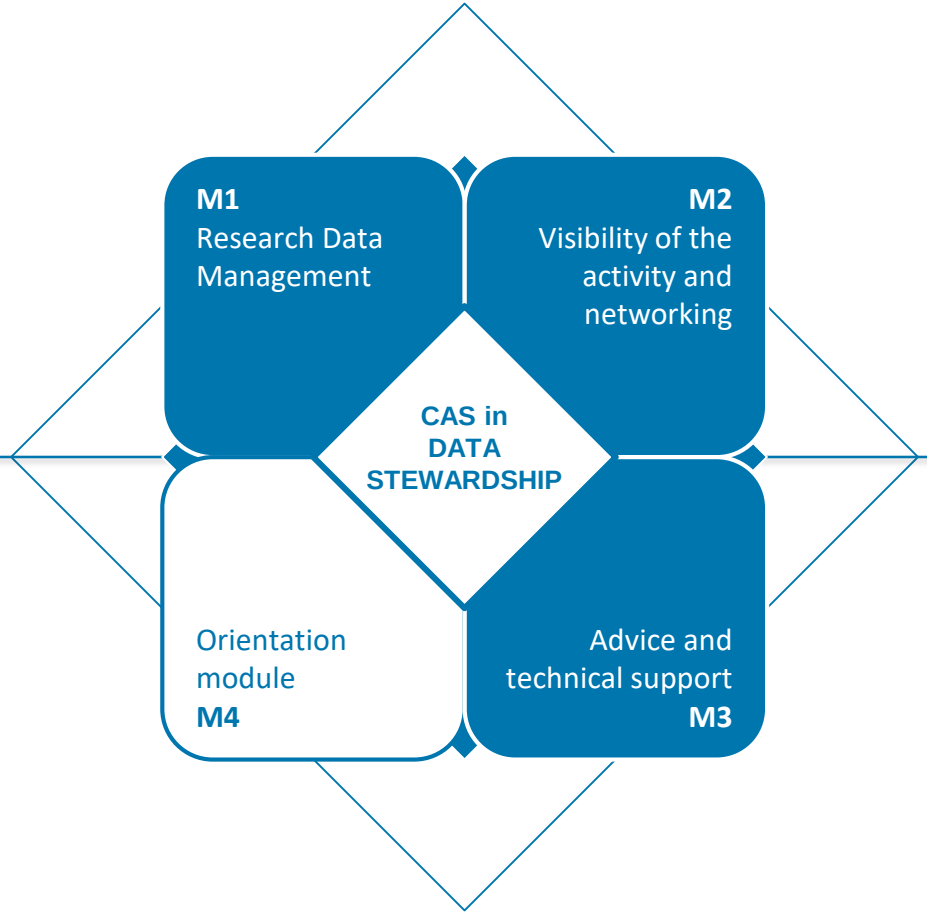


DAY TO DAY MANAGEMENT
INTRODUCTION TO VERSION CONTROL WITH GIT

Robin Engler

SIB
Swiss Institute of Bioinformatics

ABOUT THIS PRESENTATION

This course is part of the CAS in Data Stewardship. You will find more info at <add link>

Author	Dr Robin Engler
Provider	SIB Swiss Institute of Bioinformatics
Title	Introduction to version control with Git
Education level	Graduates
Language	English
License	CC-BY 4.0
Estimate total time	12h (with interactions)
DOI	<doi + link>
Version	v20240110
How to attribute	ENGLER, Robin (SIB Swiss Institute of Bioinformatics), 2025. Introduction to version control with Git. CAS Data Stewardship UNIL. 20 March 2025. DOI: <doi + link>

YOUR HOST(S)

Dr Robin Engler

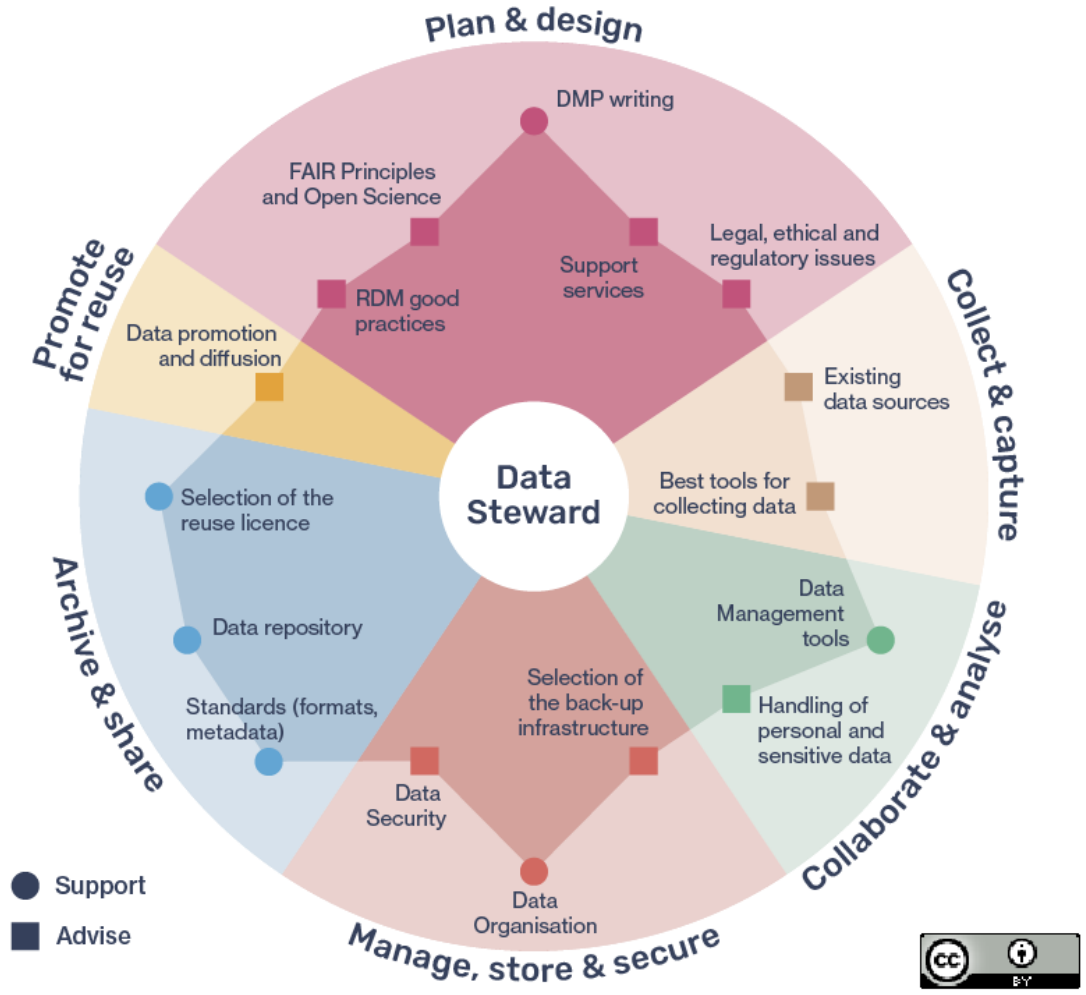
Senior Computational Biologist, SIB/Vital-IT group

Swiss Institute of
Bioinformatics

PEDAGOGICAL OBJECTIVES

In this lesson we will work on these competencies:

- PS1 - Understand the functioning and challenges of the research process in an institutional context, including policies, organization and strategy.
- PS1 - Be familiar with research-related professions and their interaction within an institution.
- PS8 - Understand the challenges of Open Science and Open Research Data and what they mean for research data management



SwissDS-ENV project: Data Stewards tasks and contributions at each step of the data lifecycle (lifecycle adapted from FORS). University of Lausanne, 2023.

Introduction to Git: course outline

- **Git basics**: creating a Git repository, making commits, and displaying a repository's history.
- **Git concepts**: commits, the HEAD pointer and the Git index.
- **Git branches**: introduction to branched workflows and collaborative workflow examples.
- **Branch management:** create and merge branches.
- **Working with remotes**: collaborating with Git and GitHub.

Course resources

Course home page: Slides, exercises, exercise solutions, command summary (cheat sheet), setting-up your environment guide.

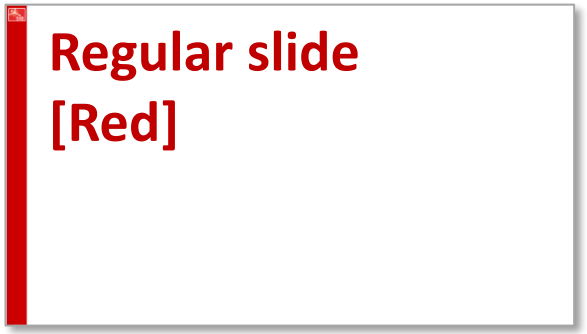
https://github.com/sib-swiss/CAS-UNIL-intro-to-git

Shared online doc: Register for the collaborative exercises. Link in the CAS Data Steward Moodle.

Questions: feel free to interrupt at anytime to ask questions.

Course slides

- 3 categories of slides:



Slide covered in detail during the course.

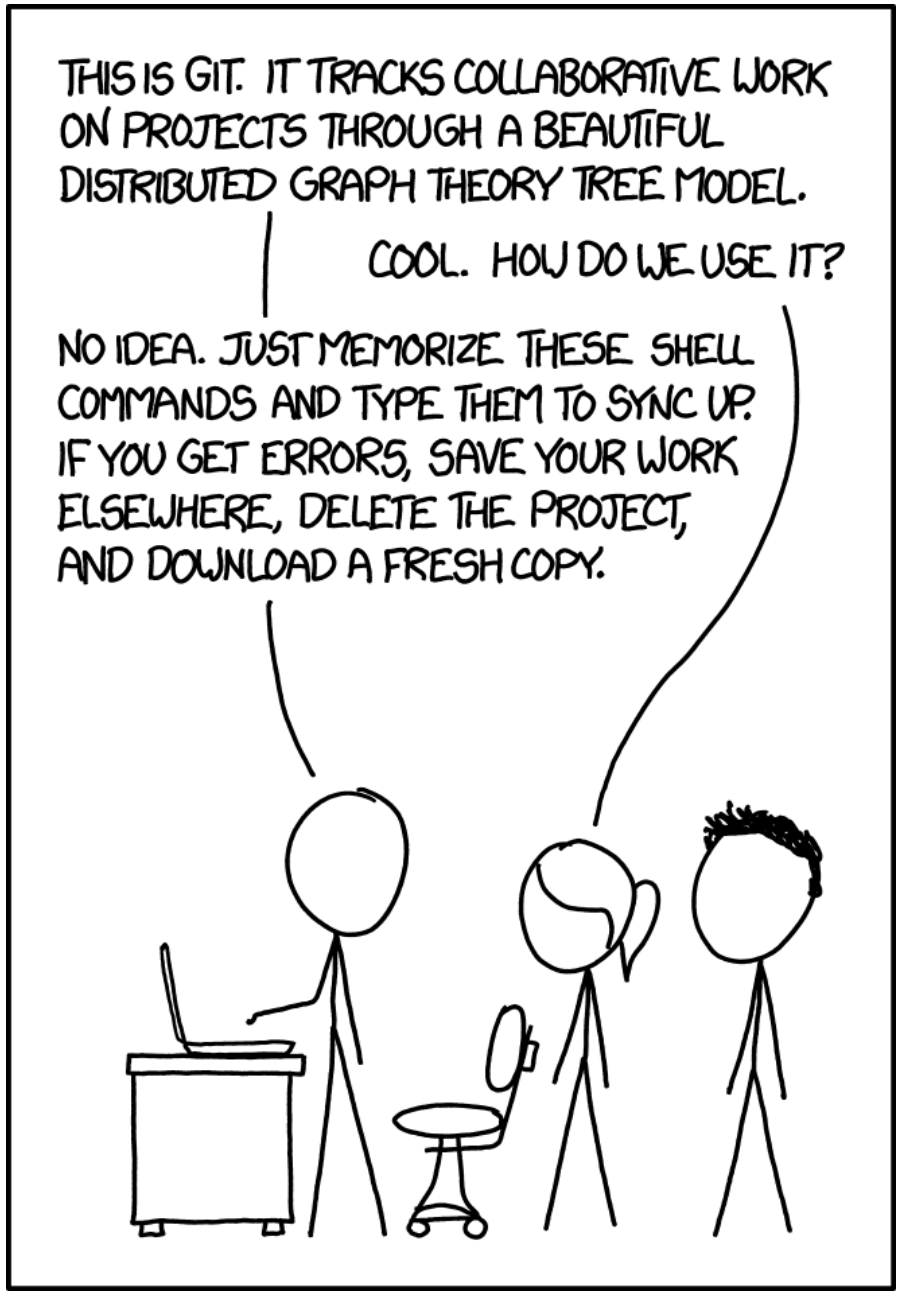
Some slides are specific to **GitHub** or **GitLab**.

Material available for your interest, to read on your own.
Not formally covered in the course.
We are of course happy to discuss it with you if you have questions.

Material we assume you know.
Covered quickly during the course.

Learning objectives

- Learn the concepts behind Git.
- Understand when and why to use each command.
- Collaborative workflows using GitHub/GitLab.



source: https://xkcd.com/1597

Command line vs. graphical interface (GUI)

- This course focuses exclusively on **Git concepts** and **command line** usage.
- Many GUI (graphical user interface) software are available for Git, often integrated with code or text editors (e.g. Rstudio, Visual Studio Code, PyCharm, …).

  It will be easy for you to start using them (if you wish to) once you know the command line usage and the concepts of Git.

Part I

Git basics

Working principle, definitions, and making your first commit

Why use version control ?

Version control systems (VCS), sometimes also referred to as *source control/code managers* (SCM), are software designed to:

- **Keep a record of changes** made to (mostly) text-based content by recording specific states of a repository's content.
- **Associate metadata to changes**, such as author, date, description, tags (e.g. version).
- **Share files** among several people and **enable collaborative work**.
- **Backup strategy**:
  - Repositories under VCS can typically be mirrored to more than one location.
  - The database allows to retrieve older versions of a document: if you delete something and end-up regretting it, the VCS can restore past content for you.
- In the case of Git, entire ecosystems such as GitHub or GitLab have emerged to offer **additional functionality**:
  - **Distribute** software and documentation.
  - **Run automated pipelines** for code testing and deployment (CI/CD).
  - **Project management features** (e.g. issue tracking, continuous integration).

Git working **principles** and **definitions**

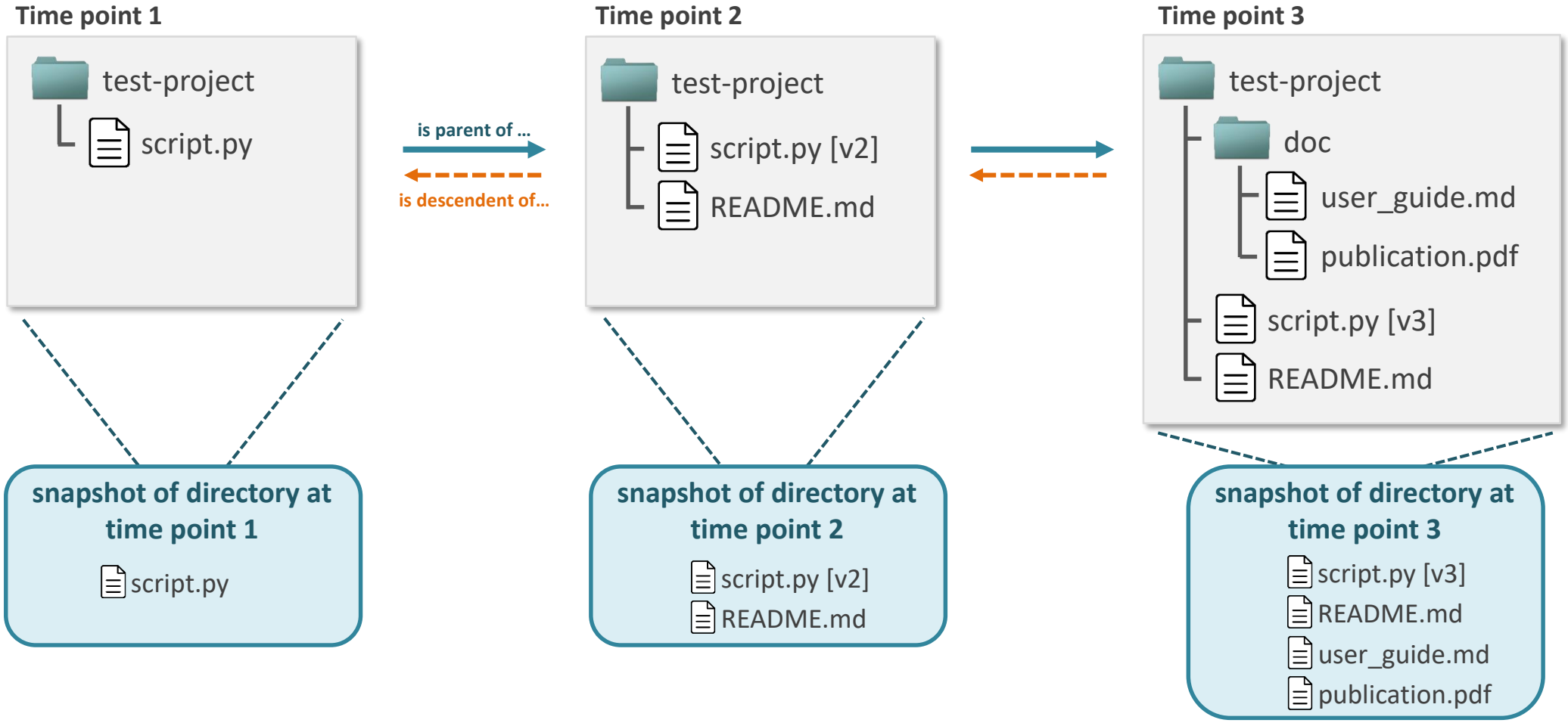
Basic principle of Git

Git can track any types of files (text or binary), but is optimized to work with not-too-large text files.

Our objective: **record the changes made to the content of a directory** on our **local machine**.

How we proceed:

- Take **snapshots** (current content of files) **at *user defined* time points – they are *not* taken automatically**.
- Keep track of **the order of snapshots** (the relation between them) so their history can be recreated.
- Associate **metadata** with each snapshot: who made it, when, description, …

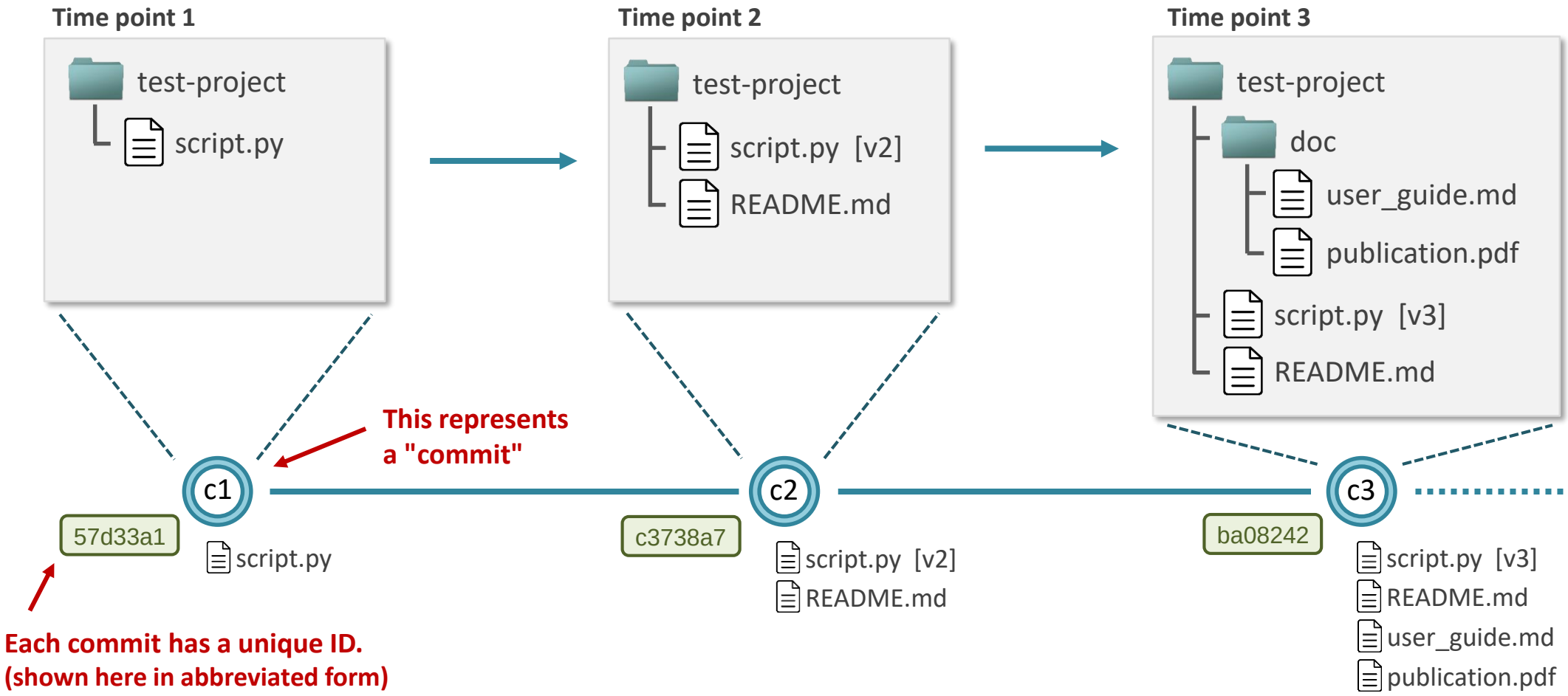


Definition: snapshots are called “commits”

* As will be seen in later slides, **this statement is not 100% correct**, but is a good-enough approximation for now.

- **Commit = snapshot + metadata** (author, time, commit message, parent commit ID, etc. …).
- Create a new commit = record a new state of the directory’s content *.
- Each commit has a unique **ID number** / **hash** (40 hexadecimal characters):

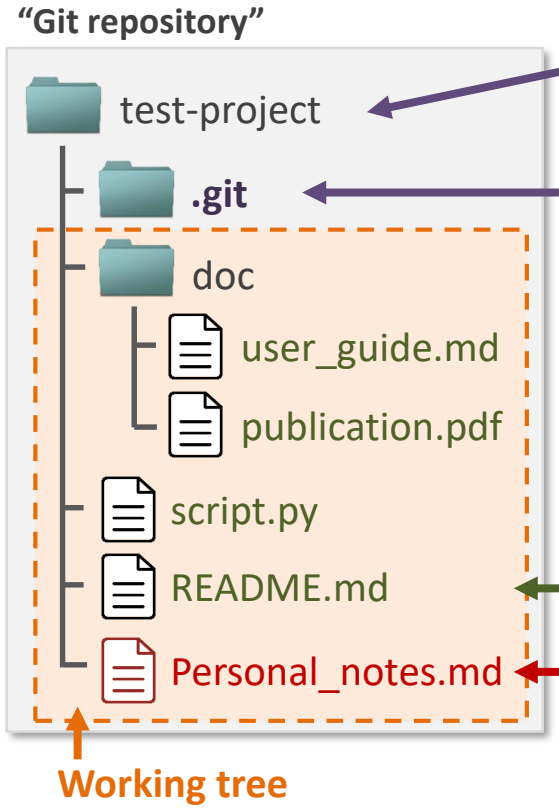
commit ID: `3c1bb0cd5d67dddc02fae50bf56d3a3a4cbc7204`



Definition: commits are stored in a repository (or "repo")

- **Git repository/repo**: version history of files in a directory under Git version control, along with metadata, and configurations necessary for version tracking and collaboration.
  - Technically, a Git repository is only the hidden ".git" directory (see figure below), but often the term is also used to refer to the entire directory under Git control ("test_project" in the example below).
  - Not all files in a directory under Git control have to be tracked: there can be a mix of **tracked** and **untracked** files.

- **Working Tree**: current content (on your computer) of a directory under Git control.
  - More exhaustive definition: state of the project files corresponding to the branch/commit that is currently checked out, augmented with uncommitted changes made to files, as well as untracked files.



Directory under Git version control
- Contains the actual Git repository, and the currently tracked and untracked files.

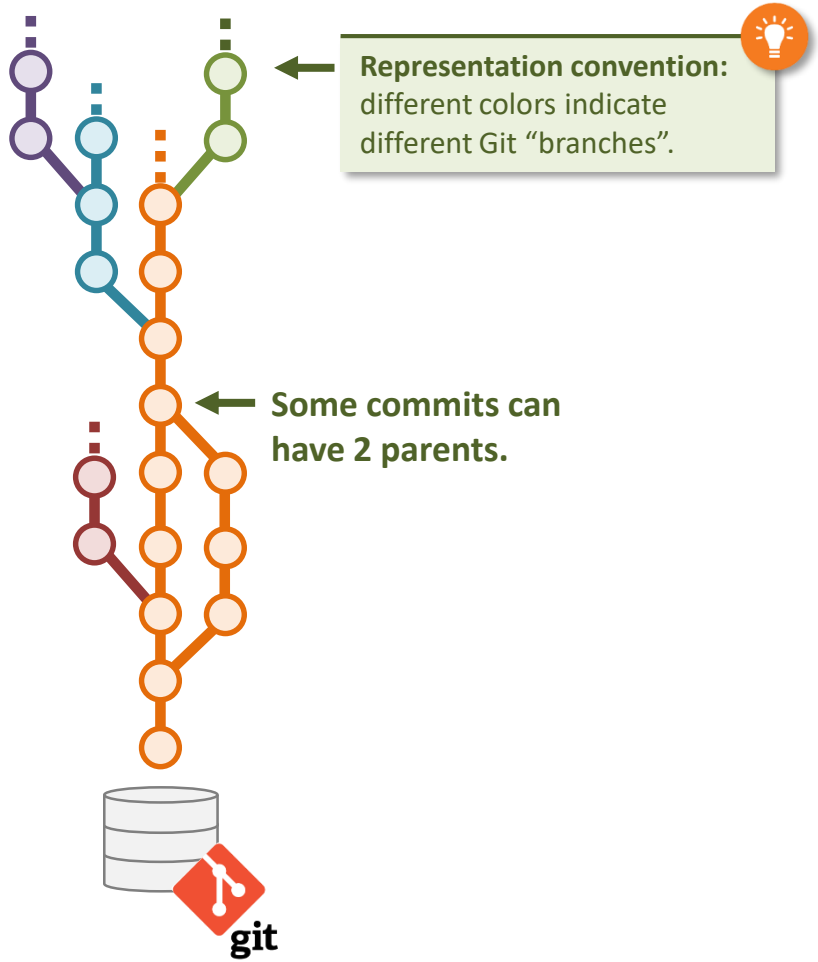
Git repository (*sensu stricto*)
- Contains the version history of all tracked files, along with metadata and configuration necessary to provide the functionalities of Git.
- Can re-create the version of all tracked files, at any commit.
- Each directory under Git control has its own repository.

Tracked file: file under Git version control.

Untracked file: file present in the git repo directory, but *not* under version control.

Definition: branches

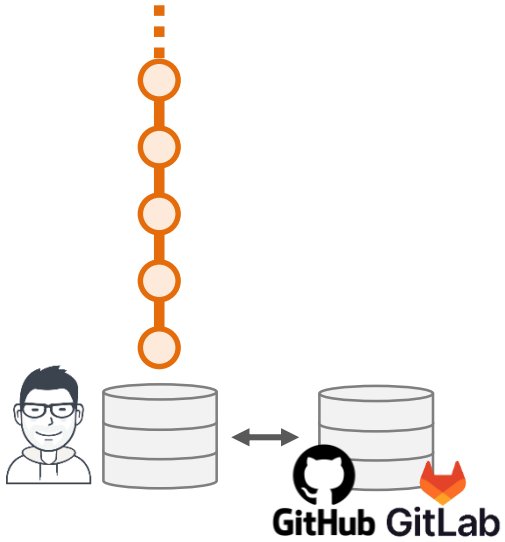
- **Repository history**: history of commits (chronology of commits).
- **Branch**: refers to a "**line of development**" within the commit history.
  - Technically a branch is simply a reference to a commit.



Git use cases: examples

Exercise 1

Single repo, single branch

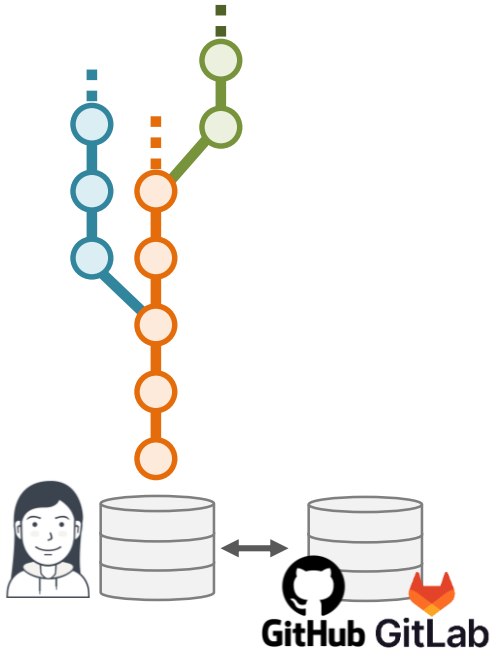


Use case

- Keep a documented log of your work.
- Go back and compare to earlier versions.
- See what changed since last commit.
- Backup (if a paired with a remote).
- Distribute your code (if paired with remote)

Exercises 2 and 3

Single repo, branched workflow (multiple development lines)

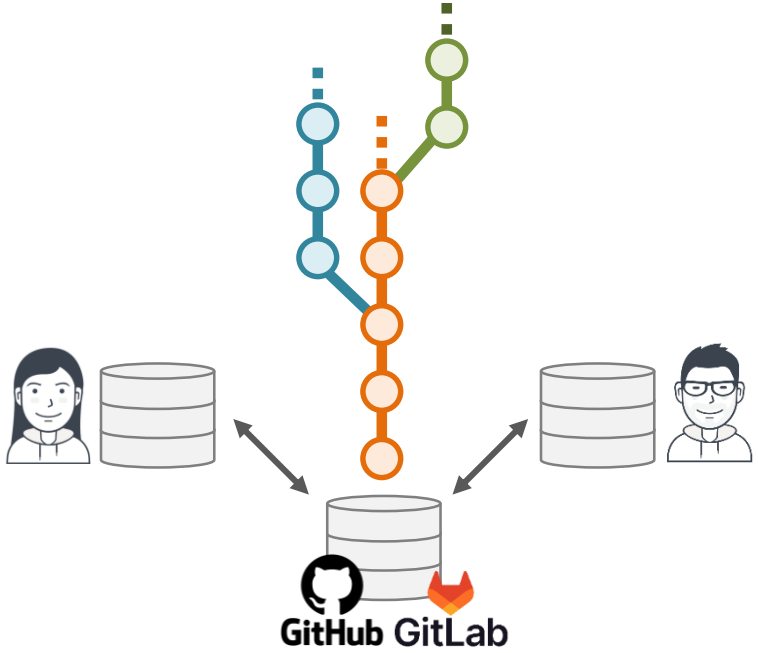


Use case

- Service in production with continued development in parallel (e.g. adding new feature).
- + all benefits of the previous use case.

Exercise 4

Collaboration with distributed and central repos.



Use case

- Collaborate with others (distributed development).
- + all benefits of the previous use case.

The local repo must be associated to a remote repository to provide backup functionality (and new commits must be regularly pushed). Highly recommended.

Each user has a full copy of the data*.

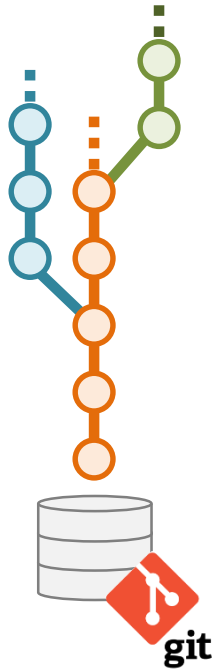
*** Provided they regularly sync their local repo.**

Local vs. Remote repository

- When creating a new Git repository on your computer, **everything is only local**.

- To get a copy of your repository online, you must take the active steps of:
  - **Creating** a new repository on a hosting service (e.g. GitHub, GitLab, Bitbucket, …).
  - **Associate** the online repository with your local repo.
  - **Push** your local content to the remote.

- By design, Git **does not automatically synchronize** a local and remote repo. Download/upload of data must be triggered by the user.



Supplementary material…

Git **configuration**

```
git config
```

Configuring Git

The minimum configuration is setting a **user name** and **email**. These will be used as default author for each commit.

- Setting user name and email:

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

> The `--global` option/flag tells Git to store the setting at the "global" (user wide) scope. Global settings apply to all Git repos on your machine.
>
> Without the `--global` option, the setting will only apply to the current Git repo.
>
> Global settings are stored in the following config file:
>
> - Linux: `/home/$USER/.gitconfig`
> - Mac OS: `/Users/<user name>/.gitconfig`
> - Windows: `C:/Users/<user name>/.gitconfig`

- Config values can be retrieved by using the `--get` option.

```
# Set user name and email at the global (user-wide) scope.
❯ git config --global user.name "Alice"
❯ git config --global user.email alice@redqueen.org

# Retrieve setting values.
❯ git config --get user.name
Alice
❯ git config --get user.email
alice@redqueen.org
```

Configuring Git: changing the default text editor

On most systems, the default editor that Git uses is "**vim**".
However, this can be configured with the following `git config` command:

```
git config --global core.editor <editor cmd>
```

- Display the current default editor used by Git:

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

- **Example**: changing the default editor to "nano" (another command line editor).

```
# Change the default editor to "nano".
$ git config --global core.editor nano

# Change the default editor to "VS Code".
$ git config --global core.editor "code --wait"

# Display the current default editor.
$ git config --global --get core.editor
nano
```

Supplementary material...

Creating a **new repo**

```
git init
```

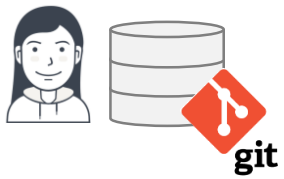
```
git clone
```

Obtaining a new Git repo: two main ways…

Turn a local directory into a Git repo
(start from scratch)

Enter the directory to version-control, then run:

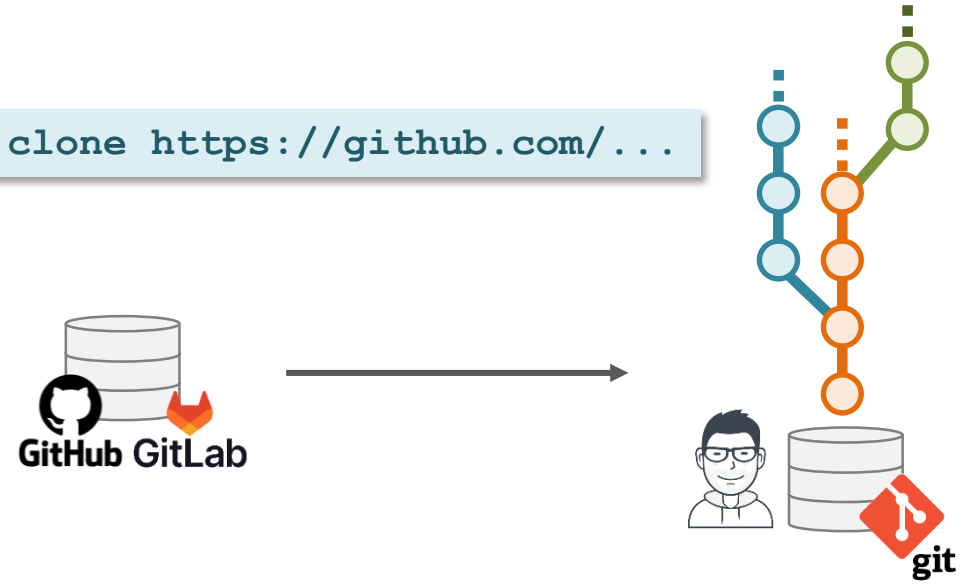
```
git init
```



- A new, empty, Git repository is created in the current directory.
- Files present in the directory can now be version-controlled. However, version-control of files is *not automatic* – more on that later.
- At this point there is *no* online remote associated with the new repo. Everything is only local.

Clone a repo from an online source
(start from an existing repo)

```
git clone https://github.com/...
```



- The entire content of the online Git repository is “cloned” (i.e. downloaded) to the local machine.
- The online repo is automatically linked (i.e. setup as a “remote”) for the local repo: we can push commits with no additional setup.
- Starting a new project on GitHub/GitLab and cloning it can also be a way to create a new empty local repository and immediately link it to a remote.

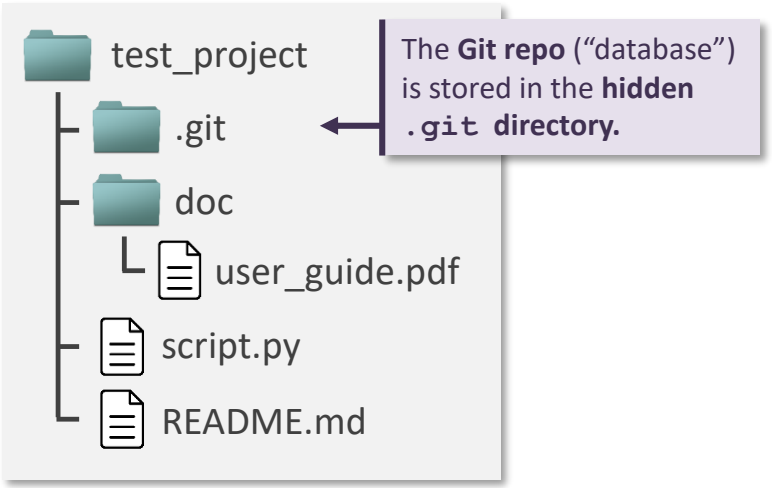
Cloning and working with remotes will be presented in more details later in these slides.

Creating a new Git repository (from scratch)

`git init` **Initializes a Git repository** in the current working directory, turning it into a version controlled directory.

Example

```
# Enter directory to version control and initialize a new repo.
❯ cd /home/alice/test_project
❯ git init
Initialized empty Git repository in /home/alice/test_project/.git/

# List the content of our directory: we now see a new .git directory.
❯ ls -a
./ ../ .git/ doc/ src/ README.md
```



- You must be located **at the root of the directory to version control** before typing `git init`
- `git init` creates a hidden `.git` directory at the root of the directory.
- **Everything** is stored in this single `.git` directory:
  - Complete version history of all tracked files.
  - All other data associated to the Git repository (e.g. branches, tags).
  - The content of `.git` can re-create the exact state of all your files at any versioned time - e.g. if you delete a file accidentally or want to go back to an earlier version.

Never delete the `.git` directory
Unless you intend to start again your repo from scratch, and accept to lose all its history.

State of the working directory (here just after `git init`)

Useful commands to assess the current status of a Git repo:

- **Show status of files** in project directory (working tree).

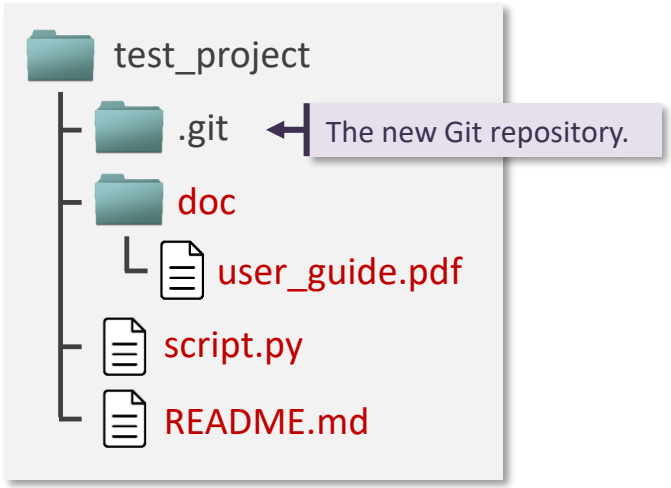
```
git status
```

```
> git status
On branch main
No commits yet
Untracked files:
      doc/
      README.md
      script.py
```

"**main**" is the default branch name.

Red = untracked files

How it looks in the file system



- **Commit history:** show log of commits, i.e. the history of the repo.

```
git log
```

```
> git status
fatal: your current branch 'main'
does not have any commits yet
```

Since we just created a new repo there are no commits yet, which is why we get this error.

Summary: when creating a new Git repo…

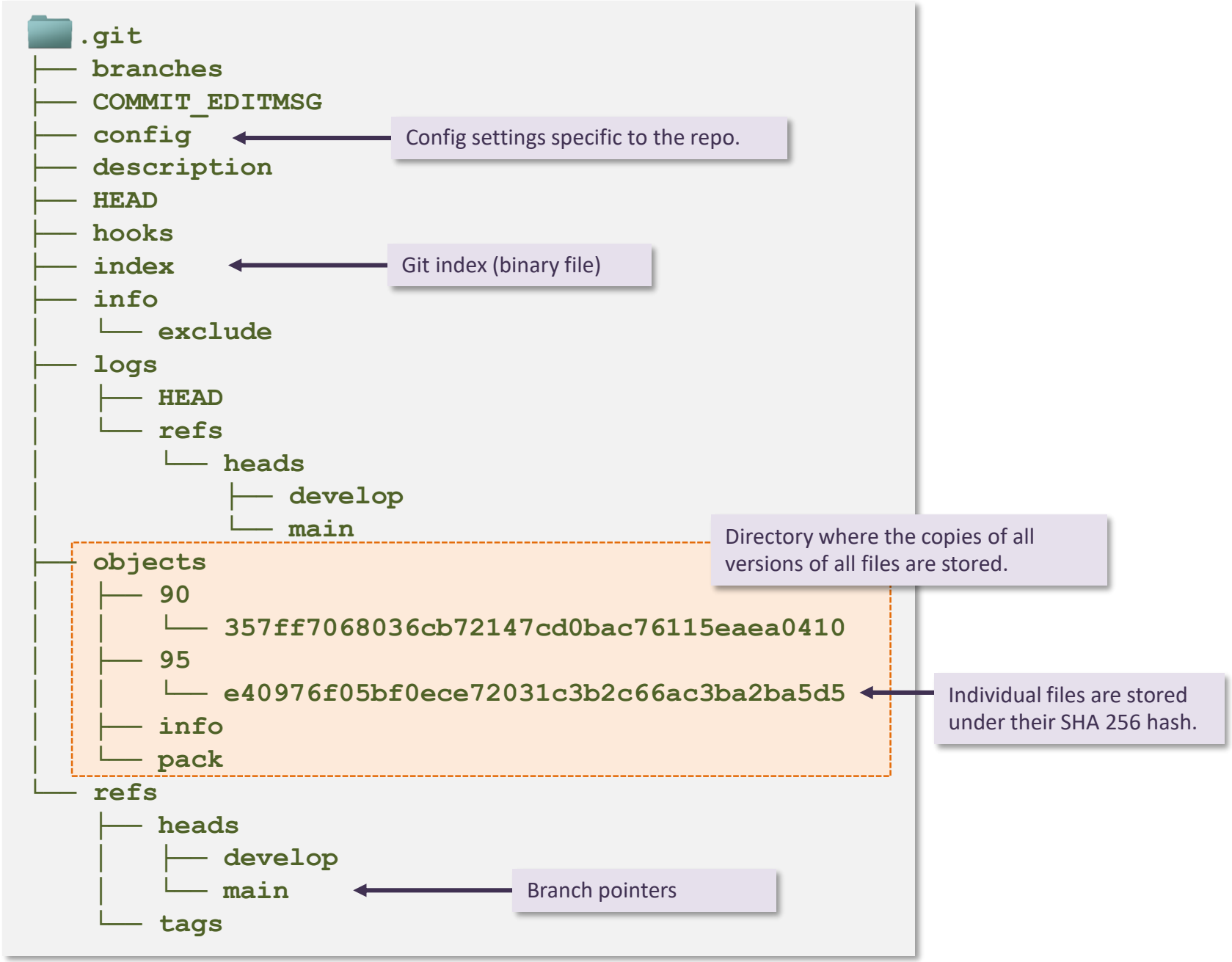
- It does not matter whether the directory is empty or already contains files/sub-directories.
- **Files** in a project directory (working tree) **are not automatically tracked** by Git (files are **untracked by default**).
- You can have both tracked and untracked files in a project directory.
- Only files located in the project directory – or one of its sub-directories – can be tracked.
- Project directories are self-contained – you can rename them or move them around in your file system.
- You can (should) have multiple Git repositories on your system – typically one per project or per code/script you develop * - don't use a single Git repo to track the entire content of your computer!
- Nesting Git repositories (i.e. having one repo inside another) is technically possible, but should be avoided unless there is a clear use-case for it.

Never delete the `.git` directory, you would lose the entire versioning history of your repository (along with all files not currently present in the working tree).

Supplementary material…

* An exception is the case of multiple projects that are tightly linked to another: in such cases it can be useful to have them all in a single repo – this is known as a **monorepo**.

Behind the scenes: the content of the `.git` directory



Supplementary material...

Making a **commit**

```
git add
```

```
git commit
```

Definition: the Git index (staging area)

In Git, creating a commit is a 2-step process:

Git index (staging area): "virtual space" where files are gathered before committing them to the repository. Acts as a buffer between the working tree and the repository, allowing to select changes to include in the next commit.

Technical note: in practice, the Git index is a file in Git's database.

Step 1 – **Staging files**

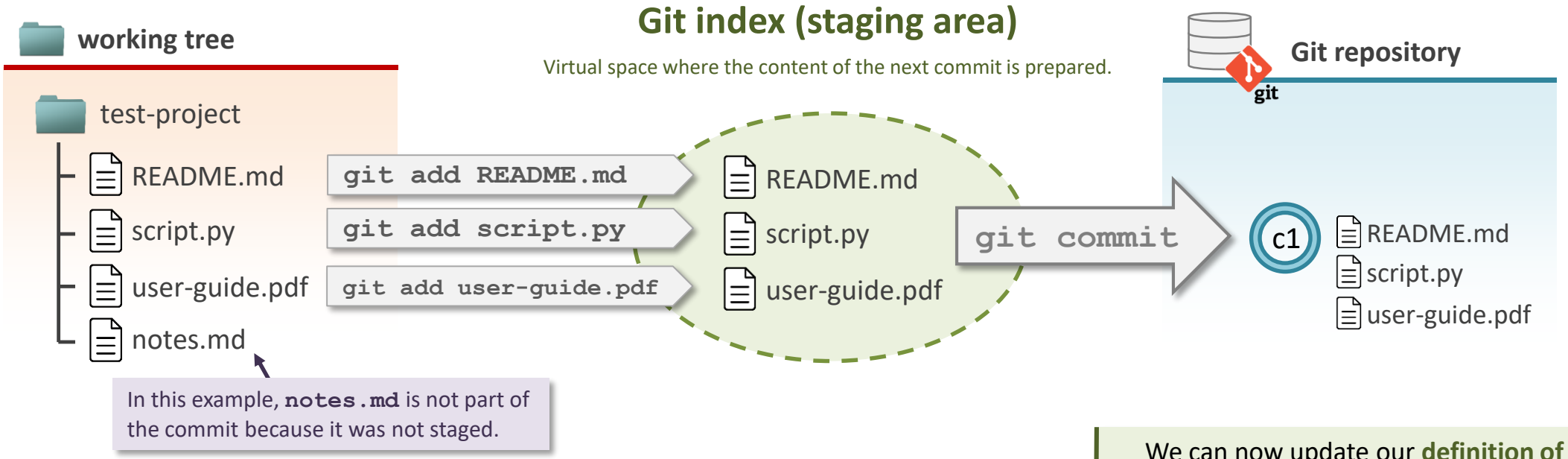
Selection of files to commit. To make a new or modified file part of the next commit, it must be added to the **Git index** (also known as the **staging area**).

```
git add <file or directory>
```

Step 2 – **Commit**

Create a commit with the current content of the Git index. A new commit (containing the current content of the Git index) is added to the repository.

```
git commit -m "commit message..."
```



We can now update our **definition of a commit**.

Commit = snapshot of the Git index at a given time.
Git index = content of your next commit.

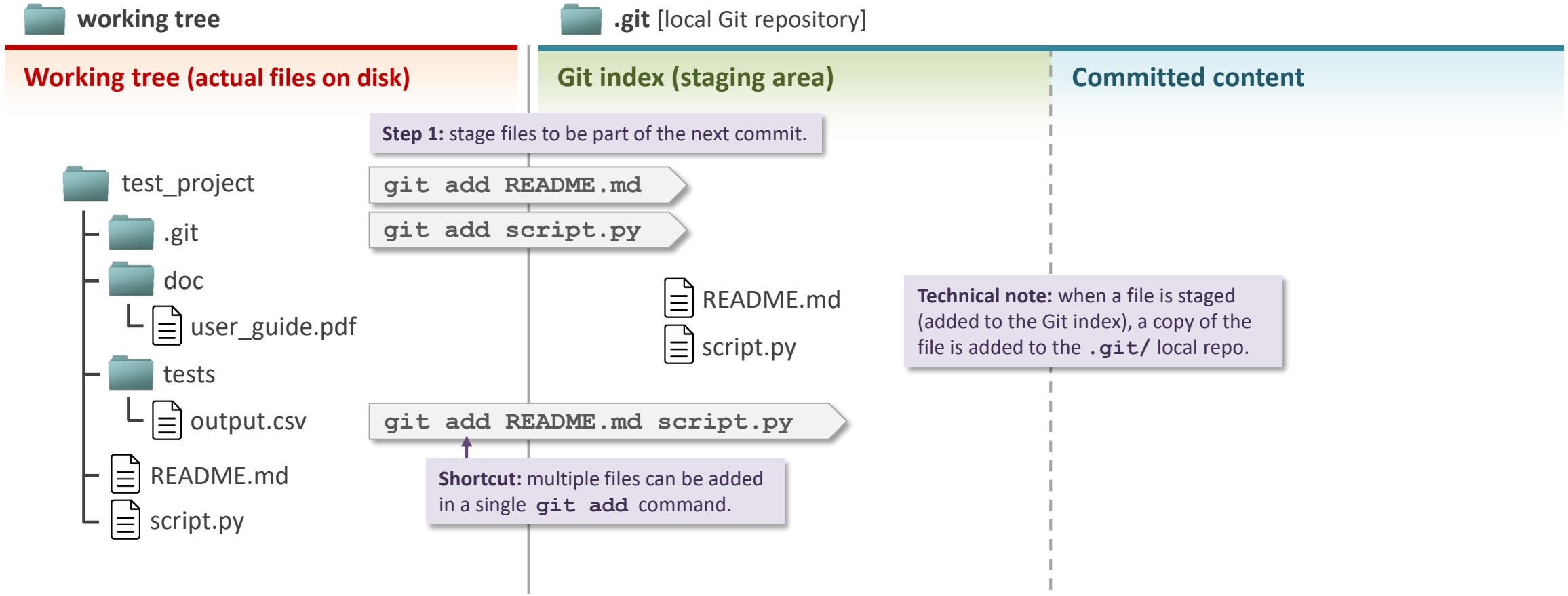
Why do we need this 2-step process ?

- Why do we need the Git index ?
- Why not simply commit the entire content of our directory ?

⇨ The objective of this 2-step procedure is to let users craft **“well thought” commits**.

- **Commits are** meant to be **meaningful units of change** in your code base (or the content you track).
- Not all current changes in the working tree need to be part of the next commit.

Staging and making a commit: a step-by-step example



File status (before staging)

```
$ git status
On branch main
No commits yet
Untracked files:
    doc/
    tests/
    README.md
    script.py
```

Initially, all files are untracked.

File status (after staging)

```
$ git status
On branch main
No commits yet
Changes to be committed:
  new file: README.md
  new file: script.py
Untracked files:
```

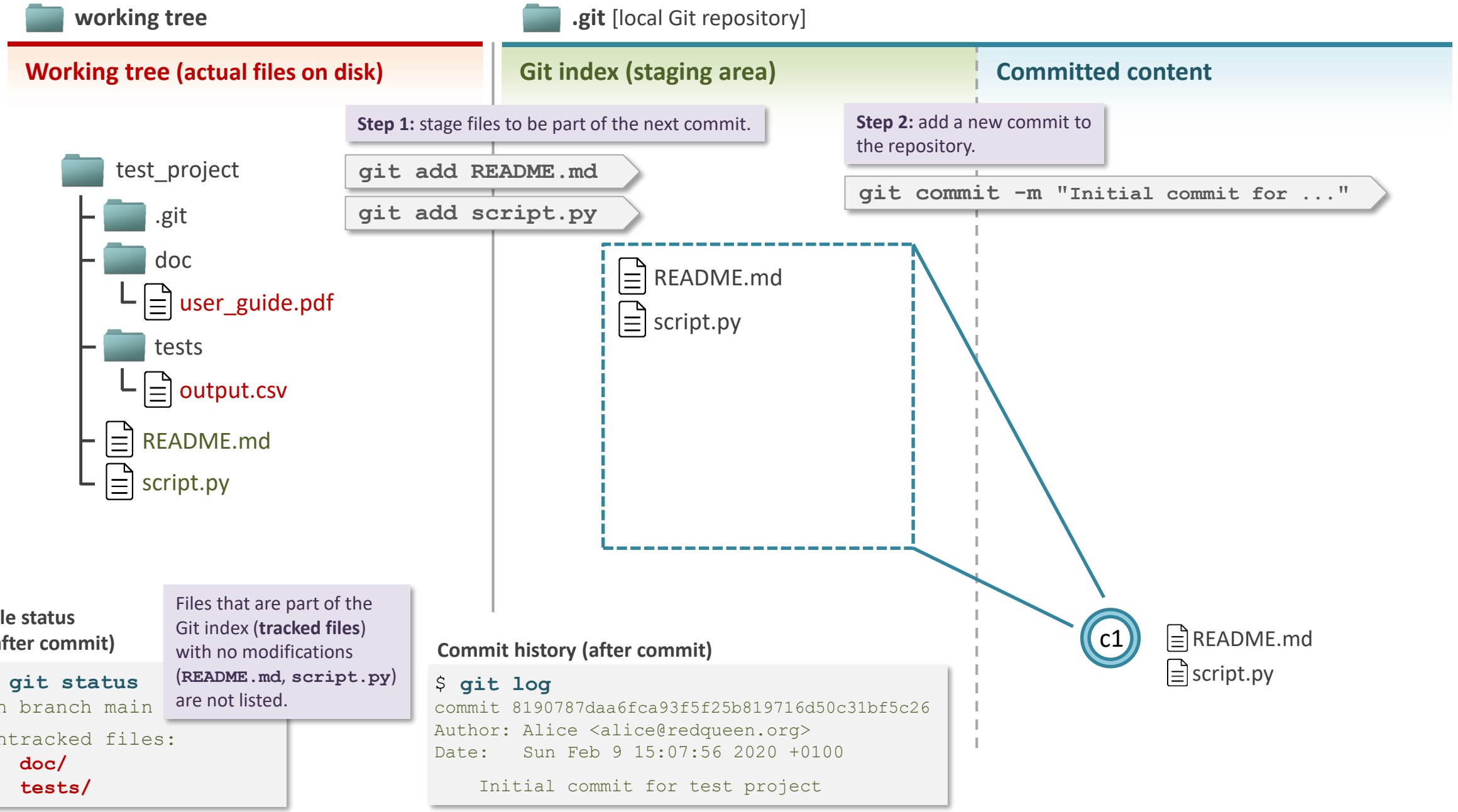
`git status` now indicates that 2 files have been newly added to the Git index.

Commit history

```
$ git log
fatal: your current branch 'main' does not have any commits yet
```

So far we only staged files, but no commit was made. This is why the output of `git log` is empty.

Staging and making a commit: a step-by-step example



Staging and making a commit: a step-by-step example

Important: once a version of a file was added to the Git index, it remains there and will be part of the next commit (unless we explicitly remove it). ***As long as a file is not modified***, there is ***no need to stage it again***.

working tree | **.git** [local Git repository]

Working tree (actual files on disk)

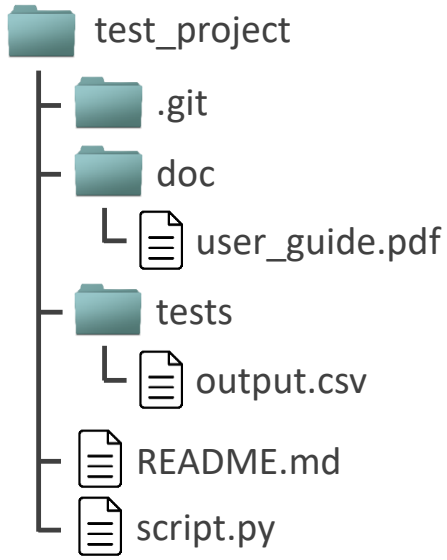
Git index (staging area)

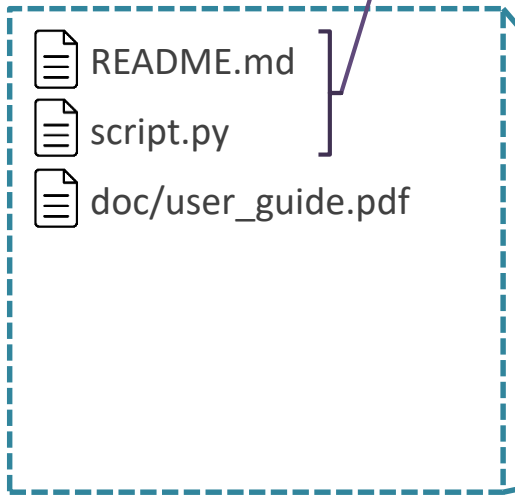
Committed content

Let's add a new file to our repo: `user_guide.pdf`

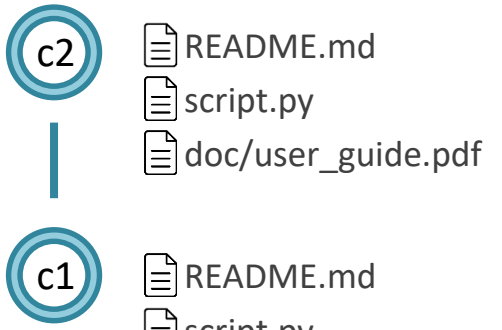
```
git add doc/
```

```
git commit -m "Add user guide"
```





Technical note: files that do not change across commits are stored only once: there is no wasteful data duplication.



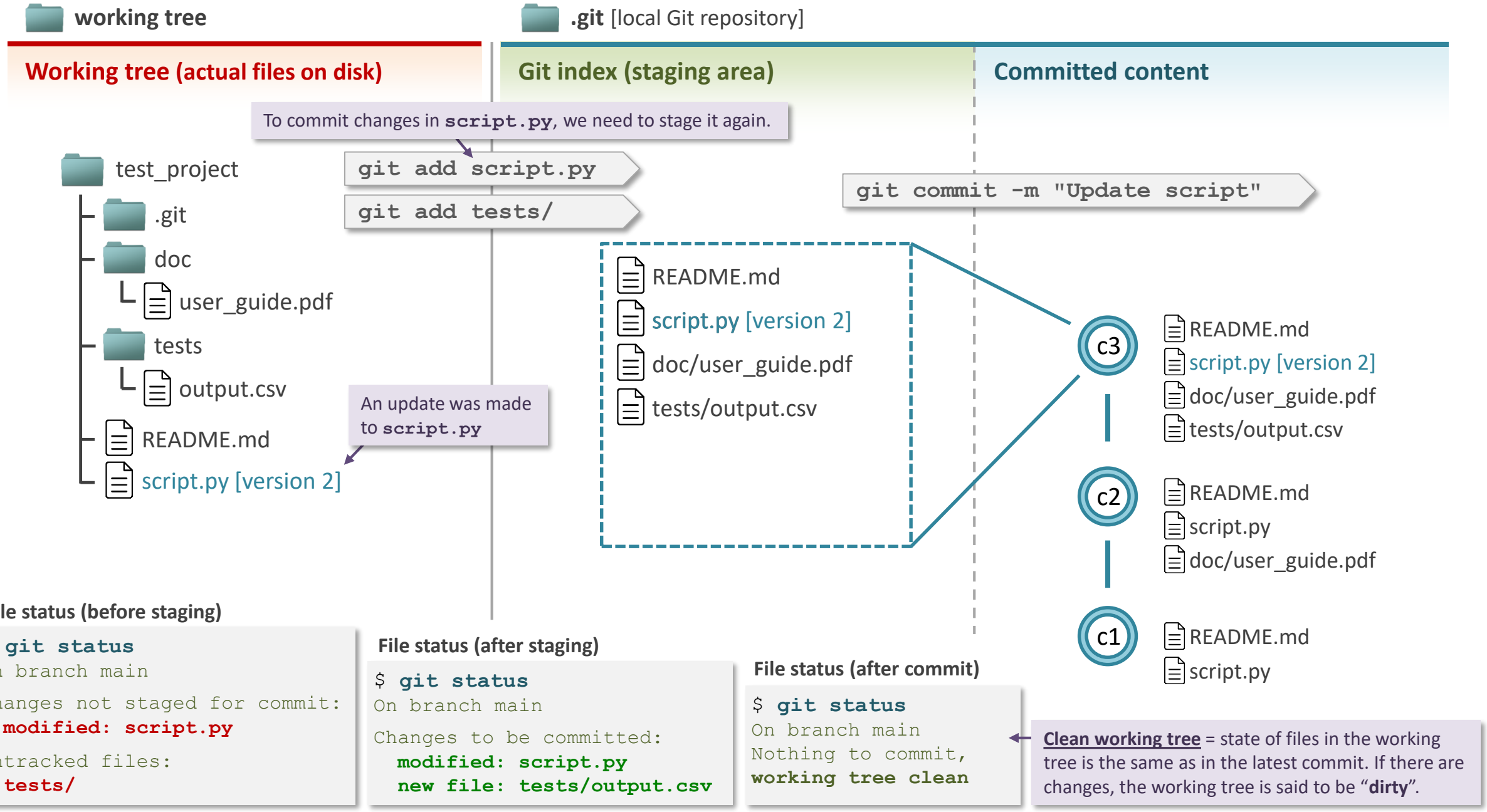
File status (after staging)

```
$ git status
On branch main

Changes to be committed:
  new file: doc/user_guide.pdf

Untracked files:
  tests/
```

Commit history (after commit)

```
$ git log
commit 04728026143ae57a71dcb7c1a503022041fb7d4d
Author: Alice <alice@redqueen.org>
Date:   Fri Oct 11 09:53:05 2024 +0200

    Add user guide

commit 8190787daa6fca93f5f25b819716d50c31bf5c26
Author: Alice <alice@redqueen.org>
Date: Fri Oct 11 08:43:15 2024 +0200

    Initial commit for test project
```

Staging and making a commit: a step-by-step example



File status (before staging)

```
$ git status
On branch main
Changes not staged for commit:
  modified: script.py
Untracked files:
  tests/
```

File status (after staging)

```
$ git status
On branch main
Changes to be committed:
  modified: script.py
  new file: tests/output.csv
```

File status (after commit)

```
$ git status
On branch main
Nothing to commit,
working tree clean
```

Clean working tree = state of files in the working tree is the same as in the latest commit. If there are changes, the working tree is said to be "**dirty**".

Summary: staging (`git add`) and committing (`git commit`) files

- By default, files in a directory under Git control are **untracked**.
- To include a file (in its current state) – or a change in file content – in the next commit, the file **must be added to the Git index (staged)** with:

```
git add <file or directory>      # Add the specified files/directories to the Git index.
```

  - Multiple files/directories can be added in a single command (by passing multiple file/directory names).
  - By default, the entire content of a file is added.
    Adding only part of a file is possible with the `--edit` or `--patch` options.

- **Staged files remain staged**, unless explicitly removed (with `git rm` or `git rm --cached`).
- **Modified files must be staged (added to the index) again**, if the new content is to be added to the next commit.

- To **create a new commit**, the command is:

```
git commit -m/--message "your commit message"
git commit
```

If no commit message is given, Git will open the default editor and ask you to enter a message **interactively**.

Reminder

commit = snapshot of the Git index

The Git index (staging area) can be thought of as a "virtual space" where the content of the next commit is prepared.

Shortcuts: for staging (`git add`) and committing (`git commit`) files

- **`git add` shortcuts:**

```
git add -u / --update     # Stages all already tracked files, but ignore untracked files.
git add -A / --all        # Stages all files/directories in the working tree (except ignored files), including file deletions.
git add .                 # Stages entire content of the current directory, except file deletions.
```

- **`git commit` shortcuts:**

This is a shortcut for:

```
git add <file or directory>
git commit -m "commit message"
```

```
git commit -m "commit message" <files or dirs>   # Stage and commit the specified files/directories in a single command.
git commit -a / -all -m "commit message"         # Stage and commit all modified tracked files in a single command.
```

`-a / --all` is a shortcut for:

```
git add -u
git commit -m "commit message"
```

It will not stage/commit untracked files.

Supplementary material...

Making commits: some advice

Git does not impose any restrictions on what and when things can be committed. *

However, it's best to:

- Make commits at ***meaningful points*** of your code/script development, for instance:
  - When a new feature was added (or a few related functions).
  - When a bug was fixed.
- Make ***multiple small commits instead of a large one*** if you are making changes that affect different functionalities of your code (this can make it easier to e.g. revert changes).
- ***Don't commit broken code on your main/master branch***, as this is the branch that others might use to get the latest version of your code.
  If you have partial work, you can commit it to a *temporary/feature* branch, and later merge it into *main/master* (more on branch management will follow later).

* One exception being that, by default, commits with zero changes are not allowed, but they are possible by using the `--allow-empty` option: `git commit --allow-empty`

Committing content: interactive commit message with the “vim” editor

```
$ git commit
```

When no commit message is specified, Git automatically opens a text editor. By default, this editor is “vim”.

```
Initial commit for test_project
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
#       new file:   README.md
#       new file:   script.py
#       new file:   doc/quick_start.md
#
```

- In the “vim” editor, press on the key “**i**” to enter edit mode
- In edit mode, you can now type your commit message.

Supplementary material...

Committing content: interactive commit message with the “vim” editor

```
Initial commit for test_project

This is the very first commit in this Git repo.
Way to go!

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
#       modified:   README.md
#       new file:   script.py
#       new file:   doc/quick_start.md
#
~
~
:wq
```

- Commit message can be entered over multiple lines.
- By convention, try to keep lines reasonably short (<= 80 chars)

- Press “**Esc**” to exit “edit” mode.
- Type “**:wq**” in the vim “command” mode.

Press “**Enter**” to exit vim and save your commit message.

```
[main (root-commit) 8190787] Initial commit for test_project
3 files changed, 6 insertions(+)
create mode 100644 README.md
create mode 100644 script.py
create mode 100644 doc/quick_start.md
```

- You are now back in the shell and your commit is done.

Supplementary material...

Demo

- Initializing a new Git repo.
- Adding content to the Git repo.
- Making a commit with interactive commit message.

exercise 1 – part A

Your first commit

This exercise has helper slides

Exercise 1 help: bash (shell) commands you may need during this course

`cd <directory>`	Change into directory (enter directory).
`cd ..`	Change to parent directory.
`ls -l`	List content of current directory.
`ls -la`	List content of current directory including hidden files.
`pwd`	Print current working directory.
`cp <file> <dest dir>`	Copy a file to directory “dest dir”.
`mv <file> <new name>`	Rename a file to <new name>.
`mv <file> <directory>`	Move a file to a different directory.
`cat <file>`	Print a file to the terminal.
`less <file>`	Show the content of a file (type “q” to exit).
`vim <file>`	Open a file with the “vim” text editor.
`nano <file>`	Open a file with the “nano” text editor.

Reminder...

Inspecting **file status**

```
git status
```

```
git diff
```

Display file status

`git status` Display the status of files in the working tree.

- *** Modified files**: files with changes in content as comparted to the latest commit.
- **** Staged files that have not been modified** since the last commit (**unmodified files**) are not listed, but they are still in the index and will be part of the next commit.
- **Ignored files** are also not listed.

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   LICENSE.txt
        modified:   README.md
        modified:   script.py
        deleted:    test/test_output.csv
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md
        modified:   doc/user_guide.md
        deleted:    test/log.txt
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        untracked_file.txt
```

new file = file is not present in latest commit.

modified = file is modified compared to latest commit.

deleted = file is present in latest commit and will now be removed

modified = file is modified compared Git index.

deleted = file is deleted on disk, but is still present in the Git index (and the latest commit).

Staged files **

Unstaged files

Modified files *

untracked files

Green = files with (changes in) content (compared to the latest commit) that has been staged and will be part of the next commit.

Red = files with (changes in) content (compared to the latest commit) that is not staged. These changes will not be part of the next commit.

Note: the (new) content of a file can be **partially staged**: some changes in the file are staged (added to the index), while some remain unstaged. This is the case in the example above for the `README.md` file (which is why it's listed in both the staged and unstaged sections). **Only the staged content will become part of the next commit**.

File status in Git: summary

Possible statuses for files in Git:

- **Tracked** – file that is currently under version control. More specifically, it is currently part of the Git index (staging area) and therefore also generally part of the latest commit *. Tracked files can be further categorized as:
  - **Unmodified** – the file is part of the latest commit * (and the Git index), and no change was made to the file since then. In other words, the content of the file in the working directory (working tree) is the same as in the latest commit. Unmodified files are not listed by the `git status` command.
  - **Modified** –the content of the file in the working directory (working tree) differs from the latest commit *. Modified files can be staged, unstaged, or partially staged.
    - **Staged**: the difference in content has been added the Git index (staging area), and will therefore be committed with the next commit.
    - **Unstaged**: the difference in content has not been staged (not part of the Git index), and will therefore not be part of the next commit.
    - **Partially staged**: some differences (but not all) have been staged (added to the Git index). Only the staged differences will be part of the next commit.
- **Untracked** - file present in the project directory (working tree), but not currently under version control by Git. More specifically, the file is not currently present in the Git index – but could be part of an earlier commit.
- Ignored - untracked file that is part of the repository's "ignore list" (`.gitignore` or `.git/info/exclude` file). Ignored files are not listed by the `git status` command.

* more precisely: the commit to which the HEAD pointer is currently pointing – this concept is explained later in the slides.

Supplementary material...

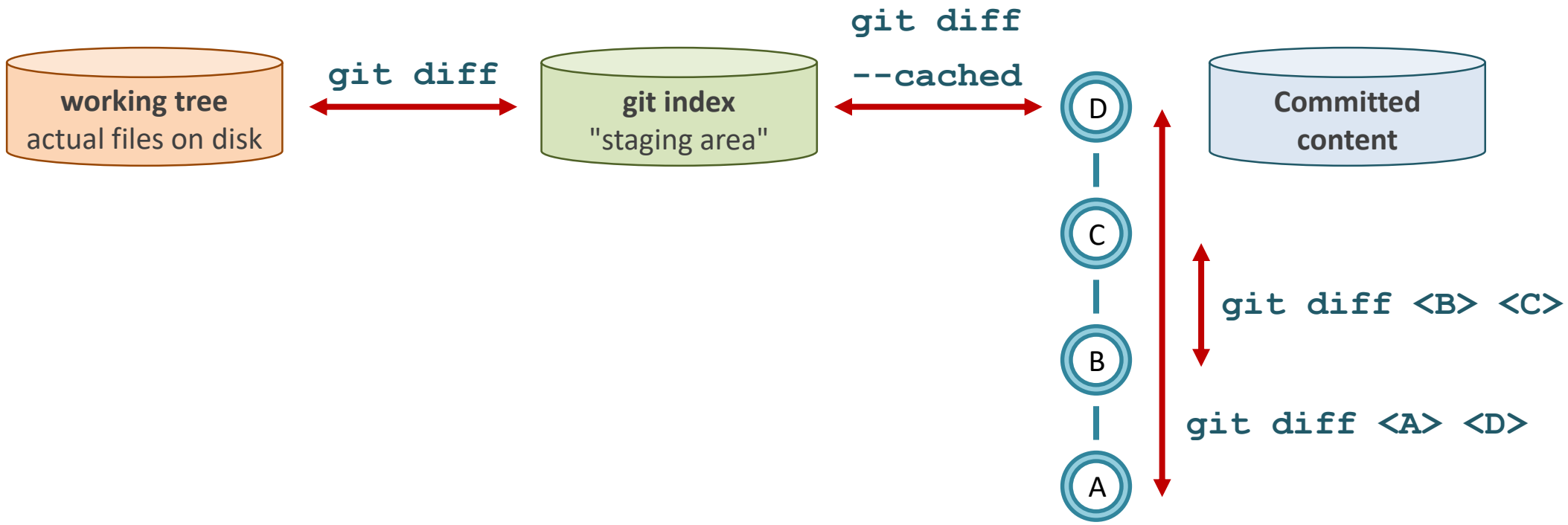
How do I know **what changed** and which changes are staged ?

`git diff` Show differences between two states of a Git repo.

```
git diff file                    # Show diff for a specific file between working tree and index.
git diff <older commit> <newer commit>    # show diff between 2 commits.

git diff --cached                # Show diff between index and last commit.
git diff --name-only             # Show only file names, not the changes.
```

Example output

```
❯ git diff
diff --git a/README.md b/README.md
index f5e333d..844d178 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,3 @@
Project description:
-This is a test
+This is a demo project
+and it's pretty useless
```



Inspecting **commits** and **history**

```
git show
```

```
git log
```

Display the “content” of a commit

`git show` Display the changes in file content introduced by a commit.

```
git show <commit reference>
git show
```

with no argument, the latest commit on the current branch is shown (i.e. HEAD)

Examples of commit references:

- A commit ID (hash): **89d201f**
- A branch name: **develop**
- A tag name: **1.0.7**
- The **HEAD** pointer.
- A relative reference: **HEAD~3**

If no commit reference is given, **HEAD** is used as default.

The detail of changes can only be shown for plain text files.

Example:

```
$ git show 89d201f
commit 89d201fd01ead6a499a146bc6da5aa078c921ecf
Author: Alice <alice@redqueen.org>
Date:   Wed Feb 19 14:00:02 2020 +0100

    Add stripe color option to class Cheshire_cat

diff --git a/script.sh b/script.sh
index d7bfdc8..fa99250 100755
--- a/script.sh
+++ b/script.sh
@@ -7,13 +7,28 @@

# def Cheshire_cat():
-   def __init__(self, name, owner="red queen"):
+   def __init__(self, name, owner="red queen", stripe_color="orange"):
+       self.stripe_color = stripe_color
```

`git show --name-only <ref>`

Only display file names (without the changes)

```
$ git show --name-only 89d201f
commit 89d201fd01ead6a499a146bc6da5aa078c921ecf
Author: Alice <alice@redqueen.org>
Date:   Wed Feb 19 14:00:02 2020 +0100

    Add stripe color option to Cheshire_cat

script.sh
```

Display commit history

Print the commit history of the repository, newest commit to oldest (i.e. newest commit at the top)

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

`git log` has many options to format its output.

See `git log --help`

Example: default view (detailed commits of current branch).

```
$ git log
commit f6ceaac2cc74bd8c152e11b9c12ada725e06c8b9 (HEAD -> main, origin/main)
Author: Alice alice@redqueen.org
Date:   Wed Feb 19 14:13:30 2020 +0100

    Add stripe color option to class Cheshire_cat

commit f3d8e2280010525ba29b0df63de8b7c2cd7daeaf
Author: Alice alice@redqueen.org
Date:   Wed Feb 19 14:11:56 2020 +0100

    Fix off_with_their_heads() so it now passes tests

commit cfd30ce6e362bb4536f9d94ef0320f9bf8f81e69
Author: Mad Hatter mad.hatter@wonder.net
Date:   Wed Feb 19 13:31:32 2020 +0100

    Add .gitignore file to ignore script output
```

Example: compact view of current branch

```
$ git log --oneline
f6ceaac (HEAD -> main, origin/main) peak_sorter: add authors to script
f3d8e22 peak_sorter: display name of highest peak when script completes
cfd30ce Add gitignore file to ignore script output
f8231ce Add README file to project
821bcf5 peak_sorter: add +x permission
40d5ad5 Add input table of peaks above 4000m in the Alps
a3e9ea6 peak_sorter: add first version of peak sorter script
```

Example: compact view of entire repo (all branches)

```
$ git log --all --decorate --oneline --graph
* fc0b016 (origin/feature-dahu, feature-dahu) peak_sorter: display highest peak at end of script
* d29958d peak_sorter: add authors as comment to script
* 6c0d087 peak_sorter: improve code commenting
* 89d201f peak_sorter: add Dahu observation counts to output table
* 9da30be README: add more explanation about the added Dahu counts
* 58e6152 Add Dahu count table
| * f6ceaac (HEAD -> main, origin/main) peak_sorter: add authors to script
| * f3d8e22 peak_sorter: display name of highest peak when script completes
|/
* cfd30ce Add gitignore file to ignore script output
* f8231ce Add README file to project
| * 1c695d9 (origin/dev-jimmy, dev-jimmy) peak_sorter: add check that input table has the ALTITUDE and PEAK columns
| * ff85686 Ran script and added output
|/
* 821bcf5 peak_sorter: add +x permission
* 40d5ad5 Add input table of peaks above 4000m in the Alps
* a3e9ea6 peak_sorter: add first version of peak sorter script
```

Adding custom shortcuts to Git

Some git commands can be long and painful to type, especially when you need them often!
To shorten a command, you can create **custom aliases**:

```
git config --global alias.<name of your alias> "command to associate to alias"
```

Example:

```
git config --global alias.adog "log --all --decorate --oneline --graph"
```

With the alias set, you can now simply type:

```
git adog
```

Ignoring files

- By default, files that are not added to a Git repo are considered **untracked**, and are always listed as such by `git status` .
- To stop Git from listing files as **untracked**, they can be added to one of the following "ignore" files:

`.gitignore`

- For files to be **ignored by every copy of the repository**.
- `.gitignore` is meant to be tracked: `git add .gitignore`
- Examples:
  - outputs of tests
  - .Rhistory, .RData
  - .pyc (compiled version of python code)

Most of the time, this is the method you will want to use to ignore files.

`.git/info/exclude`

- For files that should be **ignored only by your own local copy of the repository**.
- Not versioned and not shared.
- Examples:
  - Files with some personal notes.
  - Files specific to your development environment (IDE).

Use this method for **special cases** where a file should **only be ignored in your local copy of the repo.**

Example of a `.gitignore` file

```
my_tests.py
.Rdata
.Rhistory
*.pyc
test_outputs/
```

- Files to ignore are added by manually editing the two above-mentioned files.
- Files can be ignored based on their full name, or based on glob patterns (see next slide for examples).
  - `*.txt` ignore all files ending in ".txt"
  - `*.[oa]` ignore all files ending either in ".o" or ".a"
  - `logs/` appending a slash indicates a directory. The entire directory and all of its content are ignored.
  - `!dontignorethis.txt` adding a **!** In front of a file name means it should not be ignored (exception to rule).

Supplementary material...

exercise 1 – part B and C

Your first commit

Part II

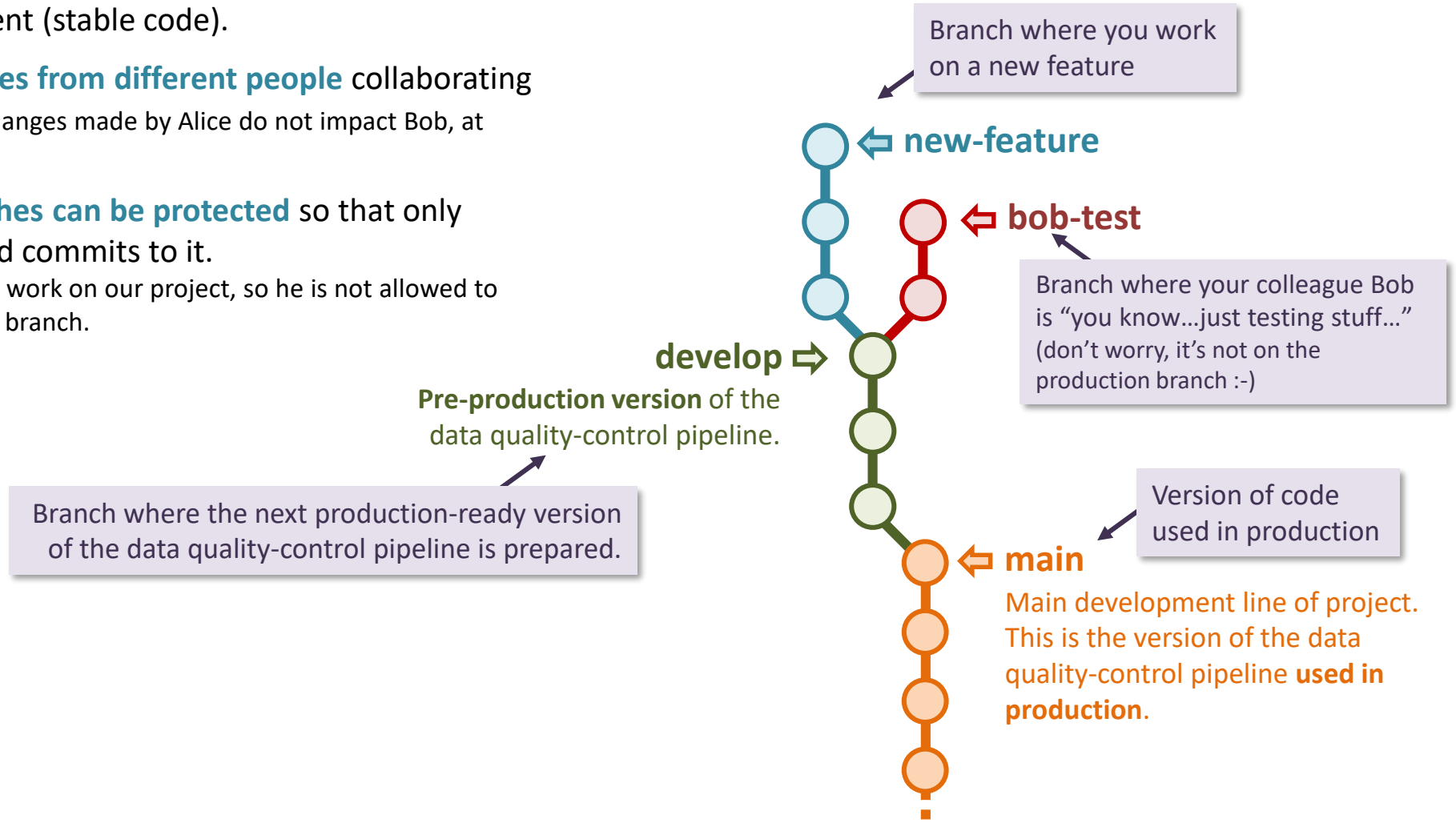
Git **branches**

Managing multiple lines of development

Why branches? An illustration with a data quality-control pipeline project

"Branching" means to **diverge from the main line of development**.

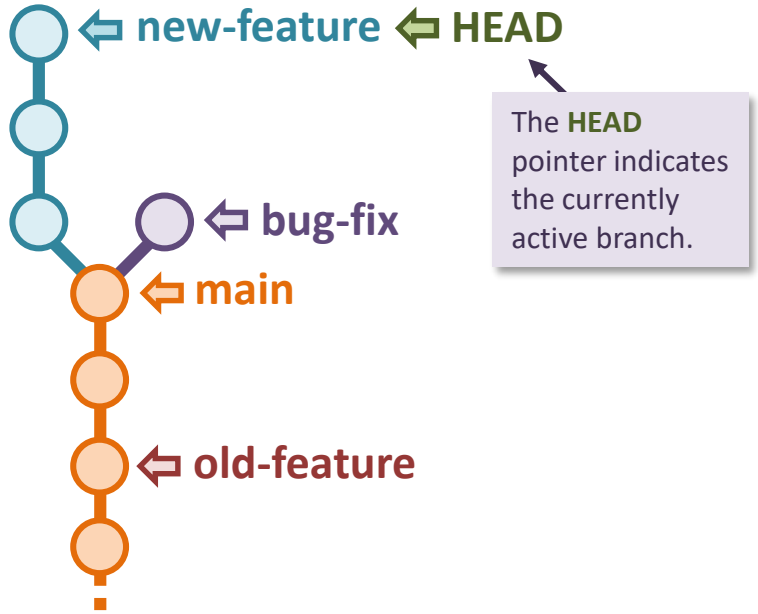
- Branches **isolate new changes** (work in progress) from the main line of development (stable code).
- Branches **isolate changes from different people** collaborating on a same project (so changes made by Alice do not impact Bob, at least not immediately).
- On online repos, **branches can be protected** so that only selected people can add commits to it.
  Use case: Bob just started to work on our project, so he is not allowed to make changes to the "main" branch.



Git is **designed to encourage branching**: branches are "cheap" (don't take much disk space) and switching between them is fast.

What are branches?

- A branch is just a pointer to a commit.
- A branch is **very lightweight** (41 bytes).
- By convention, the **main** branch is the branch representing the stable version of your work.
- To know which is the currently active branch, Git uses the **HEAD** pointer. The **HEAD** pointer always points to the currently active branch (except for the special case of "detached HEAD" mode, discussed later in the second part of this course).
- New commits are always added at the top of the currently active branch* (except for the special case of "detached HEAD" mode).



The **main** branch **is no special branch**. It is simply the default name given to the branch created when initializing a new repo [`git init`]. It has become a convention to use this branch as the stable version of a project.

Note: in earlier versions, the "**main**" branch used to be called the "**master**" branch.

Illegal characters in branch names

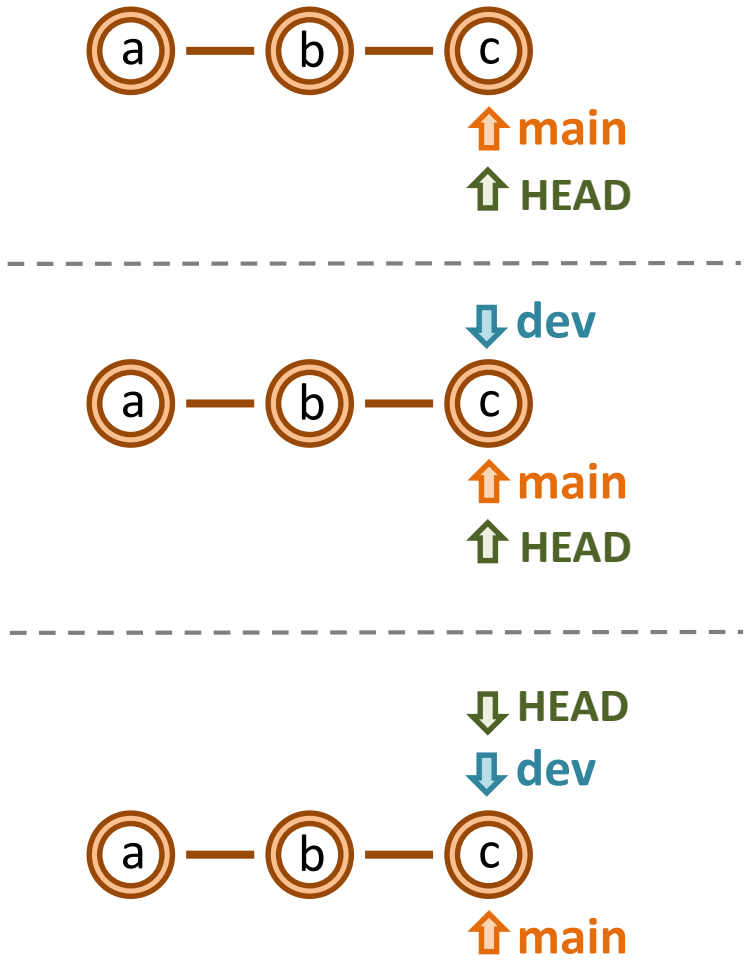
Spaces and some characters such as `,~^:?*[]\` are not allowed in branch names. It is strongly recommended to stick to lowercase letters, numbers and the "dash" character [**–**].

Create and **switch** branches

Create a new branch: `git branch <branch name>`

Switch to another branch: `git switch <branch name>`

Create a new branch **and switch** to it: `git switch -c <branch name>`



`git branch dev`

`git switch dev`

`git switch -c dev`

The `-c` option is to create and switch to the new branch immediately.

switch vs. checkout

On older Git versions the `git switch` command does not exist.
Instead, `git checkout` is used to switch branches:

```
git checkout <branch name>
git checkout -b <branch name>
```

The `git switch` command was introduced in Git version 2.23 as an replacement to `git checkout` for switching branches. This was done because the `checkout` command already has other uses (e.g. to extract older files from the Git database), and it was deemed confusing that a same command would have multiple usages. It remains nevertheless possible to switch branches with the `git checkout` command in recent Git versions.

Create and switch branches (continued)

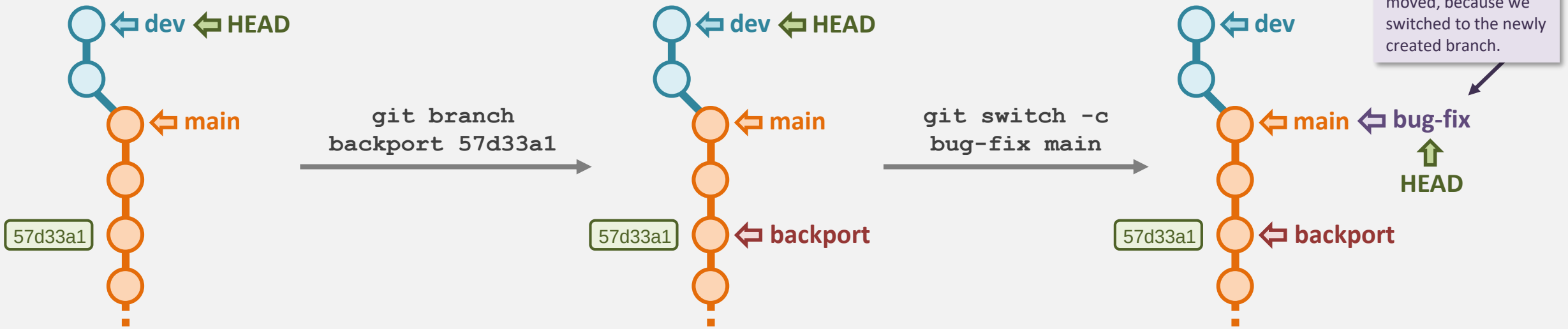
- By default new branches are created at the current position of the **HEAD** pointer (i.e. the current commit).
- But they can be created at any specified reference.

Reference to a commit, branch or tag.
The default reference is **HEAD**.

Create a new branch: `git branch <branch name> <reference>`

Create a new branch **and switch** to it: `git switch -c <branch name> <reference>`

Example:



List branches and identify the currently active branch

`git branch`	List **local** branches
`git branch -a`	List local **and remote** branches

Examples

```
❯ git branch
devel
* main
new-feature
```

The * denotes the currently checked-out (active) branch. Generally displayed in green.

```
❯ git branch -a
devel
* main
new-feature
remotes/origin/main
remotes/origin/devel
```

Remote branches (to be precise, pointers to remote branches) are shown in red and are named **remotes/<remote name>/<branch name>**

A convenient alternative: `git adog` (`git log --all --decorate --oneline --graph`) will also show all branches.
The currently active branch can be identified as it has the HEAD pointing to it.

```
* 351dca6 (HEAD -> main, origin/main, origin/HEAD) peak_sorter: added authors to script
* f3d8e22 peak_sorter: display name of highest peak when script completes
| * 076aa80 (origin/feature-dahu, feature-dahu) peak_sorter: display highest peak at end of script
| * d29958d peak_sorter: added authors as comment to script
| * 6c0d087 peak_sorter: improved code commenting
| * 89d201f peak_sorter: add Dahu observation counts to output table
| * 9da30be README: add more explanation about the added Dahu counts
| * 58e6152 Add Dahu count table
|/
* cfd30ce Add gitignore file to ignore script output
```

Demo: `git switch`

- What happens in the working directory when switching branches

git merge

get branches back together

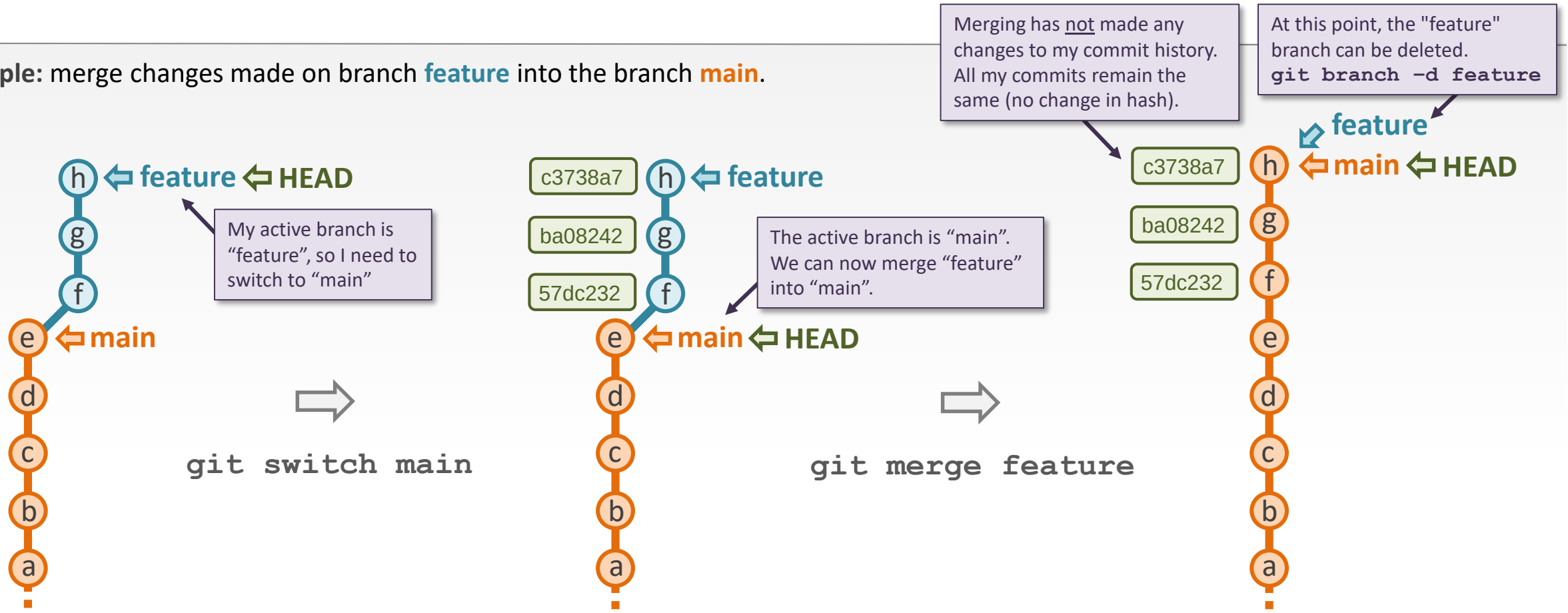
Merging branches

- **Merge:** incorporate changes from the specified branch into the currently active (checked-out) branch.

```
git merge <branch to merge into the current branch>
```

⚠ **Before running the command, make sure that the branch into which the changes should be merged is the currently active branch.** If not, use `git switch <branch>` to checkout the correct branch.

Example: merge changes made on branch **feature** into the branch **main**.



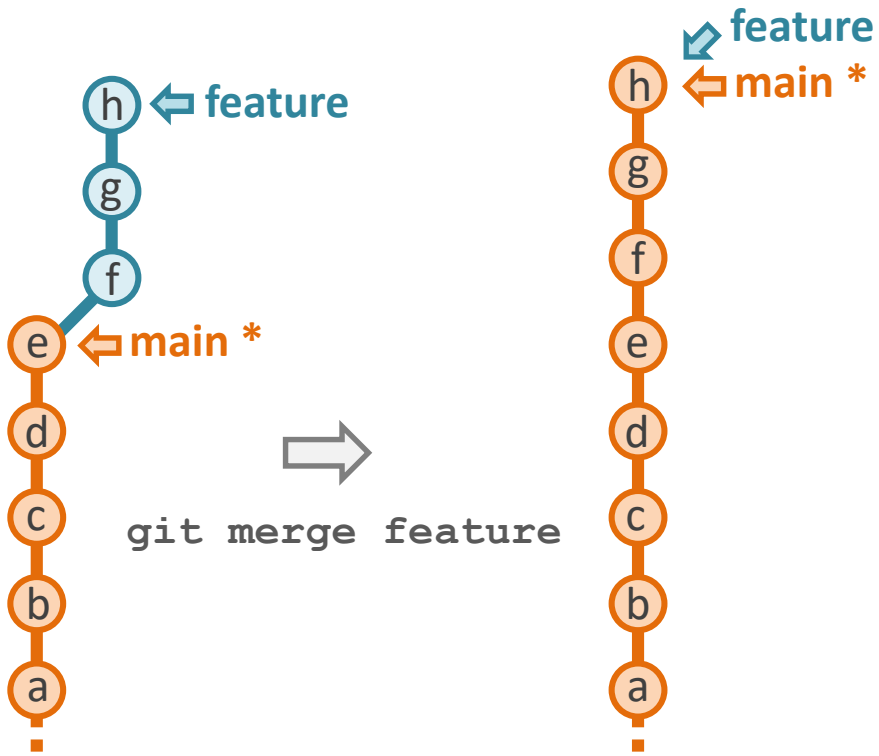
Two types of merges

- **Fast-forward merge:** when branches **have not diverged.**
  The branch that is being merged (here **feature**) is rooted on the latest commit of the branch that it is being merged into (here **main**).

- **3-way merge:** when branches **have diverged**. This introduces an extra "**merge commit**".
  The **common ancestor** of the 2 branches is not the last commit of the branch we merge into (here **main**).

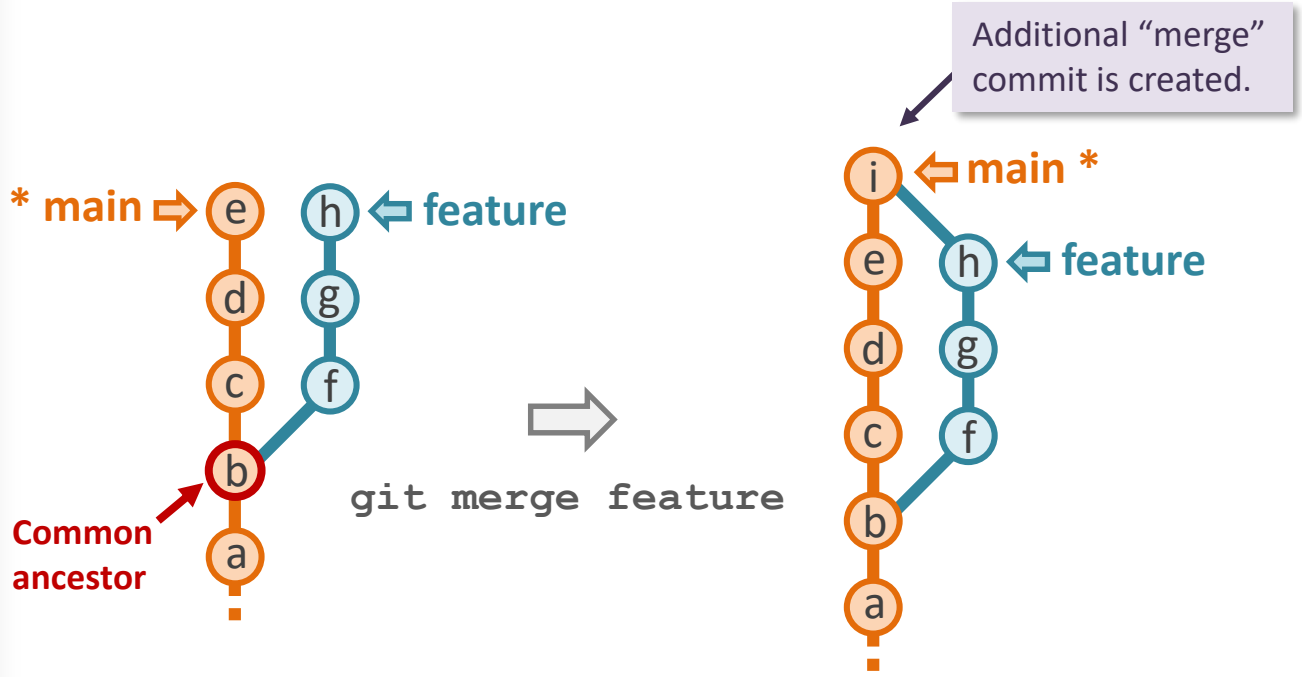
Fast-forward merge

- Guaranteed to be conflict free.



3-way merge (non-fast-forward)

- Creates an additional "merge commit" (has 2 parents).
- Conflicts may occur.



* denotes the currently active (checkout-out) branch.

Demo

- Merging branches (fast-forward and 3-way merge)

Deleting branches

Branches that are merged and are not used anymore can (should) be deleted.

`git branch -d <branch name>` ← **safe option:** only lets you delete branches that are fully merged.

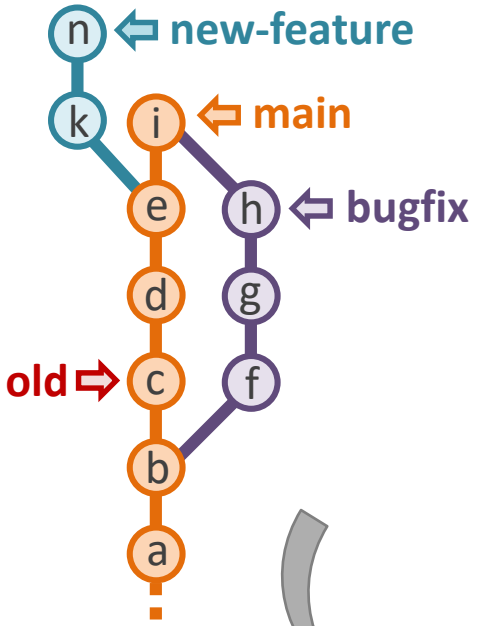
`git branch -D <branch name>` ← **YOLO option:** lets you delete any branch.

- Note: A currently active (checked-out) branch cannot be deleted. You must switch to another branch before deleting it.

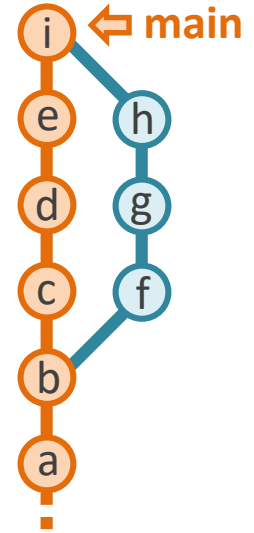
Examples

```
# The 'bugfix' and 'old' branches are fully merged. We can use -d.
› git branch -d bugfix
Deleted branch bugfix (was bd898dc)

› git branch -d old
Deleted branch old (was 75d3fed)

# Trying to delete a non-merged branch with -d will fail.
› git branch -d new-feature
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.

# -D allows deletion of a non-merged branch.
› git branch -D new-feature
Deleted branch new-feature (was f2a898b)
```

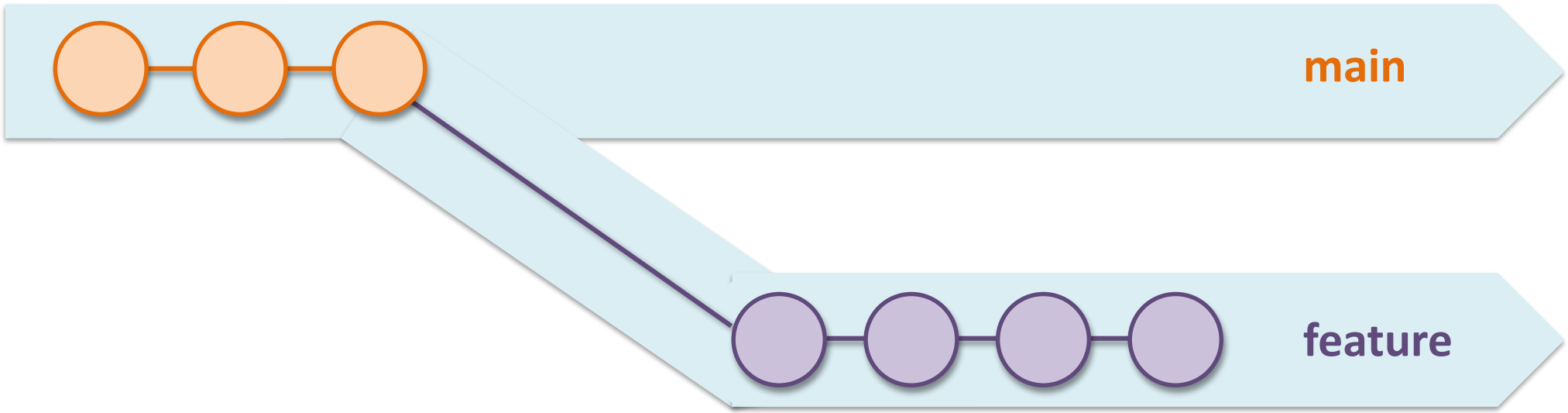
Deleted a branch by mistake ? – no panic !
This hash can be used to re-create it:
`git branch new-feature f2a898b`



Commands from "Example" box



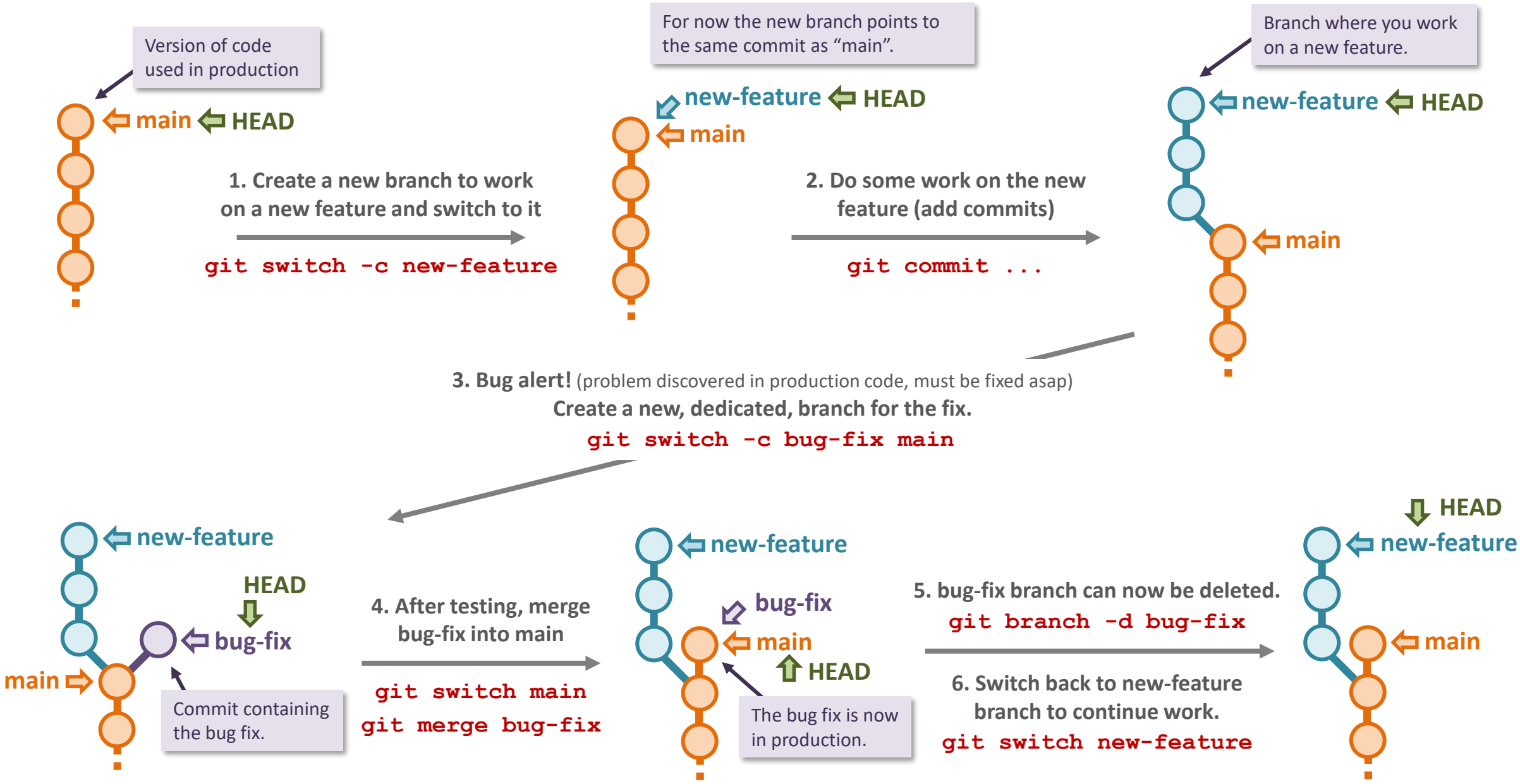
Branch management: best practices



- **Use branches** to develop and tests new changes to your code/scripts - don't test directly on main.
- **Don't hesitate to create branches**, they are "cheap" (they don't add much overhead to the git database).
- Delete branches that are no longer used.

⚠ **Don't change the history on the main branch** if your project is used by others.

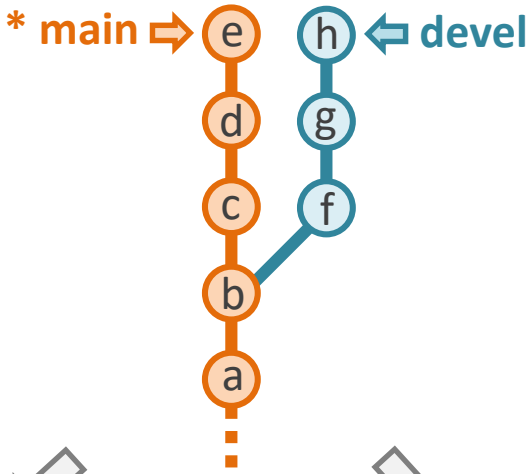
Recap - example of branched workflow: adding a new feature to an application and fixing a bug



Branch reconciliation strategies when history has diverged: merge vs. rebase

merge (3-way merge)

+ **Preserves history perfectly**.
+ Potentials conflicts must be solved only once.
- Creates an additional merge commit.
- Often leads to a "messy" history.

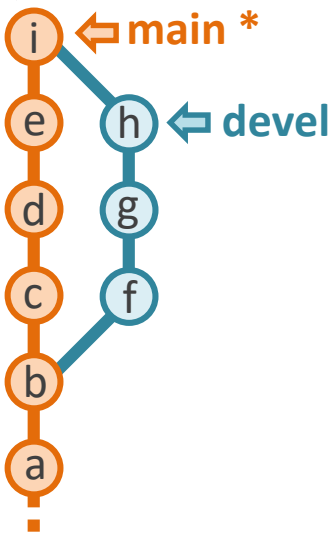


rebase + fast-forward merge

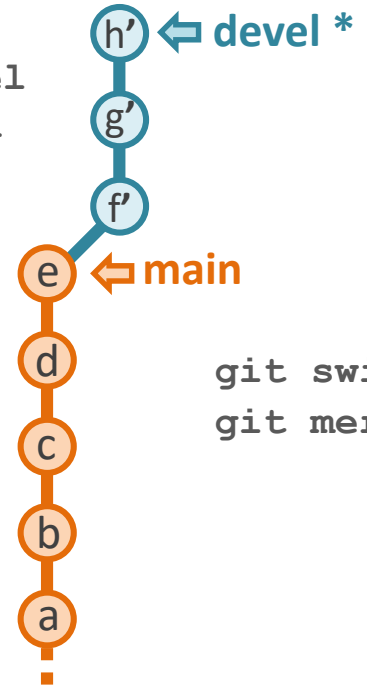
+ **Cleaner history** = easier to read and navigate.
- Conflicts may have to be solved multiple times.
- Loss of branching history.
  History of rebased branch is rewritten, not a problem in general.

`git merge devel`

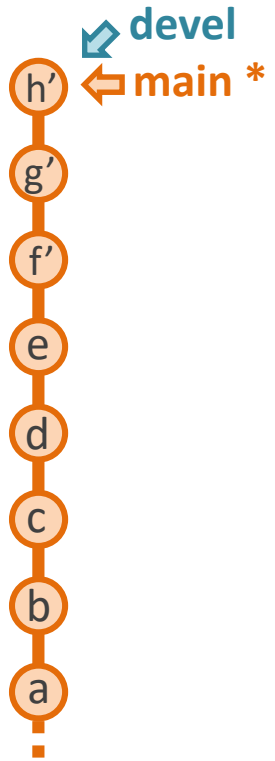
Additional "merge commit".



`git switch devel`
`git rebase main`



`git switch main`
`git merge devel`



Spoiler-alert: the end result is the same, i and h' have the same content.

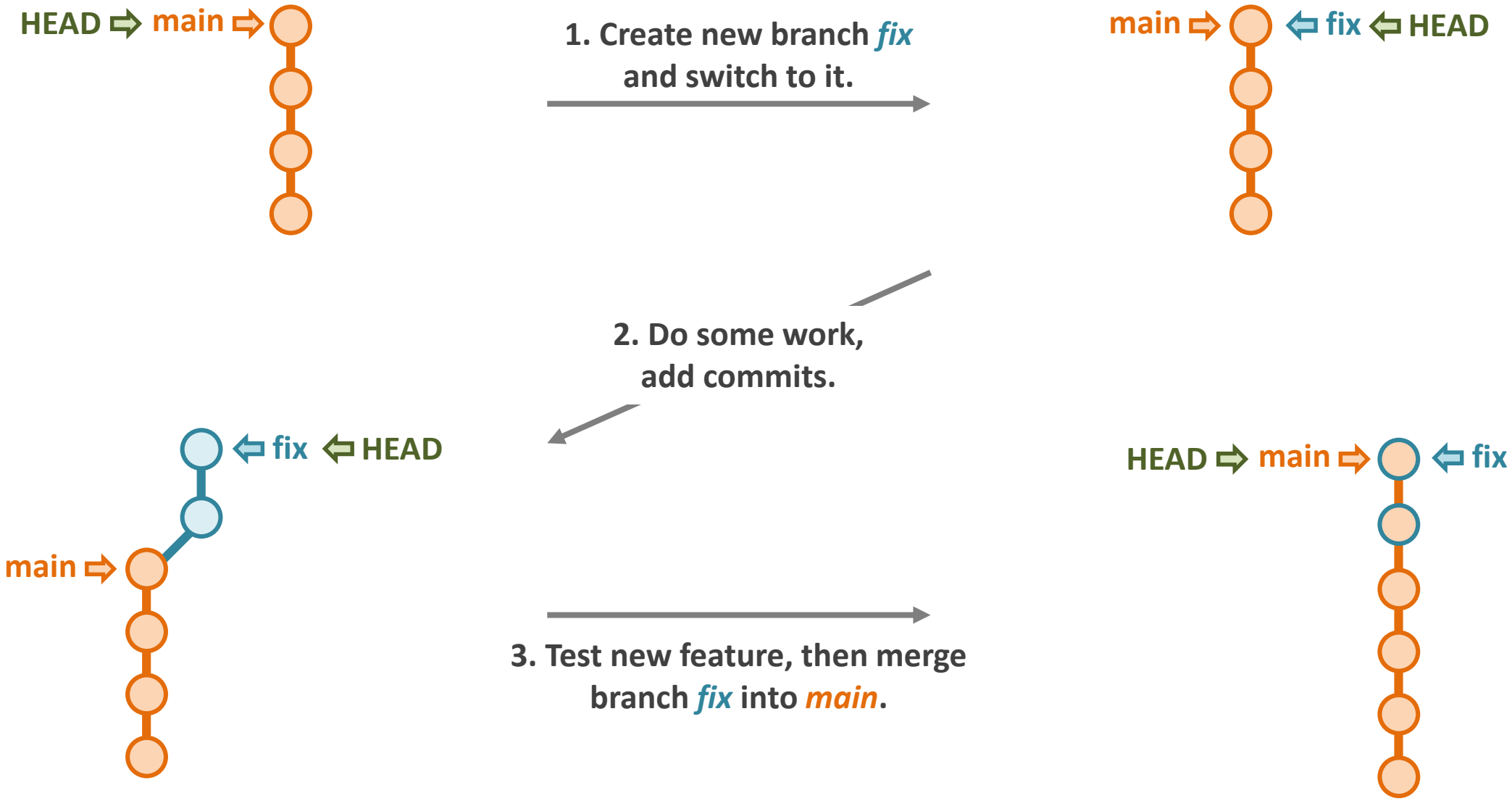
Supplementary material...

exercise 2

The Git reference webpage

This exercise has helper slides

Exercise 2 help: workflow summary



Reminder...

Part III

Working with **remotes**

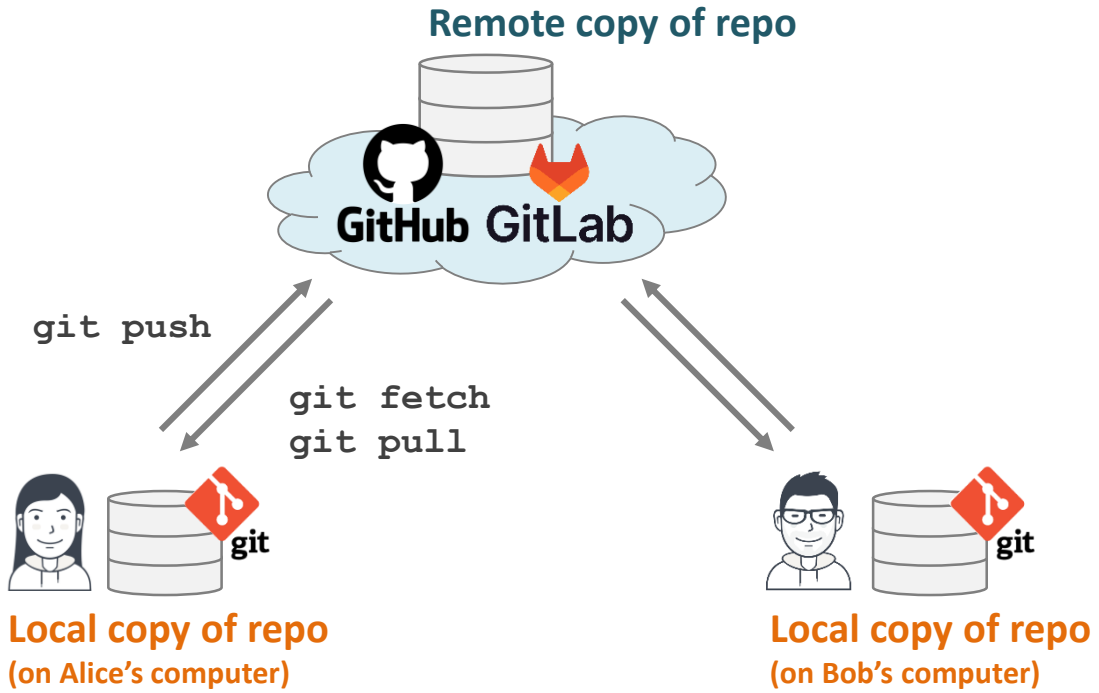
Linking your local repo with an online server

What is a “remote” ?

A **remote** is a copy of a Git repository that is stored on a server (i.e. online).

Remotes are very useful, as they allow you to:

- **Backup** your work.
- **Collaborate** and synchronize your repo with other team members.
- **Distribute** your work – i.e. let other people clone your repo (e.g. like the repo of this course).

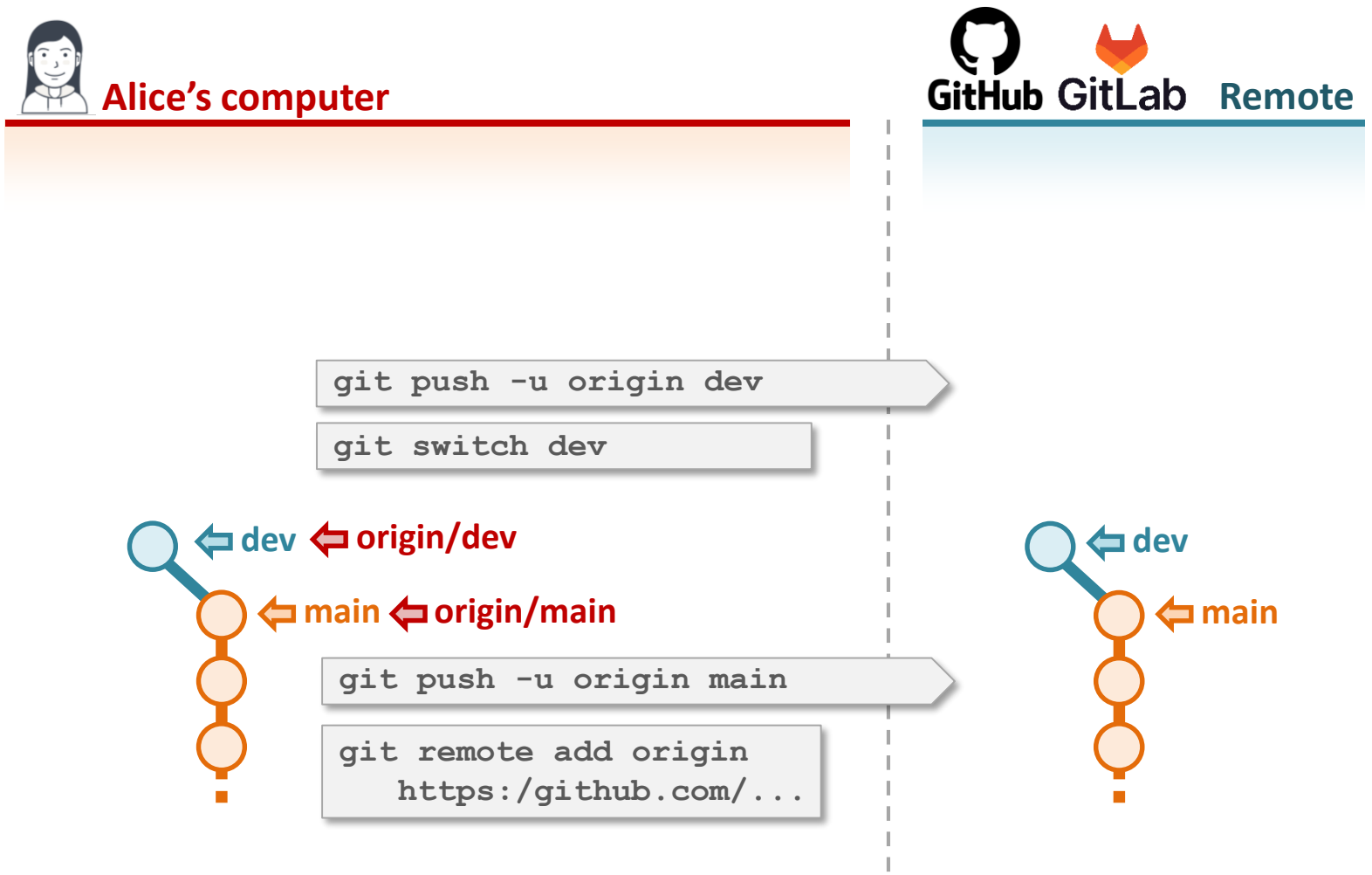


Good to know:

- Each copy of a Git repo (local or online) is a **full copy of the entire repo’s history** (provided it has been synced).
- Git does not perform any automatic sync between the local and remote repos. All **sync operations must be manually triggered**.

Remotes are generally hosted on dedicated servers/services, such as GitHub, GitLab (either gitlab.com or a self-hosted instance), BitBucket, ...

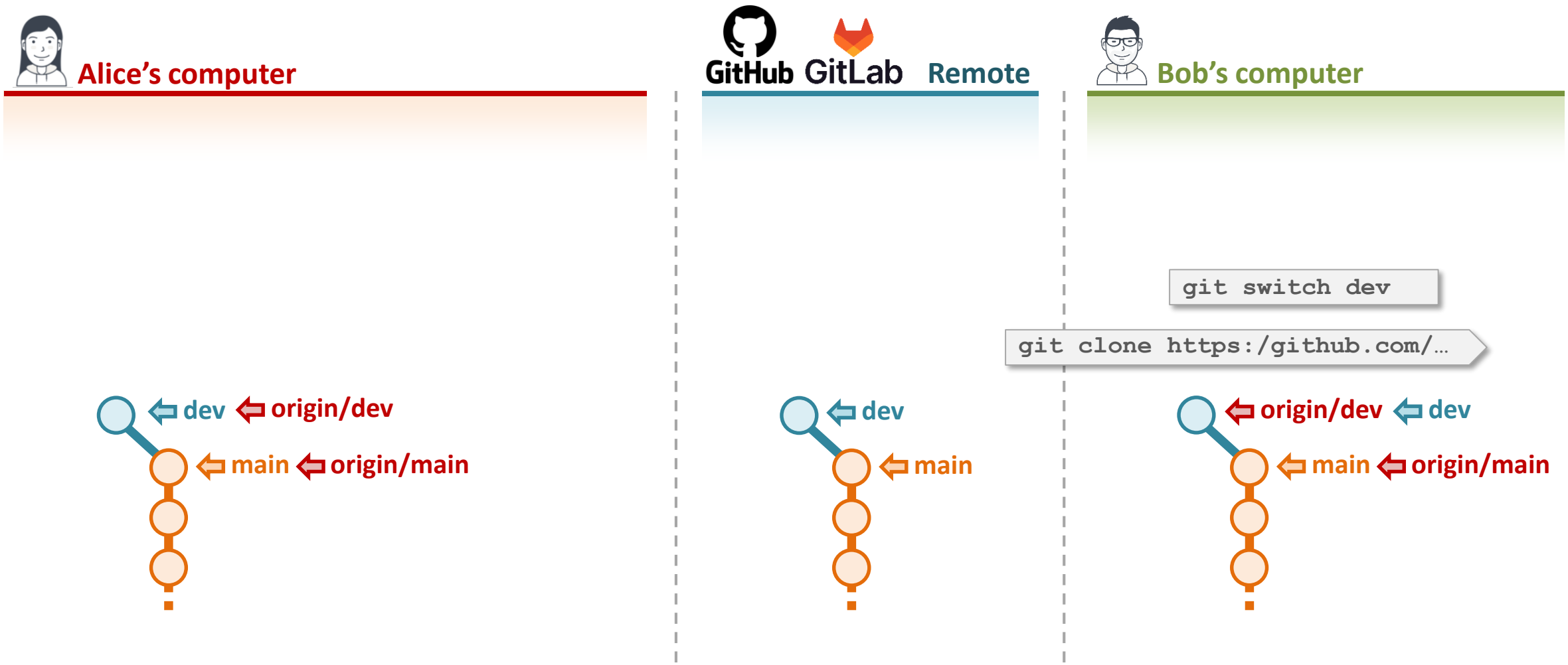
Example – part 1: creating a new remote and pushing new branches



Alice has a Git repo with 2 branches: **main** and **dev**. She now wants to store her work on GitHub, to collaborate and have a backup.

1. She creates a remote on GitHub and links it to her local repo using `git remote add origin <URL of remote>`
2. She pushes her branch **main** to the remote using `git push -u origin <branch name>`
   (at this point the branch has no upstream, so the `-u/--set-upstream` option must be used).
3. She pushes her branch **dev** to the remote.

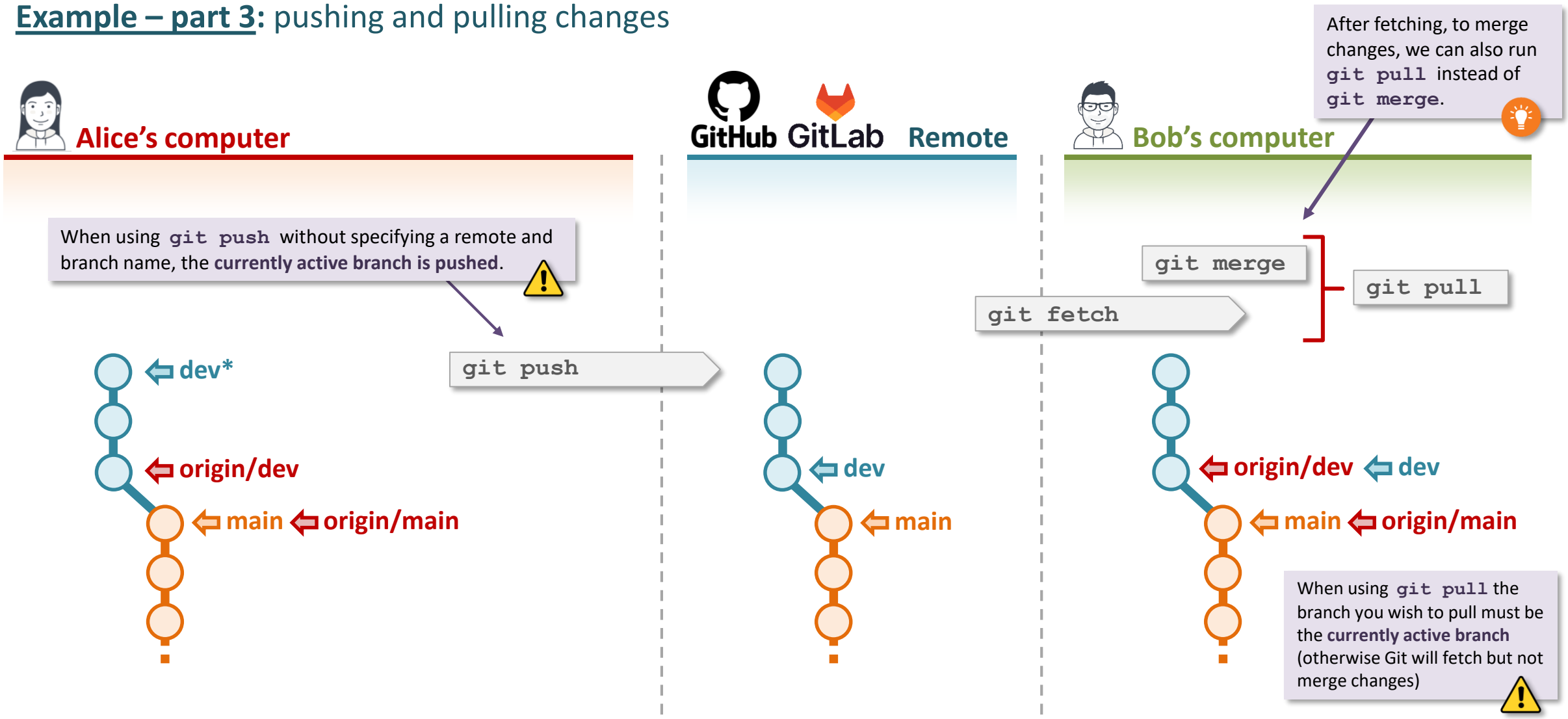
Example – part 2: cloning a remote and checking-out branches



Bob has now joined the team to work with Alice.

1. He **clones** the repo from GitHub using `git clone <URL of remote>`. At this point, Bob has no local **dev** branch - only a pointer to **origin/dev**.
2. Bob checks-out the **dev** branch to work on it. Because there is already a remote branch **origin/dev** present, Git automatically creates a new local branch **dev** with **origin/dev** as upstream (no need add the `--create/-c` option to `git switch`).

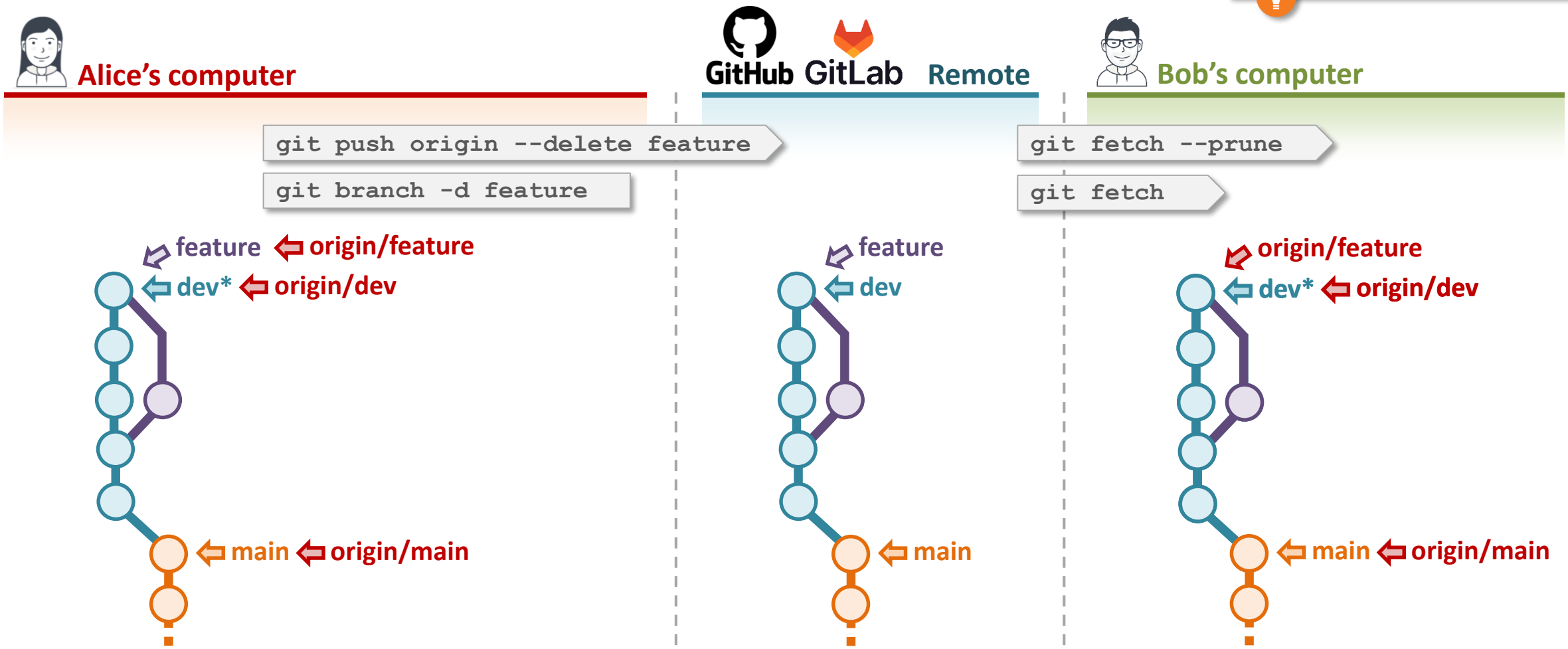
Example – part 3: pushing and pulling changes



1. In the mean time, Alice added 2 new commits to **dev**. She pushes her changes to the remote using `git push` (since her **dev** branch already has an upstream, there is no need to add the `-u/--set-upstream` option this time).
2. To get Alice's updates from the remote, Bob runs `git pull`, which is a combination of `git fetch` + `git merge`.
   Important: `git fetch` downloads all new changes/updates from the remote, but does not update your local branches.

Example – part 4: deleting branches on the remote

The `--prune` option also works with `git pull --prune`.



We are now at a later point in the development... Alice has just completed a new feature on her branch **feature**, and merged it into **dev**. She now wants to delete the **feature** branch both locally and on the remote.

1. Alice deletes her local branch with `git branch -d <branch name>`.
2. Alice deletes the feature branch on the remote with `git push origin --delete <branch name>`. This also deletes her **origin/feature** pointer.
3. Bob runs `git fetch`, but this does not delete references to remote branches, even if they no longer exist on the remote.
4. To delete his local reference to the remote feature branch (**origin/feature**), Bob has to use `git fetch --prune`.

Interacting with remotes: commands summary

Command	What it does
`git clone <URL>`	**Create a local copy** from an existing online repo. Git automatically adds the online repo as a remote.

By convention, the `<remote name>` is generally set to `origin`, but it could be anything.

`git remote add origin <remote url>`	**Add a new remote** to an existing local repo.
`git remote set-url origin <remote url>`	**Change/update the URL of a remote** associated to a local repo.
`git remote -v`	**Display the remote(s)** associated to a repo.

```
$ git remote -v
origin   https://github.com/alice/test-project.git (fetch)
origin   https://github.com/alice/test-project.git (push)
```

The fetch and push URLs should be the same.
To use different URLs (different remotes) for push and fetch, add two different remotes.

`git branch -vva` — **List branches of repo and their associated upstream** (if any).

```
$ git branch -vva
  manta-dev 18d8de0 [origin/manta-dev] manta ray: add animal name
  main      6c8d731 [origin/main] Merge pull request #44 from sibgit/dahu-dev
* sunfish   18d8de0 manta ray: add animal name
```

We can see that the branches **main** and **manta-dev** have an upstream branch. The **sunfish** branch does not.

Supplementary material...

Interacting with remotes: commands summary

Command	What it does	Where to run and comments
`git push`	**push new commits** on the current branch to the remote.	**Run on the branch that you wish to push.** (only changes on the active branch are pushed)
`git push -u origin <branch-name>`	Same as git push, but additionally sets the upstream branch to `origin/branch-name`. Only needed if no upstream is set.	`-u` option is only needed when pushing a branch to the remote for the very first time. It is not needed if you initially created the local branch from a remote branch.
`git push origin <branch-name>`	Push new commits on the specified branch to the remote.	When the remote (here `origin`) and branch names are specified, the push command **can be run from anywhere**.
`git push --force`	**Overwrite the branch on the remote** with the local version.	**Warning:** this **deletes data** on the remote!
`git fetch`	**Download all updates** from the remote to your local repo (even for non-active branches or branches for which there is no local version). Does not update your local branch pointer to `origin/branch-name`.	**Can be run from any branch.**
`git pull`	**Download all updates** and **merge changes** the upstream `origin/branch-name` into the active branch (i.e. update the active branch to its version on the remote).	**Run on the branch that you wish to update.** `git pull` is a shortcut for `git fetch + git merge origin/branch-name`
`git pull --no-rebase`	Fetch + 3-way merge active branch with its upstream `origin/branch-name`.	
`git pull --rebase`	Fetch + rebase active branch on its upstream `origin/branch-name`.	
`git pull --ff-only`	Fetch + fast-forward merge active branch with its upstream `origin/branch-name`. If a fast-forward merge is not possible, an error is generated.	

On recent versions of Git (>= 2.33), the default pull behavior is to abort the pull if a branch and its upstream are diverging.
On older versions, the default behavior is to merge them (same as `git pull --no-rebase`).

Supplementary material...

GitHub / GitLab

collaborate and share your work

GitHub / **GitLab** – an online home for Git repositories

- **GitHub** [github.com] and **GitLab** [gitlab.com] are hosting platforms for Git repositories.
- Very popular to share/distribute software.
- Allows to host public (anybody can access) and private (restricted access) repos.
- Hosting of projects is free, with some paid features.
- Popular alternatives include:
  - **A local instance of GitLab**, the same as GitLab.com but hosted by someone else.
  - **BitBucket** [bitbucket.org].

Project home page on GitHub

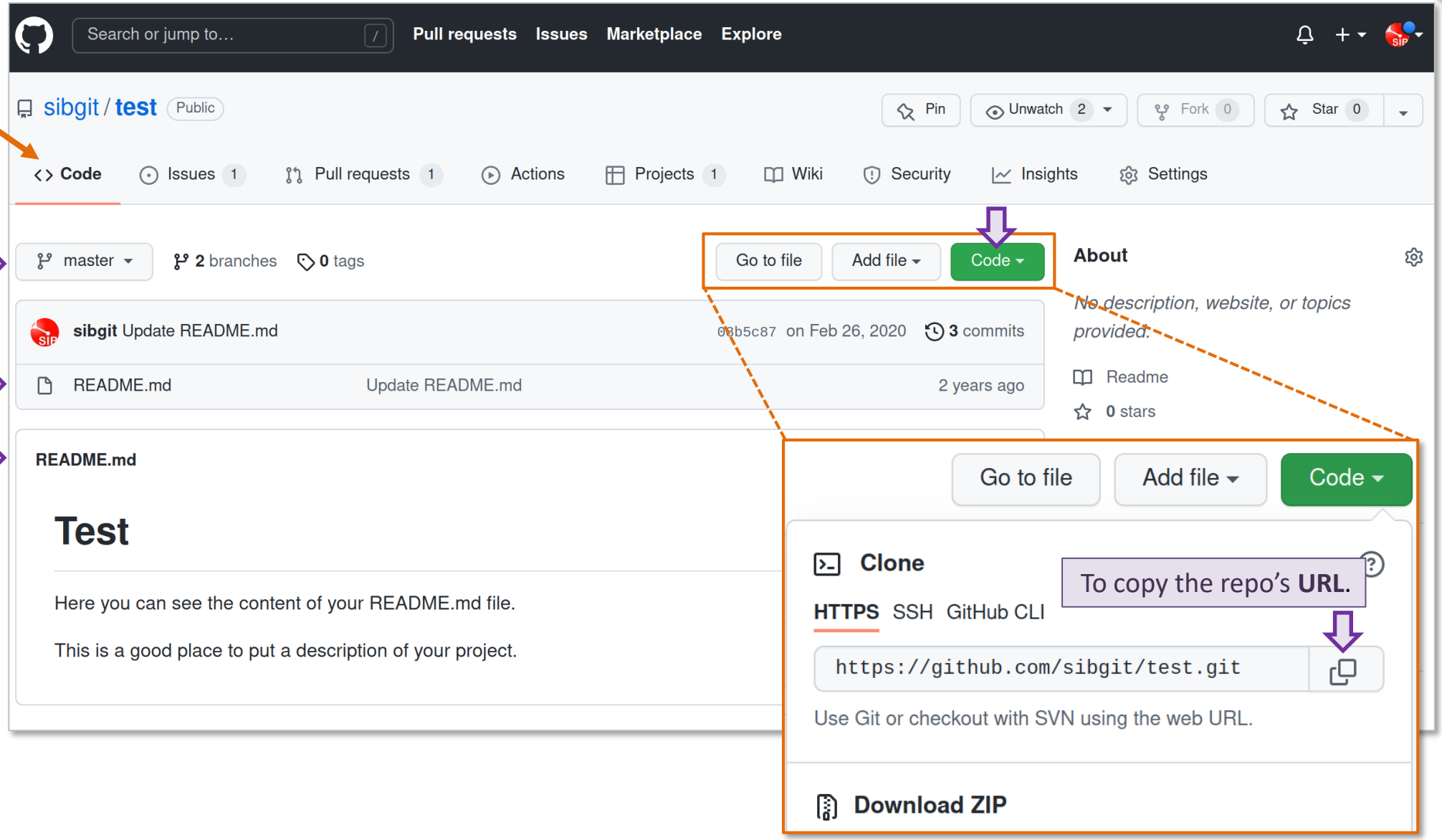
Example of the "**home page**" of a repository on GitHub

Code tab: the "home" page of your repo.

Branch you are currently viewing

List of files present in the repo.

If you have a **README.md** file, it is displayed here (with markdown rendering).



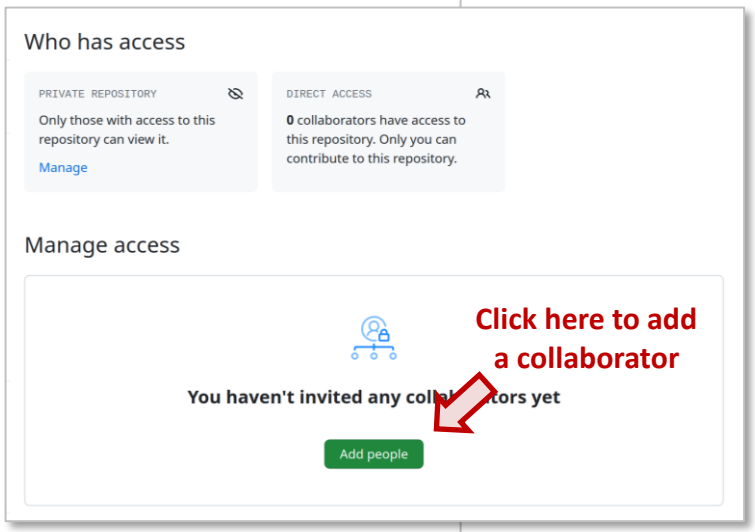
GitHub slide …

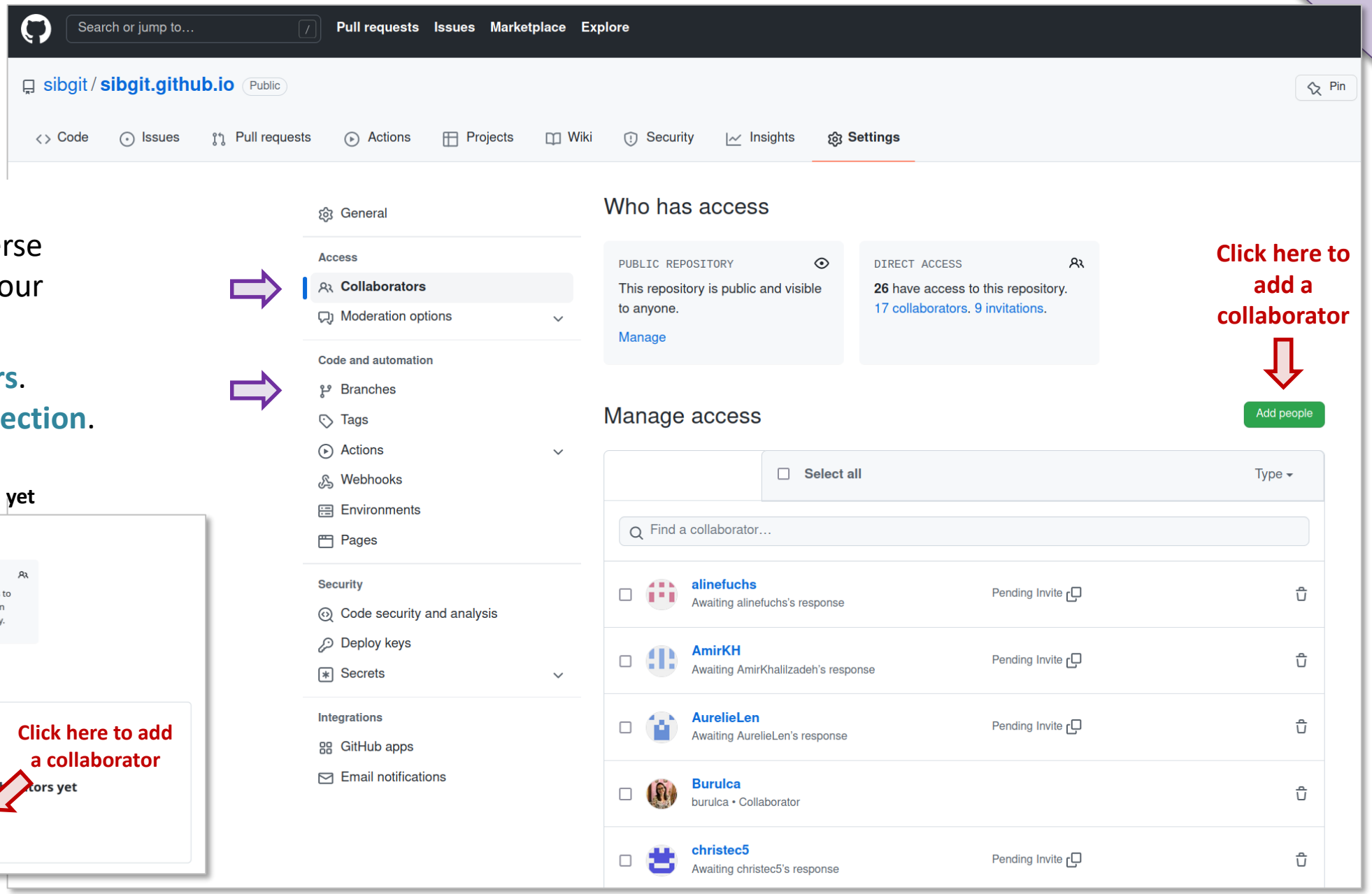
Repository settings (only available to project owner)

Here you can set diverse settings concerning your repository, e.g. :

- Invite **collaborators**.
- Setup **branch protection**.

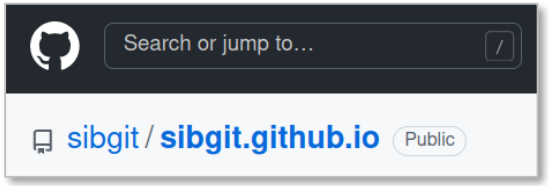
View with no collaborator added yet





GitHub slide …

Other GitHub features (some of them)



"Home" of your repo (repo content)

Issue tracker

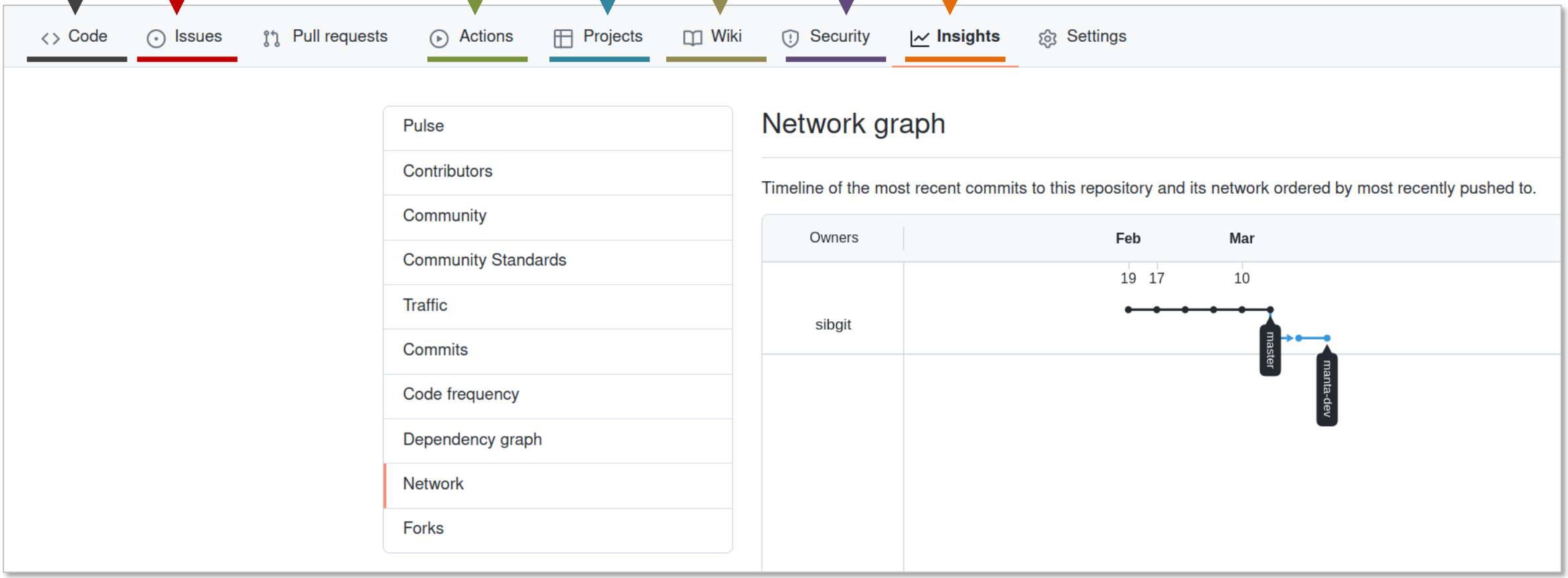
Continuous integration (automated testing)

Group issues and PR by topics.

Add a wiki for your project.

Setup automated security scanning for your code (vulnerability check).

Statistics about your repo's activity.



GitHub slide …

Creating a new repository

on GitHub / GitLab

Repositories on GitHub / GitLab

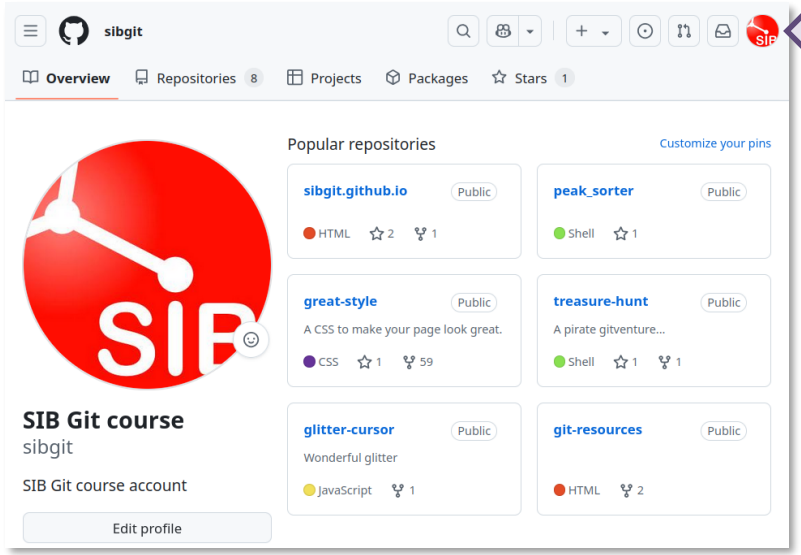
In **exercises 3** you will need to create a new repository on GitHub / GitLab.

Let's create it now together…

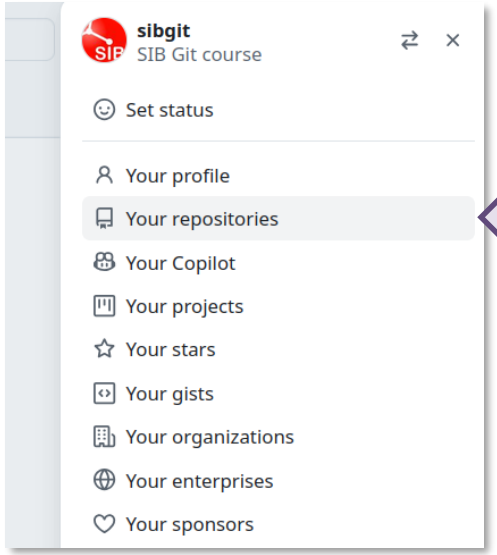
Creating a new repository on GitHub (1/3)

1. Sign-in to your user account on https://github.com.

2. Click on your **user icon** (top right), and select **Your repositories**.

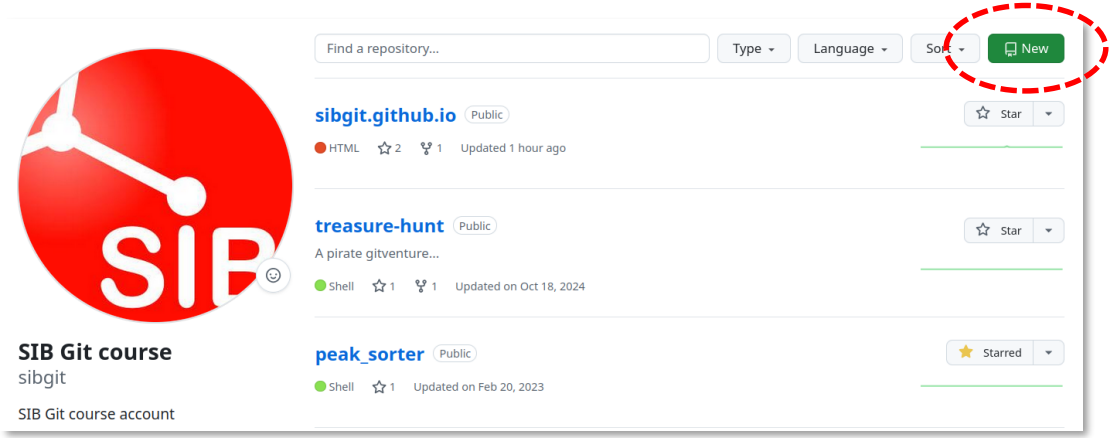


User icon



Your repositories

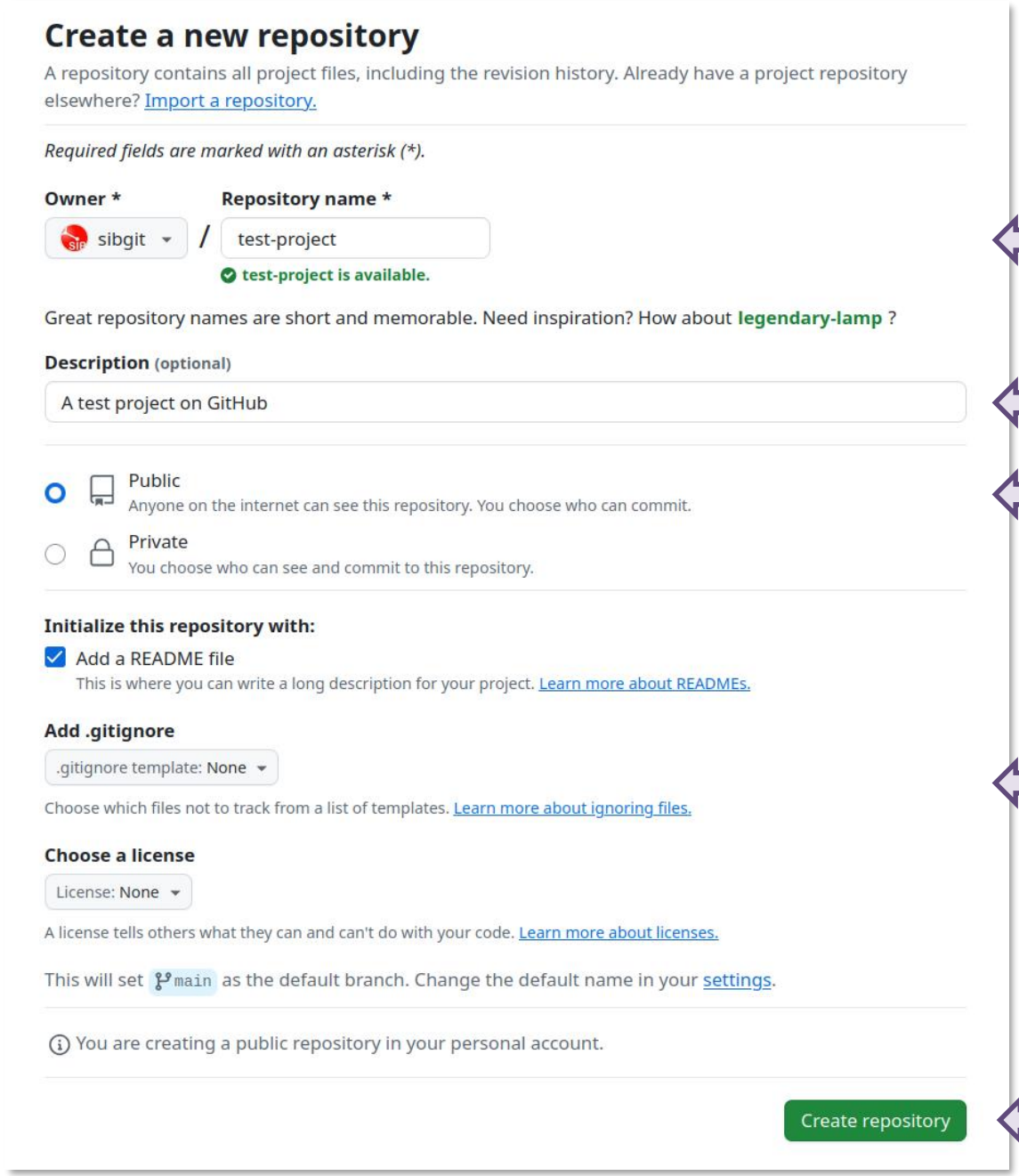
3. To create a new repo, click on 



Create new repo.

Go to next page

Creating a new repository on GitHub (2/3)



⇦ Enter a **name** for your new repository.

⇦ Add a project **description** (optional).

⇦ Select whether your repo should be:

- **Public** - anyone can access it (read from it).
- **Private** - only people you authorize can read from it.

Note: even if a repo is public, only authorized members can push changes to it.

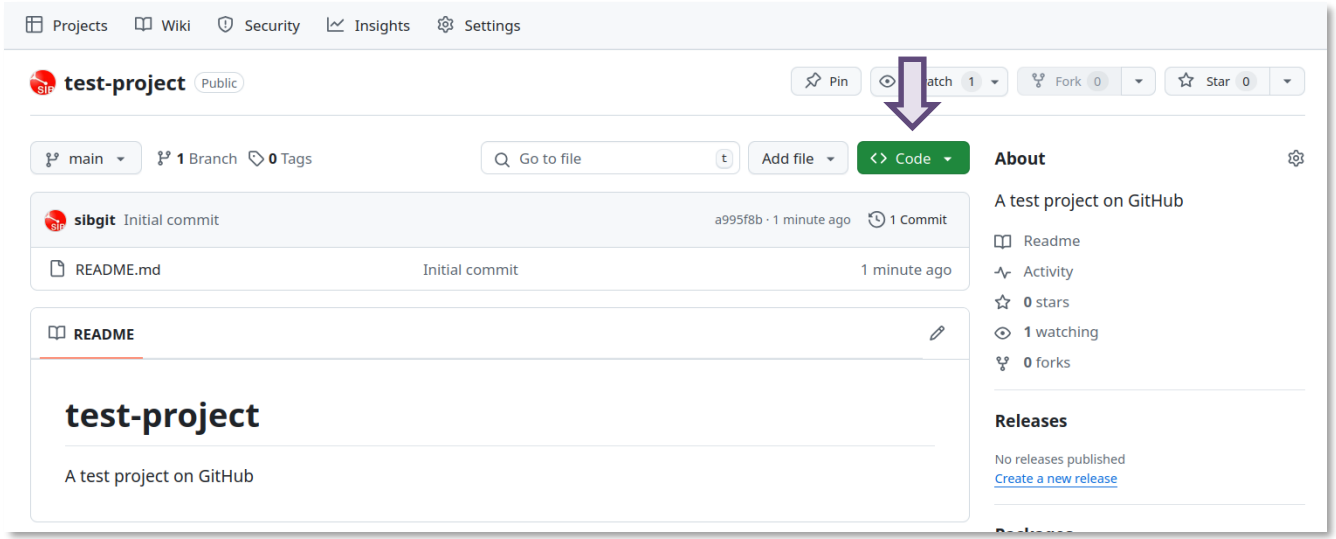
⇦ **Pre-fill the repository** with some files (optional).

- Don’t do this if you already have a local repo you want to push.
- In this example, a "README.md" file will be added to the newly created repo. Adding a file allows you to clone the repo immediately.

⇦ Click **Create repository**.

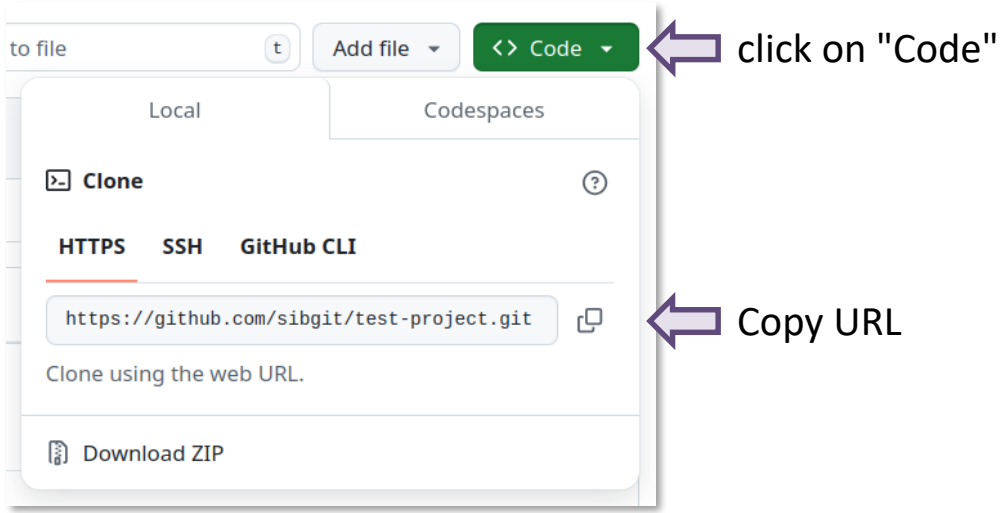
Go to next page →

Creating a new repository on GitHub (3/3)



Done: your new repository is setup and ready to be used.

To **retrieve the URL** of your repository (e.g. to **clone** it):

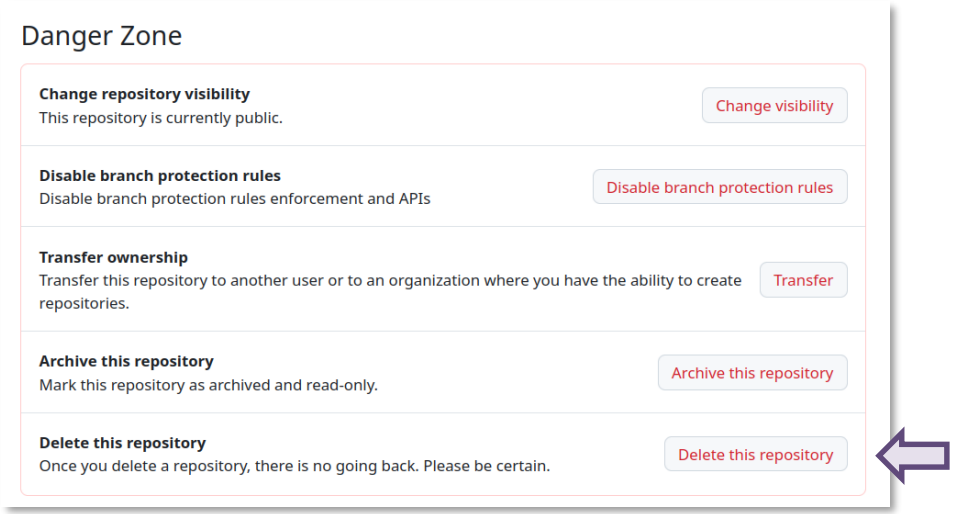


click on "Code"

Copy URL

To permanently **delete a repository**:

Settings > General (scroll to bottom) > Danger zone > Delete this repository



GitHub slide ...

Personal Access Tokens (PAT)

on GitHub / GitLab

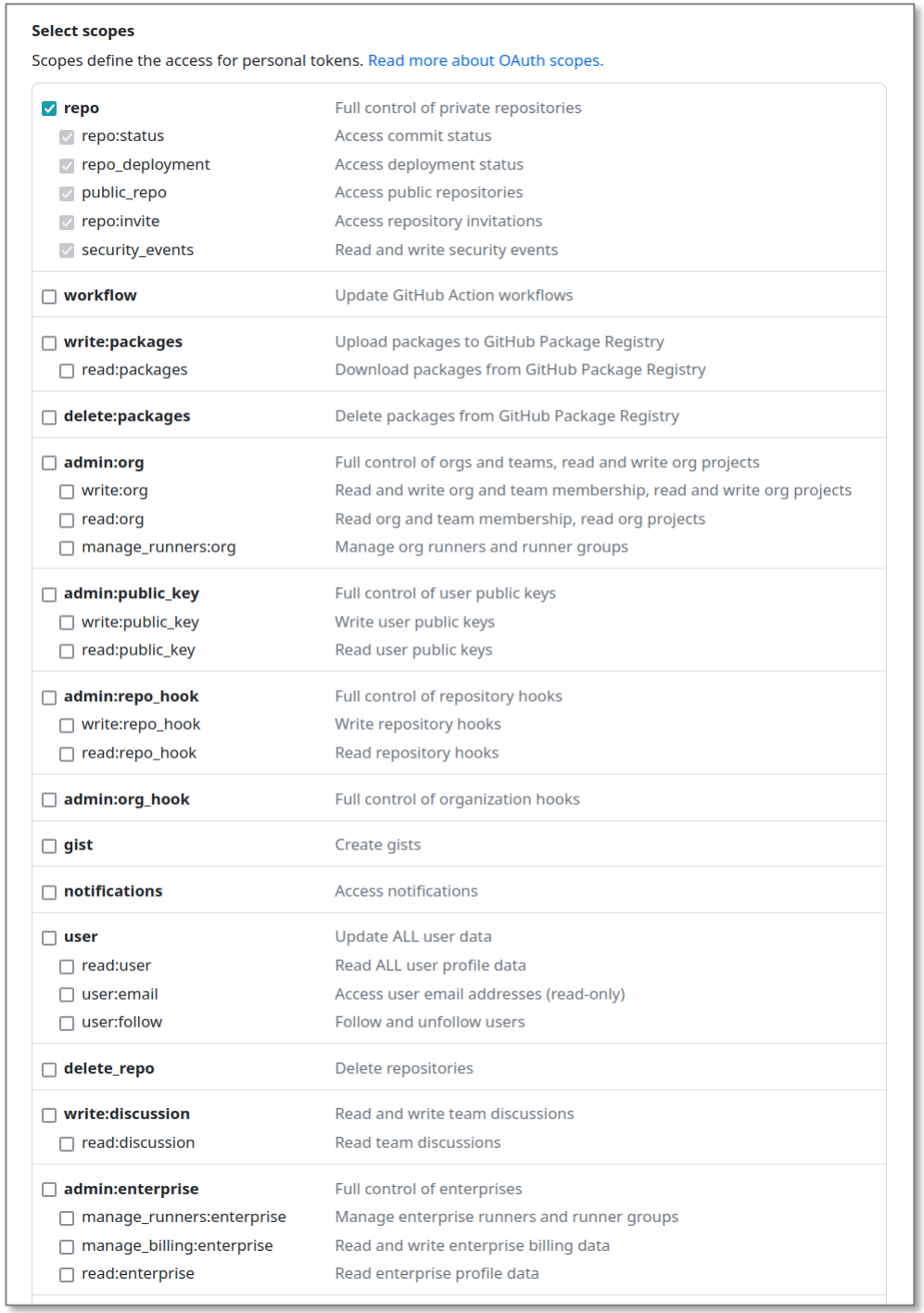
Personal access tokens (PAT) on GitHub/GitLab

Pushing data to a remote requires **some form of authentication**…
… otherwise anyone could push anything to your remotes!

For security reasons, GitHub does not allow using your user name and password for authentication when running a git push command. Instead you need to use a **personal access token (PAT)**.

In **exercises 3** and **4** you will need a PAT to push commits to GitHub/GitLab *

Let's generate a PAT together now…

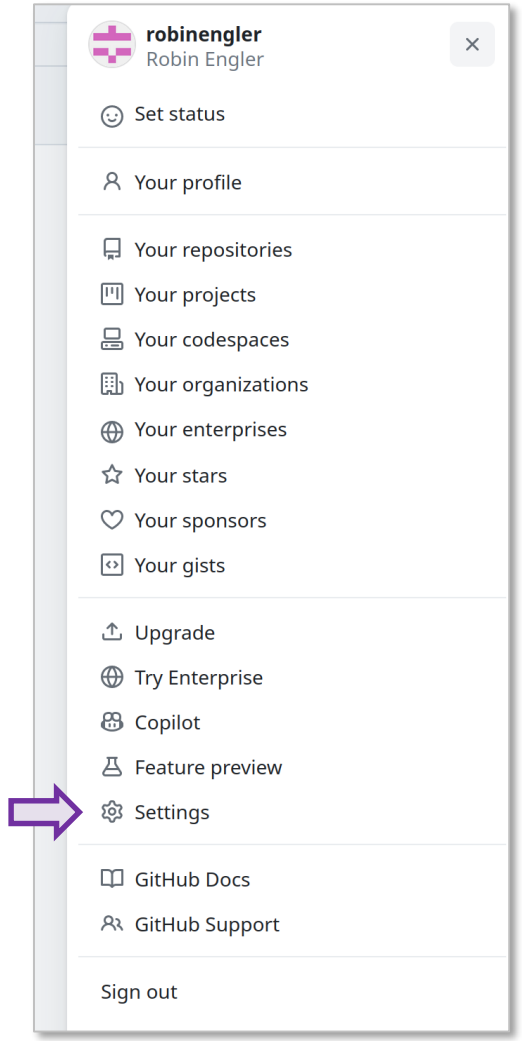


* Alternatively, you can also authenticate to GitHub/GitLab using SSH keys. If your account is already setup to use SSH keys, then you don't need a PAT.

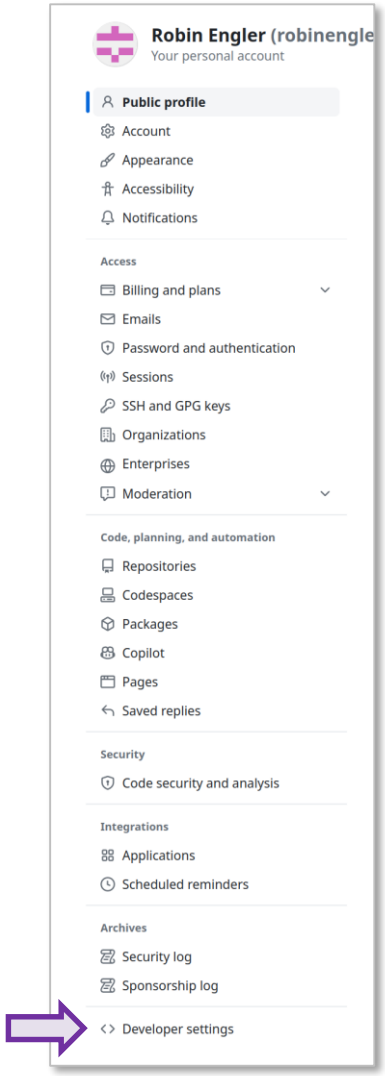
Generating a “personal access token” (PAT) on GitHub

In order to push data (commits) to GitHub, you will need a **personal access token (PAT)**.

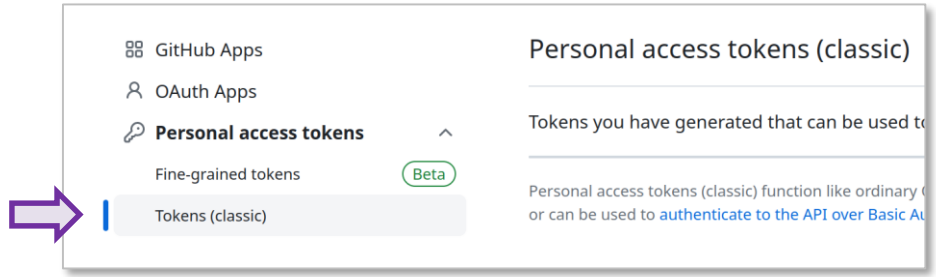
1. In your user profile (top right), click on **Settings**.



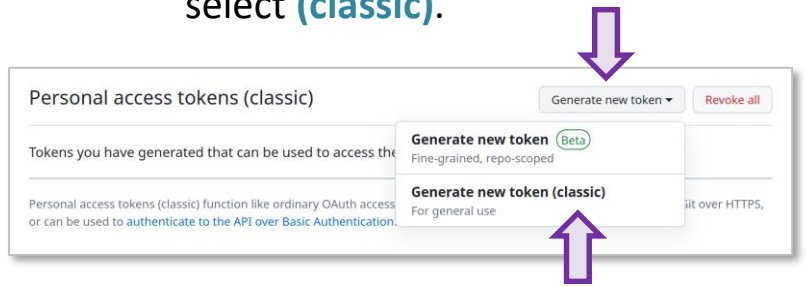
2. In your Account settings, click on **Developer settings** (at the very bottom of the list)



3. In **Developer settings**, click on **Personal access tokens**, and select **Tokens (classic)**.



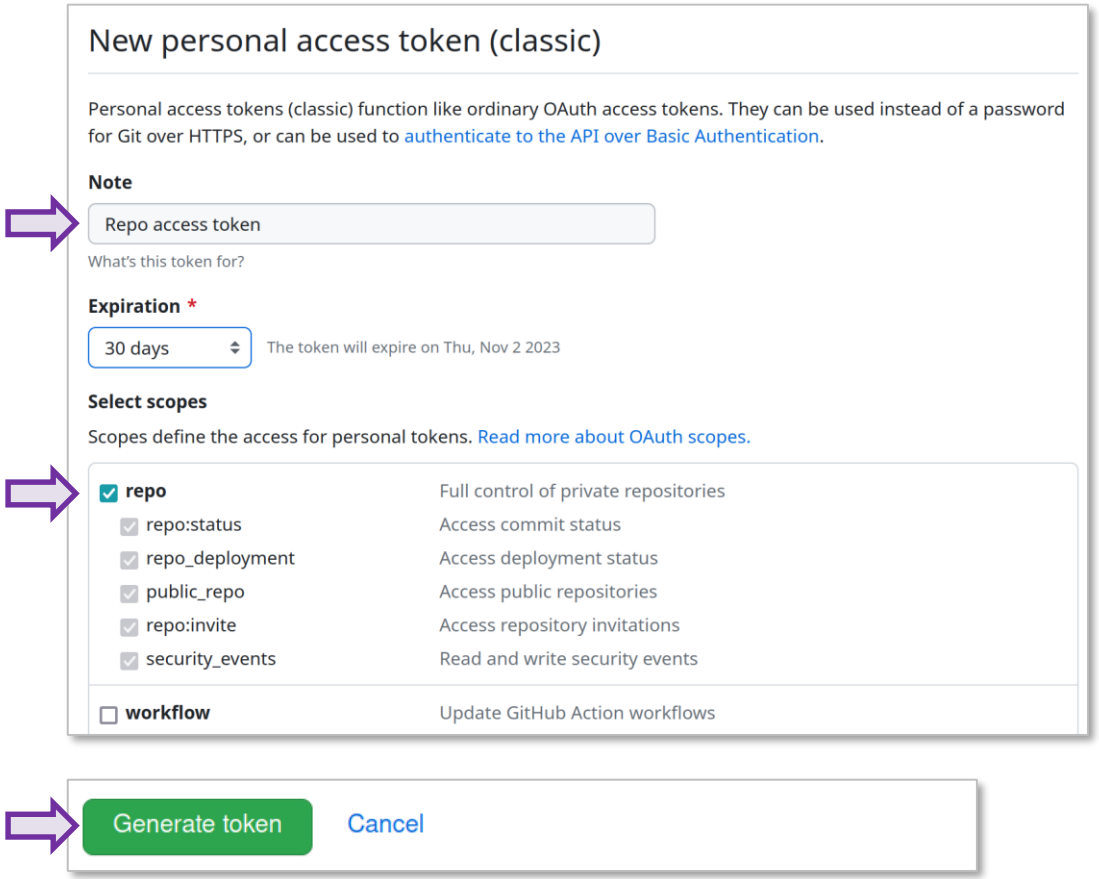
4. Click on **Generate new token**, and select **(classic)**.



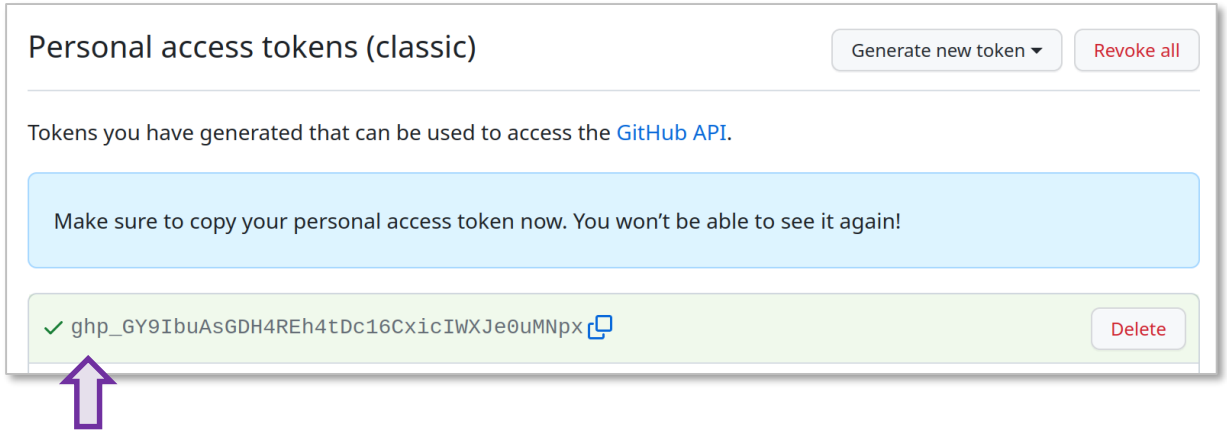
Go to next page

GitHub slide ...

5. Add a **Note** (description) to your token and select the **repo** scope checkbox. The click **Generate token**.



6. **Copy the personal access token** to a safe locations (ideally in a password manager). You will **not be able to access it again** later.



7. When you will push content to GitHub for the first time in the project, you will be asked for your user name and password. **Instead of the password**, enter the **personal access token** you just created.

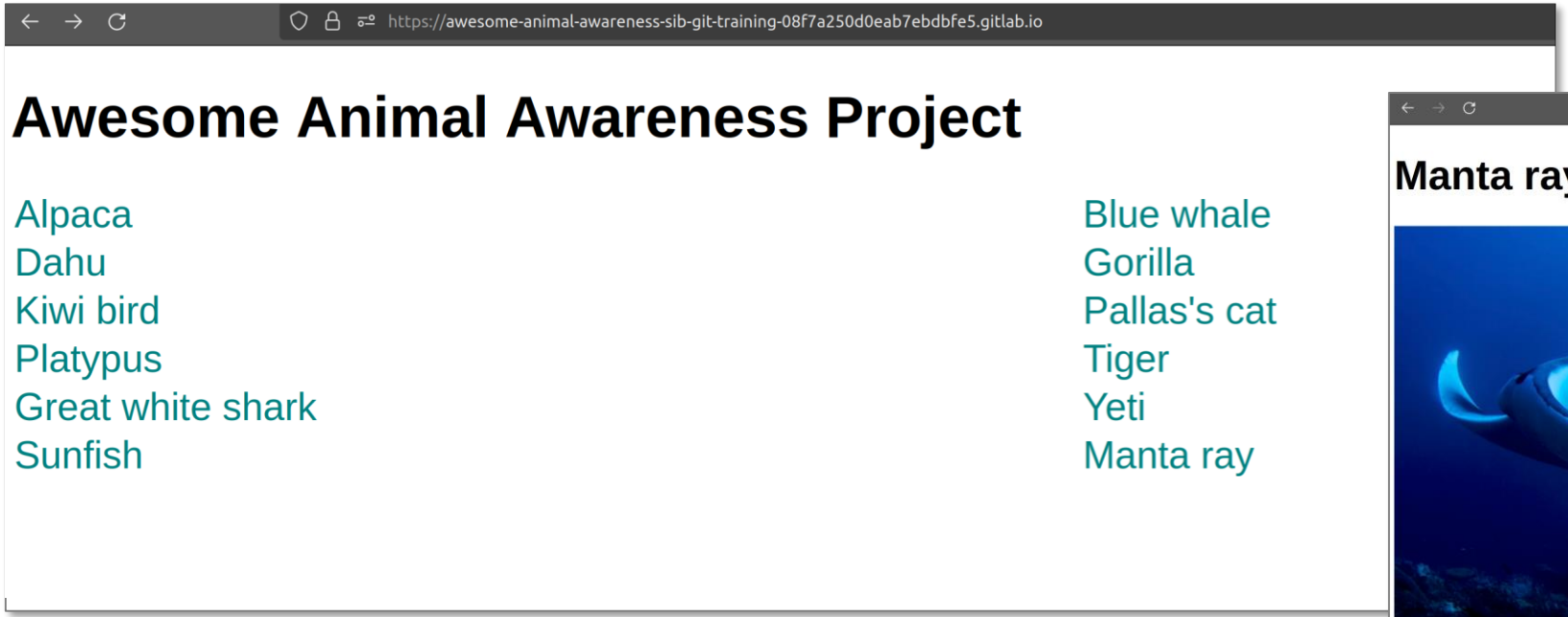
exercise 3

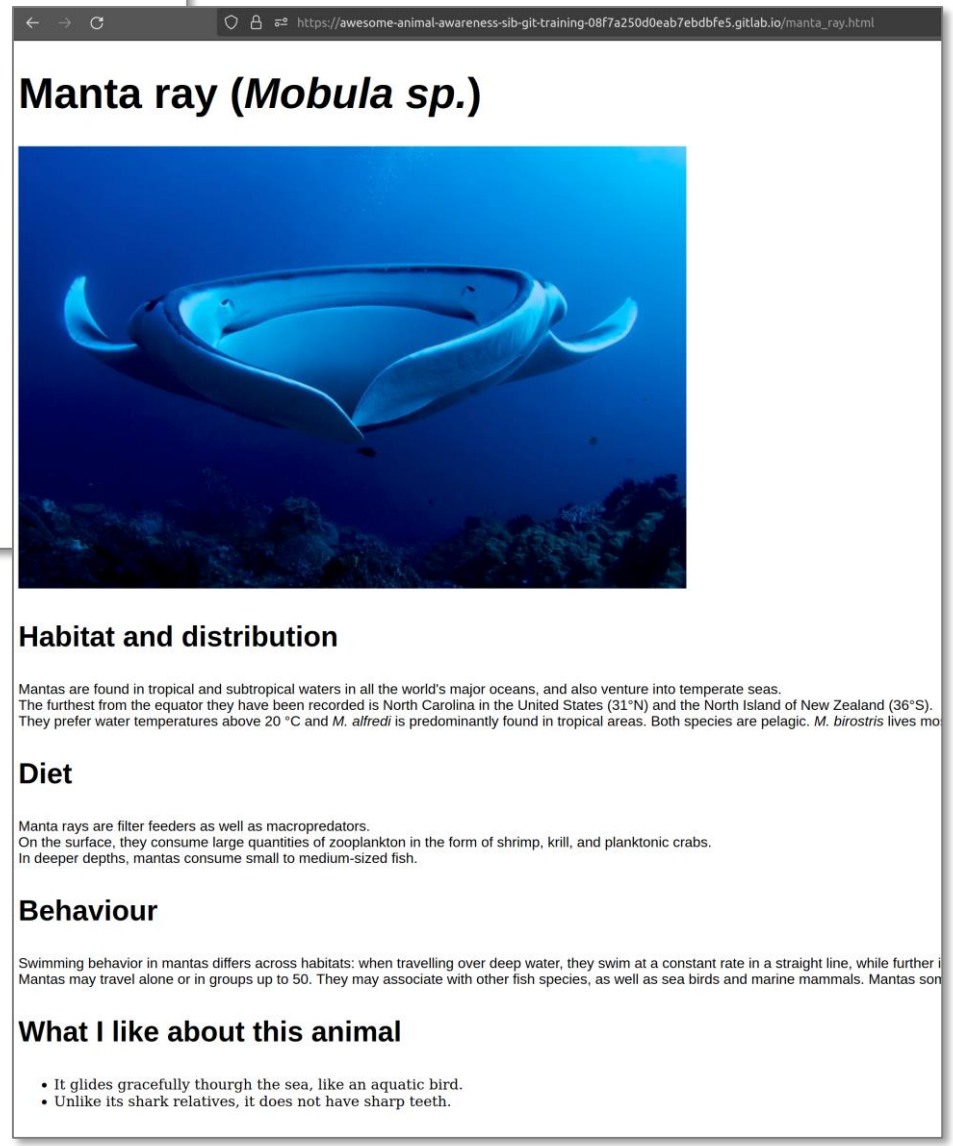
The markdown cheat-sheet

Pull Requests (GitHub) and **Merge Requests** (GitLab)

An introduction to the upcoming **exercise 4**…

In exercise 4, we will all work together on building a website for the **Awesome Animal Awareness project**!





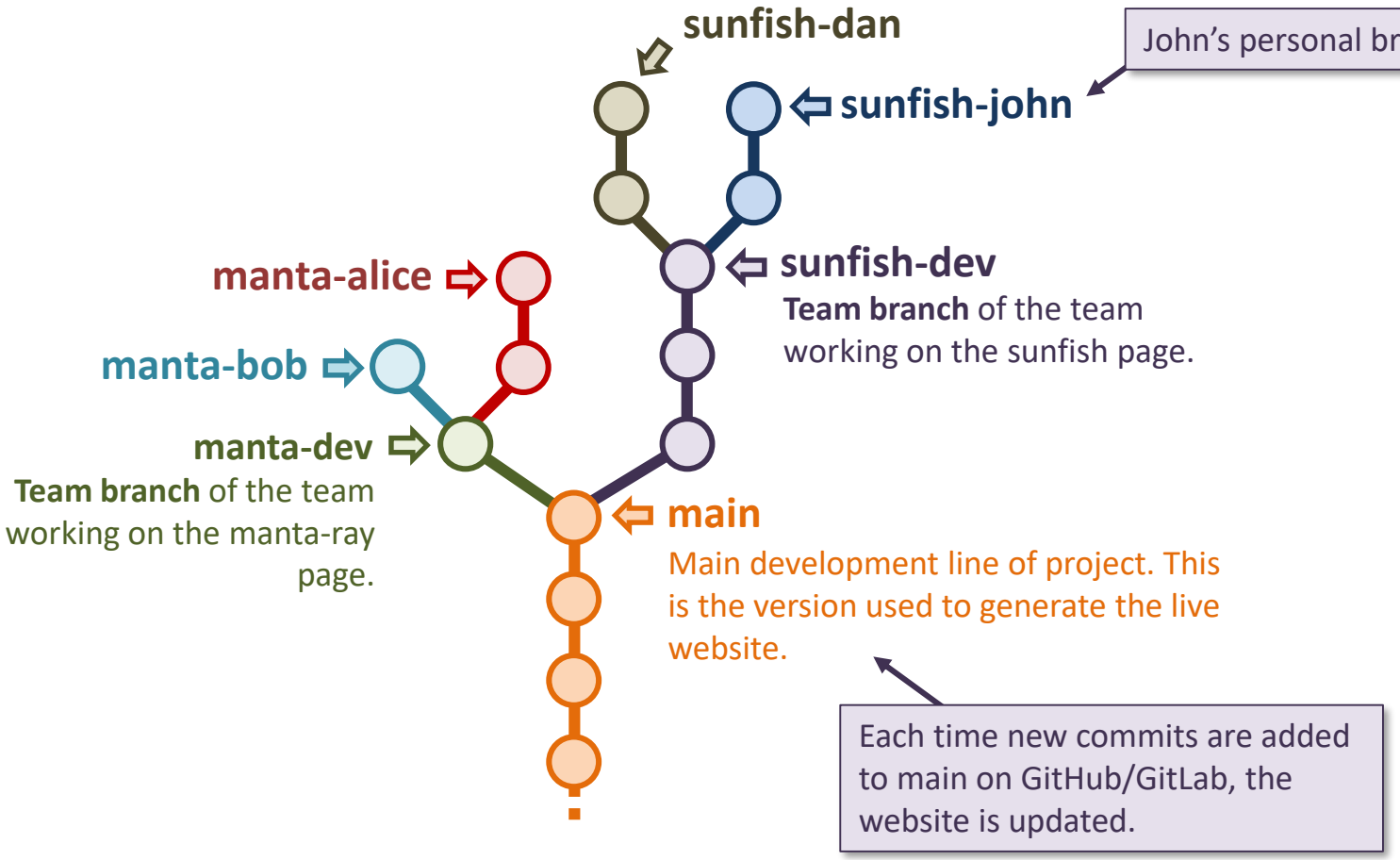
How we will work:

- We will split into teams of 2-3 people.
- Each team will be responsible for creating the page of an (awesome!) animal *.
- Within a team, each person will work on a different part of the animal's page (e.g. one person works on the "Habitat and distribution" section, while another works on the "Diet" or "Behavior").

* Note: every animal in the list is awesome – you can't go wrong!

An introduction to the upcoming **exercise 4**…

- This is how (more or less) our shared repository will look on GitHub/GitLab…
- Changes made to the **main** branch are directly reflected in the production website – so we don't want to mess-up **main** !!
- => You are **not allowed to push directly to main**.



How are we going to contribute changes from our team branches ?

Pull Requests (GitHub) / Merge Requests (GitLab)

Pull Requests (PR) and **Merge Requests (MR)** are a way to perform a merge operation on the remote (on GitHub/GitLab) instead of in your local copy of the repository.

PR/MR are the same thing, they just have different names on GitHub/GitLab.

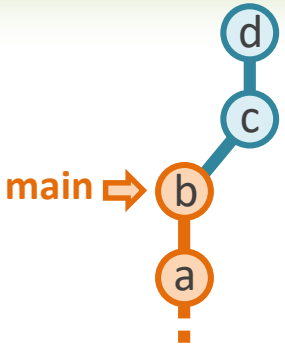
Why use a PR/MR instead of a local merge (and push) ?

- The branch you want to merge into (e.g. main) is **protected** *.
- Gives the opportunity to the repository owner(s) to **review changes** before merging them.
- Makes it easy to merge changes from a **forked** ** repository.

* **Protected** branches are branches where push operations are limited to users with enough privileges.

** A **fork** is a copy of an entire repository under a new ownership.

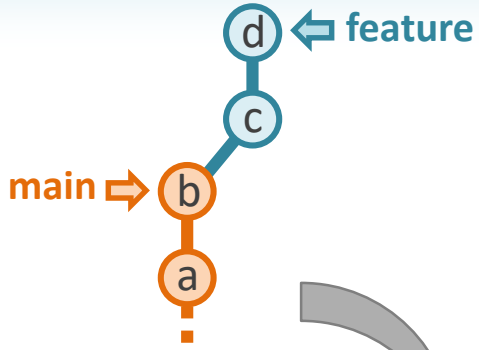
Bob's computer



`git push`

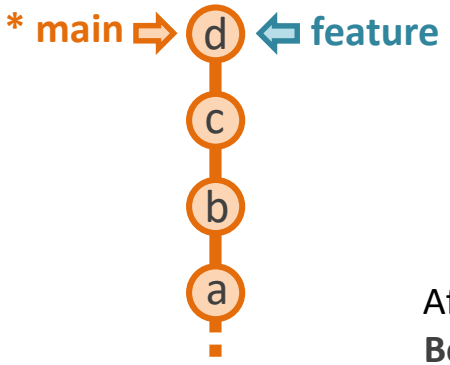
Bob has completed his work on a new feature. He **pushes his changes to the remote**.

GitHub GitLab **Remote**



The PR/MR workflow:

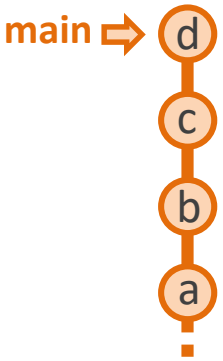
- Bob opens a PR/MR on GitHub/GitLab.
- Alice reviews the changes made by Bob on branch **feature**.
- Alice approves the PR/MR.
- Bob (or Alice) merges the PR/MR.
- On the remote, the **feature** branch is now merged into **main**. Optionally, **feature** is then deleted.



`git fetch`

`git switch main`
`git pull`

After the PR/MR is merged, **Bob updates his local repo**.



After the PR/MR is merged, you can pull the changes from the remote to update your local repo (at this point the merge is only on the remote).

```
❯ git log --all --decorate --oneline --graph
* 9afd6e4 (HEAD -> manta-dev, origin/manta-dev) manta-ray: add behavior information
* ba11531 manta-ray: add distribution and image
* 7b0516b manta-ray: add animal name and diet
* 65cb84f (origin/main, origin/HEAD, main) cicd: add .gitlab-ci.yml file
* 90c5dbe web: rename home page to Awesome Animal Awareness Project
* 4401bd7 web: add animal page templates
* 11a7390 styles: change paragraphs fonts
* 30b07cf styles: add styles.css file
* f1828a1 doc: add README.md
* 998ea08 first commit
```

Using **`git fetch`** is optional, it's useful if you want to preview the position of origin/main before merging it into your local **`main`** with **`git pull`**.

`git fetch --prune`

`--prune` deletes local references to remote branches (**`origin/manta-dev`** has been deleted).

```
❯ git log --all --decorate --oneline --graph
*   13c625f (origin/main, origin/HEAD) Merge branch 'manta-dev' into 'main'
|\
| * 9afd6e4 (HEAD -> manta-dev) manta-ray: add behavior information
| * ba11531 manta-ray: add distribution and image
| * 7b0516b manta-ray: add animal name and diet
|/
* 65cb84f (main) cicd: add .gitlab-ci.yml file
* 90c5dbe web: rename home page to Awesome Animal Awareness Project
* 4401bd7 web: add animal page templates
* 11a7390 styles: change paragraphs fonts
```

```
git switch main
git pull --prune
git branch -d manta-dev
```

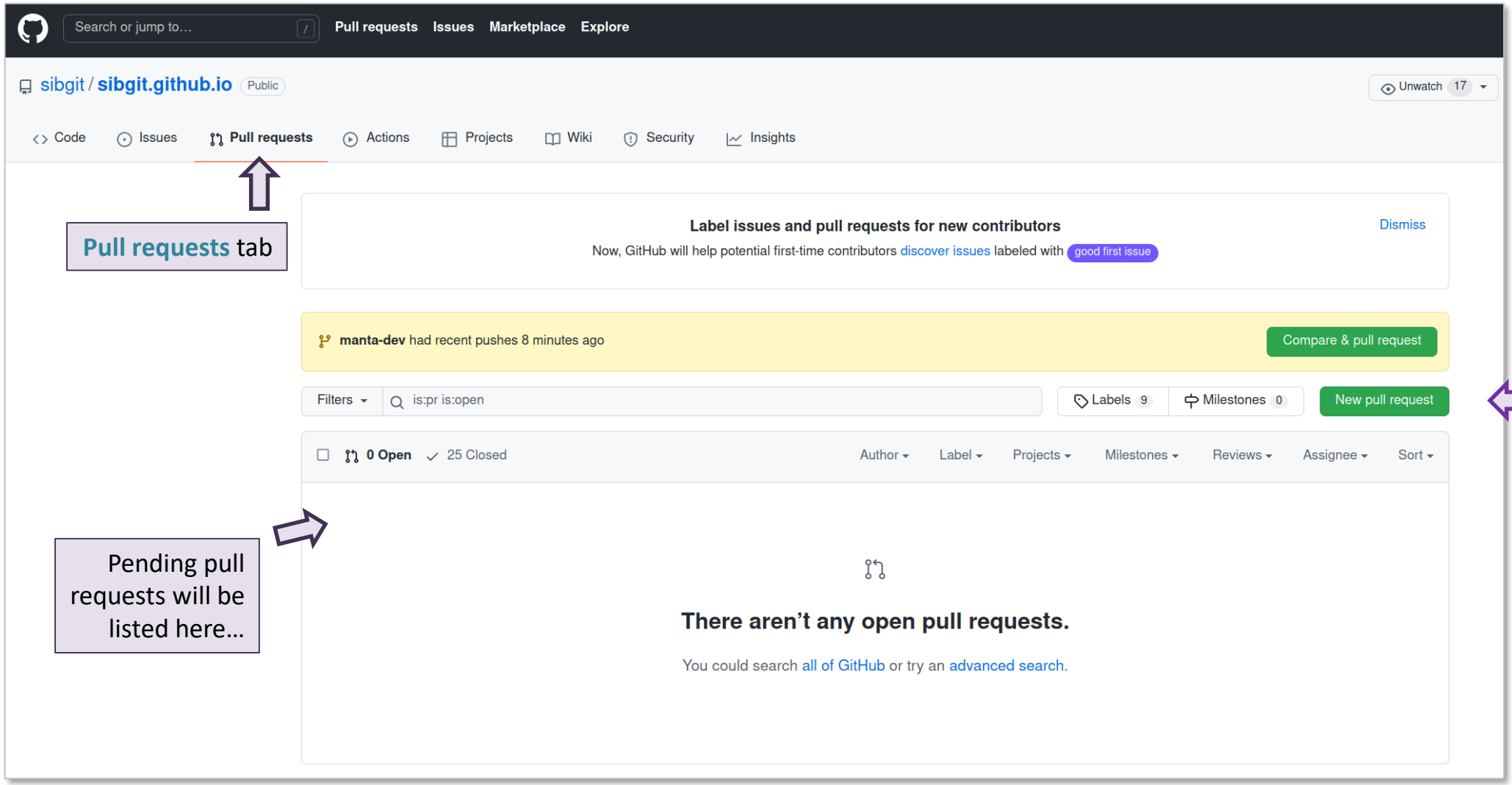
```
❯ git log --all --decorate --oneline --graph
*   13c625f (HEAD -> main, origin/main, origin/HEAD) Merge branch 'manta-dev'
|\
| * 9afd6e4 manta-ray: add behavior information
| * ba11531 manta-ray: add distribution and image
| * 7b0516b manta-ray: add animal name and diet
|/
* 65cb84f cicd: add .gitlab-ci.yml file
* 90c5dbe web: rename home page to Awesome Animal Awareness Project
* 4401bd7 web: add animal page templates
```

```
git switch main
git pull
git branch -d manta-dev
```

Open a Pull Request on GitHub: step-by-step

You will need to do this in exercise 4 !

1. On the project's page on GitHub, go to the **Pull requests** tab.



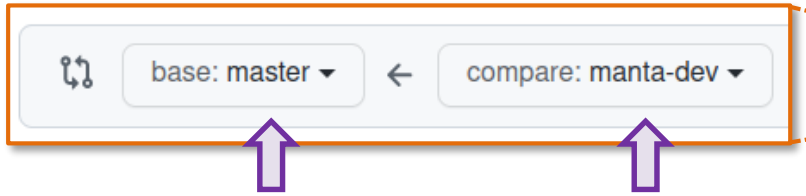
Pull requests tab

Pending pull requests will be listed here…

2. Click on **New pull request**.

GitHub slide …

3. Select the branches to merge:



Branch to merge into

Branch to merge (your contribution)

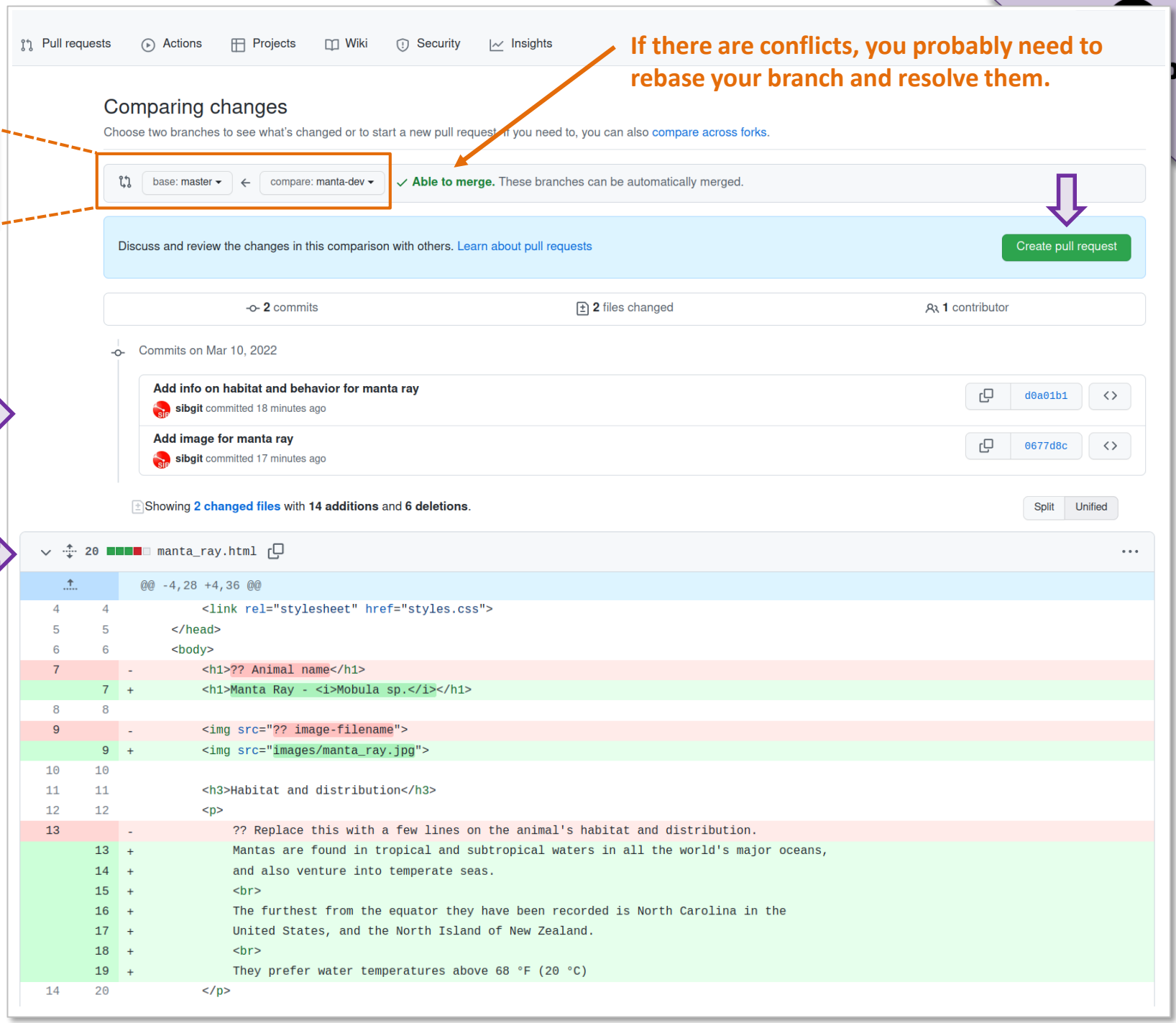
List of commits that will be merged

In this example, there are 2 commits on branch “manta-dev” that will be merged into “master”.

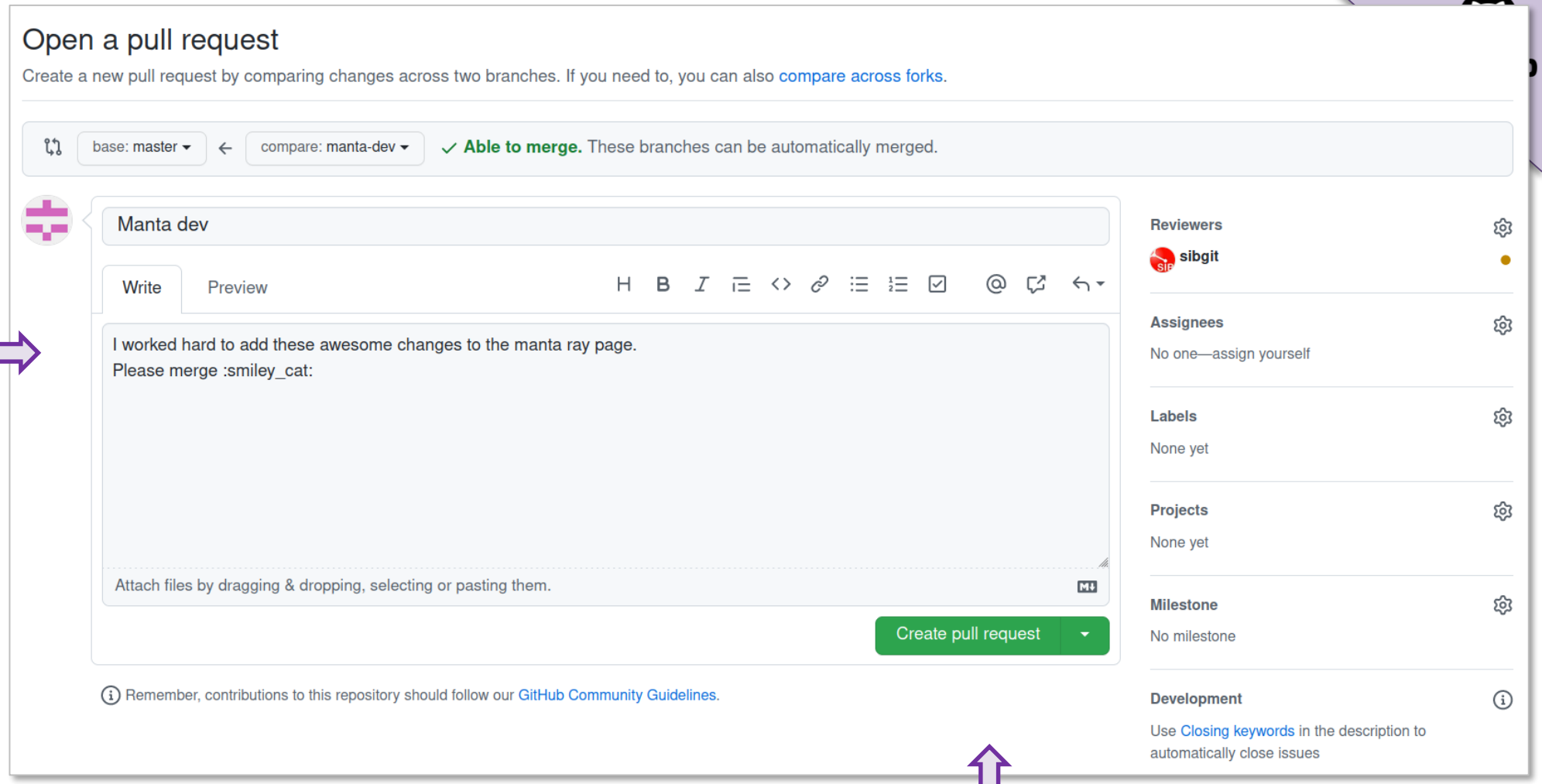
Summary of changes introduced by the pull request.

Green lines = new content.
Red lines = deleted content.

4. Click on Create pull request.



5. Optionally, enter a **message** for the people that will review your pull request.



6. Submit your pull request by clicking **Create pull request**.

The pull request is now **created**, and **awaiting approval** from an authorized person.
(e.g. the repo owner or a colleague)

Merging is **blocked**, because someone has to approve your PR.

Pull requests 1 | Actions | Projects | Wiki | Security | Insights

Manta dev #27

Open **robinengler** wants to merge 2 commits into `master` from `manta-dev`

Conversation 0 | Commits 2 | Checks 0 | Files changed 2

robinengler commented now — Collaborator

I worked hard to add these awesome changes to the manta ray page.
Please merge 😸

sibgit added 2 commits 31 minutes ago

`Add info on habitat and behavior for manta ray` `d0a01b1`

`Add image for manta ray` `0677d8c`

Add more commits by pushing to the `manta-dev` branch on **sibgit/sibgit.github.io**.

This branch has not been deployed
No deployments

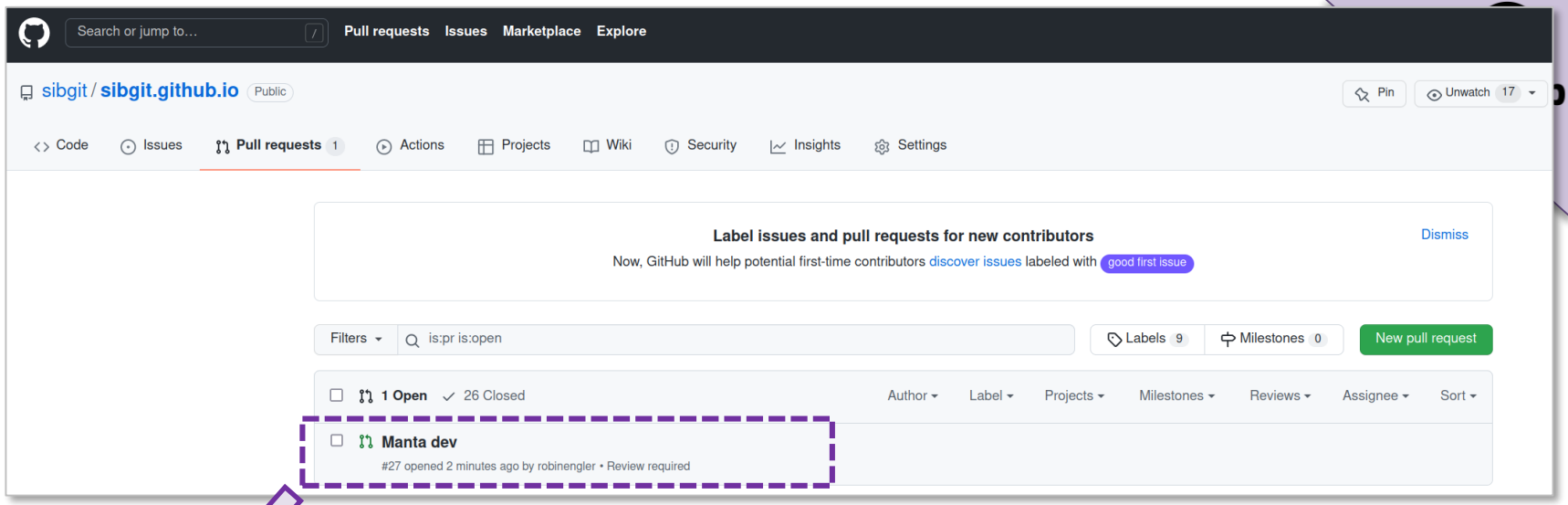
Review required
At least 1 approving review is required by reviewers with write access. Learn more.

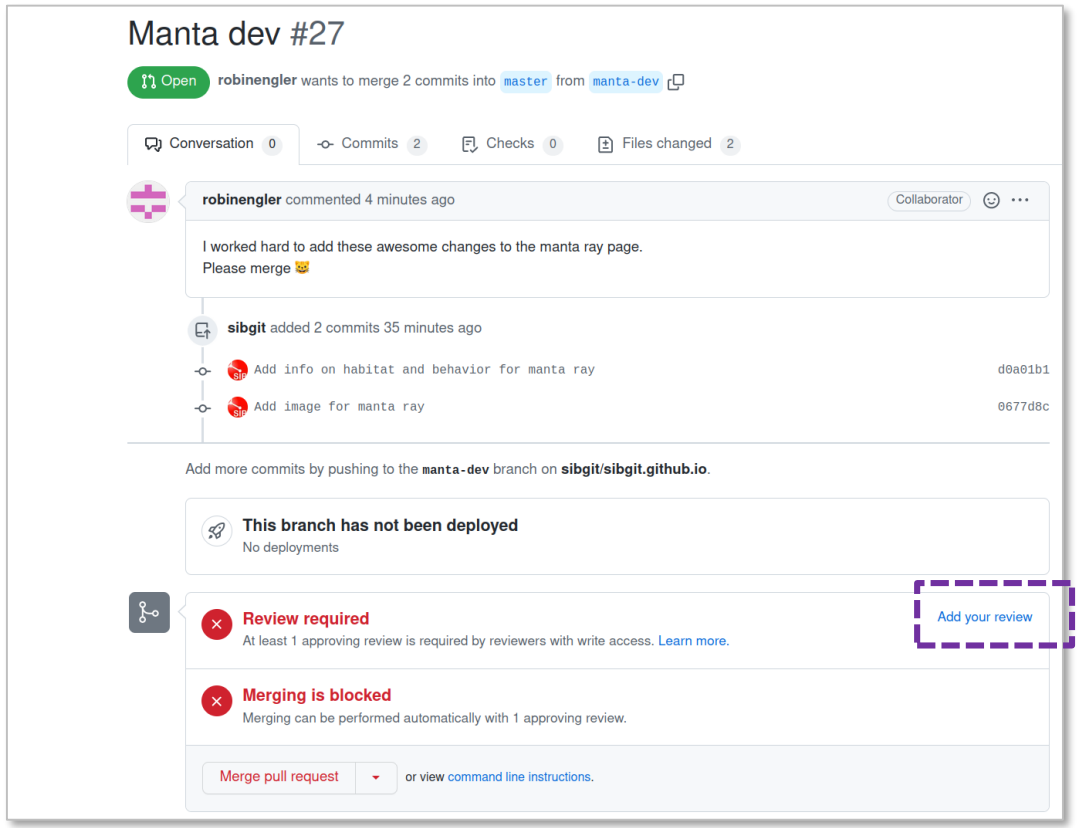
Merging is blocked
Merging can be performed automatically with 1 approving review.

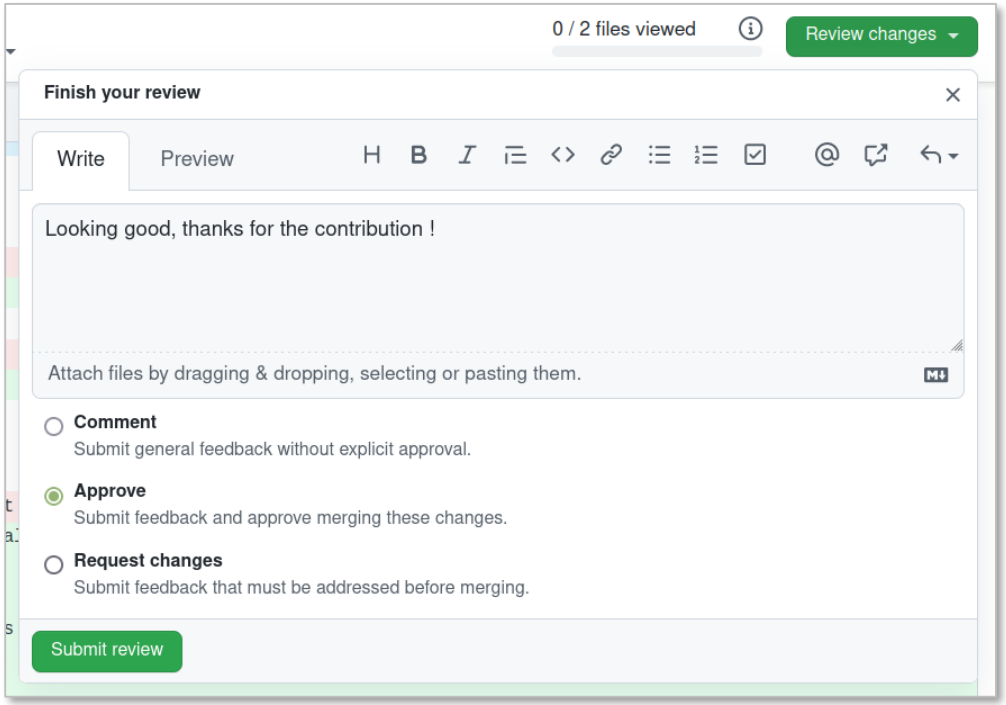
Merge pull request | or view command line instructions.

GitHub slide ...

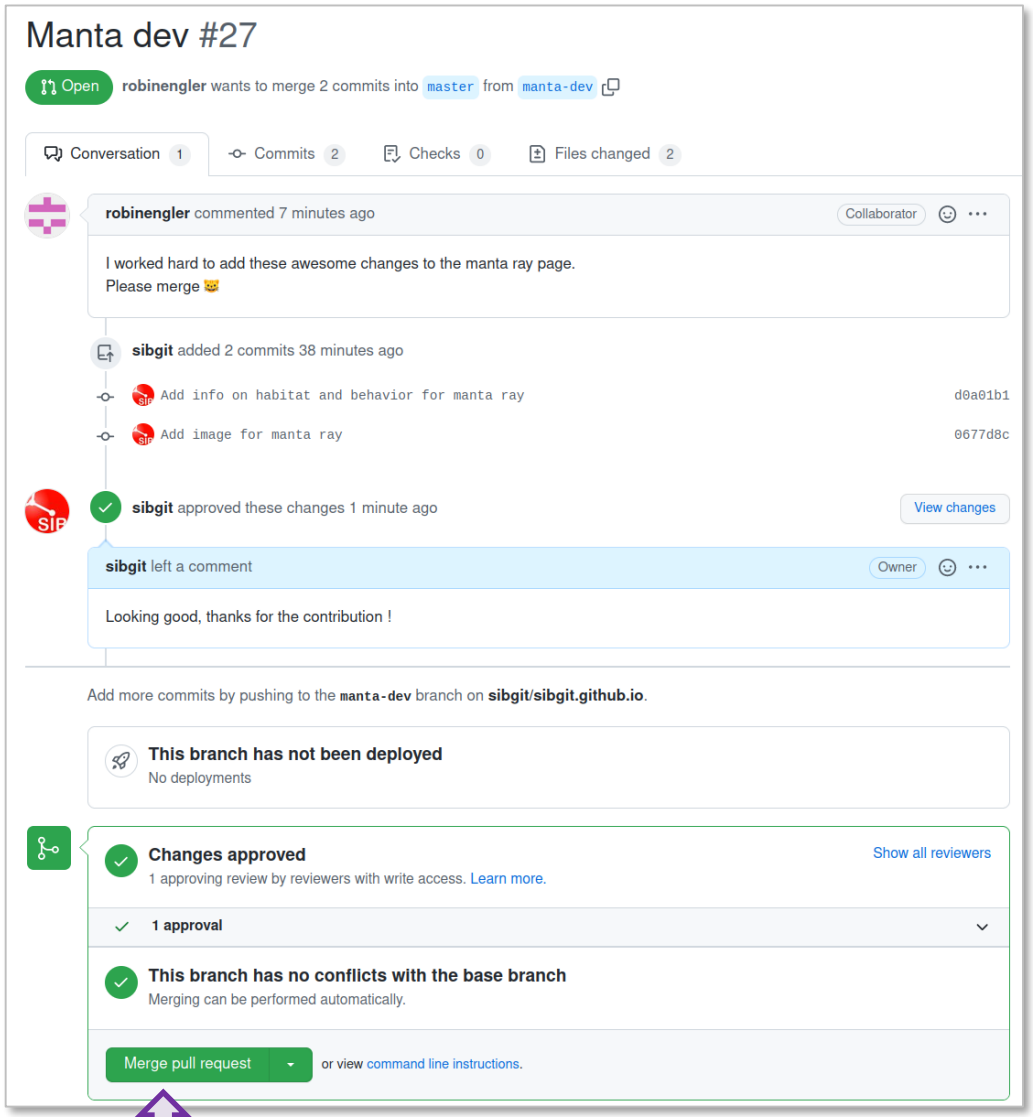
The **reviewer** of your PR will then have a look at your changes (the modifications introduced with your commits) and **approve them or request changes**.



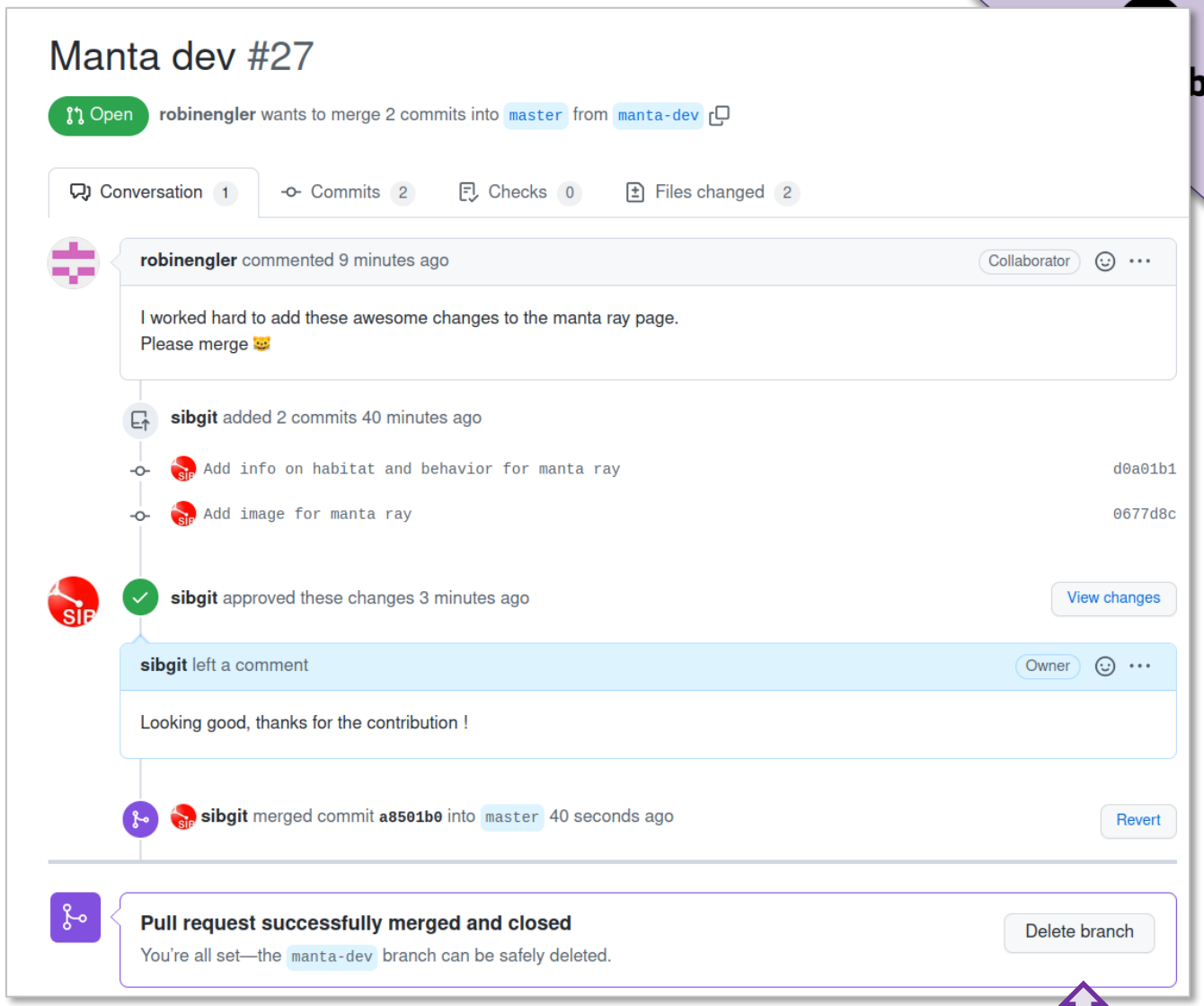




GitHub slide ...



Now that the pull request is approved, it can be merged (either by the reviewer or by you) by clicking **Merge pull request**.



Completed ! Optionally, you can **delete your branch** on the remote (this will not delete it locally).

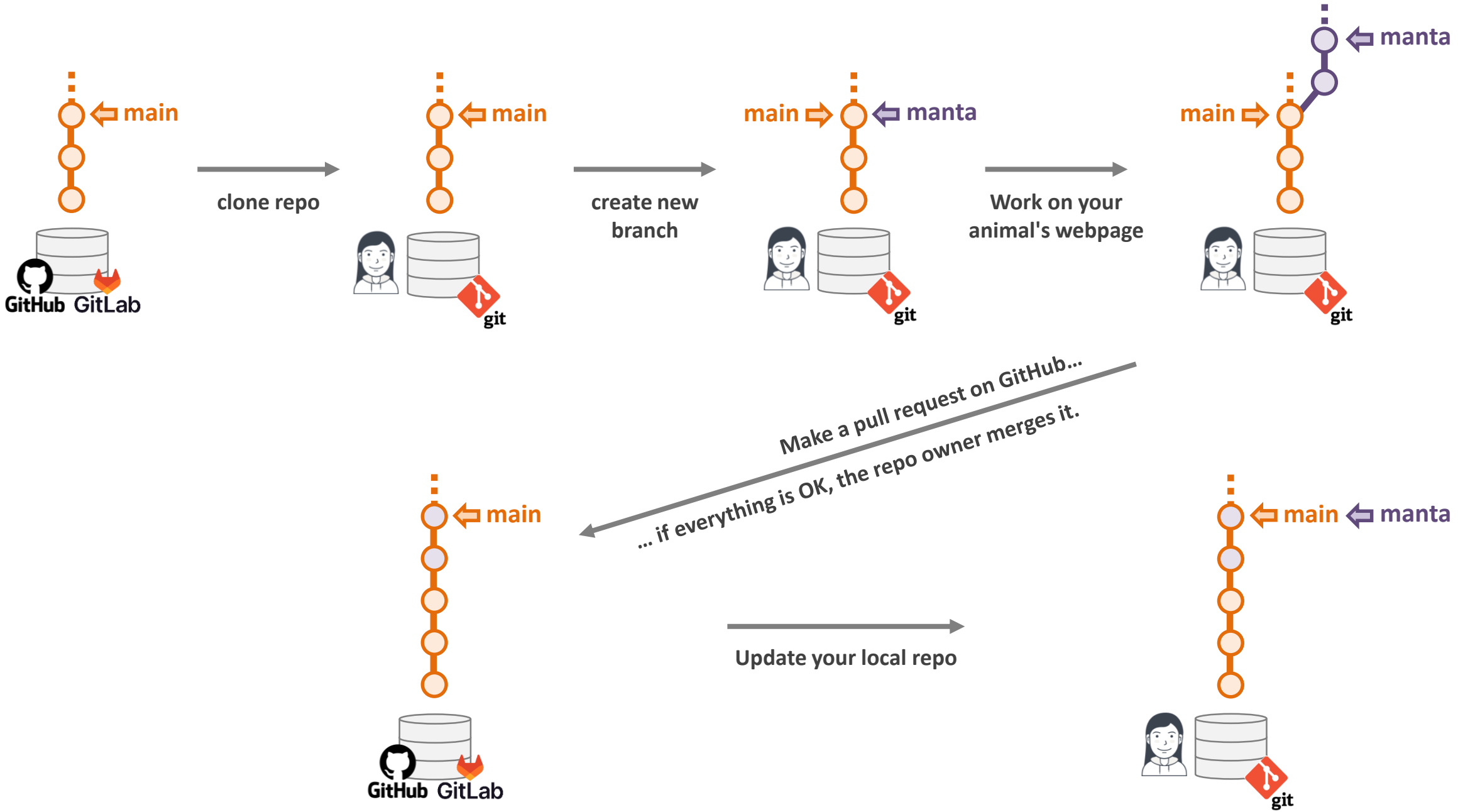
GitHub slide ...

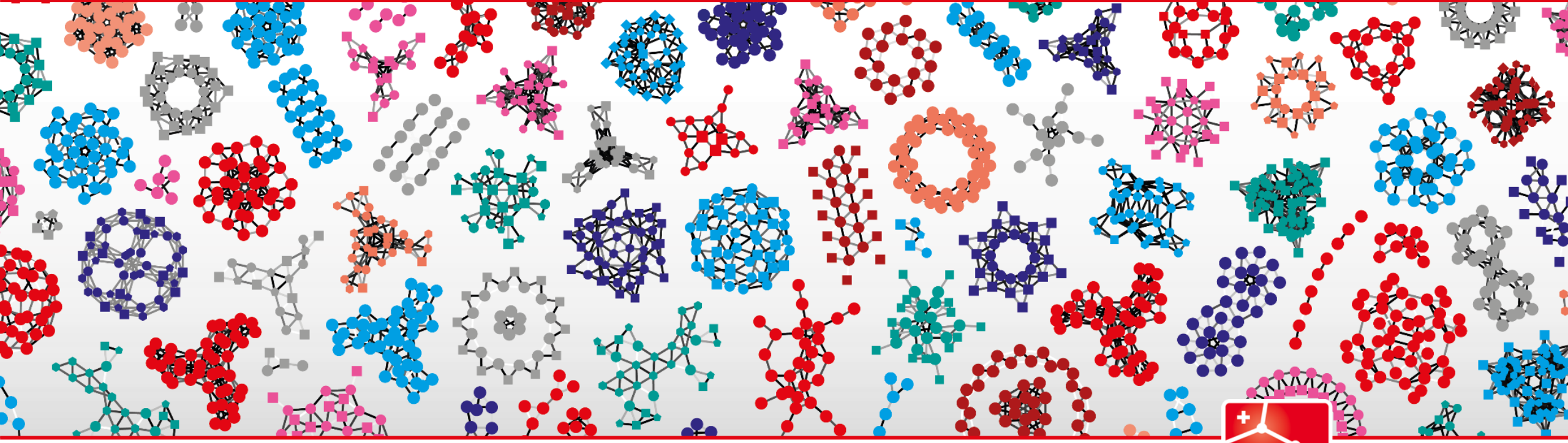
exercise 4

The Awesome Animal Awareness Project

This exercise has helper slides

Exercise 4 help: workflow summary





Thank you for attending this course

SIB
Swiss Institute of Bioinformatics