they can be replaced with a copy of the actual files and be unlocked. Git annex tracks the availability of copies of your files in different locations and allows you to "drop" local file content while preserving the surrogate symlink and the availability information. Git-annex is supported by DataLad and a few repositories like GIN.

Git LFS also uses a storing mechanism separate from the main git repo and replaces the files so that git will manage only small text pointers to the files instead of the actual files themselves. It lacks the ability to drop files like git-annex but has other features like locking files for other users, if they are modified locally, preventing merge conflicts on binary files. It is supported by mayor git services like GitLab and GitHub.

The operational mechanisms of Git LFS and git-annex and Git LFS are distinct and we are not aware of a generally applicable, easy way to migrate binary data between both data management solutions. So please choose carefully.

JupyterHub

General FAQ on JupyterLab can be found at the online documentation of the Jupyter project.

High Performance Computing (MaRC3, MaSC)

Is there a Manual for using the MaRC3?

Yes, see here. However, access to the documentation is restricted to Marburg University Network (incl. VPN) and requires login with staff account name & passwort (no 2FA).

The website of the Competence Center for High Performance Computing in Hessen (HKHLR) summarizes cluster introductions, classes and tutorials on HPC topics. For the MaRC3, there is a biannual cluster introductions, held by René Sitt. Slides of a past presentation can be downloaded here (10.11.2022, login required).

How can I transfer files to and from MaRC3?

You have multiple options for file transfer including:

- A central GitLab repository using Git LFS. This is the generally recommended way, as it helps to keep data organized, version controlled and thus transparent.
- A JupyterLab session via JupyterHub. Jupyter gives you a graphical interface to upload individual files e.g. to your home directory or download individual files from there to your local computer.
- The `scp` command. *Secure Copy* is a simple command which establishes a secured connection to a remote location like MaRC3 and allows transferring files and directories in both directions:

- General usage: `scp -r -P PORT_NUMBER /SOURCE_DIRECTORY/ USER@HOST:/DESTINATION_DIRECTORY/`
  Note: the `-r` flag is required to transfer directories instead of individual files.
- MaRC3 examples:
  `scp -r -P 223 SOURCE USER@marc3a.hrz.uni-marburg.de:DESTINATION`,
  `scp -r -P 223 USER@marc3a.hrz.uni-marburg.de:SOURCE DESTINATION`. Replace *SOURCE* by your source files or directories, *USER* by your username (staff account) and *DESTINATION* by the desired remote /local destination (e.g. your home directory `~`).

Workflow

How can I speed up the workflow?

There may be some repetitive elements in your workflow, like activation of certain modules, a specific software environment or unlocking credentials on each login (see security section). If you are using the **bash** shell (default for many Linux systems, MaRC3, and available via gitbash for other operating systems), you can edit the bash configuration file *.bashrc*. Restarting the CLI is required to apply the changes.

Example: Loading a specific software packages after each login.

Requirements: You are working on MaRC3 with bash shell.

You may want to load the miniconda module and activate a certain virtual environment by default (replace "XYZ"). As an alternative to appending the configuration, you can open it in a CLI editor like nano `nano ~/.bashrc` and insert the following:

```
# Load conda and a private environment
module load miniconda
source $CONDA_ROOT/bin/activate
conda activate XYZ'
```

Example: Aliases in .bashrc

Requirements: You are using the bash shell (CLI).

Aliases are labels which can be used as a surrogate for a longer and more complex expression. You may want to define an alias to quickly connect to MaRC3: Add the following to your .bashrc and replace USERNAME with your staff-account:

```
alias ssh_hpc='ssh -p 223 USERNAME@marc3a.hrz.uni-marburg.de'
```

Security

Passphrases on ssh keys and how to manage them

It is good practice to use passphrases on ssh keys because it adds an extra layer of security by encrypting your private key file. If someone would get hold of your private key, it would still have to be decrypted first. The downside is that also you would need to provide the passphrase whenever you use the private key (e.g. git fetch) which very inconvenient.

The tool **ssh-agent** allows to unlock the public key just once per session and keep it in memory for further usage. To use the tool in the HPC context, it is most convenient to add these lines to your .bashrc file. If you have multiple keys in your `~/.ssh` directory, you may want to add the path to the specific private key file to the command `ssh-add PATH/TO/KEY`.

```
# Activate ssh.agent and add key
eval $(ssh-agent)
ssh-add
```

In the MaRC3 HPC environment you may want to add this to your .bashrc. On your local machine you can use for instance the KeePassXC password manager to unlock your private key via ssh-agent.

How to verify checksums

When you download software from the internet, you are often provided with a hash sum for the package / installer. With these checksums you can verify the file's identity and integrity. Although it provides no absolute protection, it is good practice to compare the checksum of the downloaded file against the provided checksum.

Dependin on your operating system you have different (command line) tools to calculate a checksum for any file:

- Windows: `Get-FileHash FILE_NAME`
- Linux: `sha256sum FILE_NAME`
- Mac: `shasum -a 256 FILE_NAME`

Other Tools and Platforms

Environment virtualization with anaconda and miniconda and venv

Anaconda and its light-weight alternative **Miniconda** allow you to manage software packages and virtual environments. Depending on your working environment and its operating system you may need to install software without having administrator / root privileges. It also enables you to work in isolated software environments for different projects. A similar, purely python-based toolset is **venv** (virtual environment management) and **pip** (package installer for Python). More information on environment virtualization is provided on the NOWA School website.

If conda is not yet available on your computer, you will find the installers for all common systems plus a user guide at the conda website.

Hints:

- Activating an environment by default: see below.
- Listing private environments: `conda info --envs`.
- Deactivating environments: `conda deactivate`.
- Removing unused environments: `conda remove -n ENV_NAME --all`.

DataLad

DataLad in the DataHub

Using DataLad has been recommended to interact with the DataHubs previous storage management and data sharing platform GIN. Since we have switched to GitLab which uses a different mechanism to manage large files (Git LFS), DataLad moved out of focus for the DataHub.

What is DataLad?

DataLad is a free and open source distributed data management system that:

- keeps track of your data,
- creates structure,
- ensures reproducibility,
- supports collaboration,
- and integrates with widely used data infrastructure.

DataLad is a Python-based tool that is compatible with all major operating systems. Using Git and git-annex, it allows you to version control arbitrarily large files in datasets, without the need for third party services.

A DataLad dataset is a directory with files, managed by DataLad. You can link other datasets, known as subdatasets, and perform commands recursively across an arbitrarily deep hierarchy of datasets (see figure below). This helps you to create structure while maintaining advanced provenance capture abilities, versioning, and actionable file retrieval.

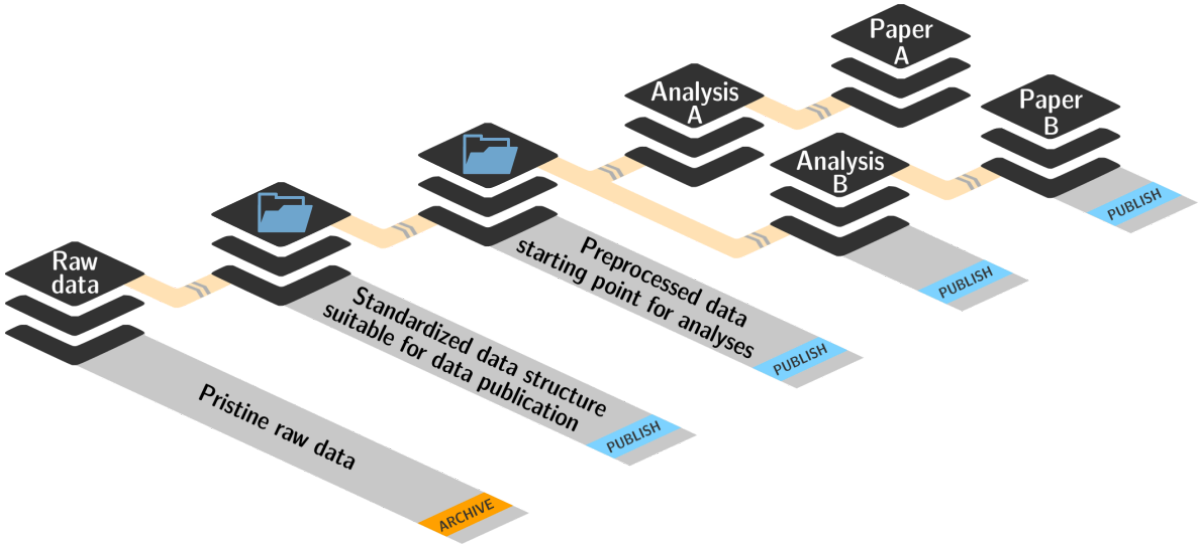


Figure 1: Managing research data with linked datasets (taken from the DataLad Handbook).

DataLad lets you consume datasets provided by others, and collaborate with them. You can install existing datasets and update them from various sources, or create sibling datasets at GIN that you can push updates to and pull from.

Basic Usage

Although DataLad has extensive functionality, you can quickly start using it with a small set of essential commands like:

- `create` a new local dataset
- `clone` or `install` existing datasets from a remote storage
- `create-sibling`: create remote copies of datasets called "siblings"
- `get` content on-demand
- `run` compute jobs in a reproducible way
- `drop` content if no longer needed

- `save` changes to datasets
- `push` changes to a remote storage.

Summarized form datalad.org.

GIN

GIN was the DataHub's designated central platform for storing, distribution and publication of research data. As the development of the platform has been discontinued and sustainable operation could not be guaranteed, our local GIN service has been replaced by the TAM-internal GitLab.

DataHub workflow instructions and tutorials

The following pages offer practical instructions on how to use the DataHub services for preparing, uploading, computing, downloading, and sharing your data. Written instructions as well as tutorial videos are provided.

However, this Manual is meant to be used after you consulted the local Data Stewards and already received an introduction by e.g., a workshop to all services. The following pages can only give general instructions but the workflow might differ for your specific case. For comprehensive background information on all services, see services pages.

Access to the DataHub

To use the DataHub services, you need to have a staff-account at the University of Marburg and the permission to use DataHub Services. Please see the section *Access* for information on how to get an account and/or permissions.

Required Installations

To make the most of the GitLab service, it is highly recommended to also use Git as a local version control system. It offers you easy up- and downloads of your data and code from your local computer. Manual uploads to GitLab might be slow, depending on the size of the files, and it is only possible in a file-by-file manner. With Git installed it is possible to only push the updates on your repository to GitLab, which is faster and saves memory and space. Also, if you don't use Git, you will always have to download the full project form GitLab to get the latest version of your GitLab-project (e.g., when your collaborator modified the project). With Git, again, you can only pull the latest updates.

Git should be installed on the platforms you actively work with during your project. This can be: Your office computer, your laptop, the computer that acquires the data, or MaRC3a of the DataHub. The advantages of using MaRC3a are outlined here and can be accessed with your UMR staff account.

Git Configuration

Please download Git on your local computer. MaRC3a already has a running Git installation.

After installation, you need to properly configure Git. Please type the following commands one after another in your command line / terminal (**note**: If you want to work with Git on your local computer as well as on MaRC3a, you have to do this configurations on both machines):

```
git --version
```

→ this tells you if the installation was successful by telling you the version number of Git you installed.

```
git config --global user.name "Example Name"
```

→ quotation marks are needed.

```
git config --global user.email example@example.com
```

```
git config --global init.defaultBranch main
```

```
git config --global core.editor ExampleEditor
```

→ this sets a default text editor when working with git. Just type in the editor you want to use. My recommendation is to use nano or vim. You don't have to download anything, it's already on your machine as it is a command-line editor. So, it's super convenient for working with git, as the git commands are also run in the command-line. It simply means that when writing commit messages you can do it in one window (i.e., your terminal) and you don't have to bother with a new window opening and making sure it's really closed etc. But of course, you can use any text editor you want. Just as an example, for making nano your global git-editor type `git config --global core.editor nano`.

```
git config --global pull.rebase false
git config --global pull.merge true
```

```
cat ~/.gitconfig
```

→ checks if your configuration was successful. It should give you this output:

```
[user]
  name = Your Name
  email = example@example.com
[init]
  defaultBranch = main
[core]
    editor = nano
[pull]
  rebase = false
  merge = true
```

The nano and vim text editors

In case you configured nano or vim as your default text editor when using git (vim is the default editor on MaRC3a/JupyterHub): Those text editors are command-line editors which means they open directly in the terminal (no new window opens). With everything, you just need to know how to operate it:

nano:
- when the editor opens for the commit message, you need to first press `enter` one time to make new line at the top.
- when you've done writing your commit message, you need to save and close it: `CTRL X` closes it but it asks you first if you want to save it by typing a capital `Y`, then `Enter`.

vim:
- when the editor opens you first need to type `i` to change from command mode to insert mode
- make a new line at the top by pressing `Enter`
- write your commit message and then press `Esc` and then `:wq` to save and exit

SSH Key Generation

To communicate with GitLab, you should have an ssh key assigned. You can follow these instructions to do so (login to the TAM GitLab required, of course). Then, go to the TAM GitLab and save this ssh key in your account preferences: Your Profile -> Preferences -> SSH Keys -> Add new key. Please check connection:

```
ssh git@tam-gitlab.online.uni-marburg.de
```

> "Welcome to GitLab, your-username".

SSH KEY GENERATION ON MARC3A

Login to the JupyterHub and open a terminal.

1. Generate ssh key: `ssh-keygen -t ed25519`
2. Display your public key and copy it: `cat ~/.ssh/id_ed25519.pub`
3. Open your ssh-key config file and add Hosts manually: `vi .ssh/config` → In-terminal text editor opens. Press `i` to enable the `insert mode`.

Your file currently contains this:

```
Added by Warewulf
Host *
    IdentityFile ~/.ssh/cluster
    StrictHostKeyChecking=no
```

Please add the following manually below (respect formatting/ spaces):

```
 Host tam-gitlab.online.uni-marburg.de
   IdentityFile ~/.ssh/id_ed25519
```

To exit the text editor press `esc` and then type `:wq` and then press `enter`.

4. Go to the TAM GitLab and save this ssh key in your account preferences: Your Profile -> Preferences -> SSH Keys -> Add new key.
5. Check connection:

```
ssh git@tam-gitlab.online.uni-marburg.de
```

> "Welcome to GitLab, your-username".

Enable Git LFS

Git LFS stands for Large File Storage and is an open source extension to git which allows it to handle large / binary files efficiently. Without such an extension, repositories easily bloat up and get impractical to use. Git LFS solves the problem by redirecting large files to a separate storage space and lets Git handle only references to the LFS files which keeps the repo small.

1. Download git-lfs. On MaRC3 / JupyterHub, Git LFS has already been installed. JupyterHub servers have it activated by default. If you connect to MaRC3 via terminal (ssh), you can enable Git LFS via its module: `module load git-lfs`.
2. Check installation: `git lfs --version`
3. Initialize git-lfs once for your username: `git lfs install`. This also applies to first usage on MaRC3 / JupyterHub.

If you do a `cat ~/.gitconfig` again, you should see the following added to your git configuration:

```
[filter "lfs"]
    smudge = git-lfs smudge -- %f
    process = git-lfs filter-process
    required = true
    clean = git-lfs clean -- %f
```

To learn more about Git LFS, check our FAQs. In the next chapters you will learn how to use Git LFS in the DataHub workflow.

Working with virtual environments

We recommend to first set up an environment using the environment and package manager conda. Creating environments basically means that you separate the work on your local computer into different environments. In these single environments you only have installed the software you actually need for this particular work and also only in the versions you need for this particular work. This is very useful as sometimes updates of a software can affect your previous work. See below for ressources and tutorials on virtual environments.

If you want to work in an environment on your local computer, please skip the steps 1.-4 and start with step 5. Step 1.-4. only apply if you want to work on the MaRC3a cluster.

By the nature of the MaRC3a cluster, you're provided only with a simple Linux environment at the beginning and you need to first activate miniconda before you can use it. You don't need to do this if you're working on your local machine.

1. Go to the JupyterLab of Marburg university and open a terminal. Alternatively, you can connect directly via SSH to the MaRC3 cluster.
2. Make sure to be in your home folder: `pwd` should give you the output `/home/your-username`
3. Activate the module Miniconda for environment activation: `module load miniconda` (In front of your username you should now see (base))
4. Activate environment: `source $CONDA_ROOT/bin/activate`
5. Create a new empty environment for your project: `conda create --name my-project`
6. Activate this new empty enviroment: `conda activate my-project` (In front of your username you should now see (my-project))
7. Install the packages you need in this environment, like so: `conda install -c conda-forge pandas==2.2.0`
hint: you can create an `requirements.txt` file that lists all the packages you need and then load this file instead of loading every single packages one after another.

!!NOTE!! Environment reload:
MaRC3a setup: Every time you restart JupyterLab / reconnect to MaRC3, you need to go through

steps 3., 4., and 6. again.
Local setup: Every time after you closed the terminal with the environment and want to continue working in this environment, you have to execute step 6. again.

In our FAQs you can find instructions on how to automate the installations described above.

RESSOURCES AND TUTORIALS ON VIRTUAL ENVIRONMENTS

- Introduction to command line tools and virtualization techniques
- Understanding Conda and Pip
- A beginner friendly intro to VMs
- Docker vs. python virtualization vs. virtual machines

Everything set up? Great! Let's prepare your data in a next step!

Preparing Your Data

Data Structure and Repository Choice

For usage of the TAM GitLab, we kindly ask you to bring your data in a certain structure.

Note: For comprehensive background information on TONIC and BIDS, please find the according entries on this website under *Data and Code Management*.

First, for overall project management you should set up a project folder that follows the TONIC Template. A template can be downloaded or forked from the TAM GitLab. Here you'll find a template for organizing multiple projects in one repository (link) or for organizing a single project (link). The folder names should be self-explanatory but if you have any questions on how to use the template, please consult your local Data Stewards. If you don't know how to use Git and GitLab yet, we kindly ask you to go through the Git and GitLab tutorial first.

The `03_data` folder should contain your data according to the BIDS standard (see BIDS documentation and the BIDS Starter Kit). Please find converters for your kind of data on the BIDS-converter website and run the BIDS-validator on your dataset to ensure that your dataset is in a valid BIDS format. For some kinds of data no BIDS-converter has been built yet. In this case you need to convert your data manually. **For help with that, please contact the Data Stewards.**

Converting your data to BIDS is an ongoing effort. It is not advisable to only convert your data to BIDS shortly before publication of the dataset. BIDS is meant to help you with managing your data throughout the active development of the project. This means, every time you collect new data you should immediately convert your data to the BIDS format and save the accompanying metadata. This ensures that your project stays fully reproducible because your analysis code is written to read in your data in BIDS format. Furthermore, there are already a lot of BIDS Apps which are automated processing pipelines that can read in BIDS datasets.

Example for a tiny BIDS-MRI dataset:

```
.
├── README
├── dataset_description.json
├── participants.json
├── participants.tsv
└── sub-01
    ├── anat
        ├── sub-01_ses-anat_run-01_T1w.json
        ├── sub-01_ses-anat_run-01_T1w.nii
        ├── sub-01_ses-anat_scans.json
        └── sub-01_ses-anat_scans.tsv
    ├── func
        ├── sub-01_task-oneback_run-01_bold.json
        ├──sub-01_task-oneback_run-01_bold.nii
        └── sub-01_task-oneback_run-01_events.tsv
```

We know that the BIDS documentation is very comprehensive and that getting started with BIDS can be a bit overwhelming. Please don't hesitate to reach out to your Data Stewards in order to get support on this matter.

FURTHER RESSOURCES AND TUTORIALS ON BIDS

- BIDS Starter Kit Tutorials
- BIDS Talks where the background and philosophy of BIDS are explained
- BIDS cookbook

Now that you know how to structure and handle your project, let's learn some Git and GitLab!

Tutorial Objectives

Topics 💡

Please try to reserve 3-4h of your time for this tutorial. If you plan on rushing through this tutorial, you will only get frustrated. We know that 3-4h seem like a lot of time, but after this tutorial you will be prepared for working collaboratively with Git and GitLab, which is exactly what the DataHub is built for. This tutorial is created in a praxis-oriented manner and under every section you'll find tasks which are nice for getting some practice. Because: You can only learn Git and GitLab by actually doing it!!

This tutorial teaches to work with Git through the command line. This has the purpose to make you understand what's really going on when you communicate with Git. However, you are welcome to use one of the many Graphical User Interfaces (Git GUIs) for Git which are more user friendly than working with the command line. A good start, for example, is to use VSCode as your code editor, as it comes with a Git integration and multiple extension packs for Git as well as for GitLab.

Please be reminded that for going through this tutorial you need to have gone through the Installation and Setup part.

Version Control and Git: Background and Theory

Objectives 📍

- What Version Control is and where it comes from
- Why it is useful
- The basic principles of Git:
  - the Git database
  - structure of the Git repository
  - the staging area
  - commits

Basic Git Workflow

Objectives 📍

- init
- add
- commit
- status
- diff
- log

Branching and merging

Objectives 📍

- branch
- checkout
- merge
- HEAD and detached HEAD
- moving in your commit history
- how to solve a merge conflict

GitLab workflow

Objectives 📍

- What is GitLab
- local vs. remote repository
- push
- pull
- clone
- fetch
- merge
- merge conflict between your local and remote repository

Contributing on GitLab

Objectives 📍

- fork
- merge requests
- gitlab for project management
- Git LFS

optional/reading/further materials

- YouTube is full of Git/GitLab/GitHub videos for all kinds of levels and features!!! For example: Brainhack Git introduction or GitHub CI
- Git cheat sheet by gitlab or github
- Atlassian tutorials and cheat sheet
- Troubleshooting: Oh shit git or Dangit git (are the same, but the latter is without swearing)
- Git branching
- Git GUIs
- Advanced Git commands
- NOWA workshops

- really, just type anything you want to know about Git in YouTube and you'll find a tutorial for it.

Introduction

Where Version Control Comes from

Version Control originally comes from software development. Of course, we develop software, too, like our experiment or data analysis code. However, we often do not develop our code in a team with hundreds of people. When developing an app like a very simple gaming app, a lot of people are involved in active development at the same time. Also, a source code for a very simple gaming app has at least 50,000 lines of code. The code for analyzing our research data likely has a lot less lines. The code for the gaming app also has different components (*modules*) and therefore the usual development process is divided in those modules, meaning that one person is responsible for one module. In the end, those modules have to come together smoothly. Also, apps usually come in different versions. We all heard about alpha-, beta-, or release-version of apps.

So professional software developers were confronted with several challenges:

- Keep track of what each person is changing in the code
- incorporate changes without breaking another module
- have multiple streams of work for different versions
- etc.

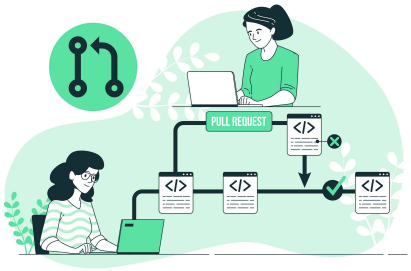
Sometimes Git might work in a way that is a bit too divided for your way of working (i.e., going in small-sized steps). There might be steps in the Git workflow that you consider unnecessary for your own needs. Then it always helps to remember where version control comes from to understand why the version control system is working in the way it does.

Centralization

Another key challenge in software development is facilitating collaborative code development independent of time and space, i.e., centralization. With Git, you are only provided with local version control, meaning it is a software that runs on your computer and no one else has access to your work (=decentralized). For collaborative code development it is necessary that all developers have access to the same project and are able to see all of the projects changes and history (=centralized).

Collaborative code development dependent on time and space

→



Collaborative code development independent from time and space[2]

For this, collaborative coding platforms such as GitHub or GitLab are the key. This is also a common misconception of Git: **Git is NOT the same as GitHub/GitLab**. They're independent of each other, in principle. You can use Git on your local machine without using GitHub/GitLab, and you can use GitHub/GitLab without using Git on your local machine. However, the combination of Git and GitHub/GitLab will give you the best of both worlds: Being able to track everything your doing on your local machine even if you don't have internet access, as well as sharing everything you did with your collaborators with just one command.

Git	**GitHub/GitLab**
- git is a software on your computer	- collaboration platforms based on the git software
- decentralized	- centralized
- local version control	- distributed version control

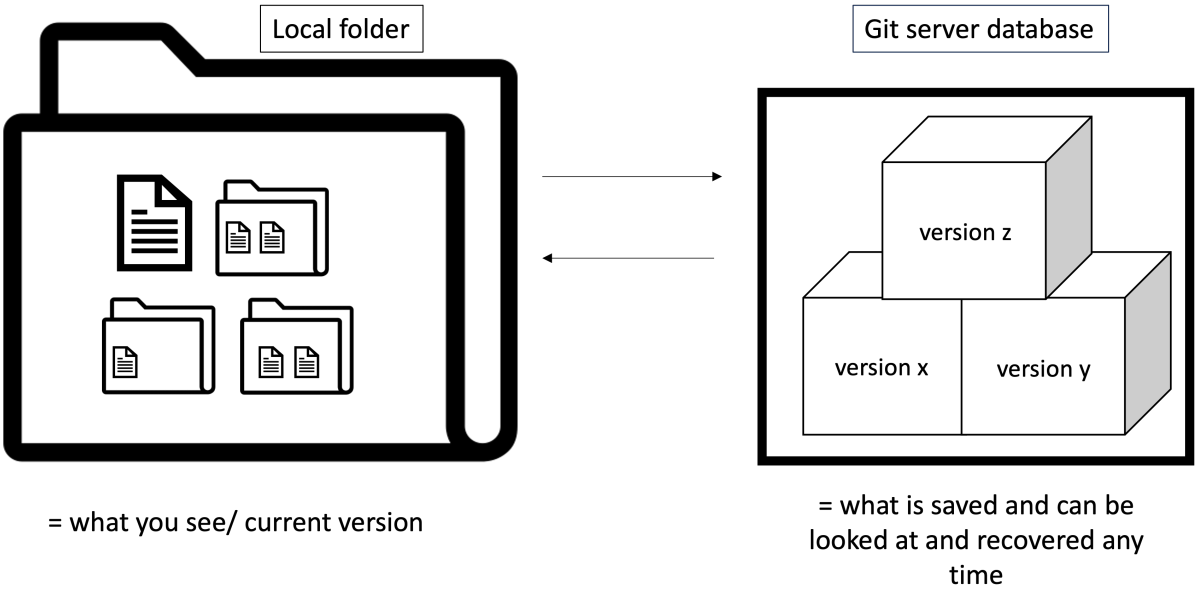
Benefits of Version Control

- Traceability
  - Track changes
  - Annotate each change with a message (=explain what and why) = a complete annotated history of the code
- multiple streams of work (or version sets) independent from each other while also being able to merge the work back together
- For collaborative work this means:

- it keeps all team members on the same page
- it makes sure that all team members can work simultaneously on the same project
- it makes sure that everybody is working on the latest version of the project

The basic principle of Git

As a version control system, what Git does is simply store different version of your project for you. Git itself is a software that provides you with a database in which the different versions of your project will be stored. All of these versions can be looked at and retrieved at any time, using the appropriate commands. So, for working with Git, it is important to know how to communicate with it, i.e., knowing the language of Git (which we will learn during this course!).



The basic principle of Git. Local folder vs. Git version database.

How the versioning works

> "Ok, so how exactly does Git store different versions of my project?"

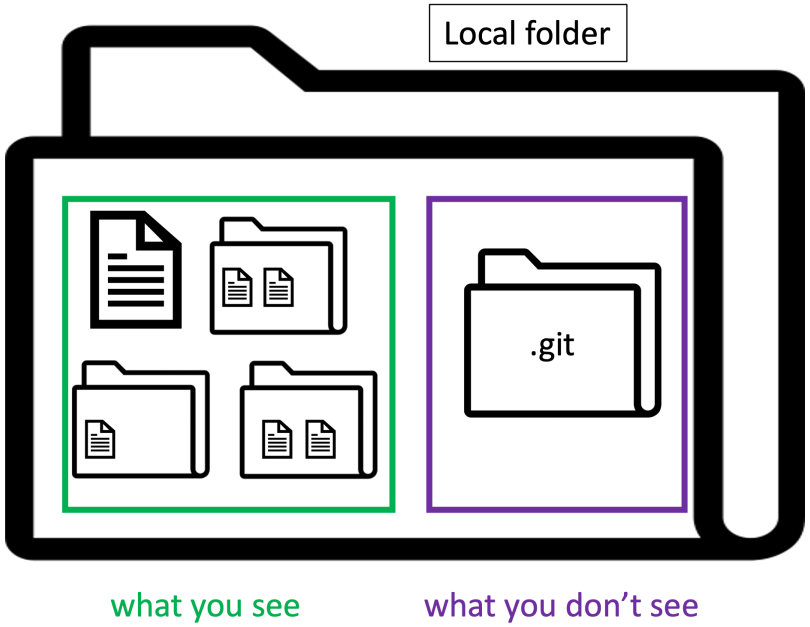
Well, every time you want to save the current state of your project (=*one version*), git takes a "snapshot" of how your folder currently looks like (=*commit* a version). The reference to this snapshot (=*commit hash*) will then be saved in your *commit history* and with this commit hash you can find this version in your version database aka *Git repository*. Committing a version means saving a snapshot of how *all* your files currently look like.

Hello Git, please save the current state of the project

- making a commit = Git's taking a snapshot of the current stage of your project and saving it in the Git repository
- commits are stand alone versions of the project
- for every commit, Git creates a hash which looks like this: 7c35a3ce607a14953f070f0f83b5d74c2296ef93
- all hashes can be found in the commit history and can be used to look at or retrieve an earlier version

> "Where is this version database?"

When you ask Git to version control your files (=*initialize Git*), it will create a hidden folder inside your local folder. This hidden folder (called *.git*) is your version database.



After initializing Git in your project folder, a hidden .git folder is placed inside your project folder. Inside this hidden .git folder the different versions of the project are stored.

> "What does "different versions" of my project even mean? How do I know when to commit a version and when not?"

First of all: The expression "committing a version" actually should be "committing changes". Because that's what you do: You start with a first version of your project and time after time you *change* things compared to previous versions. So, what you commit in the end are actually the changes to the project.

Second: For the decision on when to make a commit, it is important to know that every time you make a commit you're being asked to write a short description of what this commit is about (=*commit message*). So, what you should ask yourself before making a commit is: "Is the *change* between my current version compared to the previous version worth being saved as a stand alone version of my project?".

The commit message you give is bound to your commit, i.e., the commit message should state what you *changed*. Only then you will be able to figure out to which commit you need to go back if you're looking for a specific version of your project.

Another practice about commits is to commit changes that belong together. Namely, if you changed something in file1 and something in file2 and those changes are logically related, you should commit them together. If changes you made are not logically related, you should commit them separately.

Example for logically related changes

You are working on a project where you have to conduct an experiment, analyze the data, and publish a manuscript. You make a change to your stimuli-coding-file by coding a new stimuli shape. You insert this new stimuli shape in your experiment code as well. The changes happened in different files, yet the changes are logically related (make new shape - use new shape). You should commit those changes together, meaning *after* you changed both files.

Example for NOT logically related changes

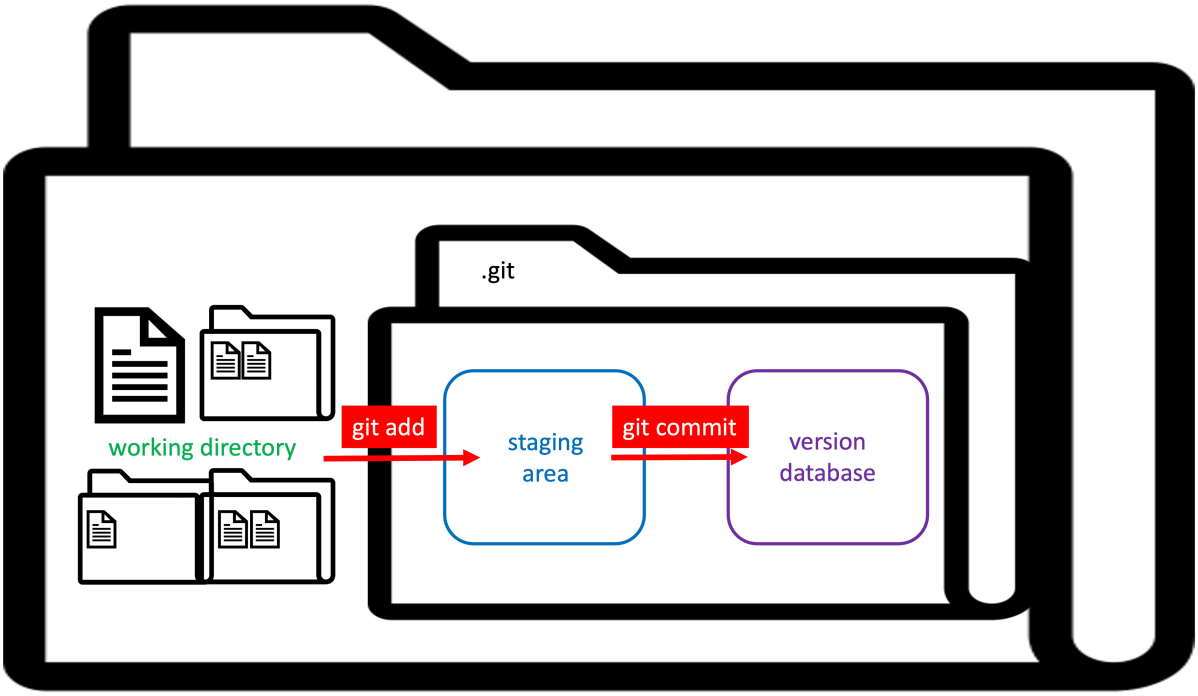
You are working on a project where you have to conduct an experiment, analyze the data, and publish a manuscript. You make a change to your stimuli-coding-file by coding a new stimuli shape. On the same day, you also make a change in our manuscript (which is awesome, btw), changing the affiliation of one of the co-authors. Those changes are not logically related. You should not put them in the same commit.

The staging area

> "But what if I worked on multiple different things since the last committed version and now I'm ready to commit my changes? Will I not have to commit all files together then?"

Luckily, no! Git has something called *the staging area* which solves this problem. *Staging* your files is basically giving your files a valid ticket for a ride to the version database. Only the files with a valid ticket for the next train (= the files you *staged*) can get onto the this train and take a ride to the version database (= commit to the version database). Every other file is left behind.

In the end, the basic Git workflow looks like this:



Basic git workflow. Every change you want to commit needs to be staged first. Only staged changes will be committed in a new version.

---

1. Text generated with ChatGPT. ↩
2. Figures retrieved from Freepik. ↩

Basic local Git Workflow

Objectives 📍

- init
- add
- commit
- status
- diff
- log

Note: For this tutorial you will need a project folder to work on.

Now that we have a basic understanding of what Git is and what it does, we want to try it. In this section we want to make changes to our project and therefore create different versions of our project. To do so, we first need to learn the basic vocabulary of Git.

Git vocabulary

- `git init`: gives git initial control over your files
- `git add`: adds a change from the working directory into the staging area
- `git commit`: saves a snapshot of the current version of your project in the repository
- `git status`: checks the status of the files
- `git diff`: let's you inspect differences between versions (in a certain way)
- `git log`: displays the commit history

First steps

The very first step at the beginning of version controlling your files is to tell git to draw attention to this folder. With `git init` we give git the power to track our files.

Navigate to your project folder in your command line/ terminal. You can also use VSCode as an IDE to work on your project. There you have an integrated terminal and are always in the folder which you currently have open in VSCode.

When you are in the right directory, type:

```
git init
```

git init

However, our files are not tracked yet, for this we have to add and commit the files with a first commit. We can check this by typing

```
git status
```

git status

Through the power we gave git by doing a `git init`, git always knows exactly what's going on in our folder. `git status` will tell us what that is. When using git only locally, we can have three different outputs of `git status`:

- *untracked files*: git knows there are files in our folder but we didn't tell git to actually track them. Either we created a new file and haven' added it yet or we simply don't want this file to be tracked.
- *changes not staged for commit*: at some point we added this file and git tracks its status. Now we made changes to this file but we haven't added them, yet.
- *changes to be committed*: these changes we already added to the staging area and will be included in our next commit.

Let's move on. Because we want EVERYTHING in our folder be tracked by git, we simply add and commit everything at once. First we type

```
git add .
```

git add

Like we learned in the previous section, `git add` allows us to add files individually to the staging area. With the `git add .` we added ALL files to the staging area (that's what the `.` does after add). If you want to stage your files individually, you simply exchange the `.` with the name of the file you want to stage.

Now that we have all our files staged and waiting for the train to version database, we can commit them. We type

```
git commit
```

and our text editor opens automatically. This is because with every commit we need to write a commit message to know later what this commit was about.

The nano and vim text editors

In case you configured nano or vim as your default text editor when using git (vim is the default editor on MaRC3a/JupyterHub): Those text editors are command-line editors which means they open directly in the terminal (no new window opens). With everything, you just need to know how to operate it:

nano:
- when the editor opens for the commit message, you need to first press `enter` one time to make new line at the top.
- when you've done writing your commit message, you need to save and close it: `CTRL X` closes it but it asks you first if you want to save it by typing a capital `Y`, then `Enter`.

vim:
- when the editor opens you first need to type `i` to change from command mode to insert mode
- make a new line at the top by pressing `Enter`
- write your commit message and then press `Esc` and then `:wq` to save and exit

Commit message structure

When the text editor opens and asks us for a commit message we have to insert a title and the actual commit message. The first line we write will be the title. The title should be short and expressive. Then we make an empty line and after this we write a longer commit message. This message can be as long as you wish or need it to be.

Because this is our first commit we can simply write a commit message like:

> "initial commit"
>
> "added all my files of the choice_rtt project to be tracked by git"

Later, we need to be more specific when describing what we changed.

NOTE FOR MAC USERS: Please make sure your text editor really saves files in `.txt` and not `.rtf`.

Please make sure that your text editor is really closed!!!

Making changes

Now we want to explore a bit the different functions of git. For this we need to make our first change. Open one of your files and make a change. Afterwards, type `git status` again. What does it say now?

So, we need to stage our change before we can commit it. We type

```
git add *path-to-your-file*
```

because we only want this file to be committed. Why? Because a commit message is bound to the changes we commit and we want to write a useful commit message that says what we changed. We will see later why this is important. Now, commit your change and give it a useful commit message.

```
git commit
```

git commit

When you do a git commit, you don't need to specify which file you are committing as every file in the staging area will be committed. If you don't want certain files to be committed together, you need to specify this by your add-commit workflow. Every time you do a `git add`, a file comes into the staging area and is included in the next commit.

Example:

You made changes on file1, file2, and file3. An add-commit workflow like this

```
git add file1
git add file2
git commit
git add file3
git commit
```

leads to file1 and file2 committed together, file3 separately.

Whereas an add-commit workflow like this

```
git add file1
git add file2
git add file3
git commit
```

leads to all three files being committed together.

Again, our text editor opens and asks us for our commit message, so we should do that.

When we do a `git status` again now, it should tells us

> "nothing to commit, working tree clean"

which basically means that the version in our working directory is the exact same version as the *last* committed version (remember the structure of the git repository from the previous section).

Task 1

1) Make a change to one of your files. Add and commit it like we just did.

2) Make another change. Don't add it.

3) Create a new file and write something in it.

4) Do a `git status`. What does it say? Do what it says needs to be done to get a clean working tree. Try the tip "git commit -m" below for committing.

git commit -m

If you type `git commit -m` instead of `git commit`, git expects you to write a very short commit message after the `m`, like `git commit -m "my commit message"`. This short commit message basically replaces the title+message form and the commit message should be short and precise. You do not have more information to this commit than the short commit message. You should always think about when to make a short commit message and when you might need a longer explanation on what you've changed.

Looking at differences

Another great thing of git, besides storing different versions for us, is its ability to show us exactly what we changed. For this we have an extra command called `git diff`. Let`s try this.

Make another change to your file, close it, but DON'T commit it for now. We can now look at the difference between the version in our working directory and the one in our version database (= last committed version). Let's type a `git diff`. The output looks weird but is actually really helpful, here's an example of a git diff with an explanation below[1]:

```
diff --git a/diff_test.txt b/diff_test.txt
index 6b0c6cf..b37e70a 100644
--- a/diff_test.txt
+++ b/diff_test.txt
@@ -1 +1 @@
-this is a git diff test example
+this is a diff example
```

- the first line just tells us which file is being compared. If you included multiple files in your commit, the differences for each file will be shown individually. So after all the differences for

one file, the differences shown for another file start with this line `diff --git name-of-the-file` again.

- `index` shows us the hashes of the files being compared. You're right, we didn't commit our current version so it shouldn't have commit hash, yet. Well, git assigns a hash to it anyway but it's not really a commit hash but you can think of it more like a "working directory hash".
- the third number after `index` tells us which kind of file it is. `100644` for example stands for an ordinary text file.
- `--- a/filename` and `+++ b/filename`: changes from `a/diff_test.txt` are marked with a `--` and the changes from `b/diff_test.txt` are marked with the `+++` symbol
- The remaining diff output is a list of diff 'chunks'. A diff only displays the sections of the file that have changes. In our current example, we only have one chunk as we are working with a simple scenario. Chunks have their own granular output semantics
- The first line is the chunk header. Each chunk is prepended by a header enclosed within `@@` symbols. The content of the header is a summary of changes made to the file. In our simplified example, we have -1 +1 meaning line one had changes
- In a more realistic diff, you would see a header like `@@ -34,6 +34,8 @@`: In this header example, 6 lines have been extracted starting from line number 34. Additionally, 8 lines have been added starting at line number 34
- The remaining content of the diff chunk displays the recent changes. Each changed line is prepended with a + or - symbol indicating which version of the diff input the changes come from

git diff

By simply typing `git diff` you will be shown the difference between the current version in the working directory and the last committed version. However, we can also look at differences between two committed versions or the difference to a staged version. For this we need the assigned commit hashes, which we can find out through a `git log`.

The commit history

`git log` will show you your commit history. Your commit history is basically a list of all your commits annotated with metadata. One of the `git log` metadata are the commit hashes which you will need to search for/ go back to older versions or creating new branches (which we will learn in a sec). To see your commit history, simply type `git log`. Here's an example and explanation of what we see:

```
commit 211a87947765cb34fe922bb39eed8e2357ea5ae9 (HEAD -> main)
Author: julia-pfarr <pfarr@staff.uni-marburg.de>
Date:   Thu Feb 8 18:57:23 2024 +0100

    change Git intro
```

- The first line is the commit hash. That's a unique identifier for this commit
- The next two lines are author and date. If you start working collaboratively on a project, you will also see entries from your collaborators
- the next line is the title you gave your commit. If you put in a longer commit message, you would also see this longer commit message.
- `(HEAD -> main)`: `HEAD` is a symbolic reference pointing to wherever you are in your commit history. When you move in your commit history, `HEAD` moves with you, like a shadow. In our example, `HEAD` is with the latest commit and is *pointing* to `main`. This means we are on the latest commit that was made on the *branch* `main`. We will learn in the next section why having the `HEAD` reference is very useful.

git log

- to exit `git log`, press `q`
- to only show a concentrated version of the commit history, type `git log --oneline`. It will only have the last 7 digits of the commit hash (which is all you need), the title of the commit, and the `HEAD` and `main` info, like this:

```
211a879 (HEAD -> main) change Git intro
c3c16d6 finish intro
e4a1e75 add online experiment section to psychopy
8644c34 add graphics to PsychoPy dir for testing
7506aa6 delete Remote WSL extension
541c70c reset
0ec1af7 fix image path in psychopy intro
```

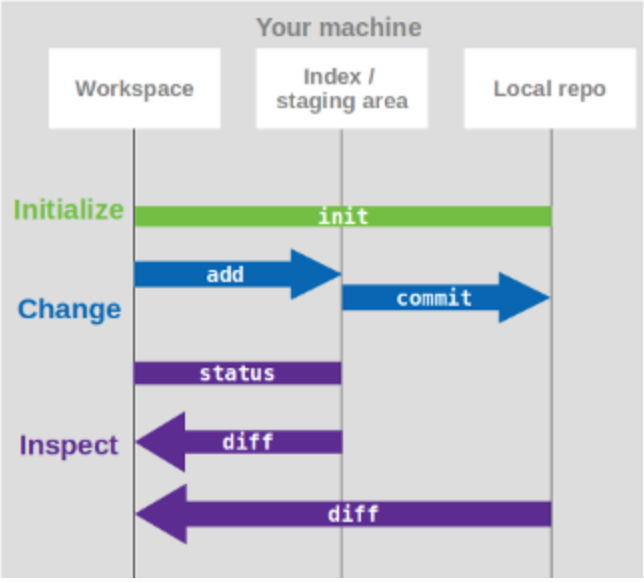
Task 2

Do a `git diff` between your second commit and your second to last commit.

Answer

- `git log --oneline` to find the commit hashes assigned to the respective commits
- `git diff HASH_OLDER_COMMIT HASH_NEWER_COMMIT`

Here`s an overview on what we did so far[2]:



Git and binary files

Git is not good at handling binary files. You can still track them using Git by adding and comitting them but Git can't do a `git diff` of binary files. The reason is that Git actually *reads* the files, thus it needs file formats that are text based (like usual code file types such as .py, .m, .R, or text files like .txt, .md, or tables in .csv, .tsv). File types like .docx, .xlsx, .pdf or images like .png, jpeg and so on are binary files which means the content of the files is actually stored in 0s and 1s. Therefore, `git diff` is not really working. There are some options how you can still diff binary files with Git (see this entry "diffing binary files") but it is just easier not to.

If you have a lot of binary files and large files in your project you might want to think about excluding those files from the diff operations of Git and also checking them into Git LFS instead.

Because Git checks all files every time and needs to come to the conclusion every time that this particular files is binary. Sometimes this can go wrong and Git could possibly destroy your binary file. To avoid this you can create a `.gitattributes` file (the `.` before the file name is important!!) and tell Git to track those files with Git LFS instead.

Task 3a

Create a `.gitattributes` files on the first level in your project folder:

1) make new file `.gitattributes`.
2) put the following content:

```
# Source files
# ============
*.pxd    text diff=python
*.py     text diff=python
*.py3    text diff=python
*.pyw    text diff=python
*.pyx    text diff=python
*.pyz    text diff=python
*.pyi    text diff=python

# Binary files
# ============
*.db     binary
*.p      binary
*.pkl    binary
*.pickle binary
*.pyc    binary export-ignore
*.pyo    binary export-ignore
*.pyd    binary

# Jupyter notebook
*.ipynb  text eol=lf

# Graphics: when git-lfs activated, exchange 'binary' with: filter=lfs diff=lfs
merge=lfs -text
*.png      binary
*.jpg      binary
*.jpeg     binary
*.gif      binary
*.tif      binary
*.tiff     binary
*.ico      binary
# SVG treated as text by default.
*.svg      text
# If you want to treat it as binary,
# use the following line instead.
# *.svg    binary
*.eps      binary

# Documents
*.bibtex   text diff=bibtex
*.doc      binary
*.DOC      binary
*.docx     binary
*.DOCX     binary
*.dot      binary
*.DOT      binary
*.pdf      binary
*.PDF      binary
*.rtf      diff=astextplain
*.RTF      diff=astextplain
*.md       text diff=markdown
*.mdx      text diff=markdown
*.tex      text diff=tex
*.adoc     text
```

```
    *.textile  text    *.mustache text    *.csv      text eol=crlf    *.tab
text    *.tsv      text    *.txt      text    *.sql      text    *.epub
diff=astextplain
```

Task 3b

1) Check that you actually installed Git LFS: `git lfs --version`.
2) Git LFS has to be initialized once for your user. If you didn't do so while going through the Installation and Setup section, please do so now: `git lfs install`. This also applies to first usage on MaRC3 / JupyterHub.
3) For your project, you can now define which files shall be handled by Git LFS by typing the following in the terminal while being in the project folder (examples, please modify according to your project and your files):

- specific files: `git lfs track "07_disseminations/myManuscript.pdf"`
- directories: `git lfs track "03_data/001_myExperiment/*"`
- file types: `git lfs track "*.pdf" "*.png"`

4) Watch the changes in the `.gitattributes` file from the note above. When Git-LFS is activated and you have instructed Git-LFS to track certain file types, you will see the configuration for that type being updated to `filter=lfs diff=lfs merge=lfs -text`. You may change it accordingly for other binary file types here too. We provide you with version of this template where this has been done for all binary file types at the TAM GitLab.

.gitignore

Another config file of Git is `.gitignore`. In this file you can state if you want certain files or whole directories excluded from being tracked by Git. Sure, you could also just not add them. But then you will always get the message of *untracked files* and that might annoy or confuse you with time. So, instead you can just list those files or directories in a `.gitignore` file.

For example, Mac users should set the `.DS_Store` files to be ignored. You can either do that globally (`echo .DS_Store >> ~/.gitignore_global`, then `git config --global core.excludesfile ~/.gitignore_global`), or in a `.gitignore` file. The `.gitignore` file has the same structure as the `.gitattributes` file. Templates can be found here.

---

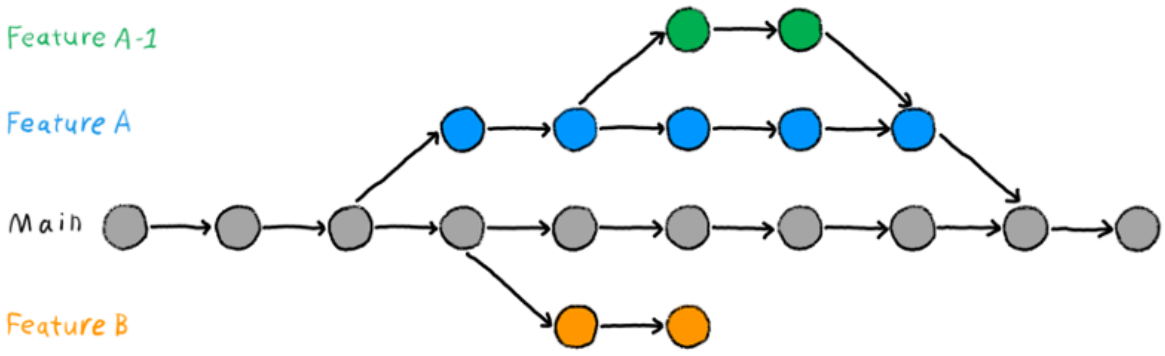
1. Example and description retrieved from Atlassian Git Tutorial ↩
2. Image by Peer Herholz ↩

Branching and Merging

Objectives 📍

- branch
- checkout
- merge
- HEAD and detached HEAD
- moving in your commit history
- how to solve a merge conflict

Now that we know how to make, record, and inspect changes with Git we want to dive into the really cool features of Git. In the introduction we heard something about "multiple streams of work". In Git this is done by `branches`. So far we only worked on the `main` branch. This is our main development stream. However, sometimes we want to try something new with our code and see if it works out without trashing our main development stream with too many commits. Or, we work collaboratively and you work on one module of the code and your collaborator works on another module of the code. To not interfere too much and having to incorporate the changes of your collaborator all the time and keep a clean main development stream, you should work on separate branches. In the end, you can `merge` your versions into the main branch.



Branching with Git. You always start your development on the main branch. Along the way you might want to try out different things while keeping your main development stream clean. For this you can create as many branches as you

need, working on them in parallel and incorporate the work you did on these features into the main branch. Or not.

Before we try this, we need some more vocabulary:

More Git vocabulary

- `git branch`: create a new branch
- `git checkout`: check out (or "replace") files in working directory with another staged or committed version AND move between branches
- `git merge`: to merge two separate streams of versions (branches) into one

Creating a new branch

Let's say you were asked by our supervisor to set up your experiment with a new feature, e.g., adding some eye tracking in addition to your EEG experiment. Because want to keep the main development stream clean and we know that we need to work on the eye tracking feature a lot, we decide to set up an extra branch for it.

Task 4

1) Create a new branch by typing `git branch eyetrack`. Check with `git branch` if it worked.
2) Switch to your new `eyetrack` branch by typing `git checkout eyetrack`. You should see *"Switched to branch eyetrack"*. Check if you are really on the new branch.
3) Make a change to your experiment code, add and commit it.
4) Do a `git log`.

What is different about the git log output compared to the last git log we did?

Why is `HEAD` not with `main`?

Git branch

- to know which branch we're currently on we can do a `git branch`
- we can switch between branches by doing `git checkout branchname`
- instead of `git branch name-of-new-branch` plus `git checkout name-of-new-branch` we could also just do a `git checkout -b name-of-new-branch`

Moving around in your commit history

`git checkout` is not only for moving between branches, but also to move on one branch between commits. You might want to do this because you realize that you like the instructions for your experiment from a previous version much better and want to bring them back without loosing all the other changes you did on the way. Or, you might want to create a branch not from the last commit but from a way earlier commit.

git checkout

- `git checkout` allows you to move around in your commit history and between branches
- `git checkout HASH` leads you to an earlier version of your project on your **current branch**
- `git ceckout branchname` leads you to the latest commit made on the branch you've chosen to go to. Every commit you make now will be recorded on the branch you checked out to. To record changes to another branch, you first have to do a `git checkout other-branchname` again
- checking out to an earlier commit brings you in the so-called *detached HEAD stage*. This means that you are currently not connected to any branch. To make changes on the version you're currently on, you first need to create a new branch, then make the changes. When you made your changes, you can merge the new branch with your `main` branch to incorporate the changes to main

Task 5

1) Create a new branch from the `eyetrack` branch and make a commit on the new branch.
2) Switch to the `main` branch and make a commit.
3) Move to your third commit. Create a branch from this commit.
4) Switch back to the `main` branch

Answer

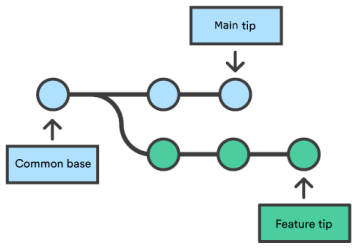
1) check with `git branch` if you're currently on the `eyetrack` branch. If not: `git checkout eyetrack`. Make a change. `git add change` & `git commit`.
2) `git checkout main`. Make a change. `git add change` & `git commit`
3) `git log` to find hash of third commit. `git checkout hash-of-third-commit`. `git branch new-branch`
4) `git checkout main`

Task 6

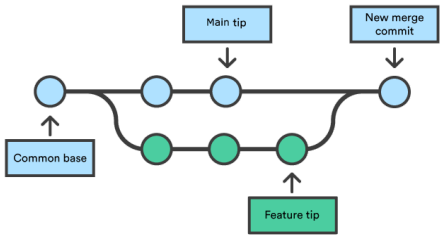
How does our development stream look like? Draw your commit history with all the branches like in Fig. 13 shown at the top of this page.

Merging and merge conflicts

Merging means bringing two independent development streams together. Previously, we said we wanted to add an eye tracking feature to our experiment code. Now that we worked on it a bit we are certain that we can incorporate eye tracking into our experiment. This means, we want to incorporate our work from the `eyetrack` branch into the `main` branch. It will look like this [1]:



Independent branches of work.



Merged branches.

Git merge

- Merging always refers to two branches
- with the command `git merge name-of-the-branch`, `name-of-the-branch` will be incorporated into the branch you're currently on
- especially when you start working collaboratively with Git, you will run into `merge conflicts`. This is something common and not a catastrophe. There is a solution to the conflict (see below).

Task 7

Merge the `eyetrack` branch into the `main` branch.

Answer

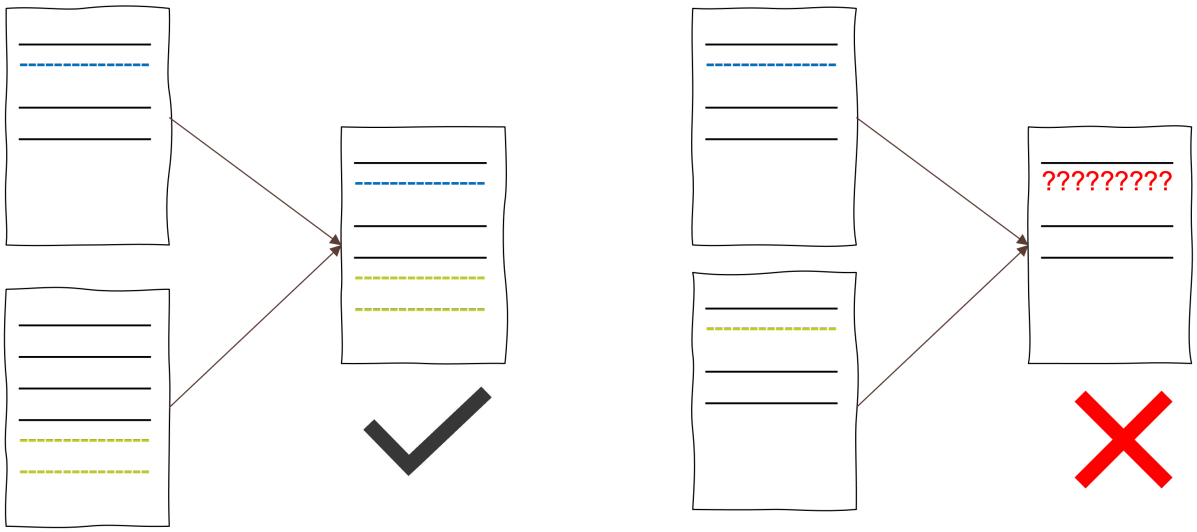
- check if you're currently on the `main` branch. If not: `git checkout main`
- `git merge eyetrack`

Did you run into a merge conflict? If not, do the optional Task.

Optional task to induce a merge conflict

1) `git checkout eyetrack`
2) change something in your code on the eyetrack branch by deleting something in your instructions and writing an alternative text on the **exact same place** as you deleted from. Add and commit.
3) `git checkout main`
4) `git merge eyetrack`

The reason why we run into merge conflicts is because of the way Git tracks the changes in our files. Git inspects our project on a file-by-file and line-by-line basis. This is the reason why a `git diff` can give us such detailed information about our changes. However, this also means that when we make changes on the `eyetrack` branch to a line that already exists on the `main` branch and then want to merge, Git simply is confused and doesn't know which one of the two versions we want to keep.



So, Git is super nice and asks us which one we want to keep by giving us a notice about a merge conflict. A merge conflict looks like this:

```
git merge eyetrack
Auto-merging crtt_exp.py
CONFLICT (content): Merge conflict in crtt_exp.py
Automatic merge failed; fix conflicts and then commit the result.
```

As you can see, Git tells us in which file the merge conflict appeared and asks us to fix the conflicts. We can fix it by opening the file and modifying it so that only one version stays.

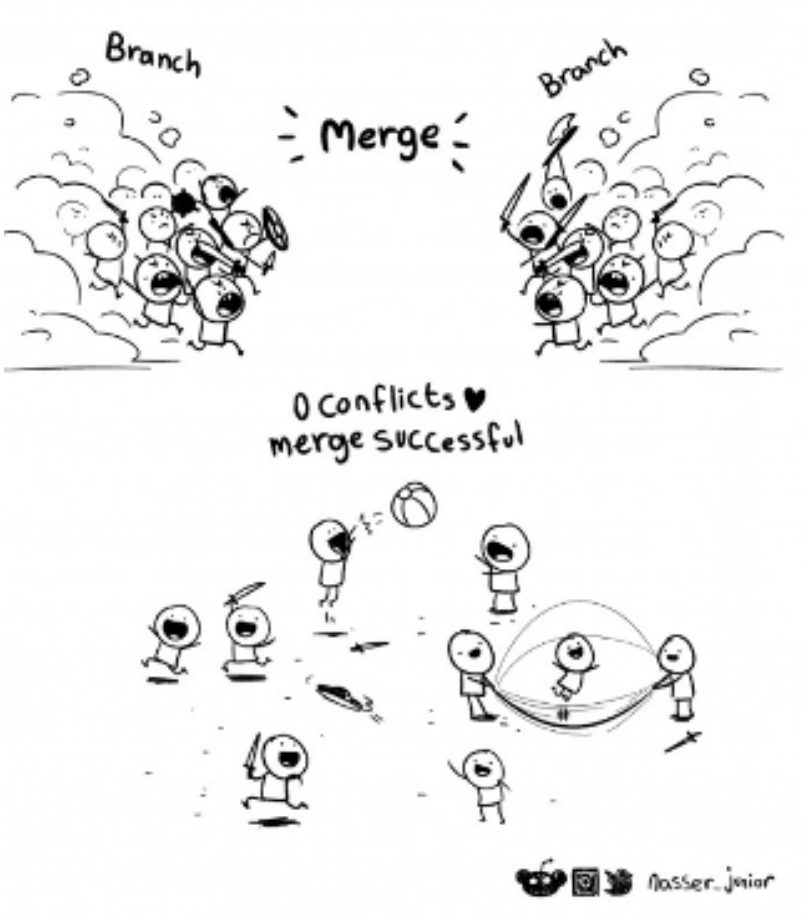
Task 8

- open the file in which the merge conflict appeared
- at the place where the merge conflict is you will see this:

> <<<<<<< HEAD
> this is the content of the file on `main`
> ...
> =======
> this is the content of the file on `eyetrack`
> . >>>>>>> eyetrack

You can see that Git tells you where the merge conflict starts and what the merge conflict is about. Thanks to VSCode you have some buttons where you can click on to choose which of the two versions you want to keep.

- solve the merge conflict
- add and commit the file
- do a `git merge` again
- Check with the Git-Graph feature of VSCode how your project development looks like



Now that we are experts in using Git locally on our machine, we're finally ready to start using Git collaboratively!!!!

P.S.: check out this tutorial for learning advanced git commands.

---

1. Figures retrieved from the Atlassian tutorial ↩

GitLab

Objectives 📍

- What is GitLab
- local vs. remote repository
- push
- pull
- clone
- fetch
- merge
- Merge conflict between your local and remote repository

As mentioned in the introduction, GitLab is a platform for collaborative code development. Everything we did so far in this Git course we did on our local machine and nobody has access to our project. Because collaboration is a major part in science, we need to be able to have a centralized workflow without needing to send all of our files back and forth with emails. This is very cumbersome and also no one ever knows what is the current version because between sending an email and actually getting feedback some time will pass. By putting your project on a centralized repository and giving your collaborators access to this repository, they will:

- always see the latest version of the project
- see the development stream of the project (aka commit history)
- they can retrieve the project and make changes and either commit them directly or ask you if you want to incorporate those changes (`merge request`)
- they can comment on it
- they can open a discussion (`issue`)
- and much more...you can even do project management on gitlab if you want to

All we have to do is to set up a remote repository on gitlab and synchronize our local work with this remote repository. First we need some more vocabulary:

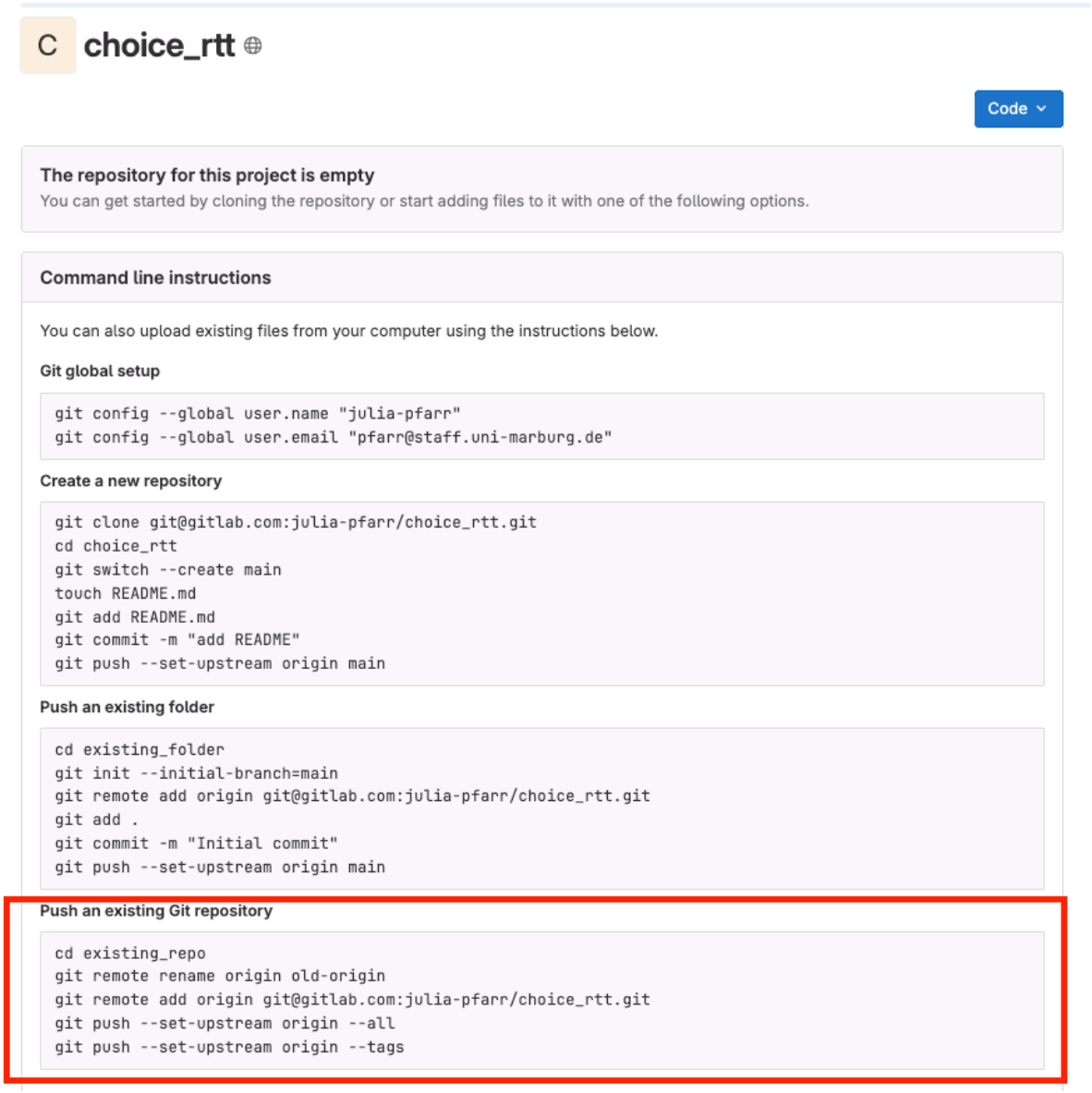
More Git vocabulary

- `git push`: synchronize your local repository with the remote repository by pushing changes in the local repository into the remote repository
- `git pull`: synchronize your local repository with the remote repository by pulling changes from the remote repository into the local repository
- `git clone`: this gives you an exact copy of a remote repository on your local computer
- `git fetch`: fetching the latest version from the remote repository into your staging area

Task 9

- go to the TAM GitLab and log in.
- click on "new project" → "create a blank project".
- give the project the name, preferrably the same name as your local folder has.
- under "Project URL" select your name.
- select "public" as visibility level.
- **uncheck** the box "Initialize with a README".

We are now presented with this page:



This page basically presents you with different options on how to fill your gitlab repository. Because you already have a Git repository, namely the Git project on your local machine, we need the last option "Push an existing Git repo".

Task 10

Follow the instructions given by GitLab for "Push an existing Git repo". After you did all of this, refresh the page.

I get an error when I push via ssh

First, please check that you followed the GitLab ssh instructions on the setup page. If you did and it doesn't work, it's likely that the ssh connection to gitlab wasn't properly stored in your `.ssh config`. Here's how you can solve this:

- navigate in the terminal to your `.ssh` folder. It's in your root directory, so `cd ~/.ssh` should work for everyone
- inside this folder, check if you have `config` file by typing `ls`
- if you see a file named `config` in the list open it. If not, create this file. Both can be done by typing `nano config` (if you have nano as a text editor). Other ways can be `touch config` or `vi config`, whichever works for you
- inside this file you have to write the following (watch the indentation!):

  Host gitlab.com HostName gitlab.com IdentityFile ~/.ssh/id_ed25519 # replace id_ed25519 with the name of your own ssh key
- save and close the file and try `git push --set-upstream origin --all` again

If that still doesn't work, you can also chose `HTTPS` instead of ssh. For this I recommend using a `Personal Access Token`. Because if you use https for push/pull/clone etc., you will always be asked for your `username` and `password`. However, if you use a personal access token, you only have to use it once and git will remember it. For creating a personal access token, follow these steps:

- click on your profile on the upper left and select `Preferences`
- go to `Access Tokens` on the left sidebar
- click `add new token`
- give it a name (e.g., the name of the machine you're currently using, like "julia's macbook"), delete the `expiration date`, and click `api` under `select scopes`
- click `create personal access token`
- copy your token (THIS IS SUPER IMPORTANT! This is the only time you can actually see your token on gitlab, after that it will never be shown to you again, so please make sure to copy and save it somewhere you can find it again!)
- go to your terminal and type the following one after another:
  - `git remote rename origin old-origin`
  - `git remote add origin https-link-of-your-repo` → you can find the https link by going to your gitlab repo and clicking on the blue `code` button on the upper right and switch to `https`
  - `git push --set-upstream origin --all`

▼ Details

- now it will ask username and password. Give your `username` **but instead of the password, put your access token!**

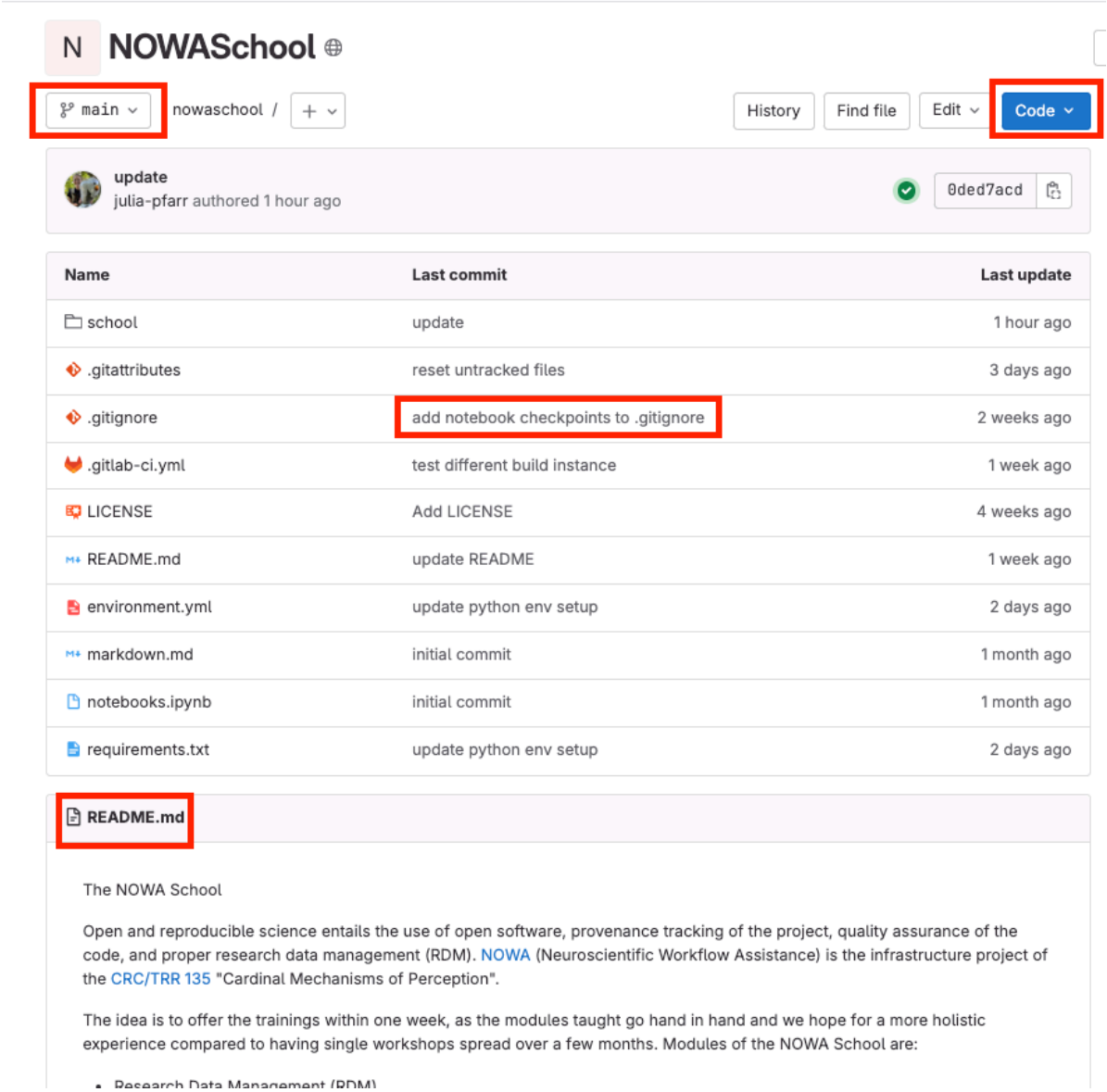
Through the command `git push --set-upstream origin --all` we set up remote branches to track our local branches. The remote branches have the same names as our local branches but have a `origin/...` added. That's how git can distinguish local and remote work.

Local vs. Remote Repository

The local and the remote repository are two different things. You need to actively initiate the synchronization between the two (push-pull).

One important thing to remember for understanding how the synchronization between Git and GitLab works is: Git makes you think you work on the same branches locally and remotely: they follow the same working stream and share the same version history, you don't need to switch between branches. But technically it's handled as if it's different branches: `main` vs. `origin/main`. This is necessary for the communication between local and remote repository to enable functions like `git merge` and `git diff`.

Let's take a look at how the main page of a GitLab repository looks like:



- at the top left we can see which branch of the project is currently displayed in the remote repository. When you click on it, you can change the branch and the files shown will automatically update according to the state of the files on this other branch
- on the top right you'll find the code for the repository which you (and others) will need to connect to the gitlab repo, for example for cloning the repository
- the marked part in the middle shows a commit message. Here it becomes evident why you should not just blindly add and commit everything at once. Because here you can see that your commits are bound to your files. If you always do `git add .` and commit, all your files will have the same commit message so the benefit of seeing at first sight what was last changed in this file is lost

- Lastly, you can see that README files in your repository will always be automatically rendered on the main page of the repo. This is another reason why you should have a nice and informative README file for your project

GitLab in VSCode

If you work in VSCode, it automatically notices that you have a remote repository involved. When you click on the git graph on the left bar you can see the remote repository under `SOURCE CONTROL REPOSITORIES` and when you click on the arrow next to `commit` you can see that now you also have the option to `commit & push` with one click. Tip: if you install the extension "git graph" you can also see your development stream in VSCode.

Working collaboratively

Because GitLab is the best to learn via working collaboratively, this is what we will do!

Task 11

- Find a partner (find instructions on how to do this exercise alone below). Choose one of your GitLab repos to be the one you will both work on during the rest of this session.
- Add the person who is not the owner of the repository as a project member with the role "Developer"

From now on the owner of the repository is called `owner` and the collaborator is called `developer`.

- `developer` clones the repository:
  - navigate with the terminal to your desktop
  - got to the `owners` GitLab repo by searching for their name. Click on the respective repo.
  - click on the `code` button on the upper right
  - copy the "clone via ssh"
  - type in your terminal `git clone repo-link-you-just-copied`
- `developer` opens a file, makes a change, adds and commits it
- `developer` pushes the change to the repo by doing `git push repo-link-you-just-copied`
- `owner` refreshes GitLab repo to see if the change was updated
- `owner` needs to incorporate the changes from the remote repo into the local repo: `git pull repo-ssh-link`
- `owner` opens a file, makes a change, adds and commits it
- `owner` pushes the change to the repo by doing `git push`

Instructions for doing this exercise alone

- create a new folder on your desktop called `developer` and `cd` in it
- clone your own repository into the `developer` folder:
  - click on the `code` button on the upper right
  - copy the "clone via ssh"
  - type in your terminal `git clone repo-link-you-just-copied`
- follow the instructions above by switching between the original folder and the cloned repo in the `developer` folder (best is to open to separate VSCode windows to not get too confused)

For the following tasks, `owner` refers to your "original" folder, and `developer` refers to the new folder that has the clone in it. So, you need to switch between those two folders regularly through changing directories in the terminal.

Project Member Roles and Permissions

The list is sorted from all to nothing, meaning the next lower level inherits the restrictions from the higher level.

- Owner: can do anything that is possible on the repository
- Maintainer: can't delete anything; can't assign, archive, transfer project or change visibility level
- Developer: + can't manage GitLab pages; can't change container registry and application security
- Reporter: + can't manage incidents
- Guest (This role applies to private and internal projects only.): + can't manage issues, merge requests; has almost no project management permissions

git clone

- if you clone a repository it will only clone main
- to get the other branches of the remote repo:
  - `git branch –a` shows you all available branches (e.g., origin/eyetrack)
  - `git checkout eyetrack`: creating a local Branch and Git automatically detects that there is a remote sibling branch
  - git output: branch ‘Party2' set up to track 'origin/eyetrack’
  - repeat for every branch you want to have locally available

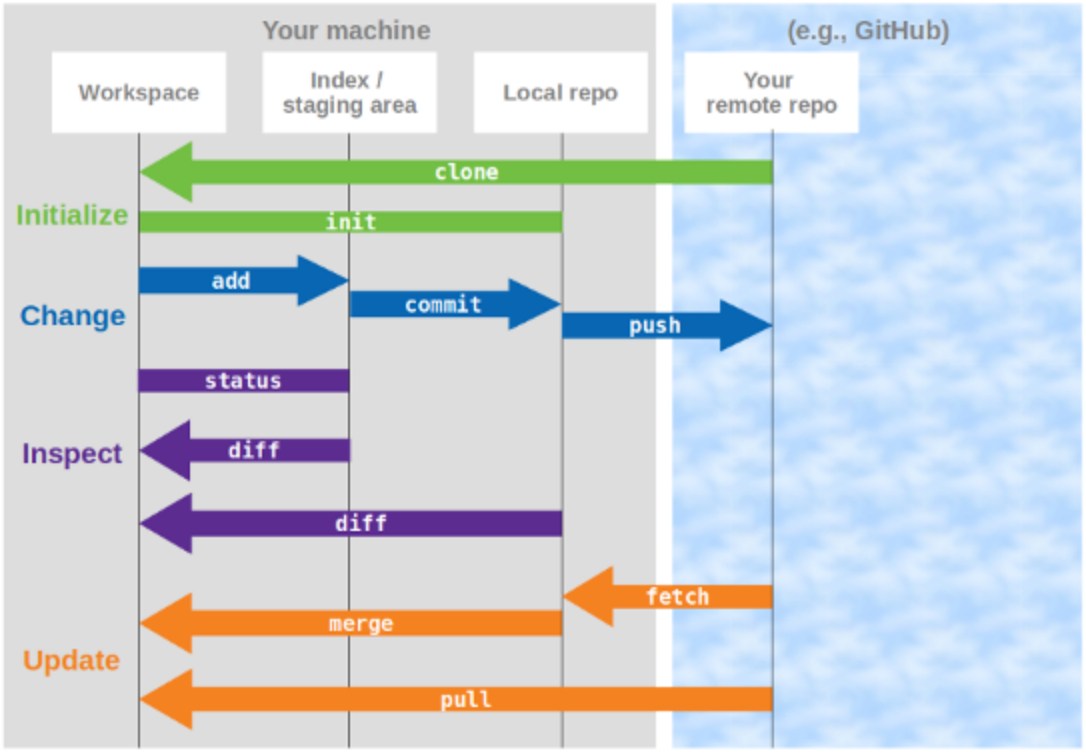
git pull

`git pull` means you are pulling the version from the remote origin branch directly into your working directory which also means an automatic merge of remote and local branch. It therefore updates your Git repo AND your working directory in one command.

git fetch

- `git fetch` means that Git is updating the remote branch in your local Git repo called `origin/main` where it contains the latest *remote version* of the project. It is not updating your local branch `main`.
- that’s why you have to (or can) merge: your corresponding local branch `main` in your working directory contains a different version of the project than the remote branch `origin/main`
- that’s also the reason why you can do git diff: it compares our `main` branch in the working directory with the `origin/main` branch in the Git repo
- `git fetch` is helpful if you want to inspect the changes from the remote repository first before incorporating them locally. Because maybe a merge conflict is hiding in the remote version

Summary gitlab workflow



Merge conflict between local and remote repository

Like mentioned previously, merge conflicts are not unusual and happen more often when you work collaboratively. It is very likely that your collaborator works on the project at the same time as you and sometimes those changes overlap. This is why we want to practice how to solve a merge conflict with the remote repository.

Task 12

- work with your partner from the previous task again (when alone: your `developer` folder is your partner, still)
- `developer` does a `git pull`
- `developer` and `owner` both make a change to the project at the same spot, e.g., modify the same line in the instructions
- `developer` does a `git push` first
- then `owner` does a `git push`
- `owner` should get a message saying:

> rejected! error: failed to push some refs to 'repo url'
> hint: Updates were rejected because the remote contains work that you do not have locally. This is usually caused by another repository pushing to the same ref. You may want to first integrate the remote changes (e.g., 'git pull ...') before pushing again.

This error message is not due to the merge conflict but due to the fact that `owner` made one more commit to the commit history and therefore expanded the commit history. As you both work on the same branch and Git works in a linear way, Git cannot put two commits at the same place but only one after another. Git expects you to have the latest version with all commits before you make another one. So, you will also receive this message when you don't have a merge conflict but simply didn't pull the latest version first.

Task 13

- work with your partner/ developer folder from the previous task again
- `owner` does a `git pull`
- `owner` should get a merge conflict message like the one we saw before
- `owner` needs to resolve merge conflict (we learned how to do that already, yeah!)
- `owner` can now do a `git push`

Tips for working collaboratively with Git

- make use of `branches`. Sometimes weird things happen when everyone is committing to the main branch all the time. Just create a branch for your own work and `merge` the updates from `main` into your branch regularly
- make use of `merge requests` (we will learn this in the next section). This way you can incorporate a second communication channel with your collaborators and give them a chance to not only inspect but also discuss the changes before merging them into the `main` branch
- before you start your work at the beginning of your day, do a `git fetch`, inspect the changes, and `merge` them

---

1. Image by Peer Herholz ↩

Contributing and Collaboration on GitLab

Objectives 📍

- fork
- merge requests
- GitLab for project management

Forking and merge requests

Sometimes you want to contribute to a project on GitLab but you don't have write access. This is usually the case if you are not added as a member to the project. Even if the project is public, you cannot contribute to it without opening a `merge request`. The same is true for your own repos: You can make them public, people can `clone` them but they cannot change something on the project without asking you first through a `merge request`.

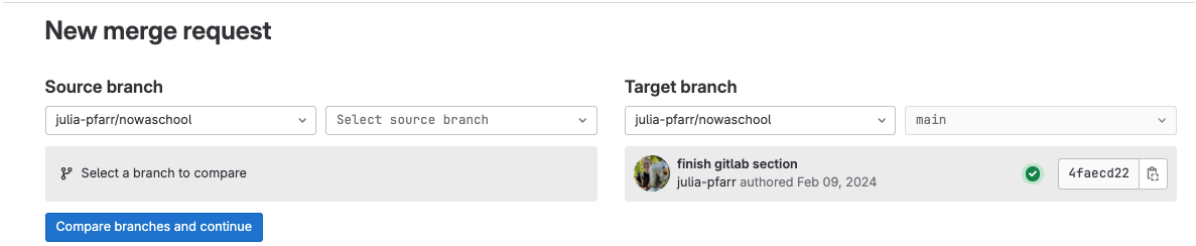
Like mentioned before, even if you are a member of the project and work together with your collaborators, it is always a good idea to work with `merge requests` to foster communication and avoid a ton of conflicts. We highly advise you to ALWAYS work with branches and merge requests!

To open a `merge request` on a repo where you don't have write access to, you first need to `fork` this project. You'll find the `fork` button on the upper right in the respective GitLab repo. Forking a project means copying a remote repository from another user as a remote repository under your username.

Task 14

- Go to this repo in the TAM GitLab
- fork the project
- clone your forked project
- make a change and push it back to your forked project
- open a `merge request` on the original project:
  - go to the original repo
  - click on `merge requests` on the left sidebar
  - select your repo and branch as `source branch` and as `target branch` select the original repo and branch
- on the original repo you should see a new merge request. Now the owner of the original repo can review it and decide to approve and merge it, or not.

This is how your collaborative workflow on GitLab should look like, even if every collaborator has access.



GitLab for project managing

GitLab is actually more than just a code development platform. It has so many additional functions. Under the section `Plan` you can manage tasks (`issues` and `issue board`), keep track of important `milestones` and assign `issues` to `milestones` . You can write a comprehensive `wiki` to store important information about the project (e.g., links to other important resources such as data repo, document meeting minutes etc.).

You can also do automated testing of your code (`continuous integration` / `CI`), build packages or container for apps and `deploy` them to e.g., docker.

It allows you to do a bunch of other stuff which is related to project analytics which is mostly used in industry and exceeds the goals of a science project in academia.

THE END

optional/reading/further materials

- YouTube is full of Git/GitLab/GitHub videos for all kinds of levels and features!!! For example: Brainhack Git introduction or GitHub CI
- Git cheat sheet by gitlab or github
- Atlassian tutorials and cheat sheet
- Troubleshooting: Oh shit git or Dangit git (are the same, but the latter is without swearing)
- Git branching
- Git GUIs
- Advanced Git commands
- NOWA workshops
- really, just type anything you want to know about Git in YouTube and you'll find a tutorial for it.

Get Your Data from GitLab for Computing

To work on your project via the uni marburg JupyterHub you simply have to do a `git clone *ssh-to-your-gitlab-repo*` and follow the Git and GitLab workflow from the tutorial, just as you would on your local machine. For this, you need to have gone through the Setup and Installations for working on the JupyterHub.

Check out these tutorials for introductions to Jupyter and JupyterLab:

- Official JupyterLab documentation
- Very short introduction in the Jupyter ecosystem
- YouTube

Sharing your Project

Contents and service description reference

This article provides user instructions on consuming and publishing research at the TAM DataHub Repository.

You might want to visit the service description to get an overview of purpose, features and status of the TAM-DataHub Repository before reading on.

Pilot (test) operation

Currently, we are implementing the TAM DataHub Repository for publication of research data and code and pilot (test-) usage has started. Please contact your Data Stewards, if you would like to participate in pilot usage.

During this phase, the workflow instructions below will be expanded and updated frequently.

Browsing the TAM DataHub Repository

You can find the TAM DataHub Repository at **tam-datahub.online.uni-marburg.de**. As this is a public service, anyone can browse and download the open access data on this repository (no VPN required).

The TAM DataHub Repository holds various types of items. For research products like *Data and Code Publications* (Datasets) and *Text Publications* you can register a DOI (login required). Other items like *Persons*, *Projects* and *Organizations* allow relating the published contents so you can browse contents not only by author, title and time of publication, but also by responsible group or project. All items are grouped in dedicated *collections* which hold a specific type of item.

Any publication consists of one or more uploaded files and the corresponding metadata. The metadata optionally include relations to various other publications or archived documents. If the authors decide that their data is too sensitive to be published with *open access*, they can decide to use *administrator access* (restricted access). In these cases, you can request permission by the authors and, if granted, will receive an access link via email. This can be helpful in review processes.

The TAM-DataHub supports full-text-indexing of text-files and metadata. Using the search (magnifier symbol), you can search for specific information across the whole platform or individual collections.

Submitting a Publication

- Use your **Marburg University Staff account**, to authenticate via Shibboleth. If you don't have a Marburg University staff account yet, visit the access section of this manual.
- If you are a TAM Member, your account will be granted permission to deposit publications. This is currently a manual process and might not happen immediately.
- As soon as you are grated permission to publish, a sidebar will be visible which allows you to add items in to one of the available collections (see figure). Alternatively, you can click the account button and open your *MyDSpace* page where you can add and manage your submissions.

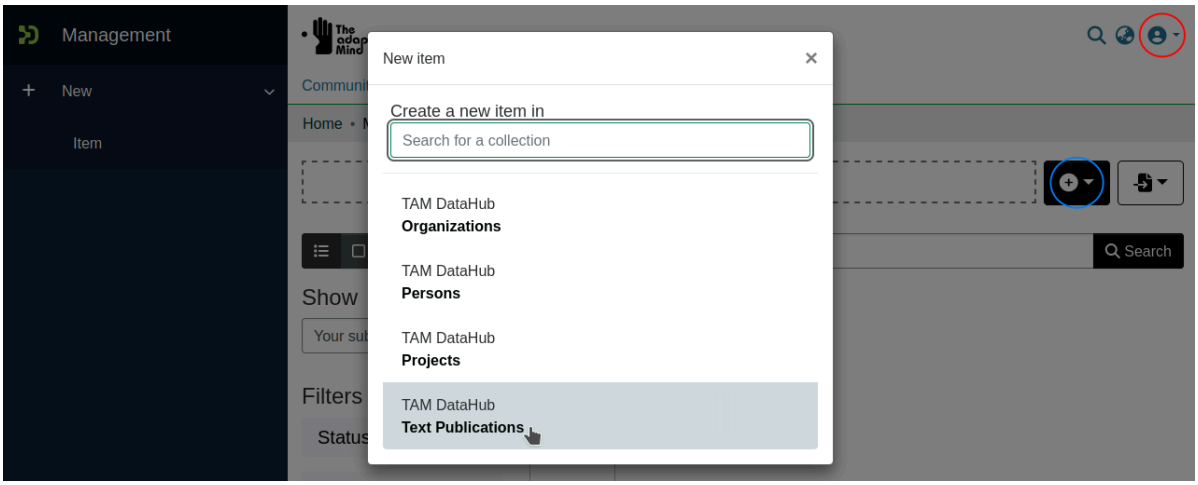


Figure: Select a collection to create a new item. The red circle indicates the "my account" button and the blue circle indicates the "add item" button.

Once you select a collection, a corresponding submission from will open and you may upload files and provide metadata for your submission. The example below shows the submission form for a *Text Publication*.

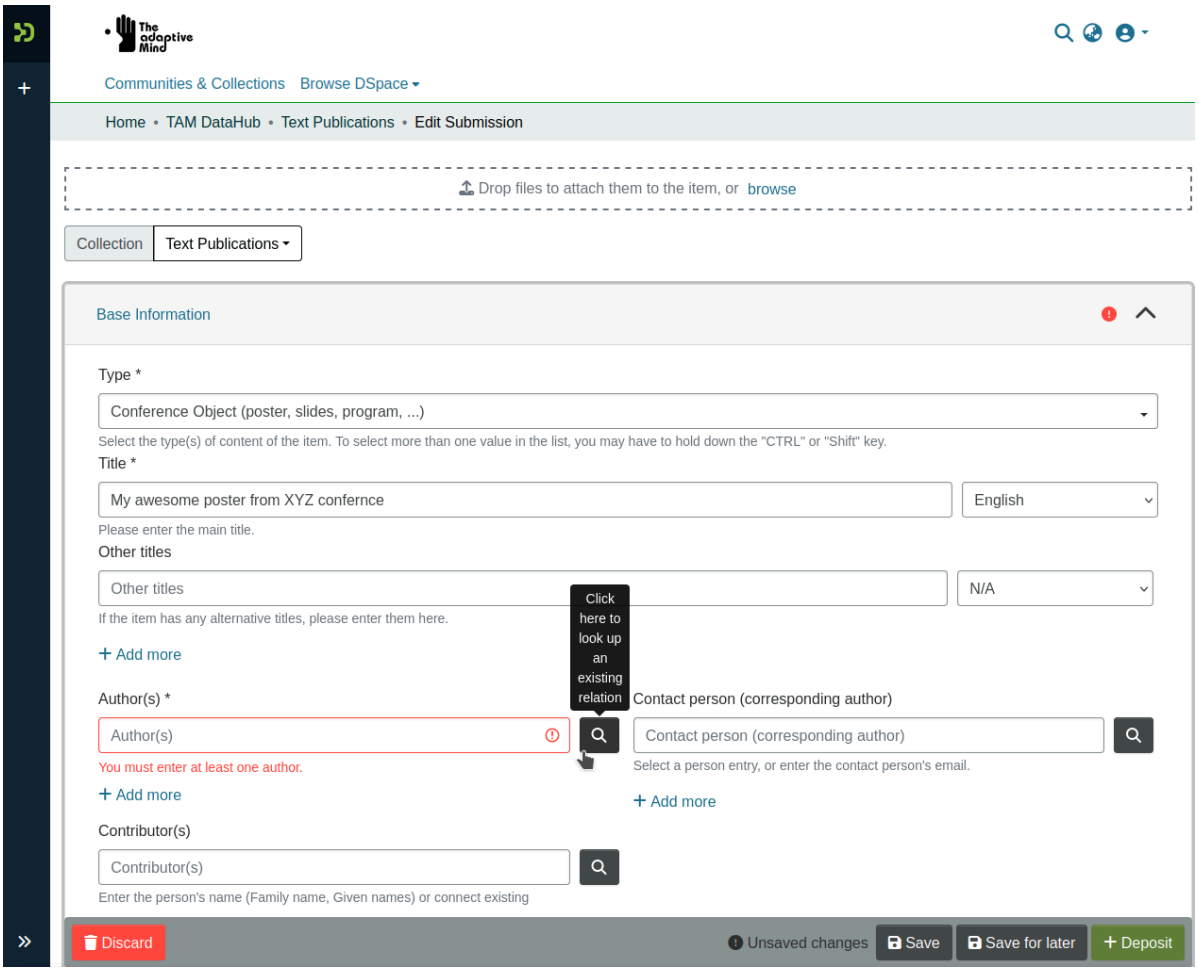


Figure: Submission workflow of a *Text Publication*.

Properties of Metadata Fields

- **Field types**: There are different types of fields e.g. for free text (*Title **), dropdown fields with controlled vocabulary or qualifiers to select from (*Type **) and fields that require specific input formats (*Date **). See also Figure below.
- **Relational fields**: Some field optionally allow to set relations to existing items in the TAM-DataHub Repository like persons, projects or other publications via filtered search (see Author(s) * field).
- **Mandatory fields**: While most fields are optional, some fields (indicated with asterisks) correspond to required metadata such as *Title **. A red hint will appear if the fields are left blank.
- **Repeatable fields**: Most fields like *Contributor* can be used multiple times. These have an "Add more" button attached. Please use this option and do not try to enter how ever separated

values into one field.

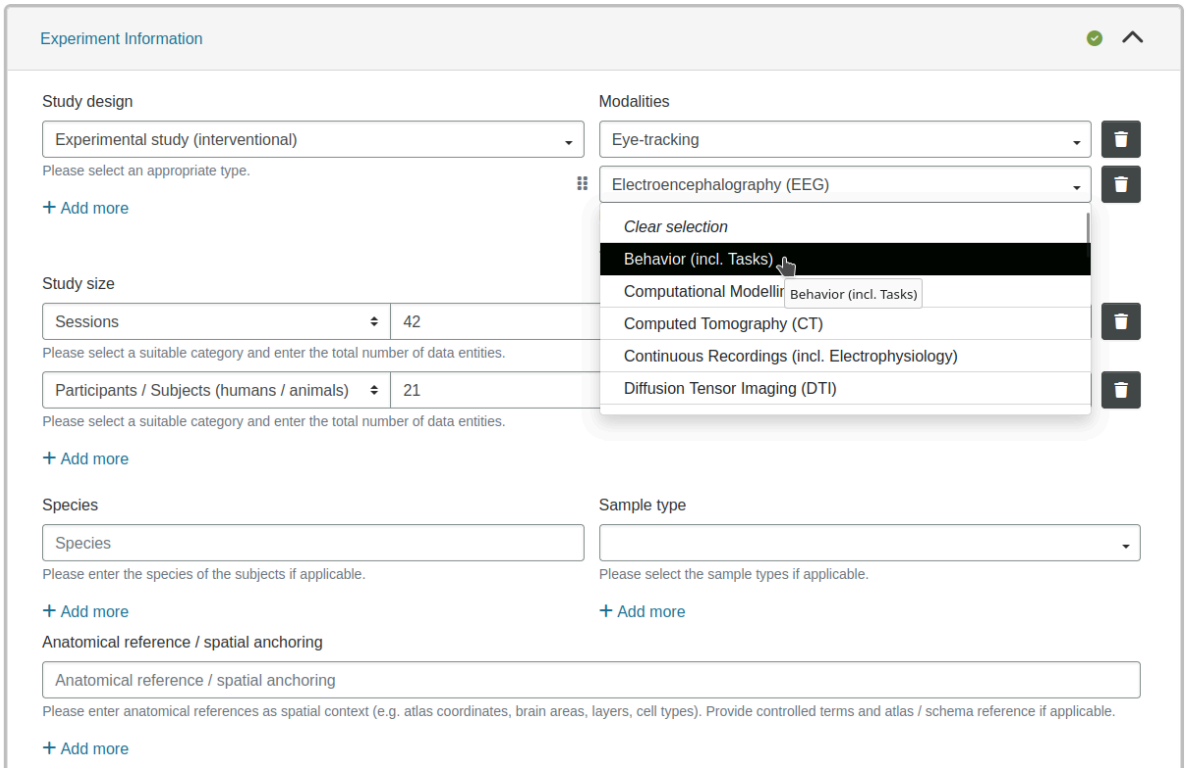


Figure: Properties of metadata fields in the submission module "Experiment Information". Multiple *Modalities* can be selected from a controlled list (dropdown) and *Study size* allows entries in different dimension such as sessions or participants (select qualifier).

Modular Submission Interface

The submission interface is build in a modular fashion according to topics. Currently the following modules are available for publication types:

- **Base Information**: Bibliographic information like title, authors and dates.
- **General Description**: Descriptive metadata like key words, classification, abstract and research questions.
- **Experiment Information**: Specific metadata regarding e.g. study design, methods and modalities.
- **Data Management and Sensitivity**: Information on data protection requirements, data structure and processing (for dataset items only).
- **Relations and References**: References to *Datasets*, *Text Publications* and other documents which are related to this submission plus the respective type of relation (references, is part of, requires, ...).

- **License**: Define the rules by which your publication can be re-used. You can select from frequently used Creative Commons licenses (recommended) or provide other licenses.
- **Item access conditions**: Define if your publication can be discovered by browsing the repository and set its access type (open access, restricted access and embargo / lease time conditions).
- **Upload files**: Browse your workstation or drag and drop individual files (or archives) and upload them to the TAM-DataHub Repository. You can also set file-based permission here once you have uploaded files.
- **Deposit license**: You need to grand a technical license to enable the TAM DataHub to store and process the data which you are about to deposit.

The Figure below shows multiple modules at once. A green checkbox appears once all required metadata of one module is provided.

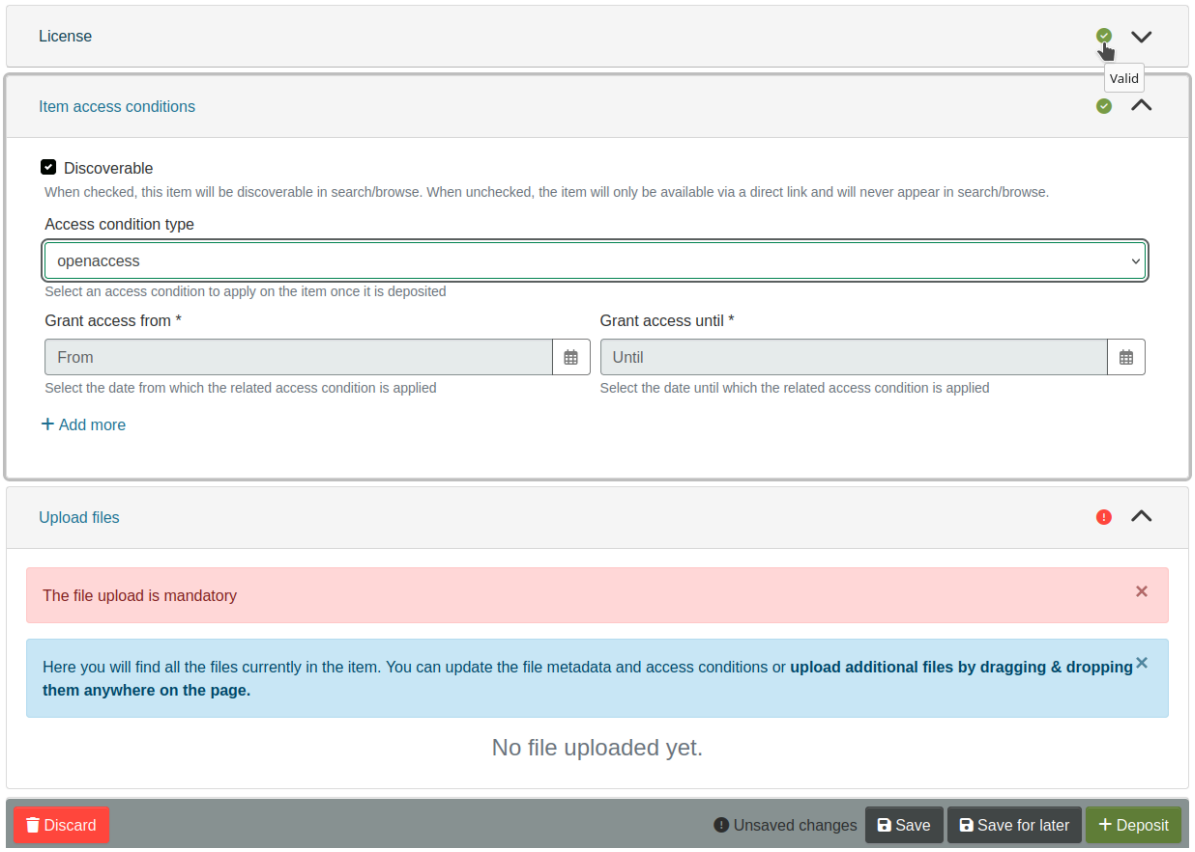


Figure: Multiple modules of a *Text-Publication* submission workflow. Modules are collapsible and red/green checkboxes indicate if all required metadata is provided

At any time, you can pause and resume the submission process by using the "Save" / "Save for later" buttons. When you have finished uploads and annotations, click the "Deposit" button to start the curation and publication workflow (next section).

Curation Workflow

When you click the "Deposit" button, the submission will enter the curation workflow. This means that the TAM Data Stewards get informed of your submission. They will have a final look on your files and metadata to check for plausibility and might provide feedback. There is also a *Private Comment* field in the *General Description* module where you can pass information, comments or questions to the DataHub Team.

You can review your submitted items and their status in your "MyDSpace" page. The figure below shows multiple submission in different stages: The upper submission is a *Dataset* item, which is still in your *Workspace* (see black "Workspace" badge). This means that the submission was saved but not yet completed.

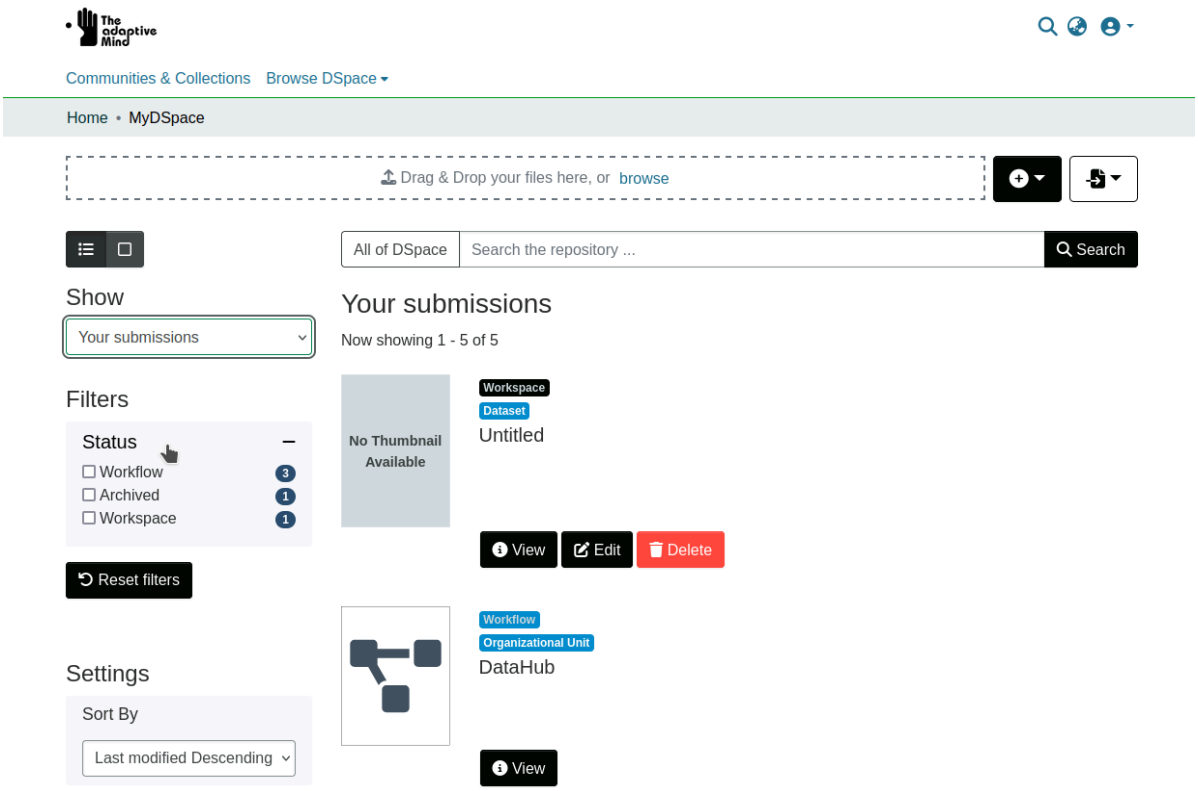
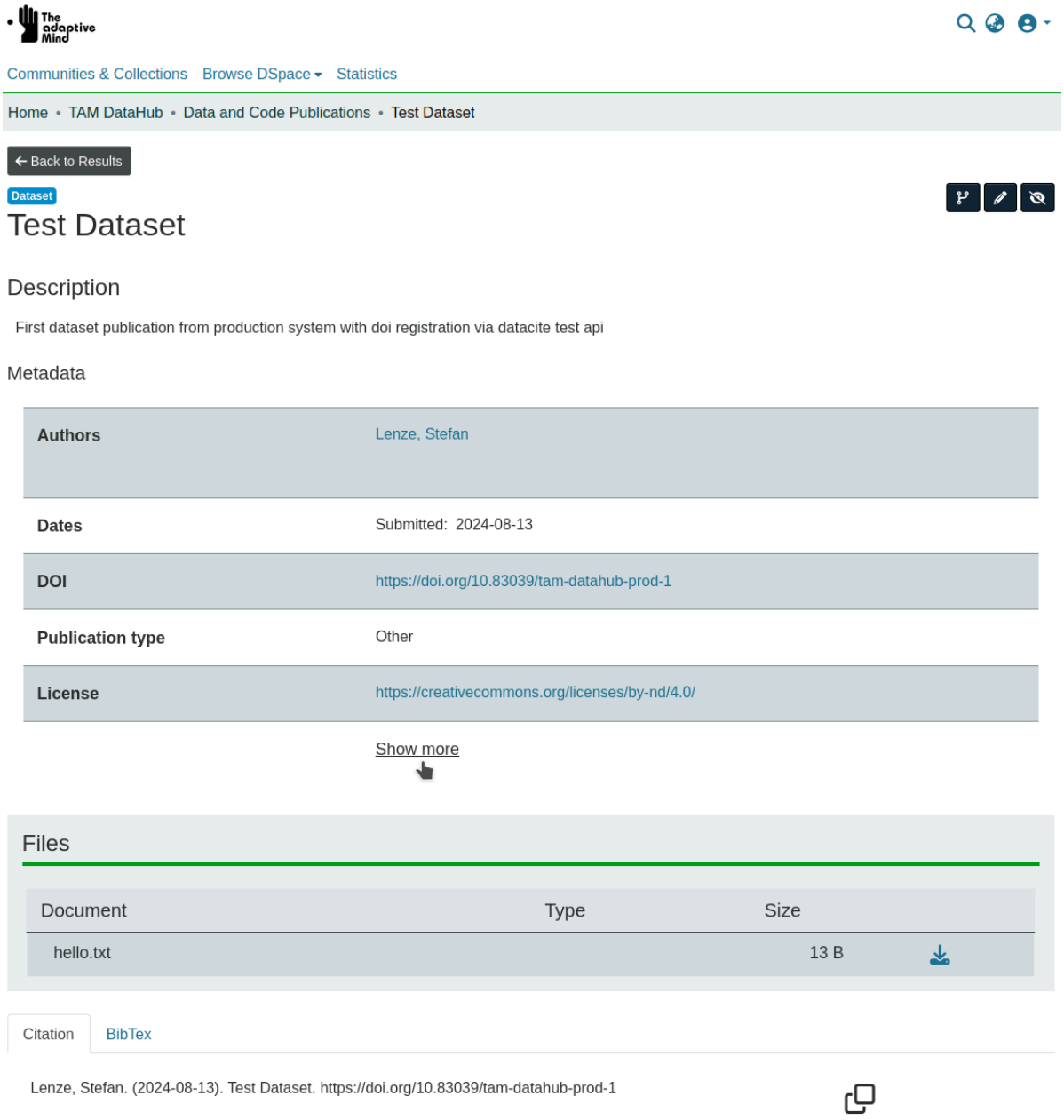


Figure: Submissions at your "MyDSpace" page.

The item below is an *Organizational Unit* which has already been deposited and is now in the process of reviewing. This is indicated by the blue "Workflow" badge. Accepted items are shown with a black "Archived" badge.

After a submission has been accepted, it is published to the TAM DSpace Repository and has specific URL on the platform. The registration of permanent identifiers (DOIs) is a separate process

which will be triggered once per day. Likewise, indexing of submitted text-files and metadata and the generation of thumbnails may take up to one day. The item-page of your submission will be automatically updated once DOI and thumbnail are available (see figure below).



Workflow changes

The future workflow may change according to user feedback.

Thank you for reading the

TAM DataHub

User Manual

Find the up-to-date version online at
https://uni-marburg.de/pMoBmv

Funding

This work was supported by “The Adaptive Mind”, funded by the Excellence Program of the Hessian Ministry of Higher Education, Research, Science and the Arts, and supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—project number 222641018 - SFB/TRR135 Project INF.