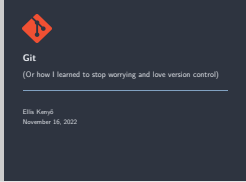
Git

(Or how I learned to stop worrying and love version control)

Ellis Kenyő

November 16, 2022



Notes

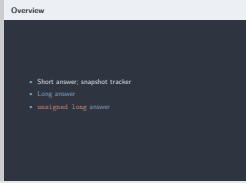
Welcome to "Git (Or how I learned to stop worrying and love version control)"; a talk on what git is, how to use it, how we *should* use it and later how we can interact with the wider community. After this, I hope you will come away with something; and hopefully a thirst for digging deeper into using git. During the talk there will some pre-computed code examples but all of the examples have been run as-is with no external scripts/tools so anything in here should work.

There will be time at the end to ask any questions, just to ensure that I actually finish the workshop.

What is Git?

Overview

- Short answer; snapshot tracker
- Long answer
- `unsigned long` answer



Notes

Git is a distributed version control tool for source code management. What does that mean in English?
In the simplest terms, a version control tool is a tool for handling changes in files by storing the changes in files between generations. git (unlike other existing version control tools) doesn't store diffs/changes, it stores *snapshots* rather then diffs. Think of it like time machine, each new "snapshot" of a file or files is called a commit. We'll dig more into these shortly.

The other links included are a short intro article expanding some basics, and a talk called "Git From the Bits Up"

which demonstrates building a git directory without using git.

Terms

Definition (Commit)
verb: The action of storing a new snapshot

noun: A point in git history; a snapshot of a file or files

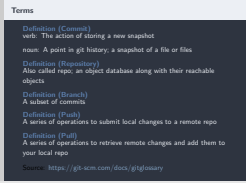
Definition (Repository)
Also called repo; an object database along with their reachable objects

Definition (Branch)
A subset of commits

Definition (Push)
A series of operations to submit local changes to a remote repo

Definition (Pull)
A series of operations to retrieve remote changes and add them to your local repo

Source: https://git-scm.com/docs/gitglossary



Notes

Included here are some terms that occur during the presentation or often enough in general git usage to warrant a definition. Included at the bottom is a link to where I based these from, along with definitions for basically anything you're likely to encounter with git, which I'd recommend at least skimming to get definitions to things you might have come across and not been sure of.

Starting at the top, we have a commit; which is used both as a noun and a verb. The noun sense refers to a single snapshot of a file or files within the history of the current repository. More concretely it refers to a "commit object", which is an object (a hashed file inside the git repo) which stores the commit as well as metadata such as committer, author etc. This will be explored in more detail in the next slide. The verb sense is simply the act of creating a commit.

Then we have a repository or repo; simply put this is just a group of files that also contains a correctly structured `.git` folder containing all the objects (including commits, branch data, etc)

Next we have a "branch", which fits with the idea of everything operating around a tree and represents an independent set of commits. A repo can have any number of branches, the only limit would come down to limitations of the file system in which the repo lies. The most recent commit on a branch is called the tip, and has a reference called "head".

Push/pull are operations performed by their respective CLI invocations and refer to taking changes from your local repo and pushing them to a remote repo; and vice versa.

Commits

Commits

What *is* a commit?

```python
import hashlib, zlib, os
string_to_hash = 'I am a file'
header = "blob " + str(len(string_to_hash)) + "\0"
blob = (header + string_to_hash).encode('utf8')
sha = hashlib.sha1(blob).hexdigest()
git_object = f".git/objects/{sha[:2]}/{sha[2:]}"
os.makedirs(os.path.dirname(git_object), exist_ok=True)
with open(git_object, 'wb') as f:
    f.write(zlib.compress(blob))

return f"Wrote: {git_object}"
```

```
Wrote: .git/objects/1f/5614948c014b5b8284aa0504fdfa770ea01dce
```

```
git cat-file -p '1f56'
```

```
I am a file
```

Source

Notes

I mentioned before what a commit is abstractly; a single snapshot in history. But that doesn't really help to understand what it is.

So I've used a trivial example to demonstrate me creating a new object in the git database for the string 'I am a file'. Skimming line-by-line for those that might not be familiar with Python:

L1: Importing libraries for hashing and general OS tasks

L2: Creating the string to hash

L3-5: Creating the actual object, which looks like "blob <string length> <null terminator> <value of the object>", then SHA1 hashing it

L6: Creating a variable for the file on disk to store in which is `.git/objects` then the first 2 characters of the hash, then everything from the second character onwards (this is done to prevent issues on file systems that don't like thousands of files in a directory)

L7: Create the path on disk (so we can write to the file)

L8-9: Open the file as writable in binary format then write the zlib-compressed hash

L11: Print the filename

And just like that, we have created an object in the database! We can prove that with the second example, which looks up a file in the git database by the hash and returns the contents, which is the original string we setup.

A commit is an object like the one created above, just with extra metadata.

OK, so *how* do I make a commit?

Creating commit

```
rm -rf sample .git/ && git init
echo "Status before: "
git status --short
echo "I am a file" > sample

echo "Status after creating: "
git status --short

echo "Status after adding: "
git add sample
git status --short

echo -e "\n"

git commit -m "Initial commit" --no-gpg-sign
```

```
Initialized empty Git repository in /tmp/test/.git/
Status before:
?? file
Status after creating:
?? file
?? sample
Status after adding:
A  sample
?? file


[master (root-commit) 467b20d] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 sample
```

Git status short format

Printing the object

```
git cat-file -p $(git log -n 1 --pretty=format:%H)
```

```
tree 446377584578f1a54bb90edaa39b144d658c2ba4
author Ellis Kenyő <me@elken.dev> 1668595275 +0000
committer Ellis Kenyő <me@elken.dev> 1668595275 +0000

Initial commit
```

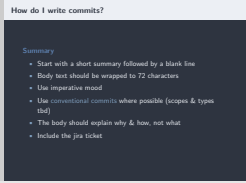

Notes

Obviously that process is quite tedious, but luckily git performs that for us (along with other cool things like redundancy checks) so how do we make a commit otherwise?
Well, we can just use the git CLI to create commits. I'm sure you've all done this before, but just so we're all on the same page let's step through the example above.
First we recreate our environment and run `git status --short` to give us the short version of the status (mostly to just save screen space, feel free to use the normal `git status` when running for yourself)
Then, we create a file with the content "I am a file" and run status again.
What's changed? Now, we see the file we created prefixed with two question marks. These two characters represent first the status of the index then the status of the working tree. For a more detailed explanation, please see the link for "git status short format".
For our purposes though, the question marks simply mean the file is untracked. Any changes we make on it are not part of the git object database.
After we use `git add` on it, we now have a single A on the left side and nothing on the other (the gap is important). This means the file has been added, but not yet committed. Any changes are not persisted to the git object database.
After that, we commit the change and get the object from the database to inspect it. As you can see, it's the same as the object we created before just with extra metadata.

How do I write commits?

Summary

- Start with a short summary followed by a blank line
- Body text should be wrapped to 72 characters
- Use imperative mood
- Use conventional commits where possible (scopes & types tbd)
- The body should explain why & how, not what
- Include the jira ticket



Notes

Now that we know *what* a commit is, we can come up to a much higher level and talk about writing commits. There are countless pages, books, articles and blogspam around these and there is no overall "best" way, the following are merely guidelines.
I will go into more detail on the following in subsequent slides.

None of these are ironclad yet, just guidelines. There are tools that we can use to help automate linting commits, so if such a system were to go live then these rules *would* be enforced. At such a point, there will be a document on the academy documenting the rules in detail.

How do I write commits?

Start with a short summary followed by a blank line

- No more than 50 chars, 72 is the hard limit (anything higher might break some tools)
- The blank line helps tools understand what's the subject and what's the body
- Short summaries help other devs to quickly parse commits when using `git log`

Notes

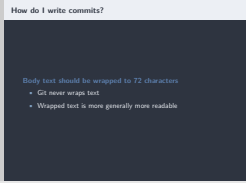
This is probably the most enforced rule, bordering on even being a guideline. It enforces a consistency that helps scan through commits when looking for a particular commit, and it gives a useful summary for people reviewing. Longer summaries are also trimmed by github.

The blank line is also important to help tools distinguish between what the subject is and what the body is.

How do I write commits?

Body text should be wrapped to 72 characters

- Git never wraps text
- Wrapped text is more generally more readable



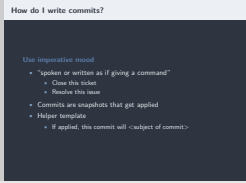
Notes

Coming from a simpler time when programs were entered onto 80-hole punchcards, the standard just sort of stuck. The 50 char body and 72 header limit also benefits when sending patches as emails. This one is quite loose, though it does improve readability and everyone should be encouraged to follow it.

How do I write commits?

Use imperative mood

- "spoken or written as if giving a command"
  - Close this ticket
  - Resolve this issue
- Commits are snapshots that get applied
- Helper template
  - If applied, this commit will <subject of commit>



Notes

This even comes from git itself, when creating a merge commit the generated message is "Merge branch 'branch-name"' and when creating a revert commit, the generated message is "Revert <subject> This reverts commit <hash>". A commit is a snapshot that is intended to be applied, so writing messages in the correct tone ensures we're sticking to convention and in general improves readability.

How do I write commits?

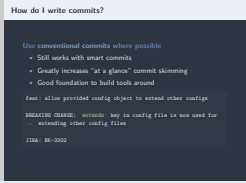
Use conventional commits where possible

- Still works with smart commits
- Greatly increases “at a glance” commit skimming
- Good foundation to build tools around

```
feat: allow provided config object to extend other configs

BREAKING CHANGE: `extends` key in config file is now used for
↪  extending other config files

JIRA: BK-3302
```

Notes

Conventional commits is a specification that dictates a rigid structure for writing commit messages by including what are referred to as types and (optionally) scopes. They match quite well with SemVer, allowing for automations to for example only pull in “fix” and “feat” commits in changelogs.
In the context of SemVer:

- a `fix` type would translate to a `PATCH` release
- a `feat` type would translate to a `MINOR` release
- A commit with `BREAKING CHANGE` in the body/footer would constitute a `MAJOR` release regardless of the type

In the example layout, you can see the basic structure. First comes a type, which can be one of `fix` for a bugfix or hotfix, `feat` for a feature, `build` for anything that affects the project’s build system or dependencies, `ci` for anything that changes CI files or scripts, `perf` for code that just improves performance, `refactor` for code that just refactors existing code, `style` for commits that resolve style issues without changing code meaning (though these should ideally be taken care of by editor tools or ci) and `test` for commits that only adjust tests. These types aren’t set in stone, and they’re not rigid; only the structure is rigid.
For example, the above commit is tagged as a feature but could also have easily been marked as a `fix` or even introduce a new `config` type, though do try and stick to the types we define.
The optional scope would be a noun describing a section of the project which helps narrow down further and would need to be defined per-project for things like `fix` and `feat`, eg one project might have scopes like “automation” or “nme”, another might have “beneficiaries” or “cards”, if you make a `build` change then scopes might be `npm` or `composer`. They are optional and should only be included when they make sense to be. `style`, `test` and `refactor` are good examples of types that would span across many sections so don’t make sense to include a scope.

How do I write commits?

The body should explain why & how, not what

- Good code is *generally* self-documenting
- When that fails, comments exist
- Having to trawl through git logs to understand a change is frustrating, as is having to read through a massive commit

Notes

Good, clean code is generally self-documenting assuming the reader has been properly acclimated to the language. So in general commits shouldn't be for explaining what some code is doing, but provide context that might not belong in a comment along with why this change has been made.

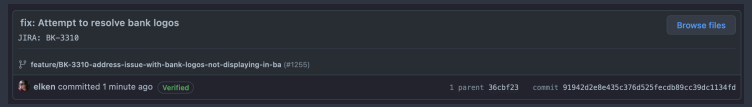
For our use case, this should be properly handled through jira; but having to trawl through jira tickets and potentially lengthy discussions isn't much better than trawling through git logs.

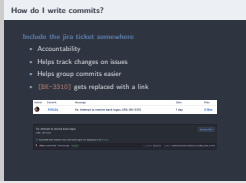
How do I write commits?

Include the jira ticket somewhere

- Accountability
- Helps track changes on issues
- Helps group commits easier
- `[BK-3310]` gets replaced with a link

Author	Commit	Message	Date	Files
	91942d	fix: Attempt to resolve bank logos JIRA: BK-3310	1 day	3 files





Notes

Touched on briefly in the last slide, including the ticket number in the commit message (ideally the footer as not to waste valuable summary space) will link it to the ticket in jira. This also applies to pull requests, and creating a pull request with the ticket ID in square brackets will cause a bot to edit your PR and insert a link to the ticket.

When do I commit?

- As often as makes sense
- Avoid massive commits
- Try and keep a commit for a single feature/bugfix

When do I commit?
- As often as makes sense
- Avoid massive commits
- Try and keep a commit for a single feature/bugfix

Notes

Frequency of commits is another thing to be aware of, committing as often as makes sense will make your life easier so you don't have to undo hundreds of times and will make later reverts much simpler. A good rule of thumb is trying to keep a single logical unit of work to a commit in such a way that if this commit were to be removed, it shouldn't require other commits first.

The *only* exception here is large linting commits, or a gigantic refactor; but in the case of the latter there's probably room to reduce the scope of that ticket.

Branches

Branches

What are they?

- A subset of commits
- A movable pointer
- More info

Notes

So we've gone through commits now, the other entity you should be familiar with is branches. Expanding on the previous example, each commit is an object that creates a tree linking it with the previous commit. As you add commits, the pointer for the branch moves forward.
When you make a new branch from a commit or other ref, you create another pointer from which commits can be added to. These branches don't have to be related, and don't even have to contain the same files.

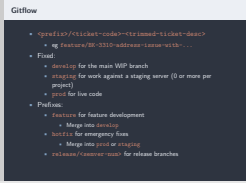
For a deeper dive, see the link on the slide.

Conventions

Conventions

Gitflow

- <prefix>/<ticket-code>-<trimmed-ticket-desc>
  - eg feature/BK-3310-address-issue-with-...
- Fixed:
  - develop for the main WIP branch
  - staging for work against a staging server (0 or more per project)
  - prod for live code
- Prefixes:
  - feature for feature development
    - Merge into develop
  - hotfix for emergency fixes
    - Merge into prod or staging
  - release/<semver-num> for release branches



Notes

Gitflow conventions have already been touched on during the last retro, but just to remind everyone/fill in those who missed it; gitflow is a branch naming model.
In terms of creating branches, there are a number of fixed branches; those being

- develop for the main WIP branch
- staging for work against a staging server (of which there could be 0, 1 or many depending on the project)
- prod for live code (akin to master or main as it is now)

In the future, these will ideally be protected in github; meaning the only way to get changes in would be via pull requests (which already require approval from one reviewer).
When dealing with everything else, there is a standard format at the top of the slide (which should be taken care of by the vscode extension that has mentioned before).

feature for branches that revolve around feature work, and hotfix for branches that fix critical issues.

Forks

- `origin` for your fork, `upstream` for the original repo
  - By default pushing goes to your fork
  - Harder to accidentally send commits to the wrong place
  - This is what `gh` CLI does
- Alternatively `origin` for the original and `<gh-username>` for your fork
  - Handles multiple forks
  - Easier to accidentally send commits to the wrong place

Forks
- origin for your fork, upstream for the original repo
  - By default pushing goes to your fork
  - Harder to accidentally send commits to the wrong place
  - This is what gh CLI does
- Alternatively origin for the original and <gh-username> for your fork
  - Handles multiple forks
  - Easier to accidentally send commits to the wrong place

Notes

Because of how we primarily use git and github, it's rare we have to handle forks but it's worth covering still.
A fork is simply a copy of a remote repository under a different user. This includes the full history and branches, but doesn't include site-specific things like issues or pull requests.
There are two main ways to handle remote naming with forks, `origin` for your forked version and `upstream` for the original version; and `origin` for the *original* version and the associated github/gitlab/bitbucket owner name (not username because repos can be forked to organisations too).
The first way is the recommended and sensible approach, it's rare you'd be working with multiple forks in the same tree and by default pushes/pulls go to/from your remote repo so there's little chance of pushing changes to the wrong place. This is also how the github CLI tool `gh` does it, so by using that you're already set.

The other approach is used in a few tools, which typically wrap around git to reduce the chances of pushing to the wrong place; but it could still happen. At the end of the day, neither is strictly better, but I would recommend the first approach.

Upstream

- Provide a MWE (Minimal Working Example)
- Provide as much info as is pertinent
- Check for contributor docs
- Search for issues first to prevent duplication

Upstream
- Provide a MWE (Minimal Working Example)
- Provide as much info as is pertinent
- Check for contributor docs
- Search for issues first to prevent duplication

Notes

Briefly touched on in the previous slide, there are some things to take into consideration when dealing with upstream. At the end of the day, a fair percentage of people contributing code are working during their free time so the more you can help them to help you the better.

A good first step is being able to provide a MWE or minimal working example to reproduce the bug. Create a repo if required, but usually just a snippet is enough. Including as much info as you can feeds into this, stack traces, OS name/version, package versions, etc. Whatever is relevant to your issue.

Nowadays if the repo is setup correctly, when making an issue for the first time github will prompt you advising you to read through contributor docs first. This is usually found at the top level but can also exist in a `docs` subfolder. This may include code conventions, style guide, how to format commits or PRs, how to reproduce issues, etc.

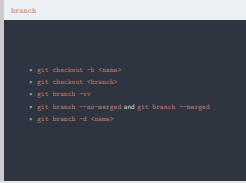
Lastly, make sure you put at least some effort in trying to find pre-existing issues or related issues. This can save everyone time and creating a new issue when one or even more exist is a good way to get an annoyed response.

Usage

Usage

branch

- `git checkout -b <name>`
- `git checkout <branch>`
- `git branch -vv`
- `git branch --no-merged` and `git branch --merged`
- `git branch -d <name>`



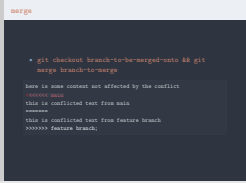
Notes

The next few slides will include a couple of relevant commands for the most common operations. First up is branches, and simply we have “create a branch and set it to be the current branch”, “checkout another branch”, “list all branches with more information including which remote branch we’re tracking”, “show me all the branches that have been merged into the default branch” and vice versa and finally “delete a branch”.
As with all these slides, you’re encouraged to read through the manual pages to see what else is possible.

merge

- `git checkout branch-to-be-merged-onto && git merge branch-to-merge`

```
here is some content not affected by the conflict
<<<<<<< main
this is conflicted text from main
=======
this is conflicted text from feature branch
>>>>>>> feature branch;
```

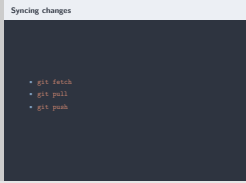
Notes

Next up is merge, and the most common usage. Most git clients have ways to do this easier, along with handling merge conflicts.
A merge conflict is simply just a result of two snapshots affecting the same section and git is unable to decide what should be applied. This is usually a result of not keeping your working branch up-to-date often enough.

Often they have to be resolved manually, but most git clients include some way to resolve these easier.

Syncing changes

- `git fetch`
- `git pull`
- `git push`



Notes

Syncing changes mostly revolves around 3 commands; fetch, push and pull.
Fetch will update the current branch (or all refs with `--all`) by downloading all objects and refs from a remote repo. Note that this won't apply any changes, only the objects.
Related to fetch, we have pull which *will* apply any changes that exist for the currently tracked branch and can also cause merge conflicts.

Last we have push, which will attempt to sync your local changes with changes on a remote repo, if your history is

different to what's remote (eg you rebased or you haven't pulled recently) then you'll get an error from the server.

stash

- `git stash`
- `git stash pop`
- `git stash apply`
- `git stash list`
- `git stash (pop|apply) <identifier>`
- `git stash -p`

stash
- git stash
- git stash pop
- git stash apply
- git stash list
- git stash (pop|apply) <identifier>
- git stash -p

Notes

git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.
So in order we have:

- "Stash all changes in my current tree"
- "Apply the most recent stash and remove it"
- "Apply the most recent stash and *don't* remove it"
- "Show me all the stashes in my tree"
- "Pop or apply a specific stash based on the identifier (the first column)"
- "Interactively decide what to stash"

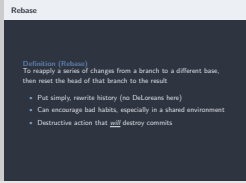
Advanced Git

Advanced Git

Rebase

Definition (Rebase)
To reapply a series of changes from a branch to a different base, then reset the head of that branch to the result

- Put simply, rewrite history (no DeLoreans here)
- Can encourage bad habits, especially in a shared environment
- Destructive action that *will* destroy commits



Notes

OK so we know what commits are now, we know how to write them, what if we make a mistake? I'll just make another commit, right?
Well, as with most guidelines the answer is "it depends". If it's just a small thing, you'd probably be better off doing what's called a rebase.
Rebasing is a fancy word with a fancy definition that simply just means writing history. This can be anything from rewording a commit, combining multiple commits together (what's referred to as "squashing"), adjusting the order of commits and just flat out editing commits.
Using the bare git CLI to rebase can be quite tricky, but tools that wrap around git often include a decent rebase interface. Regardless, the upcoming demo will just use the git CLI.

Rebase Demo

```
echo "file" > file
git add file
git commit -m "Added file"
```

```
[master 952846e] Added file
 1 file changed, 1 insertion(+)
 create mode 100644 file
```

```
git log
```

```
commit 952846eda636dc7e6d174fa36b4254aaffdfbb85
Author: Ellis Kenyő <me@elken.dev>
Date:   Wed Nov 16 10:41:15 2022 +0000

    Added file

commit 467b20d4505285307743aa462e213d594851d6c9
Author: Ellis Kenyő <me@elken.dev>
Date:   Wed Nov 16 10:41:15 2022 +0000

    Initial commit
```


Notes

The following demo will show how you can use git rebase to reword a commit. The process for more complex interactions like changing commits is quite complex and you're encouraged to either use one of the tools shown later or learn for yourself.

Building on the same repo we made earlier, we only have our initial commit. We can't rebase on more than 1 commit

(because there's nothing to rebase *onto*) so we have to just create another commit to let us play with. As before,

we're just making a dummy file and committing it.

Rebase Demo

Pre Rebase

```
git rebase -i HEAD-1
```

```
pick 0ccabad Added file

# Rebase eda7dd4..0ccabad onto eda7dd4 (1 command)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified); use -c <commit> to reword the commit message
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

```
Added file by rewriting history_

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Wed May 4 08:34:37 2022 +0100
#
# interactive rebase in progress; onto 96f1f79
# Last command done (1 command done):
#    reword 049c42d Added file
# No commands remaining.
# You are currently editing a commit while rebasing branch 'master' on '96f1f79'.
#
# Changes to be committed:
#       new file:   file
#
# Untracked files:
#       farewell.txt
#       greeting.txt
#       new_file
#
```

Post Rebase

```
git log
```

```
commit 949678607213dea4fefc7bf8632d4a8c648c9293
Author: Ellis Kenyő <me@elken.dev>
Date:   Wed May 4 08:37:53 2022 +0100

    Added file by rewriting history

commit ceab98bd5da79400982df77f6b0120cbacf77fc5
Author: Ellis Kenyő <me@elken.dev>
Date:   Wed May 4 08:37:53 2022 +0100

    Initial commit
```


Notes

Now the process begins, we invoke the command above, which translates to "`git rebase` interactively between the most recent commit and the last 1 commit". In order to perform a rebase, you have to select the range of commits that are to be affected. This can be "the last 1 commit" or two specific commit hashes.
In this case though, we're just capturing the last 1 commit.
In the first screenshot, a file has been created and opened in whatever the environment variable `GIT_EDITOR` is set to, in this case `vim`. At the top there is "pick" followed by some of the commit hash followed by the message. In the blurb below, all the possible commands are explained.
If we look there, we can see that pick just means "use this commit", which means that nothing about it changes and if we were to save this file now, no changes would be made.
As it turns out, we want to change the wording in the commit, so we need to change "pick" to "reword" and save this file.
After doing that, the bottom screenshot occurs as a new commit message is ready to be written. As you can see in the commented section underneath, this is occurring during an interactive rebase.

After then changing the message and saving this file, the rebase is completed and a fresh `git log` shows us the new history.

Rebasing Do's and Don'ts

- DO use it sparingly as it's a destructive action
- DON'T use it to amend commits that are far in the past that have been merged
  - In that case, a `revert` is preferred as it preserves the history while still removing the code
- DO be aware that rewriting already pushed commits will require you to force push
- DO use it sparingly as it's a destructive action

Notes

Some guidelines for good rebase usage, do use it sparingly as it's a destructive actions;

- DO use it sparingly as it's a destructive action

Reiterating this again because it will wipe your history, and if you haven't synced it remotely it's gone forever.

- DON'T use it to amend commits that are far in the past that have been merged. In that case, a `revert` is preferred as it preserves the history while still removing the code

Self-explanatory, rewriting history is dangerous so using it on anything already pushed is a bad idea. To undo a commit or commits use `git revert` or whatever your git client of choice has in that area

- DO be aware that rewriting already pushed commits will require you to force push

If you are working on a branch that has been synced, and you rebase something, your history and the remote history are now out of sync; so trying to push will result in an error. If this is what you are sure you want to do, the only resolution is to force push which will tell the remote repository you want your changes to take precedence.
This can result in lost work *remotely* and leads on to the last point

- DO use it sparingly as it's a destructive action

Rebasing and force pushing should only be used for small touch-ups unless you're comfortable with git. If you need to rebase a large PR that has a lot of scattered commits for example, don't be afraid to ask someone senior for help.

Closing

Closing

Clients

- Gitkraken (GUI)
- Sourcetree (GUI)
- GitHub Desktop (GUI)
- lazygit (Terminal)
- GitLens (VSCode)

Clients
- Gitkraken (GUI)
- Sourcetree (GUI)
- GitHub Desktop (GUI)
- lazygit (Terminal)
- GitLens (VSCode)

Notes

Nearly there now, I've included some recommendations for git clients in no particular order. I've used them all in the past and they're all good, though Gitkraken is what I would consider the most beginner friendly and it also has integrations for jira tickets (GitLens might too and sourcetree "should") though it has paid options and I can't comment on how much requires a license now.

Try them all and see what you like more.

Thanks for listening!

Any questions?