Git & Github tutorial

Jorge Ramírez
jorge.ramirezmedina@unitn.it

Slides: https://tinyurl.com/se2-git-tutorial

Part #1

Git basics

Getting a Git Repository

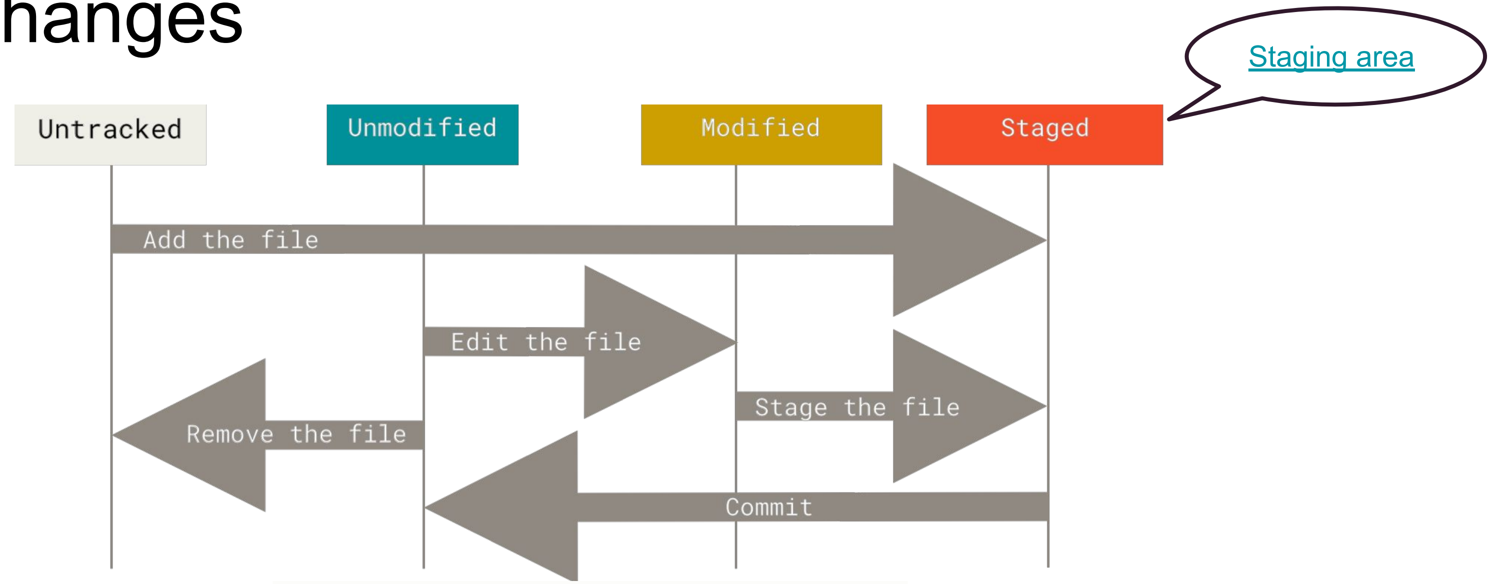
```
$ git init
$ git clone
```

Configuring git

```
$ git config
```

```
$ git config --global user.name "Mario Rossi"
$ git config --global user.email "mr@gmail.com"
```

Saving changes



The file status lifecycle. Source: The git book

```
$ git add
$ git commit
```

Checking status and changes

```
$ git status
$ git diff
```

Ignoring files

```
$ cat .gitignore
*.pyc
```

Removing files

```
$ git rm
```

Viewing commits

```
$ git log
$ git show
```

Tags

Tags can be [lightweight or annotated]

```
$ git tag
```

annotated

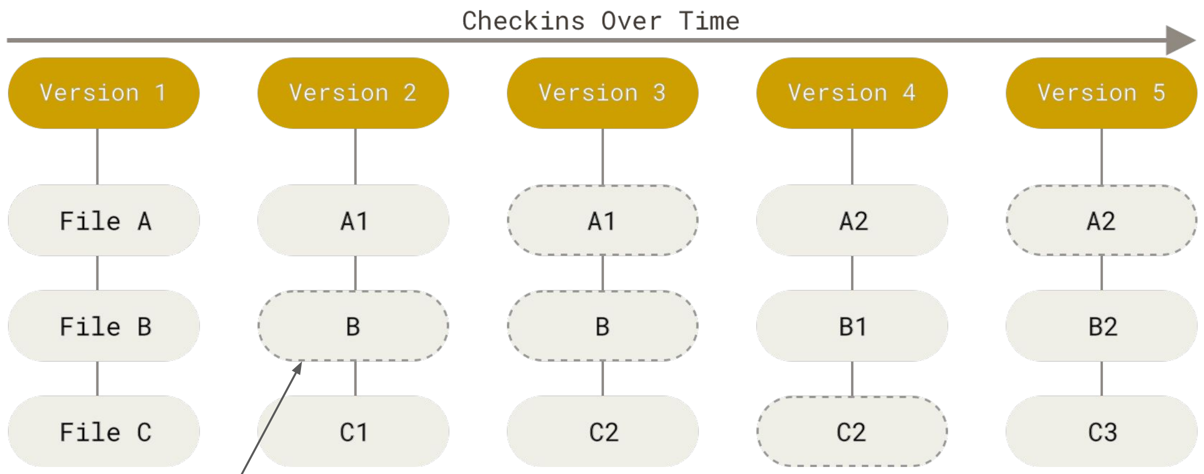
lightweight

```
$ git tag -a v0.1.0 -m "version 0.1.0"
$ git tag v0.1.0
$ git tag -a v0.1.0 6bc0f6b
```

tag a specific commit

Branching

How git works: snapshots

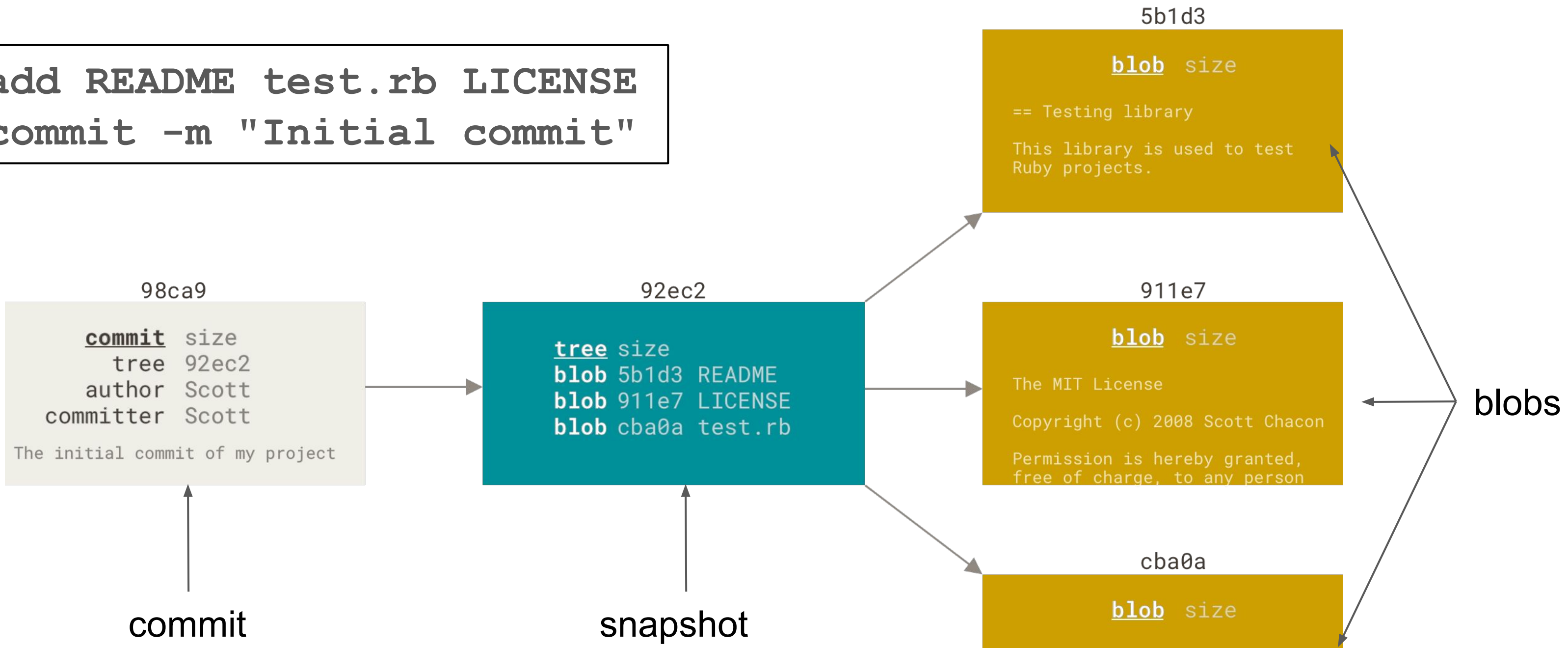


Storing data as snapshots. Source: [The git book]

Pointer to
File B

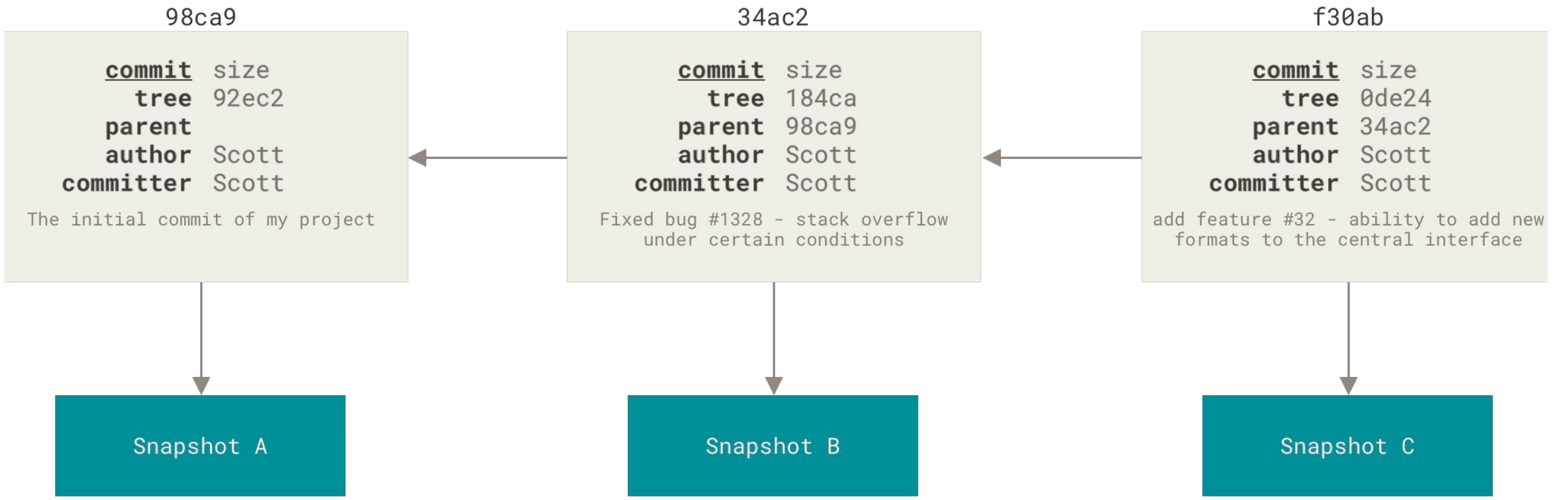
How git works: commit

```
$ git add README test.rb LICENSE
$ git commit -m "Initial commit"
```



Git commit tree. Source: [The git book]

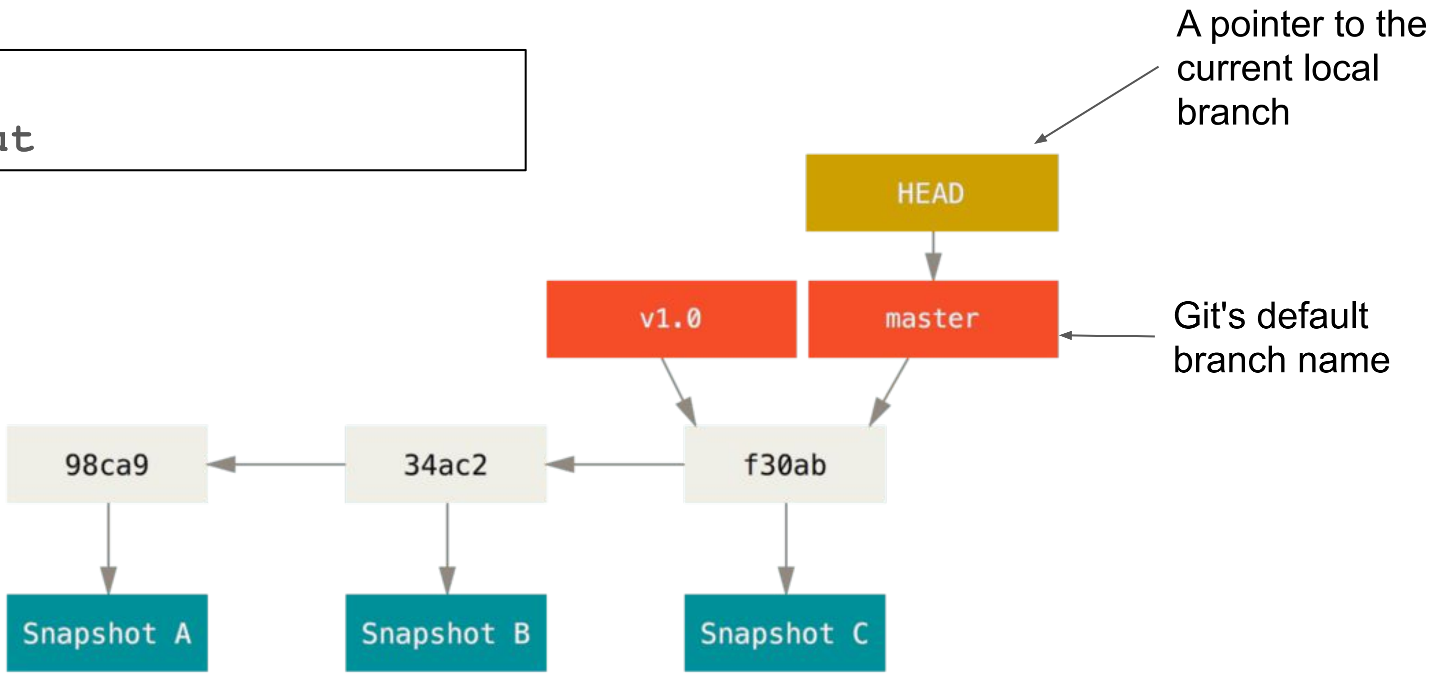
How git works: commit, commit, commit



A sequence of commits. Source: The git book

Branch = a movable pointer to a commit

```
$ git branch
$ git checkout
```


A pointer to the current local branch
HEAD
v1.0
master
Git's default branch name
98ca9
34ac2
f30ab
Snapshot A
Snapshot B
Snapshot C

A git branch and its history. Source: The git book

Working with branches

```
$ git branch f-53
$ git checkout f-53
$ … some work …
$ git commit -m "Implement load balancing".
```

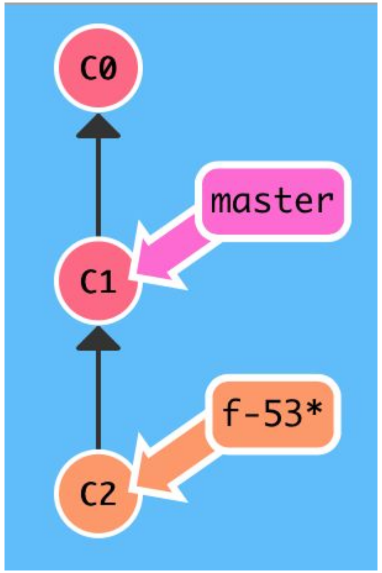
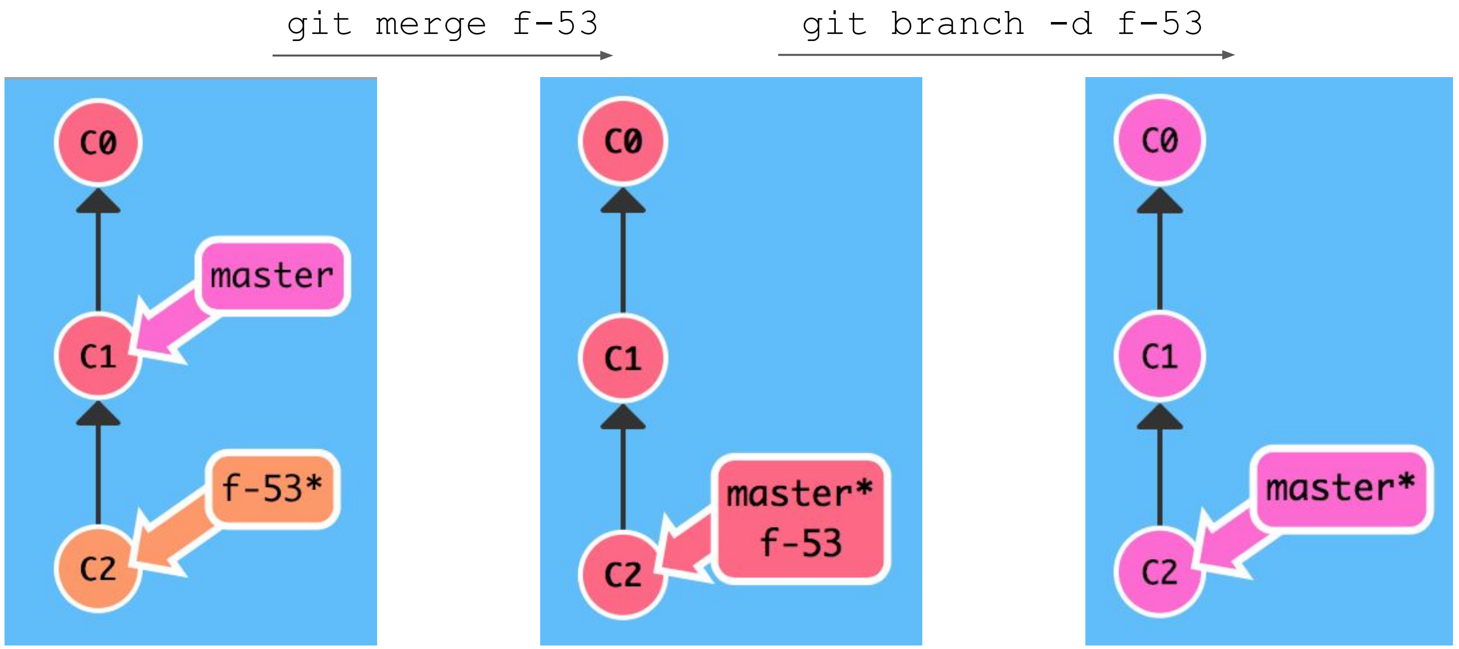


Chart generated using
learngitbranching.js.org

Part #2

Fast-forward merging



```
git merge f-53
```

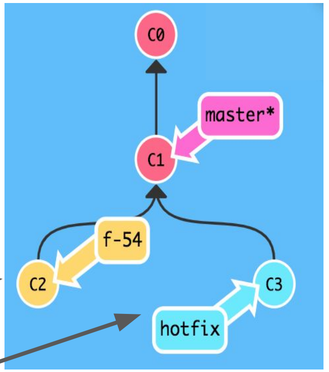
```
git branch -d f-53
```

Chart generated using learngitbranching.js.org

Working with branches: a real life example

```
$ git branch f-54
$ git checkout f-54
$ … some work …
$ … the boss asks you to fix an urgent bug…
$ git commit -m "Current progress on issue 54"
$ git checkout master
$ git checkout -b hotfix
$ … fix the bug …
$ git commit -m "Fix the user form"
$ git checkout master
$ git merge hotfix
$ … you notify the boss …
```

forward merge



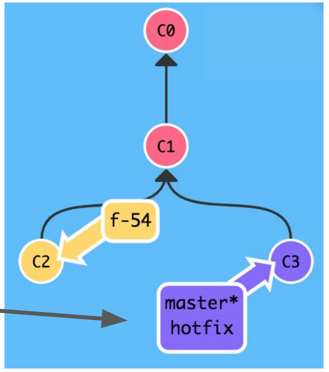
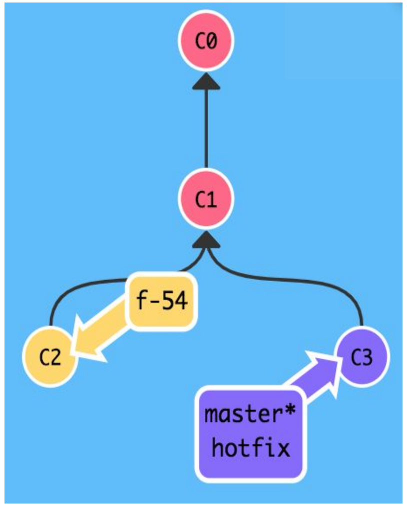


Chart generated using
learngitbranching.js.org

Recursive or three-way merging

```
git merge f-54
```



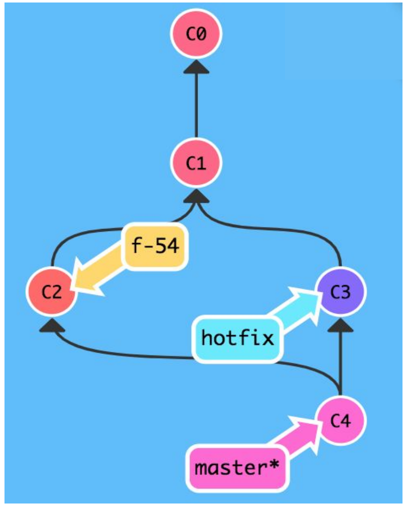


Chart generated using learngitbranching.js.org

Solving merge conflicts

Branches **f-55** and **f-56** changed the same part of a file differently.

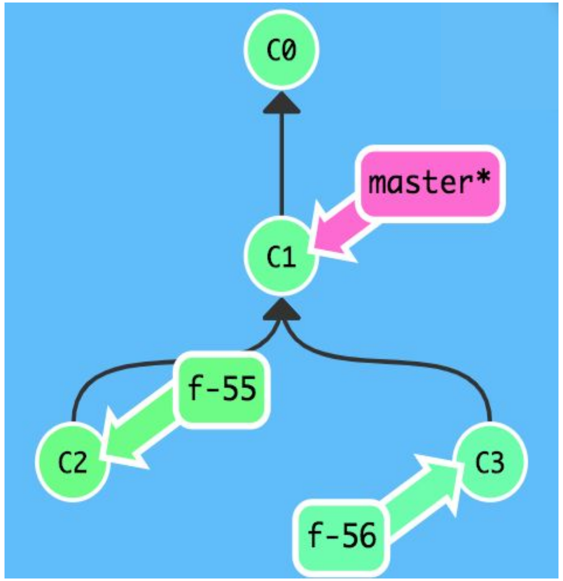


Chart generated using learngitbranching.js.org

Github

Working with remotes

```
$ git remote [add, show, rename, remove, ...]
$ git pull
$ git fetch
$ git push
```

Working with remotes

- Put things online

```
$ git push -u origin master
```

- Pull/push changes

```
$ git pull
$ git push
```

- Push a local branch

```
$ git push origin my-branch
```

- Delete a remote branch

```
$ git push -d origin my-branch
```

Some extra features of Github

- autolinked references

https://help.github.com/en/articles/autolinked-references-and-urls

- closing issues

https://help.github.com/en/articles/closing-issues-using-keywords

Branching model

Gitflow

Gitflow is a development model that defines a strict branching strategy centered around the idea of releases.

Elements

- Main branches: master branch, develop branch
- Supporting branches: feature branches, hotfix branches, release branches.

References
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
https://nvie.com/posts/a-successful-git-branching-model/

Master and Develop branches

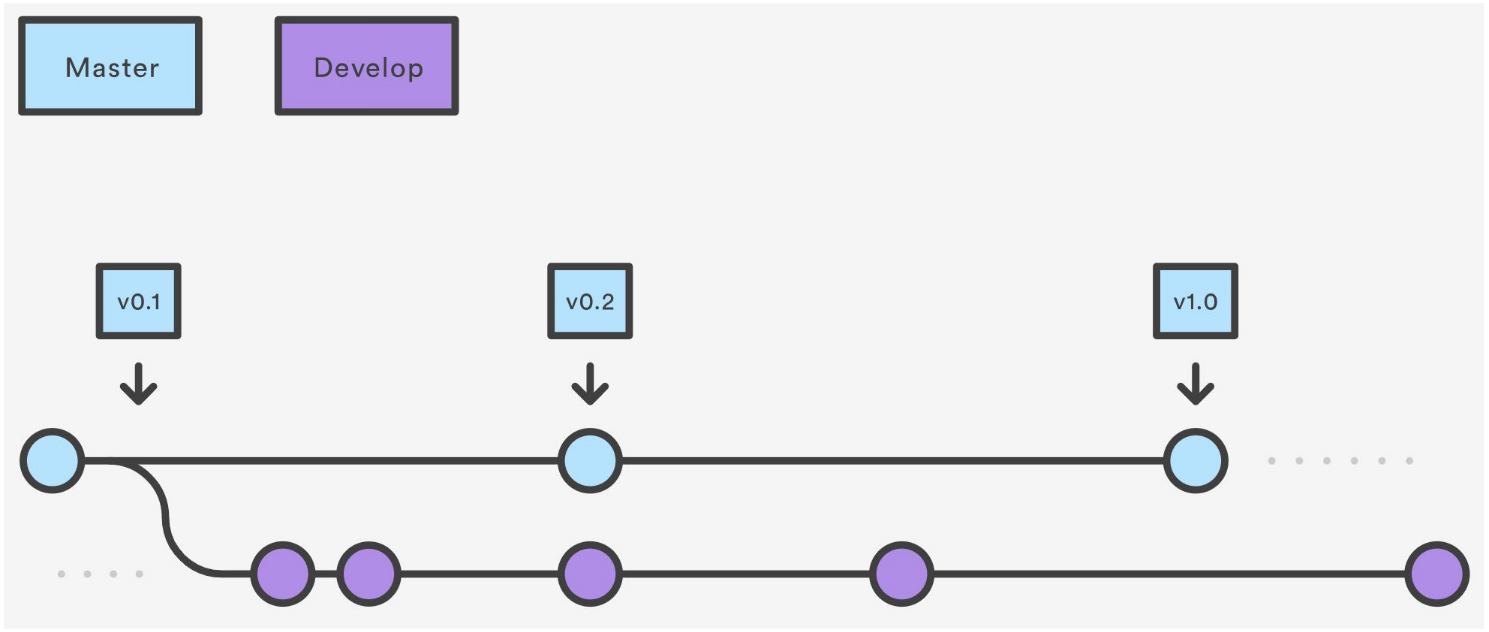


Image from Atlassian

Feature branches

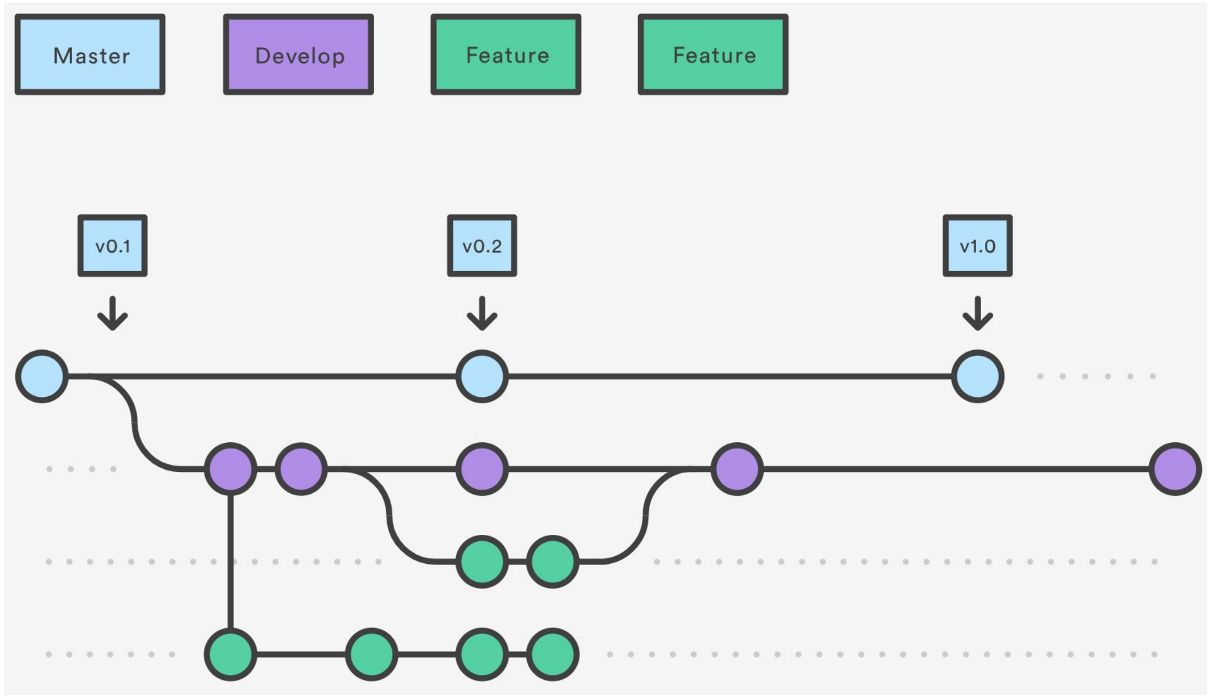


Image from Atlassian

Release branches

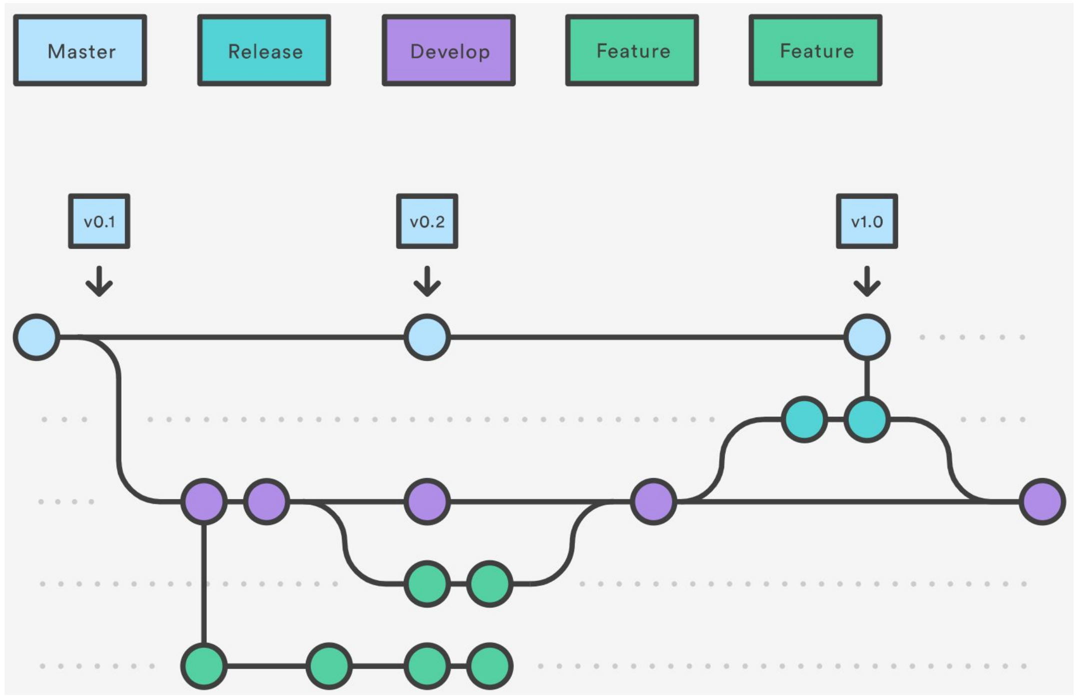


Image from Atlassian

Hotfix branches

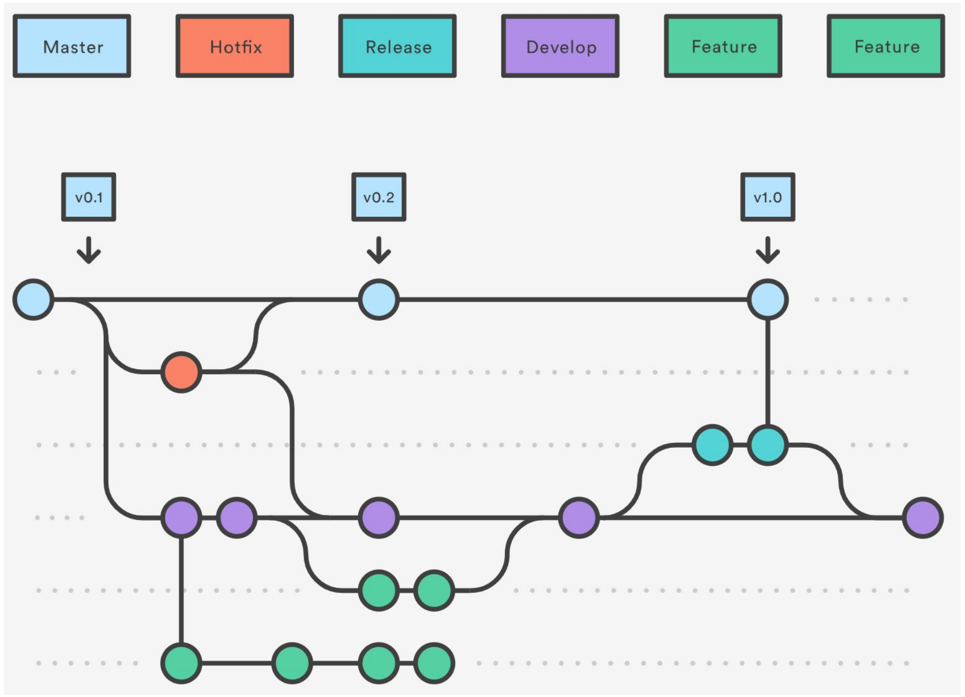


Image from Atlassian

Gitflow practice

1. Create a develop branch and push it to your Github repository.
2. Add some commits to the branch develop and push.
3. Create a hotfix branch off of master branch add a commit and push.
4. Merge the hotfix branch into master and push.
5. Merge the hotfix branch into develop.

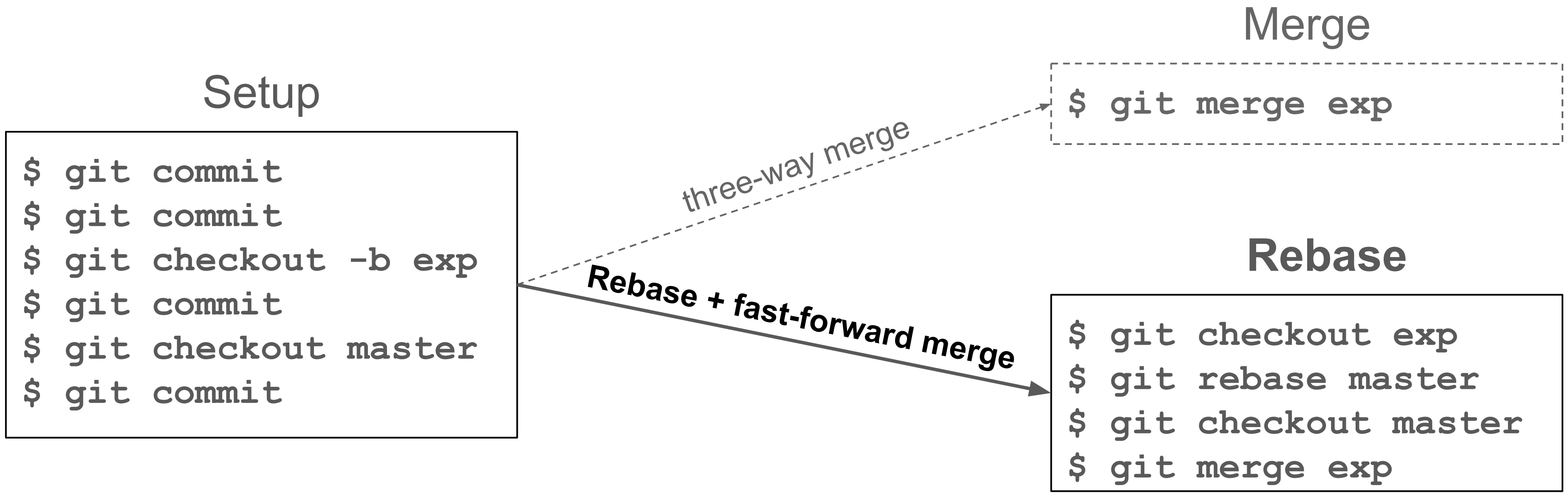
Assignment #1

https://tinyurl.com/se2-a1

(some) Advanced git

git rebase

Another way of integrating changes.



Setup

```
$ git commit
$ git commit
$ git checkout -b exp
$ git commit
$ git checkout master
$ git commit
```

Merge

```
$ git merge exp
```

Rebase

```
$ git checkout exp
$ git rebase master
$ git checkout master
$ git merge exp
```

Use learngitbranching.js.org to see the differences

git stash

Stashing allows us to store half-done work without doing a commit.

```
$ … started working on feature-58 …
$ … the boss asks you to fix an urgent bug…
$ git stash
$ … work on hotfix and push. Then …
$ git checkout feature-58
$ git stash apply
$ … continue working on feature-58 …
```

git rebase -i

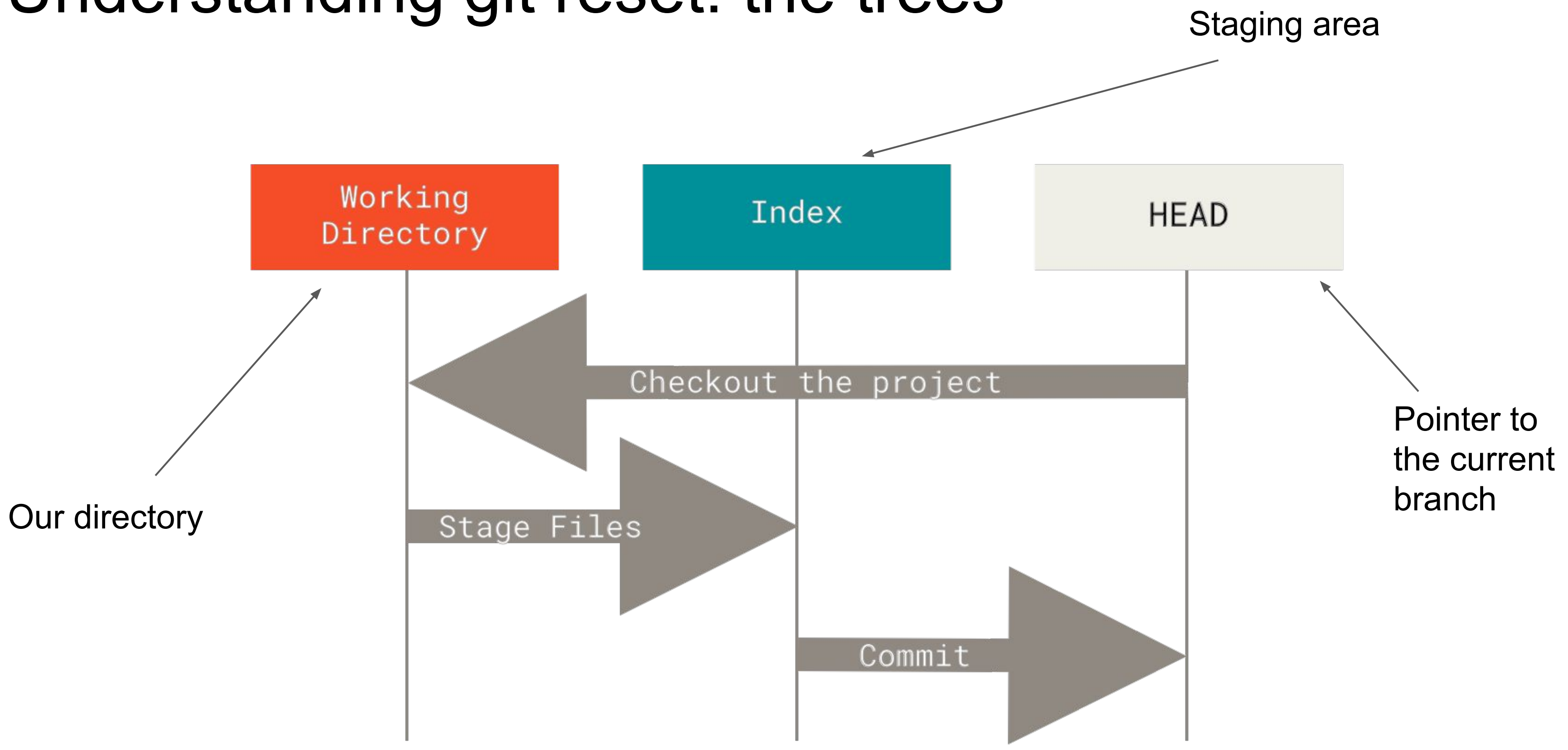
Interactive rebasing allows us, among other things, to **squash** multiple commits into one.

```
$ … started working on feature-58 …
$ git commit -m "day #1 on feature-58"
$ git commit -m "day #2 on feature-58"
$ git commit -m "day #3 on feature-58"
$ git rebase -i HEAD~3
$ … interactive rebase, pick, squash, squash …
$ git checkout develop
$ git merge feature-58
$ git push
```

References
https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History
https://www.internalpointers.com/post/squash-commits-into-one-git

Part #3

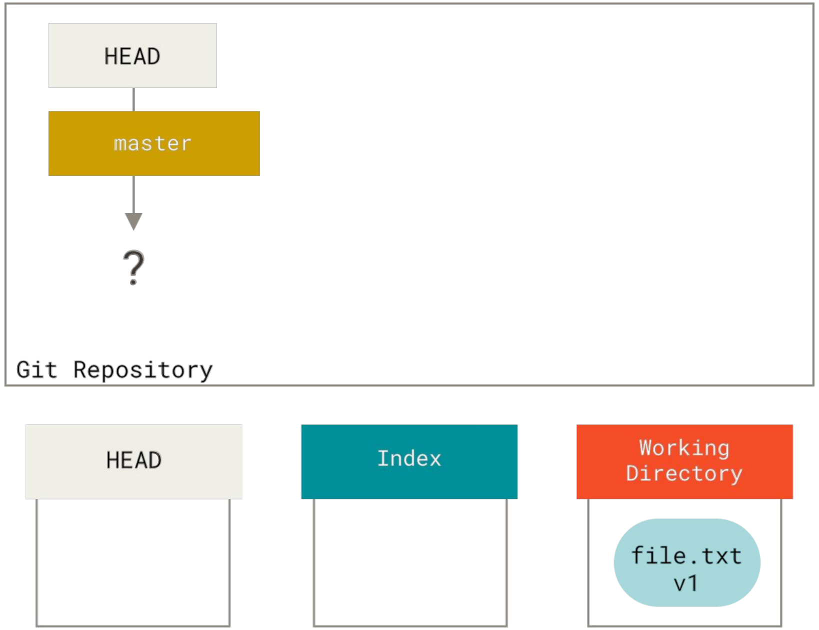
Understanding git reset: the trees



The "three trees". Source: The git book

Understanding git reset: creating a repo

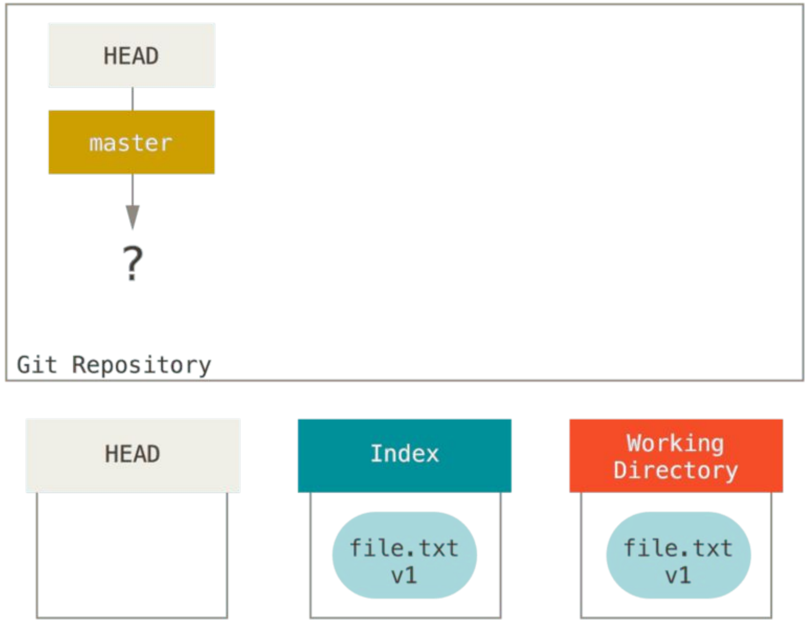
```
$ git init
```



Git basically populates the Working Directory tree.
Source: The git book

Understanding git reset: preparing for commit

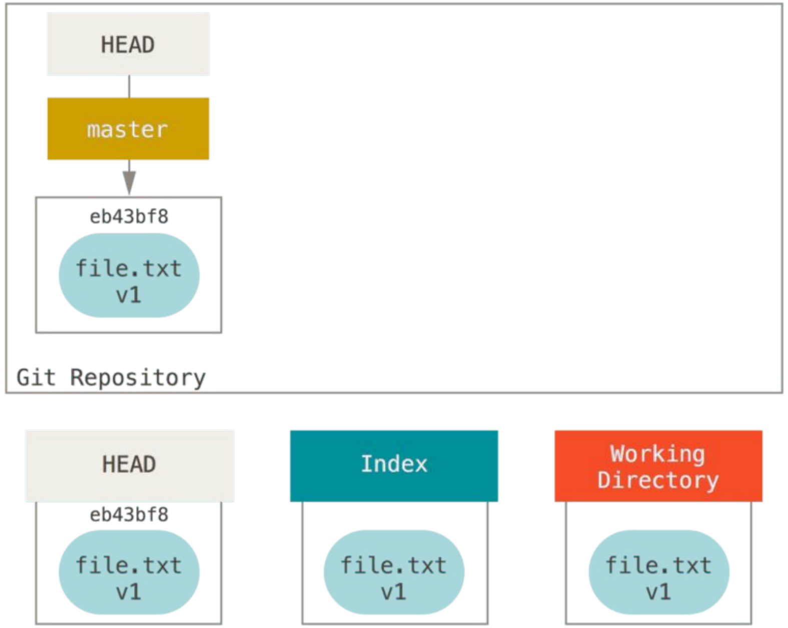
```
$ git add
```



Git moves things from the Working Directory to the Index.
Source: The git book

Understanding git reset: commit changes

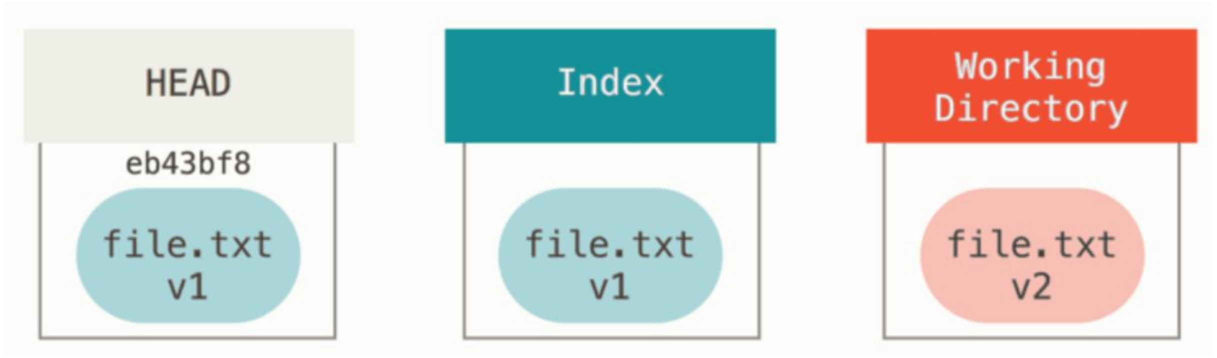
```
$ git commit
```



Git takes what's on the Index and creates an snapshot, a commit object and updates master.
Source: [The git book]

Understanding git reset: git status revisited

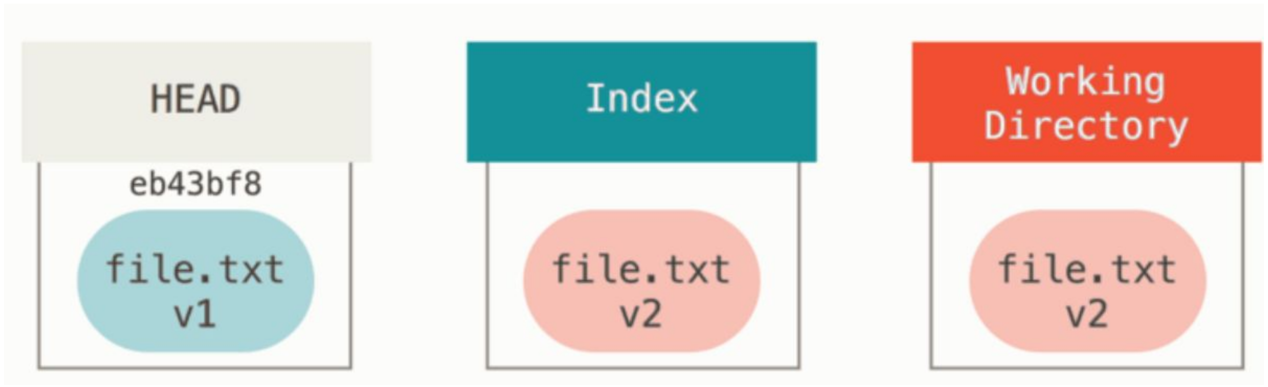
Changes not staged for commit (diff between Index and Working directory)



Source: The git book

Understanding git reset: git status revisited

Changes to be committed (diff between HEAD and Index)



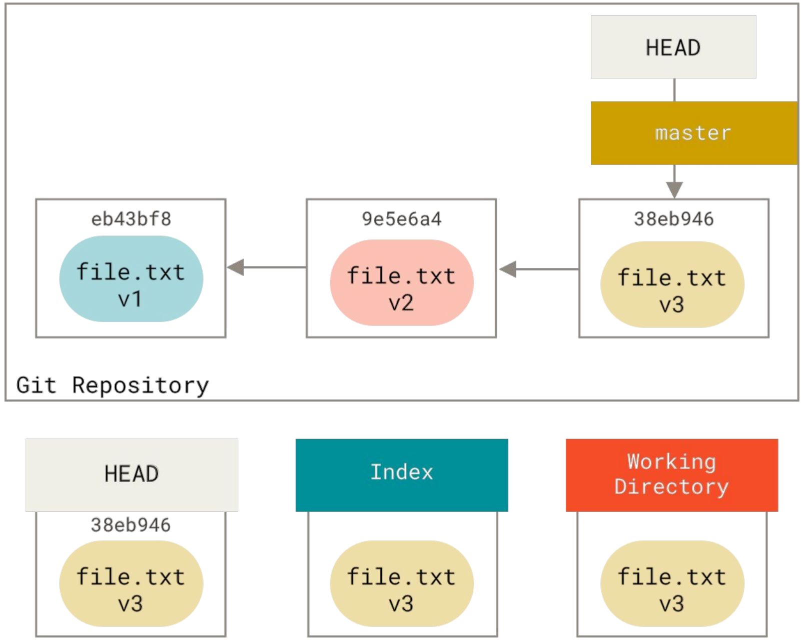
Source: The git book

The role of reset

`git reset` allows us to manipulate the "three trees", in 3 steps.

Steps

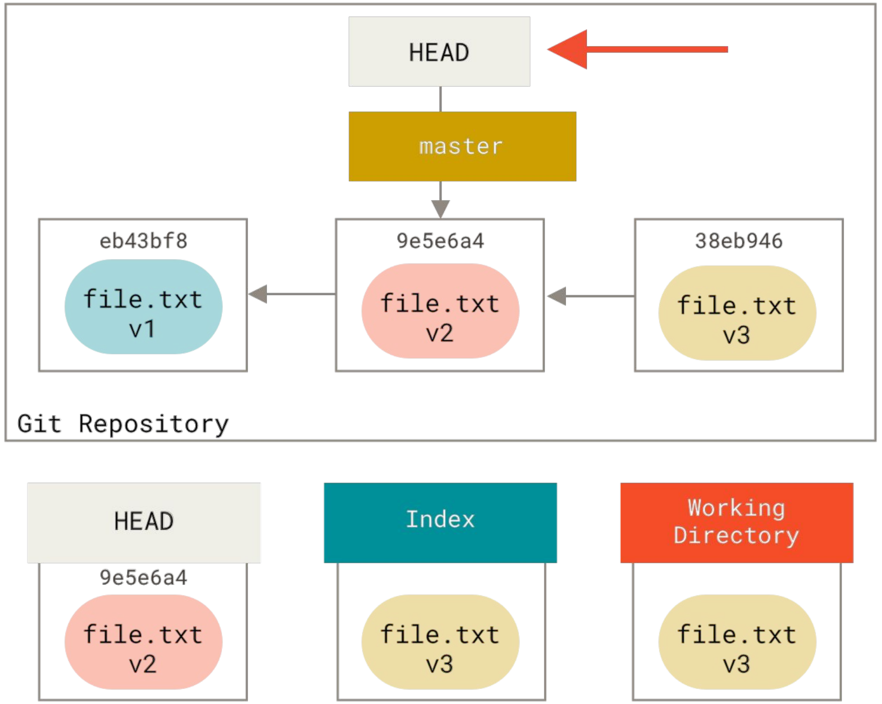
1. Move HEAD
2. Update Index
3. Update Working Directory



Source: The git book

Move HEAD: undo a commit

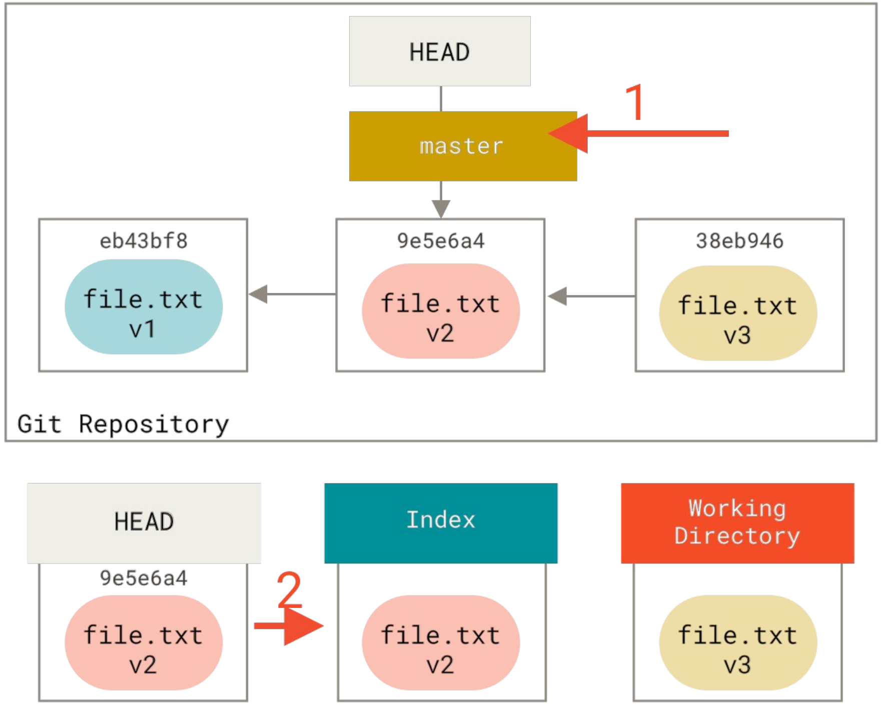
```
$ git reset --soft HEAD~
```



Source: The git book

Update Index: undo a commit and unstage

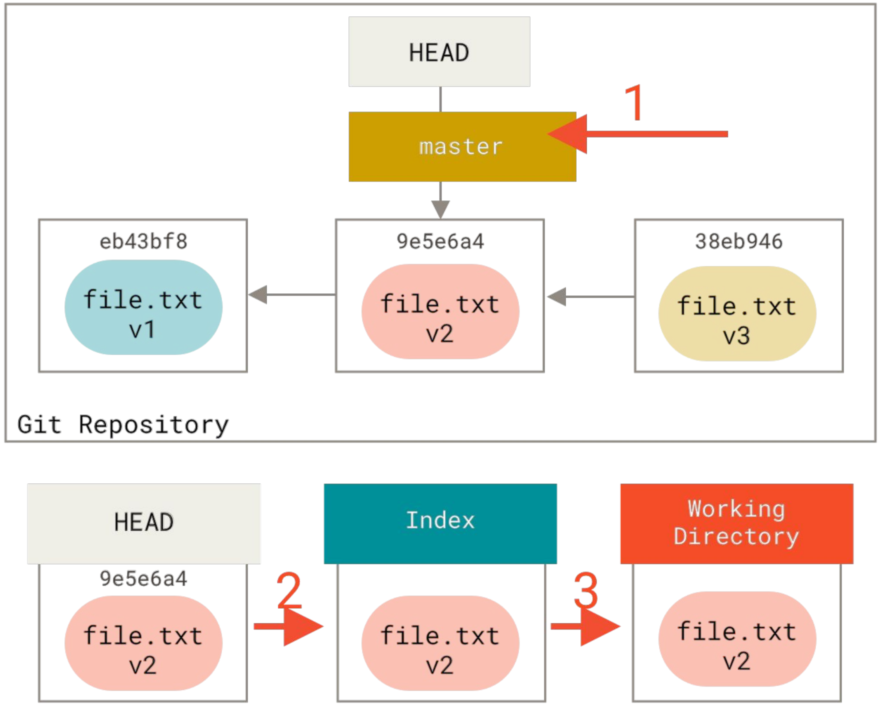
```
$ git reset --mixed HEAD~
```



Source: The git book

Update Working Directory: undo, unstage, erase

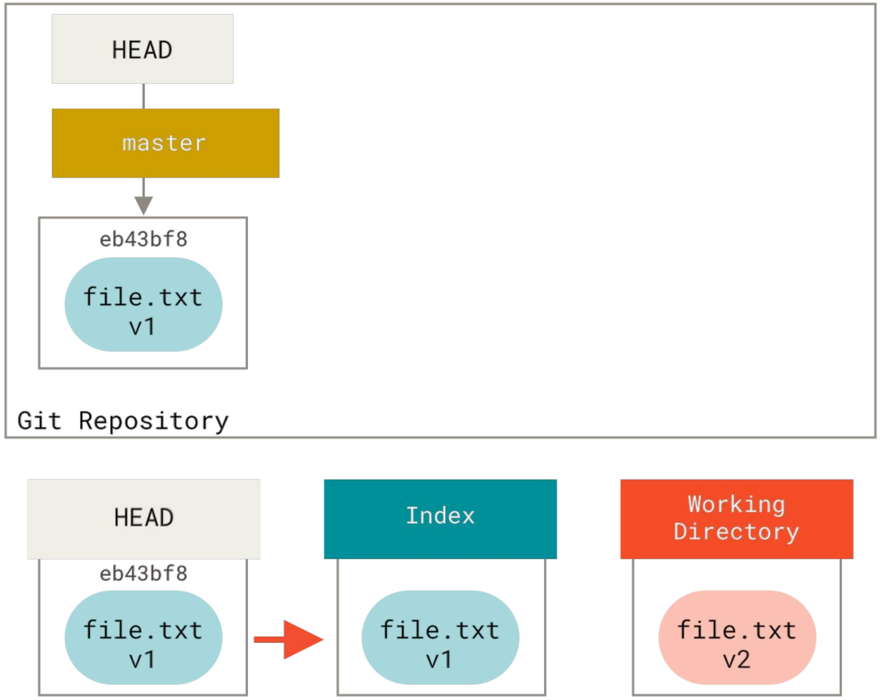
```
$ git reset --hard HEAD~
```



Source: The git book

Reset with a path

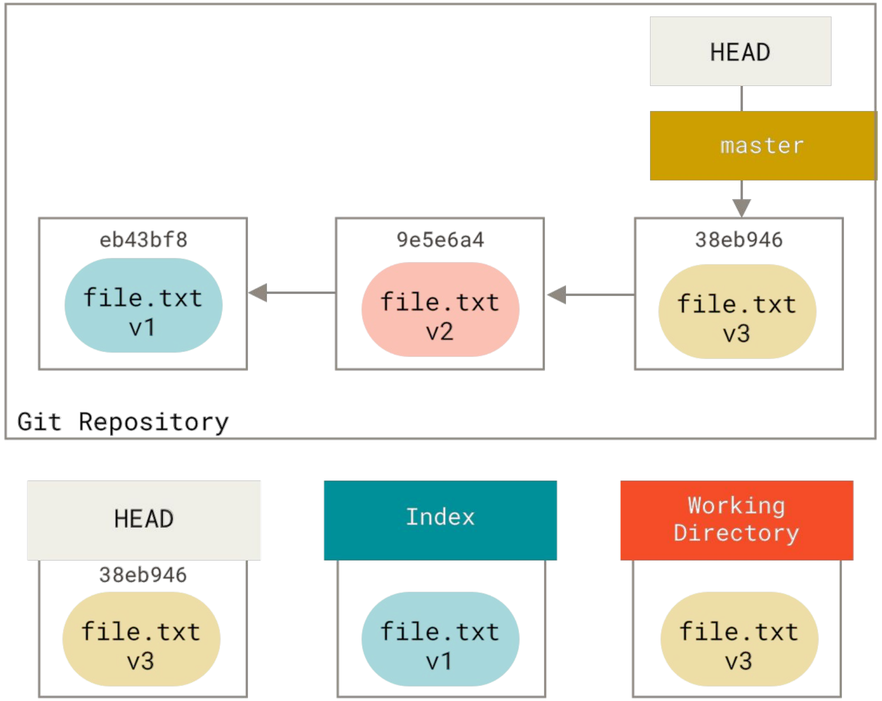
```
$ git reset HEAD file.txt
```



Source: The git book

Reset with a path

```
$ git reset eb43 README.md
```



Source: The git book

The role of checkout

`git checkout` allows us to manipulate the "three trees" too. But it is different from `git reset` depending on whether it receives a file path or not.

Moving to a branch

```
$ git checkout develop
```

Like doing `git reset --hard develop` in that git updates the "three trees" to look like the develop branch. Then `checkout` updates HEAD to point to the develop branch.

Undoing changes

```
$ git checkout -- <file>
```

Like `reset`, it does not move HEAD. And it is similar to `git reset [branch] file` BUT it also updates the Working Directory.

Editing our last commit

- Use `git commit --amend` to edit the last commit.
- **ONLY** for local commits (not yet pushed).

Thanks!

Questions from the class

How can I remove a global configuration?

- The "clean" way: remove the user section (user.name, user.email)

```
$ git config --global --remove-section user
```

- The "not-so-clean" way: edit the configuration file by manually removing the section.

```
$ vi $HOME/.gitconfig
```

How can I "see" what's on the Index? (1/2)

The Index is **our proposed changes for the next commit**, also known as the "staging area". So this question is twofold:

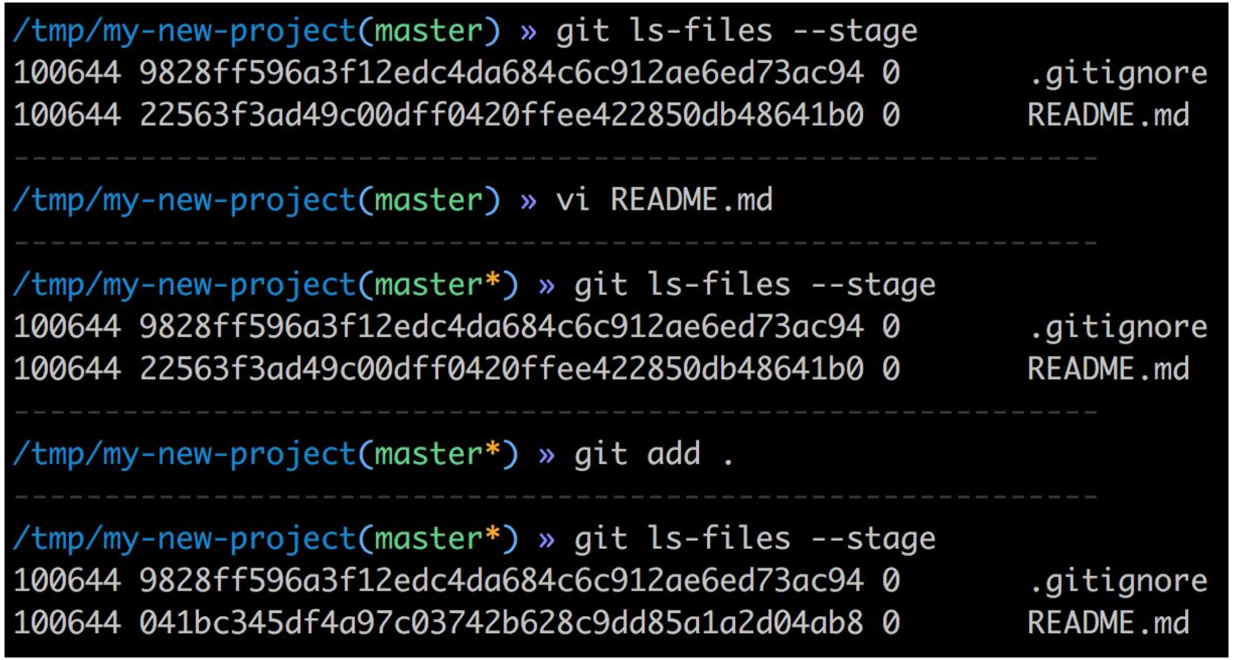
- `git status` shows what's going to be part of the next commit. But it shows only the files we changed and staged using `git add`. However, in the Index, there are also files that didn't change. And for this unchanged files, Git stores a pointer to the last snapshot of the file.
- But once we committed, we can inspect the Index by:
  - Using git show and see what was part of our last commit.
    - `git show --stat`
  - Using git ls-file and inspect how the Index currently looks like. This is a plumbing command. A concrete example of this on the next slide.
    - `git ls-files --stage`

How can I "see" what's on the Index? (2/2)

We see the index of our last commit. Here, git status will show you "nothing to commit".

We edit a file, but we DO NOT add the changes to the Index. Here git status only tells you "Changes not staged for commit".

We do git add. This updates the Index. Now you can notice that the README has changed (check the hash value). And git status will tell you "Changes to be committed".

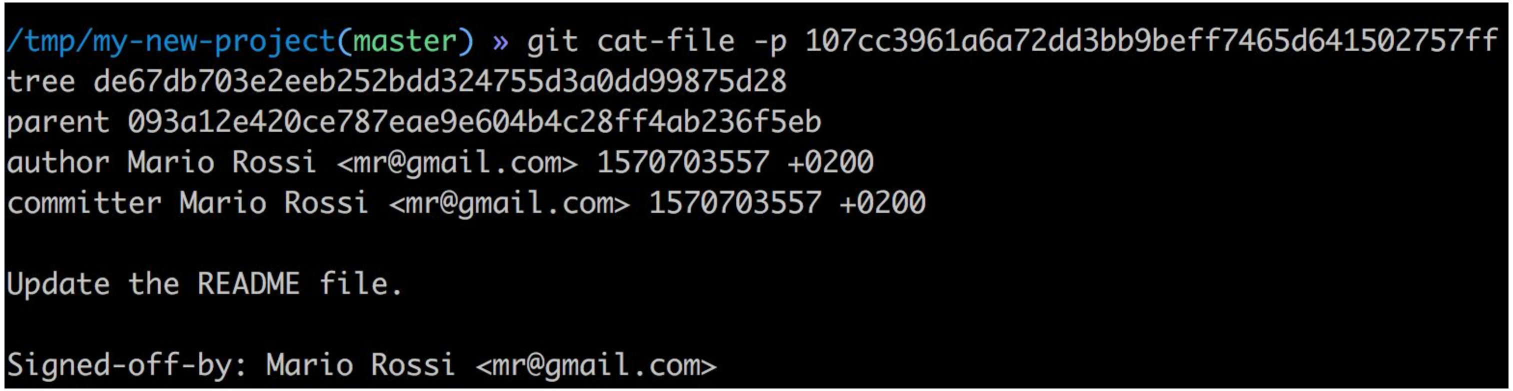


```
/tmp/my-new-project(master) » git ls-files --stage
100644 9828ff596a3f12edc4da684c6c912ae6ed73ac94 0       .gitignore
100644 22563f3ad49c00dff0420ffee422850db48641b0 0       README.md
------------------------------------------------------------
/tmp/my-new-project(master) » vi README.md
------------------------------------------------------------
/tmp/my-new-project(master*) » git ls-files --stage
100644 9828ff596a3f12edc4da684c6c912ae6ed73ac94 0       .gitignore
100644 22563f3ad49c00dff0420ffee422850db48641b0 0       README.md
------------------------------------------------------------
/tmp/my-new-project(master*) » git add .
------------------------------------------------------------
/tmp/my-new-project(master*) » git ls-files --stage
100644 9828ff596a3f12edc4da684c6c912ae6ed73ac94 0       .gitignore
100644 041bc345df4a97c03742b628c9dd85a1a2d04ab8 0       README.md
```

How can I see a git object?

- Use `git cat-file` to inspect the objects in Git's database (i.e., .git folder). This is another plumbing command.
- For example, to see a commit object:

```
$ git cat-file -p <commit hash>
```



```
/tmp/my-new-project(master) » git cat-file -p 107cc3961a6a72dd3bb9beff7465d641502757ff
tree de67db703e2eeb252bdd324755d3a0dd99875d28
parent 093a12e420ce787eae9e604b4c28ff4ab236f5eb
author Mario Rossi <mr@gmail.com> 1570703557 +0200
committer Mario Rossi <mr@gmail.com> 1570703557 +0200

Update the README file.

Signed-off-by: Mario Rossi <mr@gmail.com>
```