Git Research

Since Wendy was absolutely new to Git, she started her research from the official Git documentation:

https://git-scm.com/

Although she spent quite a lot of time familiarising with Git, Wendy decided to write a summary of the main Git functionalities in order to speed up the To-Do List project development. She then shared the document with her co-workers.

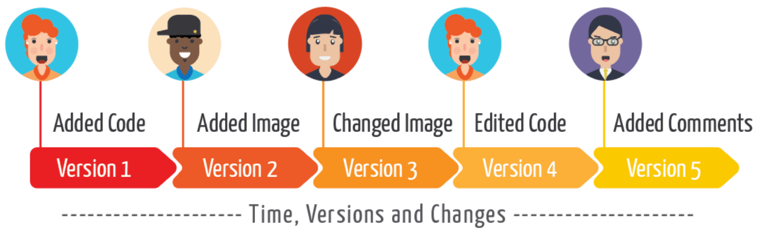
What is Git?

Git is by far the most widely used modern version-control system in the world. It is essentially designed to help you keep track of changes in source code, files and project structure during software development.

You might not know it but even if you are new to Git, you do use a similar approach on a daily basis. In fact, when working on files you create, save, edit and eventually save them again. Although this is a great strategy to manage your files, there is really no way to keep track of the history; what if you want to view how your files looked 3 saves ago?

Here is where Git shines. With Git you can keep track of all the changes you made in time to a particular project and retrieve older versions if needed. Git also facilitates collaborative changes to a project. This means that multiple people can work on the same directory and bring the changes together for a unified effect while working at the same time.

Long story short, Git can be seen as a timeline where it is possible to access different versions of a particular project, modify, remove or add functionalities. The timeline can be edited by more people at the same time, making Git a great collaboration tool.



Basic Git requirements

In order to get started with Git you will need:

- Access to your computer Terminal (also known as Command Prompt)
- The Git package installed on your machine

Git Installation

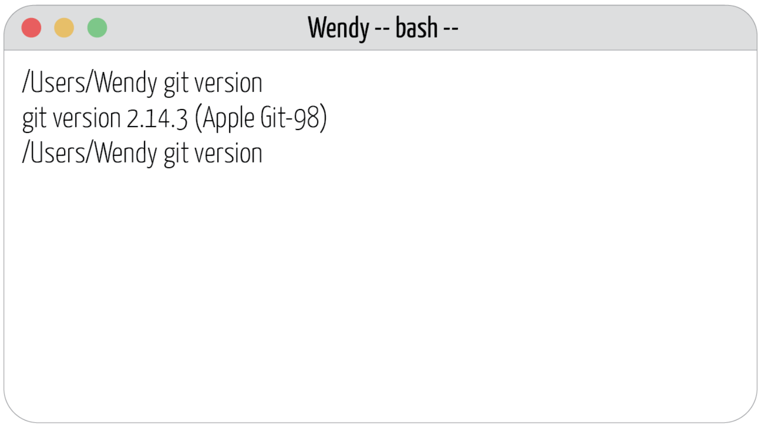
Most recent computers already come with the Git package installed. Open your computer **Terminal** and check if you have a working version of Git installed. Depending on your Operative System the **Terminal or Command Prompt** can be located as follow:

Mac: Open the Finder, navigate to **Applications/Utilities** and click on the **Terminal** application.

Windows: Click the **Start** button, navigate to **All Programs/Accessories** and click on the **Command Prompt** application.

Linux: Press Ctrl+Alt+T to open the **Terminal**

Once opened, type **git version** and press enter on your keyboard. If you have Git successfully installed on your machine, you will get the following output stating the current version of Git installed:



```
/Users/Wendy git version
git version 2.14.3 (Apple Git-98)
/Users/Wendy git version
```

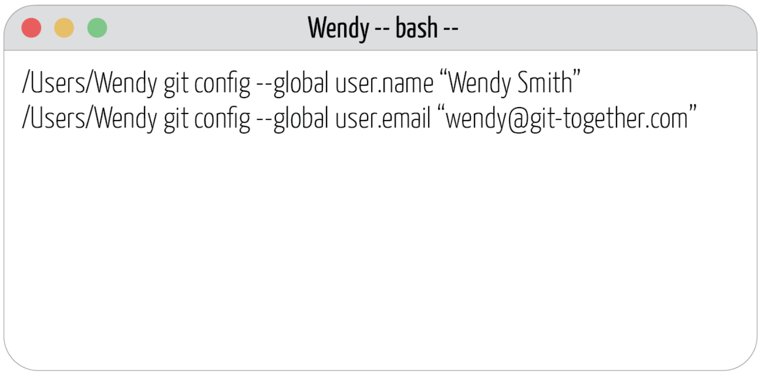
If instead you get a strange error message or "Command not found", you need to first install the Git package. You can install Git by downloading the installation package from the official page:
https://git-scm.com/downloads

Follow the installation steps and you will be all set with Git. You can always double check your installation by heading to the Terminal and type **git version**.

It is now time to configure your user name and email address. They are just records to identify your work and give you credits when making changes to your project.

Run the following two commands to configure your user name and email:

- **git config --global user.name "your full name"**
- **git config --global user.email "your email"**



```
/Users/Wendy git config --global user.name "Wendy Smith"
/Users/Wendy git config --global user.email "wendy@git-together.com"
```

Local vs Collaborative Git

A great feature of Git is that is locally enabled. This means that it does not need any complicated server setup to run like other version-control systems do. You can in fact version-control files on your desktop by simply typing commands on your **Terminal**. On the other hand, in the event that you need a server-side component to centralise your work and collaborate with your colleagues, Git allows you to do so as well.

Local Git

Let's see a really simple example of using Git locally. There are few basic commands that you have to learn to keep track of your project or files:

git init :
This command initialises the Git tracking inside a folder.

git status:
This command checks the status of the tracking. Were there any changes inside the folder?

git add :
This command adds the content of the folder to a holding zone ready to be committed. See it as a way of checking your folder content before the final save.

git reset :
This command removes the content ready to be committed from the holding zone. If you change your mind before finalising your save, you can use **git reset** to remove files from your commit.

git commit -m"project version recorded":
This command permanently records the status of the project at a specific point in time. The message in quotes can be arbitrary, it normally describes the changes that were made.

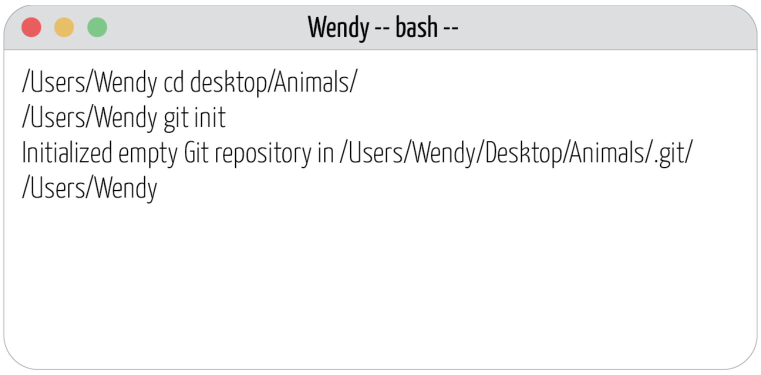
git log:
This command shows a list of all the previous commits in case you want to retrieve or check an older version of the project.

git checkout:
This command lets you navigate and eventually restore older versions of your project.

Let's pretend you want to keep track of a project folder on your desktop called **Animals** which contains different text files of animals' description.

The very first step is to set up Git with the **Animals** folder to initialise the files tracking.
Once you have created the **Animals** folder on your desktop, open the Terminal and follow the instructions below:

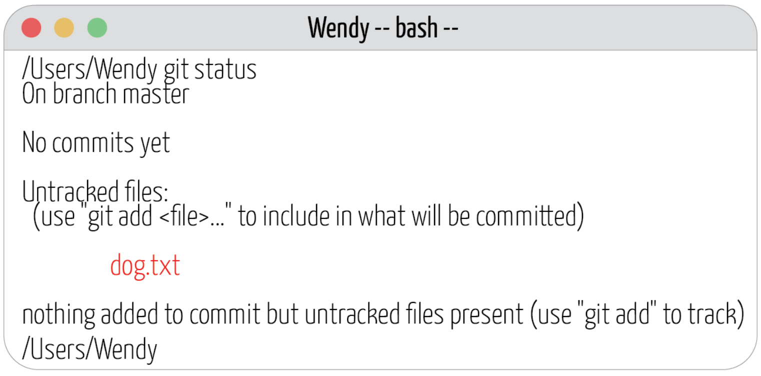
- First you need to cd inside the **Animals** folder: type **cd <your/path/to/Animals>** in your Terminal
- Next you need to initialise the Git tracking: type **git init** in your Terminal



```
/Users/Wendy cd desktop/Animals/
/Users/Wendy git init
Initialized empty Git repository in /Users/Wendy/Desktop/Animals/.git/
/Users/Wendy
```

At this point the **Animals** folder is ready to be used with Git.

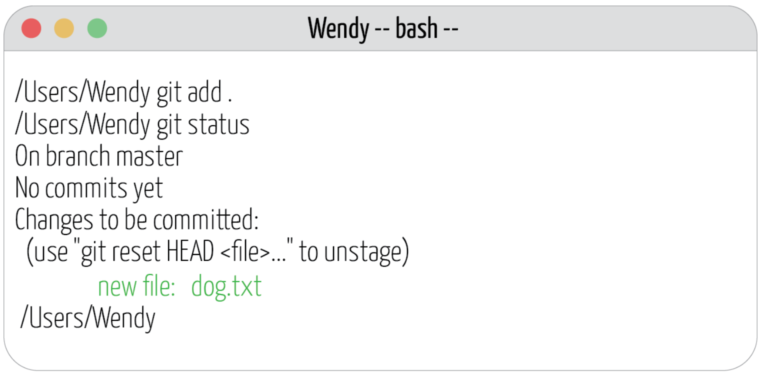
Now create a file inside the **Animals** folder called **dog.txt**.
If you run the **git status** command now, git will highlight in red that there is a new file called **dog.txt**:



```
/Users/Wendy git status
On branch master

No commits yet

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

        dog.txt

nothing added to commit but untracked files present (use "git add" to track)
/Users/Wendy
```

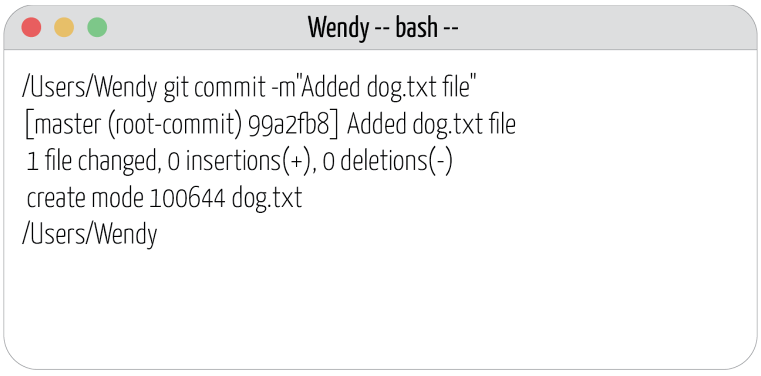
The next step is to add the **dog.txt** file to the holding zone for the final commit.
Run the **git add .** command and git will record the **dog.txt** file as ready to be committed. If the file was successfully added, running the **git status** command should list the file as green:

Note: **git add .** (the dot means add all the files to the holding zone, if you only want to add specific files you can run **git add <filename>**)



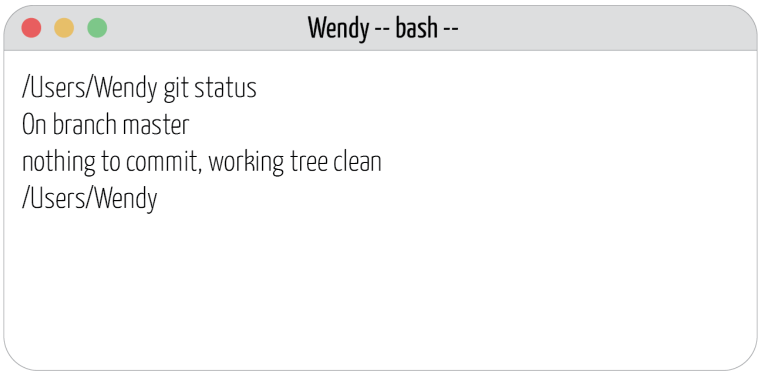
```
/Users/Wendy git add .
/Users/Wendy git status
On branch master
No commits yet
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file:   dog.txt
/Users/Wendy
```

Finally if you are happy with the changes you can run the **git commit -m"commit message"** command to finalise the changes.



```
/Users/Wendy git commit -m"Added dog.txt file"
[master (root-commit) 99a2fb8] Added dog.txt file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 dog.txt
/Users/Wendy
```

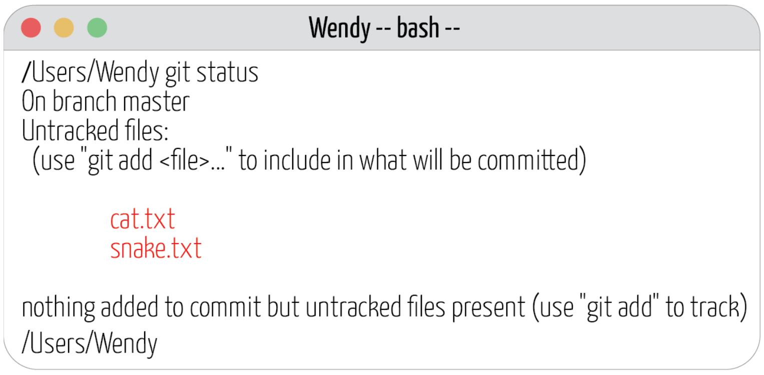
Checking the status of your project now with the **git status** command won't highlight any changes as you have nothing to commit and you are up to date with the project.



```
Wendy -- bash --
/Users/Wendy git status
On branch master
nothing to commit, working tree clean
/Users/Wendy
```

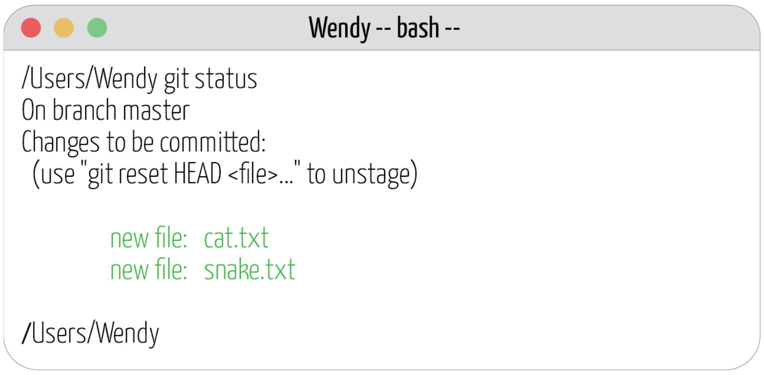
Now let's say you want to add two more animal text files to your **Animals** folder. Create a **cat.txt** file and a **snake.txt** file inside the **Animals** folder.

Running the **git status** command now will highlights that there are two new files in the **Animals** folder:



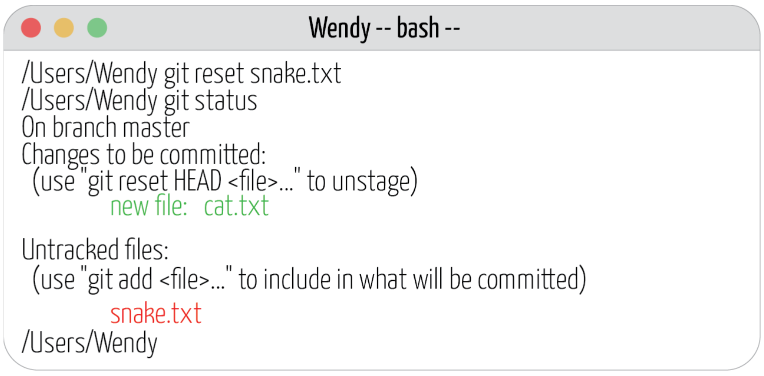
```
Wendy -- bash --
/Users/Wendy git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        cat.txt
        snake.txt

nothing added to commit but untracked files present (use "git add" to track)
/Users/Wendy
```

As usual we want to add our files to the holding zone so that they are ready to be committed; run the **git add .** command and successively the **git status** command:



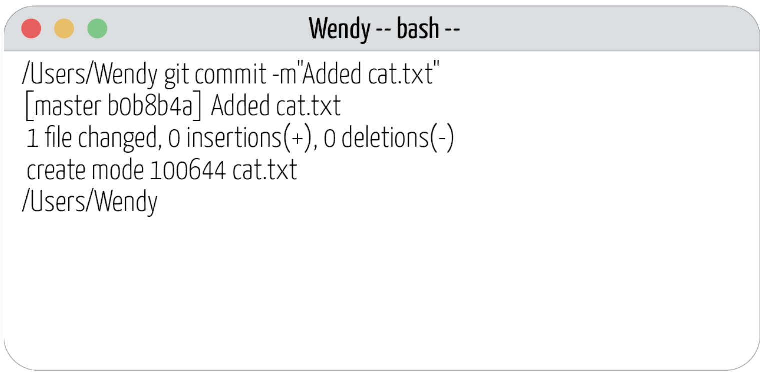
```
Wendy -- bash --
/Users/Wendy git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   cat.txt
        new file:   snake.txt

/Users/Wendy
```

Now you change your mind and you are not ready to add the **snake.txt** file to your list of animals.
Run the command **git reset snake.txt** to remove the file from the holding zone.
Running the command **git status** now will only highlight the **cat.txt** file in green:



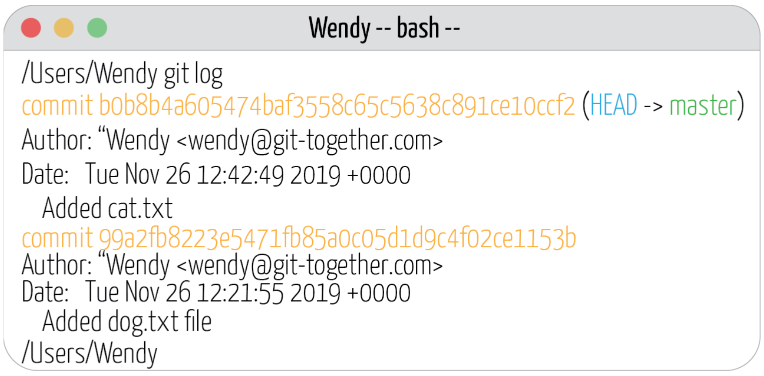
```
/Users/Wendy git reset snake.txt
/Users/Wendy git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
          new file:   cat.txt

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

At this point running the command **git commit -m"commit message"** will only commit the **cat.txt** file to the committed project version. Remember that the commit message is arbitrary and describes your latest change.



```
/Users/Wendy git commit -m"Added cat.txt"
[master b0b8b4a] Added cat.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cat.txt
/Users/Wendy
```

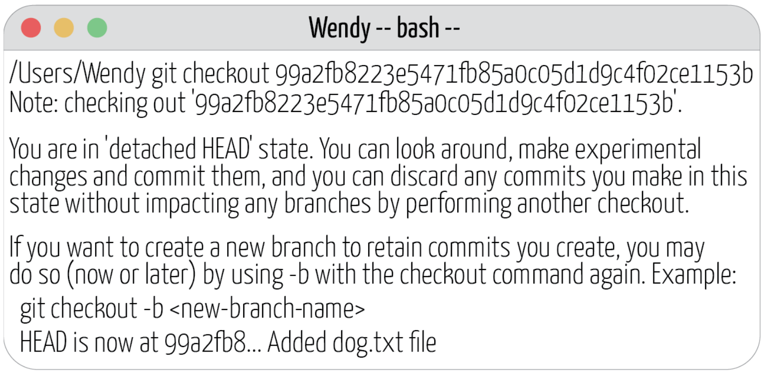
Finally if you want to check previous commits you can run the **git log** command. This will show a list of all the commits with a long string **id**, the **author** and the **message** of the commit.



```
Wendy -- bash --
/Users/Wendy git log
commit b0b8b4a605474baf3558c65c5638c891ce10ccf2 (HEAD -> master)
Author: "Wendy <wendy@git-together.com>
Date:   Tue Nov 26 12:42:49 2019 +0000
   Added cat.txt
commit 99a2fb8223e5471fb85a0c05d1d9c4f02ce1153b
Author: "Wendy <wendy@git-together.com>
Date:   Tue Nov 26 12:21:55 2019 +0000
   Added dog.txt file
/Users/Wendy
```

As you can see, at the moment we have two commits for our **Animals** project. If you want to view a previous version of the project, you can use the **git checkout** command followed by the commit **id** that you want to review. Let's say we want to go back to where we only had the **dog.txt** file inside the folder:

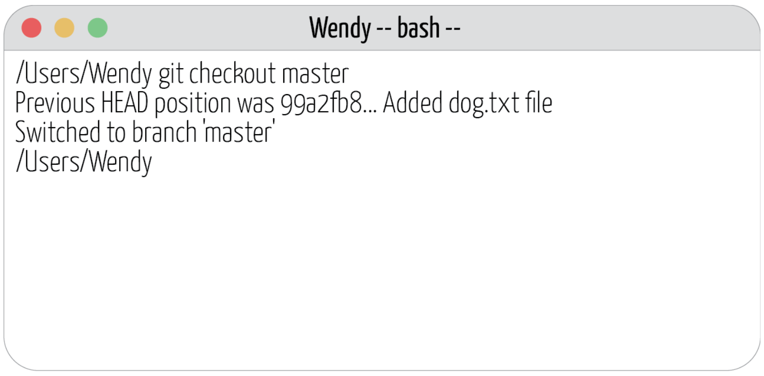
run **git checkout 99a2fb8223e5471fb85a0c05d1d9c4f02ce1153b**
(your commit id will be different)



```
Wendy -- bash --
/Users/Wendy git checkout 99a2fb8223e5471fb85a0c05d1d9c4f02ce1153b
Note: checking out '99a2fb8223e5471fb85a0c05d1d9c4f02ce1153b'.

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 performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:
 git checkout -b <new-branch-name>
 HEAD is now at 99a2fb8... Added dog.txt file
```

If you now look inside the **Animals** folder, the **cat.txt** file is no longer there. You can still see the **snake.txt** file because it was never added to the commit history.

Now let's reset back to the current version of the project. In order to do so simply run the **git checkout** command followed by the word master.



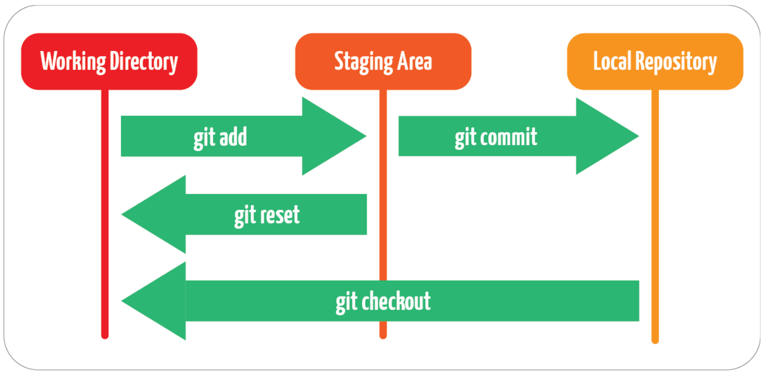
```
Wendy -- bash --
/Users/Wendy git checkout master
Previous HEAD position was 99a2fb8... Added dog.txt file
Switched to branch 'master'
/Users/Wendy
```

As you can see the **Animals** folder is now reset to the latest version of the project and it has the file **dog.txt** file again.

You can now practice the git commands shown so far. You can for example add more animals to the **Animals** folder or add some content to the existing animal files. Run the git commands and see if you can keep track of your changes.

Local Git: What happens visually?

Ok, we have seen how to use git locally to keep track of you files but what does exactly happens visually? What is the holding zone? How did the git pipeline actually look if we could draw it?



Working Directory:
The working directory corresponds to the folder which has a git tracking enabled. In the previous example the folder **Animals** was our project working directory.

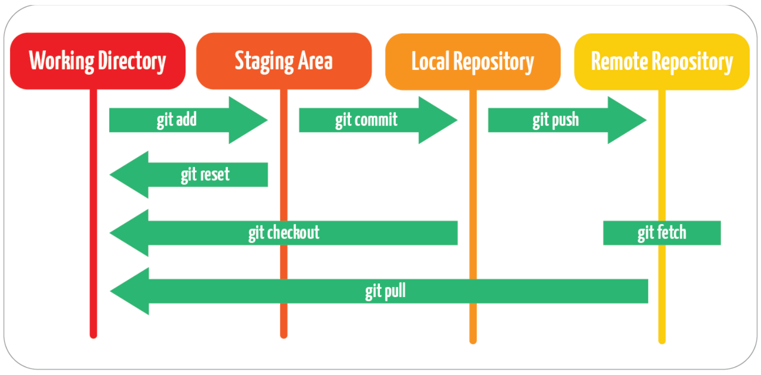
Staging Area:
The staging area corresponds to the holding zone. When running the **git add** command, the files are added to this intermediate zone. All files standing in this zone are ready to be committed.

Local Repository:
The local repository is the place where all of your committed project versions are stored. Running the **git log** command will show the content of this repository, showing each project version with corresponding id, name and description.

Collaborative Git

Great, now we know how to keep track of a project locally on our machine and understand the underlying Git pipeline. It is time to see how Git can be used in collaborative projects. What if we want to have a centralise home based project folder where multiple people can contribute to it at the same time? Luckily for us, Git offers a great way to accomplish this and the steps are not so different from what we have learnt so far. In fact, we only need a shared place where we can store our current project version and learn few other Git commands to be able to interact with it.

Collaborative Git: What happens visually?



As you can see, the collaborative Git visual pipeline is not so different from the local Git pipeline and it only adds to the latter structure.

Remote Repository:
The remote repository is a copy of your project that is hosted on the Internet. Going back to our **Animals** folder, it is like having a copy of the folder hosted on the Internet which is accessible by other people. The project owner, of course, sets different privileges to who can access the project and modify it.

You can also see that there are three new Git commands when it comes to using Git in a collaborative way:

git push:
This command permanently records the status of the hosted project at a specific point in time. It takes your latest local committed project version and adds the changes to the hosted version of the project.

git pull:
This command retrieves the latest version of the hosted project to your working directory and, if required, it merges the differences between your local version of the project and the hosted one.

git fetch:
This command is similar to git pull but instead of retreating the latest version of the hosted project, it will only synchronise your local version of the project with the hosted one.

Where can we host the project?

There are different online services that allows you to host your project and make it accessible by multiple people. The leading hosting platforms for software development which are compatible with Git are **GitHub**, **GitLab** and **Bitbucket**:

GitHub: https://github.com/

GitLab: https://about.gitlab.com/

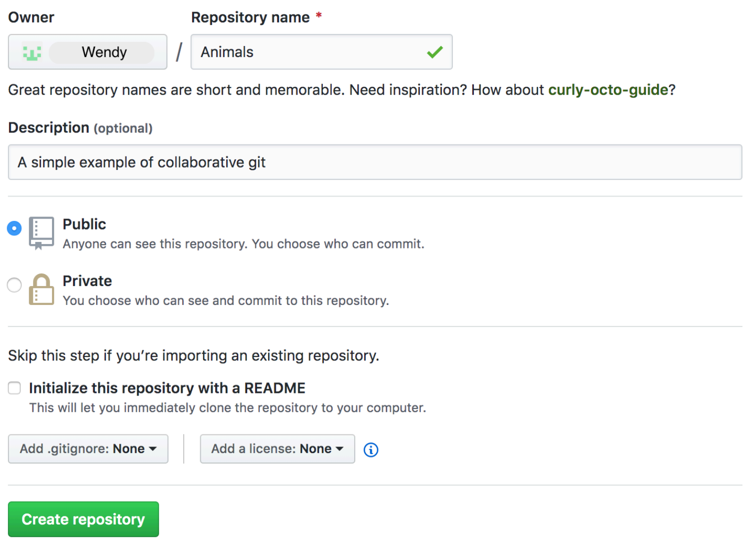
Bitbucket: https://bitbucket.org/

All of the above share the same principles when it comes to using Git but for the purpose of the following example we will use **GitHub**.

Let's say you would like to start the **Animals** project from scratch and share it with a friend so that you can both work on the project at the same time. The very first thing is for you both to create an account on **GitHub**. There are different hosting options to choose from when signing in and for this project you can use the free package version as it satisfies the requirements.

Once you have both created an account, the project leader, in this case you, will create the **Animals** remote repository. Remember: this is the version of the project that lives on the server and therefore accessible by everyone who has access to it.

In order for you to create the repository, you need to be signed in to **GitHub** and head to the **+** sign icon located on the top right of the navigation bar. Click on the **+** sign icon and select the new repository option form the submenu. You will be prompted with the following form:



Owner: Wendy / Repository name *: Animals

Great repository names are short and memorable. Need inspiration? How about curly-octo-guide?

Description (optional): A simple example of collaborative git

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

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

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

☐ Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Add .gitignore: None | Add a license: None

Create repository

Owner: the owner Github user id

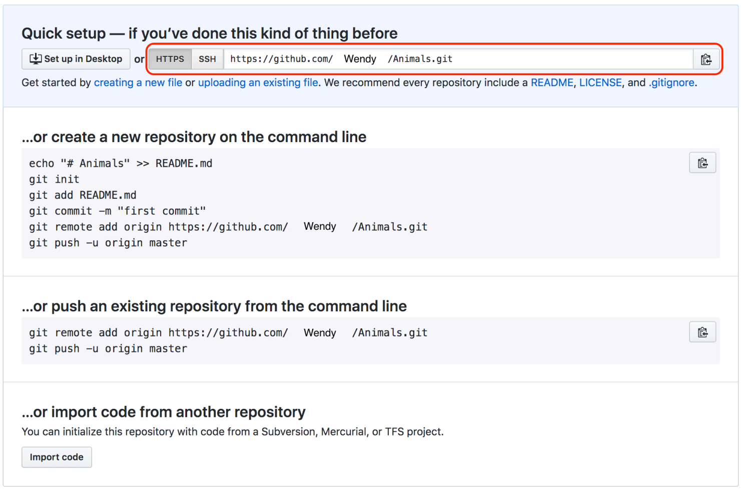
Repository name: the name of the project (Animals in this case)

Description: the project's description

Public/Private: sets whether the repository can be seen publicly or not

Initialise the repository with a README: adds a README file template to the project

Once you have filled in the form, click the **Create repository** button to finalise the process. Hurrah, you can now see that your **Animals** remote repository has been created together with a https link to access it:



Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH `https://github.com/ Wendy /Animals.git`

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# Animals" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/  Wendy  /Animals.git
git push -u origin master
```

...or push an existing repository from the command line

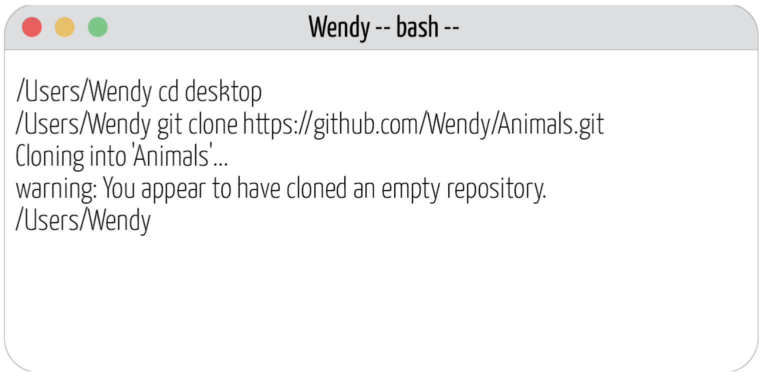
```
git remote add origin https://github.com/  Wendy  /Animals.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

The very next thing is for you and your friend to have a copy of the hosted **Animals** repository locally on your machines so that you can start collaborating on the project. Open your terminal, use the **cd** command and navigate to the path (desktop in this case) where you would like to create a local copy of the **Animals** project. Now run the git clone command followed by the https link provided on **GitHub** (highlighted in red on the above picture):



```
/Users/Wendy cd desktop
/Users/Wendy git clone https://github.com/Wendy/Animals.git
Cloning into 'Animals'...
warning: You appear to have cloned an empty repository.
/Users/Wendy
```

At this point the **Animals** folder should be visible on your desktop. The **git clone** command initialises the project working directory. It copies the version of the project hosted on the server and runs the command **git init** automatically to start the git tracking. Your friend, will have to follow the same steps and also **clone** a copy of the **Animals** folder on their desktop.

Collaborative Git: The Animals Project

The pipeline to follow in order to collaborate with the **Animals** project is exactly the same as the local git pipeline with the exception that you now need to also update the project version on the server using the newly-introduced git commands **fetch**, **push** and **pull**.

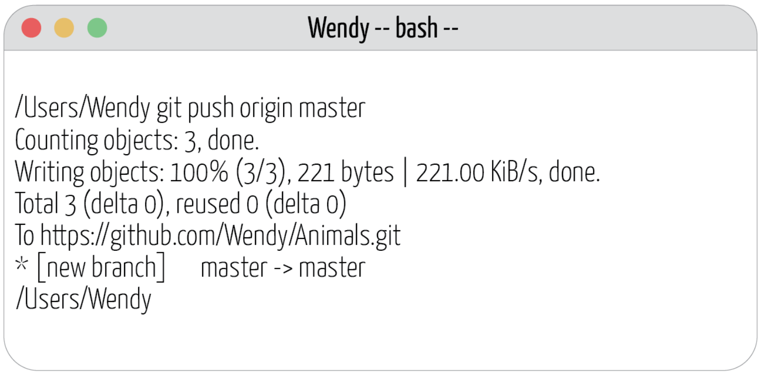
Create a **dog.txt** file inside the **Animals** folder on your desktop.

Open the terminal and run the following commands:

- **git add .** (Adds the dog.txt file to the holding zone)
- **git commit -m"Added dog file"** (Saves a version of the project on the local repository)

  Now if you want to also update the version on the server you need to push your local repository changes to the hosted repository:

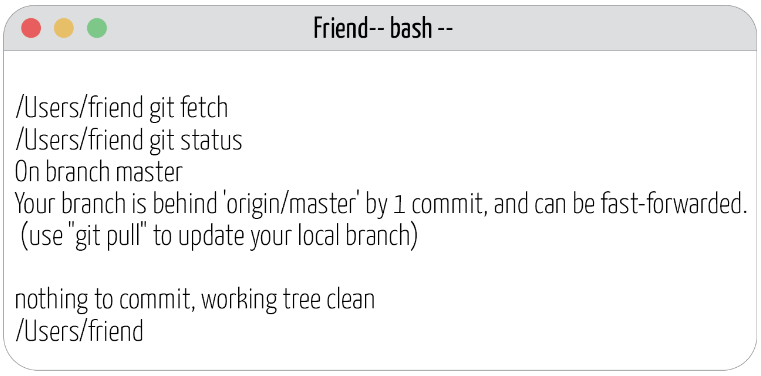
- Run the command **git push origin master** or simply **git push**



```
/Users/Wendy git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 221 bytes | 221.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/Wendy/Animals.git
* [new branch]      master -> master
/Users/Wendy
```

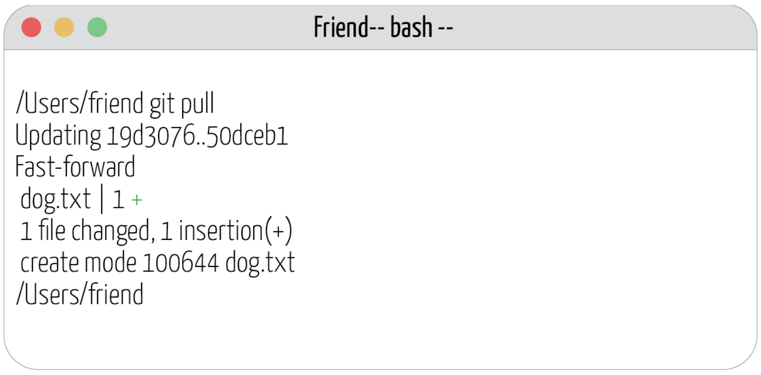
The project changes have now been added to the hosted version.
If you check the project on **GitHub** (refresh the webpage) you will see that it now contains the **dog.txt** file.

Now, in order for your friend to see the new changes, they need to synchronise their working directory with the hosted one. They should first run the commands **git fetch** and **git status** to see if there are any changes:



```
/Users/friend git fetch
/Users/friend git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
 (use "git pull" to update your local branch)

nothing to commit, working tree clean
/Users/friend
```

You can see that your friend working directory is behind by 1 commit compared to the remote directory on **GitLab**. They should now run the command **git pull** to retrieve and eventually merge the latest changes.



```
/Users/friend git pull
Updating 19d3076..50dceb1
Fast-forward
 dog.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 dog.txt
/Users/friend
```

Once the command **git pull** has been run you will both have the **Animals** folder on your desktop containing the **dog.txt** file. You will also be up to date with the **Animals** folder hosted on GitHub.

Now try and collaborate with your friend, add more files to the **Animals** folder or update existing ones, **add**, **commit** and **push** the changes to the shared repository.

There are of course other important Git features which have not been covered and I highly suggest you take a look into them once you are comfortable with the basics:

- How to resolve possible merge conflicts
- Working with Git branches
- Git privileges and Pull requests
- Add local project to already existing hosted repository

All the above listed features will not be necessary to help us with the To-do list project .

Let's go and have some fun.

Best wishes,

Your programmer Wendy