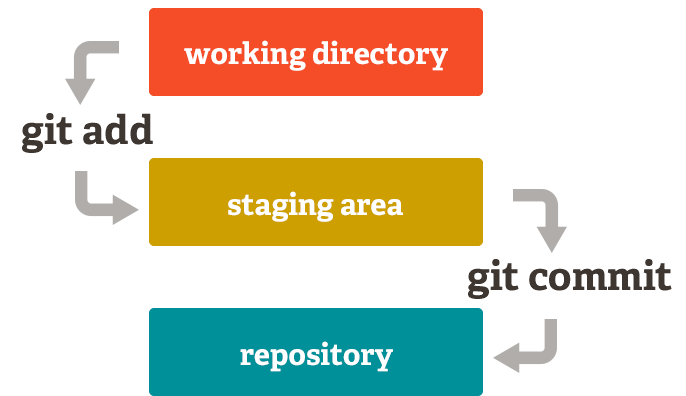
*TAREA GIT*

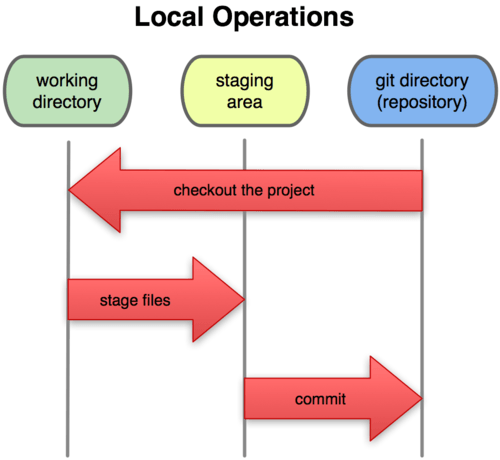
what is working tree?

The Working Tree is the area where you are currently working. It is where your files live. This area is also known as the “untracked” area of git. Any changes to files will be marked and seen in the Working Tree. Here if you make changes and do not explicitly save them to git, you will lose the changes made to your files. This loss of changes occurs because git is not aware of the files or changes in the Working Tree until you tell it to pay attention to them. If you make changes to files in your working tree git will recognize that they are modified, but until you tell git “Hey pay attention to these files,” it won’t save anything that goes on in them.



**The Staging**

The Staging Area is when git starts tracking and saving changes that occur in files. These saved changes reflect in the. git directory. That is about it when it comes to the Staging Area. You tell git that I want to track these specific files, then git says okay and moves them from you Working Tree to the Staging Area and says “Cool, I know about this file in its entirety.” However, if you make any more additional changes after adding a file to the Staging Area, git will not know about those specific changes until you tell it to see them. You **explicitly** have to tell git to notice the edits in your files.



**what is git head?**

With the first commit we make, the master of the pointer will be created automatically, pointing to this commit.

In each new commitment, the value contained in the master of aim will change, by the identifier of the following commitment.

The master branch is like any other branch. It is special because it is the default branch, but in everything else it is the same as any other branch.

When we create a new branch (git branch new branch), create a new pointer, containing the identifier of the current commitment. Creating a branch does not imply being in it. (eye, you have to jump to it, to start moving on the new branch)

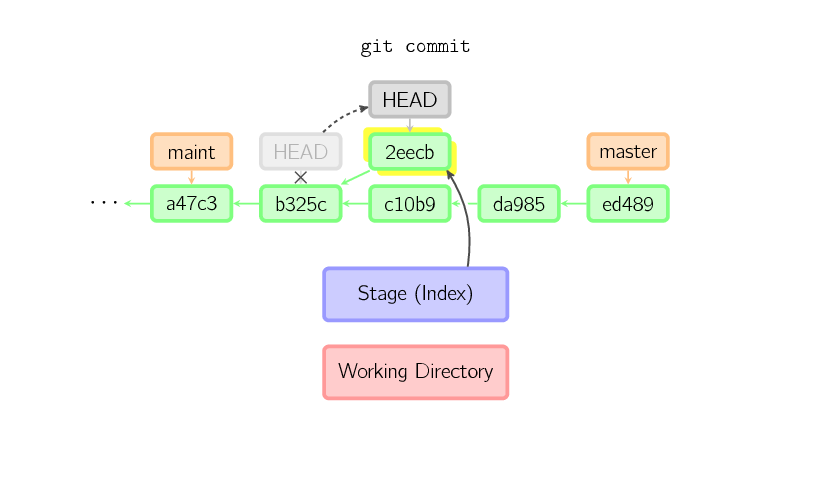
And this is where the external HEAD pointer comes into play (it is like a headlight), this pointer serves a GIT to know in which branch we are in each moment.

Then HEAD contains the identifier of the current branch, which in turn contains the identifier of the last commitment made in the branch.

HEAD POINTS TO RAMA WHERE WE ARE, AND EVERY TIME WE CHANGED FROM RAMA HEAD CHANGES ITS VALUE.

HEAD: Contains a reference to the most recent commitment of the current branch (in most cases)

When we make a jump to a specific commitment (which is not the last one where the head points), we enter a state called "separate head state", which indicates that the HEAD pointer does not contain the identifier of the last commitment of the branch where We are (it will contain the id of the commit where we have made the checkout and not the path ref: refs / heads / branch). But this is something very concrete and it is normal to have a HEAD pointing to the current branch, and to the last commit.



**what is a branch?**

To really understand how Git branches, we must first examine how it stores its data. Git does not store them incrementally (keeping only differences), but stores them as a series of snapshots (timely copies of the complete files, as they are at that time). In each confirmation of changes (commit), Git stores a control point that preserves: a pointer to the timely copy of the contents prepared (staged), some metadata with the author and the explanatory message, and one or several pointers to the confirmations (commit) who are direct parents of this (one parent in cases of normal confirmation, and multiple parents in cases of confirming a merge of two or more branches).

What is a branch modeling?

The simplest functional model A model that would meet these requirements in a really simple way is: Simple branch strategy for small teams or individuals Is practice, what is happening is: We have two long-living branches, master and develop. Each release to production comes out from master and has a tag with a named version. I wouldn’t tell a pair of programmers that they have to work on the develop branch since the beggining, first because it usually doesn’t happen, and second because the real need of a second branch starts after the first release to production (v1). This way the team can work on new features on a separate branch, while the master is intact untill the next release. The main advantage of this approach is: if you have bugs to fix, which you’ll always do, you’ll be able to provide the fix without having to stash your changes or speed up the features you are developing. So in these situations, we create a short lived branch for the fix, merge on the master and deploy to production with a new tag (v1.1), and propagate the fix to develop. When the new features are ready, they would already have the fixes, and merging with the master branch wouldn’t be a trouble (v2). It can really work on individual or small teams, considering that they may have trouble merging their changes on develop, as there’s more than one person working on the same branch. But with a simple structure of Continuous Integration concepts, this kind of issues are minimized. One disadvantage of this model is that is not really clear what is you’re working on. Let’s imagine that the develop branch has a group of commits related to a feature that you decided not to ship anymore, or not for a while. It will not be so easy to roll back the code, especially if you are working on other features at the same time. Also, it won’t be any easier to get back the discarded changes if you need them later.