

SIB

Swiss Institute of  
Bioinformatics

[www.sib.swiss](http://www.sib.swiss)

# Version control with Git – optional modules

(Git LFS, Git submodules)

Robin Engler

Vassilios Ioannidis

Lausanne, 12-14.10.2022

# git submodules

The "symlink" of Git repositories

# What are submodules ?

- Git submodules allow keeping a Git repository as a **subdirectory** of another Git repository **while version controlling the version (latest commit) of the nested repository.**
- The “super-project” and the submodule remain independent repos, and have independent remotes.

Main repository / super-project (repo containing the submodule)

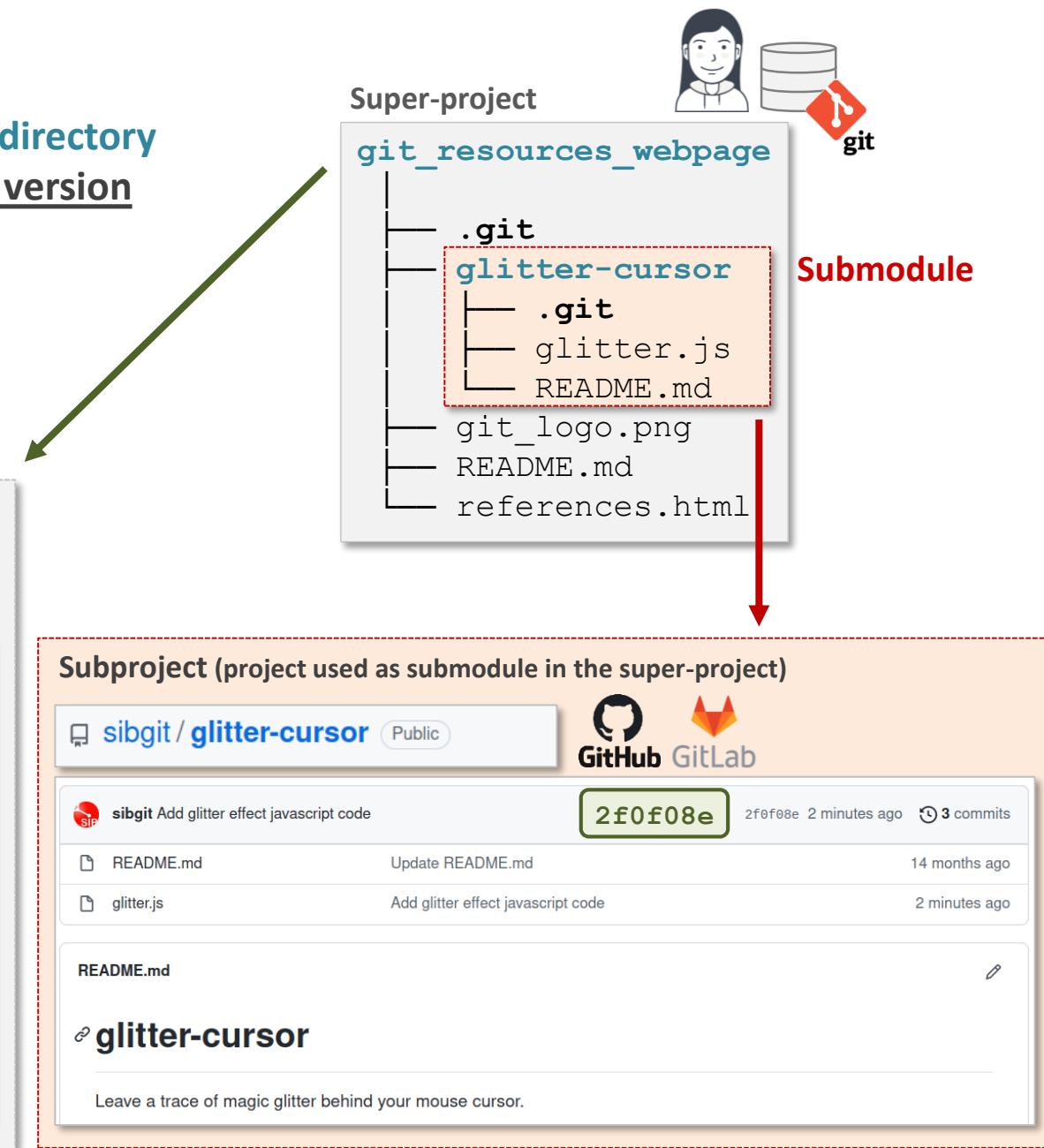


sibgit / git_resources_webpage Public		
 sibgit	Add submodule glitter-cursor	3be740e 38 seconds ago 4 commits
 glitter-cursor @ 2f0f08e	Add submodule glitter-cursor	38 seconds ago
 .gitmodules	Add submodule glitter-cursor	38 seconds ago
 README.md	Add README.md	12 hours ago
 git_logo.png	Add Git logo	12 hours ago
 references.html	Initial commit	12 hours ago

README.md

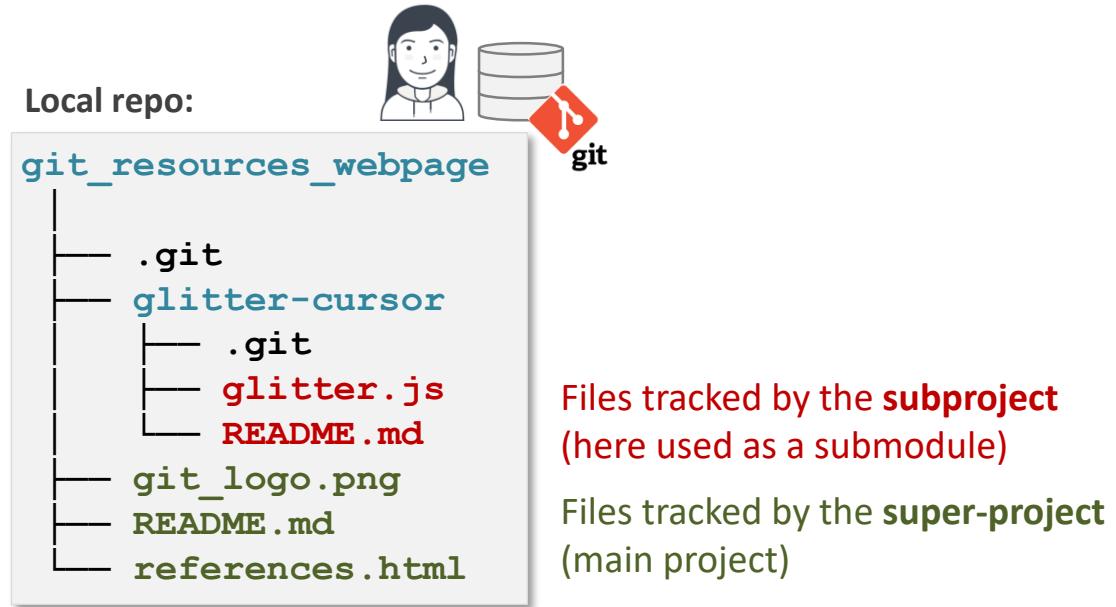
**Git resources web page**

A simple web page referencing a list of useful Git resources.



# What are submodules (continued)

- Git submodules are a **reference to another repository** at a **specific commit**. The super-project does not keep track of individual files inside the submodule.



- Because the submodule is **fixed at a specific commit** (unless explicitly changed), the maintainer of the super-project has **full control of which revision of the submodule's code they are using**.

On GitHub/GitLab, submodules are shown with the syntax:  
`<submodule dir name>@<commit hash>`

A red arrow points from the GitHub commit history screenshot to the submodule entry in the local repository diagram.

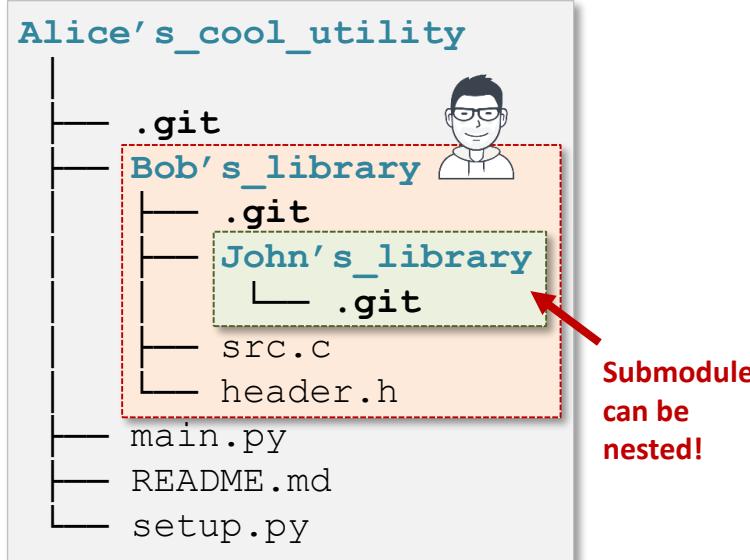
sibgit Add submodule glitter-cursor		
	glitter-cursor @ 2f0f08e	Add submodule glitter-cursor
	.gitmodules	Add submodule glitter-cursor
	README.md	Add README.md
	git_logo.png	Add Git logo
	references.html	Initial commit

# Use cases: when to use submodules

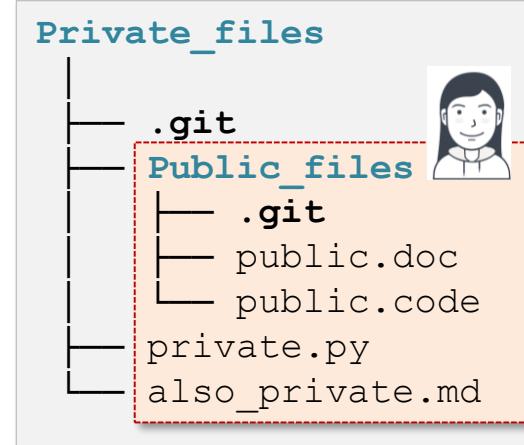
- To include external code, i.e. code maintained by someone else (e.g. on GitHub/GitLab), into your project. With Git submodules you can easily integrate it, get updates from the upstream, and stay in control of when the external code should be updated. Can also be used to re-use one of your own repos in multiple projects.
- To make public only a part of a project. You can put the part of your code/files that you want to make public in a submodule (with public access), and keep the rest of the code in a private repository.
- Large project that uses multiple subprojects maintained independently.



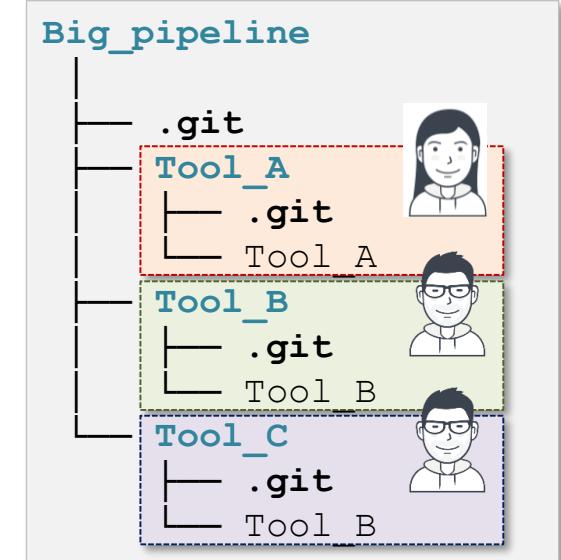
Alice uses a library maintained by Bob as a submodule



Alice wants to mix public and private files in a project.



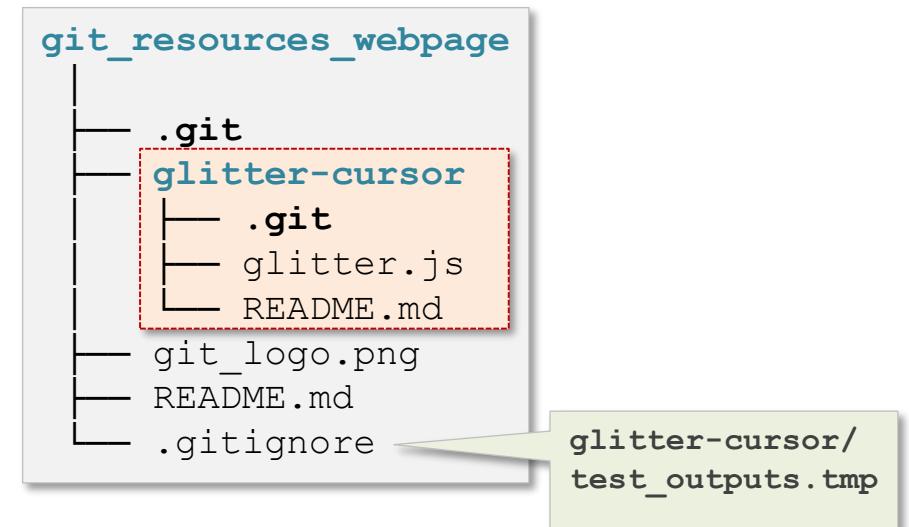
Large pipeline with multiple collaborators.



# When NOT to use submodules

- Don't use submodules when not really needed, monolithic repositories are simpler to maintain.
- If you have a sub-project that you want to use in multiple projects, it might be more efficient to create a package instead. Most programming languages have a dedicated package managers/repositories (CRAN for R, npm for javascript, PyPI for Python, etc).
- If you simply want to have a nested Git repos on your local machine (but with no link between them), you can simply add the nested repo to the `.gitignore` file of the higher-level repo.

If all you want is keeping a Git repo inside another one on your local computer with no link between them... you don't need submodules – save yourself the hassle!



# Adding/registering a submodule

To add/register a new submodule inside a Git repo:

```
git submodule add <URL of submodule repository>
```

This will:

- Add a new directory named after the submodule's repo name.
- Download the content of the submodule corresponding to the latest commit (on the default branch) into that directory.
- Create a `.gitmodules` file at the root of the super-project.



- If you add multiple submodules, you will have multiple entries in `.gitmodules`.
- `.gitmodules` should be version controlled, so that other people who clone the project know where the submodule projects are from (Git stages this file by default when adding a new submodule).

Submodule with custom name:



- Set custom name when adding submodule:  
`git submodule add <URL> <name>`
- Rename an existing submodule:  
`git mv <submodule name> <submodule new name>`

`.gitmodules`

```
[submodule "my-submodule"]
  path = my-submodule
  url = https://github.com/some-user/my-submodule.git
```

← Local path of submodule  
← URL of submodule

- Initialize the submodule in the `.git/config` file.

`.git/config`

```
[submodule "my-submodule"]
  url = https://github.com/some-user/my-submodule.git
  active = true
```

“active = true” --> module is initialized

# Adding a submodule: example

Adding “glitter-cursor” as a submodule to “git\_resources\_webpage”

Main repository / super-project  
(repo to which a submodule is added)

**sibgit / git\_resources\_webpage** Public

**sibgit Add Git logo**

- README.md Add README.md
- git\_logo.png Add Git logo
- references.html Initial commit

**README.md**

**Git resources web page**

A simple web page referencing a list of useful Git resources.

**sibgit Add submodule glitter-cursor**

- + **glitter-cursor @ 2f0f08e** Add submodule glitter-cursor
- .gitmodules Add submodule glitter-cursor
- README.md Add README.md
- git\_logo.png Add Git logo
- references.html Initial commit

**README.md**

Subproject  
(used as submodule in the super-project)

**sibgit / glitter-cursor** Public

**sibgit Add glitter effect javascript code** 2f0f08e 2 minutes ago 3 commits

- README.md Update README.md 14 months ago
- glitter.js Add glitter effect javascript code 2 minutes ago



Repo is currently at  
commit **2f0f08e**

**glitter-cursor**

Leave a trace of magic glitter behind your mouse cursor.

```
git submodule add https://.../glitter-cursor.git
git commit -m "Add submodule glitter-cursor"
git push
```

Icon and syntax indicating a submodule, which is pointing at **2f0f08e**

When a new submodule is added, it points at the latest commit of the submodule's online repository.

# How Git keeps track of the submodule's version: some more details.

Adding “glitter-cursor” as a submodule to “git\_resources\_webpage”

```
$ cd git_resources_webpage
$ git submodule add https://github.com/sibgit/glitter-cursor.git
Cloning into '/home/.../git_resources_webpage/glitter-cursor'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (9/9), done.
```

## How does Git keep track of the submodule's version ?

```
$ git status
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file: .gitmodules
    new file: glitter-cursor
```

```
$ git diff --cached
diff --git a/.gitmodules b/.gitmodules
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "glitter-cursor"]
+  path = glitter-cursor
+  url = https://github.com/sibgit/glitter-cursor.git
diff --git a/glitter-cursor b/glitter-cursor
--- /dev/null
+++ b/glitter-cursor
@@ -0,0 +1 @@
+Subproject commit 2f0f08e991d828dd27cf399c0b88edaaa48a2bf9
```

2f0f08e

## Git submodule add does the following:

- Create a new directory named “glitter-cursor”.
- Download the content of “glitter-cursor” corresponding to the latest commit (on the default branch).
- Create a **.gitmodules** file.

```
[submodule "glitter-cursor"]
  path = glitter-cursor
  url = https://github.com/sibgit/glitter-cursor.git
```

Local path of submodule

URL of submodule

- Initialize the submodule in the **.git/config** file.

```
[remote "origin"]
  url = https://github.com/sibgit/git_resources_webpage.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
  remote = origin
  merge = refs/heads/main
[submodule "glitter-cursor"]
  url = https://github.com/sibgit/glitter-cursor.git
  active = true
```

Section that was added

“active = true” --> module is initialized

- The submodule is tracked/added as a “virtual file” to the index.
- This “virtual file” contains the **commit ID** (SHA-1 checksum) to which the submodule is pointing (and nothing else).
- Individual files in the submodule are not tracked by the super-project.

# Clone a repository with submodules

```
git clone <repository>
git submodule init
git submodule update
```

or

```
git clone <repository>
git submodule update --init --recursive
```

or

```
git clone --recurse-submodules <repository>
```

- After cloning a repository that contains submodules, there will only be an empty directory for the submodules: their content is not automatically downloaded!
- You have to initialize\* the local configuration files with: **git submodule init**
- Now the content of submodule(s) can be retrieved\*\* with: **git submodule update**
- **--recursive / --recurse-submodules** means that the command also applies to nested submodules (submodules within submodules).

**Shortcut to clone, initialize and update all submodules.**

This is what you will want to use in most situations.

## Notes:

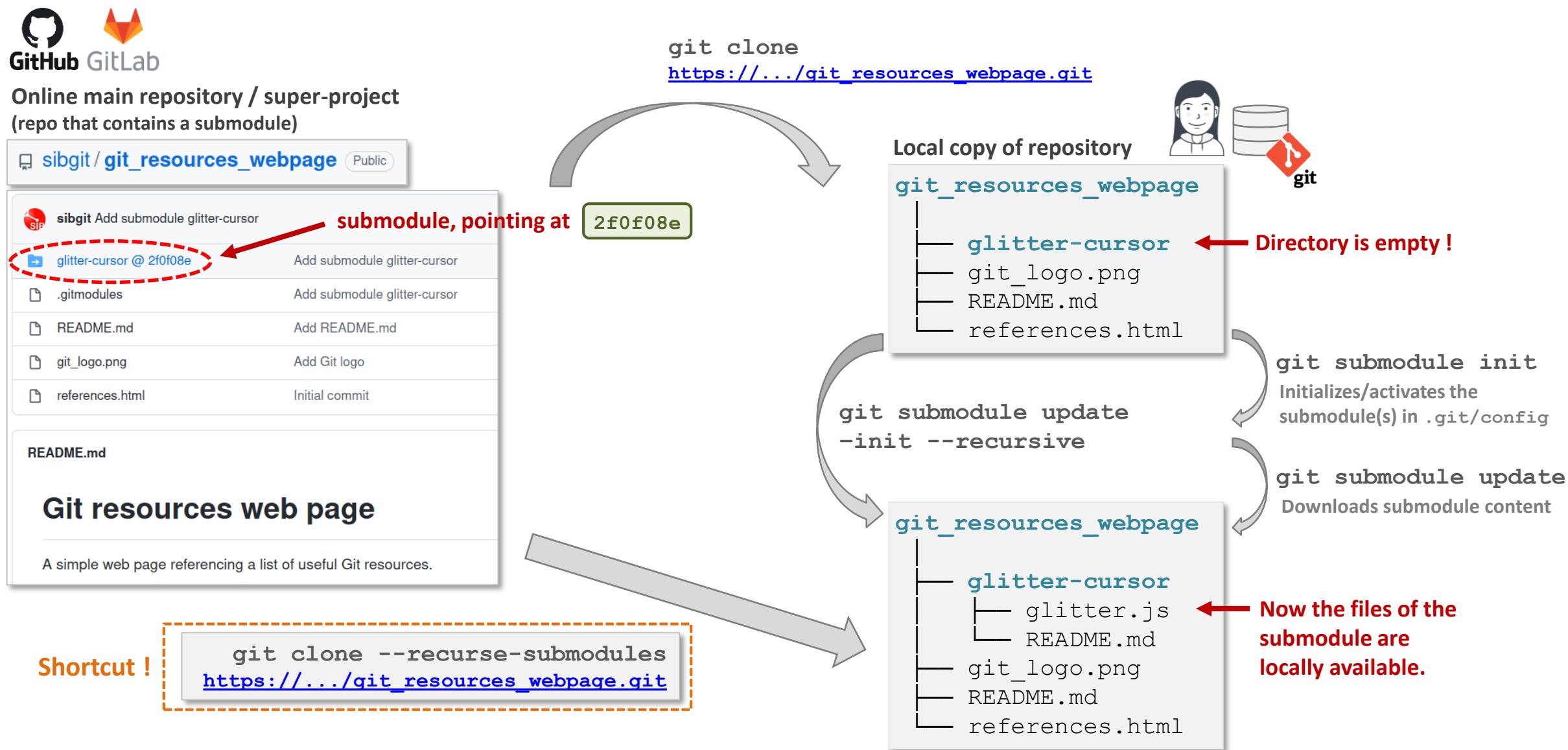
- By default, the commands **git submodule init/update** apply to all submodules of a project. To apply them only to a specific submodule, the name of the submodules can be passed: e.g. **git submodule init <submodule name>**
- What does \*initialize a submodule mean, and what exactly does **git submodule init** do?  
When Git initializes a submodule, it creates an entry for it in the **.git/config** file of the superproject repo and marks it as “active = true”.  
When working on a large project with many submodules, this makes it e.g. possible to only initialize those submodules that are really needed for your work.
- The meaning of \*\*update in **git submodule update** is to fetch updates in submodules and update the working tree of the submodules to the revision expected by the superproject. It does not mean to update the submodules to their latest version.

**.git/config**

```
[submodule "glitter-cursor"]
active = true
url = https://github.com/sibgit/glitter-cursor.git
```

## Clone a repository with submodules: example

Cloning “git\_resources\_webpage” that contains the submodule “glitter-cursor”.



# Cloned submodules are (by default) in detached HEAD state

- After cloning a repo with submodules, the submodules are in **detached HEAD** state.
- To make it point to a branch you have to explicitly checkout (switch to) that branch.

```
$ cd glitter-cursor
$ git status
HEAD detached at 2f0f08e ← Commit the submodule is currently pointing at.
$ git switch main
```



To display the revision of the submodule to which a super-project is currently pointing:

```
$ git submodule status
2f0f08e991d828dd27cf399c0b88 glitter-cursor (heads/main)
```



# Update a repository with submodules (git pull on the super-project)

Similarly to `git clone`, running `git pull` in the super-project (the main project that hosts the submodule) does not automatically update the submodules' content. You need to either:

```
git pull  
git submodule update --init --recursive
```

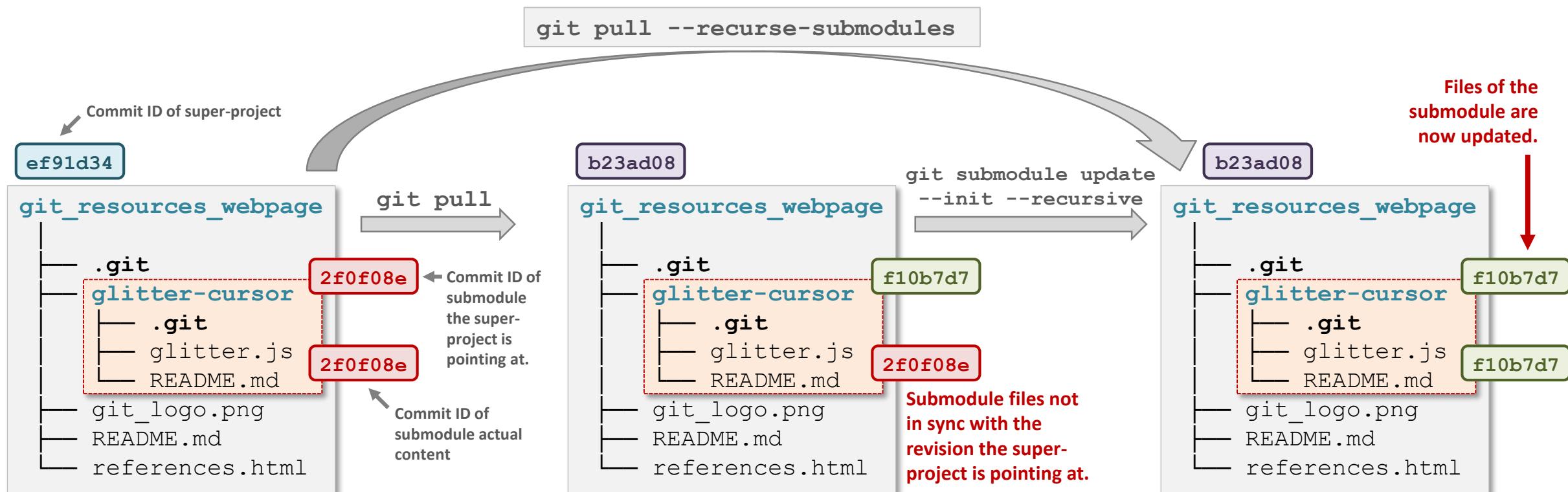
Or

```
git pull --recurse-submodules
```

Downloads the submodule's updated content

Shortcut !

This is what you will want to use in most situations.



# Working with submodules

- Submodules are regular Git repos. Once inside, you can run the same Git commands as you would on any repo.
- The super-project does not keep track of individual files in the submodule: it only keeps track of the commit to which it points.

However, the super-project will detect when changes are made inside a submodule (but not exactly which changes).



- To run the same tasks on multiple submodules, there is the handy command:

```
git submodule foreach "git command"
```

## Example:

```
$ cd glitter-cursor
# We are now in the submodule directory.
$ git status
$ git add ...
$ git commit ...
$ git push
```

## Example: files were added/modified in the submodule.

```
$ git status # run in the super-project's root!
Changes not staged for commit:
  modified: glitter-cursor (modified content,
  untracked content)
```



## Example: one or more new commits in submodule.

```
$ git status # run in the super-project's root!
Changes not staged for commit:
  modified: glitter-cursor (new commits)
```



## Example:

```
$ git submodule foreach "git status"
$ git submodule foreach "git log --oneline"
Entering 'glitter-cursor'
2f0f08e (HEAD -> main) Add glitter effect code
841e83a Update README.md
b0b66f8 Initial commit
```

# Making changes to a submodule (modifying the content of the submodule)

Let's assume we want to **modify the content** of a submodule, for instance:

- Update the submodule's content to a newer version.
- Make changes to files in the submodule.
- Point the submodule at an older version.

We proceed as follows:

1. Make the desired changes in the submodule.

If needed, pull/push the changes from/to the submodule's remote.

2. The commit ID (hash) of the submodule has now changed, so we must update the super-project by making a new commit that will indicate the update in commit ID of the submodule.

New commit to which the submodule is now pointing →

Making a new commit in the →  
super-project

Commands run in the submodule:

```
$ cd glitter-cursor
```

```
$ git pull
```

```
$ git add ...  
$ git commit ...  
$ git push
```

```
$ git checkout ...
```

Commands run in the super-project:

```
$ git status
```

On branch main

Changes not staged for commit:

**modified: glitter-cursor (new commits)**

```
$ git diff
```

diff --git a/glitter-cursor b/glitter-cursor

--- a/glitter-cursor

+++ b/glitter-cursor

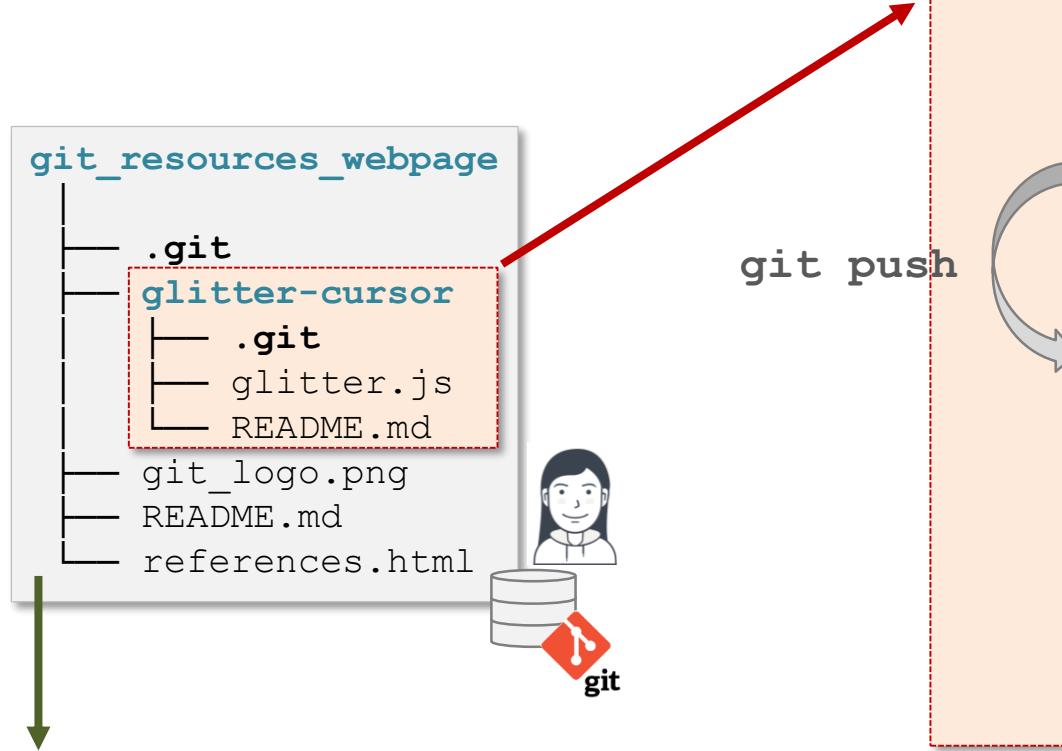
**-Subproject commit 2f0f08e991d828dd27cf399c0b88eda48a2bf9**

**+Subproject commit f10d7b772342c6a9f31390af4f8a16f71c440777**

```
$ git add glitter-cursor # go back to the super-project.  
$ git commit -m "Update submodule glitter-cursor"  
$ git push
```

# Making changes to a submodule

How things look on the online pages of the remotes



Subproject  
(used as submodule in the super-project)

sibgit / **glitter-cursor** Public



**2f0f08e**

2f0f08e 2 minutes ago 3 commits

sibgit Add glitter effect javascript code

README.md

Update README.md

14 months ago

glitter.js

Add glitter effect javascript code

2 minutes ago

sibgit Improve glitter effect

**f10d7b7**

f10d7b7 4 minutes ago 4 commits

README.md

Update README.md

14 months ago

glitter.js

Add glitter effect javascript code

8 hours ago

glitter\_improved.js

Improve glitter effect

4 minutes ago

README.md

## glitter-cursor

Leave a trace of magic glitter behind your mouse cursor.

Main repository / super-project (repo containing the submodule)

sibgit / **git\_resources\_webpage** Public



sibgit Add submodule glitter-cursor

**2f0f08e**

glitter-cursor @ 2f0f08e

git push



sibgit Update submodule glitter-cursor

**f10d7b7**

glitter-cursor @ f10d7b7

## --recurse-submodules option: automated submodules push

To avoid accidentally forgetting to push changes in a submodule when pushing in the super-project:

- `git push --recurse-submodules=check` : safeguard that will make your push fail if there are any “non-pushed” changes in submodules.
- `git push --recurse-submodules=on-demand` : automatically push all submodules when pushing the super-project.
- These options can also be permanently set in the Git configuration of the super-project:

```
$ git config push.recurseSubmodules check  
# or  
$ git config push.recurseSubmodules on-demand
```

Note: we are not using the `--global` option, so this setting only affects the current repo.

- **Important:** all these commands are run in the context (directory) of the super-project, not of the submodule!

Examples:

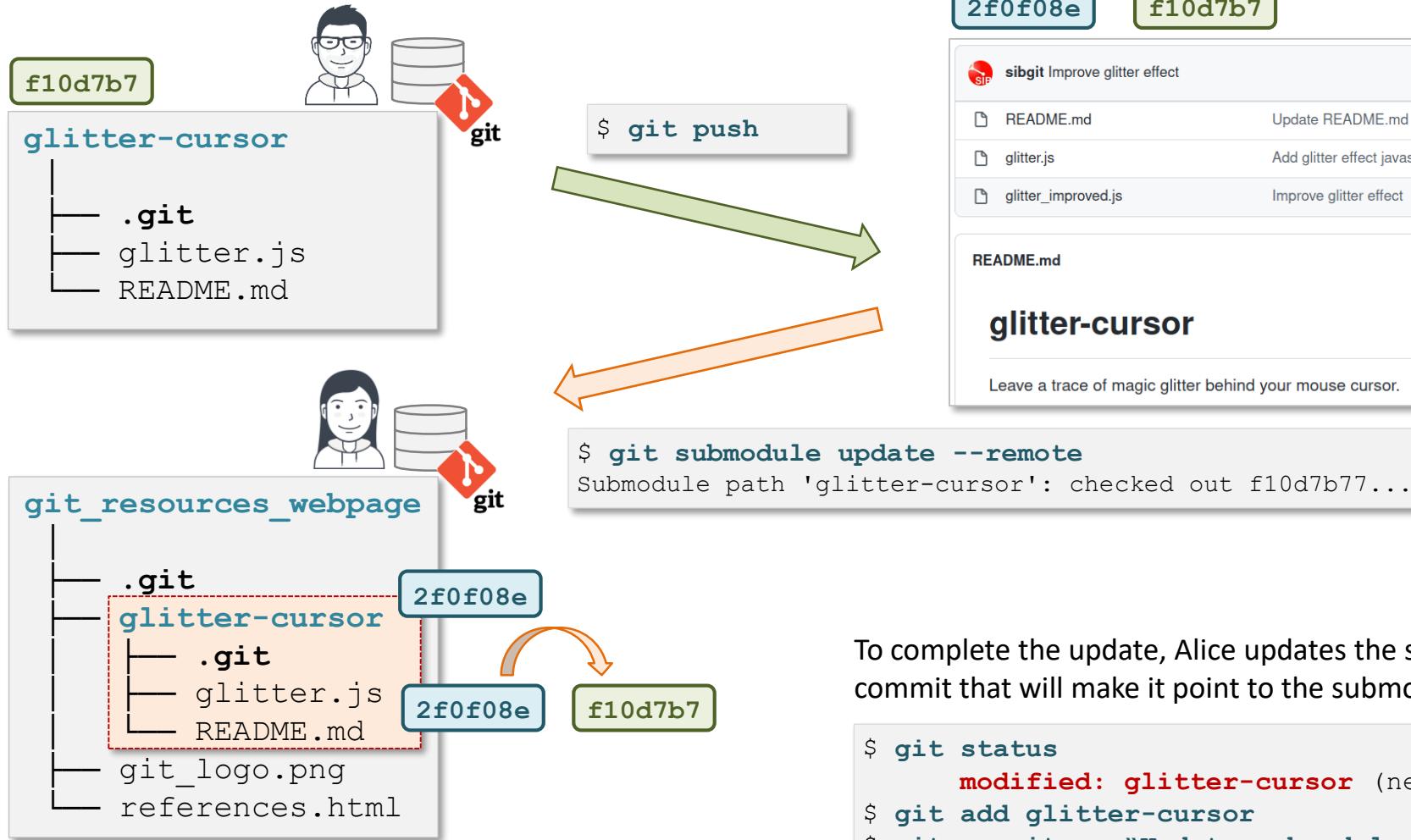
```
$ git push --recurse-submodules=check  
The following submodule paths contain changes that  
cannot be found on any remote:  
    submodule-name  
  
Please try  
git push --recurse-submodules=on-demand
```

```
$ git push --recurse-submodules=on-demand  
Pushing submodule 'submodule-name'  
...  
Pushing super-project (main project)  
...
```

# Pulling updates for a submodule

Updating a submodule to its latest commit

1. Bob, the maintainer of the “glitter-cursor” repo, pushes a new update.
2. Alice updates her submodule in the “git\_resources\_webpage” project with Bob’s new update.



`git submodule update --remote <submodule name>`

If no submodule is specified, all submodules are updated

To complete the update, Alice updates the super-project with a new commit that will make it point to the submodule commit: `f10d7b7`

```
$ git status
  modified: glitter-cursor (new commits)
$ git add glitter-cursor
$ git commit -m "Update submodule to latest version"
```

# Pulling updates for a submodule (command details)

Updating a submodule to its latest commit

To pull the latest changes for a submodule:

```
git submodule update --remote <submodule name>
```

- If no submodule is specified, all submodules are updated.
- If the local submodule has diverged from its remote (e.g. you made some commits), **--merge**/**--rebase** must be added to the command to either merge or rebase.

```
$ git submodule update --remote --merge
```

- By default Git will try to pull the changes from the master branch. To pull from another branch, you have to specify it in **.gitmodules** by setting the parameter **branch**.
- After the content of the submodule is updated, the update in its version (commit hash) must still be committed.

**.gitmodules**

```
[submodule "glitter-cursor"]
path = glitter-cursor
url = https://.../glitter-cursor.git
branch = main
```

```
$ git status
modified: my-submodule (new commits)
$ git add my-submodule
$ git commit -m "Update submodule to latest version"
```

Alternatively, the pull in the submodule can also be done manually:

```
$ cd my-submodule
$ git switch main      # If in DETACHED HEAD state.
$ git pull
```

# exercise 5

The Git reference web page gets  
better with submodules

# git LFS

large file storage

# Tracking large files can be useful...

Tracking large files together with code is an attractive proposition, e.g. in scientific applications:

- Data analysis/processing pipeline.
- Machine learning applications (training data and code in the same place).

## ... but Git does not work well with large files

- Git was designed for tracking code – i.e. relatively small text files.
- Adding large files to a Git repo is technically possible, however:
  - Since Git is a distributed VCS, **each local copy of a repository will contain a full copy of all versions of all tracked files**. Therefore, adding large files will quickly inflate the size of everyone's repository, resulting in higher disk space usage (on local hosts).
  - Git's internal data compression (i.e. packfiles) is **not optimized to work with binary data** (e.g. image or video files). Each change to a binary file will (more or less) add the full size of the file to the repo, taking disk space and slowing down operations such as repo cloning or update fetching.
  - Commercial hosting platforms **impose limits on the size of files** that can be pushed to hosted Git repos (GitHub: 100 MB, GitLab: no file limit but 10 GB repo limit).

# The solution\*: Git LFS

Git LFS (Large File Storage) is an extension for Git, specifically **designed to handle large files**.

Basic principle: large files are not stored in the Git database (the `.git` directory), instead:

- Only a **reference/pointer to large files** is stored in the Git database.
- The actual **files are stored in a separate repository or “object store”**.

Open source project: <https://git-lfs.github.com>

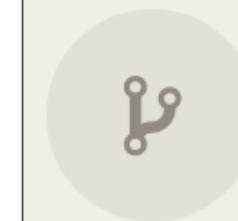
First released in 2015.



Not all hosting services support Git LFS, and when they do, storage space is limited (additional space may be purchased).

\* Alternatives to Git LFS exist, but Git LFS is the most popular.

## Features



### Large file versioning

Version large files—even those as large as a couple GB in size—with Git.

### More repository space

Host more in your Git repositories. External file storage makes it easy to keep your repository at a manageable size.

### Faster cloning and fetching

Download less data. This means faster cloning and fetching from repositories that deal with large files.

### Same Git workflow

Work like you always do on Git—no need for additional commands, secondary storage systems, or toolsets.

# GitHub and GitLab disk quotas, file size limit and pricing

- If your institution is running their own instance of GitLab, you can check with them if they offer LFS support (and how much space you can have there).
- Here are limits for 2 popular commercial Git hosting providers:

	GitHub.com	GitLab.com
Max file size	100 MB	No size limit
Max repo size	1 GB (recommended) 2 GB to 5 GB (max)	10 GB
LFS max file size	2 GB	No size limit (not sure)
LFS object store	1 GB storage for free 1 GB/month free bandwidth (download)  5 USD/month for each additional “pack” of 50 GB storage + 50 GB bandwidth	60 USD/year per 10GB

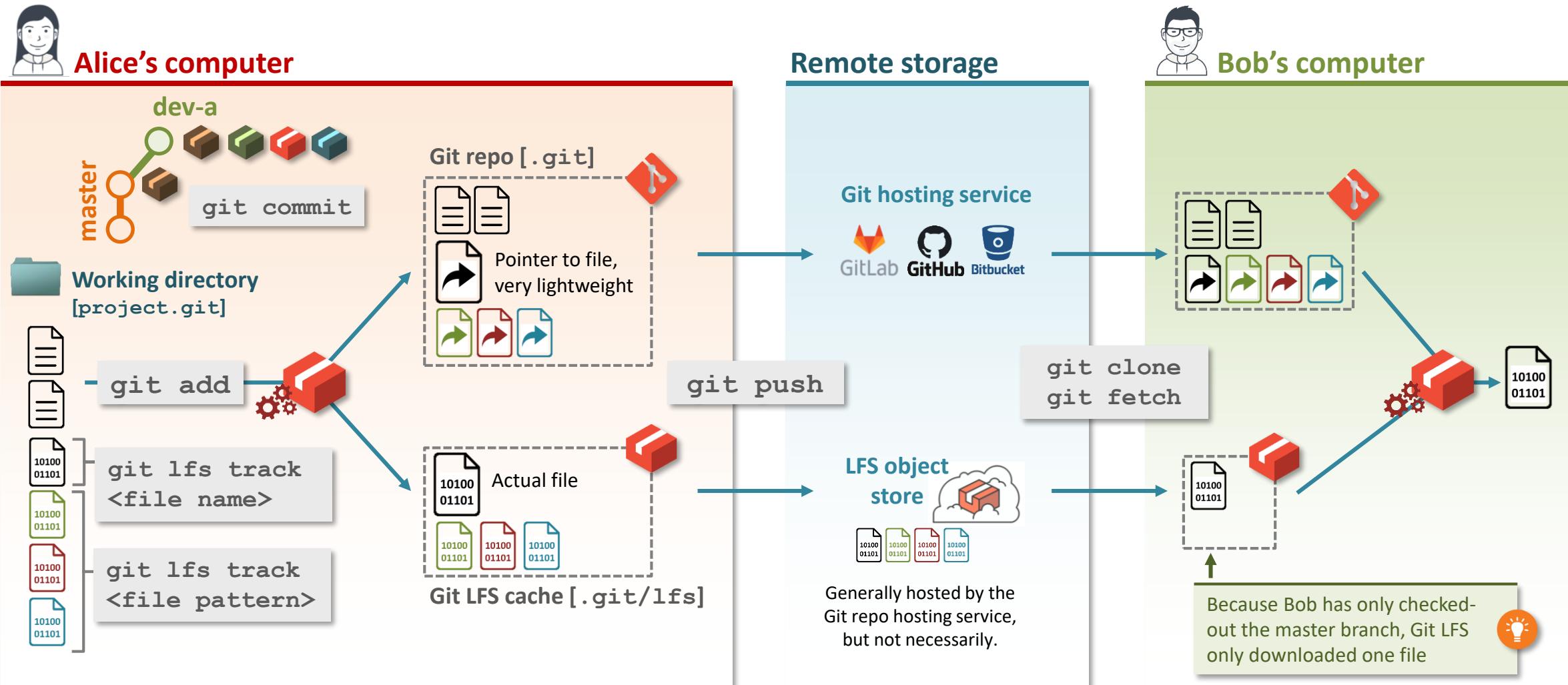
last updated on Feb 2021

You can also setup a Git LFS object store on third-party storage provider -but you need to set it up yourself and it is not a trivial task:

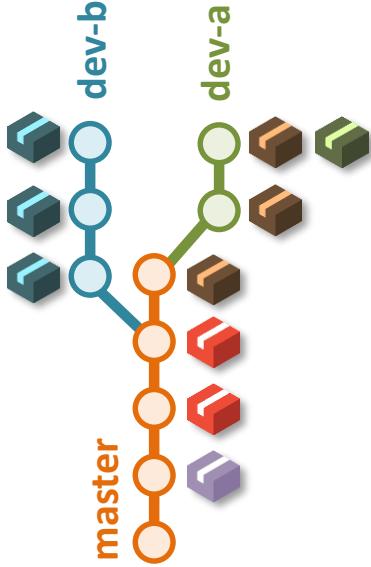
- SWITCHengines (220 CHF/TB\*year) – no backup (need to organize your own).
- AWS (amazon web services).

# Git LFS workflow overview

- Only a reference/pointer to large files is stored in the Git database.
- The large files themselves are stored in a separate repository or “object store”.
- Large files are downloaded only when needed.
- Transparent: only 1 extra command is needed for this workflow (`git lfs track`).

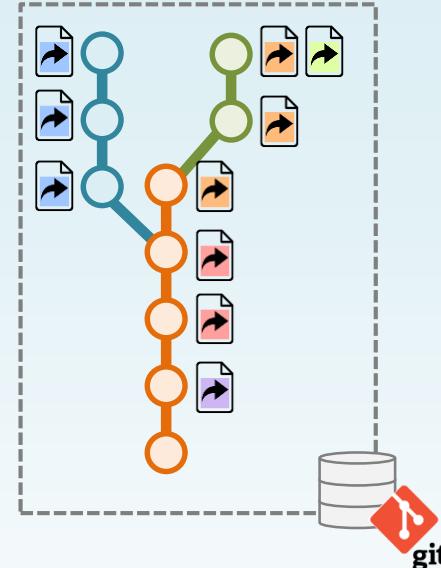


## Complete Git history of project

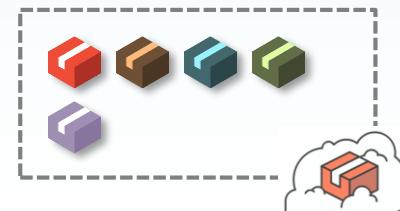


## Remote storage

### Git database content



### LFS object store content

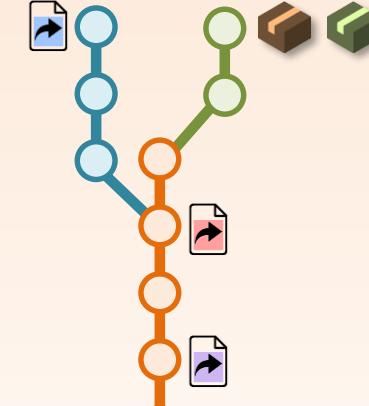


## Local Git repositories

### Alice's local repo



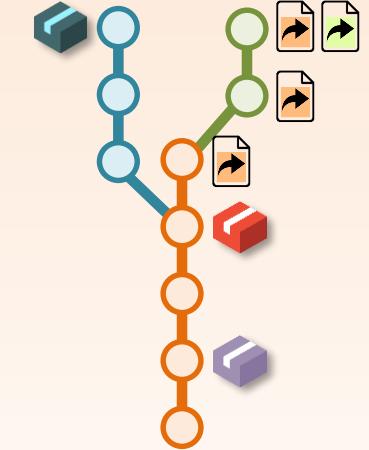
Alice just started to work on the project. She cloned the repo and created the "dev-a" branch.



### Bob's local repo



Bob contributed to the project since a while. He's currently working on "dev-b".



git checkout dev-b

git lfs purge

# Git LFS: initial setup



- One time setup: to be executed only once per user/machine, after Git LFS was installed.  
(this adds LFS Git filters to your global configuration file `~/.gitconfig`)

```
git lfs install
```

# Git LFS: tracking files



- Adding files to Git LFS:

```
git lfs track <file name or pattern>
```

- When using a file pattern (glob pattern), all files matching the pattern are tracked.
- Each call to `git lfs track` creates a new entry in the `.gitattributes` file.

- Examples:

```
$ git lfs track file_1.csv  
$ git lfs track file_2.csv file_3.csv  
$ git lfs track *.fasta  
$ git lfs track *.img  
$ git lfs track "large_file_?.txt"  
$ git lfs track "subdir/*.jpg"
```

- Track the file named exactly “file\_1.csv”
- Track the files named exactly “file\_2.csv” and “file\_3.csv”
- Track all files ending in “.fasta”
- Track all files ending in “.img”
- Track all files whose name are of the form “large\_file\_” + any single character + “.txt”
- Track all files ending in “.jpg” in sub-directory “subdir”

Content of `.gitattributes`



```
file_1.csv filter=lfs diff=lfs merge=lfs -text  
file_2.csv filter=lfs diff=lfs merge=lfs -text  
file_3.csv filter=lfs diff=lfs merge=lfs -text  
*.fasta filter=lfs diff=lfs merge=lfs -text  
*.img filter=lfs diff=lfs merge=lfs -text  
large_file_?.txt filter=lfs diff=lfs merge=lfs -text  
subdir/*.jpg filter=lfs diff=lfs merge=lfs -text
```

It is also possible to edit directly the `.gitattributes` file instead of using the `git lfs track` command.





Do not forget “quotes” when using the `git lfs track` command with a file pattern, otherwise the pattern expands when the command is run and the matching files in your current working directory (rather than the pattern) are added to `.gitattributes`.

`git lfs track "*.img"` ✓

`git lfs track *.img` ✗

content of `.gitattributes` assuming that  
“file1.img” and “file2.img” are present in the  
working directory.

`*.img filter=lfs diff=lfs merge=lfs -text`

`file_1.img filter=lfs diff=lfs merge=lfs -text`  
`file_2.img filter=lfs diff=lfs merge=lfs -text`

if we add a new file “file\_3.img” at a later  
point in time...

✓ File “file\_3.img” is tracked because it  
matches the \*.img pattern.

✗ File “file\_3.img” is not tracked because it  
matches neither file\_1.img nor file\_2.img.

- Recursively tracking an entire directory

```
git lfs track "directory_path/**"
```

← Using `/**` is important.  
Using `/` or `/*` will *not* work.

#### Content of `.gitattributes`

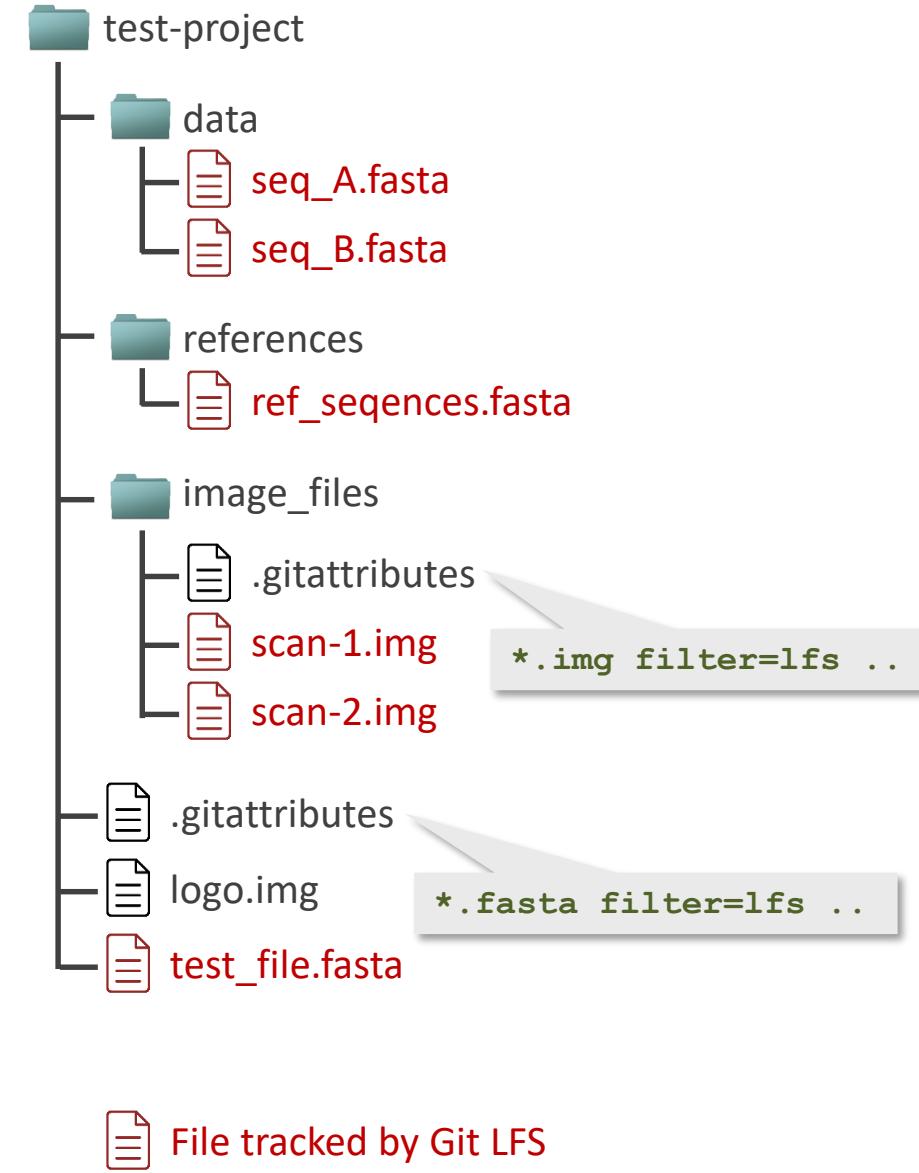
```
dir_to_track/** filter=lfs diff=lfs merge=lfs -text
```

# Git LFS file tracking: fine-grained control

- For fine-grained control, `git lfs track <file name/pattern>` can be run in sub-directories. This places `.gitattributes` files in sub-directories (similar to how `.gitignore` files behave).
- The scope of each `.gitattributes` file is its current directory and sub-directories.
- Running `git lfs track <file name or pattern>` inside a sub-directory, creates the `.gitattributes` file inside that sub-directory

The `.gitattributes` file(s) in your repo should be tracked - just like `.gitignore` file(s).

Don't forget to commit them.



# Negative pattern matching

- Unlike `.gitignore` files, `.gitattributes` files **do not support the `!pattern`** for negative matching (to tell Git LFS to not track a file).
- It is best to write `.gitattributes` files so that no negative matching is needed.
- If unavoidable, a workaround is possible by adding a line with the file/pattern that should not be tracked followed by `!filter !diff !merge` after the general pattern to track.

Example of `.gitattributes` file for tracking all “.jpg” files except “small\_logo.jpg”

```
*.jpg filter=lfs diff=lfs merge=lfs -text  
small_logo.jpg !filter !diff !merge
```

← **File that should not be tracked**

# Git LFS: untracking files



- Removing files from Git LFS:

```
git lfs untrack <file name or pattern>
```

- Calls to `git lfs untrack` remove entries from the `.gitattributes` file.
- The same result can be obtained by manually deleting lines from the `.gitattributes` file.

# Git LFS: adding and committing files



- Nothing special to do!
- Once files are tracked by LFS, adding them to git and committing them is done as usual.

```
git add ...
```

```
git commit ...
```

```
git push ...
```

# Git LFS: updating files



- Nothing special to do!
- Files tracked by Git LFS can be updated, staged and committed like any file under Git control.

```
$ git add sequence_db.fasta  
$ git commit -m "updated sequence database file"  
$ git push
```

The new version of the file is added to the local Git LFS cache. The pointer file is updated.  
The new version of the file is pushed to the remote LFS object store.

- After commits are pushed, the remote Git LFS object store contains a copy of each version of all LFS-tracked files.

## Data backup



The idea behind Git LFS is to *avoid replicating large data files* across local copies of a Git repository. This has implications for data-backup:

- For LFS-tracked files, local repos cannot be relied-upon to contain a full copy of all data.
- Therefore the remote repository has to be backed-up.

# Using Git LFS: diff-ing files



- For LFS-tracked files, `git diff` will only show the difference between pointer files, not between actual file content (even for text files).

```
git diff HEAD~1 sequences_A.fasta
diff --git a/sequences_A.fasta b/sequences_A.fasta
index a33c8a7..01f8d67 100644
--- a/sequences_A.fasta
+++ b/sequences_A.fasta
@@ -1,3 +1,3 @@
version https://git-lfs.github.com/spec/v1
-oid sha256:c1d5ab0faf552cdb3a365347093abc42a4e65718348e17eaad1584d650ae7aa6
-size 6010948
+oid sha256:fc51c1860c4341e175dcfc24fc2c653f75c5e8b3bae6cf80d3632788ccaf4379
+size 6011029
```

size of file in bytes

checksum (SHA-256) of file content.

# Listing files tracked by Git LFS



- List LFS-tracked files of **HEAD** commit (i.e. currently checked-out files).

```
git lfs ls-files
```

Example:

```
git lfs ls-files  
b04f62c7a1 * large_file_1.txt  
efdc76ef2a * sequences_B23.fasta  
e6aa57987e * subdir/logo_image.img
```

- List files associated with any reference (commit).

```
git lfs ls-files <ref>
```

Example:

```
git lfs ls-files HEAD~1  
b04f62c7a1 * large_file_1.txt  
fc51c1860c - sequences_A12.fasta  
efdc76ef2a * sequences_B23.fasta  
e6aa57987e * subdir/logo_image.img
```

\* = file is present in worktree

- = file is absent in worktree

- List all LFS-tracked files in the entire repo history.

```
git lfs ls-files --all
```

Example:

```
git lfs ls-files --all  
b04f62c7a1 * large_file_1.txt  
efdc76ef2a * sequences_B23.fasta  
e6aa57987e * subdir/logo_image.img  
e82048e6d3 - sequence_C34.fasta  
fc51c1860c - sequences_A12.fasta  
c1d5ab0faf - sequences_A12.fasta
```

# Clearing the local Git LFS cache



- Deleting files from the Git LFS local cache [`.git/lfs/objects`] can be done using:

```
git lfs prune
```

Files that are deleted by the `prune` command are those that:

- Are not currently checked-out.
- Are not part of the latest commit of a “recent” branch or tag (“recent” defaults to 10 days and can be customized via `lfs.fetchrecentcommitsdays` and `lfs.pruneoffsetdays` ).
- Are not part of a commit that was never pushed to the remote (since in this case there is not yet a copy of the file in the remote object store, and hence deleting it would amount to permanently losing the file).

- `lfs prune` command options:

```
git lfs prune --dry-run
```

- Lists the number of files that would be deleted, without actually deleting them.

```
$ git lfs prune --dry-run
prune: 6 local object(s), 4 retained, done.
prune: 2 file(s) would be pruned (12 MB), done.
```

```
git lfs prune --verify-remote
```

- Verify that files are present on the remote before deleting them.

# Pulling LFS content from a remote



- Nothing special to do!
- Just use the regular Git commands and Git LFS will download content as needed.

```
git clone ...
```

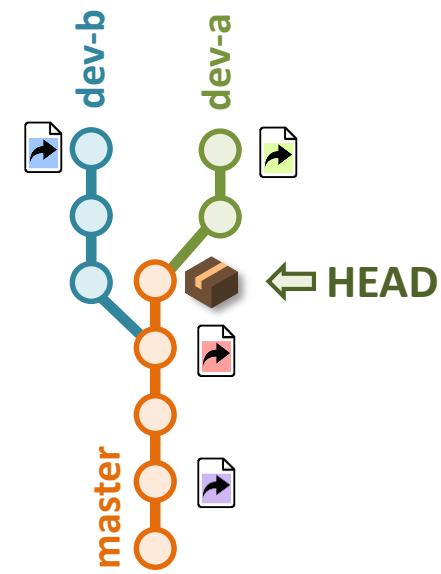
```
git fetch ...
```

```
git pull ...
```

```
git switch ...
```

- By default, only the LFS-tracked files needed for the currently checked-out branch are downloaded.

Example: if we `git clone` a new repository, only the LFS-tracked files needed for the latest commit of the “master” branch are downloaded.



# Pulling additional LFS content from a remote (files from older commits or files from other branches)



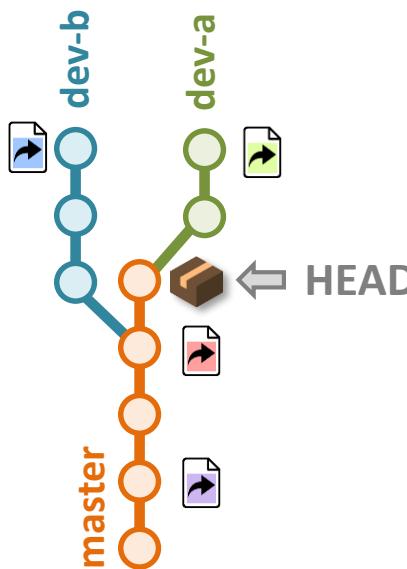
It can be useful to download LFS-tracked files to the local LFS cache, e.g. when anticipating off-line time.

`git lfs fetch --recent`

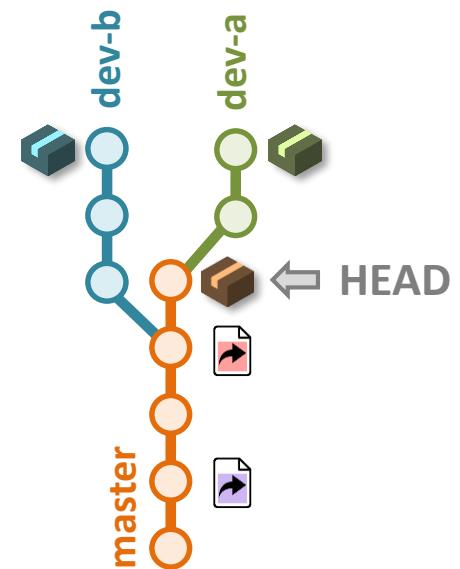
- Downloads the LFS-tracked files of *the last commit of all branches or tags that are considered “recent”*.
  - By default, “recent” is defined as no more than 7 days old.
  - The definition of “recent” can be customized via the `git config lfs.fetchrecentcommitsdays <days>` configuration option (where `<days>` = number of days).

`git lfs fetch --all`

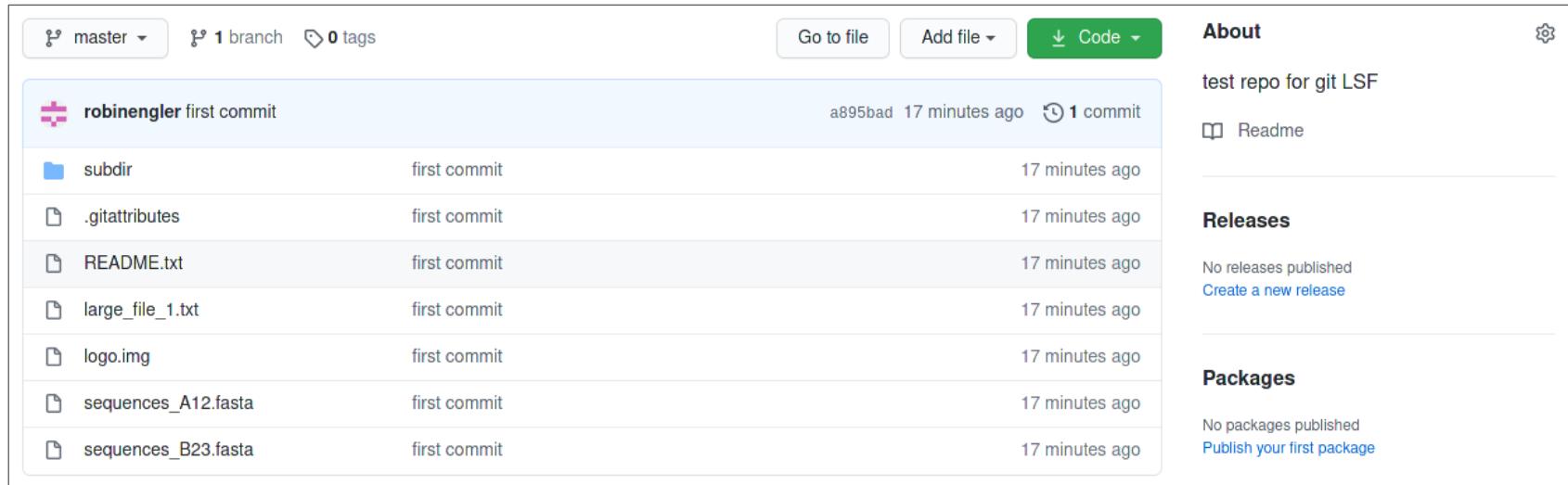
- Downloads all LFS-tracked files for all commits.



```
$ git lfs fetch --recent
fetch: Fetching reference refs/heads/master
fetch: Fetching recent branches within 7 days
fetch: Fetching reference origin/dev-a
fetch: Fetching reference origin/dev-b
```

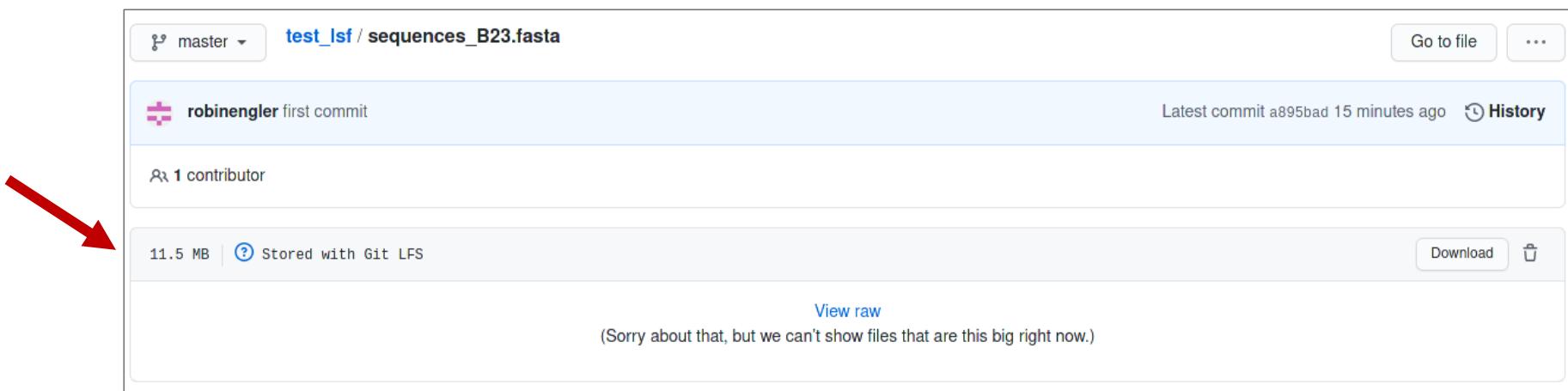


- On Git hosting platforms like GitHub or GitLab, LFS-tracked files are listed just like regular files:



A screenshot of a Git repository interface. At the top, it shows 'master' branch, '1 branch', '0 tags', 'Go to file', 'Add file', and a green 'Code' button. The main area lists files from a commit by 'robinengler': 'subdir' (first commit, 17 minutes ago), '.gitattributes' (first commit, 17 minutes ago), 'README.txt' (first commit, 17 minutes ago), 'large\_file\_1.txt' (first commit, 17 minutes ago), 'logo.img' (first commit, 17 minutes ago), 'sequences\_A12.fasta' (first commit, 17 minutes ago), and 'sequences\_B23.fasta' (first commit, 17 minutes ago). To the right, there's an 'About' section with 'test repo for git LSF', a 'Readme' link, 'Releases' (no releases), and 'Packages' (no packages). A gear icon is at the top right.

- When selecting an LFS-tracked file, the content is not shown and instead a “Stored with Git LFS” mention is listed:



A screenshot of a Git commit page for 'test\_lsf / sequences\_B23.fasta'. At the top, it shows 'master' branch, the file path 'test\_lsf / sequences\_B23.fasta', 'Go to file', and an ellipsis button. The commit details show 'robinengler first commit' and 'Latest commit a895bad 15 minutes ago'. Below, it says '1 contributor'. In the bottom left, a red arrow points to the '11.5 MB | ? Stored with Git LFS' message. In the bottom center, it says 'View raw' and '(Sorry about that, but we can't show files that are this big right now.)'. A 'Download' button and a trash bin icon are at the bottom right.

# exercise 6 A



# Tracking files already in Git

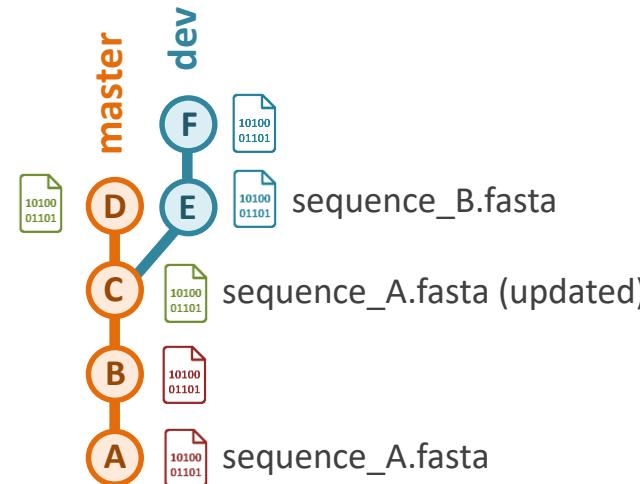
When a set of files are already part of a Git repository's history, there are two options to start tracking them with Git LFS:

1. Add the files (or file patterns) as tracked files with `git lfs track`. In this case however, the versions of the files associated with already made commits will remain in the Git database.
2. Remove the files' entire history from the Git repo, and have them tracked by Git LFS instead (over all of their history). This can be done using `git lfs migrate` command.

## Option 1

Keep files to track history in the Git repo up to the current commit.

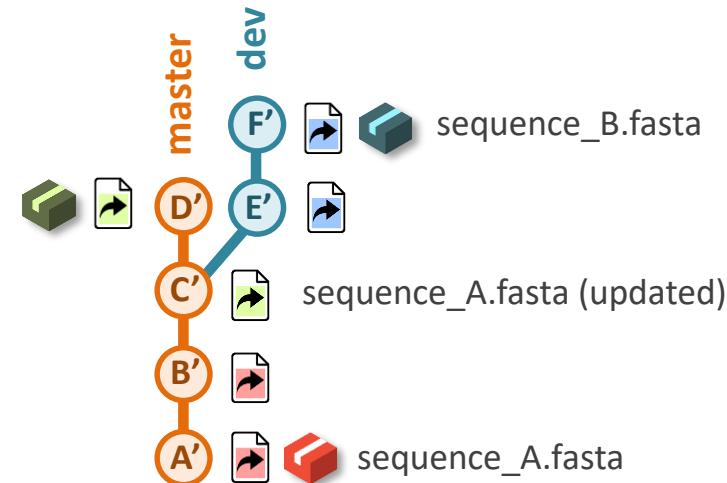
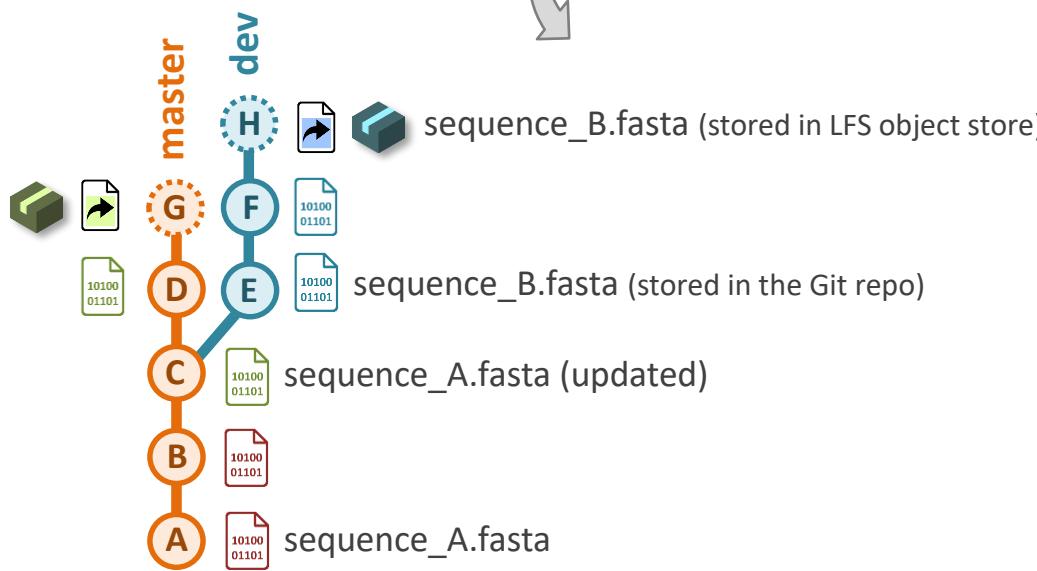
```
git lfs track "*.fasta"
git add *.gitattributes
git add *.fasta
git commit
... now do the same for branch dev
```



## Option 2

Remove files from entire Git repo history and rewrite history with files stored in LFS.

```
git lfs migrate import \
--include="*.fasta" \
--everything
git lfs checkout
```



+ The repo's history remains the same.

- Git repo size possibly still too large to push to GitHub/GitLab

- Mix of files being stored in Git repo and LFS object store = not a clean solution.

+ Large files have now their entire history saved in Git LFS.

+ Size of Git database [.git/objects] truly reduced.

- History completely changed: everyone has to reset their copy of the Git repo.



# The git lfs migrate command

```
git lfs migrate import --include=<file name or pattern> --everything
```

- List of files or file patterns to “import” into Git LFS.
- Entries in `.gitattributes` will be automatically created.
- Multiple patterns/files can be specified by separating them with a comma, e.g.: `--include="*.fasta,*.img"`

This option tells git LFS to process all (local) branches of the repository.

## Example:

```
git lfs migrate import --include="*.fasta,*.img" --everything
```

```
git lfs ls-files  
702c4c3a56 - logo.img  
6f0a4add2f - sequences_A.fasta
```

After the `migrate import` command completes, LFS-tracked files in the working directory are replaced with their pointer (indicated by the “-”).

```
git lfs checkout  
git lfs ls-files  
702c4c3a56 * logo.img  
6f0a4add2f * sequences_A.fasta
```

The content of the files can be restored with `git lfs checkout`.

# The git lfs migrate command



A couple of warnings...

## History overwrite warning !



The `git lfs migrate import` command rewrites the entire history of your repository!

- Updating a remote repo with the changes requires a `git push --force`.
- Coordinate this operation with other people working on the repo.

## Data loss warning !



- Never run `git lfs migrate import` with a non-clean working directory. All your uncommitted changes will be lost (true story)!
- To be on the safe side, it's best to **make a full copy/backup of your Git repository before running the migrate command**. In this way, should anything go wrong, you can restore your repository from your copy.



## Behind the scenes...

- Git LFS stores the tracked files in the LFS cache [`.git/lfs/objects`] rather than in the Git repo [`.git/objects`].
- A lightweight “pointer” file is saved in the git repository.

Example of “pointer” blob objects stored in the Git repo [`.git/objects`]

```
.git/objects/d4/c3cf36a1c6865ba5e4d6e82e937dc835006231  
.git/objects/a3/3c8a78275c0763d964b3a2b0facdf5909b58c3
```

126 bytes

125 bytes

```
git cat-file -p a33c8a78
```

```
version https://git-lfs.github.com/spec/v1  
oid sha256:e6aa57987e7b8340dbf0ed1f4e5f90cf58a1a98de2d7a860aeed178ea4e734b4  
size 21852324
```

```
version https://git-lfs.github.com/spec/v1  
oid sha256:c1d5ab0faf552cdb3a365347093abc42a4e65718348e17eaad1584d650ae7aa6  
size 6010948
```

The actual files are stored in the Git LFS cache [`.git/lfs/objects`]

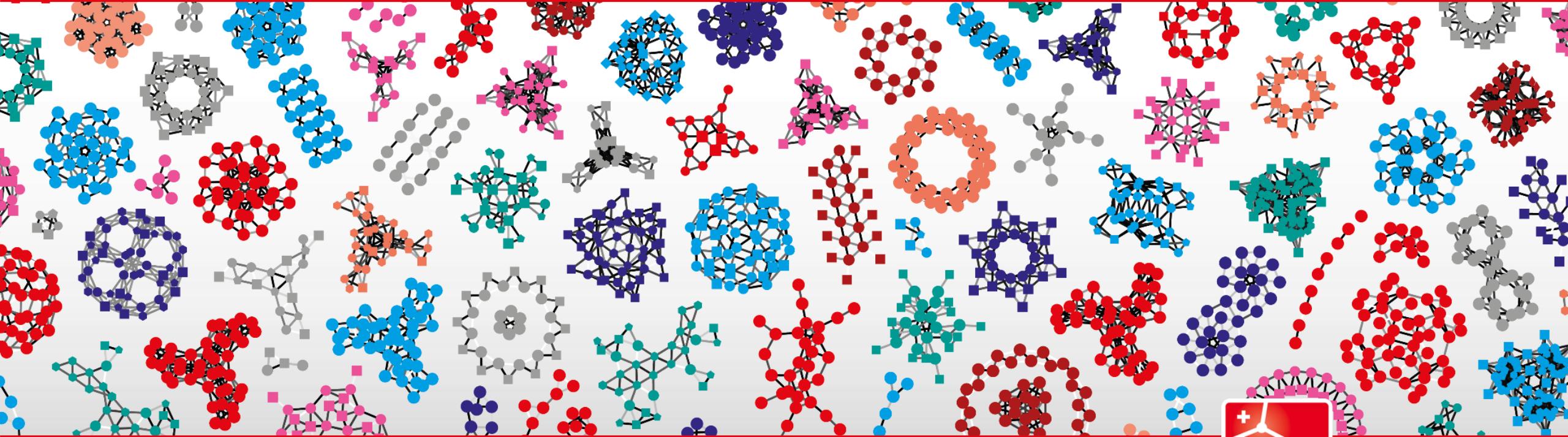
```
.git/lfs/objects/e6/aa/e6aa57987e7b8340dbf0ed1f4e5f90cf58a1a98de2d7a860aeed178ea4e734b4
```

21.8 MB

```
.git/lfs/objects/c1/d5/c1d5ab0faf552cdb3a365347093abc42a4e65718348e17eaad1584d650ae7aa6
```

6 MB

# exercise 6 B



SIB

Swiss Institute of  
Bioinformatics

Thank you for attending this course