Git and GitHub Guide: Keeping Your Repositories in Sync

This guide covers the essential Git commands and workflows to help you maintain your GitHub repositories and keep them synchronized between your local machine and GitHub.

Table of Contents

Git Basics

Git is a distributed version control system that allows you to track changes in your code, collaborate with others, and maintain different versions of your project.

Key Concepts

- **Repository (Repo)**: A storage location for your project, containing all files and their revision history.
- **Commit**: A snapshot of your repository at a specific point in time.
- **Branch**: A parallel version of your repository that allows you to work on features without affecting the main codebase.
- **Remote**: A version of your repository hosted on a server (like GitHub).
- **Clone**: Creating a local copy of a remote repository.
- **Push**: Sending your local changes to a remote repository.
- **Pull**: Fetching changes from a remote repository and merging them into your local repository.
- **Merge**: Combining changes from different branches.

Setting Up a Repository

Initializing a New Repository

To create a new Git repository in your project directory:

```
 # Navigate to your project directory
cd /path/to/your/project

# Initialize a new Git repository
git init
```

Cloning an Existing Repository

To create a local copy of an existing GitHub repository:

```
 # Clone a repository using HTTPS
git clone https://github.com/username/repository.git

# Clone a repository using SSH (requires SSH key setup)
git clone git@github.com:username/repository.git

# Clone to a specific directory
git clone https://github.com/username/repository.git my-project-folder
```

Configuring Git

Set up your identity for all repositories on your computer:

```
 # Set your username
git config --global user.name "Your Name"

# Set your email address
git config --global user.email "your.email@example.com"

# View your configuration
git config --list
```

Basic Git Workflow

Checking Repository Status

To see the current state of your repository:

```
git status
```

This command shows: - Which files have been modified - Which changes are staged for the next commit - Which files are untracked

Tracking New Files

To start tracking a new file:

```
git add filename.txt

# Add multiple files
git add file1.txt file2.txt

# Add all files in the current directory
git add .
```

Staging Changes

To stage modified files for the next commit:

```
# Stage specific files
git add filename.txt

# Stage all modified files
git add -u

# Stage all modified and new files
git add -A
```

Committing Changes

To save your staged changes to the repository:

```
# Commit with a message
git commit -m "Add a descriptive message about your changes"

# Stage all modified files and commit in one command
git commit -am "Your commit message"
```

Viewing Commit History

To see the commit history:

```
# View commit history
git log

# View a condensed history
git log --oneline

# View history with changes
git log -p

# View history with a graph
git log --graph --oneline --decorate
```

Branching and Merging

Working with Branches

Branches allow you to develop features, fix bugs, or experiment without affecting the main codebase.

```
# List all branches (* indicates the current branch)
git branch

# Create a new branch
git branch feature-name

# Switch to a branch
git checkout feature-name

# Create and switch to a new branch in one command
git checkout -b feature-name

# Delete a branch (after merging)
git branch -d feature-name

# Force delete a branch (even if not merged)
git branch -D feature-name
```

Merging Branches

To incorporate changes from one branch into another:

```
# First, switch to the target branch (usually main or master)
git checkout main

# Merge the feature branch into the current branch
```

```
git merge feature-name

# If there are conflicts, resolve them and then:
git add <resolved-files>
git commit -m "Merge feature-name into main"
```

Working with Remote Repositories

Adding a Remote Repository

To connect your local repository to a GitHub repository:

```
# Add a remote repository
git remote add origin https://github.com/username/repository.git

# View remote repositories
git remote -v
```

Pushing to a Remote Repository

To send your local changes to GitHub:

```
# Push the current branch to the remote repository
git push origin branch-name

# Push the current branch and set it to track the remote branch
git push -u origin branch-name

# Push all branches to the remote repository
git push --all origin
```

Fetching from a Remote Repository

To download changes from a remote repository without merging:

```
# Fetch changes from a remote repository
git fetch origin

# Fetch changes from all remotes
git fetch --all
```

Pulling from a Remote Repository

To download and merge changes from a remote repository:

```
 # Pull changes from the remote repository
git pull origin branch-name

# Pull changes from the tracked remote branch
git pull
```

Keeping Repositories in Sync

Regular Workflow to Keep in Sync

Follow this workflow to keep your local and remote repositories synchronized:

1. **Before starting work**: ```bash # Switch to the main branch git checkout main

Get the latest changes from the remote repository git pull origin main ```

1. **Create a feature branch**: `bash git checkout -b feature-name`

2. **Make changes and commit them**: ```bash # Make changes to files # Stage changes git add .

Commit changes git commit -m "Implement feature X" ```

1. **Stay updated with main branch**: ```bash # Switch to main branch git checkout main

Pull latest changes git pull origin main

Switch back to feature branch git checkout feature-name

Merge main into feature branch to resolve conflicts locally git merge main ```

1. **Push your feature branch to GitHub**: `bash git push -u origin feature-name`

2. **Create a Pull Request on GitHub** (through the web interface)

3. **After the Pull Request is merged**: ```bash # Switch to main branch git checkout main

Pull the latest changes (including your merged PR) git pull origin main

Delete the local feature branch git branch -d feature-name

Delete the remote feature branch (optional) git push origin --delete feature-name ```

Updating a Forked Repository

If you're working with a forked repository, keep it in sync with the original:

```
 # Add the original repository as a remote (commonly named "upstream")
git remote add upstream https://github.com/original-owner/original-repository.git

# Fetch changes from the upstream repository
git fetch upstream

# Switch to your main branch
git checkout main

# Merge changes from upstream's main branch
git merge upstream/main

# Push the updated main branch to your fork
git push origin main
```

Resolving Common Issues

Undoing Changes

```
 # Discard changes in working directory for a specific file
git checkout -- filename.txt

# Discard all unstaged changes
git checkout -- .

# Unstage a file (keep the changes in working directory)
git restore --staged filename.txt

# Undo the last commit but keep the changes staged
git reset --soft HEAD~1

# Undo the last commit and unstage the changes
git reset HEAD~1

# Completely discard the last commit and all changes
git reset --hard HEAD~1
```

Resolving Merge Conflicts

When Git can't automatically merge changes, you'll need to resolve conflicts manually:

1. Git will mark the conflicts in the affected files
2. Open the files and look for conflict markers (`<<<<<<<` , `=======` , `>>>>>>>`)
3. Edit the files to resolve the conflicts
4. Stage the resolved files with `git add`
5. Complete the merge with `git commit`

Example of a conflict in a file:

```
<<<<<<< HEAD
This is the change from the current branch
=======
This is the change from the branch being merged
>>>>>>> feature-branch
```

Stashing Changes

Temporarily store changes when you need to switch branches:

```
# Stash your changes
git stash

# List all stashes
git stash list

# Apply the most recent stash and keep it in the stash list
git stash apply

# Apply the most recent stash and remove it from the stash list
git stash pop

# Apply a specific stash
git stash apply stash@{2}

# Create a branch from a stash
git stash branch new-branch-name stash@{0}

# Remove all stashes
git stash clear
```

Advanced Git Commands

Git Rebase

Rebase is an alternative to merging that can create a cleaner project history:

```
 # Rebase the current branch onto main
git checkout feature-branch
git rebase main

# Interactive rebase for the last 3 commits
git rebase -i HEAD~3
```

Git Tags

Tags are used to mark specific points in history, typically for releases:

```
 # List all tags
git tag

# Create a lightweight tag
git tag v1.0.0

# Create an annotated tag
git tag -a v1.0.0 -m "Version 1.0.0"

# Push tags to remote
git push origin --tags

# Push a specific tag
git push origin v1.0.0
```

Git Aliases

Create shortcuts for common commands:

```
 # Create an alias for git status
git config --global alias.st status

# Create an alias for a complex log command
git config --global alias.lg "log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"
```

Git Hooks

Automate actions at certain points in the Git workflow by using hooks in the `.git/hooks` directory.

This guide covers the essential Git and GitHub commands to help you maintain synchronized repositories. Remember that Git is a powerful tool with many more features than covered here. As you become more comfortable with these basics, you can explore more advanced functionality to enhance your workflow.