Git crash course

Cherif Hassan Nousradine

Software and AI engineer

At Loria

University of Lorraine

September 12, 2023

Objectives

By the conclusion of this course, students will have achieved the following objectives:

- Grasp the basics of version control systems (VCS).
- Understand Git as a distributed VCS, including its core concepts.
- Work confidently with Git for practical development tasks.

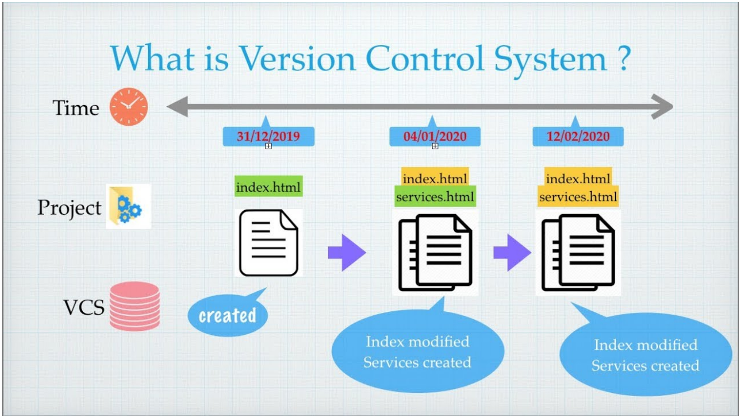
Table of content

Part 1: Version control System

- Definition
- Historical Development
- Use cases
- Fundamental concepts

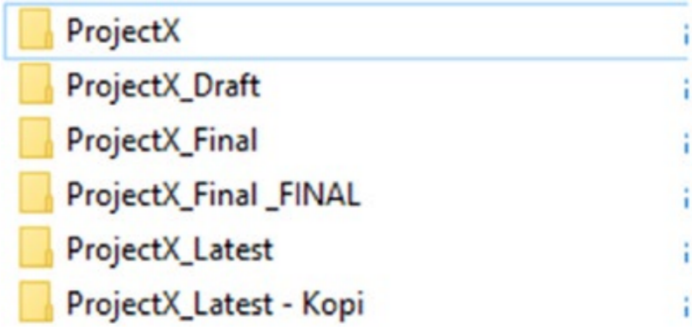
Definition

- A version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.



VCS: Intuition

- Folders in a workspace with ambiguous naming, making it nonobvious what the newest version is and how they relate



ProjectX
ProjectX_Draft
ProjectX_Final
ProjectX_Final _FINAL
ProjectX_Latest
ProjectX_Latest - Kopi

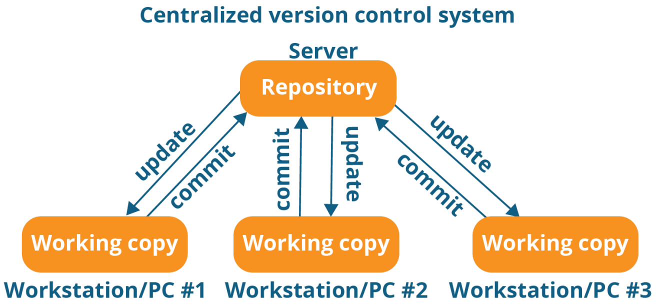
VCS:Intuition

- Without a VCS, storing project copies in multiple folders is slow and unscalable, especially for collaborative work.

John's Code
v1.0.0
v1.0.1
v1.0.2
Amy's Code
v1.0.0
v1.0.1
v1.0.2

Types of Version Control Systems

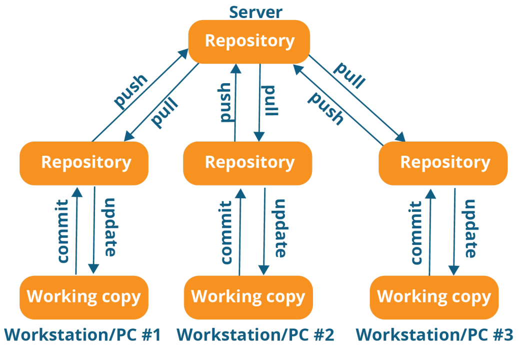
Centralized Version Control Systems: use a single global repository, requiring users to commit changes to make them visible to others, who can then update to see those changes.



Types of Version Control Systems

Distributed Version Control Systems: use multiple repositories, one per user. Changes are committed to your local repository, and to share them centrally, you must push. To update with others' changes, you need to pull those changes into your repository first.

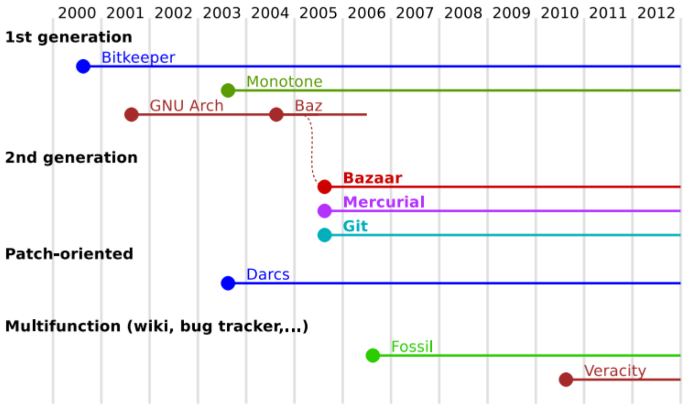
Distributed version control system



Historical Development: Centralized Tools

- First generation (single-file, local-only, lock-before-edit)
  - 1972: SCCS
  - 1982: RCS
  - 1985: PVCS
- Second generation (multiple-files, client-server, merge-before-commit)
  - 1986: CVS
  - 1992: Rational ClearCase
  - 1994: Visual SourceSafe
- Third generation (+ repository-level atomicity)
  - 1995: Perforce
  - 2000: Subversion
  - + many others

Historical Development: Decentralised tools



Use case 1: keeping a history

The life of your software or document is recorded from the beginning:

- at any moment you can revert to a previous revision (version)
- the history is browseable, you can inspect any revision
    - when was it done?
    - who was the author?
    - what was changed?
    - why?
- all deleted content remains accessible in the history

Use case 2: working with others

Version Control (VC) tools can help you:

- share a collection of files with your team members
- merge changes done by other users (developers)
- ensure that nothing is accidentally overwritten

Use case 3: branch management

In certain scenarios, you may find yourself working with various versions of the same software, represented as distinct branches. These branches serve specific purposes and evolve independently:

- The primary branch (main or master)
- A maintenance branch (dedicated to fixing bugs in older releases).
- A development branch (for implementing significant changes).
- A release branch (for stabilizing code before a new release).

Git provides essential capabilities for managing these branches effectively:

- Simultaneously manage multiple branches
- Seamlessly integrate changes from one branch into another.

Use Case 4: Collaborating with External Contributors

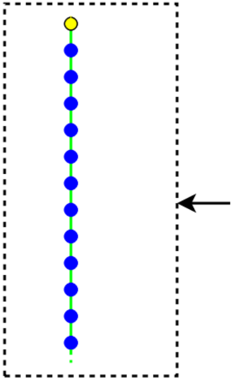
Utilizing Version Control tools facilitates collaboration with external contributors by:

- Providing them with project visibility, allowing them to stay informed about project activities.
- Streamlining the process of submitting their changes, while also aiding you in seamlessly integrating their patches.
- Enabling the option to fork the software development process and subsequently merge their contributions back into the mainline development

Summary of Benefits of VCS

- VCS works as database of all your code and make revisions instead of duplicating the files which helps you to save a lot of storage.
- It keeps all the history of all the files which gives you a full traceability and audibility of the change on what changes was made to which file, when, why and by whom.
- It provides an ability to revert back to last revision or any previous stage as per requirement
- It prevents the risk of losing funcotioning code or breaking test scripts by overwriting files as you can always take out the last working code at any point of time.

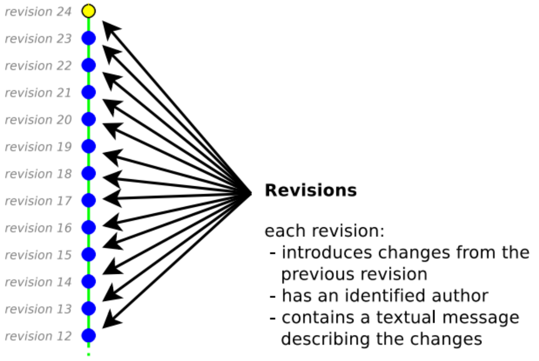
Visual Depictions: The repository



The Repository

it contains the full history of your project (all revisions from the beginning)

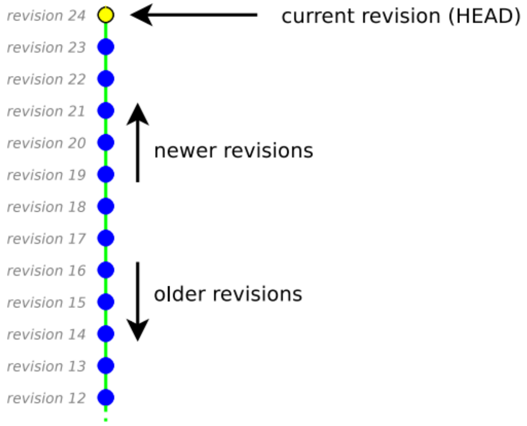
Visual Depictions: revisions



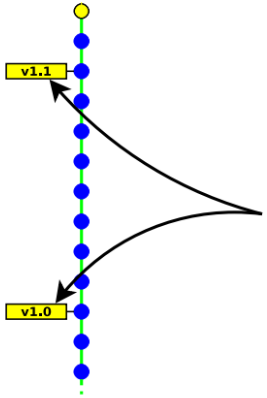
Revisions

each revision:
- introduces changes from the previous revision
- has an identified author
- contains a textual message describing the changes

Visual Depictions: Current revision



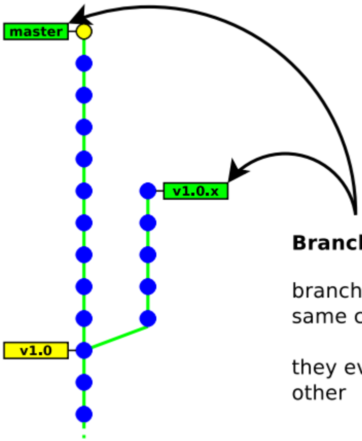
Visual Depictions: Tags



Tags

a tag identifies a particular revision
(typically each release of the software)

Visual Depictions: Branches

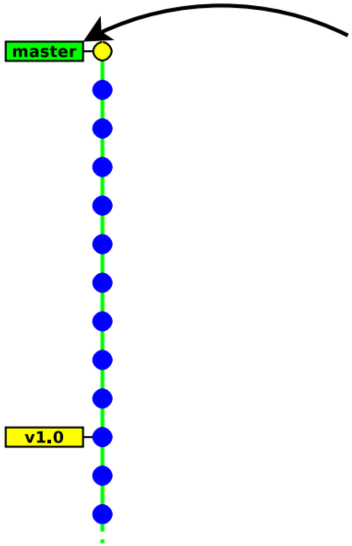


Branches

branches are different variants of the same collection of files

they evolve independently of each other

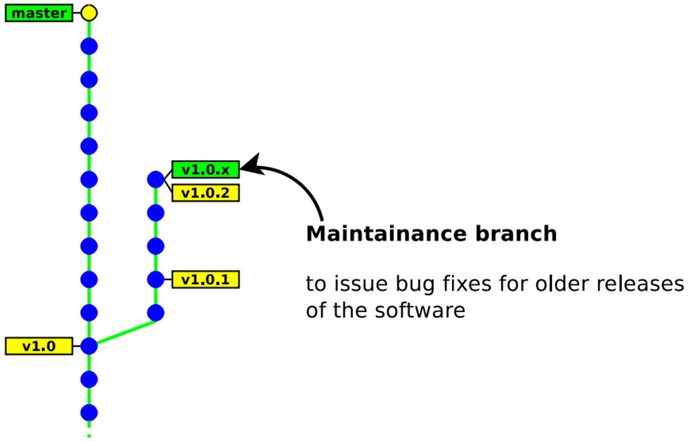
Visual Depictions: Main Branch



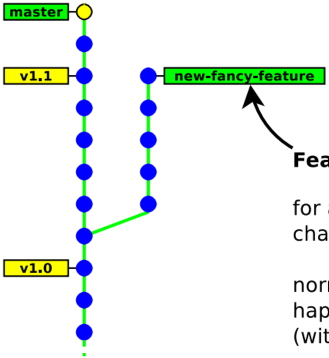
Main branch

where the everyday development happens

Visual Depictions: Maintenance Branch



Visual Depictions: Feature branch

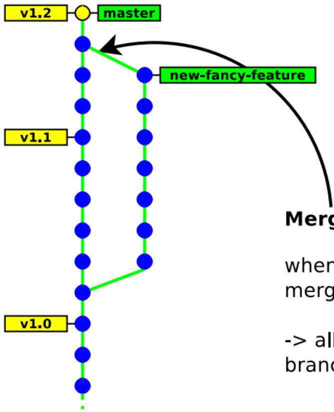


Feature branch

for a new feature requiring intrusive changes in the code

normal development continues to happen in the master branch (without disturbance)

Visual Depictions: Merging branches

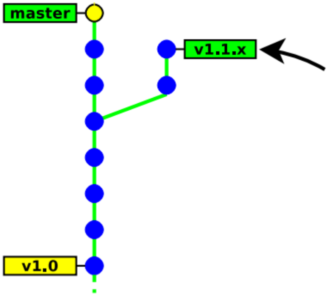


Merging

when the new feature is ready, it can merged back into the master branch

-> all changes done in the feature branch are imported

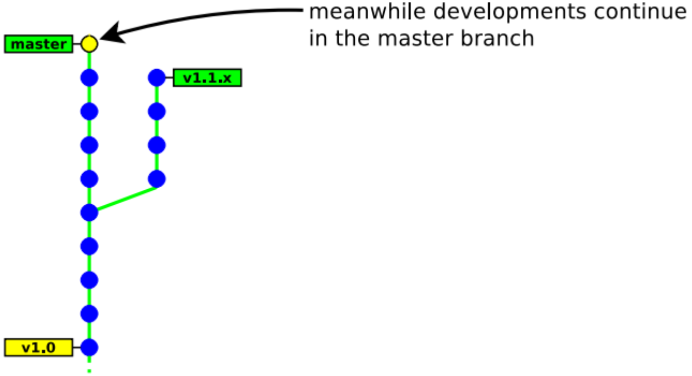
Visual Depictions: Release Branch



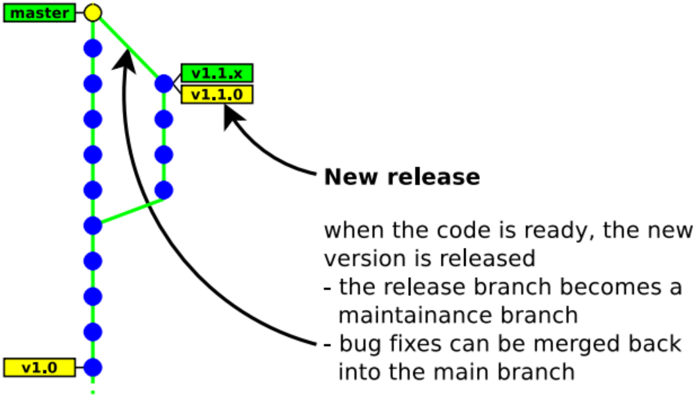
Release branch

to prepare the next release
- the code is frozen
- only bug fixes are accepted

Visual Depictions: Release and Master



Visual Depictions: New Release Branch



New release

when the code is ready, the new version is released
- the release branch becomes a maintainance branch
- bug fixes can be merged back into the main branch

Terms

Key Terminology to Familiarize Yourself with Before Embarking on the Course:

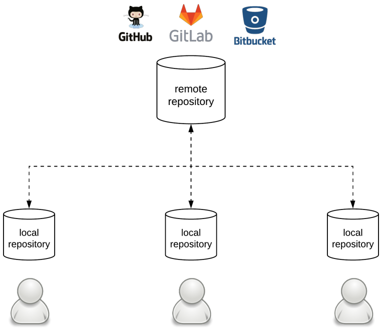
- **Directory:** Folder
- **Terminal or Command Line:** Interface for Text commands
- **cd:** Change Directory
- **Code Editor:** Word Processor for writing Code
- **Repository:** Project, or the folder/place where your project is kept
- **Github:** A website to host your repositories online

Part 2: Git: Understanding the basics

- Definition
- History
- Git's design and features

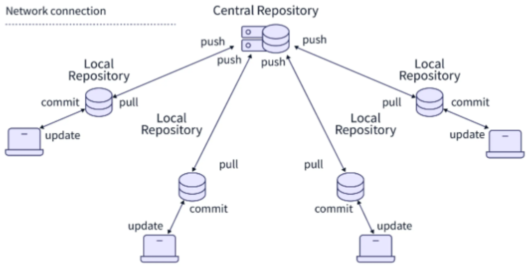
Definition:What is Git?

- Git is a free, open-source, and highly efficient version control system for projects of all sizes.
- Git is the preferred version control tool for platforms like GitHub, GitLab, and Bitbucket in the development community.



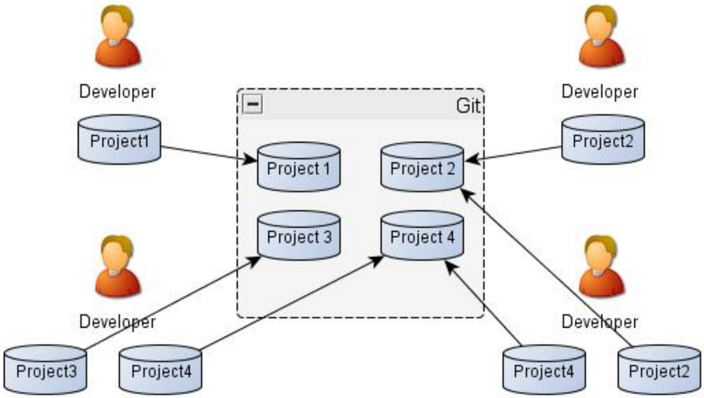
Definition:What is Git?

- It enables distributed development, allowing multiple developers to access and modify source code while keeping others in sync.
- Every Git working directory is a complete repository, offering full version tracking and independence from central servers.



Definition:What is Git?

- Git fosters collaboration by enabling multiple team members to work on the same files, helping manage changes in a coordinated manner.



Why Use Git?

Around 70% of developers worldwide use Git for development. Some of the prominent reasons for using Git are:

- Developers can work together from anywhere.
- Developers can see the full history and can compare the previous and new changes of the project.
- Developers can retreat to earlier versions of a project.

History: Git

- before 2005: Linux sources were managed with Bitkeeper(proprietary DVCS tool)
- April 2005: revocation of the free-use licence(because of some reverse engineering)
- No other tools were enough mature to meet Linux's dev constraints (distributed workflow, integrity, performance). Linus Torvald started developing Git
- June 2005: first Linux release managed with Git
- December 2005: Git 1.0 released
- Since 2016: git becomes open source

Git Design objectives

- distributed workflow (decentralised)
- easy merging (merge deemed more frequent than commit)
- integrity (protection against accidental/malicious corruptions)
- speed and scalability
- ease of use
- stability and data recovery

Git Design choices

- Distributed Version Control:
  - Git is a distributed version control system (DVCS).
  - Each user has a complete repository with history.
  - Enables offline work and robust collaboration.
- Speed and Efficiency:
  - Git is renowned for its speed and efficiency.
  - Optimized through delta compression and data structures.
  - Handles large codebases efficiently.
- Content Addressing and Integrity:
  - Git uses content-addressable storage with cryptographic hashing.
  - Data is referenced by its unique hash.
  - Ensures data integrity and tamper resistance.

Git commands

Version Control Layer	Local commands	**add** annotate apply archive bisect blame **branch** check-attr **checkout** cherry-pick **clean** **commit** **diff** filter-branch grep **help** **init** **log** **merge** **mv** notes rebase rerere **reset** revert **rm** shortlog show-branch **stash** **status** submodule **tag** whatchanged
	Sync with other repositories	**am** bundle **clone** daemon fast-export fast-import **fetch** **format-patch** http-backend http-fetch http-push imap-send mailsplit **pull** **push** quiltimport **remote** request-pull send-email shell update-server-info
	Sync with other VCS	archimport cvsexportcommit cvsimport cvsserver **svn**
	GUI	citool **difftool** **gitk** **gui** instaweb **mergetool**
VC Low-Level Layer		checkout-index check-ref-format cherry commit-tree **describe** diff-files diff-index diff-tree fetch-pack fmt-merge-msg for-each-ref fsck **gc** get-tar-commit-id ls-files **ls-remote** ls-tree mailinfo merge-base merge-file merge-index merge-one-file mergetool--lib merge-tree mktag mktree **name-rev** pack-refs parse-remotes patch-id prune read-tree receive-pack reflog replace rev-list rev-parse send-pack **show** **show-ref** sh-setup stripspace symbolic-ref update-index update-ref upload-archive **verify-tag** write-tree
Utilities		**config** var web--browse
Database Layer		cat-file count-objects hash-object index-pack pack-objects pack-redundant prune-packed relink repack show-index unpack-file unpack-objectsupload-pack verify-pack
Database (blobs, trees, commits, tags)		

Part 3: Working locally with git

- Creating a repository
- Adding and committing files
- The staging area or index

Create a new repository

git init

This command with no parameters initializes the current directory (i.e. folder) as the repository.

Example of creating new directory and initializing git repository

```
$ cd ./documents //cd to your preferred directory
$ mkdir myFirstRepo //Create a directory
$ cd myFirstRepo
$ git init
$ ls -a
```

git status

To show current branch and commits. It also allows you to see tracked vs. untracked files. It is a very useful command!

Create a new repository

git init *myRepository*

The command with parameter (fileName) creates and initializes a new repository (i.e. subdirectory/subfolder) in the current directory. You can name it whatever you please. This command creates the directory *myRepository*.

- the repository is located in myrepository/.git
- the (initially empty) working copy is located in myrepository/

Example of initializing git repository

```
$ pwd //to see your current directory
/tmp
$ git init myRepository
Initialized empty Git repository in /tmp/myRepository/.git/
$ ls -a myRepository
.  ..  .git
```

Configure the repository

Execute git config commands below:

```
$ git config --global init.defaultBranch main). //rename the
  name of the default branch in our repository to ‘main’ (a
  typical default name).
```

```
$ git branch -m main //set the name of the master branch; make
  sure you are in the subdirectory if you used git init with
  parameters. (Notice how the name "(master)" gets changed to
  "(main)" on the right the first time you do this).
```

```
$ git config --global user.name "nameHere" //you need this for
  when you commit later.
```

```
$ git config --global user.email "emailHere" //note, executing
  this command resets your email if you already have one.
```

You can verify your name and email by entering:

```
$ git config --global user.name
$ git config --global user.email
```

Activity 1: Create a git repo

1. Create a directory you want to set as your repository in a location of your choice
2. Initialize the directory as a repository and check that .git exists
3. Use config to add your name and email

Activity 2: Create a new file in the repository

1. Create a python script in your directory (working directory)

   ```
   $ nano myFirstFile.py
   ```

2. Inside the file, write code to print the text "Hello world!"

   ```
   $ print("Hello world!")
   ```

3. Save the file

   ```
   $ Ctrl + O then click enter
   ```

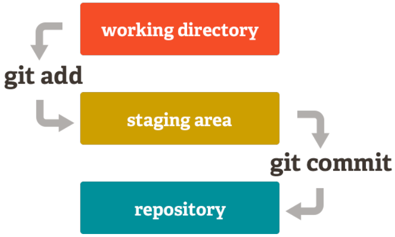
4. Exit the file

   ```
   $ Ctrl + X
   ```

The staging area or the Index

Contrary to usual version control systems, which provide two spaces, Git introduces an intermediate space: the **staging area** (also known as **index**)

- **working directory= Untracked files (red):** you can make changes in file and don't have to commit those changes; git won't care about.
- **staging area or index = Tracked files (green):** changes that are ready to be committed, meaning git will keep track of the changes in these files.
- **commit:** Officially saving the repository at a point in time (i.e. saving the current state of all the tracked files in your git record).



Add file to the index

git add files

It tracks files (copy files into the staging area).

Example of adding file into the index

```
$ cd /tmp/myRepository //cd to your working directory
$ pwd
/tmp/myRepository
$ git add myFirstFile.py
```

Commit your first file

git commit -m ”yourMessageHere”

This command commits everything that is in staging. The current state of the repository is what will be logged. Make sure to add a concise description or summary in the quotation marks of what you changed/added/deleted (i.e. the metadata); this helps other programmers who might look through your commits. If you don’t provide a description, then you will get prompted to add one.

Example of committing file into git repository

```
$ git commit -m "my first commit but not the last"
$ git status //enter git status to see what changed after
  committing.
```

Activity 3: Add, commit and do git status

Change myFirstFile.py and run it on the terminal. Then add it, commit it, and do git status and git diff.

1. Add file to staging:

```
$ git status //get used to using this command as it
  helps to verify which state the files are in
$ git add myFirstFile.py
$ git status
```

2. Commit files in staging:

```
$ git commit -m "saving original file"
$ git log //to see what our commit looks like
```

3. Change, save, and exit out of the python file:

```
$ nano myFirstFile.py
# change the text inside of the file
  print("Hello World! and Stanford!")
```

```
# save file
  Ctrl + O then click enter
# exit out of file
  Ctrl + x
# check file by running it in the terminal
  $ python myFirstFile.py
$ git diff //this will show you the difference
  between your committed file and the current staged
  file
```

4. Commit it again:

```
$ git commit -m "added more to text"
$ git log //to see what our commit looks like
$ git status
```

Show differences and histories

To see changes you make in a tracked file:

```
git diff fileName
```

Shows the differences between two revisions rev a and rev b (in a format suitable for the patch utility)

- by default rev a is the **index**
- by default rev b is the **working copy**

```
git diff [ rev a [ rev b ] ] [ -- path . . . ]
```

***Note:** git will only tell you changes that happen to tracked files (ex. modified, deleted ,...)

```
git log
```

Use git log command to see everything

Deleting files

Remove the file from the index and from the working copy

```
git rm fileName
```

Example of removing file from working directory

```
$ git rm mySecondFile.py
$ git commit -m "removed file"
```

***Note:** Don't forget to commit the index

Resetting changes

git reset cancels the changes in the index (and possibly in the working copy)

- git reset drops the changes staged into the index8, but the working copy is left intact
- git reset –hard drops all the changes in the index and in the working copy

```
git reset  [ --hard ] [ -- path . . . ]
```

Resetting changes in the working copy

This command restores a file (or directory) as it appears in the index (thus it drops all unstaged changes)

```
git checkout -- path
```

Other local commands

- `git show` show the status of the index and working copy
- `git mv` move/rename a file
- `git tag` creating/deleting tags (to identify a particular revision)

***Note:** note that git mv is strictly equivalent to: “cp src dst && git rm src && git add dst” (file renaming is not handled formally, but heuristically)

Exercise 1

1 Create a new directory and change into it.

2 Use the **init** command to create a Git repository in that directory.

3 Observe that there is now a .git directory.

4 Create a **main.py** file.

5 Look at the output of the **status** command; the **main.py** you created should appear as an untracked file.

6 Use the **add** command to add the new file to the staging area. Again, look at the output of the **status** command.

7 Now use the **commit** command to commit the contents of the staging area.

8 Create a **src** directory and add a couple of files to it.

9 Use the **add** command, but name the directory, not the individual files. Use the **status** command. See how both files have been staged. Commit them.

Exercise 1:Continue

10 Make a change to one of the files. Use the **diff** command to view the details of the change.

11 Next, add the changed file, and notice how it moves to the staging area in the status output. Also observe that the diff command you did before using add now gives no output. Why not? What do you have to do to see a diff of the things in the staging area? (Hint: review the slides if you can't remember.)

12 Now – without committing – make another change to the same file you changed in step 10. Look at the status output, and the diff output. Notice how you can have both staged and unstaged changes, even when you're talking about a single file. Observe the difference when you use the add command to stage the latest round of changes. Finally, commit them. You should now have started to get a feel for the staging area.

13 Use the log command in order to see all of the commits you made so far

14 Use the show command to look at an individual commit. How many characters of the commit identifier can you get away with typing at a minimum?

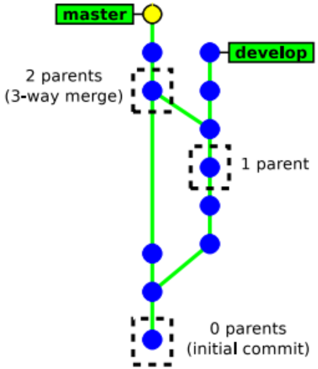
Part 4: Branching and merging in git

- How Git handles its history
- Creating new branches
- Merging and resolving conflicts

How git handles its history

Each commit object has a list of parent commits:

- 0 parent: initial commit
- 1 parent: ordinary commit
- 2 plus parents: initial commit



How git handles its history

Commits are identified with SHA-1 hash (160 bits) computed from:

- the commited files
- the meta data (commit message, author name, . . .)
- the hashes of the parent commits

A commit id (hash) identifies securely and reliably its content and all the previous revisions.

Creating a new branch

git checkout −b new_branch [starting_point]

- new_branch is the name of the new branch
- starting_point is the starting location of the branch (possibly a commit id, a tag, a branch, . . .). If not present, git will use the current location

Example of creating a new branch

```
$ git status
On branch master
nothing to commit (working directory clean)
$ git checkout -b development
Switched to a new branch 'development'
$ git status
On branch development
nothing to commit (working directory clean)
```

Switching between branches

git checkout [-m] branch_name

Example of switching to a branch

```
$ git status
On branch development
nothing to commit (working directory clean)
$ git checkout -b development
Switched to a new branch 'development'
$ git checkout master
Switched to branch 'master'
```

***Note:** : it may fail when the working copy is not clean. Add -m to request merging your local changes into the destination branch.

Merging a branch

git merge −b other_branch

This will merge the changes in other branch into the current branch.

Example of merging a branches

```
$ git status
On branch master
nothing to commit (working directory clean)
$ git merge development
Updating 8e1cf94..eb6626f
Fast-forward
file| 1 +
file2.txt | 2 ++
file3.txt | 1 +
3 files changed, 4 insertions(+)
create mode 100644 file2.txt
create mode 100644 file3.txt
```

Notes about merging

- The result of git merge is immediately committed (unless there is a conflict)
- The new commit object has two parents. → the merge history is recorded
- git merge applies only the changes since the last common ancestor in the other branch. → if the branch was already merged previously, then only the changes since the last merge will be merged

How Git merges files

If the same file was independently modified in the two branches, then Git needs to merge these two variants

- **textual files** are merged on a per-line basis:
  - lines changed in only one branch are automatically merged
  - if a line was modified in the two branches, then Git reports aconflict. Conflict zones are enclosed within «««« »»»»
- **binary files** always raise a conflict and require manual merging

Merge conflicts

In case of a conflict:

- **unmerged files** (those having conflicts) are left in the **working tree** and marked as “unmerged”
- **the other files** (free of conflicts) and the metadata (commit message, parents commits, ...) are automatically added into the **index** (the staging area)

Resolving conflicts

There are two ways to resolve conflicts:

- either edit the files manually, then run

git add **file** → to check the file into the index
or
git rm **file** → to delete the file

- or with a conflict resolution tool(xxdiff, kdiff3, emerge, ...)

```
git mergetool [file]
```

Then, once all conflicting files are checked in the index, you just need to run git commit to commit the merge.

Deleting branches

git branch -d [-m] branch_name

This command has some constraints, it cannot delete:

- the currend brach (Head)
- a branch that has not yet been merged into the current branch

Example of deleting branches

```
$ git branch -d developper1
Deleted branch developper1 (was 45149ea).
$ git branch -d developper2
error: The branch 'feature-b' is not fully merged.
If you are sure you want to delete it, run 'git branch -D
developper2
$ git checkout -b development
$ git branch -d master
error: Cannot delete the branch 'master' which you are currently
on.
```

Exercise 2

1 create a new branch named “develop”

2 make some commits in this branch

3 go back to branch “master” and make some commits

4 merge branch “develop” into “master”

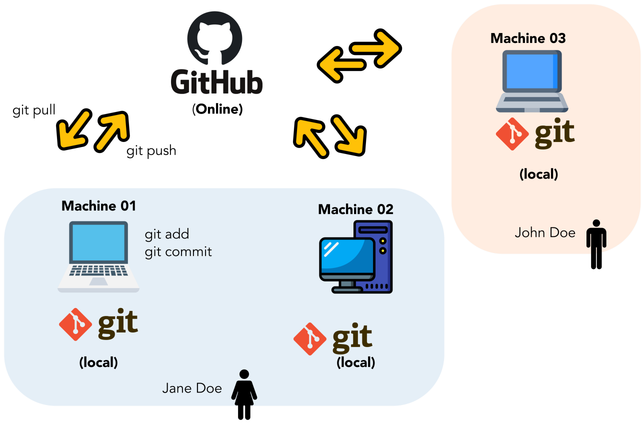
5 make a new commit in each branch so as to generate a conflict (edit the same part of a file)

6 merge branch “develop” into “master”, and fix the conflict

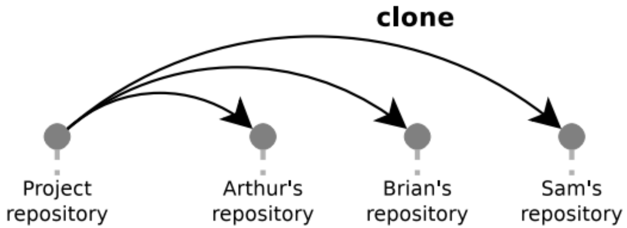
7 merge “master” into “develop”

Part 5: Interacting with a remote repository (a bit advanced)

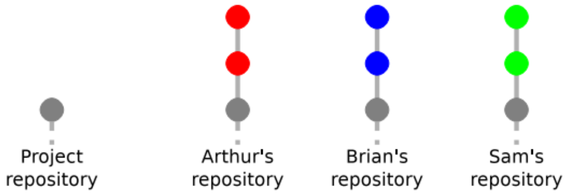
Simple workflow



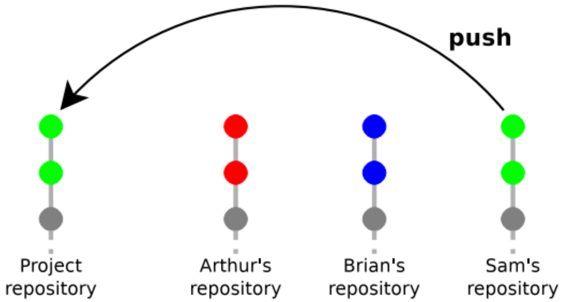
Team work


clone
Project repository
Arthur's repository
Brian's repository
Sam's repository

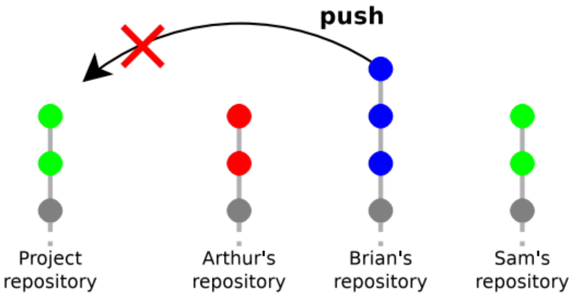
Team work


Project repository
Arthur's repository
Brian's repository
Sam's repository

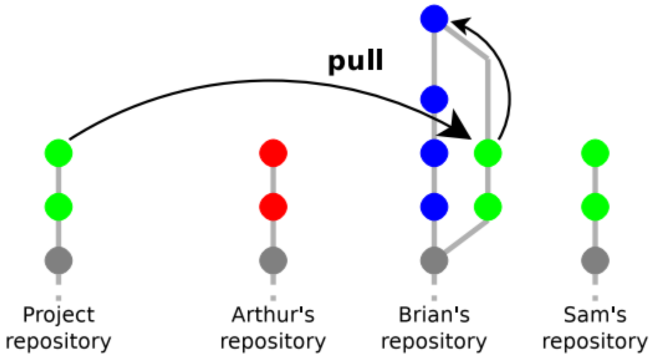
Team work


push
Project repository
Arthur's repository
Brian's repository
Sam's repository

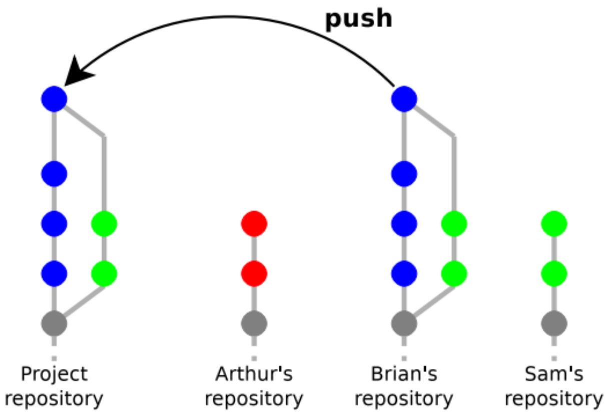
Team work


push
Project repository
Arthur's repository
Brian's repository
Sam's repository

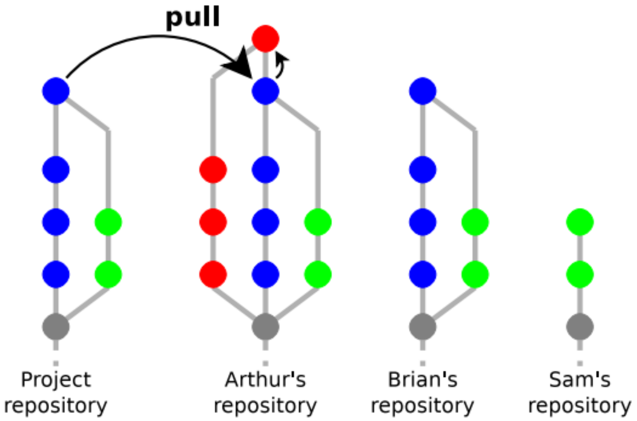
Team work


pull
Project repository
Arthur's repository
Brian's repository
Sam's repository

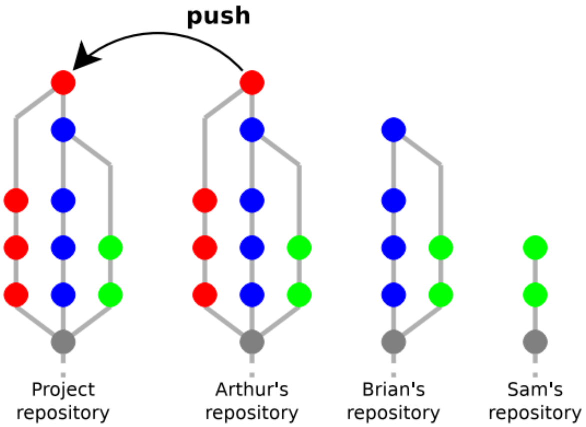
Team work


push
Project repository
Arthur's repository
Brian's repository
Sam's repository

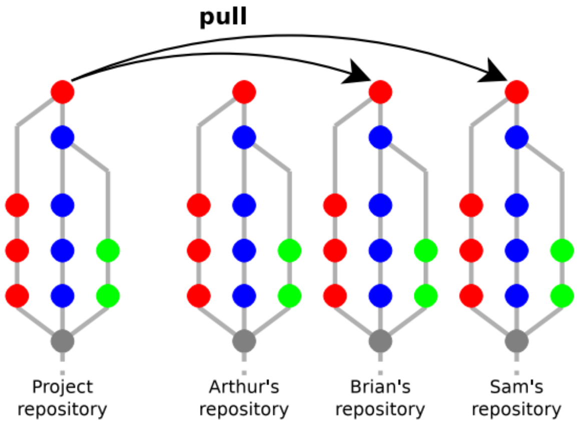
Team work


pull
Project repository
Arthur's repository
Brian's repository
Sam's repository

Team work


push
Project repository
Arthur's repository
Brian's repository
Sam's repository

Team work


pull
Project repository
Arthur's repository
Brian's repository
Sam's repository

How git handles remote repositories

- It is possible to work with multiple remote repositories
- Each remote repository is identified with a local alias. When working with a unique remote repository, it is usually named origin
- Remote repositories are mirrored within the local repository
- Remote branches are mapped in a separate namespace: remote/name/branch.
  Examples:
  - `master` refers to the local master branch
  - `remote/origin/master` refers to the master branch of the remote repository named origin

Adding a remote repository

git remote add name url

- name is a local alias identifying the remote repository
- url is the location of the remote repository

Example of adding remote repository (github)

```
$ git branch -M main //sets the target branch to main.
$ git remote add origin
https://github.com/borkounou/My_BLOG_CHERIF.git //establishing a
remote connection with github and calling it origin (a common
name used), with the address we are connecting to.
```

Adding a remote repository

git remote add name url

- name is a local alias identifying the remote repository
- url is the location of the remote repository

Example of adding remote repository (github)

```
$ git branch -M main //sets the target branch to main.
$ git remote add origin
https://github.com/borkounou/My_BLOG_CHERIF.git //establishing a
remote connection with github and calling it origin (a common
name used), with the address we are connecting to.
```

Push (upload) local changes to the remote repository

```
git push -u origin main
```

- git push examines the current branch, then:
  - if the branch is tracking an upstream branch, then the local changes (commits) are propagated to the remote branch
  - if not, then nothing is sent (new branches created locally are considered private by default)
- In case of conflict git push will fail and require to run git pull first

pull (download) changes from the remote repository

This command is used to fetch and download content from a remote repository and immediately update the local repository to match that content

```
git pull
```

Github, what is it?

A popular cloud (remote) repository where you can host your git repository and collaborate/work together with others (i.e. a social coding website).

Create a github account

Here is the link to create your github account: `https://github.com/`

Setup a new remote repository

- Setting a description is helpful and a good practice
- Make it public (it can be searched and any can see it) or private (it cannot be searched, but you can still share it amongst collaborators)
- (optional) add readME file and/or a .ignore file. Adding a readMe file is a very good practice, it helps others who are interested in what your repository is all about. These are optional because they can be added at any point. For example, if you decide later that there is a file you don't want to git to track, you can manually create a simple .ignore file.

Push an existing repository (through the terminal):

```
$ git remote add origin urlLink //establishing a remote
connection with github and calling it origin (a common name
used), with the address we are connecting to.
$ git branch -M main //sets the target branch to main.

$ git push -u origin main  //push all our contents from our local
repository to the cloud.
```

Check your pushed repository on GitHub

re-click the name of the repository you created on GitHub and your pushed data should appear

- Review the commits in your GitHub repository
- Add a file directly on GitHub
- Edit a file directly on GitHub

Updating your local repository (through terminal)

```
$ git pull // is used to fetch and download content from a remote
repository and immediately update the local repository to match
that content. (recommended.)
```

Adding collaborators

Go under your GitHub settings and you will see on the left side "collaborators" which is how you can add collaborators to your repository.

Cloning

Cloning is the act of copying a remote repository then downloading into our local git repository so we can make commits to it and push it back

> Go under your GitHub settings and you will see on the left side "collaborators" which is how you can add collaborators to your repository.

```
git clone urlLink
```

Ignoring a file in the working directory

Cloning is the act of copying a remote repository then downloading into our local git repository so we can make commits to it and push it back

- Create a `.ignore` file in your repository
- Add enter conditions (or specific files) into your .ignore file in which you do not want to sync. An example of a condition is *.txt which ignores all txt files.
- `.ignore` should appear and files you specified in .ignore should not be visible

```
git clone urlLink
```

Johan Abildskov, *Practical Git*, https://alek772.github.io/Books/Practical%20Git%20Confident%20Git%20Through%20Practice.pdf

André Sailer, *Git tutorial*, https://indico.cern.ch/event/562188/contributions/2271436/attachments/1324311/1987526/160817_GitTutorial.pdf

W3Schools, *Git tutorial*, https://www.w3schools.com/git/default.asp

Anthony Baire, *Git for begineers*, https://people.irisa.fr/Anthony.Baire/git/git-for-beginners-handout.pdf

Git Documentatin, *Git documentation*, https://git-scm.com/book/en/v2