git for Astrophysicists

by Henriette Wirth

1 Installing and setting up git

To install `git` please use the guide for your OS on:
`https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

After installing `git` you need to set your name and email address:

```
$ git config --global user.name "Jean-Luc Picard"
$ git config --global user.email jpicard@starfleet.ufp
```

2 Working with repositories

2.1 Initializing git

Imagine a visitor from Betelgeuse was coming to Europe and wants to know about the local animals. So you are tasked with creating a database for them containing all the important animals with their properties, but of course you want to be able to backup your progress on this project. So the first thing you do when starting the project is to initialize `git` in your working directory.

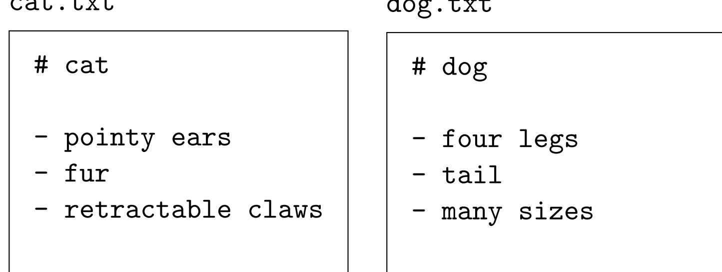
```
$ git init
```

After this command a folder with the name '`.git`' should appear in your working directory.

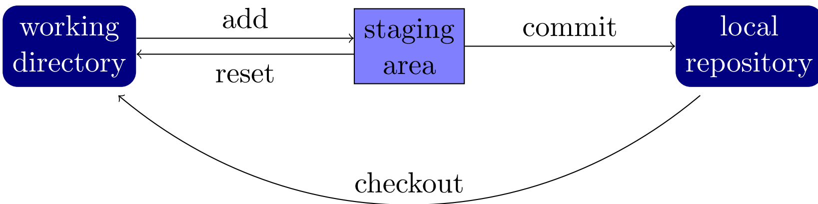
2.2 Moving files between your Working directory and the local repository

> `git` has typically three places, where your data is stored. One is the *working directory*, this is the directory that contains the files you are actively modifying. The *local repository* is contained in the `.git` directory and it contains a local copy of your repository. The *remote repository* is a copy of your repository on a server. Multiple remote repositories can exist for one project.

At the moment all three locations are empty. So let us create the first files for our project:



These files exist now in the working directory, but not in the local or remote repository. So let's look at how we can move them to the local repository. To move files to the local repository we go over a virtual space called the staging area.



To add a file to the staging area we use the command '`git add`'. Let's add the file '`cat.txt`' to the staging area:

```
$ git add cat.txt
```

We can check our staging area using the command '`git status`'. If our file was added correctly the output should look like this:

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
 new file:   cat.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
 dog.txt
```

This output tells us that we are working on the branch '`master`', which is the default

branch. We will learn how to use branches later. It also tells us that the file ‘cat.txt’ is currently staged which means that it is going to be added to the local repository during our next commit. The file ‘dog.txt’ is currently which means that it will not be added to the local repository. If we decide that we do not want to move ‘cat.txt’ to the local repository after all, we can remove it again from the staging area using ‘git remove cat.txt’.

Files can also be added and removed using the wildcard ‘*’. For example ‘git add * ’ adds all files from the working directory to the staging area, while ‘git add *.txt’ adds all files ending with ‘.txt ’.

In a lot of projects there are files that we do not ever want to commit. In software projects this might be the compiled program and the ‘.o’ files generated during compilation, in Latex projects it might be the resulting ‘.pdf’ file and the ‘.aux’, ‘.log’, ‘.out’, and the ‘.synctex.gz’ files. These files can be excluded by creating a textfile named ‘.gitignore’ on the top level of the working directory. Wildcards can also be used in the ‘.gitignore’ file. This would be an example of the ‘.gitignore’ file for a C++ project that is compiled into the file ‘prog’:

```
prog
*.o
```

Feel free to experiment with different ‘.gitignore’ files for our animal project. After staging all the files we want to add to our local repository, we can commit the files using ‘git commit -m "<comment>"’. Let’s for example move all of the files in our current project into the local directory using ‘git add *’ and then ‘git commit -m "added cats and dogs"’.

To check your local repository we can use the simple command ‘git log’, but I recommend a program with a gui like GITG. On Ubuntu and its derivatives you can easily install it using:

```
$ sudo apt install gitg
```

After the installation it can be called as ‘gitg’ from your local working directory. If run it on the current project the result should look something like in Fig. 1. On the left section of the screen we see, that we have currently one local branch called master (again, we will discuss branches later) and no remotes or tags (we will add those later too). On the top right we see the commit we just did including the comment we added, who was committing it and when and what we call the *hash*. The hash is a unique 40-digit combination of letters and numbers that git assigns to each commit. It is not relevant for us right now, but it might be useful later if we want to go back to this specific version of the project. Below is a list of files that were changed and the changes

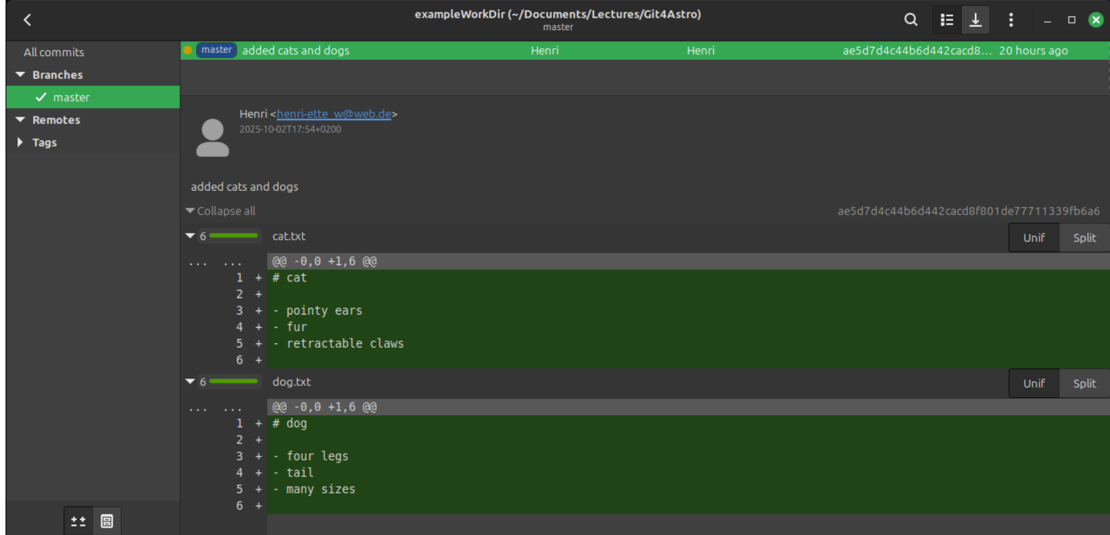


Figure 1: the first commit

made to them. In our case this is the first commit, so we only added information. On your computer, the contents of the local repository are stored in the '`.git`' folder, but luckily you will never have to look inside it.

Now that our files are in the local repository, we can always get our files back from the local repository if they happened to be lost. Feel free to delete the files and then get them back using:

```
$ git checkout master <file>
$ git checkout master *
```

The first command only allows you to *checkout* specific files, while the second one checks out all the files on the master.

For completeness let us see, what happens if we change our files and commit those changes. Let's change the file '`dog.txt`' and add '`horse.txt`'

`dog.txt`

```
# dog

- four legs
- tail
- many sizes and colours
```

`horse.txt`

```
# horse

- big mammal
- three legs
- mane
```

Let's commit these changes:

```
$ git add *
$ git commit -m "appended dogs and added horses"
[master 14d42ee] appended dogs and added horses
 2 files changed, 7 insertions(+), 1 deletion(-)
 create mode 100644 horse.txt
```

After executing this command let's have another look at gitg. It should now look like in Fig. 2. On the top-right we see the history of commits on the branch '`master`'. git repositories always store the history of all your projects so you can go back any time to see what you have done. On the bottom right we see the changes committed. As the unit git works with is the line adding the words '`and colours`' to the file '`dog.txt`' leads to the deletion of the line '`-many sizes`' and the insertion of the line '`- many sizes and colours`'. The six lines in the file '`horse.txt`' are all counted as insertions, which leads to the 1 deletion and 7 insertions mentioned in the terminal output.

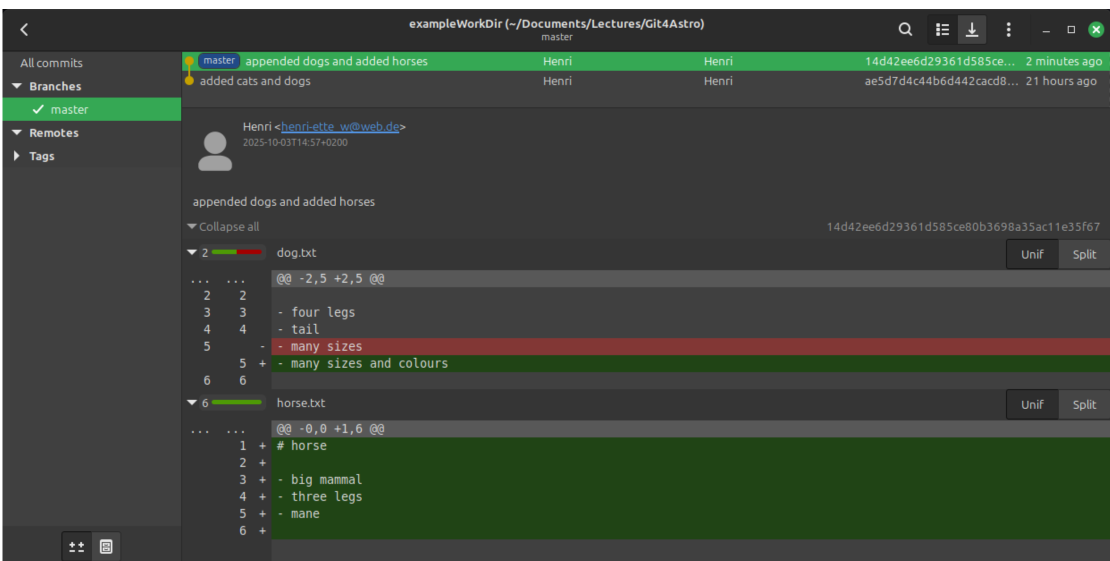


Figure 2: the second commit

2.3 Branches and Detached Heads

Our alien visitor decides that they want to expand their trip to Australia. So we bring in an expert on Australia to add their local species. But of course we want to continue adding European animals at the same time. So how can we both work on the same project? The answer is to give everyone their own branch. So we let our Australian colleague create their own branch using:

```
$ git branch Australia
```

Looking at GITG in Fig. 3, we see that a new branch named '`Australia`' is listed on the left side and from the history on the right side we see that it is identical to the original '`master`'. However, the check mark on the left tells us, that we are still following the master branch, which means that all changes we commit will still end up on the '`master`' and everything we checkout also comes from there.

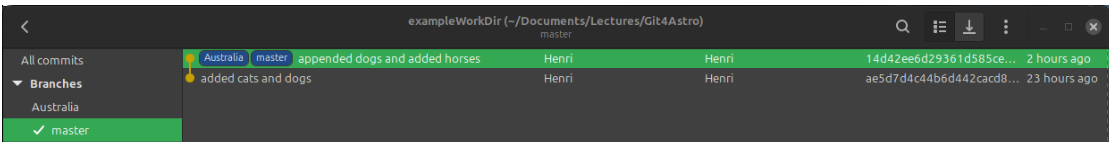


Figure 3: creating a new branch

Therefore, our Australian colleague must switch branches using:

```
$ git checkout Australia
```

> Note that checkout on a branch automatically extracts the files from the local repository to the working directory. Since '`master`' and '`Australia`' are identical, it won't make a difference in this case, however, when switching branches later on this can significantly change the content of the working directory.

The two command above can also be combined to one:

```
$ git checkout -b Australia
```

This creates a new branch and checks it out right away. A list of available branches can be viewed using '`git branch`'.

Now he adds two files:

`kangaroo.txt`

```
# kangaroo

- strong hind legs
- likes to jump
- big ears
```

`swan.txt`

```
# swan

- bird
- black feathers
- likes to swim
```

In the meantime we keep working on the 'master' and also add two animals:

cow.txt

```
# cow

- has horns
- likes grass
- is not naturally purple
```

swan.txt

```
# swan

- bird
- white feathers
- likes to swim
```

Looking at the result in GITG gives:

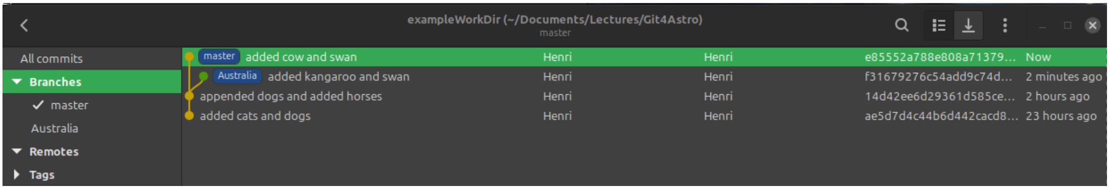


Figure 4: creating a new branch

As we can see in Fig. 4 two different versions of the project exist now on two different branches. However, the alien visitor does of course only want to read through one database. Therefore, let us merge the Australia branch into our master branch. The command for this is:

```
$ git merge Australia
Auto-merging swan.txt
CONFLICT (add/add): Merge conflict in swan.txt
Automatic merge failed; fix conflicts and then commit the result.
```

This resulted in an error message. GIT is very good at merging automatically if files are identical on both branches or exist only on one branch. However, in our case the file 'swan.txt' exists on both branches with conflicting information. Therefore git copied the file 'kangaroo.txt' over from the branch 'Australia' as it is since it is not conflicting with anything. However, it modified the file 'swan.txt' in the following way:

swan.txt

```
# swan

- bird
<<<<<<< HEAD
- white feathers
=======
- black feathers
>>>>>>> Australia
- likes to swim
```

Note that GIT leaves the lines identical as they are and only highlights the lines that differ from each other for us. We can now change this file into what we want it to look like in the final project and commit it.

However, when dealing with large files it can also help to use a merge-tool. One such tool is MELD, which can be installed using:

```
$ sudo apt install meld
```

After installing MELD we can use it as a mergetool, using the command:

```
$ git mergetool -t meld
```

This will open MELD looking like Fig. 5. The window is divided in three parts: the left part shows the state of the file as it is on the '`master`' branch that we want to merge into, the right part shows the version of the file on the branch '`Australia`' that we want to merge into the '`master`' and the middle shows the final result as it will be in the merged version.

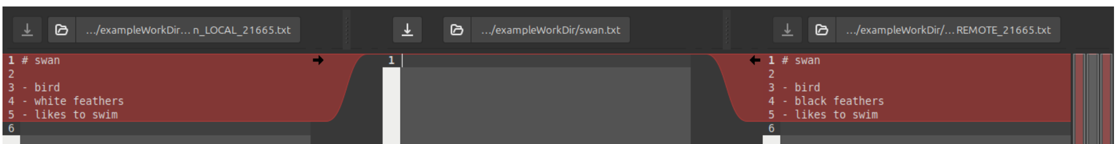


Figure 5: meld before the merge

Currently, the middle part is empty. So let's click on the black arrow between the left and the middle part to move the version currently on the '`master`' into the middle section. The result can be seen in Fig. 6. We see that meld highlights the differences between different files for us. If a line differs between two files it is highlighted in blue, while

a line that only exists in one of the files is highlighted in green. We can now use the arrows to move changes into the resulting file or we can edit it manually.

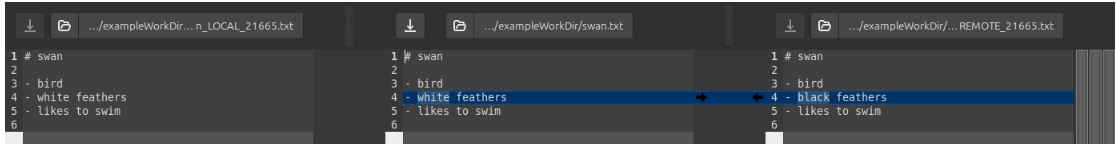


Figure 6: meld after importing the version on the '`master`' branch.

Let's edit the file as seen in Fig. 7 and then save our changes using the button above the document. After we close meld we will notice that the file '`swan.txt`' now matches the file we created in the middle of the MELD window. This method of merging files is especially useful for large files with many changes, where it is easy to loose track of the differences.

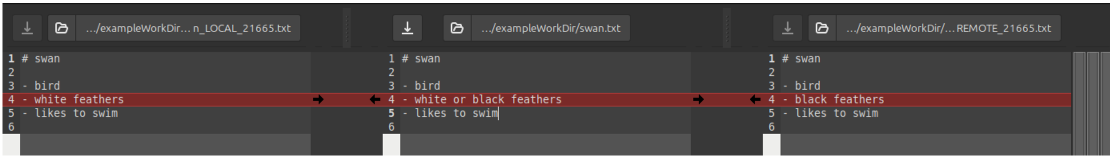


Figure 7: meld after importing the version on the '`master`' branch.

We can now commit our merge using:

```
$ git commit -m "merged Australia into master"
```

In Fig. 8 we can see the result. We can easily follow the history of the '`master`' through both the '`master`' branch itself and through the branch '`Australia`'. We also see that the merger did not affect the branch '`Australia`', so if we were to check it out again, we would end up with the same version of it as we were before merging it into the master and could continue working on it.

We can also set meld as the default mergetool using:

```
$ git config --global merge.tool meld
$ git config --global mergetool.prompt false
$ git config --global mergetool.keepBackup false
```

The first line sets the mergetool, the second one skips the confirmation dialogue before using the tool and the third one prevents git from saving a backup file after merging.

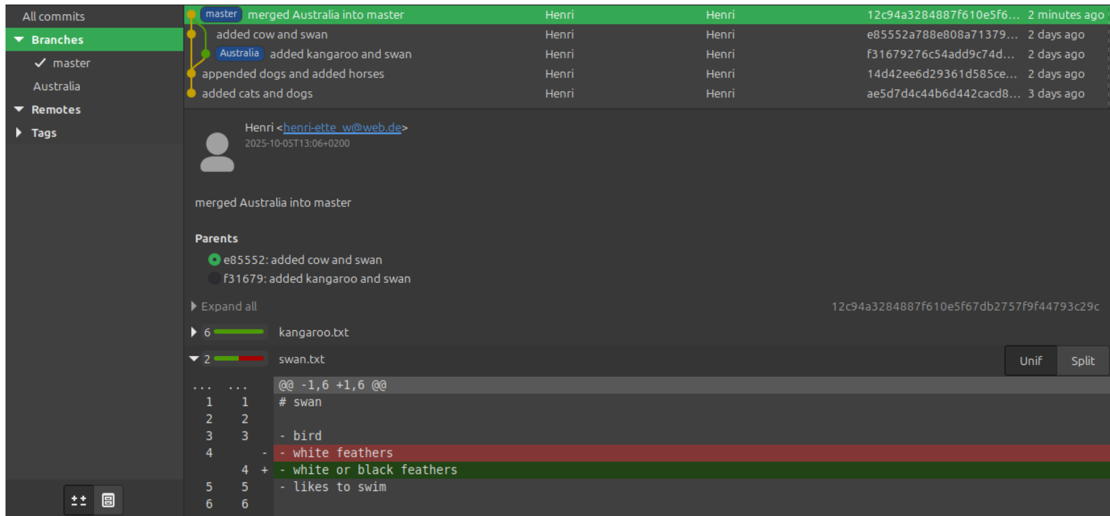


Figure 8: gitg after merging ‘`Australia`’ into ‘`master`’

Similarly we could also set meld as a difftool to show us the differences between different commits:

```
git config --global diff.tool meld
git config --global difftool.prompt false
```

This will make meld show the differences between two commits when prompted using:

```
git difftool
```

We previously only checked out branches, but we can also check out commits based on their specific hash. If for example we wanted to check out the initial commit again this leads to:

```
$ git checkout ae5d7d4c44b6d442cacd8f801de77711339fb6a6
Note: switching to 'ae5d7d4c44b6d442cacd8f801de77711339fb6a6'.

You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this state without impacting any branches by
switching back to a branch.

If you want to create a new branch to retain commits you create, you
```

```
may do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to
false

HEAD is now at ae5d7d4 added cats and dogs
```

As we are not at the '`head`' of a specific branch, we cannot commit any changes in this state, but we can create a new branch using '`git branch`' and make commits onto that.

2.4 Cherrypicking

Our visitor decides that they also want to see Africa. So we get an expert for Africa on board who makes commits visible in Fig. 9.

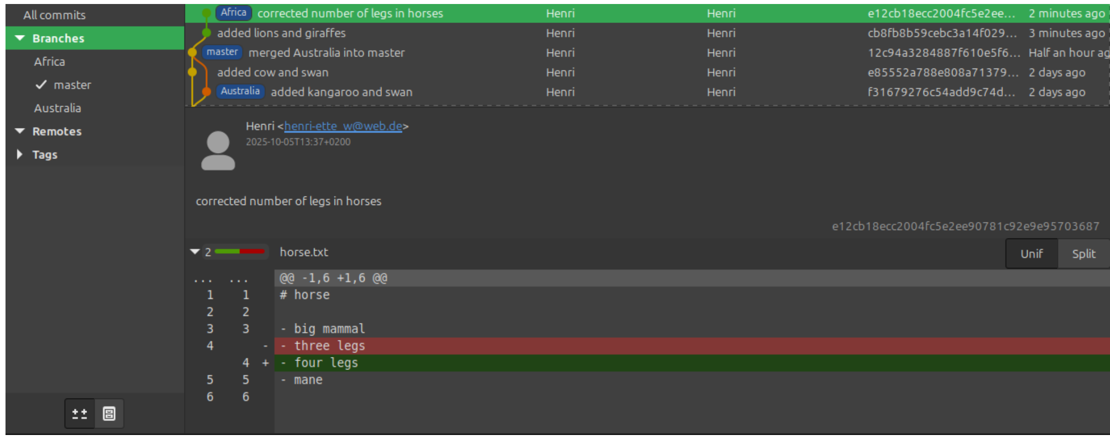


Figure 9: gitg with information about Africa

What is that? They found a bug in our old version. We accidentally wrote that horses have three leg and not four! Now, we do of course want to fix this on the '`master`' too, but we do not want to merge their half-finished work yet. We could just make the same changes they did to their files, but we also might think of using a feature of git called cherry picking. Cherry picking means copying the changes of one or multiple commits

onto another branch. The command for this is:

```
git cherry-pick e12cb18ecc2004fc5e2ee90781c92e9e95703687
```

where the number at the end is the hash we copied out of GITG. The result can be seen in Fig. 10.

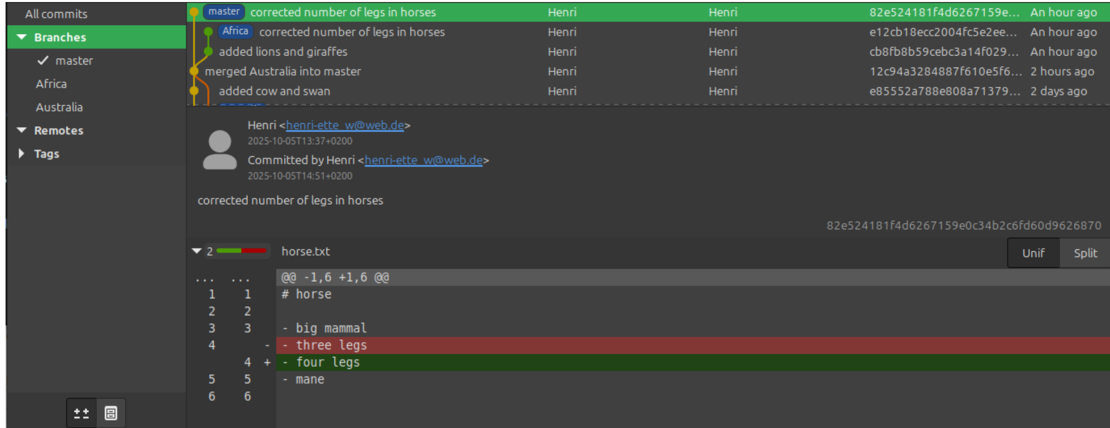


Figure 10: gitg after cherrypicking

2.5 Tags

At some point in our project we want to publish the first official version. This version shall be marked using a *tag*.

> A *tag* is a fixed name for a given commit in our repository which can be used to find that commit more easily.

We are going to use semantic versioning[1] for our tags and therefore call the first one '1.0.0'. To create the tag we use:

```
git tag 1.0.0
```

The result can be seen in Fig. 11. We can clearly see the tag 1.0.0 next to the head of '`master`'. Unlike the branches, this tag cannot be changed any more. We can ask git to list all existing tags in terminal using `git tag`.

[1] *Semantic versioning* is a versioning concept that involves three numbers: major.minor.bugfix separated by dots. The bugfix is incremented, if the change from the previous tag merely involve a bugfix, the minor if new features were added and the major in case of an interface change.

Figure 11: gitg after tagging

We can also create annotated tags. In this case the above command would change to:

```
$ git tag -a 1.0.0 -m "some comment about this tag"
```

3 The remote repository

So far we have only worked on the local repository, but eventually we want to backup our project on a server and share it with our colleagues. For this we must create a remote repository. The easiest way to create a remote repository on a server is to first create it locally and then move it to the server using SCP. To create the repository locally let's just move one level above the working directory and use the following command:

```
$ cd ..
$ git clone --bare exampleWorkDir/ repo.git
```

The command '`git clone`' copies the content from the folder '`.git`' into the folder '`repo.git`'. The '`--bare`' defines that no working directory is to be associated with this repository. Now that we have our remote repository we can move it to the server, where it ultimately shall exist.

```
$ scp -r repo.git user@server:/path/on/server/repo.git
```

We can delete '`repo.git`' on our machine after this is done. Now we want to tell git, where to find the remote repository. Therefore, we enter the following command into our terminal from the working directory:

```
$ git remote add origin user@server:/path/on/server/repo.git
```

This adds a remote named '`origin`' with the path '`user@server:/path/on/server/repo.git`' to the known repositories. A list of remotes can be viewed using:

```
$ git remote
$ git remote -v
```

with the first only showing the names and the second giving additional information like the paths. Now the remote repository is connected, but looking into GITG we find that we do not see any branches from the remote yet. To connect find the branches we have to update our information on the remote using:

```
$ git fetch
```

In Fig. 12 we can see that all the branches that we have in our local repository are also in the remote. If we make changes to out local repository we can use '`git push`' to copy changes on the current branch to the local repository.

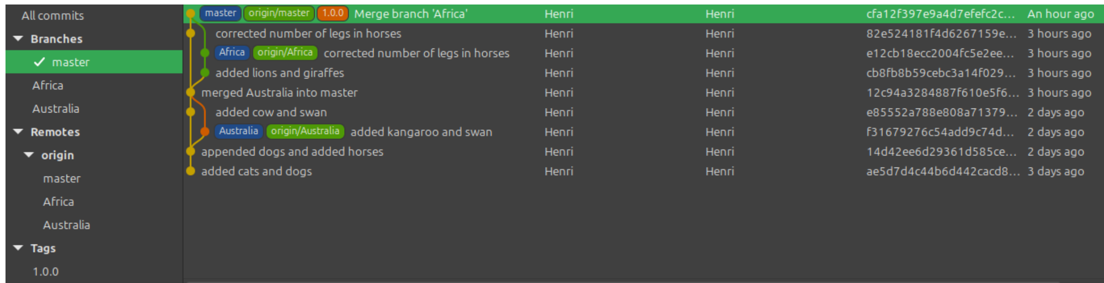


Figure 12: the remotes visible in GITG

The first time we use this command we will have to use:

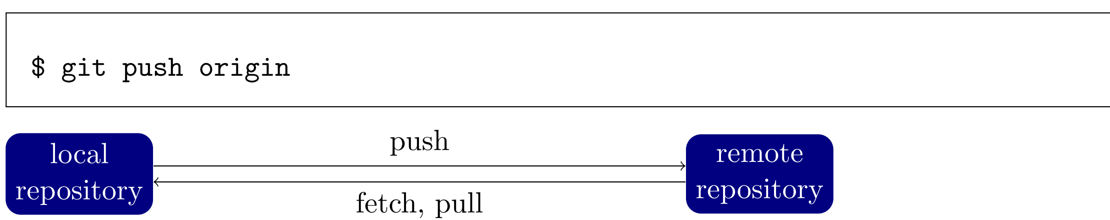
```
$ git push --set-upstream origin master
```

to clarify that we want our local '`master`' to track the '`master`' on the remote repository. To push tags we need to add the flag '`--tags`'. To move branches from the remote repository to the local repository we use '`git pull`'. If we are working with multiple remote repositories we can specify the target repository at the end of the command:

```
$ git push --set-upstream origin master
```

If we are working with multiple remotes we can specify which one we are pushing to:

```
$ git push origin
```



If a remote repository already exists and we want to create a local repository to work on it we can use:

```
$ git clone user@server:/path/on/server/repo.git
```

in the folder in which we want to create our working directory.

4 Stashing

In the beginning I mentioned that GIT has three places where data is stored *working directory*, *local repository* and *remote repository*. There is a fourth hidden location called the '*Stash*'. The *Stash* can be used to temporarily store data from the *working directory* to be free to look around the rest of the repository without having to create a commit.

Let's assume our Africa expert stumbles appon a problem in on of their commits that they want us to take a look at. We are in the middle of adding new animals for Europe. So we just move our work into the stash using:

```
$ git stash
```

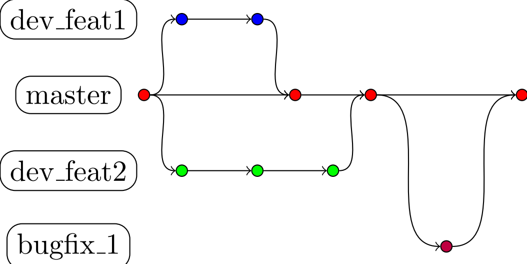
We are now free to checkout their work and have a look. After we are done with it we simply checkout our own branch again and restore the work we just did using:

```
$ git stash pop
```

5 Working in groups of developers

Now that we understand the basics of GIT, we can start developing our own projects with our colleagues. But how does one keep big repositories with many users clean? We already know half of the answer to this, which is to use multiple branches. A common strategy in software development is to declare the master a special branch that may only contain fully implemented, tested features. For every feature that gets added a new

feature branch is created, for each bugfix to be made a bugfixbranch is made. This can look something like this:



Here the naming convention is to start the name of each feature branch with 'dev_' and the name of each bugfix branch with 'bugfix_'. The developers working on those branches are not allowed to merge anything in the '`master`' unless it is fully tested and documented. Huge software companies even make their developers pear-review each others code before it can be merged. You might also find that they use branches called release branches on which specialized testing teams do a lot of additional software testing before anything gets tagged and then shipped to the customer. To make sure that only the last commit of a branch is added to the '`master`' and not the entire history the '`--no-ff`' flag should be used when merging:

```
$ git merge --no-ff def_feat2
```