A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Most commonly, forks are used to either propose changes to someone else's project or to use someone else's project as a starting point for your own idea.

[Propose changes to someone else's project](https://help.github.com/en/articles/fork-a-repo#propose-changes-to-someone-elses-project)

A great example of using forks to propose changes is for bug fixes. Rather than logging an issue for a bug you've found, you can:

* Fork the repository.
* Make the fix.
* Submit a *pull request* to the project owner.

If the project owner likes your work, they might pull your fix into the original repository!

[Use someone else's project as a starting point for your own idea.](https://help.github.com/en/articles/fork-a-repo#use-someone-elses-project-as-a-starting-point-for-your-own-idea)

At [the heart of open source](http://opensource.org/about) is the idea that by sharing code, we can make better, more reliable software.

When creating your public repository from a fork of someone's project, make sure to include a [license file](http://choosealicense.com/) that determines how you want your project to be shared with others.

# About pull requests

Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.

After initializing a pull request, you'll see a review page that shows a high-level overview of the changes between your branch (the compare branch) and the repository's base branch.

Other contributors can review your proposed changes, add review comments, contribute to the pull request discussion, and even add commits to the pull request.

## Stashing your work

The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. For example:

Popping your stash removes the changes from your stash and reapplies them to your working copy.

Alternatively, you can reapply the changes to your working copy and keep them in your stash with git stash apply:

## Stashing untracked or ignored files

By default, running git stash will stash:

* changes that have been added to your index (staged changes)
* changes made to files that are currently tracked by Git (unstaged changes)

But it will **not** stash:

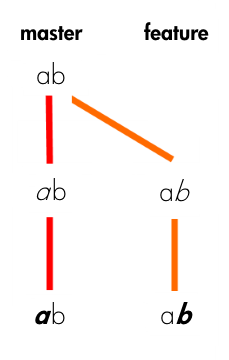
* new files in your working copy that have not yet been staged
* files that have been [ignored](https://www.atlassian.com/git/tutorials/gitignore)

Git Merge and Rebase:

[git merge](https://hackernoon.com/tagged/git-merge) and git rebase offer the same service: incorporating commits from one Git branch into another. The key distinction lies in how this result is achieved.

Often in Git workflows, developers will create feature branches to work on new features in isolation. This gives them the freedom to make incremental commits for an in-progress feature without affecting the project’s master branch (more information about using feature branches can be found in [Atlassian’s Git tutorials](https://www.atlassian.com/git/tutorials/comparing-workflows" \l "feature-branch-workflow" \t "_blank)).

In our project’s Git repository, two branches currently exist: the usual master branch, and a featurebranch that we created right after the initial commit. On the master branch, we have italicized the ‘a’, then bolded the ‘a’. On our feature branch, we have italicized the ‘b’, and then bolded the ‘b’.



The master and feature branches for the most complicated web project of 2017.



The commit logs/histories for the master and feature branches (read from top to bottom).

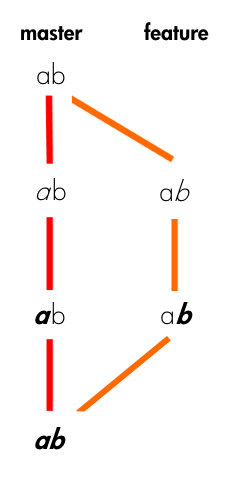
Let’s figure out how merge and rebase differ by going through a couple of typical Git workflow situations with this project.

### Merge

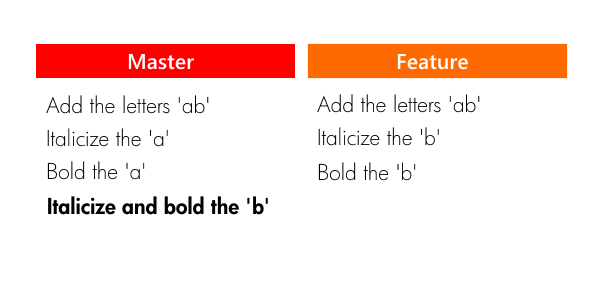
We believe our redesign of the letter ‘b’ is nothing short of a masterpiece, so we decide we want to bring our work back into the master branch, incorporating it into the actual project:

git checkout master  
git merge feature

By merging **feature**into **master**, master obtains a new commit — a “merge commit”.



Merging master into our feature branch. “Let’s just smush these branches together”.



Branch histories after the merge, with master’s new merge commit.

All by itself, the merge commit represents every change that has occurred on featuresince it branched from master. Clean and simple.

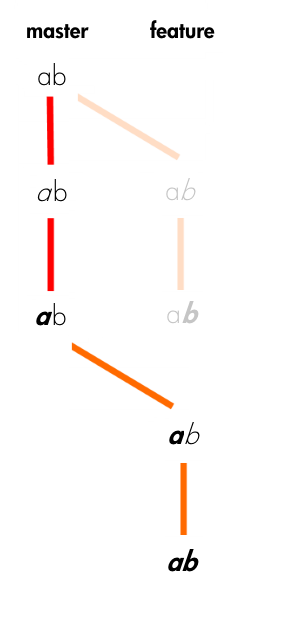
### Rebase

Let’s rewind, and pretend that we instead wanted to keep working on styling the letter ‘b’ — maybe change its size, font, color, etc. We need to get it just right — this change affects 50% of our codebase! But before we continue working on our feature branch, we decide we want to bring in the latest changes from master to keep things fresh.

Rather than merging master’s new commits into feature, we opt to rebase our feature branch onto master.

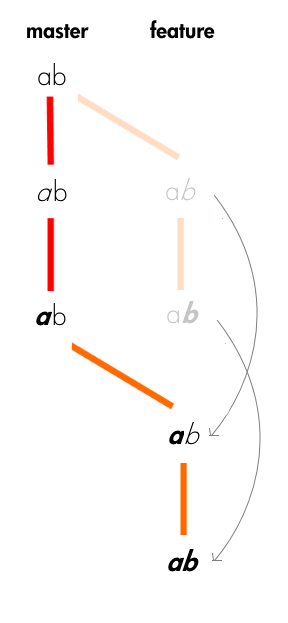
git checkout feature  
git rebase master

At a high level, rebasing can be understood as “moving the baseof a branch onto a different position”. Think of it like a redo — “I meant to start here.”



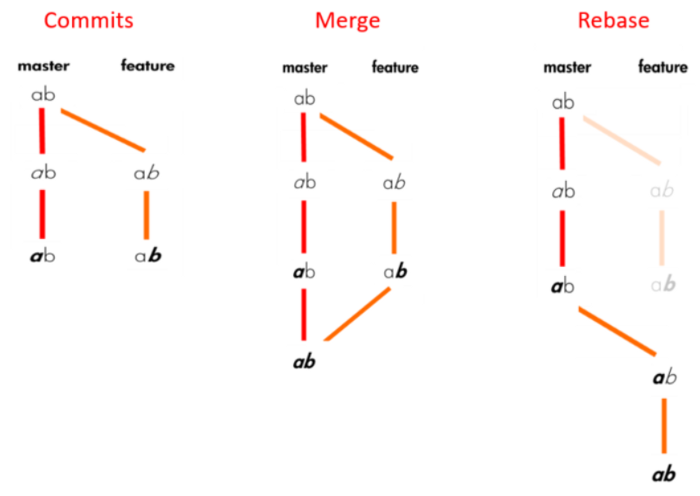
The result of rebasing our feature branch onto master.

At a lower level, what rebase actually does is pluck commits from a branch one by one (chronologically) and re-attach them to a different commit. The point at which the branch…branched has now changed.



What actually happens when rebasing our feature branch onto master. We are essentially replaying the feature branch commits from a new starting point.

You might see from the diagrams above why we would choose to rebase instead of merge in this situation. Unlike with merging, rebase does not create an extra commit. This is ideal for our situation, since all we are currently trying to do is keep our feature branch up-to-date with any new commits from master. This is definitely not a meaningful event we want to preserve in our project’s history.



**When to rebase? When to Merge?**

If the feature branch you are getting changes from is shared with other developers, rebasing is not recommended, because the rebasing process will create inconsistent repositories. For individuals, rebasing makes a lot of sense.

If you want to see the history completely same as it happened, you should use merge. **Merge preserves history whereas rebase rewrites it.**

Rebasing is better to streamline a complex history, you are able to change the commit history by [**interactive rebase**](https://medium.com/@filissen/git-interactive-rebase-e265bb55952a). You can remove undesired commits, squash two or more commits into one or edit the commit message.

Rebase will present conflicts one commit at a time whereas merge will present them all at once. It is better and much easier to handle the conflicts but you shouldn’t forget that reverting a rebase is much more difficult than reverting a merge if there are many conflicts.

### Fetch

$ git fetch origin

**git fetch** really only downloads new data from a remote repository - but it doesn't integrate any of this new data into your working files. Fetch is great for getting a fresh view on all the things that happened in a remote repository.  
Due to it's "harmless" nature, you can rest assured: fetch will never manipulate, destroy, or screw up anything. This means you can never fetch often enough.

### Pull

$ git pull origin master

**git pull**, in contrast, is used with a different goal in mind: to update your current HEAD branch with the latest changes from the remote server. This means that pull not only downloads new data; it also directly **integrates** it into your current working copy files. This has a couple of consequences:

* Since "git pull" tries to merge remote changes with your local ones, a so-called "merge conflict" can occur. Check out our in-depth tutorial on [How to deal with merge conflicts](https://www.git-tower.com/learn/git/ebook/en/command-line/advanced-topics/merge-conflicts) for more information.
* Like for many other actions, it's highly recommended to start a "git pull" only with a clean working copy. This means that you should not have any uncommitted local changes before you pull

Basic Diff:

git clone means you are making a copy of the repository in your system.

git fork means you are copying the repository to your Github account.

git pull means you are fetching the last modified repository.

git push means you are returning the repository after modifying it.

In layman's term:

git clone is downloading and git pull is refreshing.

**git clone** is used for just downloading exactly what is currently working on the remote server repository and saving it in your machine's folder where that project is placed. Mostly it is used only when we are going to upload the project for the first time. After that pull is the better option.

**git pull** is basically a (clone(download) + merge) operation and mostly used when you are working as teamwork. In other words, when you want the recent changes in that project, you can pull.