Comprehensive Guide to Working with Git in a Team

This guide will walk you through the best practices for working with Git in a team, covering everything from setting up your project to managing branches, committing changes, and collaborating effectively. By following this guide, your team will avoid common pitfalls and ensure smooth collaboration.

Table of Contents:

---

1. Setting Up Your Local Git Repository

Before you start collaborating on a project, you need to set up Git on your local machine.

Steps for initial setup:

1. **Install Git:** If you haven't already installed Git, download and install it from git-scm.com (https://git-scm.com/).

2. **Configure Git (Only once per computer):** Set up your username and email for Git, which will be used to label your commits.

```
git config --global user.name "Your Name"
git config --global user.email "your-email@example.com"
```

3. **Clone the Repository:** Once Git is installed and configured, clone the remote repository to your local machine:

```
git clone https://github.com/your-team/repository-name.git
cd repository-name
```

---

2. Understanding Branching and Its Importance

In a team environment, working with branches is essential for managing features, bug fixes, and experiments without affecting the main codebase.

Main Branch (Master / Main):

The main branch is where the stable version of your project resides. Only completed and tested features should be merged into this branch.

Feature Branches:

Feature branches are used to work on individual features or tasks. They are branched off the main branch and merged back when the feature is complete and tested.

---

3. Starting Work on a New Feature (Creating and Working on a Branch)

When you start working on a new feature or task, you should create a new branch. This isolates your work from the main branch and allows you to collaborate effectively.

Steps for creating and switching to a new branch:

1. **Ensure you're up to date with the remote main branch:** Before starting any new work, make sure your local `main` branch is up to date with the remote repository.

   ```
   git checkout main
   git pull origin main
   ```

2. **Create a new branch:** Create a new branch for your feature. Name your branch descriptively, usually related to the task or feature you are working on.

   ```
   git checkout -b feature-name
   ```

3. **Start working on your feature:** Edit your code and files as needed for the new feature.

---

4. Making Changes and Committing Them

After making some changes to your code, you need to commit those changes to your local branch.

Steps for committing your changes:

1. **Check the status of your repository:** This shows which files have been modified, added, or deleted.

```
git status
```

2. **Add your changes to staging:** Stage the changes you want to commit. This can be specific files or all files.

```
git add file1.js file2.css  # Add specific files
git add .  # Add all modified files
```

3. **Commit your changes:** After staging your changes, commit them with a meaningful commit message that explains the change.

```
git commit -m "Implemented new login feature"
```

---

5. Pulling Updates from the Remote Repository

When collaborating with others, your teammates might make changes that you need to pull into your local branch to avoid conflicts and stay up-to-date with the latest work.

Steps to pull updates from the remote repository:

1. **Switch to your local main branch:** Before pulling updates, make sure you're on the `main` branch.

```
git checkout main
```

2. **Pull the latest changes:** Fetch and merge the latest changes from the remote `main` branch.

```
git pull origin main
```

- The above sequence does **not** bring in the updates to your feature branch. Instead, you're updating your `main` branch with the latest changes from the remote repository. This is useful for ensuring your `main` is up to date, but **it doesn't bring changes to your feature branch.**

3. **Merge updates into your feature branch (optional):** If your branch was created a while ago, you might want to merge the latest changes from `main` into your feature branch to stay up to date.

```
git checkout feature-name
git merge main
```

   - The first command (`git checkout feature-name`) switches you to the `feature-name` branch (where you are working).
   - The second command (`git merge main`) will merge the latest changes from the `main` branch into your current `feature-name` branch.

- **Will the updates from the repo be merged into my feature?** Yes! The changes from `main` will be merged into your current branch (`feature-name`). However, this means that if there are any conflicts between your feature work and the updates in the `main` branch, you'll need to resolve those conflicts.

- **Can my feature work be overridden?** No, your feature work won't be overridden. If there are conflicts, Git will mark the areas of conflict, and you'll have to manually decide how to resolve them. But your changes will remain intact in your branch.

---

6. Pushing Your Changes to the Remote Repository

Once your work is complete and committed locally, push your changes to the remote repository so others can access your work.

Steps to push your changes:

1. **Push your branch to the remote repository:** Push the feature branch to the remote server.

```
git push origin feature-name
```

BEHAVE YOURSELF metaphor:

In your "room" (your feature branch), you are allowed to commit and push changes, and occasionally merge updates from the main living room. In the "house living" (main branch), you only pull to stay updated with what others are doing and make sure you're aware of all changes. You're not creating untested new things here.

---

7. Reviewing and Merging Pull Requests

When your feature is complete and ready to be merged into the main branch, you need to open a pull request (PR). A pull request is a request for the team to review and merge your changes into the main codebase.

Steps for creating a Pull Request:

1. **Push your branch to the remote repository:** (As explained above)

2. **Open a Pull Request:** Go to the repository’s page on GitHub (or your hosting platform), and you will usually see a prompt to create a pull request for the branch you just pushed. Create a PR, add a description, and request reviewers.

    - **What is a PR?** A **Pull Request (PR)** is a request for your teammates to review your changes before merging them into the main branch. It is essentially a way to "ask" others to review and approve the code before it becomes part of the main project.

    - **Are there commands to create PRs?** Creating a pull request **cannot** be done directly with Git commands. This action is done through your Git platform's interface (e.g., GitHub, GitLab, etc.). Once your feature branch is pushed, you will go to your repository’s page and manually create the PR.

    - **Do you need to ask others to pull your work?** Yes! When you open a PR, you are asking your team to review your changes. This is a manual process where you need to initiate the PR on the Git platform (like GitHub or GitLab), and then the team members can review it.

3. **Get the PR reviewed:** Team members review the PR, leave comments, and suggest changes. Make the requested changes, commit them, and push them back to your branch.

    - **What if they ignore me?** If your team doesn't review your PR in a timely manner, you may need to follow up or ask them directly. However, Git itself doesn’t enforce review processes. You can’t force them to review, but good communication and proper etiquette in a team are crucial.

    - **If I was told to make changes after a review, should I keep the PR open?** Yes, keep the PR open! Once a pull request is open, you can continue working on your feature. If your team asks you to make changes:

    1. Make the necessary changes in your local feature branch.
    2. **Commit** the changes.
    3. **Push** the changes to your remote branch:

    ```
    git push origin feature-name
    ```

    These changes will automatically update the existing PR without needing to open a new one.

4. **Merge the Pull Request:** After approval, the PR can be merged into the `main` branch by the reviewer or the person responsible for merging. This is usually done by clicking the "Merge" button on GitHub or other hosting platforms.

   - **Who will merge the PR?** Typically, a teammate (often a lead or senior developer) reviews and merges the PR. **Only the person with the necessary permissions** can perform the merge, usually after the PR is reviewed and approved.
   - **Can I merge my own PR? No**, you should not merge your own PR until it has been reviewed. This avoids conflicts of interest and ensures quality control. However, some teams allow the author to merge their PR after approval.
   - **What commands exist for merging a PR?** You typically **cannot merge a PR using Git commands directly**. Merging is done through the platform's interface (GitHub, GitLab, etc.). However, if you have the right permissions, you can merge it through the platform's UI.

   DO NOT GET CONFUSED - The **merge command in Git** is usually used when you are merging branches locally:

   ```
   git merge feature-name
   ```

   This is done when you're working on your local machine, not for merging PRs.

---

8. Resolving Merge Conflicts

Sometimes, when merging branches, Git may not be able to automatically combine changes and will flag a **merge conflict**. These need to be resolved manually.

Steps to resolve merge conflicts:

1. **Identify conflicting files:** When you pull or merge, Git will show which files have conflicts.

What is the context of merging something myself?

You generally **only need to merge** when you are working with branches locally. In collaborative scenarios, **pulling updates** (`git pull`) is often enough. However, sometimes you may need to manually merge branches if you're working directly with different branches.

Why would you need to merge manually?

- If you are not working with a PR and are merging branches locally. This is a common case when you're managing multiple feature branches and want to combine them manually before pushing to the remote.

  Example:

```
git checkout feature-name
git merge another-branch
```

This command manually merges the `another-branch` into your `feature-name` branch.

---

2. **Open and edit the conflicting files:** Conflicting areas will be marked in the file with special markers:

```
<<<<<<< HEAD
// Your changes
=======
// Changes from the branch you're merging
>>>>>>> feature-name
```

3. **Resolve the conflict:** Decide which version of the code to keep, or combine the changes manually. After resolving, remove the conflict markers.

4. **Mark the conflict as resolved:** After resolving conflicts, add the resolved files and commit the changes.

```
git add resolved-file.js
git commit -m "Resolved merge conflict"
```

5. **Finish the merge:** Once all conflicts are resolved, finalize the merge.

```
git merge --continue
```

The `git merge --continue` command is used after resolving merge conflicts. When you merge two branches and encounter conflicts, Git stops the process and asks you to resolve those conflicts. If there are multiple conflicts in different files, you would need to resolve each one. Once resolved, you use `git merge --continue` to finalize the merge.

---

9. ADVANCED - Rebasing vs. Merging

Rebasing and merging are two ways to integrate changes from one branch into another. Rebasing is a cleaner history but can be dangerous if used incorrectly.

Rebasing:

1. **Rebase onto the latest main:** If you want to keep your branch history linear, you can rebase your feature branch onto the latest `main`.

```
git checkout feature-name
git fetch origin
git rebase origin/main
```

2. **Resolve conflicts if any arise:** If there are conflicts, resolve them as explained earlier.

3. **Force-push the rebased branch:** After rebasing, you may need to force-push to overwrite the remote branch.

```
git push --force origin feature-name
```

What is the difference between `fetch` and `pull`?

- **What does `git fetch` do?** The `git fetch` command is used to **download** the latest changes from the remote repository **without merging** them into your local branch. This command updates your remote-tracking branches, such as `origin/main`, with the latest commits from the remote repository.

```
git fetch origin
```

  Why use `origin` and not just `main`?

  - `origin` refers to the **remote repository**, while `main` refers to a **branch** within the repository.
  - Using `git fetch origin` updates your tracking branches (like `origin/main`) with the latest changes without merging them into your current working branch. This allows you to review what has changed before merging.

- **What is `git pull`?** `git pull` is a combination of `git fetch` followed by `git merge`. It downloads changes from the remote repository and immediately merges them into your local branch.

```
git pull origin main
```

Merging (Recommended for most teams):

Merging retains the commit history of both branches and is simpler and safer for beginners.

---

10. Reverting Changes

If you made a mistake in a commit or need to undo some changes, you can revert to a previous commit.

Steps to revert changes:

1. **Revert the last commit:** To undo your last commit but keep the changes in your working directory:

   ```
   git revert HEAD
   ```

2. **Revert a specific commit:** If you want to revert to a specific point in time, you could use the commit hash of the point you want to go back to.

   ```
   git revert <commit-hash>
   ```

3. **Commit the revert:** Git will create a new commit that undoes the changes.

What does "Reverting a Commit" mean?

- **What is `git revert`?** `git revert` is used to **create a new commit that undoes** the changes of a previous commit.
  - When you **revert a commit**, you don't erase the commit from history. Instead, Git generates a new commit that **reverses the changes** made by a specific commit.
  - For example, if you want to undo the most recent commit, you would do:

    ```
    git revert HEAD
    ```

  - **Is it canceling my last commit?** Yes! But rather than deleting it, it creates a new commit that **undoes the changes**. This is a safer alternative because it maintains history and avoids rewriting Git history.

---

11. Best Practices and Collaboration Tips

Tips for Collaboration:

1. **Keep commits small and focused:** Each commit should do one thing. This makes it easier to track changes and understand history.

2. **Use descriptive commit messages:** A good commit message should briefly explain what the change does and why it was made.

3. **Pull frequently from the main branch:** Before starting new work, pull updates from `main` to ensure you're working on the latest code, and avoid large merge conflicts.

4. **Avoid force-pushing to shared branches:** Only use force-push (`git push --force`) on your personal feature branches. Never force-push to `main`.

This guide will help your team establish a robust workflow with Git, minimizing conflicts and improving collaboration.