&

GitHub

Workshop

How we learned to love the git

Who are we?

Assistant Professor, Research, Health Data Science (HeaDS).

Data Scientists, SUND Data Lab, Centre for Health Data Science (HeaDS).

Post-doc, Computational Biology Laboratory, Danish cancer society Research Centre (DCRC).

Jonas Sibbesen

Diana Andrejeva

Henrike Zschach

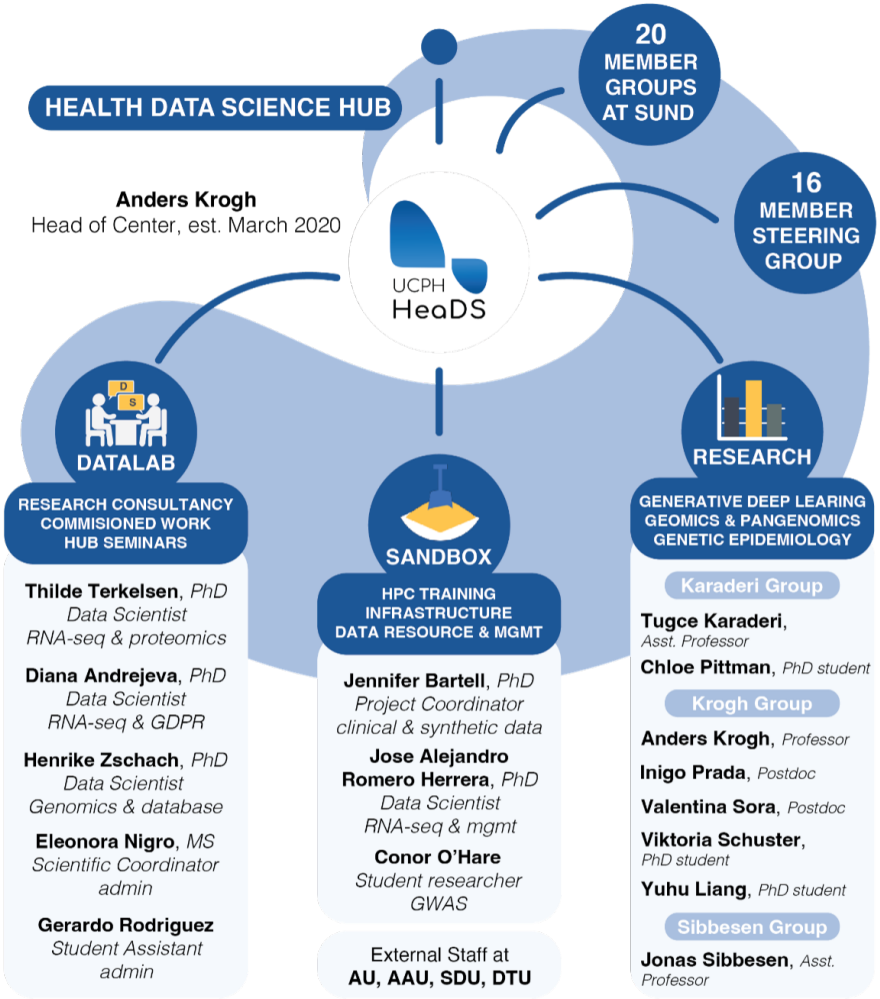
Matteo Tiberti

HeaDS

Center for Health Data Science

- **Conduct Health Data Science research.**
- **Serve as the UCPH Data Lab.**
  - Be the hub for health data science research.
  - Provide SUND researchers with health data science services & training.
- **Develop National Health Data Science Sandbox for Training and Research**

https://heads.ku.dk/



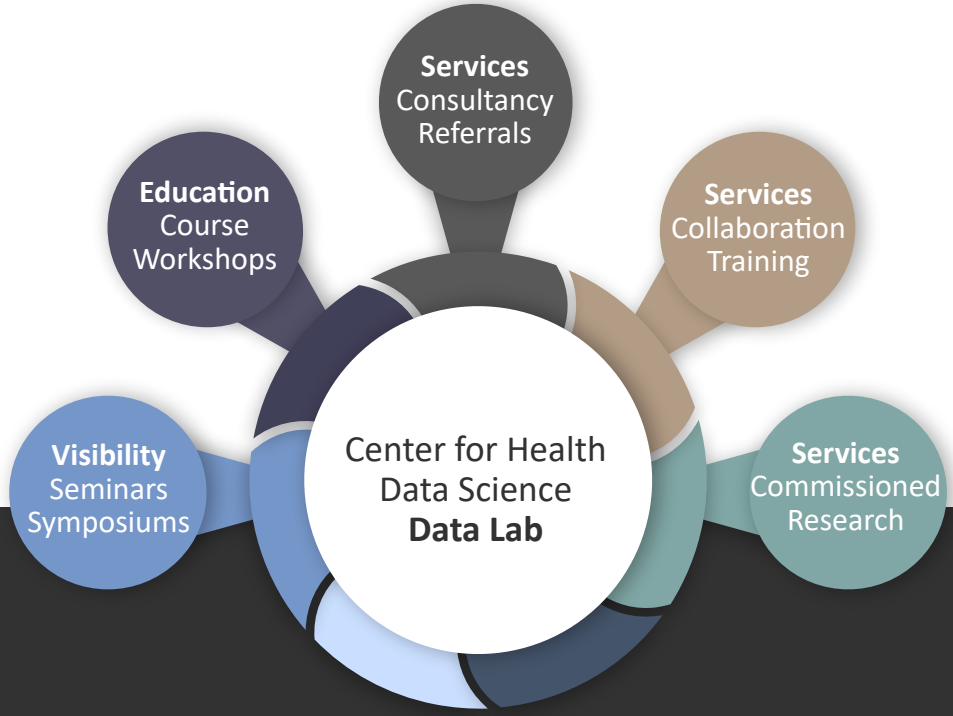
SUND DATA LAB


Diana Andrejeva


Henrike Zschach


Eleonora Nigro

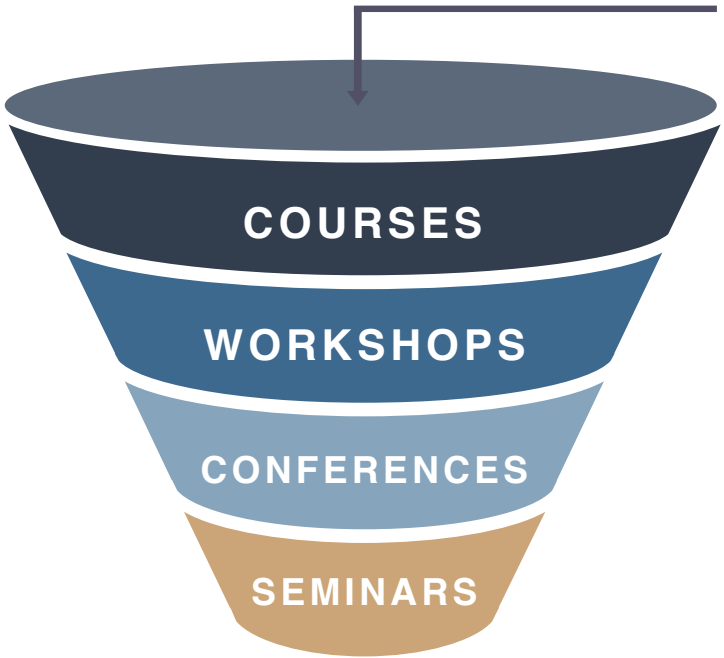

Thilde Terkelsen



Website - https://heads.ku.dk
Email - datalab@sund.ku.dk

Follow us on **Twitter** at **@ucph_heads**!
Slack channel and **mailing list**
courses, conferences, research activities, job postings, etc. within the field of data science.

Center for Health Data Science



git/Github Workshop (Reproducible Science): **Today**.

From Excel to R (module 1): **March** & **June**, 2023.

Python Tsunami: **March**

GDPR and personal data in biomedical research: **April**

Introduction to RNA-seq analysis, **January** & **June**

Program

9.00-9.45: Theoretical introduction

9.45-10.15: Creating a git repository

10.15-10.30: Coffee Break

10.30-11.15: Making changes and checking history

11.15-12:00: Creating and merging branches

12.00-13.00: Lunch break

13.00-13.30: Going to the past and fixing mistakes

13.30-14.00: Interacting with a remote repository

14.00-14.30: Working together on GitHub

14.30-14.45: Coffee Break

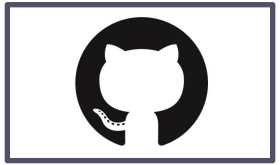
14.45-15:30: Git & RStudio

15.30-16.00: Q & A (and help with personal repository)

Resources

- Git Pro Book:
  https://git-scm.com/book/en/v2

- GitHub Skills:
  https://skills.github.com



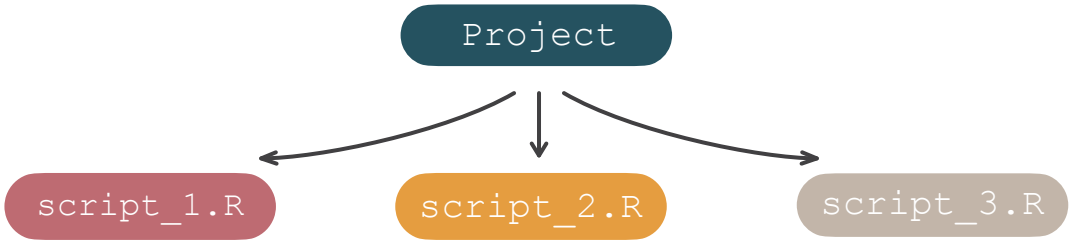
- Git Tutorials:
  https://www.atlassian.com/git/tutorials

- Git Cheat Sheet:
  https://education.github.com/git-cheat-sheet-education.pdf

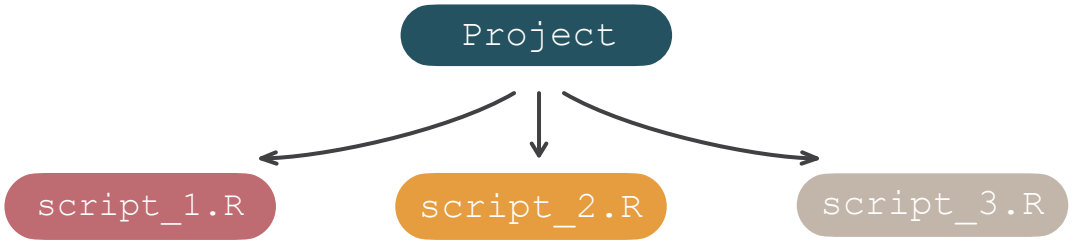
Justification

Project

Justification

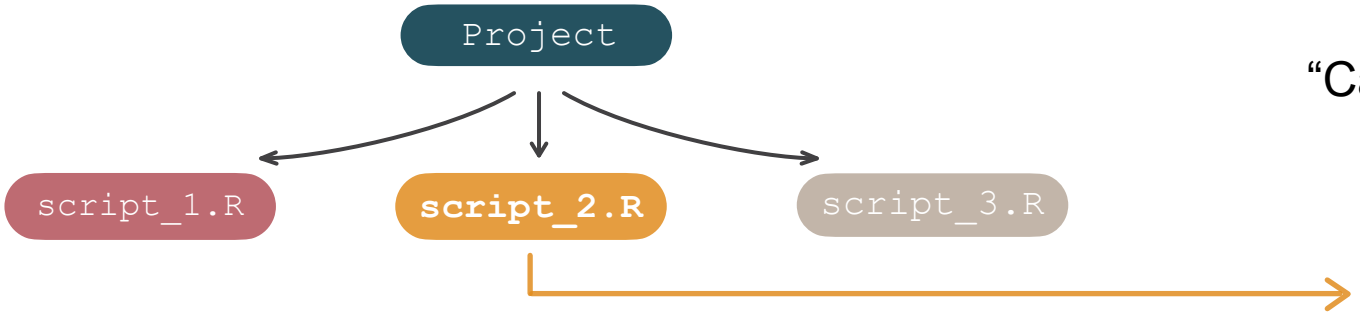


Justification



README

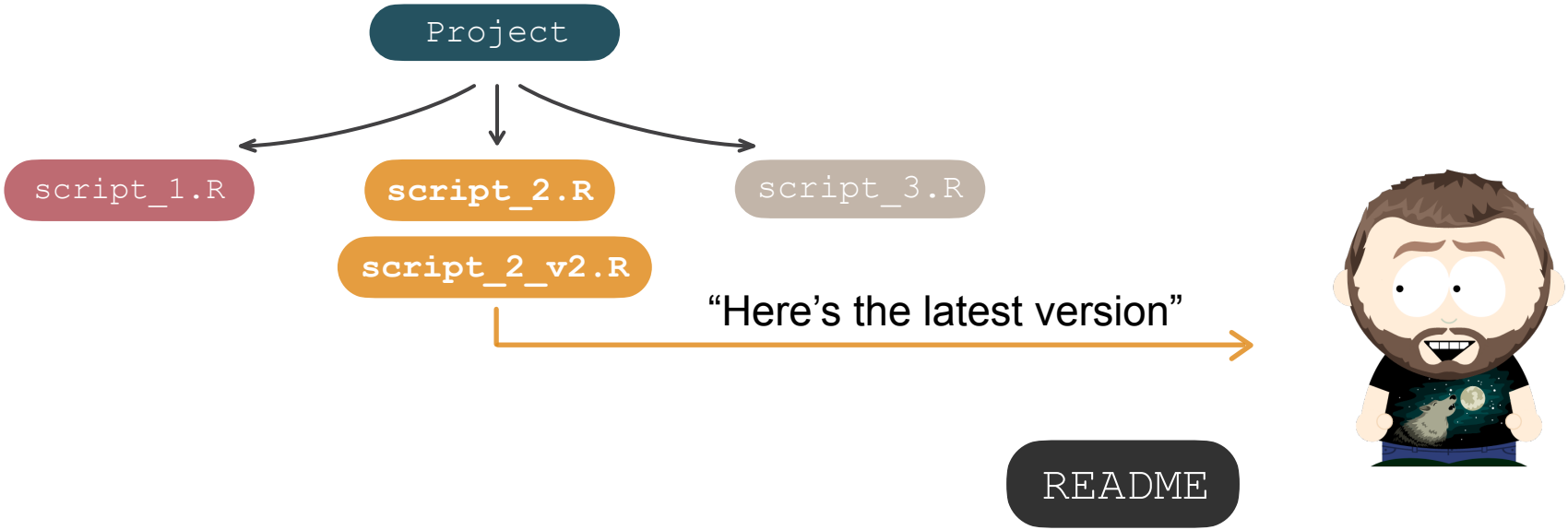
Justification



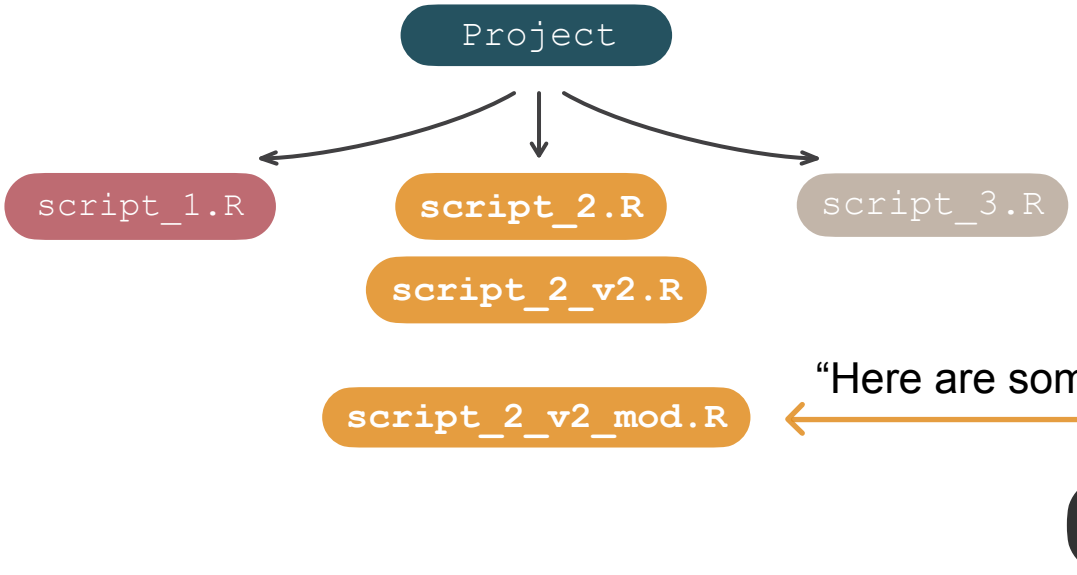
README

“Can I give it a try?”

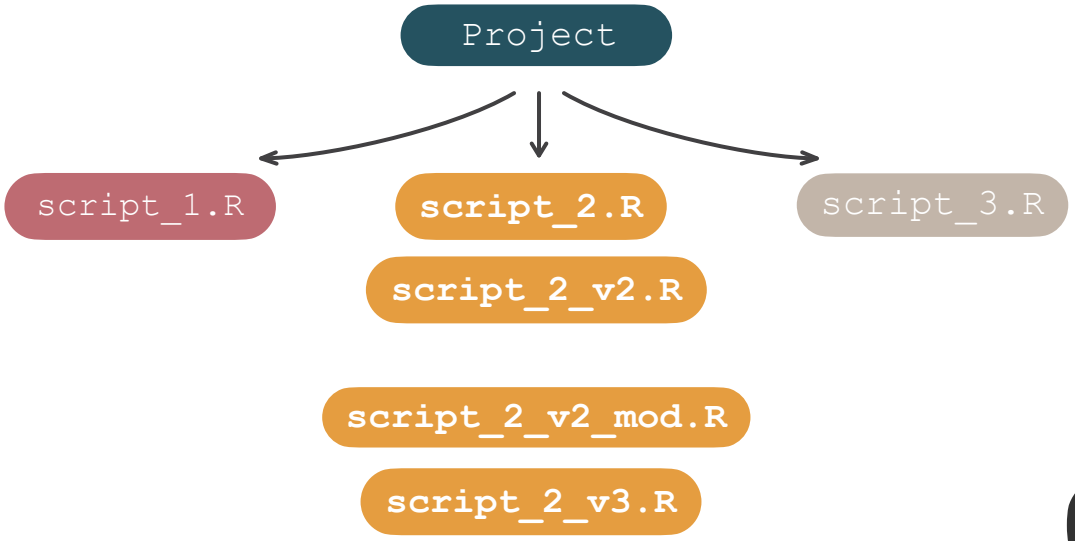
Justification



Justification



Justification



README

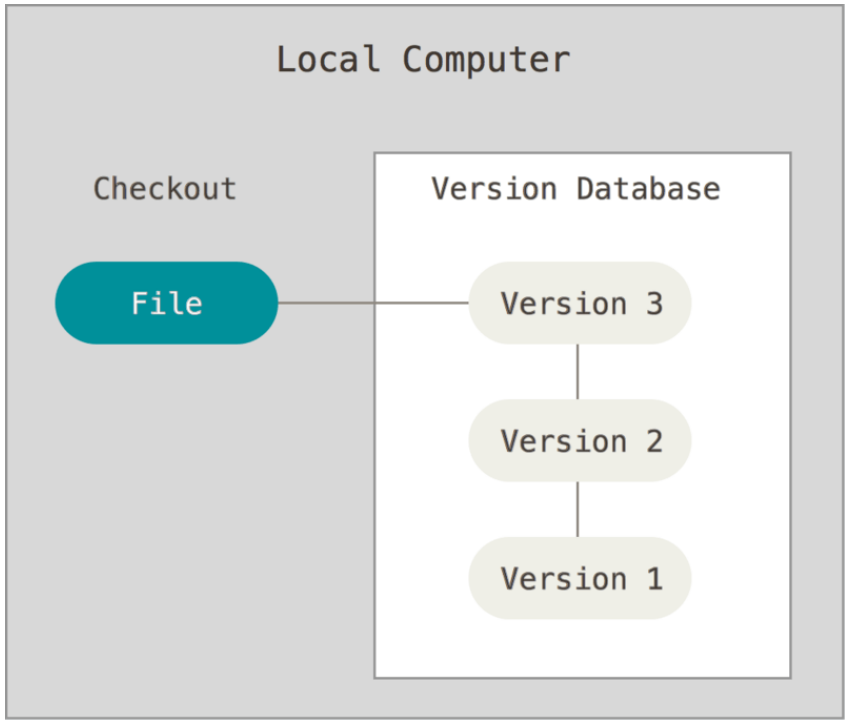
Justification

Keeping track of manually means…

- Significant overhead
  - Organization
  - Documentation
  - Communication
- Error-prone and time-consuming
- Makes collaboration difficult
- Six months from now?

Version control systems

"Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later (**history**)"



Version control systems

Referring to and keeping track of specific versions

- When was “that” change introduced (backtracking)?
- Includes metadata (who, what, when, why)

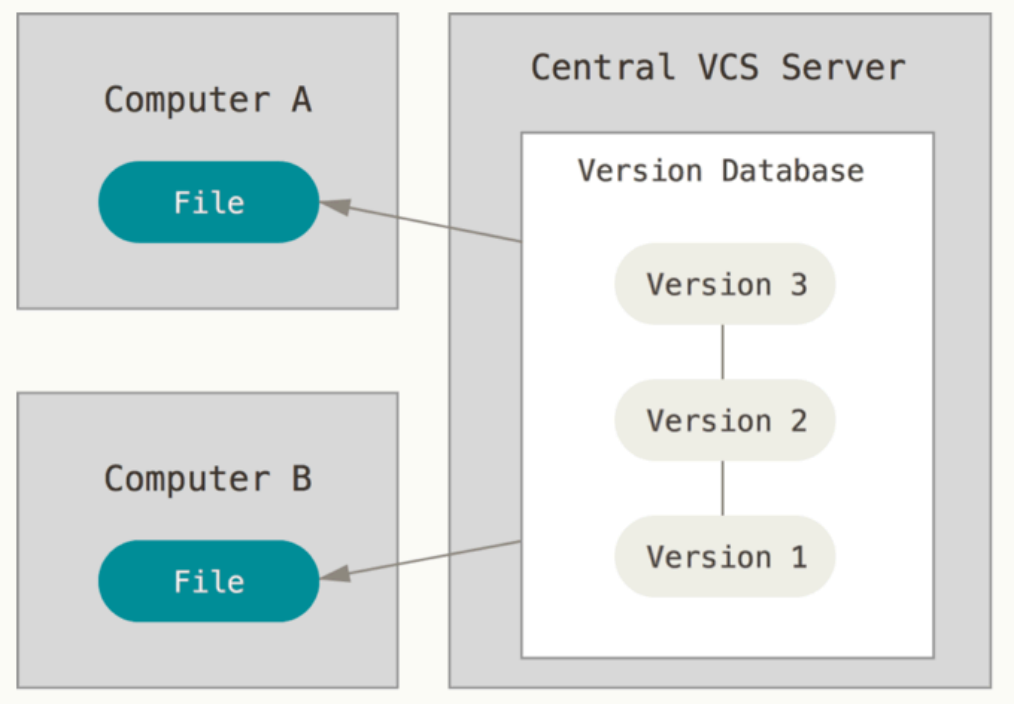
Be organized and maintain long-term control over your codebase

- Experimenting is easy and organized
- Experiments are self-contained

Keep several versions of code at the same time, <u>without clutter</u>

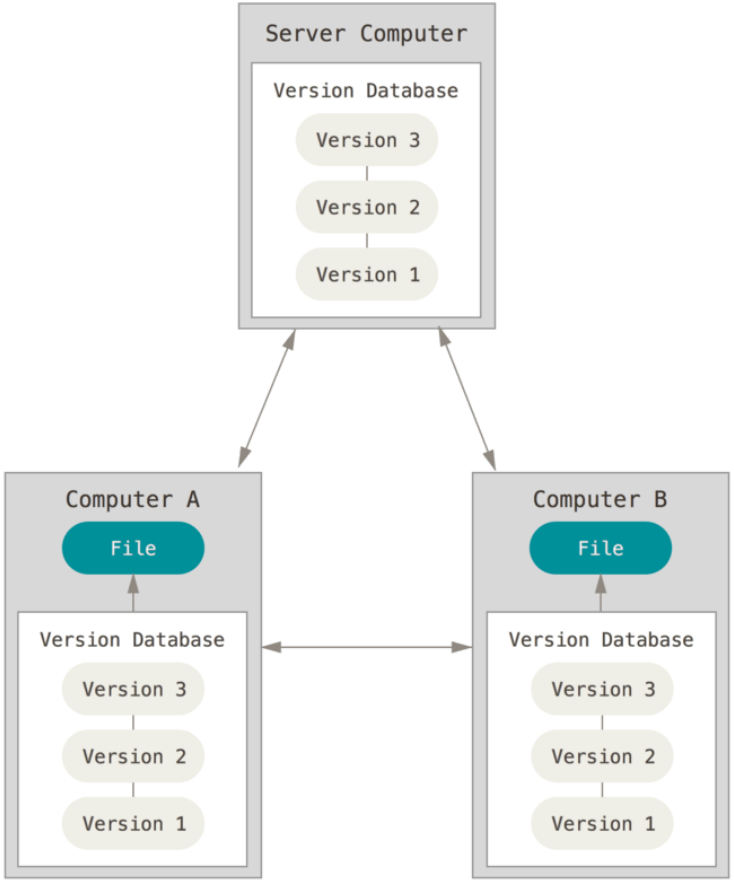
- Keeps known working versions safe
- Able to easy retrieve old versions

Centralized version control systems



Distributed version control systems (DVCS)

- No single point of failure
- Offline work is possible
- Performing most operations is fast
- The system keeps everyone synchronized…
- …but asynchronous work is possible



git and GitHub

Git is a DVCS software, created by Linus Torvalds for the management of the Linux kernel.

Other clients exist.

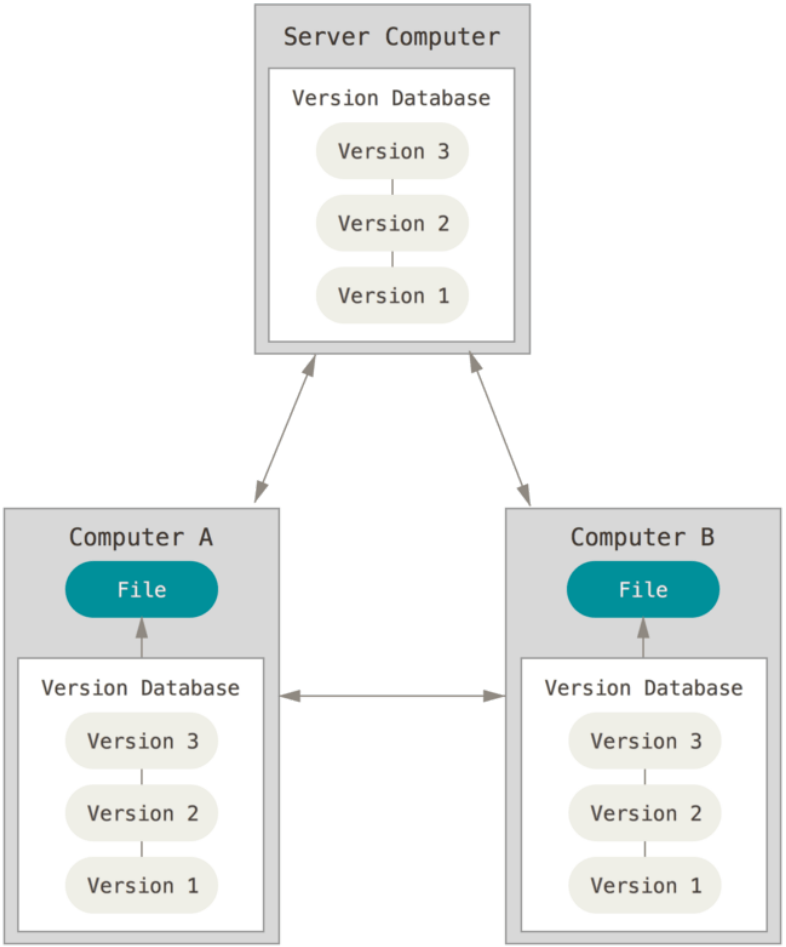
https://git-scm.com

GitHub is a web-based hosting service for version control that uses git

Other services exist.

https://github.com

git and GitHub

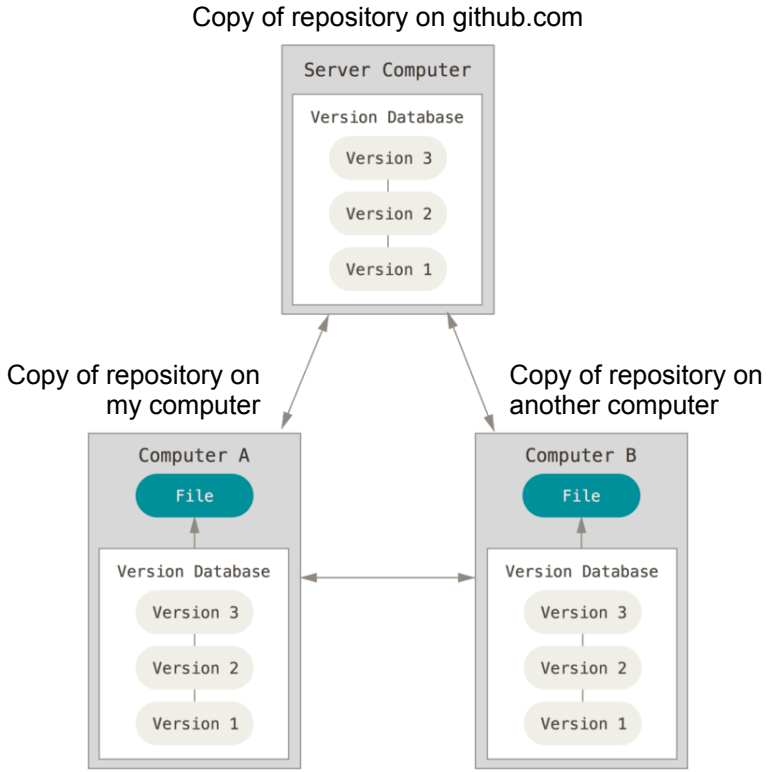


Useful concepts

Working directory (file tree):
All the files (organized in directories) that we want to keep track of, originating from a single directory.

Repository (repo)
Place in which changes to the working directory are recorded. Can be local or remote. Several copies can exist.

Remote repository
Repository that is linked to our local repository. Often a repository hosted on a site such as Github.



Why GitHub?

Helpful web interface:

- Visually browse and explore
- Community aspect (Issues, Wiki, references …)
- Granular access control

Public space: Track records, visibility, transparency, reproducibility

Easy fork/merge system; easy contributing to projects

Remote repository; backup

Easy to collaborate on the same repository

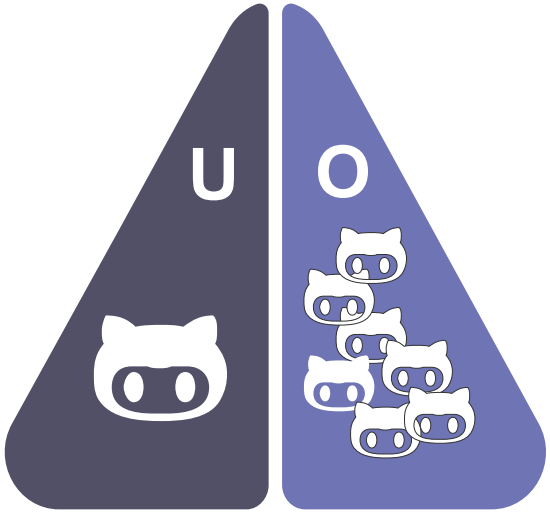
GitHub

GitHub is organized in

- **U** User accounts:

  Personal repositories that have a single owner.

- **O** Organizations:
  - Designed to better accommodate the needs of a team (e.g. lab).
  - One or multiple owners, and sophisticated access control systems to the repos.



GitHub

GitHub repositories can be:

PUBLIC

- Accessible to everyone
- Usually released with an open source license
- Still possible to have control over how many people interact with the repo.

PRIVATE

- Not accessible to the public.
- Accessible only from allowed accounts – requires password.

NOTE: Organizations have further options to limit and organize access to repositories

GitHub

MOST POPULAR

Free

The basics for individuals and organizations

- Unlimited public/private repositories
- 2,000 Actions minutes/month
  Free for public repositories
- 500MB of Packages storage
  Free for public repositories
- Community support

$0 per month

Create a free organization

Team

Advanced collaboration for individuals and organizations

- Everything included in Free, plus...
- Protected branches
- Multiple reviewers in pull requests
- Draft pull requests
- Code owners
- Required reviewers
- Pages and Wikis
- 3,000 Actions minutes/month
  Free for public repositories
- 2GB of Packages storage
  Free for public repositories
- Web-based support

$4 per user/month

Continue with Team

Enterprise

Security, compliance, and flexible deployment

- Everything included in Team, plus...
- Automatic security and version updates
- SAML single sign-on
- Advanced auditing
- GitHub Connect
- 50,000 Actions minutes/month
  Free for public repositories
- 50GB of Packages storage
  Free for public repositories

EXCLUSIVE OPTIONS

- Token, secret, and code scanning
- Premium support

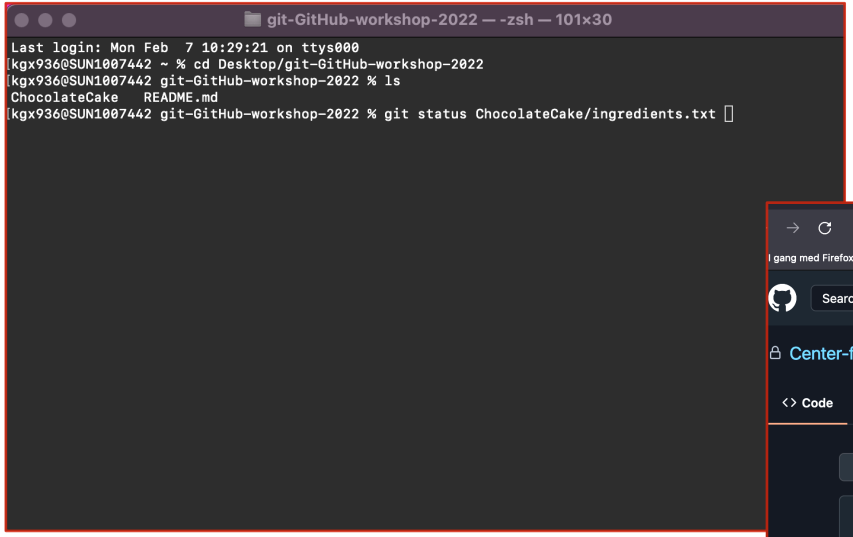
$21 per user/month

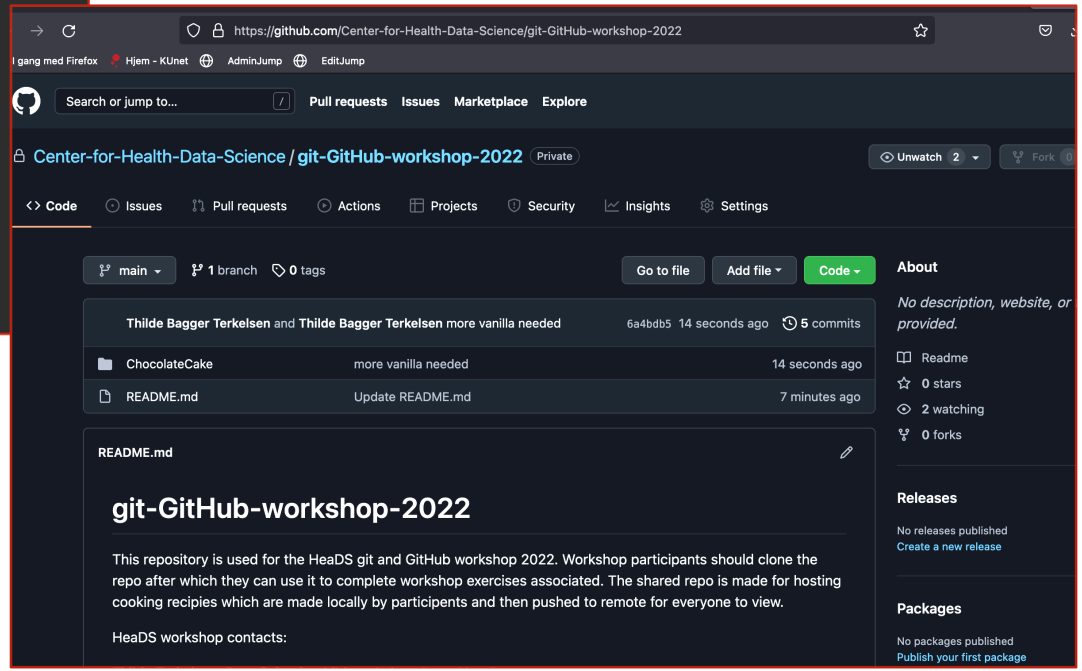
Contact Sales

Start a free trial

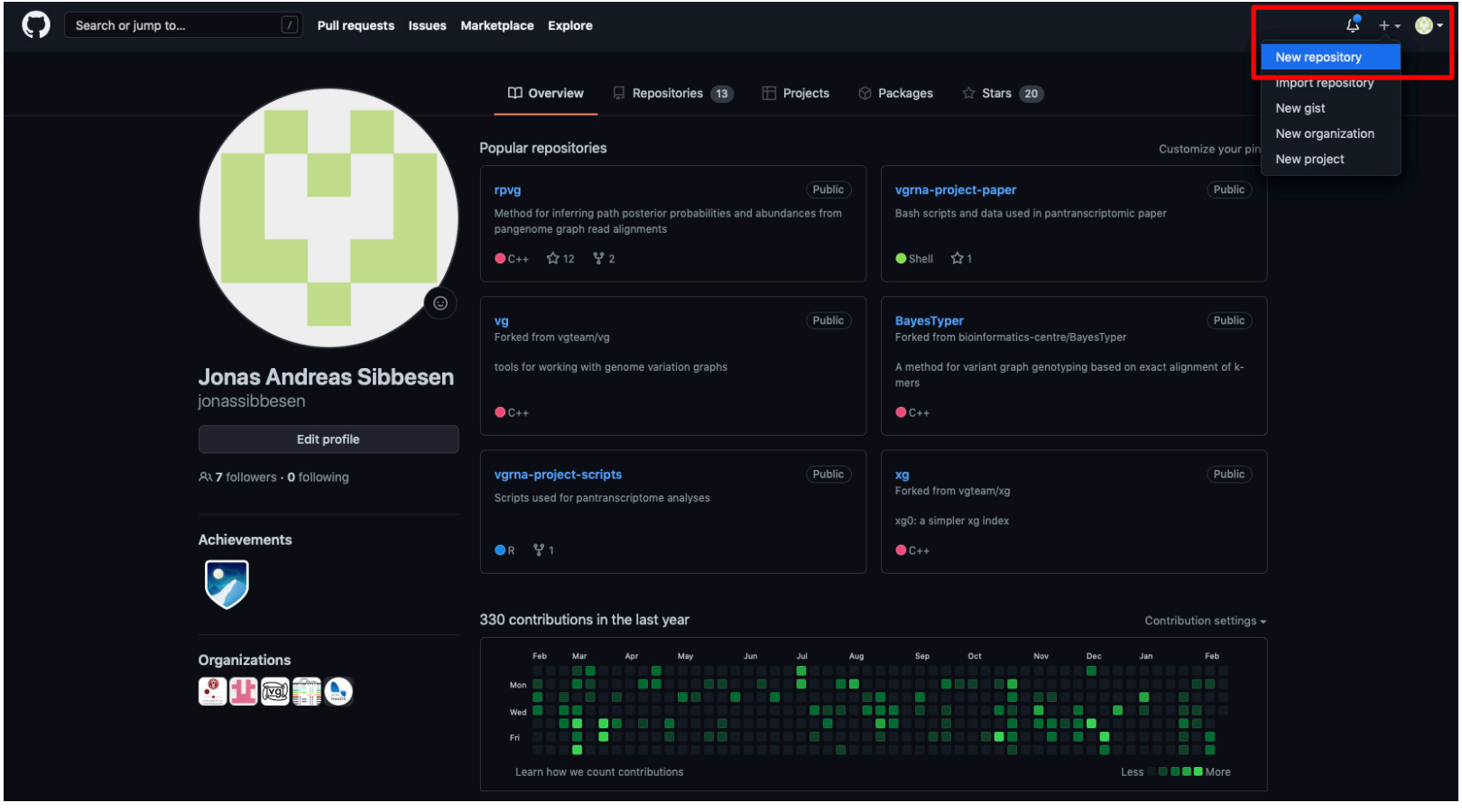
Academics get free “Team” plan!

A look at git and GitHub

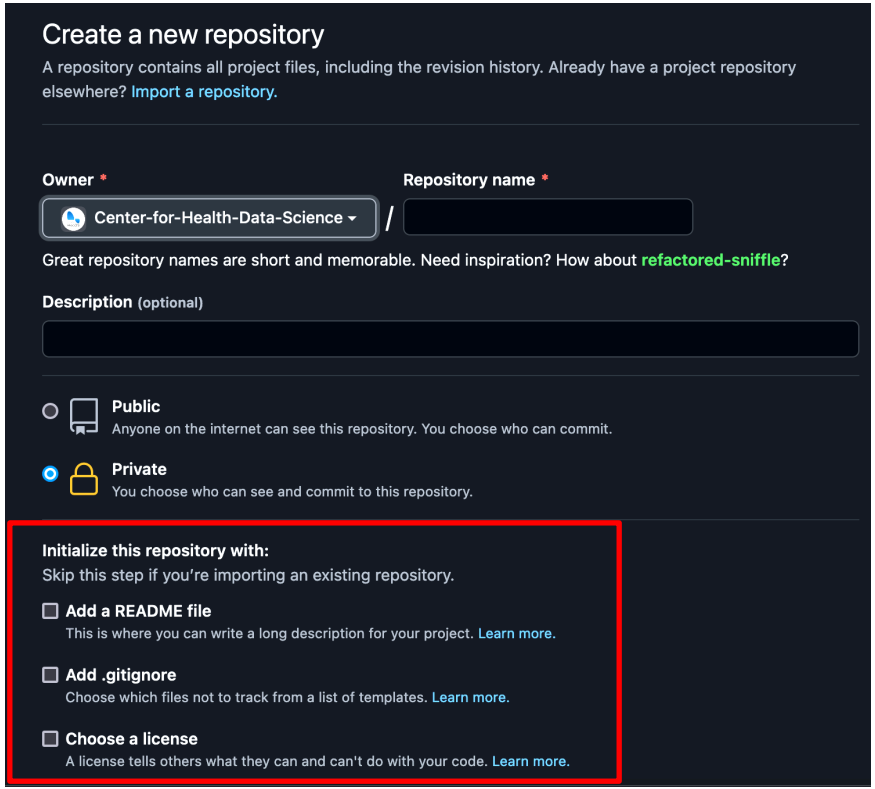




Let’s create a GitHub repo



Let’s create a GitHub repo



README file
File shown by default when someone visits the repository – good for an introduction to what the repository is about. Git markdown format.

.gitignore
Hidden file containing specifications of files/types that should be ignored at all times by git. There are some default choices.

LICENSE
Creates a LICENSE file that contains the license of the content of your repository.

https://choosealicense.com

Choose an open source license

An open source license protects contributors and users. Businesses and savvy developers won't touch a project without this protection.

Which of the following best describes your situation?

I need to work in a community.

Use the **license preferred by the community** you're contributing to or depending on. Your project will fit right in.

If you have a dependency that doesn't have a license, ask its maintainers to **add a license**.

I want it simple and permissive.

The **MIT License** is short and to the point. It lets people do almost anything they want with your project, like making and distributing closed source versions.

Babel, **.NET Core**, and **Rails** use the MIT License.

I care about sharing improvements.

The **GNU GPLv3** also lets people do almost anything they want with your project, *except* distributing closed source versions.

Ansible, **Bash**, and **GIMP** use the GNU GPLv3.

git Syntax

We will use the terminal command `git`. Syntax:

```
git <command> [<options>] [<args>]
```

Args are usually names or files or directories.

Individual commands have help pages:

```
git <command> --help
```

Examples:

```
git add file.txt
git commit -m “changes were done”
git branch -d name
```

Getting started (first time user)

Configure git with your identity:

```
git config --global user.name <username>
git config --global user.email <email>
```

This is important if you want to associate your local changes with your GitHub account. If you want to keep your email private see this guide: *https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-email-preferences/setting-your-commit-email-address*

Change the default text editor used by git (for example to Emacs):

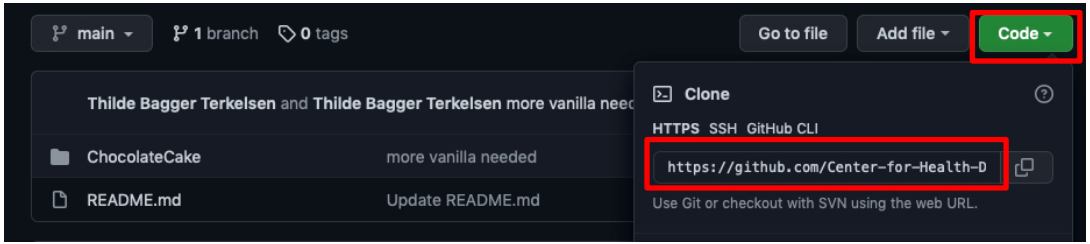
```
git config --global core.editor emacs
```

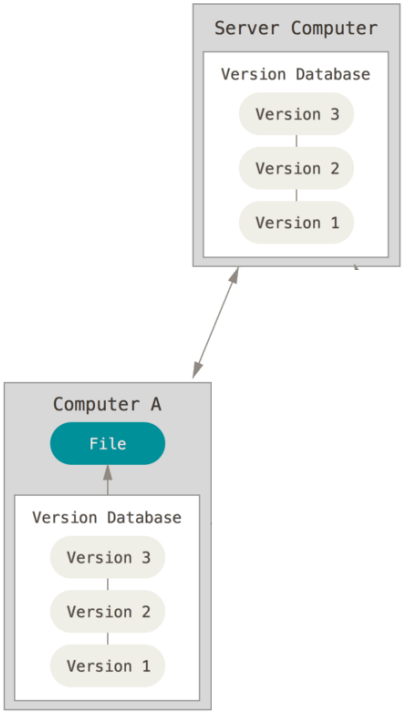
On a Windows system you must specify the full path to the text editor's executable file.

Cloning a GitHub repository

Create a local copy of a remote repository:

```
git clone <GitHub link>
cd <your repo>
```

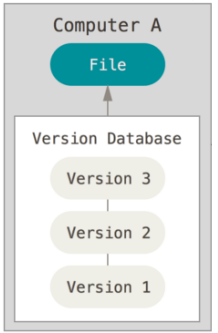




Creating a git repository

Also possible to create a repository from scratch:

```
mkdir my-repo; cd my-repo
git init
```

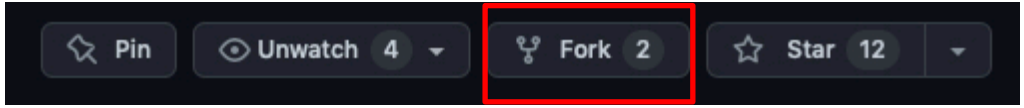


Note that in this case we would not have a remote. Creating a repository on GitHub and cloning it ensures that:

- We have a remote repository
- Remote and local copy are linked

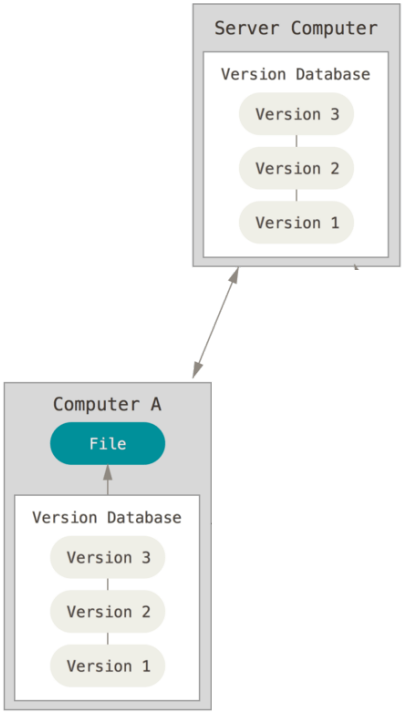
Forking a GitHub repository

Create a you own copy of someone else's repository on Github:



Forks are most commonly used to:

- Propose changes to another persons repository
- Use another persons repository as a starting point for own project



Checking status of a git repo

- Check the current state of the repository:

```
git status
```

  Gives, among other things, information on which files that have been modified.

- Check the name and location of the remote repository:

```
git remote -v
```

Exercise 1

Exercise
1

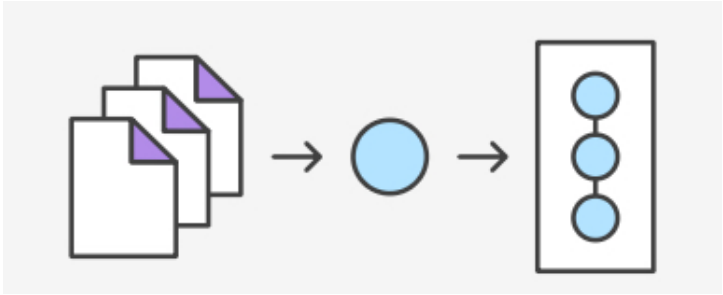
- 1. Configure git with username and email if you have not done so before.
- 2. Create a new public repository on GitHub to hold your favourite cooking recipes. Update the *README.md* file with a description of what the repo contains.
- 3. Create a local copy of your GitHub repository by **cloning** it in a folder on your computer. Make note of the location of the directory that you choose to host your local copy.
- 4. Move to the local repository. Check the status of this repository (don't worry if all the information does not make sense yet). What do you think “working tree clean” means? Check the name and location of the remote repository.

Useful concepts

Commit:
Snapshot of a particular state of a file tree.

Stage (index):
Selection of changes made on a repository to be added to a commit.

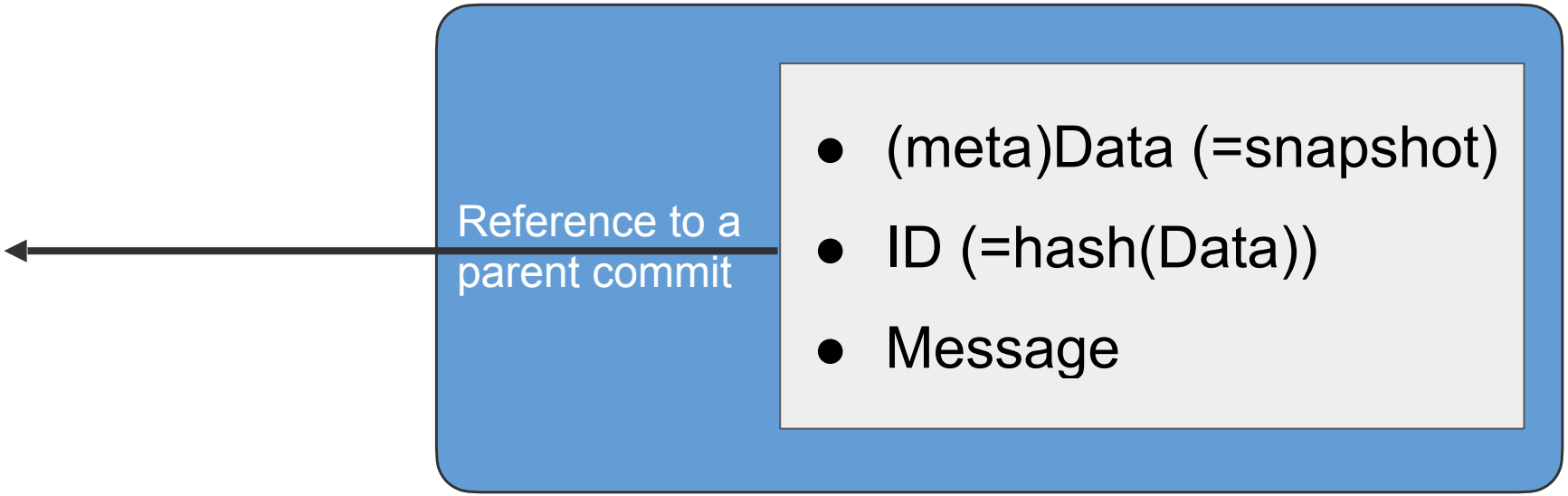
head & branch:
Reference (“pointer”) to a particular commit.



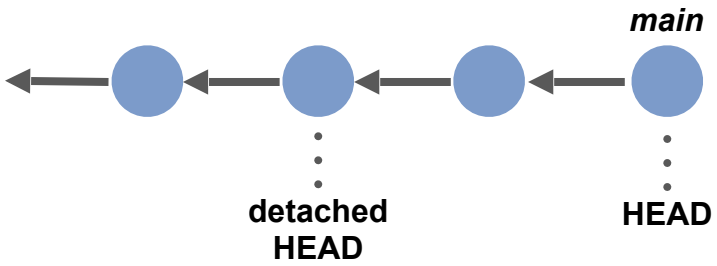
https://www.atlassian.com/git/tutorials/saving-changes

Useful concepts

Anatomy of a commit



Useful concepts



origin = original remote repository

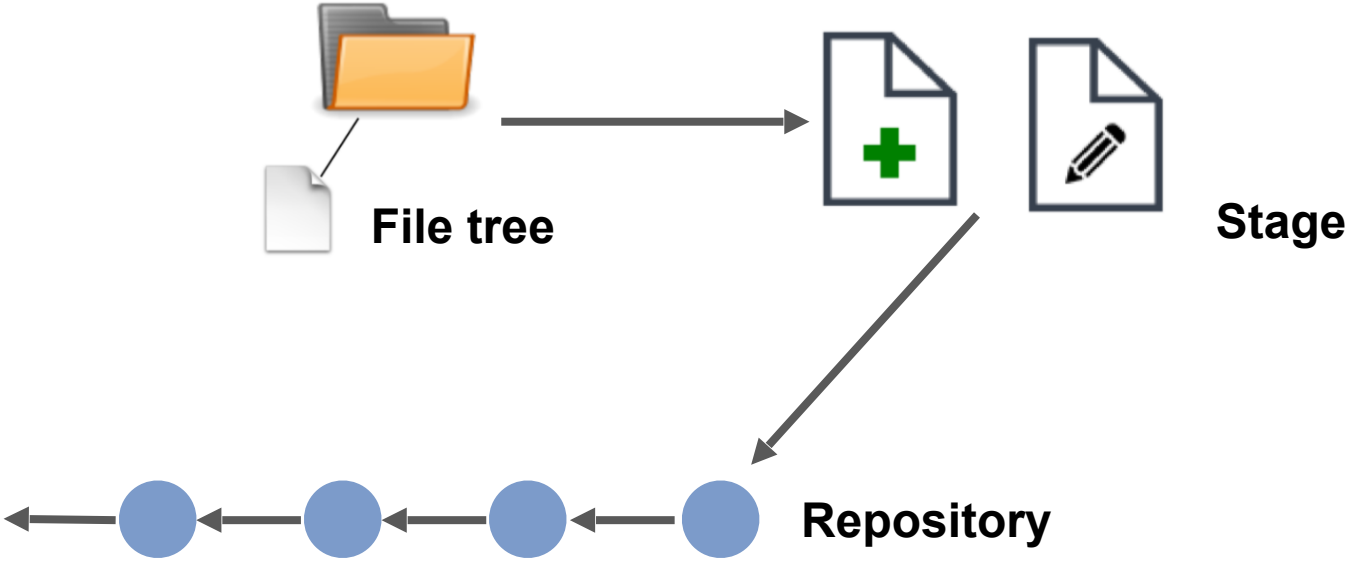
main = main branch (local and remote)

HEAD = current commit (called ***detached HEAD*** when not the tip of a branch)

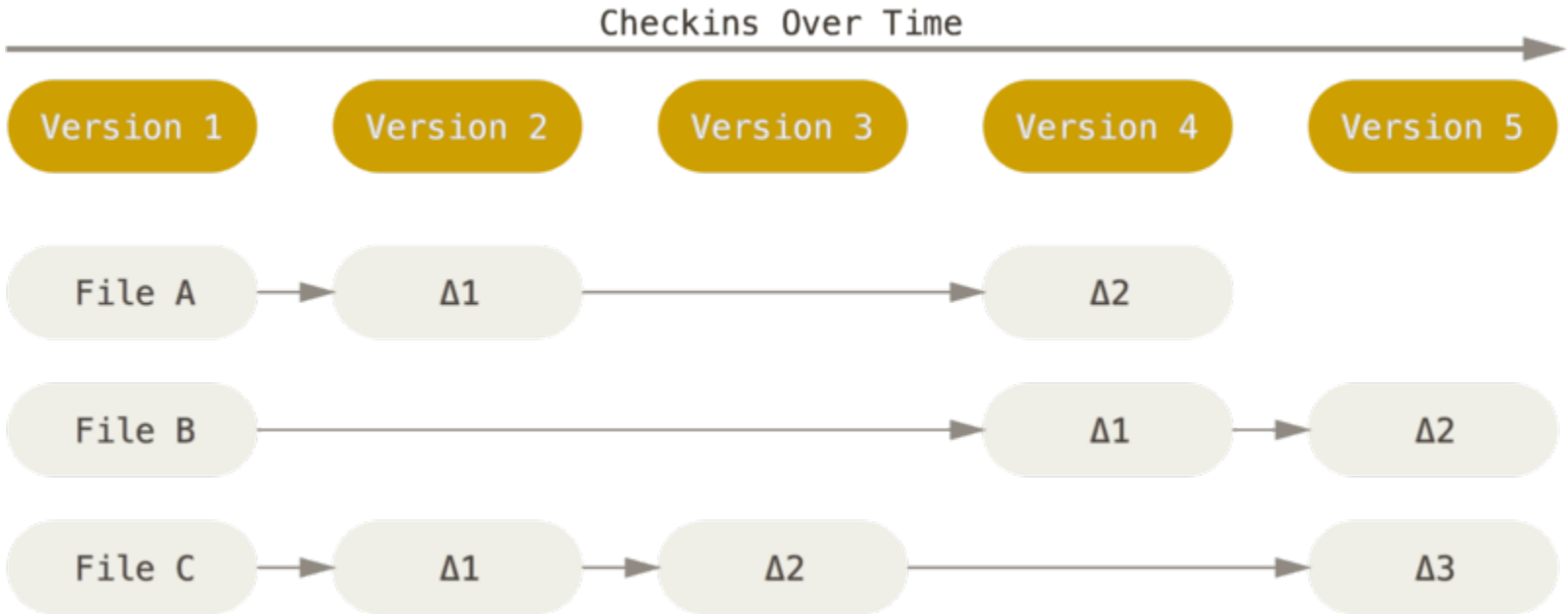
master = previous name for main on GitHub (changed in 2020)

Useful concepts

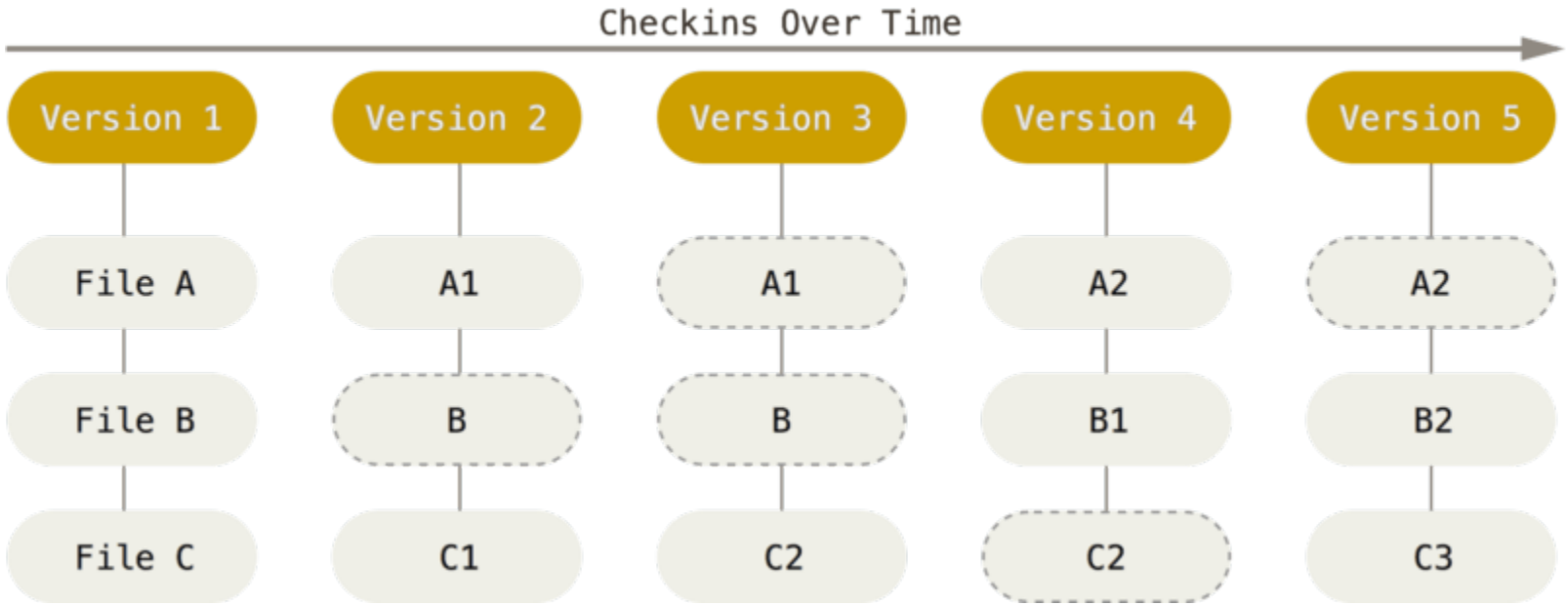
Workflow of a commit



How are differences stored?



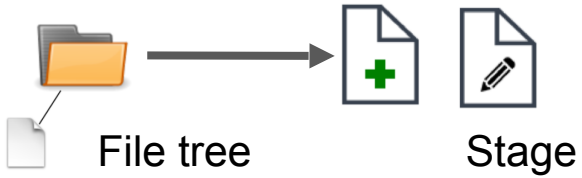
Git stores snapshots of the files



Adding files to stage

Files are added to stage using the command **`add`**

```
git add <filename>
```



- Can stage multiple files or even directories (if they contain files)
- Option `-A` stages all changes if no arguments are given
- More than one filename are allowed including wildcards

To remove or move a file (including staging the change) use:

```
git rm <filename>
git mv <filename>
```

First Commit

When you're happy with your changes and have added all the files to stage the next step is to commit the changes:

```
git commit
git commit -m "Commit Message"
```

Use `-a` to automatically stage files that have been modified or deleted. **New files you have not told Git about are not affected.**

```
git commit -a -m "Commit Message"
```

Good practices

- **Commits** should be:
  - Atomic and self-contained (or WIP), but not excessively pedantic
  - Done often (try not to have gigantic blob commits)

- **Commit messages** should:
  - Always have a short comment (more descriptive than “bugfix”, “save”, etc.)
  - Written to be understandable by someone else >1 year down the line, including you

Checking history

Check the history of a repository:

```
git log
```

A few useful options:

- To see only the commits of a certain author

  ```
  git log --author=<name>
  ```

- To see a very compressed log where each commit is one line

  ```
  git log --oneline
  ```

- To see a more complete history as a graph (useful when having multiple branches):

  ```
  git log --all --decorate --graph --oneline
  ```

Checking differences

Check

- not staged changes since the last commit:

```
git diff
```

- staged changes since the last commit:

```
git diff --cached
```

- differences between two commits:

```
git diff <ID1> <ID2>
```

- differences between second last and last commit:

```
git diff HEAD^ HEAD
```

Exercise 2

Exercise
2

- 1. Add a recipe to your repository by creating an *ingredients.txt* and a *recipes.txt* file (can contain anything you would like) in the repository folder.
- 2. Check the status of the repository. **Stage** the new files (use `git add)` and check the status. **Commit** the staged files and check the status. What did you see change between each step when checking status?
- 3. Create a directory with the recipe name and move your files to it; create a commit with your changes.
- 4. Add or remove one step in *recipes.txt* and check the change before committing (use `git diff)`. Create a commit with your change.

Exercise 2 cont.

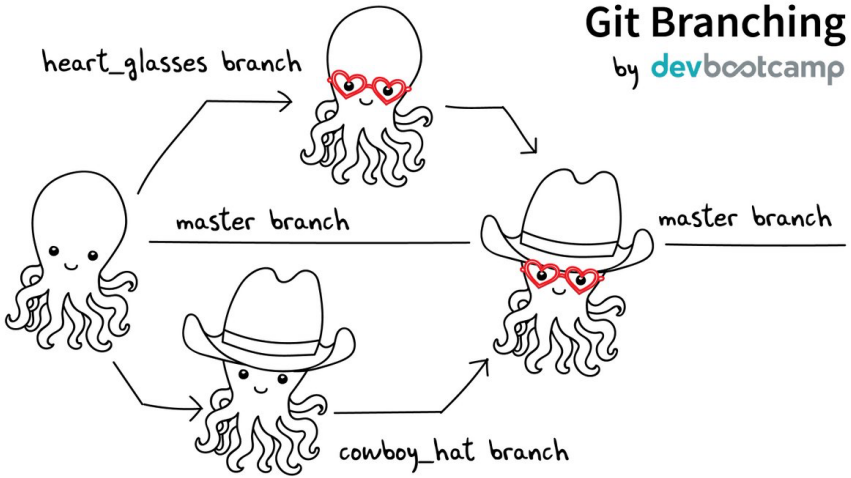
Exercise
2

- 5. Check the commit history. Can you identify the author, ID, time and date of your commits? Try experimenting with the different ways of showing the history.
- 6. Check the difference between different commits. Can you identify the filename(s) and changes?

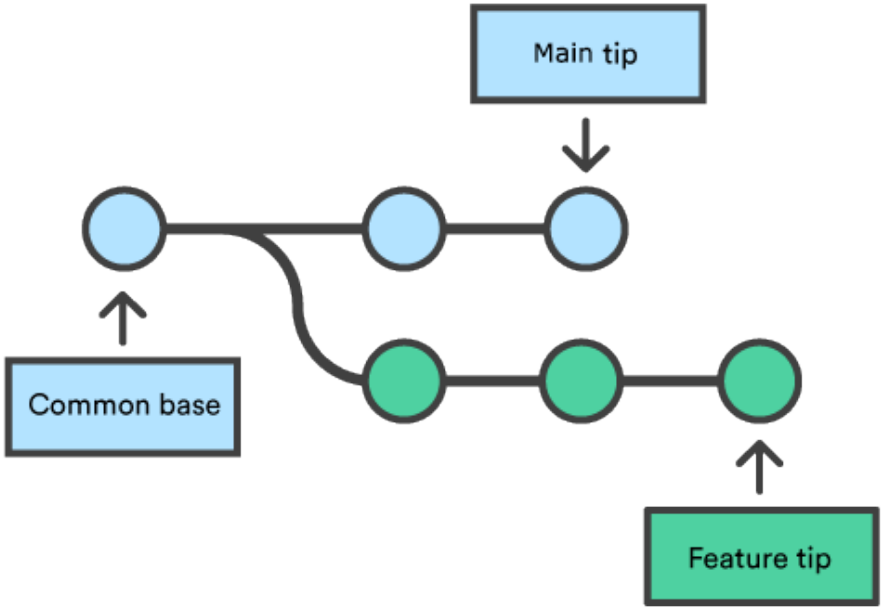
Branching

A **branch** represents an independent line/track of work on your repository.

- Keep a branch that is in working order (*main* branch).
- Isolate experiments from the *main* branch.
- Work on more than one feature at the same time, independently.
- Merge changes we like into the *main* branch.

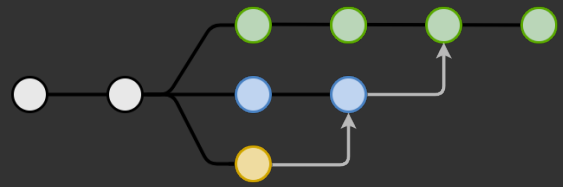


Branching



Branching

- List all the branches in your repository:
  `git branch`
- Create a new branch:
  `git branch <branch>`
- Switch to a branch:
  `git checkout <branch>`
- Safe deletion: prevents deleting if branch has unmerged changes:
  `git branch -d <branch>`
- Force delete:
  `git branch -D <branch>`



Good practices

- **Branches** should:
  - Have descriptive names.
  - Done every time you want to experiment (e.g. add a new feature or fix a bug).
  - Have a single purpose (one feature, one bug, one change).
  - Derived from a sensible commit (often the latest commit on the *main* branch)

Merging

Two main cases for merging

We want to integrate a feature we were working on in a branch, into *main*.

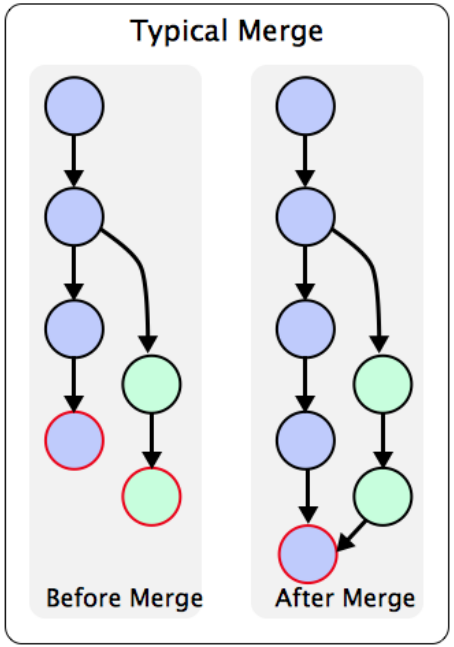
main has diverged enough from our branch that we want to integrate the new commits in it before merging our feature – so we merge *main* into our feature branch.

Merging

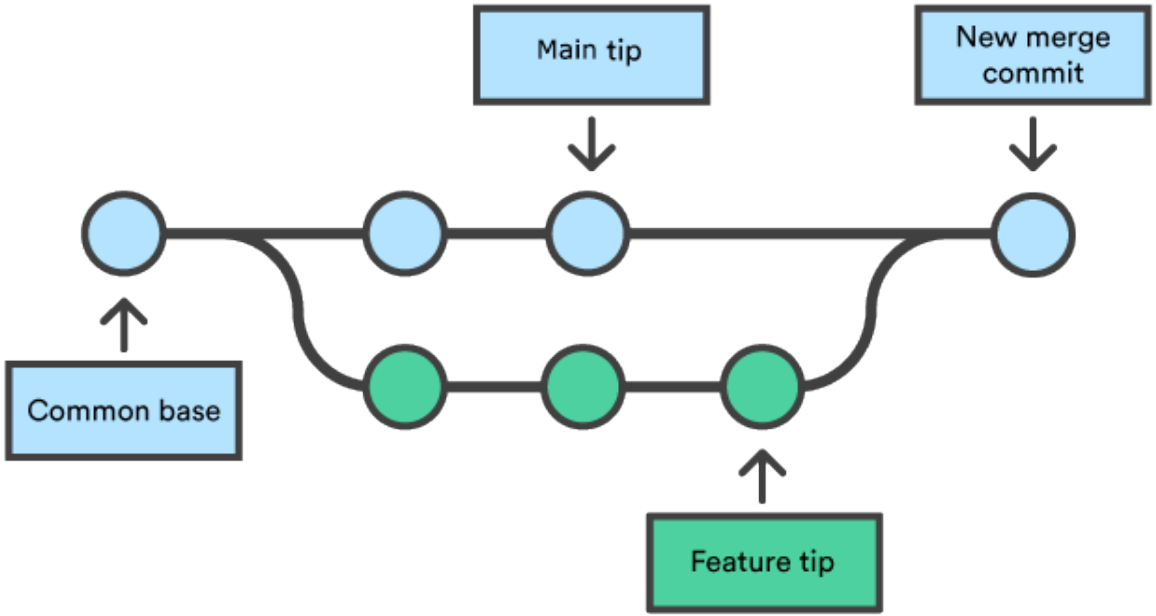
Merging incorporates changes made in another branch **into the current/active branch.**

- **Switch** to the branch you want to change (not the one you want to merge!)
- **Merge** changes made in another branch **into the current/active branch.**

```
git checkout <active branch>
git merge <branch to merge>
```

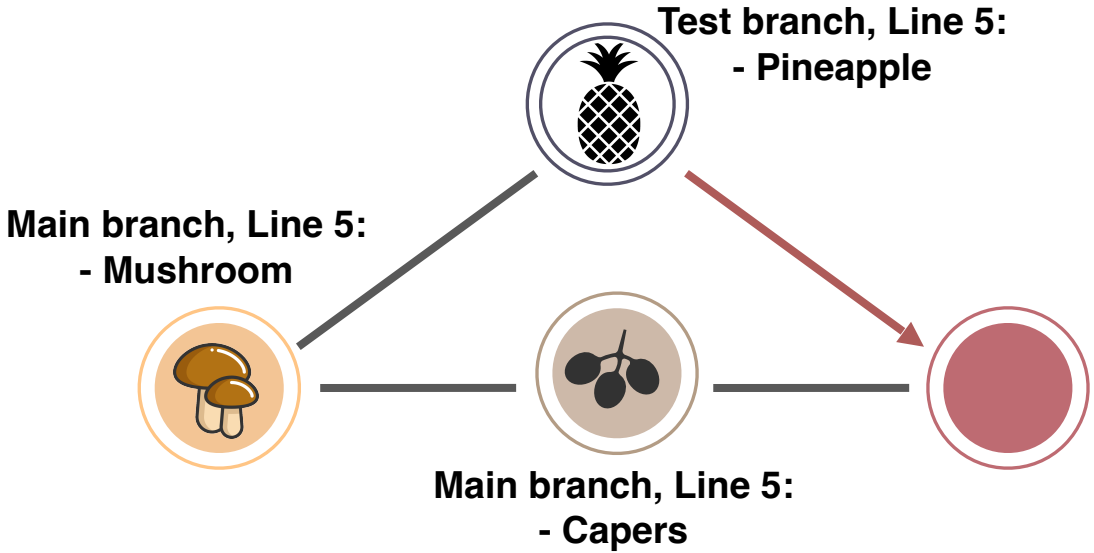


Merging



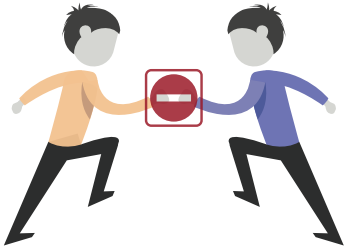
Conflicts

A merge conflict happens when trying to merge branches that have changes **in the same files and in the same lines**



Conflicts

```
(base) Matteos-MacBook-Pro-2:test matteotiberti$ git merge test
Auto-merging recipe.txt
CONFLICT (content): Merge conflict in recipe.txt
Automatic merge failed; fix conflicts and then commit the result.
```



Main branch Test branch

- Check which file(s) have conflicts.
- Modify files by hand to achieve the desired result.
- Add all files to stage once they are ready and commit. This solves the merge in a commit.
- **Remember to remove all conflict markers** and change your code in the final version you want.

```
- tomato
- mozzarella
- flour
- water
<<<<<<< HEAD
- capers
=======
- pineapple
>>>>>>> test
```

Exercise 3

Exercise
3

- 1. Create an ***experimental*** branch and switch to it. Change an ingredient in *ingredients.txt* and store the change as a commit. Try to get a list of the branches in the repository.
- 2. Go back to your ***main*** branch and add a step in *recipes.txt* and store the change as a commit. Check the difference between the two branches (use `git diff`).
- 3. Merge your ***experimental*** branch into your ***main*** branch (it will ask you to write a commit message). Check the history (try using the graph output).

Exercise 3 cont.

Exercise
3

- 4. Switch to your ***experimental*** branch and change the first line in one of your files, and commit.
- 5. Switch to your ***main*** branch and change the same first line in the same file in a different way, and commit.
- 6. Try to merge your ***experimental*** branch into your **main** branch; you should get a conflict. Check the status of the repository.
- 7. Edit the conflicting file and solve the conflict. Stage and commit to fix it. Check the history (try using the graph output).

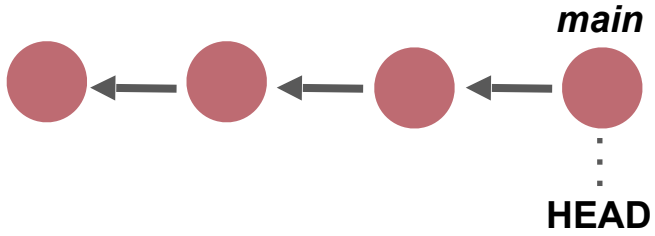
Back to the past

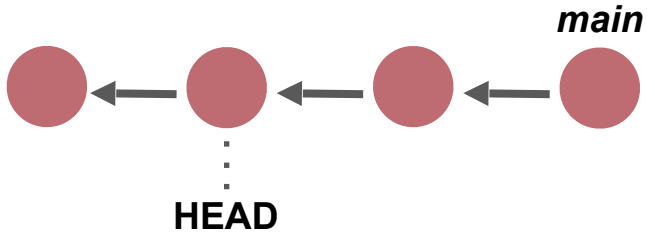
We can move around the repository using the command **`checkout`**

```
git checkout <identifier>
```

Changes the local content of the working directory to that of the reference - which means, moves **`HEAD`** to identifier. Identifier can be:

- A commit **ID**
- A branch **name**





Back to the past

Rewriting history is (usually) a bad idea

Every change you do in git is done with a forward in time perspective - the past should generally not be changed.

- Changes (even to something done a long time ago) should be done in commits added to the HEAD
- To restart from an earlier commit use branching/merging

I’ve botched staging or commit!

How to fix the last commit message:

```
git commit --amend -m “new message”
```

- Do not amend a message of a commit that have already been shared with others (e.g. uploaded to GitHub).

I’ve botched staging or commit!

`reset` is useful to “undo” some of the changes:

`git reset`

- Removes all staged changes.

`git reset <filename>`

- Removes file from staged area.

`git reset --hard <ID>`

- Moves reference to that commit (can be HEAD) and resets the working directory to match that commit. **Destructive operation** - only on local branches that have not been shared with others yet **(do not use on *main*).**

I’ve botched staging or commit!

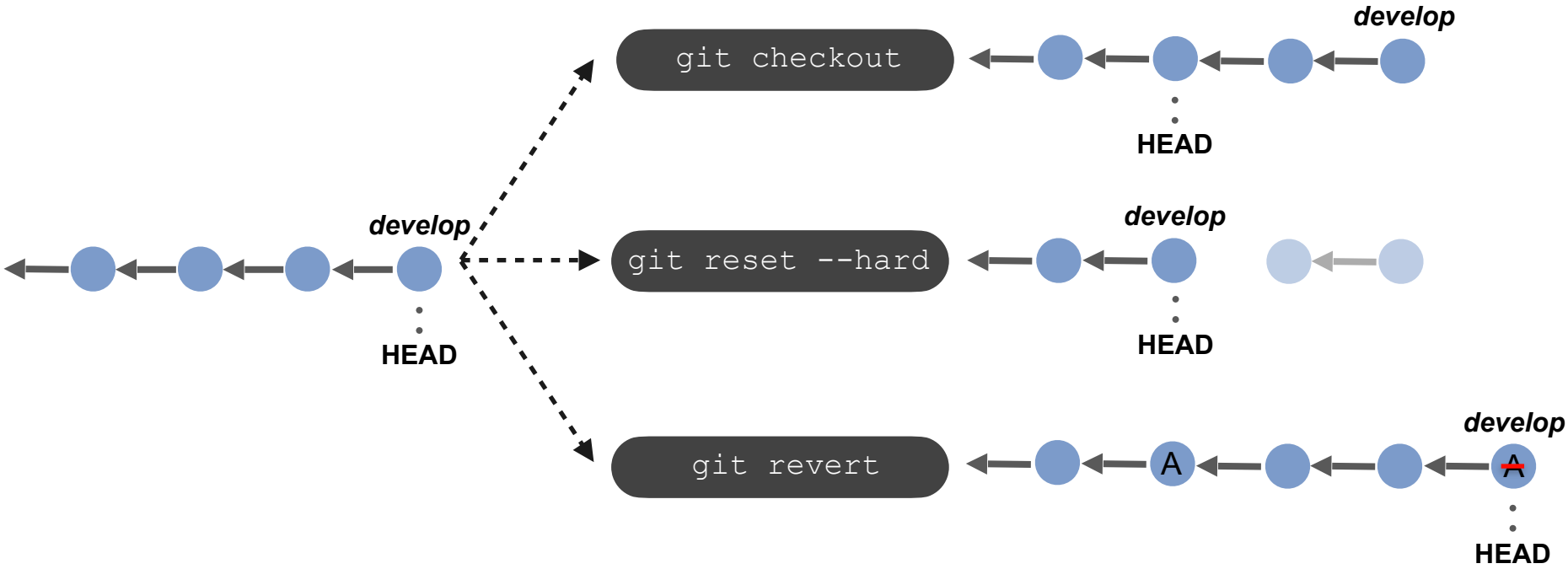
`revert` is used to add commits that “reverse” the changes done by older commits:

```
git revert <ID>
```

```
git revert <ID1>..<ID2>
```

- Non-destructive operation (history is maintained)
- Here “ID” is the commit that you want to revert
- History is polluted with commits that just revert – not the cleanest option
- Better than reset for shared repositories

Git checkout, reset & revert



Exercise 4

Exercise
4

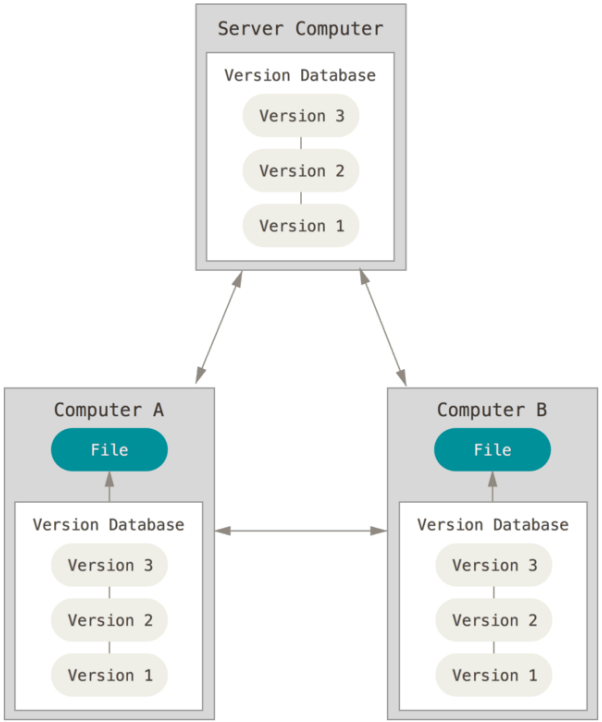
- 1. Checkout an earlier commit on the ***main*** branch. Check the content of the files. Check the history and status of the repository (do not commit). What do you notice? Checkout ***main*** to get back to the tip of the branch (HEAD).
- 2. Switch to the ***experimental*** branch. Add a step in *recipes.txt* and add the changes to stage, but **do not commit**. Check the status. Do a hard reset of **HEAD** and check the files and status. What do you notice?
- 3. Add a new ingredient to *ingredients.txt* and commit the changes. Add a new step to *recipes.txt* and commit the changes. Change the commit message of the last commit and check the history.
- 4. **Revert** the first of the two new commits (it will ask you to write a commit message). Check the files and history. What do you notice? Switch back to ***main***.

Interacting with a remote

A **remote** is a repository that is linked to our local (here our local repository started out as a copy of a remote repository on GitHub).

Changes (commits) in the local repository can be uploaded and merged into the remote repository.

Changes that have happened in the remote repository (e.g. other people's) can be downloaded and merged into the local repository.



Personal Access Token (PAT)

Personal access token to use with the command line.

Token is used to “communicate” between local repository and remote GitHub repository.

Advantage of PAT over password:

- **Unique**: Can be generated per use or per device.
- **Revokable**: Can revoke access to each one at any time.
- **Secure**: Random strings of characters that cannot be attacked by brute force i.e. less hackable than a password.
- **Quantity**: Any number of access tokens can be created, compared to only one password per user.

Interacting with a remote

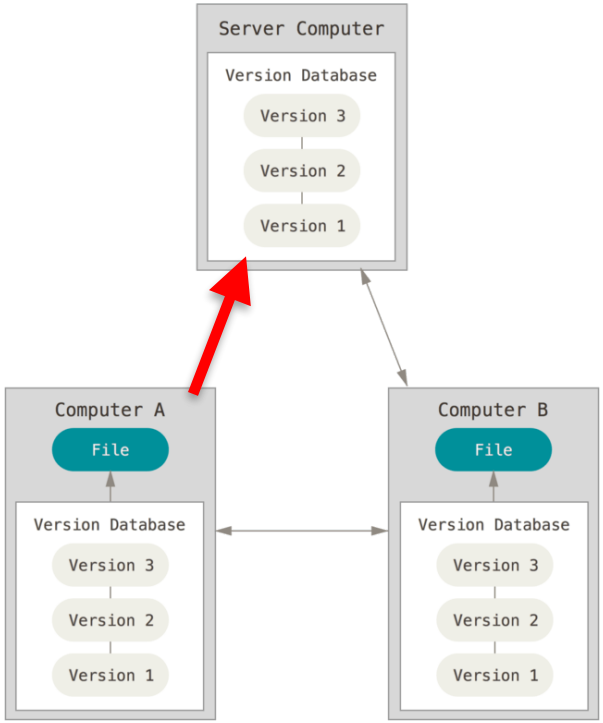
- Use **`push`** to upload local repository content to remote (i.e. export commits to remote branches)

```
git push <remote> <branch>
```

If this is your first time pushing to GitHub you need to login with your username and PAT.

- The default remote branch for the current local branch can be set when pushing (set upstream):

```
git push -u <remote> <branch>
```



Interacting with a remote - Windows users

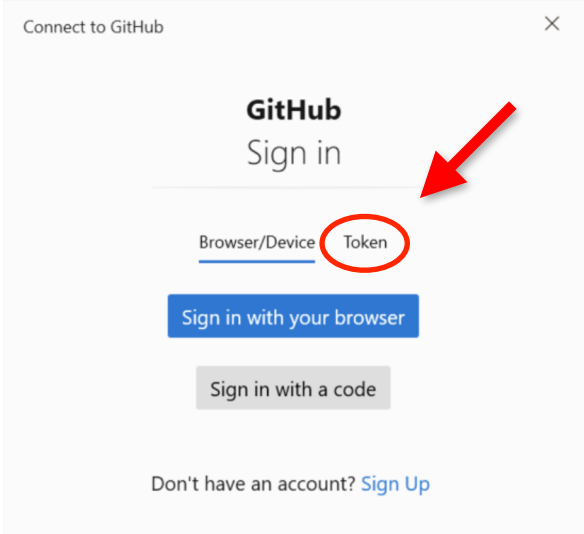
- Use **`push`** to upload local repository content to remote (i.e. export commits to remote branches)

```
git push <remote> <branch>
```

If this is your first time pushing to GitHub you need to login with your username and PAT.

- The default remote branch for the current local branch can be set when pushing (set upstream):

```
git push -u <remote> <branch>
```



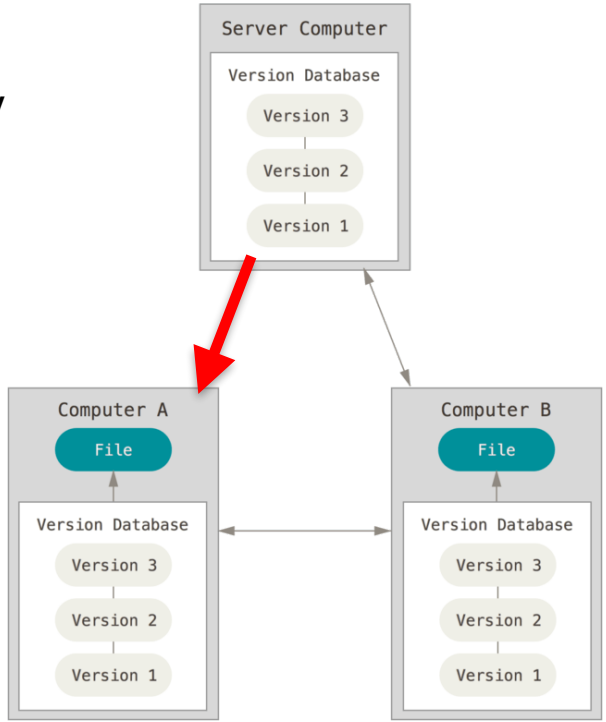
Interacting with a remote

- Use **`pull`** to download and apply the changes that were done in remote to your local repository (i.e. apply commits by other people):

```
git pull <remote> <branch>
```

- Use **`fetch`** to download without merging (remote fetched branches are named as "`<remote>/<branch>`"):

```
git fetch <remote>
```



Exercise 5

Exercise
5

- 1. Go to your GitHub account and set up your personal access token (if you have not already done it). Follow the **Creating a personal access token (classic)** step-by-step on GitHub.com:

  https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

- 2. Copy your access token and note your GitHub username as you will need both the first time you are pushing data to GitHub and for RStudio part. If you have pushed to GitHub before you might not need to use it.

Exercise 5 cont.

Exercise
5

- 3. Push your changes to the remote repository for your ***main*** branch. Visit your remote repository on github.com. Check if your changes are there.
- 4. Push the ***experimental*** branch to your repository. Check the changes on GitHub. What do you notice?
- 5. Merge the ***experimental*** branch into ***main***. Push your changes and check them on GitHub. What have changed?

Exercise 5 cont.

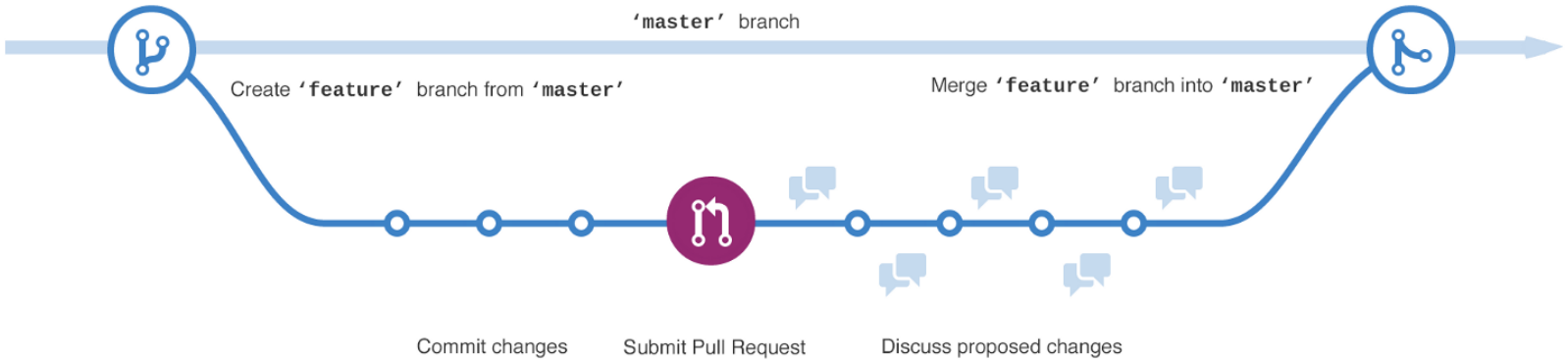
Exercise
5

- 6. Edit the **README.md file on GitHub**. Create a new commit on **main** and try to push it. What happens?
- 7. **Fetch** the changes from the remote repository. Note that the remote version of the ***main*** branch is named ***origin/main***. Look at the difference between the local and remote version of ***main***.
- 8. Merge remote version ***origin/main*** into the local version of ***main*** and push the changes to the remote repository on GitHub. Visit GitHub and check if your changes are there.

A deeper look at GitHub

Contributing and interacting with other users

- Projects on GitHub often follow a pattern of development and communication.
- The *main* branch is considered the “stable” branch and is not changeable directly - only through merges and pull requests.
- Contributions happen in branches that are merged into *main*.
- Releases happens once in a while from the *main* branch.



A deeper look at GitHub

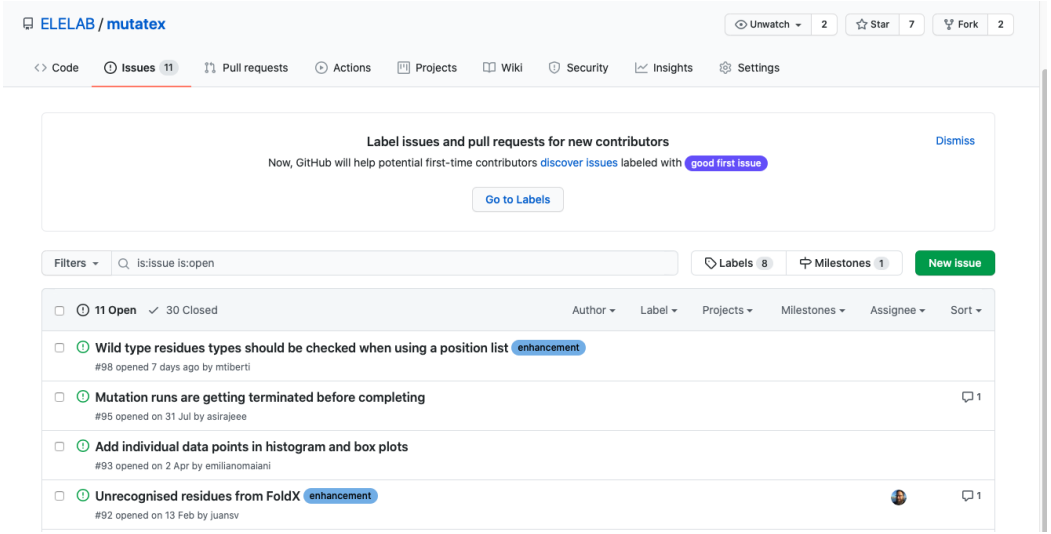
Interacting with other users: README

1. Read the documentation of the project you want to contribute to or use.

A deeper look at GitHub

Interacting with other users: Issues

Check *Issues* if you have a problem or question to see if others have already asked the same. If needed, open a new issue.



A deeper look at GitHub

Interacting with other users: Branches

Create a local copy of the repository.

- **Clone** if you have write access to the repository (direct access as *collaborator* or part of the repo *organization*). Or if you are only planning on using the method and not contributing to its development.
- **Fork** if you have no affiliation to the repository or you want your own copy to experiment with.

Create a **new branch** from the *main* branch. Add changes as commits. Push them so they are present on the remote (GitHub).

A deeper look at GitHub

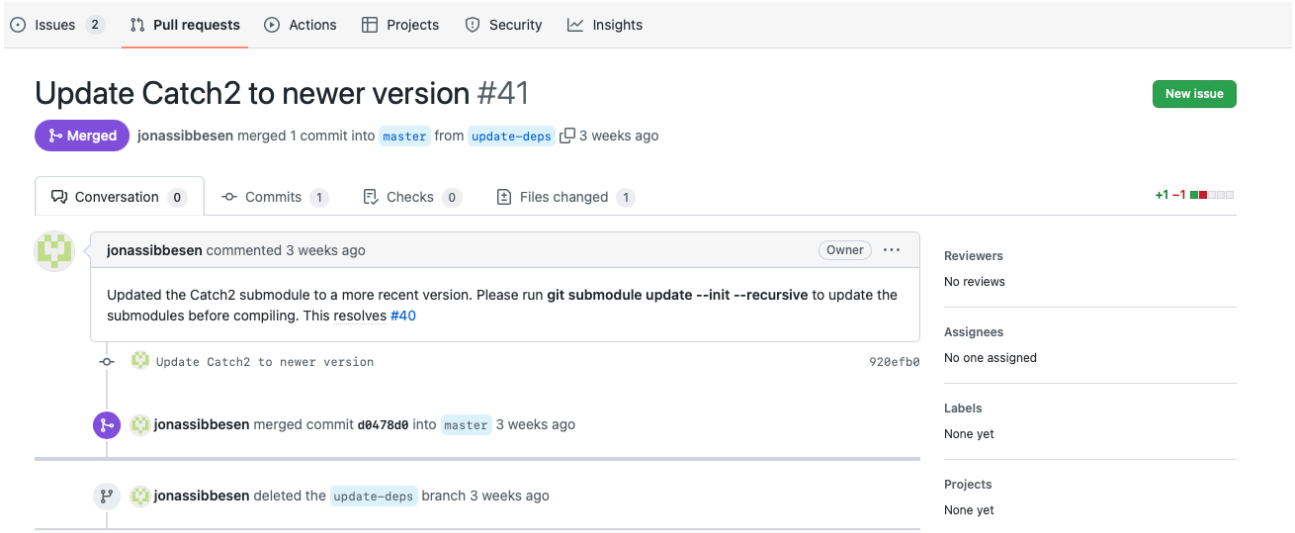
Interacting with other users: Pull requests

Create a *Pull Request (PR)* from the new branch.

- Your code might be subject to code review. If so, the maintainers will decide if/when your code is good enough to be merged.
- To update the PR just push new changes to the branch.
- It is possible to link to issues in the PR. You can use keywords (e.g. “resolves") to automatically close an issue when the PR is merged into *main*: “This PR resolves issue #40”.

A deeper look at GitHub

Interacting with other users: Pull requests



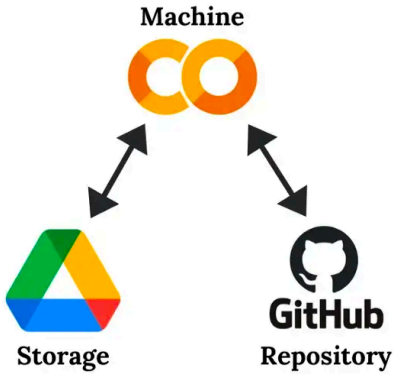
Exercise 6

Exercise
6

- 1. Go to the shared recipes repository in GitHub for which you have become a collaborator.
- 2. Open an **issue** about a recipe you think is missing.
- 3. **Clone** your local copy of this shared repository. Create a new branch. Add your recipe. Commit and push your changes.
- 4. Go back to the repository webpage and open a **Pull request (PR)** with your new branch. Link to your **issue** in the PR message. You might need to fetch and merge ***main*** because of changes made by other people.
- 5. Wait for one of us to approve the PR. Pull from the repo and check our shared recipe book.

Git & GitHub with Code Editors

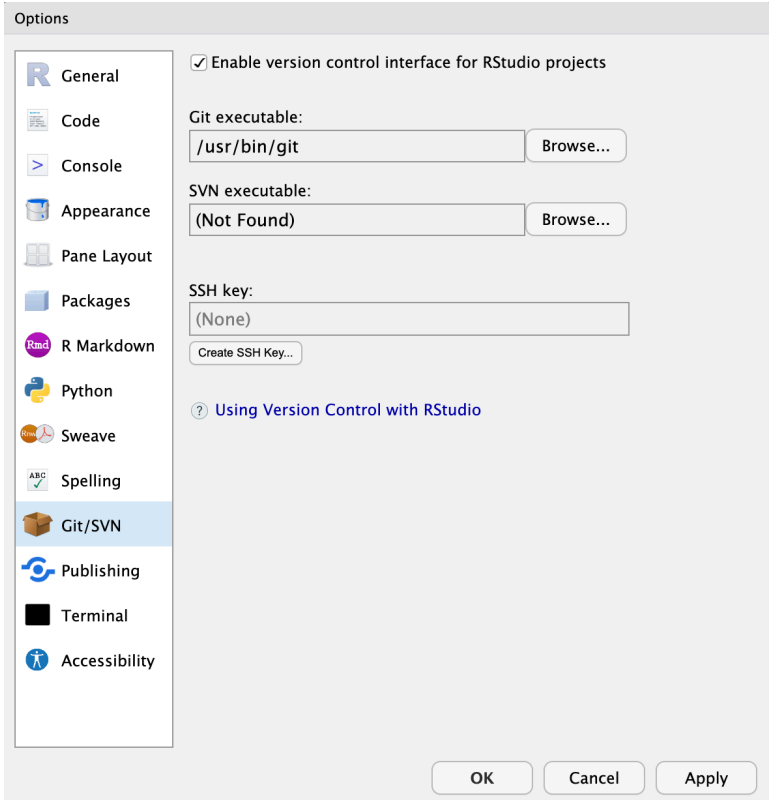
- You can work with **git/GitHub** from within code-editors, including Rstudio (R), PyCharm, Visual Studio etc. (Python).



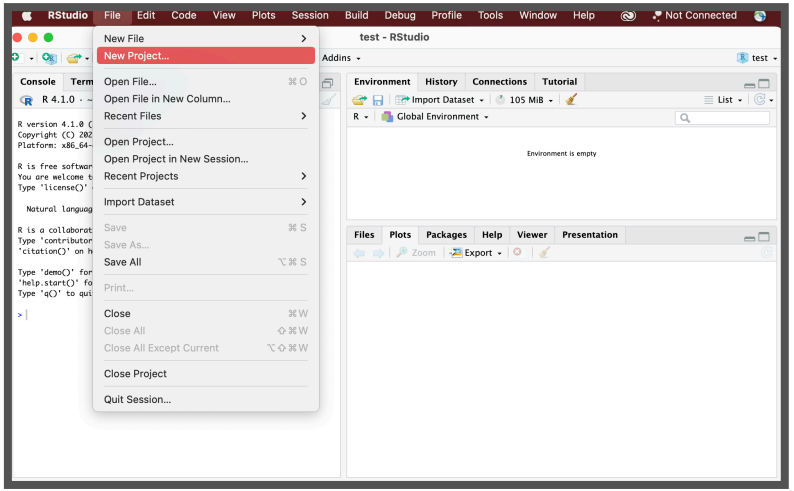
Configure RStudio

- **Make Sure RStudio Knows about Git**

- Go to the Git/SVN section. Then enable the version control interface and make sure RStudio knows where to find git.



Git & RStudio

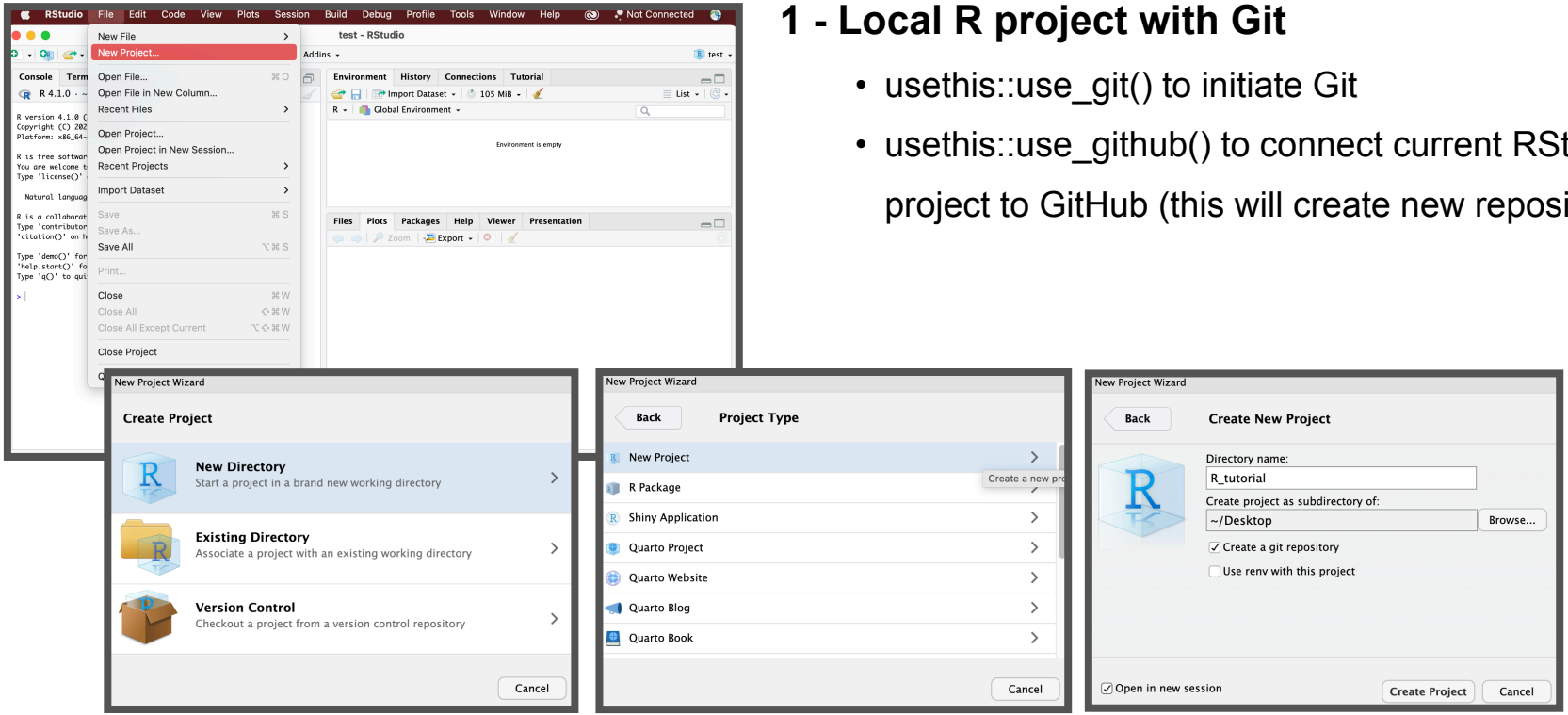


Researchers working with R benefit from using version control.

- From within RStudio Project!!!

Notice! – where the Project will be saved locally.

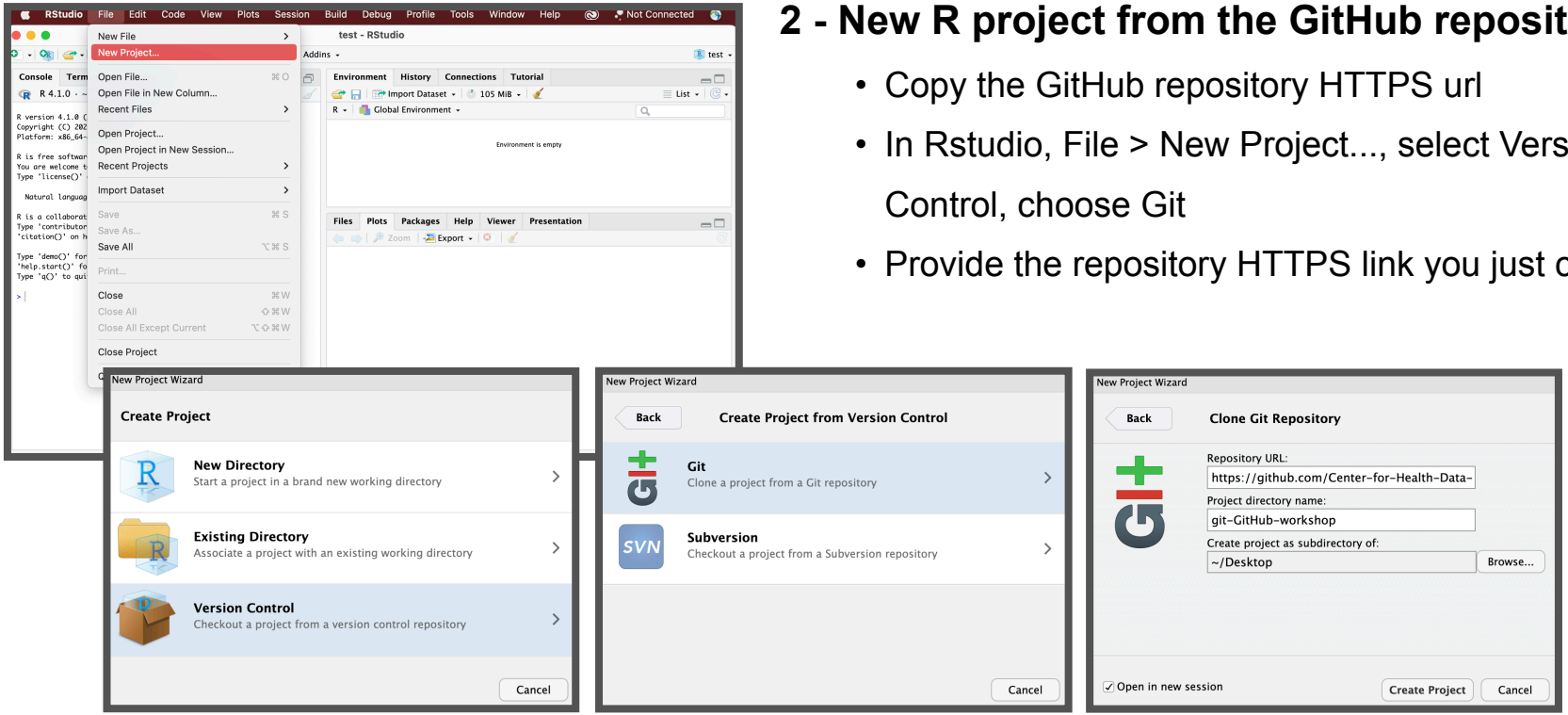
Git & RStudio



1 - Local R project with Git

- usethis::use_git() to initiate Git
- usethis::use_github() to connect current RStudio project to GitHub (this will create new repository)

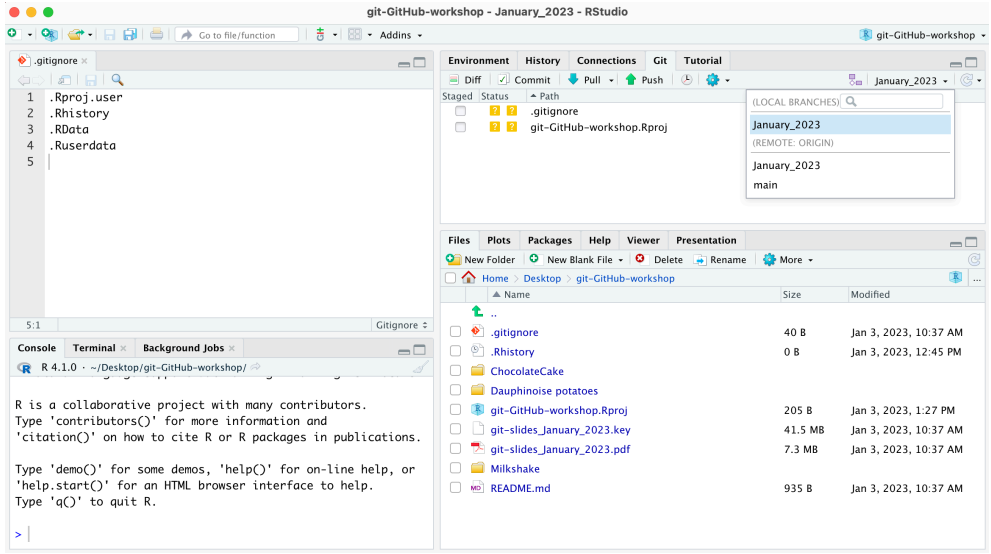
Git & RStudio

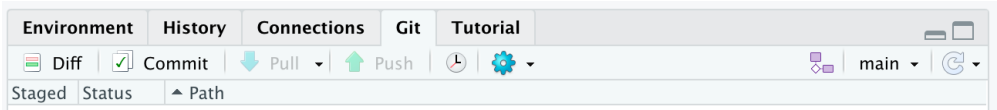


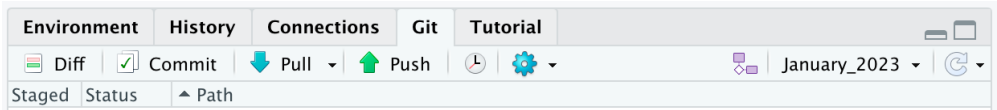
2 - New R project from the GitHub repository

- Copy the GitHub repository HTTPS url
- In Rstudio, File > New Project..., select Version Control, choose Git
- Provide the repository HTTPS link you just copied

Git & RStudio





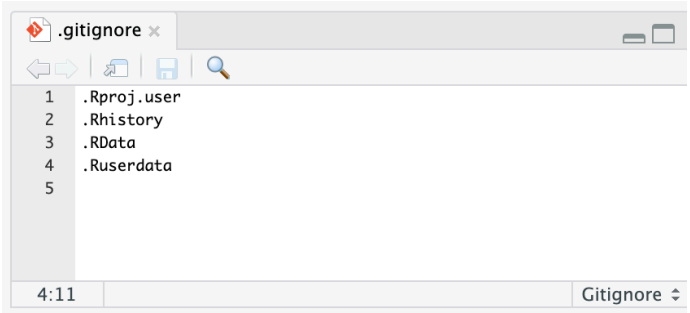


RStudio's Git panel exposes a specific subset of commands from Git.

You can use more complex commands from Shell or terminal

- Make local changes, stage and commit
- Create and switch between branches
- Changes to scripts can be reverted, and pushed to remote (e.g. GitHub)
- Click the green "Push" button to send your local changes to GitHub.

Ignore files from version control



When starting a new project in RStudio, it will always add a file .gitignore if it does not already exists and add some initial files to ignore.

Adding file names or folder names to .gitignore will ignore files that should not be taken into account by Git

Some other examples of files you probably want to ignore:

- Sensitive information (passwords,…)!
- Binary files. Git works very will with text files, but not with binary files such as .Rdata.
- Files > 50MB. Git is specifically made for code (e.g. .R) and does not intend to track all changes in large data files.
- A temp/ folder inside your project with ‘disposable’ content, e.g. draft, test or temp.

Configure Git

- You want git to know who you are so it can associate your changes with you. If you haven't configured Git already you can do it from within Studio.
- Via the usethis package in R:

```
>install.packages("usethis")
>library(usethis)
>usethis::use_git_config(user.name="Jane Doe", user.email="jane@example.org")
```

- edit_git_config() function will open your gitconfig file. Add your name and email and close this.

```
>edit_git_config()
```

Personal Access Token & RStudio

- **How to Connect RStudio Projects with GitHub Repository?**

  Your Personal Access Token will need to be stored in a way RStudio can access it to connect to your GitHub account.

- There are a couple ways to get your PAT into the Git credential store:

  - Call an R function to explicitly store (or update) your credentials.
  - Do something in RStudio that triggers a credential challenge.

Personal Access Token & RStudio

- **"gitcreds" package**
  If you've installed "usethis", you will already have gitcreds installed

- Call gitcreds::gitcreds_set().
  If you don't have a PAT stored already, it will prompt you to enter your PAT. Paste!

  If you already have a stored credential, it will reveals this and will let you inspect it.

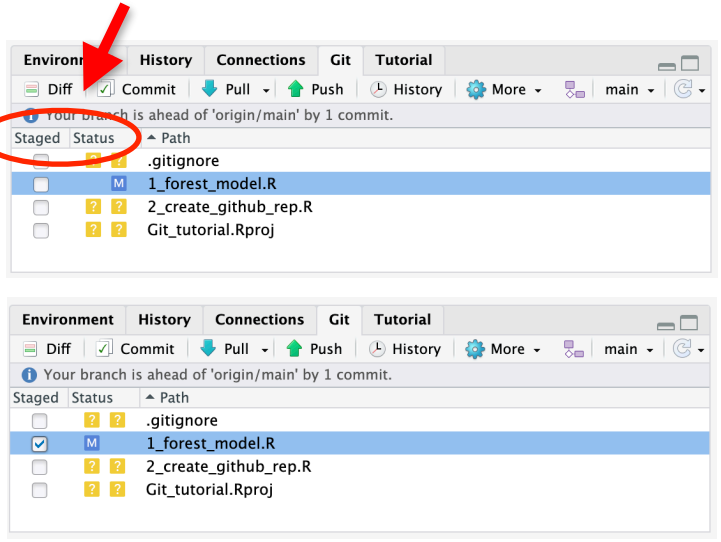
- You'll enter your GitHub username and the Personal Access Token as your password (NOT your GitHub password).

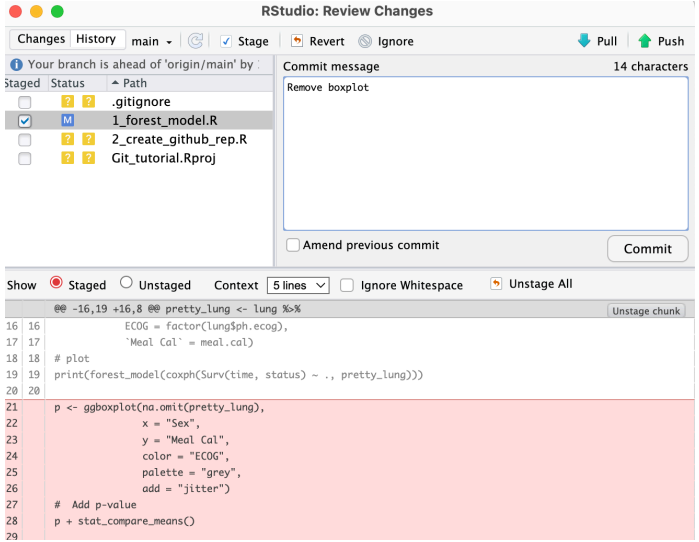
```
>install.packages("gitcreds")
>library(gitcreds)
>gitcreds_get()
#You can check that you've stored a
#credential with gitcreds_get():
```

```
>gitcreds::gitcreds_set()

? Enter password or token:
ghp_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx
-> Adding new credentials...
-> Removing credentials from
cache...
-> Done.
```

Stage and Commit





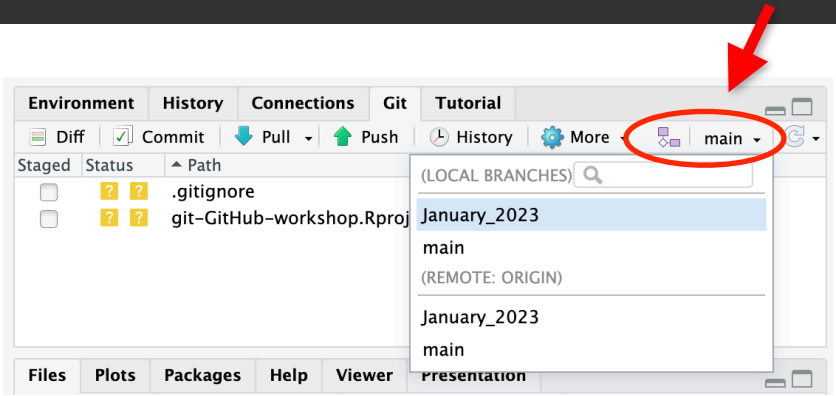
- Added
- Deleted
- Modified
- Renamed
- Untracked

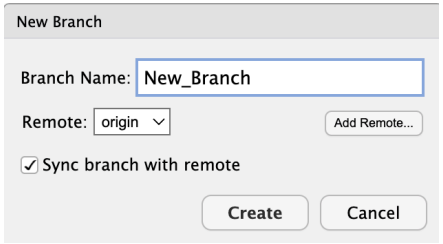
The status of the new files/changes

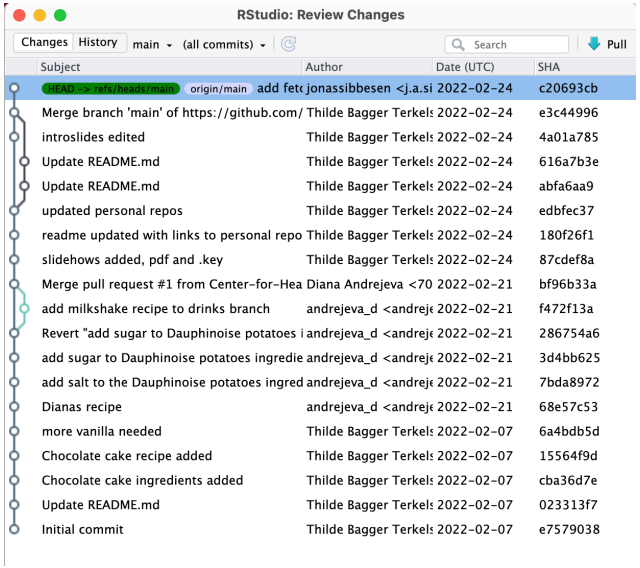
By checking the box and clicking commit, we add the file and are able to commit this with a commit message.

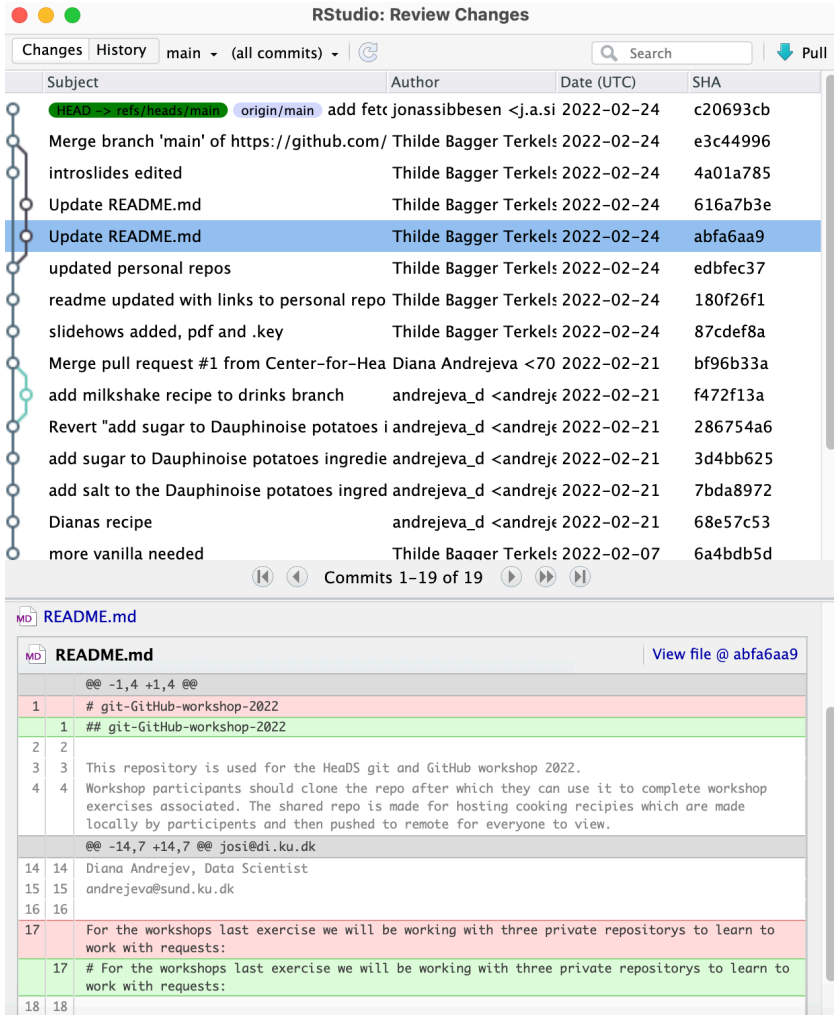
THEN FINALLY WE PUSH TO GITHUB!

Branching

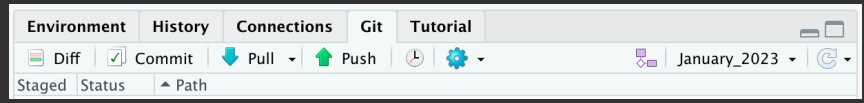

Environment
History
Connections
Git
Tutorial
Diff
Commit
Pull
Push
History
More
main
Staged
Status
Path
.gitignore
git-GitHub-workshop.Rproj
(LOCAL BRANCHES)
January_2023
main
(REMOTE: ORIGIN)
January_2023
main
Files
Plots
Packages
Help
Viewer
Presentation


New Branch
Branch Name:
New_Branch
Remote:
origin
Add Remote...
Sync branch with remote
Create
Cancel


RStudio: Review Changes
Changes
History
main
(all commits)
Search
Pull
Subject
Author
Date (UTC)
SHA
HEAD -> refs/heads/main
origin/main
add fetc
jonassibbesen <j.a.si
2022-02-24
c20693cb
Merge branch 'main' of https://github.com/
Thilde Bagger Terkels
2022-02-24
e3c44996
introslides edited
Thilde Bagger Terkels
2022-02-24
4a01a785
Update README.md
Thilde Bagger Terkels
2022-02-24
616a7b3e
Update README.md
Thilde Bagger Terkels
2022-02-24
abfa6aa9
updated personal repos
Thilde Bagger Terkels
2022-02-24
edbfec37
readme updated with links to personal repo
Thilde Bagger Terkels
2022-02-24
180f26f1
slidehows added, pdf and .key
Thilde Bagger Terkels
2022-02-24
87cdef8a
Merge pull request #1 from Center-for-Hea
Diana Andrejeva <70
2022-02-21
bf96b33a
add milkshake recipe to drinks branch
andrejeva_d <andrej
2022-02-21
f472f13a
Revert "add sugar to Dauphinoise potatoes
andrejeva_d <andrej
2022-02-21
286754a6
add sugar to Dauphinoise potatoes ingredie
andrejeva_d <andrej
2022-02-21
3d4bb625
add salt to the Dauphinoise potatoes ingred
andrejeva_d <andrej
2022-02-21
7bda8972
Dianas recipe
andrejeva_d <andrej
2022-02-21
68e57c53
more vanilla needed
Thilde Bagger Terkels
2022-02-07
6a4bdb5d
Chocolate cake recipe added
Thilde Bagger Terkels
2022-02-07
15564f9d
Chocolate cake ingredients added
Thilde Bagger Terkels
2022-02-07
cba36d7e
Update README.md
Thilde Bagger Terkels
2022-02-07
023313f7
Initial commit
Thilde Bagger Terkels
2022-02-07
e7579038



History



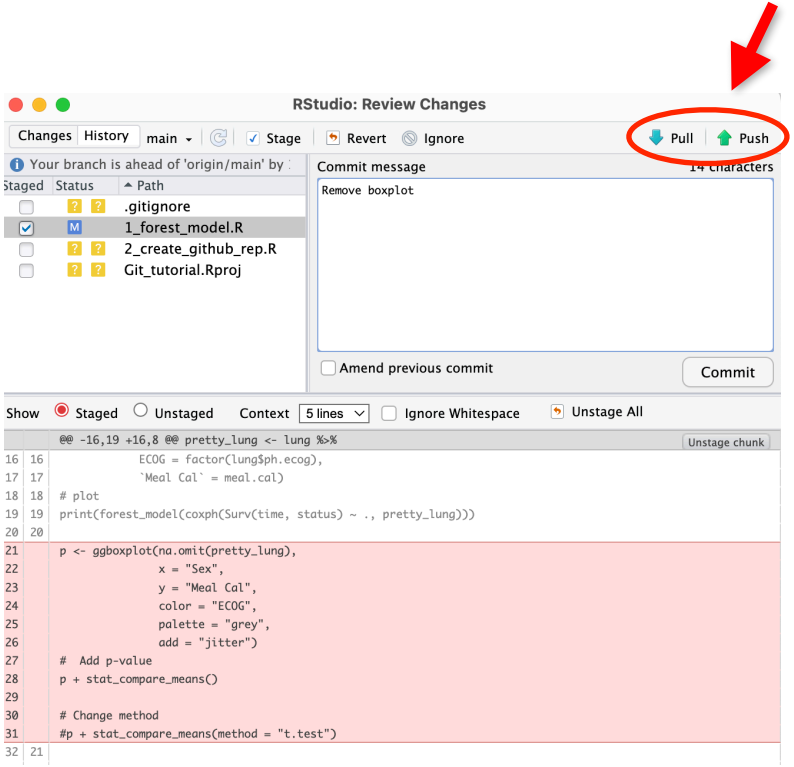
Notice the presence of History twice:

Git history vs R command history

You can check the history online, using terminal as well as **in Rstudio**.

Click on history in the **Git panel**

Interact with GitHub

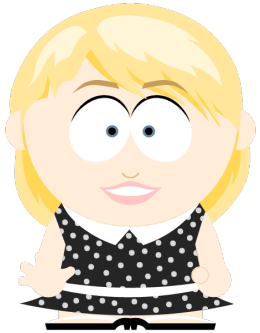


Pull updates from GitHub

- Using the down arrow button, RStudio goes to the GitHub repo, grabs the most recent code and brings it into your local editor.
- Pulling regularly is extremely important if you're collaborating.

Push changes to GitHub

- Rstudio gives you a warning about the status of your local commits versus those stored on GitHub.
- After committing, we now have a push button (the up arrow) on RStudio.



Thank you for today

See you at HeaDS