Git Feature Branch Workflow

Version: 0.38.0-Beta Date: 2025-08-20

--- Revision History ---

[0.38.0-Beta] - 2025-08-20

Added

- Added Section 9 to explain how to visualize history with `git log --graph`.

- Added Section 10 to cover pushing a feature branch to the remote.

- Added Section 11 for a basic guide on resolving merge/rebase conflicts.

- Added Section 12 to explain how to use `git stash`.

[0.37.0-Beta] - 2025-08-18

Added

- Initial versioning of the document to align with project standards.

- Added Section 7 to explain `git revert` for correcting mistakes.

- Added Section 8 to explain file management with `.gitignore` and `git rm`.

[cite_start]The feature branch workflow is a standard practice that keeps your `master` branch clean and stable. [cite: 1653] [cite_start]It lets you work on new features in an isolated environment without affecting the main codebase. [cite: 1654] [cite_start]Once a feature is complete and tested, it's merged back into `master`. [cite: 1655] [cite_start]The Git commands are the same whether you're using Windows Shell (Command Prompt, PowerShell) or a Bash shell. [cite: 1656]

---

1. Start from `master`

[cite_start]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). [cite: 1657]

```
# 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

[cite_start]Now, you'll create a new branch for your feature. [cite: 1658] [cite_start]Branch names should be short and descriptive, like `login-form` or `user-profile-page`. [cite: 1659] [cite_start]This command creates a **new branch** and **immediately switches** to it. [cite: 1660]

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

[cite_start]*You can now work safely on this branch. [cite: 1661] [cite_start]Think of it as a separate copy of the project where your changes won't affect anyone else until you're ready.* [cite: 1662]

---

3. Develop the Feature: Add and Commit

[cite_start]This is where you'll do your work—writing code, adding files, and fixing bugs. [cite: 1663] [cite_start]As you complete small, logical chunks of work, you should **commit** them. [cite: 1664] [cite_start]The process for each commit is the same: [cite: 1664]

1. [cite_start]**Stage** your changes (`git add`). [cite: 1665]
2. [cite_start]**Commit** them with a clear message (`git commit`). [cite: 1666]

```
# 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"
```

[cite_start]You can (and should) have many commits on your feature branch. [cite: 1668] [cite_start]Committing often creates a clear history of your work and makes it easier to undo changes if something goes wrong. [cite: 1668]

---

4. Keep Your Branch Synced (Optional but Recommended)

[cite_start]If you're working on a feature for a while, the `master` branch might get updated by your teammates. [cite: 1669] [cite_start]It's a good practice to pull those updates into your feature branch. [cite: 1670]

```
# 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
```

[cite_start]The **`rebase`** command keeps your project history clean and linear. [cite: 1672]

---

5. Merge Your Feature into `master`

[cite_start]Once your feature is complete, tested, and ready to go, it's time to merge it back into the `master` branch. [cite: 1672]

```
# 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
```

[cite_start]Using **`--no-ff`** (no fast-forward) is a crucial best practice. [cite: 1673] [cite_start]It creates a "merge commit" that ties the history of your feature branch together. [cite: 1674]

---

6. Push and Clean Up

[cite_start]Your `master` branch now has the new feature, but only on your local machine. [cite: 1676] [cite_start]You need to push it to the remote server. [cite: 1677] [cite_start]After that, you can delete the feature branch, since its work is now part of `master`. [cite: 1677]

```
# 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! [cite_start]□□ You've successfully created a feature, developed it in isolation, and safely merged it into the main project. [cite: 1679]

7. Correcting Mistakes (`git revert`)

[cite_start]The safest way to undo a commit that has been shared is `git revert`. [cite: 1681] [cite_start]This command creates a *new commit* that is the exact inverse of the commit you want to undo. [cite: 1682]

```
# 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 (`.gitignore` and `git rm`)

Ignoring Untracked Files (`.gitignore`)

[cite_start]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. [cite: 1684]

Removing Tracked Files (`git rm`)

[cite_start]If you have already committed a file that you now want to delete, you must use `git rm`. [cite: 1687] [cite_start]This command removes the file from both your working directory and Git's tracking index. [cite: 1688]

```
# 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"
```

---

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`.

---

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 Conflicts

If `git merge` or `git rebase` fails, it's likely due to a conflict. This happens when changes in the `master` branch and your feature branch affect the same lines in the same file.

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 rebasing: `git rebase --continue`
   - If you were merging: `git commit`

---

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.