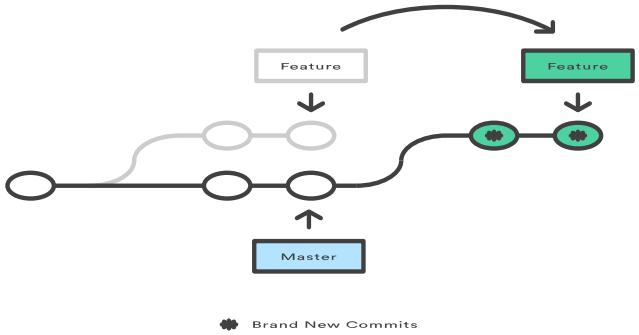
Git Rebase

Rebase is one of two Git utilities that specializes in integrating changes from one branch onto another. The other change integration utility is git merge. Merge is always a forward moving change record. Alternatively, rebase has powerful history rewriting features. For a detailed look at Merge vs. Rebase, visit our Merging vs Rebasing guide. Rebase itself has 2 main modes: "manual" and "interactive" mode.

What is Git Rebase?

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:



From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It's very important to understand that even though the branch looks the same, it's composed of entirely new commits.

Usage.

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature branch. You want to get the latest updates to the master branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest master branch. This gives the later benefit of a clean merge of your feature branch back into the master branch. Why do we want to maintain a "clean history"? The benefits of having a clean history become tangible when performing Git operations to investigate the introduction of a regression. A more real-world scenario would be:

1.A bug is identified in the master branch. A feature that was working successfully is now broken.

2.A developer examines the history of the master branch using *git log*because of the "clean history" the developer is quickly able to reason about the history of the project.

3.The developer can not identify when the bug was introduced using *git log* so the developer executes a *git bisect*.

4.Because the git history is clean,*git bisect* has a refined set of commits to compare when looking for the regression. The developer quickly finds the commit that introduced the bug and is able to act accordingly.

Configuration options

There are a few rebase properties that can be set using *git config*. These options will alter the*git rebase* output look and feel.

•**rebase.stat**: A boolean that is set to false by default. The option toggles display of visual diffstat content that shows what changed since the last debase.

•**rebase.autoSquash:** A boolean value that toggles the –autosquash behavior.

•**rebase.missingCommitsCheck:** Can be set to multiple values which change rebase behavior around missing commits.

warn	Prints warning output in interactive mode which warns of removed commits
error	Stops the rebase and prints removed commit warning messages
ignore	Set by default this ignores any missing commit warnings

•**rebase.instructionFormat:** A *git log* format string that will be used for formatting interactive rebase display

Git Rebase Standard vs Git Rebase Interactive

Git rebase interactive is when git rebase accepts an – i argument. This stands for "Interactive." Without any arguments, the command runs in standard mode. In both cases, let's assume we have created a separate feature branch.

```
# Create a feature branch based off of master
git checkout -b feature_branch master
# Edit files
git commit -a -m "Adds new feature"
```

Git rebase in standard mode will automatically take the commits in your current working branch and apply them to the head of the passed branch.

git rebase <base>

This automatically rebases the current branch onto <base>, which can be any kind of commit reference (for example an ID, a branch name, a tag, or a relative reference to HEAD).

Running git rebase with the -i flag begins an interactive rebasing session. Instead of blindly moving all of the commits to the new base, interactive rebasing gives you the opportunity to alter individual commits in the process. This lets you clean up history by removing, splitting, and altering an existing series of commits. It's like Git commit –amend on steroids.

git rebase --interactive <base>

This rebases the current branch onto <base> but uses an interactive rebasing session. This opens an editor where you can enter commands (described below) for each commit to be rebased. These commands determine how individual commits will be transferred to the new base. You can also reorder the commit listing to change the order of the commits themselves. Once you've specified commands for each commit in the rebase, Git will begin playing back commits applying the rebase commands. The rebasing edit commands are as follows:

```
pick 2231360 some old commit
pick ee2adc2 Adds new feature
# Rebase 2cf755d..ee2adc2 onto 2cf755d (9 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
```

Additional rebase commands

•*git rebase – d* means during playback the commit will be discarded from the final combined commit block.

•*git rebase – p* leaves the commit as is. It will not modify the commit's message or content and will still be an individual commit in the branches history.

•*git rebase – x* during playback executes a command line shell script on each marked commit. A useful example would be to run your codebase's test suite on specific commits, which may help identify regressions during a rebase.