Jake Page for Glasskube
Posted on Apr 10

The guide to Git I never had.

#webdev #beginners #programming #tutorial

🩺 Doctors have stethoscopes.
🔧 Mechanics have spanners.
👷 We developers, have Git.

Have you noticed that Git is so integral to working with code that people hardly ever include it in their tech stack or on their CV at all? The assumption is you know it already, or at least enough to get by, but do you?

Git is a Version Control System (VCS). The ubiquitous technology that enables us to store, change, and collaborate on code with others.

> 🚨 As a disclaimer, I would like to point out that Git is a massive topic. Git books have been written, and blog posts that could be mistaken for academic papers too. That's not what I'm going for here. **I'm no Git expert**. My aim here is to write the Git fundamentals post I wish I had when learning Git.

As developers, our daily routine revolves around reading, writing, and reviewing code. Git is arguably one of the most important tools we use. Mastering the features and functionalities Git offers is one of the best investments you can make in yourself as a developer.

So let's get started

> If you feel I missed or should go into more detail on a specific command, let me know in the comments below. And I will update this post accordingly. 🙏

While we are on the topic

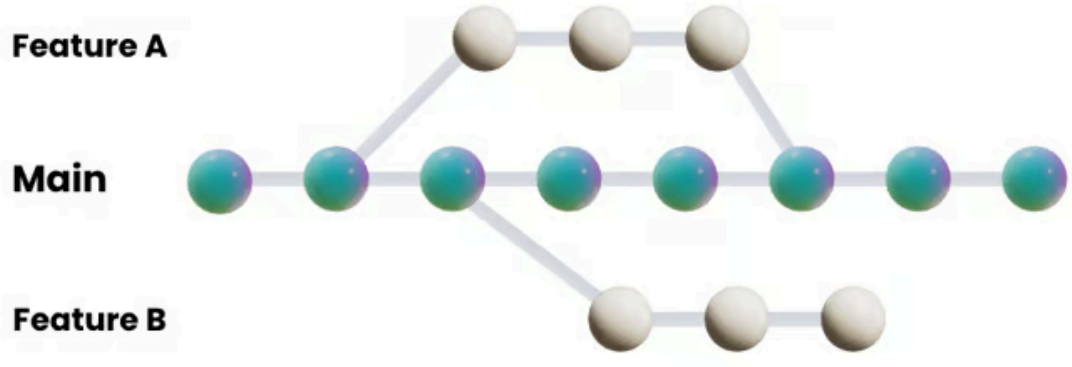
If you are looking to put your Git skills to work and would like to contribute to Glasskube, we officially launched in February and we aim to be the no-brainer, default solution for Kubernetes package management. With your support, we can make it happen. The best way to show your support is by starring us on GitHub ⭐

Let's lay down the foundations

Does Git ever make you feel like Peter Griffin?
If you don't learn Git the right way you run the risk of constantly scratching your head, getting stuck on the same issues, or rueing the day you see another merge conflict appear in your terminal. Let's ensure that doesn't happen by defining some foundational Git concepts.

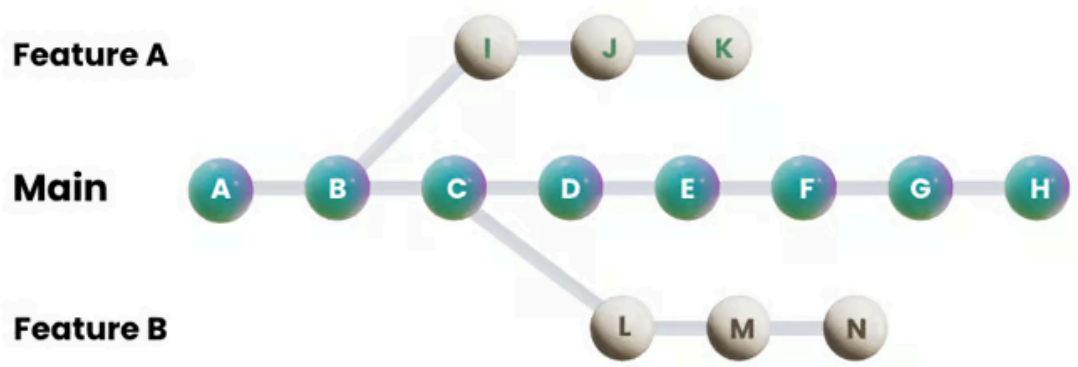
Branches

In a Git repository, you'll find a main line of development, typically named "main" or "master" (deprecated) from which several branches diverge. These branches represent simultaneous streams of work, enabling developers to tackle multiple features or fixes concurrently within the same project.



Commits

Git commits serve as bundles of updated code, capturing a snapshot of the project's code at a specific point in time. Each commit records changes made since the last commit was recorded, all together building a comprehensive history of the project's development journey.



When referencing commits you will generally use its uniquely identified cryptographic hash.

Example:

```
git show abc123def456789
```

This shows detailed information about the commit with that hash.

Tags

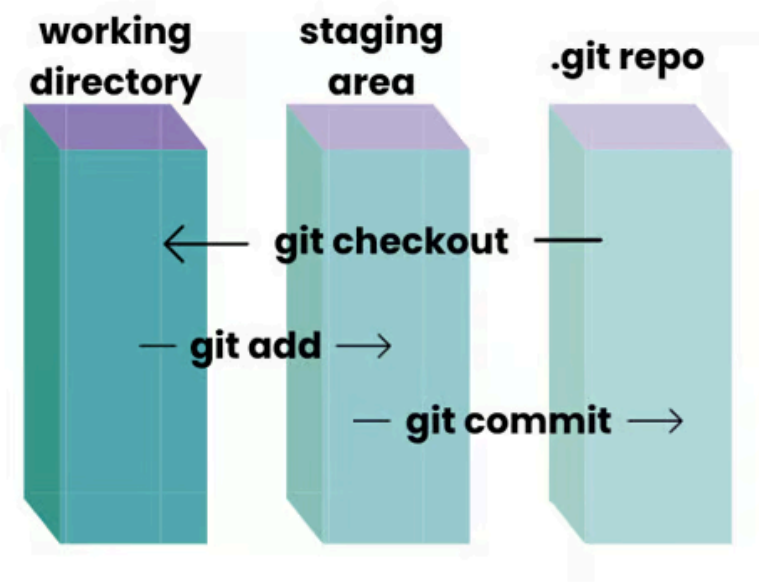
Git tags serve as landmarks within the Git history, typically marking significant milestones in a project's development, such as `releases`, `versions`, or `standout commits`. These tags are invaluable for marking specific points in time, often representing the starting points or major achievements in a project's journey.

HEAD

The most recent commit on the currently checked-out branch is indicated by the `HEAD`, serving as a pointer to any reference within the repository. When you're on a specific branch, `HEAD` points to the latest commit on that branch. Sometimes, instead of pointing to the tip of a branch, `HEAD` can directly point to a specific commit (`detached HEAD` state).

Stages

Understanding Git stages is crucial for navigating your Git workflow. They represent the logical transitions where changes to your files occur before they are committed to the repository.
Let's delve into the concept of Git stages:



Working directory 👷

The `working directory` is where you edit, modify, and create files for your project. Representing the current state of your files on your local machine.

Staging area 🚉

The `staging` area is like a holding area or a pre-commit zone where you prepare your changes before committing them to the repository.

> Useful command here: `git add`
> Also `git rm` can be used to unstage changes

Local repository 🗄️

The local repository is where Git permanently stores the committed changes. It allows you to review your project's history, revert to previous states, and collaborate with others on the same codebase.

> You can commit changes that are ready in the staging area with: `git commit`

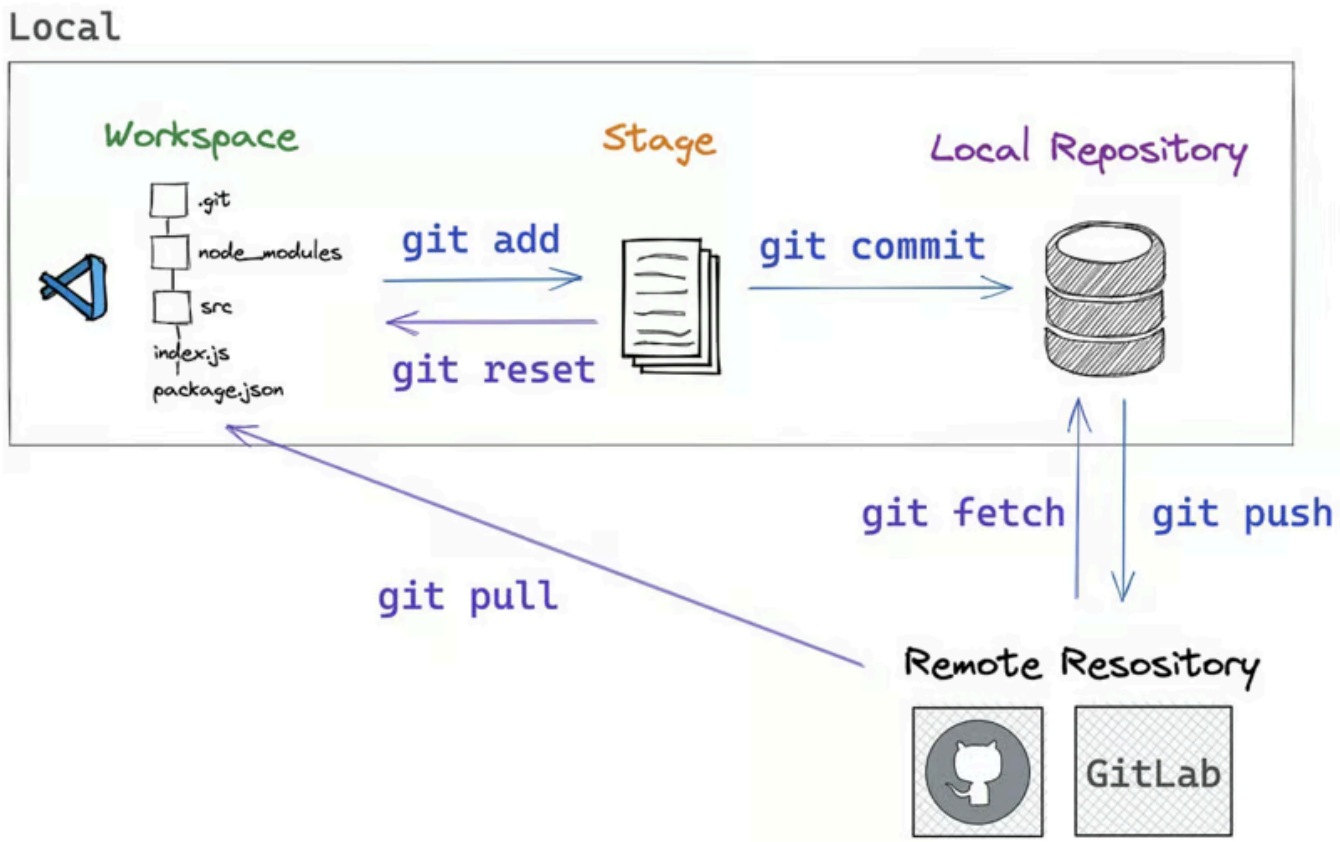
Remote repository 🛬

The remote repository is a centralized location, typically hosted on a server (like GitHub, GitLab, or Bitbucket), where you can share and collaborate with others on your project.

> You can use commands like `git push` and `git pull` to push/pull your committed changes from your local repository to the remote repository.

Getting Started with Git

Well, you have to start somewhere, and in Git that is your `workspace`. You can `fork` or `clone` an existing repository and have a copy of that workspace, or if you are starting completely fresh in a new local folder on your machine you have to turn it into a git repository with `git init`. The next step, crucially not to be overlooked is setting up your credentials.



Credentials set up

When running pushing and pulling to a remote repository you don't want to have to type your username and password every time, avoid that by simply executing the following command:

```
git config --global credential.helper store
```

The first time you interact with the remote repository, Git will prompt you to input your username and password. And after that, you won't be prompted again

> It's important to note that the credentials are stored in a plaintext format within a `.git-credentials` file.

To check the configured credentials, you can use the following command:

```
git config --global credential.helper
```

Working with branches

When working locally it’s crucial to know which branch you are currently on. These commands are helpful:

```
# Will show the changes in the local repository
git branch

# Or create a branch directly with
git branch feature-branch-name
```

To transition between branches use:

```
git switch
```

Additionally to transitioning between them, you can also use:

```
git checkout
# A shortcut to switch to a branch that is yet to be created with
git checkout -b feature-branch-name
```

To check the repository's state, use:

```
git status
```

A great way to always have a clear view of your current branch is to see it right in the terminal. Many terminal add-ons can help with this. Here is one.

Working with commits

When working with commits, utilize git commit -m to record changes, git amend to modify the most recent commit, and try your best to adhere to commit message conventions.

```
# Make sure to add a message to each commit
git commit -m "meaningful message"
```

If you have changes to your last commit, you don't have to create another commit altogether, you can use the -—amend flag to amend the most recent commit with your staged changes

```
# make your changes
git add .
git commit --amend
# This will open your default text editor to modify the commit mes
git push origin your_branch --force
```

> ⚠️ Exercise caution when utilizing `--force`, as it has the potential to overwrite the history of the target branch. Its application on the main/master branch should be generally avoided.
>
> As a rule of thumb it's better to commit more often than not, to avoid losing progress or accidentally resetting the unstaged changes. One can rewrite the history afterward by squashing multiple commits or doing an interactive rebase.

Use `git log` to show a chronological list of commits, starting from the most recent commit and working backward in time

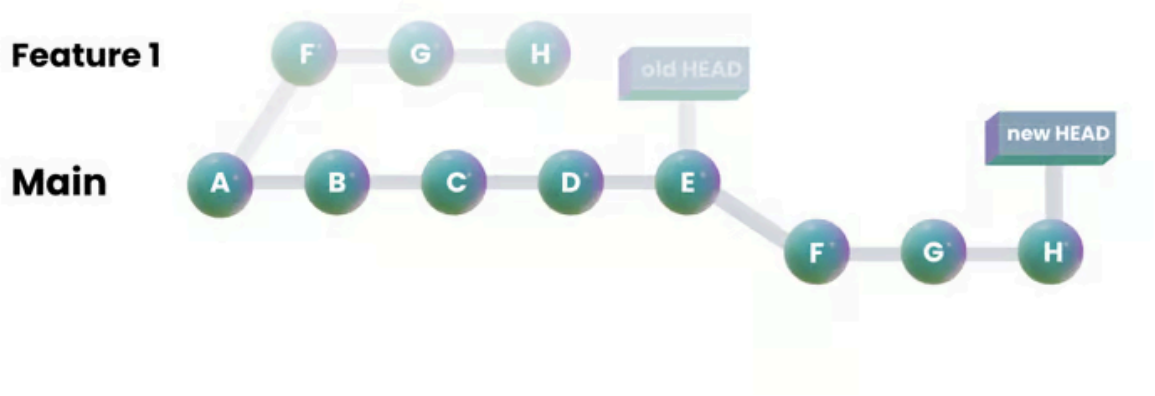
Manipulating History

Manipulating History involves some powerful commands. `Rebase` rewrites commit history, `Squashing` combines multiple commits into one, and `Cherry-picking` selects specific commits.

Rebasing and merging

It makes sense to compare rebasing to merging since their aim is the same but they achieve it in different ways. The crucial difference is that rebasing rewrites the project's history. A desired choice for projects that value clear and easily understandable project history. On the other hand, merging maintains both branch histories by generating a new merge commit.

During a rebase, the commit history of the feature branch is restructured as it's moved onto the `HEAD` of the main branch



The workflow here is pretty straightforward.

Ensure you're on the branch you want to rebase and fetch the latest changes from the remote repository:

```
git checkout your_branch
git fetch
```

Now choose the branch you want to rebase onto and run this command:

```
git rebase upstream_branch
```

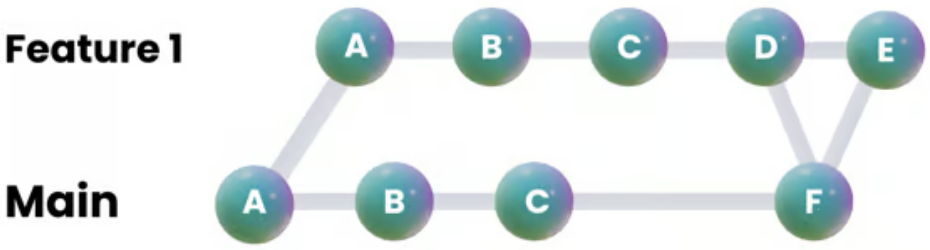
After rebasing, you might need to force-push your changes if the branch has already been pushed to a remote repository:

```
git push origin your_branch --force
```

> ⚠️ Exercise caution when utilizing `--force`, as it has the potential to overwrite the history of the target branch. Its application on the main/master branch should be generally avoided.

Squashing

Git squashing is used to condense multiple commits into a single, cohesive commit.



The concept is easy to understand and especially useful if the method of unifying code that is used is rebasing, since the history will be altered, it's important to be mindful of the effects on the project history. There have been times I have struggled to perform a squash, especially using interactive rebase, luckily we have some tools to help us. This is my preferred method of squashing which involves moving the HEAD pointer back X number of commits while keeping the staged changes.

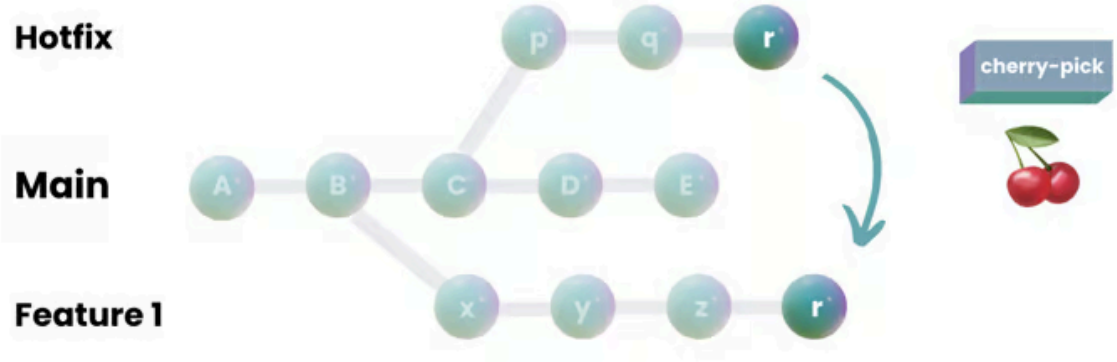
```
# Change to the number after HEAD~ depending on the commits you wa
git reset --soft HEAD~X
git commit -m "Your squashed commit message"
git push origin your_branch --force
```

> ⚠️ Exercise caution when utilizing `--force`, as it has the potential to overwrite the history of the target branch. Its application on the main/master branch should be generally avoided.

Cherry-picking

Cherry-picking is useful for selectively incorporating changes from one branch to another, especially when merging entire branches is

not desirable or feasible. However, it's important to use cherry-picking judiciously, as it can lead to duplicate commits and divergent histories if misapplied



To perform this first you have to identify the commit hash of the commit you would like to pick, you can do this with `git log`. Once you have the commit hash identified you can run:

```
git checkout target_branch
git cherry-pick <commit-hash> # Do this multiple times if multiple
git push origin target_branch
```

Advanced Git Commands

Signing commits

Signing commits is a way to verify the authenticity and integrity of your commits in Git. It allows you to cryptographically sign your commits using your GPG (GNU Privacy Guard) key, assuring Git that you are indeed the author of the commit. You can do so by creating a GPG key and configuring Git to use the key when committing.
Here are the steps:

```
# Generate a GPG key
gpg --gen-key

# Configure Git to Use Your GPG Key
git config --global user.signingkey <your-gpg-key-id>

# Add the public key to your GitHub account

# Signing your commits with the -S flag
git commit -S -m "Your commit message"

# View signed commits
git log --show-signature
```

Git reflog

A topic that we haven’t explored is Git references, they are pointers to various objects within the repository, primarily commits, but also tags and branches. They serve as named points in the Git history, allowing users to navigate through the repository's timeline and access specific snapshots of the project. Knowing how to navigate git references can be very useful and they can use git reflog to do just that.
Here are some of the benefits:

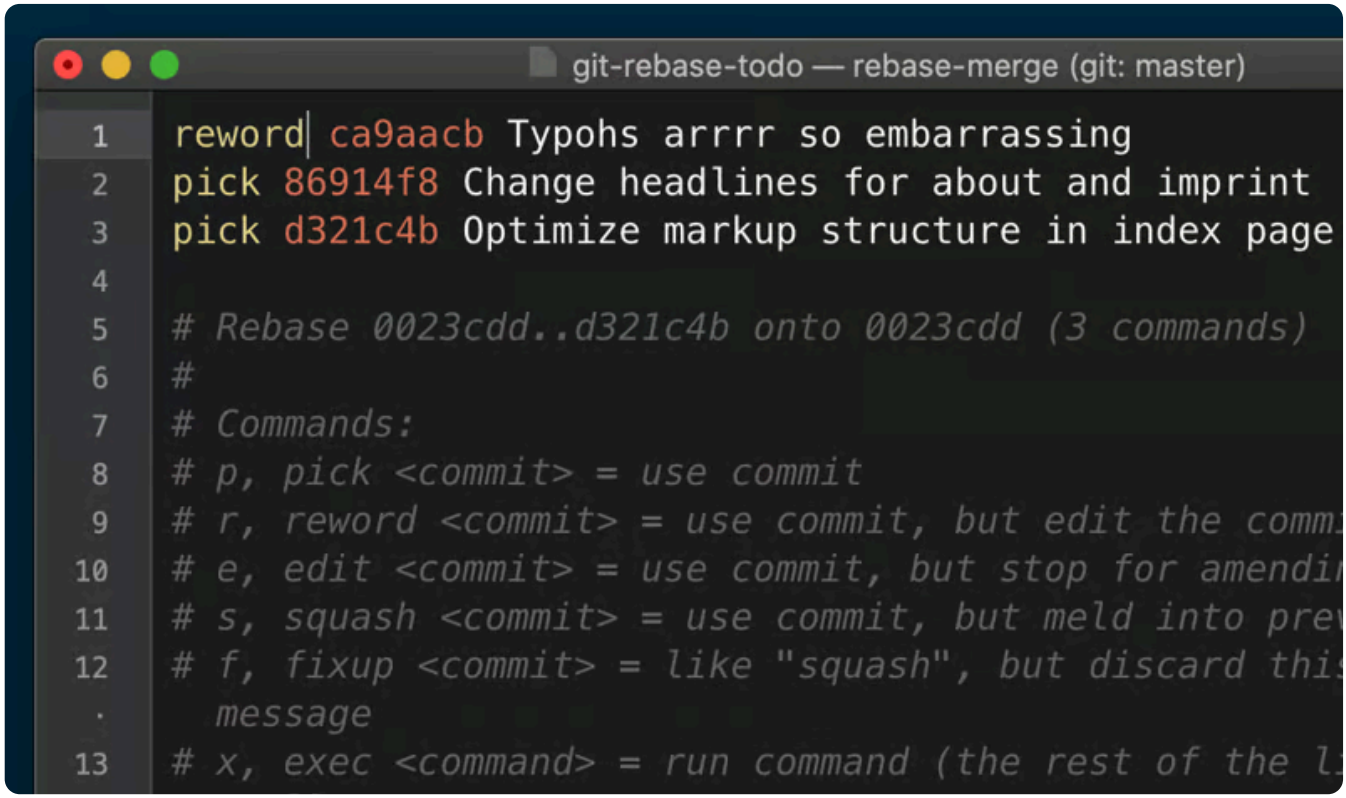
- Recovering lost commits or branches
- Debugging and troubleshooting
- Undoing mistakes

Interactive rebase

Interactive rebase is a powerful Git feature that allows you to rewrite commit history interactively. It enables you to modify, reorder, combine, or delete commits before applying them to a branch.

In order to use it you have to become familiar with the possible actions such are:

- Pick (“p“)
- Reword (“r“)
- Edit (“e“)
- Squash (“s“)
- Drop (“d“)



Here is a useful video to learn how to perform an interactive rebase in the terminal, I have also linked a useful tool at the bottom of the blog post.

Collaborating with Git

Origin vs Upstream

The **origin** is the default remote repository associated with your local Git repository when you clone it. If you've forked a repository, then that fork becomes your "origin" repository by default.

Upstream on the other hand refers to the original repository from which your repository was forked.

To keep your forked repository up-to-date with the latest changes from the original project, you git fetch changes from the "upstream" repository and merge or rebase them into your local repository.

To see the remote repositories associated with you local Git repo, run:

```
git remote -v
```

Conflicts

Don’t panic, when trying to merge or rebase a branch and conflicts are detected it only means that there are conflicting changes between different versions of the same file or files in your repository and they can be easily resolved (most times).

They are typically indicated within the affected files, where Git inserts conflict markers `<<<<<<<`, `=======` and `>>>>>>>` to highlight the conflicting sections.
Decide which changes to keep, modify, or remove, ensuring that

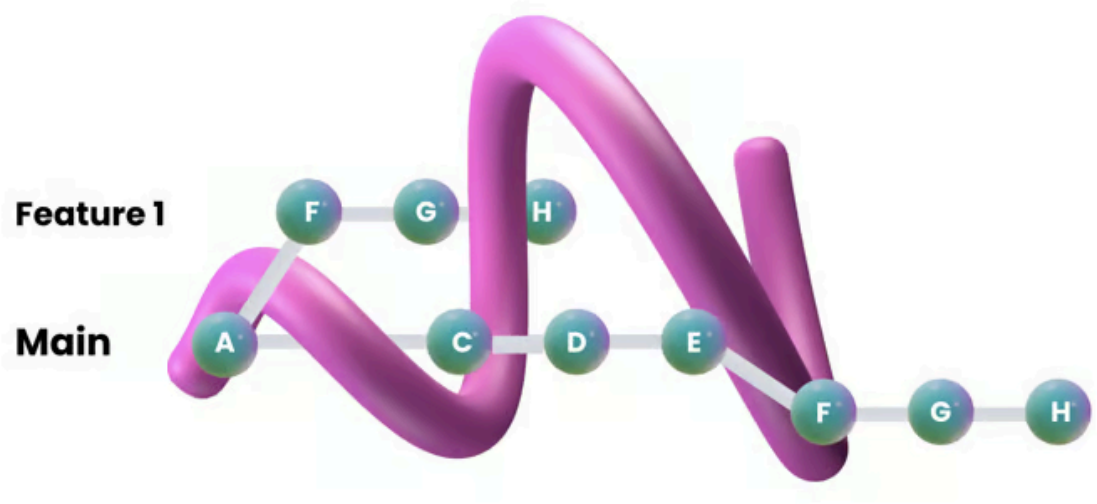
the resulting code makes sense and retains the intended functionality.

After manually resolving conflicts in the conflicted files, remove the conflict markers `<<<<<<<`, `=======`, and `>>>>>>>` and adjust the code as necessary.

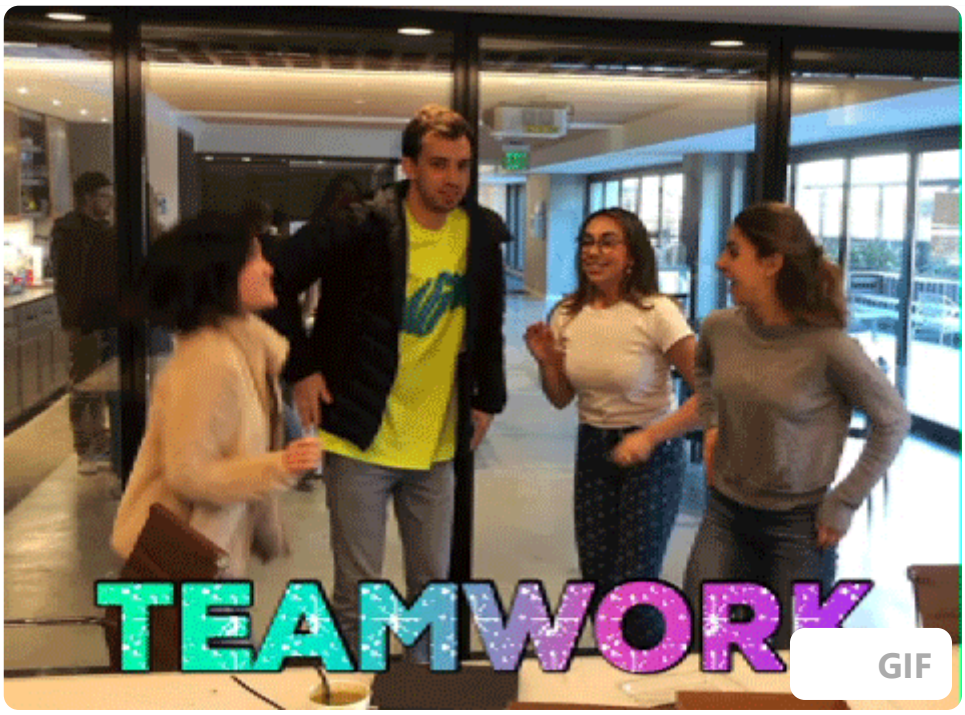
Save the changes in the conflicted files once you're satisfied with the resolution.

> If you have issues resolving conflicts, this video does a good job at explaining it.

Popular Git workflows



Various Git workflows exist, however, it's important to note that there's no universally "best" Git workflow. Instead, each approach has its own set of pros and cons. Let's explore these different workflows to understand their strengths and weaknesses.



Feature Branch Workflow 🌱

Each new feature or bug fix is developed in its own branch and then merge it back into the main branch once completed.

> - **Strength**: Isolation of changes and reducing conflicts.
> - **Weakness**: Can become complex and require diligent branch management.

Gitflow Workflow 🌊

Gitflow defines a strict branching model with predefined branches for different types of development tasks.

It includes long-lived branches such as main, develop, feature branches, release branches, and hotfix branches.

> - **Strength**: Suitable for projects with scheduled releases and long-term maintenance.
> - **Weakness**: Can be overly complex for smaller teams

Forking Workflow 🍴

In this workflow, each developer clones the main repository, but instead of pushing changes directly to it, they push changes to their own fork of the repository. Developers then create pull requests to propose changes to the main repository, allowing for code review and collaboration before merging.

This is the workflow we use to collaborate on the open-source Glasskube repos.

> - **Strength**: Encourages collaboration from external contributors without granting direct write access to the main repository.
> - **Weakness**: Maintaining synchronization between forks and the main repository can be challenging.

Pull Request Workflow ⏩

Similar to the Forking Workflow, but instead of forking, developers create feature branches directly in the main repository.

> - **Strength**: Facilitates code review, collaboration, and knowledge sharing among team members.
> - **Weakness**: Dependency on human code reviewers can introduce delays in the development process.

Trunk-Based Development 🪵

If you are on a team focused on rapid iteration and continuous delivery, you might use trunk-based development which developers work directly on the main branch committing small and frequent changes.

> - **Strength**: Promotes rapid iteration, continuous integration, and a focus on delivering small, frequent changes to

production.
- **Weakness**: Requires robust automated testing and deployment pipelines to ensure the stability of the main branch, may not be suitable for projects with stringent release schedules or complex feature development.

What the fork?

Forking is highly recommended for collaborating on Open Source projects since you have complete control over your own copy of the repository. You can make changes, experiment with new features, or fix bugs without affecting the original project.

> 💡 What took me a long time to figure out was that although forked repositories start as separate entities, they retain a connection to the original repository. This connection allows you to keep track of changes in the original project and synchronize your fork with updates made by others.

That's why even when you push to your origin repository. Your changes will show up on the remote also.

Git Cheatsheet

```
# Clone a Repository
git clone <repository_url>

# Stage Changes for Commit
git add <file(s)>

# Commit Changes
git commit -m "Commit message"

# Push Changes to the Remote Repository
git push
```

```
# Force Push Changes (use with caution)
git push --force

# Reset Working Directory to Last Commit
git reset --hard

# Create a New Branch
git branch <branch_name>

# Switch to a Different Branch
git checkout <branch_name>

# Merge Changes from Another Branch
git merge <branch_name>

# Rebase Changes onto Another Branch (use with caution)
git rebase <base_branch>

# View Status of Working Directory
git status

# View Commit History
git log

# Undo Last Commit (use with caution)
git reset --soft HEAD^

# Discard Changes in Working Directory
git restore <file(s)>

# Retrieve Lost Commit References
git reflog

# Interactive Rebase to Rearrange Commits
git rebase --interactive HEAD~3
```

Bonus! Some Git tools and resources to make your life easier.

- Tool for interactive rebasing.
- Cdiff to view colorful, incremental diffs.
- Interactive Git branching playground

If you like this sort of content and would like to see more of it, please consider supporting us by giving us a Star on GitHub 🙏

Top comments (25)

Juraj • Apr 11

I like a clean commit history, interactive rebasing was a real find for me!

vincanger • Apr 11

How did I git anything done before without this?

Jake Page • Apr 11

badumdum

Matija Sosic • Apr 11

Good stuff! I'm using git every day, but there is always something new you can learn. This is a nice cheat sheet to keep close.

Nevo David • Apr 11

This is awesome!

Awenath • Apr 11 • Edited

Great post ! I would add that you can make your life easier when conflicts happen with `git mergetool`, which can be configured to your favorite file comparator. I personnaly use p4merge that I find quite clear to use 😉

Jake Page • Apr 11

super cool @awenath I was unaware of those tool. Will check them out for sure

Rohit • Apr 12

Awesome....very interesting and knowledgeable topics all you are mentioned

Jonathan R. Mugisha • Apr 12

I see myself coming back to this.

ppaannggggg • Apr 12

Thanks, clear figures showing how git works

Daniel Hofman • Apr 12

Great post! The section on common Git commands resonated with me. CommandGit app builds on this foundation by letting users execute these commands in batches, making complex workflows simpler without hiding the command line magic.

Qyrus • Apr 11

You git it! Great read! Informative, humorous, and well written. Thanks for sharing!

Jake Page • Apr 11

Much appreciated @qyrusai !

CaptainGenesisX • Apr 12 • Edited

Hello. Thank you for publishing this great article! I am still working on reading through the guide but wanted to bring something to your attention. I am using Visual Studio 2022 and my Git is already connected to a repo with my own branch that has unstaged changes. I followed some of your sample commands in the "working with branches" section such as this:

```
# Will show the changes in the local repository
git branch
```

However, for me, doing `git branch` does not show the changes in the local repo. It simply shows me the branches available on my local machine (master, and pbi/123456).

Doing `git status` is what shows me the changes in my branch. Not sure if I am just misunderstanding or if your comment needs modified. I mean no disrespect and I appreciate your content!

Stephen Potter • Apr 13

I have used git extensively throughout the majority of my 23 year career. One thing I tell newcomers is to not be afraid of the command line, but don't feel inadequate if you use guis. GitHub desktop is a great application, especially for viewing commit contents and diffs. I use it every day along with the command line. Try it out. I also ask newcomers to make a PR on their first or second day, even just a readme fix, so they understand the process

Benoit COUETIL 💫 • Apr 14

Thank you for sharing, nice work for newbies 👏

You differentiate feature branch mode and PR mode, but they are the same from a git workflow perspective (or you did not explain how they are different).

And trunk-based-development is rarely commiting directly to trunk.

Keep up the good work 😊

Ankur Tyagi • Apr 11

Nice blog @jakepage91 👊

Jake Page • Apr 11

Cheers Ankur!

Possawat Sanorkam • Apr 13

Good job!

View full discussion (27 comments)

Some comments may only be visible to logged-in visitors. Sign in to view all comments.

Code of Conduct • Report abuse

Glasskube

🧊 The missing Package Manager for Kubernetes 📦

Support Open Source and leave us a ⭐ on GitHub ❤️‍🔥

Leave a Star ⭐

More from Glasskube

Why contributor guidelines matter.

#opensource #beginners #tutorial #github

I built a top-tier Discord server for Open Source.

#productivity #opensource #tutorial #beginners

🔥 Matomo 5 UPGRADE - A step-by-step GUIDE 💪

#tutorial #devops #opensource #kubernetes