Git Feature Branch Workflow

The feature branch workflow is a standard practice that keeps your `main` branch clean and stable. It lets you work on new features in an isolated environment without affecting the main codebase. Once a feature is complete and tested, it's merged back into `main`.

The Git commands are the same whether you're using Windows Shell (Command Prompt, PowerShell) or a Bash shell.

---

1. Start from `main`

Before you do anything, you need to make sure your local `main` branch is up-to-date with the remote repository (like GitHub or Azure DevOps).

CODE_BLOCK

Switch to your main branch

git switch main

Pull the latest changes from the remote server

git pull **CODE_BLOCK**

---

2. Create and Switch to a Feature Branch

Now, you'll create a new branch for your feature. Branch names should be short and descriptive, like `login-form` or `user-profile-page`.

This command creates a **new branch** and **immediately switches** to it.

CODE_BLOCK

The -c flag stands for "create"

git switch -c new-feature-name **CODE_BLOCK**

You can now work safely on this branch. Think of it as a separate copy of the project where your changes won't affect anyone else until you're ready.

---

3. Develop the Feature: Add and Commit

This is where you'll do your work—writing code, adding files, and fixing bugs. As you complete small, logical chunks of work, you should **commit** them. A commit is like a permanent save point.

The process for each commit is the same:

1. **Stage** your changes (`git add`).
2. **Commit** them with a clear message (`git commit`).

CODE_BLOCK

Stage a specific file

git add path/to/your/file.js

Or stage all changed files in the project

git add .

Commit the staged files with a descriptive message

git commit -m "feat: Add user login form component" **CODE_BLOCK**

You can (and should) have many commits on your feature branch. Committing often creates a clear history of your work and makes it easier to undo changes if something goes wrong.

---

4. Keep Your Branch Synced (Optional but Recommended)

If you're working on a feature for a while, the `main` branch might get updated by your teammates. It's a good practice to pull those updates into your feature branch. This makes the final merge much easier.

CODE_BLOCK

Fetch the latest changes from all remote branches

git fetch origin

Re-apply your commits on top of the latest main branch

git rebase origin/main **CODE_BLOCK**

The **`rebase`** command essentially "unplugs" your branch's changes, updates the base to the latest version of `main`, and then "re-plugs" your changes on top. This keeps your project history clean and linear.

---

5. Merge Your Feature into **`main`**

Once your feature is complete, tested, and ready to go, it's time to merge it back into the `main` branch.

CODE_BLOCK

1. First, go back to the main branch

git switch main

2. Make sure it's up-to-date one last time

git pull

3. Merge your feature branch into main

git merge --no-ff new-feature-name **CODE_BLOCK**

Using **`--no-ff`** (no fast-forward) is a crucial best practice. It creates a "merge commit" that ties the history of your feature branch together. This makes it very easy to see when a specific feature was merged into `main` and which commits belonged to it.

---

6. Push and Clean Up

Your `main` branch now has the new feature, but only on your local machine. You need to push it to the remote server. After that, you can delete the feature branch, since its work is now part of `main`.

CODE_BLOCK

1. Push the updated main branch to the remote

git push origin main

2. Delete the local feature branch

git branch -d new-feature-name

3. Delete the remote feature branch

git push origin --delete new-feature-name **CODE_BLOCK**

That's the complete lifecycle! □□ You've successfully created a feature, developed it in isolation, and safely merged it into the main project.