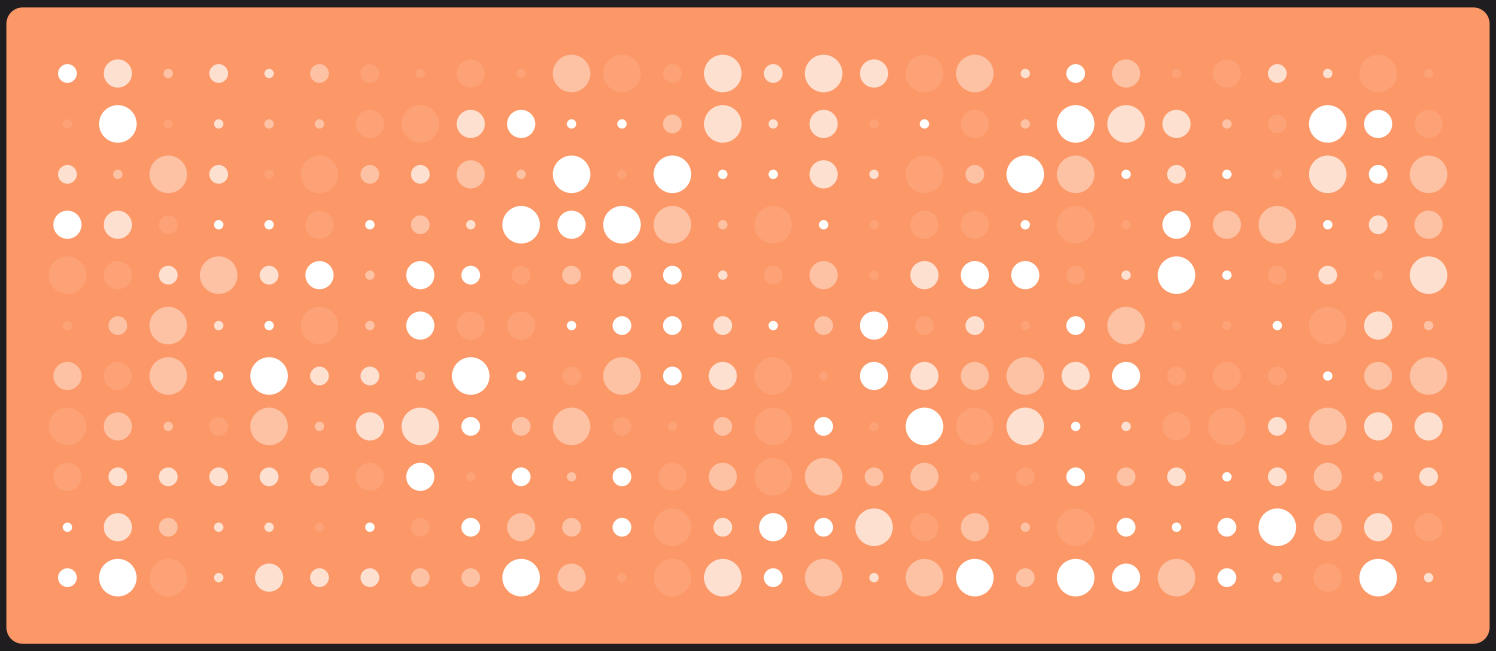

Elliott Phillips

Data Scientist
BSc OR II

Git It?

A comprehensive guide to Git versioning

Last updated: October 19, 2021

ABOUT

1.1 Why Git?

Is it worth learning Git?

TL;DR – yes, it definitely is.

Understanding how to use Git and 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, the 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

The following commands help you quicken and amend your commits prior to pushing to a remote repository

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

You can skip the git add using the -am flag to automatically add all the files in the cwd

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

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

```
$ git commit --amend -m
```

3.1 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.

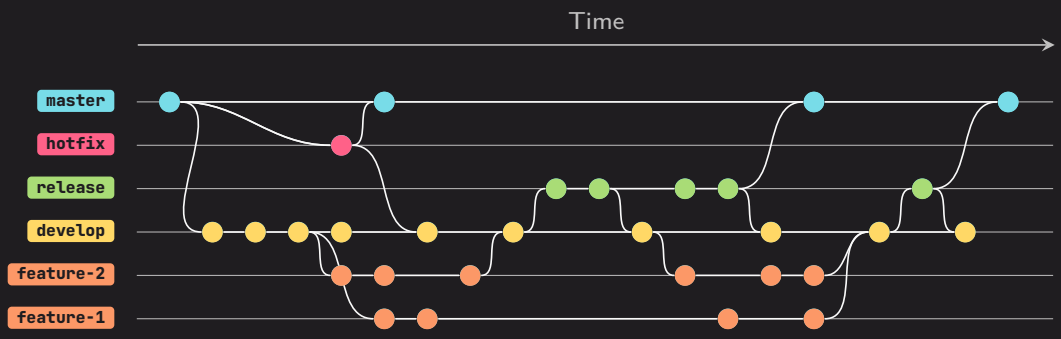

Figure 3.1: *Demonstration of the industry-standard* git-flow *branching model.*

3.1.1 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.

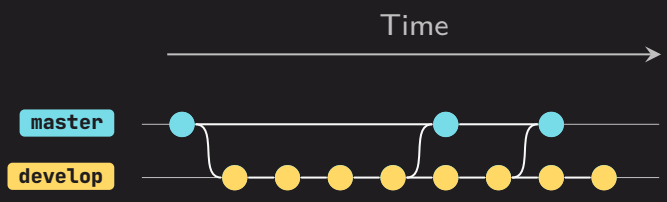

Figure 3.2: *Main branches* `master` *and* `develop` *of the* git-flow *branching model.*

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`.

3.1.2 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.

3.1.3 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 myfeature
$ git branch -d myfeature
$ 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.

`--no-ff demo`

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

4.1 Example

Demonstrative materials.