Git workflows

In this train, weâ€™ll focus on various Git workflows by exploring different strategies for collaborating and managing changes within a team.

Learning objectives

By the end of this train, you should be able to:

- Understand the key aspects of popular Git workflows, including feature branching and Gitflow.
- Evaluate the benefits of each Git workflow in different project environments.
- Apply effective strategies to resolve conflicts that arise during collaborative development using Git.

Outline

1. Git workflows

Git workflows are structured methodologies that guide how teams collaborate on software development projects. By defining specific processes for creating, merging, and managing branches within a Git repository, these workflows streamline and organise development activities. Understanding and implementing Git workflows is essential for effective team collaboration, ensuring code consistency, and maintaining project quality.

Two popular Git workflows are **feature branching** and **Gitflow**.

1.1 Feature branch workflow

The Feature branch workflow is centred around the idea that all feature development should take place in a dedicated branch instead of the main branch. This approach not only minimises the risk of destabilising the main branch but also ensures that new features are developed in isolation.

Steps:

1. **Creating a feature branch**:
   - Start by updating your local main branch (usually called `main` or `master`). Use `git checkout main` to switch to the main branch, then `git pull` to fetch the latest changes.
   - Create a new branch for your feature from the updated main branch with `git checkout -b feature-name`. This isolates your work from the main codebase.
2. **Developing the feature**:
   - Develop your feature on this new branch. Regularly commit your changes to track your progress. Use `git add .` to stage all changes, and `git commit -m "Descriptive message"` to commit.
   - To keep your feature branch updated with any changes made in the main branch, periodically merge changes from the main branch into your feature branch. Use `git merge main` while on your feature branch.
3. **Merging the feature into the main branch**
   - Before merging, update your main branch again with `git checkout main` and `git pull`.
   - Switch back to your feature branch using `git checkout feature-name` and merge the main branch into it to check if there are any conflicts. Resolve these conflicts if any.
   - Once the feature is complete and conflicts are resolved, switch to the main branch (`git checkout main`) and merge the feature branch into it with `git merge feature-name`.
   - Push your changes to the remote repository using `git push origin main`.
4. **Branch clean-up**:
   - After successfully merging your feature branch into the main branch and ensuring everything works correctly, you can delete the feature branch to keep your repository clean. Use `git branch -d feature-name` to delete it locally.
   - If you have pushed the feature branch to the remote repository, you can also delete it there using `git push origin --delete feature-name`.

Advantages:

- Offers a clean, organised way to manage new features.
- Makes reviewing and testing individual features more manageable.
- Prevents the main branch from becoming cluttered with incomplete features.

1.2. Gitflow workflow

The Gitflow workflow is a robust framework designed for managing larger projects. It introduces additional layers of branching to accommodate various stages of development, including new features, releases, and hotfixes.

Steps:

1. **Initial setup**:
   - Two primary branches are maintained at all times: `master`, which holds the official release history, and `development`, which serves as an integration branch for features.
   - Use the command `git branch development` to create the `development` branch, then push it to the remote server with `git push -u origin development`.
2. **Feature development**:
   - For each new feature, create a branch from the `development` branch. Use `git checkout -b feature-name development` to create and switch to the new branch.
   - Once work is complete, merge the feature back into `development`. First, switch to the `development` branch using `git checkout development`, then merge the feature branch with `git merge feature-name`.
   - Delete the feature branch if it's no longer needed, using `git branch -d feature-name`.
3. **Release preparation**: A release is essentially a set of features prepared for deployment. Releases can be either time-based, where they are scheduled at regular intervals, or milestone-based, where they are centered around the completion of specific goals or features.
   - When the features in `development` are ready for release, create a `release` branch from `development` using `git checkout -b release-1.0 development`.
   - In this `release` branch, you can make final tweaks, fix bugs, and prepare metadata for the release. This branch represents the next version of the product.
   - Once the `release` branch is ready to become the actual release, merge it into `master` and `development`. Merge into `master` using `git checkout master` and `git merge release-1.0`, and similarly for `development`. This ensures that new features are carried back into `development`. **Note:** If the release process is completed in one sitting, it's not necessary to merge the `release` branch back into the

development branch, as it will be an exact replica of the development branch at this point.

4. **Handling hotfixes**:
    - If a critical issue is found in `master`, create a `hotfix` branch directly from `master`. Use `git checkout -b hotfix-issue master`.
    - After fixing the issue, merge the `hotfix` branch back into both `master` and `development` to ensure that the fix is integrated into the ongoing work. Merge it into `master` using `git checkout master` and `git merge hotfix-issue`, and then into `development` using similar commands.

Advantages:

- Provides a structured, clear path for development and release, reducing confusion in large projects.
- Separates different stages of development, ensuring that the `master` branch always remains stable and release-ready.

2. Benefits of workflows in different scenarios

Small and agile projects

- **Feature branching**: Ideal for small teams where features are developed and deployed rapidly. It provides the flexibility required in an agile setting without overwhelming the main branch.

Large and complex projects

- **Gitflow**: Best suited for larger projects with defined release cycles. Its structured approach helps in managing multiple features and stages of development, ensuring a systematic workflow from development to deployment.

3. Resolving conflicts

Guide:

1. **Detecting conflicts**: Conflicts usually arise during a merge when the same lines of code have been changed in different branches.
2. **Opening conflicting files**: Git marks the areas of conflict in the file. Carefully review these sections to understand the differences.
3. **Resolving the conflict**:
    - Manually edit the file to integrate the changes from both branches. Decide which changes to keep, discard, or merge.
    - After editing, mark the conflict as resolved by staging the file with `git add <file-name>`.
4. **Completing the merge**: Finalise the merge by committing the changes. Ensure to provide a comprehensive commit message that outlines the conflict and how it was resolved.

4. Additional resources

Visit this link to explore the different Git workflows through an illustrative guide that uses visual diagrams to demonstrate each process: Comparing Git workflows