Git & GitHub Basics

Dineshkumar Thangavel
Corporate Web Development Trainer

Git Basics

Configuring Git

- **Set the global author name and email:**

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

- **View all configuration settings:**

```
git config list
```

Working with a Git Repository

1. Initializing a Repository

- **Create an empty Git repository in the current directory:**

```
git init
```

2. Checking Repository Status

- **Display the status of the repository (tracked, untracked, staged, or modified files):**

```
git status
```

3. Staging Changes

- **Stage all untracked (U) and modified (M) files:**

```
git add .
```

4. Committing Changes

- **Move staged changes to the Git repository with a message:**

```
git commit -m "Your commit message"
```

5. Viewing Commit History

- **Display a concise commit history:**

```
git log --oneline
```

6. Reverting Changes

- **Revert the repository to a specific commit (using the hash code):**

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

GitHub Basics

Creating and Linking a GitHub Repository

1. **Create a GitHub Account**
   - Sign up at GitHub if you don't already have an account.
2. **Create a New Repository**
   - Provide a unique project name in the repository creation form.
3. **Link the GitHub Repository to Git:**

```
git branch -M main  # Rename the branch to 'main' (optional)
git remote add origin <github-link>  # Link the remote repository
git push -u origin main  # Push changes and set upstream (-u for upstream)
```

Syncing Local and Remote Repositories

Pushing Local Changes to GitHub

(Ahead: Local to Remote)

1. Stage changes:

```
git add .
```

2. Commit changes:

```
git commit -m "Your commit message"
```

3. Push to GitHub:

```
git push origin <branch-name>
```

Pulling Changes from GitHub

(Behind: Remote to Local)

- Sync the local repository with the remote repository:

```
git pull origin main
```

Working with an Existing GitHub Repository

Cloning a Repository

- Clone the main branch of a GitHub repository to your local system:

```
git clone <github-link>
```

Collaborating with Team Members

Adding Collaborators

1. Go to the repository on GitHub.
2. Navigate to Settings > Collaborators > Manage Access > Add People
3. Manage access by adding the GitHub usernames or email addresses of collaborators.

Branching

Git branching is a powerful feature that enables developers to work on multiple tasks, features, or bug fixes simultaneously without interfering with the main codebase. Here's a breakdown of the concept:

What is a Branch?

A branch in Git is essentially a pointer to a specific commit. It allows you to diverge from the main codebase (usually main or master) to develop new features or fix issues independently.

Why Use Branches?

1. **Parallel Development**: Multiple developers can work on different branches without conflicts.
2. **Experimentation**: You can try out new ideas without affecting the main codebase.
3. **Version Control**: Branches make it easy to manage versions of your project.

Common Branching Commands

1. **Create a New Branch**

```
git branch branch-name
```

2. **Switch to a Branch**

```
git checkout branch-name
```

(or, with newer Git versions)

```
git switch branch-name
```

3. **Create and Switch to a Branch**

```
git checkout -b branch-name
```

(or)

```
git switch -c branch-name
```

4. **List All Branches**

```
git branch
```

5. **Merge a Branch into the Current Branch**

```
git merge branch-name
```

6. **Delete a Branch**
    - Locally:

```
git branch -d branch-name
```

    - Force delete:

```
git branch -D branch-name
```

    - Remotely:

```
git push origin --delete branch-name
```

Types of Branches

1. **Main/Master Branch**: The main codebase for production-ready code.
2. **Feature Branches**: Used to develop specific features.
3. **Bug Fix Branches**: Created to address specific issues.
4. **Release Branches**: Prepared for releases.
5. **Hotfix Branches**: Used for quick fixes on production code.

Branching Workflow Example

1. **Create a Branch for a Feature**

```
git branch feature-login

git checkout feature-login
```

2. **Work on the Branch**

   - Make changes and commit:

```
git add .

git commit -m "Add login feature"
```

3. **Merge the Branch into Main**

   - Switch to main:

```
git checkout main
```

   - Merge the feature branch:

```
git merge feature-login
```

4. **Delete the Feature Branch**

```
git branch -d feature-login
```

Best Practices

1. **Keep Branches Short-lived**: Regularly merge completed work back to the main branch.
2. **Use Clear Naming Conventions**:
    - feature/feature-name
    - bugfix/issue-number
3. **Avoid Direct Commits to Main**: Always create branches for any changes.
4. **Regularly Pull Changes from Main**: Keep your branch up-to-date to minimize merge conflicts.

Git branching used in a real-world project. Let's assume you're developing a web application with a team.

Scenario

You are working on a **blogging application**, and the team needs to:

1. Add a new feature for **user profiles**.
2. Fix a bug where images are not uploading correctly.
3. Prepare the application for the next release.

Step-by-Step Workflow

1. Clone the Repository

First, clone the repository from the remote (e.g., GitHub):

```
git clone https://github.com/username/blog-app.git
cd blog-app
```

2. Create a Branch for Each Task

Feature: User Profiles

Create and switch to a feature branch:

```
git checkout -b feature/user-profiles
```

Work on the code, then commit your changes:

```
git add .
git commit -m "Add user profiles feature"
```

Bugfix: Image Upload

Create a bugfix branch:

```
git checkout -b bugfix/image-upload
```

Fix the bug and commit your changes:

```
git add .
git commit -m "Fix image upload issue"
```

Release: v2.0

Create a release branch for version 2.0:

```
git checkout -b release/v2.0
```

Prepare for the release (update documentation, versioning, etc.), then commit:

```
git add .
git commit -m "Prepare for v2.0 release"
```

3. Test and Merge Branches

Step 1: Switch to main

```
git checkout main
```

Step 2: Merge Each Branch into main

Merge the Feature Branch:

```
git merge feature/user-profiles
```

Merge the Bugfix Branch:

```
git merge bugfix/image-upload
```

Merge the Release Branch:

```
git merge release/v2.0
```

Step 3: Push Changes to the Remote Repository

```
git push origin main
```

4. Delete Merged Branches

Once merged, delete the branches to keep the repository clean.

```
git branch -d feature/user-profiles
git branch -d bugfix/image-upload
git branch -d release/v2.0
```

Final Workflow Summary

- **Development** happens in separate branches for features, bug fixes, or releases.
- Once completed, changes are **tested** and **merged** into main.
- Branches are **deleted** after merging to maintain a clean Git history.

This approach ensures that your team can work on different tasks independently without conflicts.

Remote Workflow

In a **remote workflow**, the repository is hosted on a platform like GitHub, GitLab, or Bitbucket, and team members collaborate by cloning, branching, and pushing changes. Here's how it works:

1. Setup

Clone the Repository

Each team member clones the repository to their local machine:

```
git clone https://github.com/username/repo-name.git
cd repo-name
```

Check the Remote URL

Verify the remote repository link:

```
git remote -v
```

2. Create a Branch for Your Task

Create a branch for the specific feature, bugfix, or task:

```
git checkout -b feature/new-feature
```

Make your changes locally, then commit:

```
git add .
git commit -m "Implement new feature"
```

3. Push the Branch to the Remote Repository

Push your branch to the remote repository:

```
git push origin feature/new-feature
```

This command creates the branch on the remote and pushes your commits. Now the branch is accessible to your team.

4. Collaborate with the Team

Pull Changes from Main to Stay Updated

Switch to the main branch and pull the latest changes to keep your local repository updated:

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

Merge the latest changes into your feature branch to avoid conflicts:

```
git checkout feature/new-feature
git merge main
```

5. Create a Pull Request (PR)

Once your task is complete, create a **Pull Request** (or **Merge Request**) from your branch to main using the Git hosting platform.

- Go to your repository on GitHub (or other platforms).
- Click "Compare & Pull Request" for your branch.
- Add a title and description, then submit the PR.

6. Code Review

Other team members or reviewers:

- Review the code.
- Suggest changes (if necessary).
- Approve the PR.

If there are suggestions, you can make changes locally, commit, and push them to the same branch. The PR will automatically update.

7. Merge the Pull Request

After approval, merge the branch into `main`. This can be done via the platform's UI.

8. Delete the Branch

After merging, delete the branch both locally and remotely:

- Delete locally:

```
git branch -d feature/new-feature
```

- Delete remotely:

```
git push origin --delete feature/new-feature
```

Best Practices for Remote Workflows

1. **Use Descriptive Branch Names**:
    - feature/login
    - bugfix/image-upload
    - hotfix/critical-issue
2. **Pull Regularly**: Always pull the latest changes from `main` to reduce conflicts.
3. **Small Commits**: Make small, meaningful commits with clear messages.
4. **Review Pull Requests**: Ensure every PR is reviewed before merging.
5. **Resolve Conflicts Locally**: If there are merge conflicts, resolve them on your local machine.

Workflow Summary

1. Clone the repository and create a new branch for your task.
2. Make changes, commit, and push your branch to the remote.
3. Open a Pull Request and address feedback.
4. Merge the branch into `main` and delete it.

Team collaboration in Git

It revolves around using branches, pull requests, and best practices to work efficiently without interfering with others' work. Here's a guide for effective Git-based team collaboration:

Key Components of Team Collaboration

1. **Centralized Repository**: A remote repository (e.g., on GitHub, GitLab) acts as the single source of truth.
2. **Branching Model**: Every team member works in separate branches to avoid conflicts.
3. **Pull Requests (PRs)**: Used for reviewing and merging code changes.
4. **Code Reviews**: Ensure quality and consistency in the codebase.
5. **Continuous Integration/Continuous Deployment (CI/CD)**: Automate testing and deployment.

Steps for Team Collaboration

1. Repository Setup

- **Create the Repository**: Set up the project repository on a hosting platform (e.g., GitHub).
- **Add Collaborators**: Invite team members to the repository and assign appropriate permissions.
- **Define Branching Strategy**: Use a model like:
  - main (production-ready code)
  - develop (integration of all features before release)
  - feature/*, bugfix/*, hotfix/* for individual tasks.

2. Branch Workflow

Create Branches for Tasks

Each team member creates a branch for their task:

```
git checkout -b feature/task-name
```

Work on the Branch

Make changes, commit regularly, and write clear messages:

```
git add .
git commit -m "Implement task description"
```

Push the Branch

Push the branch to the remote repository:

```
git push origin feature/task-name
```

3. Sync with the Team

Pull Changes Regularly

Keep your branch up-to-date with the latest changes from main (or develop):

```
git checkout main
git pull origin main
git checkout feature/task-name
git merge main
```

Resolve Merge Conflicts

If conflicts occur during merging, resolve them in your local editor, then commit the changes:

```
git add .
git commit -m "Resolve merge conflicts"
```

4. Open a Pull Request

When your task is complete, create a **Pull Request (PR)** from your branch to the target branch (main or develop):

1. Go to your repository on the hosting platform.
2. Click **"New Pull Request"**.
3. Compare your branch with the target branch.
4. Add a title and description explaining your changes.
5. Submit the PR.

5. Code Review

Review Process

- **Reviewers**: Assigned team members review the PR, suggest changes, and ensure quality.
- **Author**: Responds to feedback, makes updates, and pushes them to the branch.

Approve and Merge

Once approved, the PR is merged into the target branch.

6. Clean-up

After merging, delete the branch:

- **Locally**:

```
git branch -d feature/task-name
```

- **Remotely**:

```
git push origin --delete feature/task-name
```

Best Practices for Team Collaboration

Branching Strategy

Use a clear branching model:

- **GitFlow**: Feature, release, and hotfix branches.
- **GitHub Flow**: Simple workflow with feature branches merged into main.

Commit Messages

Write meaningful commit messages:

- **Format**: <type>(<scope>): <description>
- Example:
- feat(auth): add login functionality
- fix(upload): resolve image upload error

Code Review Guidelines

- Focus on logic, readability, and adherence to standards.
- Avoid nitpicking; use automated linters for formatting issues.

Communication

- Use tools like Slack, Teams, or GitHub Issues to coordinate tasks.
- Write clear PR descriptions and link them to related issues or tasks.

Continuous Integration (CI)

Automate testing and linting for every PR using CI tools like GitHub Actions, Jenkins, or Travis CI.

Documentation

- Maintain a README.md for setup and usage instructions.
- Document team workflows, naming conventions, and coding standards.

Example Workflow for a Team

Scenario: Team of 3 working on a blog app

1. **Tasks**:
    - Dev A: Add user profile feature (feature/user-profiles).
    - Dev B: Fix image upload bug (bugfix/image-upload).
    - Dev C: Update styling (feature/update-styling).
2. **Workflow**:
    - Each dev creates their branch:
    - git checkout -b feature/task-name
    - Devs make changes, commit, and push to the remote.
    - Open PRs for their branches.
    - Team reviews and merges the changes.
    - Devs delete their branches after merging.