{JS}
MASTERY

Git & GitHub Handbook

Created by **JS Mastery**
Visit ***jsmastery.pro*** for more

Introduction

Welcome to the **Ultimate Git Guide**, your go-to resource for mastering Git, the industry-standard version control system.

Inside, you'll find clear explanations of **Git commands** with detailed flags and real-world examples.

Learn not just how to use commands like `` `git add` ``, `` `git commit` ``, and `` `git rebase` ``, but also when and why to apply them in real-world scenarios—whether you're working solo, collaborating with a team, or contributing to open-source projects.

We've included tips and tricks for maintaining a clean commit history, resolving **merge conflicts**, and using Git's `` `reset` `` and `` `revert` `` features to undo mistakes and fix bugs.

Introduction

You'll also explore advanced Git features that can boost your productivity and make troubleshooting easier.

With practical advice and best practices, this guide will help you streamline your Git workflow and use Git like a pro.

Let's get started!

Why Use Git?

Git is a version control system that allows you to track changes in your code and collaborate with others efficiently. It keeps a history of your work, making it easy to revert changes if something goes wrong.

Using Git brings numerous advantages that enhance the development process for both individual developers and teams.

Here are some compelling reasons to incorporate Git into your workflow:

- ⭐ Staging Area
- ⭐ Undoing Mistakes
- ⭐ Integration with Other Tools
- ⭐ Community and Support
- ⭐ Standard in the Industry
- ⭐ Version Control
- ⭐ Collaboration
- ⭐ Branching & Merging
- ⭐ Distributed System
- ⭐ Tracking Changes

Essential Commands

Initialize a Git repository

```
git init
```

Initializes a new Git repository. This command creates a new Git repository in the current directory. It sets up the basic files and directories needed to start tracking changes.

Clone repository

```
git clone [repository URL]
```

Clones an existing Git repository. This command creates a copy of an existing repository on your local machine. It copies the entire history and files of the specified repository to your local machine.

Essential Commands

Add file to staging

```
git add [file/directory]
```

Adds a file or directory to the staging area. This command prepares the changes for the next commit. It adds the specified file or directory to the index.

Commit with a message

```
git commit -m "[commit message]"
```

Creates a new commit with a message describing the changes made. This command creates a new commit with the changes you made to your local repository. The commit message describes the changes made in this commit.

Essential Commands

Pull changes

```
git pull
```

Updates the local repository with changes from the remote repository. It pulls the changes from the remote repository and merges them with the local changes.

Push changes

```
git push
```

This command pushes the local changes to the remote repository. It updates the remote repository with the changes you made locally.

Essential Commands

Check status

```
git status
```

Shows the current status of the repository. This command shows the status of the repository and the changes that are currently staged or unstaged.

List branches

```
git branch
```

This command lists all the branches in the current repository. It shows the current branch you're on and highlights it with an asterisk.

Essential Commands

Switch a branch

```
git checkout [branch name]
```

This command switches to the specified branch. It updates the working directory to match the contents of the specified branch.

Merge branch

```
git merge [branch name]
```

This command merges the specified branch into the current branch. It combines the changes from both branches and creates a new commit.

Essential Commands

List all commits

```
git log
```

This command shows a list of all commits in the repository. It displays the author, date, and commit message for each commit the repository has.

List remote repositories

```
git remote -v
```

This command lists all the remote repositories associated with the local repository. It shows the URL of each remote repository.

Essential Commands

Check difference

```
git diff [file]
```

This command shows the differences between the working directory and the staging area or the repository. It displays the changes made to the specified file.

Fetch changes

```
git fetch
```

This command downloads the changes made in the remote repository and updates your local repository, but it does not merge the changes with your local branch.

Essential Commands

Reset changes

```
git reset [file]
```

This removes the specified file from the staging area, effectively undoing any changes made to the file since the last commit. It does not delete the changes made to a file.

Revert changes

```
git revert [commit]
```

Creates a new commit that undoes the changes made in the specified commit. It does not delete the specified commit, but it creates a new commit that reverts the changes made in that commit.

Advanced Commands

Stash

```
git stash
```

This command allows you to save your changes without committing them, which can be useful when you need to switch branches or work on a different task.

Cherry Pick

```
git cherry-pick [commit]
```

This command allows you to selectively apply changes made in a specific commit to your current branch, which can be useful when you need to incorporate changes made in another branch without merging the entire branch.

Advanced Commands

Bisect

```
git bisect
```

This command allows you to perform a binary search through your commit history to find the commit that introduced a bug.

Blame

```
git blame
```

This command allows you to see who last modified each line of a file and when they did it, which can be useful when you need to find out who introduced a specific change or when a specific change was made.

Advanced Commands

Reflog

```
git reflog
```

This command allows you to view a log of all changes made to Git references, which can be useful when you need to recover lost commits.

Worktree

```
git worktree
```

This command allows you to work on multiple branches at the same time in separate working directories, which can be useful when you need to switch between branches quickly without losing your current changes.

Advanced Commands

Filter branch

```
git filter-branch
```

This command allows you to rewrite Git history by applying filters to branches, which can be useful when you need to remove sensitive data from your Git repository.

Merge Squash

```
git merge --squash
```

This command allows you to merge changes from one branch into another branch as a single commit, which can be useful when you want to maintain a clean commit history.

Advanced Commands

Submodule

```
git submodule
```

This command allows you to include one Git repository within another Git repository, which can be useful when you need to use code from one repository in another repository.

Submodule foreach

```
git submodule foreach
```

This command allows you to run a Git command in each submodule, which can be useful when you need to update multiple submodules at once.

Advanced Commands

Rebase

```
git rebase -i
```

This command allows you to interactively rewrite Git history by reordering, editing, or removing commits, which can be useful when you need to clean up your commit history.

Rebase

```
git rebase
```

This command allows you to apply the changes made in one branch onto another branch, which can be useful when you need to update a feature branch with changes made in the master branch.

Advanced Commands

Git tips that you might find helpful

Use Git Hooks

Git hooks are scripts that Git can run automatically before or after certain events, such as committing or pushing code. You can use Git hooks to automate tasks, run tests, or enforce code style guidelines.

Learn about Git Internals

Git is built on a set of data structures that are used to store and manage code changes. Understanding how Git works under the hood can help you troubleshoot issues and optimize your workflow.

Advanced Commands

Git tips that you might find helpful

Use Git Aliases

Git aliases are custom shortcuts for Git commands. You can use aliases to save time and make your workflow more efficient. For example, you could create an alias for git status that is shorter and easier to remember.

Use Git Configurations

Git configurations allow you to customize various aspects of how Git behaves. For example, you can configure Git to automatically rebase branches when you pull changes, or to always use a specific text editor for commit messages.

What are Flags in Git?

Flags (or options) in Git commands modify the behavior of the command or provide additional functionality. They usually start with a single dash `(-)` for short flags or double dashes `(--)` for long flags.

Using flags allows you to customize commands according to your needs, making your workflow more efficient.

Importance of Flags

- ***Customization***: Flags let you tailor commands to your workflow.
- ***Efficiency***: Combining flags can save time and reduce commands.
- ***Control***: Flags provide more control over operations.

Useful Git Flags

-m

```
git commit -m "Your commit message"
```

Allows you to provide a commit message directly in the cmd line, avoiding the need to open an editor.

-u

```
git add -u
```

Used with git add. Stages changes to tracked files only, ignoring untracked files.

Useful Git Flags

`--amend`

```
git commit --amend
```

Modifies the last commit by adding new changes or updating the commit message.

`--oneline`

```
git log --oneline
```

Used with git log. Displays each commit on a single line, making the log easier to read.

Useful Git Flags

--graph

```
git log --graph
```

Used with git log. Visualizes the commit history as a graph, helping you understand the branch structure.

-b

```
git checkout -b <branch_name>
```

Used with git checkout. Creates a new branch and switches to it in a single command.

Useful Git Flags

`--no-ff`

```
git merge --no-ff <branch_name>
```

Used with git merge. Forces a merge commit even if the merge could be performed with a fast-forward.

`--rebase`

```
git pull --rebase
```

Used with git pull. Reapplies your local commits on top of the fetched changes, creating a cleaner history.

Useful Git Flags

`--force`

```
git push --force
```

Used with git push. Forces a push to the remote repository, overwriting changes. Use with caution!

`--hard`

```
git reset --hard <commit>
```

Resets the working directory and staging area to a specific commit, discarding all changes.

Useful Git Flags

`--no-commit`

```
git revert <commit_hash> --no-commit
```

Used with git revert to apply the changes without immediately committing them.

`--no-edit`

```
git revert <commit_hash> --no-edit
```

Prevents Git from opening the commit message editor when reverting a commit.

Useful Git Flags

`--soft`

```
git reset --soft <commit_hash>
```

To move the HEAD pointer to a specific commit without changing the staging area or working directory. This keeps changes staged for a future commit.

`--hard`

```
git reset --hard <commit_hash>
```

Resets the HEAD pointer, staging area, and working directory to the specified commit. This command deletes all changes, so use it with caution!

Useful Git Flags

`--mixed` *(default)*

```
git reset --mixed <commit_hash>
```

Moves the HEAD pointer and updates the staging area to match the specified commit, but leaves the working directory unchanged. This is useful for un-staging files.

`push`

```
git stash push
```

Saves your local changes to a stash and clears your working directory. You can later apply these changes.

Useful Git Flags

`stash apply stash@{index}`

```
git stash apply stash@{2}
```

This command applies the changes from a specific stash without removing it. Replace {index} with the stash number (e.g., stash@{2}).

`list`

```
git stash list
```

Lists all stashed changes, giving you an overview of what you have saved.

Useful Git Flags

pop

```
git stash pop
```

Applies the changes saved in the latest stash and removes that stash entry from the list.

drop

```
git stash drop stash@{0}
```

Deletes a specific stash entry, allowing you to clean up your stash list.

Real-World Use Cases

Collaborating on a Team Project

When multiple developers are working on the same project, Git is essential for ensuring that everyone's changes can be integrated smoothly. For instance:

Use Case:

A team is building a new feature for an e-commerce site. Each developer can work on separate branches for their part of the feature, such as the frontend UI or backend API.

When the work is complete, they can merge their branches into the main codebase without overwriting each other's work.

Git tracks changes and makes merging easy.

Real-World Use Cases

Handling Hotfixes on a Live Site

Suppose you've deployed a website, but a critical bug appears after launch. You need to fix it without disrupting ongoing development work.

Use Case:

While the team continues to work on the next release in the development branch, a dev creates a hotfix branch from the main branch to address the bug.

Once the fix is applied, it's merged into main and immediately deployed, while also being merged back into development to keep everything in sync.

Key Git Features:

Branching, Staging, Merging

Real-World Use Cases

Rolling Back a Buggy Release

Sometimes, a new update introduces bugs that are hard to fix right away. Git lets you easily roll back to a stable version.

Use Case:

After a buggy release, the team decides to revert the project back to the last stable version. Using Git, they can either revert specific commits that introduced the bugs or reset the project to an earlier commit.

Key Git Features:

git revert, git reset, git log to view commit history.

Real-World Use Cases

Experimenting with New Features

A developer wants to experiment with a new feature without affecting the main project.

Use Case:

The developer creates a new branch **(`e.g., feature/new-auth-system`)** to test a new authentication system. This keeps the main project clean and unaffected by experimental changes. If the experiment is successful, they can merge the branch back into the main project.

Key Git Features:

Branching, Merging

Real-World Use Cases

Maintaining a Clean Commit History

In large projects, it's important to maintain an organized and clean history of changes.

Use Case:

A developer works on several small fixes and experiments, but before pushing their changes to the remote repository, they use Git's interactive rebase **`(git rebase -i)`** to combine some of their smaller commits into one, making the history easier to understand.

Key Git Features:

git rebase -i, git commit --amend.

Real-World Use Cases

Maintaining a Clean Commit History

In large projects, it's important to maintain an organized and clean history of changes.

Use Case:

A developer works on several small fixes and experiments, but before pushing their changes to the remote repository, they use Git's interactive rebase **`(git rebase -i)`** to combine some of their smaller commits into one, making the history easier to understand.

Key Git Features:

git rebase -i, git commit --amend.

Tips to Master Git & GitHub

Start with the Basics

Before diving into complex workflows, focus on mastering the core commands (**`git add`, `git commit`, `git push`, `git pull`, `git clone`**). These are the foundation of Git, and once you get comfortable with them, you'll be ready for more advanced topics.

Understand Branching

Branching is one of the most powerful features of Git. Practice creating new branches (**`git checkout -b`**), switching between branches, and merging them.

It’s a safe way to experiment and keep your main project intact.

Tips to Master Git & GitHub

Commit Often and Meaningfully

Make small, frequent commits instead of one big commit. This makes your project easier to track and troubleshoot. Include clear and descriptive commit messages, so you and your team know exactly what each change was for.

Explore GitHub Features

Take time to learn how to create Pull Requests (PRs), review code, and comment on issues. GitHub's interface provides collaboration features that streamline project management and code sharing.

Tips to Master Git & GitHub

Practice Staging and Reviewing Changes

Use the staging area **(`git add`)** to review what you're about to commit.

Check your changes with git status and git diff to ensure you're committing the right updates.

Use .gitignore Properly

Learn to set up a .gitignore file to exclude unnecessary files (like dependencies, build files, or local configs) from being tracked.

This keeps your repository clean and focused only on relevant code.

Tips to Master Git & GitHub

Get Comfortable with Undoing Changes

Don't be afraid of making mistakes! Git has many ways to undo changes (**`git reset`, `git revert`, `git checkout`**).

Practicing how to roll back commits or reset files can give you confidence in experimenting with your code.

Learn to Use Git Remotes

Get comfortable using remote repositories like GitHub Bitbucket or GitLab.

Practice pushing, pulling, and cloning repositories to collaborate on projects across different machines.

Tips to Master Git & GitHub

Use Aliases for Speed

Speed up your workflow by creating Git aliases for commonly used commands. For example, alias **`git st`** for git status or **`git co`** for git checkout.

This saves time on repetitive typing.

Read the Documentation

Git's documentation is detailed and helpful for solving issues or learning more about specific commands.

If you're ever stuck or want to explore a specific command or workflow in-depth, the official docs are an excellent resource to help you find answers and understand best practices.

Tips to Master Git & GitHub

Practice Collaboration with Others

Work on open-source or team projects to get hands-on experience with real-world Git workflows like resolving conflicts and managing branches.

Commit Often, Commit Smart

Make small, frequent commits with clear messages. This keeps your project history clean and helps you track changes more easily.

Use Branches for Everything

Always create branches for new features or bug fixes to keep your main branch stable and organized.

Branch Naming ⭐ Practices

Here are some naming best practices to keep your Git workflow organized & easy to understand:

Use Consistent Naming Conventions

- Stick to a clear and predictable format like **`feature/`**, **`bugfix/`**, or **`hotfix/`** followed by a short description.
- Example: **`feature/user-authentication`**, **`bugfix/navbar-alignment`**.

Use Lowercase Letters and Hyphens

Avoid spaces or uppercase letters; use hyphens to separate words for readability.

Example: **`feature/new-ui-design`**, **not `Feature/NewUIDesign`**.

Branch Naming ⭐ Practices

Be Descriptive, but Concise

- Use clear and meaningful names that describe the purpose of the branch.
- Avoid vague names like fix or update.
- Example: **`feature/user-profile-page`** is better than just **`feature/profile.`**

Include Issue or Ticket Numbers (If Applicable)

If you're working with project management tools like Jira, include the ticket or issue number in the branch name.

Example: **`feature/JIRA-1234-user-auth.`**

Branch Naming ⭐ Practices

Avoid Long Branch Names

- Keep it short and focused. Long branch names are hard to read and manage.
- Example: **`hotfix/critical-bug`**, not **`hotfix/critical-bug-in-header-footer`**.

Use Team-Specific Prefixes (If Needed)

For larger teams, consider using team or project-specific prefixes.

Example: **`backend/feature/api-endpoint`** or **`frontend/feature/user-dashboard`**.

Following these conventions makes collaboration easier and keeps the repository well-structured and manageable.

The End

Congratulations on reaching the end of our guide! But hey, learning doesn't have to stop here.

If you're eager to dive deep into something this specific and build substantial projects, our **special course on Next.js** has got you covered.

The Ultimate Next 14 Course

If you're craving a more personalized learning experience with the guidance of expert mentors, we have something for you — [Dev Career Accelerator](#).

Dev Career Accelerator

If this sounds like something you need, then don't stop yourself from leveling up your skills from junior to senior.

Keep the learning momentum going. Cheers! 🚀