Working with GIT

Florido Paganelli
Lund University
florido.paganelli@hep.lu.se
Fysikum, Hus A, Room 403

Support:
- send me an email or use Canvas
- personal Zoom room: https://lu-se.zoom.us/j/2485752983

Or use github!

Warning

DO NOT COPY PASTE COMMANDS

these exercises are meant for

typing on the keyboard!

(And some typesetting characters and symbols are not accepted by the terminal anyway)

Software

- **Required:**
  - **Git** - a free and open source distributed version control system
- **Optional:**
  - **gitk** – a fast git repository viewer
    - Available on Aurora as software installed using `module`
  - **tig** – a text repository browser
    - Not available on Aurora
  - There are many more even better! the above it’s only my taste.
- Installation:
  Debian/Ubuntu: `apt install git gitk tig`
  RedHat/Fedora: `yum install git gitk tig`

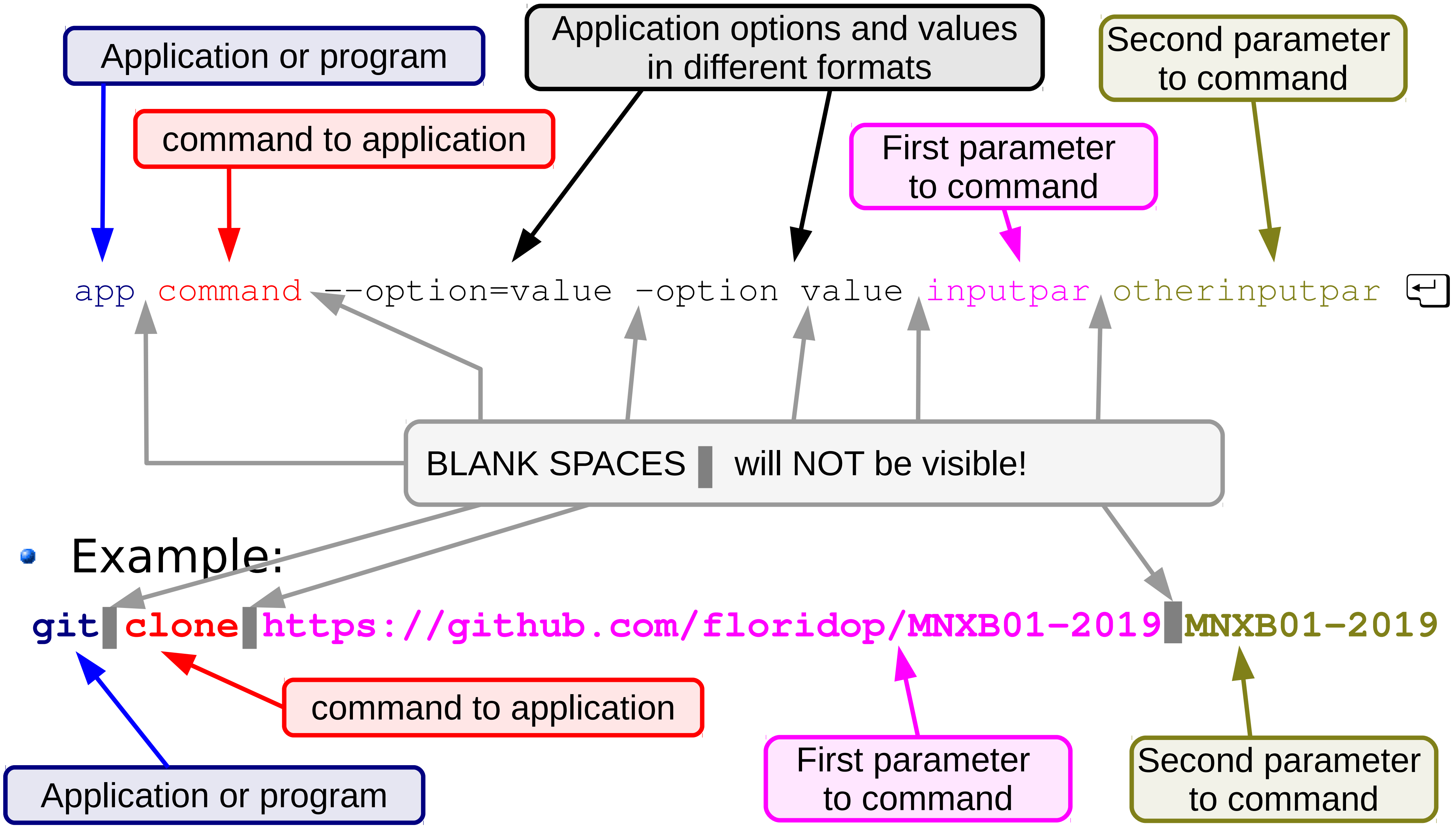
Platform	Package names
Ubuntu, Debian	**`git, gitk, tig`**
RedHat, CentOS, Fedora, SuSE	**`git, gitk, tig`**
Windows, MacOS	https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Outline

- What are version/revision control systems
  - Generic concepts of version/revision systems
- git
  - Generic concepts of git
  - git tutorial part 1: own repository
  - git tutorial part 2: collaborative repository
  - Useful git commands
  - Additional material

Notation

- I will be using the following color code for showing commands:



Application or program
command to application
Application options and values in different formats
First parameter to command
Second parameter to command
BLANK SPACES ▌ will NOT be visible!
command to application
Application or program
First parameter to command
Second parameter to command

```
app command --option=value -option value inputpar otherinputpar ↵
```

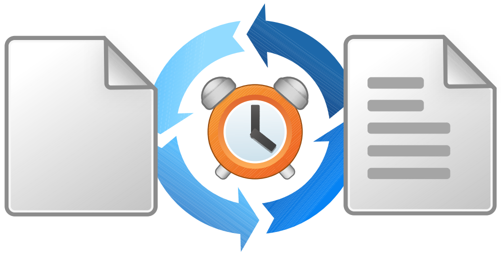
- Example:

```
git clone https://github.com/floridop/MNXB01-2019 MNXB01-2019
```

Revision systems concepts

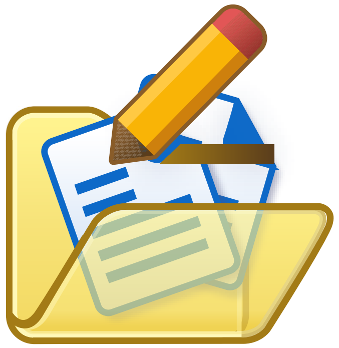
Why version/revision systems?

- Say you wrote some computer program in a text file.
- You discover a bug, something that does not work as it should, and you want to change it.
- You fix the bug, save the file. Try the program again and... **it doesn't work anymore!**
- **What went wrong?** Would be nice if you could **compare** what you **changed**...
- **Solution:** make a backup copy <u>before every (important) change</u>!
- Version systems make it easy to backup and compare **changes**



Why version/revision systems?

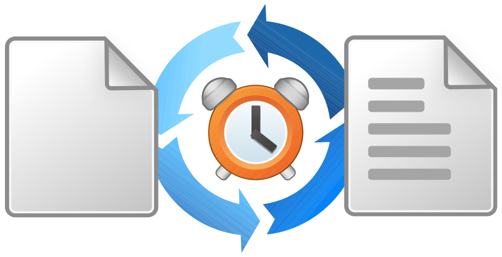
- If you do *many changes*, you might not remember what changes you made. Version systems allow you to attach a **comment** to the change.

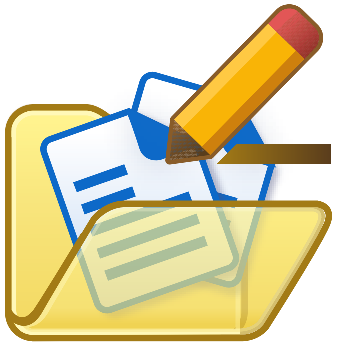


- If you want to *share* your code with *other developers*, it's nice if everybody can see who changed what. Version systems allow you to **author** the changes so the others know what you're done. This helps **sharing** code.

Why version/revision systems?

- Summary:
  - **Backup** each change in your code
  - **Compare** different versions of your code
  - **Comment** and annotate each change
  - **Share** among developers





Version systems: products and features

Product	staging	Local commit	diff	Fork/branch management	Distributed/ Collaborative	Compatibility
CVS (Current Version Stable)	N	N	Y	Y	N	?
SVN (SubVersioN)	N	N	Y	N	N	?
Git	Y	Y	Y	Y	Y	SVN CVS

There are others out there: Mercurial, Darcs, Monotone, Bazaar...

Git: vocabulary and concepts

What and why git

- Was created by Linus Torvalds especially for kernel development
  - Highly distributed community contributions
  - Lots of people *forking* (later I'll explain this term) and writing their own version of drivers
- Nowadays there are many collaborative websites systems that use it to share code (github, gitlab) and make it easier to integrate everyone's work with discussion and code revision/testing tools
- Is being used by many because is a free solution that helps distributed cooperation
- Becoming the most used among research projects
  - In other words, mostly **fashion**
- Good for text, **not good for images/archives/executables**… use dropbox or similar cloud storage for that.

Git ain't the best.

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

https://xkcd.com/1597/

Why using git in this course

- Aurora will not exist forever, and your home folders will disappear. If you want to keep your code after the course, you can move it into some cloud service. For example, you can put it on *github*.
- Today’s job market in IT and scientific programming is no longer based on your studies or experience. Most companies check if you wrote code for this and that library or framework **by checking github** or similar code sharing platforms. Better to start early!
- Suggestion: at the end of each tutorial, **push your changes to the remote github repository** we will create in the Homework.
  - If you are concerned about privacy, you can create a **private** github repository.
- **The final course project** material you will create **can be only handed out using a github repository**, so get familiar with git!

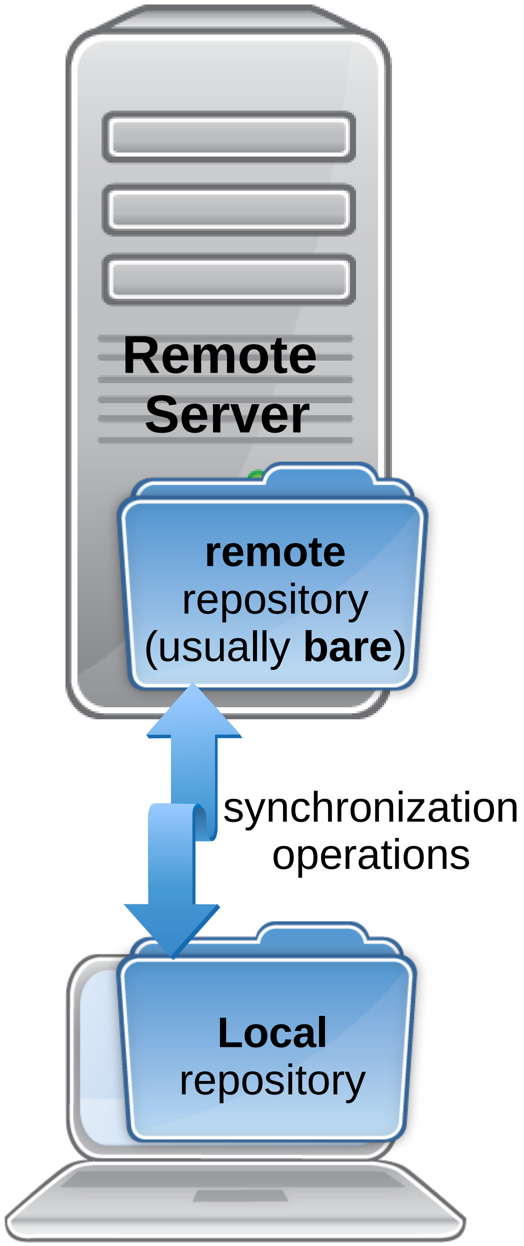
Scenarios and goals

- Scenarios (from less complicated to more):
  - 1: personal project, one user, **no** server or offline repository (see end of slides)
  - 2: personal project, one user, **single** server / repository (tutorial part 1)
  - 3: **community** project, multiple users, **multiple** servers / repositories (tutorial part 2 / homework)
- Goals:
  - track changes and version files
  - have a backup of files and tracked changes on a remote server
  - establish a way of synchronizing work between a user and the server or multiple users in a community

git tutorial part 1
scenario 2:
track and version your own code
using a remote repository

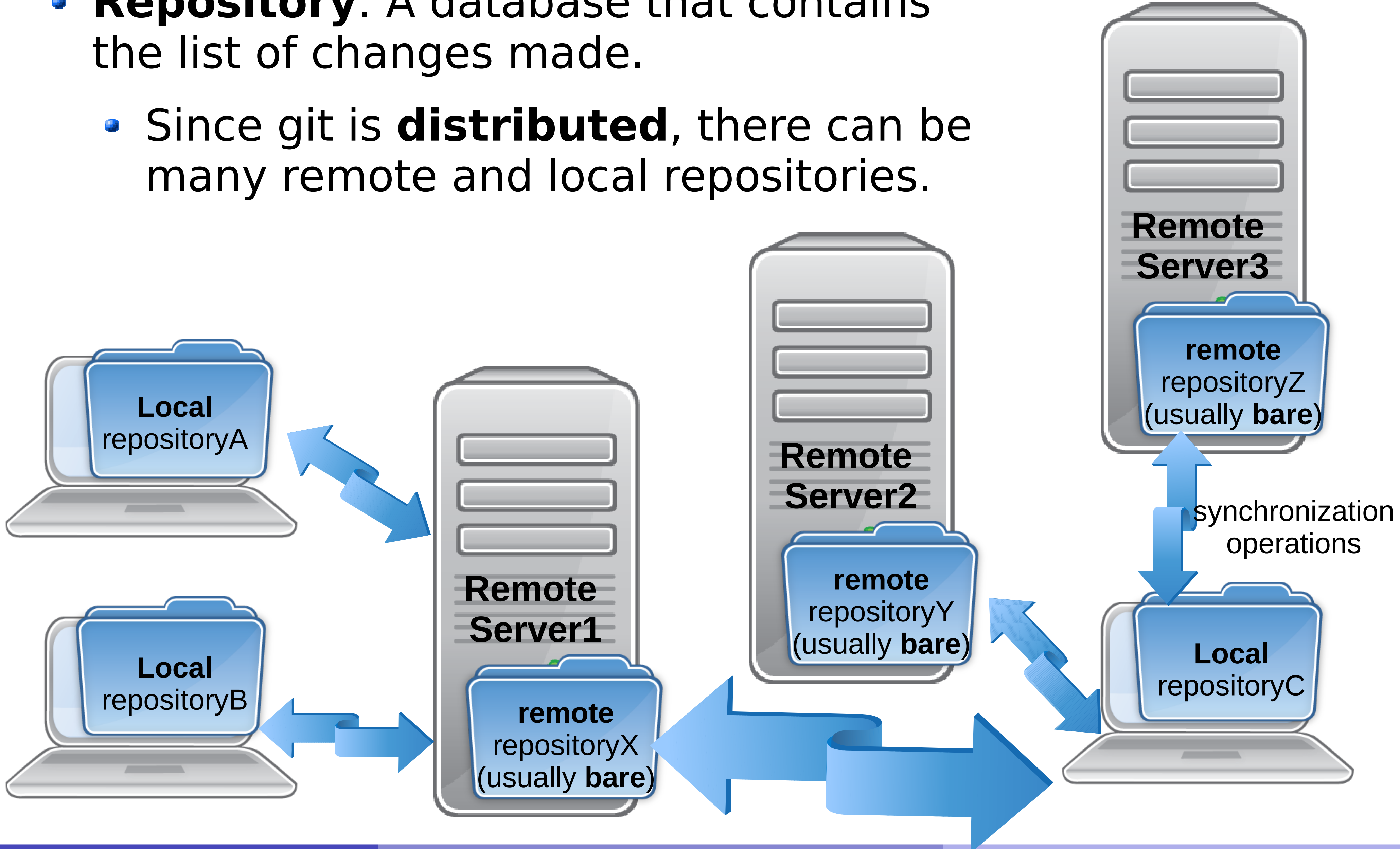
Concepts of version systems in git

- **Repository**: A database that contains the list of changes made.
  - A **local** git repository is shared *locally on your machine* in the **`.git`** invisible folder
  - A **remote** git repository is shared on a *remote server* and can be reached using a URL, like `https://github.com/floridop/MNXB01-2020.git`
  - A **bare** git repository can be stored in any folder and contains data in a form that only the git code understands. Can be used to have multiple copies of the same repository. It can be used to share a repository without GitHub or similar services.



Concepts of version systems in git

- **Repository**: A database that contains the list of changes made.
  - Since git is **distributed**, there can be many remote and local repositories.


Remote Server3
remote repositoryZ (usually bare)
synchronization operations
Remote Server2
remote repositoryY (usually bare)
Local repositoryA
Remote Server1
remote repositoryX (usually bare)
Local repositoryB
Local repositoryC

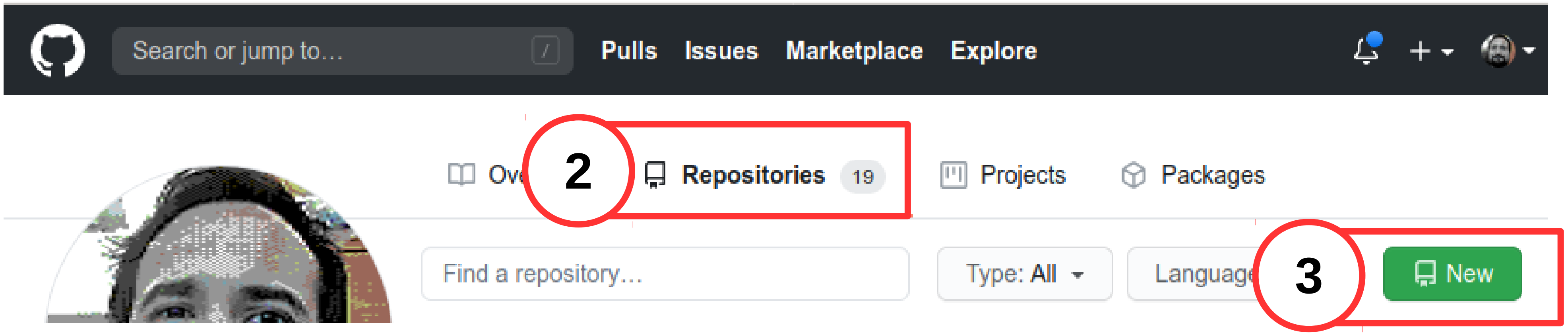
Servers and services: GitHub, GitLab

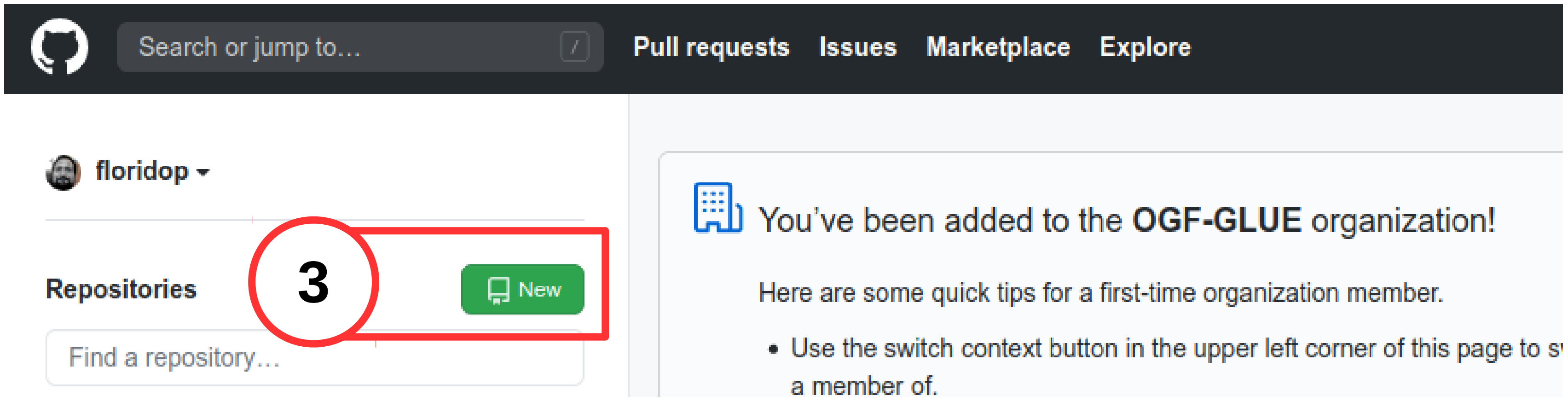
- **GitHub**, A cloud service that offers for *free*:
  - **hosting space** for git projects (they run the git server)
  - **A web interface** to collaborate on projects
- Acquired in 2018 by Microsoft, now offers also
  - Private projects (can't be seen by other users)
  - Enterprise services
- They claim they will not use your code except for the purposes of their service and that you retain all the copyrights on the code.
- It is **not** an open source project. https://www.github.com/
  - Free accounts
  - Paid dedicated server
- Open source alternatives: **Gitlab** https://about.gitlab.com/
  - Paid accounts
  - Free: run your own server
    - Nordic Project **CodeRefinery** offers free accounts to Nordic Researchers https://coderefinery.org/repository/

Octocat

Create your own repository on github 1/3

1) Login on https://www.github.com
   You may see one of the two views below. Graphics may differ a bit.
2) Click on “Repositories” (maybe not needed)
3) Click on “New”





You’ve been added to the **OGF-GLUE** organization!

Here are some quick tips for a first-time organization member.

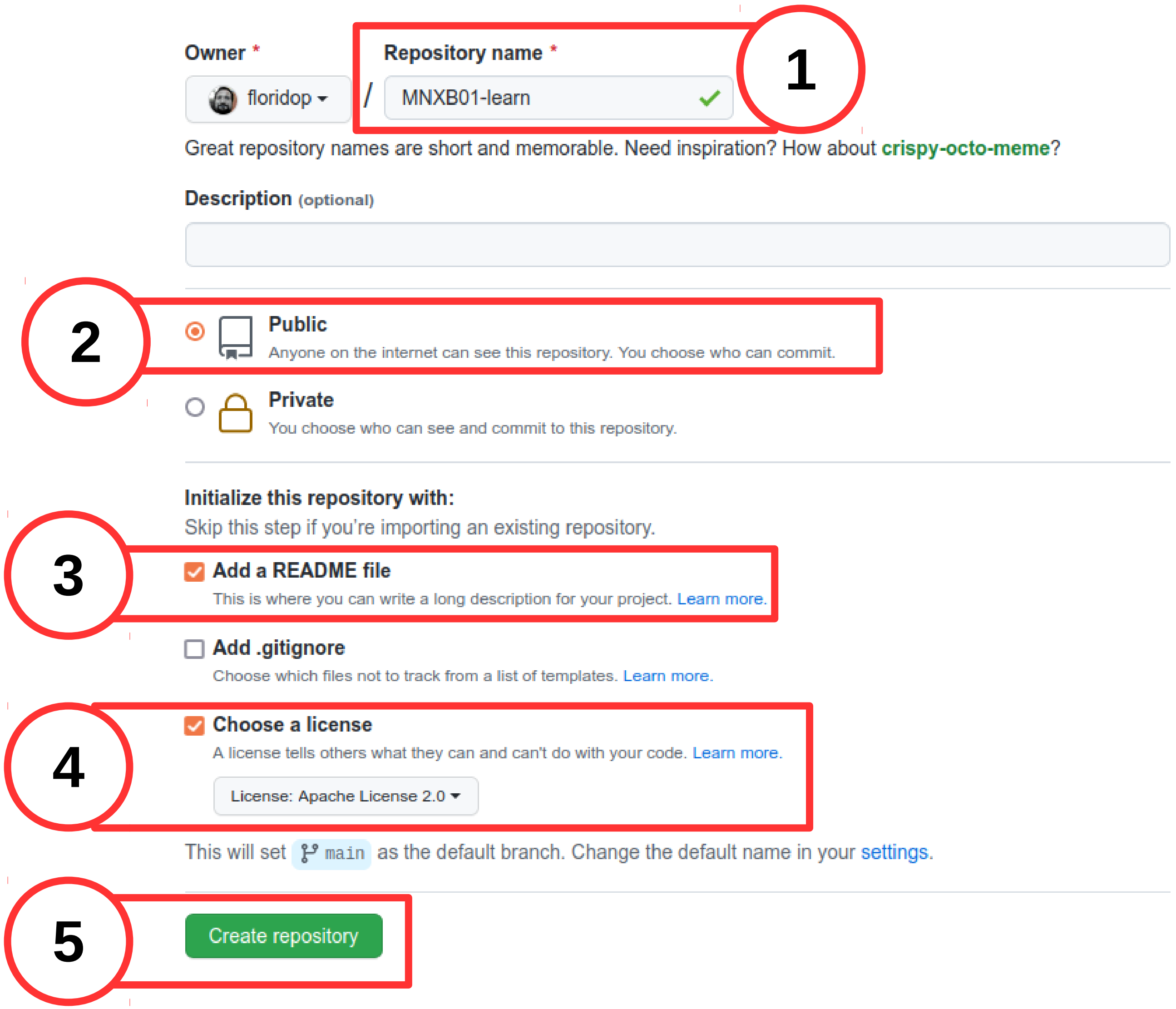
- Use the switch context button in the upper left corner of this page to s
  a member of.

Create your own repository on github 2/3

Fill the blank with this information (see also pic in next slide):

1) Repository Name:
   - MNXB01-learn
2) Make it a “Public” repository.
3) Choose a license:
   - Apache License 2.0
   - Just because it’s one that gives you some control on the code, no real reason for this tutorial.
4) Click on “create repository”

Create your own repository on github 3/3



Owner * floridop / Repository name * MNXB01-learn

Great repository names are short and memorable. Need inspiration? How about crispy-octo-meme?

Description (optional)

[✓ Public] Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

[✓] Add a README file
This is where you can write a long description for your project. Learn more.

[ ] Add .gitignore
Choose which files not to track from a list of templates. Learn more.

[✓] Choose a license
A license tells others what they can and can't do with your code. Learn more.

License: Apache License 2.0

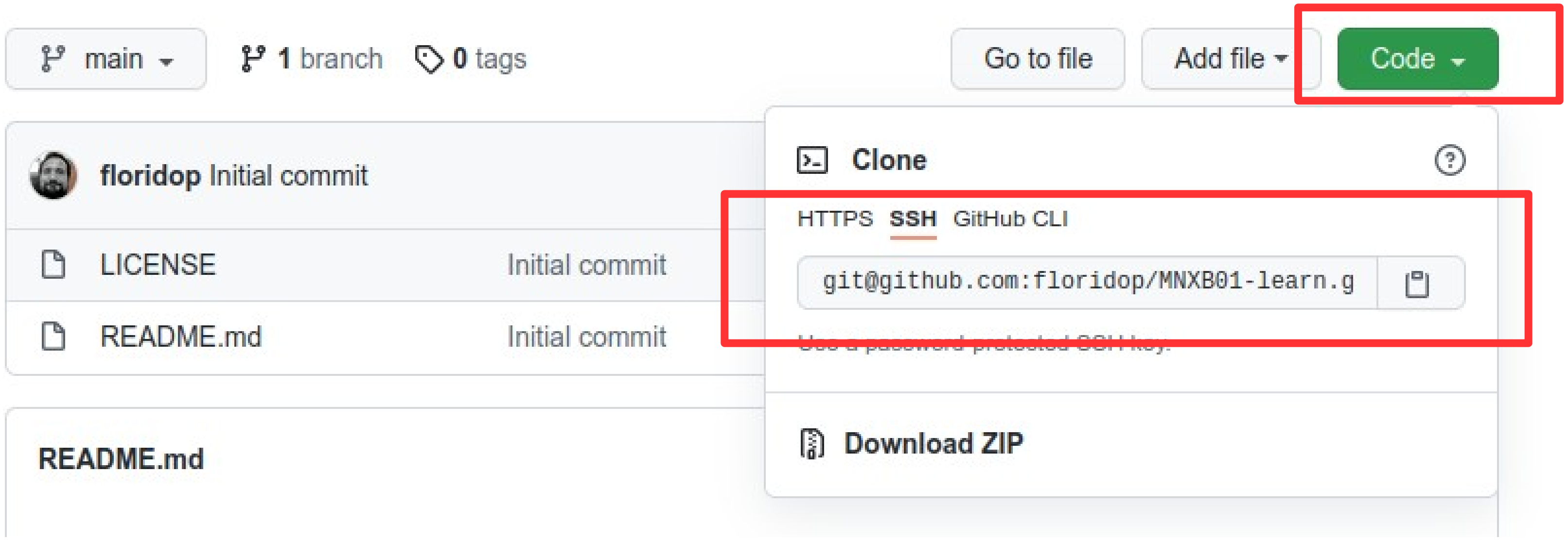
This will set `main` as the default branch. Change the default name in your settings.

Create repository

Retrieve the repository URL

1) Click on “code”
2) Select SSH
3) Copy the repo URL (a link) in the field that appears. It looks like:
   - git@github.com:YOURUSERNAME/MNXB01-learn.git
4) Save it for the coming tutorial steps!

- Leave the page open or copy paste the URL in Pluma



floridop Initial commit

LICENSE Initial commit

README.md Initial commit

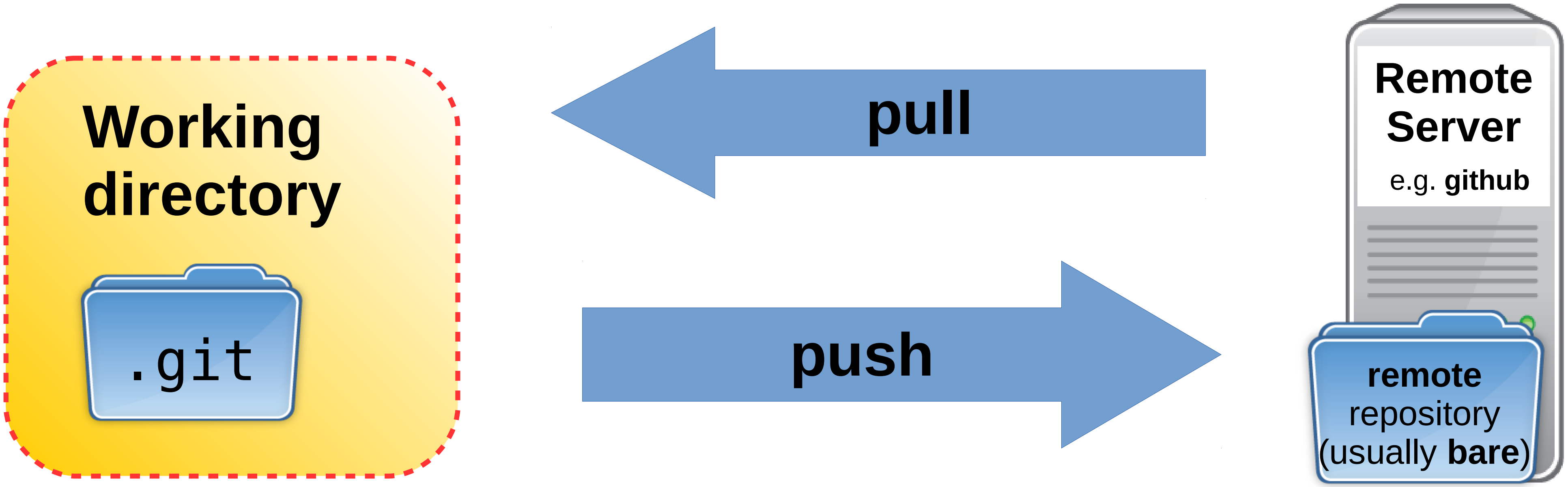
README.md

```
git@github.com:floridop/MNXB01-learn.g
```

Use a password-protected SSH key.

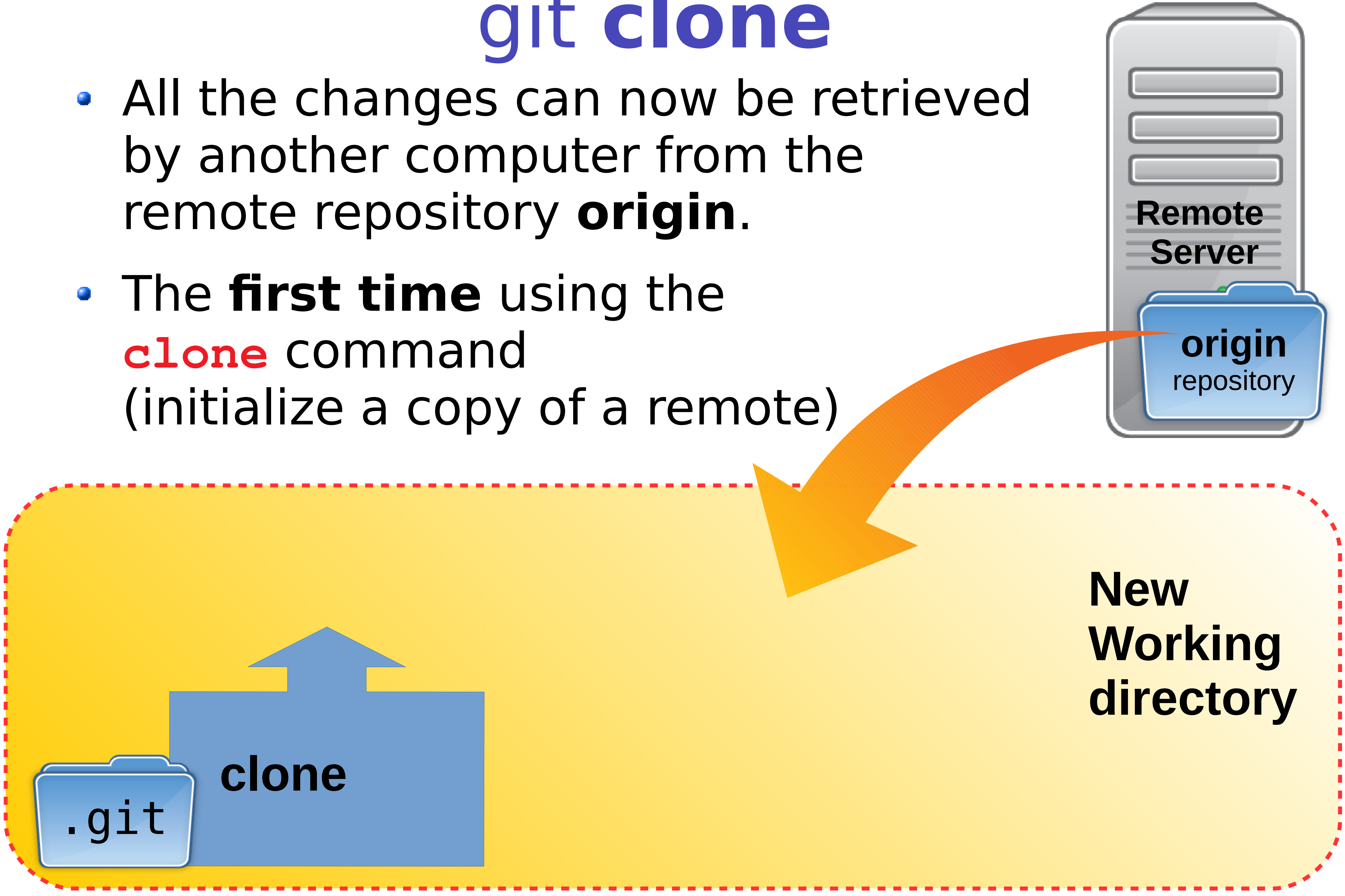
Concepts of version systems in git

- **Working directory**: the latest version of a set of files that you want to work on. This is usually **local** to your machine.
  - It is usually the result of a **clone**, an exact copy, of some remote repository
  - You can synchronize the local git repository with remote ones using the **push** (send changes) and **pull** (retrieve changes) commands.
  - A bit like DropBox but NOT automatic.



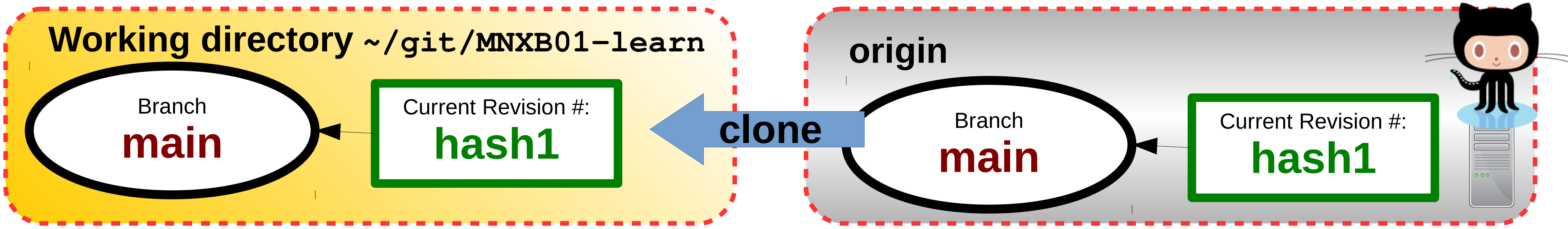
Concepts of version systems
git **clone**

- All the changes can now be retrieved by another computer from the remote repository **origin**.
- The **first time** using the `clone` command (initialize a copy of a remote)



Clone your repository on Aurora

- create a folder named git in your home and access it.
  - `mkdir ~/git`
  - `cd ~/git`
- Copy the link shown in the github webpage page after the creation
- Clone your remote repository into a local working copy
  - `git clone git@github.com:YOURUSERNAME/MNXB01-learn.git`
  - You can copy paste the URL in the terminal with Ctrl ⇧ C
  - A window or a request might appear asking to input a password. If everything is configured well, it will be your private SSH key password which you created during the preparation for the tutorial steps.
  - If the system asks you to accept the server SSH fingerprint, check this webpage to validate the fingerprint: https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/githubs-ssh-key-fingerprints
- It will be created in a subfolder with the same name `MNXB01-learn`.
- cd into it:
  - `cd MNXB01-learn`



.git database, remotes and branches

- The result of a git clone operation is the creation of a copy of your **remote** repository, by default called **origin .** You can have multiple remote repositories. You can inspect the name and URL of your *remotes* with
  - **git remote -v**
    ```
    origin git@github.com:floridop/MNXB01-learn.git (fetch)
    origin git@github.com:floridop/MNXB01-learn.git (push)
    ```
- The database of changes are kept in a hidden directory called .git . You can see it with
  - **ls -a**
    ```
    .  ..  .git  LICENSE  README.md
    ```
    ls .git/
    ```
    branches         description  hooks  logs       packed-refs
    COMMIT_EDITMSG   gitk.cache   index  objects    qgit_cache.dat
    config           HEAD         info   ORIG_HEAD  refs
    ```
- The git command accepts **subcommands** to do operations on the database.
- A **brand new git repository database** is created with the command **init.** In this case github ran this command for you in the cloud. See slides at the end of this presentation for examples.
- A brand new git repository always starts with a **branch** called **main** (formerly *master*). You can see the branches in your repository with the command
  - **git branch**
    ```
    * main
    ```
  - The asterisk * identifies the active branch we are currently working on. There can be only one active branch at a time.
  - A branch identifies a collection of files and their versions/revisions. Let's understand the revisions, but first we need to do some configuration steps.

Branch:
main

Configuring git 1/2

- Git must be configured for your personal data so that the **authoring** information in the commits can be added.
- This can be done globally or for each specific repository
  - the `--global` options are stored in your home `~/.gitconfig`
  - the `--local` options are stored in the repository's `.git/config` folder
- For this course let's configure your name, email and favourite text editor:
-
```
git config --global user.name “Name LastName”
git config --global user.email youremail@your.domain.blah
git config --global core.editor pluma
```

Configuring git 2/2

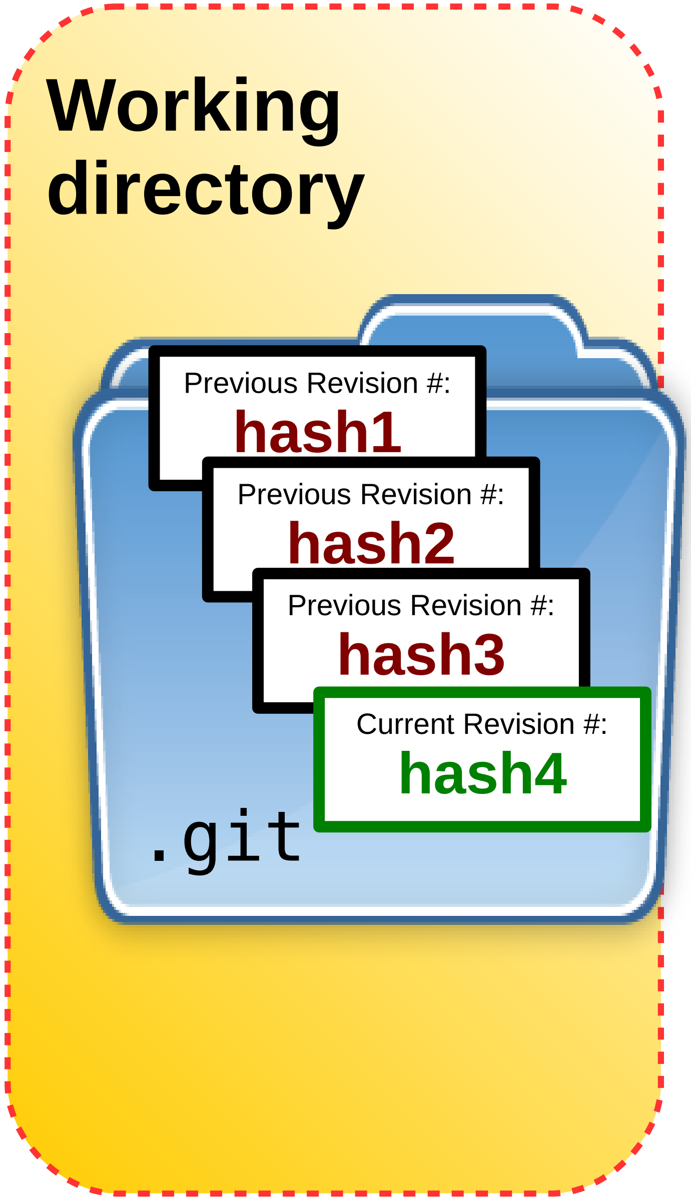
- One can inspect the config setup with:
- `git config -l`

```
user.name=Test Student 5
user.email=some.mail@testdomain.info
core.editor=pluma
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
remote.origin.url=git@github.com:floridop/MNXB01-learn.git
branch.main.remote=origin
branch.main.merge=refs/heads/main
```

Concepts of version systems: revisions/versions/commits

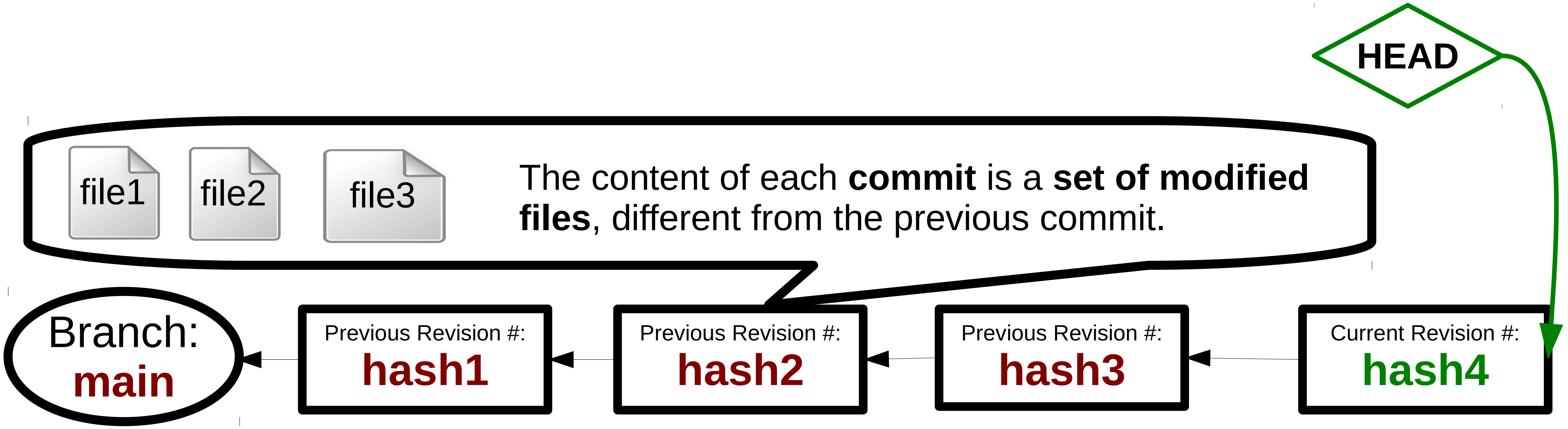
- When one is happy with the changes they made, it records them in the database by doing a **commit**
- A *committed* set of files is called a **revisions** and gets a **commit ID**: every “version” of one or more files gets a **revision tag**. This can be a number, a label, a string.
- In git usually is an **hash***, a strange sequence of symbols. It:
  - **Identifies the repository and other details** of when the changes where made
  - It’s **universally unique**, everywhere in the world that commit will represent a defined sequence of changes.
  - For this reason these systems are also known as **Revision Systems**, as every revision gets a **label** that depends on **time** and **person** who made the change.

*Hash: a special injective function that returns a value from a finite set of strings. The return values are unique under certain conditions.



Concepts of version systems git basic terminology

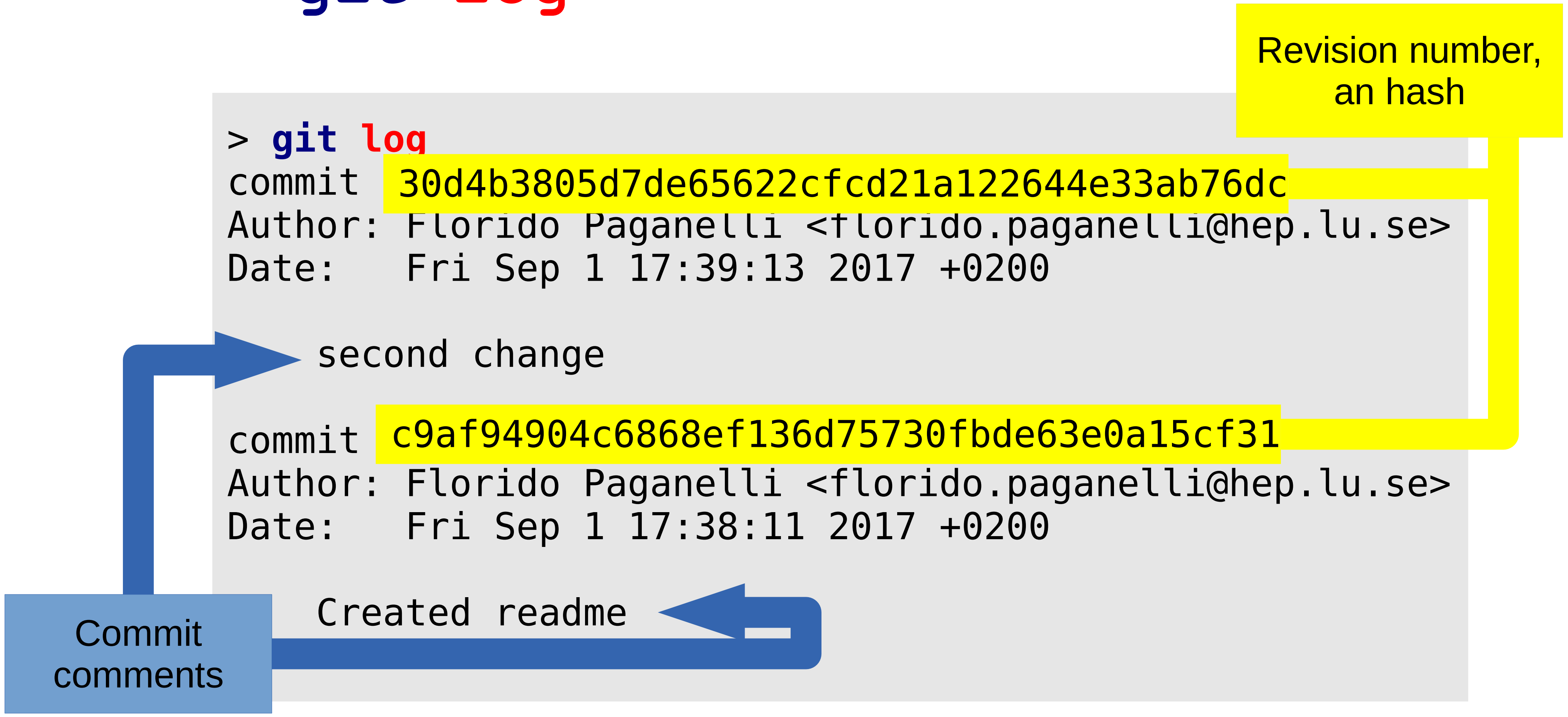
- **For every set of changes there is a `commit`.** Every commit generates a *new revision* with a different **hash**. This can be represented as an ordered graph like the one below. For every committed **change** a new hash.
- The latest commit hash is called **HEAD**.


HEAD
file1
file2
file3
The content of each commit is a set of modified files, different from the previous commit.
Branch: main
Previous Revision #: hash1
Previous Revision #: hash2
Previous Revision #: hash3
Current Revision #: hash4

Git log, commit history, revision numbers

- All the commit history with you messages can be browsed using the command

  git log



```
> git log
commit 30d4b3805d7de65622cfcd21a122644e33ab76dc
Author: Florido Paganelli <florido.paganelli@hep.lu.se>
Date:   Fri Sep 1 17:39:13 2017 +0200

    second change

commit c9af94904c6868ef136d75730fbde63e0a15cf31
Author: Florido Paganelli <florido.paganelli@hep.lu.se>
Date:   Fri Sep 1 17:38:11 2017 +0200

    Created readme
```

Git log, commit history, revision numbers

- To see which files have changed for each commit (**A**:Added **M**: Modified **D**: Deleted...):

  git log --name-status

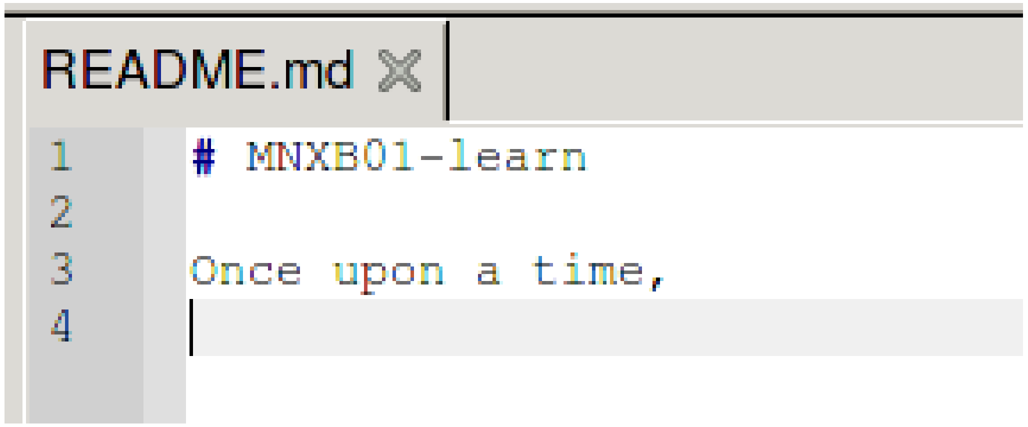
```
> git log --name-status
commit fced0d917580764b9bc72060233c60f77840a0a7 (HEAD -> main)
Author: Florido Paganelli <florido.paganelli@gmail.com>
Date:   Sat Sep 12 18:56:08 2020 +0200

    added disclaimer about input paths

M       code/smhicleaner.sh.pseudocode
M       solution/smhicleaner.sh

commit 7cd1c062daffb615a9f0d6f60d9a83dcc29e26e5
Author: Florido Paganelli <florido.paganelli@gmail.com>
Date:   Sat Sep 12 18:51:22 2020 +0200

    outputs of the command with and withour errors

A       result/output_copying
A       result/output_downloading
A       result/output_error_no_params
A       result/output_error_problems_downloading_or_copying
```

Looking at the commits: git log

- The first commit has been created by github when you ticked the “Create README.md” and chose a License.
- You can see the commits by using the command
  - **git log**
    ```
    commit 52b9bc84bf837add309437f8cd30c63521b3940c
    Author: Florido Paganelli <florido.paganelli@gmail.com>
    Date:   Tue Sep 15 13:23:04 2020 +0200

        Initial commit
    ```
- And in more detail which files have been added/changed/deleted (A/M/D) with
  - **git log --name-status**
    ```
    commit 52b9bc84bf837add309437f8cd30c63521b3940c
    Author: Florido Paganelli <florido.paganelli@gmail.com>
    Date:   Tue Sep 15 13:23:04 2020 +0200

        Initial commit

    A       LICENSE
    A       README.md
    ```

Let’s change the README

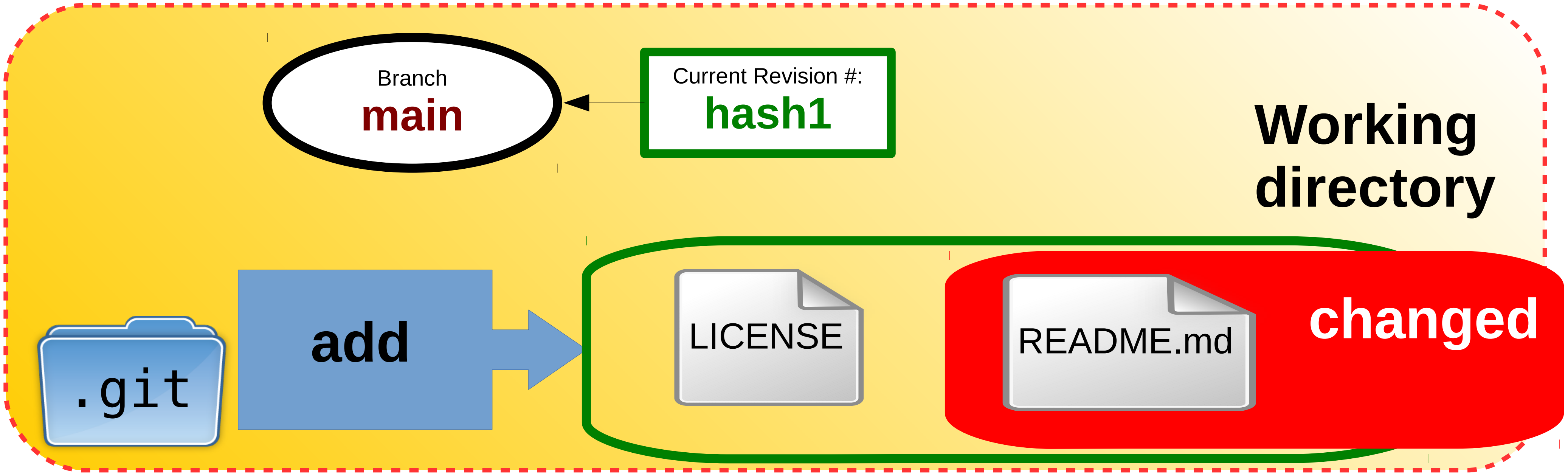
- We will write a fairy-tale. Code is boring!
- Open the geany editor to change the readme:
  - `pluma README.md&`
- Write a line “`Once upon a time,`” as in the picture and **save** (See MNXB01-manual.pdf if you don’t yet know how to do this!)



```
# MNXB01-learn

Once upon a time,

```

Concepts of version systems
git **status**

- If one modifies or changes files contained in a certain revision, git can see it, and reports to the user with the **status** command.
- Git gives the choice to **add** (include) these changes to the database.

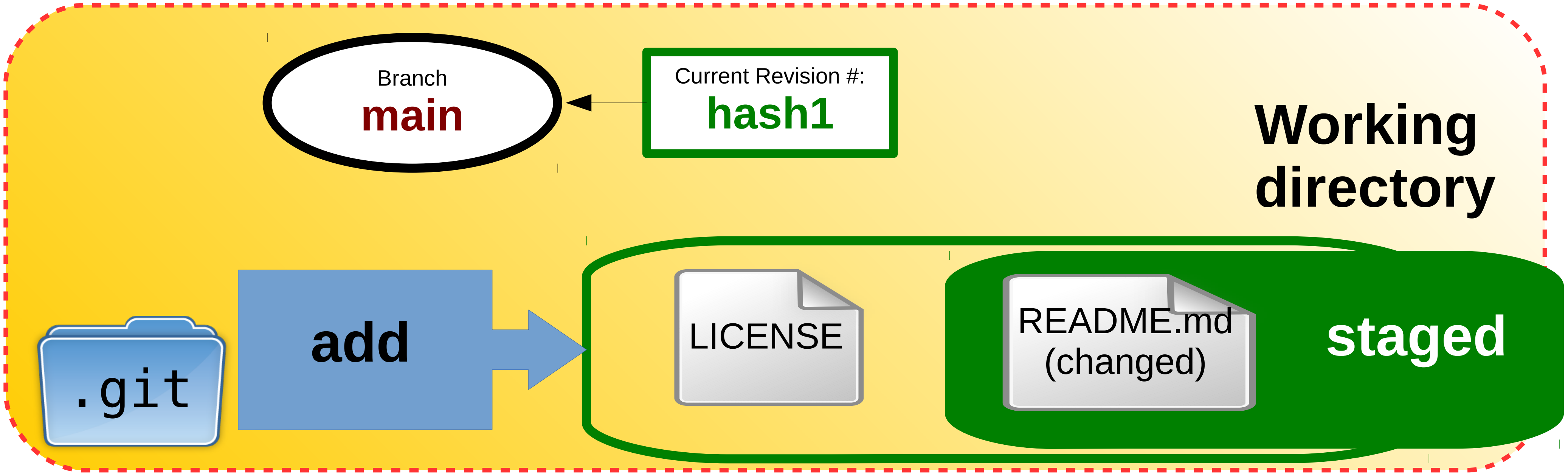


Concepts of version systems
git **status**

- Check the status of the repository with
  - `git status`
    ```
    # On branch main
    # Changed but not updated:
    #   (use "git add <file>..." to update what will be committed)
    #   (use "git checkout -- <file>..." to discard changes in
    working directory)
    #
    # modified:   README.md
    #
    no changes added to commit (use "git add" and/or "git commit -a")
    ```
- Let's add the file to be part of the next collection of changes, as git suggests above
  - `git add README.md`

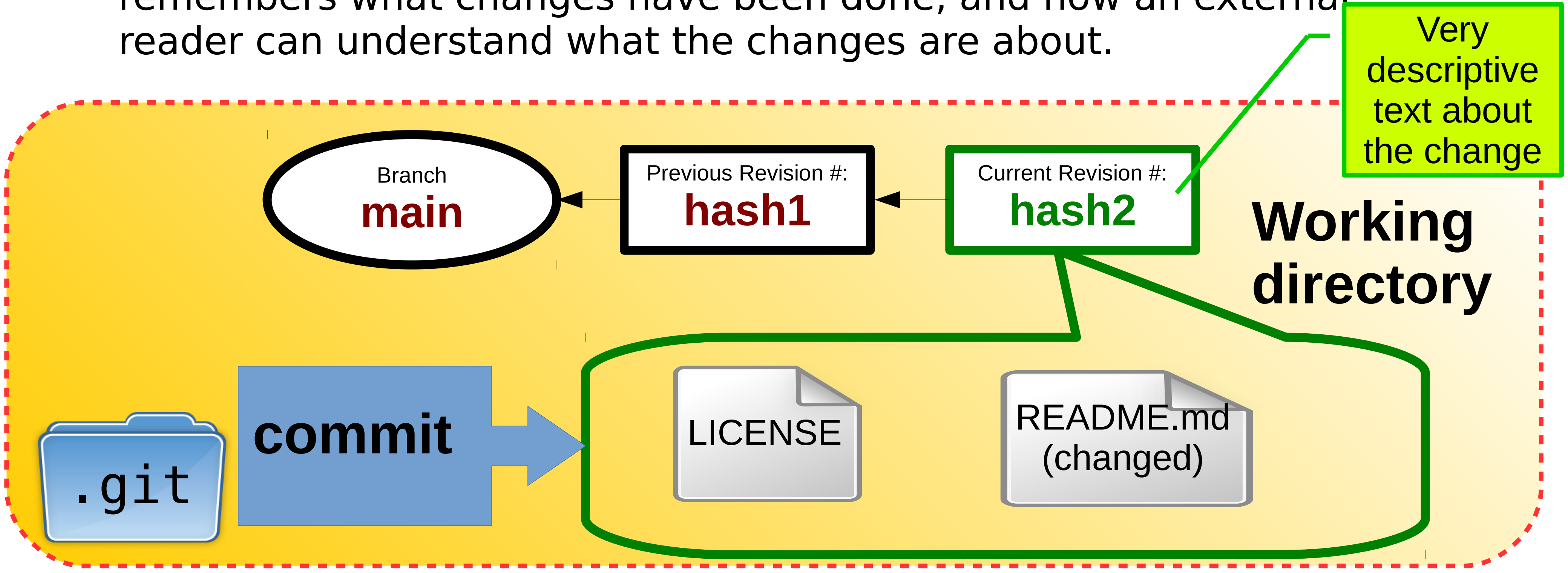
Concepts of version systems
git **add**

- Once files are **add**ed, they are *marked* to be part of a next revision, but they're not yet saved in the database.
- In git slang, they're **staged –** shortlisted to be part of the next commit.
- One may continue working, editing and keep staging other files that might be part of the same set of changes repeating this task.
- remember: staged files **are not yet versioned nor tracked**. They are not added to the changes database yet.



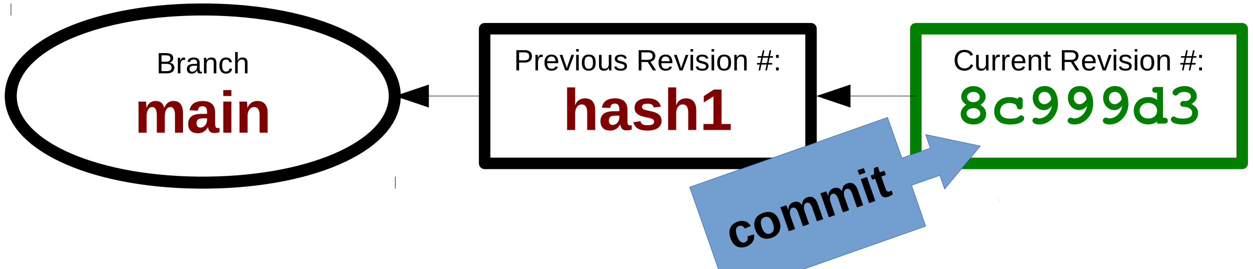
Concepts of version systems git commit

- *Staged* files will then be actually become part of a new revision in the database once the user **commits** them.
- A new commit will generate a new hash, tracking the set of changes in the database.
- When you commit you get the chance of **describing your commit with a comment**. It is **extremely important** that your commit **explains very well** what are the changes contained: this is the main way one remembers what changes have been done, and how an external reader can understand what the changes are about.



Check staged files and commit

- We can run git status again to check which files are staged:
  - **git status**
    ```
    # On branch main
    # Changes to be committed:
    #   (use "git reset HEAD <file>..." to unstage)
    #
    # modified:   README.md
    #
    ```
- README.md is ready to be added to the new set of changes. We can then commit and write an inline message with -m:
  - **git commit -m 'Added first of the fairy-tale'**
    ```
    [main 8c999d3] Added first line of the fairy-tale
     1 files changed, 3 insertions(+), 1 deletions(-)
    ```
  - Git shows a short version of the hash `(8c999d3)`, the comment, and a summary of changes.



Check committed changes

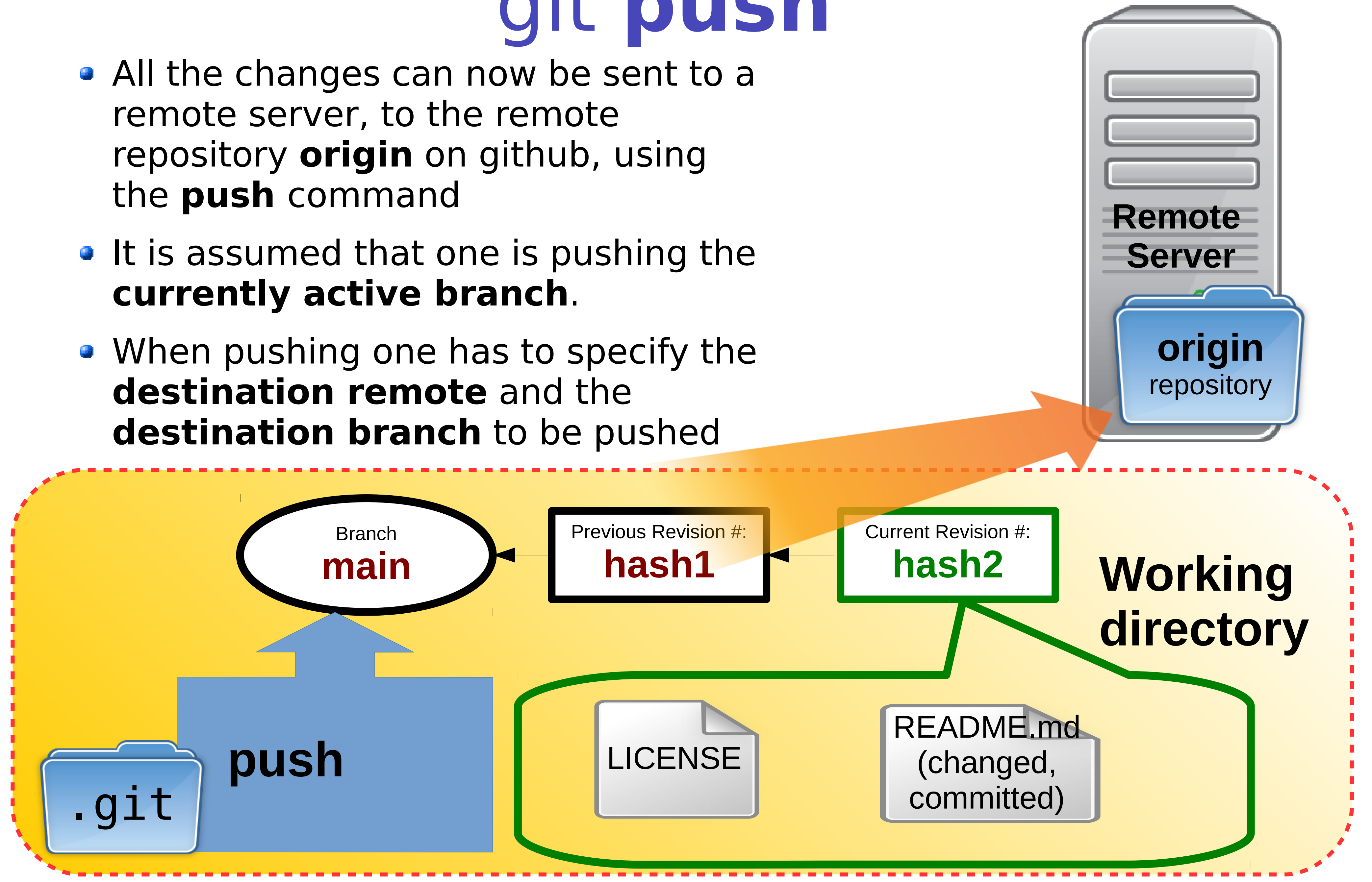
- We can run `git log` again to check the commits:
  - **git log --name-status**

```
commit 8c999d3896e8a7d01aa75c6601bc3e40e0cfa849
Author: Florido Paganelli <florido.paganelli@hep.lu.se>
Date:   Tue Sep 15 14:53:32 2020 +0200

    Added first line of the fairy-tale

M       README.md

commit 52b9bc84bf837add309437f8cd30c63521b3940c
Author: Florido Paganelli <florido.paganelli@gmail.com>
Date:   Tue Sep 15 13:23:04 2020 +0200

    Initial commit

A       LICENSE
A       README.md
```

- In the logs we can see all the metadata added to the database.

Concepts of version systems
git **push**

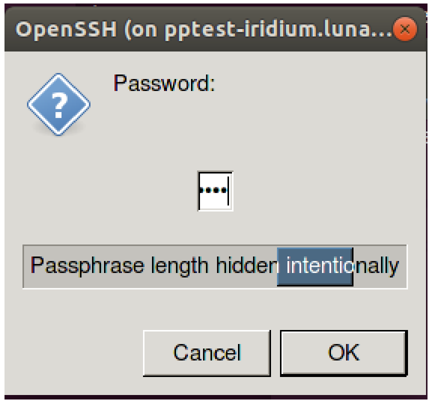
- All the changes can now be sent to a remote server, to the remote repository **origin** on github, using the **push** command
- It is assumed that one is pushing the **currently active branch**.
- When pushing one has to specify the **destination remote** and the **destination branch** to be pushed



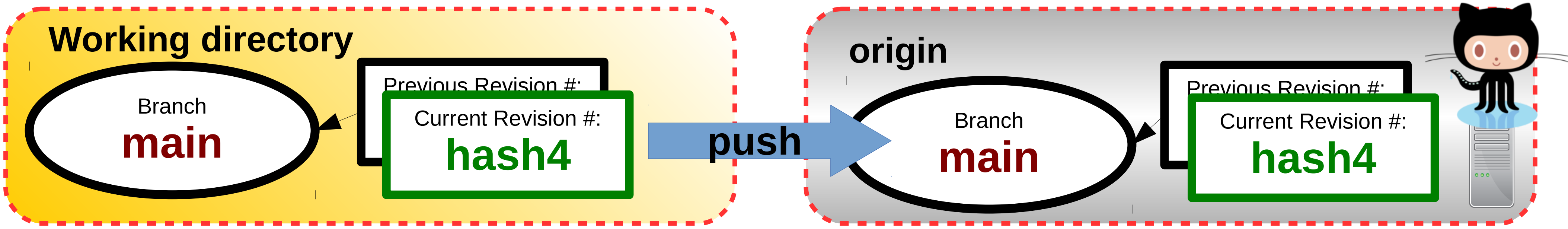
Pushing from working dir to github origin main

- We’re now ready to send our changes to the remote repository. It is important to select the correct remote and remote branch according to the branch we’re on. We’re on main hence we will push main. The system will ask for our ssh key password, which you have to input:

-
  ```
  git push origin main
  Counting objects: 5, done.
  Delta compression using up to 16 threads.
  Compressing objects: 100% (2/2), done.
  Writing objects: 100% (3/3), 338 bytes, done.
  Total 3 (delta 0), reused 0 (delta 0)
  To git@github.com:floridop/MNXB01-learn.git
     52b9bc8..8c999d3  main -> main
  ```

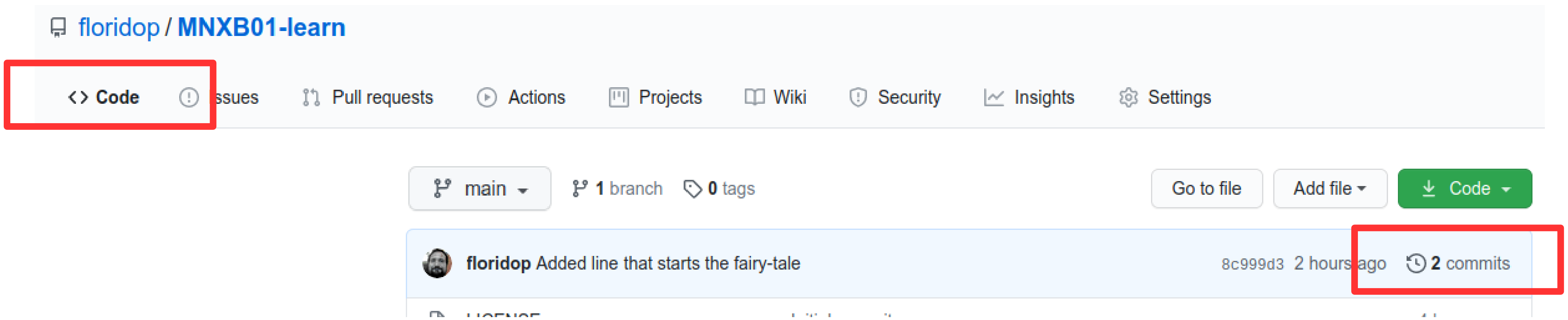

OpenSSH (on pptest-iridium.luna...
Password:
Passphrase length hidden intentionally
Cancel
OK

- Git will show some statistics about the upload and information about branch tracking.



Check the status of the repo on github

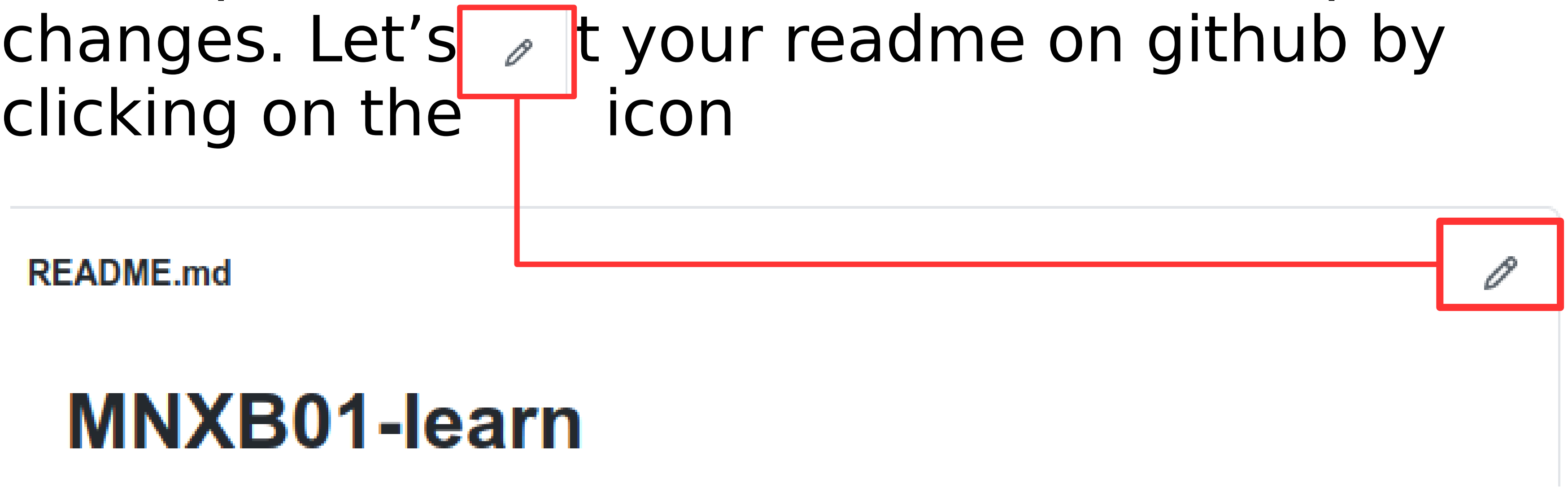
- Go back to the github page of your repo and refresh the page with F5. You should now see the modified README.md file.
- If you click on "2 commits" you can see the detail of commits
- You can always click on "<>Code" to go back to the main page of your repo.



floridop Added line that starts the fairy-tale 8c999d3 2 hours ago 2 commits

Making changes directly on github 1/4

- Github can render your readme file if you write it in a special formatting language called *MarkDown* (README.**md**)
- It interprets simple text and formats it as nice HTML.
- It also provides an online editor to make quick changes. Let's edit your readme on github by clicking on the icon



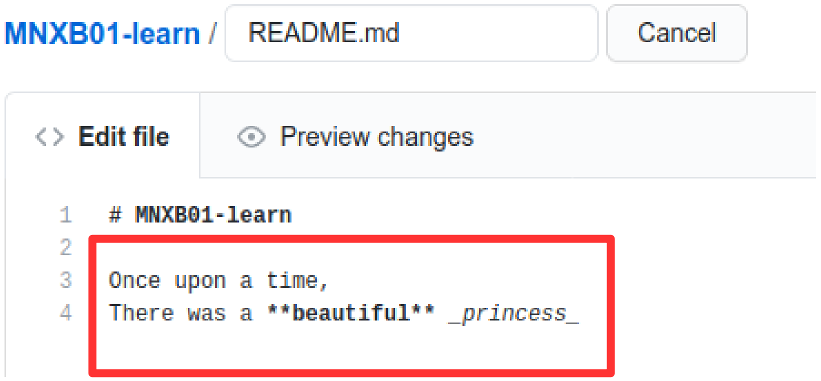
Making changes directly on github 2/4

1) Let's add an additional line of text to out fairytale:
   `there was a **beautiful** _princess_`
   - ** : bold
     _ : italic
2) You can preview the changes by clicking on "preview changes"
3) Once you're happy with the way the text looks, we're ready to **commit**.
   - In the "Commit Changes" area, write a significant comment like "Introducing the story character"
   - Then click on "commit changes"

- You can now check the commit **History** to see how your latest change has been recorded.

Making changes directly on github 3/4

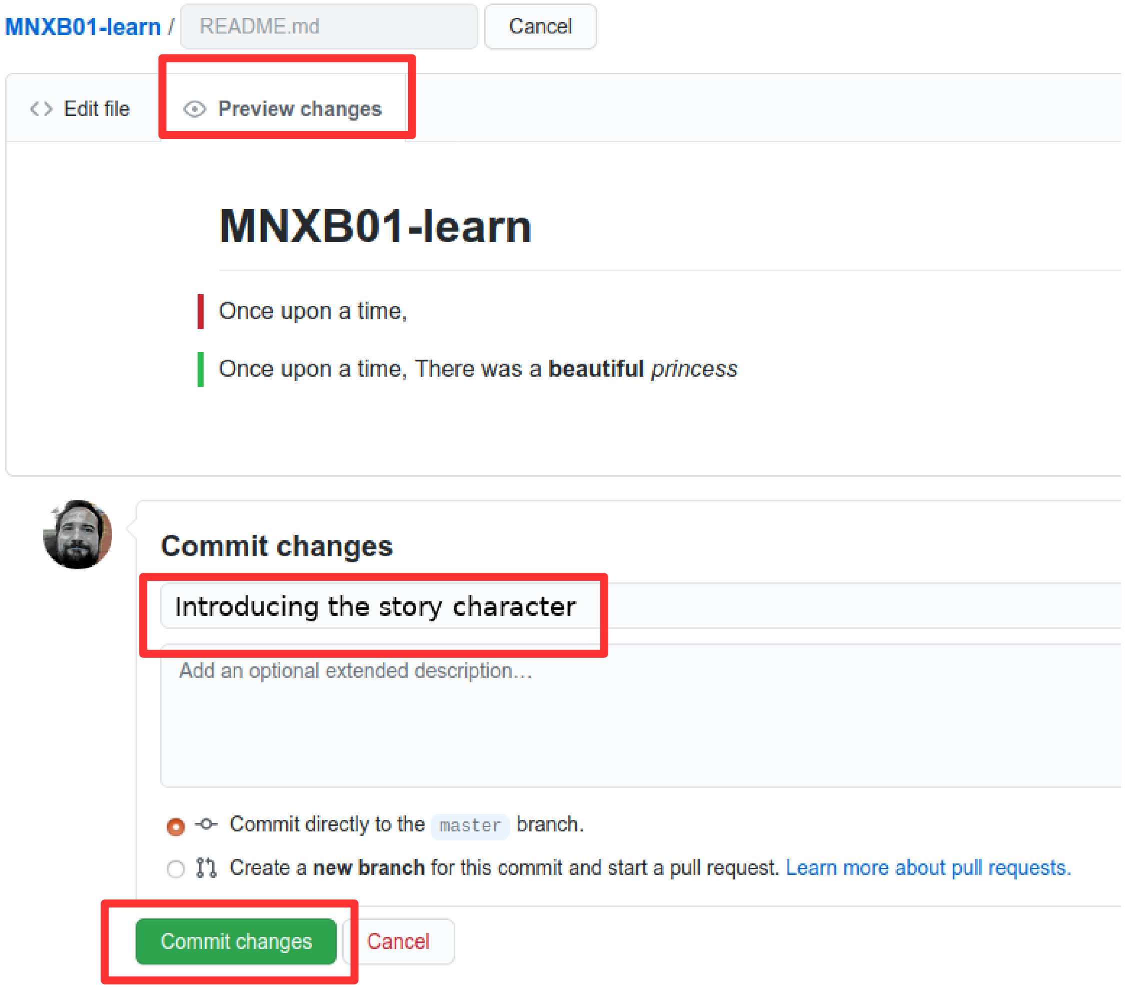
- In pictures:



MNXB01-learn / README.md Cancel

Edit file | Preview changes

```
# MNXB01-learn

Once upon a time,
There was a **beautiful** _princess_
```



MNXB01-learn / README.md Cancel

Edit file | Preview changes

MNXB01-learn

Once upon a time,

Once upon a time, There was a **beautiful** *princess*

Commit changes

Introducing the story character

Add an optional extended description…

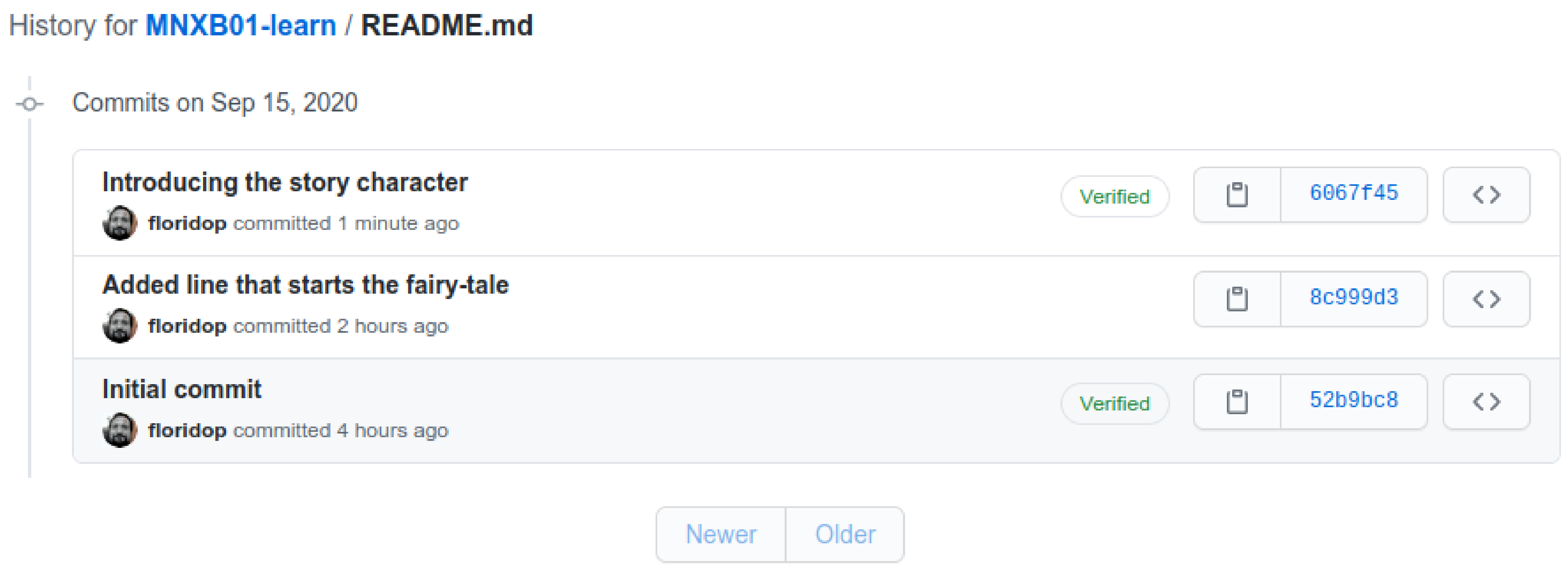
[✓ Commit directly to the `master` branch.]

Create a **new branch** for this commit and start a pull request. Learn more about pull requests.

Commit changes | Cancel

Making changes directly on github 4/4

- History


History for MNXB01-learn / README.md

Commits on Sep 15, 2020

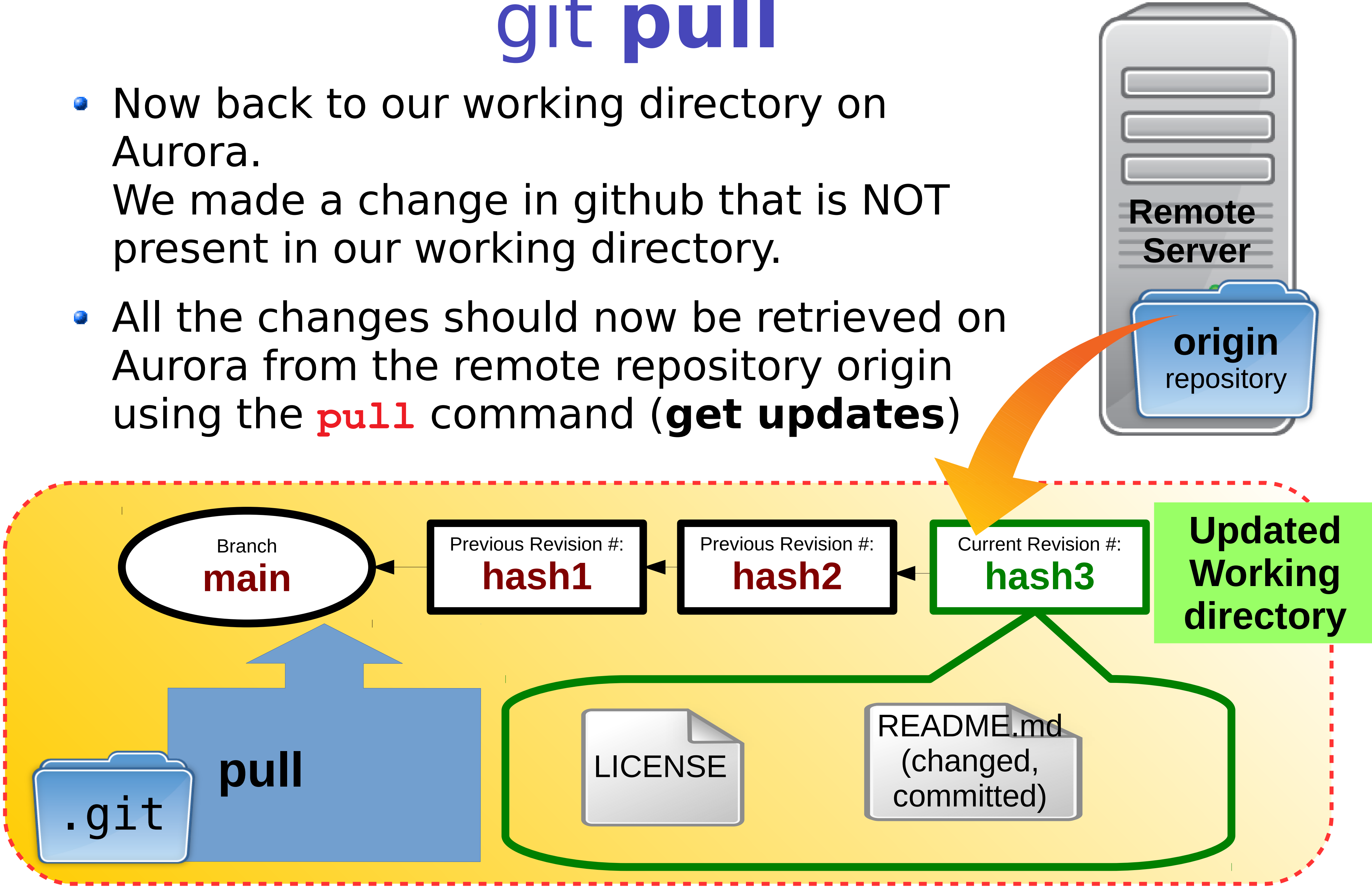
Introducing the story character
floridop committed 1 minute ago
Verified 6067f45

Added line that starts the fairy-tale
floridop committed 2 hours ago
8c999d3

Initial commit
floridop committed 4 hours ago
Verified 52b9bc8

Concepts of version systems
git **pull**

- Now back to our working directory on Aurora.
  We made a change in github that is NOT present in our working directory.
- All the changes should now be retrieved on Aurora from the remote repository origin using the `pull` command (**get updates**)

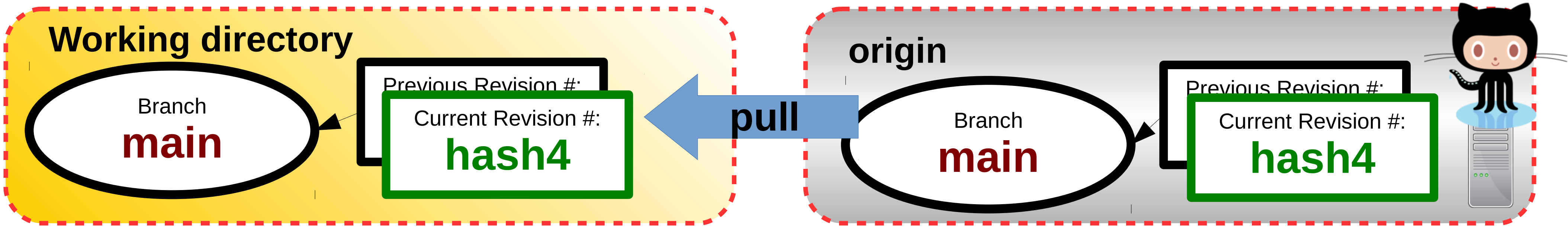


Git pull the changes from the remote server

- By the fault the pull command retrieves all tracked branches. At the moment we have that main is tracked. You will be asked the github password.
- Some modern versions of git can check if the remote is in a different state, this one on Aurora doesn't. But in general the rule of thumb is:
  - **always pull before starting your work**
- `git pull`

```
From github.com:floridop/MNXB01-learn
   8c999d3..6067f45  main       -> origin/main
Updating 8c999d3..6067f45
Fast-forward
 README.md |      1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

- git shows a summary of the retrieved changes, new HEAD, the tracked branches, the lines and changes in the file. `1 +` : added one line

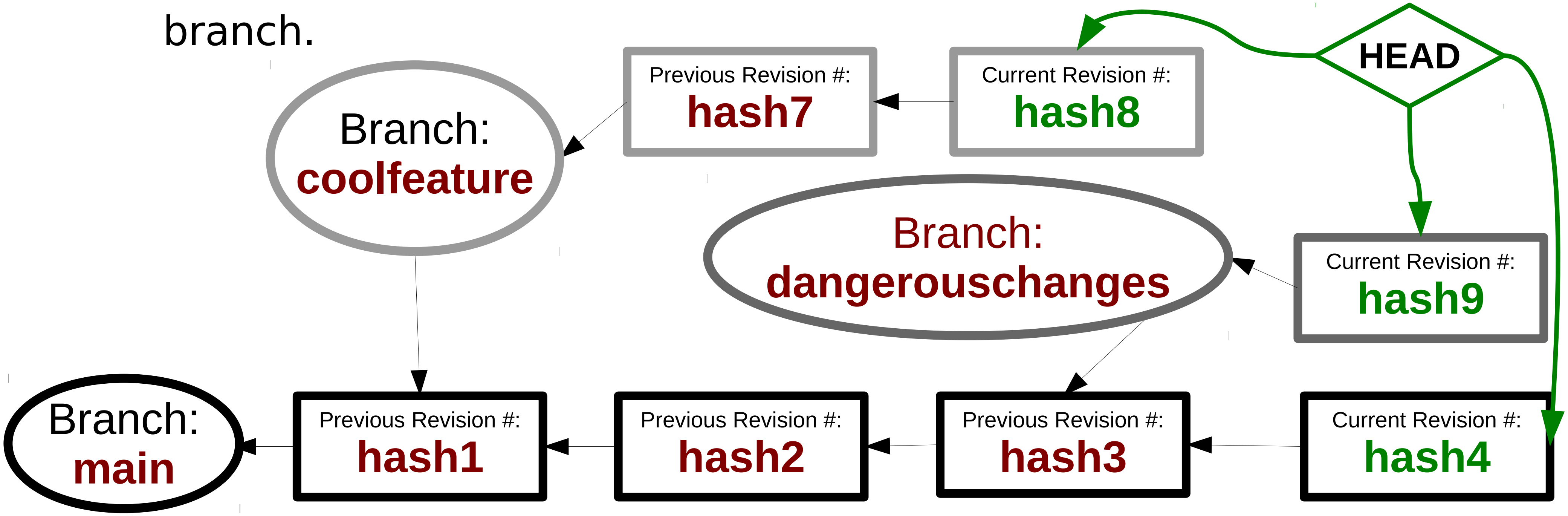


Concepts of version systems
git branches

- Keeps different developments separated so not to break the whole code
  - If I want to add a new feature to my code but I want to keep the original as it until I'm done I can create a **coolfeature** branch
  - If there are some changes that might disrupt the whole logic and I want to keep the integrity of the code I could have a **dangerouschanges** branch
  - If I just need to fix a bug and test the code with the bug fixed I might want a **bugfix** branch
- Helps contributing to code: I can create **myownbranch** and then send it to someone as a contribution.
- Command: `git branch <branchname>`

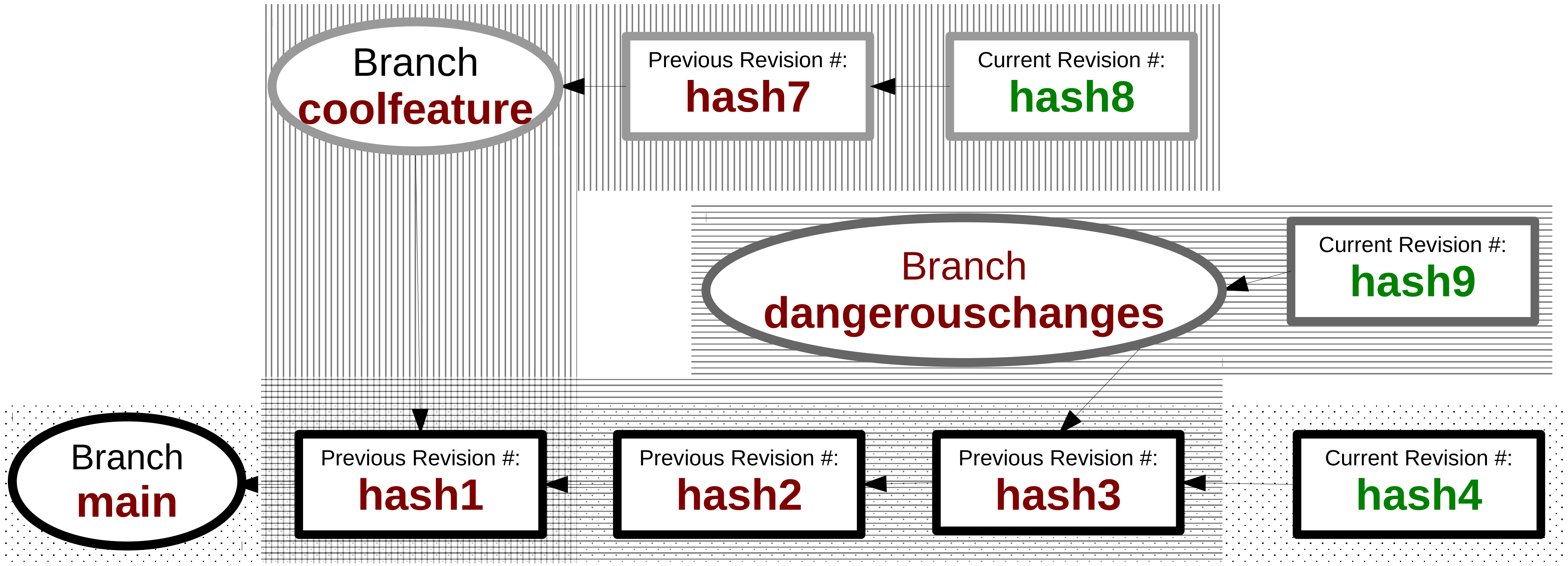
Concepts of version systems git branches

- A repository might have one or more **branch**es, that is, different version of the same repository which modify or propose different features.
  - They're called branches because they can be visualized like a *tree* as they diverge from some initial branch, usually called **main** (formerly *master*).
    Every branch has a **name**.
  - The *latest commit* of each branch is called the **HEAD** of that branch.


HEAD
Branch: coolfeature
Previous Revision #: hash7
Current Revision #: hash8
Branch: dangerouschanges
Current Revision #: hash9
Branch: main
Previous Revision #: hash1
Previous Revision #: hash2
Previous Revision #: hash3
Current Revision #: hash4

Concepts of version systems
git branch

- Every branch **history** is a continuation of the history where the main was branched.
- It is possible to branch from a branch, not just from the main. Use with care, can be confusing!

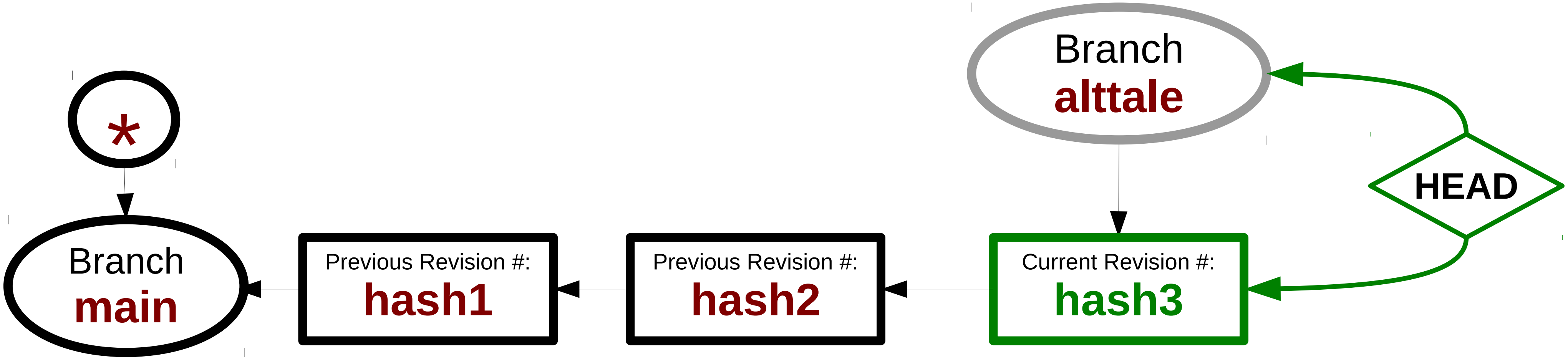


git branch our fairy-tale

- Back on Aurora, let's create a branch `alttale` out of the existing *main*:
  - **git branch alttale**
- Look at the branches:
  - **git branch**

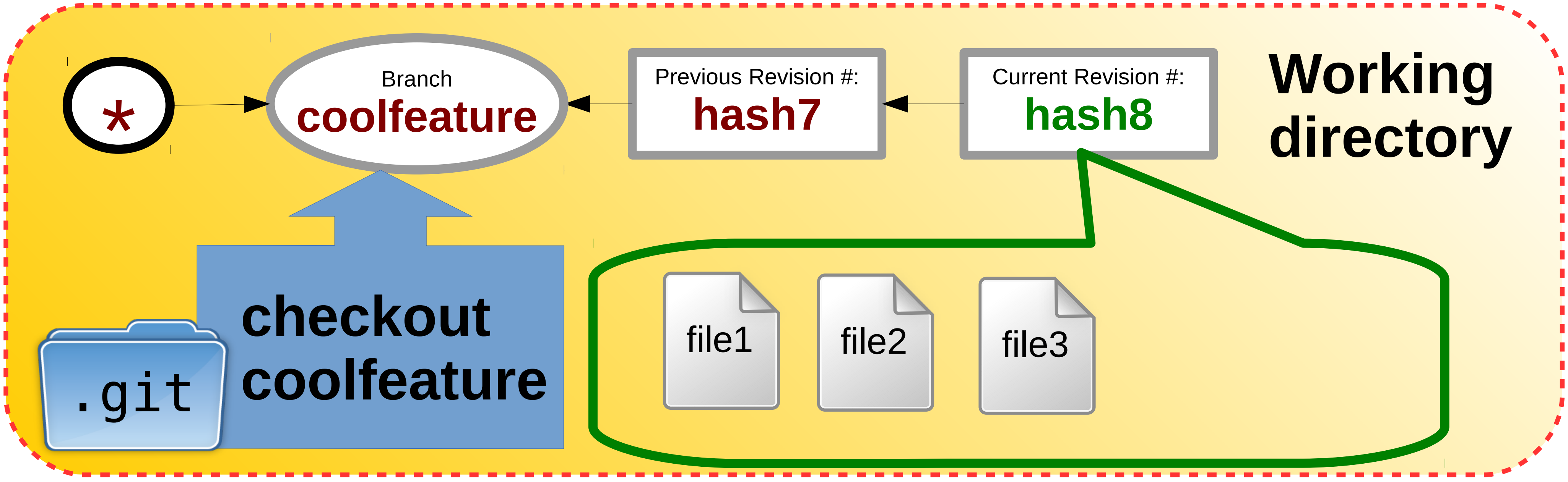
```
  alttale
* main
```

- We're still working on `main`: one can tell by the **asterisk *** next to the branch name.



Concepts of version systems
git **checkout**

- A branch can be made *active* with the `checkout` operation. When a branch is checked out you will be able to **see its files in your working directory**.
  - ✓ To `checkout` a branch means to select a certain history of changes.



Checkout the `alttale` branch

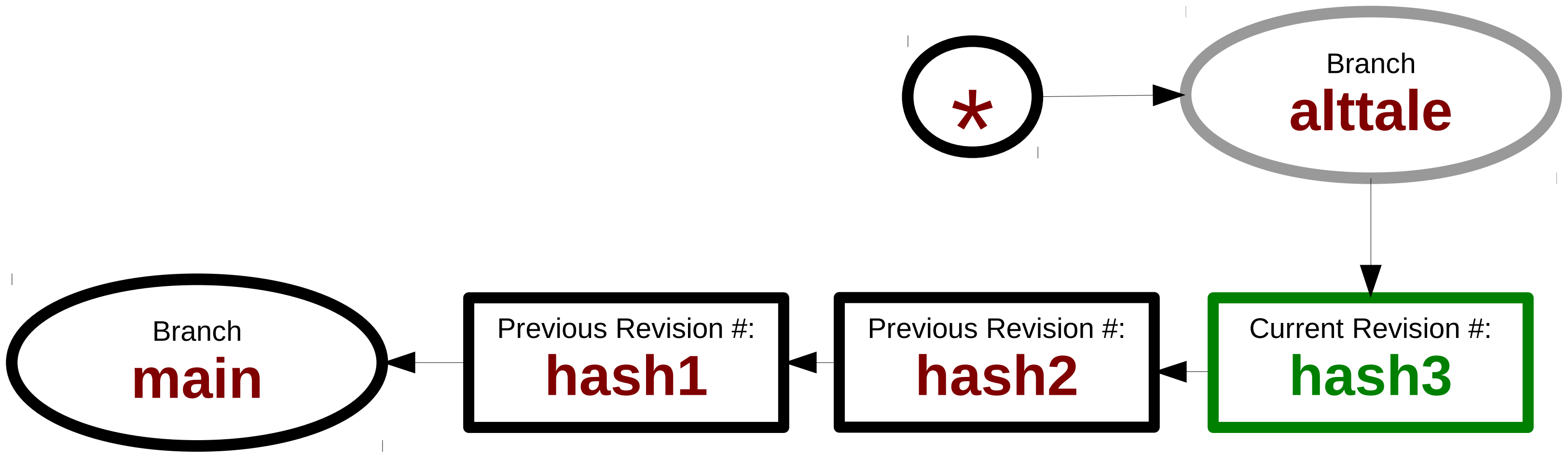
- **git checkout alttale**

```
Switched to branch 'alttale'
```

- **git branch**

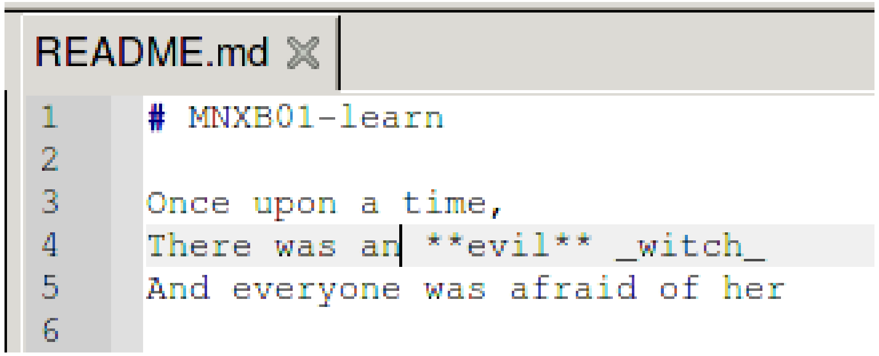
```
* alttale
  main
```

  - Now we're on the `alttale` branch
  - The last current revision of `main` is the first of `alttale`. From this point on their history **diverges**, but they **share the past** history.



Let’s change the story

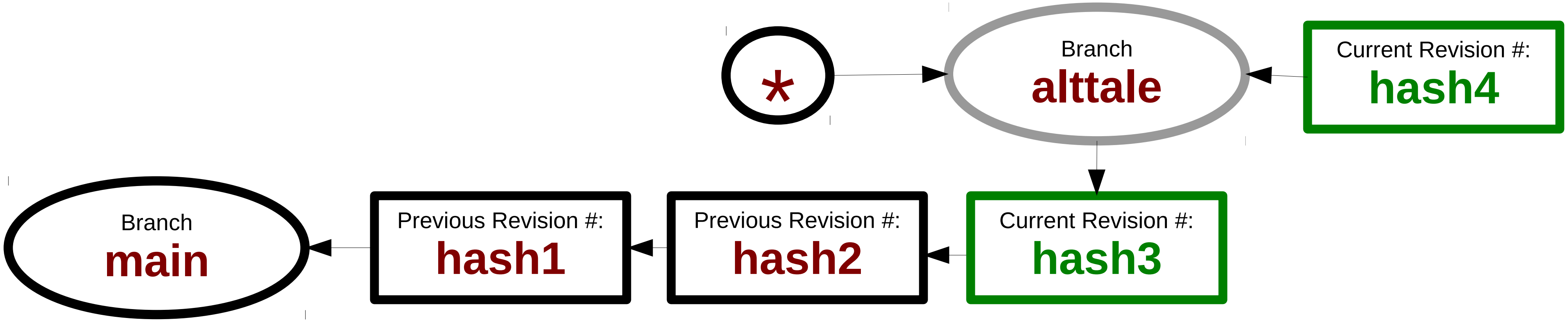
- Let’s edit README.md again
  - `pluma README.md&`
  - pluma may ask you if you want to reload the file.
    Say yes: the copy pluma has is still relative to before our changes on github, it is not up to date.
- and modify the story as such:



```
# MNXB01-learn

Once upon a time,
There was an **evil** _witch_
And everyone was afraid of her

```

- Then add and commit:

-
  ```
  git add README.md
  git commit -m 'alternate character tale'
  [alttale 8a95450] alternate character tale
   1 files changed, 2 insertions(+), 1 deletions(-)
  ```

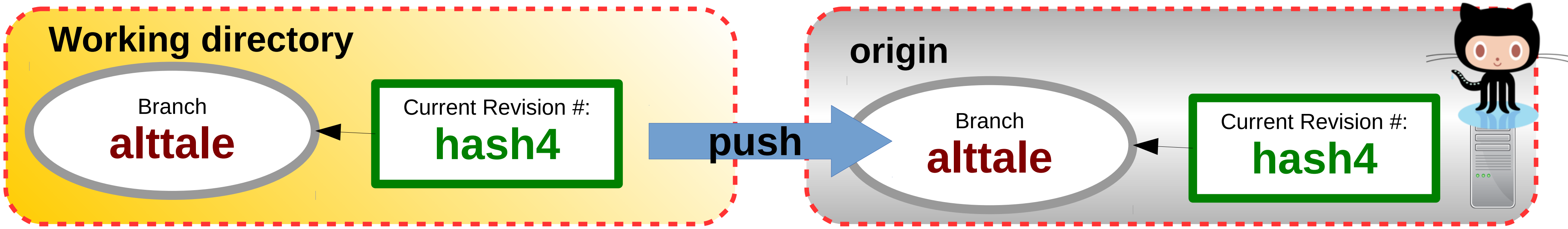


Let’s push the new branch to origin

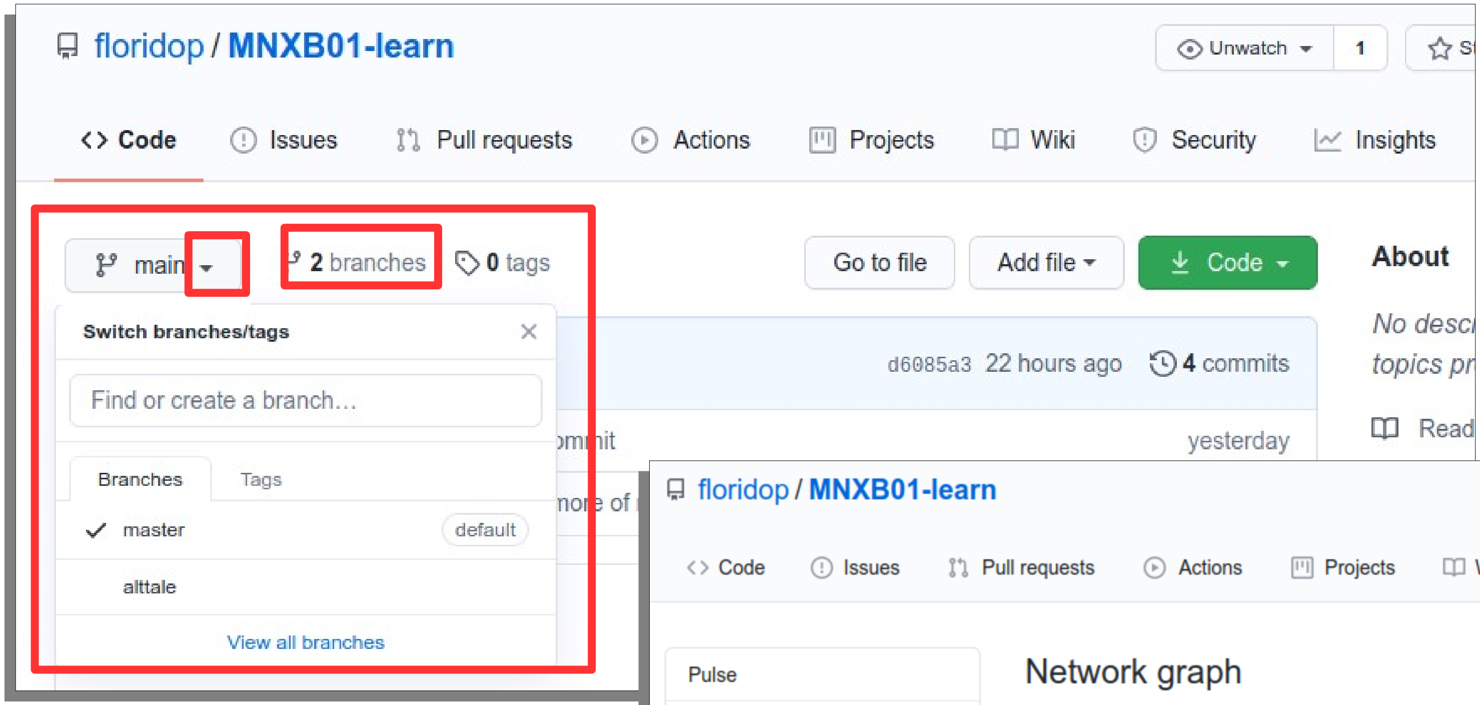
- We think this new story might work, so we want to save also this one on the remote origin.
- **git push origin alttale**

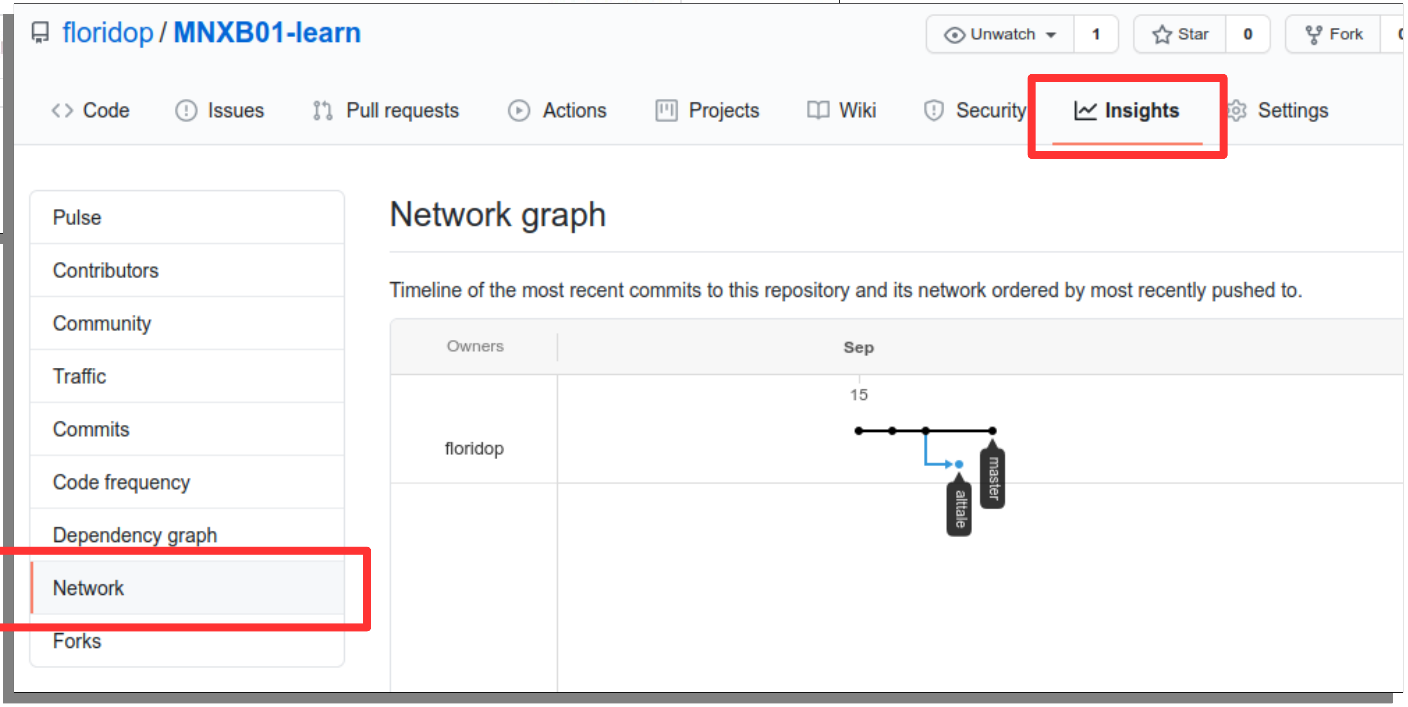
```
Counting objects: 5, done.
Delta compression using up to 40 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 367 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'alttale' on GitHub by visiting:
remote:      https://github.com/floridop/MNXB01-learn/pull/new/alttale
remote:
To git@github.com:floridop/MNXB01-learn.git
 * [new branch]      alttale -> alttale
```

- git suggest us to do a *pull request*. These are important in a collaborative environment, we will see later what they are and how to perform them.



See branches and history network on github





Typical workflows summary Scenario 2: **personal** project, **one** user, **single remote server/repo**

The one we just did in the tutorial so far.

1) **Login** into github (or you own git server)
2) **Create** a (bare) remote repository (your “**origin**”)
3) **Clone** the repository on your laptop. It will create your local **working directory**
4) Work/save on your laptop, **branch**, **add** and **commit** in your working directory
5) **Push/pull** branches and commits to/from your **origin** to synchronize with the remote server

git tutorial part 2
scenario 3:
collaborative environment

Typical workflows Scenario 3: introduction **shared project, many users, one or many servers/repository**

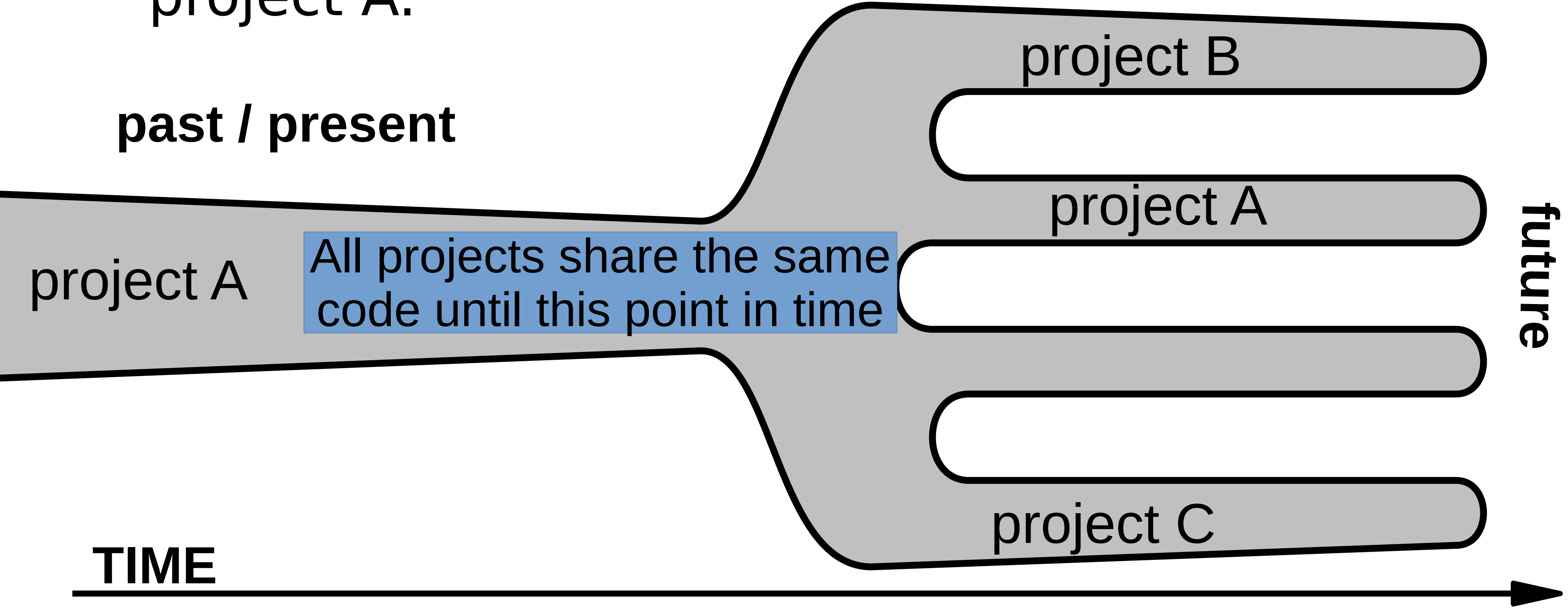
- A community creates a project repository that will involve many user to cooperate
- 2 scenarios:

1) Multiple users committing/pushing/pulling to the same git repository
   - **Pro**: quick development provided that no one is touching anyone else code.
   - **Cons**: one can generate many conflicts (changes on the same file at the same time)
   - Only works when developers do not touch each other's code
     - Good for small projects or tasks
2) Multiple users **suggesting contributions**
   - **Pro**: controlled development where one **release manager** accepts
   - **Cons**: conflicts can be avoided by the release manager
   - Only works if developers agree on a *git flow* – a way of working with git.
     - Good for large projects

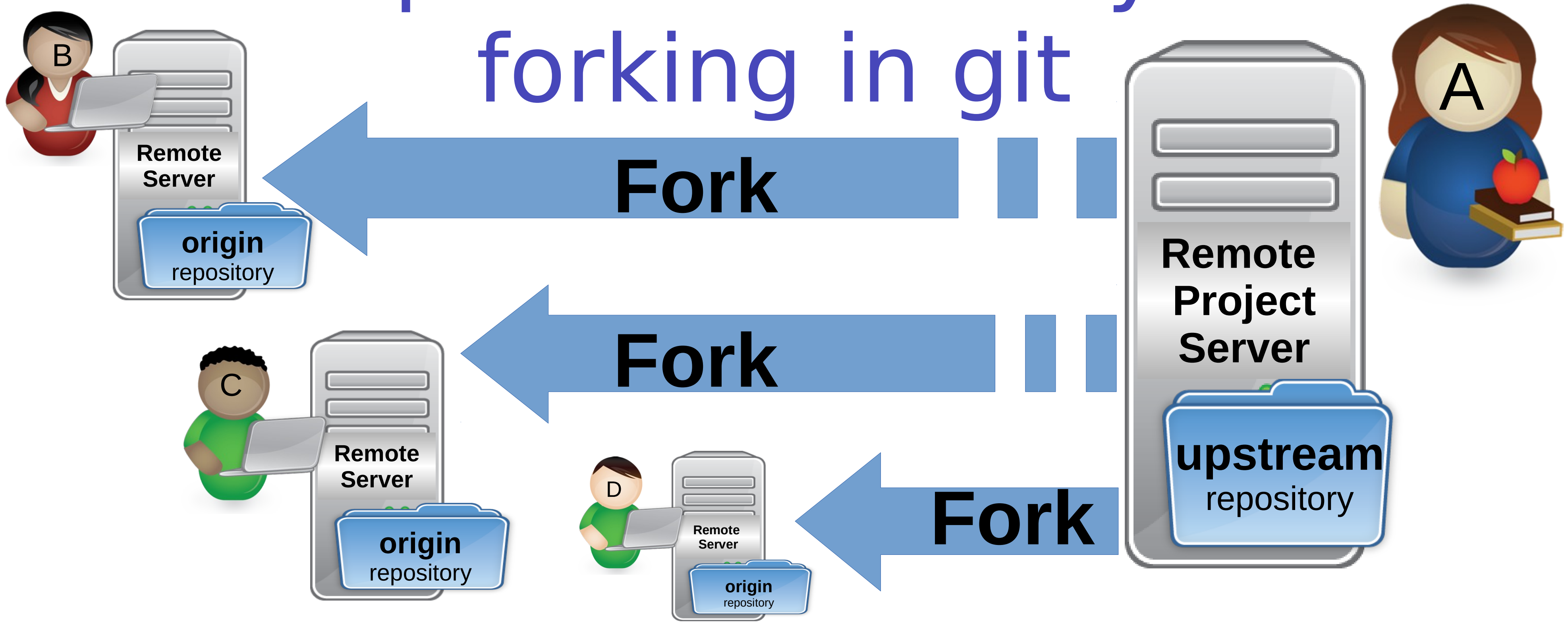
In this tutorial we will learn **scenario 2** as it works well with large scientific collaborations.

What is a software fork

- In software engineering, a **fork** of a software project A it's a copy of the software source code of A to develop features for a project B,C,... that follow completely independent choices from project A.



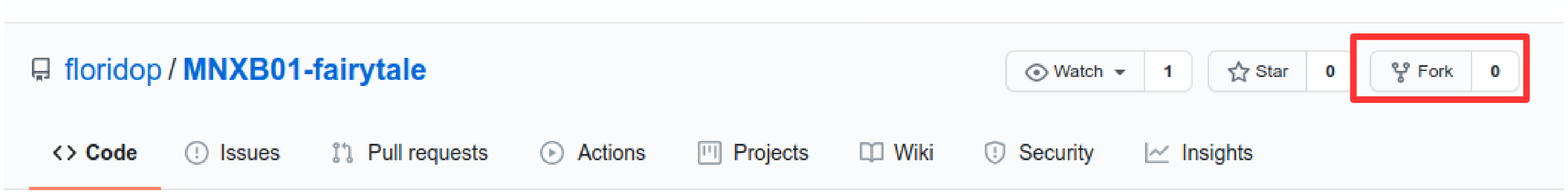
Concepts of version systems forking in git



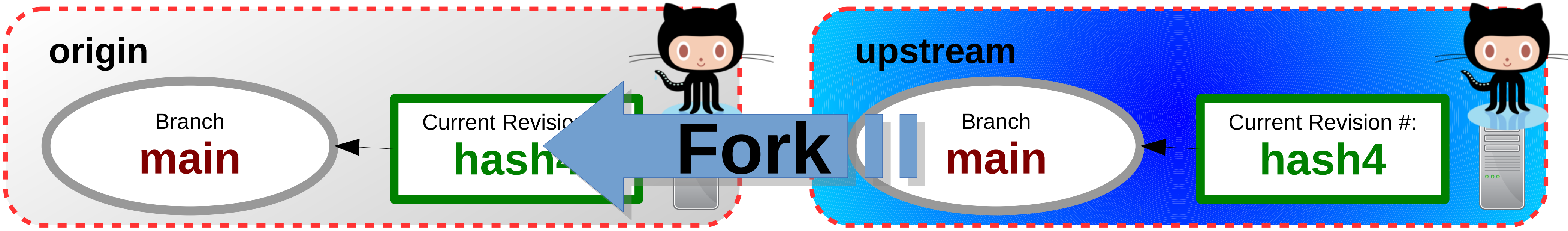
- A fork happens usually between one or more users or organizations writing software, and one of them (A) is the **release manager**.
- B,C,D **fork** A's project repository where all the contributions will be collected. In git this is done by **cloning** (duplicating) the repository of a project one wants to work on, called **upstream**
- Users B,C,D work on their forks/origins **independently**. At some point they might want to send the changes they made back to the A's upstream repository.

Fork my MNXB01-learn repository on github

- Goal: write a collaborative fairytale. Each of you will add some lines.
- Go to my github MNXB01-fairytale repo at the page: https://github.com/floridop/MNXB01-fairytale
- Click on fork:



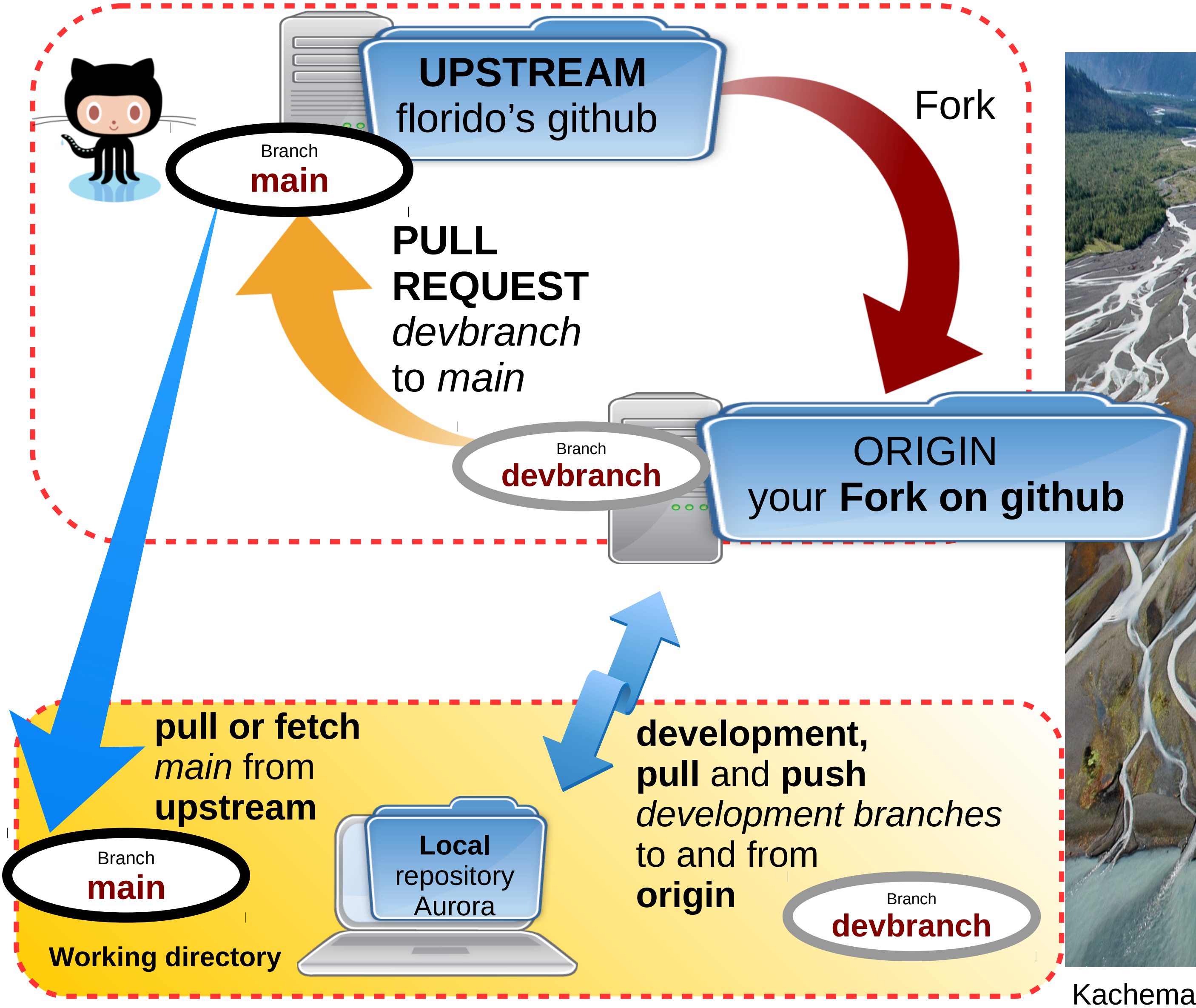
- A copy of my repository is now present in your github.
- When forking, the following naming conventions apply:
  - The repository **from which** you fork is called **upstream**
  - The copy in your personal github page is called **origin**



Setup the MNXB01-fairytale git fork

- On **your fork page**, copy the repository URL from the code button as we saw in the first part of the tutorial (slide 23).
- On Aurora, clone your origin:
  - `cd ~/git`
  - `git clone git@github.com/YOURUSERNAME/MNXB01-fairytale.git`
    - This will automatically create an **origin** with information about authentication.
  - Enter the cloned repo
    - `cd MNXB01-fairytale`
- Add my repository as **upstream remote** with this exact command:
  - `git remote add upstream https://github.com/floridop/MNXB01-fairytale.git`
    - Note that **you do not need to authenticate to my upstream repository** in this scenario. You will **never** `push` to it, only **`pull`** from it.
- list the remotes. It should look like this:
  -
```
git remote -v
origin  git@github.com/YOURUSERNAME/MNXB01-fairytale.git (fetch)
origin  git@github.com/YOURUSERNAME/MNXB01-fairytale.git(push)
upstream  https://github.com/floridop/MNXB01-fairytale.git (fetch)
upstream  https://github.com/floridop/MNXB01-fairytale.git (push)
```

A git flow model:
Upstream, origin, local, A Tale of a River



UPSTREAM
florido's github
Branch
main
Fork
PULL REQUEST *devbranch* to *main*
Branch
devbranch
ORIGIN
your **Fork on github**
pull or fetch *main* from **upstream**
Branch
main
Local repository Aurora
development, pull and **push** *development branches* to and from **origin**
Branch
devbranch
Working directory

Kachemak Bay, AK. Photo credit: Alaska Shorezone.
https://medium.com/@AKSalmonProject/where-the-river-meets-the-tides-salmon-and-estuaries-a9e7aaf78519

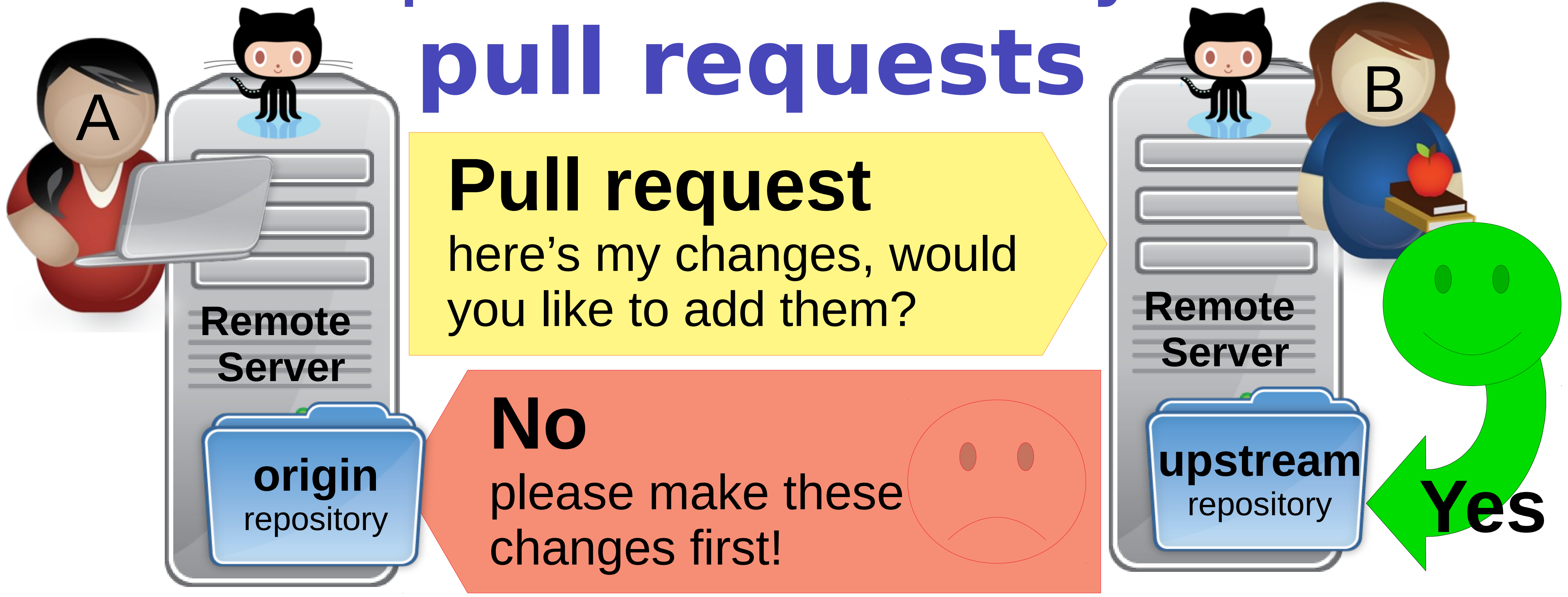
Fork, local, origin, upstream highlights

In the git flow I am showing you the following rules apply. Other flows can have different rules.

- In this model, the **upstream** *main* contains the latest and greatest version of the software. **Only the release manager is allowed to modify its content.**
- A developer *forks* some user **upstream** repository on github to obtain an personal **origin** repo in their personal account.
- The developer *clone*s **their origin** into **their working directory**
- The developer *codes* in **their working directory, but:**
  - The developer **never codes in their *main***
  - All the development happens in ***development branches*** created by the developer
- The developer *push*es and *pull*s to and from **their origin**
- The developer periodically
  - *pulls main* **from the release manager upstream** repo to keep *their* ***working directory*** *main up-to-date*
  - *push*es the *updated main* to *their* **origin** *main* so that also the copy on github is in sync with the upstream
  - After pulling, one typically will have to carefully *merge* the code in the branches with the upstream changes.
- When making changes in a *development branch*, the developer requests the upstream release manager to *apply their changes* by sending a **pull request on github** against the **upstream** *main*

Concepts of version systems

pull requests



- Pull requests are a way to **propose changes** to the forked repository so that the owner of the upstream repository can **review** them and discuss them before approval
- If they are accepted, they will be integrated, that is, the **origin** *development branch* will be merged into the **upstream** *main*
- If they are rejected, a discussion can be made about why and how to make them acceptable.
- After this process the user A will need to **pull** the changes from upstream for origin to be in **sync** with upstream.

Collaborative development 1/2

- We will write a collaborative fairy-tale.
  - Each of you will add a sentence, a line, a picture. Be creative! But only characters. If you are more into drawing than writing maybe your search the internet for “ASCII art”
  - Why text? GIT is very bad with files files that are not text. It can only handle well text. The power of git lies in the ability to use the Linux command line to do many things with text.
- Any development in this model starts with creating a a **development branch**:
  - `git checkout -b mystory`

    ```
    Switched to a new branch 'mystory'
    ```
    - The above is an equivalent and a faster way for:
      - `git branch mystory`
      - `git checkout mystory`
  - Check that you are in the *mystory* branch (the asterisk marks it!) with:
    - `git branch`

      ```
        main
      * mystory
      ```

Collaborative development 2/2

- Edit the fairytale.md file and add a piece of story (remember to save!)
  - `pluma fairytale.md&`
- add and commit
  - `git add fairytale.md`
  - `git commit -m ‘my side of the story’`

```
[mystory 1e46496] my side of the story
 1 files changed, 3 insertions(+), 0 deletions(-)
```

- push the *mystory* branch to your **origin** like we did in the first part of the tutorial (you will be asked the github password):
  - `git push origin mystory`

```
Counting objects: 5, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 349 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'mystory' on GitHub by visiting:
remote:      https://github.com/GITUSERNAME/MNXB01-fairytale/pull/new/mystory
remote:
To https://GITUSERNAME@github.com/GITUSERNAME/MNXB01-fairytale.git
 * [new branch]      mystory -> mystory
```

- You just completed the first half of your first collaborative development task!

Your first pull request 1/4

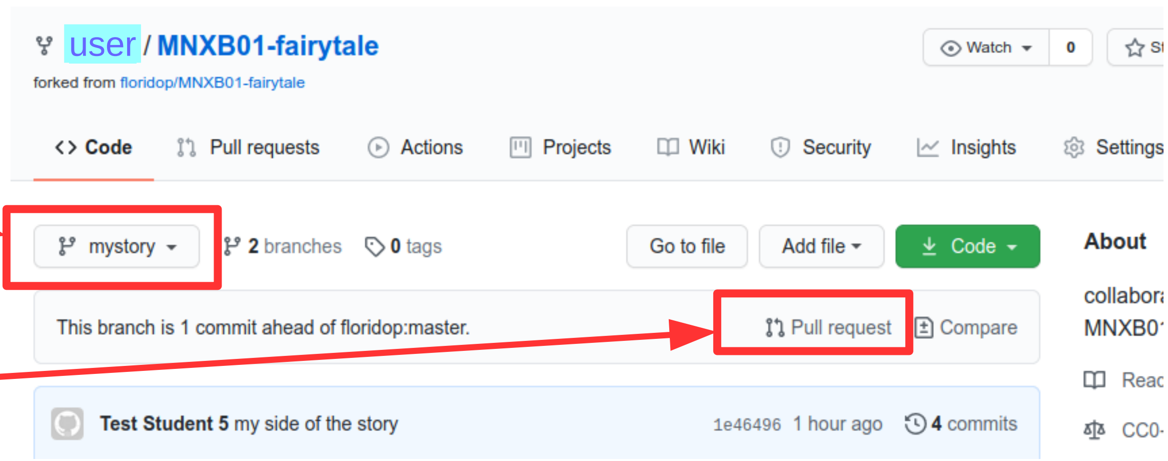
There are three possible ways of performing a **pull request**, pick **one** of them:

1) Follow git/github suggestion and click (or copy paste in the browser) the link you saw in the git output in the previous slide (like the one highlighted below but with your github user instead of GITUSERNAME):

```
remote: Create a pull request for 'mystory' on GitHub by visiting:
remote:      https://github.com/GITUSERNAME/MNXB01-fairytale/pull/new/mystory
```

2) On your fork **origin,**

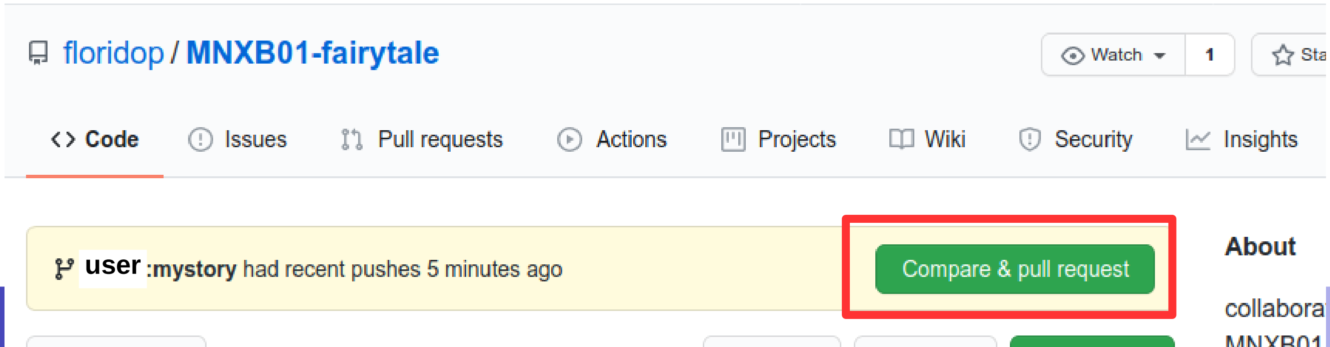
- Select the *mystory* branch
- Click on "Pull request"



This branch is 1 commit ahead of floridop:master.

Test Student 5 my side of the story

3) On my **upstream** repo page, if you're logged in in github, you will see a notification that one of your branches is ready for a pull request. Click on "Compare & pull request"

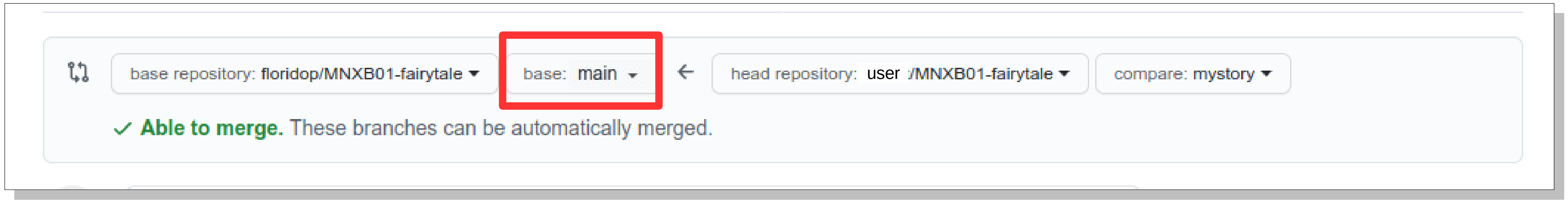


user :mystory had recent pushes 5 minutes ago

Your first pull request 2/4

The pull request submission page presents some interesting tools.

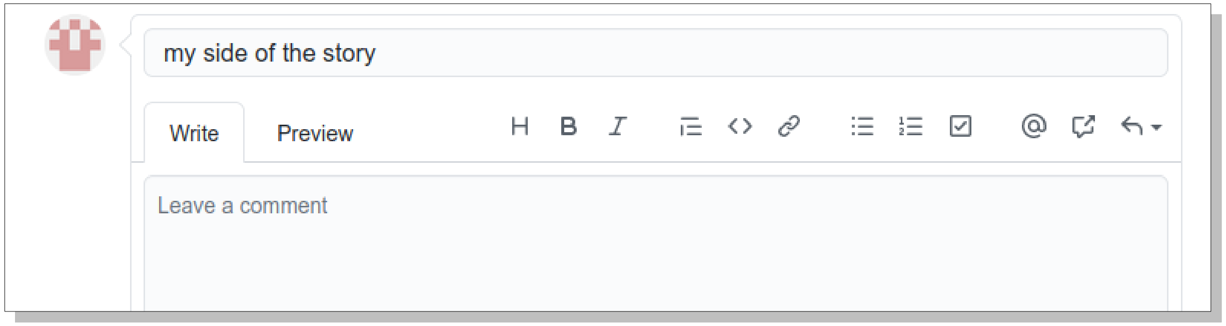
- One can choose the target branch of a pull request. Submit the pull request against my **upstream** *main*



base repository: floridop/MNXB01-fairytale | base: main | head repository: user /MNXB01-fairytale | compare: mystory

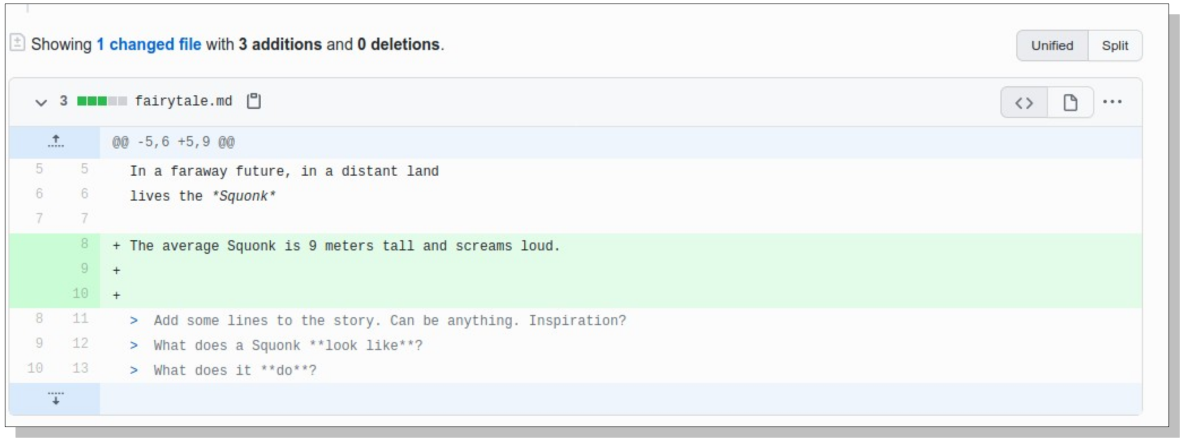
✓ Able to merge. These branches can be automatically merged.

- You can add additional comments for me to know what to do with your submission:



my side of the story

- If you scroll down the page you will see the *diff*erences between your files and mine:

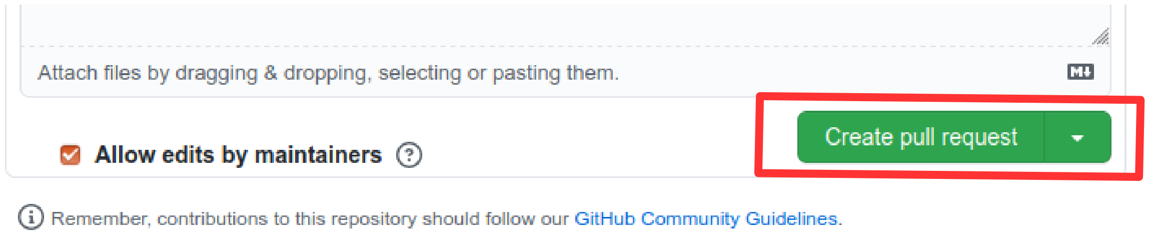


Showing 1 changed file with 3 additions and 0 deletions.

```
3 fairytale.md
@@ -5,6 +5,9 @@
 In a faraway future, in a distant land
 lives the *Squonk*

+ The average Squonk is 9 meters tall and screams loud.
+
+
 > Add some lines to the story. Can be anything. Inspiration?
 > What does a Squonk **look like**?
 > What does it **do**?
```

Your first pull request 3/4

- When done you can finally submit the pull request by clicking “Create pull request”



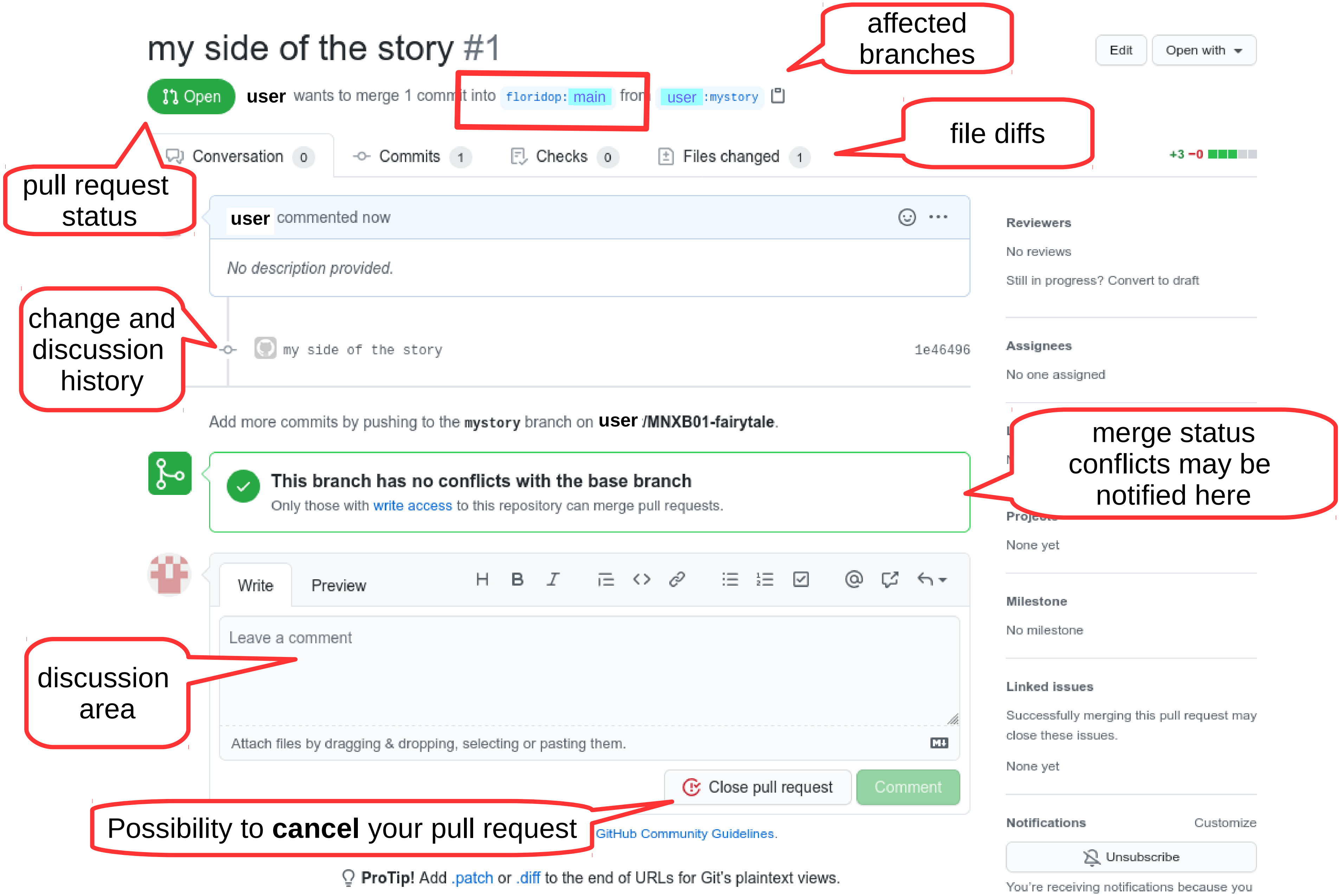
Attach files by dragging & dropping, selecting or pasting them.

☑ Allow edits by maintainers

Remember, contributions to this repository should follow our GitHub Community Guidelines.

- You just completed the second half of your first collaborative development task!
From now on the task flow **is handed over to the release manager (me)**
- You can check the progress of your pull request at the URL generated for the pull request, the page that you see after creating it. Let’s look at it in detail in the next slide...

Your first pull request 4/4



my side of the story #1

Open **user** wants to merge 1 commit into floridop:main from user:mystory

Edit | Open with

Conversation 0 | Commits 1 | Checks 0 | Files changed 1 | +3 −0

user commented now

No description provided.

my side of the story 1e46496

Add more commits by pushing to the **mystory** branch on **user /MNXB01-fairytale**.

This branch has no conflicts with the base branch

Only those with write access to this repository can merge pull requests.

Write | Preview

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Close pull request | Comment

GitHub Community Guidelines.

ProTip! Add .patch or .diff to the end of URLs for Git's plaintext views.

Reviewers
No reviews
Still in progress? Convert to draft

Assignees
No one assigned

Projects
None yet

Milestone
No milestone

Linked issues
Successfully merging this pull request may close these issues.
None yet

Notifications Customize
Unsubscribe
You're receiving notifications because you

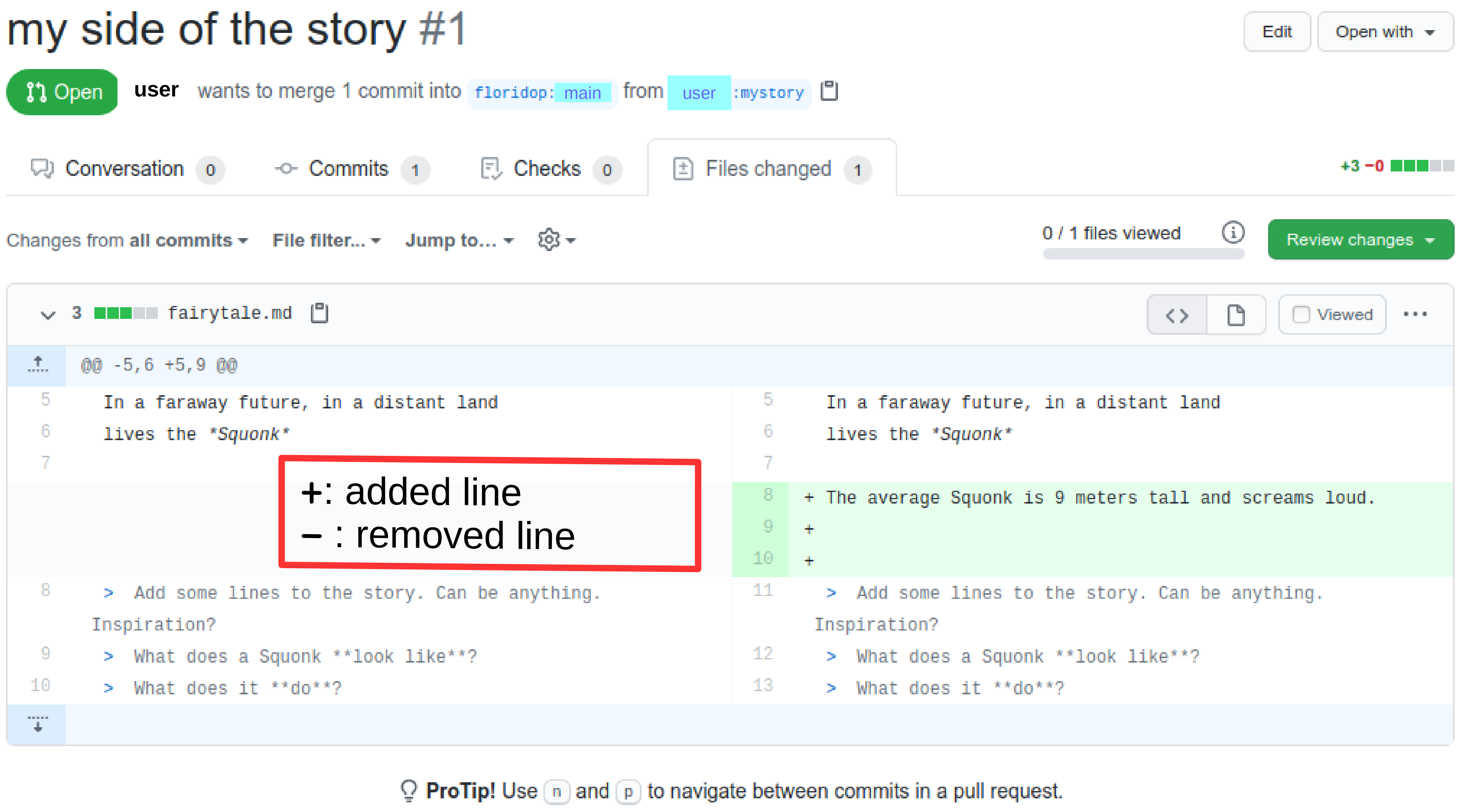
affected branches
file diffs
pull request status
change and discussion history
merge status conflicts may be notified here
discussion area
Possibility to **cancel** your pull request

Reviewing pull requests

- I will now process your pull requests as a **release manager** in order of arrival and make changes to my **upstream** *main*
- During the process, the developers can interact via the github interface to discuss changes.
- The first pull request I will accept is mine, so I can show you the process of **merging and updating** your own **origin** *main* **and** *mystory* **when the upstream repo changes**.

Diff-ing on github

- In the “files changed” tab one can see the differences (*diffs* in jargon). They are shown in a standard diff format (more about it later in these slides):


my side of the story #1

Edit · Open with

Open · user wants to merge 1 commit into floridop:main from user:mystory

Conversation 0 · Commits 1 · Checks 0 · Files changed 1 · +3 −0

Changes from all commits · File filter... · Jump to... · 0 / 1 files viewed · Review changes

3 fairytale.md · Viewed

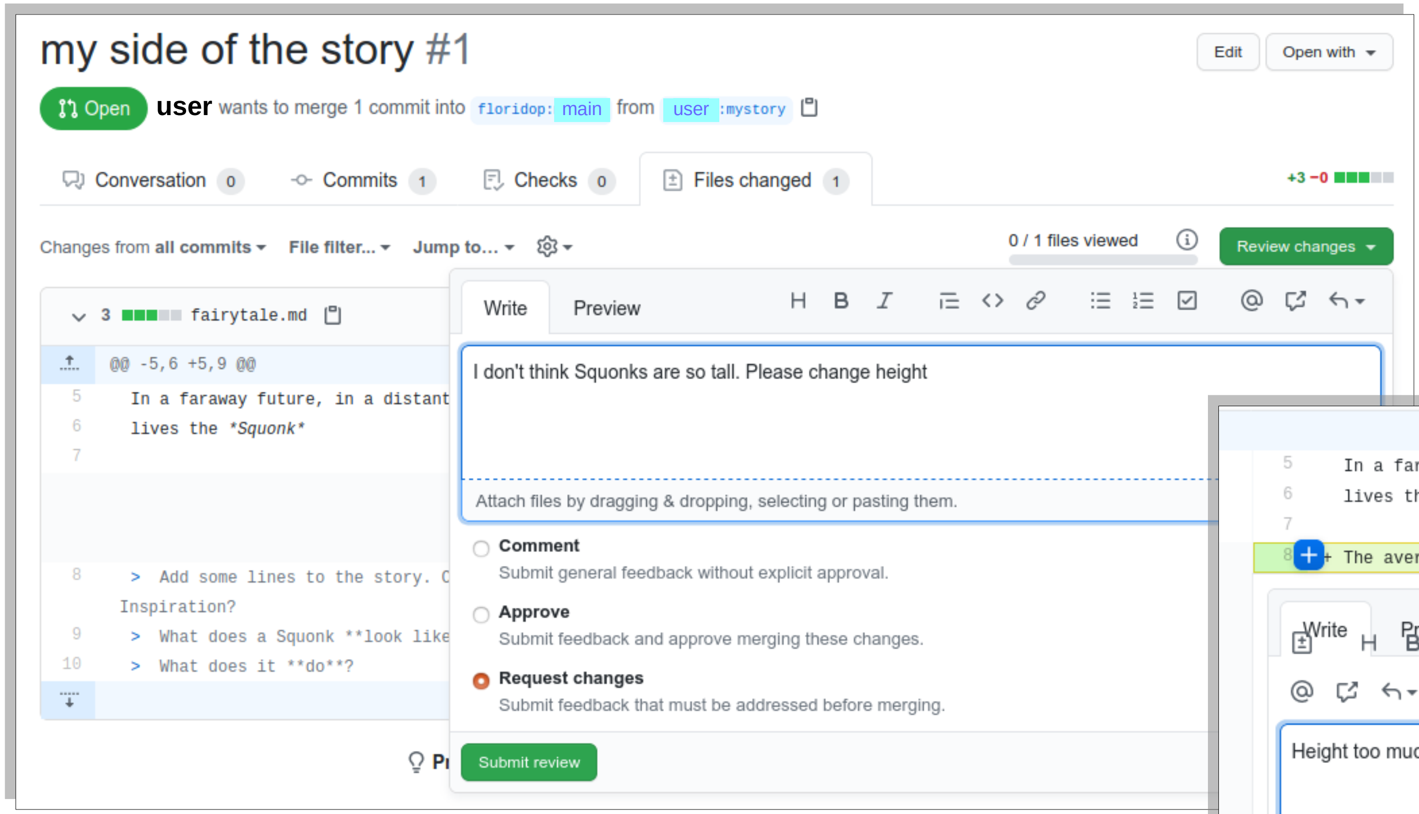
```
@@ -5,6 +5,9 @@
 5    In a faraway future, in a distant land          5    In a faraway future, in a distant land
 6    lives the *Squonk*                              6    lives the *Squonk*
 7                                                   7
                                                     8  + The average Squonk is 9 meters tall and screams loud.
                                                     9  +
                                                    10  +
 8    > Add some lines to the story. Can be anything. 11   > Add some lines to the story. Can be anything.
   Inspiration?                                         Inspiration?
 9    > What does a Squonk **look like**?            12   > What does a Squonk **look like**?
10    > What does it **do**?                         13   > What does it **do**?
```

+: added line
− : removed line

ProTip! Use n and p to navigate between commits in a pull request.

Review process

- The release manager can comment on the overall code status and even comment on single lines or blocks of code to tell the developer what to improve or change, or discuss some coding matter.



my side of the story #1

Open **user** wants to merge 1 commit into `floridop:main` from `user:mystory`

Conversation 0 · Commits 1 · Checks 0 · Files changed 1 · +3 −0

Changes from **all commits** · **File filter...** · **Jump to...** · 0 / 1 files viewed · Review changes

3 fairytale.md

```
@@ -5,6 +5,9 @@
 In a faraway future, in a distant
 lives the *Squonk*

 > Add some lines to the story. C
Inspiration?
 > What does a Squonk **look like
 > What does it **do**?
```

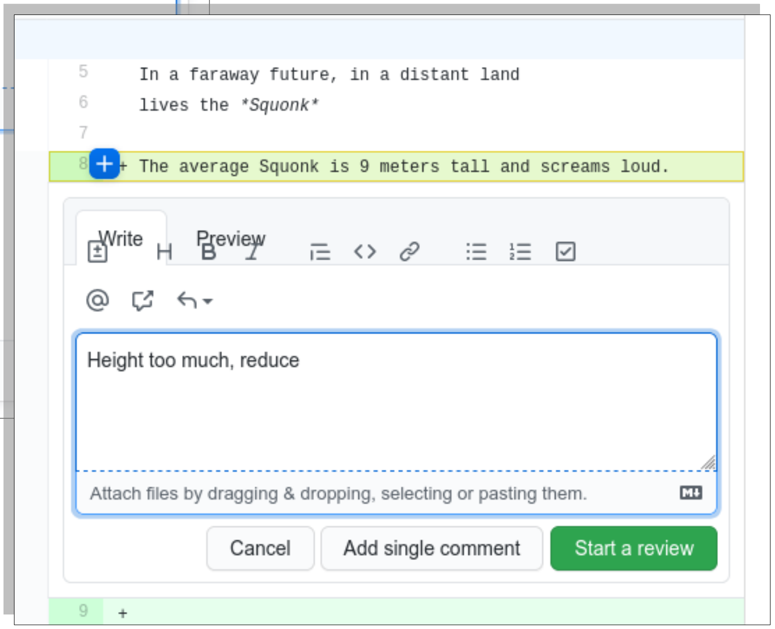
Write | Preview

I don't think Squonks are so tall. Please change height

Attach files by dragging & dropping, selecting or pasting them.

- ○ **Comment** Submit general feedback without explicit approval.
- ○ **Approve** Submit feedback and approve merging these changes.
- ● **Request changes** Submit feedback that must be addressed before merging.

Submit review



```
 In a faraway future, in a distant land
 lives the *Squonk*

+ The average Squonk is 9 meters tall and screams loud.
```

Write | Preview

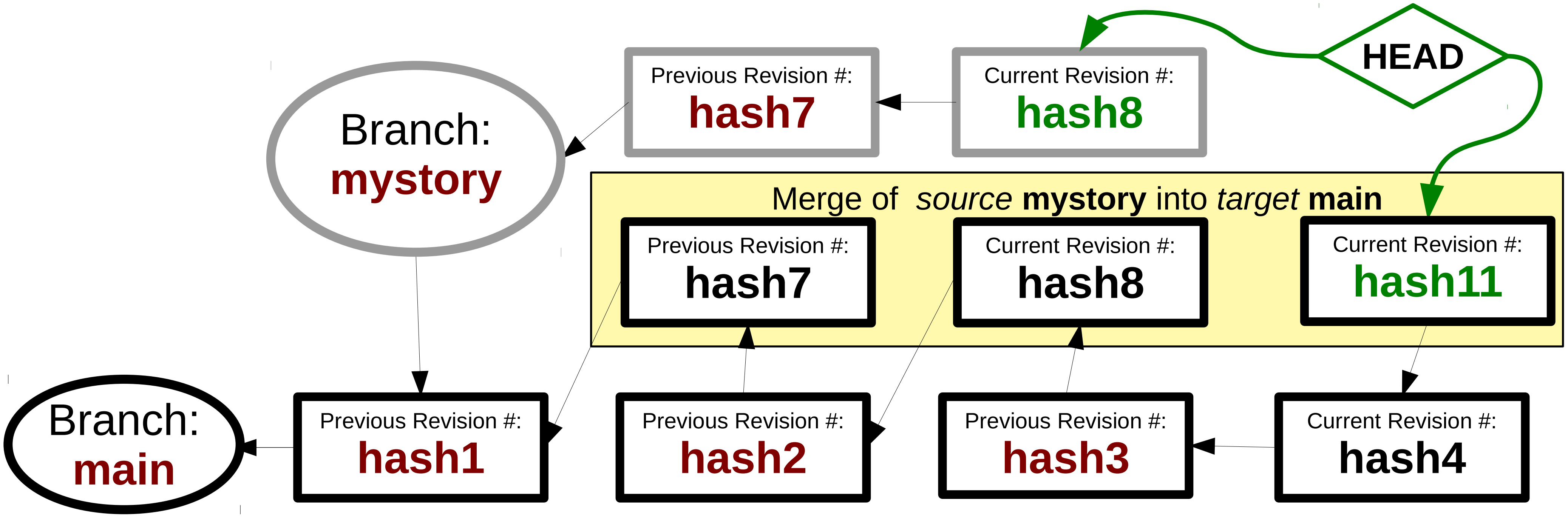
Height too much, reduce

Attach files by dragging & dropping, selecting or pasting them.

Cancel · Add single comment · Start a review

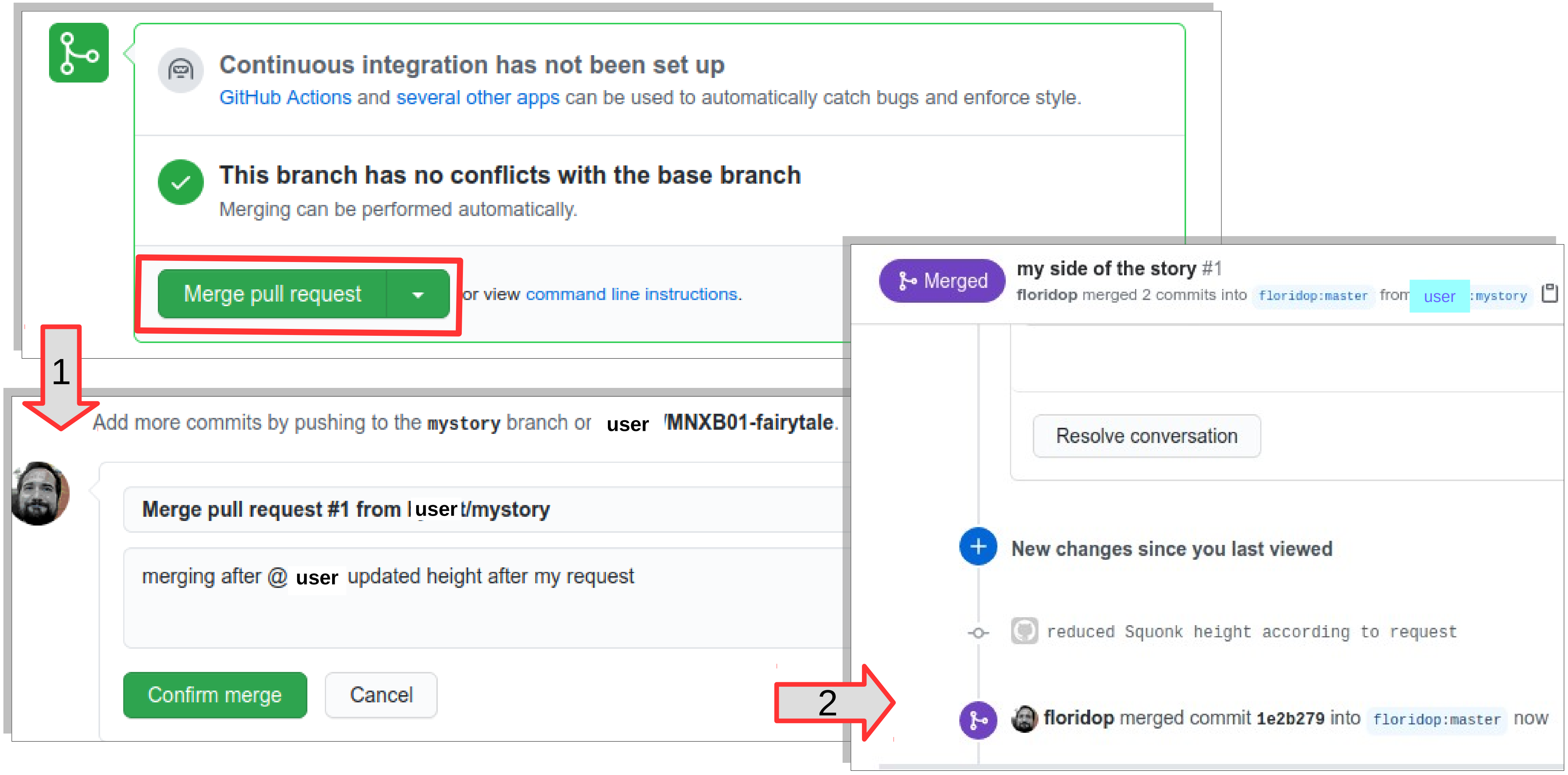
Merging – a new history is born

- Once the changes in a *development branch* are accepted, these are usually integrated back in the **upstream** *main* with the `merge` operation. This is done by the **release manager**.
- The result of a merge operation between a *source* and a *target* branch is a merged history of commits between the two branches. **The commits in the source branch are copied to the target branch.**
- A new HEAD is created with a commit that says that there was a merge in a given moment in time.
- Files are modified according to a **distributed timeline**. **Conflicts** may arise and need to be **resolved**.
- The result of these two operations (what github does behind the scenes) may look like the one in the example below
  1. Checkout *target*: `git checkout main`
  2. Merge with *source*: `git merge mystory`
- After a successful merge the *source* branch can be deleted. But you can keep it for the record.


HEAD
Branch: mystory
Previous Revision #: hash7
Current Revision #: hash8
Merge of source mystory into target main
Previous Revision #: hash7
Current Revision #: hash8
Current Revision #: hash11
Branch: main
Previous Revision #: hash1
Previous Revision #: hash2
Previous Revision #: hash3
Current Revision #: hash4

The release manager merges

- Once happy with the changes, the release manager can start the merge. It includes adding a special **commit message** concerning the merge.



Continuous integration has not been set up

GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.

This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request or view command line instructions.

Add more commits by pushing to the **mystory** branch or user MNXB01-fairytale.

Merge pull request #1 from user/mystory

merging after @ user updated height after my request

Confirm merge Cancel

Merged **my side of the story** #1

floridop merged 2 commits into `floridop:master` from user `:mystory`

Resolve conversation

New changes since you last viewed

`reduced Squonk height according to request`

floridop merged commit **1e2b279** into `floridop:master` now

`git merge` ≠ *github merge*

- It turns out they're not the same. Linus Torvalds, who wrote both git and the Linux kernel, complained in the kernel mailing list to a company submitting patches to the kernel in a mail exchange dated 4th of September 2021:

  "... I notice that you have a github merge commit in there. That's another of those things that I *really* don't want to see - github creates **absolutely useless garbage merges**, and you should **never ever use the github interfaces** to merge anything."

  This is the complete commit message of that merge:

  ```
  Merge branch 'torvalds:master' into master
  ```

  Yeah, that's not an acceptable message. Not to mention that it has a bogus "github.com" committer etc."

So github tampers (modifies without alarming the user) the authorship and the content of the commit messages. The question is, why? This is unfortunately yet another story of a service taken over by a multinational company that does not respect specifications or good practices.

Torvalds continues with a very nice reminder:

"github is a perfectly fine hosting site, and it does a number of other things well too, but merges is not one of those things.

Linux kernel merges need to be done *properly*. That means **proper commit messages with information about what is being merged and *why* you merge something**. But it also means proper **authorship** and **committer information** etc. All of which github entirely screws up."

So whenever possible do the *merge* step from the command line or using a good tool, avoid the github interface, even if it is very convenient. Or you will lose some information.

Source:
https://lore.kernel.org/lkml/CAHk-=wjbtip559HcMG9VQLGPmkurh5Kc50y5BceL8Q8=aL0H3Q@mail.gmail.com/

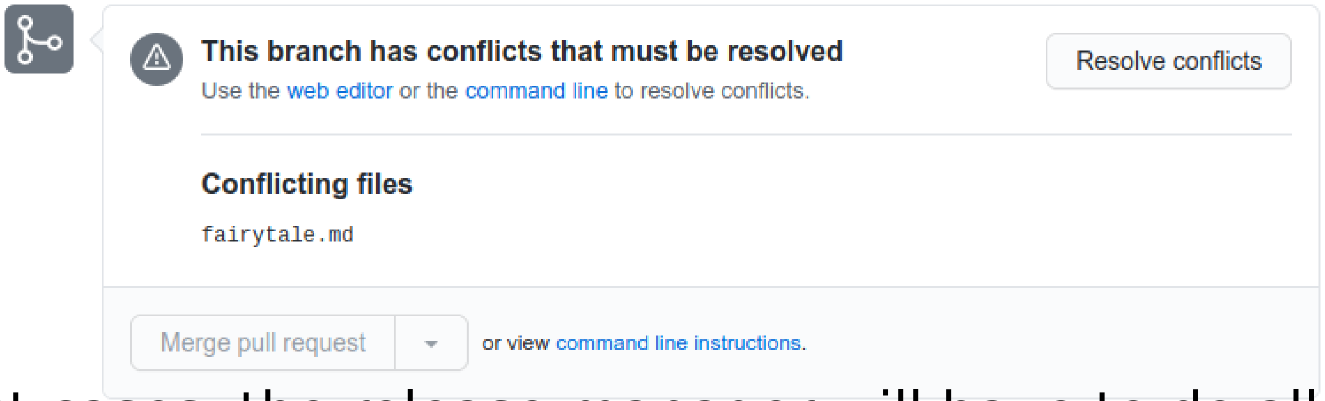
A related article: https://www.techradar.com/news/dont-use-github-to-merge-commits-suggests-torvalds
https://www.techradar.com/news/dont-use-github-to-merge-commits-suggests-torvalds

Open and closed pull requests

- As long as the pull request is open (that is, I didn't yet merged your development branch) you can still make changes and push them to your **origin** *devbranch*.
  - Github will recalculate automatically the possible outcome.

After the upstream merge 1/2: possible conflicts

- At this point, all of your branches (your **working directory** *main* and *mystory*, your **origin** *main* and *mystory)* will be **out of sync** with my **upstream** *main,* which has changed after the merge.
- For some of your pull requests that modified the same lines as the merged ones we will see the following:



This branch has conflicts that must be resolved

Use the web editor or the command line to resolve conflicts.

Conflicting files

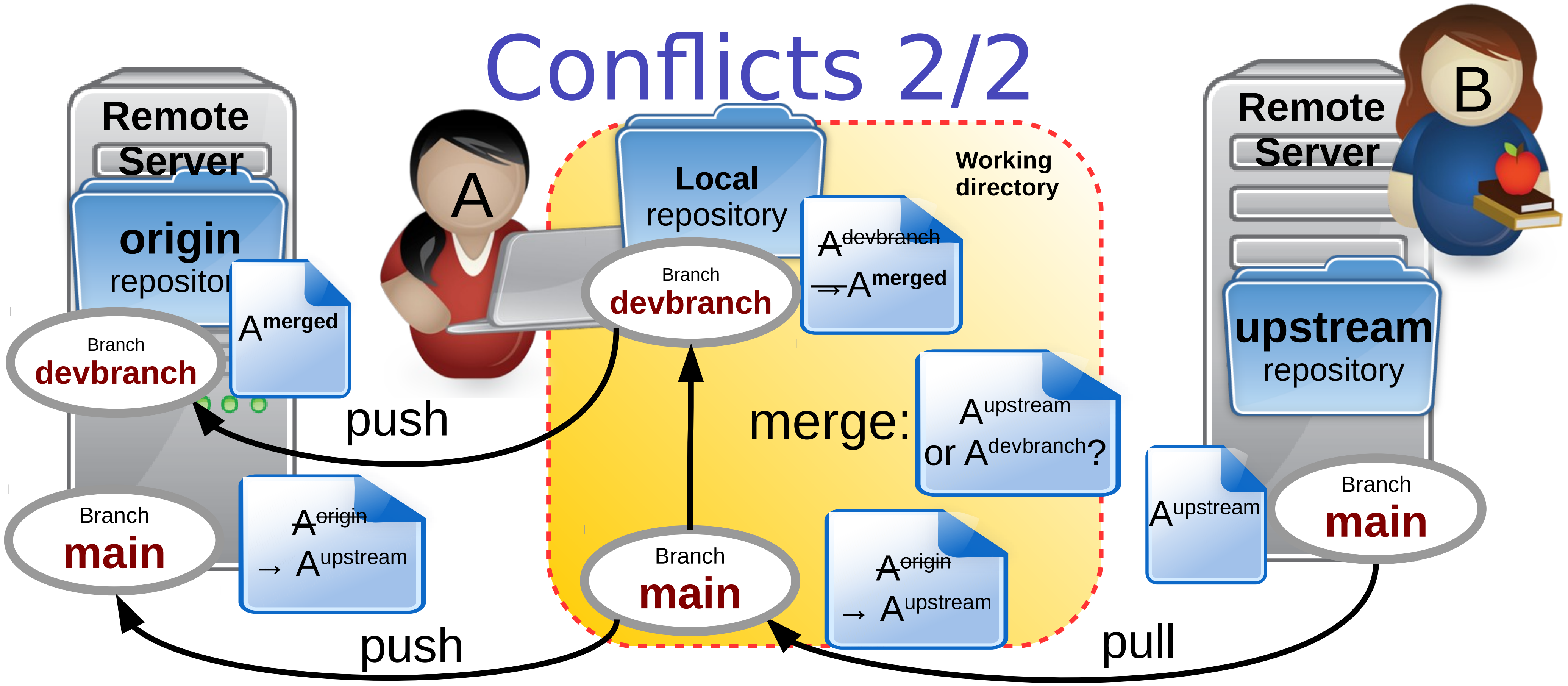
`fairytale.md`

- In most cases, the release manager will have to do all the merges manually and carefully check your code.
- In other cases, they may just tell you to sync/update your **origin** *main* to their **upstream** *main* and review the changes yourself. We will do this task to learn how to do a command line merge.

Conflicts 1/2

- In most cases, **many** developers coding on the **same file** will **cause total havoc.**
- In a distributed asynchronous collaborative environment is very likely that everyone is coding at the same time like we're doing now, so there will be **conflicts**
- **Conflicts** arise when **two or more developers edited the same lines of the same file**, and it is hard to merge the different versions of the file automatically. In that case github cannot do alone and manual intervention is required.
- A way to **avoid conflicts** is to **divide the developers' job into independent folders** and independent files, limiting as much as possible editing the same files.

Conflicts 2/2



- In a distributed development environment, **different users** may modify the **same file**
- When the **same file** is modified more or less around the **same lines**, and you try to pull from repositories or merge from branches where the modifications have been made, you may incur in a **conflict**
- A **conflict** is a set of changes that must be reviewed in order to sort out which of A^{origin},$A^{upstream}$,$A^{devbranch}$ should go into the final result of a merge
- This usually can only be solved by a developer knowledgeable of the code, and it resolves in a **n-way-merge**. An example of 3-way merge is at slide 114.
- The result is often an $A^{\mathbf{merged}}$ file that integrates all the changes all the developers made.
- In this model we try to **never pull from origin**, so that we reduce a source of the conflicts.

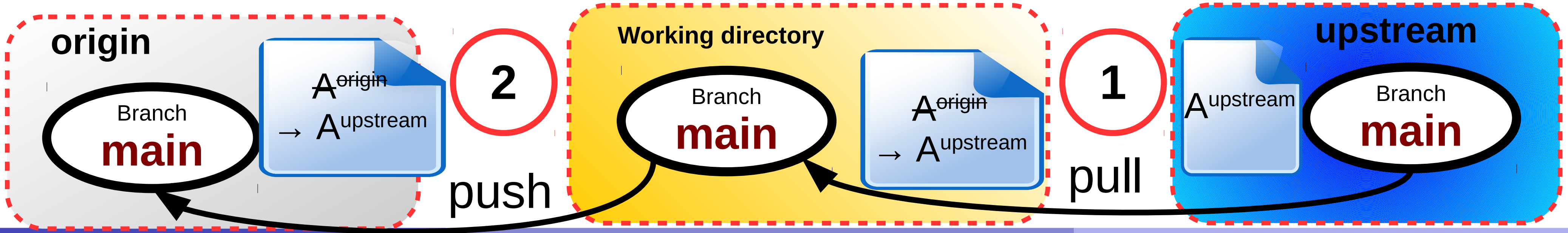
After the upstream merge 2/2: update your origin repo

- At this point, your **working directory** AND your **origin** *main* branches will be **out of sync** with my **upstream** *main*
- 1) You should update your working directory *main* to be in sync with mine by running in your work directory:

  -
    ```
    git checkout main
    Switched to branch 'main'
    ```

  -
    ```
    git pull upstream main
    remote: Enumerating objects: 1, done.
    remote: Counting objects: 100% (1/1), done.
    remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
    Unpacking objects: 100% (1/1), done.
    From https://github.com/floridop/MNXB01-fairytale
     * branch            main     -> FETCH_HEAD
    Updating abed28e..1e2b279
    Fast-forward
     fairytale.md |    3 +++
     1 files changed, 3 insertions(+), 0 deletions(-)
    ```
- 2) and then you will need to update also your **origin** *main* with the changes:

  -
    ```
    git push origin main
    Total 0 (delta 0), reused 0 (delta 0)
    To git@github.com/GITUSERNAME/MNXB01-fairytale.git
       abed28e..1e2b279  main -> main
    ```
  - See how the commits match.
- Now we're almost ready for another cycle of development.
  Next step: *merge* these updates into our *development branch*.



Visualizing changes

- Now we know that there are two different stories in *main* and *mystory*. How much do they differ?
- git provides several commands to check differences in both content and history.
- I will show you some examples that I think are useful. The main commands are:
- **`git diff`** : uses git own diffing system
- **`git difftool`**: uses a user defined git diff tool (must be configured before use).

diff a file between two branches

- To see the differences of the same file versions between two branches, use this syntax:
- **git diff branch1..branch2 filename**
- Let's try with our branches, I added some colors to the output for readability:
- **git diff main..mystory fairytale.md**

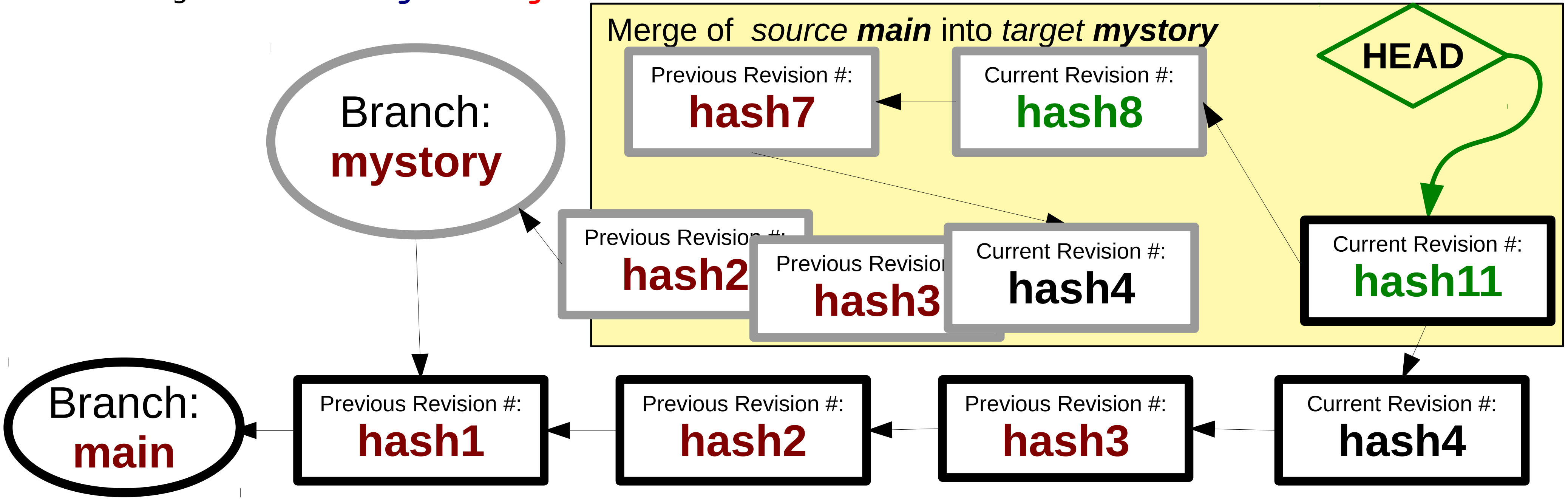
```
diff --git a/fairytale.md b/fairytale.md
index a98c877..803a9e6 100644
--- a/fairytale.md
+++ b/fairytale.md
@@ -5,8 +5,7 @@
 In a faraway future, in a distant land
 lives the *Squonk*

-The average Squonk is 7 meters tall and screams loud.
-
+I thought Squonks did not exist!

 >   Add some lines to the story. Can be anything. Inspiration?
 >   What does a Squonk **look like**?
```

-: showing lines 5 to 8 of file in in branch *main*
+: showing lines 5 to 7 of file in branch *mystory*

- : missing lines of file in branch *main*
+: added lines of file in branch *mystory*
Lines of file in common between branches

Merging updated *main* into mystory

- In this situation the **upstream** and **origin** *main* contain the latest developments and we need to incorporate those into our *development branch mystory* which is pending a pull request.
- As in the previous merge, the result of a merge operation between a *source* and a *target* branch is a merged history of commits between the two branches. **The commits in the source branch are copied to the target branch.**
- A new HEAD is created with a commit that says that there was a merge in a given moment in time.
- Files are modified according to a **distributed timeline**. **Conflicts** may arise and need to be **resolved**.
- The result of this two operations (what github does behind the scenes) may look like the one in the example below (note that it is **the opposite** of what the release manager does! They usually merge a branch into main and not viceversa.)
  1. Checkout *target*: `git checkout mystory`
  2. Merge with *source*: `git merge main`


Merge of *source* **main** into *target* **mystory**
HEAD
Previous Revision #: hash7
Current Revision #: hash8
Branch: mystory
Previous Revision #: hash2
Previous Revision #: hash3
Current Revision #: hash4
Current Revision #: hash11
Branch: main
Previous Revision #: hash1
Previous Revision #: hash2
Previous Revision #: hash3
Current Revision #: hash4

git merge on the command line

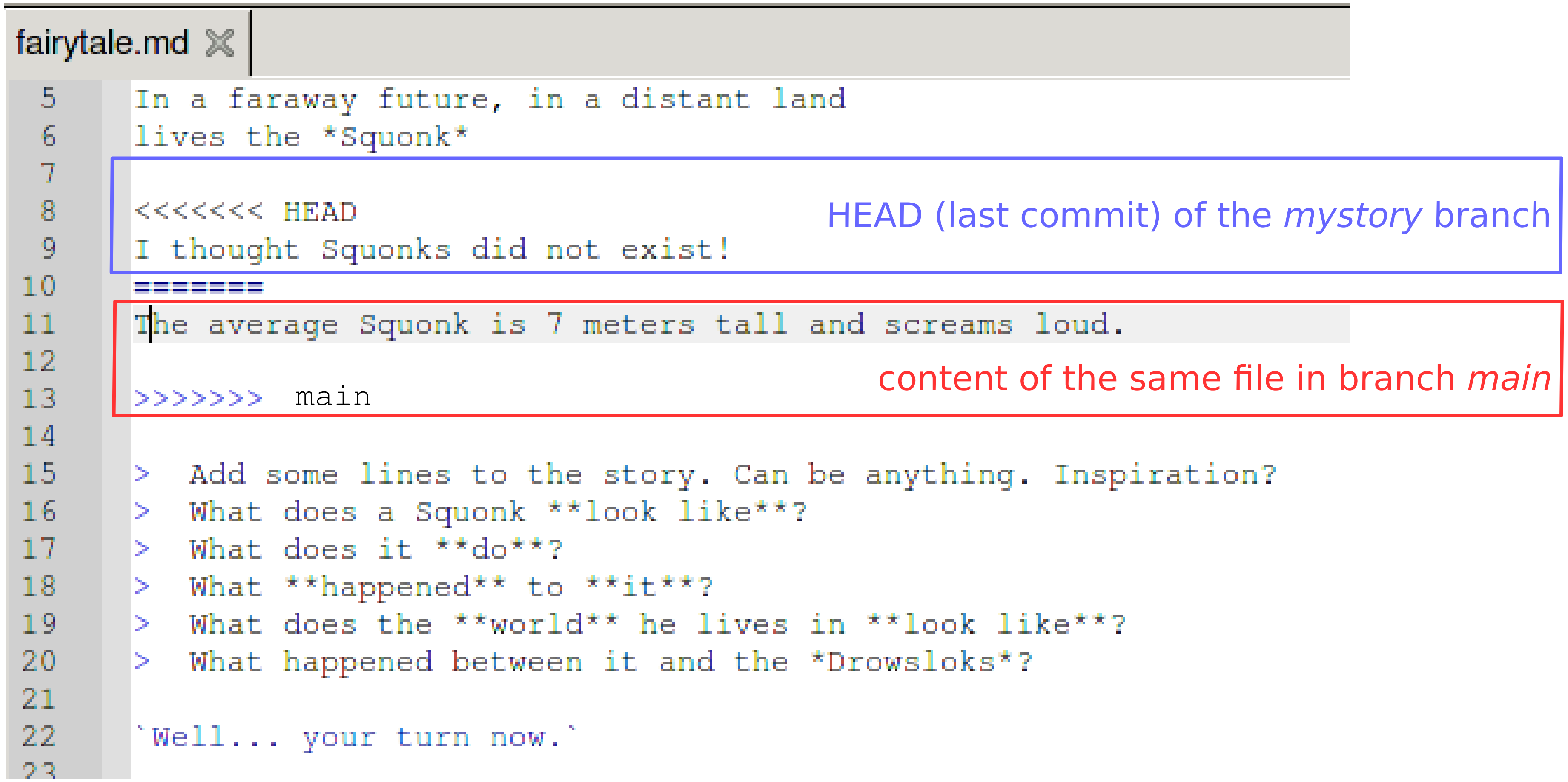
- Now we know what will happen when we merge, a conflict may be generated due to the differences in the files.
- Let's try a **git merge** and learn how to **resolve conflicts**.
- Change to the *mystory* branch and tell git to merge the content of *main*:
- `git checkout mystory`
- `git merge main`

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

- Let's open the `fairytale.md` file in `pluma` to see what the conflict caused.
- `pluma fairytale.md&`

Conflict resolution from the command line 1/4

- This is what the merge generated:



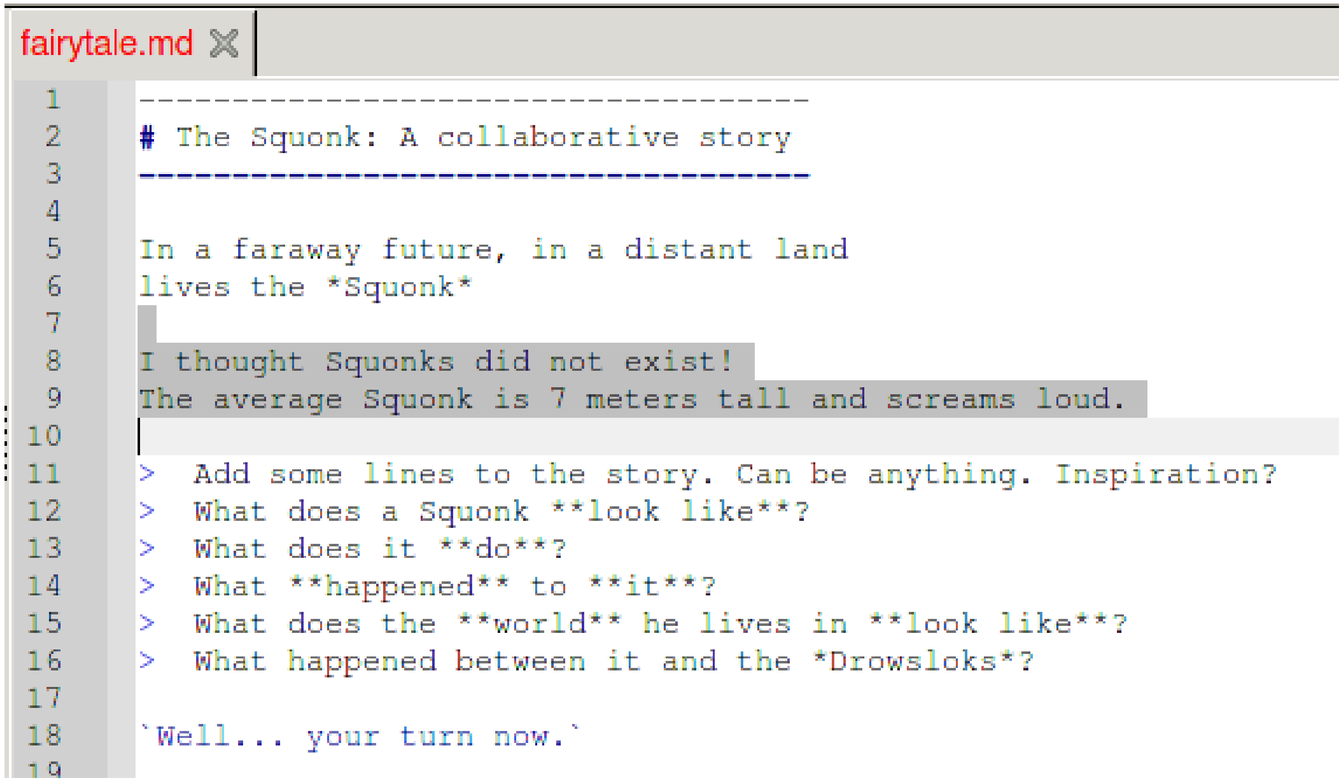
fairytale.md
HEAD (last commit) of the *mystory* branch
content of the same file in branch *main*

```
In a faraway future, in a distant land
lives the *Squonk*

<<<<<<< HEAD
I thought Squonks did not exist!
=======
The average Squonk is 7 meters tall and screams loud.

>>>>>>>  main

>  Add some lines to the story. Can be anything. Inspiration?
>  What does a Squonk **look like**?
>  What does it **do**?
>  What **happened** to **it**?
>  What does the **world** he lives in **look like**?
>  What happened between it and the *Drowsloks*?

`Well... your turn now.`
```

- The conflicts are identified with the special markers:
  <<<<<<< HEAD Beginning of the diffs in the HEAD file current branch
  ======= end of the diffs of the file in the current branch, beginning of the diffs in *main*
  >>>>>>> **main** end of the diffs in *main*
  all the other lines are **common** to the file in both branches

Conflict resolution from the command line 2/4

- The conflicts are identified with the special markers:
  <<<<<<< HEAD Beginning of the diffs in the HEAD file current branch
  ======= end of the diffs of the file in the current branch, beginning of the diffs in main
  >>>>>>> **main** end of the diffs in main
  all the other lines are **common** to the file in both branches
- To resolve the conflict, we need to edit the file to a state that we think is correct and remove all three markers.
- In my case I decided to keep all lines and then I saved:
- There are nicer tools to do merges like meld, you will find some infos about it in the end of this presentation, unfortunately there is no time to discuss it during the course.



fairytale.md

```
-----------------------------------
# The Squonk: A collaborative story
-----------------------------------


In a faraway future, in a distant land
lives the *Squonk*

I thought Squonks did not exist!
The average Squonk is 7 meters tall and screams loud.

> Add some lines to the story. Can be anything. Inspiration?
> What does a Squonk **look like**?
> What does it **do**?
> What **happened** to **it**?
> What does the **world** he lives in **look like**?
> What happened between it and the *Drowsloks*?

`Well... your turn now.`
```

Conflict resolution from the command line 3/4

- back to the terminal, we can now check with git diff the current **unstaged** changes:
- **git diff**

```
diff --cc fairytale.md
index 803a9e6,a98c877..0000000
--- a/fairytale.md
+++ b/fairytale.md
@@@ -5,8 -5,9 +5,9 @@@
  In a faraway future, in a distant land
  lives the *Squonk*

 +I thought Squonks did not exist!
+ The average Squonk is 7 meters tall and screams loud.

 -
  >  Add some lines to the story. Can be anything. Inspiration?
  >  What does a Squonk **look like**?
  >  What does it **do**?
```

  - Note how this new version of the file is not staged nor committed, so it gets **0000000** as fake commit hash.
- We can check for the file status, git will say that we're in the middle of a merge:

```
git status
# On branch mystory
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
# both modified:      fairytale.md
#
no changes added to commit (use "git add" and/or "git commit -a")
#
```

Conflict resolution from the command line 4/4

- We can now stage(add) the file, git will interpret it as if the conflict was **resolved**:

```
git add fairytale.md
git status
# On branch mystory
# Changes to be committed:
#
# modified:   fairytale.md
#
```

- Now we can commit:

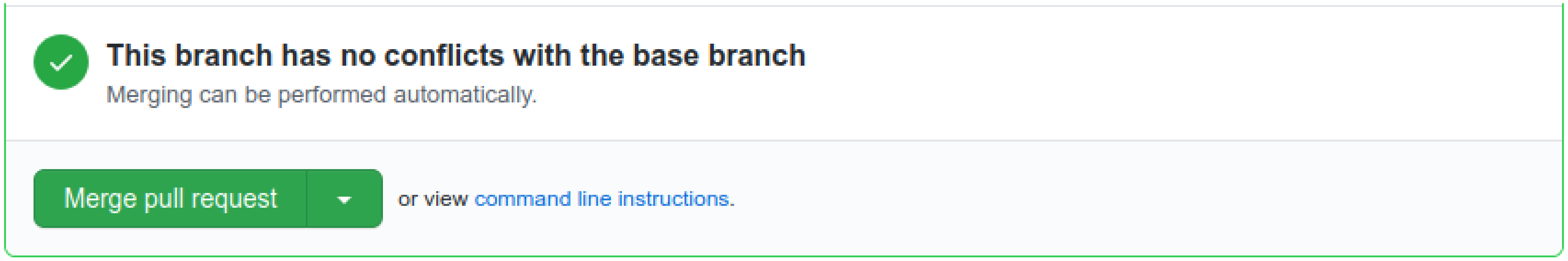
```
git commit -m 'Accepted both lines to complete the merge'
[mystory 11b512d] Accepted both lines to complete the merge
```

- And finally push the changes to origin:

```
git push origin mystory
Counting objects: 7, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 388 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local
objects.
To git@github.com:GITUSERNAME/MNXB01-fairytale.git
   2a4af07..11b512d  mystory -> mystory
```

The pull request is automatically updated

- Checking back on github, our pending pull request has been updated and now it’s hopefully good to go:



This branch has no conflicts with the base branch

Merging can be performed automatically.

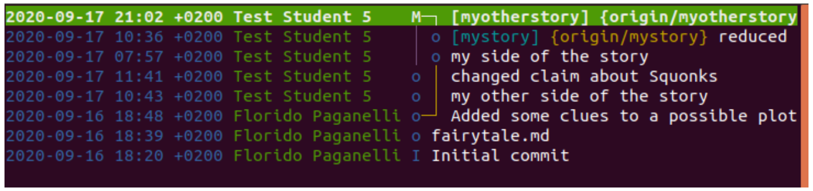
The release manager can now merge it to his **upstream** *main*.

Typical workflows summary 3/3: one project, many users, one or many servers

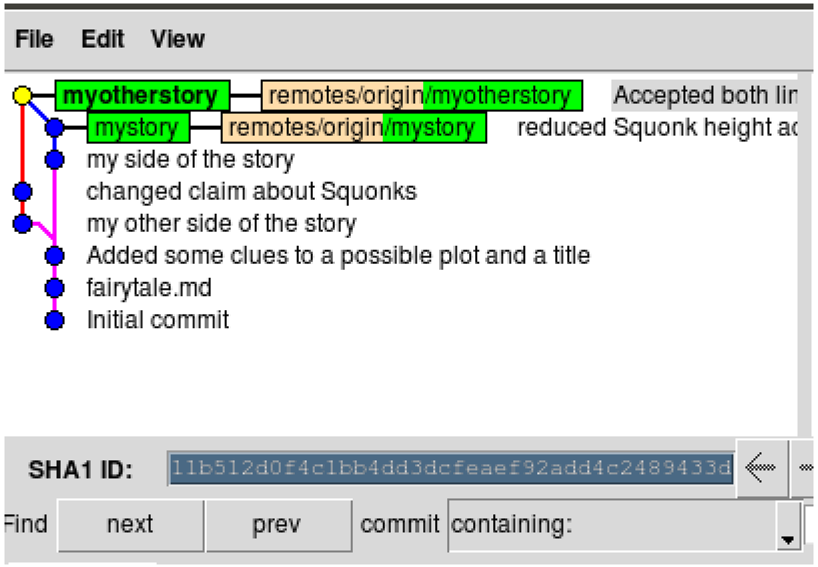
1) **Login** into github (or some git server)
2) **Fork** (make a copy of) a repository of some user community (the "**upstream**" repository).
   - The resulting forked project will be your "**origin**" remote repository
   - **IMPORTANT**: your forked origin an upstream DO NOT AUTOMATICALLY SYNC. THIS IS NOT DROPBOX.
3) **Clone** the origin repository on your computer. It will create your **working directory**
4) Set your remote "**upstream**" repository to the user community one. This is needed to keep in sync with the community.
5) **Pull (**or **fetch)** the latest updates (usually in *main*) from the **upstream**
6) Create a **branch** that is only for your changes or a specific feature you implement.
7) Work/save on your laptop, **add** and **commit** in your branch and in working directory
8) **Push/pull** branches and commits to/from your **origin**
9) When the work is done, **submit a pull request from your origin branch to the upstream *main*** (or *target branch*, depending on the development model**)** for the community to accept and review your changes.

Visualizing the changes network

- There are a lot of tools you can use to visualize the history of changes both local and on the remote repositories. They're called *git browsers*
- On Aurora I found **`gitk`**[1] (graphical). If you want a textual one is called **`tig`** but it is not available on Aurora.
- You can see the relationships between two branches by running:
  `command branch1 [branch2..]`
- Examples:
  `gitk` `main myother`
  `tig` `main myother`
- Show all branches:
  `tig` `*`
  `gitk` `*`



```
2020-09-17 21:02 +0200 Test Student 5     M┐ [myotherstory] {origin/myotherstory
2020-09-17 10:36 +0200 Test Student 5     │ o [mystory] {origin/mystory} reduced
2020-09-17 07:57 +0200 Test Student 5     │ o my side of the story
2020-09-17 11:41 +0200 Test Student 5     o │ changed claim about Squonks
2020-09-17 10:43 +0200 Test Student 5     o │ my other side of the story
2020-09-16 18:48 +0200 Florido Paganelli o─┘ Added some clues to a possible plot
2020-09-16 18:39 +0200 Florido Paganelli o fairytale.md
2020-09-16 18:20 +0200 Florido Paganelli I Initial commit
```



```
myotherstory    remotes/origin/myotherstory    Accepted both lir
mystory    remotes/origin/mystory    reduced Squonk height ac
my side of the story
changed claim about Squonks
my other side of the story
Added some clues to a possible plot and a title
fairytale.md
Initial commit
```

SHA1 ID: 11b512d0f4c1bb4dd3dcfeaef92add4c2489433d

[1] On Aurora, to enable a version of git that includes `gitk` you need to run this command:
`module load GCC/4.9.2 git/2.4.1`

Homework Tutorial 5

- This homework depends on how far in the tutorial we get during the session.
- In any case it may contain:
  - You will be required to manage a pull request submitted by me to your MNXB01-learn repository
  - You will be asked to contribute to a github repository and submit a pull request.
- **The official homework will come on canvas.**

A word on privacy and security

- When you fork my MNXB01-2022 repository and submit pull requests, everything will be public.
  Others will see your code.
  - It is perfectly ok for me because I believe one learns coding by looking at other people’s code and sharing/discussing coding with others.
  - If you’re not happy with it, you can create your own **private repository** to store the material produced during the MNXB01 course, so that **nobody else can see it**.
- The grading will be done on canvas, not on github
- If you prefer not to write your name on the github repository, you can write your nickname, but **make sure I know who you are**. I will not correct submissions if I don’t have a mapping nickname→student. You can send me this information privately if you don’t want others to know who you are.

Useful git commands

Setting your default editor with git

- If you commit without the `-m` option, git will automatically **open a text editor** for you to write a commit comment.
- It is good practice to:
  - write a commit title
  - leave a blank line
  - describe your commit in more detail.
- We will use `pluma` as the default editor, but you can use any editor you like.
- If you don't configure anything, the default is a text editor called nano, which for some is a bit weird at first. But I suggest to use it so you just use the command line. Press “CTRL + O” to save the file, ”CTRL + X” to exit.

Setting Pluma as the default git editor

- Run:

```
git config core.editor pluma
```

- Note that the commit will only happen ONCE when you save the file in geany.
- Test by running

```
git commit
```

- **If you don't like it, revert to default by writing**

```
git config --unset core.editor
```

Removing or renaming a file from the git database

- **Removing**: Sometimes one can decide that files in the directory should not be part of the repository anymore. Rather than deleting them with the rm command, one can use

  ```
      git rm filename
  ```

  - Remove a file using the above command.
  - Check the output of `git status` .
  - `git commit -m 'I have deleted file filename'`
    Remember: CLEARLY STATE that you removed some files in the commit message!

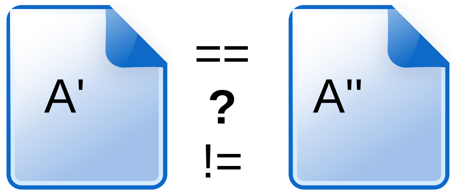
- **Renaming**: `git mv oldfilename newfilename` is equivalent to

  ```
    git rm oldfilename
  ```

  followed by

  ```
    git add newfilename
  ```

Textual and Graphical Diffing



- Run

  git diff

```
> git diff
Index: thisisfloridofile.txt
===================================================================
--- thisisfloridofile.txt (revision 6)
+++ thisisfloridofile.txt (working copy)
@@ -1 +1,2 @@
 Hello! this is florido's file.
+I am adding this change.
```

Line numbers of the two files:
-1 : showing line 1 of of file ---
+1,2 : showing lines 1 to 2 of file +++

- If you want a graphical tool to check the diffs, I suggest **meld**
  On Aurora there are two versions of meld. To enable one of the two run one of these commands:
  ```
  module load meld
  ```
- Use meld as the default diff tool:
  ```
  git config diff.tool meld
  git difftool thisisfloridofile.txt
  ```

Undoing not staged changes

- Say that we are not happy with the changes we just made **to a single file** and we want to go back to the latest commit (also called HEAD)
- Change one of the files in your repository and issue `git status`.
- The best to do is a **simple checkout** of the file from the last commit

```
git checkout thisisfloridofile.txt
git diff
```

- **Careful! You will lose all the changes done and not committed!!!**
- Note that this is equivalent to checkout the file at the **latest revision HEAD**:

```
git checkout HEAD thisisfloridofile.txt
```

- Checking out HEAD of all files in a directory will cancel all the changes done to the uncommitted files in that directory.

```
git checkout HEAD *
```

- **Play a bit with these commands by changing files and see what happens.**

Reverting to a previous revision

- Say that we don't like the current revision state, and we want to roll back the code to a state of a different revision back in time.
- The main suggestion is:
  try to never go back in the revision history.
  This is actually nice because in a collaborative environment, keeps track of who-did-what with no cheating allowed :)
  Unfortunately git allows for “cheating” by changing the revision history. It can be useful sometimes, but must be used with extreme care. **Changing the revision history gives no UNDO.**
- To experience with this, change some files and commit, then follow the next slides.

Reverting to a previous revision the safe way: revert

- The **revert** command restores the state of all files at a certain revision to the current working dir.
- Usually the output of a revert gives hints about the steps to take before committing.
- Make sure you have at least three commits (check `git log`)
- Create a fourth commit

Reverting to a previous revision the safe way: revert

- Try to git revert everything to your second commit in the log:
  `git revert commithash`
- Example:
  `git revert c9af94904c6868ef136d75730fbde63e0a15cf31`
- Read the git status output to see what changed
- Take action to make the files ready for commit, and commit
  - Git will automatically start a commit and open the text editor for you. It will add the “Revert commithash” comment to your commit and wait for your input.

Reverting to a previous revision the unsafe way: reset

- The reset command does something different. It does not preserve history and allows you to modify an existing commit. For a detailed explanation see https://www.atlassian.com/git/tutorials/undoing-changes
- Use it **only** on a **private branch** and **never** on a branch you share with others (typically a *main* or master branch)
- Additionally, I suggest to use it **only** when one of these two happen:
  - You already staged some changes to a file and you want to unstage them

    ```
        git reset filetounstage
    ```
  - You are totally unhappy with whatever you did so far and want to **unstage all staged files**:

    ```
      git reset
    ```

Fixing commit mistakes

- Commit allows you to amend or change the latest commit if, for example, you forgot a file or you wrote the wrong comment:

```
git commit --amend
```

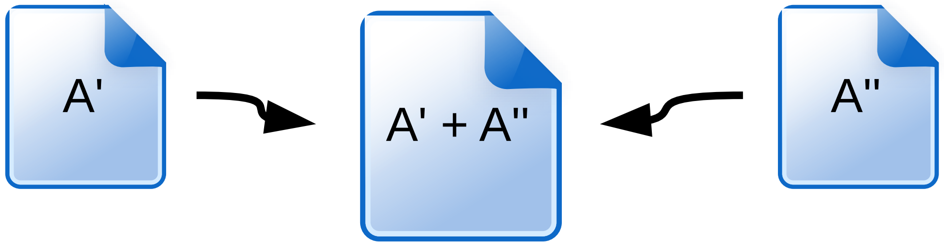
- Note that this will create a new revision hash, and will **DELETE** the previous commit hash. So be sure you are done with `amend` before you push to your remote repository.
- **NEVER DO THIS AFTER YOU PULLED YOUR LOCAL BRANCH TO A REMOTE REPOSITORY UNLESS YOU'RE THE ONLY USER OF THE REMOTE REPOSITORY.**
- Seehttps://git-scm.com/book/id/v2/Git-Basics-Undoing-Things

Graphical Clients

- Want to try a graphical client?
  - Other notable minimalistic ones:
    `gitg, qgit`
  - Feature-rich one (not available in repositories):
    `https://www.gitkraken.com/`
    - This one is NOT available on Aurora. You need to download it from the internet if you want the latest version.

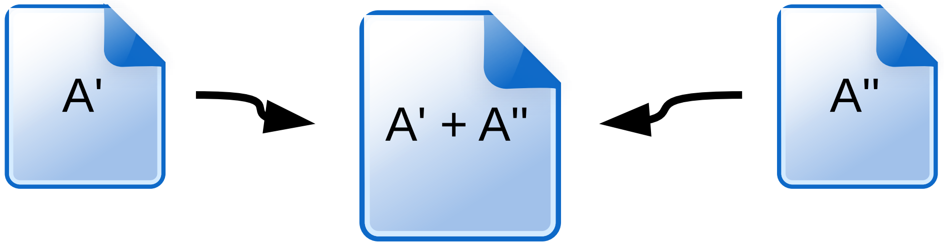
Additional material

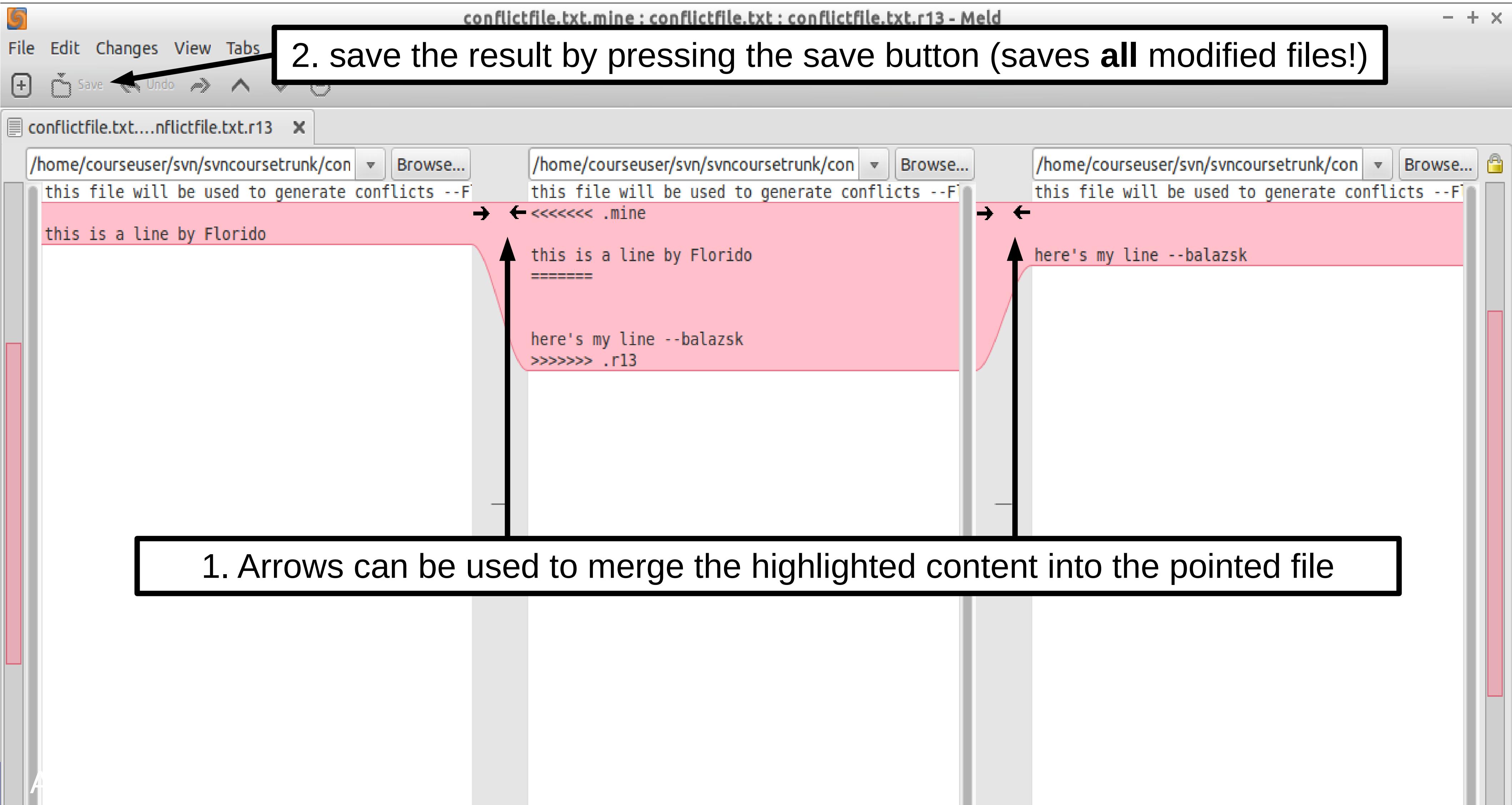
Merging



- Suppose we have two versions of a document with different contents
- We want to make one out of two
- This is often referred as **three-way-merge**
- We need to choose which part of each document we want to keep
- There exist tools to do it, for example the excellent **`meld`**
- git can attempt to do merges for us:
  - If the merges are simple, i.e. the changed content of A' can be easily mixed with that of the content of A''. For example, the documents differ a little but the changes in each document are not overlapping.
  - If we provide it with some hint on how to do the merges
  - If the above fail, it will ask us to do the merge manually, for example using `meld`
- The most frequent case of merge is in case of **conflicts** described in the slides.

Merging with meld




conflictfile.txt.mine : conflictfile.txt : conflictfile.txt.r13 - Meld
File
Edit
Changes
View
Tabs
Save
Undo
conflictfile.txt....nflictfile.txt.r13
Browse...

2. save the result by pressing the save button (saves **all** modified files!)

```
/home/courseuser/svn/svncoursetrunk/con
this file will be used to generate conflicts --F

this is a line by Florido
```

```
/home/courseuser/svn/svncoursetrunk/con
this file will be used to generate conflicts --F
<<<<<<< .mine

this is a line by Florido
=======


here's my line --balazsk
>>>>>>> .r13
```

```
/home/courseuser/svn/svncoursetrunk/con
this file will be used to generate conflicts --F


here's my line --balazsk
```

1. Arrows can be used to merge the highlighted content into the pointed file

Scenario 1
Git offline just to track your files

Create a work directory without a remote repository:

- If you have git installed, `cd` into any folder you want to track and initialize the git database, say `myongoingproject`:
  ```
  cd myongoingproject
  git init .
  ```
- This will create the hidden `.git` database folder and you can start working immediately with `add`s and `commit`s
- There will be no defined remotes, but you can add them later if you wish.

Local bare repository (optional, could be an external disk for backup):

- If you want to create your own remote, where you can push and pull, create a new folder somewhere, say `/externaldisk/mybarerepo`:
  ```
  mkdir /externaldisk/mybarerepo
  cd /externaldisk/mybarerepo
  git --bare init
  ```
- Note that you **cannot** `add` or `commit` in this repository. This is really just the database files. You can only `push` and `pull` to it. To do so, let's add it to the previous work directory:

  -
    ```
    cd myongoingproject
    git remote add mybarerepo file:///externaldisk/mybarerepo
    ```

Typical workflows summary Scenario 1 **personal** project, **one** user, **no server**

This is the simplest setup just to track your own code quickly. Instead of creating a repository on github, you create a git database directly in the directory you want to version.

1) Create a simple **local working directory** by transforming a simple directory into a **git tracked directory** with the command

```
cd dirIWantToVersion
git init .
```

2) Work/save in that working directory, **branch**, **add** and **commit** to track changes.

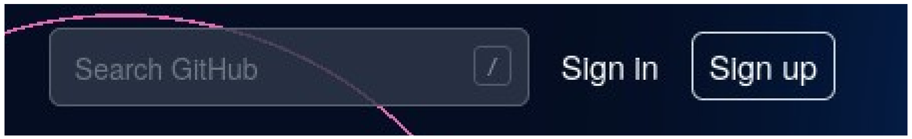
```
git add file1 file2 …
git commit -m 'done some changes'
```

Preparation for the git tutorial 1/7

- Since November 13th, 2020 Github no longer accepts usernames and passwords to upload your code from a cluster or your personal computer.
  https://github.blog/2020-07-30-token-authentication-requirements-for-api-and-git-operations/
- Among the suggested ways of logging in, there is **SSH keys**. I will teach you how to setup SSH keys for github. There are other methods but I do not think they suit this course.
  - For a detailed discussion of the SSH PKI technology and the commands read the MNXB01-manual.pdf . Here I will just show the practical commands to run for the tutorial.
  - For a detailed discussion why we suggest this method for this course read the dedicated slide about github command line access at the end of these slides.

Preparation for the git tutorial 2/7

- Create an account on https://github.com
- To create an account click on the “Sign Up” button in the upper right corner.



Preparation for the git tutorial 3/7

1) Login to Aurora (if possible...)

2) Generate a new SSH key pair with the following command:

```
ssh-keygen -b 4096 -f ~/.ssh/id_rsa_github
```

2.1) Choose a password for your private key.
!!! NEVER GENERATE KEYS WITHOUT PASSWORDS.
Read MNXB01-manual to understand why that is bad.

3) Test that your password is working with this command:

```
ssh-keygen -y -f ~/.ssh/id_rsa_github
```

3.1) If the password didn’t work, the program will say that you provided an “incorrect passphrase”. Start again from step 2

4) Copy (ctrl +c) and paste (ctrl+shift+v) this entire text below in your terminal to add to your ssh configuration information on how to login to github. You may need to press enter:

```
cat << EOF >> ~/.ssh/config

# Access to github
Host github.com
   HostName github.com
   IdentityFile ~/.ssh/id_rsa_github

EOF
```

5) check the ssh config file contains the lines added above.

```
cat ~/.ssh/config
```

Preparation for the git tutorial 4/7

6) make sure the files have the proper permissions

```
chmod 600 ~/.ssh/id_rsa_github
chmod 644 ~/.ssh/id_rsa_github.pub
```

7) Check the contents of **your public** key:

```
cat ~/.ssh/id_rsa_github.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQCjVKDNRkMUMdEsY25jfXGCMhXL/57L
XsX5Re11cJ7mMq91tTpzhV+miedOwq30+KM5iPlPoN3QpJlZ26BBcrUJ/+pury7rN/W/
YfYMb+KOez74j8eT1gNNYfArZZmKfe9XMFB73XYyChmDZZkEz7UuGPYA2TdDKGBA4cg
9MqrvXsnM8FbLfnKHBsu2rrRH51tJM7VlMkWrGwHv9UAsndoDEtaj0qaF0SaQ8qz+CK55
o7HSBSIr1/0uQwgH+yOPbaJvKORfXTp7ewIw3xDpYDtGpP744ZI+q4Bzg67c4DixHfMN
2PDbLSM1AdrfTIaLMVePAHTdptVtfl1AWHmtikqLhPLzK3H342kMauXj9ne27wh2lMf
XFIWg8vzOo+fmidjSQ9hFvczMeaKikvkpL16BF3CCS8st5TmkpyOtRohYvAehY/dpsMVV
9exbpnEt8yU6XVx25qJiuUls0p1iXtJdqESrHgS9VqFGMq9Ew9W21mPT7JX92vXpUZ0T
6yvFDfvOOd1Yy8/23ECzdyqpQyk43LrSpX38ELA3K0+8ZN0mpB+c8mxwTA0I/dCnCeS
6iiCrOhP87CA8Wb5MScS7Q94z+T3jn3wAXbR/uUbTtXJE/klykknbINfB8xo9
3cII9GIv9UxRQSMKeBWRZdH9bIAi1xoRhpAgENpAgylKr6DSQ== pflorido@aurora1
```

On your terminal:

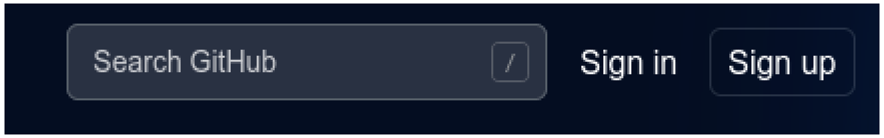
7.1) select **all the text** from `ssh-rsa` to `aurora1` (or whatever is in the end of your public key, it depends on the frontend node where you logged in)

7.2) copy the selection with with ctrl-shift-c (or right-click and then select *copy*)

We will use it later.

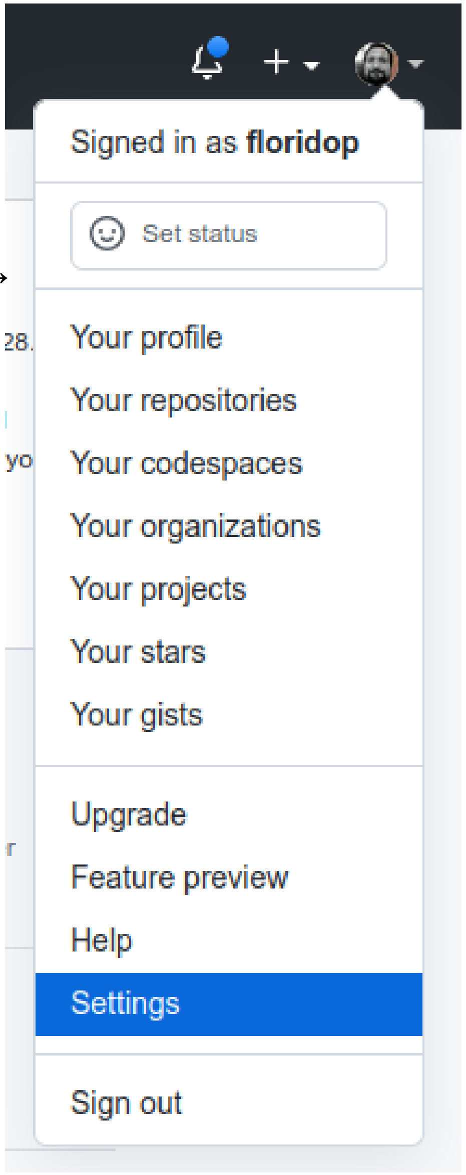
Preparation for the git tutorial 5/7

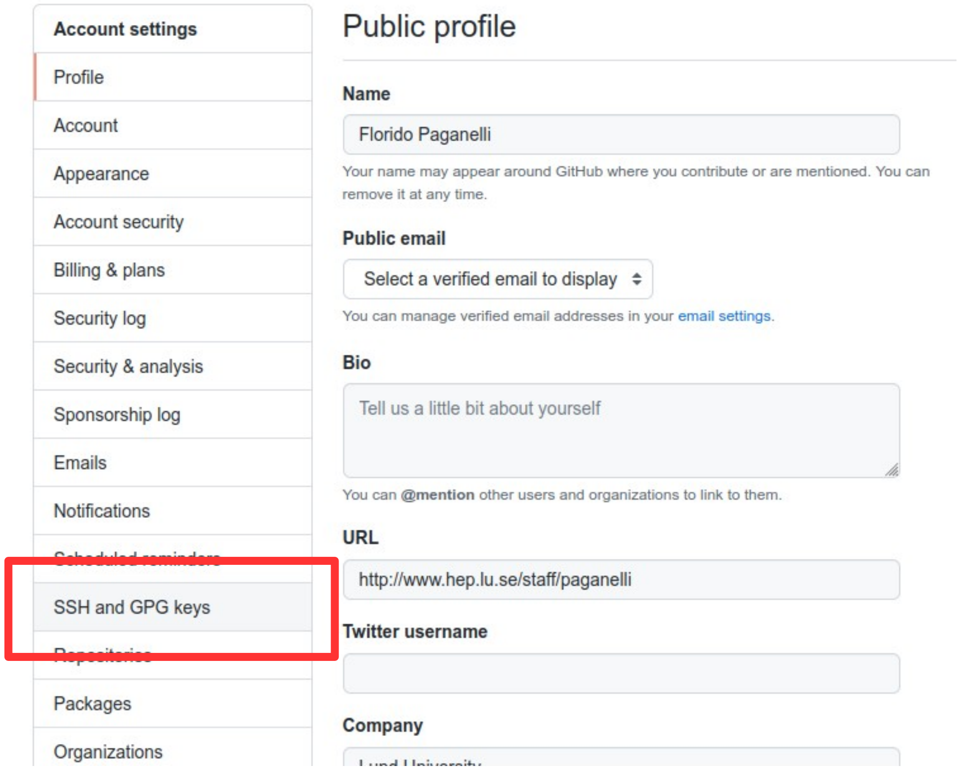
8) Sign in (login) on www.github.com



9) Click on your account avatar in the upper right corner and select "Settings" →

10) Click on "SSH and GPG Keys" in the lower left corner:





Public profile

Name: Florido Paganelli

Your name may appear around GitHub where you contribute or are mentioned. You can remove it at any time.

Public email: Select a verified email to display

You can manage verified email addresses in your email settings.

Bio: Tell us a little bit about yourself

You can @mention other users and organizations to link to them.

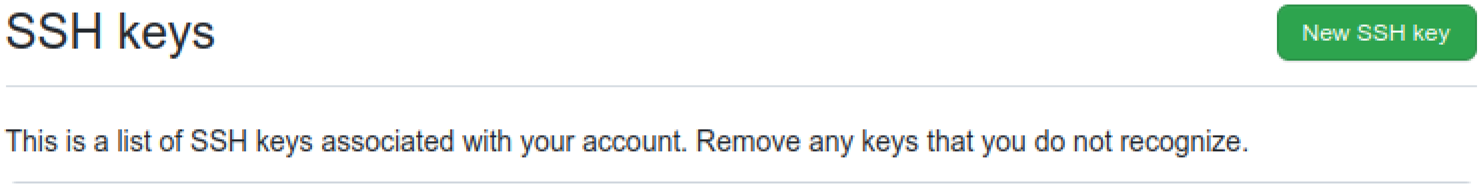
URL: http://www.hep.lu.se/staff/paganelli

Twitter username:

Company: Lund University

Preparation for the git tutorial 6/7

11) Click on New SSH key



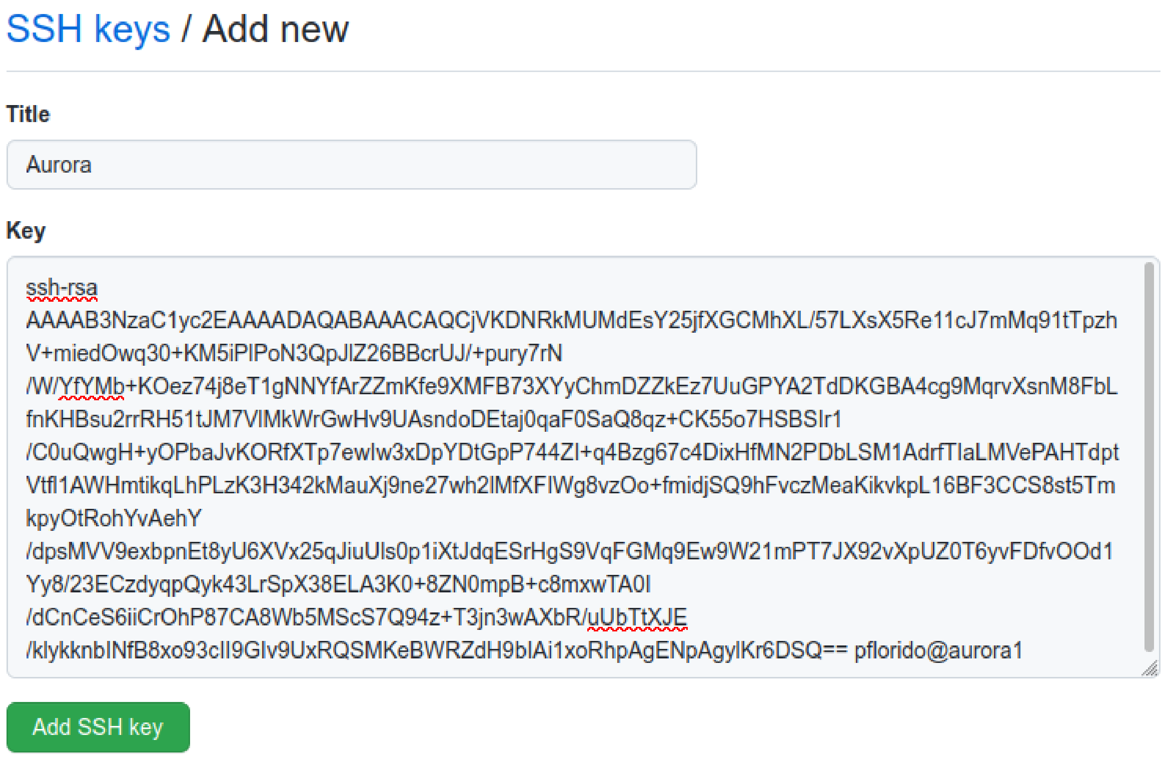
SSH keys

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

12) Paste the key copied at step (7.2) in slide 4 in the **Key** field and add the name **Aurora** to the **Title** field as in the picture below.

- PASTE YOUR KEY NOT MINE OR I WILL BE ABLE TO ACCESS AND HACK YOUR ACCOUNT! :D

- DO **NOT** PASTE YOUR **PRIVATE** KEY **EVER!**



SSH keys / Add new

Title

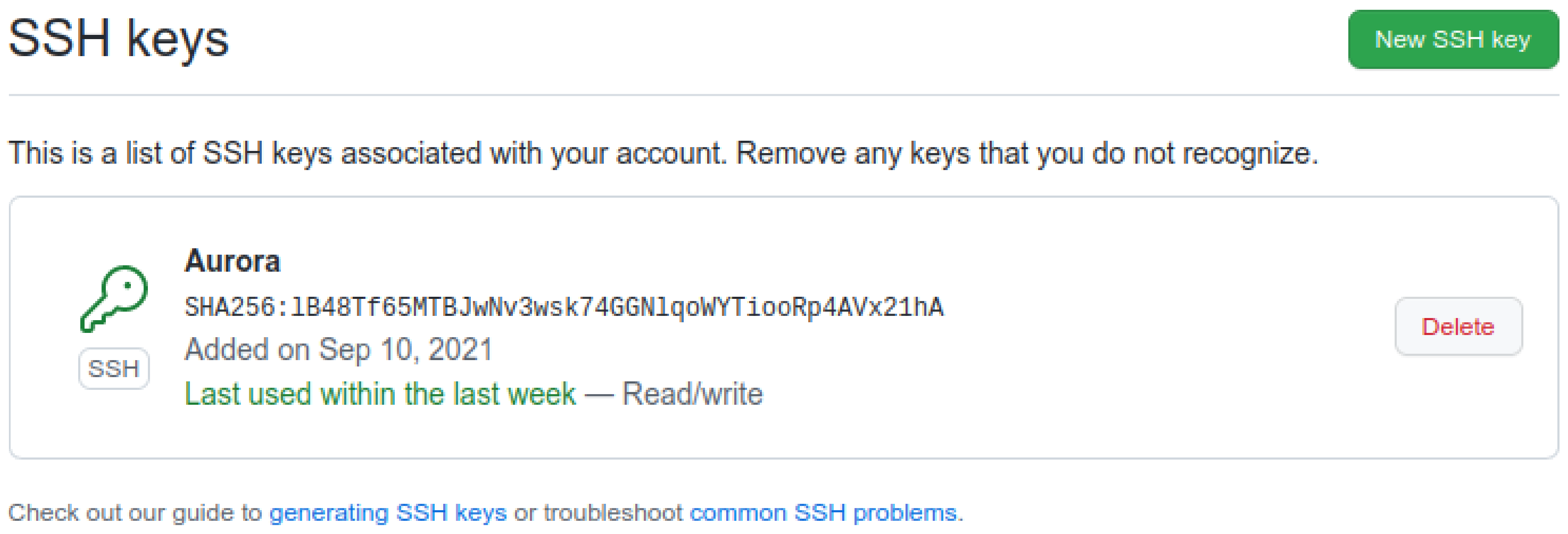
Aurora

Key

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAACAQCjVKDNRkMUMdEsY25jfXGCMhXL/57LXsX5Re11cJ7mMq91tTpzh
V+miedOwq30+KM5iPlPoN3QpJlZ26BBcrUJ/+pury7rN
/W/YfYMb+KOez74j8eT1gNNYfArZZmKfe9XMFB73XYyChmDZZkEz7UuGPYA2TdDKGBA4cg9MqrvXsnM8FbL
fnKHBsu2rrRH51tJM7VlMkWrGwHv9UAsndoDEtaj0qaF0SaQ8qz+CK55o7HSBSIr1
/C0uQwgH+yOPbaJvKORfXTp7ewlw3xDpYDtGpP744ZI+q4Bzg67c4DixHfMN2PDbLSM1AdrfTlaLMVePAHTdpt
Vtfl1AWHmtikqLhPLzK3H342kMauXj9ne27wh2lMfXFlWg8vzOo+fmidjSQ9hFvczMeaKikvkpL16BF3CCS8st5Tm
kpyOtRohYvAehY
/dpsMVV9exbpnEt8yU6XVx25qJiuUls0p1iXtJdqESrHgS9VqFGMq9Ew9W21mPT7JX92vXpUZ0T6yvFDfvOOd1
Yy8/23ECzdyqpQyk43LrSpX38ELA3K0+8ZN0mpB+c8mxwTA0l
/dCnCeS6iiCrOhP87CA8Wb5MScS7Q94z+T3jn3wAXbR/uUbTtXJE
/klykknbINfB8xo93cII9GIv9UxRQSMKeBWRZdH9bIAi1xoRhpAgENpAgylKr6DSQ== pflorido@aurora1
```

Preparation for the git tutorial 7/7

13) You should see something like this below.



SSH keys

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

Aurora

```
SHA256:lB48Tf65MTBJwNv3wsk74GGNlqoWYTiooRp4AVx21hA
```

Added on Sep 10, 2021

Last used within the last week — Read/write

Check out our guide to generating SSH keys or troubleshoot common SSH problems.

Validating the github server SSH identity

- As you know from the MNXB01-manual.pdf, there is one missing piece of information:
  What is the server SSH Key hash?
- This can be found at the following pages:
  https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/githubs-ssh-key-fingerprints
- Be sure to check that page before connecting!

Github command line authentication methods

- The official github documentation regarding these methods is here: https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/about-authentication-to-github#authenticating-with-the-command-line
- There are three suggested ways:
  - **Token Authentication**, which is the recommended way, same as username and password but you get a special password from the github website for each repository. The token is very hard to remember. It's something like
    `==mjd//#&vkhnwkbn237y61398hiw--`
    So the problem here is since no one remembers this, github suggests you to use their own client called `gh` that will securely store your tokens on whatever machine you use it.
    I personally think it is not appealing for this course. We do not want to teach you how to use one specific vendor-locked tool that only works with github. If you want to use this, read their documentation here:
    https://docs.github.com/en/get-started/getting-started-with-git/caching-your-github-credentials-in-git
    most likely, **this tool will disappear with github**. I do not consider it valuable knowledge.
  - **SSH Keys,** which is more or less how we login to Aurora already. I will teach you this because **you can use it in many other ways**, such as for logging in to Aurora, university servers, login to any other possible revision control server out there (gitlab, cvs, svn, mercurial...)
    - **It's a good piece of knowledge that will not disappear in a year or so – it has been on for more than 20!**
  - **Authorizing for SAML single sign-on**: this is only relevant for companies and it involves anyway one of the two solutions above already set up.

Pushing to github origin main: configuring the remote authentication in the case of token-based for older git versions

- github will only allow you to push if you authenticate to it. Authenticating to a remote server requires a bit of configuration.
- We will redefine the origin URL to take into account your github ID. Different versions of git do this in a different way, the one I am showing is the best on Iridium that has quite an old version.
- add a new origin URL with your full username in the URL path:

-
  ```
  git remote set-url origin https://GITUSERNAME@github.com/GITUSERNAME/MNXB01-
  learn.git
  ```
  - example for my user:
    ```
    git remote set-url origin https://floridop@github.com/floridop/MNXB01-learn.git
    ```
- Check that the setting is correct with
  ```
  git remote -v
  origin   https://floridop@github.com/floridop/MNXB01-learn.git (fetch)
  origin   https://floridop@github.com/floridop/MNXB01-learn.git (push)
  ```

References

- git cheat sheets:
  https://training.github.com/
  https://jan-krueger.net/wordpress/wp-content/uploads/2007/09/git-cheat-sheet.pdf
  https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet
- Quick guide to git
  http://rogerdudler.github.io/git-guide/
- Merging with meld
  https://lukas.zapletalovi.com/2012/09/three-way-git-merging-with-meld.html
  https://www.youtube.com/watch?v=3Qynj8WUwgs
- Reverting
  https://www.atlassian.com/git/tutorials/undoing-changes
  https://git-scm.com/book/id/v2/Git-Basics-Undoing-Things

Pictures references

- https://openclipart.org/
- http://www.libreoffice.org/