

A Comprehensive Guide to Using Git and GitHub Professionally

Introduction: Git and GitHub are integral tools for modern software development, providing an efficient and collaborative approach to version control. This guide aims to provide a clear and professional overview of Git, GitHub, and their practical usage.

What is Git?

Git is a free and open-source Version Control System (VCS) that facilitates tracking changes in code, maintaining a comprehensive history, and enabling seamless collaboration within development teams. It is widely adopted and considered a staple in the software development workflow.

What is GitHub?

GitHub is a web-based platform for hosting Git repositories. It serves as a centralized hub for developers to store, collaborate, and manage their projects online. The README.md file, written in Markdown, is a key component within GitHub repositories, providing project details and documentation.

Using Git

1. **Command Line (Most Popular):** Git's command-line interface is the most widely used method for interacting with repositories. It offers a powerful and flexible way to manage code.
2. **IDEs and Code Editors (e.g., VS Code):** Integrated Development Environments (IDEs) and code editors, such as Visual Studio Code, offer user-friendly interfaces and seamless Git integration, making version control accessible during the development process.
3. **Graphical User Interface (e.g., GitKraken):** Git can also be utilized through graphical user interfaces like GitKraken, providing a visual representation of the version control process for those who prefer a more intuitive approach.

Configuring Git

Ensure your identity is correctly set up for Git usage by configuring your global settings:

```
git config --global user.name "Your Name"
git config --global user.email your@email.com
git config --list # Displays configured name and email
```

Basic Commands

1. Clone

Clone a repository onto your local machine using the following command

```
• git clone <repository_link>
```

2. Status

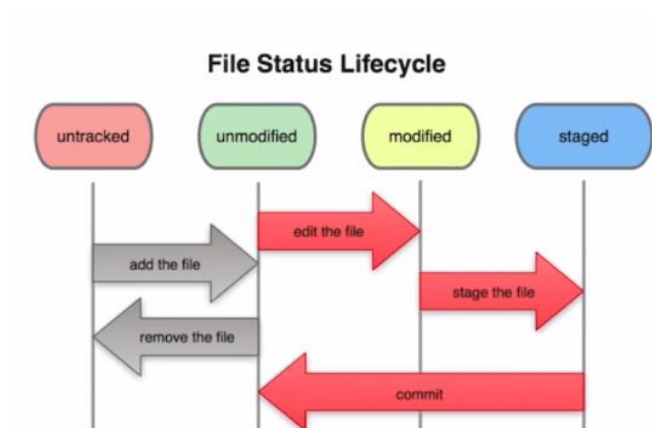
Check the state of your code with:

```
• git status
```

3. View Hidden Files

To view hidden files in Git, use:

```
ls -a
```

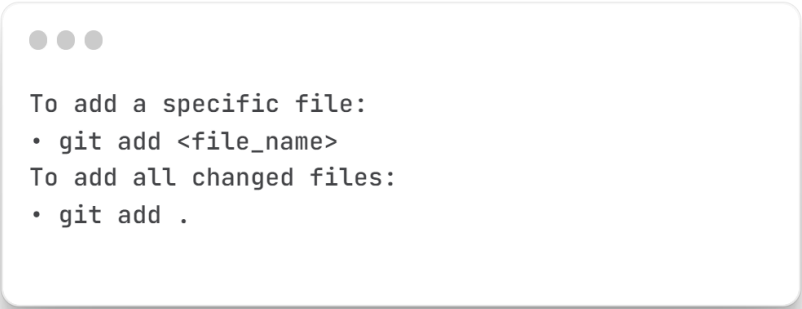


Committing Changes in Git:

After making changes to your code, it's essential to follow a structured process to incorporate those changes into your Git repository. This involves using the **add**, **commit**, and **push** commands.

1. Add

The **add** command is used to stage changes for commit. You can add specific files or all modified files in your working directory.



```
● ● ●  
  
To add a specific file:  
• git add <file_name>  
To add all changed files:  
• git add .
```

2. Commit

The **commit** command creates a record of the changes you've staged. It's important to include a meaningful commit message to describe the purpose of the changes.

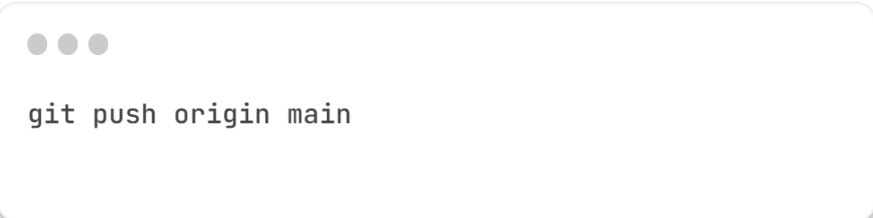


```
● ● ●  
  
git commit -m "Your descriptive commit message here"
```

Ensure your commit messages are clear and concise, providing context about the modifications made.

3. Push

After committing changes locally, you'll want to upload them to the remote repository. The **push** command accomplishes this by sending your local changes to the specified branch on the remote repository.



```
● ● ●  
  
git push origin main
```

Here, replace **main** with the branch you're working on if it's different. Ensure that your local branch is tracking the remote branch.

Initializing a New Git Repository:

The **git init** command is a fundamental step when creating a new Git repository. Below is a step-by-step guide, including commands, to help you initiate a new Git repository, add files, and interact with a remote repository on GitHub.

1. **Initialize a New Git Repository:**

Use the following command to create a new, empty Git repository in your current working directory:



```
git init
```

2. **Adding a Remote Repository (GitHub, for example):**

After creating a new repository on GitHub, use the following command to link your local repository to the remote one:

3. **Verify Remote Repository:**

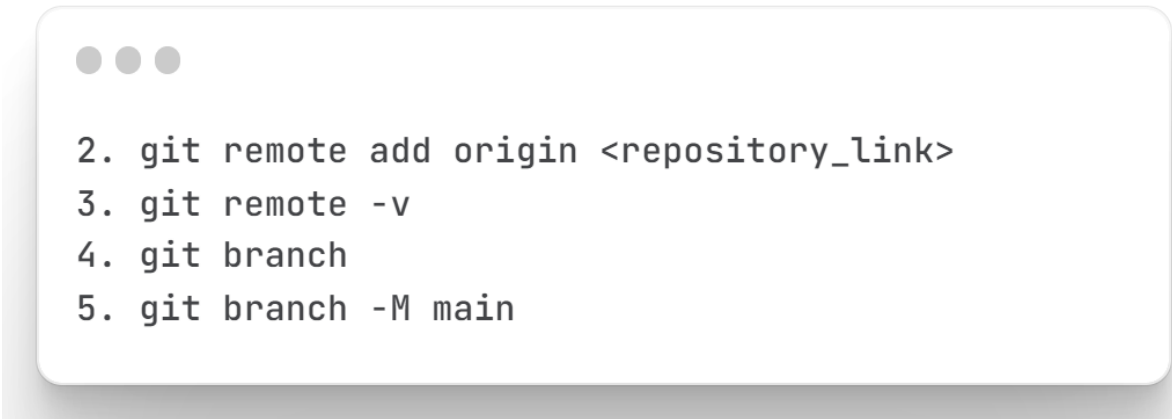
To confirm the remote repository linked, use:

4. **Checking Branches:**

To see the available branches in your repository:

5. **Rename the Default Branch (if necessary):**

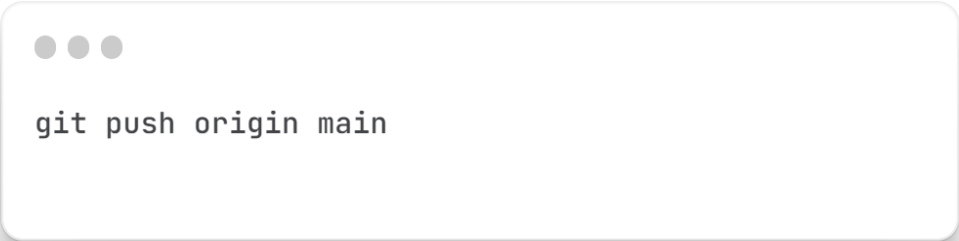
If you want to rename the default branch (e.g., from "master" to "main"), use:



```
2. git remote add origin <repository_link>
3. git remote -v
4. git branch
5. git branch -M main
```

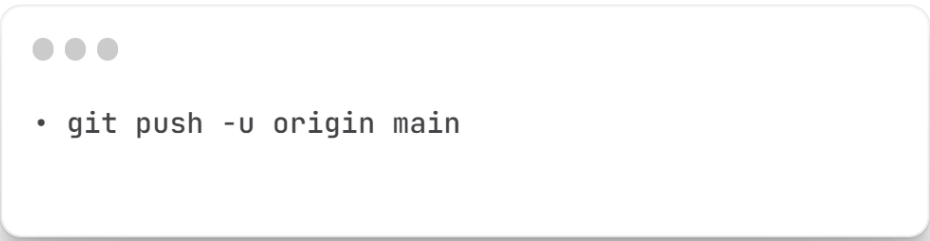
6. Pushing Changes to the Remote Repository:

After making changes and committing, use the following command to push your changes to the remote repository:



```
git push origin main
```

If it's the first push, you can use the **-u** flag to set the upstream branch. This allows you to use **git push** without specifying the remote branch and local branch names in the future:



```
• git push -u origin main
```

7. Quickly Add and Commit Changes:

If you've edited a single file and want to add and commit it in one go, you can use the **-am** flags:



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

This command stages and commits all changes, including new and modified files, with a single command.

Creating a New Directory in Your Git Repository:

When establishing a new directory within your Git repository, adhere to the following commands:

1. Create a New Directory:

Utilize the **mkdir** command to generate a new directory. For instance:

This command ensures the creation of a new folder in your existing project structure.

2. Navigate to the New Directory:

Transition into the freshly created directory by executing:

```
1. mkdir <directory_name>
2. cd <directory_name>
```

Effective Branch Management Commands in Git:

Managing branches is a crucial aspect of version control in Git. Below are key commands for branch management, along with explanations and examples:

1. **Check Available Branches:**

To view a list of existing branches in your repository:

2. **Rename a Branch:**

If you need to rename a branch, for example, from "master" to "main," use:

3. **Switch to a Different Branch:**

To navigate between branches, use the git checkout command:

4. **Create a New Branch:**

To create a new branch and switch to it in one go, use the -b option with git checkout:

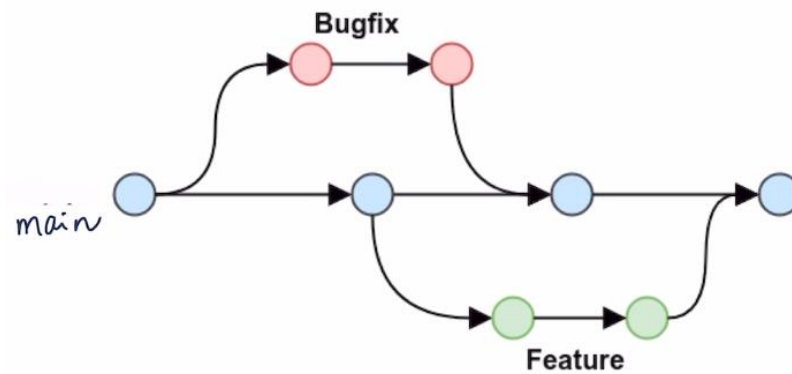
5. **Delete a Branch:**

If a branch is no longer needed, use the following command to delete it:

6. **Push a Branch with Upstream:**

If you attempt to push a branch without an upstream branch, Git will provide guidance on setting it:

```
1. git branch
2. git branch -M main
3. git checkout <branch_name>
4. git checkout -b <new_branch_name>
This creates a new branch and switches your working directory to the newly created branch.
5. git branch -d <branch_name>
Note: The branch must be fully merged into the current branch before deletion.
6. git push --set-upstream origin <branch_name>
Replace <branch_name> with the name of your branch. This command establishes the upstream
branch for subsequent pushes.
```



Efficient Code Merging and Mistake Resolution in Git:

Merging Code:

1. Compare Branches:

Utilize the **git diff** command to compare commits, branches, files, and more. To compare branches:

```
git diff <branch_name>
```

2. Merge Branches:

To merge two branches, use the **git merge** command:

```
git merge <branch_name>
```

Alternatively, create a Pull Request (PR) in GitHub to propose and merge changes.

Pull Request (PR):

1. Create a Pull Request:

Initiate changes in GitHub, creating a PR to inform others about modifications pushed to a branch:

In GitHub, click on "Compare & pull request."

Provide comments and details, allowing teammates to understand the changes.

After merging, differences between the main and feature branches can be reviewed.

2. Update Local System with Changes:

To view changes in your local system, use:

```
git pull origin main
```

This fetches and downloads content from the remote repository, updating the local repository.

Merge Conflicts:

An event that takes place when Git is unable to automatically resolve differences in code between two commits.

```
git merge <branch_name>
```

Fixing Mistakes:

Case 1: Staged Changes:

```
To unstage changes for a specific file:  
• git reset <file_name>  
To unstage all changes:  
• git reset
```

Case 2: Committed Changes (Single Commit):

If changes have been committed, use the following to uncommit:

```
• git reset HEAD~1
```

HEAD~1 points to the last commit.

Case 3: Committed Changes (Multiple Commits):

To review all commits:

- `git log`

Use the commit `hash` to reset to a specific commit:

- `git reset <commit_hash>`

To forcefully reset and discard changes:

- `git reset --hard <commit_hash>`