WHAT IS THIS SEMINAR ABOUT

What is Git and why is good

The abstract view

Saving changes

Inspecting the repository

Undoing changes

Introduction to GitHub

Push and Pull

WHAT IS THIS SEMINAR NOT ABOUT

How to setup Git on your computer

Git/GitHub Clients with GUI

Branching

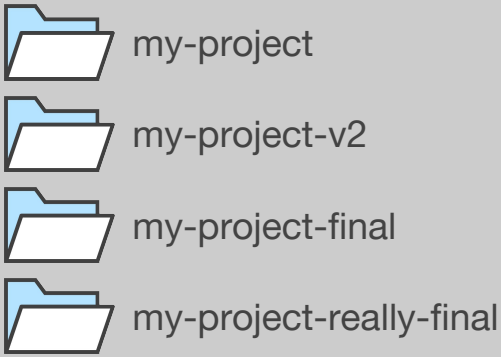
Merging and rebasing

Git team workflows

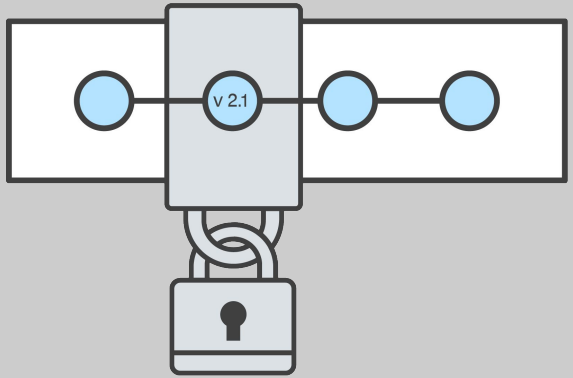
Pull requests and forks

PART 1
The Local Picture

How do you organise your code?



A typical problem for programmers is managing multiple versions of a project.



Version Control System (VCS)

A VCS allows to seamlessly manage multiple code versions. It is then possible to move between different versions, selectively save changes and reset to a specific version. At the same time it offers support to work easily with other programmers.

Git is a communication tool

Besides being a technical tool, a VCS (Git in our case) is a great communication tool. It helps people in a team to agree on a common workflow and provides a unified platform to discuss changes and fixes.

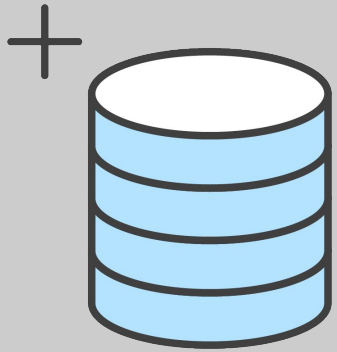
Let’s get it clear...

is the version control system

is a hosting service

Git is a program that runs in your computer and it is initialised on top of a project folder. GitHub is a hosting service provided by a company.

Let’s get started



```
$ git init
```

To initialise a folder as a Git-tracked project move into that folder (`cd /path/to/the folder/`) and run `git init`.

The abstract view

https://xkcd.com/1597/

What do YOU see when looking at a folder?

A tree of files and folders

What does GIT see when looking at a folder?

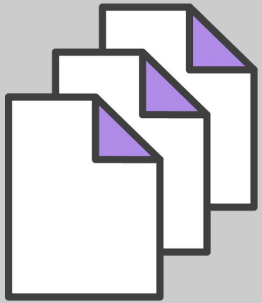
Three imaginary areas

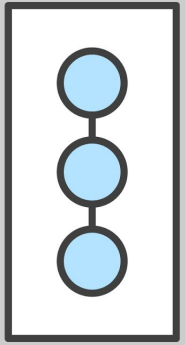
(actually four)

Three areas

Working folder from the point of view of Git: shows which files have been modified (created/edited/deleted) since the last version recorded by Git.

A list of the commits: represents all the changes that have been recorded by Git.





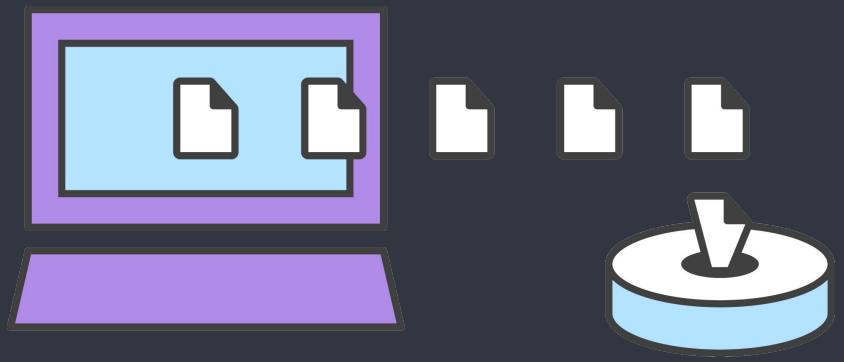
working directory

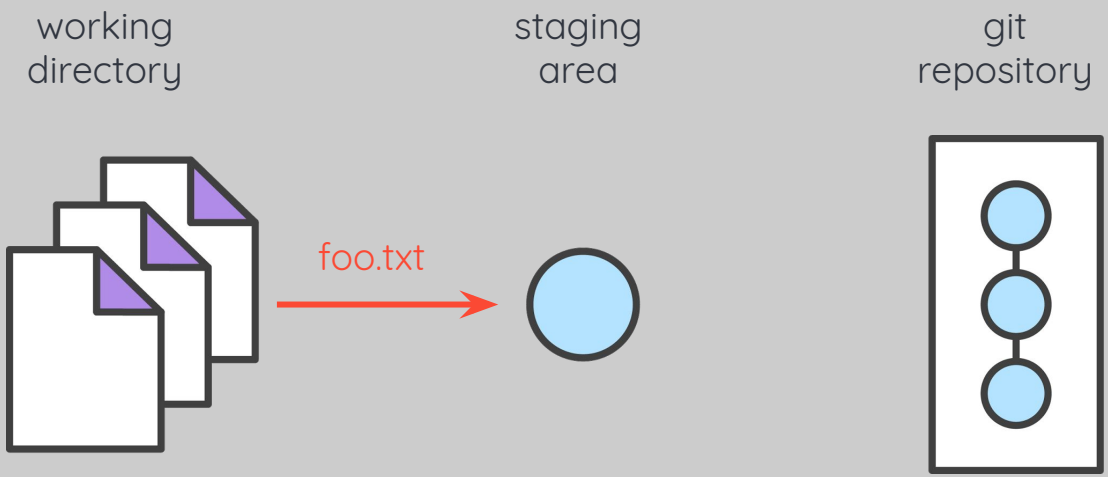
staging area

history or repository

“Middle ground” to select which files that changed should be part of the next commit.

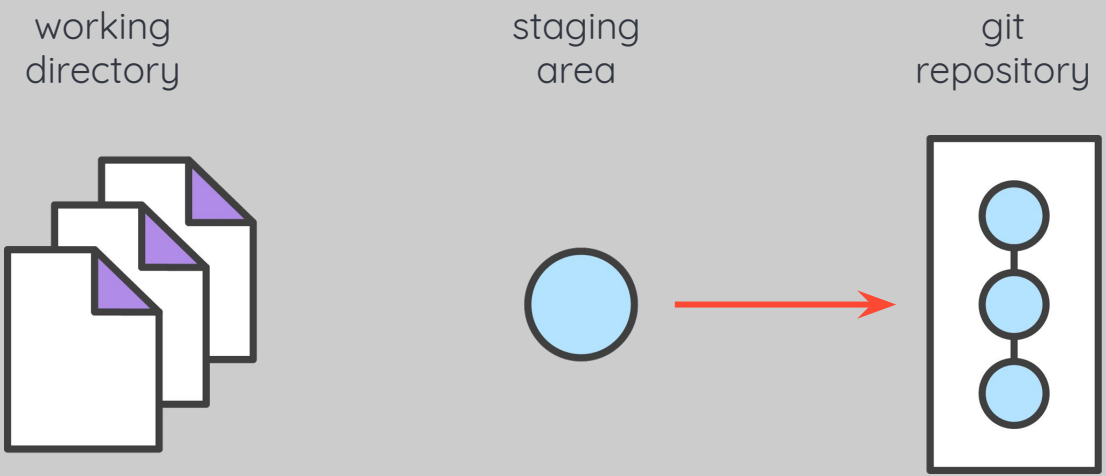
Saving changes






```
$ git add foo.txt
```

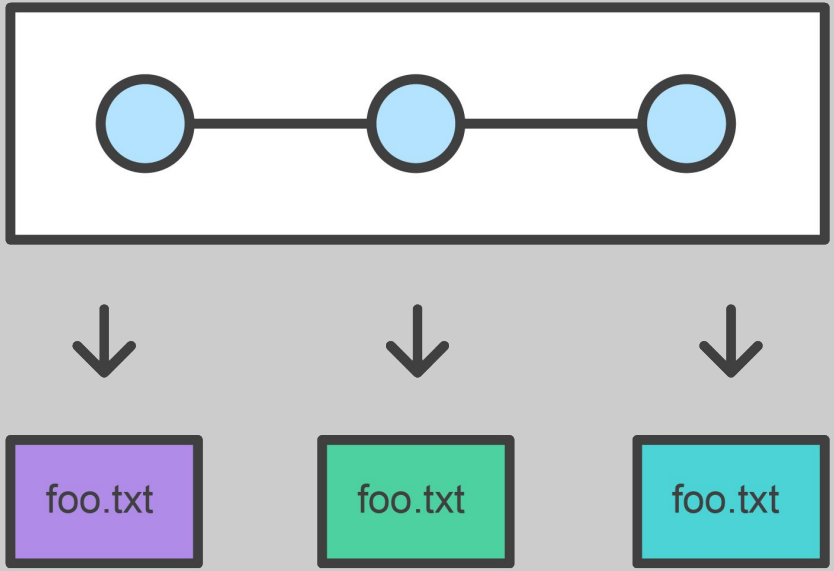
To move a (changed) file from the *working directory* to the *staging area* run `git add <name-of-the-file>`. To add all the (changed) files to the *staging area* run `git add -A`



$ git commit

To make a commit from the staged files run **`git commit`**. This will create a new commit object in the *repository*: the commit contains the set of changes that we have decided to record.
Use **`git commit -m “<write-here-the-commit-message>”`** to avoid opening an editor.

It’s all about snapshots



Git records the entire contents of each file in every commit.
Many other systems record differences between file versions.

Inspecting the repository

$ git status

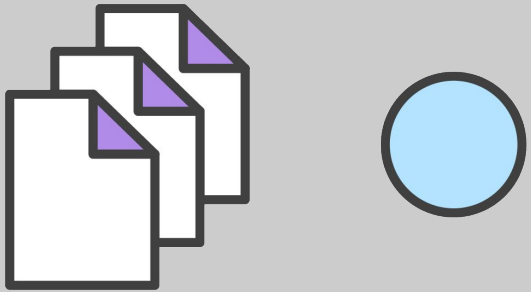
&

$ git log

Two commands are mainly used to inspect the state of the repository:
`git status` and `git log`

Inspect what?

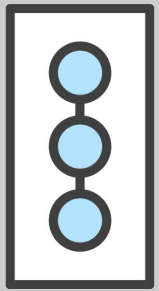
Git Status



Working Directory

Staged Snapshot

Git Log

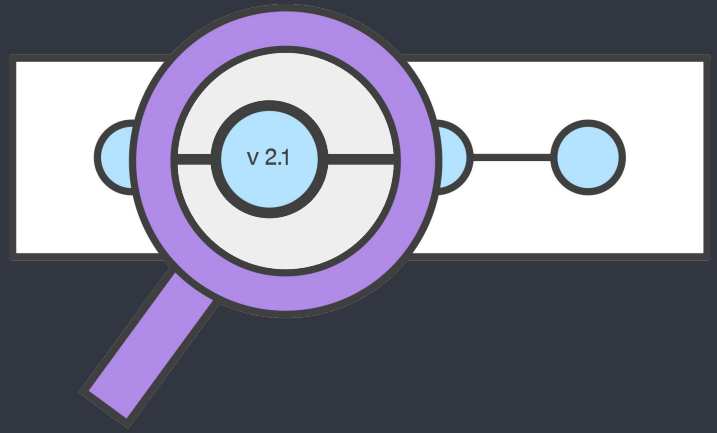


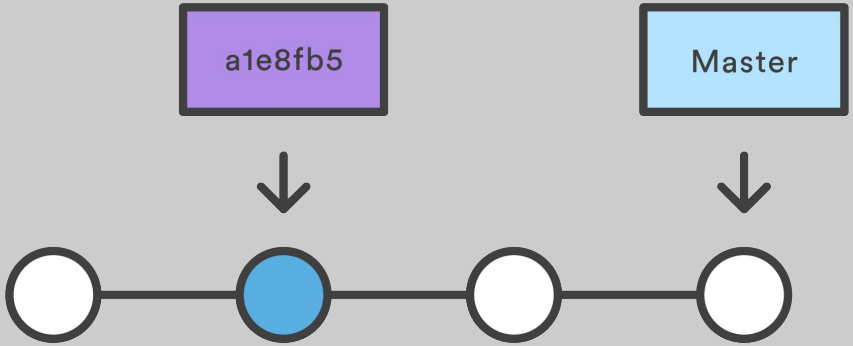
Committed History

The command **`git status`** displays information about the state of the working directory and the staging area.

The command **`git log`** returns a list of commits.

Undoing changes





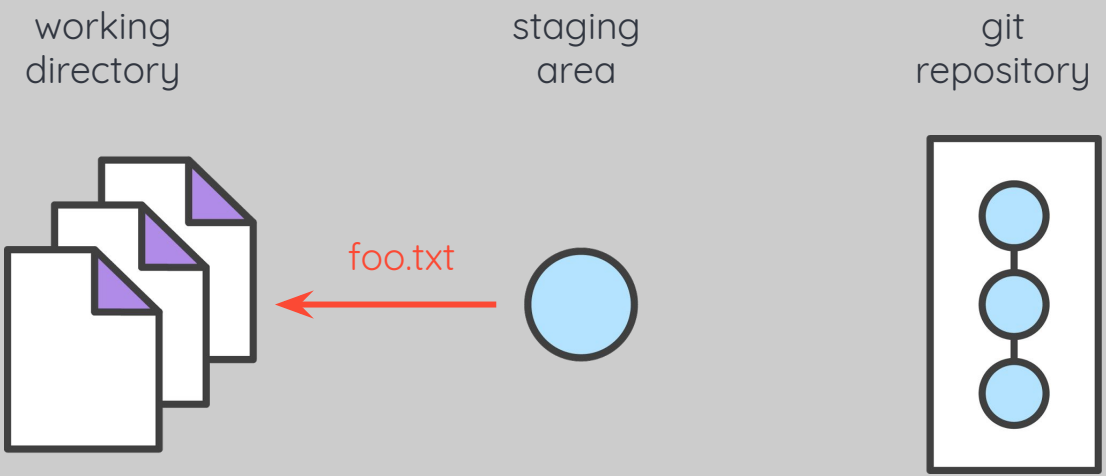

```
$ git checkout a1e8fb5
```

The command `git checkout <commit>` update all files in the working directory to match the specified commit. This will put you in a **detached HEAD** state.

Checkout commit brings to a detached head state:

READ-ONLY

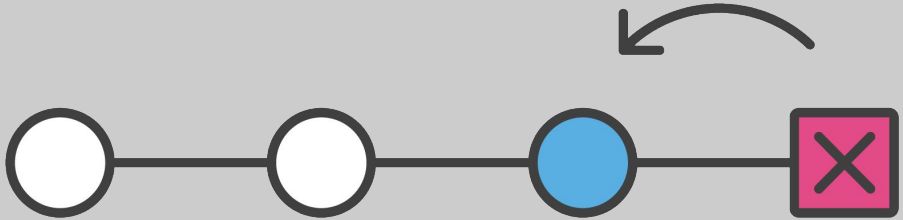
When in detached HEAD state it is only possible to look at the files in the working directory. No changes can be made (don’t change the past).




```
$ git reset foo.txt
```

The command **`git reset <file>`** remove the specified file from the staging area.

RESET ONLY STAGING AREA

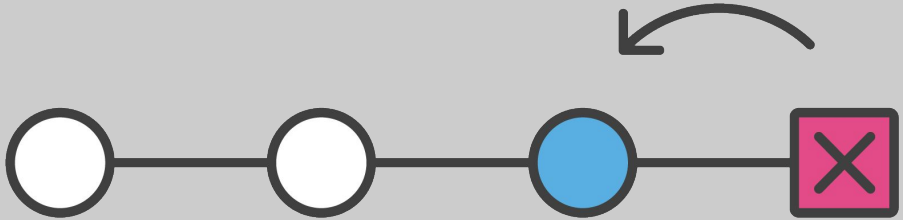


```
$ git reset a1e8fb5
```

The command `git reset <commit>` allows to remove all the committed changes up to the specified commit. The changes are only removed from Git repository and staging area. The “physical” files are still present in the folder.

DANGER:

RESET STAGING AREA & WORKING DIRECTORY



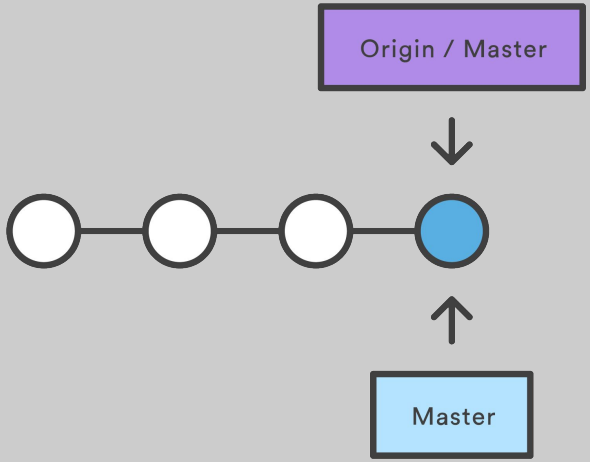
```
$ git reset --hard a1e8fb5
```

The `--hard` option instruct Git that the changes should be removed from the history and staging area and that the files in the folder should be reset to match the specified commit.

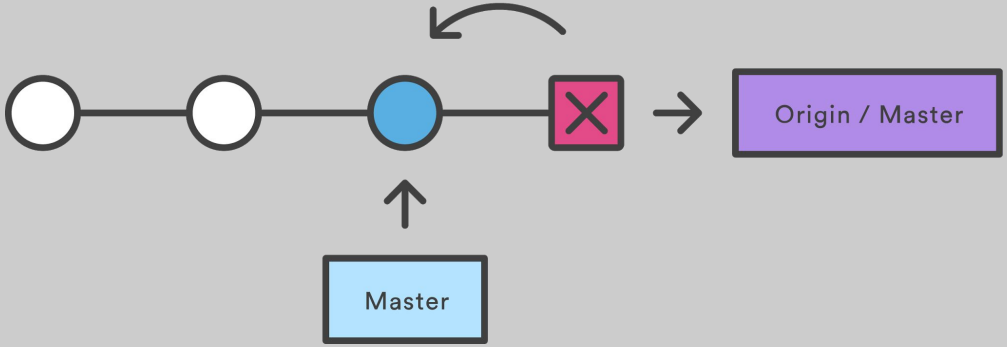
DON’T RESET PUBLIC HISTORY

After the commits have been shared with other developers (pushed to remote) the history shouldn’t be reset. Doing so would generate extensive problems to the collaboration. Other developer may have in fact coded their features on top of commits (pieces of code) that the `reset` command would remove.

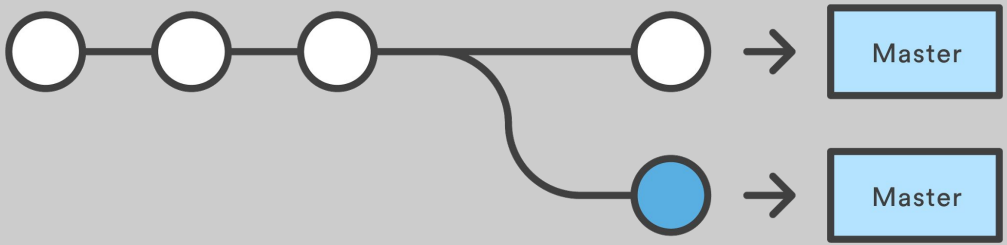
Resetting



After Resetting



After Committing



One more thing

To tell Git to ignore some files use

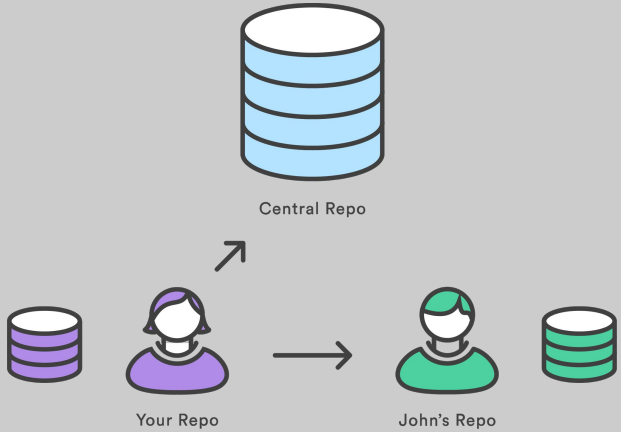
`.gitignore`

A file is ignored by Git if listed inside the standard `.gitignore` file. This feature allows to exclude from version control files that are only temporary and don't represent source code (compiled files such as `.o` or .`beam`).

PART 2
The Social Picture

Local and remote

A Git repository can reside on the local machine or in a remote hosting service like GitHub. It is common practice to have both a local and a remote repository for a single project, especially when working in a team. It is then possible to synchronize the two repositories.

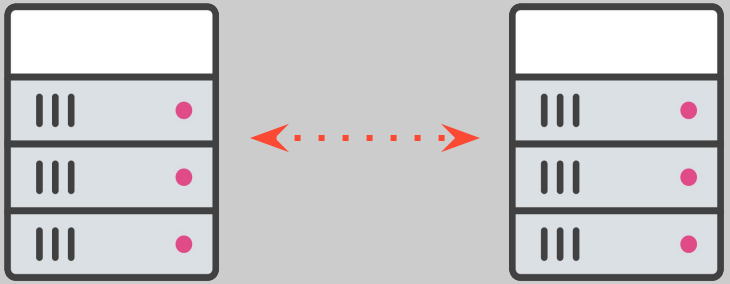


All repositories are born equal...

While it is common to refer to the remote repository as *Central*, Git makes no distinction between the working copy and the central repository. Git's collaboration model is based on repository-to-repository interaction.

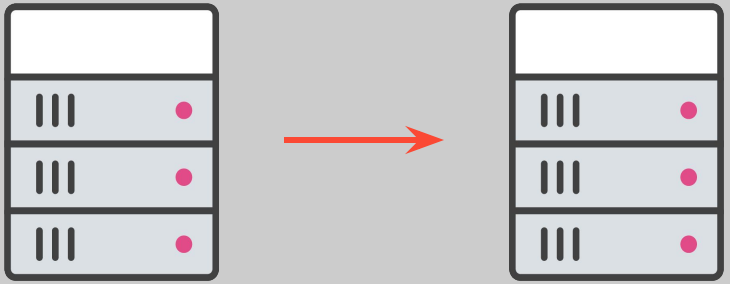
Syncing

In the upcoming slides it is assumed that an empty repository has been created on a hosting service like GitHub and that there is a local repository to work with.



$ git remote add origin <url>

The command `git remote add <remote-name> <url>` adds a remote connection with the specified name and located at the address specified by the url. This establishes a connection between the local and remote repositories. The connection provided by the url can be both over HTTP or with SSH (more secure). The name `origin` is commonly used to refer to the **own remote counterpart** of a local repository.



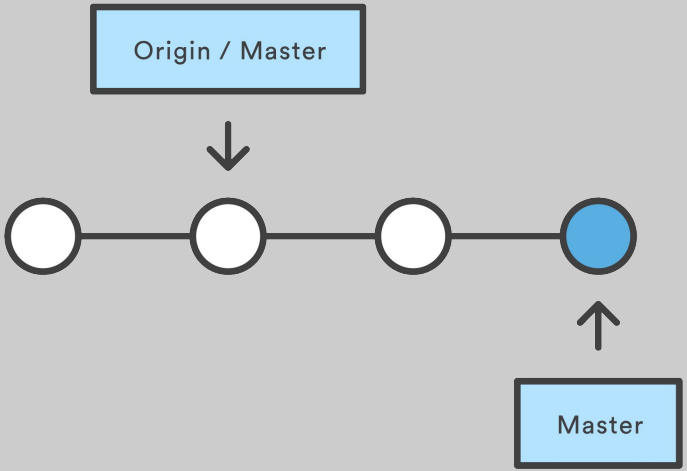
Local system

GitHub server
'Origin'

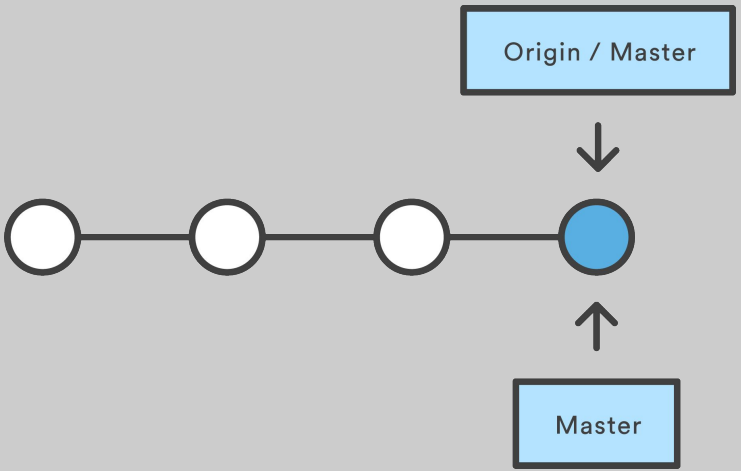
```
$ git push origin master
```

The command **`git push <remote-name> <branch>`** allows to add commits registered in the local repository (on the specified branch) to the remote repository specified by the remote name.

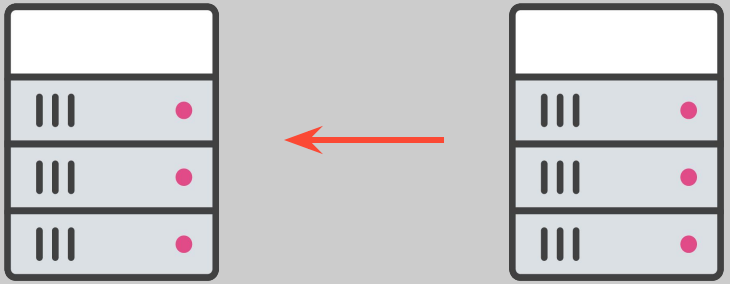
Before Pushing



After Pushing



The local repository (`Master`) is initially ahead of the remote counterpart (`Origin/Master`). After pushing they are perfectly synced.

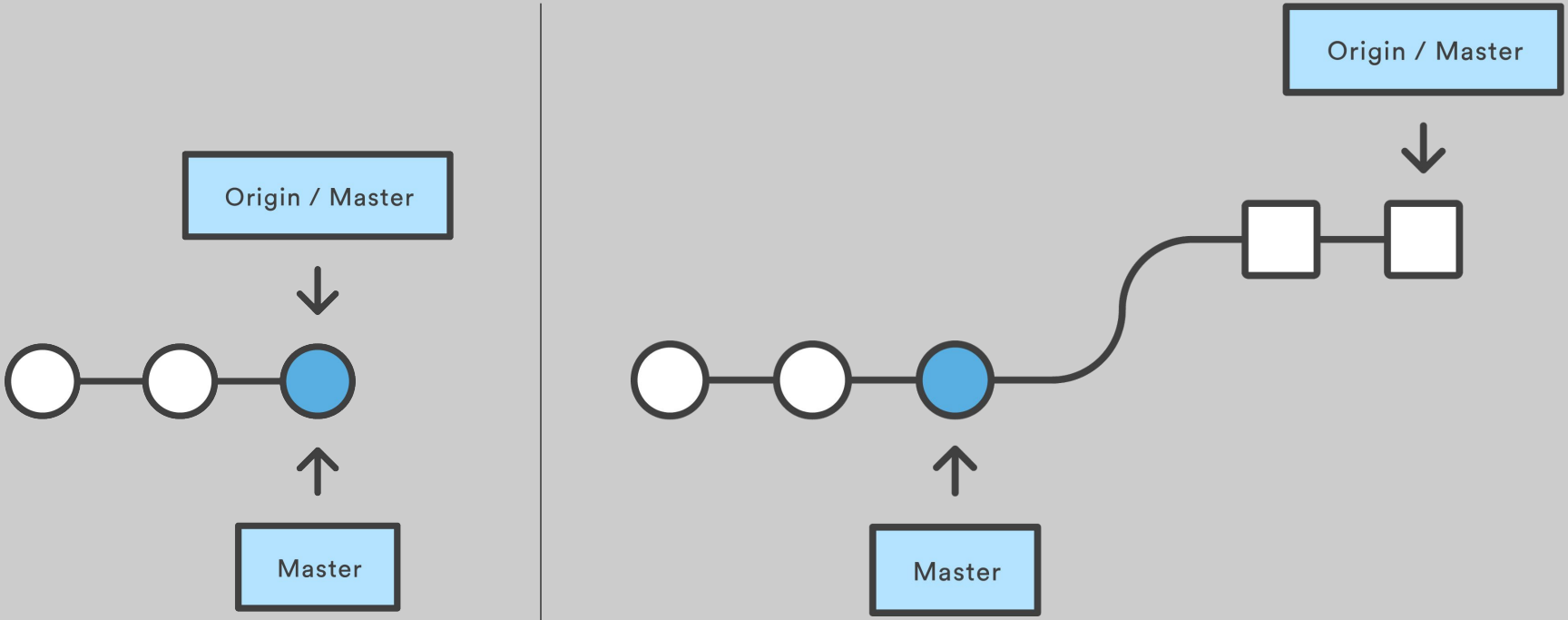


Local system

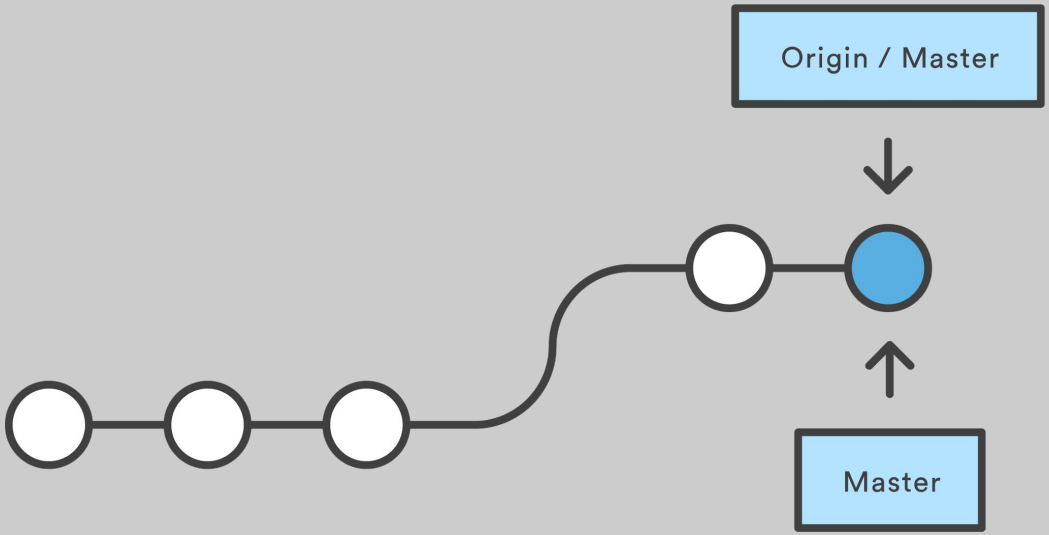
GitHub server
'Origin'

```
$ git pull origin
```

The command `git pull <remote-name>` allows to add commits registered in the remote repository specified by the remote name to the local repository.



The local repository (`Master`) is initially synchronized with the remote counterpart (`Origin/Master`). Then someone pushes commits to the remote repository. These commits are missing in the local repository.



After the pull command the two repositories are synchronized again.

```
$ git clone <url>
```

Sometimes it is needed to create a local copy of a remote repository. The command `git clone <url>` clones the remote repository specified by the url to the current directory. This action automatically sets up a connection between the two repositories naming the remote **origin**.

Credits

All the graphics in this presentation are from Atlassian *Getting Right Git* guide reachable at:

www.atlassian.com/git

The content is licensed under a
Creative Commons Attribution 2.5 Australia License