**Git Rebase**

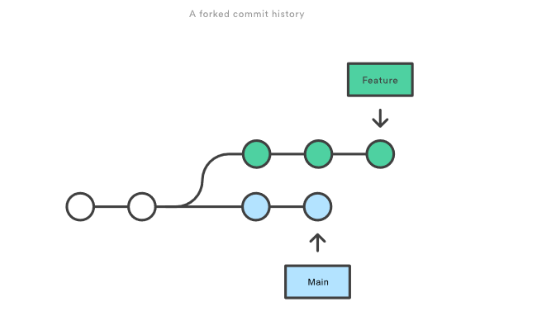
**What Is Git Rebase?**

Rebase is one of the Git utilities designed to integrate changes from one branch onto another. Rebasing is the process of combining or moving a sequence of commits on top of a new base commit. Git rebase is the linear process of merging.

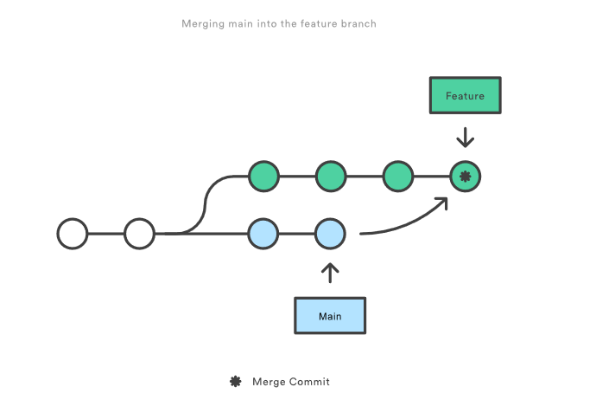
**What Does Git Rebase Do?**

A Git rebase changes the base of the developer’s branch from one commit to another, so it looks like they have created their branch from a different commit. Internally, Git creates a new commit and applies it to the specified base. However, it's essential for everyone involved to understand that although the branch appears the same, it's made up of entirely new commits. When you perform a Git rebase, you are, in effect, rewriting history.

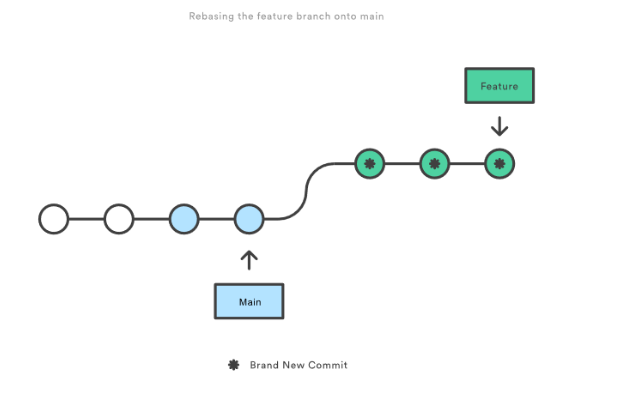
Here’s how Git rebasing compares to Git merging. Let's say you're a developer who is working on a new feature on a dedicated branch. Then, another **development team** member updates the main branch with some new commits. The situation looks like this:



Eventually, however, the team concludes that the main's new commits are relevant to the feature you are working on. So then, if you want to incorporate the new commits onto your branch, you can either do a merge or a rebase. If you decide to use Git merging, you tack on the new commits to your new branch like this:



However, if you use Git rebase, you move your whole feature branch, starting it on the tip of the main branch so that all the new commits are now part of the whole. This action rewrites the project history by making new commits for each of the original branch's commits. So, this is how the new branch looks:



**What Is Git Rebase: Git Rebase Usage**

Why do people use Git rebase? For one overriding reason: maintaining a linear project history. Let's say for instance that you've been working on a feature branch off the main branch, but the latter has progressed. But you want to get the main branch's latest updates into your feature branch while keeping your branch's history clean, so it looks like you've been working off the updated, latest main branch.

You will benefit from an eventual clean merge of your feature branch back into the main branch, perpetuating a clean history. It's essential to have a clean history, especially when conducting Git operations, to locate and investigate a possible regression introduced into the branch.

To sum it briefly, when you conduct a Git rebase, you’re saying that you want your changes to be based on what other developers have already done.

**Git Rebase Standard vs. Git Rebase Interactive**

There are two different Git rebase modes, standard and interactive. A standard mode Git rebase automatically grabs the commits present in your current working branch and immediately applies them to the head of the passed branch.

On the other hand, Interactive mode lets you change different commits in the process instead of just scooping up everything and tossing it into the passed branch. If you use interactive mode, you can remove, split, or alter existing commits and clean up the history, and we've already touched on why clean histories are essential.

So how do you perform a Git rebase?

**How to Git Rebase**

Here’s the syntax for launching a standard Git rebase:

git rebase <base>

And here’s the syntax for launching an interactive Git rebase:

git rebase --interactive <base>

This command opens an editor that lets you enter commands for each commit you want to rebase.

Later, we’ll explore a broader range of rebase commands. But before we do, we must discuss configuration.

**What Is Git Rebase: All About Configuration**

You can use git config to set some rebase properties if you want. Here are some configuration options you can take advantage of. Note that these options will alter the Git rebase output’s feel and look.

* rebase.stat- This is a boolean set to “false” by default. This option toggles the display of visual diffstat content showing what has changed since the last rebase.
* rebase.autoSquash- This boolean value toggles the --autosquash behavior.
* rebase.instructionFormat- This is a Git log format string used for formatting an interactive rebase display.
* rebase.missingCommitsCheck- This option can be set to multiple values, changing rebase behavior around missing commits. The values are:
  + Warn. This prints warning output in interactive mode and warns you of removed contents.
  + Error. This stops the rebase and prints any removed commit warning messages.
  + Ignore. This value is set by default and ignores missing commit warnings.

Now, about those rebase commands.

**Rebase Commands**

Here’s a summary of the different commands associated with Git rebase.

|  |  |
| --- | --- |
| git rebase <base> | Performs the standard rebase |
| git rebase – interactive <base> | Performs the interactive rebase |
| git rebase -- d | The commit gets discarded from the final combined commit block during playback. |
| git rebase -- p | This leaves the commit alone, not modifying the content or message, and keeping it as an individual commit in the branches’ history. |
| git rebase -- x | This executes a command line shell script for each marked commit during playback. |
| git status | Checks the rebase status. |
| git rebase -- continue | Continue with the changes that you made. |
| git rebase --skip | Skips the changes |
| git add <project file> | Adds your branch to the repository |
| git commit -m "new commit for <branch name>." | Commits the changes. |

**About Rebase Branch**

Sometimes, a developer will have many commits in different branches and want to combine them all into a single branch. There are two options: merge it or rebase it, and the latter is the best choice.

First, you need to switch to the branch in question:

git checkout <branch name>

Then, just rebase to the master.

git rebase master

**Git Advanced Rebase Application**

The  - - onto­ command activates a more powerful rebase type that lets you pass specific refs to become the tips of a rebase. For example, you can achieve this through this command:

 git rebase --onto <newbase> <oldbase>

**The Dangers of Rebase**

Git rebasing comes with risks, an understandable situation considering how it rewrites history. However, whenever users have the means to change a file that multiple users can access, it increases the risk of problems.

If your long-lived branch has strayed too far from the main, you may experience merge conflicts. In this case, you need to rebase against the main eventually, but the situation may have escalated because there are so many new commits that your branch changes will [conflict](https://www.simplilearn.com/tutorials/git-tutorial/merge-conflicts-in-git) with. You can avoid this problem by frequently rebasing your branch against the main and making more frequent commits.

When dealing with conflicts, you can reset or advance the rebase by passing the --continue and --abort command line arguments to the Git rebase.

Another rebase danger involves losing commits from interactive history writing. If you run an interactive mode rebase and execute subcommands such as drop or squash, you could remove commits from your branch’s immediate log. Use git reflog to restore the commits and undo the rebase.

Don't let these dangers deter you from rebasing in the final analysis. The only serious issue arises when you execute a history rewriting interactive rebase and end up force pushing the results onto a remote branch that other users share. Unfortunately, this could overwrite other users’ work when performing a pull.

**Recovering From Upstream Rebase**

Speaking of overwriting, how can you recover from a force push onto a branch you’re committing to? Just use git reflog and find a ref before it was rebased, then rebase the branch against the remote ref by doing the - - onto option.

**Git Rebase Master**

You can perform a Git master to branch rebase to integrate branch changes and updates into the master. Here’s the syntax for performing a master to branch rebase onto a develop branch.

 git rebase develop master

This tactic is helpful if both the master and develop branches have commits after the branch split off. The Git master to branch rebase will ensure that the master and develop branches have all the commits, regardless of where they originated.

Be careful, though. After the rebase is completed, Git creates new commits with new ids, then permanently deletes the old commits. If other users are working on branches that came from the deleted commits, they can’t push their changes back to their origin. They will need to perform a significant amount of work to push those developments back into the central depository.