Git For Dummies

Keith Homan

April 18, 2024

Contents

1 Gitflow Workflow Overview

The Gitflow Workflow is a branching model designed for projects that require a structured release cycle. It involves maintaining multiple branches to manage features, releases, and hotfixes, ensuring a clear history and organization of development efforts.

1.1 Branches in Gitflow

- **Master**: Stores the official release history.
- **Develop**: Serves as an integration branch for features.
- **Feature**: Used to develop new features.
- **Release**: Supports preparation for a new production release.
- **Hotfix**: Used to quickly patch production releases.

2 Basic Git Commands

Before delving into the specifics of the Gitflow Workflow, it's important to understand some basic Git commands that are commonly used in this workflow.

2.1 Checking Status

```
# Check the current status of your working directory
git status
```

2.2 Branching and Adding Changes

```
# Ensure you are on the correct branch
git checkout <branch_name>

# To add specific files
git add <file1> <file2> ...

# To add all changes
git add .
```

2.3 Committing and Pushing Changes

```
# Commit your changes with a descriptive message
git commit -m "Descriptive␣message␣explaining␣the␣changes"

# Push your changes to the remote repository
git push origin <branch_name>
```

Note: Directly pushing to the master/main branch in shared repositories is typically discouraged.

3 Branching Strategy in Gitflow

The Gitflow Workflow defines a strict branching model designed around project releases. This section outlines the process of creating, rebasing, and finishing feature branches, which is a core part of the Gitflow Workflow.

3.1 Creating a Feature Branch

```
git checkout develop
git checkout -b feature/your_feature_name
```

3.2 Updating a Feature Branch (Rebasing)

Before merging a feature branch back into ‘develop‘, it’s good practice to rebase the feature branch onto the latest ‘develop‘ branch. This helps to ensure a linear history and easier integration.

```
git checkout feature/your_feature_name
git rebase develop
```

3.3 Finishing a Feature Branch

After rebasing, the feature branch can be merged into ‘develop‘. It’s often recommended to use ‘–no-ff‘ to maintain the history of the feature branch.

```
git checkout develop
git merge --no-ff feature/your_feature_name
```

4 Releasing

Releases are an important part of the Gitflow Workflow, marking the culmination of development efforts into a new version of the software.

4.1 Creating a Release Branch

```
git checkout develop
git checkout -b release/your_release
```

4.2 Preparing a Release (Rebasing)

Optionally, you might want to rebase the release branch onto the ‘main‘ to ensure that any hotfixes are incorporated.

```
git checkout release/your_release
git rebase main
```

4.3 Finishing a Release Branch

After final testing and preparations, the release branch is merged into ‘main‘ and ‘develop‘. The merge into ‘main‘ marks the new release.

```
git checkout main
git merge --no-ff release/your_release
git tag -a v1.0.0 -m "Release␣1.0.0"
git checkout develop
git merge --no-ff release/your_release
git branch -d release/your_release
```

5 Hotfixes

Hotfix branches are used to quickly patch live production releases. This is a critical part of managing unforeseen issues that need immediate attention.

5.1 Creating a Hotfix Branch

```
git checkout main
git checkout -b hotfix/your_hotfix
```

5.2 Finishing a Hotfix Branch

```
git checkout main
git merge --no-ff hotfix/your_hotfix
git tag -a v1.0.1 -m "Hotfix␣1.0.1"
git checkout develop
git merge --no-ff hotfix/your_hotfix
git branch -d hotfix/your_hotfix
```

6 Versioning and Tagging

Versioning is a critical aspect of software development, allowing teams to track and manage changes to the software over time. Semantic Versioning (SemVer) is a widely adopted convention for version numbering that conveys meaning about the underlying changes.

6.1 Semantic Versioning

Semantic Versioning uses a three-part version number: MAJOR.MINOR.PATCH (e.g., ‘v1.0.0‘), where:

- **MAJOR version** increments when there are incompatible API changes, signaling that users might need to make significant changes to their own code to upgrade.
- **MINOR version** increments when functionality is added in a backwards-compatible manner, indicating that new features or improvements have been added without breaking existing functionality.

- **PATCH version** increments when backwards-compatible bug fixes are introduced, meaning that errors or issues have been corrected without adding new features or changing existing ones.

The "v" prefix is a common convention indicating "version", though it's not strictly required by Semantic Versioning.

6.2 Tagging Releases

In Git, tags are used to mark specific points in the repository's history, typically for releases. To create an annotated tag for a release:

```
# Create an annotated tag for version 1.0.0
git tag -a v1.0.0 -m "Release␣1.0.0"
# Push the tag to the remote repository
git push origin v1.0.0
```

Using tags for versioning allows teams and users to easily check out specific versions of the software, facilitating better release management and compatibility tracking.