Git Feature Branch Workflow

Version: 0.71.0-Beta Date: 2025-09-10

--- Revision History ---

[0.71.0-Beta] - 2025-09-10

Added

- Added a subsection to Section 9 explaining how to remove stale

remote-tracking branches using `git fetch --prune`.

[0.56.28-Beta] - 2025-09-01

Added

- Added a subsection to Section 9 to explain the `git branch -a` command.

[0.40.11-Beta] - 2025-08-24

Added

- Added a new subsection to "Managing Files" explaining how to view a

file's history and diffs using `git log`, `git show`, and `git diff`.

[0.40.8-Beta] - 2025-08-24

Added

- Added a subsection to "Managing Files" explaining `git restore`.

- Added Section 13 ("Cleaning Up Commits") to cover `git commit --amend`

and interactive rebase (`git rebase -i`).

- Added Section 14 ("Tagging a Release") to describe the release

tagging process.

Changed

- Enhanced Section 11 to provide a more detailed explanation of how to

handle rebase conflicts, including the use of `git`

The feature branch workflow is a standard practice that keeps your `master` 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 `master`. The Git commands are the same whether you're using Windows Shell (Command Prompt, PowerShell) or a Bash shell.

---

1. Start from `master`

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

```
# Switch to your master branch
git switch master

# Pull the latest changes from the remote server
git pull
```

---

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.

```
# The -c flag stands for "create"
git switch -c new-feature-name
```

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. The process for each commit is the same:

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

```
# 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"
```

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 `master` branch might get updated by your teammates. It's a good practice to pull those updates into your feature branch.

```
# Fetch the latest changes from all remote branches
git fetch origin

# Re-apply your commits on top of the latest master branch
git rebase origin/master
```

The **`rebase`** command keeps your project history clean and linear.

---

5. Merge Your Feature into `master`

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

```
# 1. First, go back to the master branch
git switch master

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

# 3. Merge your feature branch into master
git merge --no-ff new-feature-name
```

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.

---

6. Push and Clean Up

Your `master` 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 `master`.

```
# 1. Push the updated master branch to the remote
git push origin master

# 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
```

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

7. Correcting Mistakes (`git revert`)

The safest way to undo a commit that has been shared is `git revert`. This command creates a *new commit* that is the exact inverse of the commit you want to undo.

```
# Find the hash of the commit you want to undo (e.g., from `git log`)
# Let's say the bad commit hash is `a1b2c3d4`

# Create a new commit that undoes the changes from the bad commit
git revert a1b2c3d4
```

---

8. Managing Files

Ignoring Untracked Files (`.gitignore`)

The best way to handle files that should *never* be in the repository (like build artifacts or log files) is to use a `.gitignore` file.

Removing Tracked Files (`git rm`)

If you have already committed a file that you now want to delete, you must use `git rm`. This command removes the file from both your working directory and Git's tracking index.

```
# Remove a file that is already tracked by Git
git rm path/to/unwanted-file.txt

# Commit the deletion
git commit -m "fix: Remove obsolete file"
```

Undoing Local Changes (`git restore`)

If you have made changes to a file in your working directory that you have **not yet staged** (`git add`), you can completely discard them and reset the file to its last committed state using `git restore`.

```
# Discard all recent, un-staged changes to a single file
git restore path/to/your/file.js
```

Warning: This is a destructive operation. Once you restore the file, your local changes are gone permanently.

Viewing File History and Changes

To inspect how a specific file has changed over time, you can use variations of `git log` and `git show`.

- **See the commit history for one file:**

```
git log -- path/to/your/file.ext
```

- **See the history with the actual changes (diffs) for each commit:**

```
git log -p -- path/to/your/file.ext
```

- **See the changes from a single, specific commit:**

```
# Get the commit hash from the log first
git show <commit-hash> -- path/to/your/file.ext
```

- **See your current uncommitted changes:**

```
git diff -- path/to/your/file.ext
```

---

9. Visualizing the History (`git log`)

To see the results of your branching and merging, you can use `git log` with a few flags to create a clean, graphical view.

```
git log --graph --oneline --all
```

- `--graph`: Draws an ASCII graph showing the branch structure.
- `--oneline`: Condenses each commit to a single line for readability.
- `--all`: Shows the history of all branches. This is so useful that many developers create a global Git alias for it, like `git lg`.

Listing Branches

To see a simple list of all local and remote-tracking branches without the commit history, use `git branch -a`.

```
git branch -a
```

- `-a`: Shows all branches, including those that only exist on the remote repository.

Pruning Stale Branches

If `git branch -a` shows a remote branch that you know has been deleted from the server, your local repository has a "stale" remote-tracking branch. You can clean this up by synchronizing with the remote using the `--prune` option.

```
# Fetch the latest changes and remove any stale remote-tracking branches
git fetch --prune
```

After running this command, the stale branch will no longer appear in your `git branch -a` list.

10. Pushing a Feature Branch

Before you merge, you often need to push your feature branch to the remote repository for backup, collaboration, or to create a pull request.

```
# The -u flag sets the remote branch as the "upstream" tracking branch
git push -u origin new-feature-name
```

After running this once, you can simply use `git push` from that branch in the future.

11. Handling Merge & Rebase Conflicts

A conflict occurs when changes in the `master` branch and your feature branch affect the same lines in the same file. Git will pause the operation and ask you to resolve the conflict manually.

- **Merge Conflict:** A `git merge` conflict happens all at once. You resolve all conflicting files, make a single new commit, and the merge is done.
- **Rebase Conflict:** A `git rebase` conflict is different. Git replays your commits one by one on top of the target branch. If a conflict occurs, the rebase pauses. You must resolve the conflict for that specific commit, and then continue. You may have to repeat this process for multiple commits. The resolution process is similar for both:

1. Git will stop and tell you which files have conflicts.
2. Open the conflicting file. You will see markers like:
   ```
   <<<<<<< HEAD
   // Code from the master branch
   =======
   // Code from your new-feature-name branch
   >>>>>>> new-feature-name
   ```

3. **Edit the file manually.** Remove the conflict markers and edit the code until it is correct, keeping the changes you need from both branches.
4. **Stage the resolved file:** `git add path/to/resolved/file.js`
5. **Finish the operation:**
   - If you were merging: `git commit`
   - If you were rebasing: `git rebase --continue`
   - **If a rebase goes wrong, you can always cancel it safely with `git rebase --abort`**. This will return you to the state before the rebase began.

---

12. Temporarily Saving Changes (`git stash`)

If you have uncommitted changes but need to switch branches immediately, use `git stash`.

```
# 1. Save your uncommitted changes to a "stash"
git stash

# 2. Now your working directory is clean. You can switch branches.
git switch master

# 3. When you return to your feature branch, re-apply the changes.
git switch new-feature-name
git stash pop
```

- `git stash pop` applies the most recent stash and removes it from your stash list.
- `git stash list` shows all of your saved stashes.

---

13. Cleaning Up Commits

Before you merge your feature branch, it's good practice to clean up your commit history.

Amending the Last Commit (`git commit --amend`)

If you just made a commit and realized you had a typo in the message or forgot to include a file, you can easily fix it with `--amend`.

```
# Stage the forgotten file
git add path/to/forgotten-file.js

# Amend the previous commit to include the new file and/or change the message
git commit --amend
```

This will open your editor to let you change the commit message. The new changes will be added to the same commit.

Interactive Rebase (`git rebase -i`)

Interactive rebase is a powerful tool for rewriting history. You can use it to **squash** multiple small commits into a single larger one, **reword** commit messages, and **reorder** commits.

```
# Rebase interactively against the master branch
git rebase -i origin/master
```

This command opens an editor with a list of your branch's commits. You can change the command next to each commit (e.g., from `pick` to `squash` or `reword`) to clean up your history before it's merged.

14. Tagging a Release

When you are ready to create a release, you use tags to mark a specific point in history.

```
# 1. Switch to the master branch and make sure it's up to date
git switch master
git pull

# 2. Create an annotated tag for the new version
# The -a flag creates an annotated tag (recommended)
# The -m flag provides the tag message
git tag -a v0.41.0-Beta -m "Release of version 0.41.0-Beta"

# 3. Push the tag to the remote repository
git push origin v0.41.0-Beta
```

This creates a permanent marker for the release that everyone on the team can access.