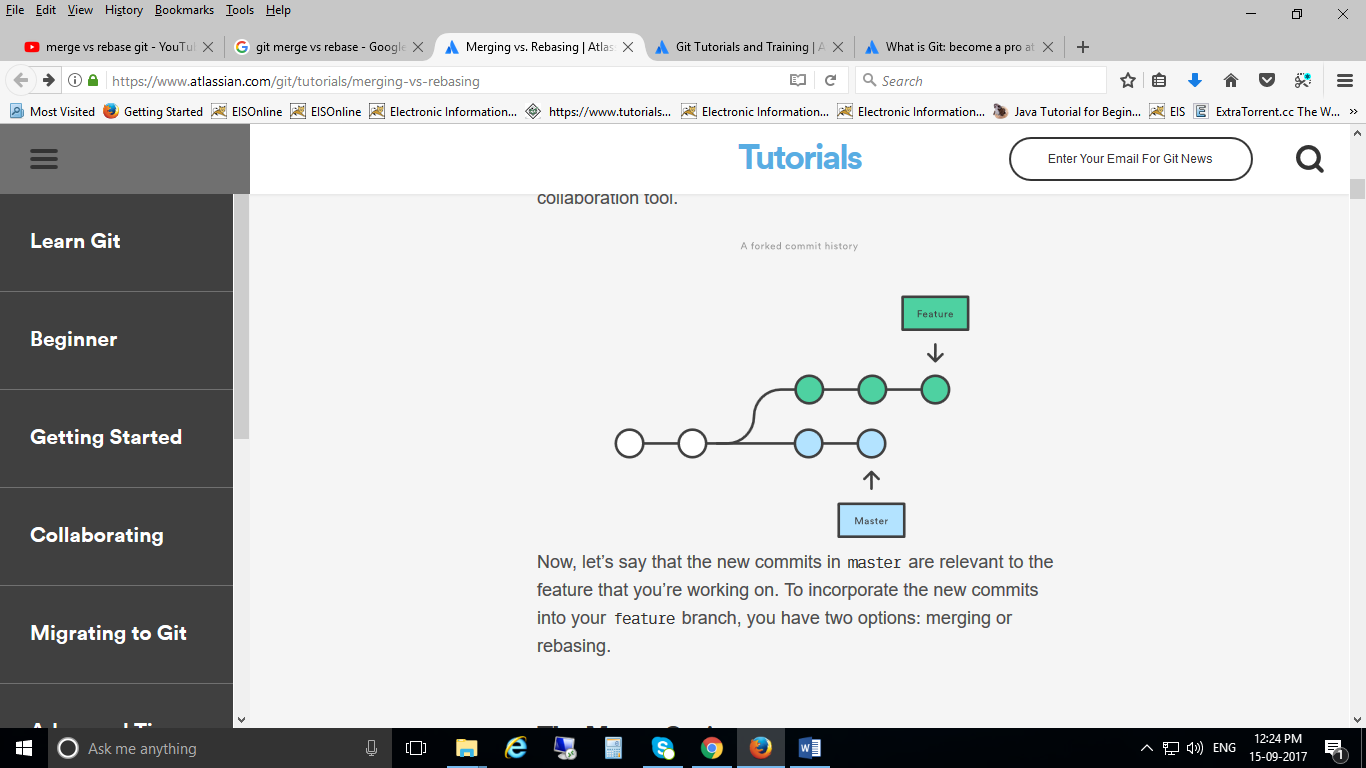
**merging-vs-rebasing (https://www.atlassian.com/git/tutorials/merging-vs-rebasing)**

when you start working on a new feature in a dedicated branch, then another team member updates the master branch with new commits



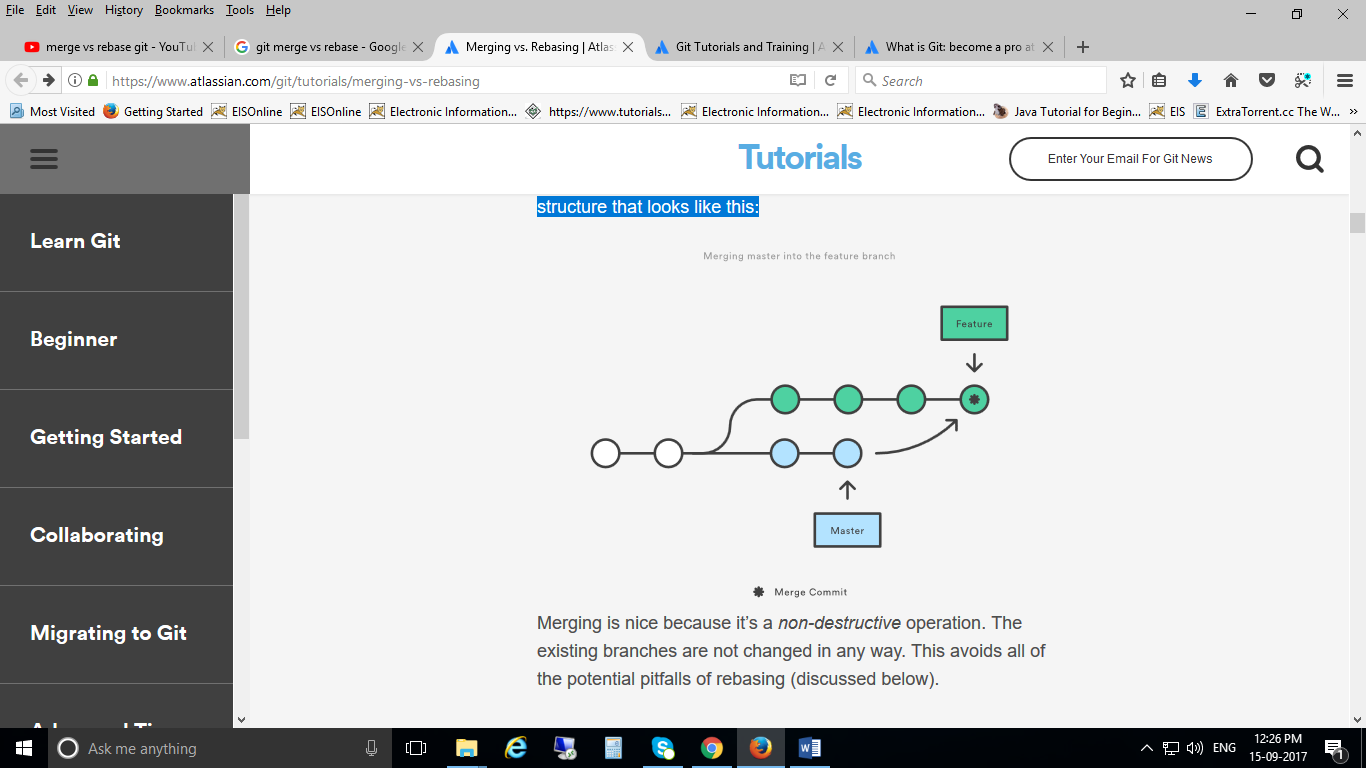
Now, let’s say that the new commits in master are relevant to the feature that you’re working on. To incorporate the new commits into your feature branch, you have two options: merging or rebasing.

The easiest option is to merge the master branch into the feature branch using something like the following

git checkout feature

git merge master

This creates a new “merge commit” in the feature branch that ties together the histories of both branches, giving you a branch structure that looks like this:



Merging is nice because it’s a non-destructive operation. The existing branches are not changed in any way. This avoids all of the potential pitfalls of rebasing (discussed below).

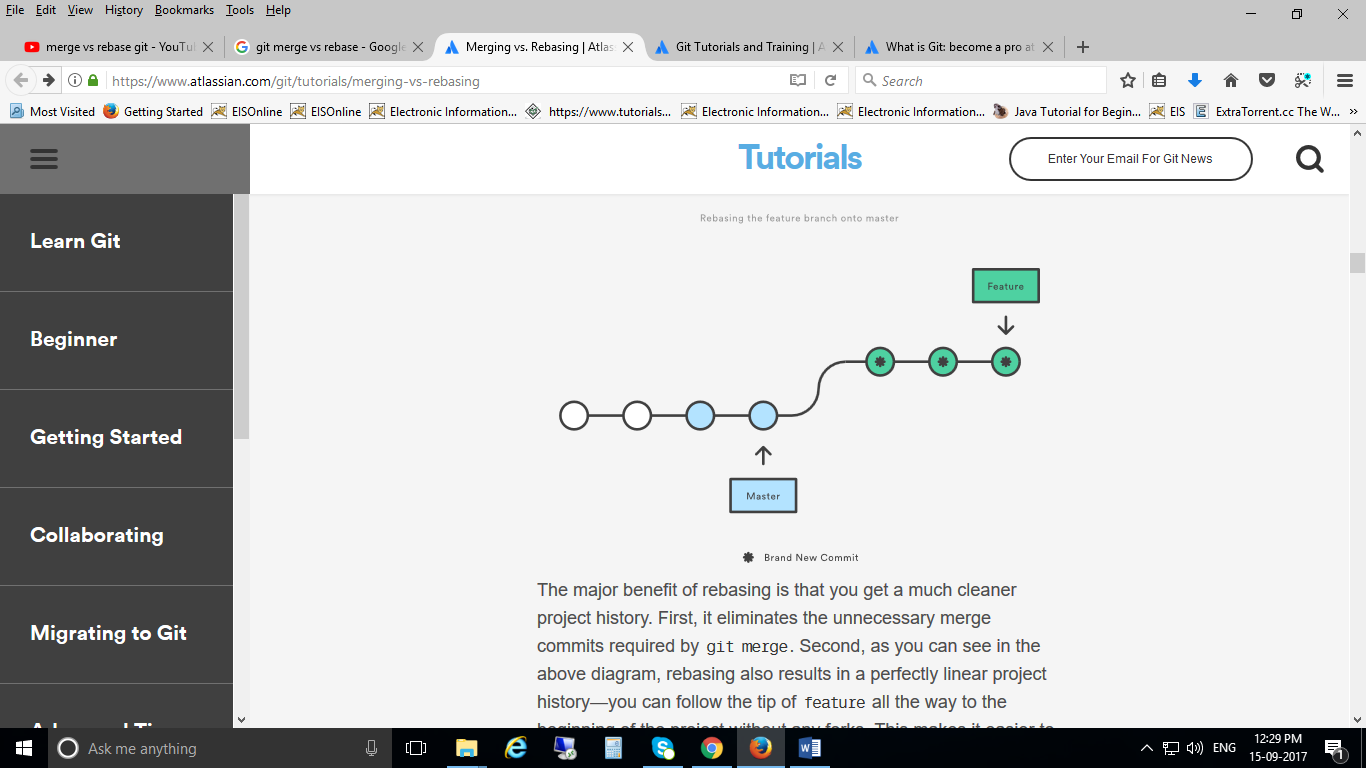
On the other hand, this also means that the feature branch will have an extraneous merge commit every time you need to incorporate upstream changes. If master is very active, this can pollute your feature branch’s history quite a bit. While it’s possible to mitigate this issue with advanced git log options, it can make it hard for other developers to understand the history of the project.

As an alternative to merging, you can rebase the feature branch onto master branch using the following commands:

git checkout feature

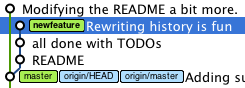
git rebase master

This moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master. But, instead of using a merge commit, rebasing re-writes the project history by creating brand new commits for each commit in the original branch.

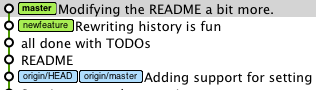
The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by git merge. Second, as you can see in the above diagram, rebasing also results in a perfectly linear project history—you can follow the tip of feature all the way to the beginning of the project without any forks. This makes it easier to navigate your project with commands like git log, git bisect, and gitk.

-----

**A git tree at default when we have not merged nor rebased**



**we get by rebasing:**



----

**Merge**

* Let's say you have created a branch for the purpose of developing a single feature. When you want to bring those changes back to master, you probably want **merge** (you don't care about maintaining all of the interim commits).

**Rebase**

* A second scenario would be if you started doing some development and then another developer made an unrelated change. You probably want to pull and then **rebase** to base your changes from the current version from the repo.