**Git Commands**

* Git config --global user.name “ “
* Git config --global user.email “ “
* Git config --list
* Git init
* Git clone
* Git add . // Git add file.ext
* Git commit –m “message”
* Git status

**Git Workflow**

* Get the fresher copy, pulling the latest changes.
* Branch it off to a Feature Branch / Bug fix / etc.
* Work in that particular feature.
* Have my Feature Branch up to date (Pulling the master every now and then, makes it easier to merge)
* Test, Share, get Feedback, Comments (Pull request)
* Perform a final Rebase Clean up
* Record and Explicit Merge

**Local Repo:**

All the files are on my computer, they exist in my machine in the local repo. Every user has the application or the code base on their machine local (“Distributed Version Control System”)

**Remote Repo:**

Not on my machine, shared by the team. We will “Push” commit to share our work with the team. And they can “Pull” so they can work with a copy of my code.

**Viewing commits and branches:**

* Git Log 🡪 To view my commits on the branch I’m on,
* Git log --Branches=\* 🡪 View all commits in branches
* Git log branch –a 🡪 Shows all branches
* Git log <branch name> 🡪 Shows particular branch

**Undo Work**

We may want to go back in time where my code was just the way I liked it. We then need to Find the ID of that code in particular and then go back there to make the changes.

* Git log –Oneline 🡪 Shows the timeline with ID numbers
* Git checkout ID 🡪 Makes my working directory match the exact same state as it was back then. Nothing I do here will be saved in my repo. This is just to test. Then to go back I must checkout to my branch or master.
* Git checkout master

Strategy 1: With CHECKOUT

* Git Checkout ID 🡪 Checkout to the previous commit ID (Before the changes were made “Crazy commit”). This will put my repo into a DETACHED HEAD state.

**Detached Head State** 🡪 Any commits I make here they will be **Orphaned** when I change branches. These Orphaned commits will be deleted by git in regular intervals. We need to prevent this by **creating a new branch** from here.

* Git checkout –b new\_branch\_without\_crazy\_commits

This repo is now on a new history timeline where the crazy commits does not longer exists. They are considered **UNDONE.**

**IF WE NEED OUR PREVIOUS BRANCH (MAYBE IT WAS OUR MASTER) THIS STRATEGY IS NOT APPORPRIATE.**

Strategy 2: With GIT REVERT

**IDEAL FOR WORKING WITH PUBLIC SHARED REPOS.**

* Git rever head 🡪 Will create a new commit with the inverse of our last commit. The previous commit will still be in our history, and we continue on the SAME branch.

**If we are required to keep a curated and minimal git history this is NOT APPROPRIATE.**

Strategy 2: With GIT RESET

**GREAT FOR LOCAL REPOS, MIGHT BE UNCOMFORTABLE FOR SHARED REPOS.**

* Git reset –hard ID 🡪 Will erase everything up top that ID

If we have a shared repo (that has the Crazy Commit pushed to it) and we git push this one, it will show an error (not up to date).

**Undoing the last Commit.**

* Git commit –amend 🡪 Modify the last commit message

**Staging Index**

Git reset –mixed 🡪 To undo changes in the staging area

**Git Ignore**

Why? We may have a directory of images we do not need to push, or exposing sensitive information, etc. We remove Files from the INDEX.

**Git Tag**

Is a tag that we put to Commits. Tipically used to mark the Release Points (v1.0, v2.0)

* Git tag 🡪 Shows all the tags we have
* Git tag –a –m “version xxx” v1.0 🡪 Annotated tag –A
* Git tag v1.1-**LW 🡪** Light weight tag

Sharing my tags:

* Git Push origin TAGNAME 🡪 git push origin v1.0

We can aswell, use tag names to checkout:

* Git checkout TAGNAME

**Creating and Merging Branches**

**MASTER:** The main branch

**Develop:** Created off Master Branch. The place where we will do all our work.

**Feature:** Created by the Develop Branch. We work on our feature, and then we merge back to the Develop Branch. After this, they are deleted.

**Release:** May be branched off Master or Develop. Then must be merged back.

**Hotfix:** Usually branched off Master. Must merge back into Develop and Master. For fixing bugs in production.

**Merging:**

First, we need to checkout to the Branch we want to RECEIVE the changes

* Git checkout master 🡪 To receive the changes
* Git merge BRANCH 🡪 master will receive this branch (ex: Develop Branch)

Ex: Working on a Feature then merging to Develop and deleting the Feature Branch

* Git commit –m “feature commit”
* Git checkout develop
* Git merge feature 🡪 Develop Receive the changes made in Feature.
* Git branch –d feature 🡪 Deletes the Branch Feature.

**PULL requests**

Let’s my team know I made changes and pushed them into a Repo. We can decline or accept the PULL requests, after checking them out. Good for CODE REVIEW.

First, we work on our Code (Feature Branch) then we make a commit “pull request”, then push it to the Repo.

* Git commit –m “pull request”
* Git push URL feature 🡪 ULR = repo location.
* Git request-pull Revision Number URL 🡪 Revision number (given by github) , ULR = repo location
* Git checkout develop 🡪 Vamos a develop
* Git merge origin/feature 🡪 Merges the remote repo, more precisely the Feature within it.
* Git branch –d feature

**GIT WORKFLOWS**

**Basic:** Central Repo, Default development branch is the Master, Does not require any other branch. Push changes as fast as possible. Pull very often. 🡪 Can be conflicts because it’s a single branch repo. Good for 2 to 3 people

**Init** – **Clone** – John **Code** – Mary **Code** – John **Push** – Mary **Push** (Mary has a **conflict** with merging, has to solve it) – Mary successfully **Pushes.**

**Forking:** Server-side Repo. Usually in open source projects. NOT a clone, but a “download”, it is a copy.

Developers **Fork** official repo – **Clone** fork locally – **Code** – **Publish – Integrate** ( official workers accept the pulls or not)

**Feature:**  Central repo, changes should be pushed into it. Work on the Feature just as a normal repo. Good for 5 to 6 people.

Branch **Feature – Code - Pull Req – Publish** in Master **– Delete Feature Branch**

**Open-source:** Forking a clone, having an “origin” but not the real one. Code on a feature branch.

**Fork** Repo – **Clone** – **add remote** for the original repo – **Feature Branch –** **Code – Commit – Push** to github – **Pull Req -**  Once Merged: **Clean up**

**GitFlow:** There are number of branches we will be using. Not the typical master-feature branches. We have a Master, Develop and a Feature. Maybe a Release and Maintenance branches.

**Develop Branch – Features Branchces –** Mary wants to **Release - Bug** appears – **Maintenance Branch –**

It is nice to use a Client rather than the Command Line. So we can see more clearly all the information of each commit, branch, pull requests, etc.