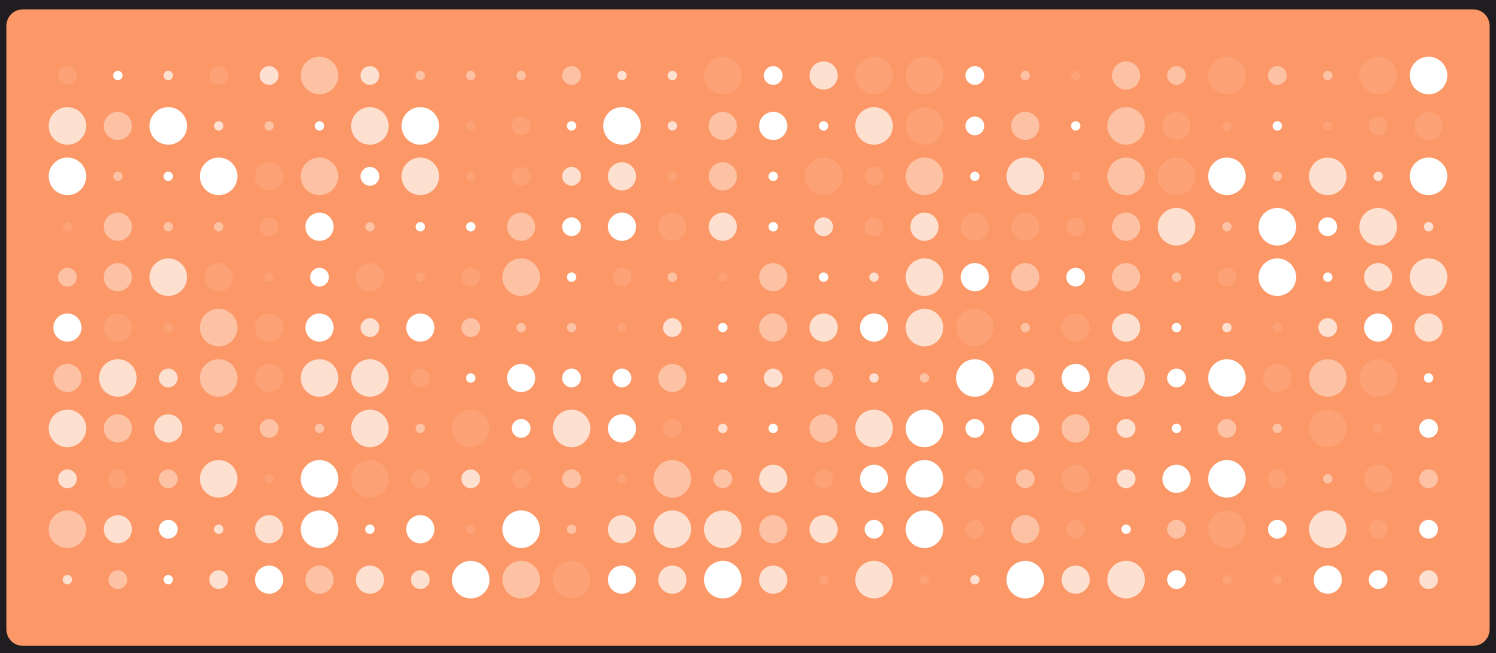

Elliott Phillips

Data Scientist
BSc OR II

Git Good

A comprehensive guide to Git versioning

Last updated: October 20, 2021

ABOUT

1.1 Why Git?

Is it worth learning Git?

TL;DR – yes, it definitely is.

Understanding how to use Git and an interface, such as GitHub, effectively is one of the most important skills for the modern developer. This book aims to equip you with the fundamentals of Git with hands-on example. By following along, you'll learn to use Git to manage your software projects and collaborate with other developers.

1.1.1 What is Git?

Git considers a repo's data as a series of snapshots. Each commit, Git captures the current state of your filesystem and stores a reference to that snapshot. For efficiency, Git links any unchanged files to the previous identical file it has already stored.

1.1.2 Integrity

Everything in Git is checksummed, using 40-character SHA-1 hash calculated based on the contents of a file or directory structure, before it is stored and is then referred to by that checksum.

```
8eb4fc5d221ff6b741afc209f14008e31adbe431
```

This functionality is built into Git at the lowest levels and is integral to its philosophy: you can't change the contents of any file or directory and hence lose information in transit or get file corruption without Git being able to detect it.

1.2 Getting started

Git, created by Linus Torvalds in 2005, is a system for keeping track of changes that happen across a set of files. To initialise a new Git repository, open a directory on your local system with an editor like VS Code then run `git init` from the command line. Git repos live in the hidden `.git` directory and keep track of all the changes that happen to files as you work on a codebase.

Commits are a method of taking snapshots of the current state of your files, and each commit has its own unique ID and is linked to its parent, allowing us to travel back in time to a previous version of our files.

The `HEAD` represents the most recent commit. If we make some changes and commit them to the repo, `HEAD` moves forward but we still have a reference to our previous commits so we can always go back to it.

Where Git really shines is enabling managed collaborative working. Software typically isn't developed linearly – you may have multiple teams working on different features for the same codebase simultaneously. Git makes that possible by branching. Create a branch by running `git branch` and then run `git checkout` to move into that branch. You can now safely work on your feature in this branch without affecting the code or files in the `master` branch.

Commits made in these branches live in an alternate universe with its own unique history. Eventually, you'll likely want to merge a branch's history with the `master`'s history. Run `git merge` on your feature branch, whose 'tip' now becomes the `HEAD` of the `master` branch, or, in other words, our fragmented universe has become one.

2.1 The perfect commit

2.1.1 Committing files

Some of the first commands you learn with Git will be to initialise and track your working directory locally.

```
$ git add .
$ git commit -m "Your message"
```

To quicken your time on the command line, you can skip the `git add` step using the `-am` flag with `git commit` to automatically add all the files in the current working directory:

```
$ git commit -am "Your message"
```

If you've committed with a spelling error, update the latest commit message with

```
$ git commit --amend -m "Updated message"
```

or if you've forgotten to add a file to your latest commit:

```
$ git add extra_file.txt
$ git commit --amend --no-edit
```

where the `--no-edit` flag retains the original commit message.

2.1.2 Stash

If you have changes that almost work but they can't really be committed yet because they break everything else or aren't up to par with the code style, or you just don't want your colleagues to see

it yet, `git stash` will remove the changes from your working directory and save them for later use without committing them to the repo

```
$ git stash
$ git pop
```

You can assign the stash a name to reference it later with `git pop` when you're ready to add those changes back into your code. Use `git stash list` to view all aliased stashes, find the stash you can to retrieve followed by `git stash apply` with the corresponding index to use it

```
$ git stash save coolstuff
$ git stash list
$ git stash apply 0
```

2.1.3 Destroy things

Imagine you have a remote repository on GitHub and a local version on your machine you've been making changes to, but things haven't been going too well and you just want to go back to the original state on the remote repo.

First do a `git fetch` to grab the latest code in the remote repo, then use `reset` with the `--hard` flag to overwrite your local code with the remote code, but be careful, your local changes will be lost forever.

```
$ git fetch origin
$ git reset --hard origin/master
```

But you might still be left with some random untracked files and build artefacts here or there. Use the `git clean` command to remove those files as well

```
$ git clean -df
```

2.1.4 Checkout

If you recently switch out of a branch and forgot its name, you can use `git checkout` followed by a dash to go directory back into the previously branch you were working on

```
$ git checkout -
```

2.1.5 Switch

If you find yourself working on a branch and need to move your current untracked changes to another (new or existing) branch, then `git switch` is the perfect use case for when you just want to test something, but you're not sure it's worth it.

```
$ git switch -c new-branch
```

You can achieve the same effect with `git checkout -b new-branch`, but `checkout` does more than just switching branches, so `switch` was created for specificity.

BRANCHES

3.1 Branches in a nutshell

Nearly every Version Control System (VCS) has some form of branching support. To "branch" means to create a new alternative development line within your project, diverging from the `master`, or main, timeline, allowing you to work on features without interacting with that main line.

Git's branching model is what sets it apart from other VCS tools – it's incredibly lightweight, making branching operations nearly instantaneous and encouraging workflows that branch and merge often.

To really understand the way Git does branching, we need to recall how Git stores its data: not as a series of changesets or differences, but instead as a series of *snapshots*.

Per commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged, the author's name and email address, the message that you typed, and pointers to the parent commits that directly came before this commit.

3.2 A successful Git branching model

The model I demonstrate here, named *git-flow*, originates from 2010 – not very long after Git itself came into being. It has become popular in its adoption and development in many software teams, perhaps even considered standard protocol. This model is geared more towards continuous delivery and integration, such as scheduled data releases, web development etc. where explicit versioning or multiple version support is required.

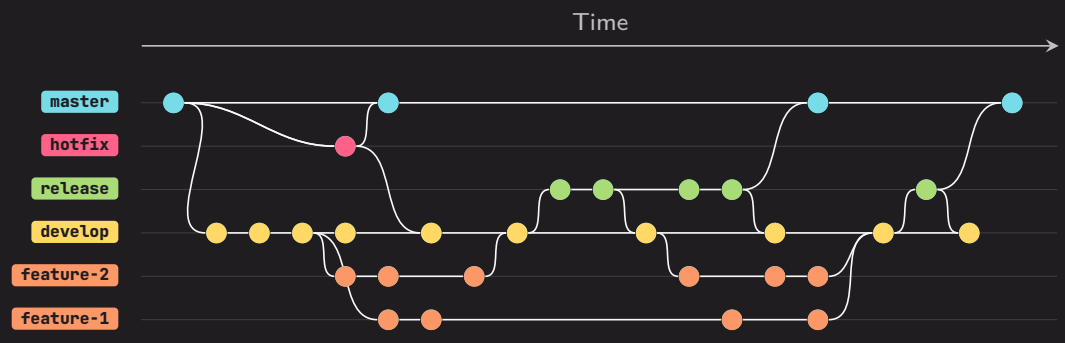

Figure 3.1: *Demonstration of the industry-standard* git-flow *branching model.*

3.2.1 The main branches

The central repo holds two main branches with an infinite lifetime, `master` and `develop`, where

`develop` runs parallel to the `master` branch. By definition, consider these two branches for source code of `HEAD` as:

- `origin/master` always reflects a production-ready state,
- `origin/develop` always reflects a state with the latest delivered development changes for the next release, enabling automatic production/CI builds overnight.

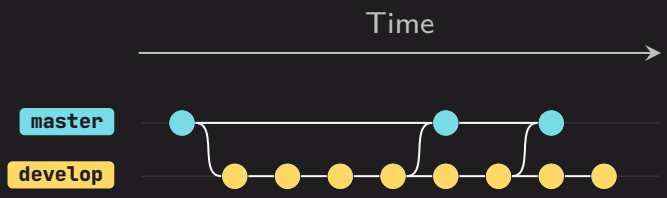

Figure 3.2: *Main branches* `master` *and* `develop` *of the* git-flow *branching model.*

When stable and release-ready, `develop` branch source code is merged back into `master` and tagged with a release number, generating a new production release *by definition*. Keeping this strict enables use of Git hooks automatically build and roll-out software to our production servers per commit to `master`.

3.2.2 Supporting branches

This development model's supporting branches lay adjacent to the main branches `master` and `develop`, enabling;

- Parallel development between team members,
- Ease of feature-tracking,
- Production release preparation,
- Effective fixing of live production problems.

These branches are `feature`, `release`, and `hotfix`.

Unlike the main branches, these branches are designed to have finite lifetime, and each have a specific purpose and are bound to strict rules to accept particular branches as origin or target destination on merging.

Remember, all Git branches are technically the same – even `master` is just an ordinary branch, created by default and most people opt to keep it – but instead these branch types are categorized by how we use them.

3.2.3 Feature branches

Feature branches are used to develop new features for the upcoming or a distant future release. They exist as long as the feature is in development and must either eventually be merged back into `develop`, or discarded.

Typically, feature branches exist in developer repos only, not in `origin`.

When starting work on a new feature, branch off from the `develop` branch.

```
$ git checkout -b new-feature develop
```

and when finished, features may be merged into `develop` for inclusion in the upcoming release:

```
$ git checkout develop
$ git merge --no-ff new-feature
$ git branch -d new-feature
$ git push origin develop
```

Where the `--no-ff flag` ensures the merge creates a new commit object, even if the merge could be performed with a fast-forward. This prevents information loss about the historical existence of a feature branch and groups together all commits that together added the feature.

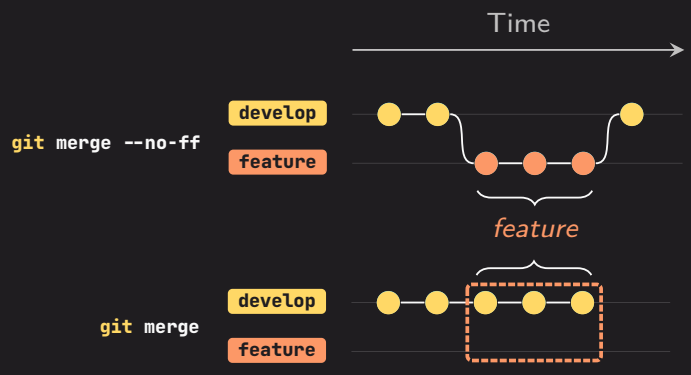

Figure 3.3: *Demonstration of fast-forwarding* `feature` *branch merges onto* `develop` *branches.*

Whilst `--no-ff` creates a few more (empty) commit objects, it enables:

- Visibility of commit objects together have implemented a feature in the Git history, eliminating laborious log messages traversal,
- Reverting a whole feature (i.e. a group of commits), with insight and reason from the Git history.

3.2.4 Release branches

Release branches support preparation of a new production release facilitating last-minute corrections and preparing meta-data for a release (version number, build dates, etc.) ready for handover to `develop`.

When `develop` reflects the desired state of the new release, create a `release` branch from `develop` that includes all completed features, ie. No partial features. It is exactly at the start of a release branch that the upcoming release gets assigned a version number – not any earlier.

```
$ git checkout -b release-1.2 develop
$ ./bump-version.sh 1.2
$ git commit -am "Bumped version number to 1.2"
```

After creating a new branch and switching to it, we bump the version number. This new branch may exist there for a while, until the release may be rolled out. Once all bug fixes have been applied in this branch, the bumped version number is committed. Adding large new features here is strictly prohibited. They **must** be merged into `develop`, and therefore, wait for the next big release.

When the state of release is ready to become a real release:

1. Merge onto `master`, since every commit on `master` is a new release by definition,
2. Then `tag` that commit on `master` for easy future reference to this historical version,
3. Merge the changes made on the `release` branch onto `develop` to carry these bug fixes into future releases.

```
$ git checkout develop
$ git merge --no-ff release-1.2
```

Completing these steps will most likely give rise to a merge conflict since the version number has been updated. Fix this and commit to finalise the release process. The `release` branch may now be removed.

```
$ git branch -d release-1.2
```

3.2.5 Hotfix branches

You can consider `hotfix` as critical `release` branches: preparation spaces for new production releases with urgency to act on a bug(s) of a live production version. When a bug is identified for immediate resolve, spawn a new `hotfix` branch from the corresponding tag on `master` that marks the production version.

This allows the development team, who operate on the `develop` branch and below, to maintain momentum whilst an assignee prepares the production hotfix.

If the current production release, say version 1.2, is live with bug that requires immediate attention, a `hotfix` branch may be created from `master`.

```
$ git checkout -b hotfix-1.2.1 master
$ ./bump-version.sh 1.2.1
$ git commit -a -m "Bumped version number to 1.2.1"
```

Ensure you bump the version number after branching off! Then, fix the bug and commit the fix in one or more separate commits.

```
$ git commit -m "Fixed severe production problem"
```

Similar to completing the `release` branch process, once the critical bug has been fixed, `hotfix` needs to be merged back both into `master` and `develop` in order to safeguard that the bugfix is included in the next release.

The key difference between the two patterns is, *when a* `release` *branch currently exists, the hotfix changes need to be merged into that* `release` *branch, instead of* `develop`.

Back-merging the bugfix into the release branch will eventually result in the bugfix being merged into `develop` too, when the release branch is finished. If work in `develop` immediately requires this bugfix and cannot wait for the release branch to be finished, you may safely merge the bugfix into `develop` now already as well.

The steps follow:

1. Update master and tag the release,

   ```
   $ git checkout master
   $ git merge --no-ff hotfix-1.2.1
   $ git tag -a 1.2.1
   ```

2. Include the bugfix in `develop`,

```
$ git checkout develop
$ git merge --no-ff hotfix-1.2.1
```

3. Remove the temporary branch.

```
$ git branch -d hotfix-1.2.1
```