# Rewriting history

###### Git commit --amend and other methods of rewriting history

## Intro

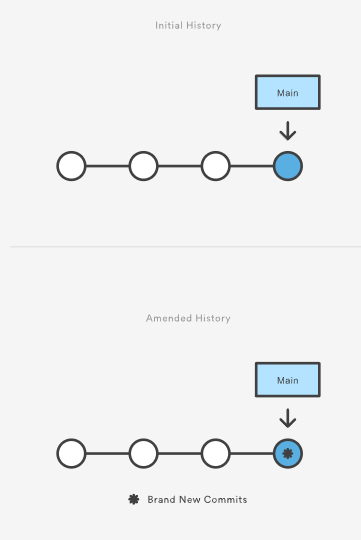
This tutorial will cover various methods of rewriting and altering Git history. Git uses a few different methods to record changes. We will discuss the strengths and weaknesses of the different methods and give examples of how to work with them. This tutorial discusses some of the most common reasons for overwriting committed snapshots and shows you how to avoid the pitfalls of doing so.

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 of losing commits. Git provides its history-rewriting commands under the disclaimer that using them may result in lost content.

Git has several mechanisms for storing history and saving changes. These mechanisms include: Commit --amend, git rebase and git reflog. These options give you powerful work flow customization options. By the end of this tutorial, you'll be familiar with commands that will let you restructure your Git commits, and be able to avoid pitfalls that are commonly encountered when rewriting history.

## Changing the Last Commit: git commit --amend

The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot. But, amending does not just alter the most recent commit, it replaces it entirely, meaning the amended commit will be a new entity with its own ref. To Git, it will look like a brand new commit, which is visualized with an asterisk (\*) in the diagram below. There are a few common scenarios for using git commit --amend. We'll cover usage examples in the following sections.



### Change most recent Git commit message

git commit --amend

Let's say you just committed and you made a mistake in your commit log message. Running this command when there is nothing staged lets you edit the previous commit’s message without altering its snapshot.

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 minor mistakes.

git commit --amend -m "an updated commit message"

Adding the -m option allows you to pass in a new message from the command line without being prompted to open an editor.

### Changing committed files

The following example demonstrates a common scenario in Git-based development. Let's say we've edited a few files that we would like to commit in a single snapshot, but then we forget to add one of the files the first time 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.py   
git add main.py   
git commit --amend --no-edit

The --no-edit flag will allow you to make the amendment to your commit without changing its commit message. 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.

### Don’t amend public commits

Amended commits are actually entirely new commits and the previous commit will no longer be on your current branch. This has the same consequences as resetting a public snapshot. Avoid amending a commit that other developers have based their work on. This is a confusing situation for developers to be in and it’s complicated to recover from.

### Recap

To review, git commit --amend lets you take the most recent commit and add new staged changes to it. You can add or remove changes from the Git staging area to apply with a --amend commit. If there are no changes staged, a --amend will still prompt you to modify the last commit message log. Be cautious when using --amend on commits shared with other team members. Amending a commit that is shared with another user will potentially require confusing and lengthy merge conflict resolutions.

## Changing older or multiple commits

To modify older or multiple commits, you can use git rebase to combine a sequence of commits into a new base commit. In standard mode, git rebase allows you to literally rewrite history — automatically applying commits in your current working branch to the passed branch head. Since your new commits will be replacing the old, it's important to not use git rebase on commits that have been pushed public, or it will appear that your project history disappeared.

In these or similar instances where it's important to preserve a clean project history, adding the -i option to git rebase allows you to run rebase interactive. This gives you the opportunity to alter individual commits in the process, rather than moving all commits. You can learn more about interactive rebasing and additional rebase commands on the [git rebase page](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase).

#### Changing committed files

During a rebase, the edit or e command will pause the rebase playback on that commit and allow you to make additional changes with git commit --amend Git will interrupt the playback and present a message:

Stopped at 5d025d1... formatting  
You can amend the commit now, with  
git commit --amend  
Once you are satisfied with your changes, run  
 git rebase --continue

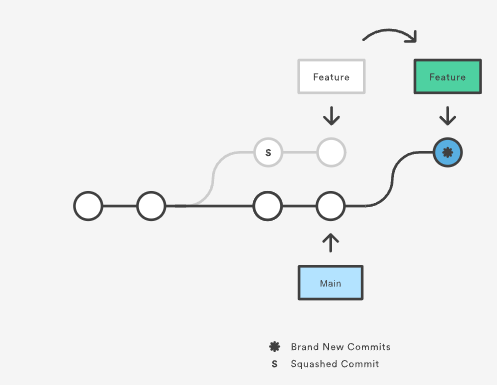
#### Multiple messages

Each regular Git commit will have a log message explaining what happened in the commit. These messages provide valuable insight into the project history. During a rebase, you can run a few commands on commits to modify commit messages.

* Reword or 'r' will stop rebase playback and let you rewrite the individual commit message during.
* Squash or 's' during rebase playback, any commits marked s will be paused on and you will be prompted to edit the separate commit messages into a combined message. More on this in the squash commits section below.
* Fixup or 'f' has the same combining effect as squash. Unlike squash, fixup commits will not interrupt rebase playback to open an editor to combine commit messages. The commits marked 'f' will have their messages discarded in-favor of the previous commit's message.

#### **Squash commits for a clean history**

The s "squash" command is where we see the true utility of rebase. Squash allows you to specify which commits you want to merge into the previous commits. This is what enables a "clean history." During rebase playback, Git will execute the specified rebase command for each commit. In the case of squash commits, Git will open your configured text editor and prompt to combine the specified commit messages. This entire process can be visualized as follows:



Note that the commits modified with a rebase command have a different ID than either of the original commits. Commits marked with pick will have a new ID if the previous commits have been rewritten.

Modern Git hosting solutions like Bitbucket now offer "auto squashing" features upon merge. These features will automatically rebase and squash a branch's commits for you when utilizing the hosted solutions UI. For more info see "[Squash commits when merging a Git branch with Bitbucket](https://bitbucket.org/blog/git-squash-commits-merging-bitbucket)."

### Recap

Git rebase gives you the power to modify your history, and interactive rebasing allows you to do so without leaving a “messy” trail. This creates the freedom to make and correct errors and refine your work, while still maintaining a clean, linear project history.

## The safety net: git reflog

Reference logs, or "reflogs" are a mechanism Git uses to record updates applied to tips of branches and other commit references. Reflog allows you to go back to commits 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. Every time your branch tip is updated for any reason (by switching branches, pulling in new changes, rewriting history or simply by adding new commits), a new entry will be added to the reflog. In this section we will take a high level look at the git reflog command and explore some common uses.

### Usage

git reflog

This displays the reflog for the local repository.

git reflog --relative-date

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

### Example

To understand git reflog, let's run through an example.

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

The reflog above shows a checkout from main 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 accidentally moved back, the reflog will contain the commit main pointed to (0254ea7) before you accidentally dropped 2 commits.

git reset --hard 0254ea7

Using Git reset, it is now possible to change main back to the commit it was before. This provides a safety net in case the history was accidentally changed.

It's important to note that the reflog only provides a safety net if changes have been committed to your local repository and that it only tracks movements of the repositories branch tip. Additionally reflog entries have an expiration date. The default expiration time for reflog entries is 90 days.

For additional information, see our [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog) page.

## Summary

In this article we discussed several methods of changing git history, and undoing git changes. We took a high level look at the git rebase process. Some Key takeaways are:

* There are many ways to rewrite history with git.
* Use git commit --amend to change your latest log message.
* Use git commit --amend to make modifications to the most recent commit.
* Use git rebase to combine commits and modify history of a branch.
* git rebase -i gives much more fine grained control over history modifications than a standard git rebase.

Learn more about the commands we covered at their individual pages:

* [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase)
* [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog)

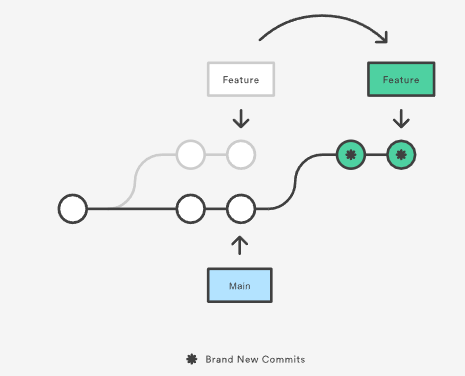
# git rebase

This document will serve as an in-depth discussion of the git rebase command. The Rebase command has also been looked at on the [setting up a repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository) and [rewriting history](https://www.atlassian.com/git/tutorials/rewriting-history) pages. This page will take a more detailed look at git rebase configuration and execution. Common Rebase use cases and pitfalls will be covered here.

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](https://www.atlassian.com/git/tutorials/merging-vs-rebasing). Rebase itself has 2 main modes: "manual" and "interactive" mode. We will cover the different Rebase modes in more detail below.

## 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.

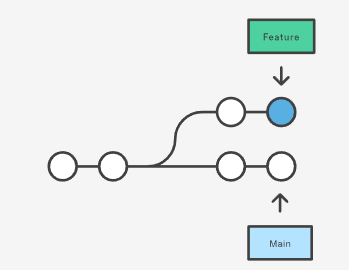
## Usage

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the main branch has progressed since you started working on a feature branch. You want to get the latest updates to the main 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 main branch. This gives the later benefit of a clean merge of your feature branch back into the main 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 main branch. A feature that was working successfully is now broken.
2. A developer examines the history of the main 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.

Learn more about [git log](https://www.atlassian.com/git/tutorials/git-log) and [git bisect](https://git-scm.com/docs/git-bisect) on their individual usage pages.

You have two options for integrating your feature into the main 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 the main branch facilitates a fast-forward merge.



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 previously in [rewriting history](https://www.atlassian.com/git/tutorials/rewriting-history), you should never rebase commits once they've been pushed to a public repository. The rebase would replace the old commits with new ones and it would look like that part of your project history abruptly vanished.

### 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 main   
git checkout -b feature\_branch main  
# 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 , 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  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

As detailed in the [rewriting history page](https://www.atlassian.com/git/tutorials/rewriting-history), rebasing can be used to change older and multiple commits, committed files, and multiple messages. While these are the most common applications, git rebase also has additional command options that can be useful in more complex applications.

* 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.

### Recap

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.

Most developers like to use an interactive rebase to polish a feature branch before merging it into the main code base. This gives them the opportunity to squash 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.

The real power of interactive rebasing can be seen in the history of the resulting main branch. To everybody else, it looks like you're a brilliant developer who implemented the new feature with the perfect amount of commits the first time around. This is how interactive rebasing can keep a project's history clean and meaningful.

### 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 rebase.
* 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

### Advanced rebase application

The command line argument --onto can be passed to git rebase. When in git rebase --onto mode the command expands to:

 git rebase --onto <newbase> <oldbase>

The --onto command enables a more powerful form or rebase that allows passing specific refs to be the tips of a rebase.  
Let’s say we have an example repo with branches like:

   o---o---o---o---o  main  
        \  
         o---o---o---o---o  featureA  
              \  
               o---o---o  featureB

featureB is based on featureA, however, we realize featureB is not dependent on any of the changes in featureA and could just be branched off main.

 git rebase --onto main featureA featureB

featureA is the < oldbase >. main becomes the < newbase > and featureB is reference for what HEAD of the < newbase > will point to. The results are then:

                      o---o---o  featureB  
                     /  
    o---o---o---o---o  main  
     \  
      o---o---o---o---o  featureA

## Understanding the dangers of rebase

One caveat to consider when working with Git Rebase is merge conflicts may become more frequent during a rebase workflow. This occurs if you have a long-lived branch that has strayed from main. Eventually you will want to rebase against main and at that time it may contain many new commits that your branch changes may conflict with. This is easily remedied by rebasing your branch frequently against main, and making more frequent commits. The --continue and --abort command line arguments can be passed to git rebase to advance or reset the the rebase when dealing with conflicts.

A more serious rebase caveat is lost commits from interactive history rewriting. Running rebase in interactive mode and executing subcommands like squash or drop will remove commits from your branche's immediate log. At first glance this can appear as though the commits are permanently gone. Using git reflog these commits can be restored and the entire rebase can be undone. For more info on using git reflog to find lost commits, visit our [Git reflog documentation page](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog).

Git Rebase itself is not seriously dangerous. The real danger cases arise when executing history rewriting interactive rebases and force pushing the results to a remote branch that's shared by other users. This is a pattern that should be avoided as it has the capability to overwrite other remote users' work when they pull.

## Recovering from upstream rebase

If another user has rebased and force pushed to the branch that you’re committing to, a git pull will then overwrite any commits you have based off that previous branch with the tip that was force pushed. Luckily, using git reflog you can get the reflog of the remote branch. On the remote branch's reflog you can find a ref before it was rebased. You can then rebase your branch against that remote ref using the --onto option as discussed above in the Advanced Rebase Application section.

## Summary

In this article we covered git rebase usage. We discussed basic and advanced use cases and more advanced examples. Some key discussion points are:

* git rebase standard vs interactive modes
* git rebase configuration options
* git rebase --onto
* git rebase lost commits

We looked at git rebase usage with other tools like [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog), [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch), and [git push](https://www.atlassian.com/git/tutorials/syncing/git-push). Visit their corresponding pages for further information.

# git reflog

This page provides a detailed discussion of the git reflog command. Git keeps track of updates to the tip of branches using a mechanism called reference logs, or "reflogs." Many Git commands accept a parameter for specifying a reference or "ref", which is a pointer to a commit. Common examples include:

* git checkout
* git reset
* git merge

Reflogs track when Git refs were updated in the local repository. In addition to branch tip reflogs, a special reflog is maintained for the Git stash. Reflogs are stored in directories under the local repository's .git directory. git reflog directories can be found at .git/logs/refs/heads/., .git/logs/HEAD, and also .git/logs/refs/stash if the git stash has been used on the repo.

We discussed git reflog at a high level on the [Rewriting History Page](https://www.atlassian.com/git/tutorials/rewriting-history). This document will cover: extended configuration options of git reflog, common use-cases and pitfalls of git reflog, how to undo changes with git reflog, and more.

## Basic usage

The most basic Reflog use case is invoking:

git reflog

This is essentially a short cut that's equivalent to:

git reflog show HEAD

This will output the HEAD reflog. You should see output similar to:

eff544f HEAD@{0}: commit: migrate existing content  
bf871fd HEAD@{1}: commit: Add Git Reflog outline  
9a4491f HEAD@{2}: checkout: moving from main to git\_reflog  
9a4491f HEAD@{3}: checkout: moving from Git\_Config to main  
39b159a HEAD@{4}: commit: expand on git context   
9b3aa71 HEAD@{5}: commit: more color clarification  
f34388b HEAD@{6}: commit: expand on color support   
9962aed HEAD@{7}: commit: a git editor -> the Git editor

Visit the [Rewriting History page](https://www.atlassian.com/git/tutorials/rewriting-history) for another example of common reflog access.

### Reflog references

By default, git reflog will output the reflog of the HEAD ref. HEAD is a symbolic reference to the currently active branch. Reflogs are available for other refs as well. The syntax to access a git ref is name@{qualifier}. In addition to HEAD refs, other branches, tags, remotes, and the Git stash can be referenced as well.

You can get a complete reflog of all refs by executing:

git reflog show --all

To see the reflog for a specific branch pass that branch name to git reflog show

git reflog show otherbranch 9a4491f otherbranch@{0}: commit: seperate articles into branch PRs 35aee4a otherbranch{1}: commit (initial): initial commit add git-init and setting-up-a-repo docs

Executing this example will show a reflog for the otherbranch branch. The following example assumes you have previously stashed some changes using the git stash command.

git reflog stash 0d44de3 stash@{0}: WIP on git\_reflog: c492574 flesh out intro

This will output a reflog for the Git stash. The returned ref pointers can be passed to other Git commands:

git diff stash@{0} otherbranch@{0}

When executed, this example code will display Git diff output comparing the stash@{0} changes against the otherbranch@{0} ref.

### Timed reflogs

Every reflog entry has a timestamp attached to it. These timestamps can be leveraged as the qualifier token of Git ref pointer syntax. This enables filtering Git reflogs by time. The following are some examples of available time qualifiers:

* 1.minute.ago
* 1.hour.ago
* 1.day.ago
* yesterday
* 1.week.ago
* 1.month.ago
* 1.year.ago
* 2011-05-17.09:00:00

Time qualifiers can be combined (e.g. 1.day.2.hours.ago), Additionally plural forms are accepted (e.g. 5.minutes.ago).

Time qualifier refs can be passed to other git commands.

git diff main@{0} main@{1.day.ago}

This example will diff the current main branch against main 1 day ago. This example is very useful if you want to know changes that have occurred within a time frame.

## Subcommands & configuration options

git reflog accepts few addition arguments which are considered subcommands.

### Show - git reflog show

show is implicitly passed by default. For example, the command:

git reflog main@{0}

is equivalent to the command:

git reflog show main@{0}

In addition, git reflog show is an alias for git log -g --abbrev-commit --pretty=oneline. Executing git reflog show will display the log for the passed .

### Expire - git reflog expire

The expire subcommand cleans up old or unreachable reflog entries. The expire subcommand has potential for data loss. This subcommand is not typically used by end users, but used by git internally. Passing a -n or --dry-run option to git reflog expire Will perform a "dry run" which will output which reflog entries are marked to be pruned, but will not actually prune them.

By default, the reflog expiration date is set to 90 days. An expire time can be specified by passing a command line argument --expire=time to git reflog expire or by setting a git configuration name of gc.reflogExpire.

### Delete - git reflog delete

The delete subcommand is self explanatory and will delete a passed in reflog entry. As with expire, delete has potential to lose data and is not commonly invoked by end users.

## Recovering lost commits

Git never really loses anything, even when performing history rewriting operations like rebasing or commit amending. For the next example let's assume that we have made some new changes to our repo. Our git log --pretty=oneline looks like the following:

338fbcb41de10f7f2e54095f5649426cb4bf2458 extended content 1e63ceab309da94256db8fb1f35b1678fb74abd4 bunch of content c49257493a95185997c87e0bc3a9481715270086 flesh out intro eff544f986d270d7f97c77618314a06f024c7916 migrate existing content bf871fd762d8ef2e146d7f0226e81a92f91975ad Add Git Reflog outline 35aee4a4404c42128bee8468a9517418ed0eb3dc initial commit add git-init and setting-up-a-repo docs

We then commit those changes and execute the following:

#make changes to HEAD git commit -am "some WIP changes"

With the addition of the new commit. The log now looks like:

37656e19d4e4f1a9b419f57850c8f1974f871b07 some WIP changes 338fbcb41de10f7f2e54095f5649426cb4bf2458 extended content 1e63ceab309da94256db8fb1f35b1678fb74abd4 bunch of content c49257493a95185997c87e0bc3a9481715270086 flesh out intro eff544f986d270d7f97c77618314a06f024c7916 migrate existing content bf871fd762d8ef2e146d7f0226e81a92f91975ad Add Git Reflog outline 35aee4a4404c42128bee8468a9517418ed0eb3dc initial commit add git-init and setting-up-a-repo docs

At this point we perform an interactive rebase against the main branch by executing...

git rebase -i origin/main

During the rebase we mark commits for squash with the s rebase subcommand. During the rebase, we squash a few commits into the most recent "some WIP changes" commit.

Because we squashed commits the git log output now looks like:

40dhsoi37656e19d4e4f1a9b419f57850ch87dah987698hs some WIP changes 35aee4a4404c42128bee8468a9517418ed0eb3dc initial commit add git-init and setting-up-a-repo docs

If we examine git log at this point it appears that we no longer have the commits that were marked for squashing. What if we want to operate on one of the squashed commits? Maybe to remove its changes from history? This is an opportunity to leverage the reflog.

git reflog 37656e1 HEAD@{0}: rebase -i (finish): returning to refs/heads/git\_reflog 37656e1 HEAD@{1}: rebase -i (start): checkout origin/main 37656e1 HEAD@{2}: commit: some WIP changes

We can see there are reflog entries for the start and finish of the rebase and prior to those is our "some WIP changes" commit. We can pass the reflog ref to git reset and reset to a commit that was before the rebase.

git reset HEAD@{2}

Executing this reset command will move HEAD to the commit where "some WIP changes" was added, essentially restoring the other squashed commits.

## Summary

In this tutorial we discussed the git reflog command. Some key points covered were:

* How to view reflog for specific branches
* How to undo a git rebase using the reflog
* How specify and view time based reflog entries

We briefly mentioned that git reflog can be used with other git commands like [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout), [git reset](https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting), and [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge). Learn more at their respective pages. For additional discussion on refs and the reflog, [learn more here](https://www.atlassian.com/git/tutorials/refs-and-the-reflog).