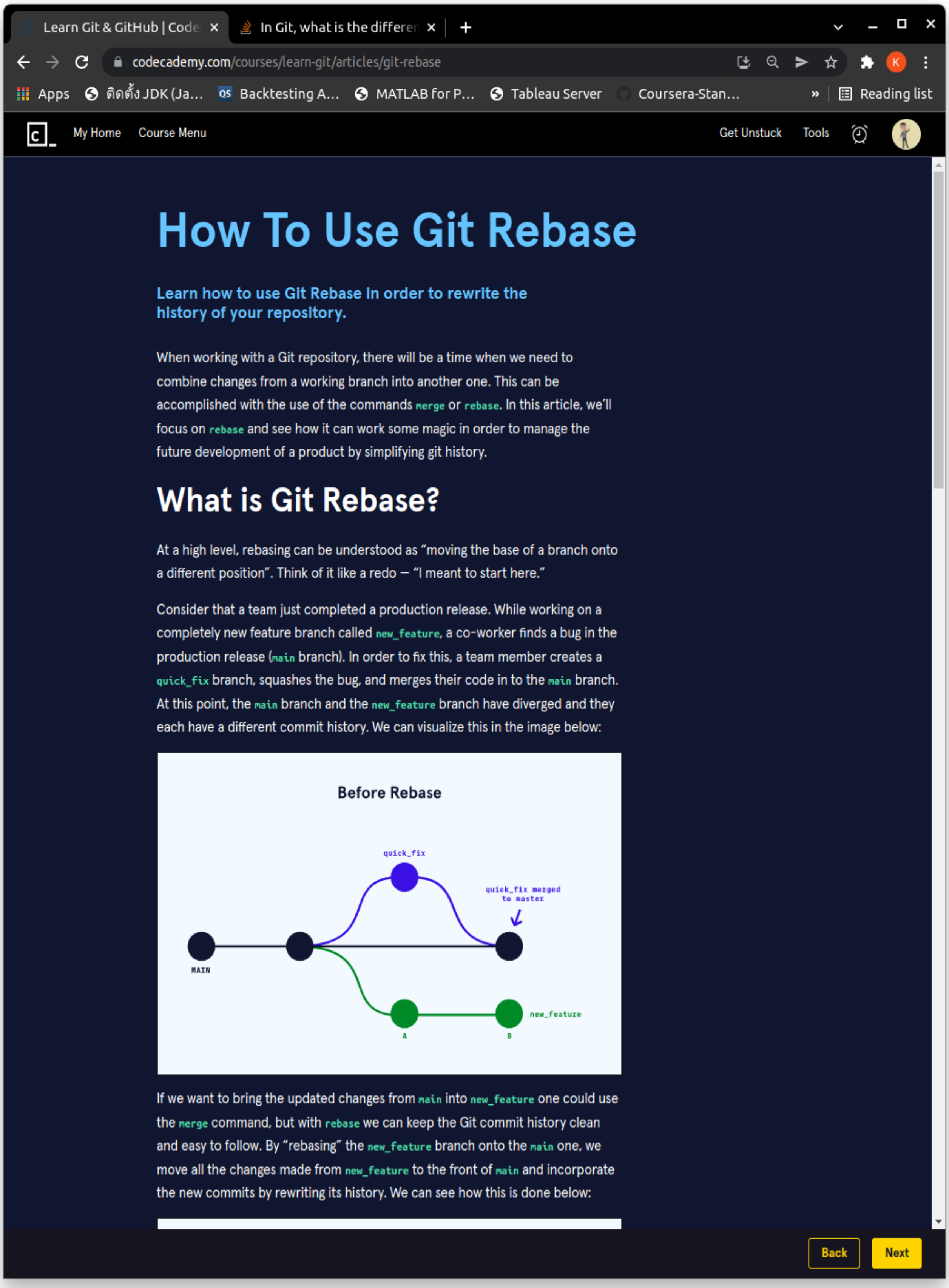


How To Use Git Rebase

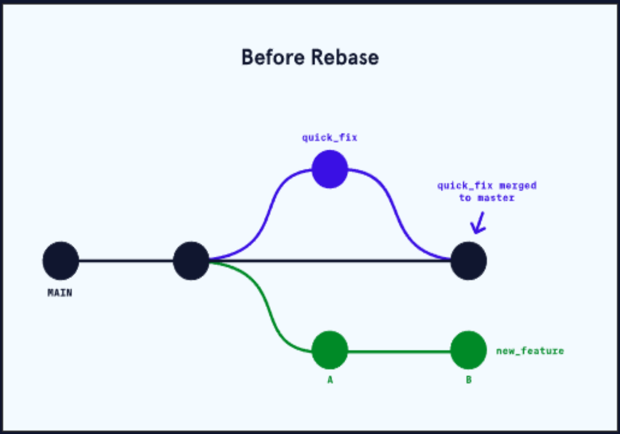
Learn how to use Git Rebase in order to rewrite the history of your repository.

When working with a Git repository, there will be a time when we need to combine changes from a working branch into another one. This can be accomplished with the use of the commands `merge` or `rebase`. In this article, we'll focus on `rebase` and see how it can work some magic in order to manage the future development of a product by simplifying git history.

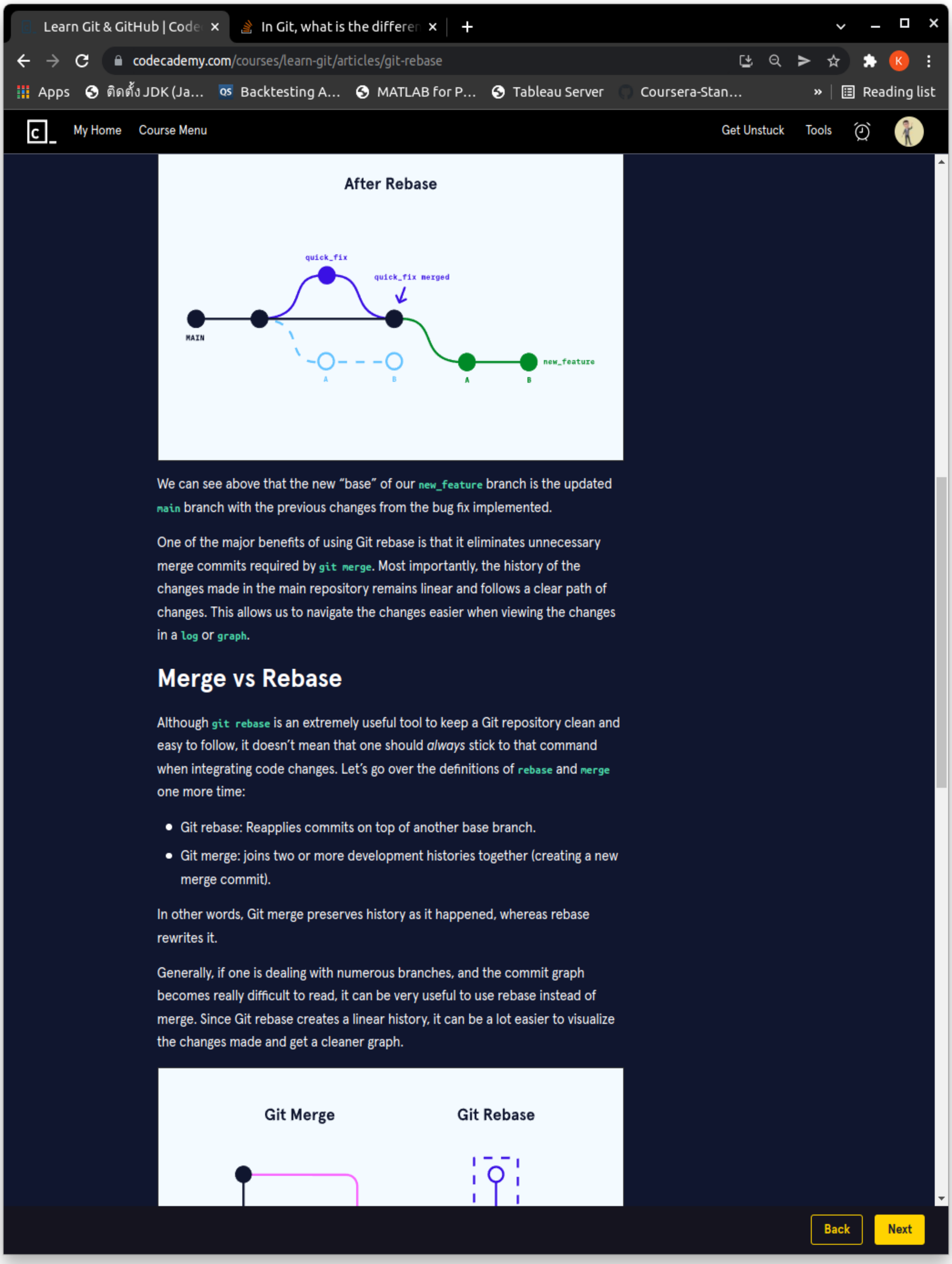
What is Git Rebase?

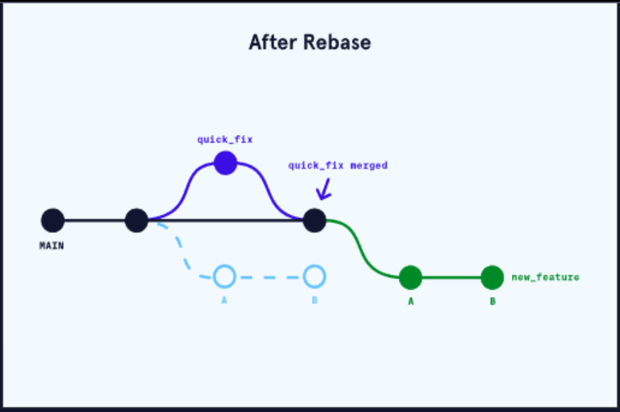
At a high level, rebasing can be understood as "moving the base of a branch onto a different position". Think of it like a redo – "I meant to start here."

Consider that a team just completed a production release. While working on a completely new feature branch called `new_feature`, a co-worker finds a bug in the production release (`main` branch). In order to fix this, a team member creates a `quick_fix` branch, squashes the bug, and merges their code in to the `main` branch. At this point, the `main` branch and the `new_feature` branch have diverged and they each have a different commit history. We can visualize this in the image below:



If we want to bring the updated changes from `main` into `new_feature` one could use the `merge` command, but with `rebase` we can keep the Git commit history clean and easy to follow. By "rebasing" the `new_feature` branch onto the `main` one, we move all the changes made from `new_feature` to the front of `main` and incorporate the new commits by rewriting its history. We can see how this is done below:





We can see above that the new “base” of our `new_feature` branch is the updated `main` branch with the previous changes from the bug fix implemented.

One of the major benefits of using Git rebase is that it eliminates unnecessary merge commits required by `git merge`. Most importantly, the history of the changes made in the main repository remains linear and follows a clear path of changes. This allows us to navigate the changes easier when viewing the changes in a `log` or `graph`.

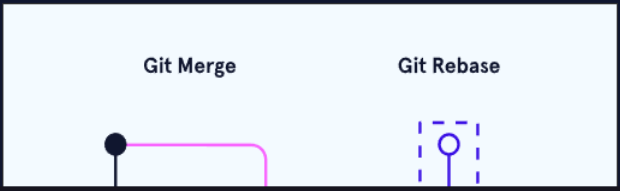
Merge vs Rebase

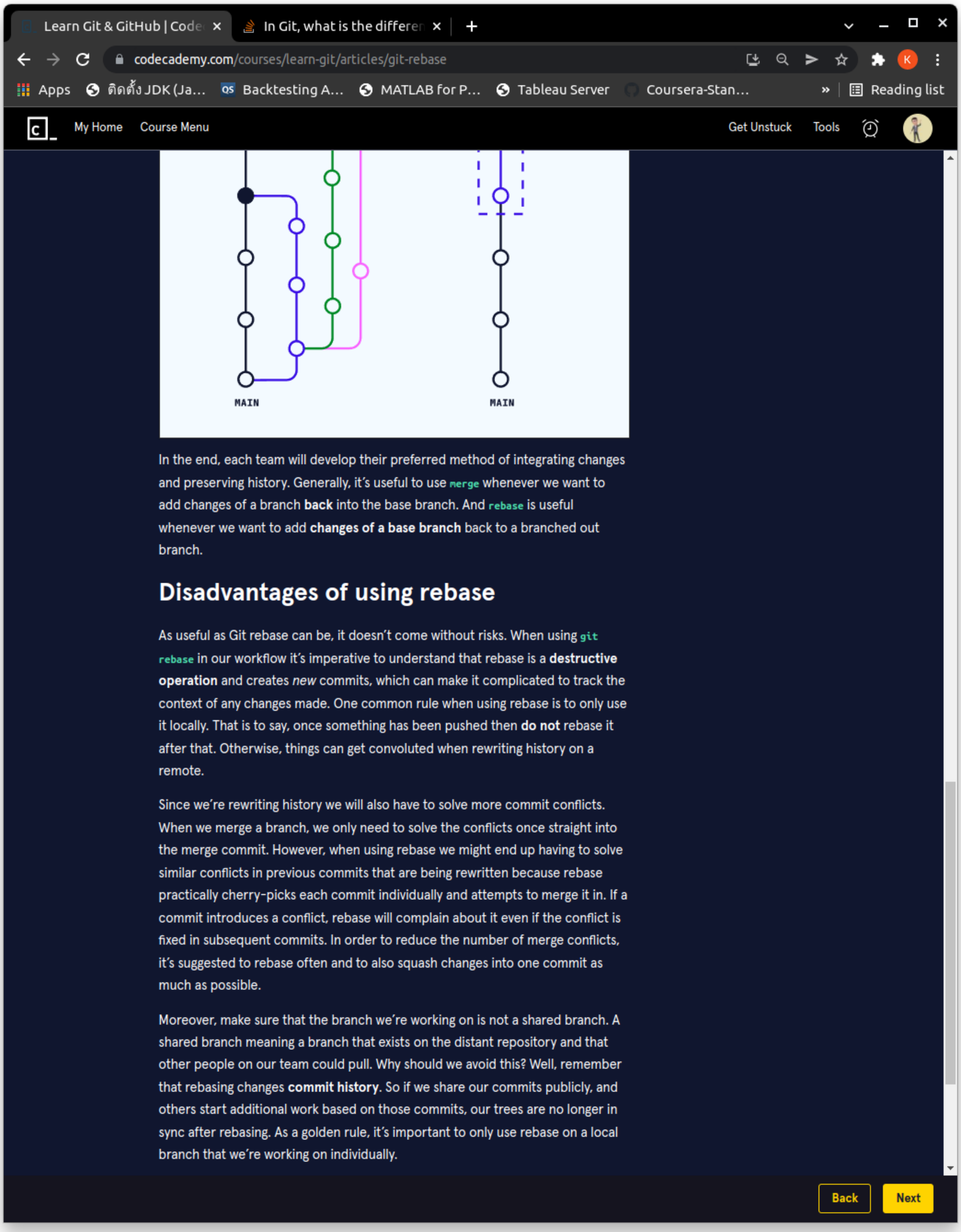
Although `git rebase` is an extremely useful tool to keep a Git repository clean and easy to follow, it doesn’t mean that one should *always* stick to that command when integrating code changes. Let’s go over the definitions of `rebase` and `merge` one more time:

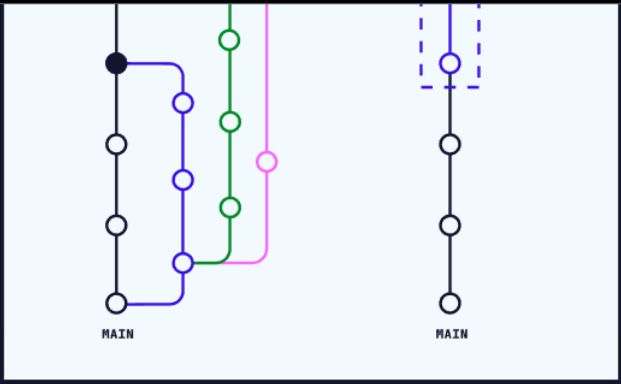
- Git rebase: Reapplies commits on top of another base branch.
- Git merge: joins two or more development histories together (creating a new merge commit).

In other words, Git merge preserves history as it happened, whereas rebase rewrites it.

Generally, if one is dealing with numerous branches, and the commit graph becomes really difficult to read, it can be very useful to use rebase instead of merge. Since Git rebase creates a linear history, it can be a lot easier to visualize the changes made and get a cleaner graph.







In the end, each team will develop their preferred method of integrating changes and preserving history. Generally, it's useful to use `merge` whenever we want to add changes of a branch **back** into the base branch. And `rebase` is useful whenever we want to add **changes of a base branch** back to a branched out branch.

Disadvantages of using rebase

As useful as Git rebase can be, it doesn't come without risks. When using `git rebase` in our workflow it's imperative to understand that rebase is a **destructive operation** and creates *new* commits, which can make it complicated to track the context of any changes made. One common rule when using rebase is to only use it locally. That is to say, once something has been pushed then **do not** rebase it after that. Otherwise, things can get convoluted when rewriting history on a remote.

Since we're rewriting history we will also have to solve more commit conflicts. When we merge a branch, we only need to solve the conflicts once straight into the merge commit. However, when using rebase we might end up having to solve similar conflicts in previous commits that are being rewritten because rebase practically cherry-picks each commit individually and attempts to merge it in. If a commit introduces a conflict, rebase will complain about it even if the conflict is fixed in subsequent commits. In order to reduce the number of merge conflicts, it's suggested to rebase often and to also squash changes into one commit as much as possible.

Moreover, make sure that the branch we're working on is not a shared branch. A shared branch meaning a branch that exists on the distant repository and that other people on our team could pull. Why should we avoid this? Well, remember that rebasing changes **commit history**. So if we share our commits publicly, and others start additional work based on those commits, our trees are no longer in sync after rebasing. As a golden rule, it's important to only use rebase on a local branch that we're working on individually.