Git DevOps and Shared Development Workflows

Marco Melis

marco.melis@unica.it

https://www.linkedin.com/in/melismarco/

Industrial Software Development A.Y. 2020-2021
M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

October 2020

Outline

About Myself

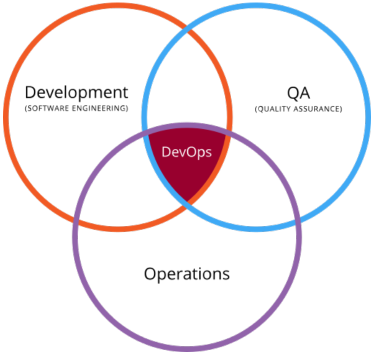
- M.Sc. Electronic Engineering @ UNICA (April 2017)
- PhD student @ UNICA on Adversarial and Explainable Machine Learning
- +5 years of study on DevOps and Shared Development Workflows
- DevOps and Development Workflows consultant @ PRALab since 2014

DevOps

Development (Dev) and Software Operation (Ops)

DevOps

DevOps is a software engineering culture and practice that aims at unifying software development (Dev) and software operation (Ops)



DevOps

As per Professors Bass, Weber, and Zhu definition, DevOps is:

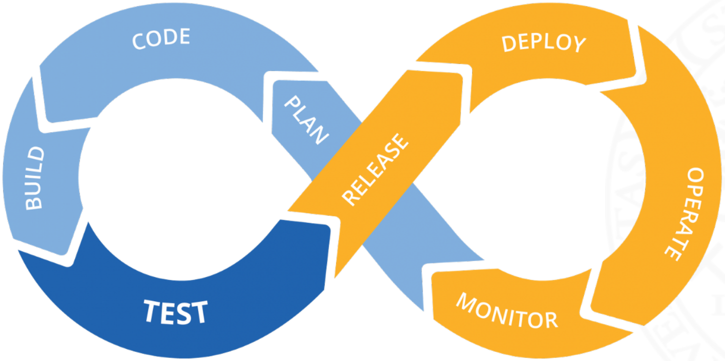
- "a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality."[1]

DevOps is a cross-functional set of toolchains

A toolchain is the set of programming tools that is used to
automatically
perform a specific simple or advanced task

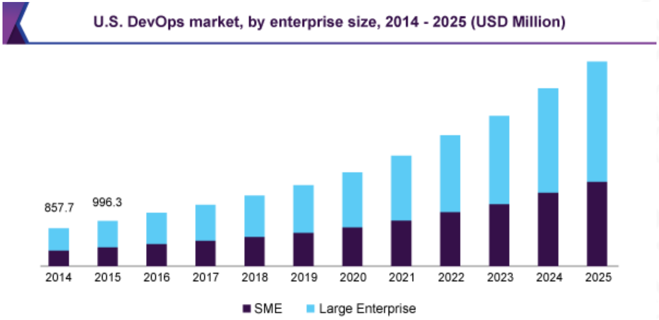
[1]Bass; Weber; Zhu. DevOps: A Software Architect's Perspective.

DevOps Toolchain



Why study DevOps?

You can find work!

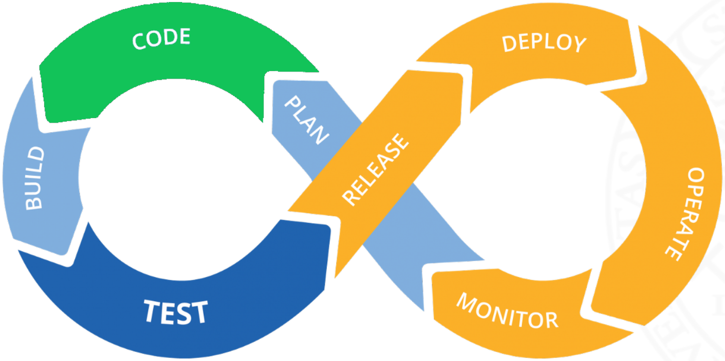


Grand View Research 2018, San Francisco, CA, United States

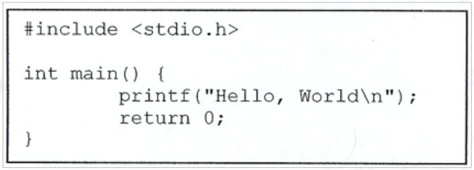
Version Control Systems

How to keep control of the software evolution

Code Development in the DevOps Toolchain



At the beginning...



```
#include <stdio.h>

int main() {
        printf("Hello, World\n");
        return 0;
}
```

1 file, 4 lines

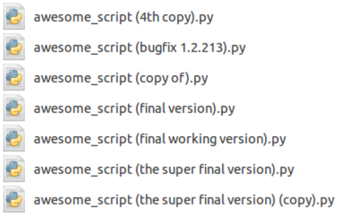
Any code change is easy to manage, right?

...then things start to get dirty

Alone developer working on millions of code lines

Update the source code: version history

How to properly manage code updates?


awesome_script (4th copy).py
awesome_script (bugfix 1.2.213).py
awesome_script (copy of).py
awesome_script (final version).py
awesome_script (final working version).py
awesome_script (the super final version).py
awesome_script (the super final version) (copy).py

Not the best way to track the previous versions

Version Control Systems!

Version Control Systems (VCSs)

Version Control System

A system that records the **changes** to a file or to a set of files over time so that a *specific version* can be recalled later

Office productivity

- Google docs/sheets
- CMS (WordPress, Joomla)
- Wikipedia's page history

DevOps examples

- Revision Control System (RCS, 1982)
- Concurrent Version System (CVS, 1986)
- Subversion (SVN, 2000)
- Mercurial (2005)
- Git (2005)

What we are going to learn?

How to use a VCS for:

1. local and remote **backup** of software code (or normal documents!) and **restore**
2. managing **two or more versions** of software code (es. development and production)
3. managing a **shared project**

Git: the stupid content tracker

By far, the **most widely used** modern version control system in the world today

From the README, Git means:

- a random pronounceable three-letter combination, not used by any UNIX command
- stupid, despicable, simple
- *global information tracker*, but only when you're in good mood (*angels sing*)
- *goddamn idiotic truckload of sh*t*, when it breaks

Short history of Git

- The Linux kernel is an open source software project
- (1991/2002) changes to the Linux kernel were passed around as patches and archived files
- In 2002, the Linux kernel project began using a proprietary VCS called BitKeeper
- In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tools free-of-charge status was revoked
- Linux development community (and in particular Linus Torvalds, the creator of Linux) started to work on their own open-source VCS: **Git**

Git: the stupid content tracker

Key Concepts of Version Control Systems and Git

Key concepts: The Repository

A repository is like a **database** for your project

Your project's repository contains all of your project's files and stores the *history* of the project

In Git, it's a generic directory containing the `.git` folder

Key concepts: Commits and Revisions

Changes to files and folders are stored by a **commit** operation

Committing $\approx$ Saving

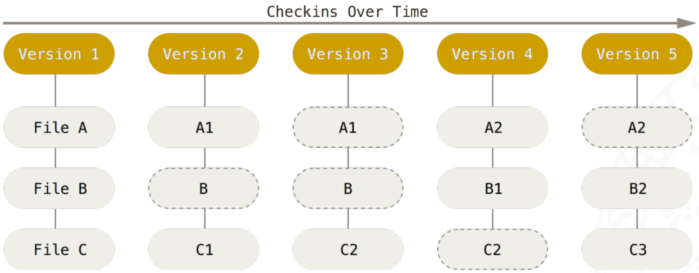
Commits are *described logical sets* of actions that can be **uniquely identified** in the VCS repository

These actions can be:

- Replace a word on line X with another word
- Delete line X
- ...

Each commit corresponds to a **revision** of the codebase
The latest revision is often called `HEAD`

Storage of revisions in Git



Storage of a new revision

- *on commit*, a snapshot of all files *content* is stored

✓ A **reference** to the previous version is stored if no changes

✓ If two files have the exact same content, **data is stored once**

Git: the stupid content tracker

Getting Started with Git

Installing the Git package

Install on Windows or MacOS

Download and run the installation package from
`https://git-scm.com/downloads`

Install on Linux

Debian/Ubuntu: `sudo apt-get install git`
Fedora/CentOS: `sudo yum install git`

The terminal

Git is a **command line program**

To work with it a **terminal** window should be opened

Test Git installation

1. Create a new folder for the project
2. Open a terminal (Win: right click → 'Git **Bash** Here')
3. Write `git --version` (latest is `2.23`)

Useful terminal commands

- `ls` (linux, mac) `dir` (windows): display the content of the current directory
- `mkdir`: create a new folder inside the current directory
- `cd` *`foldername`*: switch to *foldername*

Configuring the text editor for Git

The default text editor is usually `vim`

Let's use something bett... ehm *easier*!

Windows

1. (if available) `git config --global core.editor nano`
2. (`git config --global core.editor notepad`)

Linux or MacOS

1. `git config --global core.editor nano`

User identification

Let's identify ourselves!

- Author identity is *always* attached to each `commit`
- This is **foundamental**, especially while working in a team

Configuring name and email

```
1 git config --global user.name "Paolo Rossi"
2 git config --global user.email "rossi@mail.com"
```

Initializing a new repository: `git init`

- To create a new repo use the command: `git init`
- *one-time-only* command that creates a new *.git* subdirectory in your current working directory

Try yourself!

In the project directory run:

1. `git init`

Output of the terminal will be *Initialized empty Git repository in ...*

Inspect a repository: `git status`

The `git status` command displays the state of the repository:

- ✓ New files, changed files
- ✓ Info about *current* commit

- ✗ It will *not* output the revision history (use `git log` instead)

Try yourself!

1. `git status`

Output of the command will be similar to:

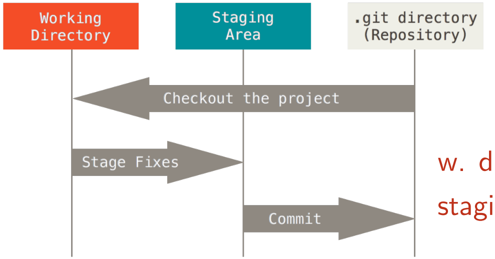
```
On branch master
nothing to commit
```

The edit-stage-commit cycle

How editing of files works in Git?

The edit-stage-commit cycle

Git editing workflow is *virtually* split in 3 areas



.git	:	the repository (objects database)
w. directory	:	the playground
staging area	:	the "buffer" area

The working directory

- The "space" where files and folders are edited
- Changes in the working directory **do not affect** the repository
  - Updates are *not stored* until instructed to do so
- Newly added files are called *untracked*
  - Meaning that are not yet part of the repository

Try yourself!

1. Create a new *text file* and place it in the project directory
2. `git status`

Output of the command will list the untracked files

The staging area (`git add`)

Changes cannot be directly stored (committed) into the repository!

- A **buffer/review** phase comes before each commit
- The files which should be part of a commit must be moved to the **staging area** using the command `git add`
    - To add specific files: `git add file1 file2`
    - (Less preferable) To add all files: `git add .`

Staging is the most important phase!

The staging area (`git add`)

Why so important?

- Changes must be **always reviewed** before commit!
- Changes must be grouped is some *logical ways*

Atomic commits

Its important to commit only **small and related** changes:

✓ to easily track down bugs (which commit(s) caused a problem)

✓ to revert with minimal impact on the rest of the project (undo only the problematic commit)

The staging area (`git add`)

Try yourself!

1. Stage the previously created text file (`git add` *`file`*)
2. Create two new *text files* and place it in the project directory
3. `git add .` to add them to staging
4. Use git status to check the staged files

Output of the command will list the *changes to be committed*

Going back from staging area (`git reset`)

To revert one or more changes staged for commit call:

```
git reset file1 file2 ...
```

Try yourself!

1. Reset one of the currently staged files (`git reset` *file*)
2. Call `git status` to see if the correct files appear in the list of staged for commit and in the list of untracked files

`git reset` can be used for more advanced restoring tasks!

Commit changes (`git commit`)

To permanently store the staged changes call `git commit`

The commit interactive dialog

- The `git commit` command will open an interactive dialog in the default text editor
- Describe each commit **extensively**
  - ✗ New changes
  - ✓ Modified main.py to fix a bug with input chars

A shortcut is provided to bypass the interactive dialog:

```
git commit -m "commit message"
```

Explore the revision history (`git log`)

To get the history of commits *for the entire repository* call `git log`

Useful extensions

- `git log -2`: displays the last 2 commits (-3 the last 3 commits, etc.)
- `git log -p`: displays the changes introduced by each commit
- `git log --stat`: displays (in a short form) the changes introduced by each commit
- `git log --graph`: displays the branches and merges history of commits

Info about the *last commit only* can be displayed using `git show`

Commit changes and explore history

Try yourself!

1. Call `git commit` to commit the staged files
   Remember to write a commit message which **makes sense**
2. Look for the infos about the last commit with `git show`
3. Look for the short repository history `git log`
4. Look for the detailed repository history `git log -p`
5. Play with `git log --stat` and `git log --graph`

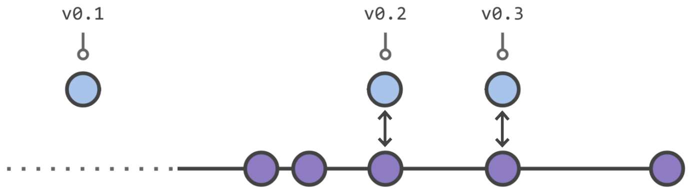
The edit-stage-commit cycle (recap)

Basic editing workflow in Git (local)

1. (first time only) Initialize the repository (`git init`)
2. Edit files in the *working directory*
3. Add files, or parts!, to staging area (`git add` *`file`*)
4. Commit changes to repository (`git commit`)

Tagging (`git tag`)

- Tags are named ref's to specific commits
- Are generally used to mark a point wich represents a specific version of the code (i.e. v1.0.1).
- To create a tag: `git tag` ***`tagname`***
- To get the hash of the commit attached to a tag:
  `git rev-parse` ***`tagname`***



Tagging (`git tag`)

Try yourself!

1. Create a new tag: `git tag` *`tagname`*
2. Commit new changes to some files
3. List curren tags: `git tag`
4. Access an old tag: `git checkout` *`tagname`*
5. Observe that the new changes are gone
6. Go back to the latest commit: `git checkout master`

What if something goes wrong?

Git is all about **recovering** from mistakes
There is always a fast *go-back* method

- Revert *not staged* changes (to a tracked file):
  `git checkout` *`file`*
- Untrack a file: `git rm --cached` *`file`*
- Committed changes can be amended: `git commit --amend`

Try yourself!

1. Stage new/additional changes
2. `git commit --amend`
3. Notice from `git log` that no new commits are present

Revert commits (`git revert`)

Entire commits can be reverted using:

- `git revert` *`commit hash`*
- This will generate a *revert commit*

Revert commits

- The commit message of a revert commit contains the hash and the description of the reverted commit
- Revert commits are useful to keep code history (it is *not unusual* to revert a revert commit...)

Try yourself!

1. Commit changes to the **content** of a previously committed file
2. Revert the last commit (`git revert` *`hash`*). To get the hash use `git log` (look at date/time)

Revert *to* commit (`git reset --hard`)

Another option is to revert the *entire repository* to a certain point in time, identified by a specific commit:

- `git reset --hard` *`commit hash`*

Pay attention!

This command will **completely reset** the repository by deleting all current changes!

Try yourself!

1. Manually delete all the content of the repository (excluding the `.git` folder)
2. Revert to a previous commit (`git reset --hard` *`hash`*). To get the hash use `git log` (look at date/time)

Recap: Git for local backup and restore

Try yourself!

1. Create a new folder and initialize a repository
2. Create a new code or text file
3. Modify it and commit each change when appropriate
4. Tag one of the versions
5. Restore the repository to a specific commit or tag

Going remote

Work with Remote Repositories

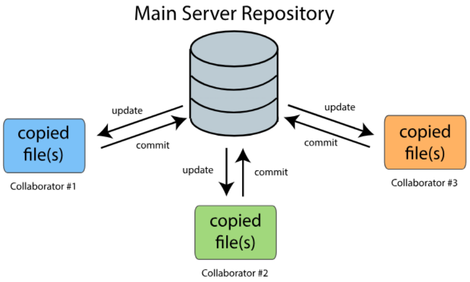
C-VCSs vs D-VCSs

Centralized Version Control Systems vs
Distributed Version Control Systems

Centralized Version Control Systems

Let's move the code to remote...

- The database storing the changes, the *repository*, is on a **single** remote server
- Each operation **requires** a connection to remote



Examples: CVS, SVN

CVCSs: downsides

A major downside:

✗ code history is stored **exclusively** on the server

What if...

...a developer ragequits and destroys our servers and backup disks?

The entire code history is *lost* except for the latest local revision!

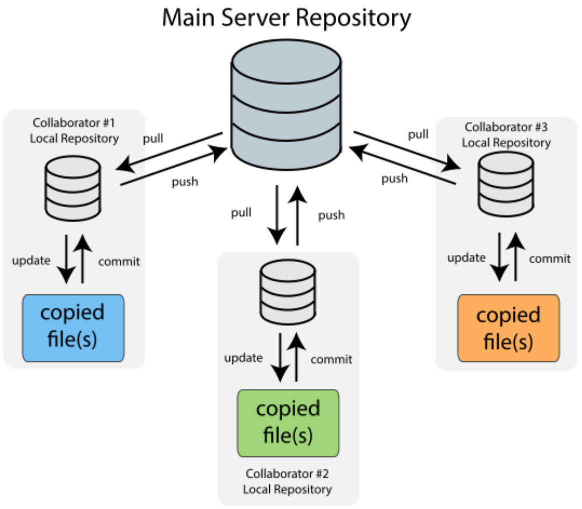
Distributed Version Control Systems

Let's be safe!

- The entire remote repository is **copied** locally
- The local repository is a *full backup* of the remote
- All operations can be fully performed locally so...
  - ✓ ...development is not affected by server status
  - ✓ ...if the server is destroyed, development can continue

Examples: Mercurial, Git

Distributed Version Control Systems



Git remote repositories

Git is a *Distributed Version Control System*:
the local repository can be **copied** to a remote server,
along with history and tags

This means that git can be used for *backup* (to remote)
and *recovery* (from remote)

The copy of a repository stored on a server is usually called
remote repository

Online Platforms

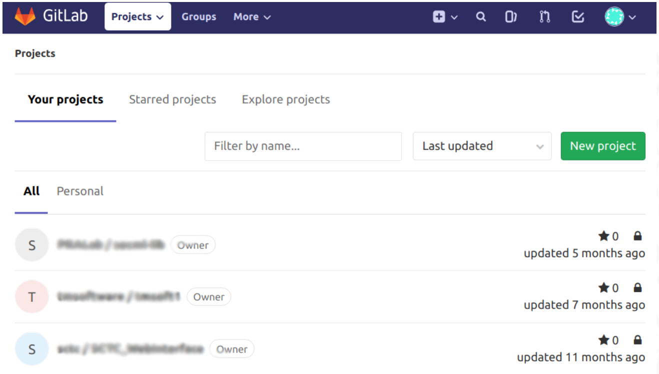
Different online platforms provide the Git tools on the cloud, for free!

GitHub GitLab

Let's create a GitLab account!

Create a GitLab account

`https://gitlab.com/users/sign_in`



Projects

Your projects Starred projects Explore projects

Filter by name... Last updated New project

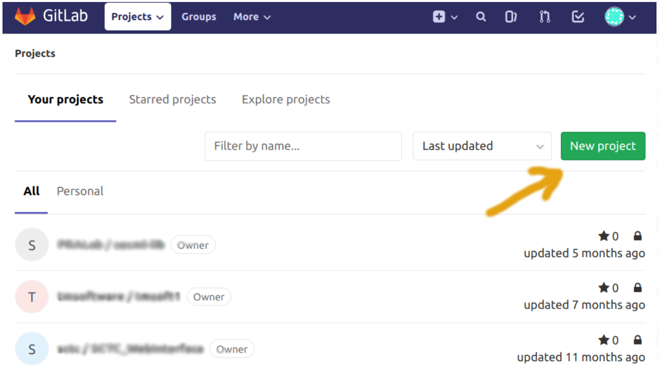
All Personal

S	[redacted]	Owner	★0 updated 5 months ago
T	[redacted]	Owner	★0 updated 7 months ago
S	[redacted]	Owner	★0 updated 11 months ago

Do not use external identity providers (Facebook, Google, etc.)

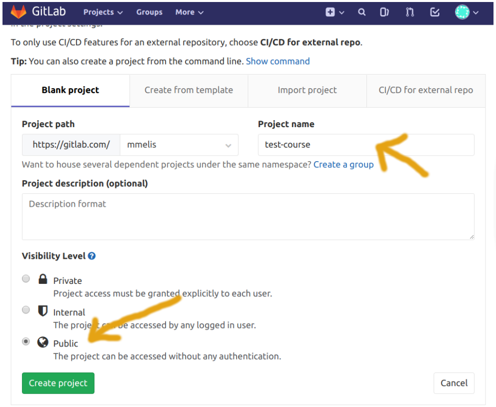
Create a GitLab Repository

Use the *new project* tool



Create a GitLab Repository

Create a new *public* repository



To only use CI/CD features for an external repository, choose **CI/CD for external repo**.

Tip: You can also create a project from the command line. Show command

Project path: https://gitlab.com/ mmelis

Project name: test-course

Want to house several dependent projects under the same namespace? Create a group

Project description (optional)

Description format

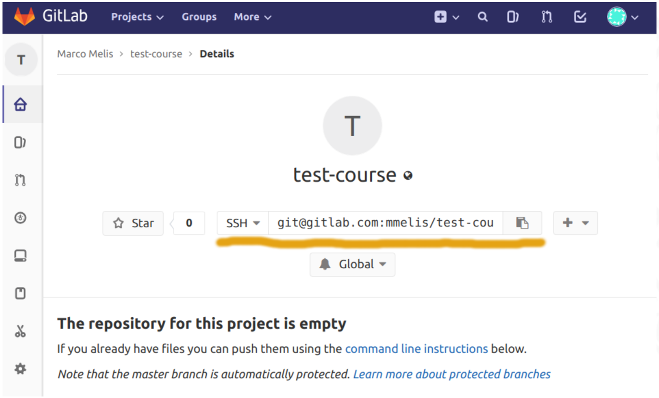
Visibility Level

- Private — Project access must be granted explicitly to each user.
- Internal — The project can be accessed by any logged in user.
- [✓ Public] — The project can be accessed without any authentication.

Do **NOT** use `"Initialize with a README"` option!

Create a GitLab Repository

The new repository has an unique address!



```
git@gitlab.com:mmelis/test-cou
```

The repository for this project is empty

If you already have files you can push them using the command line instructions below.

Note that the master branch is automatically protected. Learn more about protected branches

Working with remotes (`git remote`)

The `git remote` command lets you create, view, and delete connections to remote repositories

- Remote connections are more like *bookmarks* rather than direct links to repositories
- They serve as *convenient names* that can be used to reference the real long repository URL
- Multiple remotes can be added to a local repository

Add a remote repository

To add a new connection to a remote repository use:

- `git remote add` *`repoalias repourl`*

The default alias for the main remote is `origin`

Publish changes to remote (`git push`)

After committing the local changes, these can be *published* to the remote repository

Push changes

The operation of uploading changes to a remote repository is called **pushing**:

- `git push`

Only the committed changes will be pushed!

✗ staged changes and untracked files *are not uploaded*

✓ tags are uploaded separately by using: `git push --tags`

Publish changes to a remote (git push)

Try yourself!

1. Add your GitLab remote repository (https address) using:
   `git remote add origin` *`repourl`*
2. (the first time) Push the local repository to the remote using:
   `git push --set-upstream origin master`
   You will be asked for gitlab.com username/password
3. Add a new commit
4. Push the newest changes to remote using: `git push`
5. Push tags to remote using: `git push --tags`

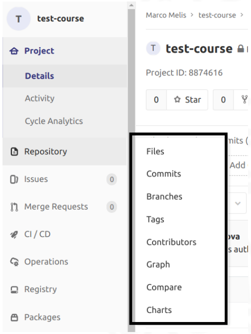
Explore remote repositories

- Now navigate to gitlab.com and explore the `Repository` menu on the left
- All repository data (files, history of commits, tags) are available online

Restore from remote

To restore the local repo from the remote:

1. `git fetch origin`
2. `git reset --hard origin/master`

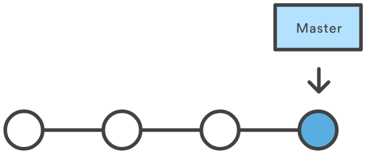


Git for managing multiple versions

Commit changes without nightmares: **branches**

What is a branch?

A branch is a sequence of commits
It is the set of changes that led to a specific state of the repo



- Each repository has at least one branch, the **main trunk**, which is often called `master`
- By default, the `master` branch is the *active branch* upon repository creation

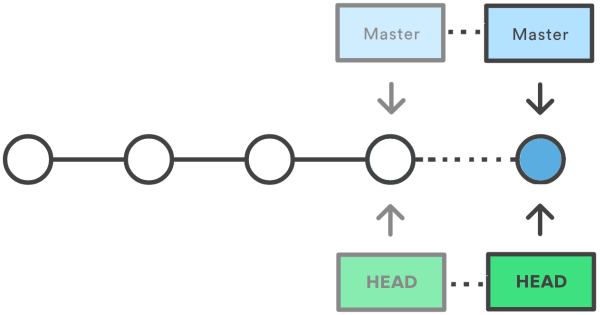
Try yourself!

1. Use `git status` to get the name of the active branch

Output of the command will be: `On branch master`

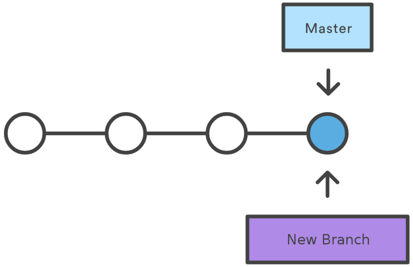
Branch and commits

New commits are "attached" to the `HEAD`, which is by default the latest commit of the *active branch*



Branching

Every VCS defines the **branching** operation,
i.e. the creation of *new branches*

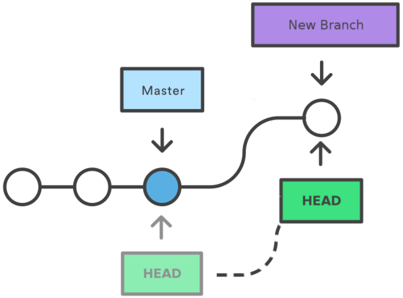


- Think of each new branch as a *copy* of the repository
- The branch from which the new branch is created is called *source*
- Initially, the new branch and the source branch will *share* the same sequence of commits

Branching

By default, the new branch does not **automatically** become *active*

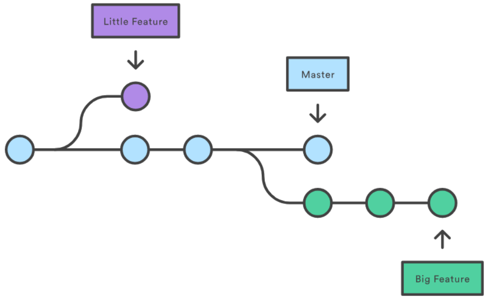
- you still see the files as per the *source branch*'s sequence
- `HEAD` is still the latest commit of the *source branch*



- But after setting the new branch as the *active branch*, the repository state will reflect the sequence of commits of the *new branch*
- `HEAD` will now be the latest commit of the *new branch*
- The source branch **will not be affected** by commits until becomes active again

Branching

Each branch is an **isolated** working environment

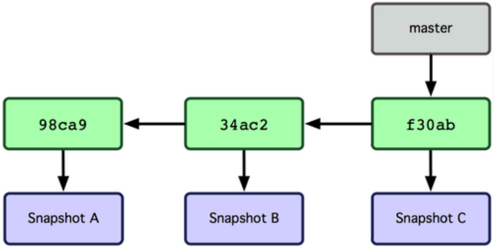


Where you can do:

- New feature development
- Code testing
- Crazy experiments!
- ...

Branching in Git

In Git, each branch is a **pointer** to a specific commit



Branch creation (`git branch`, `git checkout`)

To create a new branch *from the current branch* call:

- `git branch` *`branchname`*

This only creates the new branch

To *activate* it use: `git checkout` *`branchname`*

Try yourself!

1. Start a new branch: `git branch` *`branchname`*
2. Switch to the new branch: `git checkout` *`branchname`*
3. Use `git status` to check if the new branch is active

Play in the repository using the new branch (stage new changes, add new files, commits, check history)

Remote branches

Branches are in fact always published to remote repositories along with commits (and tags) each time we use `git push`

Each local branch, however, *must be manually mapped* to a **remote branch** before (or during) push

Remote branches are *prefixed* automatically using the name of the remote they belong to (`origin` by default), avoiding mixing up with local branches:

- `master` → `origin/master`

Remote branches

The first time a branch is uploaded to the remote, the `--set-upstream` flag should be used:

- `git push --set-upstream origin master`

The first argument of the `git push --set-upstream` command is the alias of the remote repository, while the second is the **desired name for the remote branch**

Remember that all git commands are relative by default to the *active branch*, so **only the active branch will be pushed**

Merging

After work on a branch is finished, we want to integrate the changes into the main trunk (`master` branch)

This procedure is called merging

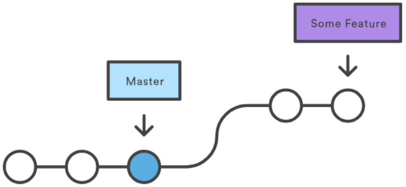
Merge the *target branch* into the *active branch* using:

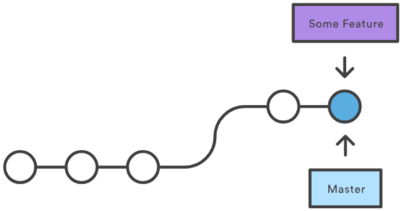
- `git merge` *`target-branch-name`*

Fast-Forward Merging

If the active branch *has no new commits* with respect to the branch to merge, Git does a **Fast-Forward Merge**

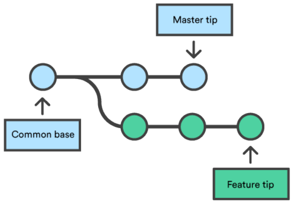
- This works by *moving the pointer* of the active branch to the latest commit of the branch to merge

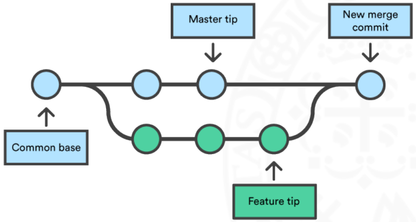




No Fast-Forward Merging

If the active branch *has* new commits with respect to the branch to merge, Git creates a **merge commit**





Merge commits are special

- They always have two parent commits
- They contain the "history" of the merge process

Merging

Try yourself!

1. Switch to the `master` branch (`git checkout`)
2. Merge the previosuly created branch into `master`:
   `git merge` *`branchname`*
3. Check the status of the master branch and the repository log
4. Delete the secondary branch: `git branch -d` *`branchname`*

Give a role to branches

While branches are normally used for development of features in an isolated environment...

...we can give them **specific roles**!

A specific branch for:

1. the unstable code in development (`develop`, `master`)
2. stable code (`stable`, `master`)
3. released code (`production`, `release`, `master`)

Multiple versions can coexists in separate specific branches

✗ Do not left undeleted branches without a specific purpose!

Manage a development and a production version

Try yourself!

1. Switch to the `master` branch and synchronize with remote
2. Create a new `develop` branch and synchronize with remote
3. Make changes to the code and commit to `develop`
4. Once satisfied with the code, merge `develop` branch with `master` and synchronize with remote.
   Only stable/final code should be on `master`.

Project sharing

Develop a shared project with Git

Project sharing

Git works great for *single-developer* projects!

- Allows data backup and restore
- Helps enforcing a clear organization of the project
- Keeps history of the project for easier bug fixing and documentation purposes

However, Git is all about **collaboration**!

Project sharing

Shared project: multiple developers working **at the same time on the same code** using the same remote repository

Remote repositories allow each developer to access the same code history, same tag list, etc.

But the most important step to successful partecipate in a shared project is to *make others know what are you working on*

The best way to do so is **pushing "personal" branches**

Check the status of a remote (`git fetch`)

The status of a *shared* remote repository
can change **at any time**!

We should **periodically** check for new changes from the remote:

- `git fetch`

This command will retrieve all the necessary informations from the available remotes (commits history, branch structure, etc.)

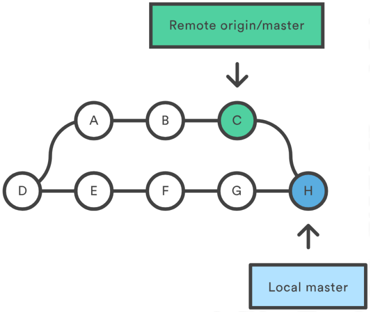
✓ `git fetch` only *downloads* the changes from the remote... **does not apply them!**

Pull changes from a remote (`git pull`)

In order to integrate the eventual changes from the remote into the local branch, another tool must be used:

- `git pull`

This command will **merge** the local branch with the remote counterpart



Extreme attention!

This is the *not safe* version of the `git fetch` command, as it **actually updates** your local files

Merge conflicts (and how to resolve them)

Sometimes we have to merge remote changes
for files we are already editing in local

When changes in the active branch *overlap* with changes pulled from remote or, more generally, with the changes in the branch to merge, a **merge conflicts** will occur!

A merge conflict will be generated if:

- the same line of a file has been edited in both branches
- a file has been deleted in a branch and edited in another

The merge process will be aborted *until all the conflicts* have been resolved

Merge conflicts (and how to resolve them)

Resolution of conflicts is often a
manual only process!

...an IDE can automatically manage few cases but not always works...

How conflicts are presented?

```
here is some content not affected by the conflict
<<<<<<< master
this is conflicted text from master
=======
this is conflicted text from otherbranch
>>>>>>>
```

Merge conflicts (and how to resolve them)

How conflicts are presented?

```
here is some content not affected by the conflict
<<<<<<< master
this is conflicted text from master
=======
this is conflicted text from otherbranch
>>>>>>>
```

- The content *before* the ======= it is from the active branch
- The content *after* ======= it is from the branch to merge

Merge conflicts (and how to resolve them)

How to fix merge conflicts

For each merge conflict (section starting with <<<<<<<):

1. Edit the conflicted file as you expect it to be after merge
2. Remove the <<<<<<<, ======= and >>>>>>> lines
3. Stage the conflicted files: `git add`
4. Generate the merge commit: `git commit`
5. Push the changes to remote: `git push`

Pull changes from remote and resolve conflicts

Try yourself!

1. Commit some changes to one local file
2. Go to gitlab.com and **edit the same lines of the same file edited locally** (use the online editor)
3. Pull the changes from remote (`git pull`)
   Merge conflicts for the edited file should be raised by Git
4. Fix the merge conflicts
5. Commit the merge changes
6. Publish changes to remote using `git push`

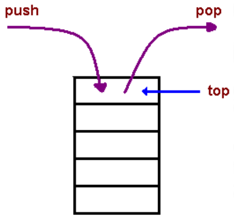
Stashing changes (`git stash`)

What if the remote changes generate a conflict with *not yet committed* local changes?

✗ Pulling is not possible until all conflicts are resolved

Solution: stash changes using `git stash`

- Temporarily store changes in the working directory
  - Untracked files will *not* be stashed!
- Stash works using a *stack* philosophy (push and pop)
  - `git stash` stores the current changes
  - `git stash pop` restores the latest stashed changes



Stashing changes (`git stash`)

Try yourself!

1. Edit a local file (**do not stage/commit**)
2. Edit the same file on gitlab.com **and commit**
3. Try to call `git pull`. The following error will be displayed: `error: Your local changes to the following files would be overwritten by merge: ...`
4. Temporarily store change with `git stash`
5. Try `git pull` again. This time should work
6. Restore local changes with `git stash pop`
7. Fix local conflicts and use `git add` *`file`* to mark resolution

Forking a repository

Sometimes we want to create a copy of a remote repository and upload it to the same remote

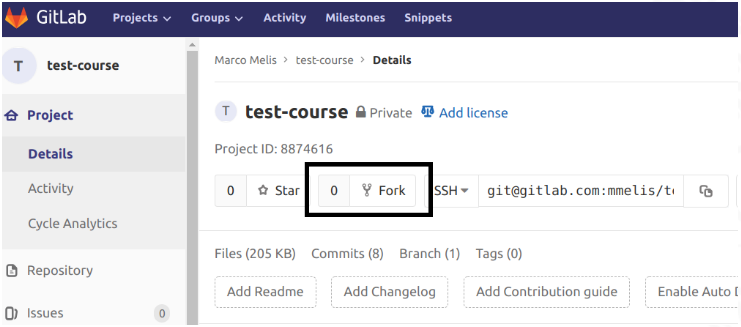
- Using standard Git tools, the remote repo can be cloned locally and then pushed using a new name

Online Git platforms however, provide the **forking** option, to directly create a copy of a repository under a new namespace

The repository created using the forking tool will not receive any update from the source repository!

Forking a repository

Forking in Gitlab:



Git Workflows

Git development workflows for teams

Git Workflows

What is a Git workflow?

- It is a recipe or *recommendation* for how to use Git to accomplish work in a *consistent and productive* manner
- Workflows *encourage* developers to use Git tools *effectively*

There is not an **universal** Git workflow!

The team should *discuss* which workflow to use

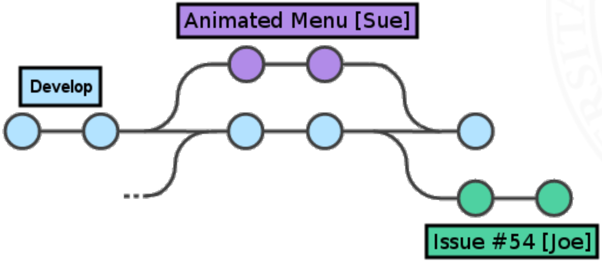
But there are some common choices..

The Feature Branch Workflow

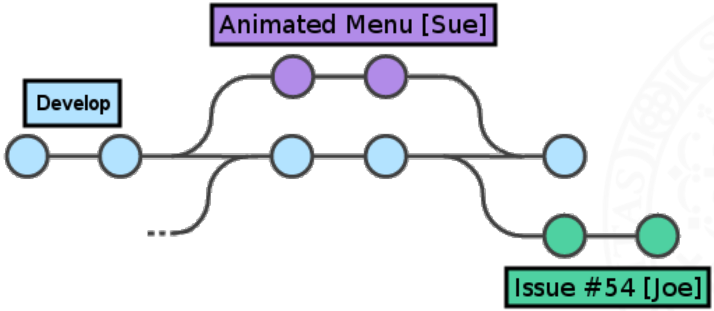
A *dedicated* **feature branch** must be **always** created when you want to add a new feature or fix a bug in your codebase...

no matter how big or how small it is

✗ No direct commits to the development branch



Feature Branches



Parallelization

This workflow allows parallel development of different features!

Our first shared project

Let's try shared development

https://github.com/unica-isde/isde-git

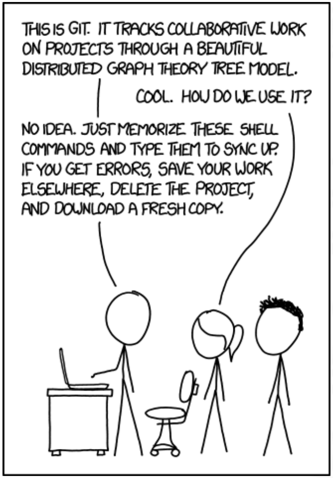
Cloning a repository

To retrieve a copy of a remote repository:

- `git clone` *`repository url`*

Git commands: a sum up

`git clone` *`url`*: mirror a repo
`git checkout` *`branch`*: activate branch
`git add` *`file`*: add files to stage
`git commit`: commit staged files
`git pull/push`: get/send changes from/to remote
`git merge` *`branch`*: merge target branch to current branch, creating a merge commit
`git stash`: store changes for later
`git reset`: remove from stage
`git revert` *`commit-id`*: revert specific commits


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.