Git Guide

Grant Holtes

March 25, 2024

1 Introduction

Git is a tool for collaborative software development. This guide helps technical understand its key features and help teams work with git in a consistent manner.

2 Getting Started

2.1 Account Creation

To get started with Git using GitHub as the remote repository provider, sign up at `https://github.com` by clicking "Sign Up".

2.2 Creating a Repository

Create a new project space (repository) by clicking the "+" icon on GitHub and selecting "New repository".

2.3 Cloning a Repository

To work on a project locally, clone the repository using `git clone <repository_url>`.

3 Fundamental Git Operations

3.1 Adding Files

Add files to the repository directory and stage them with `git add`.

3.2 Committing Changes

Commit changes with `git commit -m "message"`.

3.3 Pushing Changes

Share changes with `git push origin master`.

4 Publishing an existing codebase

If you have existing code you want to publish to github, you can follow the above steps to create a new repository, but rather than cloning the empty repository, you can follow the steps below to push your existing code to the new remote repository.

1. Create the (hidden) files that git needs using `git init`. This command needs to be run in the command line / terminal in the root folder of the codebase you wish to publish.
2. `Add` and `commit` your code with `git add .` and `git commit -m "Init repo"`. These steps could also be done with a git GUI.
3. Create a new (empty) repository in github
4. Set the URL of the remote repository that you have created by running the following in the command line: `git remote add origin <remote repository URL>`
5. `Push` your changes to the remote

5 Collaboration with Git

5.1 Branches

A branch is set of changes (commits) on top of a given set of past changes. Create branches for new features or fixes using `git checkout -b <branch_name>`.

5.2 Merging

Merge branches with `git merge <branch_name>`.

5.3 Rebasing to Avoid Conflicts

Rebasing is recommended before raising a pull request. Use `git rebase <branch_name>` to apply your changes on top of the latest changes from the main branch, maintaining a cleaner commit history and reducing the likelihood of conflicts when merging branches

5.4 Pull Requests

Use pull requests (PRs) to propose changes. On GitHub, select "New pull request" and choose the branch you want to merge. PRs allow team members to review changes before merging.

6 Suggested collaboration workflow

Putting the above operations together we can define a workflow that allows for concurrent collaboration on a repository while minimising conflicts[1].

These steps start with a repository with a `main` branch that is taken as the current correct version of the codebase, and should be done for each standalone change that is made. In an ideal case you should be working through these steps multiple times per day, which allows for others to incorporate your changes quickly, reducing time spent resolving conflicts.

1. `Fetch` the latest changes from the remote repository
2. `Checkout` the latest commit on the `main` branch
3. Create a new branch with a (preferably descriptive) name that matches the feature / change you are going to make. for example `FIX-making-the-button-green` and checkout this branch. You should now see that this branch is a continuation from the latest commit on `main`.
4. Make your changes
5. `Add` and `commit` your changes to your feature branch. Give your commit a meaningful name, as the branch will be deleted later, so this commit name is how you will find your changes later, if you need to.
6. Fetch changes again. If anyone else has made changes to `main`, you should rebase your branch onto the latest commit to main and double check that your code still runs given the changes that others have made[2]. This may result in the rebase exposing some conflicts, which will need to be resolved - See conflict resolution section.
7. `Push` your changes, which will require creating a new remote branch.
8. Create a `Pull request` using the github interface, to merge your new branch into `main`.
9. Once reviewed and approved, merge your changes into `main`. You can use the `Rebase and merge` method in github, but give you have already rebased, this shouldn't be any different to a vanilla `merge` or the `Squash and merge` methods.

[1] For the technically minded, this aims to ensure a linear git history - searching for this terminology will provide more detail

[2] If you have made multiple commits, you should be combining (`squashing`) your commits into single commit before moving ahead with the pull request, but new users can find this overly complex so this isn't called out explicitly here. If you have rebased but have multiple commits, the `Squash and merge` merge method on github will do this for you

7 Conflict resolution

A conflict is created when git cannot unambiguously determine which version of a line of a file should be used. This is most commonly the case when there have been two different changes to the same line or section of a file between the two commits being merged. This can be intimidating, so its worth remembering that our job during conflict resolution is simply to tell git which version we want to keep. Git GUIs often provide a conflict resolution workflow, prompting you to select which version of each line or file you want to keep.

7.1 Multiple commits, rebasing and conflicts

Rebasing multiple commits is a common cause of conflicts, and these must be resolved (often repetitively) for every commit in the rebase[3]. The effort can be massively reduced and the conflicts often avoided by simply combining the commits into a single commit before rebasing, an operation referred to as `squashing`.

8 Other suggestions

8.1 Git GUIs

A git GUI is any tool that provides a graphical interface to trigger git operations. My preferred tool is Fork[4], which has both free and paid versions.

8.2 Command line git

Knowing some basic command line git commands and being comfortable in interacting with git in this way is a useful skill for any user that works with remote machines, where the use of a GUI may not be possible.

8.3 Dev, Test and Production branches

For systems where the `main` branch is built and run as a system, rather than simply being a filestore, it is recommended to have multiple branches that are built and run, so that changes can be tested fully before being integrated into the production version of the system. A typical system will have a `dev` branch where feature branches are frequently merged in. Once a tranche of features have been completed, the `dev` branch is merged into the `test` branch for final testing, then the `main` branch to be made available to the users. This flow of changes through the branches is referred to as "promotion".

[3]The reason for this is that the chain of commits are recorded as changes from the previous commit, so each conflict that is resolved creates a new conflcit in the next commit

[4]Download from https://git-fork.com/

8.4 Branch restrictions

The ability to merge changes into branches is usually restricted where there are many developers. Some typical restrictions include:

1. Require reviews and approval to merge any branch into `dev, test or main` branches
2. Only allowing `dev` to merge into `test`, and only allowing `test` to merge into `main`, to ensure that changes are promoted correctly.

8.5 Environment specifications

Just having the code shared in a repository is one part of collaboration, being able to run the code is another matter. Having a method to allow users to setup an identical environment is crucial. Each language offers their own selection of methods to provide this, with some listed below for Python specifically.

- Python venv allows for different packages and package versions to be installed for each project, and requirements.txt files allows for the specific versions of each package to be specified as a file in the repository. This requires the same major version of python to be installed and used by all users.
- Anaconda environment.yaml files allow for a similar behavior as Python venv and requirements.txt, but I dislike anaconda and avoid this method.
- Docker allows for the specific build of Python and the underlying OS to be specified, as well as packages and versions. This is underselling Docker massively, but at a high level it completely fixes the "but it worked on my machine" problem, at the expense of added complexity and a steep learning curve[5].

8.6 The .gitignore file

The gitignore file is a simple text file with the name `.gitignore` which declares files and folders that should not be scanned or processed by git. For example, adding a line with `*.xlsx` will ignore all excel files in the folder structure.

It is used to avoid committing and uploading inappropriate files, such as environments and packages, data files, and binary files. It is usually best to find an example file for the language you are developing in which should cover most usual cases, then adding extra items as required.

[5] With the notable exception of chip architecture differences...

8.7 The README.md file

A file called `README.md` in the root of the repository is the standard way to include high level documentation, and commonly includes a high level summary of the code, instructions on how to set up the environment, and use the code. This is written in `Markdown`, which allows the plain text to be rendered in a slightly nicer format, with everything from headings and text formatting to images and tables[6].

[6] See https://www.markdownguide.org/ for a detailed guide