Collaborative code editor

Made by: Hasan Omar Masoud

Table Content

Introduction

1.1 Background

In today's fast-paced development environment, collaboration among programmers is essential for efficient software creation. Traditional coding environments often lack the necessary features to support multiple users working on the same codebase simultaneously, leading to version control issues, miscommunication, and project delays.

The rise of collaborative coding tools has transformed how developers work together, allowing for real-time updates and streamlined workflows. By leveraging web technologies, teams can interact seamlessly, regardless of their geographical locations. Our project aims to address these challenges by creating a robust collaborative code editor that allows multiple users to write, edit, and execute code in real time. This platform is designed not only for individual coding tasks but also for collaborative code reviews and discussions, fostering a more interactive and productive programming environment.

1.2 Objectives

The primary objectives of the collaborative code editor project are as follows:

Real-Time Collaboration: To develop a web-based platform where multiple users can edit code simultaneously, with live updates reflecting changes made by any user in real time.

Version Control: To implement an advanced version control system that tracks changes, allows users to revert to previous versions, and maintains a detailed log of contributions by each user.

Code Execution: To provide an environment where users can execute code snippets in multiple programming languages, with the output displayed directly in the editor.

Code Reviews and Comments: To facilitate collaborative code reviews by enabling users to leave comments on specific lines of code, promoting discussion and improvement.

1.3 Technologies Used

The collaborative code editor project utilizes a diverse set of technologies to achieve its objectives. Key technologies include:

Java: The primary language for backend development, using the Spring Boot framework to create a robust and scalable server application.

Spring Boot: A popular Java framework that simplifies the development of web applications by providing built-in features for RESTful APIs, security, and WebSocket support.

Nginx: A high-performance web server used to serve static files and act as a reverse proxy, improving performance and enabling secure access to the backend APIs.

Angular: A powerful JavaScript framework for building dynamic and responsive user interfaces. Angular is used for the frontend development of the collaborative code editor.

WebSocket: A communication protocol that enables real-time, full-duplex communication between clients and the server, allowing for live code collaboration.

OAuth 2.0: An authentication framework used to secure user login processes and manage access permissions.

MySQL/MongoDB: Database technologies used for data storage and management, enabling the application to persist user data, code files, and version histories.

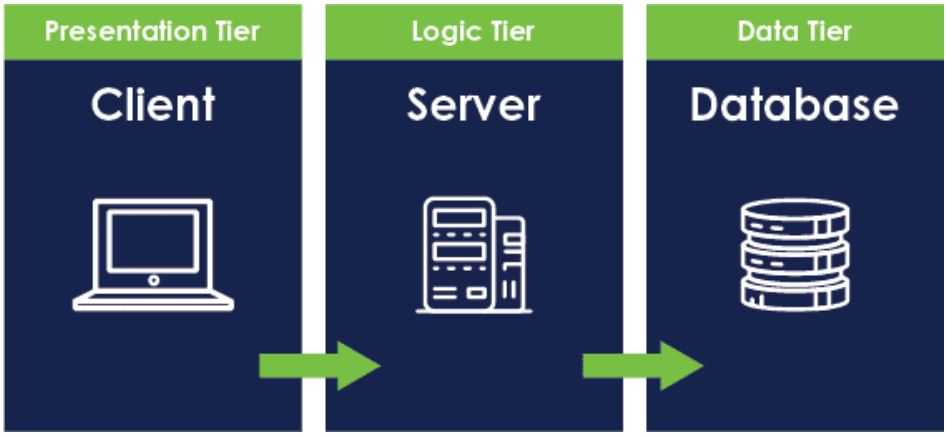
Docker: A containerization platform used to package the application and its dependencies, ensuring consistency across development and production environments.

Language Server Protocol (LSP): A protocol that allows the editor to communicate with language servers, providing features like code completion, error checking, and code navigation. This enhances the user experience by offering intelligent code assistance across various programming languages.

Project Structure

2.1 Overview

This collaborative code editor is designed around a robust 3-tier architecture that separates the application into three distinct layers: Presentation Layer, Application Layer, and Data Layer. This architecture promotes scalability, modularity, and maintainability by decoupling the components, allowing each layer to focus on its specific responsibilities.



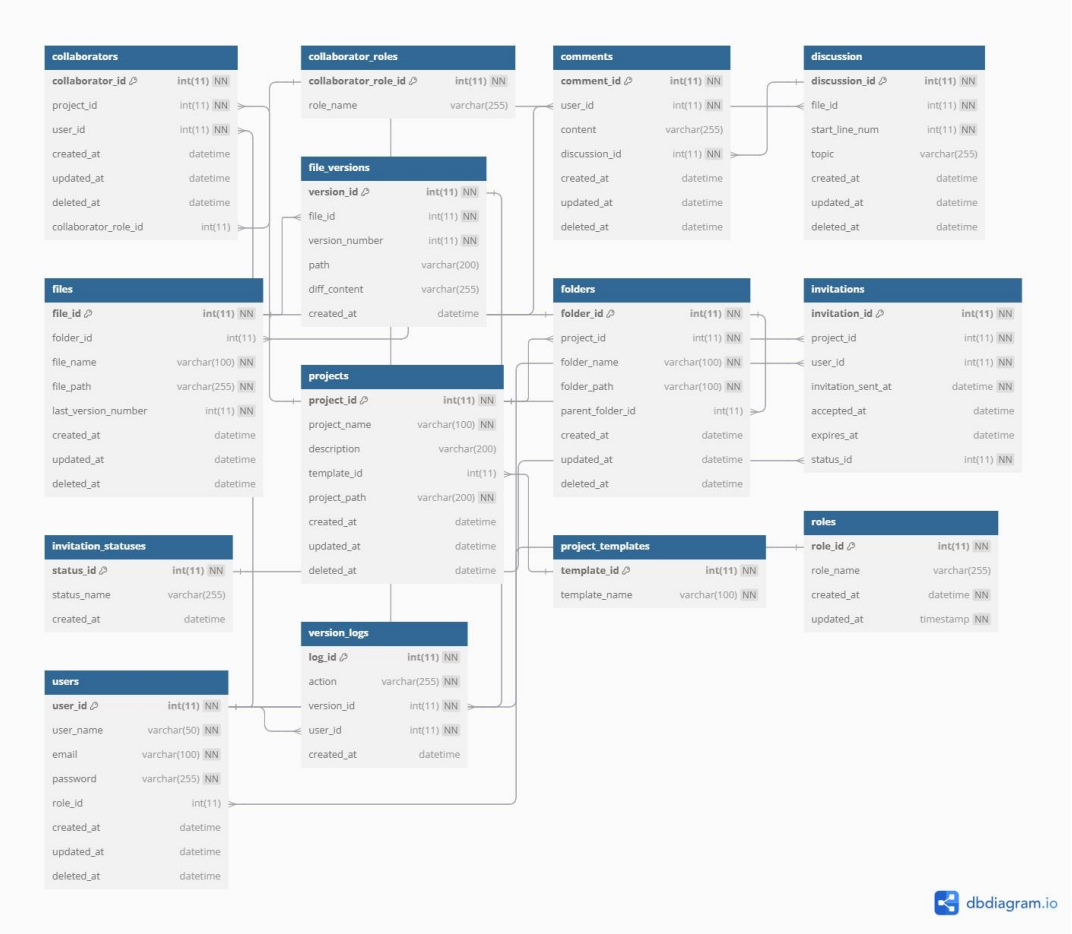
Presentation Layer: The frontend of the application is built using Angular, providing a dynamic and responsive user interface that allows users to write, edit, and execute code in real time. This layer interacts with users through a browser-based client, handling user input, displaying real-time updates, and providing code review and commenting functionalities.

Application Layer: The backend server, developed with Java and Spring Boot, forms the core of the application's logic. It handles client requests, manages collaborative editing sessions, enforces role-based access control (RBAC), and integrates OAuth 2.0 for secure authentication. The server also manages communication with the presentation layer via RESTful APIs for HTTP requests and WebSocket for real-time updates, ensuring consistent synchronization across all users.

Data Layer: The data storage layer uses a relational (MySQL) database to store persistent information, including user data, path files, and path version histories. This layer ensures data integrity and provides the application layer with quick, concurrent access to stored information.

2. MySQL Database Design

The MySQL database design for the collaborative code editor project consists of several interconnected tables, each tailored to support critical application functionalities, such as user authentication, file management, version control, and collaborative code review. By organizing data into structured tables with clear relationships, the database ensures efficient data retrieval, storage, and management.



Database Tables:

- **Collaborators**: Stores information about users who collaborate on projects, including their roles and timestamps for when they joined or left a project.
- **Collaborator Roles**: Defines specific roles (e.g., owner, member) assigned to collaborators, which helps manage permissions.
- **Comments**: Holds individual comments made by users on specific discussions, tracking the commenter, content, and timing.

• **Discussion**: Contains discussions related to specific code files or lines, allowing users to create discussion threads tied to particular areas of code.
• **Files**: Manages the metadata of files within the project, including names, locations, and version numbers.
• **File Versions**: Tracks each version of a file, allowing the project to manage and reference specific file versions.
• **Folders**: Organizes files within projects into a folder structure, supporting nested folders with parent-child relationships.
• **Invitations**: Stores invitations for users to join projects, with statuses (e.g., accepted, pending) and expiry details.
• **Invitation Statuses**: Defines statuses (e.g., accepted, pending, expired) for tracking the state of project invitations.
• **Projects**: Holds project information, including names, descriptions, and optional templates.
• **Project Templates**: Provides templates that can be applied to new projects for a pre-configured setup.
• **Roles**: Lists possible user roles within the system (e.g., admin, user), which defines user permissions.
• **Users**: Contains user account details, including usernames, emails, and role assignments.
• **Version logs**: Logs actions taken on file versions, tracking who modified which file version and when.

Database optimizations and best practices:

Indexes:

- **Primary Keys**: Indexes on primary keys such as `user_id`, `project_id`, and `file_id` enhance query performance by allowing fast lookups.
- **Secondary Indexes**: Implement secondary indexes on frequently queried fields, such as `email`, to improve login and search speeds.

Constraints:

- **Foreign Key Constraints**: Enforce referential integrity with foreign key constraints (e.g., `owner_id` in the Project table, `user_id` in the Comment table) to ensure related records remain valid.
- **Unique Constraints**: Apply unique constraints on fields like `email` in the User table to prevent duplicate user entries.

Soft Deletes:

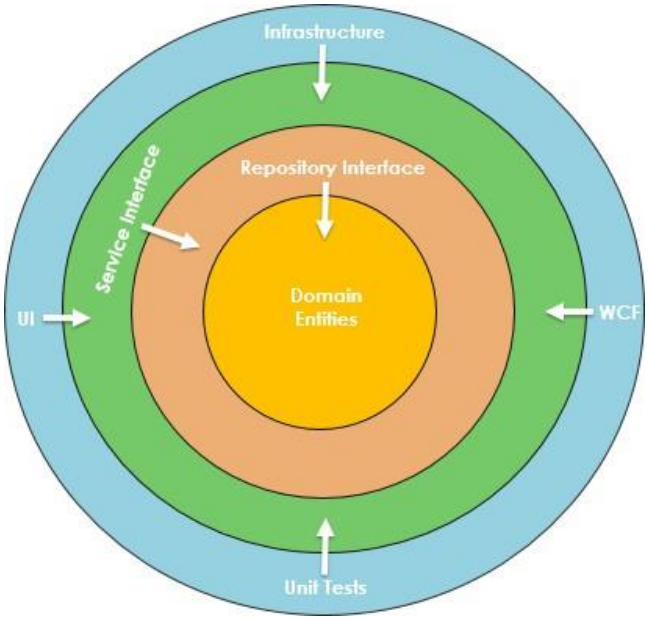
- **Deleted_at Column**: Introduce a `deleted_at` column to mark records as "soft deleted" instead of permanently removing them. This approach preserves data history for potential recovery.

Transactions:

- **Use of Transactions**: Implement transactions for critical operations (such as version control updates) to ensure atomicity. This ensures that operations either fully complete or fail, preventing partial changes to the database.
- **Isolation Level**: Use the **Serializable** isolation level for operations like editing files. This prevents phenomena such as dirty reads, non-repeatable reads, and phantom reads, ensuring that each transaction is completely isolated from others. This is particularly important during collaborative edits, as it maintains data integrity when multiple users attempt to modify the same file concurrently.

3. Backend

3.1 System Architecture Design: Onion Architecture is a software design pattern that promotes a clear separation of concerns within an application. This architecture is centered on the principle that the core logic of the application should be independent of external frameworks, user interfaces, and data sources. By organizing the codebase into concentric circles, Onion Architecture ensures that changes in one part of the application have minimal impact on other parts. This enhances maintainability, testability, and flexibility in the application's design.



In our implementation, we utilize three main layers: the **Core Layer**, the **Infrastructure Layer**, and the **Presentation Layer**.

1. Core Layer:

The **Core Layer** is the heart of the application, containing the essential business logic and domain entities. It is designed to be independent of external frameworks, ensuring that core functionalities remain stable and reusable.

- **Models or Entities**: These are the core domain objects representing the fundamental business concepts within the application. For instance, classes like `User`, `Project`, and `File` encapsulate the properties and behaviors relevant to their respective domains. They form the basis of the data model and are typically mapped to database tables.
- **DTOs (Data Transfer Objects)**: DTOs are simple objects that carry data between processes, particularly between the server and client. They help in decoupling the internal data representation from the external representation, enhancing security and performance by only exposing necessary data fields.
- **Mappers**: Mappers are utility classes responsible for converting entities to DTOs and vice versa. They streamline data transformation processes, allowing for clean and maintainable code by separating the mapping logic from the core business logic.
- **Service Interfaces**: These define the contracts for the business logic operations. Service interfaces specify the methods available for use, such as user registration, authentication, and project management, allowing for implementation flexibility while adhering to defined behaviors.
- **Exceptions**: This component includes custom exception classes that represent specific application errors. For example, exceptions like `UserNotFoundException` or `InvalidInputException` help to manage and handle errors more effectively within the application logic.

2. Infrastructure Layer:

The **Infrastructure Layer** provides the necessary implementations for the business logic defined in the core. It connects the application with external systems, such as databases and external services.

- **Service Implementations**: This component includes concrete classes that implement the service interfaces defined in the core layer. For instance, `UserServiceImpl` may provide methods for registering users, handling authentication, and managing user profiles. It contains the business logic and orchestrates operations using repositories.
- **Repositories**: Repositories are responsible for data access and persistence. They define methods for interacting with the database, such as saving, retrieving, updating, and deleting records. By abstracting data access logic, repositories enable the application to work with various data sources without affecting core business logic. For example, `UserRepository` and `ProjectRepository` interface with the database, typically extending JPA or similar ORM frameworks.

3. Presentation Layer:

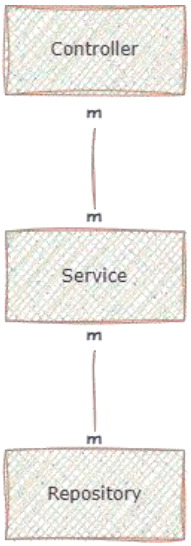
The **Presentation Layer** serves as the interface between the application and its users or external systems. It handles HTTP requests, manages user interactions, and presents data.

- **Controllers**: Controllers are responsible for handling incoming API requests. They process the requests, validate input data, call the relevant service methods, and return appropriate responses. For example, `UserController` handles user-related requests, such as registration and login, while `ProjectController` manages project-related operations.
- **Global Exception Handler**: This component captures and manages exceptions that occur across the application. It provides a centralized way to handle errors, allowing for consistent responses and logging. The global exception handler enhances user experience by returning meaningful error messages rather than exposing stack traces or system errors.
- **Config Package**: The `config` package is responsible for defining various configurations necessary for the application to function correctly.
- **WebSocket :** This component enables real-time communication between clients and the server. It manages WebSocket connections, allowing users to receive live updates and collaborate within the code editor. The WebSocket implementation consists of various handlers for specific actions and interceptors that process or modify messages before they reach their intended handlers. Interceptors enhance flexibility and security.

- **Filters:** The filter package contains components like the `JwtAuthenticationFilter`, which is essential for managing security within the application. This filter intercepts incoming HTTP requests, checks for a valid JWT (JSON Web Token) in the `Authorization` header, and authenticates users accordingly. By managing authentication at this layer, the filter ensures secure access to the application's resources while allowing the Core and Infrastructure layers to focus on business logic and data handling.

- **Detailed Overview of REST Controllers:**

In our application architecture, REST controllers serve as the entry point for handling HTTP requests. Each controller connects with the service layer to process business logic. The service layer, in turn, interacts with the repository layer to perform data access and manipulation. This layered approach ensures a clear separation of concerns, allowing for maintainability and scalability.



- **Detailed Overview of WebSocket Handlers:**

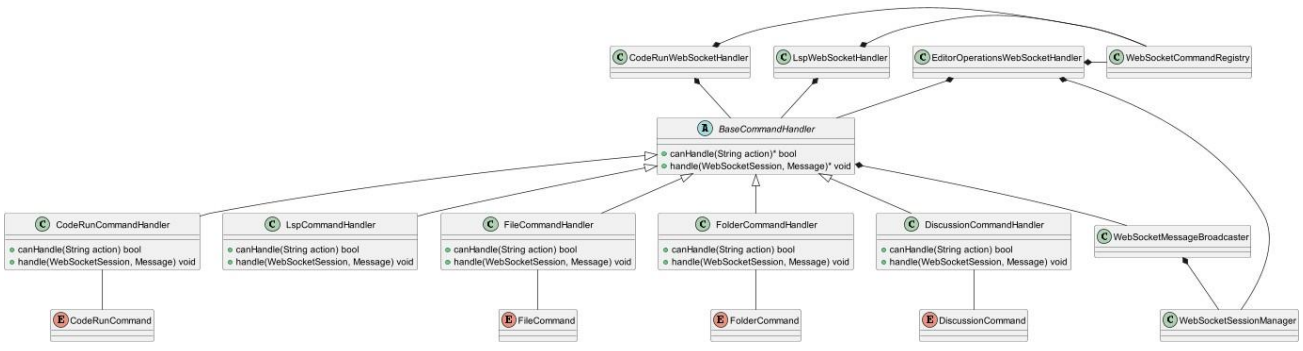
In my application, I am implementing a design that effectively addresses the complexity of handling WebSocket messages by utilizing several design patterns, specifically the Handler Pattern, Command Pattern, Register Pattern, and Factory Pattern. This approach minimizes the reliance on large switch statements within WebSocket handlers, enhancing both clarity and maintainability.

I have designed three primary WebSocket handlers: `CodeRunWebSocketHandler`, `EditorOperationsWebSocketHandler`, and `LspWebSocketHandler`, each responsible for managing distinct types of messages and commands. This segregation of responsibilities promotes cleaner code and simplifies future expansions.

To facilitate the execution of commands, I utilize the Command Pattern through specialized command handlers, such as `FileCommandHandler`, `DiscussionCommandHandler`, and `CodeRunCommandHandler`, each inheriting from a base handler class. A key aspect of this pattern is the use of enums to define commands, encapsulating command strings and providing methods to retrieve the corresponding enum value based on a given string. This promotes type safety and reduces string comparison logic.

To manage these command handlers efficiently, I employ the Register Pattern, which allows for dynamic retrieval of command handlers based on the action specified in incoming messages. This pattern works in tandem with the Factory Pattern, utilized to create instances of command handlers, ensuring that the correct handler is instantiated based on the command being executed.

Overall, this architectural design leverages established design patterns to create a flexible, maintainable, and scalable framework for managing complex WebSocket interactions in real time.



- **BaseCommandHandler:** An abstract class providing a template for handling WebSocket commands, requiring implementations for command handling and action checking.
- **WebSocketMessageBroadcaster:** Manages broadcasting messages to connected WebSocket sessions, facilitating communication between clients in projects and discussions.
- **WebSocketSessionManager:** Handles the management of WebSocket sessions for projects and discussions, allowing for adding and removing sessions efficiently.
- **FileCommandHandler:** A concrete command handler for file-related operations, extending `BaseCommandHandler` to manage create, edit, and delete file commands.
- **FolderCommandHandler:** A command handler that processes folder-related actions like creating, deleting, and retrieving project folder structures via WebSocket messages.
- **LspCommandHandler:** Handles Language Server Protocol (LSP) commands over WebSocket, managing LSP server initialization, communication, and shutdown.
- **DiscussionCommandHandler:** A command handler for managing discussions and comments, extending `BaseCommandHandler` to process discussion-related actions.
- **CodeRunCommandHandler:** Handles code execution commands and interacts with processes, extending `BaseCommandHandler` to manage running and inputting code.
- **CodeRunWebSocketHandler:** A WebSocket handler that manages connections and message parsing specifically for code execution commands, utilizing `CodeRunCommandHandler`.
- **EditorOperationsWebSocketHandler:** Manages WebSocket connections for editor-related operations, routing commands to various handlers, including those derived from `BaseCommandHandler`.
- **LspWebSocketHandler:** Facilitates communication for Language Server Protocol (LSP) operations, routing messages to appropriate command handlers, including those extending `BaseCommandHandler`.
- **WebSocketCommandRegistry:** A registry that keeps track of command handlers and retrieves the appropriate handler based on the action.

3.2 Multithreading: In the Collaborative Code Editor project, effective management of multithreading was crucial to support real-time collaboration among multiple users. Given the concurrent nature of the application, it was essential to ensure that multiple threads could operate simultaneously without causing data inconsistency or corruption.

- **Concurrent Data Structures:** To manage WebSocket connections and sessions effectively, we utilized a **`ConcurrentHashMap`**. This concurrent collection allows for safe and efficient retrieval and updates of user sessions in a multi-threaded environment. The `ConcurrentHashMap` enables multiple threads to read and write data concurrently, minimizing the need for external synchronization while maintaining thread safety. For instance, when a user connects or disconnects, their session information is stored in the `ConcurrentHashMap`, which handles the necessary locking internally. This approach not only simplifies the code but also improves performance by allowing more granular access to the map.
- **Isolation Levels:** To further ensure thread safety and maintain data integrity during concurrent operations, we implemented **serializable isolation levels**. This isolation level is the highest in the context of transaction management and guarantees that transactions are completely isolated from one another. It prevents phenomena such as dirty reads, non-repeatable reads, and phantom reads, ensuring that users always see a consistent view of the data. By adopting serializable isolation, we could effectively manage concurrent file edits and code executions. When multiple users attempt to edit the same file, the system can ensure that only one operation is processed at a time, thereby preventing conflicts and maintaining a consistent state across all user sessions.

3.3 Security: In the Collaborative Code Editor, security was a top priority to ensure that user data and code integrity were maintained throughout the application. The following measures were implemented to provide a secure environment for users:

- **Authentication and Authorization:** To manage user access effectively, we implemented a robust authentication and authorization system. The key components of this system include
  1. **Login and Registration**: Users can create accounts and log in to the application. During registration, we enforced strict validation rules on user input, using annotations like `@NotBlank(message = "Name is required")` in our

Data Transfer Objects (DTOs). This ensures that essential fields are not left empty, enhancing input integrity and user experience.

2. **Token-Based Authentication**: After successful login, the application generates a JSON Web Token (JWT) that is used for subsequent requests. This token is sent back to the client and stored securely. It serves as proof of identity and is included in the Authorization header for protected endpoints, ensuring that only authenticated users can access sensitive operations.
3. **Role-Based Access Control (RBAC)**: The application employs RBAC to enforce permission levels based on user roles (e.g., admin, coder). This means that only users with the appropriate roles can perform specific actions, such as deleting or modifying projects. For example, an editor might have permission to edit files but not to delete them, while an admin would have full access.

- **Access Validation:** To ensure that user's only access projects and features they are authorized to use, we implemented validation checks against user identifiers. Each token contains user details, such as user ID and role, allowing the application to validate permissions effectively. For instance, when a user attempts to delete a project, the system checks their token to verify if they have the necessary role and ownership of the project.
- **WebSocket Security:** Given the real-time nature of the application, securing WebSocket connections was critical. We employed interceptors to validate the JWT token upon establishing a WebSocket connection. This interceptor checks the validity of the token, ensuring that only authorized users can initiate or participate in collaborative sessions. This added layer of security prevents unauthorized access and maintains the integrity of user interactions in real-time.

3.4 File Persistence: In the Collaborative Code Editor, the organization of files is essential for effective project management and user experience. Our file persistence structure is designed to ensure that users can easily manage their projects while providing safety mechanisms for deleted files.

- **Storage Structure :** The file persistence system is organized as follows
  1. **Code Editor Storage**: At the top level, we have a directory named `code-editor-storage`, which serves as the primary storage area for all user-related files and projects.
  2. **User-Specific Folders**: Within the `code-editor-storage`, there are individual folders for each user, named after their respective usernames. This

organization ensures that users can only access their own files, promoting data integrity and security.

3. **Projects Folder**: Each user's folder contains a single **projects folder**, which encompasses two main subfolders:

    - **Code Folder**: This folder holds the current working code for each project, organized by project name. Each project has its specific subfolder, allowing users to edit and collaborate in real time on their ongoing work.
    - **Archive Folder**: This folder is designated for files and folders that have been deleted through a "soft delete" mechanism. When users choose to delete files or folders, they are moved to the archive folder specific to the project instead of being permanently removed. This process ensures that deleted items can be restored if needed, providing a safety net for users' work.

- **Benefits of the Structure**

- **Organized Storage**: Users can easily navigate their projects, with clear separations between current working files and archived materials.
- **Data Recovery**: The archive folder allows users to recover accidentally deleted files, enhancing data integrity and user confidence in managing their work.
- **User-Friendly Interface**: The logical organization of files makes it intuitive for users to manage their projects without confusion, facilitating a smoother collaborative coding experience.

3.5 SOLID Principles:

In developing the Collaborative Code Editor, we prioritized the principles of Clean Code and SOLID design to create a robust, maintainable, and scalable application. Our approach emphasizes clarity, separation of concerns, and adaptability, which are fundamental for effective software engineering.

- **Single Responsibility Principle (SRP):** In our codebase, each class and method follows the single-responsibility principle for clarity and ease of modification. Each component serves a distinct role: `UserController` handles HTTP requests for user-related actions, `UserService` manages business logic for user operations, `UserRepository` interacts directly with the database, and `UserMapper` facilitates conversions between User and `UserDTO` objects.
- **Open/Closed Principle (OCP):** Our architecture supports extensibility by adhering to the Open/Closed Principle, enabling new functionalities without modifying existing code. For example, Service Interfaces like `ProjectService` define core project management methods (create, update, delete), allowing us to add different implementations or features without altering the interface itself. This approach facilitates introducing new capabilities or storage options seamlessly.
- **Liskov Substitution Principle (LSP):** In our implementation of the Liskov Substitution Principle, we ensure that subclasses can replace their superclass without affecting program correctness. This is achieved with `FileCommandHandler` and `FolderCommandHandler`, both inheriting from a shared `BaseCommandHandler` class. This setup enables us to use either handler interchangeably; for example, methods accepting a `BaseCommandHandler` parameter can work seamlessly with `FileCommandHandler` or `FolderCommandHandler`, preserving expected behavior across the application.
- **Interface Segregation Principle (ISP):** emphasizes that clients should not be forced to depend on interfaces they do not use. In our design, we achieve this through granular interfaces, such as `FileService` for file operations and `UserService` for user management. This approach ensures that classes only implement the interfaces relevant to their specific functionalities, thereby minimizing unnecessary dependencies and fostering clearer contracts.

- **Dependency Inversion Principle (DIP):** (DIP) states that high-level modules should not depend on low-level modules; instead, both should rely on abstractions. We implement this through Dependency Injection in the `UserController`, where `UserService` is injected via the constructor instead of being instantiated directly. This practice enhances testability by allowing for service mocking during unit tests and decouples the controller from the specific implementation of the service.

3.6 Clean Code

- **Meaningful Names:** Use clear and descriptive names for classes, methods, and variables. For example, the `AuthService` clearly indicates its role in handling authentication logic, making it immediately understandable to developers.
- **Avoiding Code Duplication:** Mappers can centralize the conversion logic, preventing duplication of mapping code in various service methods. This promotes maintainability and adheres to the DRY (Don't Repeat Yourself) principle.
- **Readable Code:** The use of enums for specific commands, like `FileCommand`, enhances readability by clearly defining the allowed actions in a structured manner. For example, having commands like `CREATE_FILE`, `DELETE_FILE`, and `EDIT_FILE` makes it evident what operations are possible, reducing the need for additional comments or documentation.
- **Meaningful Error Messages:** Our global exception handler provides clear and descriptive error messages, such as for `ResourceNotFoundException`. This approach improves user experience and aids developers in diagnosing issues, contributing to a more robust application.
- **Functions Should Do One Thing:** Each method in our service classes (e.g., `UserService`, `ProjectService`, `FileService`) is designed to perform a single task, ensuring clarity and maintainability. This separation of concerns makes it easy for developers to understand the functionality without navigating through unnecessary complexity.

3.7 Design patterns:

- **Singleton Pattern:** This pattern is used across services, controllers, and repositories to ensure that only one instance of each component is created, reducing memory consumption and avoiding redundant object instantiation. This choice enhances the efficiency of commonly used components, such as `AuthService` and `ProjectRepository`, making them thread-safe and accessible across the application.
- **Builder Pattern:** For complex model creation, the builder pattern is applied to streamline object construction. By using builders with classes like `Project` and `User`, this pattern enhances code readability and flexibility, especially when creating objects with multiple optional fields.
- **Factory Pattern:** The factory pattern is implemented in components like `CodeExecutorFactory`, where it determines the appropriate `CodeExecutor` based on the selected programming language. This factory centralizes instantiation logic, allowing for a clean and extendable way to support multiple languages. In the `FileCommand` logic, the factory similarly encapsulates command creation, improving modularity.
- **Handler Pattern:** The handler pattern is applied in processing various command operations within the file and folder functionalities, allowing each command type to have its own handler. This separation simplifies the addition of new commands and keeps handlers focused on specific tasks, maintaining single responsibility for each handler.
- **Registry Pattern:** The registry pattern is used to manage instances of command handlers and other key components, making them accessible throughout the application. This pattern enables efficient access and retrieval of components, promoting a clean and modular design while reducing dependencies on hardcoded references.

3.8 Effective Java:

- **Favor Composition Over Inheritance:** we applies composition by using services, repositories, and other components as dependencies within other classes, avoiding tight coupling. Rather than relying on inheritance, you achieve flexibility by injecting components into classes where needed.
- **Use Interfaces to Define Types:** Interfaces are used for service classes (e.g., `ProjectService` or `UserService`), allowing flexible implementations and making it easy to swap or mock service implementations, which also aids testing and adherence to SOLID principles.
- **Minimize Mutability:** By making DTOs and certain key classes immutable (using `final`), you safeguard them from unintended changes. Immutable objects are easier to reason about and reduce concurrency issues.
- **Favor Exceptions Over Returning Error Codes:** Your application consistently uses exceptions for error handling, like `ResourceNotFoundException`, making error management more expressive and simplifying debugging, rather than using error codes.
- **Consider Factories Instead of Constructors:** The factory pattern is implemented in components such as `CodeExecutorFactory`, where you create executors based on the language selected. This allows for better extensibility and abstraction over instantiation logic.
- **Always Override `toString`:** For DTOs and model classes, you use records or overrides for `toString`, enabling better logging and debugging since these implementations can provide clear, structured output of object states.
- **Minimize the Accessibility of Classes and Members:** By limiting access to components, such as marking classes or methods as `private` or `protected` where applicable, you encapsulate internal workings, making the code more secure and easier to refactor without unintended interactions.
- **In Public Classes, Use Accessor Methods, Not Public Fields:** Instead of exposing fields directly, your classes use getters and setters, which promotes encapsulation, ensuring data consistency and adherence to object-oriented principles.
- **Use Enum Instead of `int` Constants:** Enums are used to define constants like user roles, project states, or command types, making your code more readable, type-safe, and less error-prone compared to `int` constants.

4. DevOps

4.1 Docker: In our project, we are utilizing Docker to create a containerized environment, which significantly enhances the development, deployment, and scalability of our application. Specifically, we have designed three separate containers: one for the frontend, one for the backend, and one for the database (MariaDB). This separation of concerns allows us to manage each component independently while ensuring that they can communicate effectively.

- **Frontend Container:** The **frontend container** is responsible for serving the user interface of our application, which is built using **Angular**. This container includes all the necessary files, libraries, and dependencies required to run the Angular code. We use **Nginx** as a lightweight web server to serve the built static assets.
- **Backend Container:** The **backend container** houses the server-side logic of our application. It includes APIs, business logic, and database interactions. This container runs a web server or framework (such as Spring Boot for Java or Express for Node.js) and is equipped with **Docker IO** installed to facilitate running Docker within Docker (DinD).