Project Description

The Collaborative Code Editor is a sophisticated web application designed to facilitate real-time collaborative coding among multiple users. Developed primarily using Java with the Spring Boot framework, the application emphasizes robust back-end development and adheres to advanced software engineering principles. The front-end is built using HTML, CSS, and JavaScript (ReactJS), providing an intuitive and responsive user interface. Key features include real-time code editing, version control, user authentication via OAuth 2.0, role-based access control, and the ability to execute code in various programming languages using Docker containers. The project also incorporates DevOps practices, including CI/CD pipelines and Docker containerization, ensuring seamless deployment and scalability on AWS.

Objective

The objective of this report is to provide a comprehensive overview of the Collaborative Code Editor project, highlighting its adherence to clean code principles, Effective Java practices, SOLID design principles, and the implementation of various design patterns. The report will also discuss how the project handles multithreading, ensuring thread safety and concurrency management. Additionally, it will cover the DevOps practices employed, including Docker containerization, AWS implementation, and CI/CD pipeline setup, which contribute to the project's scalability and maintainability.

Application Workflow

Starting the Application

First, navigate to your project directory in the terminal. This directory should contain the necessary `docker-compose.yml` file for the application.

Run the following Docker command to build and start the entire application stack:

```
docker-compose up --build
```

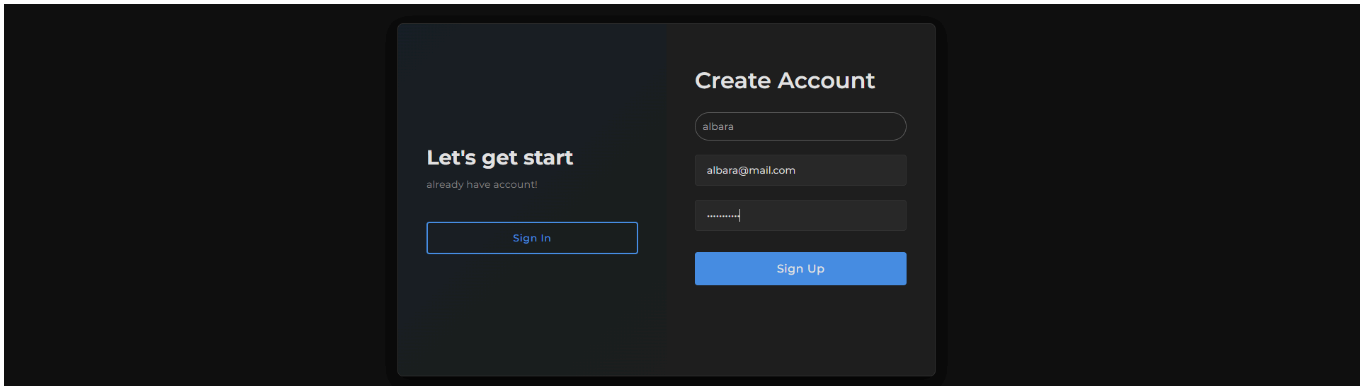
This command will build all required Docker images and launch the necessary services such as the frontend, backend, and database containers.

Once the services are up and running, open your browser and go to:

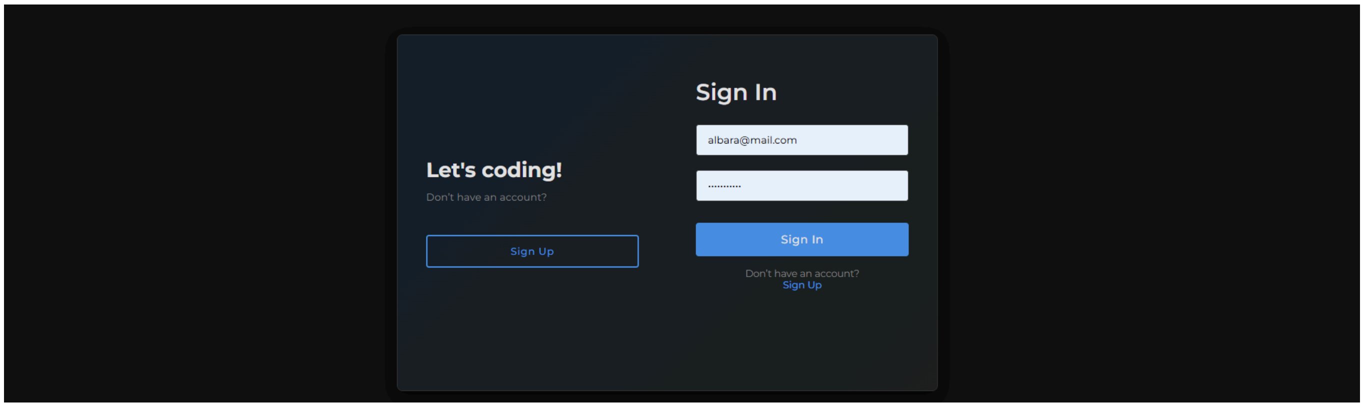
```
http://localhost:3000/
```

Sign Up / Sign In

Upon visiting the application in your browser, you'll be greeted with a **Sign Up** page.
If you are a new user, you can create an account by providing your email and a secure password.

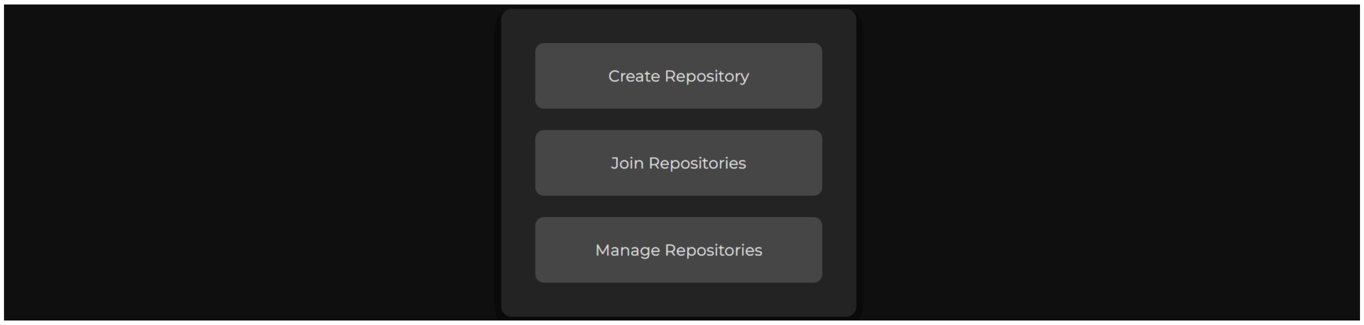


If you already have an account, simply click on the **Sign In** option and enter your credentials to log in.



Home Page Navigation

After signing in, you will be directed to the **Home Page**, which provides access to the core features of the application:

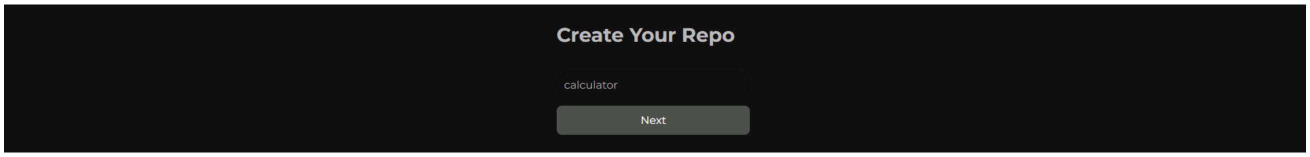


- **Create Repository**
- **Join Repository**
- **Manage Repository**

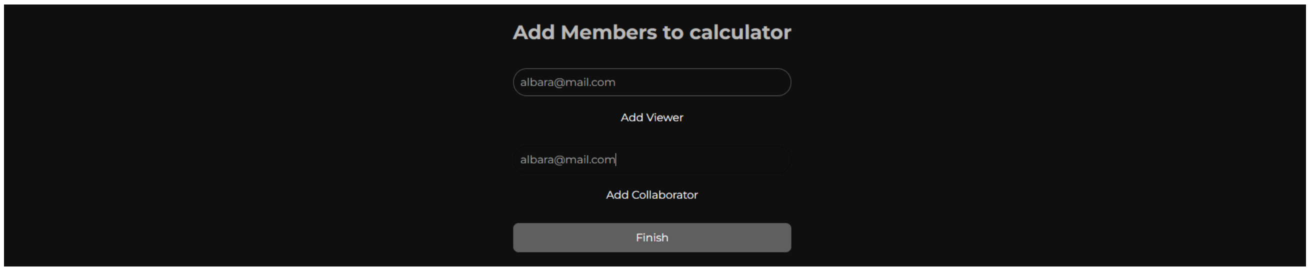
Creating a Repository

If you click on **"Create Repository"**, the application will prompt you to:

- Enter a name for your new repository.



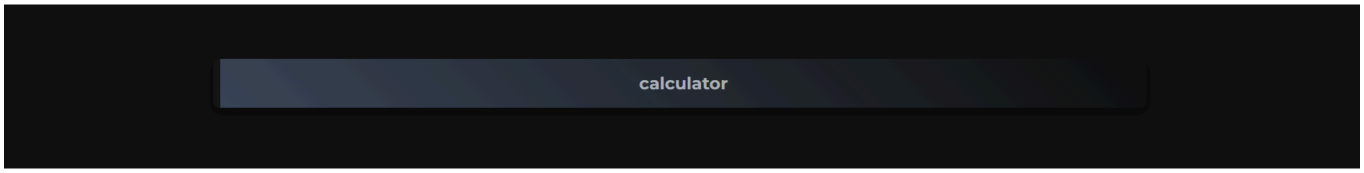
- Optionally, add **viewers** or **collaborators** by entering their email addresses.
- Click **Add Viewer** or **Add Collaborator** to grant them access.



This sets up your collaborative coding environment with access controls in place.

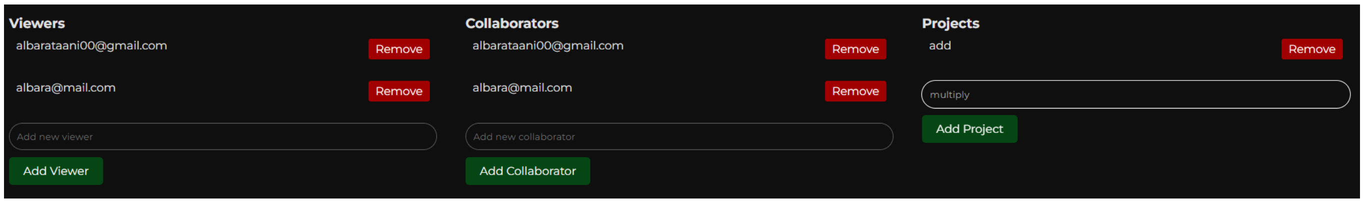
Managing a Repository

To work on a repository you own or have access to, click on **"Manage Repository"** from the home page.



Here, you can:

- Select any of your repositories.
- Add new **projects** under a repository.
- Assign or invite more **editors** and **viewers**.
- Manage existing collaborators.



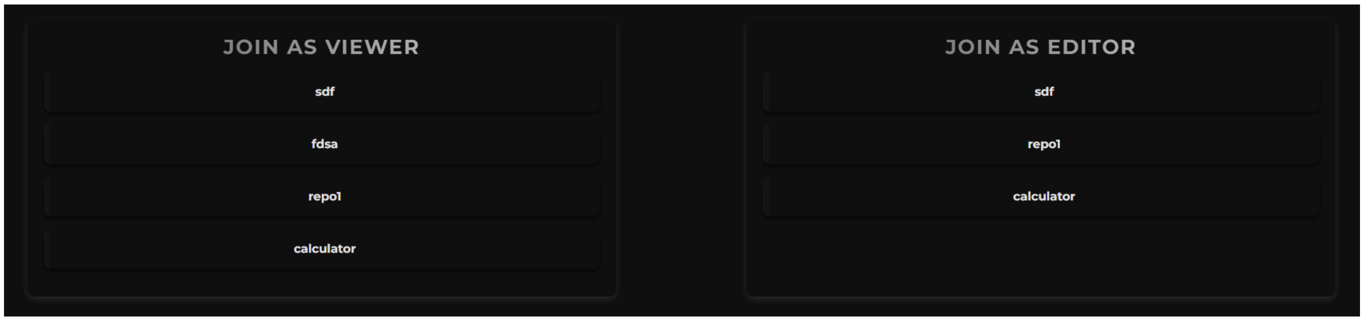
This section is essential for maintaining your workspace and permissions.

Joining a Repository

If you've been invited to a repository or want to collaborate on an existing one, choose **"Join Repository"** from the home page.

You'll be able to:

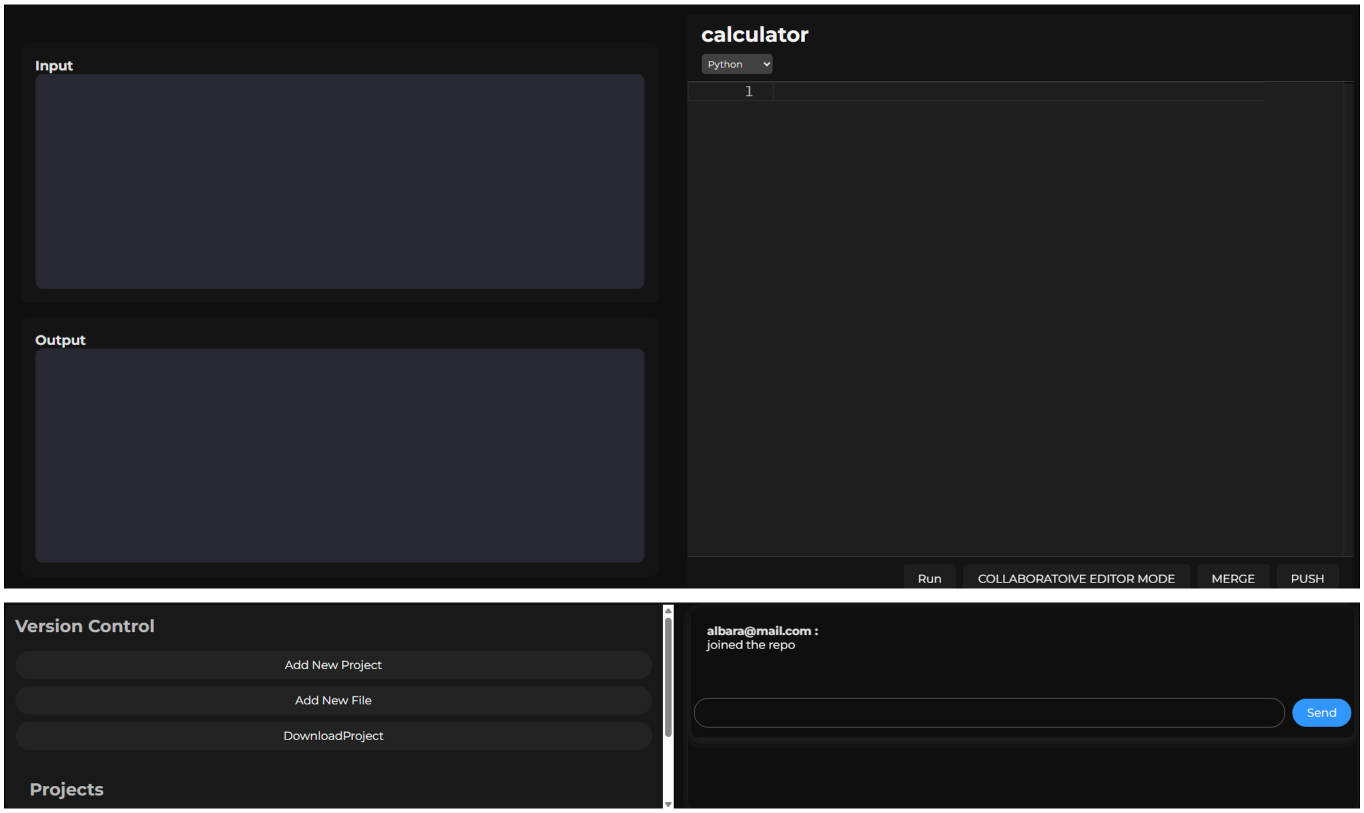
- Browse available repositories.
- Join as a **viewer** (read-only access) or **editor** (write access).



Once joined, you will be redirected to the **Code Page**, where collaboration happens in real time.

The Code Page – Your Collaborative Workspace

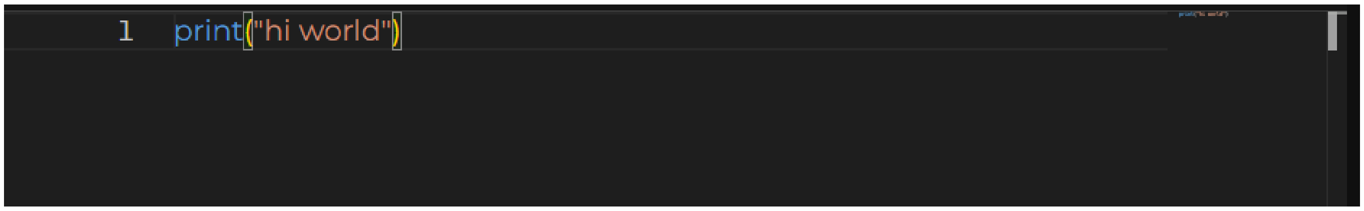
The code page is where the actual coding and collaboration takes place. It's divided into four key sections:



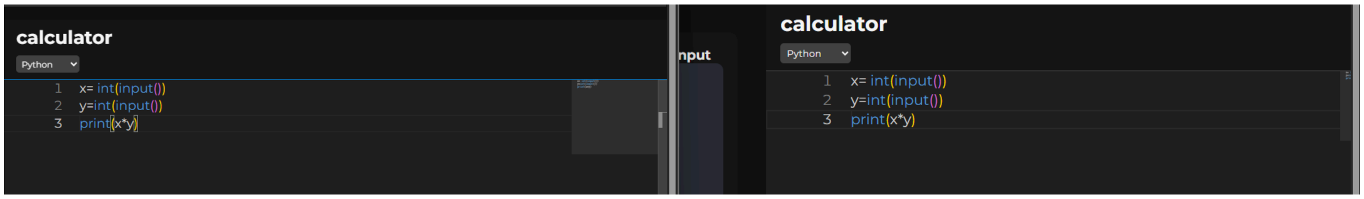
albara@mail.com :
joined the repo

1. Code Editor

This is the main area where you write your code.


```
print("hi world")
```

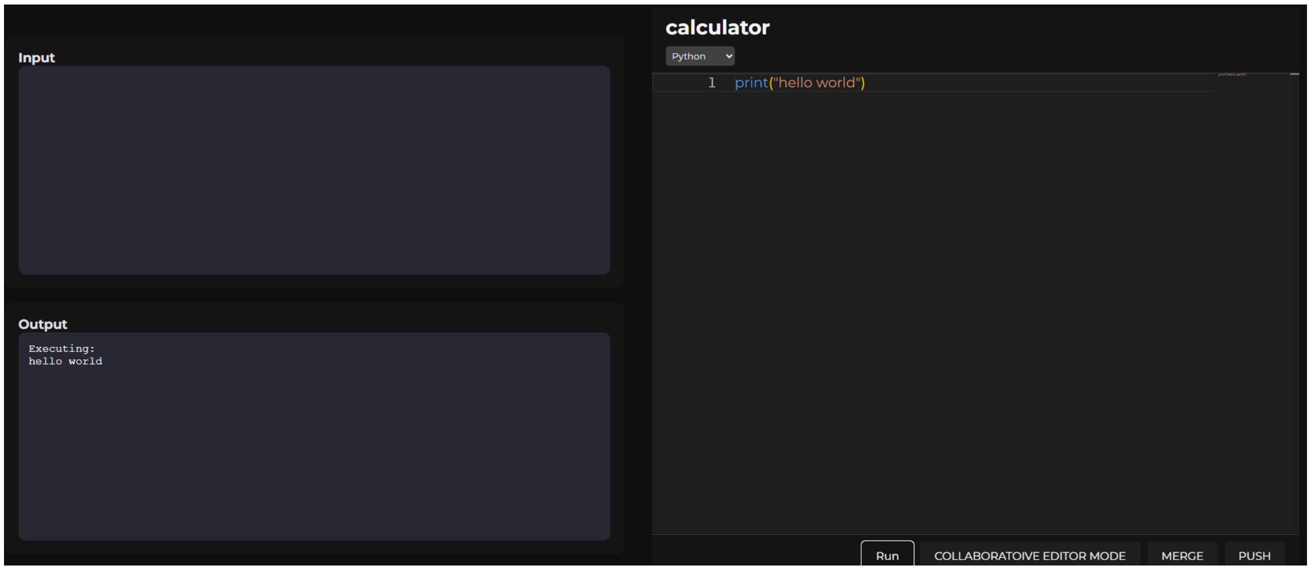
If **Editor Mode** is enabled, changes you make will be visible in real-time to other collaborators working on the same file.



```
x= int(input())
y=int(input())
print(x*y)
```

2. Input and Output Boxes

- Here, you can enter input values for your code.
- After clicking **Run**, the output will be displayed in the output box.
- Great for testing functions and logic on the spot.



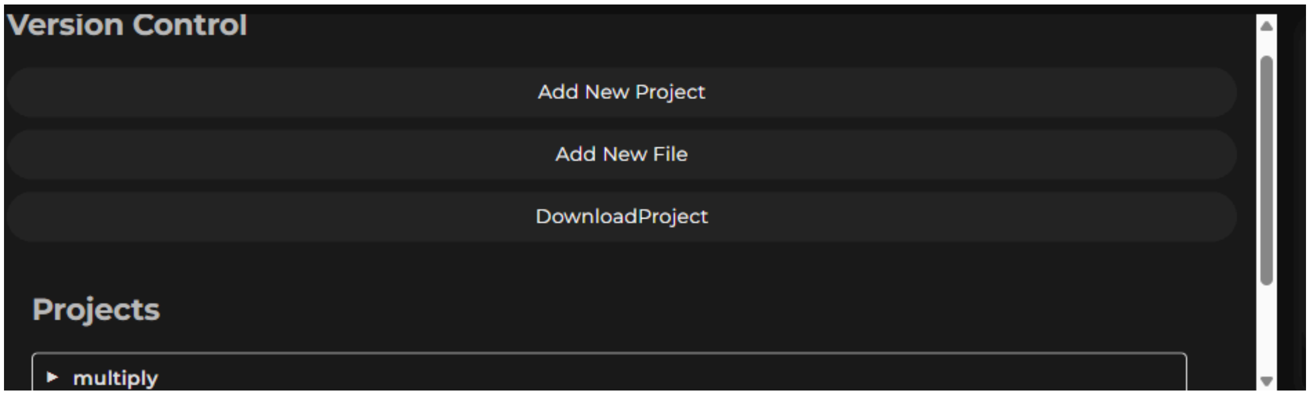
```
print("hello world")
```

```
Executing:
hello world
```

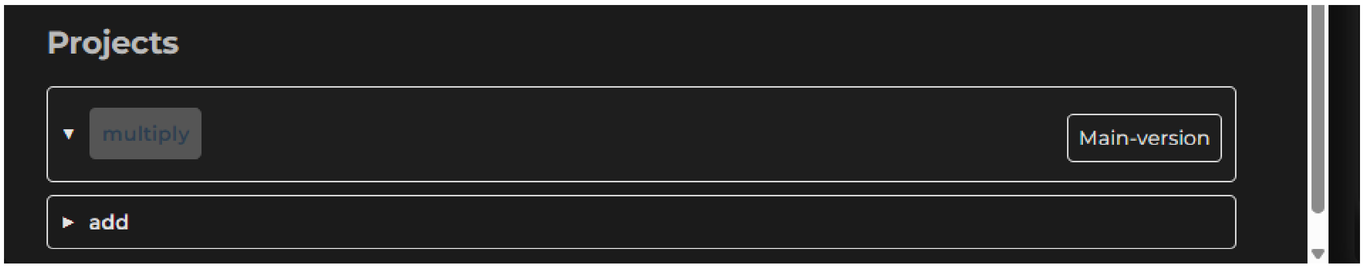
3. Version Control Section

This section helps you manage files and updates:

- **Add Project** or **Add File** to expand your repository.



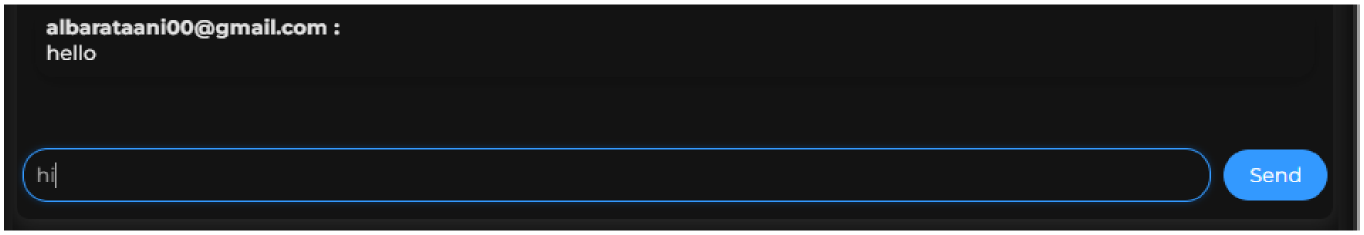
- Use **Pull** to fetch the latest updates from the remote repository and sync your workspace.



4. Chat Box

Collaborate and communicate with team members in real-time:

- Discuss changes.
- Share ideas.
- View activity logs and recent updates to the code.



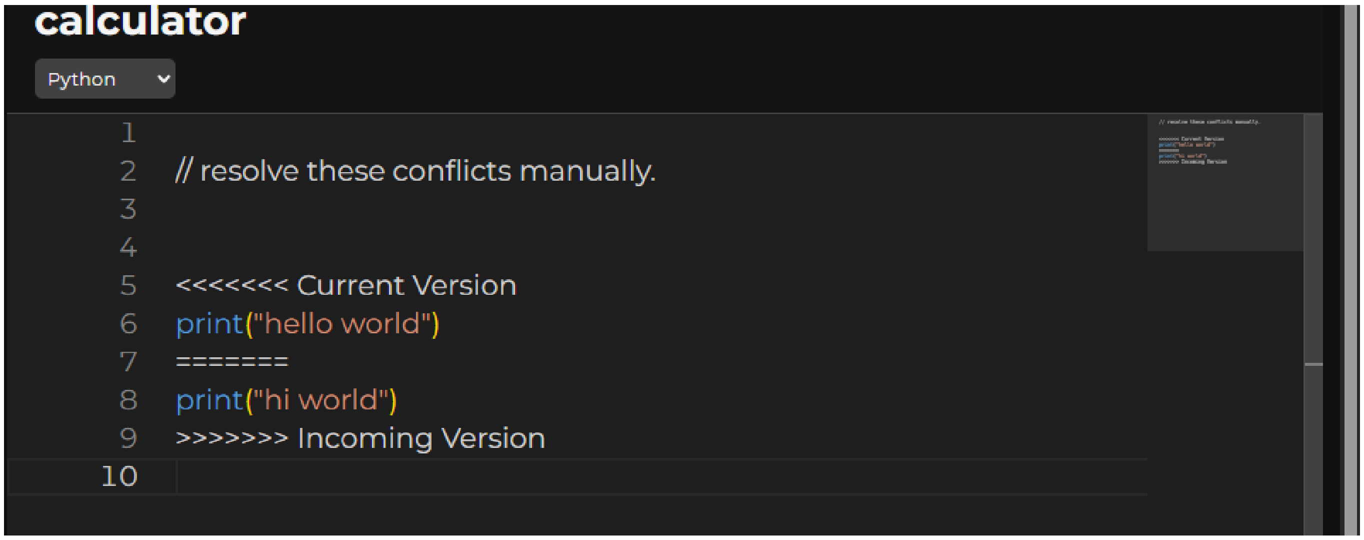
albarataani00@gmail.com :
hello

hi

Push and Merge

- To save your latest work to the shared repository, click on the **Push** button.
- If there are changes from others that you haven't incorporated yet, click **Merge** to combine your current changes with the latest version.

The merge process is smartly structured, so you'll see a side-by-side comparison of:



```

// resolve these conflicts manually.


<<<<<<< Current Version
print("hello world")
=======
print("hi world")
>>>>>>> Incoming Version

```

- Your current code.
- The last saved version.

This helps to resolve conflicts easily and maintain clean, collaborative code.

Clean Code

```
@Service("ProjectServiceImpl")

public class ProjectServiceImpl implements ProjectService {

    private final Map<String, Object> projectLocks = new ConcurrentHashMap<>
();

    @Autowired
    private FileService fileService;

    @Autowired
    private ProjectDownloadService projectDownloadService;
    private final ProjectRepository projectRepository;
    private final RepoRepository repoRepository;
    private final FileRepository fileRepository;

    @Override
    @Transactional
    public void createProject(ProjectDTO project) {...}

    private Project buildNewProject(ProjectDTO project, Repo repo) {...}
    private void createDefaultFile(ProjectDTO project) {...}

    @Override
    public void deleteProject(ProjectDTO projectDTO) {...}

    @Override
    public List<ProjectDTO> getProjects(String repoId) {
        Repo repo = repoRepository.findByRepoId(repoId)
                .orElseThrow(() -> new RepositoryNotFoundException("Repo not
found for ID " + repoId));
        return repo.getProjects().stream()
                .map(project -> new ProjectDTO(project.getName(), repoId))
                .collect(Collectors.toList());
    }

    @Override
    public void downloadProject(ProjectDTO projectDTO, HttpServletResponse
response) throws IOException {...}

    private List<File> findFilesByProjectAndRepo(String projectName, String
repoId) {...}
}
```

1. Names (Chapter 2)

Descriptive and Intention-Revealing Names

- **Class/Interface Names**:
  - `ProjectServiceImpl` clearly indicates it's a service implementation for project-related operations.
  - DTOs like `ProjectDTO` and `FileDTO` signal they are data transfer objects.
- **Variables**:
  - `projectRepository`, `repoRepository`, and `fileRepository` explicitly describe their purposes (e.g., database interactions).
  - Constants like `MAIN_VERSION` and `DEFAULT_EXTENSION` are self-documenting.
  - `projectLocks` (a `ConcurrentHashMap`) clearly indicates it manages concurrency locks for projects.
- **Methods**:
  - `createProject`, `deleteProject`, `getProjects`, and `downloadProject` are verbs that directly describe actions.

    - Helper methods like `buildNewProject` and `createDefaultFile` reveal their specific responsibilities.

Avoid Disinformation

- No abbreviations or ambiguous terms (e.g., `repoId` is clear, not `rid`).
- `ProjectNotFoundException` and `RepositoryNotFoundException` unambiguously describe error conditions.

Consistency

- Uses `repoId` and `projectName` consistently across parameters and method calls (e.g., `findByRepoIdAndProjectName`).
- `ProjectDTO` and `FileDTO` follow a consistent naming pattern for data transfer objects.

---

2. Functions (Chapter 3)

Small and Single Responsibility

- **Short Methods**:
    - Most methods are concise (e.g., `getProjects`, `downloadProject`).
    - Complex logic is delegated to helper methods like `buildNewProject` and `createDefaultFile`.
- **Single Responsibility**:
    - `createProject` handles project creation, while `deleteProject` focuses on deletion. Neither mixes concerns.
    - `buildNewProject` isolates the object-construction logic.
    - `getProjectLock` encapsulates lock management for concurrency.

Descriptive Function Signatures

- **Parameters**:
    - `ProjectDTO projectDTO` and `HttpServletResponse response` clearly define inputs.
    - Avoids overly generic types (e.g., `String repoId` instead of `String id`).
- **Return Types**:
    - `List<ProjectDTO>` in `getProjects` signals a collection of DTOs.
    - `void` for actions like `deleteProject` where no return value is needed.

Error Handling

- **Exceptions**:
  - Specific exceptions like `ProjectNotFoundException` are thrown instead of generic ones.
  - Wraps lower-level exceptions in meaningful messages (e.g., `throw new RuntimeException("Failed to create project...")`).

Abstraction Level

- **High-Level Flow**:
  - `createProject` reads like a recipe: validate repo → build project → save → create default file.
  - `downloadProject` delegates file retrieval and downloading to other services (`projectDownloadService`).
- **Low-Level Details**:
  - Helper methods (e.g., `findFilesByProjectAndRepo`) hide database interaction details.
  - Concurrency control (`synchronized (getProjectLock(fileKey))`) is isolated in `deleteProject`.

Avoid Side Effects

- **Thread Safety**:
  - Uses `synchronized` blocks with project-specific locks to prevent race conditions in `deleteProject`.
  - `ConcurrentHashMap` ensures thread-safe access to `projectLocks`.
- **Transactional Boundaries**:
  - Annotates `createProject` with `@Transactional` to maintain database consistency.

Avoiding Duplication – DRY Principle.

Duplication in code can lead to maintenance challenges and bugs. The code avoids duplication by using functions and abstractions to encapsulate common logic.

Example: `FileDTO.java`

Copy

```
@Data
@Getter
```

```java
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class FileDTO {
    private String filename;
    private String repoId;
    private String projectName;
    private String extension;
}
```

The `FileDTO` class uses Lombok annotations to avoid duplicating boilerplate code for getters, setters, and constructors. This makes the class definition cleaner and easier to maintain.

Comments and Documentation

Avoid unnecessary comments, use comments to explain complex decisions.
there is a lack of comments. This is intentional
I believe that when code adheres to SOLID principles, Clean Code practices, and the guidelines from *Effective Java*, it becomes self-explanatory, making most comments redundant.

Effective Java

Consider Static Factory Methods Instead of Constructors

Static factory methods provide more flexibility and control over object creation compared to constructors. They can return an existing instance, a subclass, or even null, which is not possible with constructors.

Defense

Dependency Injection: Classes such as `JwtUtil`, `SecurityConfig`, and `RepoController` rely on Spring's framework for instantiation and management of dependencies. This approach avoids the direct use of constructors and aligns with the principle by using a factory method (Spring's `@Bean` and `@Component` annotations) for object creation.

Appropriate Use of Constructors: The code does use constructors, but in appropriate places. For example, constructors in the `Project`, `MessageLog`, `Repo`, `RepoMembership`, `User`, and `CodeUpdate` classes are simple and necessary for initializing entities. For more complex object creation, the code follows a builder pattern (e.g., `ProjectDTO.builder()`), which is aligned with the principle.

Constructor Injection: The classes (e.g., `SignInController`, `SignUpController`, `UserController`, `FileController`, etc.) use constructor injection for their dependencies (such as `AuthenticationService`, `UserServiceImpl`, and `UserService`). This follows the principles of dependency injection.

Example: `RepoController.java`

```java
@RestController
@RequestMapping("/api/repos")
public class RepoController {

    private final RepoService repoService;

    @Autowired
    public RepoController(RepoService repoService) {
        this.repoService = repoService;
    }

    @PostMapping("/create")
    public ResponseEntity<String> createRepo(@RequestBody RepoDTO repoDTO) {
        repoService.createRepo(repoDTO);
        return ResponseEntity.ok("Repo created successfully");
    }
}
```

The `RepoController` class uses Spring's `@Autowired` annotation to inject the `RepoService` dependency. This approach avoids the direct use of constructors and aligns with the principle of using static factory methods for object creation.

Minimize the Accessibility of Classes and Members

Minimizing the accessibility of classes and members is crucial for maintaining encapsulation and hiding the internal representation of a class. This practice enhances the modularity and maintainability of the code.

Defense

Encapsulation: The controllers are appropriately marked as private, such as the `authenticationService` in the `SignInController`. This is in line with the principle of encapsulation and keeping the class's internal representation hidden.

Example: `SignInController.java`

```
@RestController
@RequestMapping("/api/auth")
public class SignInController {

    private final AuthenticationService authenticationService;

    @Autowired
    public SignInController(AuthenticationService authenticationService) {
        this.authenticationService = authenticationService;
    }

    @PostMapping("/signin")
    public ResponseEntity<String> signIn(@RequestBody LoginRequest
loginRequest) {
        authenticationService.authenticate(loginRequest);
        return ResponseEntity.ok("Signed in successfully");
    }
}
```

The `SignInController` class encapsulates the `authenticationService` dependency, marking it as private. This practice hides the internal representation of the class and aligns with the principle of minimizing the accessibility of classes and members.

Prefer Dependency Injection to Hardwiring Resources

Dependency injection promotes loose coupling and improves testability by allowing the injection of different implementations of a dependency.

Defense

Dependency Injection: The controller classes, like `UserController`, `FileController`, and services like `DockerExecutorService`, all depend on Spring's `@Autowired` or constructor-based dependency injection.

Example: `UserController.java`

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    private final UserService userService;
    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/{userId}")
    public ResponseEntity<UserDTO> getUserById(@PathVariable String userId) {
        UserDTO userDTO = userService.getUserById(userId);
        return ResponseEntity.ok(userDTO);
    }
}
```

The `UserController` class uses Spring's `@Autowired` annotation to inject the `UserService` dependency. This approach promotes loose coupling and improves testability by allowing the injection of different implementations of the `UserService`.

Try-with-Resources to try-finally

The `try-with-resources` statement is a cleaner and more concise way to handle resource management, ensuring that resources are closed automatically.

Defense

Resource Management: The `DockerExecutorService` loads a command template via an input stream. However, this could be enhanced using a `try-with-resources` statement instead of relying on implicit closure.

Example: `DockerExecutorService.java`

```
@Service
public class DockerExecutorService {

    public String executeCommand(String command) {
        ProcessBuilder processBuilder = new ProcessBuilder(command.split("
"));
        try (BufferedReader reader = new BufferedReader(new
InputStreamReader(processBuilder.start().getInputStream()))) {
            StringBuilder output = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                output.append(line).append("\n");
            }
            return output.toString();
        } catch (IOException e) {
            throw new RuntimeException("Failed to execute command", e);
        }
    }
}
```

The `DockerExecutorService` class uses a `try-with-resources` statement to manage the `BufferedReader` resource. This ensures that the resource is closed automatically, aligning with the principle of using `try-with-resources` instead of `try-finally`.

In Public Classes, Use Accessor Methods, Not Public Fields

Using accessor methods (getters and setters) instead of public fields promotes encapsulation and allows for better control over how the fields are accessed and modified.

Defense

Accessor Methods: The entire classes I used private fields with `@Getter` and `@Setter` annotations.

Example: `User.java`

```java
package com.collaborative.editor.dto;
import lombok.*;
import javax.persistence.*;
import java.util.List;

@Data
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String email;
    private String password;
    private String role;
    @OneToMany(mappedBy = "user")
    private List<RepoMembership> repoMemberships;
}
```

The `User` class uses private fields with `@Getter` and `@Setter` annotations to generate accessor methods. This practice promotes encapsulation and aligns with the principle of using accessor methods instead of public fields.

Favor Composition Over Inheritance

Composition is generally more flexible and promotes better design than inheritance. It allows for the creation of more modular and reusable code.

Defense

Composition: Entity relationships (e.g., `RepoMembership` to `Repo` and `User`) are represented with composition, not inheritance. The relationships are modeled with `@ManyToOne` and `@OneToMany`, which reflects a correct design choice.

Example: `RepoMembership.java`

```java
package com.collaborative.editor.dto;
import lombok.*;
import javax.persistence.*;


@Data
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
public class RepoMembership {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "repo_id")
    private Repo repo;
    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;
    private String role;
}
```

The `RepoMembership` class uses composition to model the relationship between `Repo` and `User`. This practice promotes better design and aligns with the principle of favoring composition over inheritance.

Prefer Interfaces Over Abstract Classes

Interfaces are generally more flexible than abstract classes because they allow for multiple inheritance and provide a clear contract for implementing classes.

Defense

Interfaces: Interfaces are preferable to abstract classes because Java only allows single inheritance. Therefore, I used interfaces in most of the services.

Example: `UserService.java`

```
public interface UserService {
    UserDTO getUserById(String userId);
    void createUser(UserDTO userDTO);
    void updateUser(String userId, UserDTO userDTO);
    void deleteUser(String userId);
}
```

The `UserService` interface defines a contract for the `UserServiceImpl` class. This practice promotes flexibility and aligns with the principle of preferring interfaces over abstract classes.

Use Enums Instead of Int Constants

Enums provide a type-safe way to represent fixed sets of constants, enhancing code clarity and type safety.

Defense

Enums: The `RepoRole` enum in methods like `addMember` and `removeMember` is excellent. It enhances code clarity and type safety.

Example: `RepoRole.java`

```
public enum RepoRole {
    ADMIN,
    COLLAPORATOR3,
    VIEWER;
}
```

The `RepoRole` enum represents a fixed set of user roles in the application. This practice enhances code clarity and type safety, aligning with the principle of using enums instead of int constants.

Prefer Lambdas to Anonymous Classes

Lambdas provide a cleaner and more concise way to implement functional interfaces compared to anonymous classes.

Defense

Lambdas: I used lambdas quite often, especially when working with streams.

Example: `CodeMetricsDisplay.java`

```
import java.util.List;
import java.util.stream.Collectors;

public class CodeMetricsDisplay {
    public List<String> getFunctionNames(List<String> codeLines) {
        return codeLines.stream()
                .filter(line -> line.contains("def "))
                .map(line -> line.split("def ")[1].split("\\(")[0])
               .collect(Collectors.toList());
    }
}
```

The `getFunctionNames` method uses a lambda expression to filter and map lines of code. This practice makes the code more concise and expressive, aligning with the principle of preferring lambdas to anonymous classes.

Document All Exceptions Thrown by Each Method

Documenting exceptions thrown by each method helps users understand the potential errors that can occur and how to handle them.

Defense

Centralized Exceptions: I used them quite often, especially when working with the `CentralizedExceptions`.

Example: `UserServiceImpl.java`

```java
@Service
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;
    @Autowired
    public UserServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDTO getUserById(String userId) {
        return userRepository.findById(userId)
                .orElseThrow(() -> new UserNotFoundException("User not found
with ID: " + userId));
    }
}
```

The `getUserById` method throws a `UserNotFoundException` if the user is not found. This practice documents the potential exception and aligns with the principle of documenting all exceptions thrown by each method.

Check Parameters for Validity

Checking parameters for validity helps prevent errors and ensures that the method behaves as expected.

Defense

Parameter Validation: For example, validating that `LoginRequest` and `User` objects are not null and that essential fields are populated could be done before proceeding with further logic.

Example: `AuthenticationService.java`

```
@Service
public class AuthenticationService {
    public void authenticate(LoginRequest loginRequest) {
        if (loginRequest == null || loginRequest.getUsername() == null ||
loginRequest.getPassword() == null) {
            throw new IllegalArgumentException("Login request is invalid");
        }
        // Authentication logic
    }
}
```

The `authenticate` method checks if the `LoginRequest` object and its essential fields are not null before proceeding with the authentication logic. This practice helps prevent errors and ensures that the method behaves as expected.

Optimize Method Signatures

Method signatures should be optimized with minimal parameters to enhance readability and maintainability.

Defense

Minimal Parameters: This is especially true for the methods handling simple requests like `signIn`, `createAccount`, and `runCode`.

Example: `UserController.java`

```
@PostMapping("/signin")
public ResponseEntity<String> signIn(@RequestBody LoginRequest loginRequest) {
    authenticationService.authenticate(loginRequest);
    return ResponseEntity.ok("Signed in successfully");
}
```

The `signIn` method has a minimal parameter list, enhancing readability and maintainability. This practice aligns with the principle of optimizing method signatures.

Synchronize Access to Shared Mutable Data

Synchronizing access to shared mutable data ensures thread safety and prevents concurrent modification exceptions.

Defense

Synchronized Access: Using the `fileLocks` map and the `synchronized` keyword in methods like `pushFileContent` and `mergeFileContent`.

Example: `FileServiceImpl.java`

```java
@Service
public class FileServiceImpl implements FileService {

    private final Map<String, Object> fileLocks = new ConcurrentHashMap<>();

    @Override
    public void pushFileContent(String fileId, String content) {
        synchronized (getLock(fileId)) {
            // Push file content logic
        }
    }

    @Override
    public void mergeFileContent(String fileId, String content) {
        synchronized (getLock(fileId)) {
            // Merge file content logic
        }
    }

    private Object getLock(String fileId) {
        return fileLocks.computeIfAbsent(fileId, k -> new Object());
    }
}
```

The `pushFileContent` and `mergeFileContent` methods use the `synchronized` keyword to ensure thread-safe access to shared mutable data. This practice prevents concurrent modification exceptions and aligns with the principle of synchronizing access to shared mutable data.

SOLID Principles

Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) states that a class should have only one reason to change, meaning it should have only one job or responsibility. This principle promotes cohesion and reduces the likelihood of unintended side effects from changes.

Example: `EditorServiceImpl.java`

```
@Service
public class EditorServiceImpl implements EditorService {

    @Override
    public CodeMetrics calculateMetrics(String code, String language) {
        int lines = code.split("\\n").length;
        int functions = countFunctions(code, language);
        int variables = countVariables(code, language);
        int cyclomaticComplexity = calculateCyclomaticComplexity(code,
language);
        return new CodeMetrics(lines, functions, variables,
cyclomaticComplexity);
    }

    private int countFunctions(String code, String language) {
        switch (language.toLowerCase()) {
            case "java":
                return
code.split("\\bpublic\\b|\\bprivate\\b|\\bprotected\\b").length - 1;
            case "python":
                return code.split("\\bdef\\b").length - 1;
            default:
                return 0;
        }
    }

    private int countVariables(String code, String language) {
        switch (language.toLowerCase()) {
            case "java":
                return
code.split("\\bint\\b|\\bString\\b|\\bdouble\\b|\\bboolean\\b").length - 1;
            case "python":
                return code.split("\\b=\\b").length - 1;
            default:
                return 0;
        }
    }

    private int calculateCyclomaticComplexity(String code, String language) {
        String[] controlFlowKeywords;
        switch (language.toLowerCase()) {
            case "java":
                controlFlowKeywords = new String[] { "if", "else", "for",
"while", "switch", "case", "catch" };
                break;
            case "python":
                controlFlowKeywords = new String[] { "if", "elif", "else",
"for", "while", "try", "except" };
                break;
            default:
                controlFlowKeywords = new String[] {};
        }

        int complexity = 1;
        for (String keyword : controlFlowKeywords) {
            complexity += code.split("\\b" + keyword + "\\b").length - 1;
        }

        return complexity;
    }
}
```

The `EditorServiceImpl` class adheres to SRP by having separate methods for different responsibilities. For instance, `calculateMetrics` is responsible for calculating code metrics, while `countFunctions`, `countVariables`, and `calculateCyclomaticComplexity` each handle specific aspects of metric calculation.
Each method has a single, well-defined purpose, making the class cohesive and focused. This separation of concerns makes the code modular and easier to maintain or extend.

Open/Closed Principle (OCP)

The Open/Closed Principle (OCP) states that software entities should be open for extension but closed for modification. This means that the behavior of a module can be extended without modifying its source code.

Example: Extending `EditorServiceImpl`

```
public interface EditorService {
    CodeMetrics calculateMetrics(String code, String language);
    void insertComment(CodeUpdate codeUpdate);
    void saveCodeUpdate(CodeUpdate codeUpdate);
}

public class ExtendedEditorServiceImpl implements EditorService {

    private final EditorServiceImpl editorServiceImpl;

    public ExtendedEditorServiceImpl(EditorServiceImpl editorServiceImpl) {
        this.editorServiceImpl = editorServiceImpl;
    }
    @Override
    public CodeMetrics calculateMetrics(String code, String language) {
        // Extended behavior can be added here
        return editorServiceImpl.calculateMetrics(code, language);
    }

    @Override
    public void insertComment(CodeUpdate codeUpdate) {
        // Extended behavior can be added here
        editorServiceImpl.insertComment(codeUpdate);
    }

    @Override
    public void saveCodeUpdate(CodeUpdate codeUpdate) {
        // Extended behavior can be added here
        editorServiceImpl.saveCodeUpdate(codeUpdate);
    }

    public void newFeature() {
        // New features can be added without modifying the original class
    }
}
```

The `EditorService` interface defines a contract for the `EditorServiceImpl` class. This allows for the creation of new implementations, such as `ExtendedEditorServiceImpl` , which can extend the functionality of the original class without modifying its source code.
By adhering to OCP, the codebase becomes more flexible and easier to maintain, as new features can be added without altering existing code.

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. This principle ensures that subclasses adhere to the behavior defined by their superclasses.

Example: `CodeMetrics` and Extended Classes

```
public class CodeMetrics {
    private int lines;
    private int functions;
    private int variables;
    private int cyclomaticComplexity;
    // Getters and setters
}

public class ExtendedCodeMetrics extends CodeMetrics {
    private int additionalMetric;

    // Getters and setters for additionalMetric
}
```

The `ExtendedCodeMetrics` class extends `CodeMetrics`, adding an additional metric.
According to LSP, any instance of `CodeMetrics` should be replaceable with an instance of `ExtendedCodeMetrics` without altering the behavior of the program.
This adherence to LSP ensures that the code remains robust and that extensions do not introduce unintended side effects.

Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) states that many client-specific interfaces are better than one general-purpose interface. This principle promotes the creation of small, focused interfaces that are tailored to specific use cases.

Example: `EditorService` Interface

```
public interface EditorService {
    CodeMetrics calculateMetrics(String code, String language);
    void insertComment(CodeUpdate codeUpdate);
    void saveCodeUpdate(CodeUpdate codeUpdate);
}

public interface CommentService {
    void addComment(CodeUpdate codeUpdate);
    void deleteComment(CodeUpdate codeUpdate);
}
```

Explanation:
The `EditorService` interface is focused on operations related to code metrics and updates, while the `CommentService` interface is focused on operations related to comments.
By adhering to ISP, the codebase avoids large, general-purpose interfaces, making it more modular and easier to understand. Clients that use these interfaces only need to implement the methods that are relevant to their use case.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules but on abstractions. This principle promotes the use of interfaces and abstract classes to decouple high-level and low-level modules.

Example: Dependency Injection in `EditorServiceImpl`

```java
@Service
public class EditorServiceImpl implements EditorService {

    private final CodeUpdateRepository codeUpdateRepository;

    @Autowired
    public EditorServiceImpl(CodeUpdateRepository codeUpdateRepository) {
        this.codeUpdateRepository = codeUpdateRepository;
    }

    @Override
    public void saveCodeUpdate(CodeUpdate codeUpdate) {
        codeUpdateRepository.save(codeUpdate);
    }
}
```

The `EditorServiceImpl` class depends on the `CodeUpdateRepository` interface, which is injected through the constructor. This adherence to DIP ensures that the `EditorServiceImpl` class is not tightly coupled to a specific implementation of the repository.
By depending on abstractions, the codebase becomes more flexible and easier to test, as different implementations of the repository can be injected without altering the `EditorServiceImpl` class.

Spring MVC (Model-View-Controller)

Separates responsibilities into three layers:
Model: Domain data and business logic.
View: React frontend handles rendering and user interactions.
Controller: REST endpoints orchestrate requests and responses.

Controllers are organized by feature:

SignInController – authenticates users (`/api/auth/signin`).

```java
@RestController
@RequestMapping("/api/auth")
public class SignInController {
    @PostMapping("/signin")
        public ResponseEntity<String> signIn(@RequestBody LoginRequest
loginRequest) {...}
    }
```

SignUpController – registers new users (`/api/auth/signup`).

```
@RestController
@RequestMapping("/api/auth")
public class SignUpController { ... }
```

ExecutionController – runs submitted code in Docker (`/api/execute`).

```
@RestController
@CrossOrigin
@RequestMapping("/api/execute")
public class ExecutionController { ... }
```

LogsController – retrieves code metrics and logs (`/api/logs/metrics`).

```
@RestController
@RequestMapping("/api/viewer")
@CrossOrigin
public class LogController { ... }
```

RepoController – manages collaborative repos and repo details (`/api/repos`).

```
@RestController
@CrossOrigin
@RequestMapping("/api/repos")
public class RepoController { ... }
```

UserController – fetches authenticated user profiles (`/api/users/{userId}`).

```
@RestController
@RequestMapping("/api/user")
public class UserController { ... }
```

ProjectController, **FileController**, **VersionController**, and **WebsocketController** – handle project CRUD, file versioning, and real-time collaboration via WebSocket.

```java
@Controller
public class WebsocketController { ... }
```

Models:

CodeUpdate:
A MongoDB document that logs code changes for specific files within a project and repo. It captures details such as user ID, filename, repo ID, project name, line number, and the updated code content.
Repo:
Represents a collaborative coding space where multiple users can work on shared files. It manages user membership and access to projects.
User:
Defines a user in the application, including their profile and authentication details. User-related operations are handled by the `UserService`.
RepoMembership:
Tracks users' roles (e.g., viewer, collaborator) within a repo, managing their level of access and participation.
File:
Stores file data and version history in a MongoDB collection, enabling real-time collaboration and content management.
Project:
Represents a project stored in a MySQL database. It connects to repos and users and uses JPA annotations for ORM and database interactions.

View:

The frontend of the application is built with **React**, following the Spring MVC architecture. Styling is managed with **CSS**, and **JavaScript** handles user interactions. Data is exchanged between frontend and backend using **REST APIs** managed by Spring controllers.

Here’s a refined and cleaner version of your explanation on the Builder Pattern usage:

Builder Pattern

Overview:
The Builder design pattern is implemented throughout the project using Lombok’s `@Builder` annotation. This approach streamlines object creation, removing the need for complex constructors and enhancing code readability, maintainability, and flexibility.

Key Applications:
The pattern is applied to several core entities such as **CodeUpdate**, **File**, **Project**, **Repo**, and **User**, allowing for method chaining and simplified instantiation of objects with multiple fields.

Additionally, `@Builder` is used across various **request/response DTOs** like **LoginRequest**, **FileDTO**, and **ProjectDTO**, improving code scalability and making it easier to handle optional parameters and complex fields. It also contributes to immutability by enabling clear and controlled object construction.

```
@Data
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class CodeDTO {
    private String userId;
    private String filename;
    private String repoId;
    private String projectName;
    private String lineNumber;
    private String column;
    private String lineContent;
    private String code;
}
```

DTO (Data Transfer Object) Overview

The **DTO (Data Transfer Object)** pattern is used to carry data between different parts of an application — especially between the client and the server. Instead of sending whole complex objects, DTOs help us send only the needed data in a clean and efficient way. This keeps internal details safe, reduces the amount of data sent over the network, and makes the app easier to manage and update.

DTO Classes and Their Purpose

CodeUpdate
Holds details about a change made to the code.
MessageLog
Keeps track of messages related to actions in a repo.
LoginRequest / LoginResponse
Used to send and receive login information.

CodeDTO
Carries code data from one part of the app to another
CodeExecution
Holds the information needed to run a piece of code
CodeMetrics
Stores performance stats and results after running the code
CodeMetricsRequest
Used to ask for performance data about code.
FileDTO
Sends and receives data related to files
ProjectDTO
Used to transfer project-related information.
RepoDTO
Holds data related to a specific repo.
CreateRepoRequest / AddMemberRequest
Used when creating a new repo or adding someone to it.

Strategy Design Pattern

In this project, the **Strategy Design Pattern** is used to handle code execution differently depending on the programming language (e.g., Python, Java, C++). Each language has its own way of being run, so a custom command template is defined for each one.

The `DockerExecutorService` selects the right strategy (i.e., command template) **at runtime**, based on the language specified in the `CodeExecution` request. This makes the system flexible and easy to extend when adding support for more languages.

Key Methods Involved in the Strategy Pattern

`getCommandTemplateForLanguage(String language)`
Chooses the correct command template based on the programming language.

`prepareCommand(String dockerComman, String code, String input)`
Fills in the command template with the actual code and input to prepare it for execution.

`buildDockerCommand(String dockerImage, String command)`
Creates the final Docker command that will run the code inside the appropriate container.

Singleton Pattern

The **Singleton Pattern** is used in two main ways:

Lock Management:
Shared maps like `repoLocks`, `fileLocks`, and `projectLocks` are used to control access and avoid conflicts. These are globally available and ensure only one thread can work on a specific resource at a time.

Service Dependency Injection:
Spring automatically creates service classes as **singletons**, meaning only one instance of each service exists throughout the application. This allows consistent, shared access to business logic.

Observer Pattern

In the context of **WebSockets**, the application follows the **Observer Pattern** to keep clients updated in real-time.

Subjects (Producers):
Methods in the `WebsocketController` marked with `@MessageMapping` act as the **subjects**. These methods receive updates (like code changes or messages), process them, and send them to specific WebSocket topics (e.g., `/topic/file/updates/...`). This broadcast informs all subscribed clients (observers) about the update.

Observers (Subscribers):
Clients connected to topics like `/topic/file/updates/{repoId}/{projectName}/{filename}` or `/topic/chat/{repoId}` act as **observers**. Whenever a change happens (code update or new chat message), the server pushes updates to all connected clients in real time.

```
@MessageMapping("/code/updates/{repoId}/{projectName}/{filename}/COLLABORATOR"
)
@SendTo("/topic/file/updates/{repoId}/{projectName}/{filename}")
public CodeUpdate handleCollaboratorCode(
        @DestinationVariable String repoId,
        @DestinationVariable String projectName,
        @DestinationVariable String filename,
        @Payload CodeUpdate codeUpdate) {

    String lineKey = repoId + "-" + projectName + "-" + filename + "-" +
codeUpdate.getLineNumber();

    executorService.submit(() -> {
        synchronized (getLineLock(lineKey)) {
            editorService.saveCodeUpdate(codeUpdate);
        }
    });
    return codeUpdate;
}
```

Purpose:
To handle live collaboration. For example, when a collaborator updates code, the server saves the update using `editorService.saveCodeUpdate(codeUpdate)` and notifies all subscribers via `/topic/file/updates/...`.

Observer Role:
Any client connected to the file's WebSocket topic receives the update immediately, ensuring everyone sees the latest changes without needing to refresh or poll.

Multithreading

Handling Multithreading

Concurrent Access Management:
The backend code handles multiple client connections concurrently, ensuring that the application can scale with an increasing number of users. This is achieved through the use of asynchronous processing and thread management techniques.

ExecutorService for Asynchronous Processing:
The `WebsocketController` class employs an `ExecutorService` to manage asynchronous processing of WebSocket messages. This service executes tasks in separate threads, allowing the main thread to remain responsive to incoming requests. By offloading tasks to the `ExecutorService`, the application can handle a high volume of concurrent operations without blocking the main thread.

```
@Controller
public class WebsocketController {
    private final ExecutorService executorService;
    ...
}
```

```
    executorService.submit(() -> {
        synchronized (getLineLock(lineKey)) {
            editorService.saveCodeUpdate(finalCodeUpdate);
        }
    });
```

Ensuring Thread Safety

Synchronized Locks:
To ensure thread safety, I uses synchronized locks to manage concurrent access to shared resources. This is evident in the `EditorService` class, where synchronized locks are used to control access to code files.

```
    private Object getCodeLock(String fileKey) {
        return CodeLocks.computeIfAbsent(fileKey, k -> new Object());
    }
```

The `getCodeLock` method returns a lock object associated with a specific file key, ensuring that only one thread can modify the file at a time.

Similarly, the `WebsocketController` class uses synchronized locks to manage access to specific lines of code during collaborative editing sessions.

```
    executorService.submit(() -> {
        synchronized (getLineLock(lineKey)) {
            editorService.saveCodeUpdate(finalCodeUpdate);
        }
    });
```

The `getLineLock` method returns a lock object associated with a specific line key, preventing race conditions and ensuring data consistency.

Atomic Operations:
While not explicitly shown in the snippets, the use of atomic variables and operations can further enhance thread safety. Atomic operations ensure that updates to shared variables are performed in a single, indivisible step, preventing race conditions and ensuring data consistency.

Immutable Data Structures:
The use of immutable data structures to enhances thread safety by ensuring that objects cannot be modified after creation. Immutable objects are inherently thread-safe, as their state cannot be changed, eliminating the risk of concurrent modification exceptions.

Concurrency Control in Databases:
When interacting with databases, your code ensures thread safety by using transactions `@Transactional` and isolation levels to control concurrent access. This is particularly important in the `ProjectServiceImpl` and `RepoServiceImpl` classes, where database operations are

performed concurrently. By using appropriate isolation levels, the application prevents dirty reads, non-repeatable reads, and phantom reads, ensuring data consistency and integrity.

example: use `@Transactional` to make database operations are performed concurrently.

```
    @Transactional
    public void createProject(ProjectDTO project) {
        Repo repo = repoRepository.findByRepoId(project.getRepoId())
                .orElseThrow(() -> new RepositoryNotFoundException("Repo not 
found for ID " + project.getRepoId()));
        Project newProject = buildNewProject(project, repo);
        try {
            projectRepository.save(newProject);
        } catch (Exception e) {
            throw new RuntimeException("Failed to create project: " + 
e.getMessage());
        }
        createDefaultFile(project);
    }
```

Thread Safety via Synchronization and Locks

Because users may edit identical files or even the same line of code simultaneously, we employ locking strategies at multiple granularities to prevent race conditions:

Locking

Each line, file and Repository is represented by a unique lock key (`repoId + projectName + filename + ...).

Locks are stored in a `ConcurrentHashMap`, and access to each lock is guarded with a `synchronized` block so that only one thread can modify a given line at a time.

```
        String lineKey = repoId + "-" + projectName + "-" + filename + "-" + 
codeUpdate.getLineNumber();

        codeUpdate = editorService.insertComment(codeUpdate);
        CodeUpdate finalCodeUpdate = codeUpdate;

        executorService.submit(() -> {
            synchronized (getLineLock(lineKey)) {
                editorService.saveCodeUpdate(finalCodeUpdate);
            }
        });
```

Pessimistic Database Locking

When writing file content to the database, we annotate repository methods with `@Lock(LockModeType.PESSIMISTIC_WRITE)` This forces any concurrent transaction to wait until the current write completes.
for example on FileRepository interface:

```
        @Modifying
        @Update(update = "{ 'filename' : ?0, 'projectName' : ?1, 'repoId' : ?
2, 'content' : ?3, 'createdAt' : ?4, 'lastModifiedAt' : ?5, 'extension': ?6
}")
        @Query(value = "{ 'filename' : ?0, 'projectName' : ?1, 'repoId' : ?
2}")
        @Lock(LockModeType.PESSIMISTIC_WRITE)
        void upsertFileContent(@Param("filename") String filename,
                                @Param("projectName") String projectName,
                                @Param("repoId") String repoId,
                                @Param("content") String content,
                                @Param("createdAt") Long createdAt,
                                @Param("lastModifiedAt") Long lastModifiedAt,
                                @Param("extension") String extension);
```

Multithreading in WebSocket

Multithreading is crucial for supporting real-time collaboration over WebSocket, allowing multiple users to interact with the same codebase concurrently without blocking each other. We leverage an `ExecutorService` to dispatch tasks in parallel, ensuring responsive handling of incoming requests and updates.

- **Thread Pool Configuration**
  We instantiate a fixed-size thread pool via

```
    @Bean
    @Qualifier("webSocketExecutorService")
    public ExecutorService webSocketExecutorService() {
        return Executors.newFixedThreadPool(50);
    }
```

Limiting the pool to 100 threads prevents resource exhaustion while permitting up to 100 simultaneous WebSocket event handlers.

```
        executorService.submit(() -> {
            synchronized (getLineLock(lineKey)) {
                editorService.saveCodeUpdate(codeUpdate);
            }
        });
```

- `docker-compose.yml` spins up five services on a single user-defined network (`backend`), wiring dependencies, ports, and persistent volumes.
- The **frontend Dockerfile** builds a Node/React dev container with live-reload.
- The **backend Dockerfile** uses a multi-stage build: first compile with Maven, then run with a slim JDK image (plus Docker CLI for any container orchestration your app might do).

Here's a **summarized version (approx. 5 pages)** of the detailed Docker Compose + Dockerfile explanation, formatted for a README or repository documentation. It includes code snippets and concise explanations to maintain clarity and usefulness.

Docker Setup Summary – Collaborative Editor

This project uses Docker Compose to run a full-stack collaborative editor environment with a Spring Boot backend, React frontend, MySQL, MongoDB, and UI tools (phpMyAdmin and mongo-express).

---

docker-compose.yml Overview

Services

```
version: "3.8"
services:
```

Defines multiple services (containers) that work together. Each service is a containerized piece of the app.

Backend – `app`

```
  app:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: collaborative-editor
    ports:
      - "8080:8080"
    depends_on:
      - mysql
      - mongo
    networks:
      - backend
    environment:
      - DOCKER_HOST=tcp://host.docker.internal:2375
    volumes:
      - ./editor/src:/editor/src
      - ./editor/target:/editor/target
      - "${USERPROFILE}/.m2:/root/.m2"
```

Builds from `./backend` with its Dockerfile.

Maps `8080` → Spring Boot's default port.

Depends on `mysql` and `mongo` containers.

Uses `DOCKER_HOST` to allow Docker CLI access.

Mounts source/target and `.m2` for Maven caching.

Frontend – `frontend`

```
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: collaborative-frontend
    ports:
      - "3000:3000"
    depends_on:
      - app
    networks:
      - backend
    volumes:
      - ./frontend/src:/app/src
```

React app running on port `3000`.

Depends on `app` service (backend).

Volume maps frontend source for live reload.

MySQL Database – `mysql`

```yaml
mysql:
  image: mysql:8.0
  container_name: collaborative-mysql
  restart: always
  environment:
    MYSQL_DATABASE: mysql_db
    MYSQL_ROOT_PASSWORD: root
    MYSQL_ROOT_USER: root
  ports:
    - "3307:3306"
  volumes:
    - mysql_data:/var/lib/mysql
  networks:
    - backend
```

Runs MySQL on host port `3307`.

Initializes root user and a database.

Stores data in `mysql_data` volume.

MongoDB – `mongo`

```yaml
mongo:
  image: mongo
  container_name: collaborative-mongo
  environment:
    MONGO_INITDB_ROOT_USERNAME: devroot
    MONGO_INITDB_ROOT_PASSWORD: devroot
    MONGO_INITDB_DATABASE: mongo_db
  ports:
    - "27017:27017"
  volumes:
    - mongo_data:/data/db
    - ./mongo_init:/docker-entrypoint-initdb.d
  networks:
    - backend
```

MongoDB with admin credentials.

Seeding via `./mongo_init` folder.

Exposed on default port `27017`.

Mongo UI – `mongo-express`

```yaml
mongo-express:
  image: mongo-express
  container_name: collaborative-mongo-express
  environment:
    ME_CONFIG_MONGODB_SERVER: collaborative-mongo
    ME_CONFIG_MONGODB_PORT: 27017
    ME_CONFIG_MONGODB_AUTH_USERNAME: devroot
    ME_CONFIG_MONGODB_AUTH_PASSWORD: devroot
    ME_CONFIG_BASICAUTH_USERNAME: dev
    ME_CONFIG_BASICAUTH_PASSWORD: dev
  depends_on:
    - mongo
  ports:
    - "8888:8081"
  networks:
    - backend
```

Web UI to view MongoDB.

Secured with basic auth (`dev` / `dev`).

Runs at `http://localhost:8888`.

MySQL UI – `phpmyadmin`

```yaml
phpmyadmin:
  image: phpmyadmin/phpmyadmin
  container_name: collaborative-phpmyadmin
  environment:
    PMA_HOST: collaborative-mysql
    PMA_PORT: 3306
    PMA_USER: root
    PMA_PASSWORD: root
  ports:
    - "8082:80"
  depends_on:
    - mysql
  networks:
    - backend
```

Access MySQL visually at `http://localhost:8082`.

Networks & Volumes

```
networks:
  backend:
    driver: bridge

volumes:
  mysql_data:
    driver: local
  mongo_data:
    driver: local
```

All containers share a custom bridge network.

Volumes persist DB data.

Dockerfile – Frontend

```
FROM node:18
WORKDIR /app
COPY package.json package-lock.json ./
RUN yarn install
RUN yarn add --dev nodemon
COPY src ./src
COPY public ./public
EXPOSE 3000
CMD ["yarn", "start"]
```

Node 18-based React app.

Uses `yarn` with `nodemon` for hot reload.

Starts development server on port `3000`.

Dockerfile – Backend (Multi-stage)

```
# Stage 1: Build
FROM maven:3.8.5-openjdk-17 AS build
WORKDIR /editor
COPY pom.xml /editor/
RUN mvn dependency:go-offline
COPY src /editor/src/
RUN mvn clean install -DskipTests

# Stage 2: Runtime
FROM openjdk:17-jdk-slim
RUN apt-get update && apt-get install -y docker.io
WORKDIR /editor
COPY --from=build /editor/target/editor-0.0.1-SNAPSHOT.jar /editor/editor.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/editor/editor.jar", "-
Dspring.devtools.restart.enabled=true"]
```

Stage 1: Uses Maven to build the app and fetch dependencies.

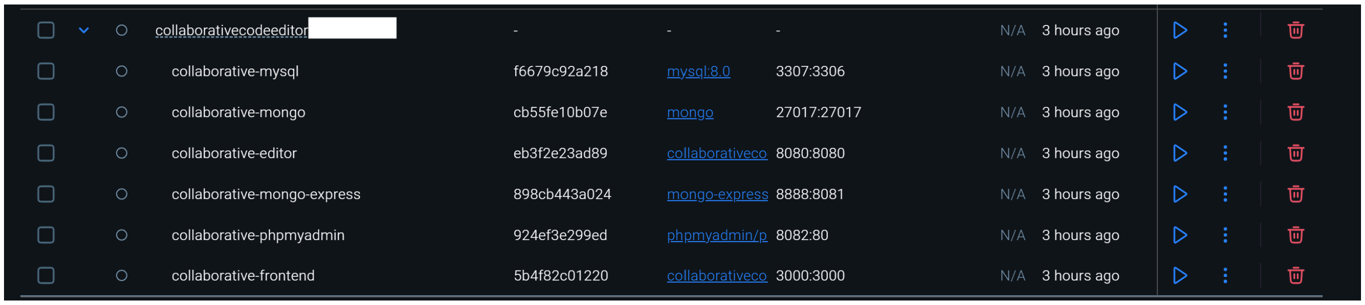
Stage 2: Slim runtime with Docker CLI installed.

Runs your Spring Boot app with DevTools reload enabled.

Build and Run

```
cd {project path}
docker-compose up --build
```

Access services:



collaborativecodeeditor[redacted]	-	-	-	N/A	3 hours ago
collaborative-mysql	f6679c92a218	mysql:8.0	3307:3306	N/A	3 hours ago
collaborative-mongo	cb55fe10b07e	mongo	27017:27017	N/A	3 hours ago
collaborative-editor	eb3f2e23ad89	collaborativeco	8080:8080	N/A	3 hours ago
collaborative-mongo-express	898cb443a024	mongo-express	8888:8081	N/A	3 hours ago
collaborative-phpmyadmin	924ef3e299ed	phpmyadmin/p	8082:80	N/A	3 hours ago
collaborative-frontend	5b4f82c01220	collaborativeco	3000:3000	N/A	3 hours ago

Frontend: `http://localhost:3000`

Backend: `http://localhost:8080`

Mongo UI: `http://localhost:8888`

MySQL UI: `http://localhost:8082`