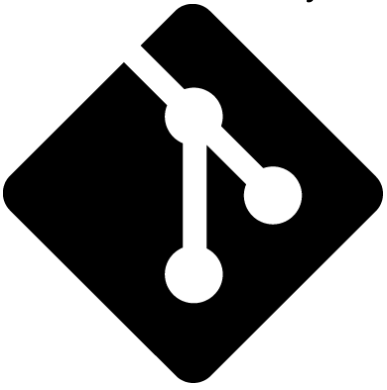
Git for Graduates

Joe Fuller-Gray



2024

Contents

About

A key part of professional software development is collaborating with others. The most common way to do this is with version control software, the presently dominant one being `git`.

Many universities don't teach `git` to the degree required for professional work. This piece aims to provide a simple guide to using `git` for working with other developers, complete with a simple workflow.

This document assumes some faculty with the command line and is aimed towards developers using a Linux distribution or MacOS for their work. It is intended to take someone who can push a history to GitHub and turn them into someone who can contribute meaningful changes to a repository.

This is not intended to be an exhaustive guide to Git. For a more in-depth manual, read "Pro Git" by Scott Chacon, available at https://git-scm.com/book/en/v2.

All Git commands are valid as of `git` version 2.45.2.

What Git is

> Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals.
>
> Description from https://git-scm.com/docs/git.html

Git is version control software. It is designed to track versions of human readable text, although if it's a file, it can be tracked by Git.

In 2023's Stack Overflow survey, it was noted that 94% of all software developers use Git. The alternative to using Git seems to be not using version control.

Why you should use it

Version control software is a vital tool because it allows multiple developers to share a history of a particular project.

Let's suppose you don't use it, and you want to collaborate with someone else on a document you have written. You make your changes, and they make theirs. Then you send them to each other. How do you know who made which changes? How do you know which version is the definitive original document? How do

you share your reasoning with the other person?

Version control software such as Git solves this problem by requiring both users to mark their changes and send them to a central repository so that both users can see the history of the project.

Where you can get it from

The most common way to get it is to download it through your operating system's package manager.

Linux

Linux distribution	Command
Debian / Ubuntu	`apt install git`
Fedora	`yum install git`
Arch Linux	`pacman -S git`

MacOS

MacOS package manager	Command
Homebrew	`brew install git`
MacPorts	`port install git`

Windows

On Windows, you can download an installer from https://git-scm.com/downloads, which also includes some GUI tools for using Git.

How to configure it

On first run, it will prompt you to set your name and your email address.

In the terminal, run the following:

```
git config --global user.name "<Your-name>"
git config --global user.email "<Your-email>"
```

This tells Git to use your name and your email when you write to the history.

This is all you need right now. Customisation can come later once you're more familiar with the software.

How to use it

Initialising a repository

Inside your folder, you can initialise a Git repository by simply running `git init`.

You will need to make a first commit. Common practice is to simply make a barebones README and commit

that.

```
echo "$YOUR_PROJECT_NAME" > README.md
git init
git add README.md
git commit -m "init: Add README.md"
```

Terminology

HEAD refers to the topmost commit on a currently checked-out branch.

A **commit** is a change, consisting of a diff and a commit message.

Checking out refers to a change in repository state.

A **diff** is a text representation of the difference between two files. Git also treats a change on the same file as a diff.

A **branch** refers to a history of the repository forked from a common ancestor from another branch.

The original branch is usually called **main**. Git doesn't require the original branch to be called `main`. It has been called `master` before, and it's sometimes referred to as `trunk`, from previous version control systems. For the sake of consistency, refer to it as `main`.

Making changes

Once you have your repository, you can begin to make your changes. Standard practice is to only make changes to human-readable files. Binary files or generated files should not be committed to the Git history, because they can be quite large and they're harder to visualise when browsing the Git history.

You can check what changes are being made to track files by viewing the diff.

```
git diff
```

Now that you have your changes, you can add them to the staging area then make your commit.

```
git add <file>
git commit
```

If you want to add files in chunks, you can use `git add -p` to patch-add changes.

If you want to add everything that currently is not being tracked, you can use `git add .`, although it is suggested that you don't do this because it's difficult to track what's actually going in there.

Removing a file is easy: `git rm` removes the file and adds its removal to the staging area. This is preferred over a simple `rm` followed by a `git add <removed-file>` for simplicity's sake.

Moving a file is similar: `git mv` moves one file to another location and adds that change to the staging area, and is preferred over `mv` followed by `git add` for similar reasons.

Commit editors

Without prior configuration, most Linux distributions set their `$EDITOR` environment variable to `nano`.

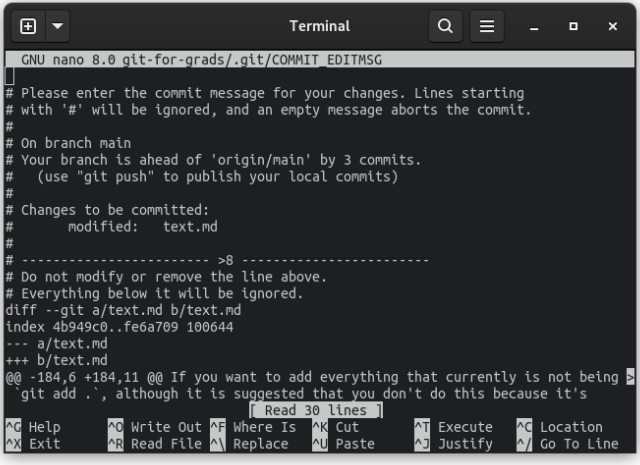


Figure 1: GNU Nano as a commit editor

If you want to use another editor, set the editor in your Git config:

```
git config --global core.editor "<editor>"
```

Anatomy of a commit

A commit looks like this:

```
commit df8653aa
Author: joefg <joefg@example.com>
Date:   Fri Apr 26 16:05:12 2024 +0100

    init: Add .gitignore

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..a136337
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1 @@
+*.pdf
```

The commit hash (or SHA) is the unique reference for a commit. It is derived from the content of the commit plus the date plus the author (and a few more things than that, but that is past the scope of this simple gude). Changing these things changes the commit hash. The general rule here is that if you change the commit hash, you change the commit. Avoid changing already-pushed commits!

The author and email comes from your Git configuration.

The commit message is what the developer describes the commit as.

The diff is the textual representation of the changes to the repository in the commit.

To view a single commit, use `git show`.

The staging area

`git add` adds changes to the staging area. Git is not an atomic version controller: the developer ultimately dictates which changes enter a commit.

`git reset HEAD <file>` removes that file from the staging area but keeps its changes in the file. To remove everything from the staging area, `git reset HEAD` or just `git reset`.

Ignoring files

The `.gitignore` file is used to define what files and what sorts of files should not go into the commit history.

```
file # Ignore all files called file

*.exe # Ignore all files ending in `.exe`

build/* # Ignore everything in `build/`
```

```
build/ # Ignore directories called `build`
```

A good list of sample ones can be found on https://github.com/github/gitignore.

It's advantageous to keep a folder structure in version control, even if you don't want to keep the files within a specific folder in there.

For that reason, it is recommended to keep a `.gitignore` inside each folder and handling ignored files on a per-folder basis. If you just want the folder, ignore everything in that folder, but keep the gitignore in there.

Viewing history

Viewing a repository's history is simple. Just use `git log`.

By default this shows a list of commits on the current branch. To see all commits in the repository, use `git log --all`.

For a more compact view with just the header of each commit, use `git log --oneline`.

There are GUI tools available to view a history in a more intuitive fashion: `gitk`, which ships with Git, is one such tool.

Searching

To finding a string inside tracked files, use `git grep`. Some common flags to use with a `git grep` include:

```
# Case insensitive
git grep -i "<string>"

# Returns line on which the string was found
git grep -n "<string>"
```

Merging and Rebasing

There are two ways to introduce changes made on other branches into your branch: through a **merge** or a **rebase**.

Merge

A **merge** adds a single commit to the current branch containing all of the changes made onto another branch. Using our branch and main as an example, we want to introduce changes made on our branch into main.

This new commit is a merge commit. It doesn't contain any of the commits made on the feature branch, but it does contain the diff and a commit message.

Because this doesn't introduce any additional commits or rewrite history, this is preferred when making

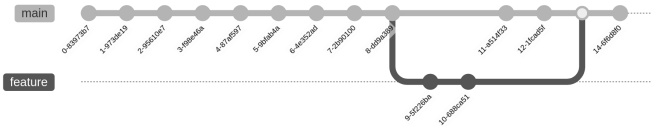


Figure 2: Merging feature into main

changes that affect `main`.

Rebase

A **rebase** changes the base of the current branch to the head of the branch you're rebasing onto.

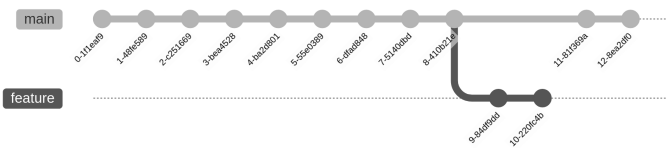


Figure 3: Main and feature

A rebase replays commits made from your branch on top of the other branch, meaning conflicts appear in your commits, rather than in the other branch's commits.

This does change the commits that you put on top of the other branch. It does not introduce merge commits, which makes tracking which commits were introduced

Figure 4: Main and feature after rebase

in which changes difficult. It is for this reason that rebasing is preferred on your own branches.

Rule of thumb

A good rule of thumb is **merge globally, rebase locally**.

If you've pushed your changes and other people are using the branch, merge.

If your branch is something that only you and at most a few other people are using, rebase.

Make sure everyone who uses your branch is aware of this before rebasing.

Fetching changes

Fetching changes on your branch from the remote source is done with a `git pull`. By default, this works as a fetch from the remote to `origin/main` followed by a merge of `origin/main` into your `main`. This can

be split up into two commands: `git fetch` and `git merge origin/main`.

Instead of a fetch merge, a user can fetch rebase: `git fetch && git rebase`. Doing this means that merge conflicts have to be resolved at rebase time, not after the merge.

It is possible to use the rebase behaviour in a `git pull` by adding this to your configuration:

```
git config --global pull.rebase true
```

Merge conflicts

When the introduction of one commit makes changes that another commit will make, you see a **merge conflict**, in the form of this appearing in a file:

```
<<<<<<< HEAD
change
=======
conflicting change
>>>>>>> branch
```

This is accompanied by an error message which looks like this:

```
$ git rebase main
Auto-merging README.md
CONFLICT (content): Merge conflict in <file>
error: could not apply 73cc672... <commit>
```

At this point you have to decide which change you wish to keep going forward. You do this by removing the change you don't want, then using `git add <file>`, then continuing with the merge or rebase.

Conflict resolution tools

In addition to manually editing a conflicting file, there are tools which provide a cleaner conflict resolution mechanism.

Here, we call the file from the current branch `LOCAL`; the common ancestor between two branches `BASE`; the file that you wish to merge `REMOTE`; and the final result `MERGED`.

VSCode has a tool to do this: it can edit the merge conflict by prompting the user to select which chunk to keep, or it can show the conflict on the file side-by-side and then prompt the user to select which chunk to keep.

Vim (and its derivative Neovim) has `vimdiff` which can be used as a mergetool, which puts `LOCAL` on the top left, `BASE` on the top centre, `REMOTE` on the top right, with `MERGED` on the bottom.

Regardless of which one you use, you can configure Git to use the right one to use by setting the `diff.tool` variable in your config.

```
git config --global diff.tool "<diff-tool>"
```

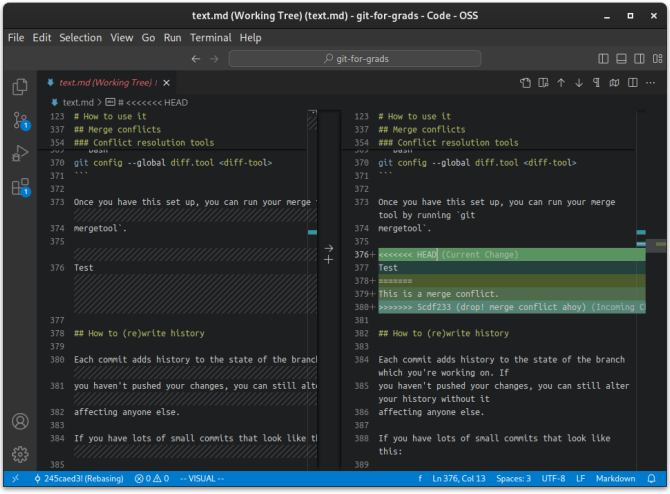


Figure 5: VSCode Mergetool

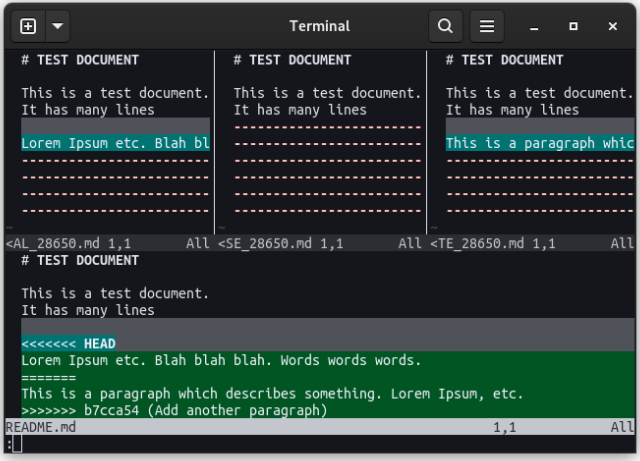


Figure 6: Vimdiff Mergetool

Once you have this set up, you can run your merge tool by running `git mergetool`.

How to (re)write history

Each commit adds history to the state of the branch which you're working on. If you haven't pushed your changes, you can still alter your history without it affecting anyone else.

If you have lots of small commits that look like this:

```
<HEAD> #05 - styling
       #04 - make tests work
       #03 - remove broken stuff
       #02 - add more stuff
       #01 - add stuff
       #00 - draft: new widget
<main> #-1 - Merge branch feature/whatever
```

It's poor form to push these to a main branch, so you should squash these into a single commit. You can rewrite history by resetting the pointer of the current branch's state to where the first commit was.

```
git reset --soft #-1
git commit
```

`git reset --soft #-1` takes the history back to after commit #-1, but keeps all of the changes in the staging area.

You can do the same thing by referring to `HEAD~7`, which could be read as `HEAD - 7`, or 7 commits back from `HEAD`. `HEAD~N` is shorthand for `N` commits back from `HEAD`.

You can also refer to branches. When you refer to the branch, you are referring to the top-most commit on that branch, so you could achieve the same effect by using `git reset --soft origin/main`, which resets to the common ancestor of `origin/main`.

After this, you can `git commit` and write a commit message.

If you haven't pushed already, `git push` would work because nobody else has the old history.

If you have pushed already, use `git push --force-with-lease`. A `--force` makes the Git host replace the old history with the new history, and anyone else pulling that branch would have their history replaced with this history. To avoid cases where one user is pulling and another is pushing, a `--force-with-lease` makes the Git host check that there is only one user pushing or pulling at a time.

Rebasing

Interactive Rebasing

Another way to do this is witih an interactive rebase. The `rebase` command changes the history between `HEAD` and the given commit.

An interactive rebase shows you which commits are being moved when the rebase happens. To do this, use `git rebase -i <commit>`, and you will be presented with a menu that looks like this:

```
pick <commit> <commit-header>

# Rebase <start-commit>..<end-commit>
# onto <new-base>
#
# Commands:
# p, pick <commit>
# r, reword <commit>
# f, fixup <commit>
# s, squash <commit>
# e, edit <commit>
# d, drop <commit>
```

A `pick` uses that commit as-is.

A `reword` allows the developer to reword the commit without changing its contents.

A `fixup` allows the developer to merge that commit

into the commit above it without modifying its commit message.

A `squash` allows the developer to merge that commit into the commit above, appending its commit message into the commit above it (or editing the combined commit message).

An `edit` stops the rebase at that commit, allowing the developer to add or alter the staging area, before using `git rebase --continue` to resume the rebase.

A `drop` removes the commit from the history.

What a commit should contain

Each commit should represent the known *good state* of the repository: the functionality should be working if introduced and nothing should be broken if removed. **Each commit should pass all automated testing**.

Commits should be as small as possible: it should introduce one feature at a time. A big commit which introduces more than one feature is harder for a reviewer to understand and the change would appear to be larger than it actually is.

Writing a good commit message

The commit message will be attached to your change, so you need to make sure the following:

1. It makes sense.

Someone in the future might want to see why you made a change, and one of the first things they'll do is find the commit in which a change was made. It takes seconds to write a simple "makes thing do x" without an explaination, but it could take hours to find exactly why. Sticking the explaination in saves the future developer potentially hours of work.

2. It is written in an imperative tone in the present tense.

You should view a commit message as what applying the changes in the commit message will do, rather than what it has done. This takes some getting used to. The reason for this is that git itself uses the imperative style in the present tense for merging and rebasing (which will be discussed later), so it makes sense to also write in this style.

A good way to tell if your message fits this style is to add "This commit, when applied, will" before the message. If it makes sense after this, it is in the imperative tone in the present tense.

3. It includes some reference to the thing it's trying to solve.

Your commit might form part of work inside a work item, or a ticket. Putting a reference to that work item inside your commit allows a future developer to see

which work item the commit was part of so they can get more context as to why a change was made.

Broadly, it should follow:

> Title: Summary, imperative, start uppercase, no full stop (no more than 50 characters)
>
> Body: Explain what the change does and why it was needed. Keep it to 72 characters per line.
>
> You can add more paragraphs if you want after a blank line.
>
> - Bullet points are good.
> - Blank lines between each bullet point
>
> Part-of: ticket-id

Bad examples

```
commit df8653a (HEAD -> main)
Author: joefg <joefg@example.com>
Date: Fri Apr 26 16:05:12 2024 +0100

I fixed this button so it works
```

This is a particularly poor commit message. It's in the wrong tense, so it assumes that the commit has done something already rather than stating what it will do.

“This button” is too vague. “It works” is too vague and flippant. The “I” is ultimately not important. There’s no description. It would be rejected in the change request.

```
commit df8653a (HEAD -> main)
Author: joefg <joefg@example.com>
Date: Fri Apr 26 16:05:12 2024 +0100

Fixes bug with the widget button handler
that prevented it from firing,
updated test to match.
```

This is still partly in the past tense (“prevented”), and it’s all wedged into the header of the commit. There’s no reference to a work item either. Like above, this would be rejected in the change request.

A good example

```
commit df8653a (HEAD -> main)
Author: joefg <joefg@example.com>
Date: Fri Apr 26 16:05:12 2024 +0100

Fixes widget button handler bug

The widget button's handler would not fire
the event if the overall widget was still
loading. The fix is to remove the lock on
widget load so that the button handler can
```

```
cancel the widget loading and re-send it
with the new state of the button.

Amends tests in:

- Widget test

- Multiple widget test

- Weather widget test

Part-of ticket/100
```

This is better. The title is succint so the future developer can see where a bug was fixed while browsing quickly through the commit history. If the future developer wants to see why the fix went in, he can read the body. If he wants to see the ticket in which the issue was raised, he can refer to the ticket in the part-of at the bottom.

This is a good commit message, and would likely pass a review.

Pushing your changes

Once your have made your changes, you can push your branch.

If you don't already have one set up, you can create your repository on the Git host of your choice, then

run the following in the shell.

```
git remote add origin <repo-url>
```

Once you have done this, you can push it with `git push`.

How to work with other developers

This is fine for when you're the only developer working on a project, but what about when other developers show up?

For this, you'll need to change your workflow. The most basic workflow is Trunk-Based Development, which uses `main` as the current state of work prior to release with branches coming off that for features.

Figure 7: Trunk-Based Development

Branching

Imagine two developers pushing to the `main` branch at the same time. They will both run into problems

because the one who pushed first blocks the person who pushes second.

It is very rare in industry to push straight to main. You tend to work on branches instead.

To create a new branch from main and use it, do the following:

```
# Make sure main is up to date
git fetch main    # Fetch main's updates
git checkout main # Go to main
git rebase

git checkout -b your-branch
```

Now you're on that branch, you can make your commits all you like without affecting main.

Remember to rebase main often so that any merge conflicts are minimal when it comes time to raise the change request.

Once you're done, you can push that branch, and your Git repository host will be able to see that branch, and many offer the ability to raise a change request for it following a template.

Sending and applying patches

It is possible to send the output of a `git diff` to someone and for them to apply it. Doing this allows a de-

veloper to send a patch to another developer without commiting it.

```
# Generate our patch
git diff > diff.patch

# Apply it
git apply diff.patch
```

This is particularly helpful when discussing a small change on a change request.

Cherry picking

It is possible to add a commit from another branch to the current branch with the `cherry-pick` command.

```
git cherry-pick <commit-sha>
```

This is particularly useful when bringing a fix from a private branch into the current branch.

Change requests

Some hosts call these “merge requests”, some call them “pull requests”. The term “change request” works better because not all of these will use the `git merge` or `git pull` commands depending on workflow.

Once you create a Change Request, it needs a few things.

1. A summary. Expand on what your commit messages were. Why would the reviewer want this change? Does it meet the acceptance criteria? Include screenshots for any UX work.
2. Test instructions. How should a reviewer test these changes? Include any test files, any test configurations, and detailed instructions. If it's a bug, include steps to reproduce it on `main`, and those same steps should work on the branch without that bug occurring.
3. Caveats. Are there any compromises with this change request? Does there need to be any follow-on work?
4. Deploy instructions. Will there be any further work required to get this change into production? Does this require any migration steps?

Once you have your change request, ask for a review from a colleague.

Reviewing change requests

Suppose you're now reviewing a colleague's change request. You should approach it like this:

1. Ask: does the summary make sense? What is it that you're testing? If it's unclear, ask for a clarification in the comments.

2. Are the test instructions clear? You should be able to follow them from your existing computer. If it requires additional setup, this should be clear in the instructions. If they're not, ask for a clarification. Then follow the test instructions. If it fails, mark it on the change request. Keep going until you've exhausted the instructions.

3. Caveats. Are they acceptable? Does this introduce any unwanted precedent in the development process? Should this caveat be documented elsewhere?

4. Deploy instructions. Has the developer missed anything out?

Add your comments to the change request. Unless it's a simple one-line fix, change requests almost never go in first time round. The nature of software means someone always misses something. This is OK. This is just part of the process. If it was easy, there wouldn't be this process.

Fixes in your change request

Your change request has been reviewed, and the reviewer asks for some fixups.

One approach to use for this is to use the `--fixup` and `--squash` arguments to `git commit`. Ideally, it should be one commit per point raised in the change request.

The `git commit --fixup` command works like this:

```
# Make your change and
# add it to staging
git add <file>

# Make a fixup commit by taking the
# commit hash and using that here.
git commit --fixup <commit>
```

If your commit was this:

```
commit 00000 (branch)

This is a test commit
```

`git commit --fixup 00000` creates a commit that looks like this:

```
commit 11111 (branch)

fixup! This is a test commit
```

The `git commit --squash` command works the same way, except you can add a commit message body that gets put into the commit that the commit is being squashed into.

Once you have made your fixups, push them, and see if they address the points raised. **Avoid force-pushing at this stage.** Force-pushing changes the state on the remote, messing with the diff and making

it less clear which commits are fixes and which ones aren't.

If you get the all clear, you can then run the following to automatically squash your history.

```
git rebase -i --autosquash main
```

This opens an interactive rebase menu which should have already arranged your `fixup!` commits in order. Once you have rebased, you should be back down to the number of commits you already had when going into the review first time round.

Tagging

Git's `tag` functionality provides a useful way to mark a specific point in history with a human-readable string of text.

In Trunk-Based Development, this is used to mark the current state of production, and is set during release. In many projects this is done by a project maintainer, usually through the repository host's UI.

You can show tags by running `git tag`; you can tag a commit by running `git tag -a <tag-name> <commit>`, and you can delete a tag by running `git tag -d <tag-name>`.

Sometimes you need to push a tag to test an on-tag pipeline or similar. Make sure this isn't confused with

a new version number by using the following structure: `<ticket-number>-<your-initials>-<tag-number>`.

The "accepted" versioning system, commonly found in open-source projects, is the Semantic Versioning standard. Broadly speaking, this is:

`<MAJOR>.<MINOR>.<PATCH>`

where a Major version is for breaking changes, Minor versions are for backward-compatible changes, and Patch versions are for backward-compatible bug fixes.

A good history

The whole point of using Git in this way is to maintain a log of which *features* went into a particular project, at which dates, as part of which work orders.

Keeping a good history allows both experienced developers and newbies to see the evolution of the project. In theory, it should be possible for someone to check out to the very first commit and step their way through the history to see how a product was built.

Keeping a good history from day one will pay dividends in the long term. Don't accept anything less than a good development log.

Extras

Bisection

A `git bisect` allows a developer to perform a binary search on a branch to find a commit which introduced or removed a specific behaviour between two commits.

You can do this by running the following:

```
git bisect start
git bisect bad HEAD
git bisect good <commit>
```

This starts at the midpoint between the good commit (earlier in the history) and the bad commit (later in the history). If it's "good", the developer can mark it with `git bisect good` and the bisect moves to the middle between that midpoint and the bad commit, and so on. This takes what would be an `O(N)` bug-finding mission into an `O(logN)` mission.

This can be streamlined further with `git bisect run`, which allows a script to determine whether the run was "good" or "bad".

Semantic branches

Many organisations have a branch naming structure. This should be clear to you when onboarding. The standard the author uses is:

Type	Branch name
Feature	`feature/<branch-name>`
Bug Fix	`fix/<branch-name>`
Private	`private/<name>/<branch-name>`

A `private` branch isn't truly private – once you push, it's visible to all with access to the repo to see.

Other ones I've seen include putting your username in the branch name. This is an open-source trick. It depends on the organisation and project. For open-source projects, there's usually some guidance in the documentation on this matter.

Semantic commits

Sometimes, commit messages headers follow a pattern.

The author tends to use the following:

Type	Header
Feature	`feat: <header>` or `feature: <header>`
Bug fix	`fix: <header>`
Chore	`chore: <header>`
Tech	`tech: <header>`

Like above, it depends entirely on the project and organisation.

Conventional commits

Conventional Commits is another standard for structuring commits, with the benefit of being able to use committizen to enforce the Conventional Commit style.

Broadly speaking, Conventional Commits follows this pattern:

```
<type>[scope][breaking]: <description>

[body]

[footer(s)]
```

Where the **type** is either **fix** for patches or **feat** for features, with the optional scope being the thing that's being modified, a breaking change being represented with an `!` after that, followed by a terse description (all of which should fit on one line).

There are variations on this discussed on the Conventional Commits page. It is also perfectly acceptable to follow the Angular Convention here.

Done properly, using Conventional Commits enables the use of a Conventional Changelog, which save time when building release notes.

Linting commit messages

As your team gets larger, it may help to introduce some linting to the commit messages so that all commits adhere to a house style. One such tool to do this is *commitlint*. Add this to your CI and ensure that it has been run with every push to origin.

Aliases

Git offers the ability to alias certain commands to shorthand.

One is an `unstage` command which removes a file from the staging area:

```
git config --global \
    alias.unstage 'reset HEAD --'
```

This allows you to use `git unstage` in place of `git reset HEAD -- <file>`.

Reflog

The reflog is a way of viewing the history of wherever your HEAD was pointing to. It tracks where your HEAD was at various stages.

For example: suppose a developer checkout to a previous commit (a detached HEAD), then you run `git`

`reflog`: it will show you where your HEAD was before, so you can check out that reflog to re-attach the head.

It may look like this:

```
98513ee HEAD@{0}: commit: I broke the thing
e12553d HEAD@{1}: commit: Thing works
bb046eb HEAD@{2}: commit: Add another thing
a21b4af HEAD@{3}: commit: Add thing
```

If you want to reset the repository to a point prior to HEAD, you can run `git reset --hard HEAD{1}`. The usual caveats apply: you will lose those commits.

Local configuration

It is possible to set per-repository configurations. Using `git config --local` sets a configuration value in the repository's own `.gitconfig` file, found in `.git/.gitconfig`.

Git looks for a configuration in the following order:

1. System-wide: usually in `/etc/gitconfig`, or the equivalent on your operating system.
2. User-wide: in `~/.gitconfig`.
3. Repository: in `.git/.gitconfig`.

Hooks

Git allows you to set automations to run either locally or remotely when a specific action has been triggered. Locally, these include:

- `pre-commit`: This is run prior to making a commit. This can be used to trigger a run of a linter or a type checker to ensure code quality at a local level.
- `commit-msg`: This is used to generate a commit message.

On a server, a `pre-receive` hook can be used to automatically run a script to check for any credentials that may have ended up committed. Ensuring code quality is the work for CI, not a hook.

You can find examples of these hooks in `.git/hooks`. They tend to be implemented as shell scripts.

Porcelains

Git isn't just a command line tool. Your favourite text editor probably either has support for it, or a plugin is available for it which supports it.

- Microsoft's Visual Studio Code has support for it built-in, including support for GitHub.
- Emacs has a plugin called `Magit` which provides

an interface for interacting with a Git branch in Emacs.

- Vim has `Vimagit`, which does above but for Vim.
- gitg and gitk offer a GUI over Git.
- Sublime Merge is a commercial offering which offers a graphical interface to Git.

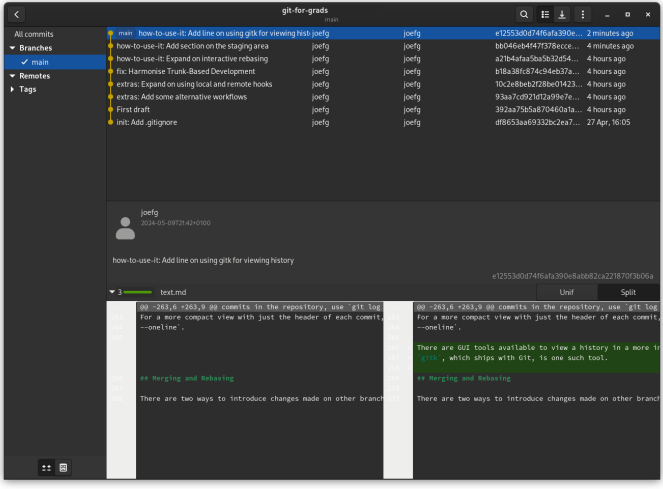


Figure 8: gitg

Alternative workflows

Trunk-Based Development isn't the only way to use Git.

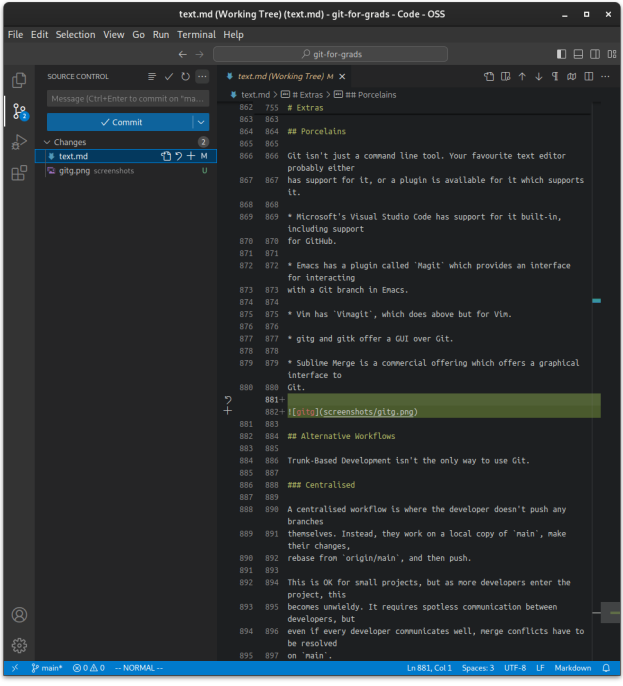


Figure 9: VSCode

Centralised

A centralised workflow is where the developer doesn't push any branches themselves. Instead, they work on a local copy of `main`, make their changes, rebase from `origin/main`, and then push.

This is OK for small projects, but as more developers enter the project, this becomes unwieldy. It requires spotless communication between developers, but even if every developer communicates well, merge conflicts have to be resolved on `main`.

Short-Lived Feature Branching

This is similar to Trunk-Based Development except feature branches last no longer than a few hours. This is done to prevent branches from becoming stale, and it keeps feature size to a minimum.

Personal Branching

Instead of a branch per feature, each developer has their own branch. The benefit is that there are fewer branches to manage, and the developer is solely responsible for managing their history. Trunk-Based Development can use this too: having a long-running personal branch is helpful for research and development tasks, but the developer must take care to rebase

their branch often to resolve merge conflicts before they become big.

Forking

This requires the use of a Git host's Fork feature. Each developer makes a repository-level fork of the project and pushes their changes to that, with the developer of the project's main repository integrating changes from that fork when required.

This is more complex than the others. It's most often used in open-source software development of complex projects.