ADDIS ABABA SCIENCE AND TECHNOLOGY UNIVERSITY

COLLEGE OF ENGINEERING

SOFTWARE COMPONENT DESIGN

Group Member	ID
Bereket Zerihun	ETS 0220/13
Bemnet Retta	ETS 0196/13
Afomia Tadesse	ETS 0089/13
Bemnet Kebede	ETS 0194/13
Aklesia Tewedros	ETS 0197/13

Table of Contents

Leveraging GitHub for Agile Software Development: A Case Study of the Blog Website Project

1. Introduction

In this paper, we explore how **GitHub**, a distributed version control platform, was instrumental in supporting the agile development process in our blog website project. Agile methodologies, with their focus on iterative development, flexibility, and collaboration, were at the heart of our workflow. By integrating GitHub's powerful features, such as branching, pull requests, and issues, we were able to maintain a high level of collaboration, efficiency, and continuous delivery throughout the project. This paper discusses our team's approach to utilizing GitHub in line with agile principles, providing insights into version control practices, collaboration techniques, and project management/ mmm using GitHub.

2. Project Setup

2.1 Initializing the Repository

To start, one member of the team initialized a GitHub repository to serve as the project's central hub. This repository was set up with the following steps:

1. **Creating the Repository on GitHub**: A new repository, blog-website, was created on GitHub to host the codebase. The repository was public to enable easy collaboration.
2. **Cloning the Repository Locally**: Once the repository was created, team members cloned it to their local machines:
3. git clone https://github.com/blogteam/blog-website.git

   This provided each team member with a local copy of the project that could be synchronized with the central repository.

2.2 Configuring the Repository

After the repository was cloned, we ensured that everyone had set up their local environments, including the necessary dependencies and development tools. This was essential for maintaining consistency across the project, especially when multiple people were contributing to the same codebase.

3. Branching Strategy

The key to maintaining an organized and conflict-free project was the adoption of a **branching strategy**. This strategy allowed us to work on separate features or bug fixes without affecting the main codebase.

3.1 Main Branch

The main branch acted as the stable version of the code that would be deployed. Any feature or bug fix that was merged into the main branch had to be thoroughly tested and reviewed.

3.2 Feature Branches

Each feature, whether it was a new login page or the creation of a blog post section, was developed on a separate branch:

git checkout -b feature/login-functionality

This approach isolated each piece of work, allowing team members to focus on individual tasks while avoiding interference from others' work.

3.3 Bug Fixes and Hotfixes

Bug fix branches were also created whenever a bug was discovered, such as incorrect form validation. These were similarly isolated in their own branches, e.g., bugfix/login-error.

3.4 Merging Feature Branches

Once a feature was completed, it was merged back into the main branch through a **pull request (PR)**. The PR allowed for peer reviews before the changes were integrated.

4. GitHub Workflow

4.1 Pulling the Latest Code

To stay up-to-date with changes made by other team members, each member regularly pulled the latest updates from the main branch:

git pull origin main

This step ensured that no one worked on outdated code, minimizing the risk of conflicts.

4.2 Committing and Pushing Changes

When working on a feature branch, changes were committed locally using the following workflow:

1. **Staging Changes**: After making modifications, team members staged the changes using git add:
2. git add .
3. **Committing Changes**: Changes were committed with a detailed message that described the modifications:
4. git commit -m "Add login functionality with JWT"
5. **Pushing to Remote**: After committing the changes, the feature branch was pushed to GitHub:
6. git push origin feature/login-functionality

4.3 Creating Pull Requests

Once a feature was completed, a **pull request (PR)** was created on GitHub to propose merging the feature branch into the main branch. The PR provided a space for team members to:

- Review the changes.
- Suggest improvements.
- Ensure code quality through continuous integration (CI) tests.

Example PR title:

- "Feature: Implement login functionality with JWT Authentication"

4.4 Resolving Merge Conflicts

Merge conflicts occurred when multiple developers modified the same lines of code. To resolve these conflicts:

1. **Fetch Latest Changes**: The developer working on the conflict fetched the latest code:
2. git fetch origin
3. **Merge Locally**: After resolving the conflicts in the editor, the changes were merged:
4. git merge origin/main
5. **Commit the Resolutions**: The resolved conflicts were committed and pushed back to the remote branch.

4.5 Pulling and Merging Updates

When new updates were pushed by other team members, each team member pulled the latest changes from the main branch to incorporate them into their local work:

git pull origin main

5. Agile Development Workflow Integration

The GitHub workflow was designed to integrate seamlessly with the **agile** practices used in the project. We followed a **sprint-based approach**, where each sprint typically lasted two weeks and focused on delivering a set of features.

5.1 Sprint Planning

At the beginning of each sprint, features and tasks were planned and assigned to team members. This included:

- Defining **user stories** and **acceptance criteria**.
- Breaking down features into smaller, manageable tasks.

5.2 Daily Standups

Each day, we held a 10-15 minute standup meeting where team members:

- Shared progress updates.
- Raised any blockers they encountered.
- Coordinated tasks for the day.

This helped maintain a clear understanding of what each team member was working on and ensured transparency in the development process.

5.3 Feature Completion and Pull Requests

At the end of a sprint, completed features were pushed to GitHub and merged into the main branch. The pull requests provided a space for code reviews and validation, ensuring that the final code met the required standards.

5.4 Sprint Review and Retrospective

After each sprint, we held a **sprint review** to demonstrate the completed features to the stakeholders. The feedback was then incorporated into the next sprint planning session.

6. GitHub Features Used

6.1 Pull Requests

Pull requests were central to our development process. They allowed for **peer reviews**, ensured code quality, and facilitated collaboration between team members.

6.2 Issues and Project Boards

We used **GitHub Issues** to track bugs, tasks, and feature requests. These issues were tagged and categorized into **milestones**:

- **Feature**: New features to be developed.
- **Bug**: Issues that needed to be fixed.
- **Enhancement**: Improvements to existing features.

6.3 Branch Protection Rules

To prevent direct commits to the main branch, we set up **branch protection rules** on GitHub. These rules ensured that:

- Only pull requests from feature branches could be merged into main.
- CI tests had to pass before merging.
- At least one code reviewer had to approve the pull request.

6.4 Continuous Integration (CI)

We integrated **CI tools** with our GitHub repository to automatically run tests whenever a pull request was opened or updated. This ensured that new code changes did not break the existing functionality.

7. Challenges Faced and Solutions

7.1 Merge Conflicts

Challenge: Merge conflicts arose when multiple team members worked on the same file simultaneously. **Solution**: We resolved these conflicts by regularly pulling the latest code and communicating any shared file changes.

7.2 Delayed Code Reviews

Challenge: Sometimes, code reviews were delayed, which impacted the development timeline. **Solution**: We set deadlines for pull request reviews and ensured that team members were assigned as reviewers in advance.

7.3 Feature Integration

Challenge: Integrating multiple features at once caused unexpected bugs. **Solution**: We adopted an incremental approach, merging smaller, well-tested features one at a time.

8. Conclusion

GitHub was a key tool in enabling efficient collaboration and project management for our blog website project. By adhering to an agile development methodology and integrating GitHub's version control, branching, pull requests, and CI capabilities, our team successfully delivered a fully functional blog website. The combination of GitHub's features and agile practices ensured that the project remained on track, with clear communication, efficient problem-solving, and quality code delivered at the end of each sprint.