combined_readme.md

01- Introduction

Course Topics

1. Fundamental concepts
2. Creating snapshots
3. Browsing project history
4. Branching & merging
5. Collaborating using GitHub
6. Rewriting history

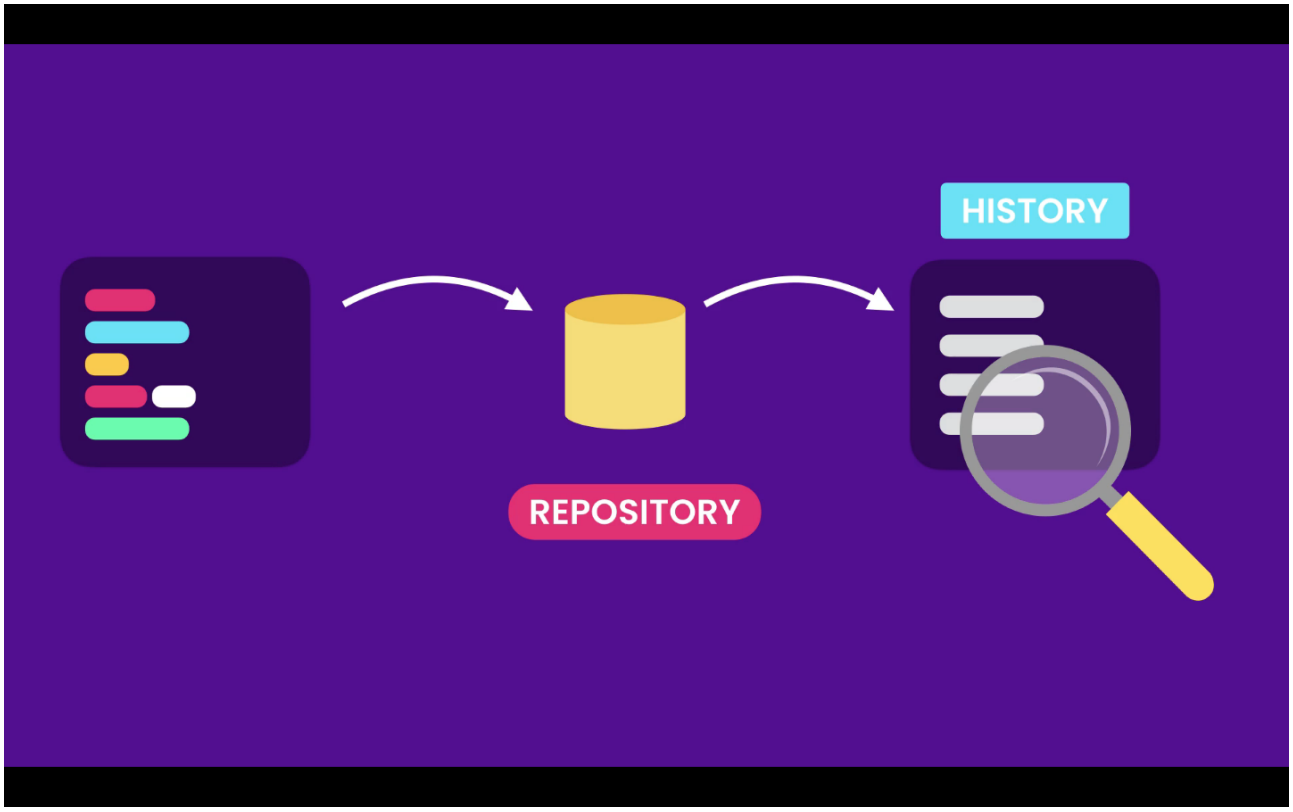
02- How to Take This Course

Mosh advice "DO NOT SKIT ANY LESSONS"!# 03- What is Git

What is Git

Git is the most popular **Version Control System (VCS)** in the world.

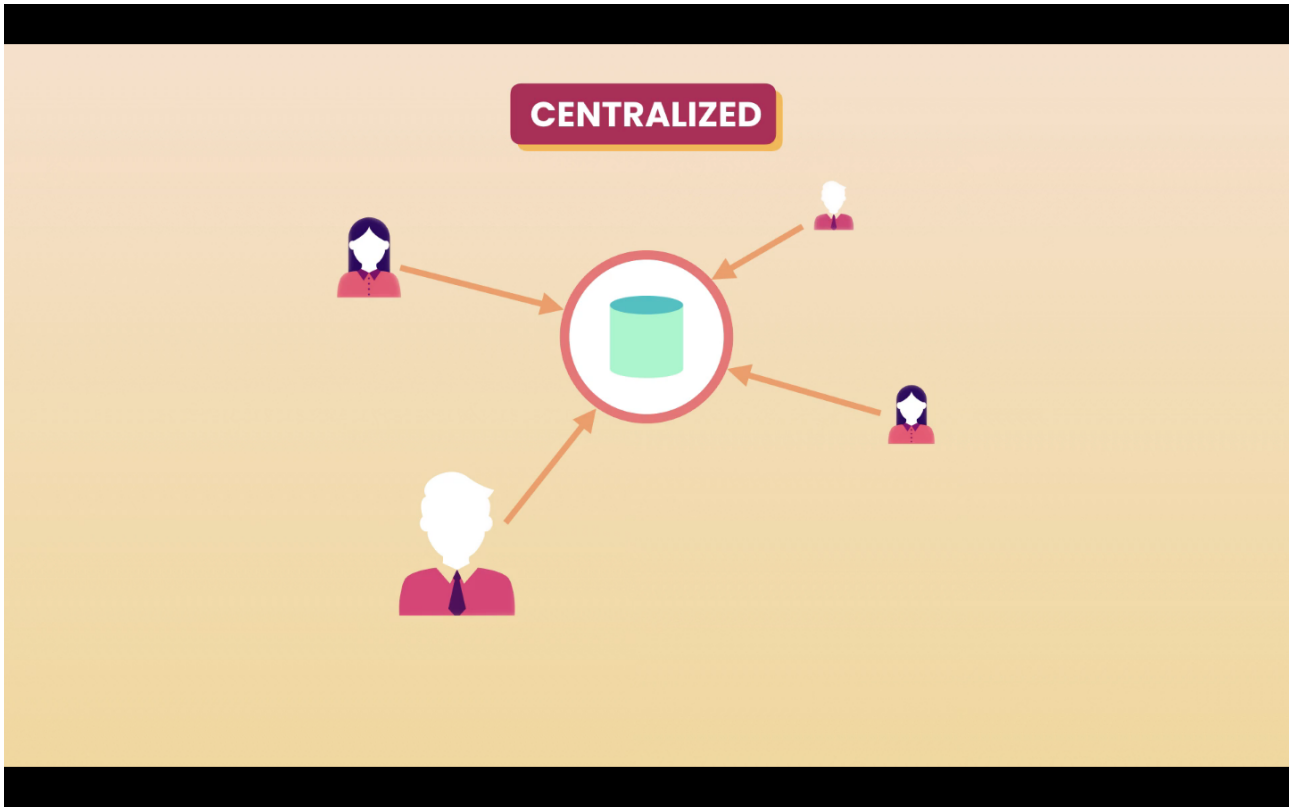
Git records the changed made to our code in a special database called repository. It enable us to look into our project history, and see what changes where made, by whom, when and why. And if we mess something it we can easily revert our project back to an earlier state.



Version Control Systems categories

Centralized

In a **Centralized** system all team members connect to a central sever to get the latest copy of the code, and to share their changes with others.



Examples:

- Subversion
- Microsoft Team Foundation Server

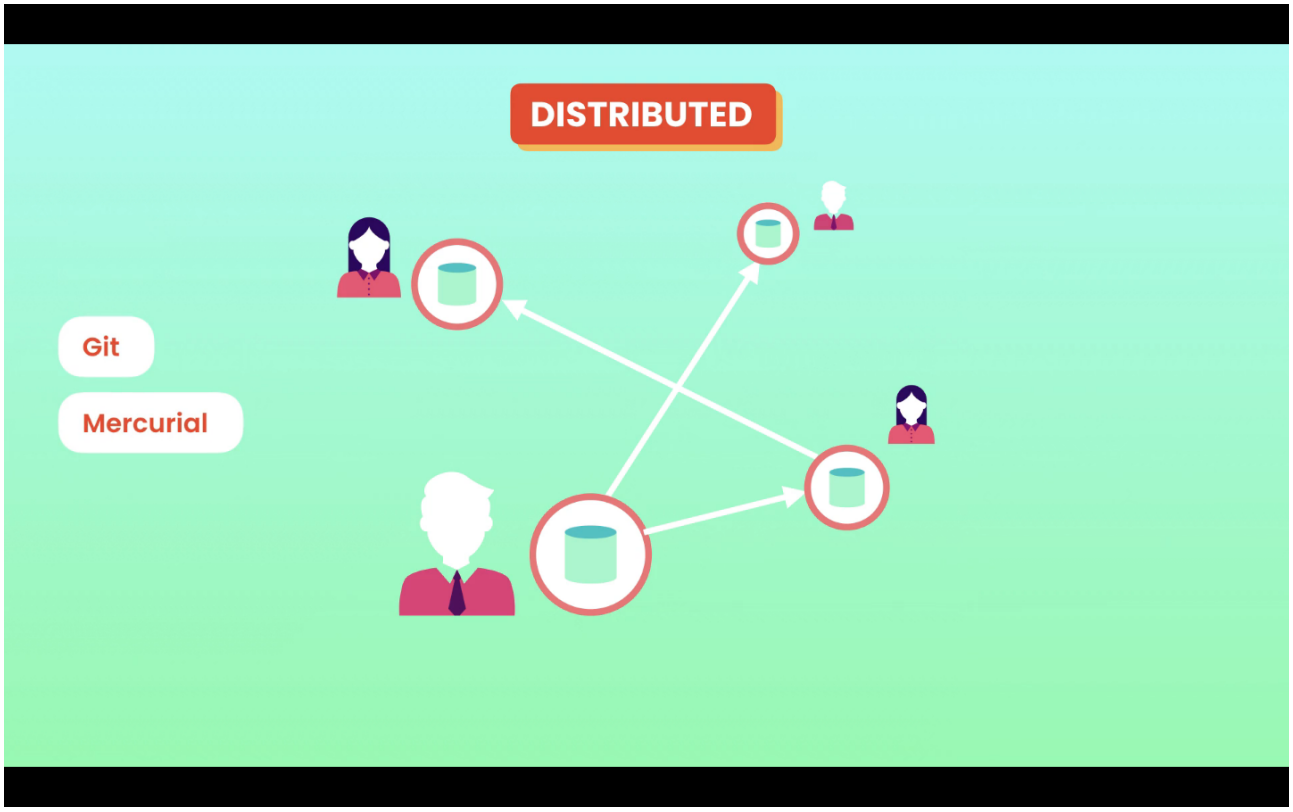
The problem with a Centralized VCS is the single point of failure, if the server gos offline it is impossible to collaborate, or continue to take snapshots of the code.

Distributed

In a **Distributed** system each team member as a full copy of the code and repository in ts machine, if the server goes offline they can keep on working

Examples:

- Git
- Mercurial



Why Git?

Reasons to choose Git

- It is free
- Open Source
- Super Fast
- Scalable

Operations like branching and merging are painful in other VCS.# 04- Using Git

Ways to use Git

Command-line

The command-line is the most common, and probably the fastest way to use Git. Most GUI tools have limitations.

Sometimes GUI tools might not be available. Ins case for example if we connect to a server remotely and do not have permission to install a GUI tool.

Code Editors & IDEs

Most code editors these days have built-in GUI tools and extensions to use Git.

- Vs code Extensions:
  - GitLens

Graphical User Interfaces

In the Git website there is a complete list of GUI tools for all the main platforms Mac, Linux and Windows.

The most popular ones are:

- GitKraken
- SourceTree

05- Installing Git

On macOS

In macOS it is recommended to use **Homebrew**.

Install **Homebrew**, and run the following command `brew install git` .

06- Configuring Git

Setting

The fist time we use git we have to specify a few configuration setting.

- Settings:
- Name
- Email
- Default editor
- Line ending

Settings Level

We can specify these configurations setting at 3 different levels

1. System ---> Apply to all users of the current computer.
2. Global ---> Apply to all repositories of the current user.
3. Local ---> Apply to the current repository.

Command to apply settings

The command to apply setting is `git config --global <setting> <value>` . The flag `--global` specifies we are applying setting the the Global level.

- In the terminal type:
  - `git config --global user.name "Miguel Pimenta"`
  - `git config --global user.email my-email@code.com`
  - `git config --global core.editor "code --wait"` code for **VS Code** the `--wait` tells the terminal to wait until the window is closed.
  - `git config --global --edit` command to open the config file in the editor.

End of lines

To manage end of line correctly we have to configure a property called `core.autocrlf` . These is a very import setting, so git can properly handle end of lines.

Windows

On Windows end of lines are marked with two special characters:

- Carriage Return: `\r`
- Line Feed: `\n`

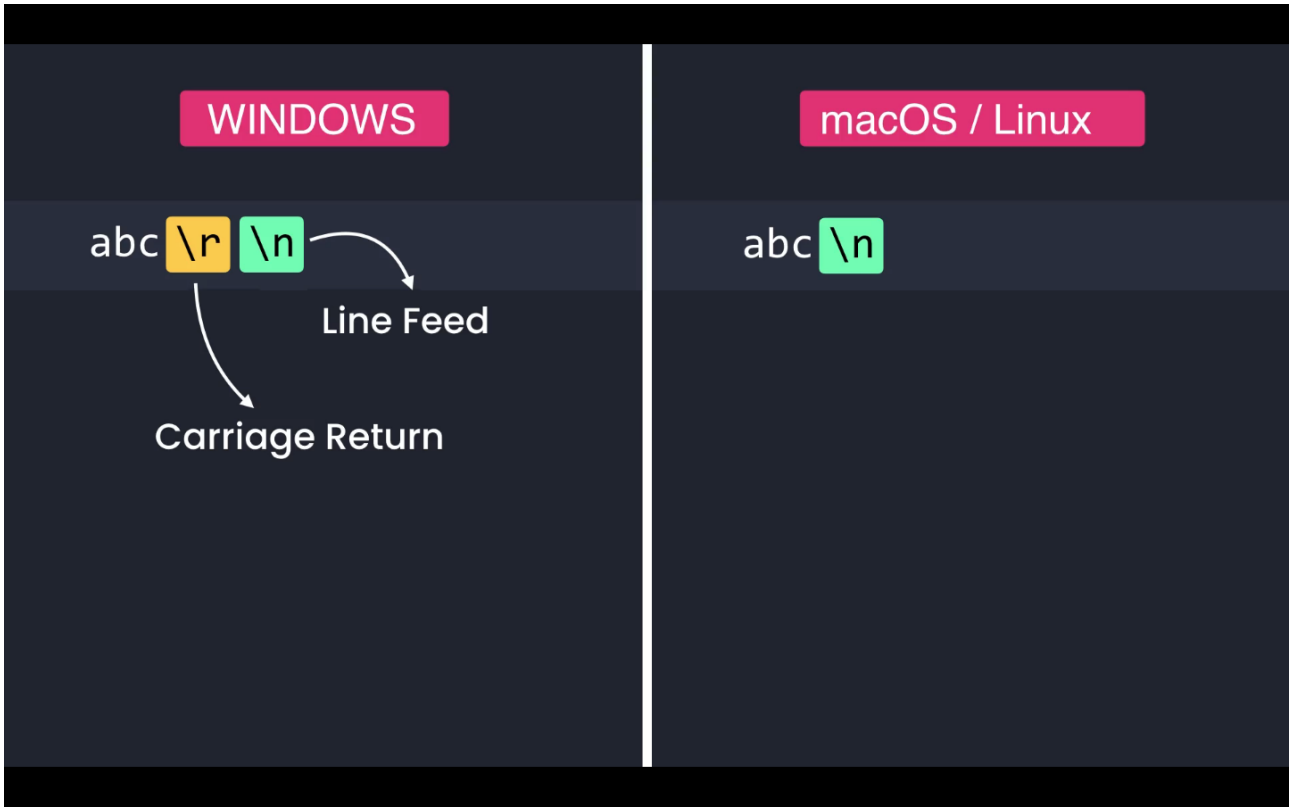
```
git config --global core.autocrlf true
```

macOS / Linux

On macOS and Linux end of lines are marked with one special character:

- Line Feed: `\n`

```
git config --global core.autocrlf input
```



07- Getting Help

To get help about git command we can type a command followed by the `--help` flag. For example `git config --help` will give us the help topics about the `config` command.
Press `space` to go to the next page and `esc` to exit.
If we use the flag `-h` we will get a shorter summary of the help topics.

combined_readme.md

01- Introduction

Creating Snapshots and the fundamental concepts in Git

02- Initializing a Repository

Initialize a git repository

Create a directory and put it any where. In may case `/Users/jmschp/code/mosh-ultimate-git-course/` .

Inside the directory run `git init` . This command should return the following message:

```
❯ git init
Initialized empty Git repository in /Users/jmschp/code/mosh-ultimate-git-course/.git
```

Inside our project folder the command `git init` as created a folder `.git` , by default it is hidden because we are not supposed to touch it. With the command `ls -a` we can see the git sub directory.

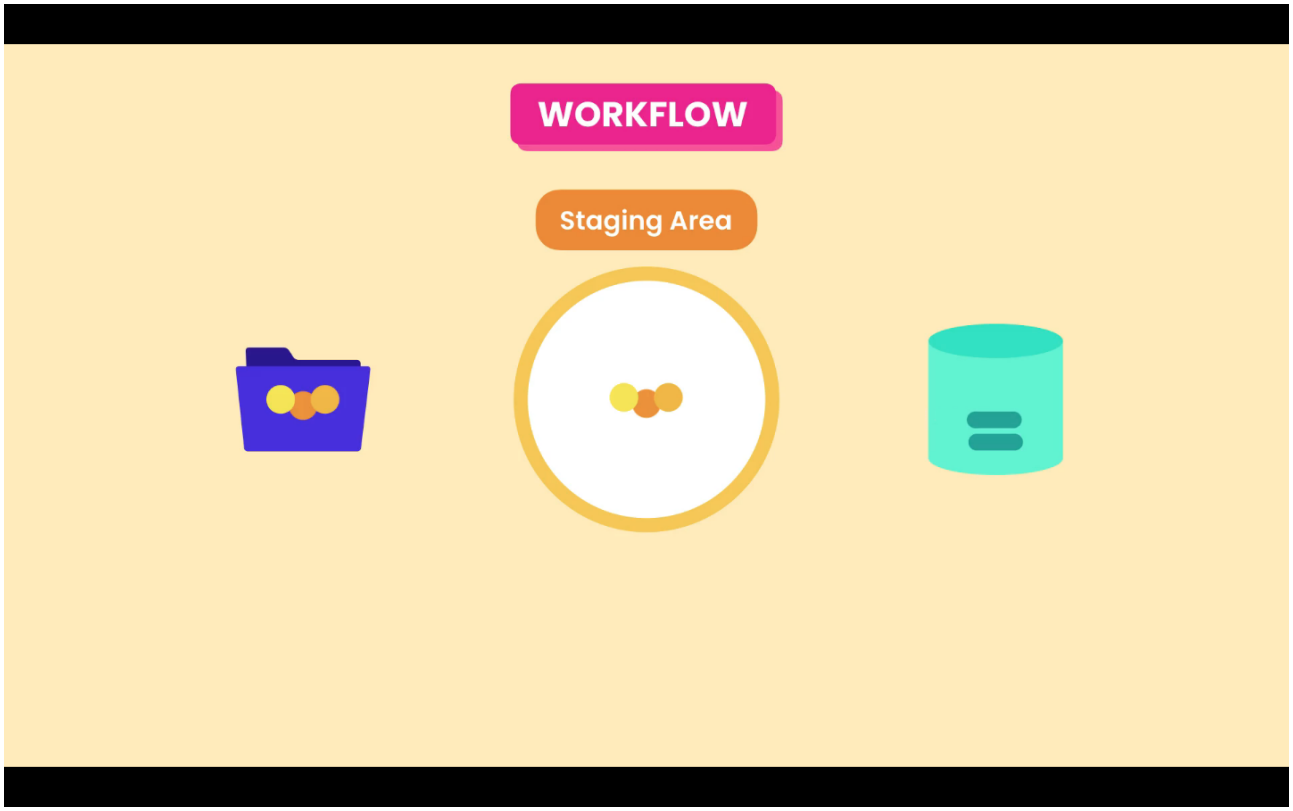
If we corrupt or delete this directory we lose the project history.

03- Git Workflow

Workflow

In the daily work, we modify one or more files. When we believe that we reach a point that we want to record we commit those changes into the repository. Creating a commit it is like taking a snapshot of our project.

In Git there is a special area, an intermedial area, between the ***Working Directory*** and the ***Repository***. Its called the ***Staging*** or ***Index*** area. Files in these are are the one that will be in the next coommit.



First we add our changes to the **Staging Area**, and there we are able to review our changes, and them we make a commit. The commit will we save a snapshot to the repository. If some of the changes should not the saved in the next commit, we can remove that file from the **Staging Area**.

After we make the commit the **Staging Area** it is not emptied. What we have there now is the same snapshot the was record in the repository.

Git commands

git add

To add files to the staging area we use the `git add` command:

```
git add file1 file2
```

We can add multiple files at the same time or use `git add .` to add all files, that have been changed, in the current directory recursively.

git commit

To record the snapshot to the repository we use the `git commit` command, with the `-m` flag and a meaningful message, about the changes we have made.

```
git commit -m "Initial commit"
```

Each commit contains a unique identifier, and also information of what was changed:

- ID
- Message
- Date / time
- Author
- Complete snapshot of the project

```
commit f289d8ab48e5530b659931667edbc73e3eb10e29 (HEAD -> main, origin/main)
Author: Miguel Pimenta <my-email@code.com>
Date:   Sat Feb 27 23:15:48 2021 -0300

    First commit
```

Unlike other VCS Git stores, in each commit, the full snapshot of the project. Other VCS only store deltas or what was changed. Git those these in a very efficient manner, compressing the content and it does not store duplicates.

04- Staging Files

Git those not track files automatically we have to tell Git to track new files we add. Even when we start a new repository with `git init`, in project that already as several files, Git will only track them, we need to add them.

We can run the `git status` command to see the status of the **Working Directory** and **Staging Area**.

The output will be something like these:

```
❯ git status
On branch main

Untracked files:
```

```
  (use "git add <file>..." to include in what will be committed)
        02 Creating Snapshots/04- Staging Files.md

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

The above message is telling us we have untracked files.

Use the `git add` command to stage that file. In my case I have to quotation marks `""`, because there are whitespace, in the file path

```
git add "02 Creating Snapshots/04- Staging Files.md"
```

We can also use patterns to add files.

```
git add *.md
```

These will add all the files with an `.md` extension.

Now if we run `git status` again, we will have the following output:

```
❯ git status
On branch main

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   02 Creating Snapshots/04- Staging Files.md
```

If we make changes to the same file, after adding it to the **Staging Area** and run `git status`, we will have the following output:

```
❯ git status
On branch main

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   02 Creating Snapshots/04- Staging Files.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   02 Creating Snapshots/04- Staging Files.md
```

We have the same file in the **Staging Area** and also marked as modified, and not staged. When we run the `git add` command Git took a snapshot of that file and added it to the **Staging Area**. So now we have a one version of the file in the staging area and another version in the **Working Directory**.

We can run `git add` one more time to add our changes to the **Staging Area**.

```
❯ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   02 Creating Snapshots/04- Staging Files.md
```

05- Committing Changes

Now that we have files in the **Staging Area**, we can commit them to the repository, with the command `git commit -m "My commit message"`

```
❯ git status
On branch main

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   02 Creating Snapshots/05- Committing Changes.md
```

With the command `git commit -m "New lesson started"` a snapshot will be saved to the repository.

```
❯ git commit -m 'New lesson started'
[main cb4a472] New lesson started
 1 file changed, 4 insertions(+)
 create mode 100644 02 Creating Snapshots/05- Committing Changes.md
```

If we commit to more than one file, we will se the following message:

```
❯ git commit -m 'Lesson completed'
[main bc32e03] Lesson completed
 2 files changed, 25 insertions(+)
```

```
 create mode 100644 02 Creating Snapshots/06- Committing Best Practices.md
 create mode 100644 02 Creating Snapshots/07- Skipping the Staging Area.md
```

When a short, one line, message is not sufficient, because we need to explain in detail the changes that where made we can use the command `git commit`, without the `-m "My message"` part. These will open the default editor with a file named `COMMIT_EDITMSG`.

In the first line we add a short description, ideal less than 80 characters, then we addd a line break and the more detail message. After we save and close the file the changes are committed. And we will hae in the terminal and ouput like the following:

```
❯ git commit
[main e0a1a79] Continuing lesson 5 Committing Changes
 1 file changed, 18 insertions(+)
```

Example detailed commit:

```
Continuing lesson 5 Committing Changes

Adding more details to the lesson number 5 Committing Changes of section 2 Creating
Lectured by Mosh Hamedani.
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Your branch is ahead of 'origin/main' by 2 commits.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#       modified:   02 Creating Snapshots/05- Committing Changes.md
#
# ------------------------ >8 ------------------------
# Do not modify or remove the line above.
# Everything below it will be ignored.
diff --git a/02 Creating Snapshots/05- Committing Changes.md b/02 Creating Snapshots
index 10da8dc..21c6f07 100644
--- a/02 Creating Snapshots/05- Committing Changes.md
+++ b/02 Creating Snapshots/05- Committing Changes.md
@@ -2,3 +2,21 @@

 Now that we have files in the **Staging Area**, we can commit them to the repositor

+```zsh
+On branch main
+
```

```
+Changes to be committed:
+  (use "git restore --staged <file>..." to unstage)
+        new file:   02 Creating Snapshots/05- Committing Changes.md
+```
+
+With the command `git commit -m "New lesson started"` a snapshot will be saved to t
+
+```zsh
+git commit -m 'New lesson started'
+[main cb4a472] New lesson started
+ 1 file changed, 4 insertions(+)
+ create mode 100644 02 Creating Snapshots/05- Committing Changes.md
+```
+
+When a short, one line, message is not sufficient, because we need to explain in de
\ No newline at end of file
```

06- Committing Best Practices

Our commits should not be to small neither to big. We do not want to commit every time we change a file, neither we want to wait until we implement a featured end to end before committing.

The all point of committing is to record checkpoint as we go.

Commit often when we believe the project or file is at a state we want to record.

For example if we are fixing a **Bug** than we find a **Typo** we should make separate commits. One for the **Bug** and another for the **Typo**.

Wording

Most people like to use the present tense for commit messages. But other conventions can be used.

- PRESENT: Fix the bug
- PAST: Fixed the bug

Conventional Commits

More in depth detail about commit messages in Conventional Commits.

07- Skipping the Staging Area

We don't always have to stage our changes before committing. But do this only if we are sure our changes do not need to be reviewed.

To do this we run the command `git commit -a -m "My message"`, we supply the flag `-a`, that means all modified files. Or we can use `git commit -am "My message"`, combining `-a -m`.

```
❯ gc -am 'Add details to lesson'
[main d0dd608] Add details to lesson
 1 file changed, 10 insertions(+)
```# 08- Removing Files

To delete a file from the project we use the same workflow. Just delete teh file nor

I have added and committed a file names `test.txt` and then deleted that file. When

```zsh
❯ git status
On branch main

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   02 Creating Snapshots/08- Removing Files.md
        deleted:    test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

In the example we have one file marked **modified** and another file marked as **deleted**.

If we run `git ls-files` we will see a list of files in the **Staging Area**, in my case:

```
❯ git ls-files
01 Getting Started/01- Introduction.md
01 Getting Started/02- How to Take This Course.md
01 Getting Started/03- What is Git.md
01 Getting Started/04- Using Git.md
01 Getting Started/05- Installing Git.md
01 Getting Started/06- Configuring Git.md
01 Getting Started/07- Getting Help.md
01 Getting Started/images/03-01.png
01 Getting Started/images/03-02.png
```

```
01 Getting Started/images/03-03.png
01 Getting Started/images/06-01.png
02 Creating Snapshots/01- Introduction.md
02 Creating Snapshots/02- Initializing a Repository.md
02 Creating Snapshots/03- Git Workflow.md
02 Creating Snapshots/04- Staging Files.md
02 Creating Snapshots/05- Committing Changes.md
02 Creating Snapshots/06- Committing Best Practices.md
02 Creating Snapshots/07- Skipping the Staging Area.md
02 Creating Snapshots/08- Removing Files.md
02 Creating Snapshots/images/03-01.png
README.md
test.txt
```

Even after deleting `text.txt` from the **Working Directory** it still exist in the **Staging Area**.

Use `git add test.txt` to add that file to the staging area, to be committed.

```
❯ git status
On branch main

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    test.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   02 Creating Snapshots/08- Removing Files.md
```

And now `git commit -m "Delete unnecessary file" .`

```
❯ git commit -m "Delete unnecessary file"
[main 632354b] Delete unnecessary file
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 test.txt
```

To remove a file we have to remove it both form the **Working Directory** and the **Staging Area**.

We can perform this operation with a single command `git rm test.txt` # 09- Renaming or Moving Files

When we rename a file and the run `git status` we will see the following output:

```
❯ git status
On branch main

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    02 Creating Snapshots/09- Renaming or Moving.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        02 Creating Snapshots/09- Renaming or Moving Files.md

no changes added to commit (use "git add" and/or "git commit -a")
```

We have to changes and both are unstaged. One that is a delete operation, and a new untracked file.

If we both files to the staging area and run `git status` one more time, we will see the following output:

```
❯ git status
On branch main

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    02 Creating Snapshots/09- Renaming or Moving.md -> 02 Creating S
```

Git recognizes that we renamed the file ank marks it as renamed. Like in the delete operation, there is a Git command to rename files:

Then we commit the changes:

```
❯ git commit -m 'Rename file'
[main aec2762] Rename file
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename 02 Creating Snapshots/{09- Renaming or Moving Files.md => 09- Renaming or Mc
```

```
git mv <current file> <new name>
```

When we use the `git mv` command the changes are applied to both the **Working Directory** and the **Staging Area**.

10- Ignoring Files

In almost every project there are some files we do not want git to track them. Like for example `.log` files.

For example I added a `logs` folder and a `.log` file, in te root directory of the project. `git status` will mark the new folder as untracked.

```
❯ git status
On branch main

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

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

To ignore files we must create a special file called `.gitignore` in the root of the project. This file as no name just as an extension. In that we will add the files or folders we want git to ignore.

We can include as many files or folder we want and use patterns as well like `*.log` to include all log files.

In this case we add the folder `logs/` to `.gitignore` . Now if we run `git status` we will no longer see the `logs/` folder, Instead it marks a new file `.gitignore` .

```
❯ git status
On branch main
Your branch is ahead of 'origin/main' by 14 commits.
  (use "git push" to publish your local commits)

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

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

Now we must add and commit the `.gitignore` to the repository.

This only works if the file or directory to be ignored has not been included in the repository. If by accident we included a file in the repository and only later add it to `.gitignore`, Git is not going to ignore it, because it is already in the repository.

To ignore first we have to remove it from the **Staging Area**, with the `git rm --cached`. So lets suppose we had added the `logs/` folder the repository by accident. We can run:

```
git rm --cached -r logs/
```

And them commit the changes. The `-r` flag is to allow recursive removal.

In GitHub repository gitignore we can see a list of `.gitignore` templates for different programming languages.

11- Short Status

With the `git status` command we see the status of the **Staging Area** and **Working Directory**.

```
❯ git status
On branch main
Your branch is ahead of 'origin/main' by 19 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   02 Creating Snapshots/11- Short Status.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md

no changes added to commit (use "git add" and/or "git commit -a")
```

We can pass the flag `-s` to `git status -s` to have a shorter version.

```
❯ git status -s
 M "02 Creating Snapshots/11- Short Status.md"
?? "02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md"
```

In this output we have two columns, the left column represents the **Staging Area** and the right column the **Working Directory**.

We have modified file `02 Creating Snapshots/11- Short Status.md` , thats why we have a red `M` in the right column, but the left column is empty because we don not have staged this modifications.

File `02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md` is a new file, that why we have red `?` in both columns.

Now if we run `git add "02 Creating Snapshots/11- Short Status.md"` and add this file to the **Staging Area**, the output of `git status -s` will be:

```
❯ gi status -s
M  "02 Creating Snapshots/11- Short Status.md"
?? "02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md"
```

Now The `M` is green and is in the left column, meaning the modifications are in the **Staging Area**.

If we keep modifying the file and run `git status -s` we will se to `M` one green and another red. And if we add the new file to the **Staging Area** we will see a green `A` .

```
❯ git status -s
MM "02 Creating Snapshots/11- Short Status.md"
A  "02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md"
```

12- Viewing Staged and Unstaged Changes

Before committing code it is a best practice to review your code. To do that we can use the `git diff` command.

Comparing the Staging Area

To view changes in files that we have added to the **Staging Area** we run `git diff --staged` . This command will give us the following output:

```
❯ git diff --staged
diff --git a/02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md b/02 C
index 561ffc0..3661360 100644
```

```
--- a/02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md
+++ b/02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md
@@ -1,3 +1,9 @@
 # 12- Viewing Staged and Unstaged Changes

+## git diff command

+Before committing code it is a best practice to review your code. To do that we can
+
+```zsh
+git diff --staged
+```
\ No newline at end of file
```

In the first line we can see that the `diff` utility as called and with which arguments, what files we are comparing. The **`a/...`** is what we have in the last commit and **`b/...`** is what we currently have in the **Staging Area**.

```
diff --git a/02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md b/02 C
```

After that we have a legend, changes in the old copy are marked with a red **`-`**, and changes to the new copy are marked with green **`+`**.

```
--- a/02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md
+++ b/02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md
```

The legend is followed by a header, that tells us what parts of our file have been changed. The changes are split in chunks and there is a header for each chunk.

The **`-1,3`** referes to the old copy. It means starting from line one **`1`** three **`3`** lines have been extracted and shown here.

The **`+1,5`** referes to the new copy. It means starting from line one **`1`** five **`5`** lines have been extracted and shown here.

```
@@ -1,3 +1,9 @@
```

Comparing the Working Directory

To compare what we have in the working directory with what we have in the **Staging Area** we run `git diff`.

The output follows the same concept as `git diff --staged`.

13- Visual Diff Tools

The most popular visual `diff` tools out there are:

- KDiff3
- P4Merge
- WinMerge (Windows only)
- VS Code

To set VS Code as our default diff tool we have to set to Git configurations:

1. `git config --global diff.tool vscode` with this configuration we are giving a name to our default diff tool.
2. `git config --global difftool.vscode.cmd "code --wait --diff $LOCAL $REMOTE"` with this configuration we are telling git how to open VS Code. `code` is the VS code in `PATH` `--wait` to tell the terminal to wait for us to close VS Code, `--diff` We are telling VS Code we are comparing to files, and `$LOCAL $REMOTE` are to placeholders for the old file and new file.

Now we can run `git difftool` or `git difftool --staged` to open VS Code to see changes.

14- Viewing History

Log

We can use the `git log` command to view the commit history. This command produces and output like the following. It is order by the newest to oldest commit.

```
commit 3e3c6c3fced4e94579e7c506e742b480bd2c682c (HEAD -> main, origin/main)
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Sun Feb 28 17:39:31 2021 -0300
```

```
    Start lesson

commit 529f5c183f9084caa3f5eafd2435f67094888a94
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Sun Feb 28 17:13:15 2021 -0300

    Added new lessons
```

Each commit ad a unique identifier, and hexadecial, 40 character that git generates automatically.

```
commit 3e3c6c3fced4e94579e7c506e742b480bd2c682c
```

Next to the first commit we have **`(HEAD -> main)`**:

- `HEAD` is a reference to the current branch.
- `main` beeing the current branch.

Log one line

With the `git log --oneline` we can see a shorter version of the `log` command.

```
3e3c6c3 (HEAD -> main, origin/main) Start lesson
529f5c1 Added new lessons
da5f08b Lessons start
12156df Add details to lesson
0ed2ae3 Lesson completed
ebdd235 Add gitignore
4dec09d Add more details to lesson
```

Reverse history

The `--reverse` flag reverts the log display history. It can be applied both to the log command and to the log one line command.

- `git log --reverse`
- `git log --oneline --reverse`

15- Viewing a Commit

Show commit

To view what was changed in a commit we can use the `git show` command. We have to pass teh commit as an argument. There are two ways to reference a commit:

1. By the unique identifier, for example `git show 3e3c6c3`. We don't have to type all the characters, we can type fewer character as long they are unique.
2. Another way is using the `HEAD` pointer. `HEAD` is in front of the last commit, so we can type how many steps we want to go back `git show HEAD~2`, for example 2.

This will produce a similar output to `git diff`.

```
commit da5f08b76c559362400177e88d3a01b6d7511531
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Sun Feb 28 16:52:55 2021 -0300

    Lessons start

diff --git a/02 Creating Snapshots/11- Short Status.md b/02 Creating Snapshots/11- S
new file mode 100644
index 0000000..3b350e3
--- /dev/null
+++ b/02 Creating Snapshots/11- Short Status.md
@@ -0,0 +1,2 @@
+# 11- Short Status
+
```

Show commit single file

If we want to see the exact version of a file saved in commit instead of seeing the differences we can use `git show HEAD~1:README.md`. So the `git show` command followed by `:` and the path to a file, in this example `README.md`, or another example, `git show HEAD~1:"02 CreatingSnapshots/12- Viewing Staged and Unstaged Changes.md"`.

Show commit all files and directories

If we want to see all the files and directories in a commit we use `ls-tree`. A **tree** is a data structure for representing hierarchical information. These trees can have nodes and the nodes can have children. If we use `git ls-tree HEAD~1`, we will have an output like this:

```
❯ git ls-tree HEAD~1
100644 blob 333c1e910a3e2bef1b9d0d4587392627d8388974    .gitignore
040000 tree 4c24e363fb92a146d90b33f5e6be484eda876cb1    01 Getting Started
040000 tree fedd8b67c40b90365ca1b2fc1357b15a56b8c9b6    02 Creating Snapshots
100644 blob abefdb35d01f55ffdabca93f53748b84f9d10e14    README.md
```

Files are represented using `blob` and directories are represented by `tree` . All of theses are object saved in git database. The `333c1e910a3e2bef1b9d0d4587392627d8388974` is a unique identifier of the corresponding file. We can use this identifier to vew the content of the file.

We can use `git show 333c1e` to view the content of the `.gitignore` , in that commit.

If we run this command on a `tree` like for instance `git show 4c24` we will get:

```
tree 4c24

01- Introduction.md
02- How to Take This Course.md
03- What is Git.md
04- Using Git.md
05- Installing Git.md
06- Configuring Git.md
07- Getting Help.md
images/
```

Git objects

Using `git show` we can view **Git Objects**, these objects can be:

- Commits
- Blobs (files)
- Tree (directories)
- Tags

16- Unstaging Files

When we want to unstage a file (remove the file from the **Staging Area**), in other word undo the `git add` command, we use the `git restore --staged` command. We can pass to it as arguments a specific file or multiple files, with a `.` (dot) for all files, or patterns like for example `*.txt` for all text files.

In this example I have the `02 Creating Snapshots/16- Unstaging Files.md` file changes in the **Staging Area**. We can see that with `git status` the file is marked as modified.

```
❯ git status
On branch main

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   02 Creating Snapshots/16- Unstaging Files.md
```

Running `git restore --staged "02 Creating Snapshots/16- Unstaging Files.md"` will remove this changes from the **Staging Area**.

When we run `git status` again, we can see the changes are not staged for commit.

```
❯ git status
On branch main

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   02 Creating Snapshots/16- Unstaging Files.md

no changes added to commit (use "git add" and/or "git commit -a")
```

When we run the `git restore --staged` git removes that file from the **Staging Area** and places there the copy from the last commit with that file.

17- Discarding Local Changes

When we want to discard local changes, in other others, changes in the **Working Directory**, we use the `git restore` , without the `--staged` flag.

When we run this command, Git is going to take a copy from the next environment in this case the **Staging Area**, and copy it in the **Working Directory**.

We can pass to it as arguments a specific file or multiple files, with a `.` (dot) for all files, or patterns like for example `*.txt` for all text files.

In case of new files (untracked files), Git does not change anything, because it does not know where to get a previous version of this file, it does not exist in the **Staging Area** or **Repository**. To remove untracked files we can use `git clean` with the `-fd` flags, `-f` for force and `-d` for whole directories.

If we run this command without the flags on untracked files we will get a fat error:

```
fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given; refusin
```

This is a way of git ro warns us that this can not be undone.

18- Restoring a File to an Earlier Version

Once git tracks a file it stores every version of that file in the database. With Git there are two way to restore a earlier version of file:

1. Restore a file to previous version
2. Undoing a commit

In this lesson we are going to look at restoring a file. For example if we delete a file by accident.

Using the `git rm` command I have deleted a file. And ist now marked as `deleted` in the **Staging Area**. We can see ti with `git status`

```
❯ git status
On branch main

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    02 Creating Snapshots/17- Discarding Local Changes.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        02 Creating Snapshots/18- Restoring a File to an Earlier Version.md
```

After committing `git commit -m 'delete file'`, and fi we do a `git log --oneline`

```
b02e494 (HEAD -> main) delete file
cb02fd6 Lesson complete
9b67da8 Lesson start
001ce95 Lesson complete
0746554 Start lesson
[...]
```

We can recover that file with `git restore` . By default git will restore a file from the next environment, so if the file we want to restore is in the **Working Directory**, Git will restore it from the **Staging Area**, and if it is in the **Staging Area** Git will restore it from the repository, or last commit.

In this case with `git restore --source=HEAD~1 "02 Creating Snapshots/17- Discarding Local Changes.md"` , we override the default behavior and restore the last commit that is when we deleted the file. With `git status` we will see the recovered file marked as untracked.

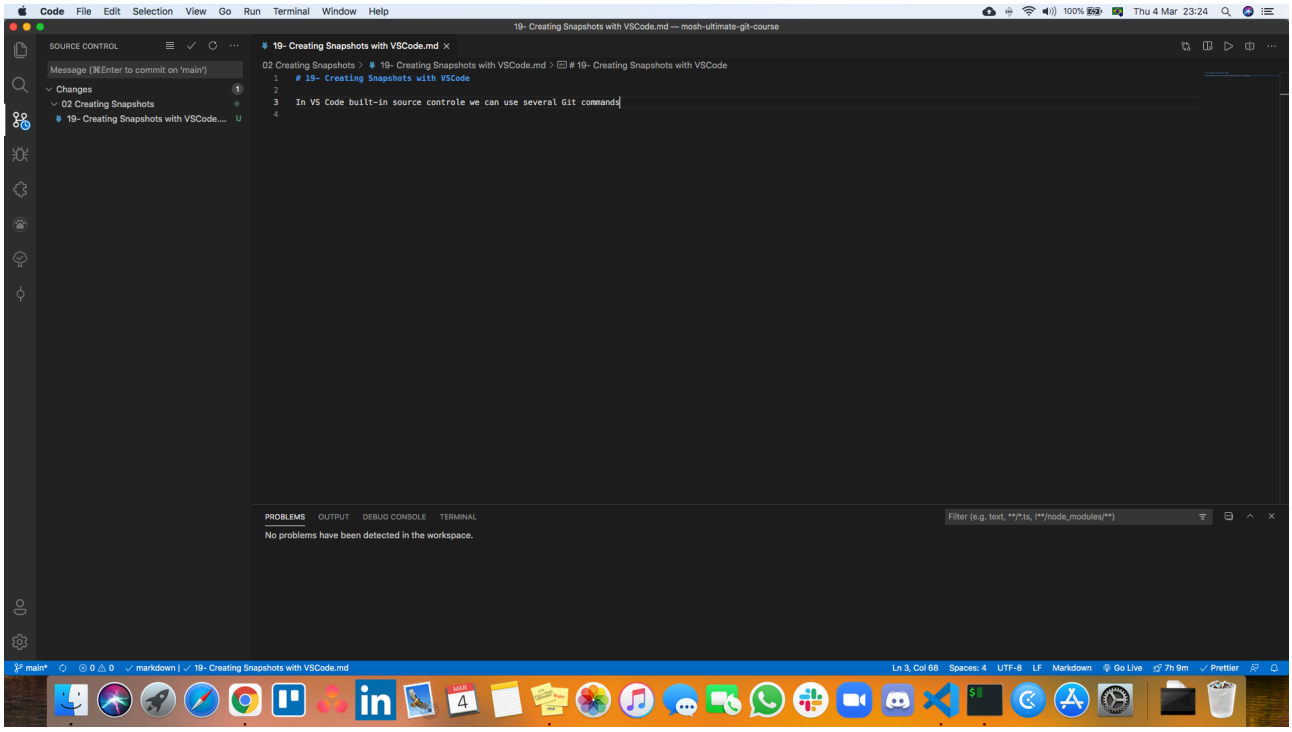
```
❯ git status
On branch main

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        02 Creating Snapshots/17- Discarding Local Changes.md
        02 Creating Snapshots/18- Restoring a File to an Earlier Version.md

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

19- Creating Snapshots with VSCode

In VS Code built-in source controle we can use several Git commands.



The view in the left panel is similar to the `git status` output.

If we hover the mouse hover a file we will see a `+` sign. That will add files to the **Staging Area**. If that file is already in the **Staging Area** a `-` sig will be displayed, to remove the file from the **Staging Area**.

combined_readme.md

01- Introduction

In this section we will look into different ways to browse a project history.

1. Search for commits by author, date, message, etc
2. View a commit
3. Restore project to an earlier point
4. Compare commits
5. view the history of a file
6. Find a bad commit that introduced a bug

03- Viewing the History

We are going to view in more detail the `git log` command. Wit this command, as we know we can see in more detail the commit history. It gives us teh following output.

```
commit b9e25dff4873a1db8cd615784a485d4ab7854e14 (HEAD -> main, origin/main)
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Thu Mar 4 23:45:40 2021 -0300

    start new lesson

commit 9f618634b2592513a9da3da313f554abec2e18cc
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Thu Mar 4 23:42:01 2021 -0300

    style: change tab to spaces

commit bf77b4e007ecaa30da382ff80e7afdc1fb1f6fc9
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Thu Mar 4 23:35:58 2021 -0300

    lesson completed
```

Given we can have multiple pages of commits pressing `space` will take us to the nex page, we can also use the up and down arrows to navigate, and `q` to exit.

Apart fom the `--oneline` options that show a summary of the commits history, que can use other options.

Option --stat

With the `--stat` options we can se the list of file that have been changed in each commit. The `git log --stat` will give the following output.

```
commit b9e25dff4873a1db8cd615784a485d4ab7854e14 (HEAD -> main, origin/main)
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Thu Mar 4 23:45:40 2021 -0300

    start new lesson

 03 Browsing History (44m)/01- Introduction.md | 1 +
 1 file changed, 1 insertion(+)

commit 9f618634b2592513a9da3da313f554abec2e18cc
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Thu Mar 4 23:42:01 2021 -0300

    style: change tab to spaces

 02 Creating Snapshots/15- Viewing a Commit.md | 8 ++++----
 1 file changed, 4 insertions(+), 4 deletions(-)

commit bf77b4e007ecaa30da382ff80e7afdc1fb1f6fc9
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Thu Mar 4 23:35:58 2021 -0300

    lesson completed

 02 Creating Snapshots/19- Creating Snapshots with VSCode.md     |   9 +++++++++
 02 Creating Snapshots/20- Creating Snapshots with GitKraken.md |   0
 02 Creating Snapshots/images/19-01.png                         | Bin 0 -> 375202 by
 3 files changed, 9 insertions(+)
```

Or combined with the `--online` option `git log --oneline --stat` .

```
b9e25df (HEAD -> main, origin/main) start new lesson
 03 Browsing History (44m)/01- Introduction.md | 1 +
```

```
 1 file changed, 1 insertion(+)
9f61863 style: change tab to spaces
 02 Creating Snapshots/15- Viewing a Commit.md | 8 ++++----
 1 file changed, 4 insertions(+), 4 deletions(-)
bf77b4e lesson completed
 02 Creating Snapshots/19- Creating Snapshots with VSCode.md     |   9 +++++++++
 02 Creating Snapshots/20- Creating Snapshots with GitKraken.md |   0
 02 Creating Snapshots/images/19-01.png                         | Bin 0 -> 375202 by
 3 files changed, 9 insertions(+)
```

Option --patch

With the `--path` option we can see the actual changes in each commit. The `git log --online --patch` wil give the following ouput.

```
bf77b4e lesson completed
diff --git a/02 Creating Snapshots/19- Creating Snapshots with VSCode.md b/02 Creati
new file mode 100644
index 0000000..bac12c2
--- /dev/null
+++ b/02 Creating Snapshots/19- Creating Snapshots with VSCode.md
@@ -0,0 +1,9 @@
+# 19- Creating Snapshots with VSCode
+
+In VS Code built-in source controle we can use several Git commands.
+
+![VS Code Source Controle](./images/19-01.png "VS Code Source Controle")
+
+The view in the left panel is similar to the `git status` output.
+
+If we hover the mouse hover a file we will see a **`+`** sign. That will add files
diff --git a/02 Creating Snapshots/20- Creating Snapshots with GitKraken.md b/02 Cre
new file mode 100644
index 0000000..e69de29
diff --git a/02 Creating Snapshots/images/19-01.png b/02 Creating Snapshots/images/1
new file mode 100644
index 0000000..292335c
Binary files /dev/null and b/02 Creating Snapshots/images/19-01.png differ
```

04- Filtering the History

In a project with a long history we can have hundreds or even thousands of commits. We use we can filter commits, so we do not have to view the al history.

Filter by last n commits `-number`

We can pass a number to the `git log` command. For example with `git log -3` we can view the last 3 commits. The following output is from the command `git log --oneline -3`.

```
4acfee7 (HEAD -> main, origin/main) lesson complete
dd6b4b2 fix: typo
b9e25df start new lesson
```

Filter by author `--author=`

With the option `--author=` we can filter commits by author, `git log --author="Miguel"`.

Filter by date `--before` and or `--after`

Wit he option `--before` and or `--after` we can filter the commit history by date, we do not have to pass both options. For example `git log --oneline --after="2021-03-06"`. It is also possible to specify relative dates, like `git log --oneline --after="yesterday"`, `git log --oneline --after="one week ago"`, `git log --oneline --after="two day ago"`.

Filter by message `--grep`

We can filter commits by keywords in their message with the `--grep` option. For example `git log --online --grep="typo"`. This will return all the commits tha have the word `typo` in their message. This command is case sensitive.

Filter by content `-S`

We can filter commits by content of the file with the `-S` option. For instance if we need to find a commit that added or remove a certain function declaration,for example the function `hello()` we use `git log -S"hello()"`. Notice that we do not pass an `=` sign to the `-S` option. Here with the command `git log --oneline -S"VS Code"`, we can see all the commits that have the `VS Code` in the file content.

```
4acfee7 (HEAD -> main, origin/main) lesson complete
bf77b4e lesson completed
15478b9 Lesson completed
f289d8a First commit
```

We could add the `--patch` option to see the actual changes made, `git log --oneline -S"VS Code" --patch` .

Filter by commit range

It is also possible to filter a range of commits, using the commit hash `git log 15478b9..b02e494` .

Filter by file

We can filter commits that applied changes to a particular file or several files, just by passing the name of the file, with the relative path at the end of the command, like so `git log -- "02 Creating Snapshots/16- Unstaging Files.md"` . The `--` double hyphens separating the command from the file is optional, but if we do not pass it, depending on the file name Git can throw an error of ambiguity. The file name should be the last option given to the command.# 05- Formatting the Log Output

Option `--pretty=format:`

We can personalize the output of the log command with the option `git log --pretty:format:` . We must pass it a format string that where we can use plain text combined with placeholder that will be replaced by Git with information. In Git PRETTY FORMATS we can see a list of all the place holders. We can also pass colors to the output

The following command (courtesy of Le Wagon bootcamp setup).

```
git log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
```

Will give teh following output:

```
* 4acfee7 - (HEAD -> main, origin/main) lesson complete (57 minutes ago) <Miguel Pim
* dd6b4b2 - fix: typo (57 minutes ago) <Miguel Pimenta>
* b9e25df - start new lesson (3 days ago) <Miguel Pimenta>
* 9f61863 - style: change tab to spaces (3 days ago) <Miguel Pimenta>
```

```
* bf77b4e - lesson completed (3 days ago) <Miguel Pimenta>
* 0aa8fd5 - restore deleted file (3 days ago) <Miguel Pimenta>
* 40d94ee - deleted file (3 days ago) <Miguel Pimenta>
* 2d077eb - lessons complete (3 days ago) <Miguel Pimenta>
* b02e494 - delete file (3 days ago) <Miguel Pimenta>
* cb02fd6 - Lesson complete (5 days ago) <Miguel Pimenta>
* 9b67da8 - Lesson start (5 days ago) <Miguel Pimenta>
```

06- Aliases

We can set aliases from frequently used commands so we do not have to type them in the long form. That is done the config property `alias`, like so.

```
git config --global alias.l "log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)
```

We we are setting the `l` as an alias of `log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit`.

So ater setting that property we can use `git l`.

07- Viewing a Commit

As we seen before we can use `git show` to the infos about a specific commit. Section: Creating Snapshot - Lesson: 15- Viewing a Commit

Final version of a file

We can see the final version of a file in a particular commit with the `git show HEAD~n:<path to the file>`. For example `git show HEAD~2:"02 Creating Snapshots/12- Viewing Staged and Unstaged Changes.md`. In here are 2 commits before `HEAD` which is the last commit.

Files changed in a commit `--name-only`

To view the files that were changed in a given commit we use `git show HEAD~4 --name-only`, The output is something like:

```
commit bf77b4e007ecaa30da382ff80e7afdc1fb1f6fc9
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Thu Mar 4 23:35:58 2021 -0300

    lesson completed

02 Creating Snapshots/19- Creating Snapshots with VSCode.md
02 Creating Snapshots/20- Creating Snapshots with GitKraken.md
02 Creating Snapshots/images/19-01.png
```

Files changed in a commit `--name-status`

Using the option `--name-status`, we have a similar output than above but wit information about the file, if it was added, modified, deleted or renamed.

```
commit bf77b4e007ecaa30da382ff80e7afdc1fb1f6fc9
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Thu Mar 4 23:35:58 2021 -0300

    lesson completed

A       02 Creating Snapshots/19- Creating Snapshots with VSCode.md
A       02 Creating Snapshots/20- Creating Snapshots with GitKraken.md
A       02 Creating Snapshots/images/19-01.png
```

08- Viewing the Changes Across Commits

To see what has been changed across a range of commits we use the `diff` command. For example `gif diff HEAD~2 HEAD`, will return all the changes from the last two commits `HEAD~2` until the most recent commit `HEAD`.

We can add a particular file to that command to only see the changes made to that file, like `git diff HEAD~2 HEAD "03 Browsing History (44m)/06- Aliases.md"`

Like with the `log` command, we can pass `--name-only` and `--name-status` here to see the list of file that have been changed.

09- Checking Out a Commit

When we need to see the complete project in a given point in time we can check out a given commit, and it will restore our **Working Directory** to that point in time. Using the command `git checkout <commit>`. For example `git checkout 12156df`.

This will bring a warning:

```
Note: switching to 'b9e25df'.

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:

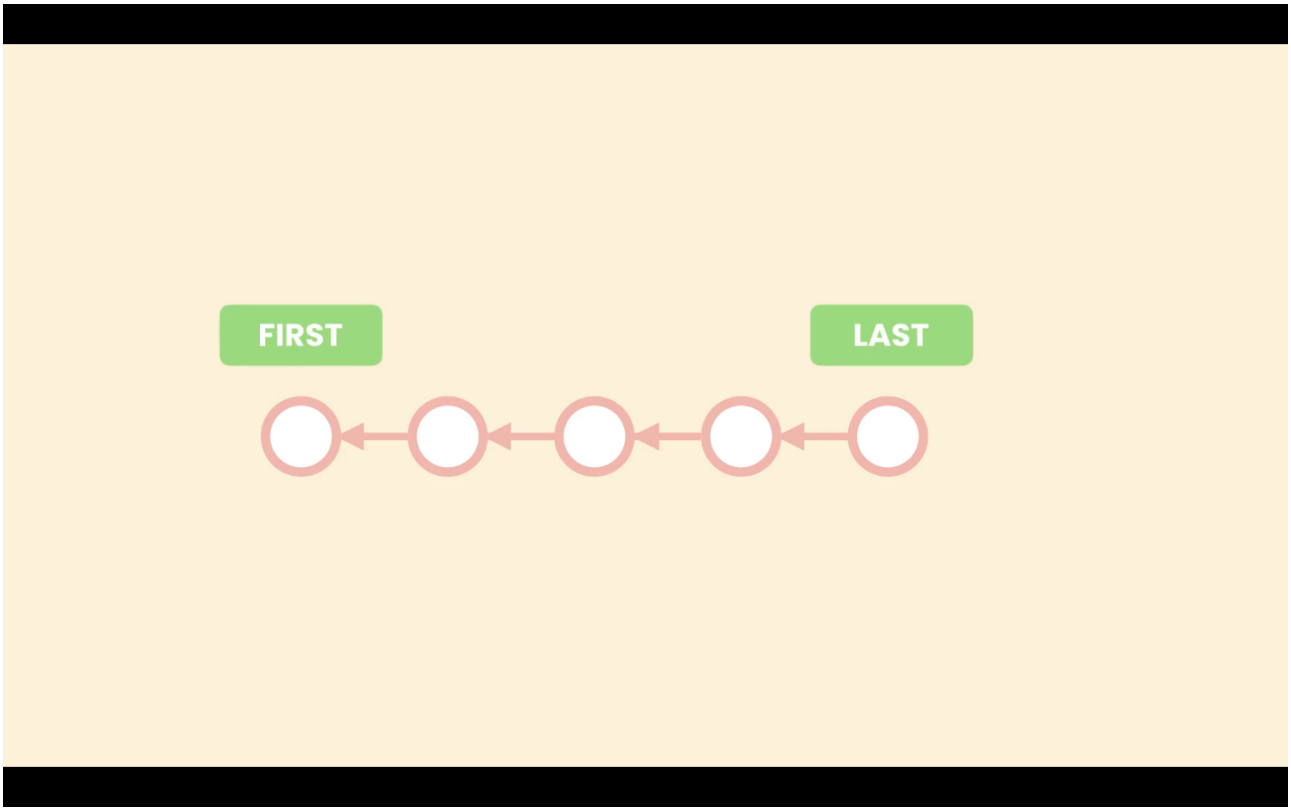
  git switch -

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

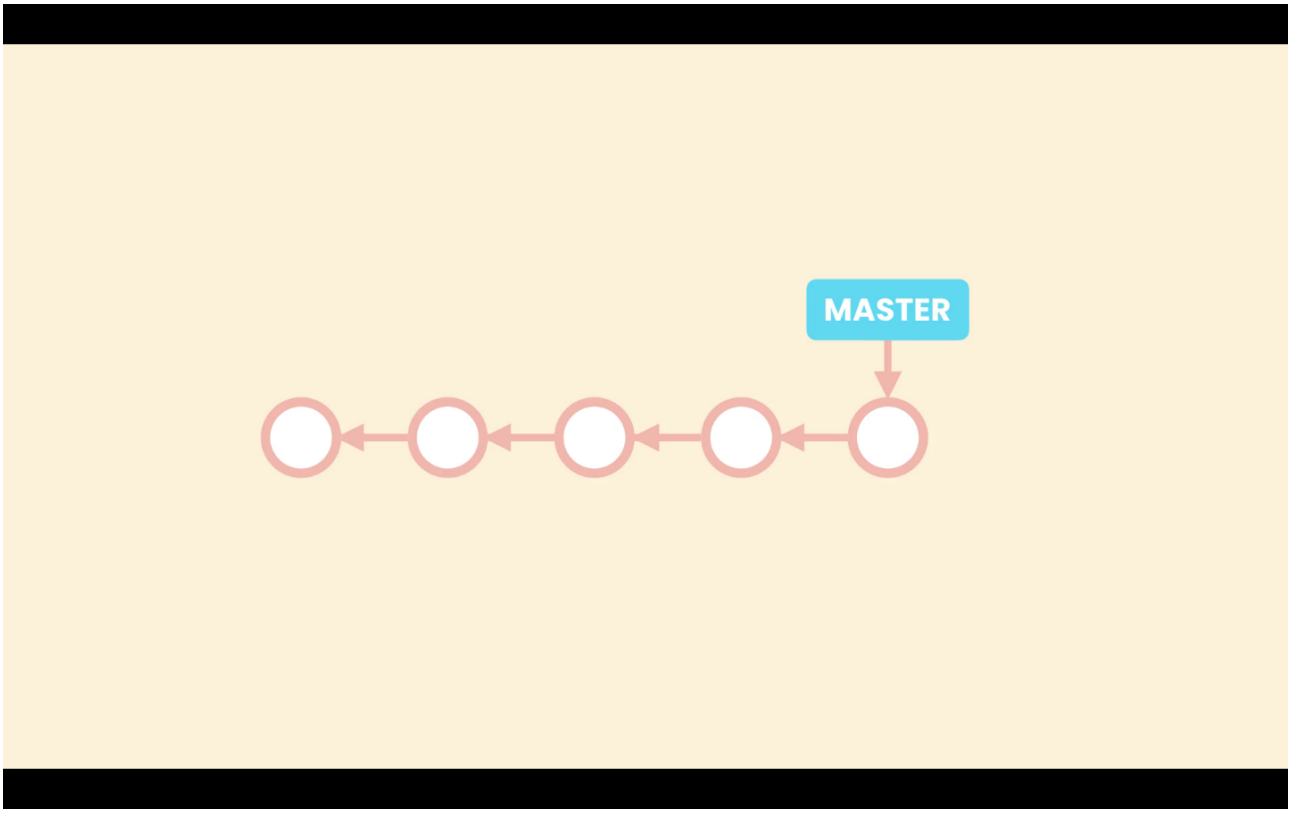
HEAD is now at b9e25df start new lesson
```

Detached HEAD state

In Git each commit is pointing to the last commit. That is how Git maintains history.



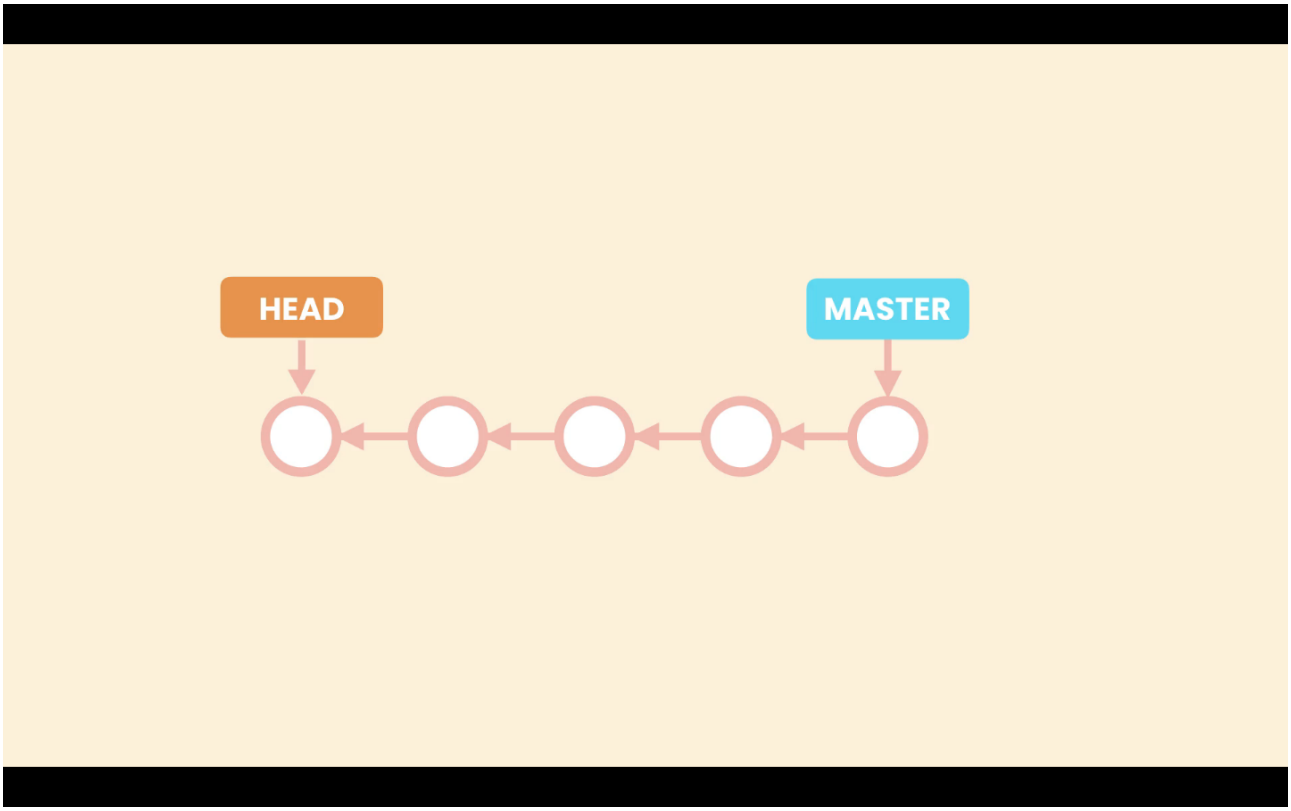
Until now all the commits created are part of a branch, the `master` or `main`. The way Git represents branches is using a pointer, so `master` is pointing to the last commit created. As we create new commits master moves forward to point to the last commit.



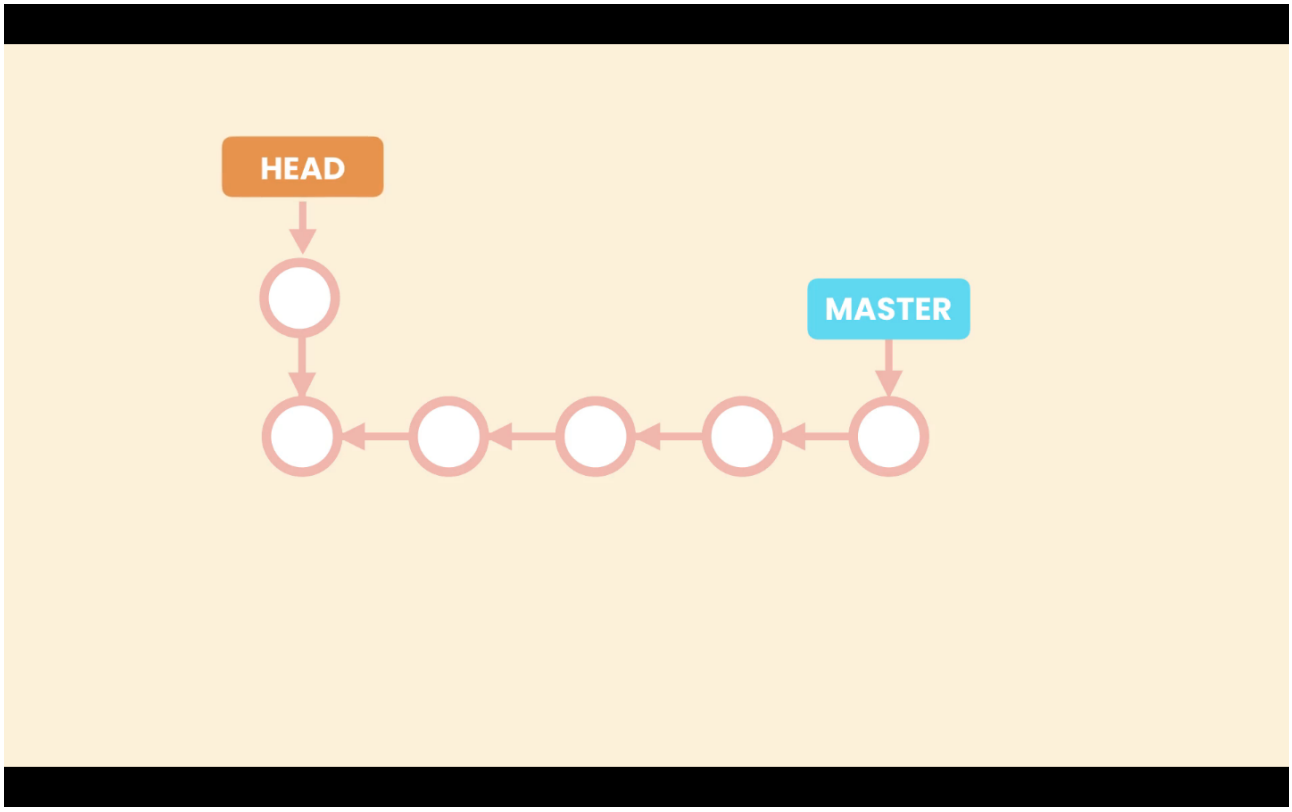
Because we can have multiple branches Git needs to know in which branch we are working at the moment. To do that Git uses another special pointer called `HEAD`, so `HEAD` points to the current branch we are working on. In this case `master`. We have seen this in the `git log` command **`(HEAD -> main)`**.

```
2ee3bb6 (HEAD -> main) add details to lesson
18869aa lesson complete
f2150f7 lesson complete
4acfee7 lesson complete
```

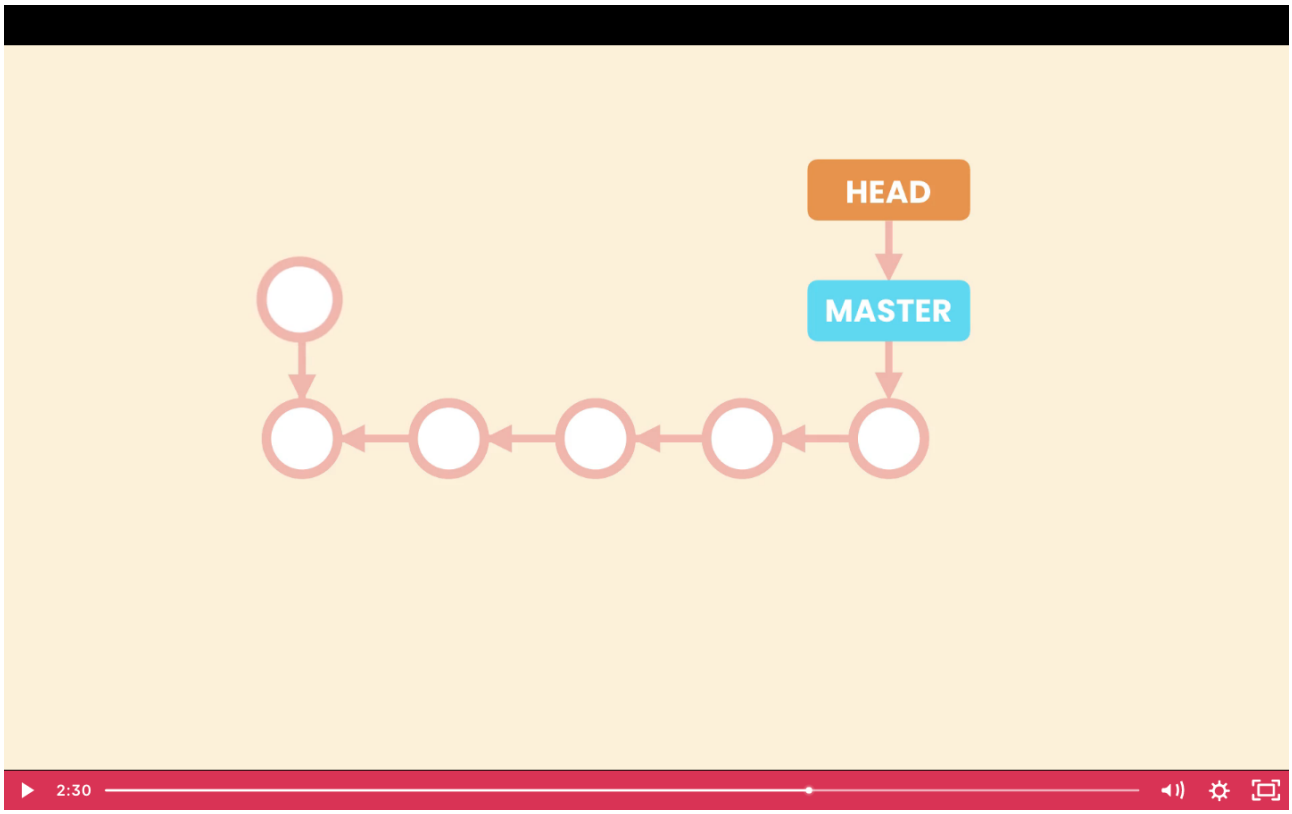
When we checkout a particular commit, the `HEAD` pointer will move to that commit, thats is what is called a `detached HEAD`. `HEAD` is not pointing to a branch but is pointing to a specific commit.



In this situation we should not create new commits. If we make changes and create a new commit, that commit will be added where the `HEAD` pointer is at the moment. But eventual we will attach the HEAD pointer to a branch, so the commit created it is not reachable by any other commit or pointer, ist like a dead commit. Git checks for commits like these periodically and removes them to save space.



Lost commit.



If we run `git log --oneline --all` command while in a detached `HEAD` state we can see the `master` (main in my case) pointing to the last commit and `HEAD` point to the specific commit. If we use `git log --oneline` without the the `--all` flag, we will not see the commits after made after the commit `HEAD` is pointing to.

```
ebbd4b1 (main) add details to lesson
66cf5ab add details to lesson
2ee3bb6 add details to lesson
18869aa lesson complete
f2150f7 (HEAD) lesson complete
4acfee7 lesson complete
dd6b4b2 fix: typo
b9e25df start new lesson
9f61863 style: change tab to spaces
[...]
```

To attache the `HEAD` pointer to the branch, use `git checkout <name of the branch>` in this case `git checkout main` .

10- Finding Bugs Using Bisect

Git provides a great tool to find bugs quickly **Bisect**.

Image that we have a bug in an application, but we do not know where the bug was introduced. Using the **Bisect** to we can narrow our search.

First we have to tell it that the current state, being the last commit, is a bad commit. And them we have to give it a good commit, as teh good state.

Running `git log --oneline` , let say that the good state or good commit is `9f61863` . at that point in time the application was ok.

```
ebbd4b1 (HEAD -> main) add details to lesson
66cf5ab (origin/main) add details to lesson
2ee3bb6 add details to lesson
18869aa lesson complete
f2150f7 lesson complete
4acfee7 lesson complete
dd6b4b2 fix: typo
b9e25df start new lesson
9f61863 style: change tab to spaces
bf77b4e lesson completed
[...]
```

So first we run `git bisect start`, this will initialize the the **`bisect`** operation.

Then we tell it the bad commit, witch is the current one, run `git bisect bad`.

Then we give it a good commit run `git bisect good 9f61863`. This will give the following output:

```
❯ git bisect good 9f61863
Bisecting: 4 revisions left to test after this (roughly 2 steps)
[18869aaa853d0e3b2383a9104184f05eadd61722] lesson complete
```

If we run the `git log --oneline --all`. We can see that the `HEAD` is detached. So Git as made a checkout to the middle of the history, between the bad and good commit we gave `bisect`. So our **Working Directory** will be restored to that point in time.

```
54b967c (main, refs/bisect/bad) add details to lesson
9325623 add details to lesson
ebbd4b1 add details to lesson
66cf5ab add details to lesson
2ee3bb6 add details to lesson
18869aa (HEAD) lesson complete
f2150f7 lesson complete
4acfee7 lesson complete
dd6b4b2 fix: typo
b9e25df start new lesson
9f61863 (refs/bisect/good-9f618634b2592513a9da3da313f554abec2e18cc) style: change ta
bf77b4e lesson completed
0aa8fd5 restore deleted file
40d94ee deleted file
2d077eb lessons complete
```

At this point we can run our application and automated tests to see if the problem is still there. If the bug persists it means that it is somewhere between commit `18869aa` and commit `b9e25df`. If the bug is gone it means it was introduced in the other half.

Suppose this was a good commit, so we run `git bisect good`. Meaning the bug is in the upper half. Between commit `2ee3bb6` and the latest commit. This will have the following output:

```
❯ git bisect good
Bisecting: 2 revisions left to test after this (roughly 1 step)
[66cf5ab9240156e59aca2923bc0d73a6583af68d] add details to lesson
```

So now running `git log --oneline --all`, Git as moved the `HEAD` to the middle of the half that should have the problem.

```
54b967c (origin/main, main, refs/bisect/bad) add details to lesson
9325623 add details to lesson
ebbd4b1 add details to lesson
66cf5ab (HEAD) add details to lesson
2ee3bb6 add details to lesson
18869aa (refs/bisect/good-18869aaa853d0e3b2383a9104184f05eadd61722) lesson complete
f2150f7 lesson complete
4acfee7 lesson complete
dd6b4b2 fix: typo
b9e25df start new lesson
9f61863 (refs/bisect/good-9f618634b2592513a9da3da313f554abec2e18cc) style: change ta
bf77b4e lesson completed
0aa8fd5 restore deleted file
40d94ee deleted file
2d077eb lessons complete
```

Now `HEAD` in the `66cf5ab` commit, lets image we run our test and the bug is still there, so this is a bad commit, we run `git bisect bad`. The out put is:

```
❯ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[2ee3bb6004b4f8e681693b35f81df16ae425fa10] add details to lesson
```

With `git log --oneline --all`, we see now that we have one commit marked as **`(refs/bisect/bad)`**, and Git as moved `HEAD` to the `2ee3bb6` commit. So the bug must be in commit `66cf5ab` or `2ee3bb6`.

```
54b967c (origin/main, main) add details to lesson
9325623 add details to lesson
ebbd4b1 add details to lesson
66cf5ab (refs/bisect/bad) add details to lesson
2ee3bb6 (HEAD) add details to lesson
18869aa (refs/bisect/good-18869aaa853d0e3b2383a9104184f05eadd61722) lesson complete
f2150f7 lesson complete
4acfee7 lesson complete
dd6b4b2 fix: typo
b9e25df start new lesson
9f61863 (refs/bisect/good-9f618634b2592513a9da3da313f554abec2e18cc) style: change ta
bf77b4e lesson completed
0aa8fd5 restore deleted file
```

```
40d94ee deleted file
2d077eb lessons complete
```

Let's suppose the bug is still there, meaning the bug was introduced in commit `2ee3bb6`. So we run `git bisect bad`. Git will output the info about that commit like the following example:

```
❯ git bisect bad
2ee3bb6004b4f8e681693b35f81df16ae425fa10 is the first bad commit
commit 2ee3bb6004b4f8e681693b35f81df16ae425fa10
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Sun Mar 7 15:37:14 2021 -0300

    add details to lesson

 03 Browsing History (44m)/09- Checking Out a Commit.md | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

After we are done we have to attach the `HEAD` pointer to the branch with the command `git bisect reset`.

With `git bisect` we can split our history in half, to see various commit, and find the commit that introduced a problem.

11- Finding Contributors Using Shortlog

If we need to see everyone that made contributions to our project we can use the `git shortlog` command. This will output the following:

```
Fernando Moraes (14):
      html do index/show
      Merge branch 'main' of github.com:jmschp/kids-time into main
      index e show html
 [...]
      bttns
      atualicazao seed
      bttns com link, pagina activitys

Miguel Pimenta (135):
      Initial commit with minimal template from https://github.com/lewagon/rails-tem
      Initialized rails app
      Added and configured Devise gem
```

```
      Configured Cloudinary
      Added Models Activity Order, and added columns to users
  [...]
```

The contributor, the number of commits and the commits messages.

The command accepts various options, for example `git shortlog -n` will sort output according to the number of commits per author.

The command `git shortlog -s` , suppress commit descriptions, only provides commit count and authors.

We can also filter using `--before=` and `--after=` , to view teh contributor in a date range.

12- Viewing the History of a File

To view a history of a particular file we run `git log <file>`

13- Restoring a Deleting File

If we want to recover a deleted file we can do it by checkout the previous commit, of the one that deleted the file.

For example I have run `git rm "03 Browsing History (44m)/10- Finding Bugs Using Bisect.md"` and delete this file. Wih `git log --oneline` , we can see that the file was deleted in the last commit.

```
dacc1ec (HEAD -> main) delete file
33bceb5 start new lesson
1ce178a lesson complete
b5f716e lesson complete
54b967c add details to lesson
```

Now to recover that file we checkout the previous commit (or teh parent commit), in this case `33bceb5` , passing the nane file.

```
git checkout 33bceb5 -- "03 Browsing History (44m)/10- Finding Bugs Using Bisect.md"
```

After running this command we can see that we have in the **Staging Area** one new file with `git status .`

```
❯ git status
On branch main

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   03 Browsing History (44m)/10- Finding Bugs Using Bisect.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        03 Browsing History (44m)/13- Restoring a Deleting File.md
```

14- Finding the Author of Line Using Blame

Git Blame is a command to find the author of a crappy line of code. Run `git blame <file>`

```
git blame "03 Browsing History (44m)/13- Restoring a Deleting File.md"
```

This command will give an output like the following:

```
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300  1) # 13- Restoring a Deleting Fi
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300  2)
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300  3) If we want to recover a delet
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300  4)
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300  5) For example I have run `git r
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300  6)
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300  7) ```zsh
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300  8) dacc1ec (HEAD -> main) delete
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300  9) 33bceb5 start new lesson
```

For each line of code in the file we can see

1. The commit ---> b89cef85
2. The author ---> Miguel Pimenta
3. Date & Time ---> 2021-03-08 23:30:10 -0300
4. Line number ---> 1
5. And the code it self.

Git blame with email `-e`

We can use other option with `git blame`, for example `-e` will also return the email of the author.

```
git blame -e "03 Browsing History (44m)/13- Restoring a Deleting File.md"
```

```
b89cef85 (<jmiguelpimenta@gmail.com> 2021-03-08 23:30:10 -0300  1) # 13- Restoring a
b89cef85 (<jmiguelpimenta@gmail.com> 2021-03-08 23:30:10 -0300  2)
b89cef85 (<jmiguelpimenta@gmail.com> 2021-03-08 23:30:10 -0300  3) If we want to rec
b89cef85 (<jmiguelpimenta@gmail.com> 2021-03-08 23:30:10 -0300  4)
b89cef85 (<jmiguelpimenta@gmail.com> 2021-03-08 23:30:10 -0300  5) For example I hav
```

Git blame filter by lines `-L`

We can filter the lines we want using the `-L` flag and we have to pass to it the start line and end line separated by a comma.

```
git blame -L 1,3 "03 Browsing History (44m)/13- Restoring a Deleting File.md"
```

The above example will return the first three lines of code.

```
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300 1) # 13- Restoring a Deleting Fil
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300 2)
b89cef85 (Miguel Pimenta 2021-03-08 23:30:10 -0300 3) If we want to recover a delete
```

15- Tagging

Tags are use to mark a certain point in the project. Most of the times they are used to mark release version, like v1.0.

There are two type of tags lightweight, and annotated tag. The lightweight as only the tag name, and a annotated tag has more properties, like tagger name, email and message. Because a annotated tag can have more info they are preferes to lightweight.

Tagging in the last commit

If we want to tag the last commit run `git tag <name of the tag>`

```
git tag v1.0
```

Tagging an earlier commit

To tag an earlier commit we have to pass the commit after the tag like `git tag <name of teh tag> <referent to commit>` .

```
git tag v1.0 b4ef25f
```

With `git log --oneline` we can see the tag.

```
b89cef8 (HEAD -> main) lesson complete
b4ef25f (tag: v1.0) restore file
dacc1ec delete file
33bceb5 start new lesson
1ce178a lesson complete
b5f716e lesson complete
```

Referente a commit using a tag

It is also possible to reference a commit by it tag.

```
git checkout v1.0
```

See all the tags

To see all the tag in the project run:

```
git tag
```

Annotated tag `-a`

To create a annotated tag we use the option `-a`, followed by the tag name and then `-m` and a message.

```
git tag -a v1.2 -m 'Release version 1.1'
```

Tags messages `-n`

To view the tags and messages use the `-n` option.

```
git tag -n
```

This will ouput something like the following example.

```
v1.0            restore file
v1.2            Release version 1.2
```

The lightweight tag is associated with the commit message that it point to. And the annotated tag has a custom message.

Show Tag

Run `git show <tag name>` to view all the infos of that tag. Similar to `git show <commit>`. If the command is ran on an annotated tag besides the commit info we also have the tag info.

```
tag v1.2
Tagger: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Tue Mar 9 00:00:03 2021 -0300

Release version 1.1

commit b89cef852dde2c83d81609d981c35ce6f8b7fba0 (HEAD -> main, tag: v1.2)
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Mon Mar 8 23:30:10 2021 -0300

    lesson complete
```

Delete a tag `-d`

To delete a tag use the `-d` option.

```
git tag -d v1.0
```

This returns the deleted tag like so:

```
Deleted tag 'v1.0' (was b4ef25f)
```

combined_readme.md

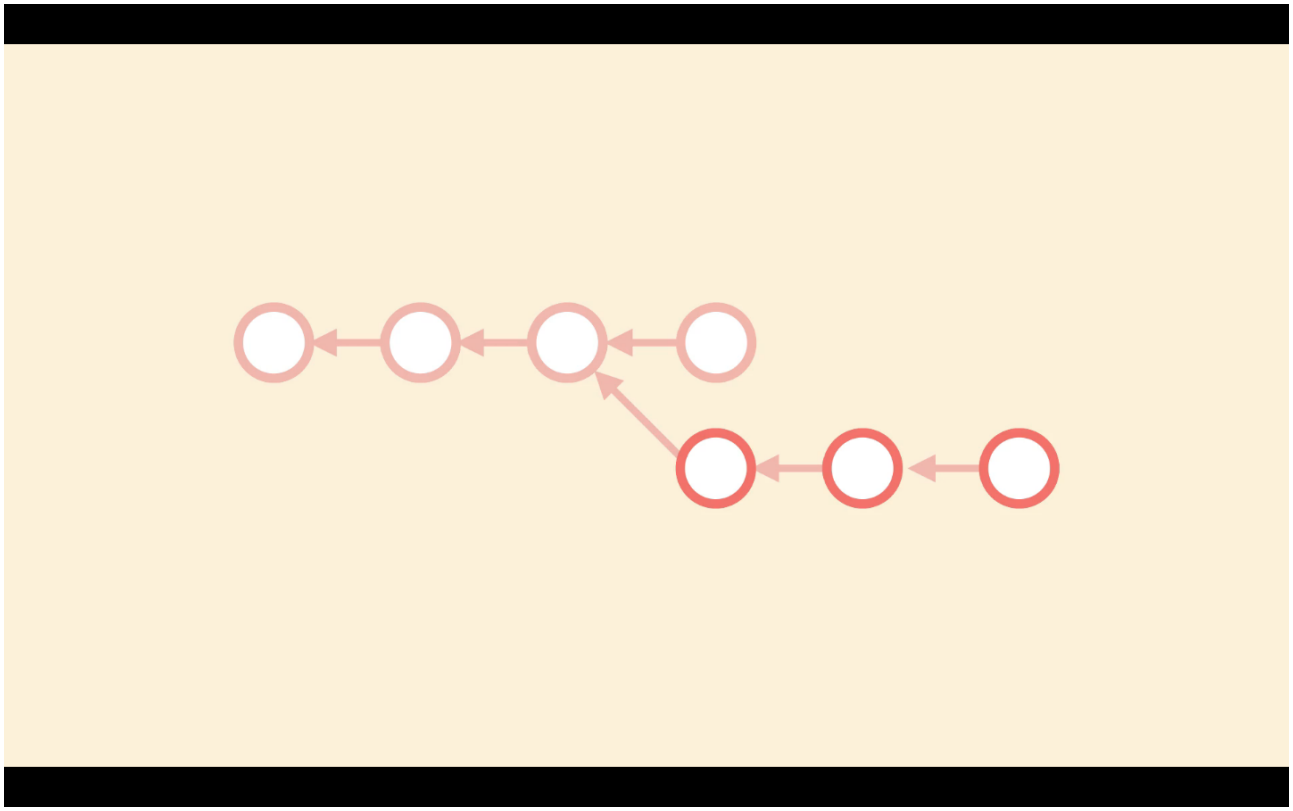
1- Introduction

In this section we will cover.

1. Use branches
2. Compare branches
3. Merge branches
4. Resolve conflicts
5. Udo a faulty merge
6. Essential tools (stashing, cherry picking)

02- What are Branches

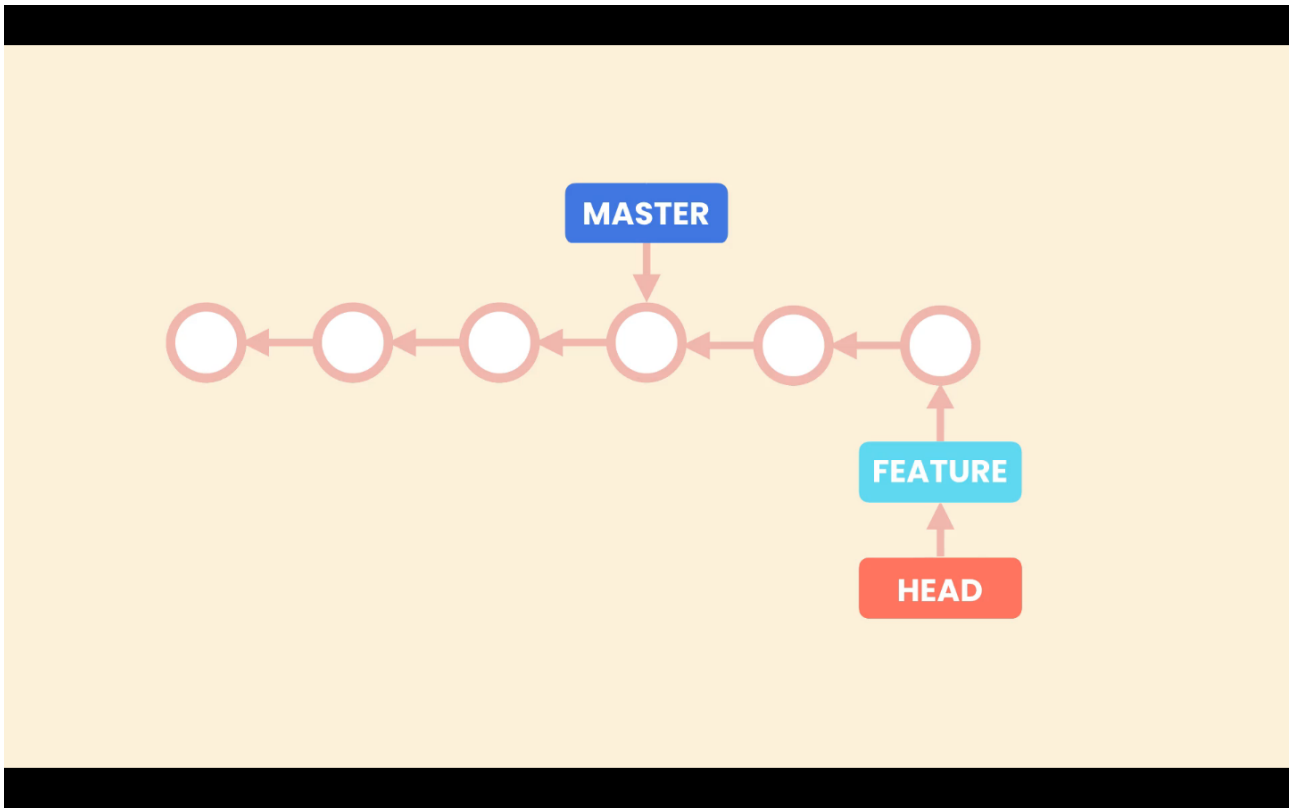
Branches allows us to work in an isolated environment from the main line. So branches are a kind of separate isolated workspace.



The principal line of work (branch) is called ***master*** or ***main***. When we need to work, for example, on a new feature we can create a new branch to work on that feature. And when we are done we merge the new feature branch into the ***main*** branch.

How does git manages branches

When we create a new branch Git creates a new pointer for that branch. When we work in that branch and make new commits Git moves that pinter forward, and the master pointer stays where it is. This way Git knows the latest code in each branch. To know in which branch we are working on, Git uses the **HEAD** pointer, when we switch banchs Git points the **HEAD** pointer to the branch we want to work on, and updates the **Working Directory** accordingly.



04- Working with Branches

Create new branch

To create a new branch run `git branch <name-of-branch>`

```
git branch bugfix
```

View Branches

To view all the available branch run `git branch`

```
❯ git branch
  bugfix
* main
```

The `*` in front of the ***main*** means that at the moment we are in that branch. It is also possible to view the current branch with `git status` .

Change branches

Nowadays the command to changes branches is `git switch <name-of-branch>` . It used to be `git checkout <name-of-branch>` , it is possible to still use the old command.

```
❯ git switch bugfix
Switched to branch 'bugfix'
```

Rename a branch `-m`

The branches name should represent the work that is being performed on it. To rename a branch run the command `git branch -m <old-name> <new-name>`

```
git branch -m bugfix bugfix-signup-form
```

Commit to branch

When we commit to a branch this branch moves forward and the ***main*** branch stays where it is, we can see that with `git log --oneline` . The ***HEAD*** pointer will be pointing to the new branch head of ***main***.

```
54bb974 (HEAD -> bugfix-signup-form) start new lesson
6d5df20 (main) lesson complete
b89cef8 lesson complete
b4ef25f restore file
dacc1ec delete file
33bceb5 start new lesson
```

```
1ce178a lesson complete
b5f716e lesson complete
```

If we switch back to ***main*** our **Working Directory** will be restored to that point.

Delete a brach `-d` or `-D`

To delete, first we need to change to a different branch usually ***main***, then we use the `-d` option, but if this branch as unmerged changes with ***main***, Git will throw an error warning us.

```
❯ git branch -d bugfix-signup-form
error: The branch 'bugfix-signup-form' is not fully merged.
If you are sure you want to delete it, run 'git branch -D bugfix-signup-form'.
```

To force the deletion use the `git branch -D bugfix-signup-form`.

05- Comparing Branches

Log differences

We can compare branches to see different commits between them. To do so use the command `git log <first-branch>..<second-branch>`. This will return all the commits that are in `<second-branch>`, and not in `<first-branch>`.

```
❯ git log main..bugfix-lesson
commit 6572ee7da7ce8b30c0c42cf2d6f27f6968c6b470 (HEAD -> bugfix-lesson)
Author: Miguel Pimenta <jmiguelpimenta@gmail.com>
Date:   Wed Mar 10 20:18:39 2021 -0300

    add details to lesson
```

It is also possible to use the `--oneline` option, `git log --oneline main..bugfix-lesson`.

See differences

To compare the actual changes between branches we use the `diff` command, like so `git diff main..bugfix-lesson` . This will produce an output just like the normal `diff` , but comparing the two branches. If we are in the ***main*** branch we do not need to specify it i the command, we can run `git diff bugfix-lesson` , this will have the same output.

We can also use the `--name-only` and `--name-status` optins here, like so `git diff --name-status main..bugfix-lesson` # 06- Stashing

Git stashing docs

Create stash

When we switch branches Git restores our **Working Directory** to the last snapshot of the target branch. In case we have local changes in the **Working Directory** that have not yet been committed, they could be lost. In such situations Git does not allows us to switch branch, ti will throw an erro if we try.

```
❯ git switch main
error: Your local changes to the following files would be overwritten by checkout:
    04 Branching (76m)/06- Stashing.md
Please commit your changes or stash them before you switch branches.
Aborting
```

Let's imagine that this changes are still work in progress, and we do not want to commit them yet. In this situation we should stash our changes. Stashing changes means save them in a Git safe place, but they will be not part of the history. To do so we run `git stash push -m <stashing-message>` . For example:

```
❯ git stash push -m "lesson details"
Saved working directory and index state On bugfix-lesson: lesson details
```

This command will stash (save) our changes, but they will not be displayed in the **Working Directory**. At this point we could switch branches and the changes will be safe.

If we have new untracked files, by default, they are not included in the stash, to include them we have to use the `--all` or `-a` option.

```
❯ git stash push -a -m "lesson details"
```

List stashes

To view the stashes we run `git stash list`. Each stash as a unique identifier, `stash@{0}`.

```
❯ git stash list
stash@{0}: On bugfix-lesson: lesson details stash list
stash@{1}: On bugfix-lesson: lesson details
```

Show stash changes

Before applying the changes to the **Working Directory** we can view them with the command `git stash show stash@{i}`, where `i` is the stash index. Or we just pass the index, like so:

```
❯ git stash show 1
04 Branching (76m)/06- Stashing.md | 19 ++++++++++++++++++
1 file changed, 19 insertions(+)
```

Apply stash changes to Working Direcotry

To apply the stash we use `git stash apply i` like so:

```
❯ git stash apply 0
On branch bugfix-lesson
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   04 Branching (76m)/06- Stashing.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Delete a stash

After applying the stash we can delete it running `git stash drop i`. Or we may not need to apply the changes from the stash, so we can delete it without applying.

```
❯ git stash drop 0
Dropped refs/stash@{0} (9206126462da8a1cfc1777768c767edf54c49cdd)
```

Alternately we can delete all stash running `git stash clear` .# 07- Merging

Git merging docs

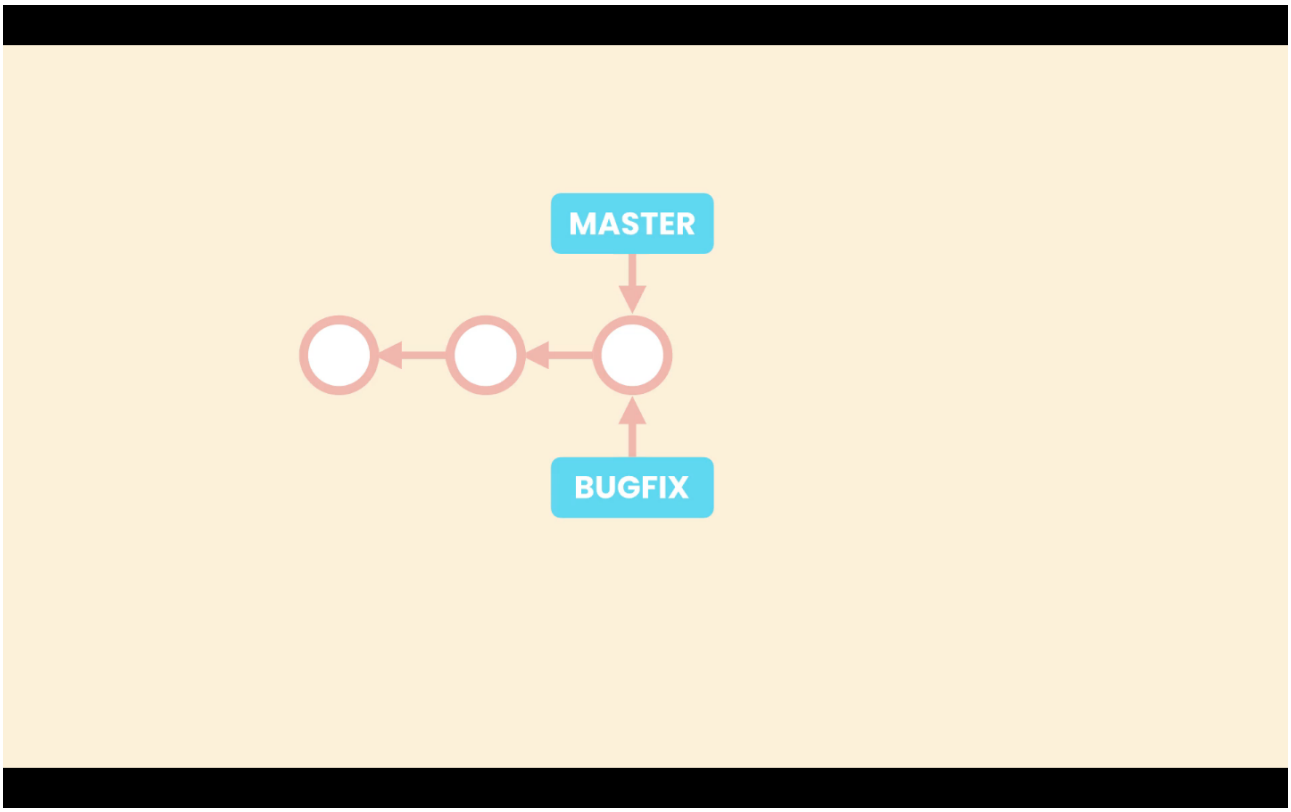
Merging is about bringing changes from one brach to another.

In Git we have two types of merges:

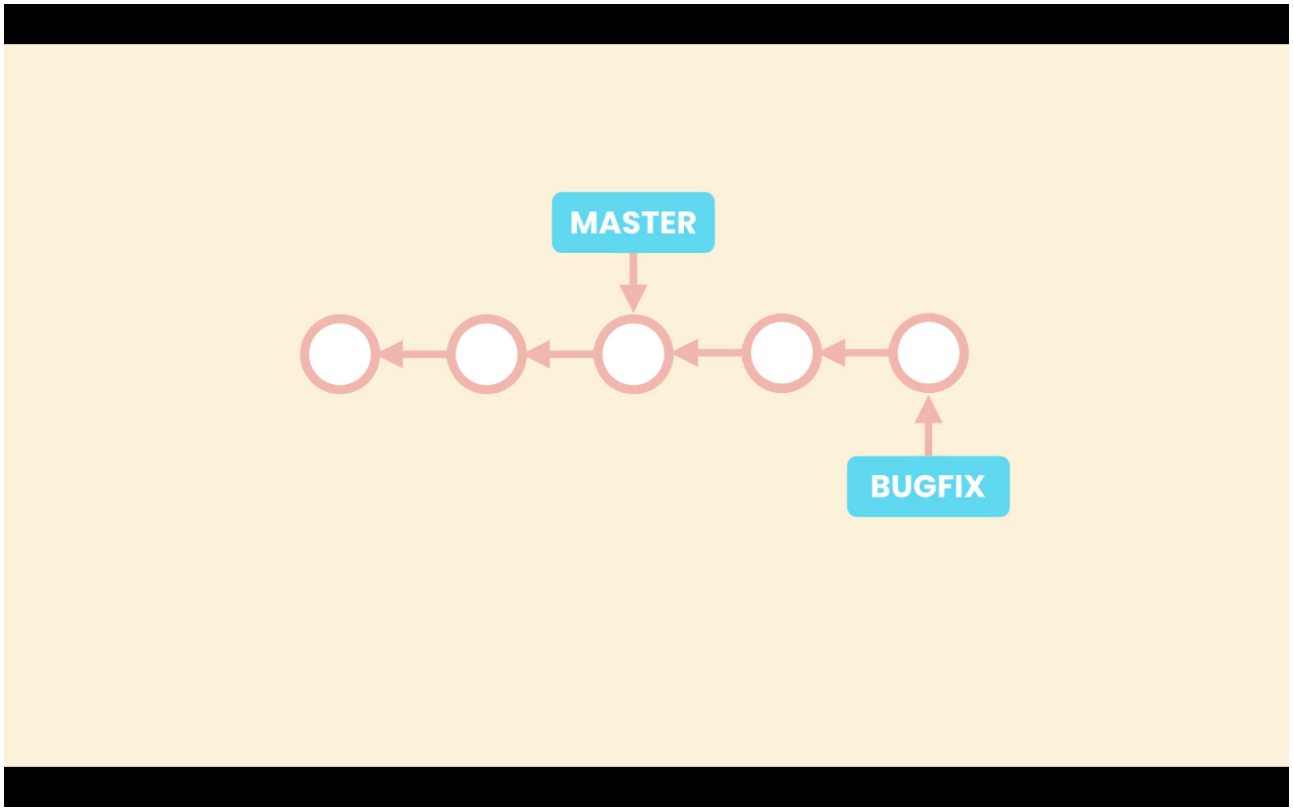
1. Fast-forward merges
2. 3-way mergers

Fast-forward merges

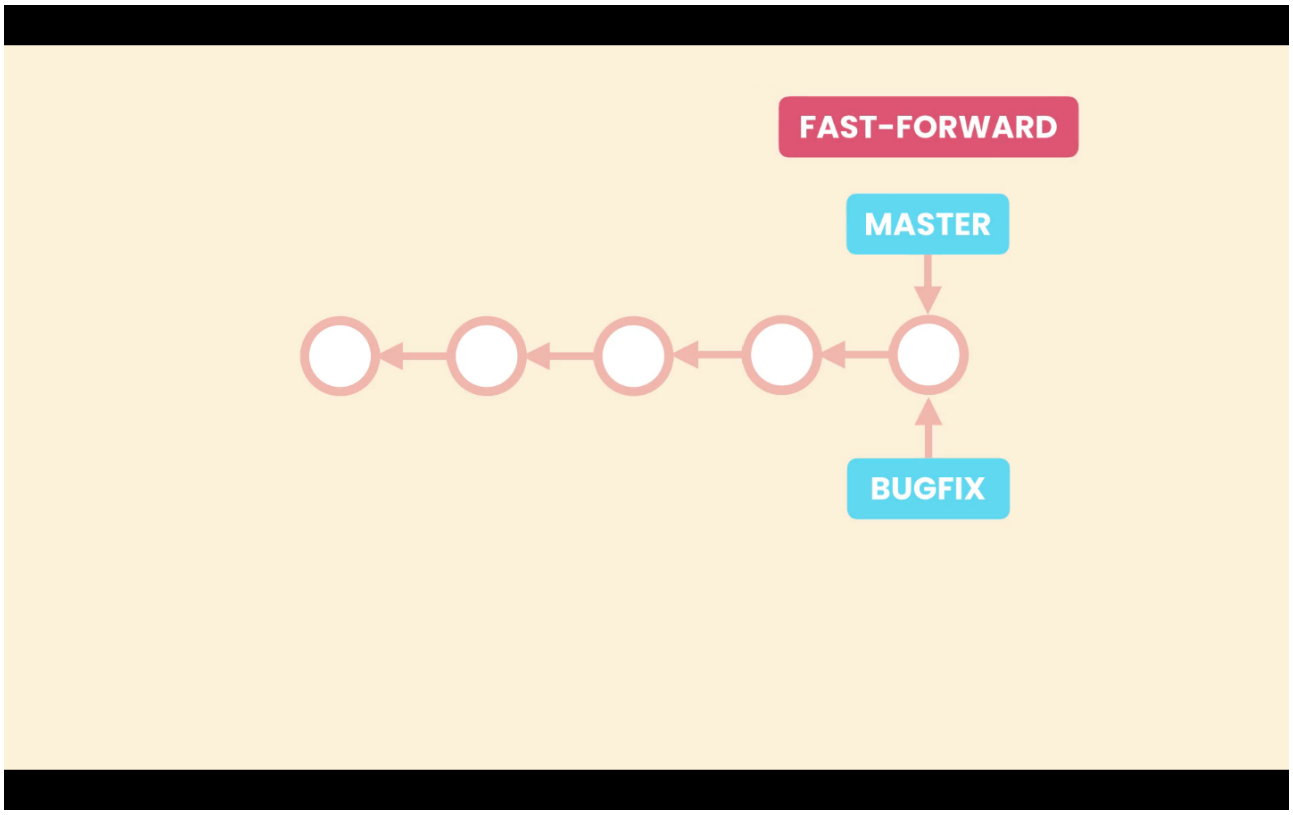
When we create new brach, let's call it ***bugfix***, from the ***main*** branch, both branches, ***main*** and ***bugfix***, will be pointing to the same commit.



Then when we switch to ***bugfix*** and start working on it and committing to it, the ***bugfix*** branch moves forward and **main** stays in the same place. This branches have not diverged and there is a direct linear path between them.

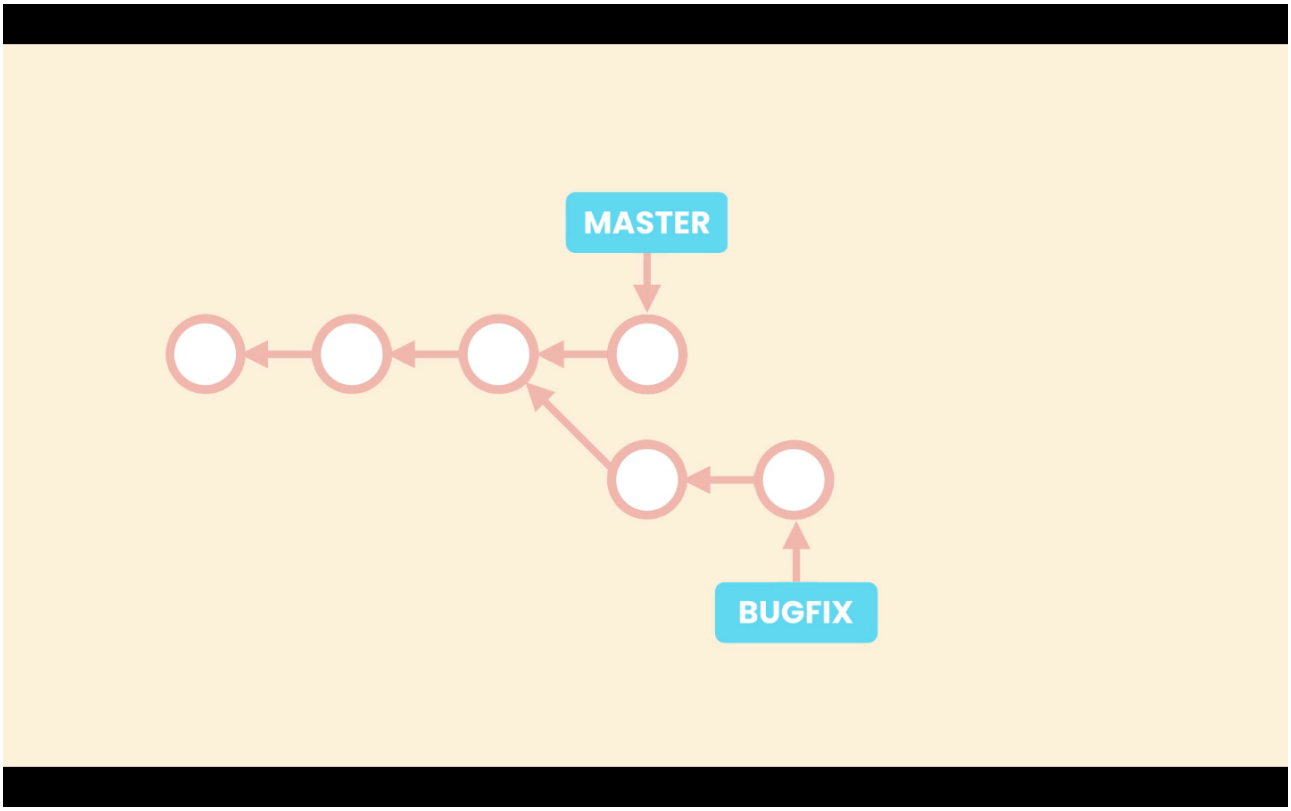


This way all Git has to do to merge the changes is to bring the master pointer forward. This is the fast-forward merge. Git run this type of merge when there is a direct linear path between the two branches.

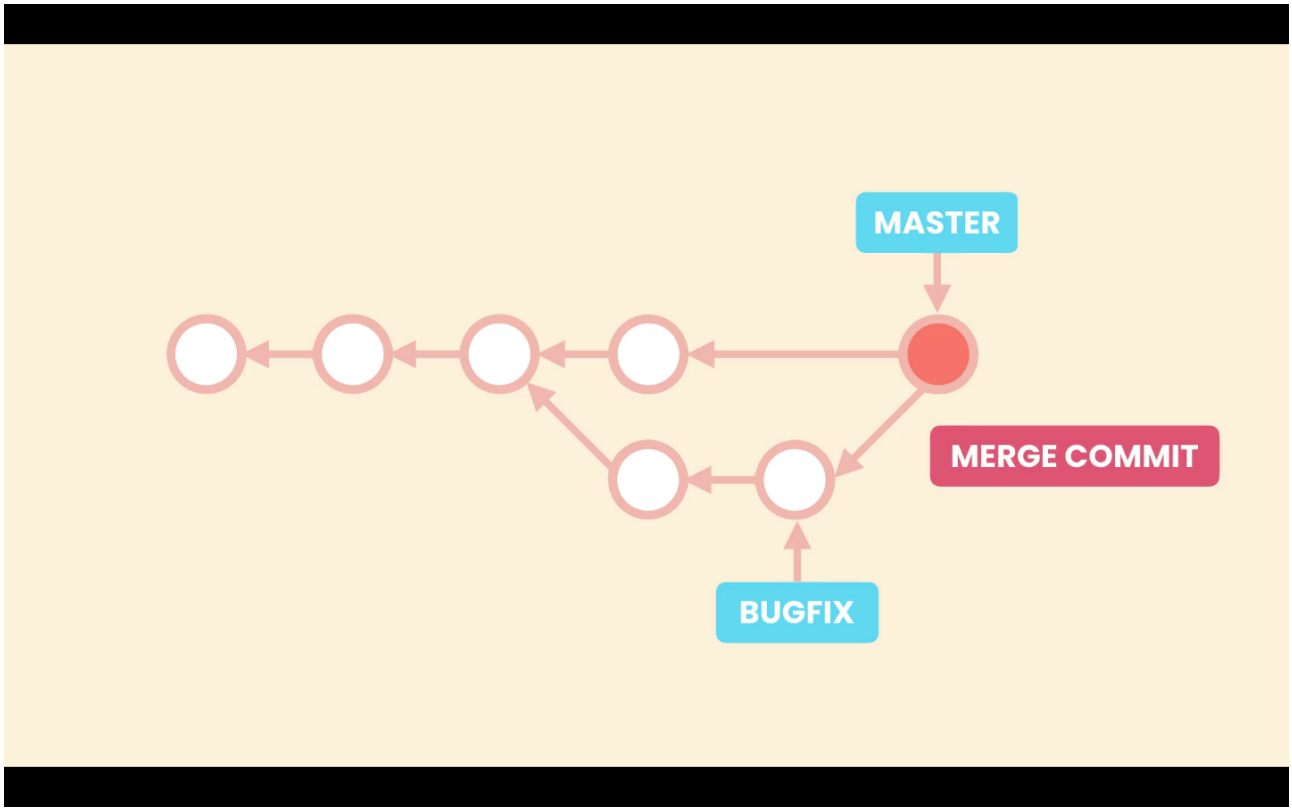


3-way merge

A 3-way merge happens when we apply some changes to the ***main*** branch, and commit them, after we created the ***bugfix*** branch. So we have some changes in ***main*** that do not exist in the ***bugfix*** branch. In this situation the two branches are diverged.



In this situation, when we run a merge git create a new commit that merges this two branches. This is called a 3-way merge because this merge commit is base in three different commit, the common ancestor of both branches, which has the before code, and the tips of both branches, tha contain the after code.



08- Fast-forward Merges

Git log for branches `--graph`

To have a better visualizations of branches with `git log`, it is better to include the `--graph` option. It will produce an output where we are abe to better view the branch path.

```
❯ git log --oneline --graph
* 23c694d (HEAD -> fast-forward-merge) add details to lesson
* 6d91b25 (origin/main, main) add details to lesson
* 932b8dd lesson complete
* 1433f35 lesson complete
* 6d7a701 style: add final empty line
* 18c119d lesson start
* 176761d lesson completed
[...]
```

In this log we can see that the branch ***`fast-forward-merge`*** is one commit ahead of ***`main`***, and that there is a linear path between them. So the could merger by the fast-forward method.

Merge branch fast-forward

To merge the ***fast-forward-merge*** branch into ***main***, we first should have committed all of our work in that branch and them we switch to ***main***. In the ***main*** branch we run the merge with the command:

```
git merge fast-forward-merge
```

After we can run `git log` to see the result. As we can see in the log now both branches point to the same commit.

```
* 3c5127f (HEAD -> main, fast-forward-merge) add details to lesson
* 8c79b5d add details to lesson
* 23c694d add details to lesson
* 6d91b25 add details to lesson
* 932b8dd lesson complete
* 1433f35 lesson complete
```

Merge without fast-forward `--no-ff`

It is also possible to enforce a non fast-forward merge with the command `git merge --no-ff <branch-name>`.

```
git merge --no-ff no-fast-forward-merge
```

With this we tell Git that, although it is possible to have a fast-forward merge, don't do it, and that it should create a new commit to merge ***main*** with another branch.

Here as well we can see the ***no-fast-forward-merge***, branch is three commit ahead of ***main*** and there is a linear path between them.

```
❯ git log --oneline --graph
* 00a98bb (HEAD -> no-fast-forward-merge) add details to lesson
* 9aff663 add details to lesson
* 2694107 add details to lesson
* d8fbb20 add details to lesson
* 3c5127f add details to lesson
* 8c79b5d add details to lesson
* 23c694d add details to lesson
* 6d91b25 add details to lesson
* 932b8dd lesson complete
```

```
* 1433f35 lesson complete
[...]
```

When we run the `git merge --no-ff no-fast-forward-merge`, our default editor will open, with the default merge commit message ***Merge branch 'no-fast-forward-merge'***. We do not need to modify the message, and we can add details if needed.

Now when we run `git log` we can see the new merge commit.

```
❯ git log --oneline --graph
*   afba0c3 (HEAD -> main) Merge branch 'no-fast-forward-merge'
|\
| * 00a98bb (no-fast-forward-merge) add details to lesson
| * 9aff663 add details to lesson
| * 2694107 add details to lesson
|/
* d8fbb20 add details to lesson
* 3c5127f add details to lesson
* 8c79b5d add details to lesson
* 23c694d add details to lesson
```

The new branch for this example called ***no-fast-forward-merge***, was created from commit **d8fbb20**, then there were three commits on it, and at last was merged into ***main*** in the commit **afba0c3**, that was triggered by the merge command with the `--no-ff` option.

Disable fast-forward merges

It is possible to disable fast-forward merges, that way all mergers Git performs will be non fast-forward, even if it is possible to have a fast forward merge.

1. Disable for a single repository: `git config merge.ff false`
2. Disable globally `git config --global merge.ff false`

09- Three-way Merges

The 3-way-merge is implicit when the branches to merge diverge. This happens when changes are made to the original branches after the creation of the new branch.

In this example I have created a new branch from ***main*** (the original branch), called ***3-way-merge***, in the moment of creation both branches are pointing to the same commit.

```
❯ git log --oneline --graph
* 2827b4c (HEAD -> 3-way-merge, main) add details to lesson
* 80a0972 add details to lesson
*   afba0c3 Merge branch 'no-fast-forward-merge'
|\
| * 00a98bb add details to lesson
| * 9aff663 add details to lesson
| * 2694107 add details to lesson
|/
* d8fbb20 add details to lesson
[...]
```

When we commit to ***3-way-merge***, this branch will move forward, but will have a linear path to ***main***. This linear path is broken when changes are applied to ***main*** before the merge.

Here I have switched to ***main*** and made 2 commits. So now the ***main*** branch has diverged from ***3-way-merge***.

```
❯ git log --oneline --graph
* e81cc69 (main) add lesson title
* b9063c7 add file for lesson
| * 08e6d5e (HEAD -> 3-way-merge) add details to lesson
|/
* 2827b4c add details to lesson
* 80a0972 add details to lesson
*   afba0c3 Merge branch 'no-fast-forward-merge'
|\
| * 00a98bb add details to lesson
| * 9aff663 add details to lesson
| * 2694107 add details to lesson
|/
* d8fbb20 add details to lesson
[...]
```

Now when we run a merge Git will run a 3-way-merge. It will open the default editor with a commit message, when add more details if needed to the commit message.

```
❯ git log --oneline --graph
*   e26d5a7 (HEAD -> main) Merge branch '3-way-merge'
|\
| * 2f37a10 (3-way-merge) add lesson title
| * c99eb28 add file for lesson
| * 08e6d5e add details to lesson
* | e81cc69 add lesson title
```

```
* | b9063c7 add file for lesson
|/
* 2827b4c add details to lesson
* 80a0972 add details to lesson
*   afba0c3 Merge branch 'no-fast-forward-merge'
|\
| * 00a98bb add details to lesson
| * 9aff663 add details to lesson
| * 2694107 add details to lesson
|/
* d8fbb20 add details to lesson
```

In the log we can see the merge commit. After a merge we can deleted the merged branch `3-way-merge`, in this case.

10- Viewing Merged and Unmerged Branches

List merge branches `--merged`

When we are finished working in a branch, we should merge it into `main`, and afterwards delete it.

To view the list of merged branches run the following command:

```
git branch --merged
```

List unmerged branches `--no-merged`

To view the list of unmerged branches run

```
git branch --no-merged
```

11- Merge Conflicts

Very often when we are merging branches we run into conflicts. Conflicts happen when:

1. The same line of code was been changed in two different ways, in the merged branches.
2. A given file is changes in one branch, but delete in the other branch.
3. The same file is added in two different branches with different content.

When conflict happens Git can not merge the branches automatically, and we must step in. After running `git merge <name-of-branch>` we will be warned and in `git status` we can see the following:

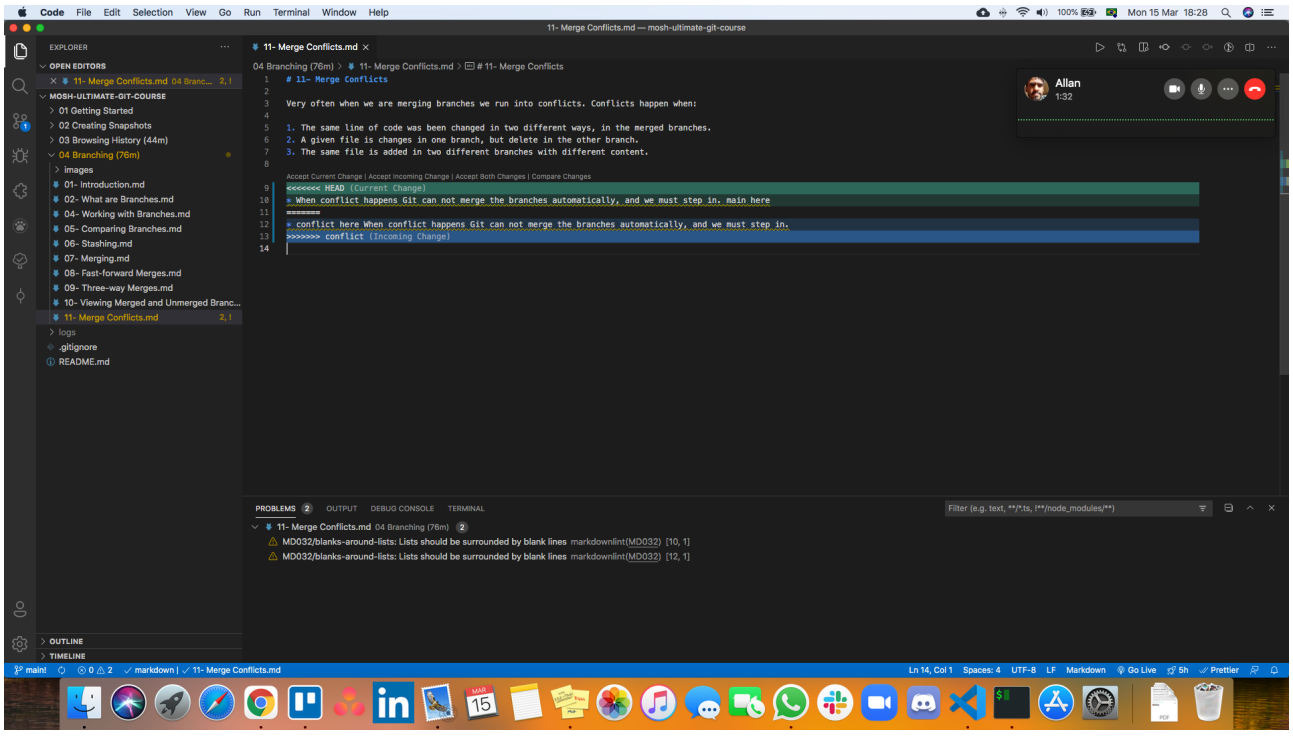
```
❯ git status
On branch main

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   04 Branching (76m)/11- Merge Conflicts.md

no changes added to commit (use "git add" and/or "git commit -a")
```

As we can see `Unmerged paths` is were the conflict is. If we open the file in VS Code, during the merge operation, we will see the conflict



Here we can use VS Code options, or edit the file manually to solve the conflict. When solving conflicts we should avoid at all cost adding new code.

After finishing resolving the conflict, we must add the file to the **Staging Area**, with `git add <file>`, and commit it. Because this is a merge commit, we do not need to pass a message, we can accept the default message, just run `git commit`.

12 - Graphical Merge Tools

We can use external merge tools like:

1. Kdiff
2. P4Merge
3. WinMerge (Windows only)

Other tools like GitKraken or IDEs, may even have better functionalities.

13- Aborting a Merge

In case we want to abort a merge that has conflicts we can use the `--abort` option.

```
git merge --abort
```

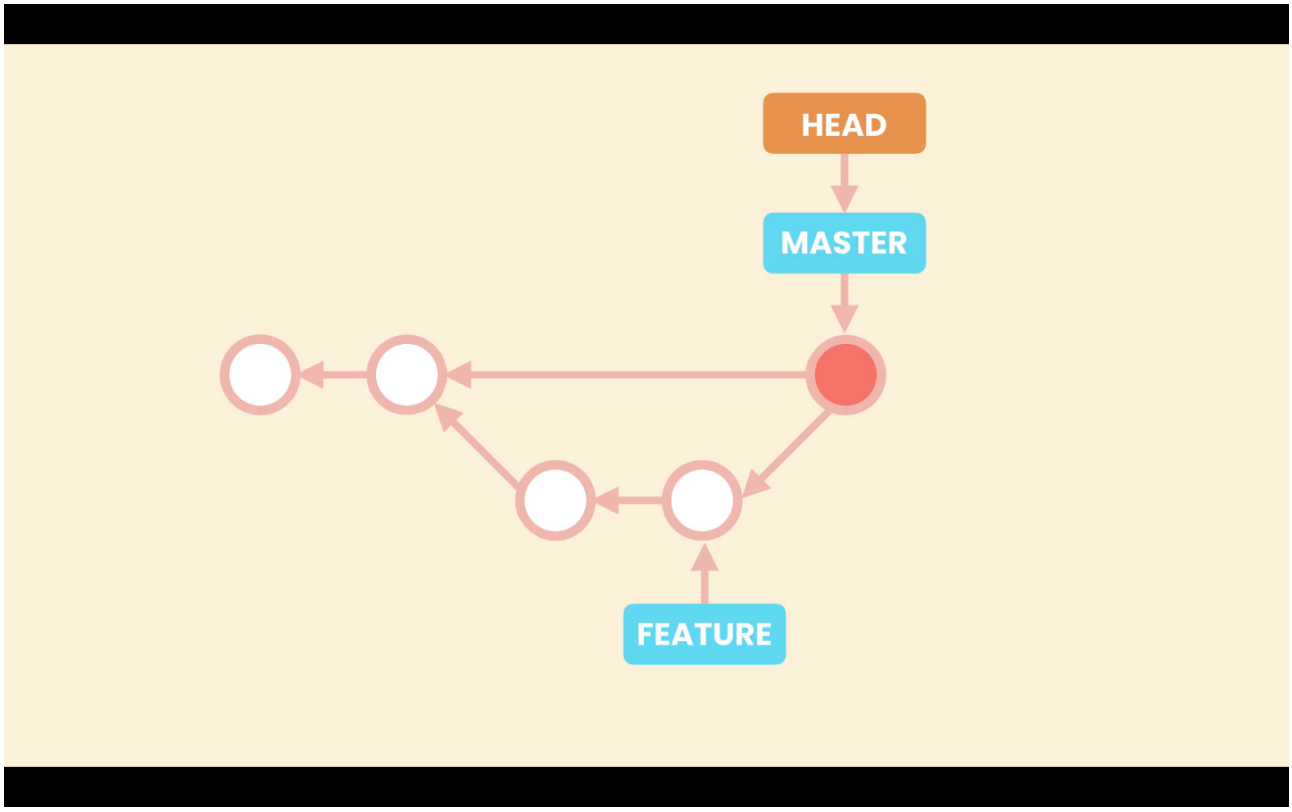
14- Undoing a Faulty Merge

In case we need to undo a merge we have two options:

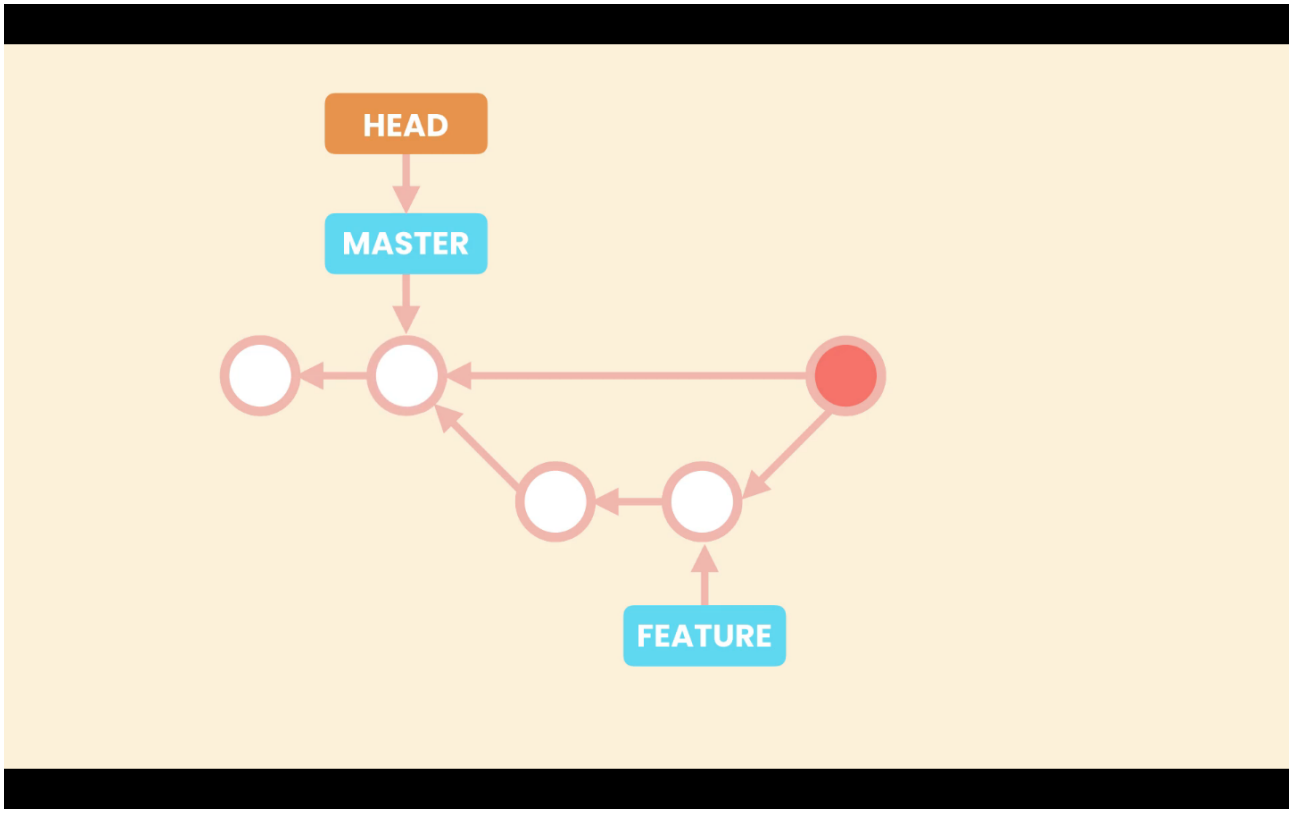
1. Remove the merge commit, as if it was never there (only if we did not shared the repository with others)
2. Revert the merge commit

1. Remove the merge commit

After a merge we have the ***main*** and ***HEAD*** pointers, pointing to the last commit, that is the merge commit.



With the reset command we will move both pointers to the last commit before the merge. And remove the merge commit.



Run the command:

```
git reset --hard HEAD~1
```

After we run the command the merge commit does not have any commit or pointer pointing to it. So for Git this garbage, once in a while Git looks for commits like this and removes them.

Resting the *HEAD* options

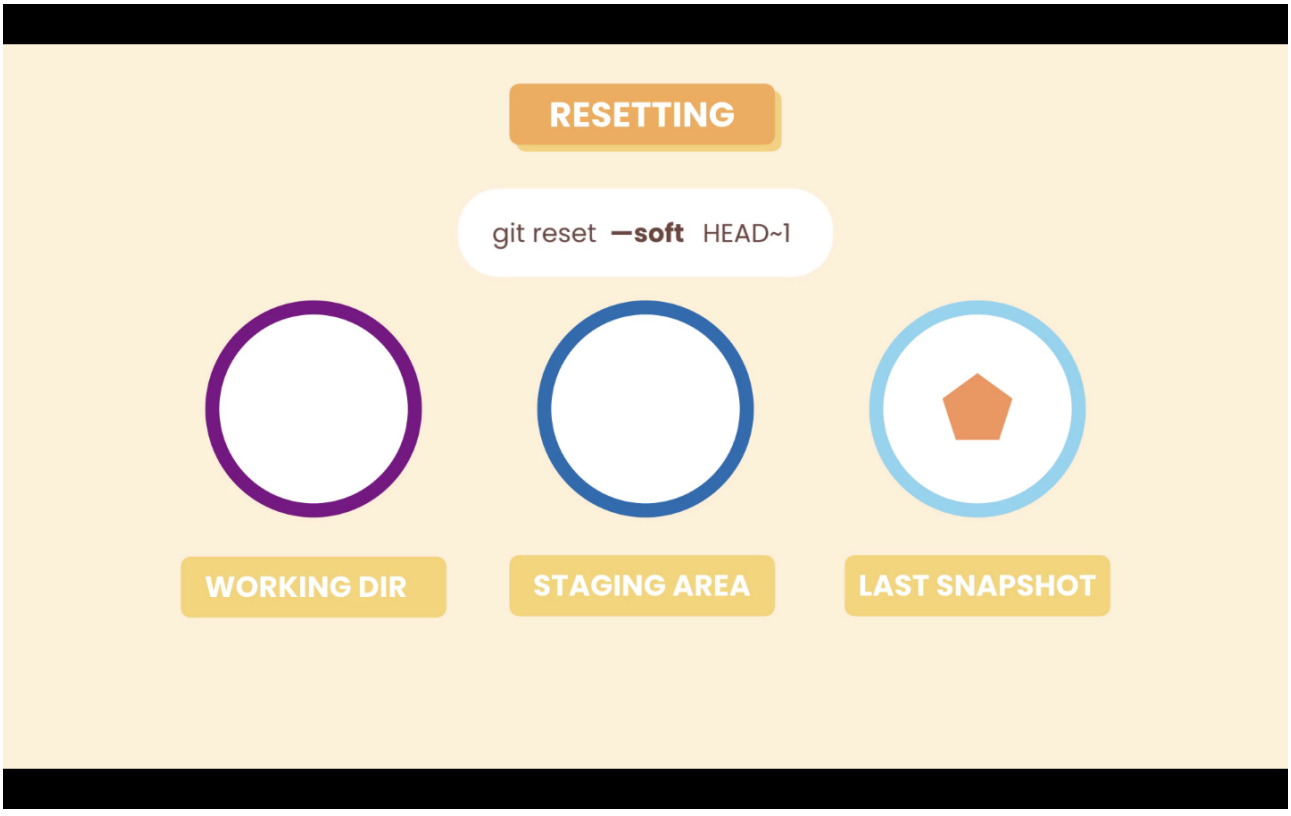
When resting the ***HEAD*** pointer, we have three option:

1. soft
2. mixed
3. hard

Option `--soft`

When we reset the ***HEAD*** using the `--soft` option, Git will have the ***HEAD*** pointer pointing to the indicated commit, in this case `HEAD~1` , so go back one commit, in the ***Repository***. But the ***Staging Area*** and ***Working Directory*** are not affected.

```
git reset --soft HEAD~1
```

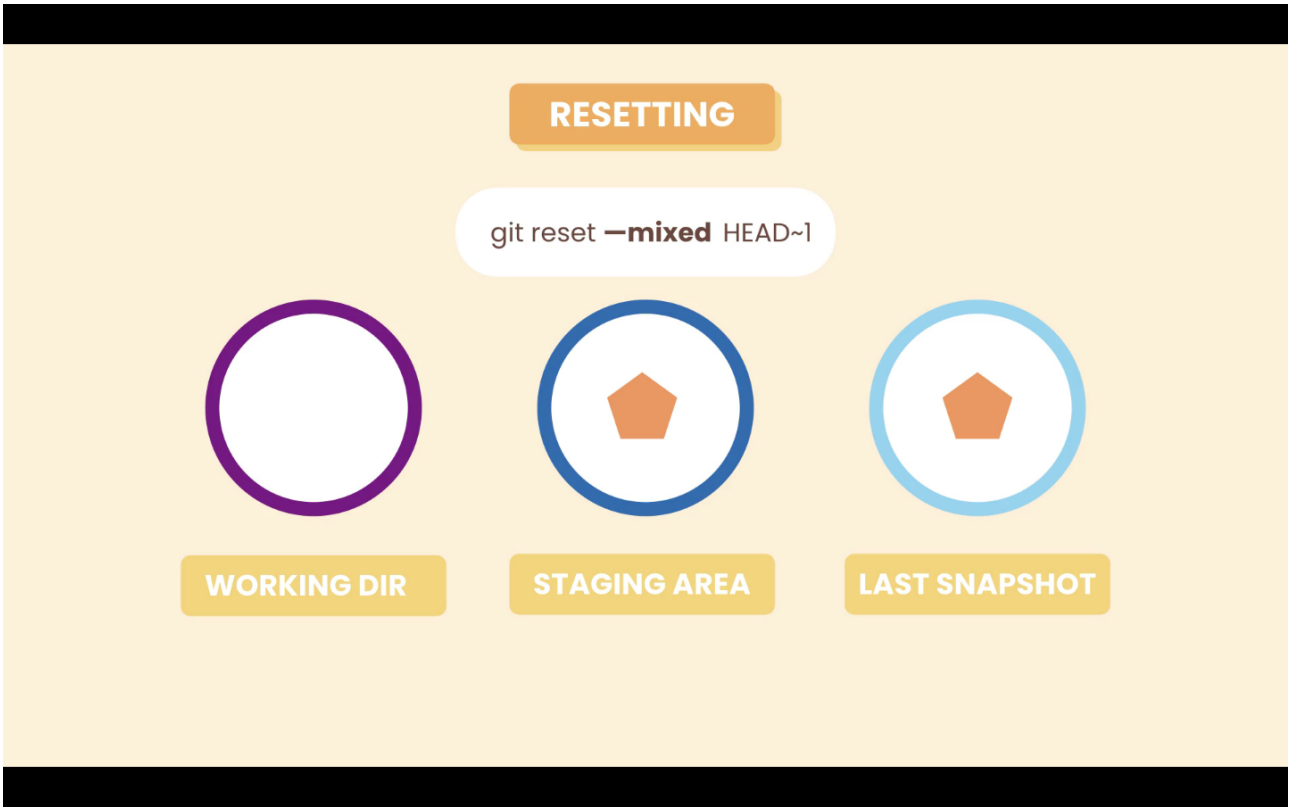


Option `--mixed`

In this case Git is going to apply the new snapshot to both the ***Repository*** and ***Staging Area***, local changes in the ***Working Directory*** will not be affected.

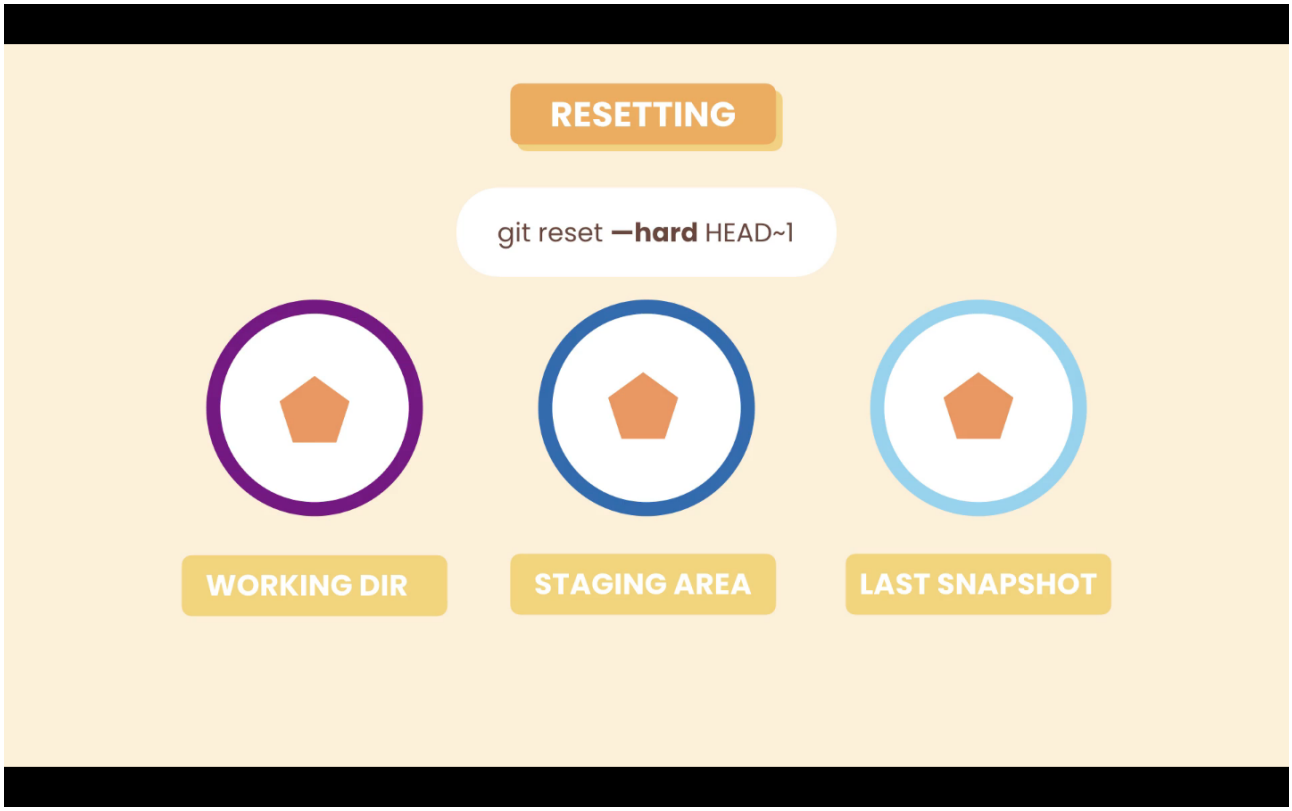
Using the `--mixed` option, which is the default option so we do not need to specify it.

```
git reset HEAD~1
```



Option `--hard`

Using the `--hard` option Git will apply the new snapshot to all environments (***Working Directory***, ***Staging Area***, ***Repository***).This was the state we where before stating the merge.



2. Revert the merge commit

If we have shared our history instead of undoing the merge commit we have to revert the last commit.

A merge commit has two parents so have to tell git how we want to revert the changes. Because our merge commit is in the `main` branch the first parent should also be in the `main` branch. To do this we use the the following command:

```
git revert -m 1 HEAD
```

In the `-m 1`, we are specifying the first parent. And `HEAD` is representing the target commit, the last commit.

Running this Git will open the default editor with a default message for the revert commit.

Option `-m`

From `git revet --help`

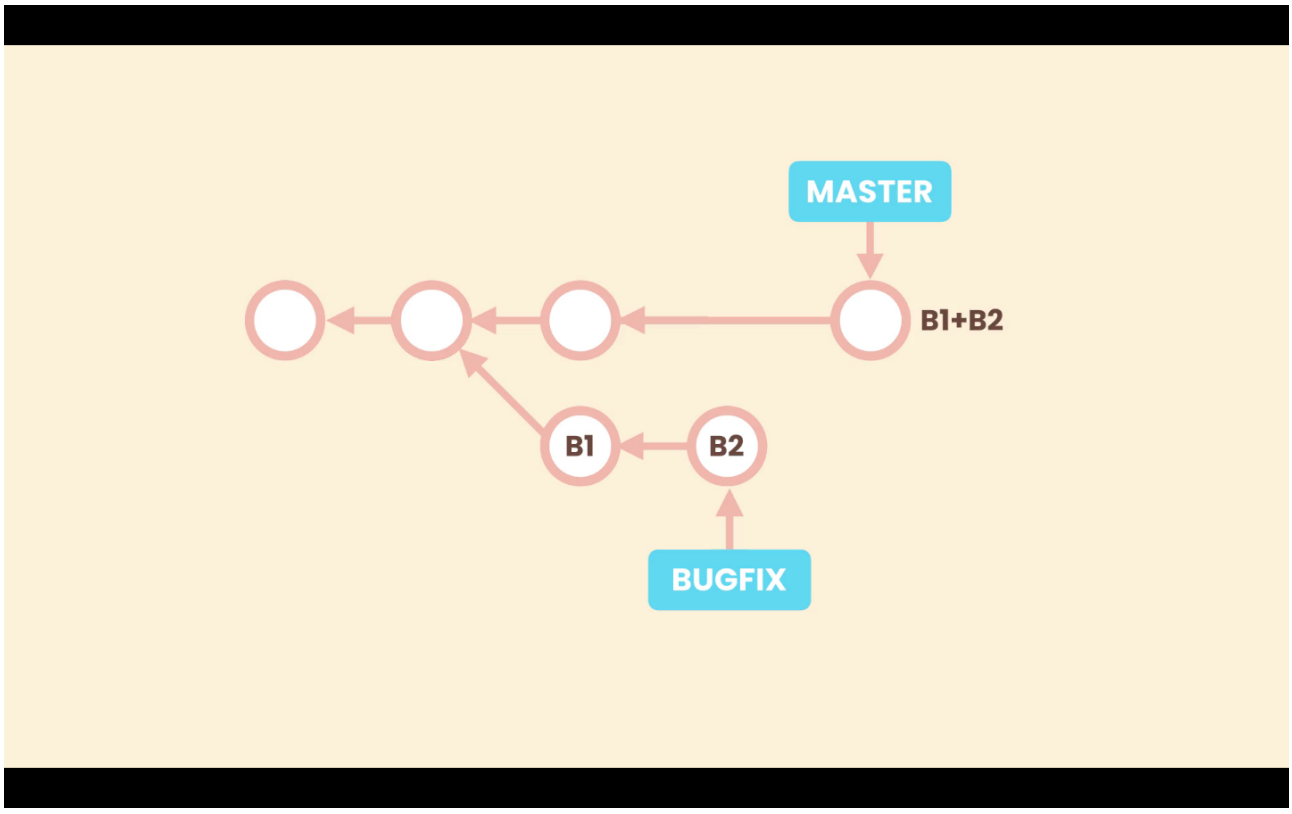
> -m parent-number, --mainline parent-number Usually you cannot revert a merge because you do not know which side of the merge should be considered the mainline. This option specifies the parent number (starting from 1) of the mainline and allows revert to reverse the change relative to the specified parent.

15- Squash Merging

In squash merging we first combine the commits from the branch, and then we merge. This is useful in situations where the commits in the branch are not good quality commits, or simply we do not need the history from the branch.

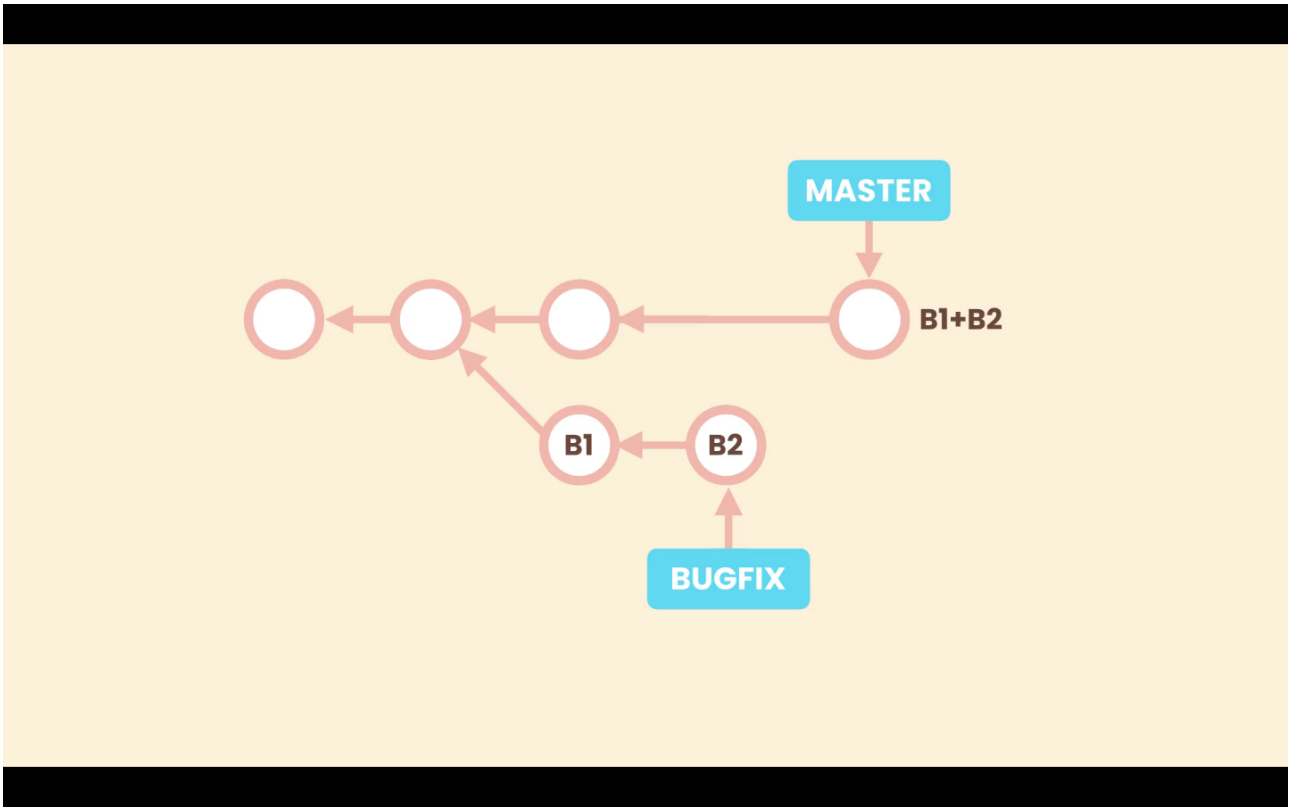
For example commits are

- To fined grain
- Maybe we have mixed different things in each commit



This new commit is not a merge commit, because it does not have two parents. It is lacking the reference to **B2**, the last commit from the **bugfix** branch. It is just a regular commit added on top of `main` that combines the commits from the other branch.

When we delete the **bugfix** branch, we are left with a clean linear history. This is the benefit fo Squash merging. But usually we should only apply it to short lived branches with bad history.



To perform a squash merge use the following command `git merge --squash <name-of-branch>` . Git will create a new commit, called a ***`Squash commit`*** , that combines the changes made in the merged branch, and it will and the changes to the ***Staging Area***. Then we just need to commit then normall.

```
git merge --squash bugfix
```

List merged and unmerged branches

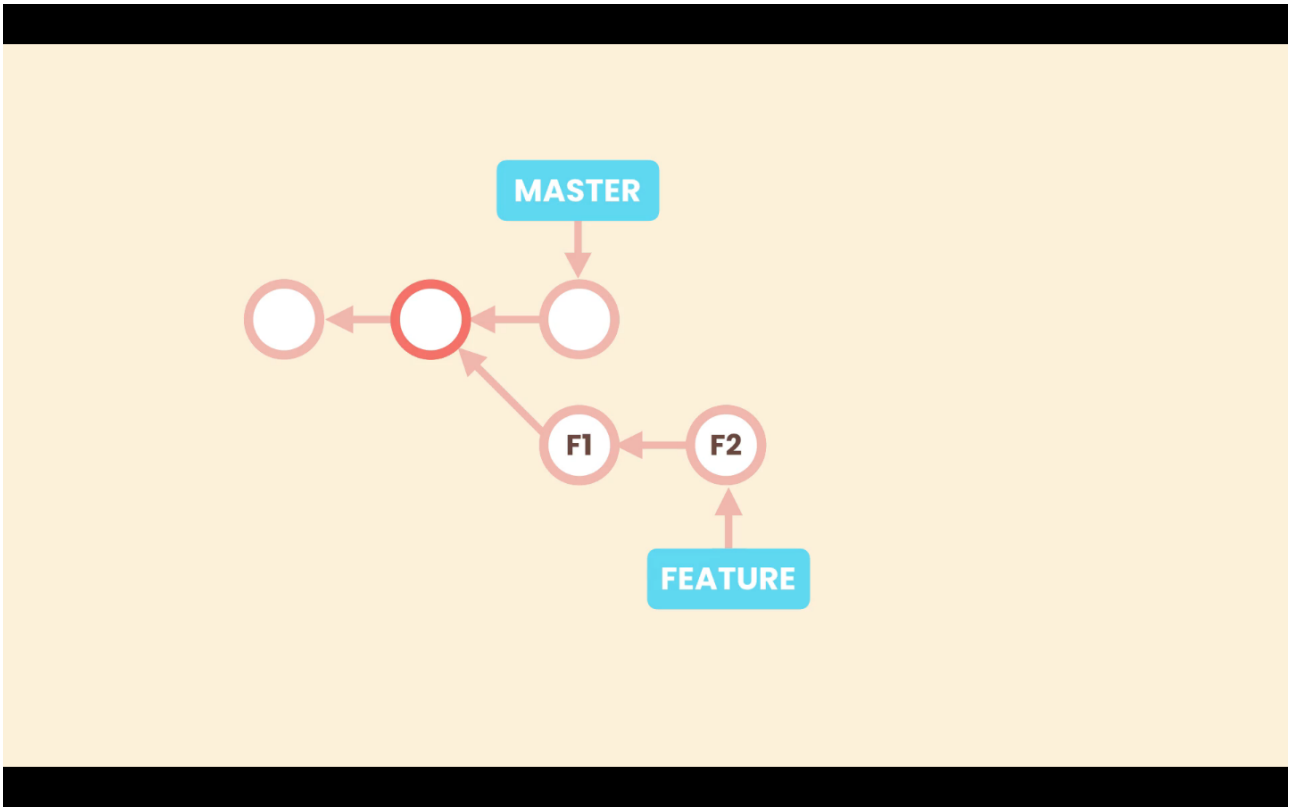
If we run `git branch --merged` , to list all the merged branches we will not see the **bugfix** branch. Because this branch was not actually merged. So it best to delete it after the squash merge, but in this situation we have to use `-D` instead of `-d` , or Git will throw an error. So `git branch -D bugfix` .

Conflict in Squash merge

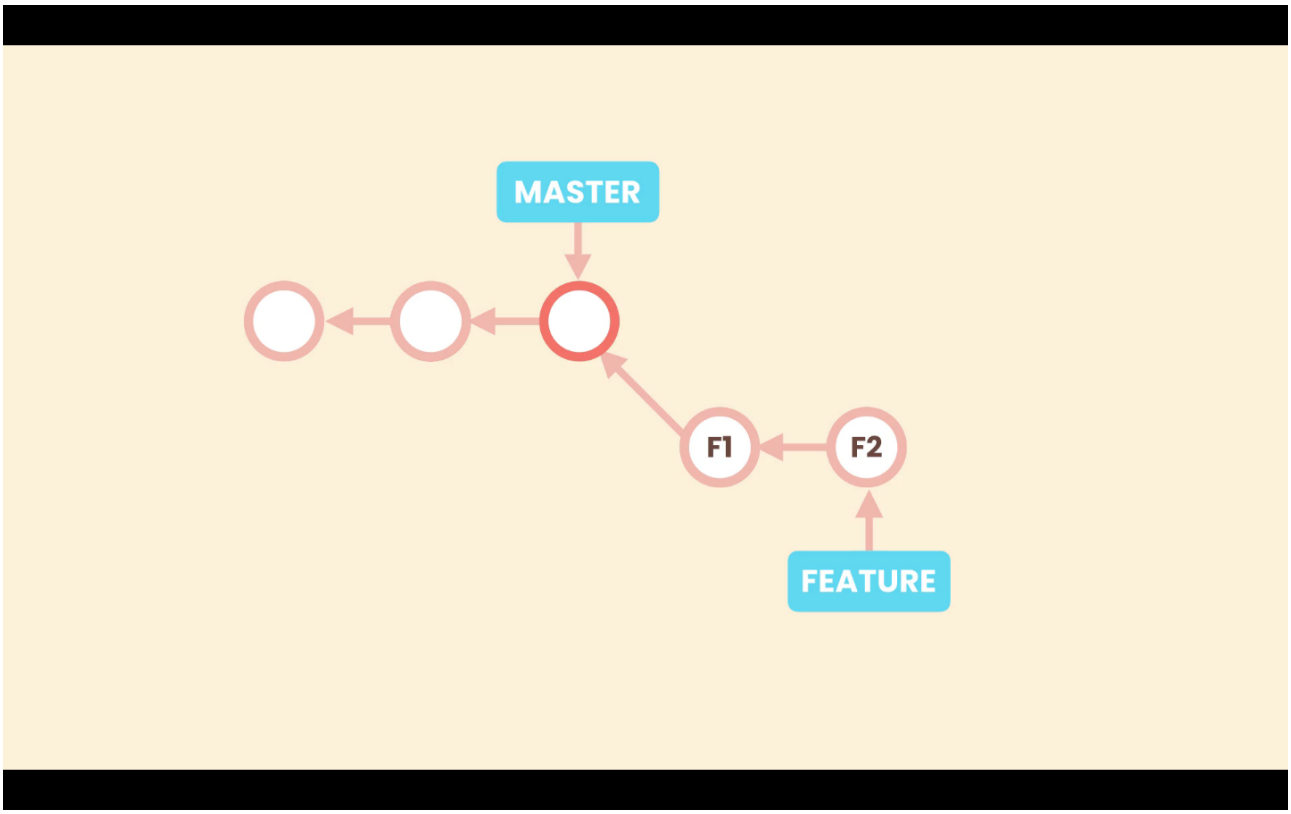
In case we run into conflicts when running a Squash merge, we can resolve this conflicts as in a normal merge.

16- Rebasing

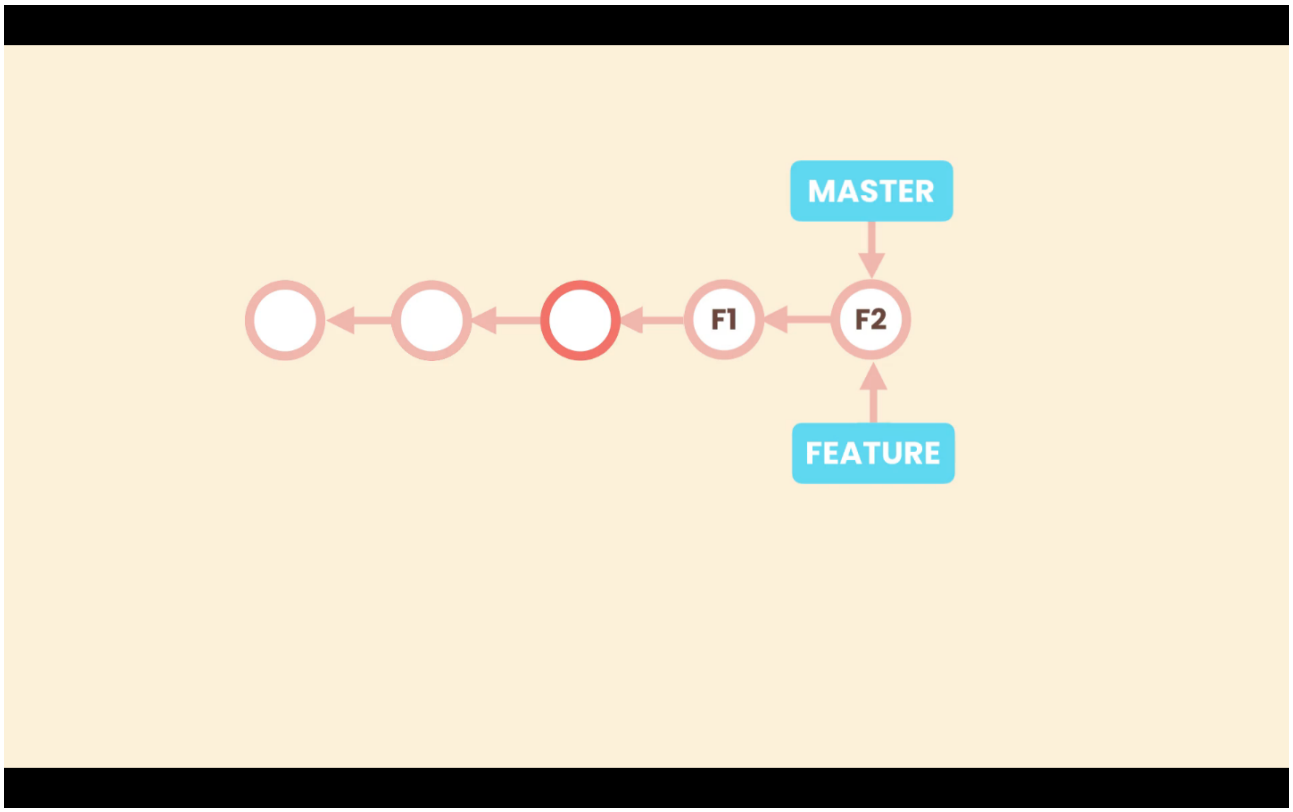
Suppose we have `main` and a new branch, let's call it **feature**, that have diverged. With the rebase technic we can change the base of the **feature** branch, making the base of the branch the latest commit on `main`. This will result in a linear history.



After rebasing the base commit of **feature** will be the latest commit on `main`.



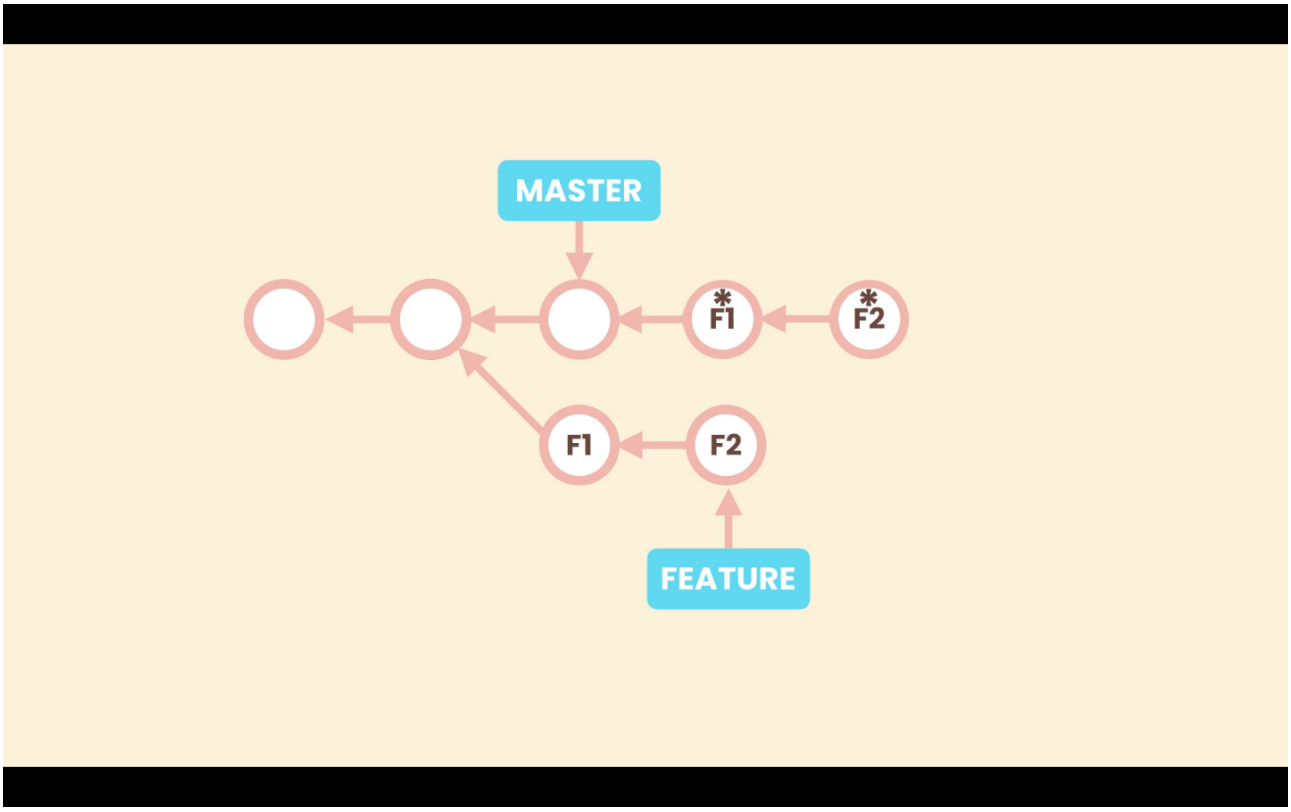
Then when we merge **feature** into *`main`*, it is like a fast-forward merge, with linear history.



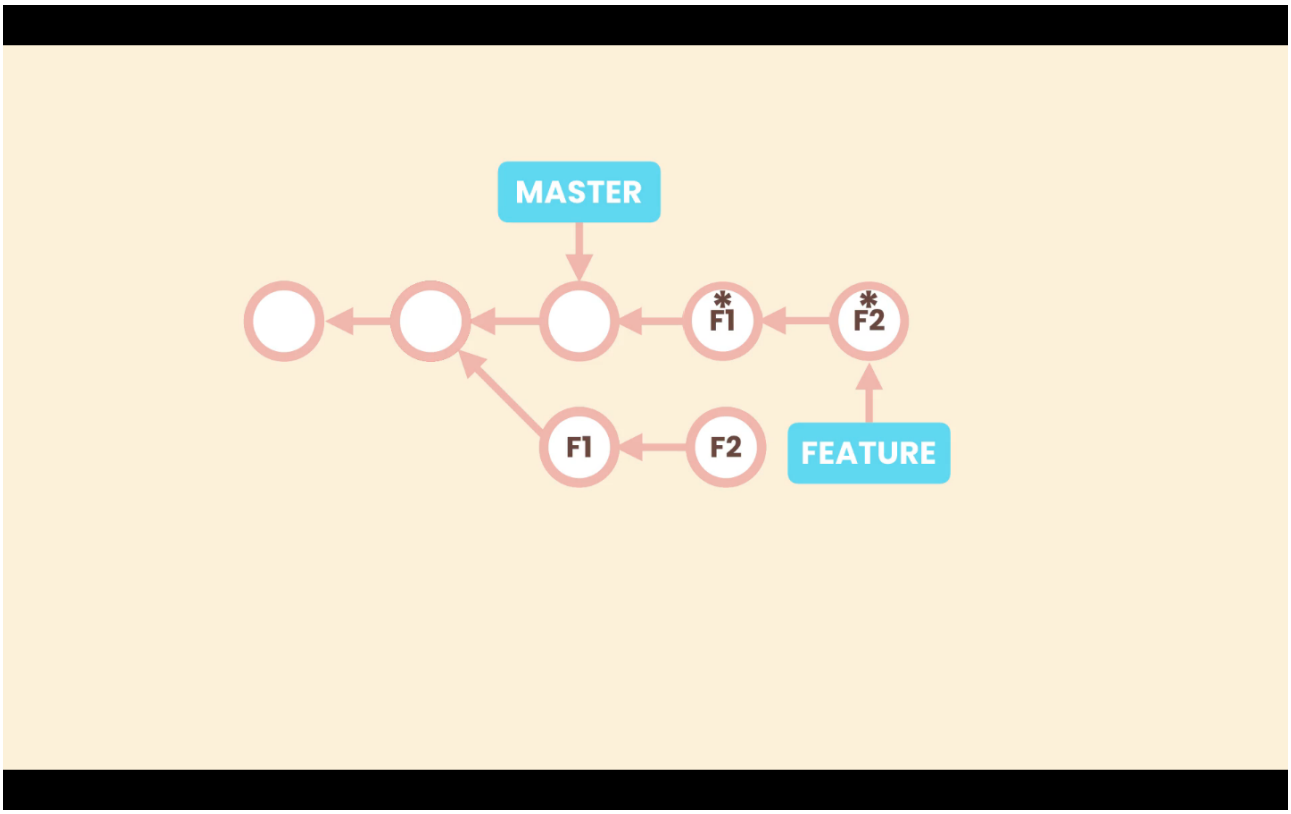
How does Rebase merge work

We should be caution with rebasing because it rewrites history, and so we should only apply it, if we have not shared our history, with other people.

Git commits are immutable, so what actually happens with rebasing, is that Git creates new commits that look like the commits in our branch **feature**, and applies them in top of *`main`*.



Then Git move the pointer of **feature** to the latest commit of new commit that it has created. The original commit, in this case **F1** and **F2**, will be considered garbage for Git and it will eventually remove them.



Rebase command

First in the new branch in this case **feature** we run `git rebase main`.

```
❯ git rebase main
Successfully rebased and updated refs/heads/feature
```

Then we switch to ***main*** and run `git merge <name-of-branch>`.

```
git merge feature
```

Resolving conficts

Rebase `--continue`

Resolving conflicts is similar as in other situations. When we run `git rebase main`, we will be warned of conflicts. After resolving this conflicts we run:

```
git rebase --continue
```

This will make Git apply the next commit on top of ***main***. It is possible that, the next commit also as a conflict, so we must resolve it once more.

Rebase `--skip`

We can use the `--skip` option to skip the current commit and move to the next commit. For example if the conflict appear in a particular commit, but we do not care about that commit.

```
git rebase --skip
```

Rebase `--abort`

We can abort the rebase operation with the option `--abort`. For example if we have to many conflict and do not want to go throw a complete rebase. Aborting the rebase will take us back to the previous state before starting rebasing.

```
git rebase --abort
```

Merge tool backup file

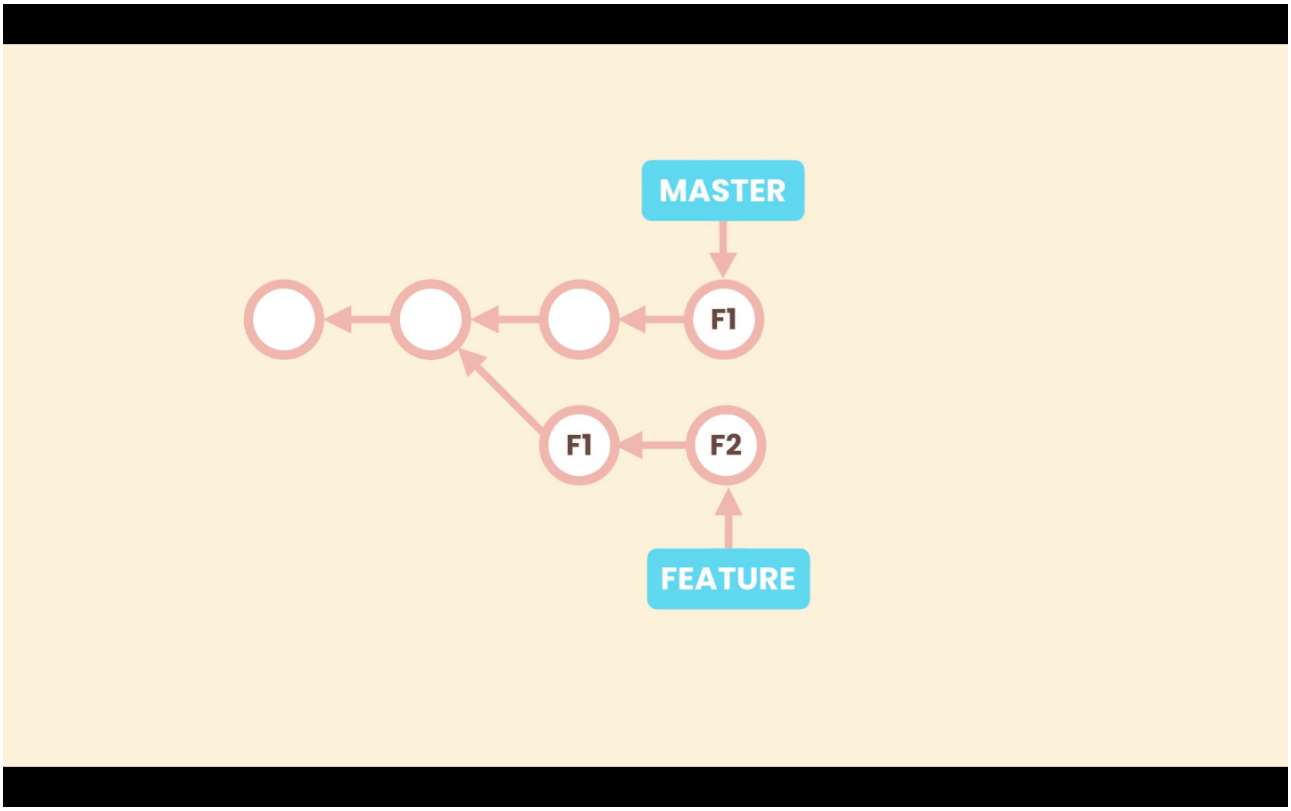
Depending on the merge tool, after we abort the rebase operation, it can create backup files from the conflict. If we do not need this file we can just remove it.

To prevent the merge tool from automatically creating this files we can set:

```
git config --global mergetool.keepBackup false
```

17- Cherry Picking

Imagine we are working in a branch that is diverged from ***main***, and we need one of the commits from this branch in ***main***. Just one commit, not a full merge.



To achieve this we use cherry picking.

First we must be in the ***main***, and then there we run `git cherry-pick <commit-ID>`

```
git cherry-pick 5670ecc
```

Then we must make a commit. We do not need to specify a message, wen we run `git commit`, the default editor will open with a default message based in the commit we are cherry picking, we can accept it or change it.

18- Picking a File from Another Branch

Sometimes we may need a single file from one branch in another branch. Unlike cherry picking that merges a commit, we can bring only one file. For this operation we use the `restore` command. In the branch that needs the files we run `git restore --source=<name-of-branch> -- <file-name>`, git will update the ***Working Directory*** with the latest version of that file from the target branch

Let's suppose we need a file named `config.rb` from an unmerged branch called **feature** in ***`main`***, so we run:

```
git restore --source=feature -- config.rb
```

combined_readme.md

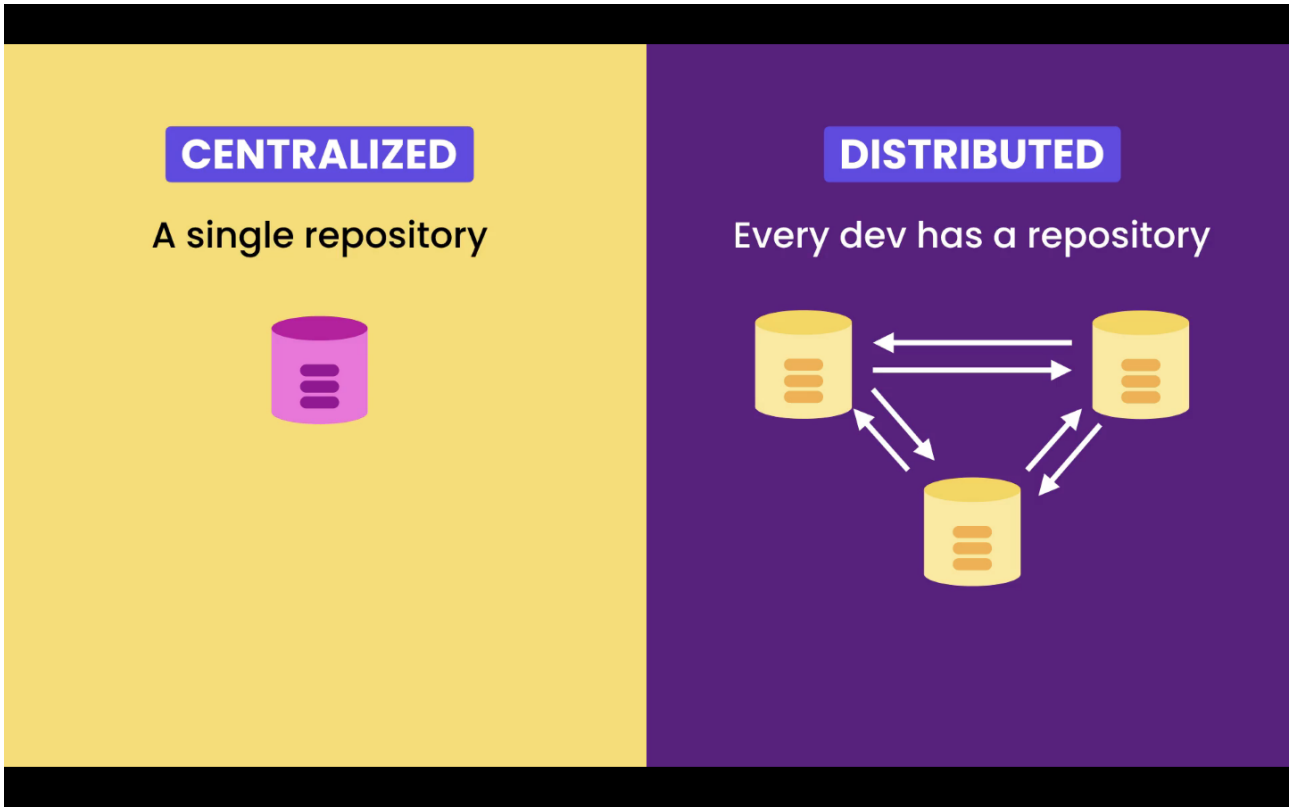
01- Introduction

In this section we will go through:

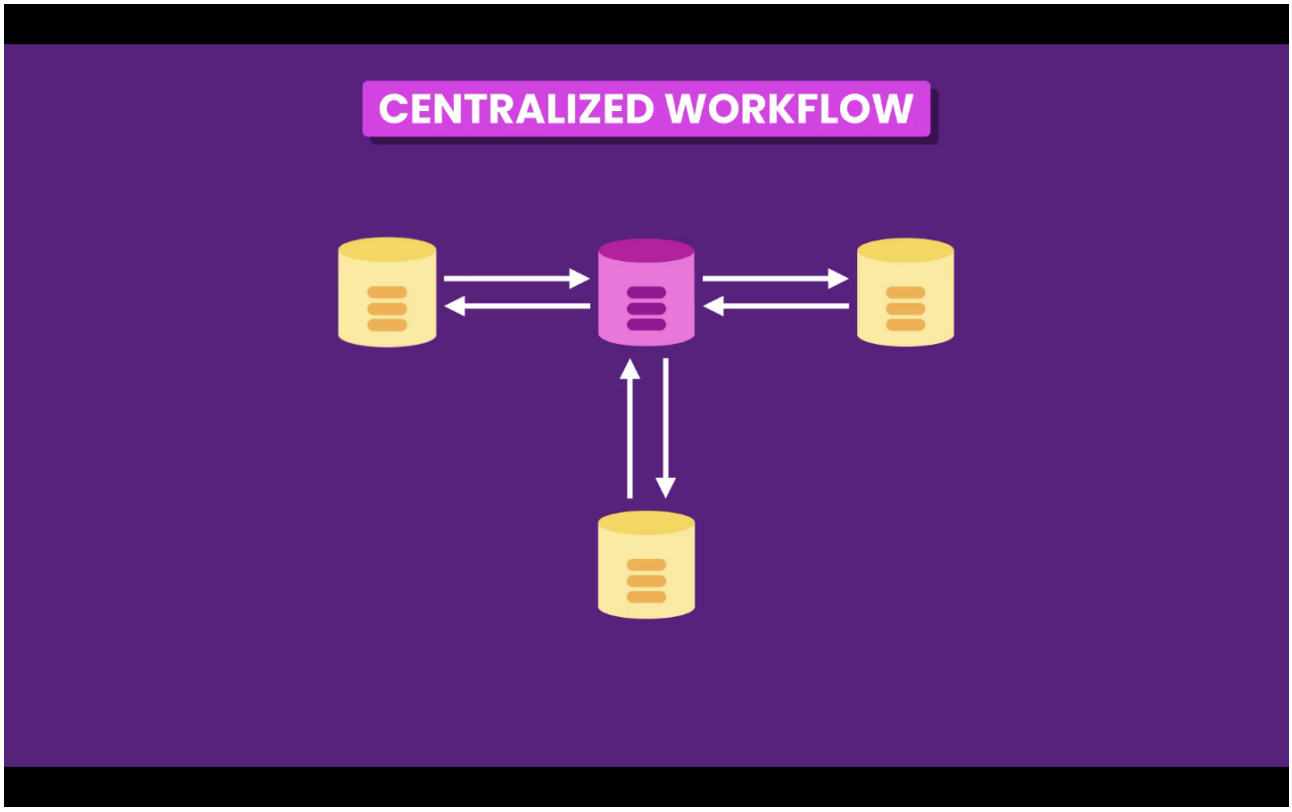
1. Collaboration workflows
2. Pushing, fetching and pulling
3. Pull requests, issues and milestones
4. Contributing to open-source projects.

02- Workflows

With a distributed VCS like Git it would be possible for each developer to synchronize his work with each member of the team.



But, usually, this is not the best solution, it's to complex and more susceptible to errors. Instead it is better to use, what is called, a **Centralized Workflow**, each developer still keeps a full local copy of the repository, but instead of syncing with each other they sync with a **Remote Repository**.



With this model we don't have a single point of failure. If the **Remote Repository** fails we still can sync the repository with each other.

Remote Repository examples:

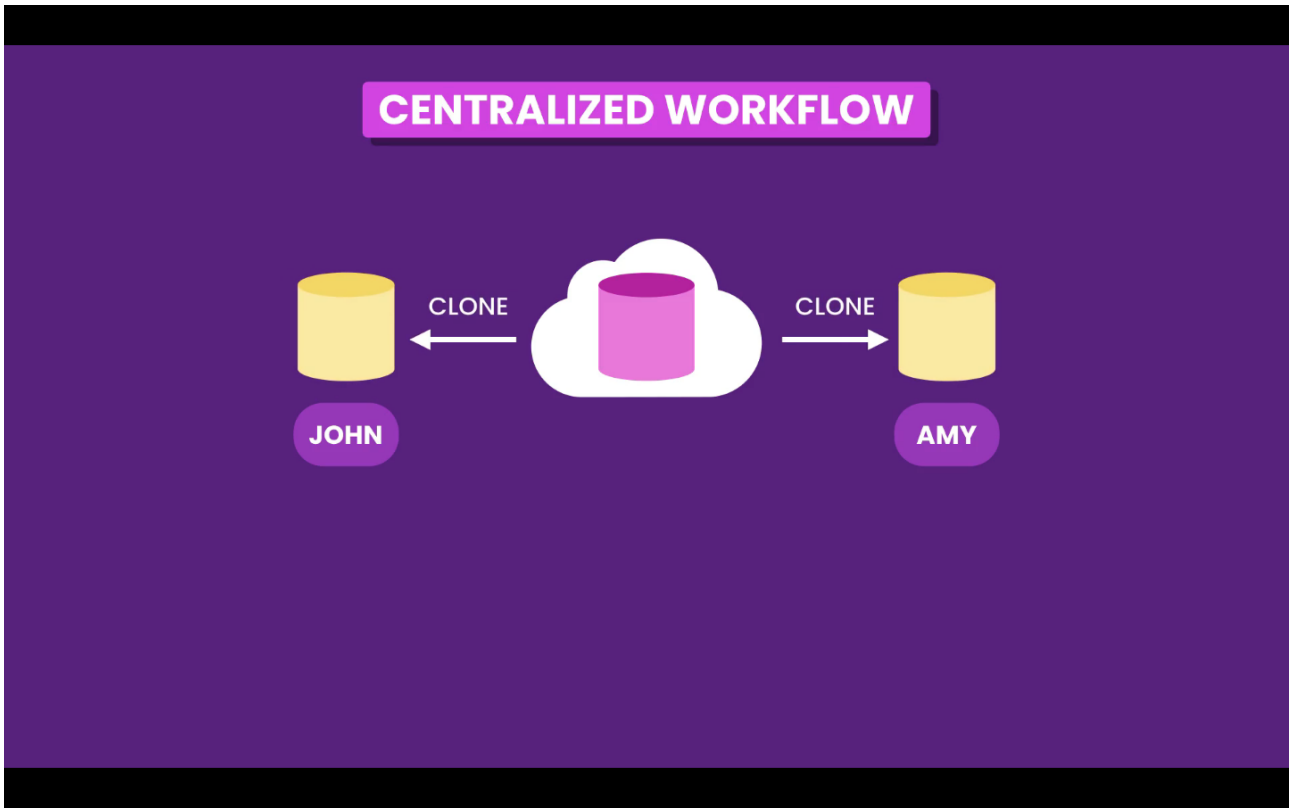
- Company private servers
- Cloud hosting services
  - GitHub
  - GitLab
  - Bitbucket
  - etc...

Centralized Workflow

The collaborating workflow usually follows this steps.

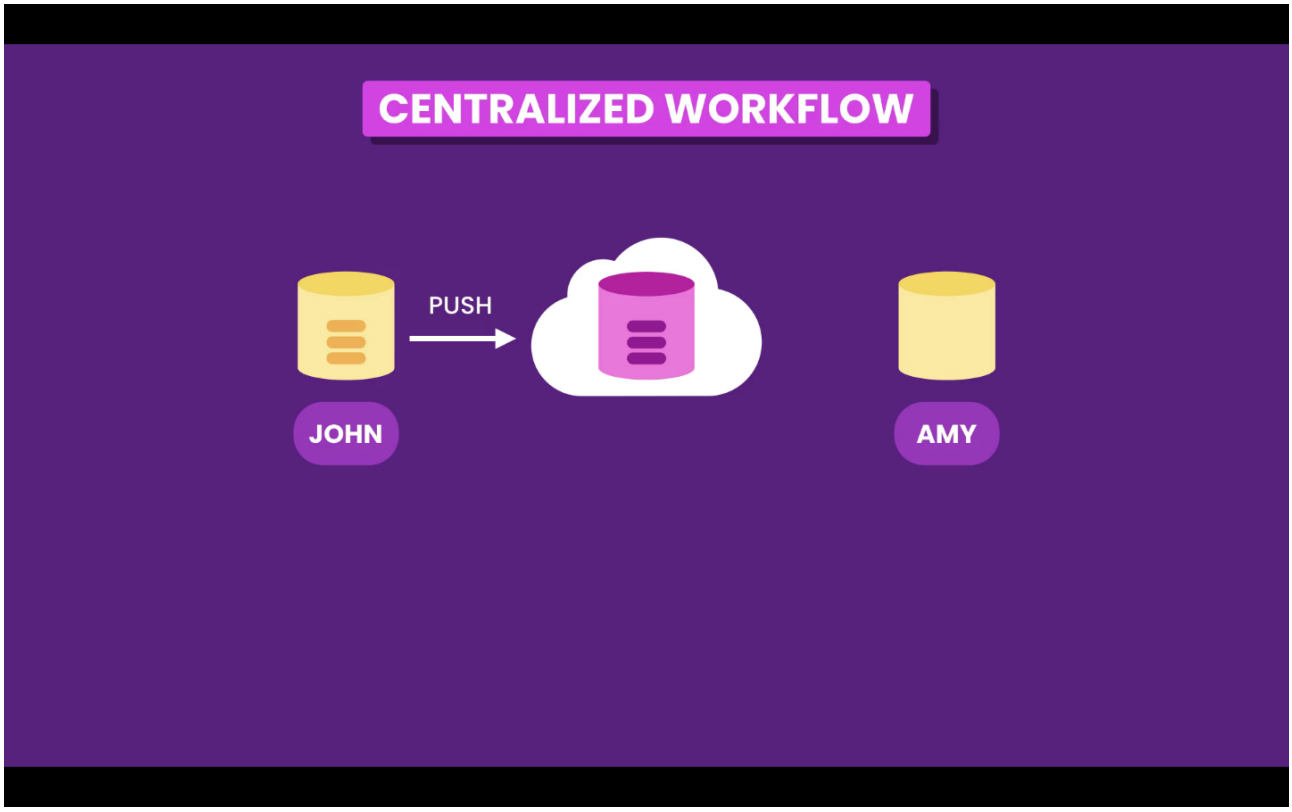
1. Clone the Remote Repository

First team members clone the repository from the **Remote Repository**. By cloning, tey will have a full copy of the repository on their machine.



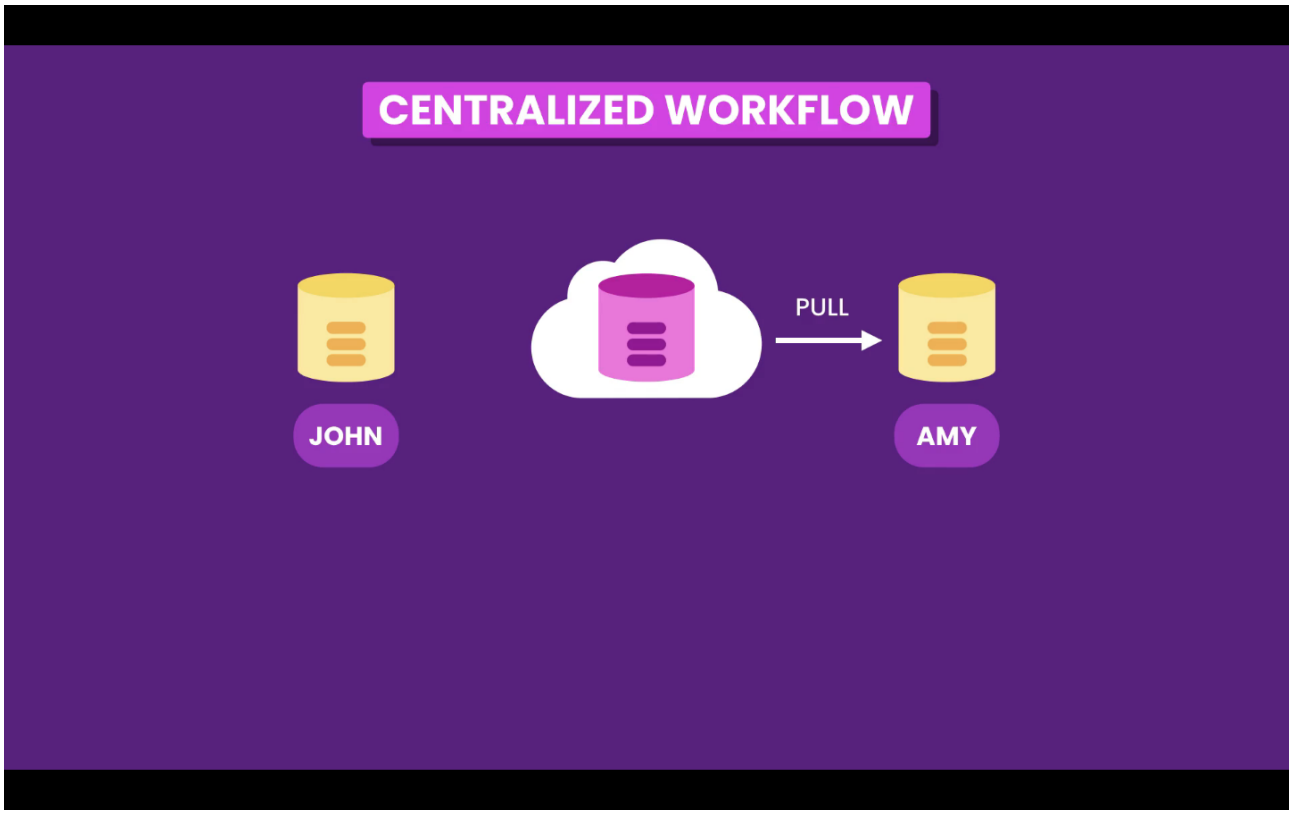
2. Push to Remote Repository

Team members start working and commit to their **Local Repository**. At any time they can use the `push` command to sync (upload) their work to the **Remote Repository**, so it is shared with team members.



3. Pull from Remote Repository

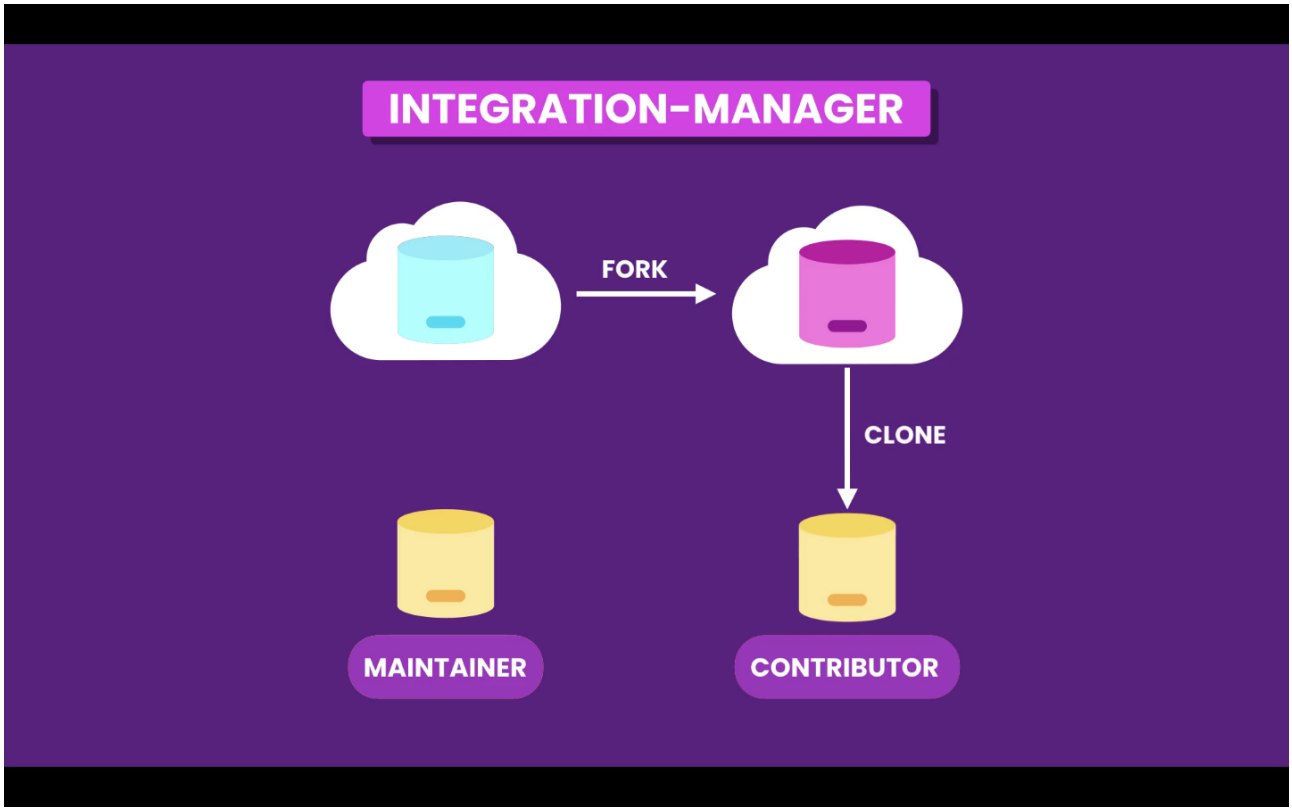
Any other team member can use the `pull` command to bring (download) new changes to their **Local Repository**.



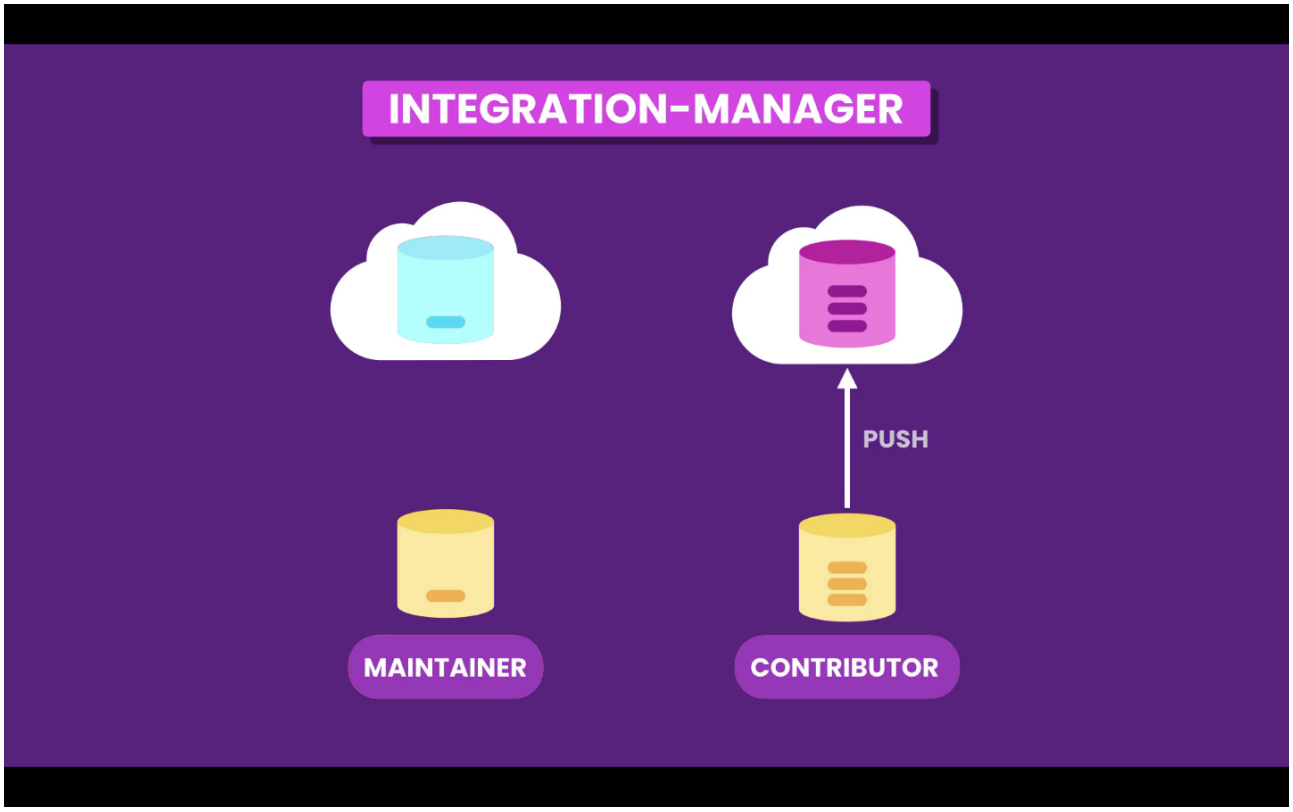
Workflow for open-source projects

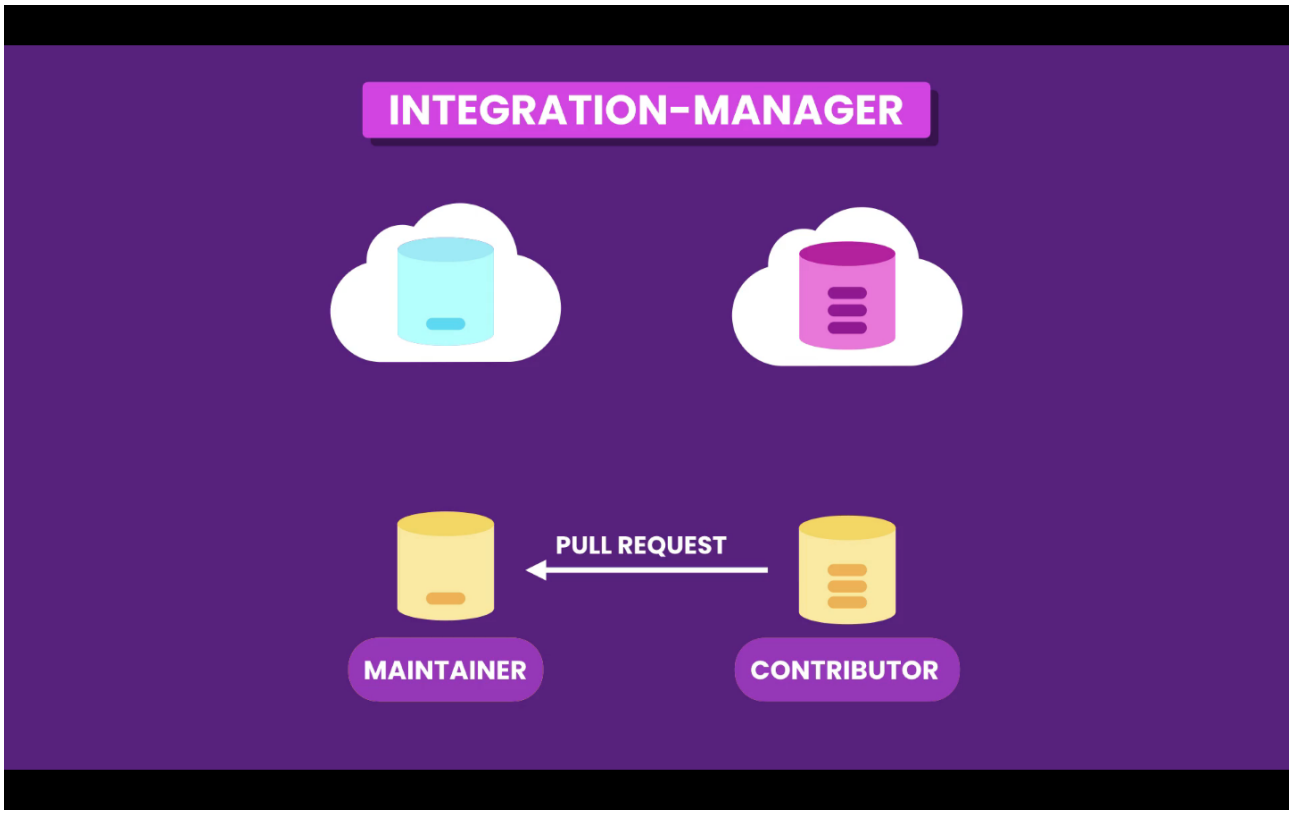
In a open-source project, usually only the maintainers have push access to the original **Remote Repository**. Other developers that want to contribute to the project first have to **fork** the **Remote Repository**, in order to have a copy of it.

Afterwards a contributor developer can clone the clone the repository and start working.



When he is done we can `push` is changes to the forked **Remote Repository**, and then send a **Pull Request** to the maintainers of the project. Who are notified, and can pull and review the changes. And if they agree with the changes they can merge them with the original **Remote Repository**.





05- Cloning a Repository

To clone a repository we need the repository url. For example:
`https://github.com/jmschp/mosh-ultimate-git-course.git` .

Then in our machine we run the command `git clone <url>`, this will create a local copy of the repository. This command only copies the ***main*** branch, which is the default branch, even if there are other branches in the **Remote Repository**.

```
❯ git clone https://github.com/jmschp/mosh-ultimate-git-course.git
Cloning into 'mosh-ultimate-git-course'...
remote: Enumerating objects: 489, done.
remote: Counting objects: 100% (489/489), done.
remote: Compressing objects: 100% (308/308), done.
remote: Total 489 (delta 236), reused 422 (delta 169), pack-reused 0
Receiving objects: 100% (489/489), 5.90 MiB | 1.71 MiB/s, done.
Resolving deltas: 100% (236/236), done.
```

Changing the default directory

When using the `clone` command Git will create a directory with the same name of the repository. In ths case `mosh-ultimate-git-course`. We can change it by passing a new name after the url `git clone <url> <my-folder>`

```
git clone https://github.com/jmschp/mosh-ultimate-git-course.git mosh-git
```

The above command will copy the repository to a new folder called `mosh-git`.

Remote Repository

When we clone a **Remote Repository**, in this case from GitHub, Git names this source repository ***origin***.

```
❯ git log --oneline --graph
* 0065a18 (HEAD -> main, origin/main, origin/HEAD) start new lesson
* 488cbba lesson complete
* 910d1d3 renamed folder
* a32bc1f style: removed white spaces
* 2d4c7af fixed merge conflict
* 5a090b1 rename file
```

Reference `origin/main`

The ***origin/main*** pointer, tell us where is the ***main*** branch in the **Remote Repository**. If we start to work and commit to the **Local Repository**, this one will move forward, but ***origin/main*** , will stay here it is until we push our work.

Technically this is called a remote tracking branch, we can not switch to it or commit to it.

List remote repositories

We can have more than one **Remote Repositories**, with the command `git remote` we can list all the **Remote Repositories** connected to our **Local Repository**.

```
❯ git remote -v
origin  https://github.com/jmschp/mosh-ultimate-git-course.git (fetch)
origin  https://github.com/jmschp/mosh-ultimate-git-course.git (push)
```

Using the `-v` option we get a more verbose output, showing more details. In this example we only hev one **Remote Repository**.

06- Fetching

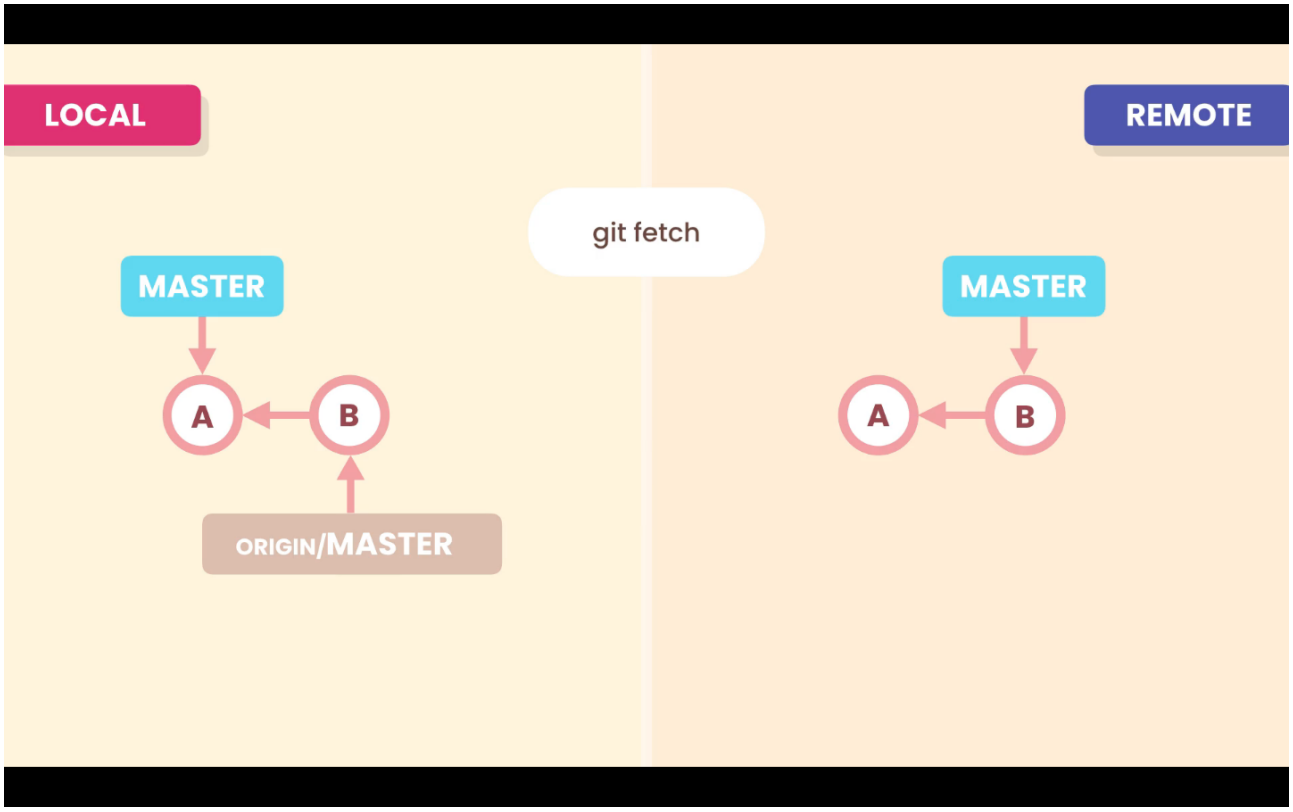
The **Local Repository** and **Remote Repository** work independently. If we have new commits in our **Remote Repository**, because another team member pushed is work, our **Local Repository** will not be aware of it.

We have to use the `git fetch <remote-repository>` command to download the new commits. When we do so the ***origin/main*** pointer will move forward, and point to the new commits. But our working directory will not be updated

```
git fetch origin
```

Optionally we can specify a branch to the `fetch` command, like `git fetch <remote-repository> <branch-name>` .

```
git fetch origin bugfix
```



To update our branch with the changes downloaded from the `fetch` command, we have to merge them with our branch, with the command:

```
git merge origin/master
```

Remote and Local branches

With the command `git branch -vv` we can se how the remote and local branches are diverging.

```
❯ git branch -vv
* main bbe3812 [origin/main: ahead 2] add details to lesson
```

In the above output from the `git branch -vv` we can see that our local ***main*** branch is connect to the remote ***origin/main*** branch. And the local branch is ahead by 2 commits.

07- Pulling

`git pull`

The `pull` command combines the `fetch` and `merge` together. With this command Git will download the commits in the **Remote Repository** branch and merge them with the **Local Repository** branch.

By default Git will perform a fast-forward merge if possible, if it is not possible like in the image above, git will run a 3-way merge.

`git pull --rebase`

With the command `git pull --rebase` Git will rebase **Local Repository** branch on top of the **Remote Repository** branch.

08- Pushing

With the `git push <remote-repository> <name-of-branch>` command we can send (upload) our changes to the **Remote Repository**.

```
❯ git push origin main
Enumerating objects: 30, done.
Counting objects: 100% (30/30), done.
Delta compression using up to 8 threads
Compressing objects: 100% (24/24), done.
Writing objects: 100% (25/25), 649.54 KiB | 19.68 MiB/s, done.
Total 25 (delta 11), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (11/11), completed with 3 local objects.
To https://github.com/jmschp/mosh-ultimate-git-course.git
   0065a18..a170331  main -> main
```

It is also possible to abbreviate this command and only run `git push`, by default Git assumes the **Remote Repository** ***`origin`***, and it will also assume teh current branch.

Reject push

In some situations our `push` may be rejected. For example if some team member pushed before us. So the **Remote Repository** and **Local Repository** histories have diverged.

To resolve this, first we have to `pull` the **Remote Repository** and merge the changes, and them we can `push` .

09- Storing Credentials

We can store the credentials for access to the **Remote Repository**, and tell git where to find them. For this we can set `credential.helper` in the configuration.

Cache

With the `cache` configuration Git will save the credentials for 15 minutes in memory.

```
git config --global credential.helper cache
```

MacOs

We can use macOS keychain, to permanently store our credential. First run the following commando to know if the macOS keychain helper is installed.

```
❯ git credential-osxkeychain
usage: git credential-osxkeychain <get|store|erase>
```

If it is installed we can set it, with the following command, if not Git will give us instructions on how to install it.

```
git config --global credential.helper osxkeychain
```

10- Sharing Tags

Push tag

By default the `push` command does not transfer tags to the **Remote Repository**. We have to explicit `push` them, with the command `git push origin <tag-name>` .

```
git push origin v1.0
```

Deleted pushed tag

To delete an already pushed tag from the **Remote Repository**, we use the command `git push origin --delete <tag-name>` .

```
git push origin --delete v1.0
```

This only deletes the tag from the **Remote Repository**, it will still be present in the **Local Repository**.

12- Sharing Branches

Push branch to Remote Repository

When we create a new branch it will only be available in our **Local Repository**. If we want to share our branches with team members we have to `push` them to the **Remote Repository**.

If we try to `push` a branch, that it is not in the **Remote Repository**, with `git push` , we will get an error.

For example I have created a branch named `bugfix` and tried to `push` it. Git will throw an error:

```
❯ git push
fatal: The current branch bugfix has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin bugfix
```

The error message ***`The current branch bugfix has no upstream branch.`*** means that this branch is not linked to a remote tracking branch in origin, if we run `git branch -vv` , we can see this.

```
❯ git branch -vv
* bugfix 9edbb2f lesson complete
  main   9edbb2f [origin/main] lesson complete
```

To set the remote tracking branch we run the command Git suggested `git push --set-upstream origin <name-of-branch>` , we only have to pass `--set-upstream` option the first time.

We can abbreviate the option `--set-upstream` to `-u` .

```
❯ git push -u origin bugfix
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1.56 KiB | 1.56 MiB/s, done.
Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'bugfix' on GitHub by visiting:
remote:      https://github.com/jmschp/mosh-ultimate-git-course/pull/new/bugfix
remote:
To https://github.com/jmschp/mosh-ultimate-git-course.git
 * [new branch]      bugfix -> bugfix
Branch 'bugfix' set up to track remote branch 'bugfix' from 'origin'.
```

And them again `git branch -vv` to see the result.

```
❯ gb -vv
* bugfix 9edbb2f [origin/bugfix] lesson complete
  main   9edbb2f [origin/main] lesson complete
```

Delete a branch from Remote Repository

To delete a branch from the **Remote Repository** we run `git push -d origin <name-of-branch>` . This will only detete the branch in the **Remote Repository**, it will still be available in the **Local Repository**.

```
git push -d origin bugfix
```

We can check with `git branch -vv` .

```
❯ gb -vv
* bugfix 133c99c [origin/bugfix: gone] add details to lesson
  main   9edbb2f [origin/main] lesson complete
```

13- Collaboration Workflow

For this example I have created a new branch directly on GitHub, we can use the `fetch` command to download this new branch.

```
❯ git fetch
From https://github.com/jmschp/mosh-ultimate-git-course
 * [new branch]      feature    -> origin/feature
```

When we run `fetch` we got a remote tracking branch. We can run `git branch` to list all the local branches, but this new fetched branch will not be displayed, because it is a remote tracking branch. We can see it with `git branch -r`

```
❯ git branch -r
  origin/HEAD -> origin/main
  origin/feature
  origin/main
```

Now we can create a new local branch that maps to this remote tracking branch. To do that we use the the following command, `git switch -c <local-branch> <remote-tracking-branch>` .

```
❯ git switch -c feature origin/feature
Branch 'feature' set up to track remote branch 'feature' from 'origin'.
Switched to a new branch 'feature'
```

After all team members have set up the branch in their local machine, they can collaborate in this branch.

If one team member deletes this branch from the **Remote Repository**, other team member will still have the the remote tracking branch in their machine. To remve remote tracking branches that are not in the **Remote Repository**, run `git remote prune <remote-repository>` .

```
git remote prune origin
```

20- Keeping a Forked Repository Up to Date

Add a Remote Repository

To keep a fork up to date we can add a remote repository linked to the original repository. To do that we use the `remote` command. With `git remote -v` we can list all the **Remote Repositories**. To add a new remote we use the `add` command, like so `git remote add <remote-name> <url>`. We can name the **Remote Repository** what ever we want, but in this situations is is usually called `upstream`.

```
git remote add upstream https://github.com/username/repository-name.git
```

Rename a Remote Repository

To rename a remote repository use the `rename` command, `git remote rename <remote-old-name> <remote-new-name>`.

```
git remote rename upstream base
```

In this example we rename the **Remote Repository** `upstream` to `base`.

Removing a Remote Repository

To remove a remote **Remote Repository** we use the `rm` command. `git remote rm <remote-name>`

```
git remote rm base
```

combined_readme.md

01- Introduction

In this section we will go through:

1. Why and when to rewrite history
2. Undo or revert commits
3. Use interactive rebasing
4. Recover lost commits

02- Why Rewrite History

Why do we need the history?

With Git history we can see:

- What was changed
- Why it was changed
- When it was changed

Bad commits

Bad commits make it difficult to read the project history.

- Poor commits messages, with no meaning
- To large commits, with unrelated changes
- To small commits, scattered all over the history

We need a clean history to be ale to see how our project evolved from day one.

Tools

To make our history cleaner we can use several Git operations.

- Squash small, related commits
- Slipt large commits with unrelated changes
- Reword commits messages
- Drop unwanted commits
- Modify the content of a commit

03- The Golden Rule of Rewriting History

Don't rewrite public history

This rules means that commits that have been pushed to a public repository, and shared with other developers should not be modified.

04- Example of a Bad History

As an example of bad history:

```
❯ git log --oneline --graph
* 088455d (HEAD -> master) .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

Commit:

- `6fb2ba7 Render restaurants the map.` -> Wording issue should be `"...restaurants on the map."``
- `af26a96 Fix a typo.` -> We shouldn't have a typo in the first place, commits like these pollute the history. So we can combine them with other commit.
- `8527033 Change the color of restaurant icons.` -> This part of the of the same line of work in the restaurants, So all this commits should be combined.
- `8441b05 Add a reference to Google Map SDK.` -> The problem here is that if we checkout commit `6fb2ba7`, our application is not going to work, because the Google

Map SD reference comes afterwards. We should either move this commit down before `6fb2ba7` , or combine both commits.

- `72856ea WIP` -> A "Work In Progress" commit it is just a noisy commit, we Should either drop it, change the message or combine it.
- `111bd75 Update terms of service and Google Map SDK version.` -> Ideally we should separate these commit in tow commits, Because updating terms of service, as nothing to do with updating google map SDK.
- `088455d (HEAD -> master) .` -> A mysterious message commit, we should either drop it or change this message.

05- Undoing Commits

As we seen before if we have pushed a commit to a public **Remote Repository** we should not remove it.

```
❯ git log --oneline --graph
* 088455d (HEAD -> master) .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

For example if we want to undo the last commit `088455d (HEAD -> master) .` we have two options:

1. Revert commit -> The has been pushed to a public **Remote Repository**
2. Reset commit -> The has not been pushed to a public **Remote Repository**

Reset commit

Resetting a commit removes this commit from the history. We use the `reset` command and we have to give it the target commit. We can do that using the `HEAD~n` syntax, which means `n` commits back from the commit `HEAD` is pointing to usually the last commit.

```
git reset --hard HEAD~1
```

Options to the reset command

We have the following options:

- `--soft` -> Removes the commit only
- `--mixed` -> Unstages files
- `--hard` -> Discards local changes

For example, let's say we have two commits in our **Local Repository** `A` and `B`, and `B` is the last commit because `HEAD` is pointing to it. In the **Staging Area** and **Working Directory** we have the same code as in the last snapshot (**Local Repository**).

Option --soft

The `--soft` option only changes the **Local Repository**. Removes the commit only.

If we use `git reset --soft HEAD~1`, Git will point `HEAD` to the target location, in the **Local Repository**, but it is not going to touch the **Staging Area** and **Working Directory**.

This is the same state as before we committed `B`. We have some changes in the **Staging Area** and **Working Directory**, that have not yet been committed.

Option --mixed

The `--mixed` option will change the **Local Repository** and the **Staging Area**. We go one step back, it will unstage the changes.

With the `--mixed` option Git will move the `HEAD` pointer, from `A` to `B` as in the `--soft` option, and will put the last snapshot in the **Staging Area** as well, but it will not touch the **Working Directory**.

This is the same state as before we staged our changes. We have some changes in the **Working Directory**, that have not yet been staged.

Option --hard

The `--hard` option will change the **Local Repository**, **Staging Area**, and the **Working Directory**. Even one more step back, discards local changes.

With the `--hard` option Git will move the `HEAD` pointer, from `A` to `B` as in the `--soft` option, and it will put the last snapshot in the **Staging Area** and the **Working Directory**. So the new changes in the **Working Directory** are gone.

06- Reverting Commits

In case we have pushed our commits to a public **Remote Repository** we should not use the `reset` command. We should use the `revert` command, this will create a new commit base on our target commit.

Revert one commit

To revert a single commit we pass that commit to the revert command. Either by the commit ID or by the `HEAD~n` syntax. This will create new commit.

History before revert:

```
❯ git log --oneline --graph
* 088455d (HEAD -> master) .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

```
❯ git revert HEAD~1
[master 1b7aed7] Revert "WIP"
 1 file changed, 1 deletion(-)
```

History after revert:

```
❯ git log --oneline --graph
* 1b7aed7 (HEAD -> master) Revert "WIP"
* 088455d .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
```

```
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

The above command reverts the last commit, so it will create a new commit undoing the changes made by the last commit.

Revert a range of commits

We can revert a range of commits using the `..` notation, like:

```
git revert HEAD~4..HEAD
```

or

```
git revert 8441b05..1b7aed7
```

Note that the first commit `HEAD~4` or `111bd75`, in the above example are not included.

This operation will create a new commit for each commit reverted.

History before revert:

```
❯ git log --oneline --graph
* 088455d (HEAD -> master) .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

History after revert:

```
❯ git log --oneline --graph
* 645357e (HEAD -> master) Revert "WIP"
* 685ec4e Revert "Update terms of service and Google Map SDK version."
```

```
* 6a65dda Revert "WIP"
* a0262f0 Revert "."
* 088455d .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

Range revert with single commit `--no-commit`

Instead of creating one new commit for each reverted commit (which might pollute the history) we can use the option `--no-commit`, that will create only one commit, for all the reverted commits. With this options Git will add the required changes to the **Staging Area**, for each reverted commit.

```
git revert --no-commit HEAD~4..HEAD
```

If we run `git status`, we can see the changes of the reverse of the last 4 commits.

```
❯ git status
On branch master
You are currently reverting commit 72856ea.
  (all conflicts fixed: run "git revert --continue")
  (use "git revert --skip" to skip this patch)
  (use "git revert --abort" to cancel the revert operation)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   map.txt
    modified:   package.txt
    deleted:    terms.txt
```

If we are happy with the changes we use the `--continue` option.

```
git revert --continue
```

Or if we want to abort we use th `--abort` option.

```
git revert --abort
```

History after revert:

```
❯ git log --oneline --graph
* b7ed3ee (HEAD -> master) Revert last 4 commits
* 088455d .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

07- Recovering Lost Commits

With the `reflog` command we can see a log of how a reference (or pointer) as moved in our history. If we do not supply any options we will see how the `HEAD` pointer moved in history. `git reflog <reference>`

```
❯ git reflog
088455d (HEAD -> master) HEAD@{0}: reset: moving to HEAD~1
b7ed3ee HEAD@{1}: commit: Revert last 4 commits
088455d (HEAD -> master) HEAD@{2}: reset: moving to HEAD~4
645357e HEAD@{3}: revert: Revert "WIP"
685ec4e HEAD@{4}: revert: Revert "Update terms of service and Google Map SDK version
6a65dda HEAD@{5}: revert: Revert "WIP"
a0262f0 HEAD@{6}: revert: Revert "."
088455d (HEAD -> master) HEAD@{7}: reset: moving to
[...]
088455d (HEAD -> master) HEAD@{27}: commit: .
f666091 HEAD@{28}: commit: WIP
111bd75 HEAD@{29}: commit: Update terms of service and Google Map SDK version.
72856ea HEAD@{30}: commit: WIP
8441b05 HEAD@{31}: commit: Add a reference to Google Map SDK.
8527033 HEAD@{32}: commit: Change the color of restaurant icons.
af26a96 HEAD@{33}: commit: Fix a typo.
6fb2ba7 HEAD@{34}: commit: Render restaurants the map.
70ef834 HEAD@{35}: commit (initial): Initial commit
```

Every entry of the `reflog` command starts with the commit `HEAD` is point to after the operation, then a unique identifier. In this example we have `HEAD@{0}` for the first entry, `HEAD@{1}` for the second entry, and so on. In front of the identifier we see what happened.

Commit ID	Identifier	Message
645357e	HEAD@{1}	reset: moving to HEAD~4

So we can revert any of theses operations to recover lost commits. In this example the operation `HEAD@{0}` is resetting `HEAD~1` . We can recover the lost commit with the `reset` command. We can use the identifier or the the commit ID

History before:

```
❯ git log --all --oneline --graph
* 088455d (HEAD -> master) .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

```
❯ git reset --hard HEAD@{1}
HEAD is now at b7ed3ee Revert last 4 commits
```

History after:

```
❯ git log --all  --oneline --graph
* b7ed3ee (HEAD -> master) Revert last 4 commits
* 088455d .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

08- Amending the Last Commit

In situations where we made a mistake in the last commit, like a typo in the message, or adding a file that shouldn't be there, we can amend the commit. We don't actually modify the last commit, in really Git creates a new commit, because git commits are immutable.

Amend a commit message or include changes

With the `--amend` option we can add more changes to the last commit or modify the commit message. In case we need to add more changes in the code to the last commit, first we need to stage them and the run `git commit --amend -m <message>`. Optionally we can omit the message and accept the last message, so run `git commit --amend` will open the default editor with the previous commit message.

```
git commit --amend
```

Remove a file from the last commit

To remove a file from the last commit, first we need to use the `reset` command with the `--mixed` option, `git reset --mixed HEAD~1`. With this command Git will unstage our changes, but the **Working Directory** will not be affected.

Then we can restage the changed we need and perform a normal commit.

```
git reset --mixed HEAD~1

git add <files>

git commit -m <commit message>
```

Here we do not use the `--amend` option, because we have reseated the `HEAD`, so the last commit is not there anymore.

09- Amending an Earlier Commit

To amend an earlier commit we use interactive rebasing. With rebasing we can replay other commits on top of a commit. First we choose the commit we need to amend and pass it parent to the `rebase` command with the `-i` option, like `git rebase -i <commit>`.

We can modify one or more commits. Git will recreate each commit that goes through the rebase operation even if the are not edit.

In the above image suppose we only changed commit `B`, Git will recreate `C` and `D` to point not the new `B` commit.

Rebasing is a destructive operation because it rewrites history.

The `-i` option means we are going to interact with the rebase operation, stop it, make changes, continue it or abort it.

This command will open de default editor with a script, listing all the commits we need to rebase, and instructions to perform the `rebase` operation. Just like the following example:

```
pick 8f1440a Add a reference to Google Map SDK.
pick 098a4bc Render restaurants the map.
pick 0bf60fe Fix a typo.
pick dd8f07e Change the color of restaurant icons.
pick ba55176 Update terms of service and Google Map SDK version.
pick f820221 WIP
pick 65dbb96 .

# Rebase 70ef834..65dbb96 onto 70ef834 (7 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
```

```
# However, if you remove everything, the rebase will be aborted.
#
```

After configuring the script the rebase operations starts.

```
❯ git rebase -i 8527033
Stopped at 8441b05...  Add a reference to Google Map SDK.
You can amend the commit now, with

  git commit --amend '-S'

Once you are satisfied with your changes, run

  git rebase --continue
```

Here we can make the necessary changes, and then amend the commit. when we are done we run `git rebase --continue`, to continue the operation.

To abort the rebase operation in any point use the `--abort` option. `git rebase --abort`.

In a rebase operation, a change introduced in a earlier commit will be carried on thought the history.

10- Dropping Commits

To drop commits we use interactive `rebase` operation. So we have to pass to the `rebase` operation the parent of the commit we want to drop, like `git rebase- i <parent-commit>`.

In interactive `rebase` operations conflicts might arise. For this example we want to drop the commit `72856ea`, but this commit introduces a new file that is used in the next commit `111bd75`, so this will provoque a conflict.

```
❯ git log --oneline --graph --all
* b7ed3ee (HEAD -> master) Revert last 4 commits
* 088455d .
* f666091 WIP
* 111bd75 Update terms of service and Google Map SDK version.
* 72856ea WIP
* 8441b05 Add a reference to Google Map SDK.
* 8527033 Change the color of restaurant icons.
* af26a96 Fix a typo.
```

```
* 6fb2ba7 Render restaurants the map.
* 70ef834 Initial commit
```

To select the parent of commit `72856ea`, we can use several syntaxes, `72856ea~1`, or `72856ea^`. When we run the interactive `rebase` operations, and select `drop` in the script operation, we will be prompt with the conflict warning message.

```
❯ git rebase -i 72856ea~1
CONFLICT (modify/delete): terms.txt deleted in HEAD and modified in 111bd75 (Update
error: could not apply 111bd75... Update terms of service and Google Map SDK version
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 111bd75... Update terms of service and Google Map SDK version.
```

In the short status we can see the `terms.txt` file as two changes `D` for `deleted`, because the rebase operations drops the commit where the file was introduced, and `M` for `modified`, because these file is modified in the next commit.

```
❯ git status -s
M  package.txt
DU terms.txt
```

To resolve this conflict we use the `mergetool` command.

```
❯ g mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
tortoisemerge emerge vimdiff nvimdiff
Merging:
terms.txt

Deleted merge conflict for 'terms.txt':
  {local}: deleted
  {remote}: modified file
Use (m)odified or (d)eleted file, or (a)bort?
```

In this case we want the `m` option for modified. Then we continue the rebase operation, with `git rebase --continue`.

11- Rewording Commit Messages

To reword commits we also use the interactive `rebase` operation, and in the script choose the `reword` option.

For each commit we want to reword, Git will open the default editor and give us a chance to rewrite the commit message.

12- Reordering Commits

To reorder commits we use the interactive `rebase` operation, all we have to do is change the commits order in the script's operation.

13- Squashing Commits

Squash rebase

To squash commits together we use the interactive `rebase` operation. In the rebase script we select the `squash` option only for the child commits, the parent commit keeps the `pick` option. All commits that need to be squash have to be in sequence, so if needed we can reorder them first.

```
pick 098a4bc Render restaurants the map.
squash 0bf60fe Fix a typo.
squash dd8f07e Change the color of restaurant icons.
pick ba55176 Update terms of service and Google Map SDK version.
pick f820221 WIP
pick 65dbb96 .
```

In the above example commits `098a4bc` , `0bf60fe` and `dd8f07e` , will be combined together. After combined the commits, the rebase operation will open the default editor to edit the commit message.

Fixup rebase

The `fixup` rebase option works like `squash` option, but instead of allowing us to edit the commit message, it will use the message form the parent squashed commit, in this case the message from commit `098a4bc` .

```
pick 098a4bc Render restaurants the map.
pick fixup Fix a typo.
pick fixup Change the color of restaurant icons.
pick ba55176 Update terms of service and Google Map SDK version.
pick f820221 WIP
pick 65dbb96 .
```

14- Splitting a Commit

To split a commit we use the interactive `rebase` operation. We pass to the rebase operation the parent of the commit we need to split. In the script we select the `edit` option for the commit we need to split.

Then in the `rebase` operation we unstage the changes of the commit with `git reset --mixed HEAD^` . Now the changes applied in this commit ar in our **Working Directory**.

With the unstaged changes we slipt the commit to our needs with normal `git add <file>` and `git commit -m <message>` . This will create new commits. Then continue the `rebase` operation to finish `git rebase --continue` .