Git: basics

Agenda

1. Repositories and changes
2. Branches and timelines
3. Section name

Git

Git is a distributed version control system to handle all kind of projects.

It is so popular that had become the *de facto* industry standard system to version control software projects.

Version control means that it can handle the history of changes that happen to a project, keeping track of all the details about what, when and who performed them. Also, it allows to branch the history and to manage the merging of conflicting changes.

Distributed means that the repository is stored not only in a central place, but a copy of the code exists in all the developer's computers.

What is version control

Repositories and changes

Repository

The place where you store your work.

At first, it’s a **folder on your computer.** When you add version control, it becomes a repository and can keep track of the changes you make.

Getting a Git Repository

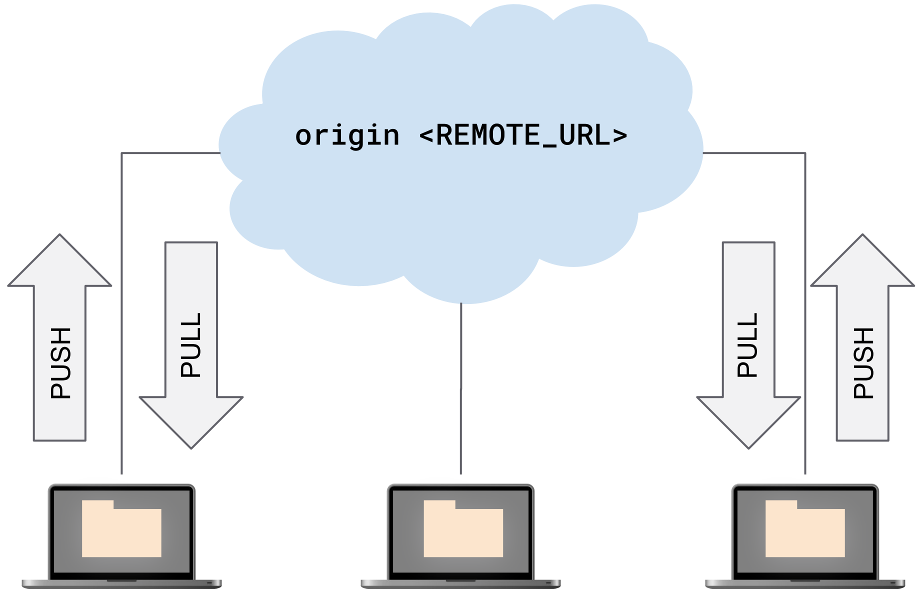
```
// create a folder

mkdir myProject

// move into the folder

cd myProject

// set up the repository

git init

// set up the repository renaming
master branch to “main”

git init -b main
```

codurance

Remote

A shared repository that is synchronized with the local one and allows a team working together.

Remote repositories can be provided by third parties (Github, Gitlab, BitBucket...) or you can setup your company git server.



Add locally hosted code to remote github

codurance

Changes

Changes are modifications to the contents of the repository.

They can be of three types:

- Addition of files
- Deletion of files
- Modification of files

(Empty folders don’t count)

You need to tell git to put an eye on them, tracking the files.

Recording changes

```
// after making some changes…

// stage the files for tracking

git add <changed files>

// stage all the changes

git add .

// add changes under a folder

git add src/folder

// add all changes and stage untracked
files

git add -all
```

codurance

Changes

Whenever a file changes, it will be considered changed, but it's not until you stage the change that it will be taken into account in order to "remember" it. In fact, if you continue modifying a concrete file after it is added, **the new content won't be part of the next commit**, you must add them again to register the new changes.

Changes are **provisional**, so to speak. To fix a point in the git history you must **commit** the changes you've made.

```
// after making some changes…
// stage the files for tracking
git add <changed files>


// stage all the changes
git add .
// add all changes and stage untracked files
git add --all
```

Commit

A commit is a point in the history of a repository.

A commit is a way to register a set of changes in the versioning system. It can range from the tiniest change on a single file to dozens affecting several files. A commit makes permanent those changes.
It has a name or id that identifies it uniquely and can have a message that describes the meaning of the changes.

Commit

```
// Make changes permanent

git commit -m “Message describing the
set of changes”

// Stage all the changes and make
changes permanent

git commit -a -m “Message”
```

codurance

What’s in a commit

A commit marks a point in the history giving it a unique identifier

Commit message is a human readable label, but what uniquely identifies an individual commit is a SHA1 hash.

Commit contains information about all the changes. Also, it stores a pointer to the previous commit.

Anatomy of a git commit (internals)

```
// Make changes permanent

git commit -m “Message describing the
set of changes”

// Stage all the changes and make
changes permanent

git commit -a -m “Message”
```

Working with existing remote

You clone an existing remote repository to have a local one linked to the remote and start contributing to it.

```
// Get the repository REMOTE_URL

git clone git@github.com:user/project.git

// A folder is created for project

cd project

// You're all set
```

Create a remote from local repository

You link your local repository with the remote, in order to share changes among teammates.

Add locally hosted code to remote github

```
// Setup project in remote server and get
the repository REMOTE_URL

// In your local project

// Set the remote

git remote add origin  <REMOTE_URL>

// Verify

git remote -v

// Send local committed changes to remote

git push -u origin main
```

Getting changes from remote

Working with a team you will need to update your local repository with the changes published by your teammates.

```
// Get all the changes, merging them
with your current work

git pull

// Get the changes, without actually
merging them

git fetch

// After that you will need to merge
explicitly

git merge origin/<branch_name>
```

Syncing with git

Sharing changes with remote

After committing some changes, you probably want them to be shared with your team.

```
// Make some changes

// Stage the changes

git add <changes>

// Fix the point in repo history

git commit -m “My important feature”

// Send local committed changes to
remote repository

git push -u origin the_branch

git push
```

Syncing with git

codurance

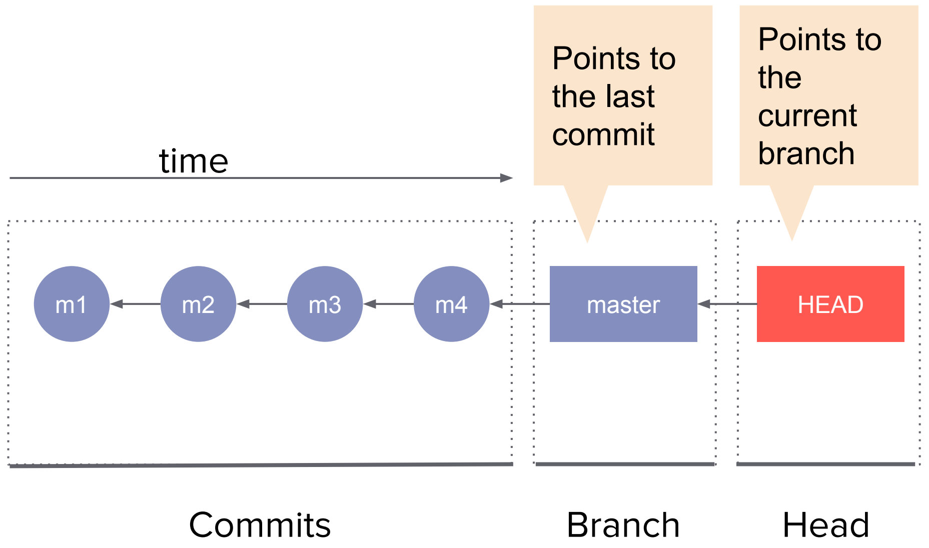
Branches and timelines

Branch

The sequence of commits is the history of your project. From the beginning this history happens in a branch named master (by default).

A pointer to the current commit, or point in the history, is named HEAD.

But you can use other branches too.



Syncing with git

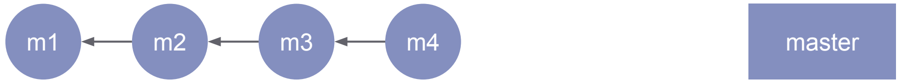
codurance

Branches

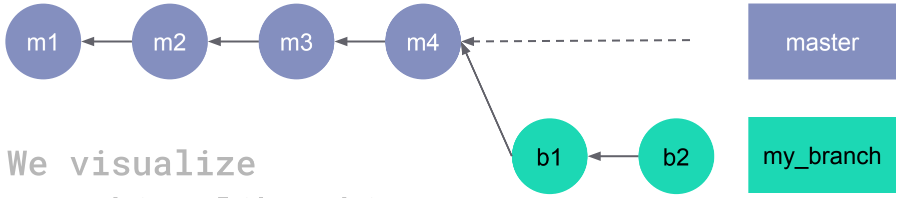
Many times you'll want to explore things or work on your project without having to mess around with the work you have done until that moment. In that case, a good thing to do is to create a branch.

A branch is a deviation from the course of history reflected by master. The master branch and the new one can evolve differently from that point in history.

You have these commits in master branch



```
// Start branch my_branch and jump into it
git checkout -b my_branch
```



We visualize something like this

codurance

What is a branch

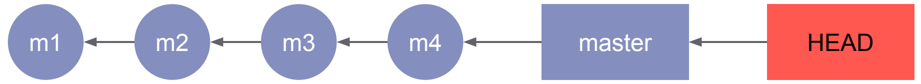
Internally, branches are pointers to commits.

Each commit also points to its parent commit, so git can figure out the full history.

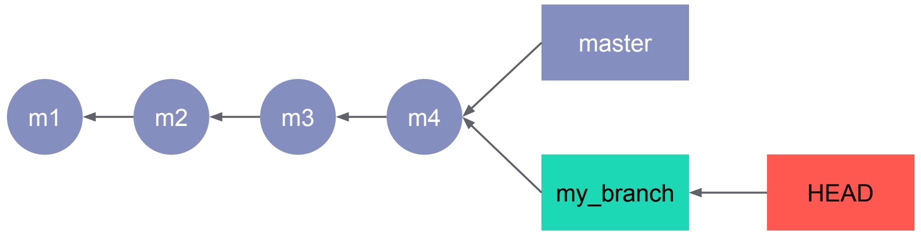
There is an special HEAD pointer that points to the currently checked out branch (or commit) and changes when we checkout another branch.

Understanding branches in Git

But this is what really happens…



```
// Start branch my_branch and jump into it
git checkout -b my_branch
```



codurance

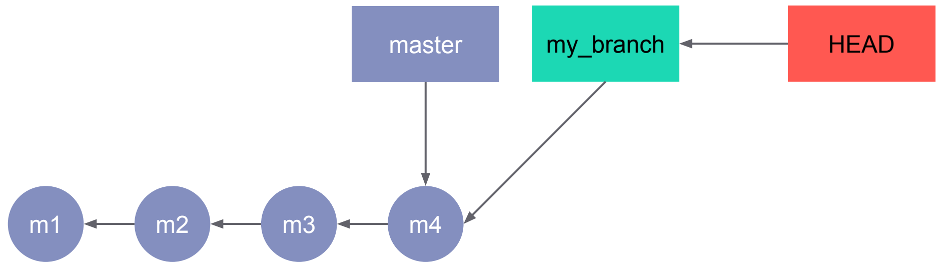
Committing changes to a branch

You can add commits to the checked out branch.

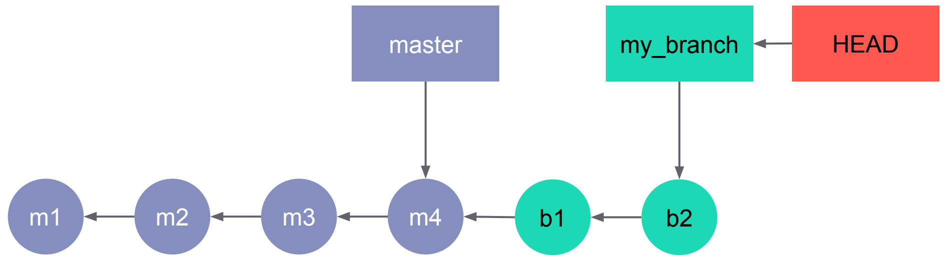
When you add commits to a branch, they are not present in other branches until you merge them.

Understanding branches in Git

```
git checkout my_branch
```



```
git commit -m “My changes”
```

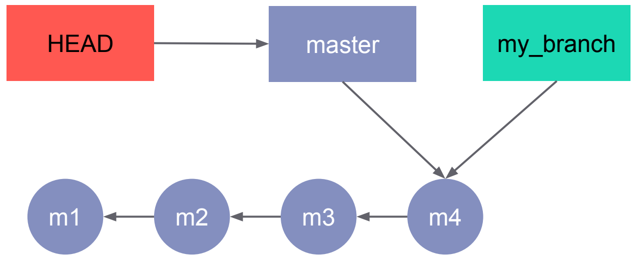


Moving between branches

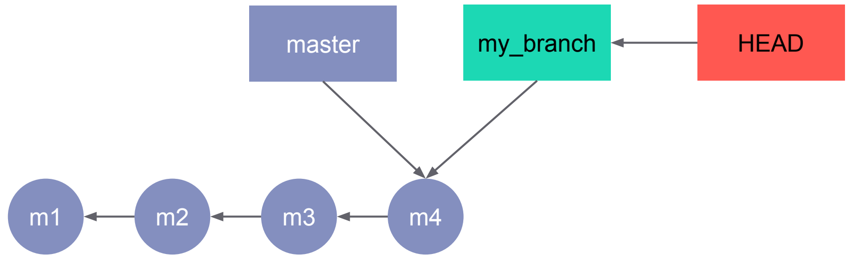
You can move to different branches to see a different timeline.

The HEAD pointer simply changes to the desired branch.

Understanding branches in Git



```
// Move to another branch

git checkout my_branch
```



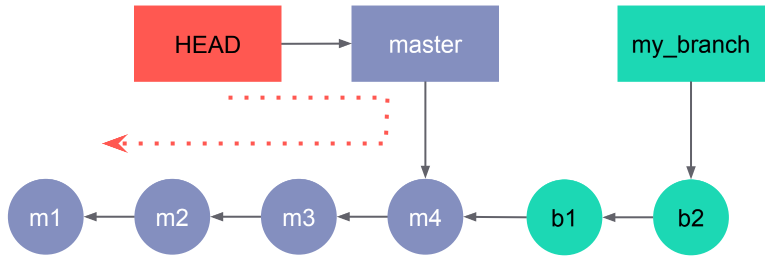
Moving between branches (2)

When you checkout another branch you don’t loose the commits in the previous one, you only stop seeing them.

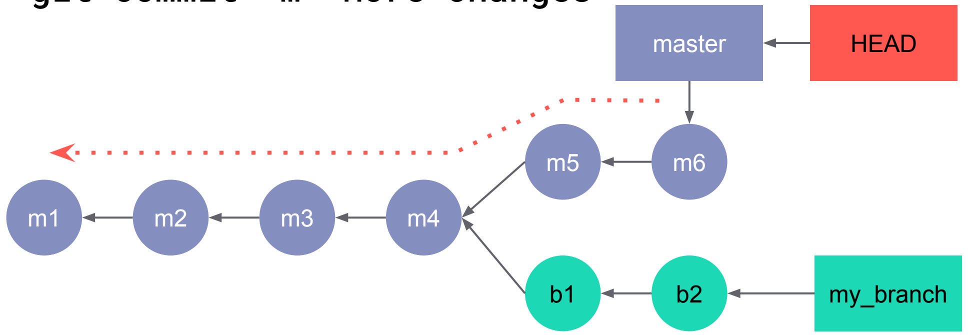
The HEAD pointer changes. You can follow the arrow from HEAD to know which commits are visible now.

Understanding branches in Git

```
git checkout master
```



```
git commit -m “More changes”
```



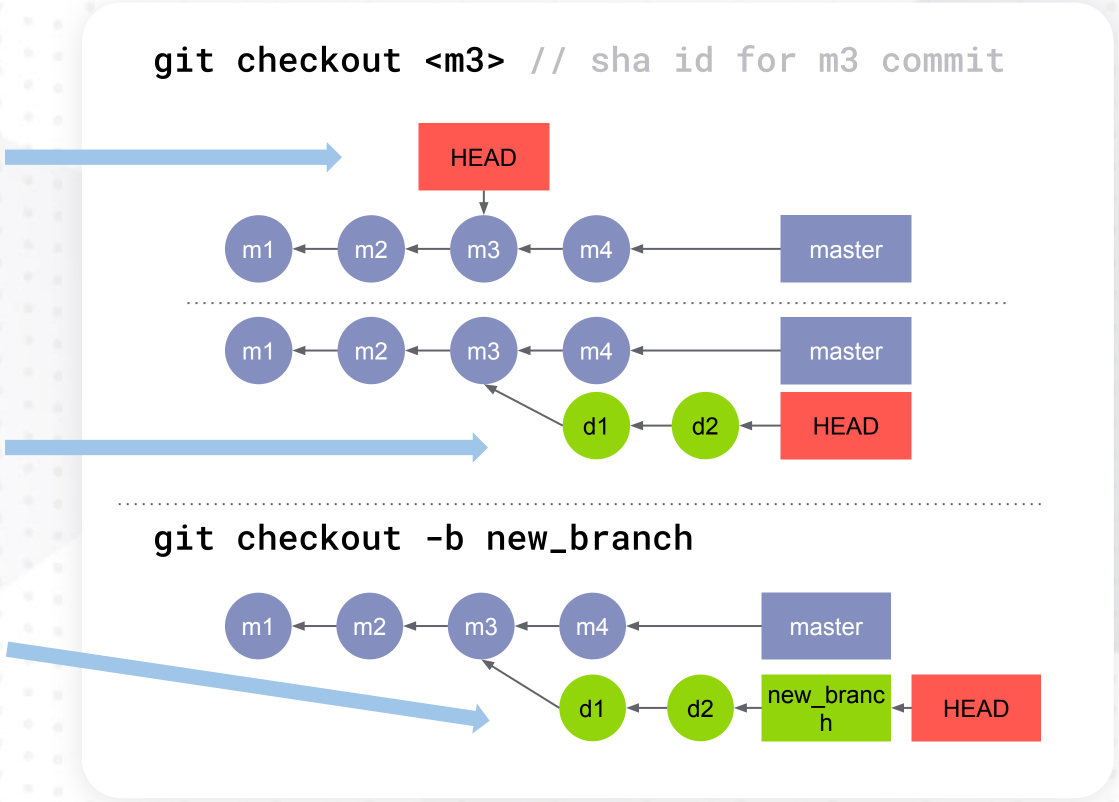
Understanding Detached HEAD

Detached HEAD happens when the HEAD points to a specific commit, not a branch.

You can add commits when detached.

And create a new branch from them to stop being detached.

Understanding branches in Git



```
git checkout <m3> // sha id for m3 commit
```

```
git checkout -b new_branch
```