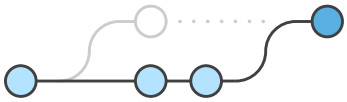


Rewriting history

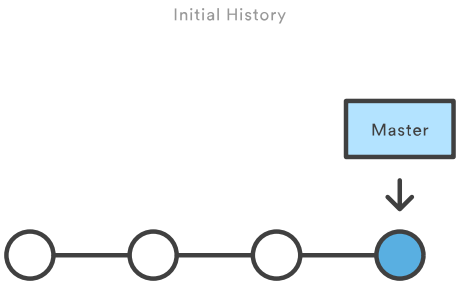
Intro

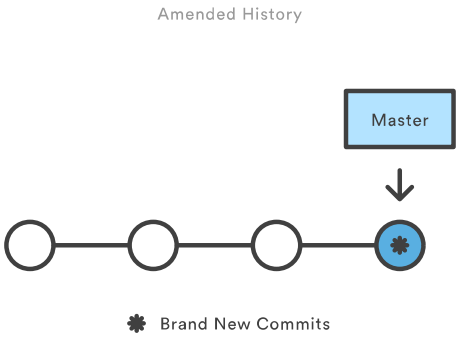
Git's main job is to make sure you never lose a committed change. But, it's also designed to give you total control over your development workflow. This includes letting you define exactly what your project history looks like; however, it also creates the potential to lose commits. Git provides its history-rewriting commands under the disclaimer that using them may result in lost content.

This tutorial discusses some of the most common reasons for overwriting committed snapshots and shows you how to avoid the pitfalls of doing so.

Tutorials

The `git commit --amend` command is a convenient way to fix up the most recent commit. It lets you combine staged changes with the previous commit instead of committing it as an entirely new snapshot. It can also be used to simply edit the previous commit message without changing its snapshot.





But, amending doesn't just alter the most recent commit—it replaces it entirely. To Git, it will look like a brand new commit, which is visualized with an asterisk (*) in the diagram above. It's important to keep this in mind when working with public repositories.

Usage

Tutorials

and replace the previous commit with the resulting snapshot. Running this when there is nothing staged lets you edit the previous commit's message without altering its snapshot.

Discussion

Premature commits happen all the time in the course of your everyday development. It's easy to forget to stage a file or to format your commit message the wrong way. The `--amend` flag is a convenient way to fix these little mistakes.

Don't Amend Public Commits

On the `git reset` page, we talked about how you should never reset commits that have been shared with other developers. The same goes for amending: never amend commits that have been pushed to a public repository.

Amended commits are actually entirely new commits, and the previous commit is removed from the project history. This has the same consequences as resetting a public snapshot. If you amend a commit that other developers have based their work on, it will look like the basis of their work vanished from the project history. This is a confusing situation for developers to be in and it's complicated to recover from.

Example

The following example demonstrates a common

around. Fixing the error is simply a matter of staging the other file and committing with the `--amend` flag:

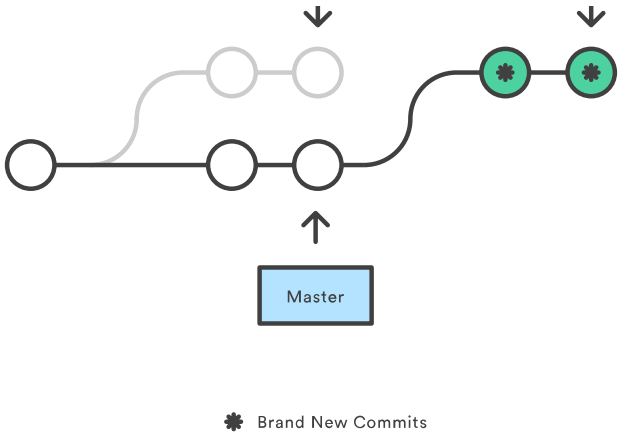
```
# Edit hello.py and main.py
git add hello.py
git commit

# Realize you forgot to add the changes from main
git add main.py
git commit --amend --no-edit
```

The editor will be populated with the message from the previous commit and including the `--no-edit` flag will allow you to make the amendment to your commit without changing its commit message. You can change it if necessary, otherwise just save and close the file as usual. The resulting commit will replace the incomplete one, and it will look like we committed the changes to `hello.py` and `main.py` in a single snapshot.

git rebase

Rebasing is the process of moving a branch to a new base commit. The general process can be visualized as the following:

Tutorials



From a content perspective, rebasing really is just moving a branch from one commit to another. But internally, Git accomplishes this by creating new commits and applying them to the specified base—it's literally rewriting your project history. It's very important to understand that, even though the branch looks the same, it's composed of entirely new commits.

Usage

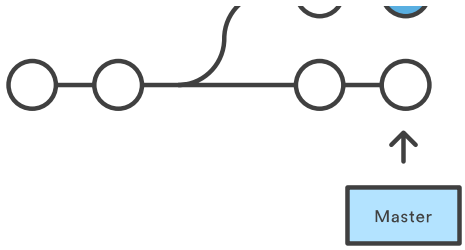
```
git rebase <base>
```

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

Discussion

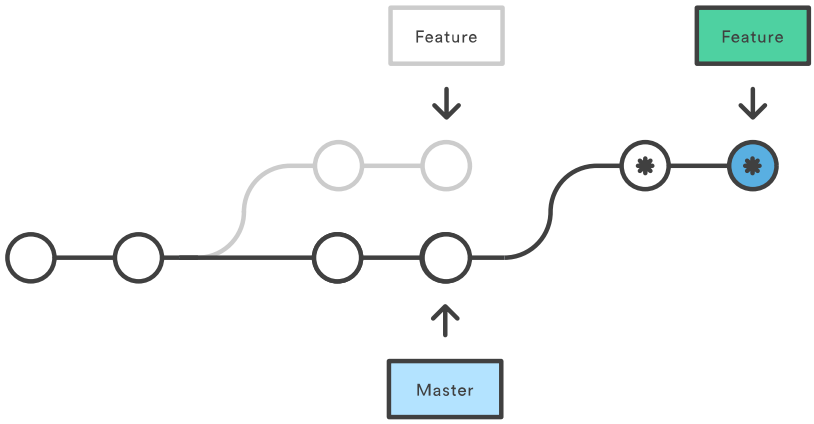
The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature:

Tutorials

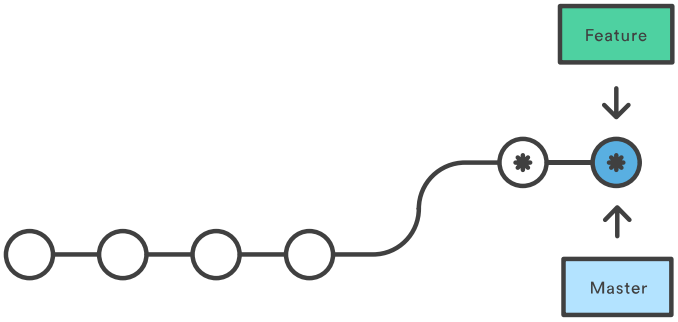


You have two options for integrating your feature into the `master` branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history. The following diagram demonstrates how rebasing onto `master` facilitates a fast-forward merge.

Tutorials



After a Fast-Forward Merge



Brand New Commits

Rebasing is a common way to integrate upstream changes into your local repository. Pulling in upstream changes with `git merge` results in a superfluous merge commit every time you want to see how the project has progressed. On the other hand, rebasing is like saying, "I want to base my changes on what everybody has already done."

Don't Rebase Public History

As we've discussed with `git commit --amend` and `git reset`, you should never rebase commits that have

Tutorials

vanished.

Examples

The example below combines git rebase with git merge to maintain a linear project history. This is a quick and easy way to ensure that your merges will be fast-forwarded.

```
# Start a new feature
git checkout -b new-feature master
# Edit files
git commit -a -m "Start developing a feature"
```

In the middle of our feature, we realize there's a security hole in our project

```
# Create a hotfix branch based off of master
git checkout -b hotfix master
# Edit files
git commit -a -m "Fix security hole"
# Merge back into master
git checkout master
git merge hotfix
git branch -d hotfix
```

After merging the hotfix into master, we have a forked project history. Instead of a plain git merge, we'll integrate the feature branch with a rebase to maintain a linear history:

```
git checkout new-feature
git rebase master
```

This moves new-feature to the tip of master, which lets us do a standard fast-forward merge from master:

Tutorials

git rebase -i

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.

Usage

```
git rebase -i <base>
```

Rebase the current branch onto `<base>`, but use 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.

Discussion

Interactive rebasing gives you complete control over what your project history looks like. This affords a lot of freedom to developers, as it lets them commit a "messy" history while they're focused on writing code, then go back and clean it up after the fact.

Tutorials

insignificant commits, delete obsolete ones, and make sure everything else is in order before committing to the “official” project history. To everybody else, it will look like the entire feature was developed in a single series of well-planned commits.

Examples

The example found below is an interactive adaptation of the one from the non-interactive `git rebase` page.

```
# Start a new feature
git checkout -b new-feature master
# Edit files
git commit -a -m "Start developing a feature"
# Edit more files
git commit -a -m "Fix something from the previous

# Add a commit directly to master
git checkout master
# Edit files
git commit -a -m "Fix security hole"

# Begin an interactive rebasing session
git checkout new-feature
git rebase -i master
```

The last command will open an editor populated with the two commits from new-feature, along with some instructions:

```
pick 32618c4 Start developing a feature
pick 62eed47 Fix something from the previous comm
```

You can change the pick commands before each commit to determine how it gets moved during the rebase. In our case, let’s just combine the two commits with a squash command:

Tutorials

Getting Started

Setting up a repository

Saving changes

Inspecting a repository

Viewing old commits

Undoing Changes

Rewriting history

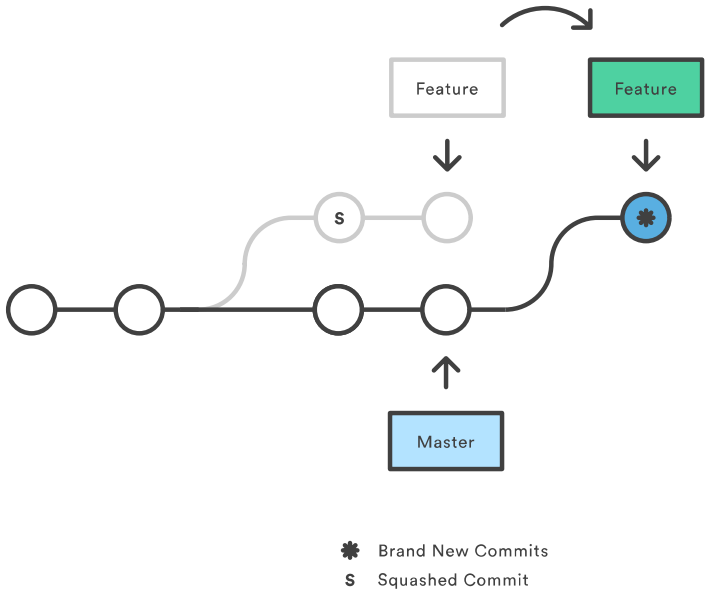
- git commit --amend
- git rebase
- git rebase -i
- git reflog

Collaborating

Migrating to Git

Advanced Tips

Save and close the editor to begin the rebase. This will open another editor asking for the commit message for the combined snapshot. After defining the commit message, the rebase is complete and you should be able to see the squashed commit in your `git log` output. This entire process can be visualized as follows:



Note that the squashed commit has a different ID than either of the original commits, which tells us that it is indeed a brand new commit.

Finally, you can do a fast-forward merge to integrate the polished feature branch into the main code base:

```
git checkout master
git merge new-feature
```

The real power of interactive rebasing can be seen in the history of the resulting master branch—the extra `62eed47` commit is nowhere to be found. To everybody else, it looks like you're a brilliant developer who

meaningful.

git reflog

Git keeps track of updates to the tip of branches using a mechanism called reflog. This allows you to go back to changesets even though they are not referenced by any branch or tag. After rewriting history, the reflog contains information about the old state of branches and allows you to go back to that state if necessary.

Usage

```
git reflog
```

Show the reflog for the local repository.

```
git reflog --relative-date
```

Show the reflog with relative date information (e.g. 2 weeks ago).

Discussion

Every time the current HEAD gets updated (by switching branches, pulling in new changes, rewriting history or simply by adding new commits) a new entry will be added to the reflog.

Tutorials

example.

```
0a2e358 HEAD@{0}: reset: moving to HEAD~2
0254ea7 HEAD@{1}: checkout: moving from 2.2 to mas
c10f740 HEAD@{2}: checkout: moving from master to
```

The reflog above shows a checkout from master to the 2.2 branch and back. From there, there's a hard reset to an older commit. The latest activity is represented at the top labeled `HEAD@{0}`.

If it turns out that you accidentially moved back, the reflog will contain the commit master pointed to (0254ea7) before you accidentially dropped 2 commits.

```
git reset --hard 0254ea7
```

Using `git reset` it is then possible to change master back to the commit it was before. This provides a safety net in case history was accidentially changed.

It's important to note that the reflog only provides a safety net if changes have been commited to your local repository and that it only tracks movements.

Tutorials

Enter Your Email For Git News

Getting Started

Setting up a repository

Saving changes

Inspecting a repository

Viewing old commits

Undoing Changes

Rewriting history

- git commit --amend
- git rebase
- git rebase -i
- git reflog

Collaborating

Migrating to Git

Advanced Tips