Software Development Version Control and GitHub Collaboration Best Practices

For The City of Coral Gables by Joseph Ruiz & Anthony Castillo

1. Overview

This document outlines the new conventions for managing software development version control within our organization. It focuses on establishing a structured process for using GitHub for collaborative development, ensuring code integrity, and promoting efficient teamwork.

2. Objectives

- To define a standardized version control workflow for all development teams.
- To ensure consistency and clarity in collaborative work using GitHub.
- To implement best practices for branch management, commits, and pull requests.

3. Version Control Strategy

We will be adopting a **GitHub-based version control** system. The following conventions and practices will govern our approach to managing source code repositories:

3.1 Repository Structure

- **Organization-Level Repository**: One GitHub repository for each project, hosted within a GitHub organization. This ensures all team members have access to the same repository and can collaborate in a centralized manner. We will then delegate access for each team member that is within the organization for each repository they will be working on.

3.2 Branching Strategy

The following branching strategy will be enforced to maintain clean and organized repositories:

- **Main Branch (main)**: This will represent the stable production-ready version of the code. All releases will be tagged from the main branch. All code being pushed onto this branch MUST work, for code to be approved and pushed to the main branch a senior or project manager delegated to the repo should delegate and review the code. Based on the scope of the project, the best convention would be to set up a CI/CD pipeline for continuous development and deployment.

- **Development Branch (dev)**: All new features and bug fixes will first be merged into the dev branch for testing before being merged into main. This branch should simulate main, with the appropriate hosting conventions as main.
- **Testing Branch (test):** Most unstable branch, all experimental features and ideas to be implemented. For each feature branch off experimental and then merge into Testing branch for testing. This branch should simulate local development environment, i.e. running the codebase off a local computer, and each feature should branch off testing and merge into test once the code is feature complete.
- **Feature Branches (test/feature_name)**: Each feature or bug fix will have its own branch off the test branch, named clearly according to the feature (e.g., test/login-auth).

3.3 Commit and Pull Request Guidelines

To maintain code quality and transparency, the following commit and pull request guidelines should be followed:

- **Commits**: Commits should be small, frequent, and descriptive. Use clear and concise commit messages following the format:

[TYPE] Brief description of changes:

~ Description going over changes in more granular detail

[TYPE]: Use a short descriptor to indicate the nature of the commit. Examples include:

- **[FEAT]**: A new feature or functionality.
- **[FIX]**: A bug fix.
- **[REFACTOR]**: Refactoring existing code without changing functionality.
- **[DOCS]**: Changes to documentation.
- **[STYLE]**: Code style changes (e.g., formatting, missing semicolons, etc.).
- **[TEST]**: Adding or modifying tests.
- **[CHORE]**: Miscellaneous changes that don't affect the code's logic (e.g., updating dependencies).
    - Example: [FEAT] Implemented user login validation
    - Example: [FIX] Resolved null reference in user profile

- **Pull Requests (PR)**:
    - All feature branches should go through a pull request process before being merged into the dev or main branch.
    - **Code Reviews**: Every PR must be reviewed by at least one other developer before being merged. Either by senior or project manager.
    - Ensure that tests pass and any conflicts are resolved before submitting a PR.

3.4 .gitignore Best Practices

To ensure that only relevant files are tracked in the repository and to avoid committing unnecessary or sensitive information, we will adopt the following .gitignore conventions:

Purpose of .gitignore

The .gitignore file is used to specify which files and directories Git should ignore. This is essential for keeping the repository clean and preventing:

- **Large or unnecessary files** (e.g., build artifacts, compiled code, or temporary files) from being added to the repository.
- **Sensitive information** (e.g., credentials, API keys, configuration files) from being accidentally committed.

.gitignore Conventions

1. **IDE-specific files**: Files generated by code editors and IDEs should not be committed. Common examples include:
    - .vscode/
    - .idea/
    - .DS_Store (macOS)
    - *.suo, *.user (Visual Studio)
2. **Build and Dependency Artifacts**: Avoid committing compiled code, dependency libraries, and other generated files. Examples:
    - node_modules/ (for JavaScript projects)
    - bin/, obj/ (for .NET projects)
    - dist/, build/ (for frontend or backend builds)

3. **Configuration Files**: Exclude machine- or environment-specific configuration files. Instead, use sample config files like config.example.json for developers to copy and modify locally.
    - *.env
    - config.json
4. **Log Files**: Exclude log files and temporary debugging information.
    - *.log
    - logs/
5. **Secrets and Credentials**: Avoid committing API keys, passwords, or any sensitive data by using environment variables or services like GitHub Secrets for automated builds and deployments.
    - .env
    - secrets.json
6. **Platform-Specific Files**: Some files are automatically generated by the operating system and should not be tracked in Git.
    - .DS_Store (macOS)
    - Thumbs.db (Windows)

Best Practices

- **Keep .gitignore project-specific**: While there are common patterns, ensure that your .gitignore is tailored to the needs of each specific project (e.g., different technologies or frameworks).
- **Version .gitignore files**: Commit and version the .gitignore file itself so all team members have the same ignore patterns.
- **Use global .gitignore for system-wide ignores**: Developers can configure a global .gitignore file on their local machine to exclude files generated by their operating system or specific tools (e.g., .DS_Store or *.swp files).

Example .gitignore for a .NET Core Project:

```
# IDE-specific files
.vscode/
.idea/

# Build artifacts
bin/
obj/

# User-specific settings
*.suo
*.user
*.userosscache
*.sln.docstates

# Log files
*.log

# Secrets
*.env
secrets.json

# OS-generated files
.DS_Store
Thumbs.db
```

3.5 Merge Strategy

To maintain a clean commit history and prevent unnecessary merge conflicts:

- **Squash Merging**: For most feature branches, squash merging should be used to combine all commits into a single commit when merging into the dev or main branch. Likewise Squash Merge for all commits that is branched off for each feature branch from test.
- **Rebase**: Developers should frequently rebase their feature branches against dev to minimize merge conflicts. Likewise rebase for experimental based on each Squash Merge.

4. GitHub Workflow for Collaborative Development

1. **Clone**: Each developer must clone the repository locally on their respective machine.
2. **Create a Feature Branch**: Use the following naming convention: feature/short-description or bugfix/short-description.
3. **Push Commits**: Push commits to the feature branch in the remote repository.
4. **Submit Pull Request**: Once development is complete, submit a pull request to merge into the corresponding branch, providing detailed information on the changes and referencing any relevant issues or tickets.
5. **Review and Approval**: Another team member must review the PR before it can be merged.
6. **Merge**: After approval, the feature branch will be merged into the corresponding branch. Upon testing completion, it can be merged into main for deployment.

5. Code Reviews and Testing

To maintain high code quality and prevent defects:

- **Code Reviews**: All pull requests must be reviewed by another developer.
- **Automated Testing**: Automated tests must be executed on all commits, and all tests should pass before merging any changes into the corresponding branch.

6. Security and Access Control

- **Permissions**: Developers will have write access to the dev, test & the corresponding feature branches but only maintainers can merge into main.

- **Sensitive Data**: No sensitive information (such as credentials or API keys) should be committed to the repository. Use environment variables and GitHub Secrets where necessary.

7. Conclusion

This convention establishes a solid foundation for collaborative development using GitHub, ensuring consistency, clarity, and code quality. By adhering to these best practices, we can streamline our development processes and improve team productivity while minimizing errors and conflicts.