GIT Guidelines

Best Practices to follow

by Hue Learn, Wise Work, DNAi World

Contents

1. Introduction

Git, the powerhouse of version control, is a vital tool for developers. However, using Git effectively can be a daunting task. In this guide, we'll explore essential tips and best practices to help you navigate the Git universe like a pro, ensuring efficient and error-free code management.

There are a few key concepts:

1. **Repository**: This is where the code and its entire history are stored.
2. **Commit**: A commit is a record of changes made to the repository.
3. **Branch**: A branch is like a separate line of development, allowing you to work on features or fixes in isolation.
4. **Merge**: This is the process of integrating changes from one branch into another.
5. **Conflict**: Conflicts happen when changes from different branches contradict each other, requiring manual resolution.

Here's a quick start guide:

1. **Install Git**: Download and install it from git-scm.com.
2. **Create a Repository**: Initialize a new repository with git init or clone an existing one with git clone <repository-url>.
3. **Make Changes**: Add changes to the staging area with git add <file>, and commit them with git commit -m "commit message".
4. **Branching**: Create a new branch with git branch <branch-name> and switch to it with git checkout <branch-name>.
5. **Merging**: Merge changes from one branch to another with git merge <branch-name>.
6. **Pushing and Pulling**: Push changes to a remote repository with git push and pull changes from a remote repository with git pull.

Here are some basic Git commands:

Commands	Description
git notes add	Creates a new note and associates it with an object (commit, tag, etc.).
git restore <file>	Restores the file in the working directory to its state in the last commit.
git reset <commit>	Moves the branch pointer to a specified commit, resetting the staging area and the working directory to match the specified commit.
git reset –soft <commit>	Moves the branch pointer to a specified commit, preserving the changes in the staging area and the working directory.
git diff	Shows the changes between the working directory and the staging area (index).
git diff <commit1> <commit2>	Displays the differences between two commits.
git diff –staged or git diff –cached	Displays the changes between the staging area (index) and the last commit.
git add <file>	Adds a specific file to the staging area.
git add . or git add –all	Adds all modified and new files to the staging area.

Commands	Description
git status	Shows the current state of your repository, including tracked and untracked files, modified files, and branch information.
git status –ignored	Displays ignored files in addition to the regular status output.
git commit	Creates a new commit with the changes in the staging area and opens the default text editor for adding a commit message.
git commit -m “<message>” or git commit –message “<message>”	Creates a new commit with the changes in the staging area and specifies the commit message inline.
git commit -a or git commit –all	Commits all modified and deleted files in the repository without explicitly using git add to stage the changes.

2. Commit

1. Document the Purpose of Commits with Examples

Commit messages should be clear and descriptive. Use the imperative mood and be concise but specific about what has been changed and why.

Wrong Coding Style.

```
fix
```

Right Coding Style.

```
git commit -m "Fix login bug

Fix bug causing 404 error on login page
```

2. Commit Often, but with Purpose

Small, frequent commits make it easier to track changes and revert to previous states if needed.

3. Commit Message Format

Use the format: `[Type] Short description`

Right Coding Style.

```
`feat: add user profile page`
```

4. Revert Commits Carefully

`git revert` is safer for undoing changes in shared branches as it preserves the commit history.

Wrong Coding Style.

```
Using `git reset` to undo changes in a shared repository.
```

Right Coding Style.

Using `git revert` to create a new commit that undoes a previous commit

```
Revert a specific commit
git revert <commit-hash>
```

5. Document the Purpose of Commits with Examples.

Descriptive commit messages are as important as a change itself. Write detailed commit messages, provide context and help understand changes later.

Wrong Coding Style.

```
Generic commit messages like `update files`
```

Right Coding Style.

- Providing context and examples in commit messages.
- Commit message with detailed description

```
git commit -m "Fix login bug
```

- Issue: Null pointer exception
- Solution: Added null checks in login validation
- Example: User receives error message when submitting an empty form

2.1. Syncing Code

- Daily Sync the code by following below rules
  - Pull the latest code from the `dev` branch every morning.
  - Push your changes to your feature/bug branch every evening.
  - More frequent syncing is encouraged to minimize merge conflicts.

2.2. Pre-commit hooks

Pre-commit hooks can automatically enforce code quality rules before committing changes.

Wrong Coding Style.

```
Committing code without checking for linting errors or running tests
```

Right Coding Style.

Using pre-commit hooks to automate code quality checks

```
// Example .pre-commit-config.yaml
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v3.4.0
    hooks:
    -   id: trailing-whitespace
    -   id: end-of-file-fixer
```

2.3. Cherry-Pick commits

`git cherry-pick` allows you to apply specific commits from one branch to another without merging entire branches.

Right Coding Style.

- Cherry-picking specific commits from one branch to another

```
git checkout target-branch
git cherry-pick <commit-hash>
```

2.4. Commiting Large Files

Wrong Coding Style.

Committing large files directly to the repository

Right Coding Style.

Using Git Large File Storage (LFS) for large files..

```
git lfs track "*.psd"
```

2.5. Revert Commits

`git revert` is safer for undoing changes in shared branches as it preserves the commit history.

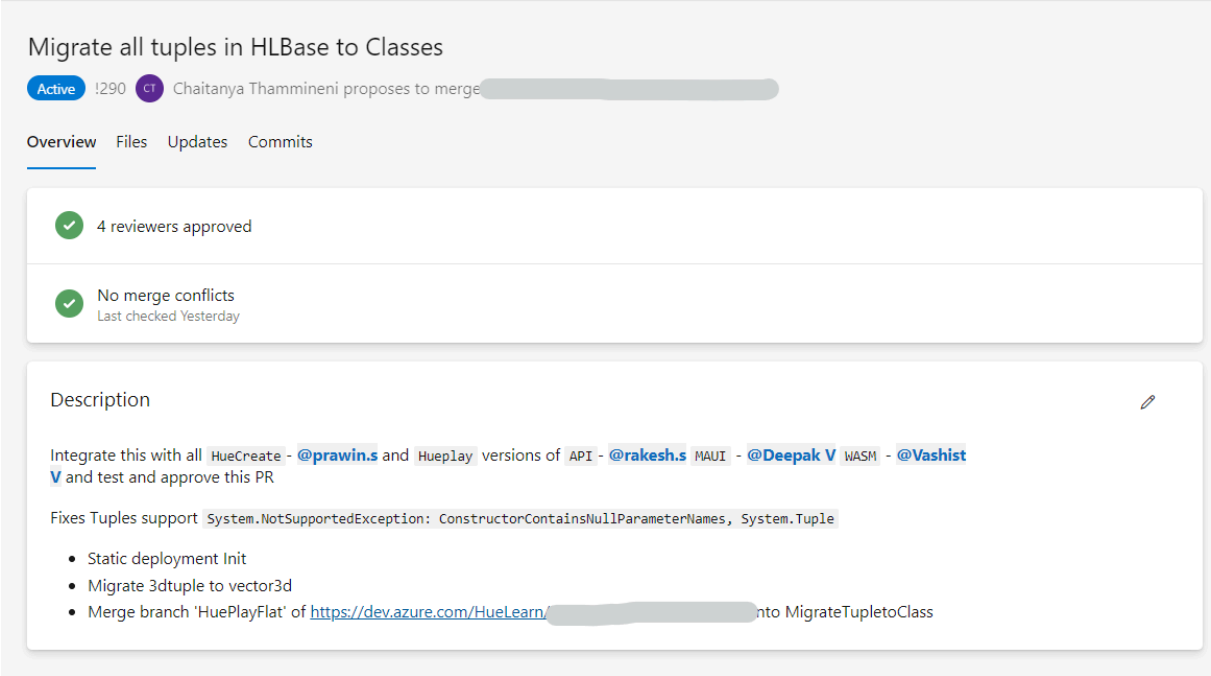
Wrong Coding Style.
Using `git reset` to undo changes in a shared repository

Right Coding Style.
Using `git revert` to create a new commit that undoes a previous commit

```
# Revert a specific commit
git revert <commit-hash>
```

2.6. Pre Push Request

When a single code base is shared among different applications, Changes made by one team should inform other team leads, to check their app functionality before raising the PR. This will make sure of common working code block for all the apps is implemented, instead of apps breaking after deployment.



2.7. Resolve Merge Conflicts

Careful conflict resolution ensures that the final code is correct and functional. Use Visual Studio's merge tool to handle conflicts.

When conflicts occur, Visual Studio will prompt you to resolve them using its merge tool.

Wrong Coding Style.
Resolving conflicts without understanding the changes

Right Coding Style.
Reviewing changes carefully and testing after resolving conflicts

```
git add resolved-file
git commit
```

3. Uncomitted work

Use `git stash` to Save Uncommitted Work. `git stash` helps manage work-in-progress without committing incomplete changes.

Wrong Coding Style.

```
Committing incomplete or experimental changes
```

Right Coding Style.

```
Save current changes
     git stash

Switch to another branch
     git checkout another-branch

Apply saved changes
     git stash pop
```

4. Periodic Synchronization

4.1. Visual Studio Code (VSCode)

4.1.1. Use the Source Control Panel

Wrong Coding Style.

Making changes without checking or committing frequently.

Right Coding Style.

- Open the Source Control panel by clicking on the Source Control icon in the Activity Bar on the side of the window.
- Stage changes by clicking the + icon next to files.

- Commit changes by entering a commit message and clicking the checkmark icon.
- Stage and commit changes

git add filename.txt

git commit -m "Describe your changes here"

4.1.2. Configure Auto Fetch

Right Coding Style.
Enable auto-fetch in VSCode settings.

- Go to File > Preferences > Settings (or Code > Preferences > Settings on macOS).
- Search for Git: Auto Fetch and check the box to enable it.

4.1.3. Use Extensions

Install extensions like **GitLens** from the Extensions Marketplace to enhance Git features.

Search for "**GitLens**" in the Extensions view and click Install.

4.2. Visual Studio

4.2.1. Use Team Explorer

Use Team Explorer to manage Git operations.

Right Coding Style.

Open **Team Explorer** from **View** > Team Explorer.

Use the **Home button** to navigate to the **Sync page** for committing and pushing changes.

4.2.2. Configure Git Settings

Wrong Coding Style.

Not configuring Git settings to fit your workflow.

Right Coding Style.

- Configure Git settings via Visual Studio.
- Go to Tools > Options > Source Control > Git Global Settings.

- Adjust settings such as user name and email.

Configure Git user settings via command line

git config –global user.name “Your Name”

git config –global user.email “you@example.com”

4.2.3. Use Git Commit Templates

Wrong Coding Style.

Using inconsistent commit messages.

Right Coding Style.

Set up a commit template to standardize commit messages.

Create a .**gitmessage** file with your desired commit message format.

git config –global commit.template ~/.gitmessage

5. Git Hooks for Automation

Git hooks automate repetitive tasks and help enforce coding standards.

Wrong Coding Style.

Configure Git Settings

```
Manual steps for common tasks like formatting or testing
```

Right Coding Style.

Using Git hooks to automate tasks like formatting and testing

```
.git/hooks/pre-commit
!/bin/sh
npm run lint
```

6. Develop using branches

Using branches, software development teams can make changes without affecting the main code line. Branching organizes development and separates work in progress from stable, tested code in the main branch.

Determining a single branching strategy is the solution to a chaotic development experience.

While there are several approaches to development, the most common are:

- **Centralized workflow**: Teams use only a single repository and commit directly to the main branch.
- **Feature branching**: Teams use a new branch for each feature and don't commit directly to the main branch.
- **Personal branching**: Similar to feature branching, but rather than develop on a branch per feature, it's per developer. Every user merges to the main branch when they complete their work.

Follow a Consistent Branching Model. Avoid using inconsistent branch names and strategies.

Right Coding Style.

- Branch names should include the developer's name and a brief description of the feature or bug, grouped under a `work` folder.
- Feature Branch Example: `work/deep/feature-login`
- Bug fix Branch Example: `work/pav/bugfix-header`

Here are some Git branching commands:

Commands	Description
git branch	Lists all branches in the repository.
git branch <branch-name>	Creates a new branch with the specified name.
git branch -d <branch-name>	Deletes the specified branch.
git branch -a	Lists all local and remote branches.
git checkout <branch-name>	Switches to the specified branch.
git checkout — <file>	Discards changes made to the specified file and revert it to the version in the last commit.

Right Coding Style.

Branch Strategy

We follow a three-branch strategy: `dev`, `stag`, and `prod`.

- dev: For local testing and development.
- stag: For product owner/CXO level review.
- prod: For public release.

Branching Rules

1. Development Branch (`dev`):

- Use this branch for all local development and testing.
- Periodically sync your code with the dev branch to ensure you have the latest updates.
- Example branch name: `work/john/feature-login`, `work/jane/bugfix-header`.

2. Staging Branch (`staging`):
   - This branch is for product owner/CXO level review.
   - Any code that needs to be merged into `staging` must be reviewed and approved by either Mr.Shivaram or Mr.Chaitanya.
3. Production Branch (`prod`):
   - This branch is for public release.
   - Any code that needs to be merged into `prod` must be reviewed and approved by either Mr.Shivaram or Mr.Chaitanya.

6.1. Branch Life

1. Keep Branches Short-Lived

   Short-lived branches reduce the risk of merge conflicts and keep the project organized.

Wrong Coding Style.
Keeping feature branches open for **months**

Right Coding Style.
Merging feature branches back to `develop` or `main` as soon as the feature is complete and tested.

```
git checkout develop
git merge feature/short-lived-branch
```

6.2. Stale Branches

Cleaning up stale branches helps keep the repository organized. It has to be done in the **presence of more than two persons.**

Wrong Coding Style.
Not cleaning Up Stale Branches regularly.

Right Coding Style.
Deleting branches that have been merged and are no longer needed.

```
# Delete a local branch
git branch -d old-feature-branch

# Delete a remote branch
git push origin --delete old-feature-branch
```

6.3. Shared branches

Avoid Force Pushing to Shared Branches.

Wrong Coding Style.
Using `git push --force` on `main` or `develop` branches.

Right Coding Style.
Using `--force-with-lease` ensures that you don't accidentally overwrite changes made by others.

```
# Force push with lease
git push --force-with-lease
```

6.4. Branch protection rules

Protecting important branches helps ensure code quality and stability. Implementing these additional practices can further enhance your Git workflow and ensure a well-maintained code base.

Wrong Coding Style.
Allowing direct pushes to important branches like `main`

Right Coding Style.
Enforcing branch protection rules in your repository settings

- Create draft pull requests to monitor merge conflicts.

Regularly pull from the source branch into your working branch to keep it in sync and reduce conflicts.

- Enforce successful build checks before merging

Commands	Description
git checkout <branch-name>	Switches to the specified branch.
git checkout -b <new-branch-name>	Creates a new branch and switches to it.
git checkout — <file>	Discards changes made to the specified file and reverts it to the version in the last commit.
git merge <branch>	Merges the specified branch into the current branch.

Commands	Description
git log	Displays the commit history of the current branch.
git log <branch>	Displays the commit history of the specified branch.
git log –all	Displays the commit history of all branches.
git stash	Stashes the changes in the working directory, allowing you to switch to a different branch or commit without committing the changes.
git stash list	Lists all stashes in the repository.
git stash pop	Applies and removes the most recent stash from the stash list.
git stash drop	Removes the most recent stash from the stash list.
git tag	Lists all tags in the repository.
git tag <tag-name>	Creates a lightweight tag at the current commit.

7. Git Managing History

Here are some Git managing history commands:

Commands	Description
git revert <commit>	Creates a new commit that undoes the changes introduced by the specified commit.
git revert –no-commit <commit>	Undoes the changes introduced by the specified commit, but does not create a new commit.
git rebase <branch>	Reapplies commits on the current branch onto the tip of the specified branch.

8. Review Changes

Reviewing changes helps catch errors and ensures code quality. Use `git diff` to Review Changes

Wrong Coding Style.
Merging branches without reviewing changes

Right Coding Style.

```
# Show differences between working directory and index.
git diff

# Show differences between two branches
git diff branch1 branch2
```

9. Conclusion

By utilizing this resource, developers can enhance their productivity and efficiency in working with Git, ultimately leading to smoother and more successful software development projects.