MiniGitHub

Information Systems Development

Name: Phat Tran Dai
Login: TRA0163

Content

1. Vision

1.1. Introduction

The purpose of this document is to define the high-level needs and features of MiniGitHub, a web-based code collaboration platform. MiniGitHub is designed to help developers and teams manage code repositories, track changes, and collaborate on software projects.

This document provides an:
- overview of the product's purpose,
- positioning in the market,
- stakeholders and users,
- and its key features.

1.2. Product Overview

1.2.1. Product Perspective

MiniGitHub is an independent system designed as a lightweight GitHub alternative. It consists of three major layers:

- Presentation Layer: Web interface and desktop application for interaction.
- Domain Layer: Business logic for repository, commit, and issue management.
- Data Layer: Database with persistence services.

Interfaces (for now):

- Web browser (for end users)

(would be nice to have):
- Desktop application (for end users)
- REST API (for integration with Git clients)

1.2.2. Assumptions and Dependencies

System assumes internet-connected users with modern web browsers.

Database system (SQL Server or PostgreSQL) must be available. However, in my project I'm using Sqlite for simplicity.

Hosting environment must support a modern .NET runtime.

Full Git implementation is out of scope; repository management will be simplified.

1.3. Positioning

1.3.1. Problem Statement

The problem of	fragmented and inefficient collaboration on code projects
affects	developers, open-source contributors, and organizations that rely on version control and collaboration tools
the impact of which is	code conflicts, slower development cycles, miscommunication, lack of version tracking, and difficulties in managing distributed teams
a successful solution would be	a simple and accessible platform that enables users to store, manage, and share code repositories, track changes through versioning, and collaborate effectively with features such as issue tracking and pull requests.

Table 1: Problem statement

1.3.2. Product Position Statement

For	developers, students, and organizations seeking a lightweight but structured collaboration platform
Who	need a reliable and accessible version control and project collaboration tool
The MiniGitHub system	is a repository hosting and collaboration platform
That	provides version control, issue tracking, and lightweight collaboration features
Unlike	existing heavyweight platforms such as GitHub or GitLab, which may be overly complex for small teams
Our product	offers a simple solution with only the essential collaboration tools.

Table 2: Product Position Statement

1.4. Stakeholder and User Descriptions

1.4.1. Stakeholder Summary

Name	Description	Responsibilities
Project Sponsor	Provides funding or backing for development	Defines goals and approves overall direction
Development Team	Engineers building the system	Implements features, ensures maintainability
System Administrator	Manages hosting and deployment	Keeps system online and secure

Table 3: Stakeholder Summary

1.4.2. User Summary

Name	Descriptions	Responsibilities	Stakeholder
Developer	Individual writing and committing code	Creates repositories, commits changes, opens issues	Development Team
Maintainer	Oversees project repositories	Reviews contributions, manages issues and pull requests	Development Team
Contributor	External collaborator	Suggests changes, reports bugs, submits pull requests	Development Team

Table 4: User Summary

1.4.3. User Environment

Number of users per project	Typically 2–10, may scale for open collaboration.
Task cycle	Continuous; developers frequently commit, push, and review code.
Environment constraints	Primarily desktop/laptop through a web browser
Platforms in use	Modern web browsers for front-end, PostgreSQL/SQL Server for data, ASP.NET backend.

1.5. Product Features

User management	Registration, login, authentication.
Repository management	Create, list, delete repositories.
Commits	Track file version and changes.
Issue	Open, comment on, and close issues.
Pull requests (not implemented)	Propose and merge proposed changes.
Search and browse	Find repositories and issues.
Collaboration tools	Comments, activity feeds.
Basic analytics	Number of commits, contributors, and issues per repo.

1.6. Other Product Requirements

Standards	Follows MVC architecture for the web front end, layered design (Data Access, Domain, Presentation), and selected patterns (Service Layer, Mapper, Table Data Gateway, Unit of Work).
Performance	Must support 50–100 concurrent users with acceptable response time.
Robustness	Transactions must ensure consistent repository state.
Platform requirements	Runs on Linux/Windows server, database is hosted on a separate node.
Usability	Clean and intuitive UI, minimal learning curve.
Documentation	Basic user guide and technical developer documentation.
Constraints	Project scope limited to core GitHub-like features (full Git replication is out of scope). Only core repository and collaboration features are included.

2. Functional Analysis – Use Case Model

2.1. Main Actors

Actor	Description
User	A registered user of the system who can create repositories, make commits, manage issues, and pull requests.
Visitor	A non-logged-in user who can browse public repositories.
Administrator	A system administrator who oversees the operation of the application, manages users, and handles incidents.

2.2. Use-Case Diagram

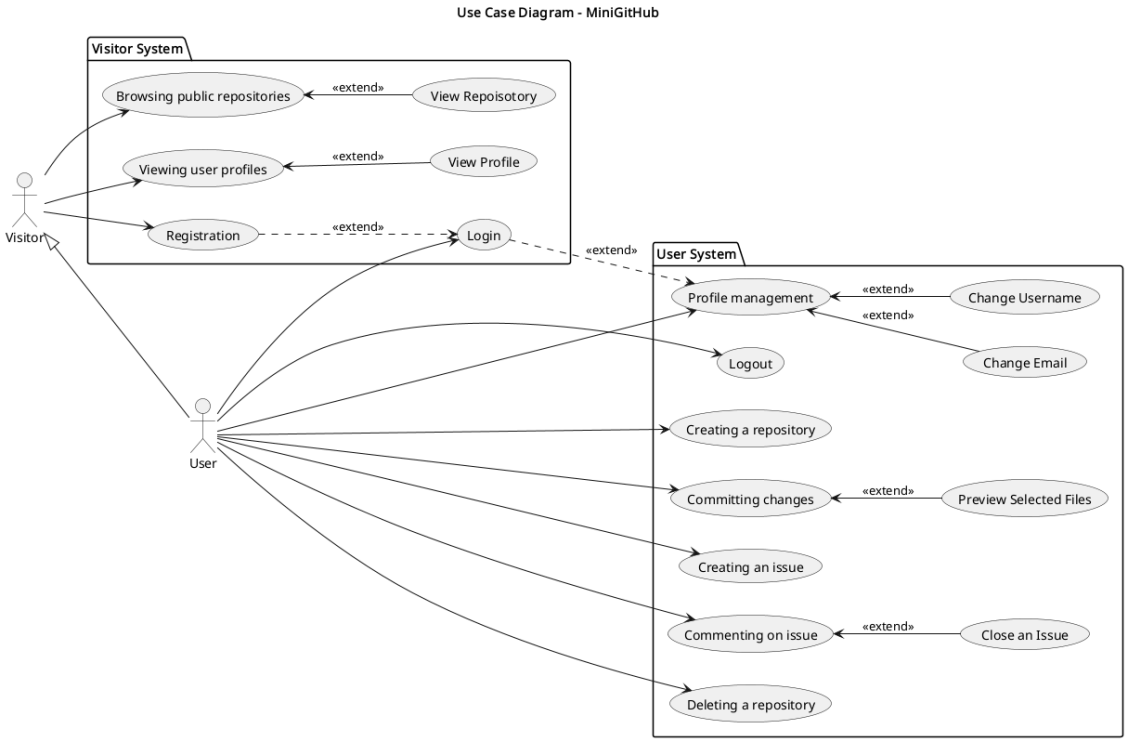

Figure 1: Use Case Diagram

2.3. Use Case 1: Creating a Repository

- **Name and Identification:** Creating a Repository
- **Actor:** User
- **Goal:** Create a new repository where the user can store and manage source code.
- **Preconditions:**
  - The user is logged into the system.
  - The repository name is not already used in the user's account.
- **Triggering Event:** The user clicks the "New Repository" button.
- **Main Scenario:**
  - The system displays a form for creating a repository.
  - The user enters a name, an optional description, and sets the visibility (public/ private).
  - The user confirms the creation.
  - The system verifies that the entered information is valid.
    - The repository name is unique within the user's repositories.
  - The system creates a new repository.
  - The system displays the page of the newly created repository.
- **Alternative Scenario:** The name is invalid or already exists - the system displays an error message and allows correction.
- **Postconditions:**
  - A new repository record is created in the database.
  - The user becomes the owner of the repository.

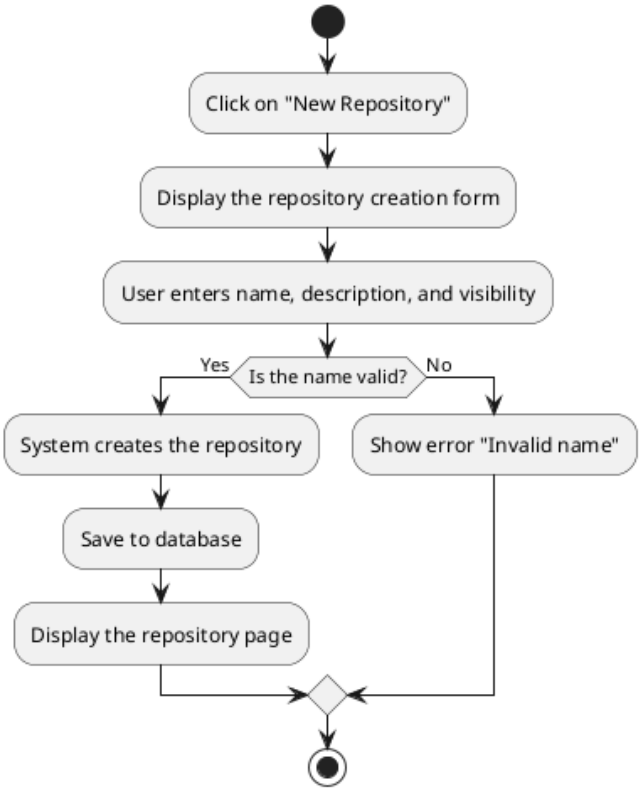

Figure 2: Use-case diagram – Creating a Repository

2.4. Use Case 2: Submitting Commits (Commit Changes)

- **Name and Identification:** Submitting commits to the repository
- **Actor:** User
- **Goal:** Save changes in source files to the repository's version history.
- **Preconditions:**
  - The user has write access to the repository.
  - The repository exists and contains at least one branch.
- **Triggering Event:** The user makes changes in a local copy of the repository and wants to save them.
- **Main Scenario:**
  - The user selects the files they want to commit.
  - They enter a description of the change (commit message).
  - They confirm the submission.
  - The system verifies the user's access rights.
  - The system creates a new commit and links it to the previous one.
  - The system updates the version history of the branch.
  - The system displays a confirmation of a successful commit.
- **Alternative Scenarios:**
  - The user does not have write permissions → the system displays the error "Access denied".
  - A conflict with the current version occurred → the system requests conflict resolution.
- **Postconditions:**
  - The new commit is saved in the database.
  - The repository history is extended with the new state.

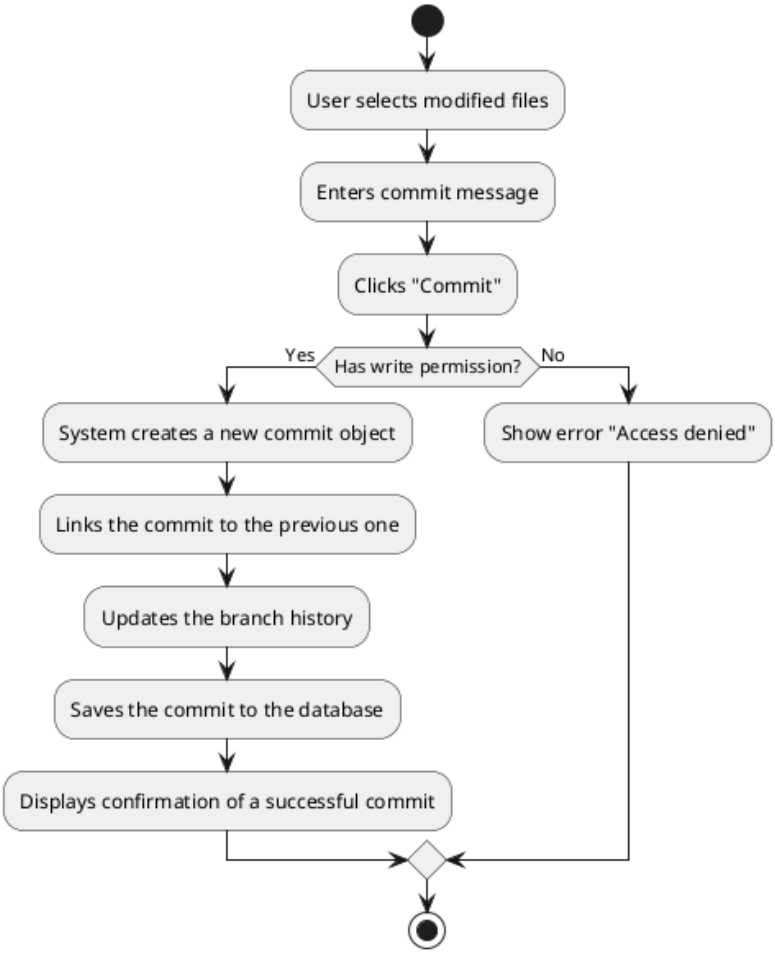

Figure 3: Use-case diagram — Submitting commits to the repository

2.5. Use Case 3: Creating an Issue

- **Name and Identification:** Creating an Issue
- **Actor:** User (Contributor or Maintainer)
- **Goal:** Report a problem, bug, or enhancement proposal in the repository so it can be tracked and resolved.
- **Preconditions:**
  - The repository exists.
  - The user has permission to create issues in the repository.
- **Triggering Event:** The user discovers a bug or has an improvement idea and clicks "New Issue".
- **Main Scenario:**
  - The user opens the repository page.
  - Clicks "New Issue".
  - The system displays the issue creation form.
  - The user fills in the title, description (and optionally adds labels or assigns an assignee, not implemented).
  - The user confirms the creation of the issue.
  - The system validates the inputs (e.g., forbidden characters, title length).
  - The system stores the issue in the database with the default status **Open**.
  - The system displays the newly created issue in the list.
- **Alternative Scenarios:**
  - The input contains invalid or empty values $\rightarrow$ the system displays an error message and allows correction.
  - A database write error occurs $\rightarrow$ the system displays the error "Failed to save issue".
- **Postconditions:**
  - The new issue is visible in the list of all issues.
  - Users can comment on it or change its status (e.g., Closed).

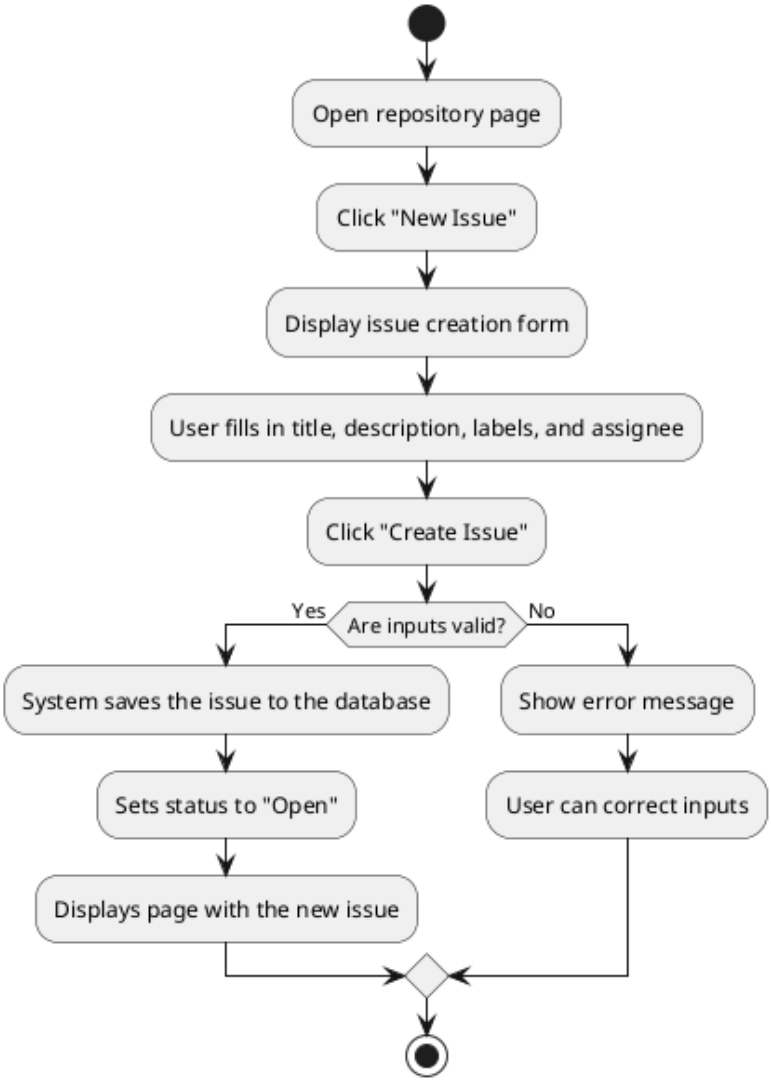

Figure 4: Use-case diagram – Creating an issue

3. Analysis – Technical Requirements

3.1. Conceptual Model of the Domain

3.1.1. Main Entities:

- **User** – represents a system user (owns an account, login credentials, role).
- **Repository** – a project space containing commits and issues.
- **Commit** – a record of a source code change, linked to the author.
- **Issue** – an item for tracking problems, bugs, or improvement proposals.
- **Comment** – a comment on an issue or commit.
- **File** – a file stored in a repository, with version history.

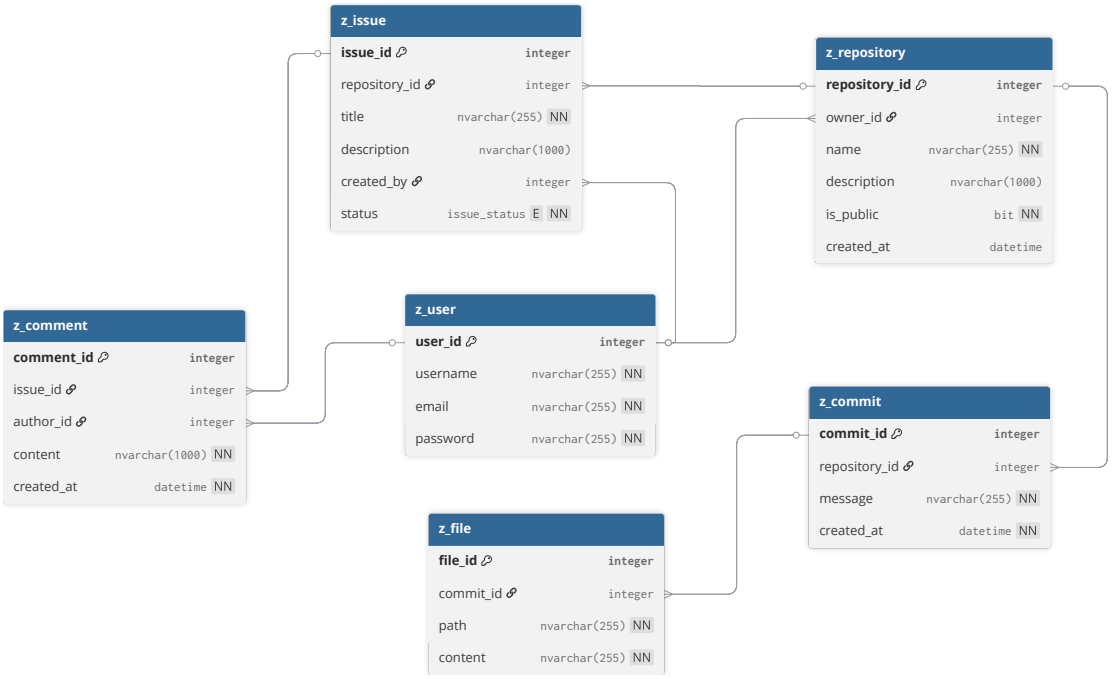

Figure 5: Conceptual model (entity model)

3.2. Estimated Sizes of Entities and Their Quantities

Entity	Estimated Number of Records	Size of One Record	Note
User	5000	~2 KB	basic profile
Repository	15000	~3 KB	metadata
Commit	500000	~5 KB	message, reference to files
Issue	50000	~3 KB	problem title, description, status
Comment	200000	~1 KB	text, author, linked to issue/commit
File	1000000	~4 KB	considering metadata only, not binary content

- **Total estimated data size:** 6–8 GB in production (including indexes and metadata).
- + 1000000 Files that can be simple text files (small size), images (bigger files), zipped archives (huge)

3.3. Estimated Number of Concurrent Users

- The system targets small development teams and individuals, therefore:
  - Normal load: 50–100 concurrently active users
  - Peak load: up to 300 users
  - Inactive accounts: up to 10x more than active users
- Given the asynchronous nature of repository operations (most are short), this range is realistic for a standard server deployment.

3.4. Types of User Interactions with the System and Their Complexity

Interaction Type	**Description**	**Estimated Complexity**	**Frequency**
Viewing Repository	reading metadata, list of commits and issues	low	frequent
Committing Changes	writing new version to DB	medium	medium
Creating Issue	writing record	low	medium
Commenting	writing text, updating	low	frequent
Creating Repository	checking for duplicates, initializing DB records	medium	less frequent
Viewing Commit History	data aggregation	high	medium
Searching	full-text search on various records in DB	medium	frequent

Table 6: Types of user interactions

3.5. Initial System Layout and Platform Choices

3.5.1. Architecture:

- Multi-layer architecture with separation of concerns:
  - Data Layer:
    - Implementation using SQLite (development) or PostgreSQL (production)
    - Patterns used: Table Data Gateway, Data Connector, Unit of Work
  - Domain Layer:
    - System logic (repositories, commits, issues, ...) using .NET
    - Pattern: Service Layer, Mapper (interfacing Data layer)
  - Presentation Layer:
    - Web UI: ASP.NET Core MVC

Layer	Technology	Platform
Data	SQLite / PostgreSQL	Linux
Domain	.NET C# Class Library	cross-platform
Web UI	ASP.NET Core MVC	Browser

Table 7: Platform Targeting

4. Wireframe

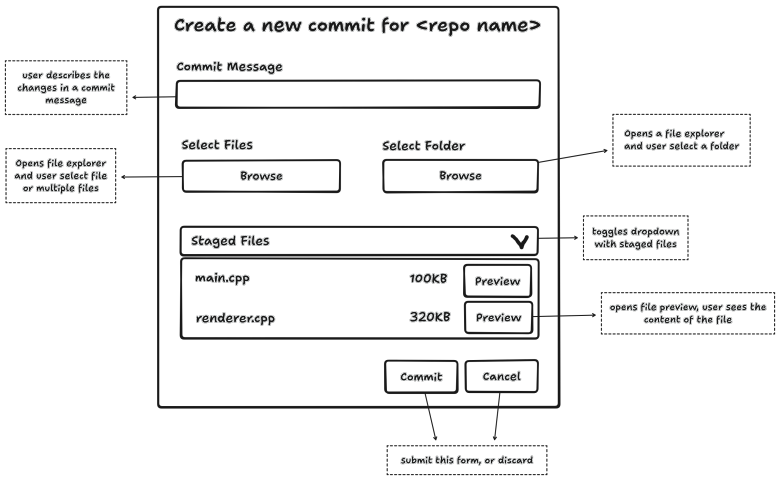

Figure 6: Writeframe - Add Commit

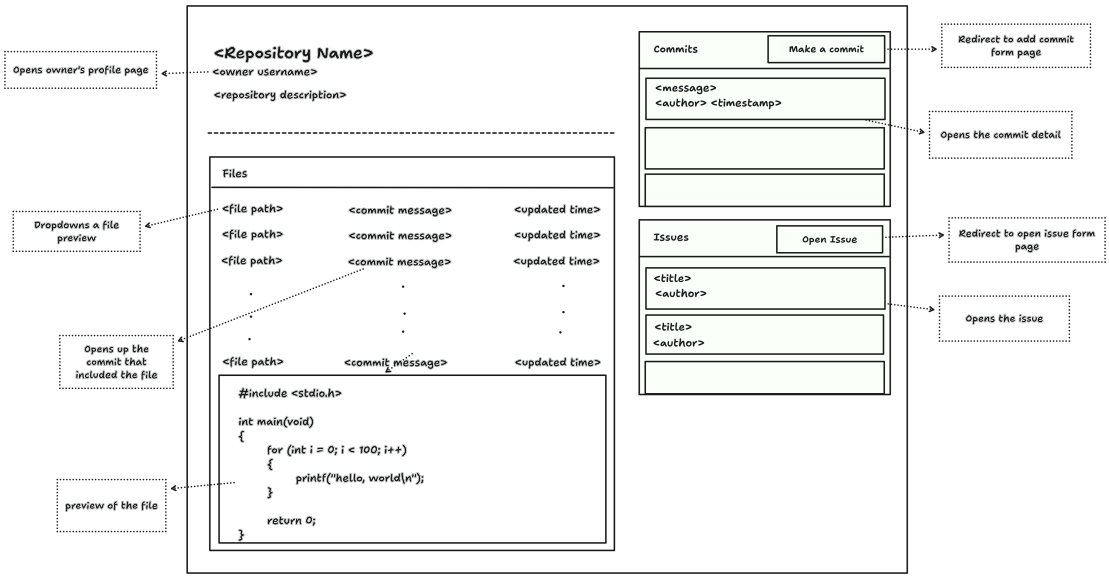

Figure 7: Writeframe - Repository View

5. Domain Model

There are 3 main groups of classes:

- Entities
- Mappers
- Repositories

5.1. Entities

Entities represent the model classes in the domain layer. They are mostly the same with the models in the data layer, but they also include some other methods and have compound attributes. For example a Commit model also contains Files.

5.2. Mappers

They provide seamless mapping between models

- from the data layer to the domain layer
- from the domain layer to the data layer

They are used for abstracting the models internal workings. The client is only ever exposed to the model from the domain layer when working in the domain layer.

The opossite is true. The client is only exposed to the data models when working in the data layer.

5.3. Services

These classes provide actions that we can do basic CRUD operations upon our models. These CRUD operations are translated into database calls, that are not necessarily only trivial operations. Some actions that these services provide deal with DB transactions and insert multiple records at once. Again the client doesn't have to worry about the inner workings of the DB, they just call upon the action and it either succeeds or it fails and throws an error.

5.4. Class Diagram

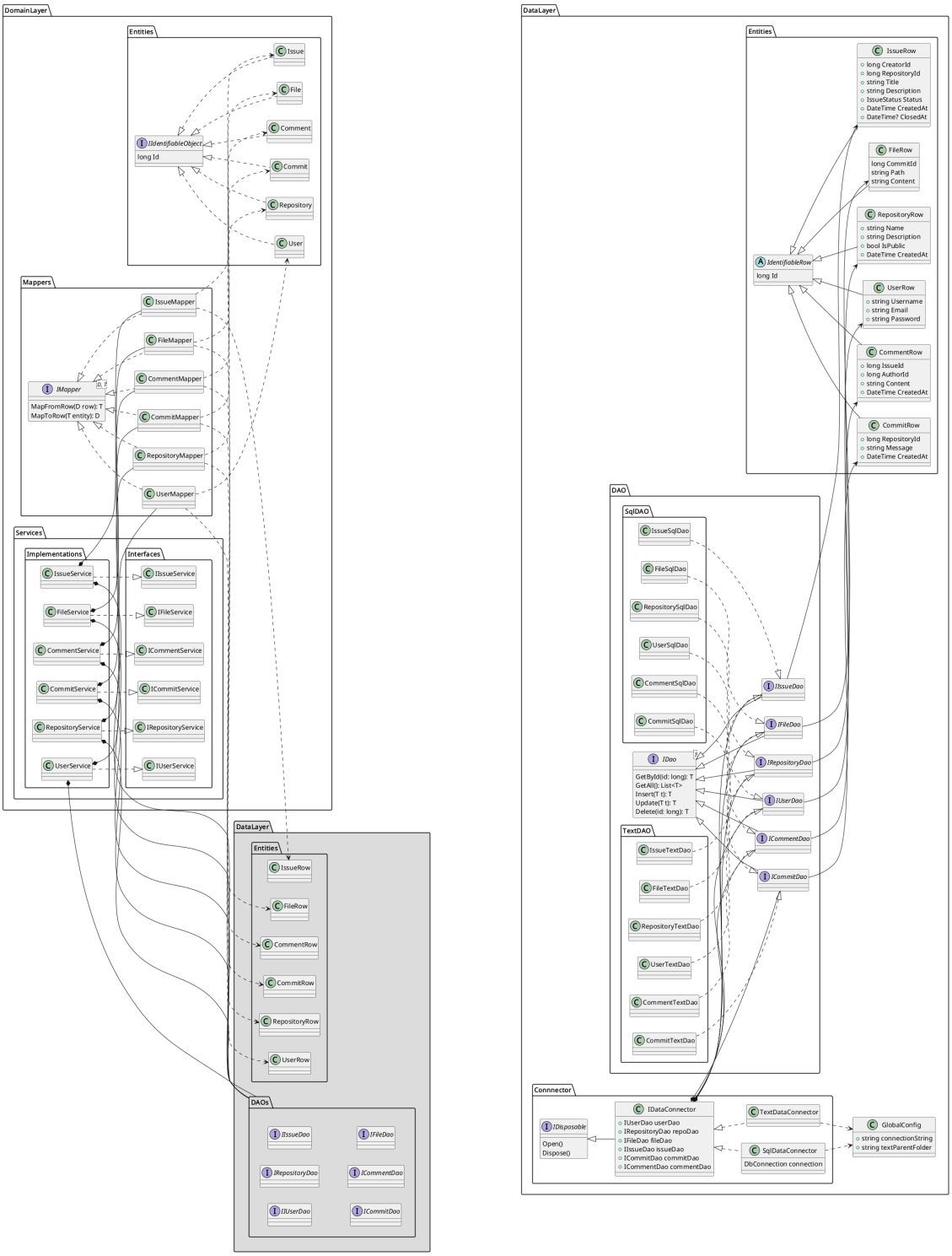

Figure 8: Class Diagrams of Data and Domain Layers

5.5. Sequence Diagram

Here's a sequence diagram for adding a commit.

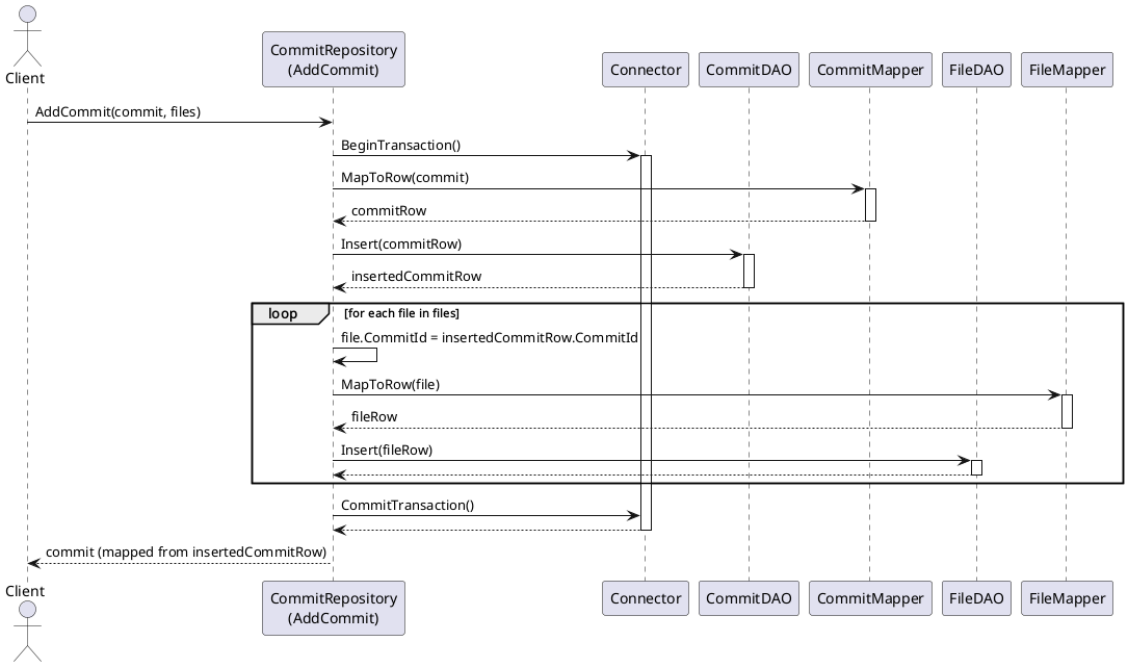

Figure 9: Sequence Diagram - Add Commit

6. Architecture

6.1. Components

This project follows a very standard architecture. It has 3 main components - Data, Domain and Presentation layers.

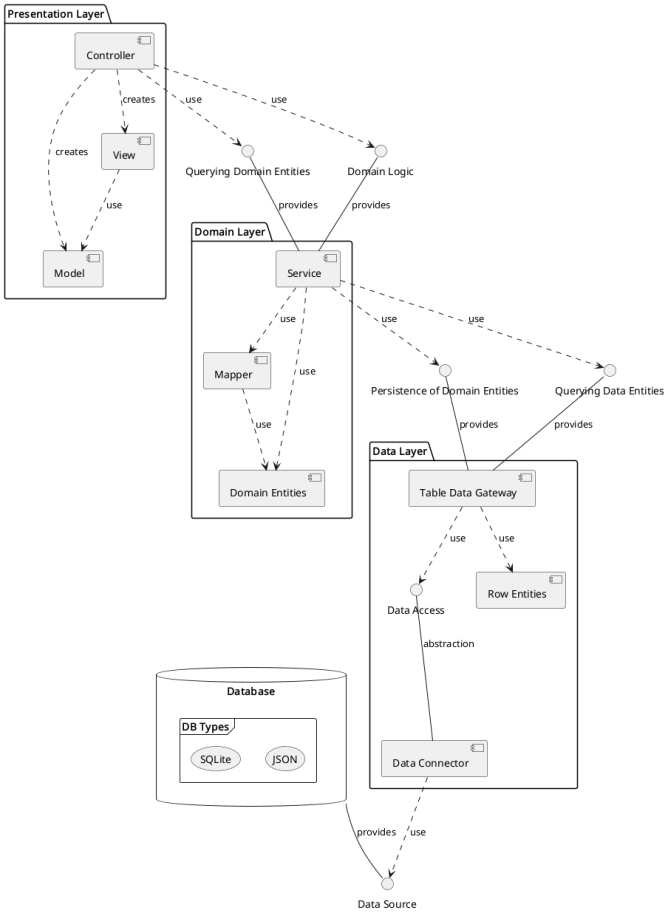

Figure 10: Component Diagram

The data layer depends on a database for it is its data source. The data source is provided by a database and used by the data connector. Data connector serves as an abstraction over the raw data source and it itself provides a data access. Table data gateways use this data access and provide these interfaces - persistence of data entities and querying for data entities. These interfaces are used by services in the domain layer. Services provide the domain logic interface. This interface is used by the presentation layer.

6.2. Deployment

This whole application will by deployed on two computational nodes. One is hosting the database and the other is running an instance of our application. This application connects to the node hosting the database. This deployment scheme distibutes the workload thus reduces computaional load on individual nodes. Client devices connect to the application via a browser.

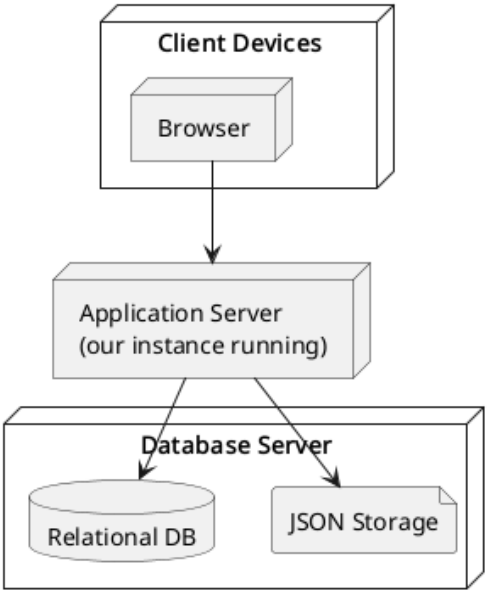

Figure 11: Deployment Diagram