

Collaborate. Create. Code in Real-Time.

Unlock the power of collaboration. Code, review, and merge projects in multiple languages without leaving your browser.

[Start Coding Now](#)

Software Engineering and DevOps – spring 2024

Prepared By Abdullah Ayman

Atypont Final Project



[/in/abdullah-ayman](https://www.linkedin.com/in/abdullah-ayman)



abdayman1092002@gmail.com



+962-796489732

TECHNOLOGIES USED IN OUR COLLABORATIVE CODE EDITOR



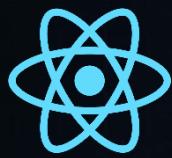
JAVA



SPRING BOOT



SPRING SECURITY



REACT.JS



JAVASCRIPT



CSS



MYSQL



MONGODB



GitHub



Docker



AWS



GitHub Actions



OAuth 2.0

CONTENTS

Introduction	4
Web Application Overview	5
User Roles and Permissions	6
Owner: Responsibilities	7
Viewer: Viewing, Logs	9
Collaborator: Editing, Versioning	14
System Design and Architecture	20
Agile project roadmap	20
Overview of System Architecture	21
Database Design	22
Security Design	25
Exception Handling Design	31
Version Control Design	33
Code Executor Design	36
Code Metrics Feature	38
Design Patterns	39
Spring MVC Pattern	39
Builder Pattern	44
DTO design pattern	45
Template Method design pattern	46
Observer Websocket Pattern	47
Global Exception Handling Stratigy Design Pattern	48
Multithreading	49
Multithreading in WebSocket Code Updates	50
Thread Safety with Synchronization and Locks	51
Optimistic Locking for Operations.....	52
Solid Principles	53
Clean Code	64
Effictive Java	72
DevOps(GitHub)	77
DevOps(Docker)	79
DevOps(CI/CD)	86
DevOps(AWS)	98

Introduction

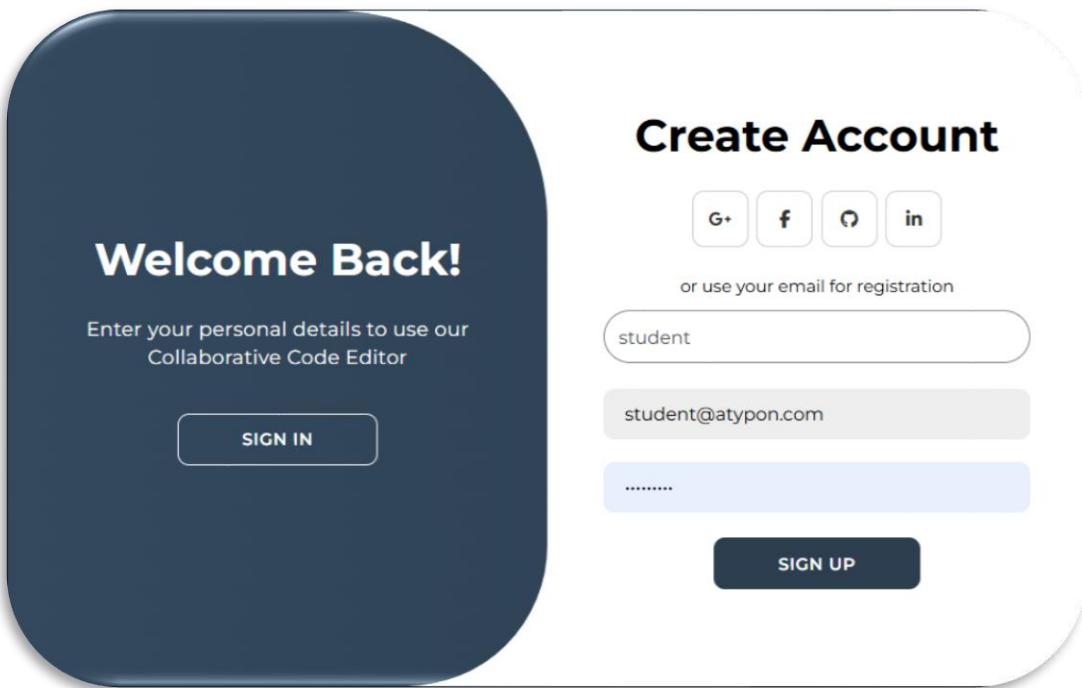


This report outlines the development of a Collaborative Code Editor, a web application that empowers multiple users to write, edit, and execute code together in real-time, regardless of their physical location. It supports real-time communication using WebSockets, version control, and various user roles such as Owner, Viewer, and Collaborator, each with distinct responsibilities. It also incorporates essential features such as code execution in multiple programming languages, robust file management, and real-time updates.

In this report, we will cover:

- **System Design & Architecture:** The architecture decisions, including security, data schema design, and communication protocols.
- **Clean Code:** How the project adheres to best practices for writing clear, readable, and maintainable code.
- **Effective Java:** How principles from Effective Java were applied to improve performance and reliability.
- **SOLID Principles:** How the code structure follows the key principles of software design.
- **Design Patterns:** The use of common patterns that enhance flexibility and maintainability.
- **Multithreading:** How we handled concurrency to support multiple users editing files simultaneously, ensuring data consistency.
- **DevOps Practices:** The use of Docker, AWS, and CI/CD pipelines for deployment and scaling the application.

Web Application Overview



Before you get started with our collaborative code editor, you'll first encounter the login form. Our platform allows you to sign up for a new account or conveniently log in using your existing accounts from GitHub, Facebook, Google, or LinkedIn. Once you've successfully authenticated, you'll have full access to the powerful features of the collaborative code editor.

Sign in

to continue to [Collaborative Code Editor](#)

Sign in with Google

Email or phone
abdullahayman@gmail.com

[Forgot email?](#)

Before using this app, you can review Collaborative Code Editor's [privacy policy](#) and [terms of service](#).

[Create account](#) [Next](#)

Sign in to GitHub to continue to Collaborative Code Editor

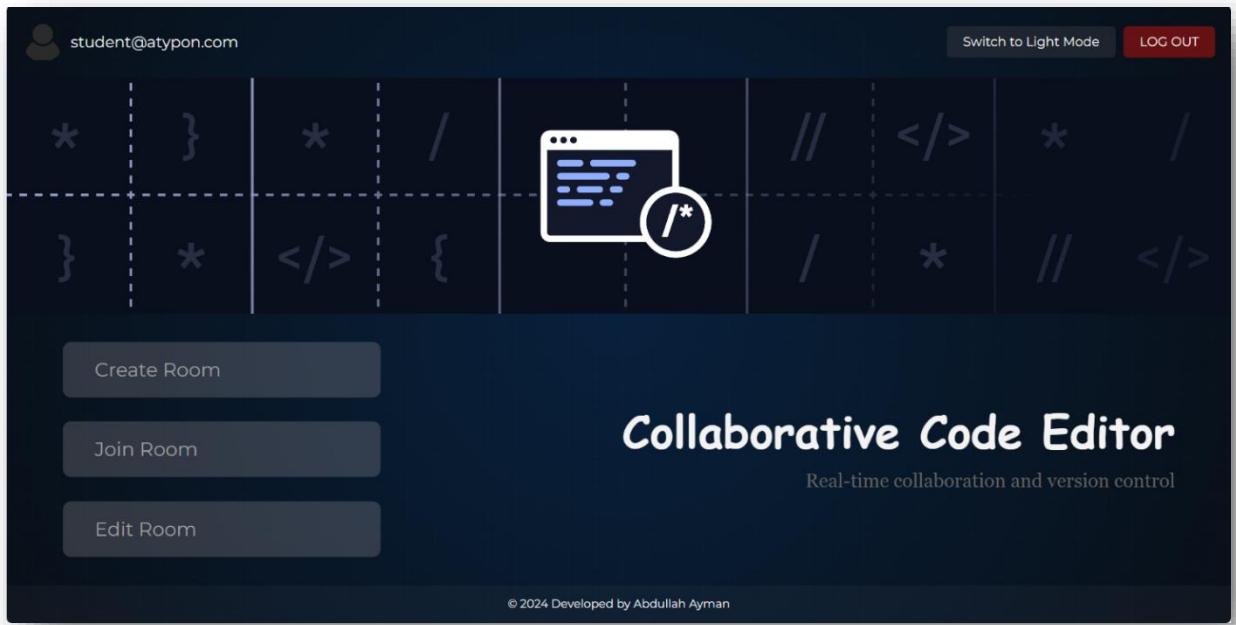
Username or email address

Password [Forgot password?](#)

[Sign in](#)

[Sign in with a passkey](#)
New to GitHub? [Create an account](#)

2.1 User Roles and Permissions



Once authorized, users can access the platform's features. Depending on their role—Owner, Collaborator, or Viewer—they can create, join, or manage rooms and projects, as outlined in the table below. Each role comes with specific permissions to enhance collaboration.

ROLE	PERMISSIONS	RESPONSIBILITIES
OWNER	Create room, add/remove projects, rename/remove room	Full control of the room, project management, adding/removing users
COLLABORATOR	Edit files, manage versions, PULL, PUSH, MERGE, RUN and DOWNLOAD version files and projects	Contribute to project files, handle version control, and real time editing.
VIEWER	View project, assist collaborators, view logs, run the codes	Monitor project progress, view code complexity, help collaborators

2.2 Owner: Responsibilities



Once you create a room as an Owner, you are given full control over the room's management and its members.

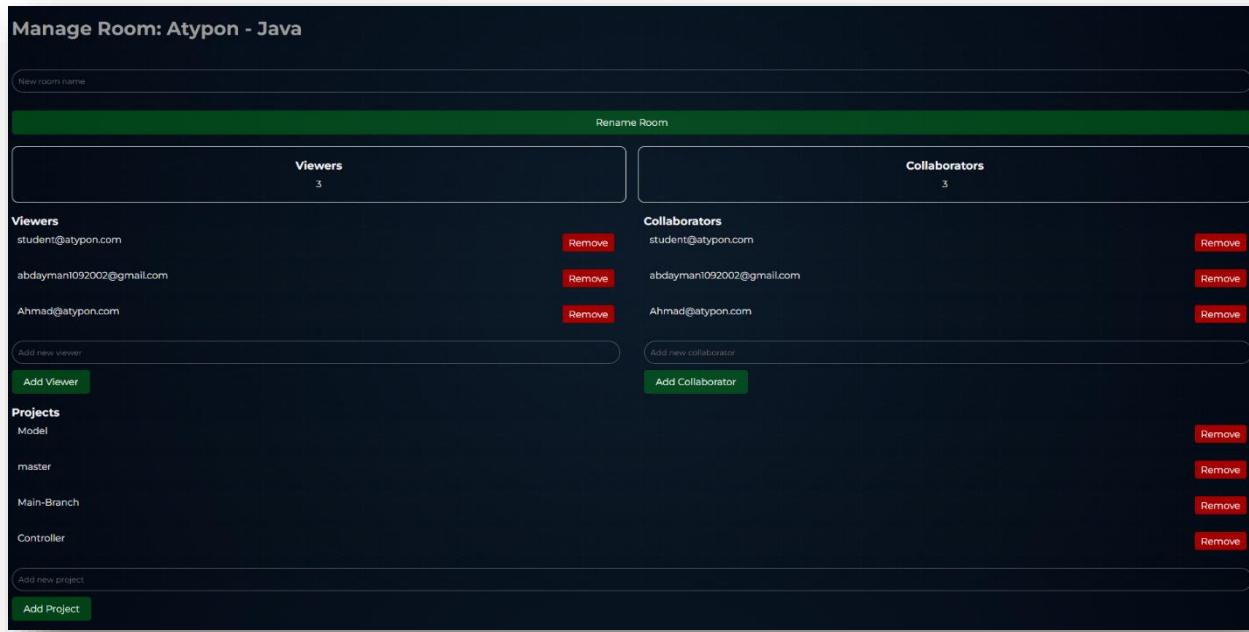
Atypon - Java ID: 9dabcd21-dc95-4b2a-bdf6-aaaca6733a00	Atypon - Python ID: f2fc5c7a-6490-450e-95c5-b7085d83a5e1	Atypon - C++ ID: 91aa4be7-520f-4275-8085-36cd2d0e5ae	Atypon - PHP ID: d15e52d6-4473-48bf-b2f7-a48533ad615f	Atypon - REACT.JS ID: 93a58866-69dc-4e35-b8c7-46c112617432
Atypon - Angular ID: 64bd3fb8-3660-4b30-b2c4-3209931abb17	Atypon - Machine Learning ID: ee849491-eb87-4c07-bed4-e5a7a6ce7030	Atypon - neuralNetwork ID: 9c233693-3ebf-4b97-ab68-8828b83ff551	Atypon - Dart ID: f5b0d6c8-a182-428c-8b27-59d65bcfdf93	

Switch to Light Mode [HOME](#) [LOG OUT](#)

© 2024 Developed by Abdullah Ayman

Hit what you want from your own room's dashboard then you can manage it.

2.2 Owner: Responsibilities



Owner Abilities in the Collaborative Code Editor

- Add New Members: This typically involves sending an invitation link or directly adding the member's email address through a user interface.
- Remove Members: The Owner can select members from a list and choose an option to remove them, which revokes their access to the room.
- Rename the Room: This usually involves entering a new name in a designated field within the settings of the room.
- Download Projects and Logs: This is often facilitated through a download button that packages the project files and logs into a downloadable format (e.g., ZIP file).
- Manage Projects: This involves a user interface where Owners can create new project entries, edit details, or delete projects as needed.

Note

It's important to mention that some of the room control functionality has not yet been implemented in the frontend (like download logs, projects and remove the room). Future enhancements will focus on completing these controls and developing a more creative dashboard to improve user experience.

2.3 Viewer: Viewing, Logs and Assessment

VIEWER ROOMS

- Room Name : Atypon - Java
Room Id: 9dabcd21-dc95-4b2a-bdf6-aaaca6733a00
- Room Name : Atypon - Python
Room Id: f2fc5c7a-6490-450e-95c5-b7085d83a5e1
- Room Name : Atypon - C++
Room Id: 91aa4be7-520f-4275-8085-36cdd2d0e5ae
- Room Name : Atypon - REACT.JS
Room Id: 93a58866-69dc-4e35-b8c7-46c112617432
- Room Name : Atypon - Dart
Room Id: f5b0d6c8-a182-428c-8b27-59d65bcfdf93

COLLABORATOR ROOMS

- Room Name : Atypon - Java
Room Id: 9dabcd21-dc95-4b2a-bdf6-aaaca6733a00
- Room Name : Atypon - Python
Room Id: f2fc5c7a-6490-450e-95c5-b7085d83a5e1
- Room Name : Atypon - C++
Room Id: 91aa4be7-520f-4275-8085-36cdd2d0e5ae
- Room Name : Atypon - REACT.JS
Room Id: 93a58866-69dc-4e35-b8c7-46c112617432
- Room Name : Atypon - Dart
Room Id: f5b0d6c8-a182-428c-8b27-59d65bcfdf93

© 2024 Developed by Abdullah Ayman

Viewers, once they join a room, they can engage with the project by accessing the latest code, adding comments, and reviewing important metrics. This functionality not only enhances collaboration but also ensures that Viewers can contribute effectively to the overall project quality.

Atypon - Java

```
1 import java.util.Scanner;
2 class Main {
3     public static void main() {
4         Scanner in = new Scanner(System.in);
5         System.out.println("Enter your user name");
6         String userName = in.nextLine(); //Please re-write this line
7         System.out.println("The username is: " + userName);
8     }
9 }
10
11 }
```

As you can see the viewer can not edit only add comment

Version Control

- Add New Branch
- Add New File
- Download Project

Branches

- Controller
 - RoomController
 - WebSocketController
 - ProjectController
- Model
- Main-Branch

File Name: RoomController.java
Project: Controller
Created At: 10/21/2024, 2:57:23 PM
Last Modified At: 10/21/2024, 2:57:23 PM

student@atypon.com (VIEWER): joined the room 3:01:09 PM

Ahmad@atypon.com (COLLABORATOR): joined the room 3:01:14 PM

Type a message... Send

2.3 Viewer: Viewing, Logs and Assessment

Viewer Features:

➤ Joining the Room:

Action: When a Viewer joins the room, a message is logged to indicate their entry.

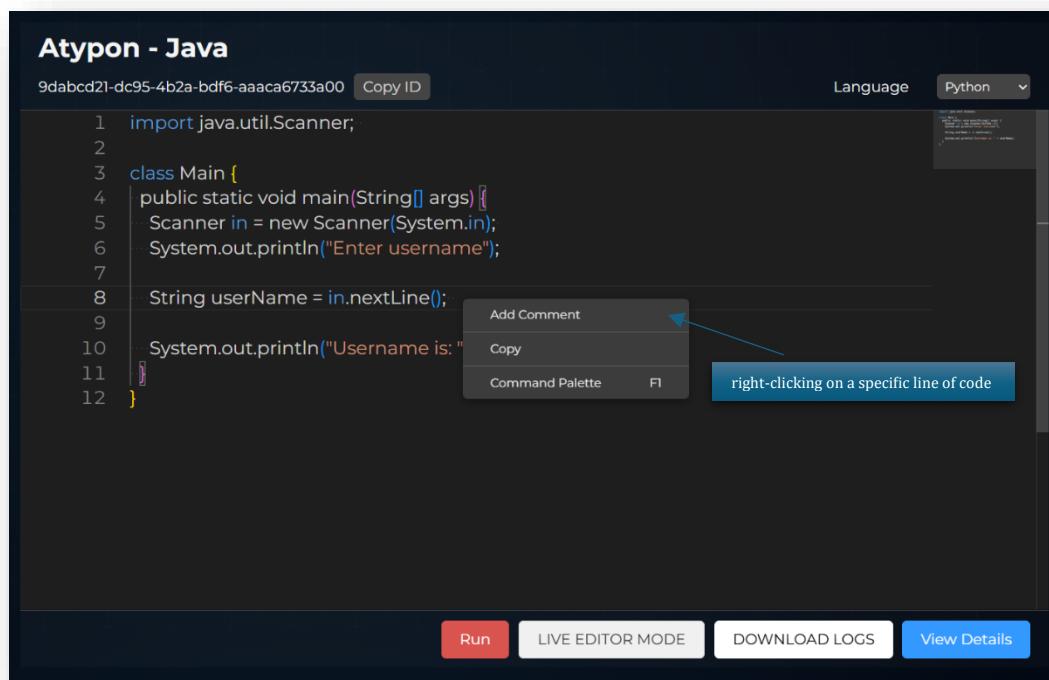
Process: The system automatically records this event in the logs file, providing a history of room activity.



➤ Adding Comments:

Action: Viewers can provide feedback or notes directly in the code by adding comments.

Process: By right-clicking on a specific line of code, Viewers access a menu with various features, including the option to "add comment." This allows them to insert comments on any line, fostering clear communication.



2.3 Viewer: Viewing, Logs and Assessment

The screenshot shows a Java code editor interface. The code is as follows:

```
1 import java.util.Scanner;
2
3 class Main {
4     public static void main(String[] args) {
5         Scanner in = new Scanner(System.in);
6         System.out.println("Enter username");
7
8     String userName = in.nextLine();
9
10    System.out.println("Username is: " + userName);
11 }
12 }
```

A modal window titled "Add a comment to Line 8" is displayed over the code. It contains the text "please rewrite this Line of code" and two buttons: "Submit Comment" and "Cancel".

At the bottom of the editor, there are four buttons: "Run", "LIVE EDITOR MODE" (which is highlighted in green), "DOWNLOAD LOGS", and "View Details".

Note

It's important to mention that, make sure press **LIVE EDITOR MODE** Button to allow you to add comments in live mode.

The screenshot shows the same Java code editor interface. The code remains the same, but now the line "8 String userName = in.nextLine(); //please rewrite this line of code" has a green annotation "As you can see the comment added successfully at line number eight." with an arrow pointing to it.

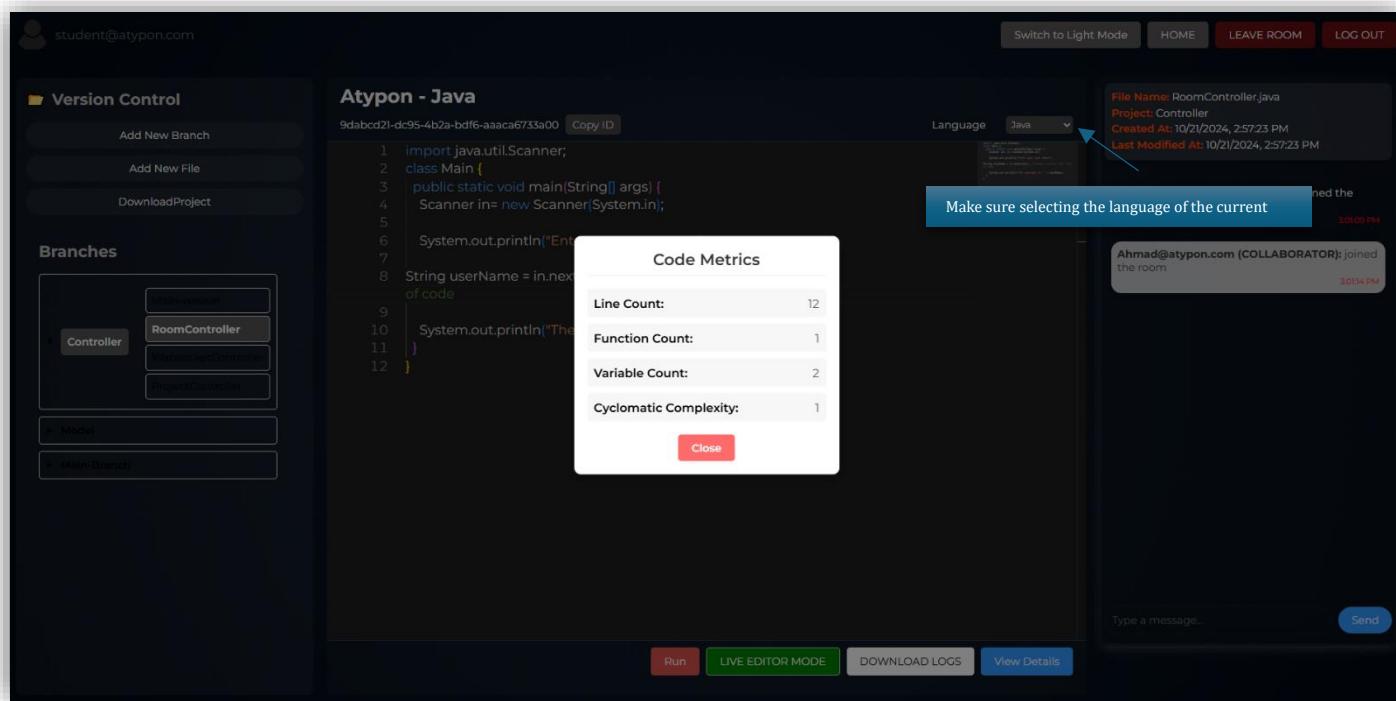
At the bottom of the editor, there are four buttons: "Run", "LIVE EDITOR MODE" (which is highlighted in green), "DOWNLOAD LOGS", and "View Details".

2.3 Viewer: Viewing, Logs and Assessment

➤ Displaying Code Metrics:

Action: Viewers can assess the code quality and complexity.

Process: The system displays metrics such as the number of lines, functions, variables, and the cyclomatic complexity. This information aids Viewers in evaluating the contributions of Collaborators and understanding the code structure (for now this feature is allowed only to java and python).

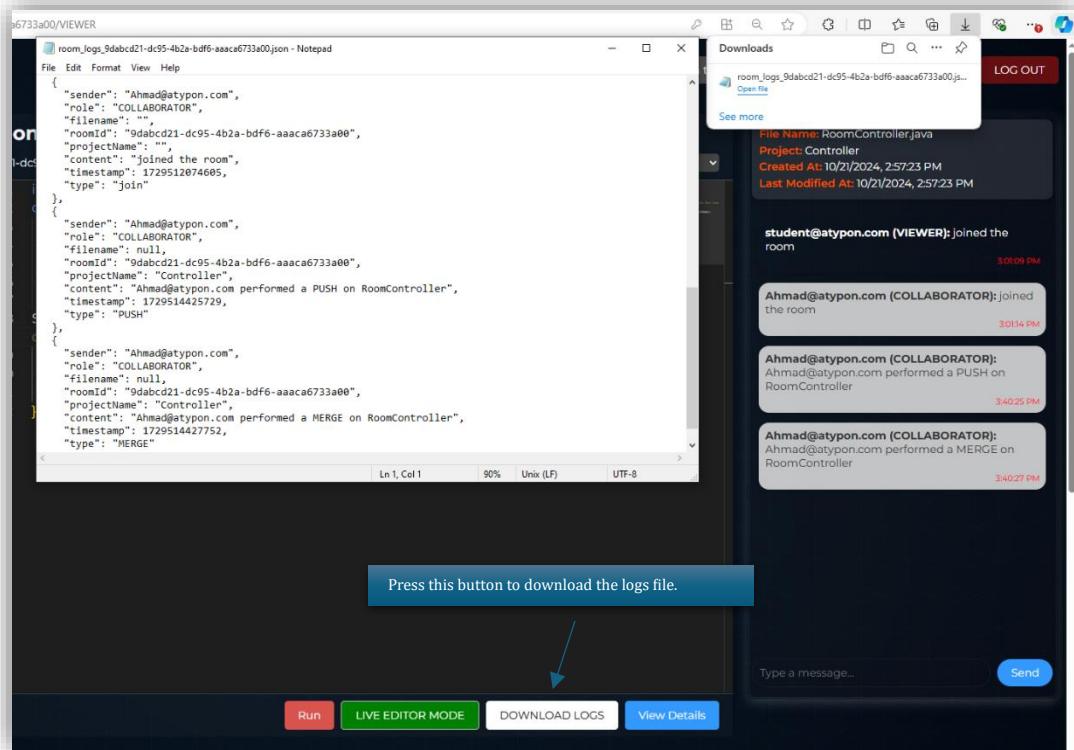


➤ Downloading Logs:

Action: Viewers can download a log file that contains detailed information about project activity.

Process: This log file includes data on who pushed or merged files, as well as who joined or left the room. Such logs are essential for tracking changes and contributions.

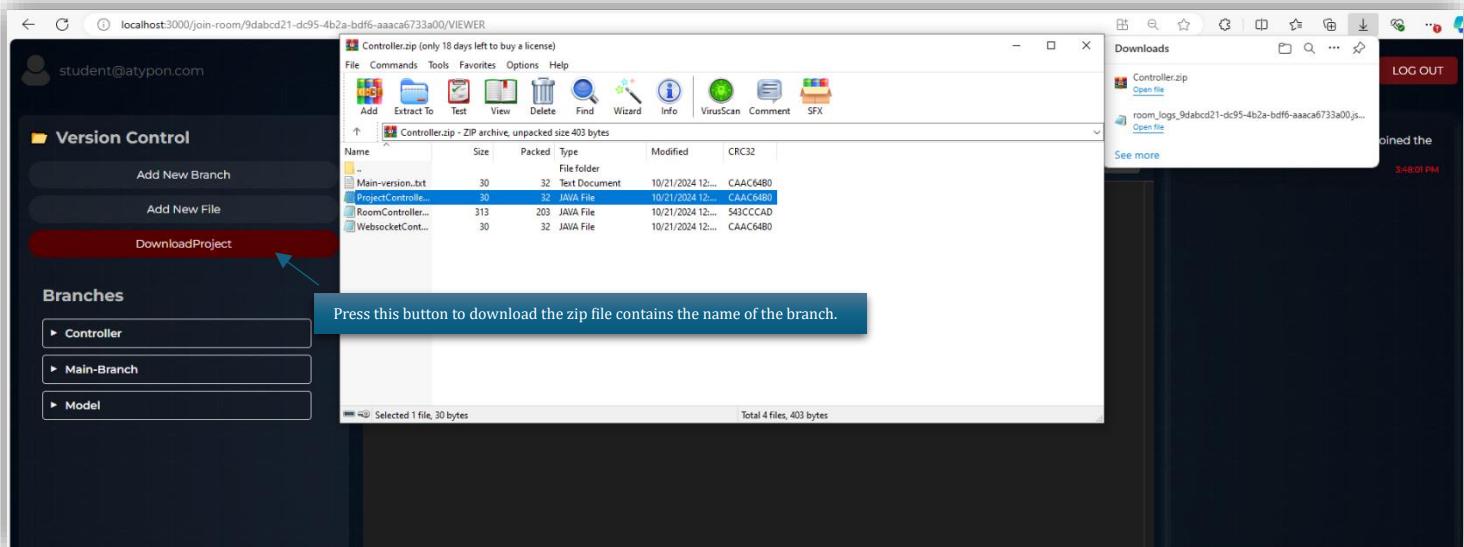
2.3 Viewer: Viewing, Logs and Assessment



➤ Downloading Branches:

Action: Viewers have the option to download any branch of the project.

Process: This feature enables them to access specific versions of the code for local review or testing.



2.4 Collaborator: Editing, Versioning

The screenshot shows two main sections: 'VIEWER ROOMS' and 'COLLABORATOR ROOMS'.
In 'VIEWER ROOMS', there are five listed rooms:

- Room Name : Atypon - Java
Room Id: 9dabcd21-dc95-4b2a-bdf6-aaaca6733a00
- Room Name : Atypon - Python
Room Id: f2fc5c7a-6490-450e-95c5-b7085d83a5e1
- Room Name : Atypon - C++
Room Id: 91aa4be7-520f-4275-8085-36cd2d0e5ae
- Room Name : Atypon - REACT.JS
Room Id: 93a58866-69dc-4e35-b8c7-46c112617432
- Room Name : Atypon - Dart
Room Id: f5b0d6c8-a182-428c-8b27-59d65bcfdf93

In 'COLLABORATOR ROOMS', there are also five listed rooms:

- Room Name : Atypon - Java
Room Id: 9dabcd21-dc95-4b2a-bdf6-aaaca6733a00
- Room Name : Atypon - Python
Room Id: f2fc5c7a-6490-450e-95c5-b7085d83a5e1
- Room Name : Atypon - C++
Room Id: 91aa4be7-520f-4275-8085-36cd2d0e5ae
- Room Name : Atypon - REACT.JS
Room Id: 93a58866-69dc-4e35-b8c7-46c112617432
- Room Name : Atypon - Dart
Room Id: f5b0d6c8-a182-428c-8b27-59d65bcfdf93

At the bottom center, it says "© 2024 Developed by Abdullah Ayman".

Collaborators, once they join a room, they can engage with the project by accessing the latest code. Collaborators play a key role in the code editing process, actively contributing to project development in real time. Download, PUSH and merge then handling everything from writing code to resolving conflict. This functionality enhances collaboration, enabling multiple contributors to work efficiently within a shared environment.

The screenshot shows the 'Atypon - Java' code editor interface. On the left, there's a sidebar with 'Version Control' (Add New Branch, Add New File, DownloadProject), 'Branches' (Main-Branch, Model, Controller), and a 'Controller' section with sub-options: Main-version, RoomController, WebSocketController, and ProjectController. The main area displays the following Java code:import java.util.Scanner;
class Main {
 public static void main(String[] args) {
 Scanner in= new Scanner(System.in);
 System.out.println("Enter your user name");
 String userName = in.nextLine(); //Please re-write this line of code
 System.out.println("The username is: " + userName);
 }
}

Below the code, a blue button says 'New options for the collaborators' with a downward arrow pointing towards the bottom right. At the bottom right, there are four buttons: 'Run', 'LIVE EDITOR MODE', 'MERGE', and 'PUSH'. In the top right corner, there's a message: 'student@atypon.com (COLLABORATOR): joined the room'.

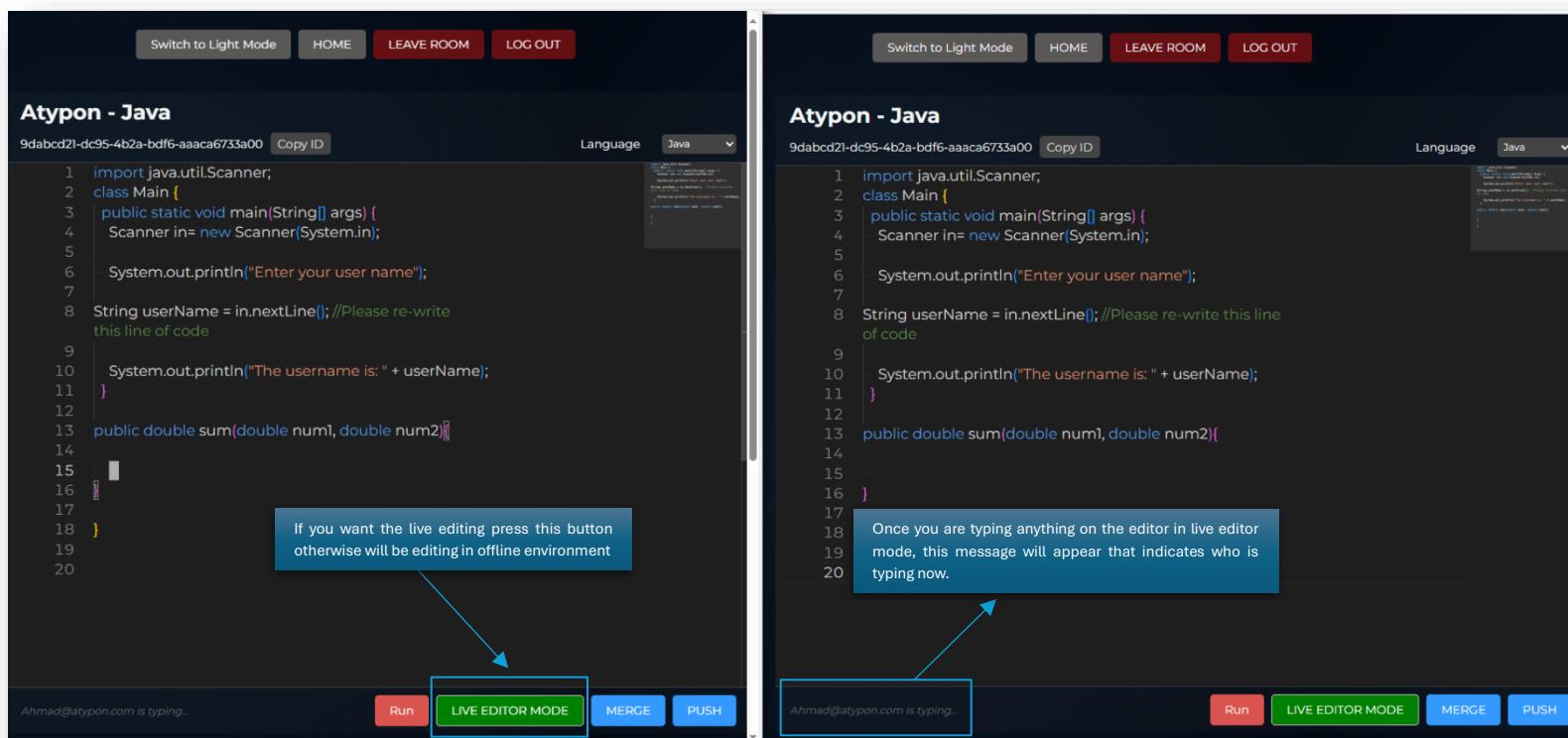
2.4 Collaborator: Editing, Versioning

Collaborator Features:

➤ Joining the Room and Live Editing:

Action: Once a Collaborator joins the room and selects a branch and file, they can immediately begin editing the code in real time.

Process: Any changes made by the Collaborator are instantly reflected in the editor, allowing other Collaborators to see updates live.

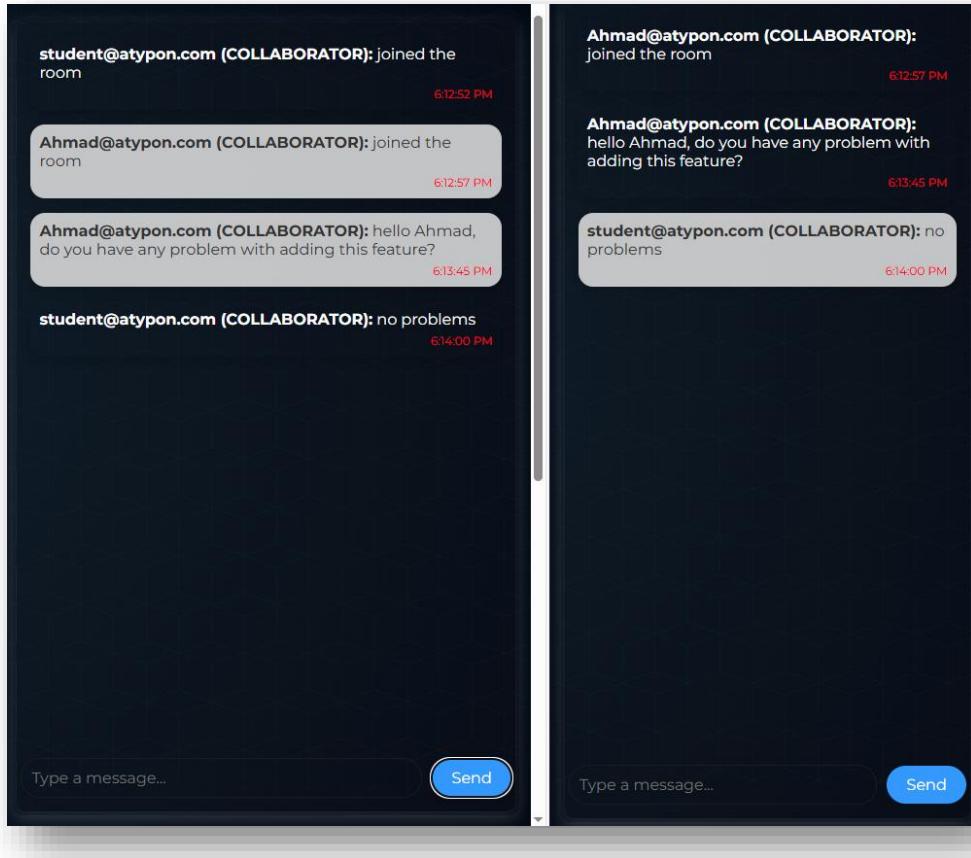


➤ Live Chat and Discussion:

Action: Collaborators can engage in live discussions with other members using the chat box on the right side of the page.

Process: This feature facilitates communication and decision-making, ensuring that Collaborators stay aligned while working on the code.

2.4 Collaborator: Editing, Versioning



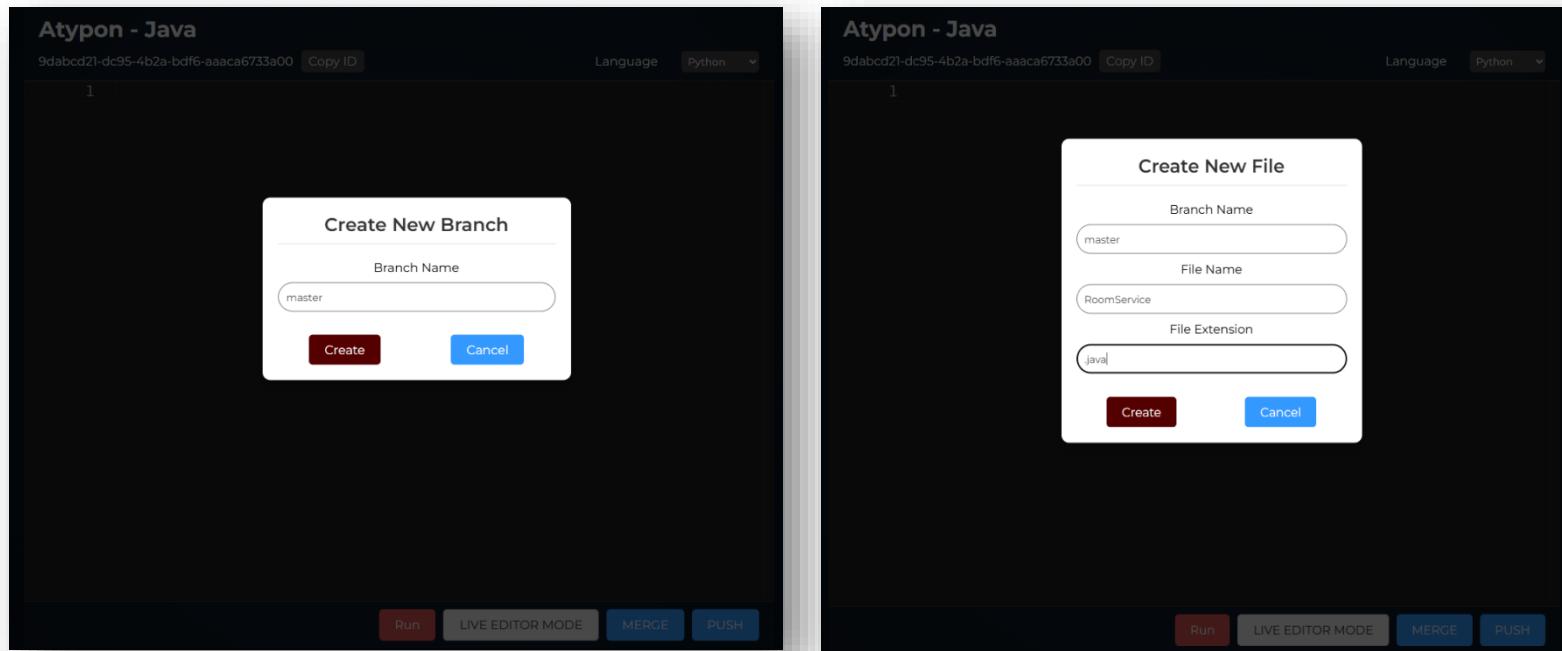
➤ Version Control (GitHub-like):

Action: Collaborators have full version control functionality similar to GitHub.

Key Processes:

- **Creating Files:** Collaborators can create new files by clicking “Create File”.
- **Creating Branches:** New branches can be created by clicking “Create project”.
- **Merging Code (MERGE):** Collaborators can merge the live-edited code, which will replace the persistent version stored in the database. If conflicts arise, they must be resolved manually within the editor.
- **Handling Conflicts (PUSH it after handling):** After addressing any code conflicts, the Collaborator must press “PUSH” to update the database with the new changes. This ensures that the final version is saved and reflected across the project.
- **Downloading Branches:** download branches in a ZIP file for offline review or backup.

2.4 Collaborator: Editing, Versioning



Note

It's important to mention that, for simplicity I have developed like this in the future work I want to make it in more details and features.

The image shows two side-by-side screenshots of the Atypon Java application interface, illustrating a merge conflict resolution process.

Screenshot 1 (Left): Conflict Resolution Dialog

A modal dialog titled "You must handle the conflicting manually, then press PUSH to save it in data base." is displayed over the code editor. The code editor shows Java code with a conflict resolution note:

```
1 /**
2 * =====
3 * WARNING: MANUAL CONFLICT RESOLUTION REQUIRED
4 * =====
5 * The programmer/user must resolve these conflicts
6 * manually,
7 * ensuring the code is correct. After resolving the
8 * conflicts,
9 * push the resolved version using the PUSH button
10 * in the interface.
11 */
12 import java.util.Scanner;
13 // 1. Import the Scanner class
14
15 class Main {
16     public static void main(String[] args) {
```

Screenshot 2 (Right): Current and Incoming Versions

The code editor shows the current version of the code and an incoming version from another source. The incoming version contains a method definition for "sum".

```
20 System.out.println("Enter your user name");
21 String userName = myObj.nextLine();
22 // 3. Read the user input with .nextLine()
23
24 System.out.println("The username is: " + userName);
25 }
26 =====
27 <<<<< Current Version
28 =====
29 >>>> Incoming Version
30 <<<<< Current Version
31 }
32 =====
33 public double sum(double x, double y){
34     >>>> Incoming Version
35     return x+y;
36 }
```

Both screenshots show a dark-themed interface with standard navigation buttons at the bottom: Run, LIVE EDITOR MODE, MERGE, and PUSH.

2.4 Collaborator: Editing, Versioning

The screenshot shows a Java code editor interface with the following details:

- Title:** Atypon - Java
- Code Area:** Displays Java code with line numbers 20 to 41. The code reads user input from `System.out` and prints it back.
- File Info:** File Name: RoomService.java, Project: master, Created At: 10/21/2024, 6:31:49 PM, Last Modified At: 10/21/2024, 6:32:45 PM
- Collaboration Log:** Shows messages from Ahmad@atypon.com (COLLABORATOR):
 - joined the room (6:32:57 PM)
 - performed a MERGE on RoomService (6:34:06 PM)
 - performed a PUSH on RoomService (6:36:39 PM)
- Buttons:** Switch to Light Mode, HOME, LEAVE ROOM, LOG OUT
- Status Bar:** Pushed to server successfully!
- Bottom Right:** Type a message... and Send button

➤ Running Code:

Action: Collaborators could run the code directly in the editor.

Process: The interface provides an input area where Collaborators can enter data, and an output area where the results of the code execution are displayed.

The screenshot shows a code editor interface with the following components:

- Input:** A text area containing "Abdullah Ayman Obaid".
- Output:** A text area showing the result of the code execution: "Executing: The username is: Abdullah Ayman Obaid".
- Top Buttons:** Run, LIVE EDITOR MODE, MERGE, PUSH
- Right Side:** Type a message... and Send button

2.4 Collaborator: Editing, Versioning

The image shows two instances of the Atypon Java editor interface. Both instances have the title 'Atypon - Java' and a file ID '9dabcd21-dc95-4b2a-bdf6-aaaca6733a00'. The top instance has a green arrow pointing to the code completion dropdown for the word 'Scanner'. The bottom instance has a green box highlighting the code completion dropdown for the variable 's', which lists options like 'static', 'Scanner', 'String', and 'System'.

Atypon - Java

9dabcd21-dc95-4b2a-bdf6-aaaca6733a00 Copy ID

```
1 import java.util.Scanner;
2
3 class Main {
4     public static void main(String[] args) {
5         // Scanner in = new Scanner(System.in);
6         System.out.println("Enter username");
7
8         String userName = in.nextLine(); //please rewrite this Line
9         of code
10
11         System.out.println("Username is: " + userName);
12     }
13
14 }
15 }
```

Atypon - Java

9dabcd21-dc95-4b2a-bdf6-aaaca6733a00 Copy ID

```
1 import java.util.Scanner;
2
3 class Main {
4     public static void main(String[] args) {
5         // Scanner in = new Scanner(System.in);
6         System.out.println("Enter username");
7
8         String userName = in.nextLine(); //please rewrite this Line
9         of code
10
11         System.out.println("Username is: " + userName);
12
13         int s
14             * static
15             * Scanner
16             * String
17             * System
18     }
19
20 }
21 }
```

Run LIVE EDITOR MODE MERGE PUSH

Run LIVE EDITOR MODE MERGE PUSH

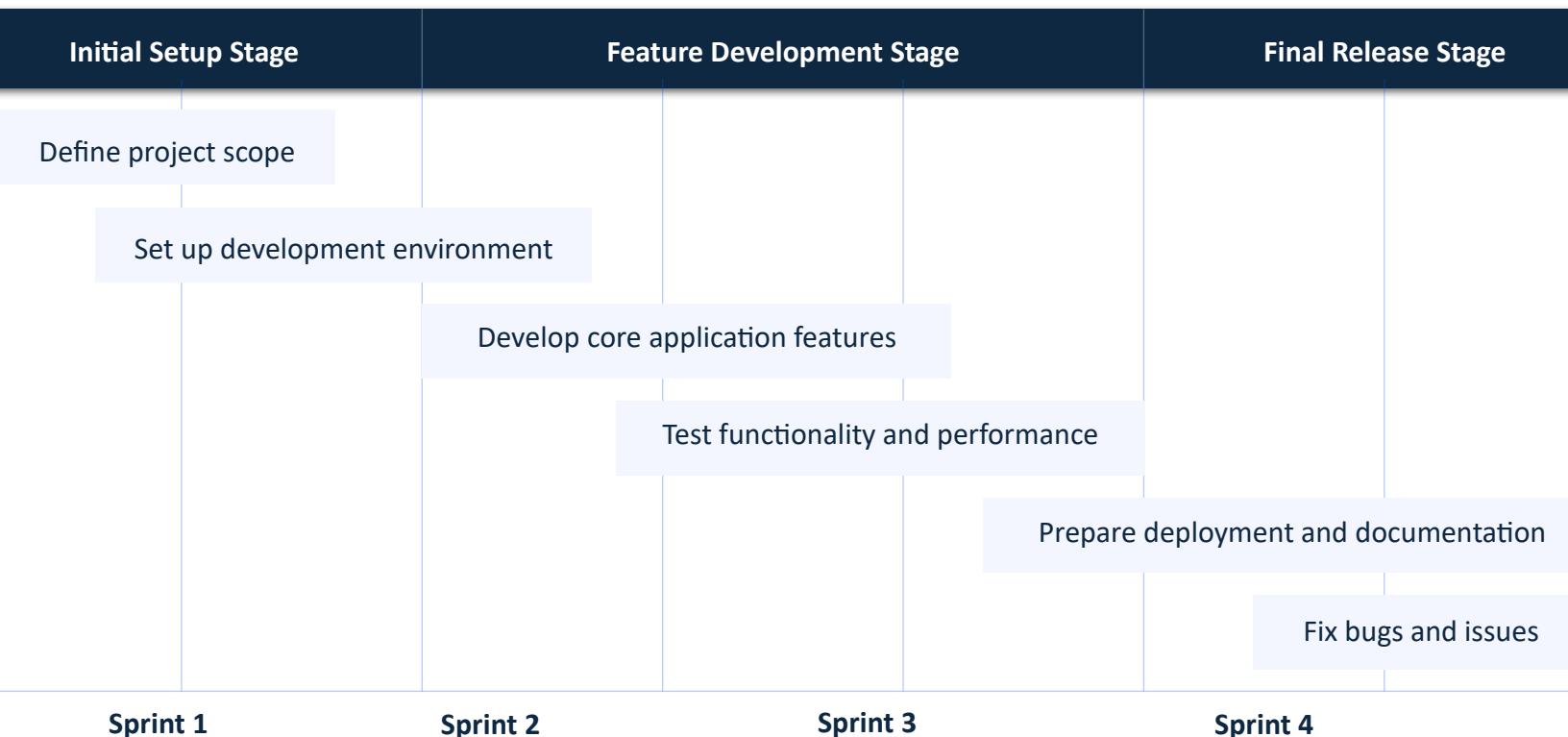
Note

it's important to mention that our editor has different functionalities that make developer's life easier, these features like those ones in VS Editor.

System Design and Architecture

Before starting any software project, it's essential to plan effectively, and Agile methodology provides the framework to do so in a structured yet flexible manner. Our project will be divided into four **one-week sprints**, each focused on delivering key features incrementally. This short sprint cycle allows for frequent feedback. By organizing the work into manageable, focused sprints, we ensure consistent progress.

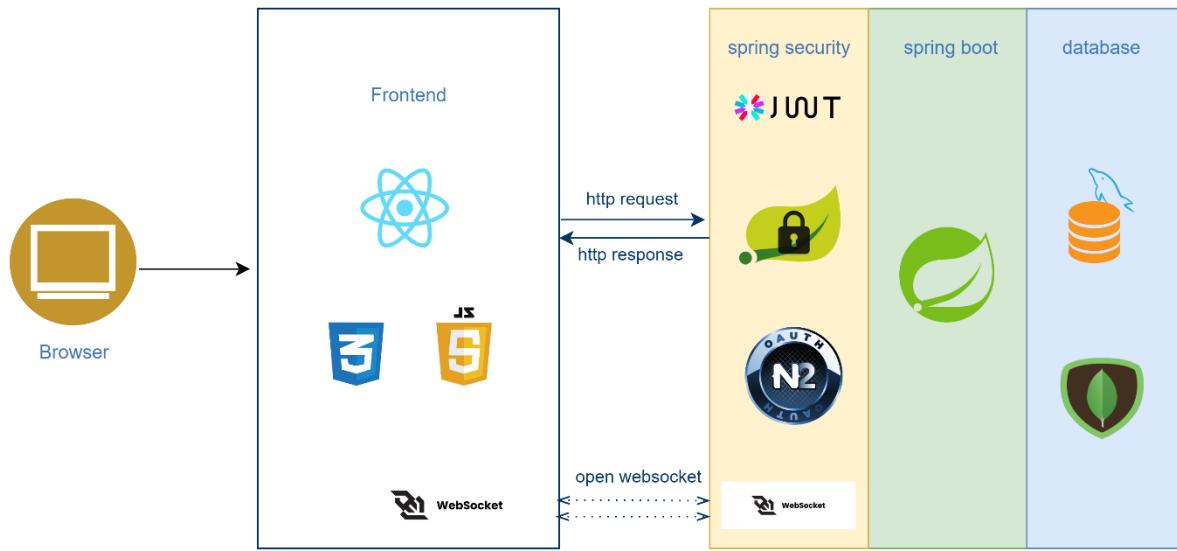
3.1 Agile project roadmap



OVERVIEW

This Agile project focuses on delivering a robust software solution in three key stages: initial setup, feature development, and final release. Over four weeks, I will define the project scope, develop core features, test performance, fix bugs, and prepare for deployment, ensuring timely and efficient delivery.

3.2 Overview of System Architecture



- **Client-Side (Frontend):**
 - **WebSocket for Real-Time Collaboration:** Upon accessing a room, a WebSocket connection is opened between the frontend and the server. This connection supports real-time code editing, live updates, and chat functionality among Collaborators.
 - **User Authentication:** The login process initiates when a user enters their credentials (username and password or OAuth provider), which are sent to the backend via an HTTP request.
- **Backend (Server-Side):**
 - **Spring Boot (Java):** The server is built using the Spring Boot framework.
 - **Authentication Flow:**
 - **Spring Security (OAuth 2.0, JWT).**
 - **JWT Token for Authorization.**
 - **Role-Based Access Control (RBAC).**
- **Database Layer:**
 - **MongoDB (NoSQL)** for File Versions and Code Updates.
 - **MySQL (Relational)** for Room and Project Management.
- **WebSocket Communication:**
 - **Real-Time Code Editing:**
 - **Live Chat.**

3.3 Database Design



In this section, I will explain the schema design choices for both MySQL and MongoDB, along with the challenges encountered, and propose diagram suggestions to illustrate the structure. The database design of the collaborative code editor integrates both relational and non-relational databases, making use of MySQL for structured relationships (**users, rooms, projects**) and MongoDB for unstructured, high-volume **data (file versions, real-time code updates, logs)**. This hybrid database approach ensures a balance between the structured data relationships and flexible, dynamic data storage for file versioning and collaboration.

1. MySQL Schema Design:

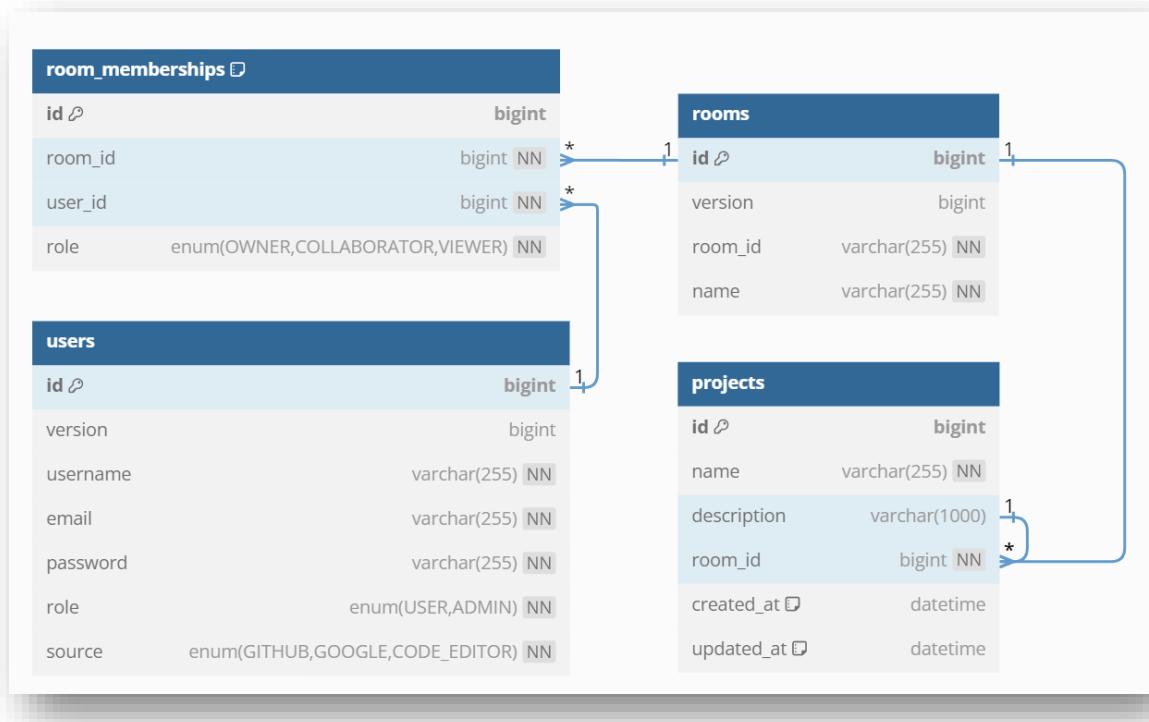
The relational database in MySQL is used to manage users, rooms, projects, and their relationships. Each entity has its own table with structured relationships to represent ownership, project management, and membership roles.

Key Entities in MySQL:

- **User:** Represents users in the system, each having a unique username, email, and role (e.g., USER, ADMIN). A user can have different memberships in various rooms.
- **Room:** Each room is a collaborative space where projects are managed. Rooms are associated with multiple users through the RoomMembership table, defining their roles (Owner, Collaborator, Viewer).
- **Project:** Each room contains multiple projects. Projects have a relationship with a room and store details such as name, description, and timestamps (createdAt, updatedAt).
- **RoomMembership:** This table handles the many-to-many relationship between users and rooms, allowing users to join multiple rooms with specific roles (**Owner, Collaborator, Viewer**).

3.3 Database Design

- ❖ **Schema Diagram for MySQL:** In this diagram, tables like User, Room, Project, and RoomMembership represented with relationships:



2. MongoDB Schema Design:

MongoDB is used for dynamic, high-volume data, such as file versions, real-time code updates, and chat logs. The flexibility of MongoDB allows storing various file versions and real-time code changes efficiently.

Key Collections in MongoDB:

- **File:** Stores information about individual file versions. Each document stores details such as filename, roomId, projectName, content, creation and modification timestamps, and file extension.
- **CodeUpdate:** This collection stores real-time code updates during live editing. It tracks which user made changes, which file and project were affected, the line number, and the specific changes made.
- **Project:** Each room contains multiple projects. Projects have a relationship with a room and store details such as name, description, and timestamps (createdAt, updatedAt).
- **MessageLog:** Stores logs of messages sent in the chat, tracking the sender, role, filename, room, and project. This helps in keeping a detailed history of collaboration activities.

3.3 Database Design

❖ Design Considerations and Problems Faced:

- **Cascading in Relational Models:** One issue encountered was managing cascading operations between rooms and users. To prevent unwanted deletions when removing a room, a **RoomMembership** entity was introduced to **separate concerns between rooms and users**.
 - **File Updates in MongoDB:** Updating documents in MongoDB with dynamic content, especially file updates, proved challenging. The requirement to pass the entire document for updates **does not align with clean code principles**, leading to a more verbose update logic compared to MySQL's simpler save operation.

3.4 Security Design



In this section, I will explain the security architecture of the collaborative code editor project:

1. General Technologies Used: For the security design in this project, I employed the following technologies to ensure secure authentication and access control:

- **OAuth 2.0** for third-party authentication with GitHub and Google.
- **JWT (JSON Web Tokens)** for token-based authentication to maintain stateless sessions between the front end and back end.
- **Role-Based Access Control (RBAC)** to manage permissions for three user roles: Owner, Collaborator, and Viewer.
- **Spring Security** to handle HTTP requests, authentication, and authorization.
- **BCrypt** for password hashing and storage, ensuring secure user credentials.

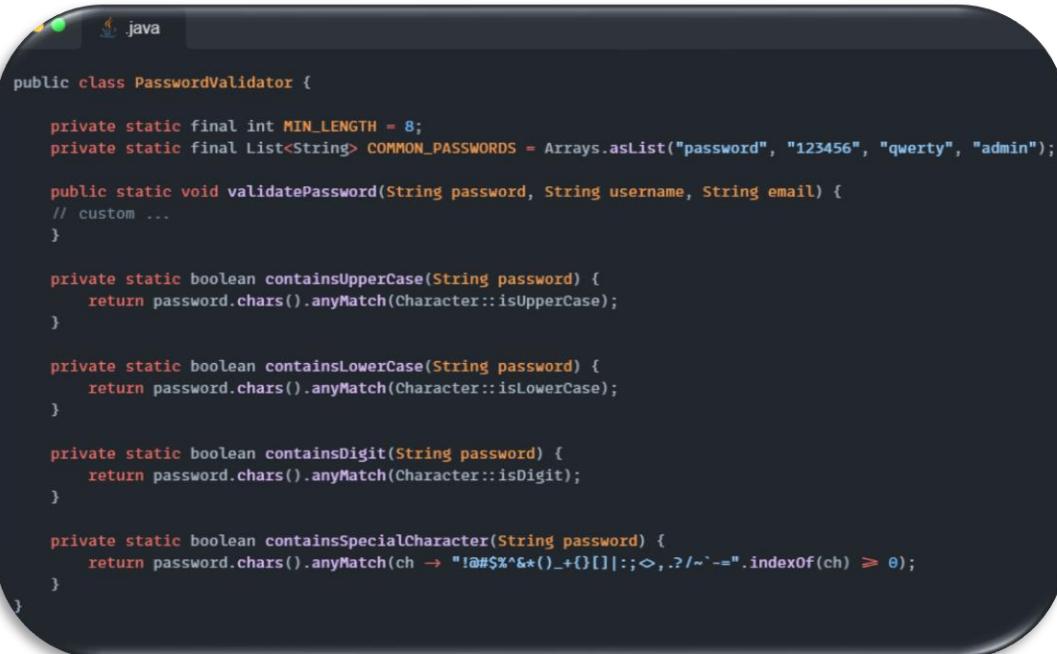
2. Custom Sign-Up Implementation: The system supports its own user registration process alongside OAuth 2.0. In the custom sign-up process, users provide their email, username, and password. These are validated using specific criteria before being stored in the database.

Sign-Up Workflow:

- The password must meet certain security standards, such as having a minimum length of 8 characters, containing uppercase and lowercase letters, digits, and special characters.
- **Email Validator:** Ensures that the email format is valid.

3.4 Security Design

- **Password Validator:** Checks if the password is secure enough, avoiding common passwords, and not allowing personal information (username or email) within the password.
- The password is hashed using **BCrypt** for security, ensuring no plaintext passwords are saved in the database.



```
java

public class PasswordValidator {

    private static final int MIN_LENGTH = 8;
    private static final List<String> COMMON_PASSWORDS = Arrays.asList("password", "123456", "qwerty", "admin");

    public static void validatePassword(String password, String username, String email) {
        // custom ...
    }

    private static boolean containsUpperCase(String password) {
        return password.chars().anyMatch(Character::isUpperCase);
    }

    private static boolean containsLowerCase(String password) {
        return password.chars().anyMatch(Character::isLowerCase);
    }

    private static boolean containsDigit(String password) {
        return password.chars().anyMatch(Character::isDigit);
    }

    private static boolean containsSpecialCharacter(String password) {
        return password.chars().anyMatch(ch -> !"!@#$%^&*()_+{}[];:<,.?/-`-= ".indexOf(ch) >= 0);
    }
}
```

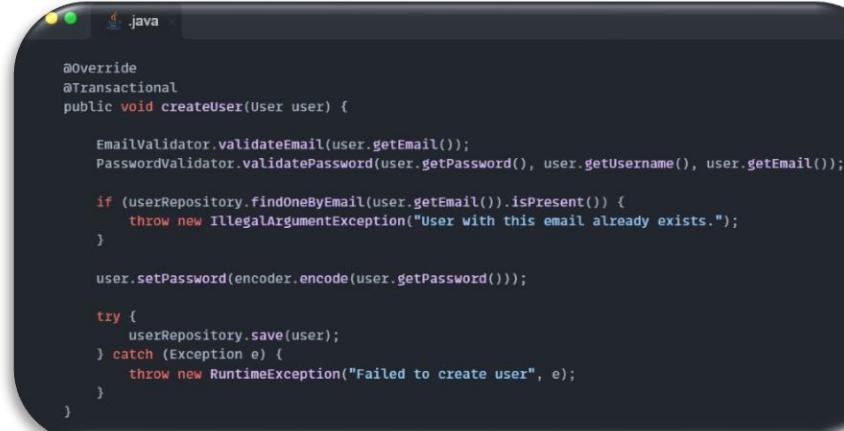


```
java x

public class EmailValidator {

    private static final String EMAIL_REGEX = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9+_.-]+$";
    private static final Pattern EMAIL_PATTERN = Pattern.compile(EMAIL_REGEX);

    public static void validateEmail(String email) {
        if (email == null || !EMAIL_PATTERN.matcher(email).matches()) {
            throw new IllegalArgumentException("Invalid email format");
        }
    }
}
```



```
java

@Override
@Transactional
public void createUser(User user) {
    EmailValidator.validateEmail(user.getEmail());
    PasswordValidator.validatePassword(user.getPassword(), user.getUsername(), user.getEmail());

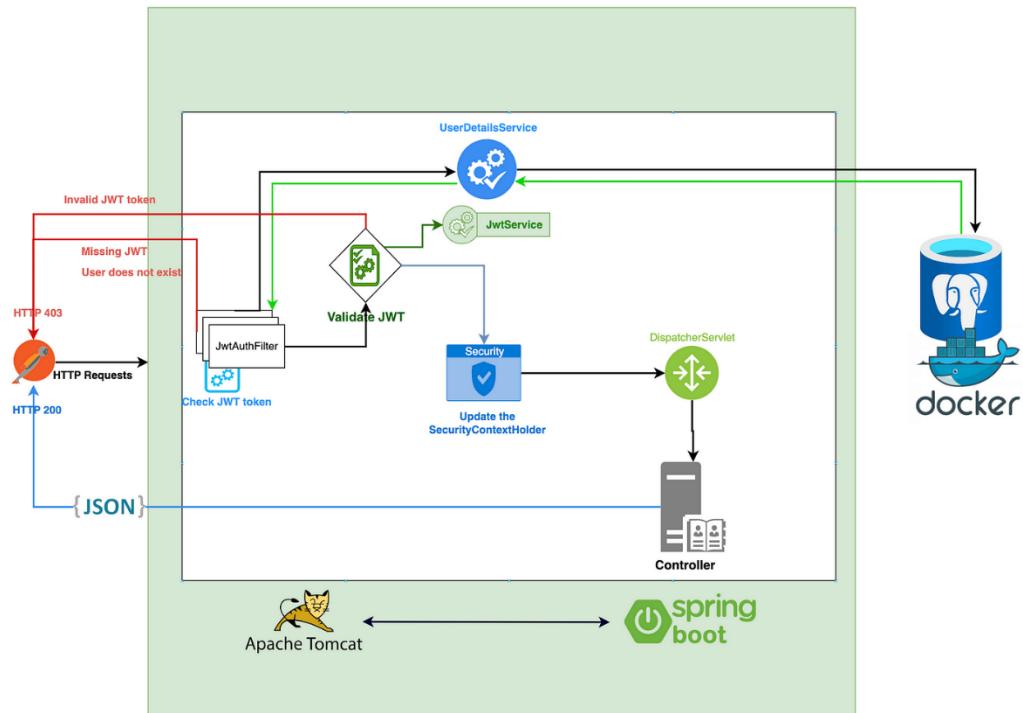
    if (userRepository.findOneByEmail(user.getEmail()).isPresent()) {
        throw new IllegalArgumentException("User with this email already exists.");
    }

    user.setPassword(encoder.encode(user.getPassword()));

    try {
        userRepository.save(user);
    } catch (Exception e) {
        throw new RuntimeException("Failed to create user", e);
    }
}
```

3.4 Security Design

3. JWT-Based Authentication Flow: Once a user logs in using their credentials, a JWT token is generated and returned to the front end. This token is used for all future API requests, enabling stateless authentication between the client and server.



JWT Authentication Workflow:

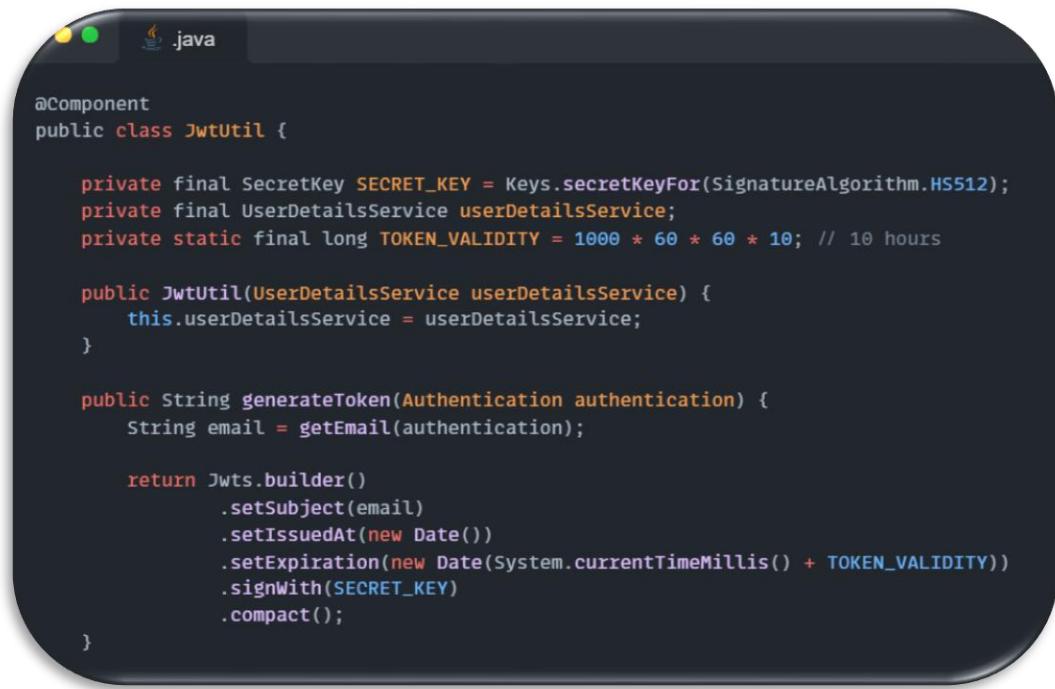
- The user provides login credentials (email and password) via a /sign-in endpoint.
- The AuthenticationService verifies the credentials using Spring Security's AuthenticationManager and returns a JWT upon successful verification.
- The JWT is generated using the user's email and an expiration time (10 hours in this case).
- The JWT is then sent back to the front end and stored (typically in the browser's local storage).
- For any future API requests, the JWT is included in the request's Authorization header (Bearer <token>).
- The back end validates the JWT and processes the request if the token is valid.

A screenshot of a terminal window showing Java code. The window title is '.java'. The code is a method named 'verify' that takes a 'LoginRequest' and returns a string. It uses an 'authenticateUser' method to get an 'Authentication' object and then calls 'jwtUtil.generateToken' on it. The code ends with a closing brace '}'.

```
public String verify(LoginRequest loginRequest) {
    Authentication authentication = authenticateUser(loginRequest);
    return jwtUtil.generateToken(authentication);
}
```

3.4 Security Design

- The token contains information like the user's email, issued time, and expiration, and it's signed using a **HMAC SHA-512** algorithm.



```
@Component
public class JwtUtil {

    private final SecretKey SECRET_KEY = Keys.secretKeyFor(SignatureAlgorithm.HS512);
    private final UserDetailsService userDetailsService;
    private static final long TOKEN_VALIDITY = 1000 * 60 * 60 * 10; // 10 hours

    public JwtUtil(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    public String generateToken(Authentication authentication) {
        String email = getEmail(authentication);

        return Jwts.builder()
            .setSubject(email)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + TOKEN_VALIDITY))
            .signWith(SECRET_KEY)
            .compact();
    }
}
```

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvG4gRG91IiwiYWRtaW4iOnRydWUsImIhdCI6MTUxNjIzOTAyMn0.VFb0qJ1LRg_4ujbZoRMXnVkJUgiuKq5KxWqNdbKq_G9Vvz-S1zZa9LPxtHWKa64zD12ofkT8F6jBt_K4riU-fPg
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS512",
  "typ": "JWT"
}
```

PAYOUT: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

VERIFY SIGNATURE

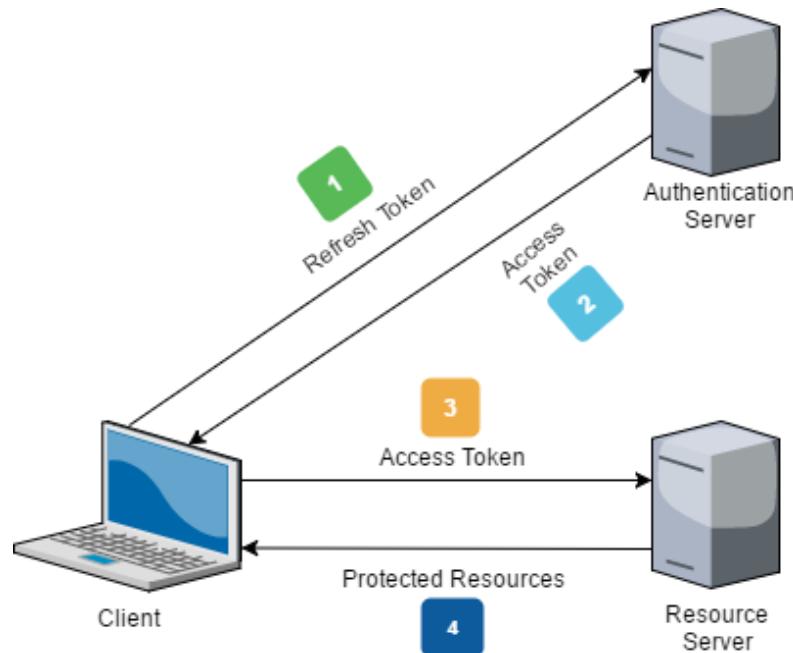
```
HMACSHA512(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-512-bit-secret
) □ secret base64 encoded
```

⌚ Signature Verified

SHARE JWT

3.4 Security Design

4. **OAuth 2.0 Authentication Flow:** In addition to custom sign-in, the system integrates OAuth 2.0 to allow users to authenticate via third-party providers like Google and GitHub.



OAuth 2.0 Workflow:

- When a user selects Google or GitHub login, they are redirected to the respective provider's OAuth authorization page.
- The authorization code is exchanged for an access token from Google or GitHub.
- The access token is used to retrieve user details (e.g., email, username) from the OAuth provider
- If the user exists in the system, they are authenticated and a JWT token is generated for the session.
- If the user is new, a new account is created in the system.

JWT Token with OAuth:

- After successful OAuth login, a new JWT token is generated for the user and sent back to the front end for future authenticated requests, just as it would be for a regular login.

```
java
@Override
public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response)
    if (authentication instanceof OAuth2AuthenticationToken oauthToken) {
        String registrationId = oauthToken.getAuthorizedClientRegistrationId();
        if ("google".equalsIgnoreCase(registrationId)) {
            handleGoogleLogin(oauthToken.getPrincipal());
        } else if ("github".equalsIgnoreCase(registrationId)) {
            handleGitHubLogin(oauthToken.getPrincipal());
        }
    }
    String jwtToken = generateJwtToken(authentication);
    response.sendRedirect("http://localhost:3000/oauth2/redirect?token=" + jwtToken);
}
```

3.4 Security Design

5. Role-Based Access Control: In this project, Role-Based Access Control (RBAC) has **three roles** in the system: **Owner, Collaborator, and Viewer**. These roles define what actions a user can perform within a room, such as editing files, managing projects, or simply viewing content.

Security Configuration

- The security configuration enables global method security, allowing us to use annotations like `@PreAuthorize` to control access at the method level.



```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
// Enable @PreAuthorize annotation
public class SecurityConfig {
    // Security setup and configurations go here
}
```

Role Checks in the Service Layer: In the service layer, the **RoomSecurityService** interacts with the **RoomMembershipService** to determine the roles of users within rooms.

```
public List<RoomRole> getUserRolesForRoom(String userEmail, String roomId) {  
    ...  
    if (userRoles.isEmpty() || userRoles.get().isEmpty()) {  
        throw new AccessDeniedException("User does not have access to this room");  
    }  
    ...  
}  
  
public boolean isOwner(String userEmail, String roomId) {  
    return roomMembershipService.isOwner(userEmail, roomId);  
}  
  
public boolean isCollaborator(String userEmail, String roomId) {  
    return roomMembershipService.isCollaborator(userEmail, roomId);  
}  
  
public boolean isViewer(String userEmail, String roomId) {  
    return roomMembershipService.isViewer(userEmail, roomId);  
}
```

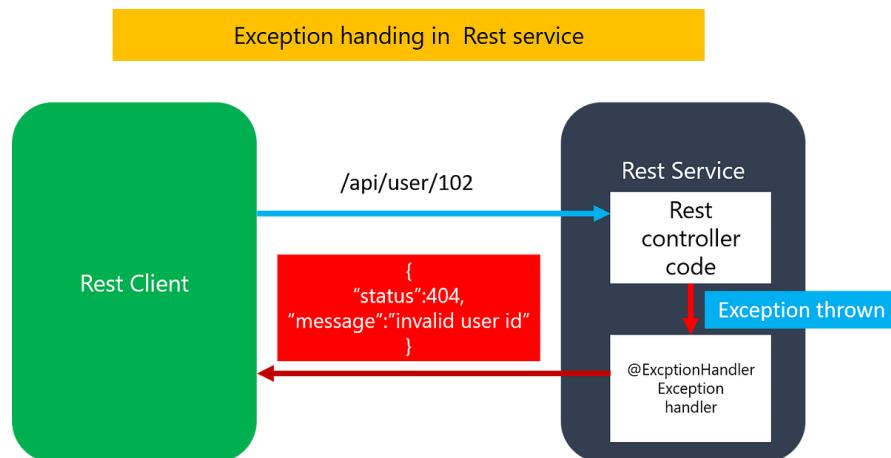
Room Access Control

The key access methods are protected using @PreAuthorize annotations. These annotations check the user's role in the room before allowing access to certain

```
java

@PreAuthorize("#roomSecurityService.canViewRoom(principal.username, #roomId)")
@PostMapping("/join-room/{roomId}")
public ResponseEntity<String> joinRoom(@PathVariable("roomId") String roomId,
    return ResponseEntity.ok("Room");
}
```

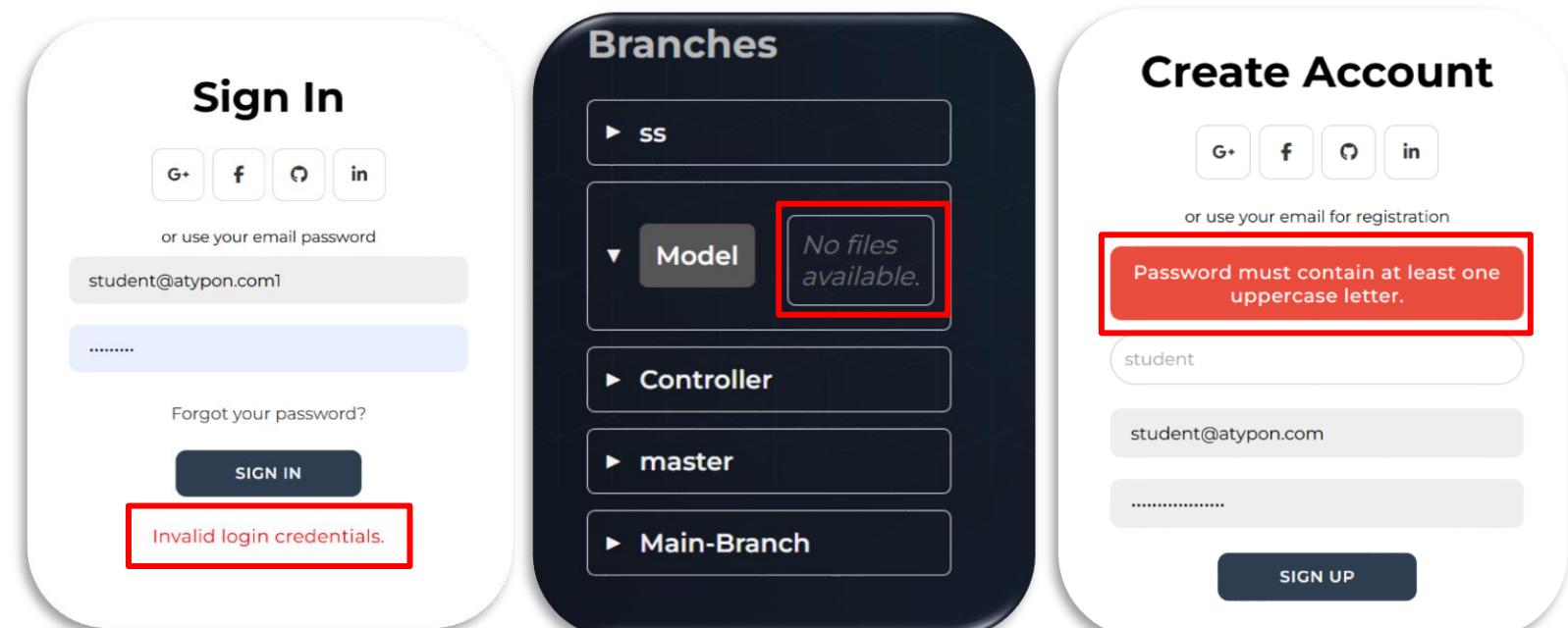
3.5 Exception Handling Design



In my project, I've implemented a centralized exception handling mechanism using **@RestControllerAdvice** and a **GlobalExceptionHandler** class. This approach enables consistent and organized handling of exceptions that may arise throughout the application. By using this design, exceptions are handled in a clean, maintainable manner.

Exception Types: The system handles multiple types of exceptions:

- **User-Related Exceptions:** E.g., `UserNotFoundException`, `InvalidCredentialsException`.
- **File and Project Exceptions:** E.g., `FileNotFoundException`, `FileAlreadyExistsException`, `ProjectNotFoundException`.
- **Room Exceptions:** E.g., `RoomNotFoundException`, `RoomCreationException`, `RoomDeletionException`.
- **General Exceptions:** A fallback handler for uncaught exceptions, which returns a generic error message.



3.5 Exception Handling Design



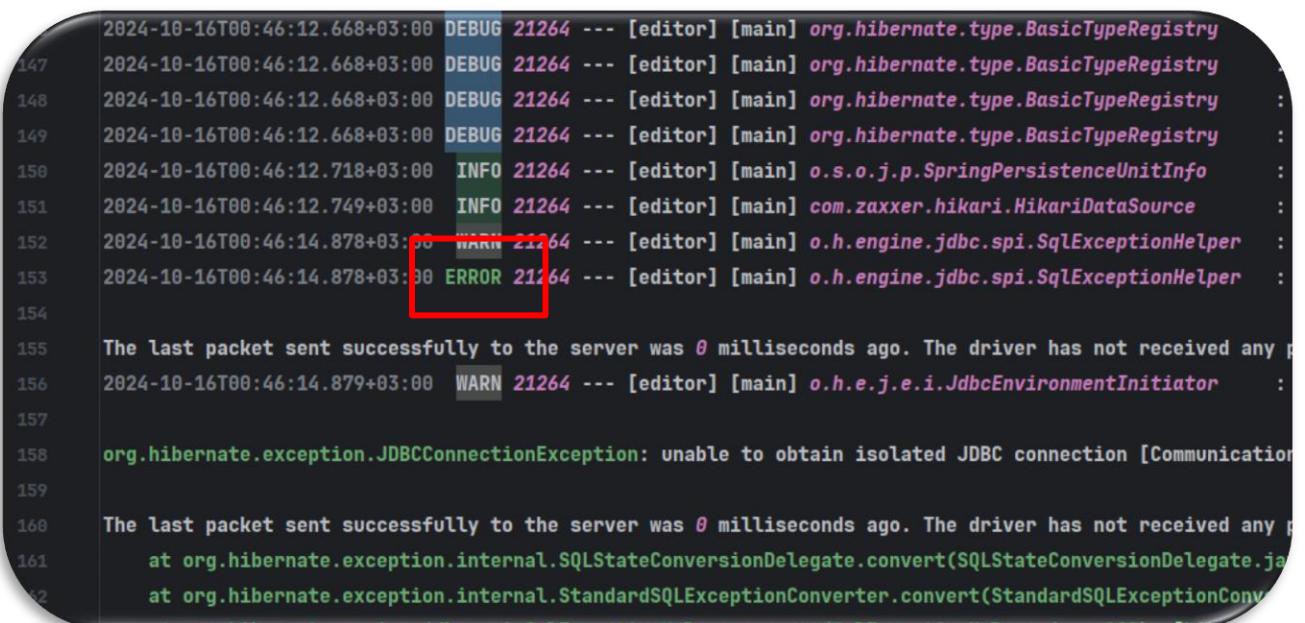
```
ExceptionHandler(UserNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public Map<String, String> handleUserNotFoundException(UserNotFoundException ex) {
    logger.warn("User not found: {}", ex.getMessage(), ex);
    return createErrorResponse("User Not Found", ex.getMessage(), "USER_NOT_FOUND");
}

ExceptionHandler(RoomCreationException.class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
public Map<String, String> handleRoomCreationException(RoomCreationException ex) {
    logger.error("Room creation failed: {}", ex.getMessage(), ex);
    return createErrorResponse("Room Creation Failed", ex.getMessage(), "ROOM_CREATION_FAILED");
}
```

Logging (LoggerFactory SLF4)

plays a role in allowing the system administrator to trace errors and improve the system. By utilizing SLF4J's LoggerFactory, the GlobalExceptionHandler logs each exception according to its severity:

- **Warnings:** These are for recoverable issues like invalid credentials or failed room updates. These logs help the admin identify areas where the user experience might be improved or where minor issues are occurring.
- **Errors:** Critical problems such as missing resources (e.g., user, project, room not found) or internal server errors are logged as errors.



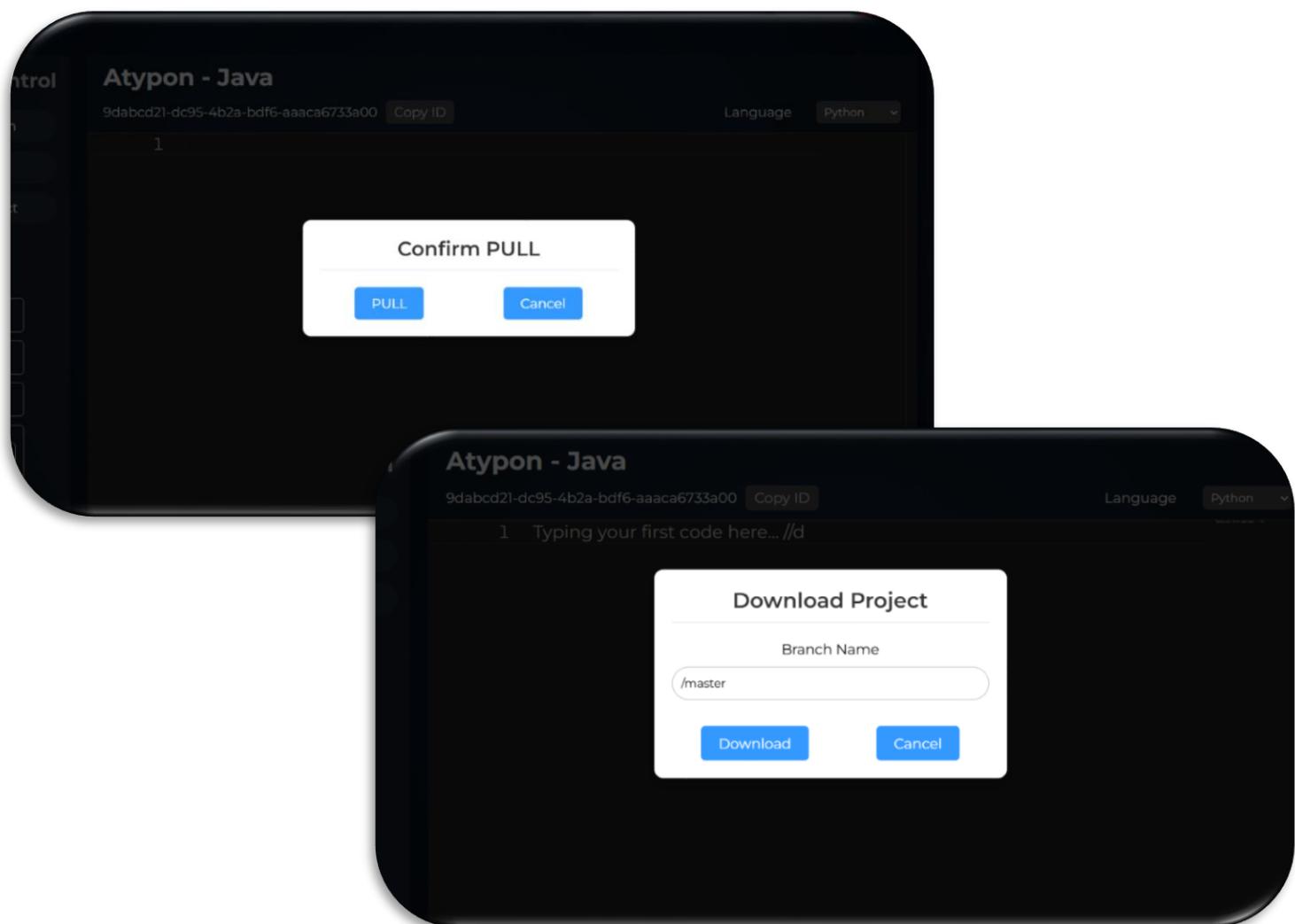
```
2024-10-16T00:46:12.668+03:00 DEBUG 21264 --- [editor] [main] org.hibernate.type.BasicTypeRegistry
147 2024-10-16T00:46:12.668+03:00 DEBUG 21264 --- [editor] [main] org.hibernate.type.BasicTypeRegistry
148 2024-10-16T00:46:12.668+03:00 DEBUG 21264 --- [editor] [main] org.hibernate.type.BasicTypeRegistry
149 2024-10-16T00:46:12.668+03:00 DEBUG 21264 --- [editor] [main] org.hibernate.type.BasicTypeRegistry
150 2024-10-16T00:46:12.718+03:00 INFO 21264 --- [editor] [main] o.s.o.j.p.SpringPersistenceUnitInfo
151 2024-10-16T00:46:12.749+03:00 INFO 21264 --- [editor] [main] com.zaxxer.hikari.HikariDataSource
152 2024-10-16T00:46:14.878+03:00 WARN 21264 --- [editor] [main] o.h.engine.jdbc.spi.SqlExceptionHelper
153 2024-10-16T00:46:14.878+03:00 ERROR 21264 --- [editor] [main] o.h.engine.jdbc.spi.SqlExceptionHelper
154
155 The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any p
156 2024-10-16T00:46:14.879+03:00 WARN 21264 --- [editor] [main] o.h.e.j.e.i.JdbcEnvironmentInitiator
157
158 org.hibernate.exception.JDBCConnectionException: unable to obtain isolated JDBC connection [Communication
159
160 The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any p
161     at org.hibernate.exception.internal.SQLExceptionConversionDelegate.convert(SQLExceptionConversionDelegate.java:102)
162     at org.hibernate.exception.internal.StandardSQLExceptionConverter.convert(StandardSQLExceptionConverter.java:53)
```

3.6 Version Control Design

In my project, I designed the version control system to mimic GitHub-like version control but with the added feature of real-time editing. The following explains the key functionalities and how they are implemented:

1. Pulling and Downloading Code

- The pull action in my system allows users to pull the code currently being edited in real time. This brings the live version of the code into the editor, enabling collaborators to see and modify it together. However, if a user wishes to **pull an existing version stored in the database**, they can use the **Download** button. The code for downloading the project is handled by ProjectDownloadService:
 - The response headers are set to ensure that the files are downloaded as a ZIP. The files are written into the ZIP using a ZipOutputStream, allowing the user to download the entire project in a compressed format.



3.6 Version Control Design

4. Merging Code:

The merge functionality in my system compares the code currently being edited in the live editor with the original version stored in the database. If there are conflicts, the system identifies and inserts conflict markers. This requires the user to manually resolve the conflicts before proceeding. The merging process works as follows:

- The original version is retrieved from the database.
- The system compares the lines from the original version with the new version from the editor.
- When there are differences, the system adds conflict markers (**<<<<<**, **=====**, **>>>>>**), similar to Git, and returns the merged content to the user, who must **manually resolve the conflict**.

The figure consists of three screenshots of the Atypon Java editor interface, showing the process of merging code.

Screenshot 1: Shows the initial Java code in the editor. The code reads a username from standard input and prints it back. It includes a `sum` method.

```
3 class Main {  
4     public static void main(String[] args) {  
5         Scanner myObj = new Scanner(System.in); // Create a  
6         Scanner object  
7         System.out.println("Enter username");  
8  
9         String userName = myObj.nextLine(); // Read user input  
10        System.out.println("Username is: " + userName); //  
11        Output user input  
12    }  
13    public double sum(double x, double y){  
14        return x+y;  
15    }  
16 }
```

Screenshot 2: Shows a confirmation dialog titled "Confirm Code Merge" with "MERGE" and "Cancel" buttons.

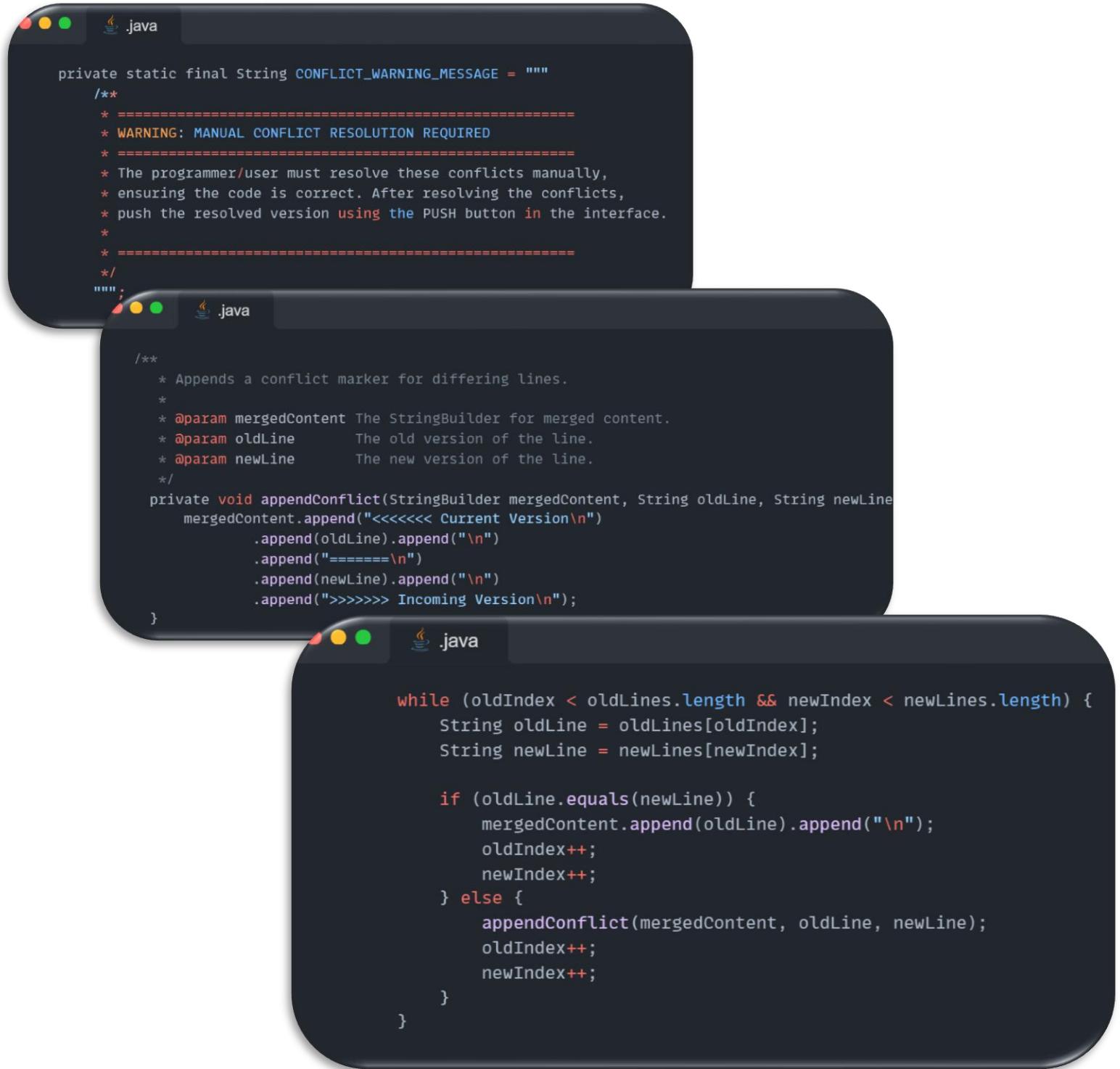
Screenshot 3: Shows the merged code after the merge button was clicked. Conflict markers (`<<<<<`, `=====`, `>>>>>`) are inserted at the beginning of the incoming version's code. The merged code now includes both the original code and the incoming version's code.

```
17 System.out.println("Enter username");  
18  
19 String userName = myObj.nextLine(); // Read user input  
20 System.out.println("Username is: " + userName); //  
21         Output user input  
22 <<<<< Current Version  
23  
24 =====  
25 | public double sum(double x, double y){  
26 |>>>>> Incoming Version  
27 <<<<< Current Version  
28 |}  
29 =====  
30 | return x+y;  
31 |>>>>> Incoming Version  
32 |}  
33 |}  
34 }
```

3.6 Version Control Design

Conflict Handling

I implemented a conflict resolution system within the **FileMergeHandler**, which automatically detects and marks conflicting sections of the code.



The image shows three overlapping Java code editor windows, each with a .java file icon in the title bar. The windows are arranged in a staggered fashion, with the top-left window being the smallest and the bottom-right window being the largest.

The code in the windows illustrates a conflict resolution mechanism:

- Top Window:** Contains a static final string `CONFLICT_WARNING_MESSAGE` with a multi-line comment explaining that manual conflict resolution is required and the programmer/user must resolve conflicts manually after ensuring the code is correct.
- Middle Window:** Contains a private method `appendConflict` that appends a conflict marker for differing lines. It takes three parameters: `mergedContent` (StringBuilder), `oldLine` (String), and `newLine` (String). The method appends the current version, followed by the old line, a separator, the new line, and the incoming version.
- Bottom Window:** Contains a loop that iterates through two arrays of strings, `oldLines` and `newLines`. It compares the current old line and new line. If they are equal, it appends the old line and increments both indices. If they are not equal, it calls the `appendConflict` method and then increments both indices.

```
private static final String CONFLICT_WARNING_MESSAGE = """  
/**  
 * ======  
 * WARNING: MANUAL CONFLICT RESOLUTION REQUIRED  
 * ======  
 * The programmer/user must resolve these conflicts manually,  
 * ensuring the code is correct. After resolving the conflicts,  
 * push the resolved version using the PUSH button in the interface.  
 *  
 * ======  
 */  
""";  
  
/**  
 * Appends a conflict marker for differing lines.  
 *  
 * @param mergedContent The StringBuilder for merged content.  
 * @param oldLine The old version of the line.  
 * @param newLine The new version of the line.  
 */  
private void appendConflict(StringBuilder mergedContent, String oldLine, String newLine)  
    mergedContent.append("<<<< Current Version\n")  
        .append(oldLine).append("\n")  
        .append("=====\\n")  
        .append(newLine).append("\n")  
        .append(">>>> Incoming Version\\n");  
}  
  
while (oldIndex < oldLines.length && newIndex < newLines.length) {  
    String oldLine = oldLines[oldIndex];  
    String newLine = newLines[newIndex];  
  
    if (oldLine.equals(newLine)) {  
        mergedContent.append(oldLine).append("\n");  
        oldIndex++;  
        newIndex++;  
    } else {  
        appendConflict(mergedContent, oldLine, newLine);  
        oldIndex++;  
        newIndex++;  
    }  
}
```

Note

It's important to mention that once conflicts have been resolved, users need to **push** the final version of the code to the database.

3.7 Code Executor Design

```
Input
Abdullah
22
1000

Output
Executing:
Name: Abdullah
Age: 22
Salary: 1000.0
```

In my project, the Code Executor Design relies on Docker to run code in an isolated environment, ensuring security and scalability. Here's how I structured the system:

```
Dockerfile

# Install Docker CLI for running Docker commands from within the container
RUN apt-get update && apt-get install -y docker.io
```

1. Execution Handling

I used a **Docker-in-Docker** approach to execute code within containers based on the language being used (e.g., Java, Python, C++). This design pulls the appropriate Docker image and executes the code in an isolated environment. This ensures that the code runs safely without affecting the main server.

Problems Faced in this approach:

- The main downside is that handling execution on the same server as the rest of the application can cause performance issues. For example:
- **Image preparation time:** Pulling Docker images and preparing containers takes time, which could lead to delays in execution.
- **Server load:** Running multiple containers on the main server may cause resource bottlenecks, especially with many users.

Note

Despite these trade-offs, **I chose this approach because it fits the project requirements and provides a controlled, isolated environment for code execution.** By running the code inside Docker, **it ensures that no malicious code can affect the system, as every execution happens in a sandboxed container.**

3.7 Code Executor Design

2. JSON-Based Command Template

One of the features of my design is the use of **JSON-based templates** to handle **commands for different programming languages**. This allows **flexibility in adding new languages or modifying commands without changing the codebase**.

Each language's Docker image and corresponding command template is defined in a JSON file. This structure allows me to easily extend the system with new languages or modify the execution logic for existing ones. For example:

- The template for Python pulls the python:3.9 image and runs the code using python3.
- The template for C++ pulls the gcc image and compiles and runs the code within a single command.

This design gives me the flexibility to dynamically support additional languages by simply adding new entries to the JSON configuration file.



```
{  
    "python": {  
        "dockerImage": "python:3.9",  
        "commandTemplate": "echo \"{input}\" | python3 -c \"{code}\""  
    },  
    "javascript": {  
        "dockerImage": "node:14",  
        "commandTemplate": "echo \"{input}\" | node -e \"{code}\""  
    },  
    "java": {  
        "dockerImage": "openjdk:17",  
        "commandTemplate": "echo \"{code}\" > Main.java && javac Main.java && echo \"{input}\" | java Main"  
    },  
    "cpp": {  
        "dockerImage": "gcc:latest",  
        "commandTemplate": "echo \"{code}\" > main.cpp && g++ main.cpp -o main && echo \"{input}\" | ./main"  
    },  
    "go": {  
        "dockerImage": "golang:latest",  
        "commandTemplate": "echo \"{input}\" > main.go && echo \"{input}\" | go run main.go"  
    },  
    "ruby": {  
        "dockerImage": "ruby:latest",  
        "commandTemplate": "echo \"{code}\" > main.rb && ruby main.rb"  
    }  
}  
  
@PostConstruct  
public void loadCommandTemplates() throws Exception {  
    ObjectMapper objectMapper = new ObjectMapper();  
    InputStream templateStream = new ClassPathResource("data/commandTemplates.json").getInputStream();  
    commandTemplates = objectMapper.readValue(templateStream, new TypeReference<>() {});  
}
```

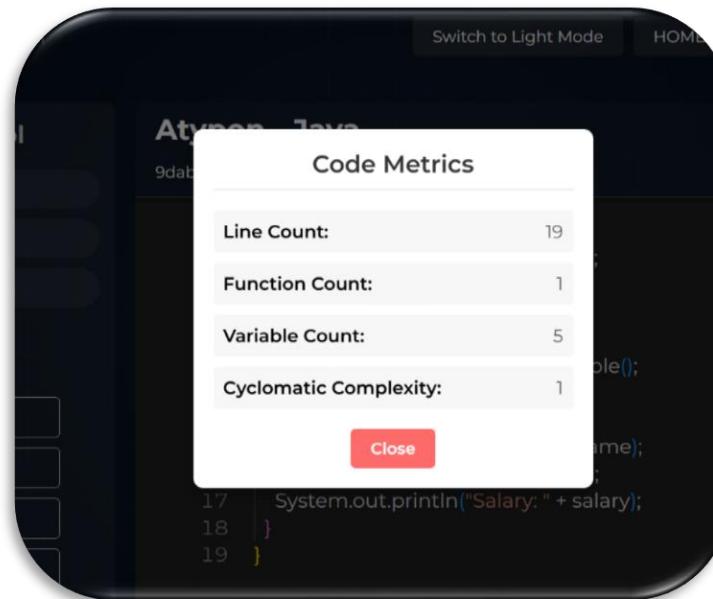
3.8 Code Metrics Feature

I implemented a Metrics and Cyclomatic Complexity feature, which is available specifically for the **Viewer role**. This feature provides valuable insights into the code being worked on, helping Viewers better understand the complexity, number of lines, and variables in the code. Currently, this is implemented for **Java and Python**, but in the future, I plan to extend it to other languages.

Code Metrics Details

The **calculateMetrics** method computes key details such as:

- Number of lines: This simply counts the lines in the code.
- Number of functions: This counts the functions based on language-specific syntax (e.g., public, private for Java and def for Python).
- Number of variables: Variables are identified using typical keywords (e.g., int, String, =).
- Cyclomatic Complexity: A measure of the code's complexity, based on the control flow structures such as if, for, while, etc.



Cyclomatic Complexity Calculation

For each supported language, specific control flow keywords are analyzed to calculate the cyclomatic complexity. For example:

- Java: Keywords like if, else, for, while, switch, catch are counted.
- Python: Keywords like if, elif, else, for, while, try, except are used.

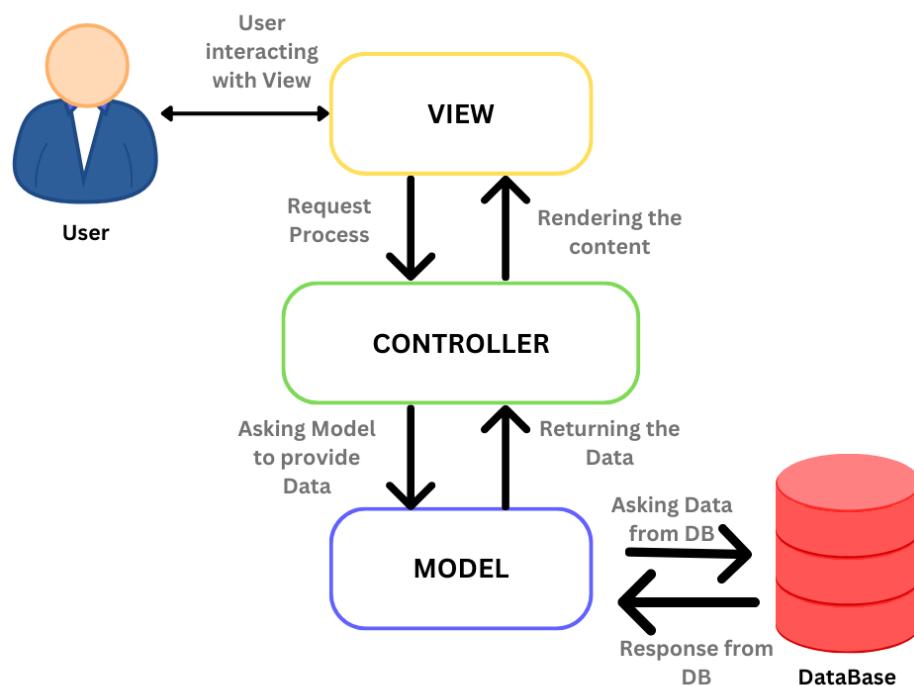
The complexity starts at 1 and increases with every conditional or branching structure found in the code.

Design Patterns

I used various design patterns in my code to structure and organize it effectively. The primary design pattern is Spring MVC, which provides a clean separation between the controllers (responsible for handling HTTP requests), services (business logic), and repositories (data persistence). Let me break down the design patterns I have implemented:

4.1 Spring MVC Pattern

I follow the Model-View-Controller (MVC) pattern using Spring Boot as the framework.



- **Controller Layer:**

I have structured my controllers in a way that each one handles a specific aspect of the application. Here's how they are organized:

1. SignInController:

- Handles user sign-in requests.
- It uses `@PostMapping` for login and processes the `LoginRequest` payload.
- It interacts with the `AuthenticationService` to validate user credentials and generate tokens.
- Exception handling is done inside the method itself to provide meaningful HTTP responses (`UNAUTHORIZED`, `NOT_FOUND`, etc.).

4.1 Spring MVC Pattern



```
@RestController  
@CrossOrigin  
@RequestMapping("/api")  
public class SignInController {  
    // Handles sign-in logic and returns JWT tokens or error messages  
}
```

2. SignUpController:

- Handles user sign-up requests.
- When a user signs up, it uses UserService to create a new user in the database.
- It also includes exception handling to return proper responses based on the error.



```
@RestController  
@CrossOrigin  
@RequestMapping("/api")  
public class SignUpController {  
    // Handles sign-up and user registration logic  
}
```

3. ExecutionController:

- This controller is responsible for handling code execution requests.
- It calls DockerExecutorService to run code inside Docker containers and returns the output back to the frontend.

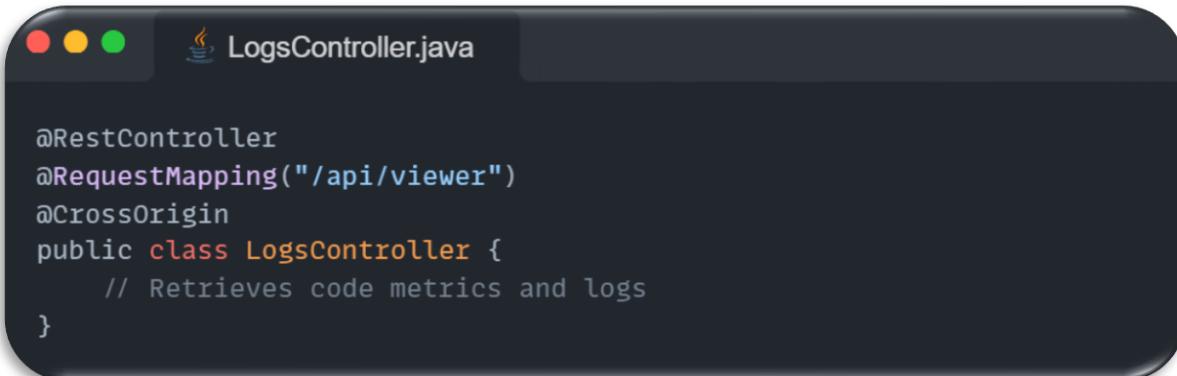


```
@RestController  
@CrossOrigin  
@RequestMapping("/api/execute")  
public class ExecutionController {  
    // Handles code execution requests  
}
```

4.1 Spring MVC Pattern

4. LogsController:

- Manages the retrieval of code metrics and logs.
- The EditorService is responsible for calculating code metrics like lines of code, complexity, etc.

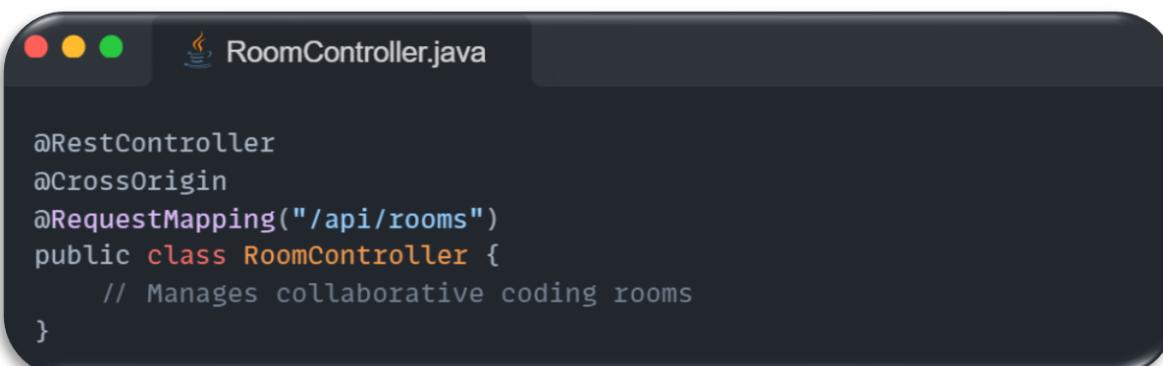


The screenshot shows a dark-themed code editor window titled "LogsController.java". The code defines a REST controller with annotations: `@RestController`, `@RequestMapping("/api/viewer")`, and `@CrossOrigin`. The class is named `LogsController` and contains a single method with a comment indicating it retrieves code metrics and logs.

```
@RestController  
@RequestMapping("/api/viewer")  
@CrossOrigin  
public class LogsController {  
    // Retrieves code metrics and logs  
}
```

5. RoomController:

- Handles room management in collaborative coding.
- It interacts with RoomService, ProjectService, and UserService to manage room membership, project access, and room security.

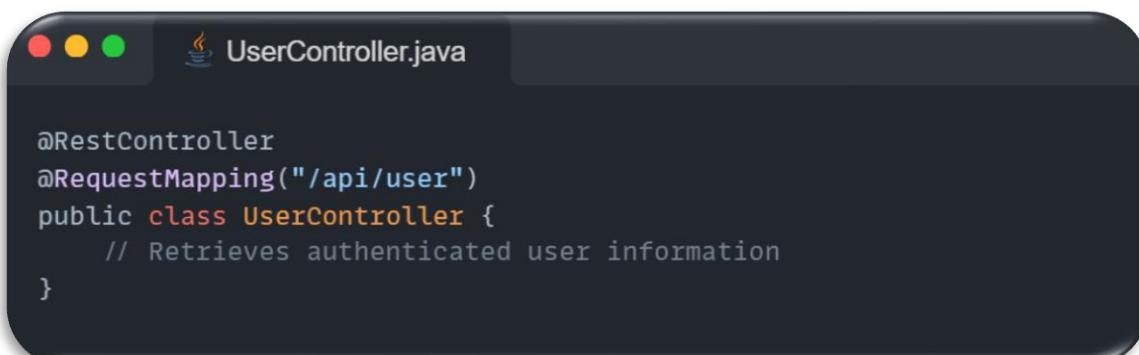


The screenshot shows a dark-themed code editor window titled "RoomController.java". The code defines a REST controller with annotations: `@RestController`, `@CrossOrigin`, and `@RequestMapping("/api/rooms")`. The class is named `RoomController` and contains a single method with a comment indicating it manages collaborative coding rooms.

```
@RestController  
@CrossOrigin  
@RequestMapping("/api/rooms")  
public class RoomController {  
    // Manages collaborative coding rooms  
}
```

6. UserController:

- Retrieves authenticated user information and builds a response with their profile information, such as email, name, and profile picture.
- It uses Spring Security to authenticate the user and fetch their details from the SecurityContext.



The screenshot shows a dark-themed code editor window titled "UserController.java". The code defines a REST controller with annotations: `@RestController`, `@RequestMapping("/api/user")`. The class is named `UserController` and contains a single method with a comment indicating it retrieves authenticated user information.

```
@RestController  
@RequestMapping("/api/user")  
public class UserController {  
    // Retrieves authenticated user information  
}
```

4.1 Spring MVC Pattern

7. VersionController

A) ProjectController:

- Handles project-related requests like creating, retrieving, and updating project information.

```
@RestController  
@RequestMapping("/api/projects")  
@CrossOrigin  
public class ProjectController {  
    // Manages project creation and updates  
}
```

B) FileController:

- Manages file operations, such as pulling and pushing files, and handles version control for files.
- It ensures that only users with the appropriate roles (e.g., collaborators) can modify files.

```
@RestController  
@RequestMapping("/api/files")  
@CrossOrigin  
public class FileController {  
    // Manages file operations and versioning  
}
```

C) WebsocketController:

- Manages WebSocket events for real-time collaboration.
- This includes sending and receiving code updates, handling locks on code lines, and communicating changes in real-time to all connected clients.

```
@Controller  
public class WebsocketController {  
    // Handles real-time WebSocket communication  
}
```

4.1 Spring MVC Pattern

- **Models:**

1. **CodeUpdate:**

- This is a MongoDB document that tracks changes in the code for a specific file in a project and room.
- It includes details like the user ID, filename, room ID, project name, line number, and content of the code being updated.

2. **Room:**

- This entity represents a coding room where multiple users collaborate on different files. It handles membership and project access.

3. **User:**

- This entity represents a user in the application and includes fields for the user's profile and credentials. It is managed by UserService for user-related operations.

4. **RoomMembership:**

- This entity tracks user membership in rooms and their roles (e.g., viewer, collaborator).

5. **File:**

- This model stores file information, including its versions, in a MongoDB collection. It helps in managing file content during collaboration.

6. **Project:**

- This entity represents a project in the MySQL database and is linked to rooms and users. It uses JPA annotations for ORM and database operations.

The image shows two dark-themed code editors side-by-side. The left editor is titled 'Room.java' and contains the following code:

```
@Getter  
@Setter  
@Entity  
@Table(name = "rooms")  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class Room {  
    // Represents a room for collaboration  
}
```

The right editor is titled 'File.java' and contains the following code:

```
@Document(collection = "file_versions")  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@CompoundIndex(def = "{filename: 1, 'projectName': 1}")  
public class File {  
    // Represents file versions and metadata  
}
```

- **View:**

- The view part of the Spring MVC pattern in my application is handled by React on the frontend. I use CSS for styling and JavaScript to manage user interactions. The backend controllers are responsible for sending and receiving data via REST APIs, which the frontend consumes.

4.2 Builder Pattern

Overview of Builder Pattern Usage

The Builder design pattern is applied across your entities using **Lombok's @Builder** annotation, which simplifies object creation by eliminating the need for complex constructors and making code more readable and maintainable.

Key Entities and Their Builder Applications:

The Builder design pattern is used across various entities in my project, leveraging **Lombok's @Builder** annotation to simplify object creation. This pattern improves readability and flexibility by allowing method chaining and reducing the need for complex constructors, especially for **entities like CodeUpdate, File, Project, Room, and User**. It also enhances immutability and scalability by making it easy to manage complex fields and optional parameters across **different request/response classes like LoginRequest, FileDTO, ProjectDTO, and others**.

```
11  @Document(collection = "file_versions")
12  @Data
13  @NoArgsConstructor
14  @AllArgsConstructor
15  @Builder
16  @Compound
17  public class Room {
18      ...
19  }
20  private final String roomName;
21  private final String roomId;
22  private final List<RoomMembership> roomMemberships;
23  private final Set<Project> projects;
```

```
11  @Override
12  @Transactional
13  public String createRoom(User owner, String roomName) {
14      try {
15          String roomId = UUID.randomUUID().toString();
16          Room room = Room.builder()
17              .name(roomName)
18              .roomId(roomId)
19              .roomMemberships(new ArrayList<>())
20              .projects(new HashSet<>())
21              .build();
22          ...
23      } catch (Exception e) {
24          throw new RoomCreationException("Failed to create room: " + e.getMessage());
25      }
26      return room.getRoomId();
27  }
```

```
11  private void createDefaultFile(Room room) {
12      try {
13          fileService.createFile(
14              FileDTO
15                  .builder()
16                  .filename("README")
17                  .projectName("Main-Branch")
18                  .roomId(room.getRoomId())
19                  .extension(".md")
20                  .build()
21                  ...
22          );
23      } catch (Exception e) {
24          throw new FileCreationException("Failed to create default file: " + e.getMessage());
25      }
26  }
```

4.3 DTO design pattern

DTO (Data Transfer Object) Overview

The DTO (Data Transfer Object) design pattern is used to **encapsulate data** and transfer it between different layers of an application, particularly between the client and server. **This pattern simplifies data management** by providing specific classes for different data types, allowing for efficient communication without exposing internal entities. **It helps reduce the amount of data sent over the network** and establishes a clear contract for the data structures exchanged, enhancing maintainability and flexibility in the application.

Class Name	Purpose
CodeUpdate	Stores information about a code update.
MessageLog	Logs messages for each action in a room.
LoginRequest & LoginResponse	Represents login request and response data structures.
CodeDTO	Represents data transfer object for code.
CodeExecution	Represents data for code execution requests.
CodeMetrics	Represents performance data of the code.
CodeMetricsRequest	Represents request structure for code metrics.
FileDTO	Represents file-related data transfer objects.
ProjectDTO	Represents project-related data transfer objects.
RoomDTO	Represents room-related data transfer objects.
CreateRoomRequest & AddMemberRequest	Represents requests for room creation and adding members.

```
Dto.java
```

```
public class ProjectDTO {
    private String name;
    private String description;
}
```

```
Dto.java
```

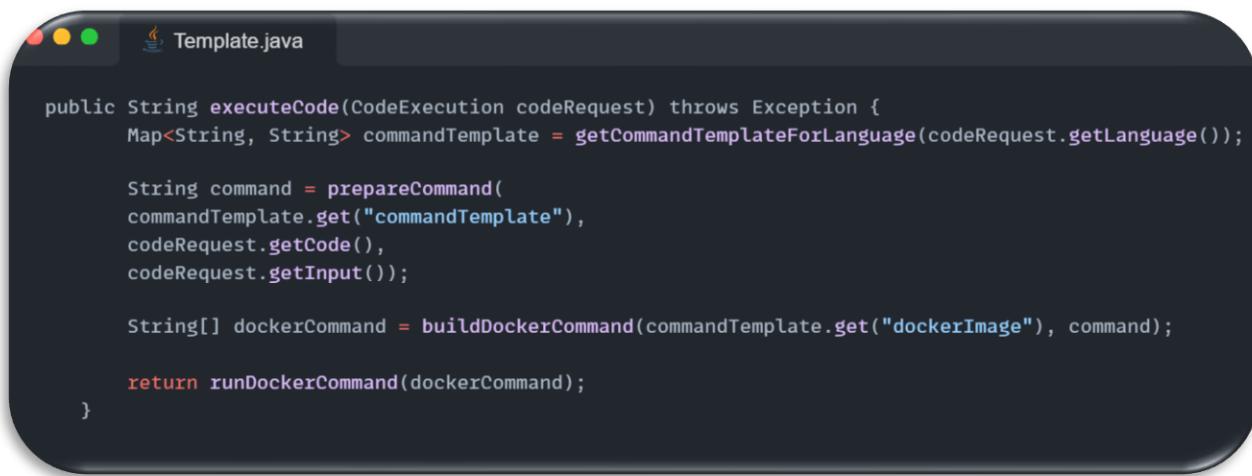
```
public class CreateRoomRequest {
    private String memberName;
    private String roomName;
}
```

```
Dto.java
```

```
public class CodeExecution {
    private String language;
    private String code;
    private String input;
}
```

4.4 Template Method design pattern

The DockerExecutorService implements the Template Method design pattern for executing code in different programming languages using Docker containers.



```
Template.java

public String executeCode(CodeExecution codeRequest) throws Exception {
    Map<String, String> commandTemplate = getCommandTemplateForLanguage(codeRequest.getLanguage());

    String command = prepareCommand(
        commandTemplate.get("commandTemplate"),
        codeRequest.getCode(),
        codeRequest.getInput());

    String[] dockerCommand = buildDockerCommand(commandTemplate.get("dockerImage"), command);

    return runDockerCommand(dockerCommand);
}
```

Key methods:

- executeCode
- getCommandTemplateForLanguage
- prepareCommand
- buildDockerCommand
- runDockerCommand

Note

By following this template, you will be allowed to add new languages commands in the future.

4.5 Strategy design pattern

In this case, the strategy for executing code differs based on the language (e.g., Python, Java, C++). Each language has a unique command template that defines how the code should be executed inside a Docker container. The DockerExecutorService dynamically chooses the correct execution strategy (command template) at **runtime based on the programming language in the CodeExecution request**.

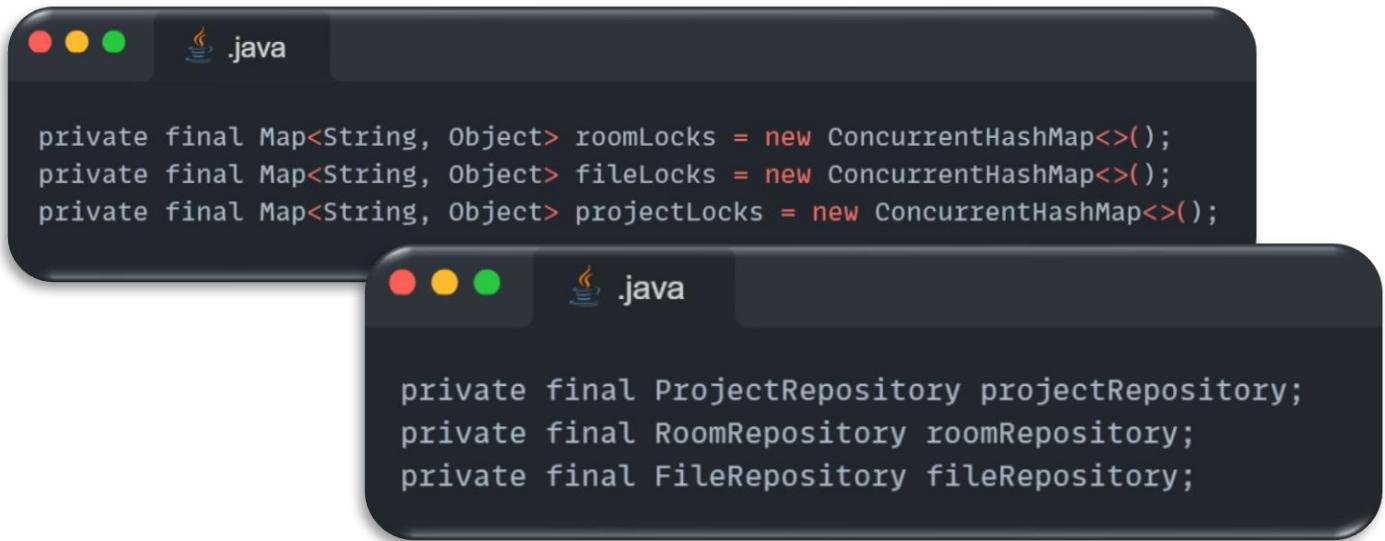
Key Methods Involved in the Strategy Pattern

- getCommandTemplateForLanguage(String language)
- prepareCommand(String commandTemplate, String code, String input)
- buildDockerCommand(String dockerImage, String command)

4.6 Singleton Pattern

The Singleton Pattern is applied in two areas:

- **Lock management:** Using globally shared maps (roomLocks, fileLocks, projectLocks) to manage locks.
- **Service dependency injection:** Spring ensures that services are instantiated as singletons, allowing shared access to service logic across the application.



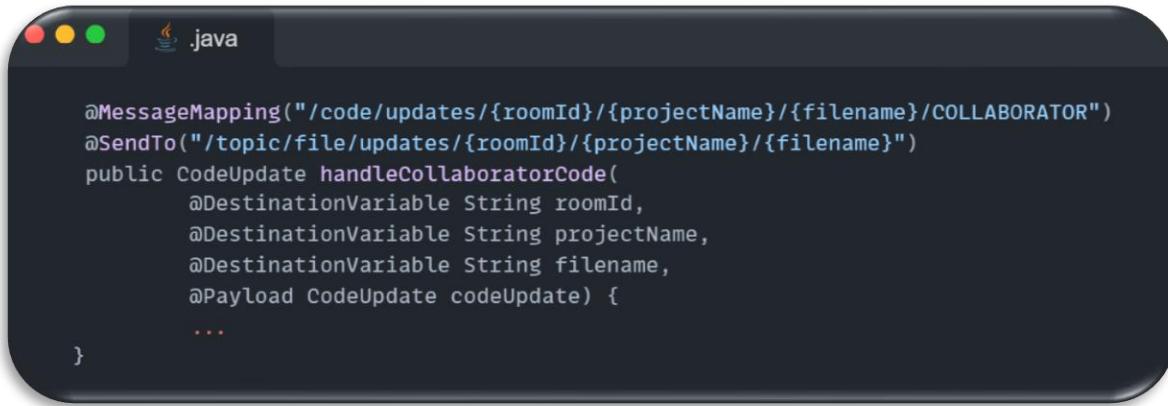
- This pattern helps ensure thread safety and consistency when managing resources such as files, rooms, and projects in a collaborative, multi-threaded

4.7 Observer Pattern

In the **WebSocket context**, the server acts as the **subject** that broadcasts updates to clients (the **observers**). The clients are notified whenever a certain event (such as a code update or chat message) occurs on the server.

- **Subjects (Producers):** The `@MessageMapping` annotated methods in `WebsocketController` represent the subjects (or event handlers) that produce updates, this method receives code updates for a file and processes the changes. It then sends the update to a specific WebSocket topic (`/topic/file/updates/...`), notifying all subscribed clients (the observers) about the code change.
- **Observers (Subscribers):** Clients connected to the WebSocket topic (`/topic/file/updates/{roomId}/{projectIdName}/{filename}`) act as observers. They automatically receive real-time updates from the server when changes are made to the code. Similarly, chat messages are broadcast to `/topic/chat/{roomId}`, where all connected clients (observers) receive the new message in real-time.

4.7 Observer Pattern



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

- **Purpose:** This method handles code updates from collaborators. After processing the update (saving it to the database using editorService.saveCodeUpdate(codeUpdate)), it broadcasts the update to all clients subscribed to the topic /topic/file/updates/....
- **Observer Role:** It notifies all relevant observers (clients connected to that file's WebSocket topic) about the update.

4.8 Global Exception Handling Design Pattern

In this design, the Global Exception Handling pattern is essentially a combination of the Chain of Responsibility and Strategy patterns:

- **Chain of Responsibility:** Different exception handlers are responsible for handling different types of exceptions, creating a chain where an exception will be passed to the next handler until it is handled.
- **Strategy:** Each @ExceptionHandler provides a different strategy for dealing with specific exceptions.



```
@RestControllerAdvice
public class GlobalExceptionHandler {

    private static final Logger logger = LoggerFactory.getLogger(GlobalExceptionHandler.class);

    @ExceptionHandler(UserNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public Map<String, String> handleUserNotFoundException(UserNotFoundException ex) {
        logger.warn("User not found: {}", ex.getMessage());
        return createErrorResponse("User Not Found", ex);
    }

    @ExceptionHandler(InvalidCredentialsException.class)
    @ResponseStatus(HttpStatus.UNAUTHORIZED)
    public Map<String, String> handleInvalidCredentialsException(InvalidCredentialsException ex) {
        logger.warn("Invalid credentials attempt: {}", ex.getMessage());
        return createErrorResponse("Invalid Credentials");
    }
}

public class RoomUpdateException extends RuntimeException {
    public RoomUpdateException(String message) {
        super(message);
    }
}

public class InvalidCredentialsException extends RuntimeException {
    public InvalidCredentialsException(String message) {
        super(message);
    }
}

public class FileAlreadyExistsException extends RuntimeException {
    public FileAlreadyExistsException(String message) {
        super(message);
    }
}
```

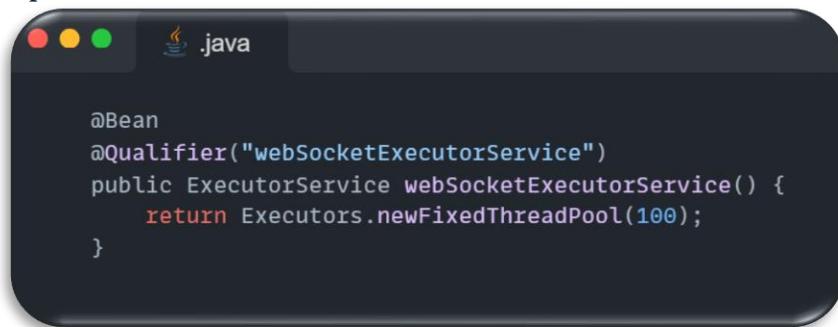
Multithreading

- **Multithreading in WebSocket Code Updates**

Multithreading is essential for handling simultaneous operations on the same file or line of code by multiple users. The **ExecutorService** is used to execute tasks in parallel, which allows multiple clients to collaborate in real time without blocking other operations.

Example:

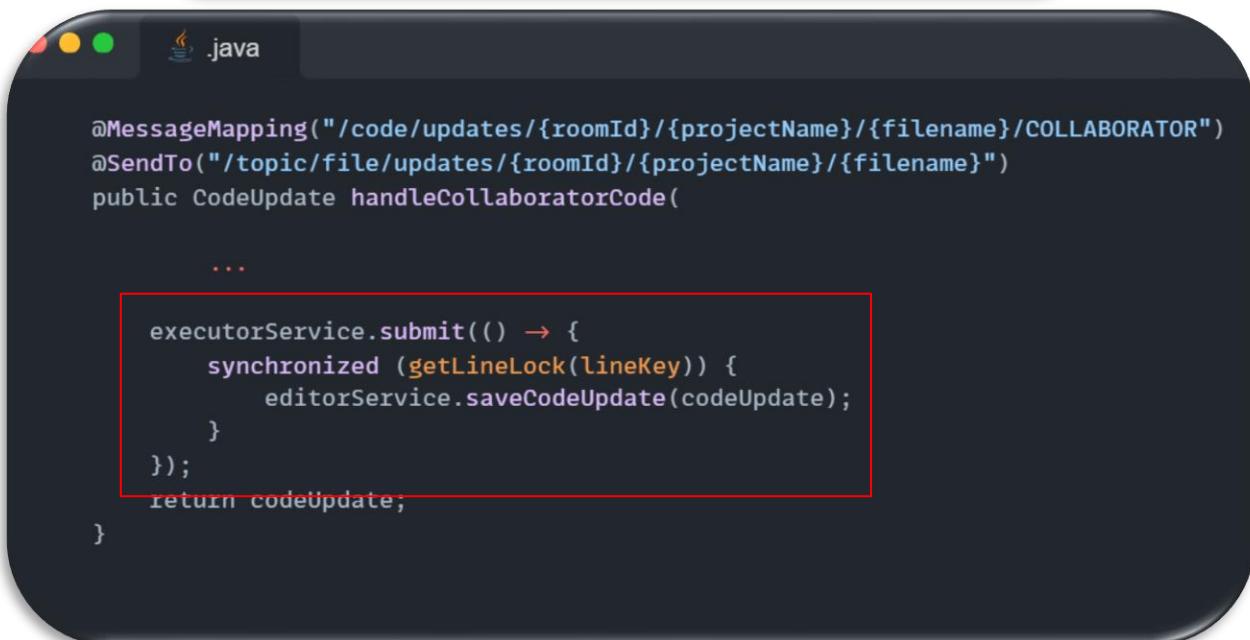
ExecutorService is initialized with a fixed thread pool to limit the number of concurrent threads handling WebSocket events. By using **Executors.newFixedThreadPool(100)**, we ensure that a maximum of 100 threads can run concurrently for handling incoming requests and updates.



```
java
@Bean
@Qualifier("websocketExecutorService")
public ExecutorService websocketExecutorService() {
    return Executors.newFixedThreadPool(100);
}
```



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



```
java
@MessageMapping("/code/updates/{roomId}/{ projectName}/{filename}/COLLABORATOR")
@SendTo("/topic/file/updates/{roomId}/{ projectName}/{filename}")
public CodeUpdate handleCollaboratorCode(
    ...
    executorService.submit(() -> {
        synchronized (getLineLock(lineKey)) {
            editorService.saveCodeUpdate(codeUpdate);
        }
    });
    return codeUpdate;
}
```

5.2 Thread Safety with Synchronization and Locks

Since multiple users can edit the same file simultaneously, the risk of race conditions is mitigated by using locks. In cases where multiple users try to update the same line of code or the same file, PUSH the same code or managing the room, synchronized blocks and locks ensure that only one thread can modify a particular section of code at a time.

- 1. Line-level Locking:** When users update a specific line in the editor, a line-based lock is applied using a **unique key** that identifies the line (**roomId + projectName + filename + lineNumber**). The **ConcurrentHashMap** is used to store these locks, and the synchronized keyword ensures that only one thread can hold the lock for a specific line while performing operations on it.

The image displays two screenshots of a Java code editor window. Both screenshots show code snippets with syntax highlighting and a 'java' icon in the title bar.

Top Screenshot:

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

private Object getLineLock(String lineKey) {
    return lineLocks.computeIfAbsent(lineKey, k -> new Object());
}
```

Bottom Screenshot:

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

synchronized (getLineLock(lineKey)) {
    editorService.saveCodeUpdate(codeUpdate);
}
```

- 2. File-level Locking:** For file operations such as pushing or retrieving file content, a similar locking mechanism is used to ensure that two clients cannot modify the file content concurrently. Each file operation uses a unique lock based on the roomId, projectName, and filename.

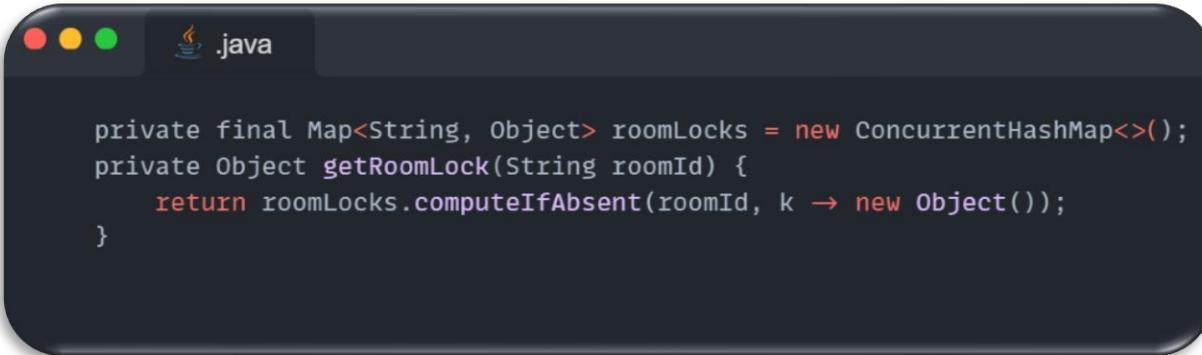
The image shows a screenshot of a Java code editor window with a 'java' icon in the title bar.

```
String fileKey = file.getRoomId() + "-" +
                file.getProjectName() + "-" +
                file.getFilename();

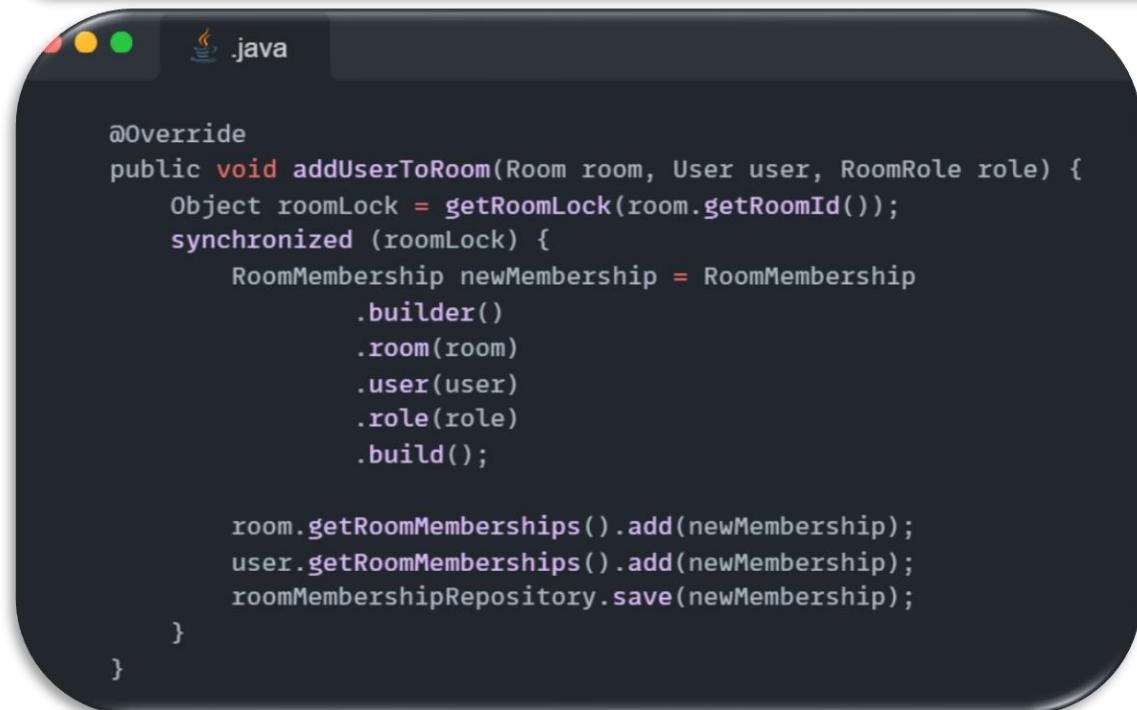
synchronized (getFileLock(fileKey)) {
    fileVersionRepository.upsertFileContent( ... );
}
```

5.2 Thread Safety with Synchronization and Locks

3. **Room-level Locking:** For room operations such as adding or removing member in the room, a similar locking mechanism is used to ensure that two clients cannot modify the file content concurrently. Each file operation uses a unique lock based on the roomId only.



```
private final Map<String, Object> roomLocks = new ConcurrentHashMap<>();
private Object getRoomLock(String roomId) {
    return roomLocks.computeIfAbsent(roomId, k -> new Object());
}
```



```
@Override
public void addUserToRoom(Room room, User user, RoomRole role) {
    Object roomLock = getRoomLock(room.getRoomId());
    synchronized (roomLock) {
        RoomMembership newMembership = RoomMembership
            .builder()
            .room(room)
            .user(user)
            .role(role)
            .build();

        room.getRoomMemberships().add(newMembership);
        user.getRoomMemberships().add(newMembership);
        roomMembershipRepository.save(newMembership);
    }
}
```

4. **Pessimistic Locking:** For database operations, particularly when updating file content, pessimistic locking is used to prevent conflicts. The `@Lock(LockModeType.PESSIMISTIC_WRITE)` annotation ensures that once a thread has locked a record, other threads must wait until the current operation is



```
@Lock(LockModeType.PESSIMISTIC_WRITE)
@Modifying
@Update(...)
void upsertFileContent(...);
```

5.3 Optimistic Locking for Operations

By combining optimistic and pessimistic locking strategies, the system ensures that both database-level and in-memory data remain consistent, even in a concurrent environment, this overall strategy ensures that file content updates, code updates, and room modifications are safely handled in a multithreaded environment, preventing race conditions and ensuring data consistency.

The figure consists of three separate windows, each representing a Java code editor with a dark theme. Each window has a title bar with a red, yellow, and green button on the left and a small icon on the right. The first window shows a single line of code: `@Version private Long version;`. The second window shows a method implementation with several annotations highlighted by red boxes: `@Override @Transactional`, `String fileKey = file.getRoomId() + "-" + file.getProjectName() + "-" + file.getFilename();`, `synchronized (getFileLock(fileKey)) {`, `try {`, `} else {`, `throw new RuntimeException("File not found for pushing content.");`, `} catch (OptimisticLockException e) {`, and `throw new RuntimeException("Failed to push content due to concurrent modification.", e);`. The third window shows another method implementation with annotations highlighted by red boxes: `@Override @Transactional(readOnly = true)`, `public List<String> getViewers(Room room) {`, `return room.getRoomMemberships().stream()`, `.filter(rm → rm.getRole() == RoomRole.VIEWER)`, `.map(RoomMembership::getUser)`, `.map(User::getEmail)`, `.collect(Collectors.toList());`, and `}`.

In the context of room management, versioning is implemented using the `@Version` annotation in the Room entity. This approach ensures **optimistic locking**, where the application checks if the **version number** of an entity matches before committing a transaction. If another thread has modified the room in the meantime, an `OptimisticLockException` is thrown, preventing concurrent modifications.

SOLID Principles

I've applied SOLID principles to ensure that everything is structured in a way that's easy to maintain and extend. When I think about these principles, it's all about keeping each piece of code focused on one responsibility, making sure new features can be added without breaking the existing ones, and keeping things flexible for future changes. I've broken down logic across different classes, so responsibilities are separated, and interfaces ensure that

6.1 Authentication Defense (SOLID Principles)

Here's a breakdown of how SOLID Principles are applied in Authentication classes:

1. Single Responsibility Principle (SRP)

SRP Defense: Each class related to authentication has a single, well-defined responsibility.

- **SignInController:** Responsible for handling the user sign-in process.
- **SignUpController:** Responsible for handling the user sign-up process.
- **AuthenticationService:** Responsible for handling authentication-related logic (validating credentials, generating tokens).
- **JwtUtil:** Handles token generation and parsing.
- **PasswordValidator and EmailValidator:** Validates password and email formats independently, focusing only on validation logic.
- **UserService:** Manages user creation, retrieval, and related user logic.

The image shows three Java code editors side-by-side, each with a dark theme and syntax highlighting. The first editor contains the `AuthenticationServiceImpl` class, which implements the `AuthenticationService` interface. It includes methods for sign-in and sign-up, and injects `AuthenticationManager` and `JwtUtil`. The second editor contains the `EmailValidator` class, which contains static methods for validating email addresses based on length and character presence. The third editor contains the `PasswordValidator` class, which contains static methods for validating passwords against common patterns and character requirements.

```
@Service("AuthenticationServiceImpl")
public class AuthenticationServiceImpl implements AuthenticationService {

    private final AuthenticationManager authManager;
    private final JwtUtil jwtUtil;

    @Autowired
    public AuthenticationServiceImpl(AuthenticationManager authManager, JwtUtil jwtUtil) {
        this.authManager = authManager;
        this.jwtUtil = jwtUtil;
    }

    @Override
    public String signIn(String username, String password) {
        Authentication authentication = authManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));
        SecurityContextHolder.getContext().setAuthentication(authentication);
        return jwtUtil.generateToken(authentication);
    }

    private AuthenticationManager getAuthManager() {
        return authManager;
    }
}

public class EmailValidator {

    private static final int MIN_LENGTH = 6;

    public static boolean validateEmail(String email) {
        if (email.length() < MIN_LENGTH) {
            throw new IllegalArgumentException("Email must be at least " + MIN_LENGTH + " characters long");
        }
        return true;
    }
}

public class PasswordValidator {

    private static final int MIN_LENGTH = 8;
    private static final List<String> COMMON_PASSWORDS = Arrays.asList("password", "123456", "qwerty", "admin");

    public static void validatePassword(String password, String username, String email) {
        // contains check
    }

    private static boolean containsUpperCase(String password) {
        return password.chars().anyMatch(Character::isUpperCase);
    }

    private static boolean containsLowerCase(String password) {
        return password.chars().anyMatch(Character::isLowerCase);
    }

    private static boolean containsDigit(String password) {
        return password.chars().anyMatch(Character::isDigit);
    }

    private static boolean containsSpecialCharacter(String password) {
        return password.chars().anyMatch(ch -> !"!@#$%^&*()_+{}[]|;:<,.?/-=".indexOf(ch) >= 0);
    }
}
```

6.1 Authentication Defense (SOLID Principles)

2. Open/Closed Principle (OCP)

OCP Defense: Classes are open for extension but closed for modification.

- **SignUpController:** can add new authentication providers in the providerSignIn method by adding a new provider option without altering the logic.
- **AuthenticationService:** could be extended by creating new implementations (e.g., OAuth2, SAML and LDAP) without modifying existing logic.
- **UserServiceImpl:** Similarly, additional user creation or validation strategies can be implemented and injected without changing the core class.
- **JwtUtil:** can easily support different token generation strategies by extending or modifying methods without changing the core authentication service.

The image displays four overlapping Java code snippets, each showing a different part of the application's architecture related to authentication and user management.

- AuthenticationServiceImpl.java**:

```
@Service("AuthenticationServiceImpl")
public class AuthenticationServiceImpl implements AuthenticationService {

    private final AuthenticationManager authManager;
    private final JwtUtil jwtUtil;

    @Autowired
    public AuthenticationServiceImpl(@Lazy AuthenticationManager authManager, JwtUtil jwtUtil) {
        this.authManager = authManager;
        this.jwtUtil = jwtUtil;
    }

    @Override
    public String verify(LoginRequest loginRequest) {
        Authentication authentication =
            SecurityContextHolder.getContext();
        return jwtUtil.generateToken(authentication);
    }
}
```
- AuthenticationService.java**:

```
public interface AuthenticationService {
    String verify(LoginRequest loginRequest);
}
```
- SignUpController.java**:

```
@PostMapping("/sign-up")
public ResponseEntity<Map<String, String>> createAccount(@RequestBody User user) {
    try {
        userService.createAccount(user);
        return buildResponse("Account created successfully", HttpStatus.OK);
    } catch (Exception e) {
        return buildResponse("Registration failed, please try again", HttpStatus.BAD_REQUEST);
    }
}

@GetMapping("/sign-in/provider/{provider}")
public RedirectView providerSignIn(@PathVariable String provider) {
    return new RedirectView("/oauth2/authorization/" + provider);
}
```
- UserService.java**:

```
public interface UserService {
    void createUserAccount(User user);
    Optional<User> findUserByEmail(String email);
    User findUserByUsername(String username);
    User getUser(Authentication authentication);
}
```

6.1 Authentication Defense (SOLID Principles)

3. Liskov Substitution Principle (LSP)

LSP Defense: Subtypes (implementations of **AuthenticationService**, **UserService**, etc.) can replace their base types without altering the correctness of the program.

For instance, **AuthenticationServiceImpl** can be substituted by another implementation, such as **OAuth2AuthenticationServiceImpl**, and the system would still work without breaking the contract. The base interface ensures that new implementations of authentication services adhere to the required behavior.

```
java

public interface AuthenticationService {
    String verify(LoginRequest loginRequest);
}

// Implementation 1: Default JWT-based authentication
@Service("AuthenticationServiceImpl")
public class AuthenticationServiceImpl implements AuthenticationService {
    @Override
    public String verify(LoginRequest loginRequest) {
        // Default JWT authentication logic
    }
}

// Implementation 2: OAuth2-based authentication (can replace the default implementation)
@Service("OAuth2AuthenticationServiceImpl")
public class OAuth2AuthenticationServiceImpl implements AuthenticationService {
    @Override
    public String verify(LoginRequest loginRequest) {
        // OAuth2 authentication logic
    }
}
```

4. Interface Segregation Principle (ISP)

ISP Defense: Interfaces are kept small, specific, and focused.

- **AuthenticationService** focuses only on authentication logic.
- **UserService** focuses on user-related operations like creation, retrieval, etc.

```
java

public interface UserService {
    void createUser(User user);
    Optional<User> findUserByEmail(String email);
    User findUserByUsername(String username);
    User getUser(Authentication authentication);
}

java

public interface AuthenticationService {
    String verify(LoginRequest loginRequest);
}
```

6.1 Authentication Defense (SOLID Principles)

5. Dependency Inversion Principle (DIP)

DIP Defense: High-level modules (like AuthenticationServiceImpl) depend on abstractions (AuthenticationManager, JwtUtil) rather than concrete implementations. This makes the system more flexible and easier to refactor, as lower-level components can be swapped out without changing the high-level logic.

- **SignUpController:** the UserService is injected using the @Qualifier annotation, meaning the controller depends on an abstraction (UserService interface) rather than a concrete implementation (UserServiceImpl).
- **SignInController:** The SignInController depends on the AuthenticationService abstraction, not the concrete implementation. This allows for different authentication services to be injected.
- **UserServiceImpl:** Similarly, the UserServiceImpl depends on the UserRepository abstraction, allowing different repository implementations to be used without changing the core service logic.

The image shows three overlapping Java code editor windows, each with a dark theme and a title bar indicating it's a .java file. The code in each window illustrates the Dependency Inversion Principle (DIP) by showing how high-level modules depend on abstractions rather than concrete implementations.

- Top Window (AuthenticationServiceImpl.java):** Shows the implementation of the AuthenticationService interface. It uses @Service("AuthenticationService") and @Autowired annotations. It depends on abstractions: AuthenticationManager and JwtUtil. The constructor takes these abstractions and injects them into the class.
- Middle Window (SignUpController.java):** Shows the SignUpController. It uses @RestController, @CrossOrigin, and @RequestMapping("/api"). It depends on the UserService abstraction, which is injected via a @Qualifier("UserService") annotation.
- Bottom Window (UserController.java):** Shows the UserController. It uses @RestController and @RequestMapping("/api/user"). It depends on the UserServiceImpl abstraction, which is injected via a @Qualifier("UserService") annotation.

```
@Service("AuthenticationService")
public class AuthenticationService implements AuthenticationService {
    private final AuthenticationManager authManager; // Depends on abstraction
    private final JwtUtil jwtUtil; // Abstracts token management

    @Autowired
    public AuthenticationService(AuthenticationManager authManager, JwtUtil jwtUtil) {
        this.authManager = authManager; // Concrete implementation can be injected
        this.jwtUtil = jwtUtil;
    }
}

@Service("UserService")
public class UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserService(@Qualifier("UserRepository") UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @GetMapping("/users")
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }
}
```

```
@RestController
@CrossOrigin
@RequestMapping("/api")
public class SignUpController {

    private final UserService userService;

    public SignUpController(@Qualifier("UserService") UserService userService) {
        this.userService = userService;
    }

    @PostMapping("/sign-up")
    public ResponseEntity<User> signUp(@RequestBody User user) {
        User savedUser = userService.signUp(user);
        return ResponseEntity.ok(savedUser);
    }
}
```

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

    private final UserServiceImpl userService;

    @Autowired
    public UserController(@Qualifier("UserService") UserServiceImpl userService) {
        this.userService = userService;
    }

    @GetMapping("/user")
    public ResponseEntity<User> getUser() {
        User user = userService.getUser();
        return ResponseEntity.ok(user);
    }
}
```

6.2 Room Management Defense (SOLID Principles)

Here's a breakdown of how SOLID Principles are applied in Room Management classes:

1. Single Responsibility Principle (SRP)

SRP Defense: In the Room, File, and Project management classes, each class has a single responsibility.

- **RoomController:** handles HTTP requests related to rooms and delegates tasks to RoomService, UserService, and ProjectService.
- **RoomServiceImpl:** manages core business logic related to rooms, such as creation, deletion, and membership management.
- **FileServiceImpl:** focuses on managing files associated with projects, providing methods to fetch, push, and merge file content.
- **ProjectServiceImpl:** handles creating, deleting, and managing projects linked to a room.

The screenshot shows a Java IDE interface with two code files open in tabs:

- FileService.java**:

```
public interface FileService {  
    List<FileDTO> getFiles(ProjectDTO project);  
  
    void createFile(FileDTO fileDTO) throws FileAlreadyExistsException;  
  
    void pushFileContent(File file);  
  
    File mergeFileContent(File newVersion) throws FileAlreadyExistsException;  
  
    File pullFileContent(FileDTO fileDTO);  
}  
  
public interface ProjectService {  
  
    void createProject(ProjectDTO project) throws NoSuchFieldException;  
  
    List<ProjectDTO> getProjects(String roomId);  
  
    void deleteProject();  
  
    void downloadProj();  
}
```
- RoomController.java**:

```
@RestController  
@RequestMapping("/api/rooms")  
public class RoomController {  
    private final RoomService roomService;  
    private final UserService userService;  
    private final ProjectService projectService;  
  
    @Autowired  
    public RoomController(RoomService roomService, UserService userService, ProjectService projectService) {  
        this.roomService = roomService;  
        this.userService = userService;  
        this.projectService = projectService;  
    }  
  
    @PostMapping("/createRoom")  
    public ResponseEntity<Map<String, String>> createRoom(@RequestBody CreateRoomRequest request) {  
        // LOGIC HERE  
    }  
}
```

6.2 Room Management Defense (SOLID Principles)

2. Open/Closed Principle (OCP)

OCP Defense: In cases where new features need to be added, the existing structure can accommodate changes without altering existing code.

- **RoomService interface** can be extended by adding new methods, and any new services (such as notifications, or logging) can be implemented in separate classes.
- **The FileServiceImpl** allows for file versioning, and new functionalities like file encryption can be added by introducing a new **FileEncryptionHandler** without modifying the original service.
- Similarly, for FileService and ProjectService.

The image shows two code editor windows. The left window displays the `FileServiceImpl.java` file, which implements the `FileService` interface. It contains several overridden methods with annotations like `@Override` and `@Transactional`. The right window displays the `RoomService.java` file, which defines an interface with methods for creating, getting, deleting, and renaming rooms.

```
FileServiceImpl.java
@Service("FileServiceImpl")
public class FileServiceImpl implements FileService {

    private static final String DEFAULT_CONTENT_FILE = "Typing your first code here...";
    private final FileRepository fileVersionRepository;
    private final FileMergeHandler fileMergeHandler;
    private final CodeUpdateRepository codeUpdateRepository;

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

    @Override
    public List<FileDTO> getFiles(ProjectDTO project) {
    }

    @Override
    @Transactional
    public void createFile(FileDTO fileDTO) throws FileAlreadyExistsException {
    }

    @Override
    @Transactional
    public void pushFileContent(File file) {
    }

    @Override
    public File mergeFileContent(File newVersion) {
    }

    @Override
    public File pullFileContent(File oldVersion) {
    }
}

RoomService.java
public interface RoomService {
    String createRoom(User owner, String roomName);
    Optional<Room> getRoomById(String roomId);
    void deleteRoom(String roomId);
    void rename(Room room);
}
```

6.2 Room Management Defense (SOLID Principles)

3. Liskov Substitution Principle (LSP)

LSP Defense is followed by ensuring that subclasses or implementations of interfaces can be substituted for their parent classes without affecting the functionality.

For instance, The **RoomServiceImpl** and **FileServiceImpl**, **ProjectServiceImpl** classes both adhere to their respective service interfaces, ensuring that any class implementing RoomService or FileService can be substituted without breaking the system.



A screenshot of a Java code editor window titled "RoomServiceImpl.java". The code defines a class that implements the RoomService interface. It includes methods for creating rooms, saving them to a repository, adding users to rooms, and creating default projects and files. The code uses annotations like @Service and @Override, and imports from packages such as java.util.UUID, Room, RoomRepository, User, RoomRole, and ProjectDTO.

```
@Service("RoomServiceImpl")
public class RoomServiceImpl implements RoomService {
    @Override
    public String createRoom(User owner, String roomName) {
        String roomId = UUID.randomUUID().toString();
        Room room = Room.builder()
            .name(roomName)
            .roomId(roomId)
            .roomMemberships(new ArrayList<>())
            .projects(new HashSet<>())
            .build();

        roomRepository.save(room);
        addUserToRoom(room, owner, RoomRole.OWNER);

        createDefaultProject(room);
        createDefaultFile(room);
    }
}
```

4. Interface Segregation Principle (ISP)

ISP Defense: is applied by ensuring that clients are not forced to depend on interfaces they do not use. This is seen in how each service interface (RoomService, FileService, ProjectService) is designed to include only methods related to its specific functionality.

- **FileService** includes methods related to file management (pushFileContent, pullFileContent).
- **RoomService** focuses on room operations like creation, deletion, and membership management.



A screenshot of a Java code editor window titled "FileService.java". It shows the definition of a public interface named FileService with three methods: getFiles, pushFileContent, and mergeFileContent. The interface also includes a closing brace at the bottom.

```
public interface FileService {
    List<FileDTO> getFiles(ProjectDTO project);
    void pushFileContent(File file);
    void mergeFileContent(File file);
}
```

6.2 Room Management Defense (SOLID Principles)

5. Dependency Inversion Principle (DIP)

DIP Defense: in the RoomController, FileController and ProjectController, services are injected via their interfaces, not concrete classes (I used different Dependency Inversion methods field injection and Constructor injections)

- By injecting interfaces rather than implementations, the controllers can easily work with different service implementations without being tightly coupled to them.

The image shows two Java code editors side-by-side. The top editor contains the code for `RoomServiceImpl`, and the bottom editor contains the code for `RoomController`. Both files have a `.java` extension and a coffee cup icon in the title bar.

```
@Service("RoomServiceImpl")
public class RoomServiceImpl implements RoomService {

    @Autowired
    private RoomRepository roomRepository;

    @Autowired
    private RoomMembershipRepository roomMembershipRepository;

    @Autowired
    private ProjectService projectService;

    @Autowired
    private FileService fileService;
```



```
@RequestMapping("/api/rooms")
public class RoomController {

    private final RoomService roomService;
    private final UserService userService;
    private final ProjectService projectService;
    private final RoomSecurityService roomSecurityService;

    @Autowired
    public RoomController(
            @Qualifier("RoomServiceImpl") RoomService roomService,
            @Qualifier("UserServiceImpl") UserService userService,
            @Qualifier("ProjectServiceImpl") ProjectService projectService,

            this.roomService = roomService;
            this.userService = userService;
            this.projectService = projectService;
            this.roomSecurityService = roomSecurityService;
    }
```

6.3 Editor Defense (SOLID Principles)

Here's a breakdown of how SOLID Principles are applied in Editor classes:

1. Single Responsibility Principle (SRP)

SRP Defense: In the Editor environment each class has a single responsibility.

- **WebsocketController:** Handles WebSocket communication for code updates and chat messages.
- **LogsController:** Manages log retrieval for rooms, users, projects, and files.
- **ExecutionController:** Executes code in Docker containers.
- **EditorService:** Responsible for business logic related to code editing, saving updates, and calculating metrics.
- **DockerExecutorService:** Executes Docker commands and runs code within isolated containers.



The screenshot shows two Java code files in a code editor:

```
ExecutionController.java
```

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

    private final DockerExecutorService dockerService;

    public ExecutionController(DockerExecutorService dockerService) {
        this.dockerService = dockerService;
    }

    @PostMapping("/run")
    public ResponseEntity<Map<String, String>> runCode(@RequestBody CodeExecution codeRequest) {
        try {
            String output = dockerService.executeCode(codeRequest);
            return ResponseEntity.ok(Map.of("output", output));
        } catch (Exception e) {
            return ResponseEntity.unprocessableEntity().build();
        }
    }
}
```

```
EditorController.java
```

```
@RestController
@RequestMapping("/api/viewer")
@CrossOrigin
public class EditorController {

    private final EditorService editorService;

    public EditorController(@Qualifier("EditorServiceImpl") EditorService editorService) {
        this.editorService = editorService;
    }

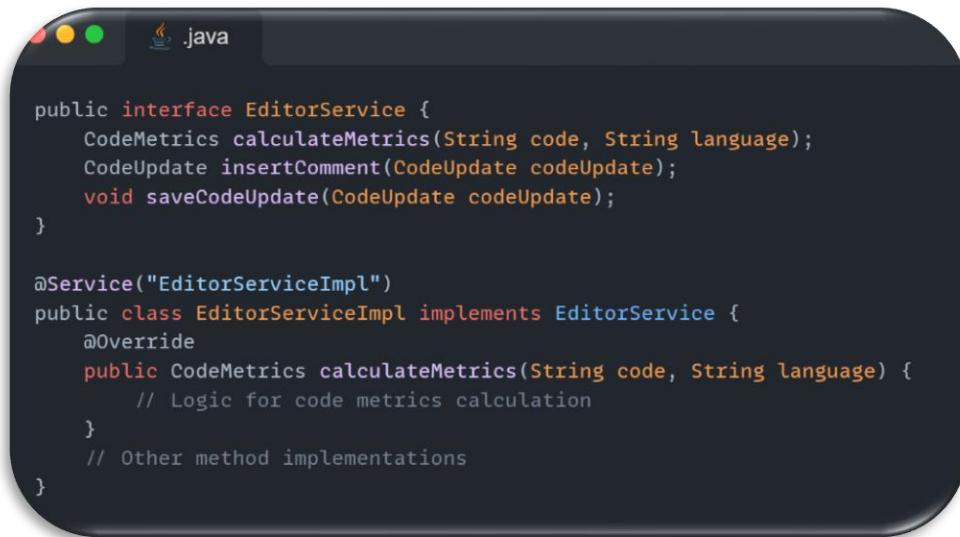
    @PostMapping("/CodeMetrics")
    public ResponseEntity<Map<String, CodeMetrics>> getCodeMetrics(@RequestBody CodeMetricsRequest request) {
        try {
            CodeMetrics metrics = editorService.calculateMetrics(request.getCode(), request.getLanguage());
            return ResponseEntity.ok(Map.of("metrics", metrics));
        } catch (Exception e) {
            return ResponseEntity.unprocessableEntity().build();
        }
    }
}
```

6.3 Editor Defense (SOLID Principles)

2. Open/Closed Principle (OCP)

OCP Defense: By using interfaces and abstractions, we ensure that we can extend the system without modifying existing code.

- **EditorService:** This interface allows for different implementations (EditorServiceImpl). New behaviors can be added without changing the existing service logic.
- **DockerExecutorService:** The JSON-based command template allows the addition of new languages or commands without altering the core class.



```
java

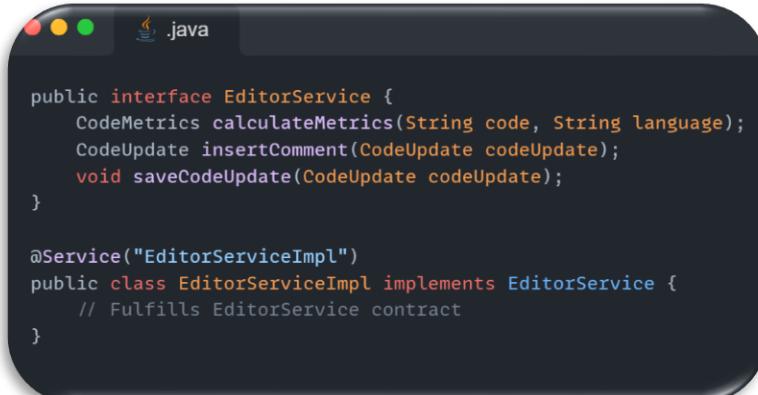
public interface EditorService {
    CodeMetrics calculateMetrics(String code, String language);
    CodeUpdate insertComment(CodeUpdate codeUpdate);
    void saveCodeUpdate(CodeUpdate codeUpdate);
}

@Service("EditorServiceImpl")
public class EditorServiceImpl implements EditorService {
    @Override
    public CodeMetrics calculateMetrics(String code, String language) {
        // Logic for code metrics calculation
    }
    // Other method implementations
}
```

3. Liskov Substitution Principle (LSP)

LSP Defense is followed by ensuring that subclasses or implementations of interfaces can be substituted for their parent classes without affecting the functionality.

- **EditorServiceImpl:** Fulfils the EditorService contract. Any other class implementing this interface can be substituted without breaking functionality.
- **DockerExecutorService:** Can be replaced by another execution service as long as it adheres to the method signatures and behavior defined by its usage.



```
java

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

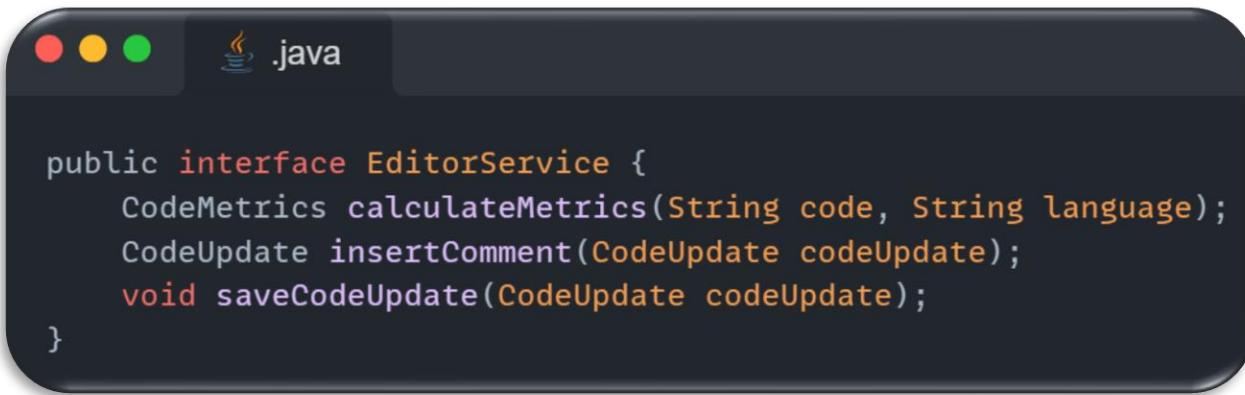
@Service("EditorServiceImpl")
public class EditorServiceImpl implements EditorService {
    // Fulfils EditorService contract
}
```

6.3 Editor Defense (SOLID Principles)

4. Interface Segregation Principle (ISP)

ISP Defense: The classes depend only on the methods they need. Large interfaces are broken into smaller, specific ones.

- **EditorService:** Contains only methods related to code updates and metrics, without forcing the implementation of unnecessary methods.
- **Other controllers** (LogsController, ExecutionController, WebsocketController) focus only on their responsibilities, avoiding coupling with irrelevant methods.



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

5. Dependency Inversion Principle (DIP)

DIP Defense: High-level modules should not depend on low-level modules but on abstractions. This is achieved through dependency injection and the use of interfaces. (I used different Dependency Inversion methods field injection and Constructor injections)

- **Controllers (WebsocketController, LogsController, ExecutionController)** depend on services via their interfaces, not concrete implementations, allowing for flexibility and testability.



```
@RestController  
public class WebsocketController {  
    private final EditorService editorService;  
    private final LogsService logService;  
  
    public WebsocketController(@Qualifier("EditorServiceImpl") EditorService editorService,  
                               @Qualifier("LogsServiceImpl") LogsService logService) {  
        this.editorService = editorService;  
        this.logService = logService;  
    }  
}
```

Clean Code

My approach to software development is based on the idea that code should be easy to read, understand, and maintain. I believe that this can be achieved by following a set of principles and practices, such as:

7.1 Meaningful Names

❖ [Principle] Use Intention-Revealing Names:

- **Defend:** Method names like *createProject*, *listProjects*, *deleteProject*, and *downloadProject* in *ProjectController* clearly describe their purposes.

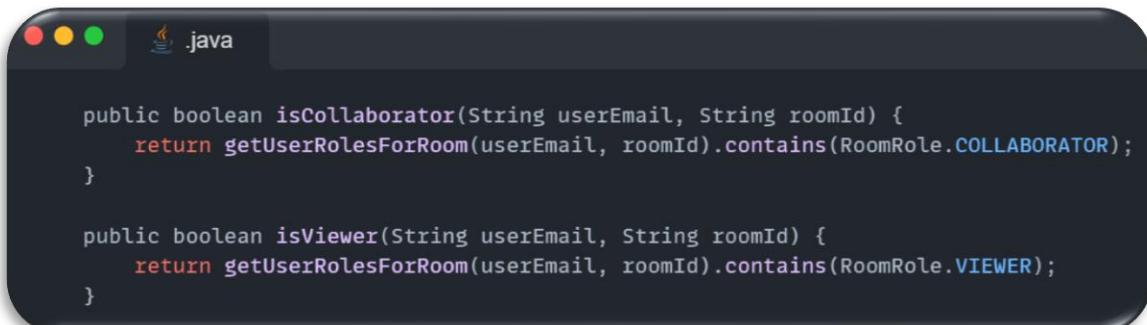
Example:



```
@PostMapping("/create-project")
public ResponseEntity<String> createProject(@RequestBody ProjectDTO request) {
    try {
        projectService.createProject(request);
        return ResponseEntity.ok("Project created successfully!");
    } catch (Exception e) {
        return ResponseEntity.badRequest().body("Failed to create project.");
    }
}
```

- **Defend:** Method names like *getUserRolesForRoom*, *isOwner*, *isCollaborator*, and *isViewer* in *RoomMembershipService* clearly describe their purposes.

Example:



```
public boolean isCollaborator(String userEmail, String roomId) {
    return getUserRolesForRoom(userEmail, roomId).contains(RoomRole.COLLABORATOR);
}

public boolean isViewer(String userEmail, String roomId) {
    return getUserRolesForRoom(userEmail, roomId).contains(RoomRole.VIEWER);
}
```

7.1 Meaningful Names

- **Defend:** Method names like `handleCollaboratorCode`, `handleChatMessage`, and `handleViewerComment` in `WebsocketController` clearly describe their purposes.

Example:



```
@Controller
public class WebsocketController {
    private final ExecutorService executorService;
    private final EditorService editorService;
    private final LogService logService;
    private final Map<String, Object> lineLocks = new ConcurrentHashMap<>();
    private final RoomSecurityService roomSecurityService;

    @MessageMapping("/chat/{roomId}")
    @SendTo("/topic/chat/{roomId}")
    public MessageLog handleChatMessage(@DestinationVariable("roomId") String roomId,
                                         @Payload MessageLog messageLog) {
    }
}
```

- **Defend:** `AuthenticationServiceImpl` contains a method `verify`, which reflects its role in verifying login requests.

Example:



```
@Override
public String verify(LoginRequest loginRequest) {
    Authentication authentication = authenticateUser(loginRequest);
    SecurityContextHolder.getContext().setAuthentication(authentication);
    return jwtUtil.generateToken(authentication);
}
```

- **Defend:** The `RoomService` interface uses meaningful names for its methods, such as `createRoom`, `getRoomById`, `deleteRoom`, `addUserToRoom`, and `getCollaborators`.

Example:



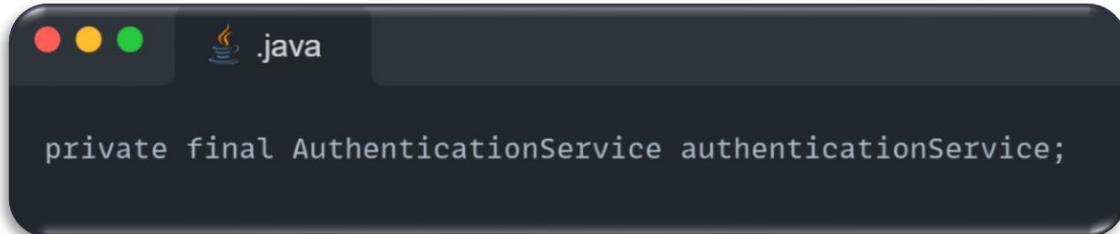
```
public interface RoomService {
    String createRoom(User owner, String roomName);
    Optional<Room> getRoomById(String roomId);
    void deleteRoom(String roomId);
    List<String> getCollaborators(Room room);
}
```

7.1 Meaningful Names

❖ [Principle] Pronounceable and Searchable Names:

- **Defend:** Variable names like token, email, and authenticationService are pronounceable and easy to search for, improving readability and maintainability.

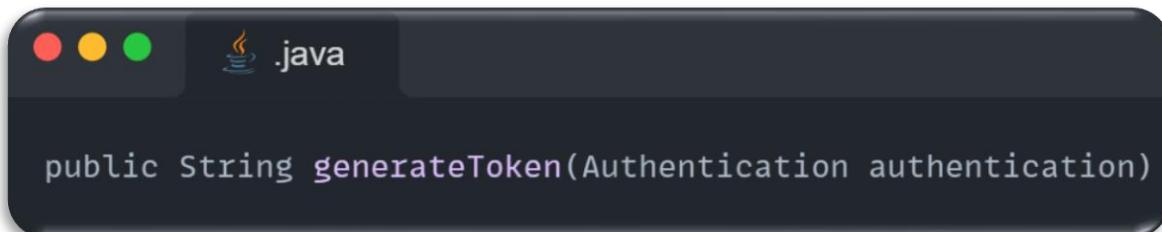
Example:



```
private final AuthenticationService authenticationService;
```

- **Defend:** Names like generateToken, validateToken, getEmail, and getUsernameFromToken in the JwtUtil class clearly express what the method does.

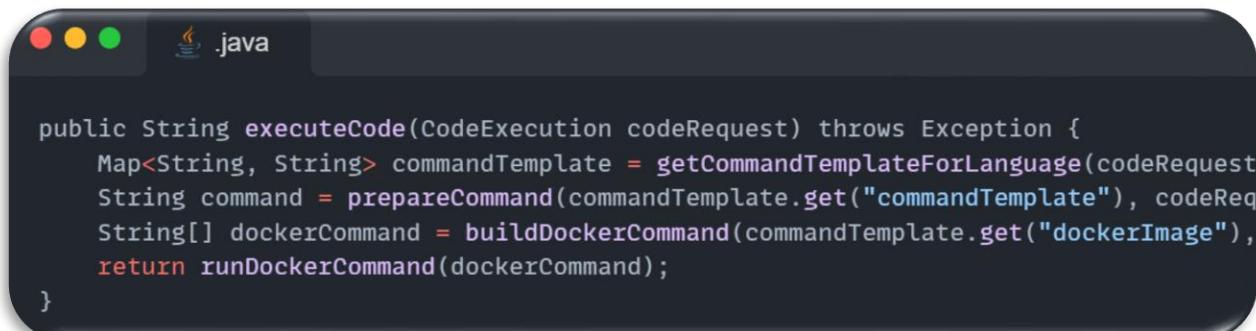
Example:



```
public String generateToken(Authentication authentication)
```

- **Defend:** In DockerExecutorService, the variables like commandTemplate, dockerCommand, and commandTemplates are descriptive and searchable.

Example:



```
public String executeCode(CodeExecution codeRequest) throws Exception {
    Map<String, String> commandTemplate = getCommandTemplateForLanguage(codeRequest);
    String command = prepareCommand(commandTemplate.get("commandTemplate"), codeRequest);
    String[] dockerCommand = buildDockerCommand(commandTemplate.get("dockerImage"));
    return runDockerCommand(dockerCommand);
}
```

7.2 Functions (Methods)

❖ [Principle] Small Functions & Doing One Thing:

- **Defend: getEmail(Authentication authentication):** This method is focused only on extracting an email from the authentication object, maintaining a singular responsibility. Its clear objective is to return the email, with no additional logic.

Example:

```
public String getEmail(Authentication authentication) {  
    if (authentication.getPrincipal() instanceof UserDetails userDetails) {  
        return userDetails.getUsername();  
    }  
    if (authentication instanceof OAuth2AuthenticationToken oauthToken) {  
        return oauthToken.getPrincipal().getAttribute("email");  
    }  
    throw new IllegalArgumentException("Unsupported authentication principal type");  
}
```

- **Defend: saveCodeUpdate(CodeUpdate codeUpdate)** This method is focused only on saving the temporary updating the code in live editing.

Example:

```
@Override  
public void saveCodeUpdate(CodeUpdate codeUpdate) {  
  
    String codeKey = codeUpdate.getRoomId() + "-" + codeUpdate.getProjectName()  
  
    synchronized (getCodeLock(codeKey)) {  
        updateCode(codeUpdate);  
    }  
}
```

- **Defend: createDefaultFile(Room room)** This method is focused only on create Default File.

Example:

```
private void createDefaultFile(Room room) {  
    try {  
        fileService.createFile(  
            FileDTO  
                .builder()  
                .filename("README")  
                .projectId(room.getRoomId())  
                .extension(".md")  
                .build()  
        );  
    } catch (Exception e) {  
        throw new RuntimeException("Failed to create default file: " + e.getMessage());  
    }  
}
```

7.2 Functions (Methods)

❖ [Principle] Methods should take the minimum number of arguments necessary.:

- **Defend:** signIn(LoginRequest loginRequest): The signIn method only takes a LoginRequest object, which encapsulates all the necessary data (email, password) in a single object. This reduces complexity and improves readability by avoiding multiple arguments.

Example:



```
public ResponseEntity<LoginResponse> signIn(@RequestBody LoginRequest loginRequest)
```

- **Defend:** executeCode(CodeExecution codeRequest): The executeCode method accepts just one object, CodeExecution, which contains the necessary properties like language, code, and input. This keeps the method signature clean and limits unnecessary arguments.

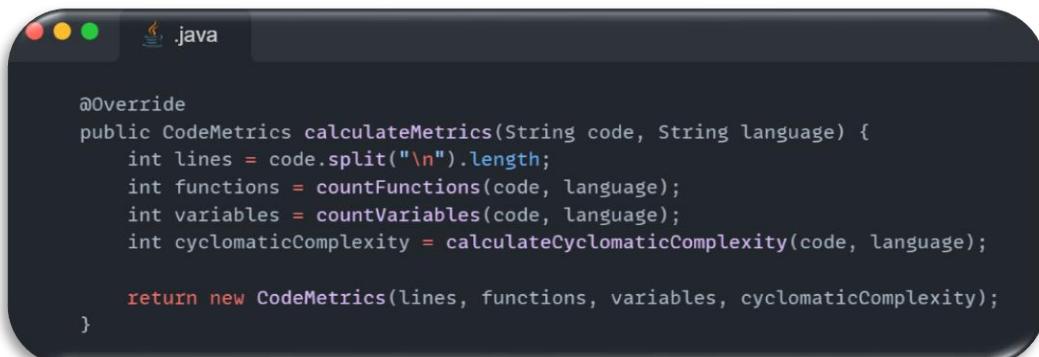
Example:



```
public String executeCode(CodeExecution codeRequest)
```

- **Defend:** calculateMetrics(String code, String language): The calculateMetrics method accepts just two objects.

Example:



```
@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);
}
```

7.3 Comments

- ❖ [Principle] Avoid unnecessary comments, use comments to explain complex decisions.

- **Defend:** If you have the chance to visit my code you could observe that you will not be able to find comments, because I believe it the code with applied (SOLID PRINCIPLES, CLEAN CODE And JAVA EFFECTIVE) It is not necessary to add any comments.

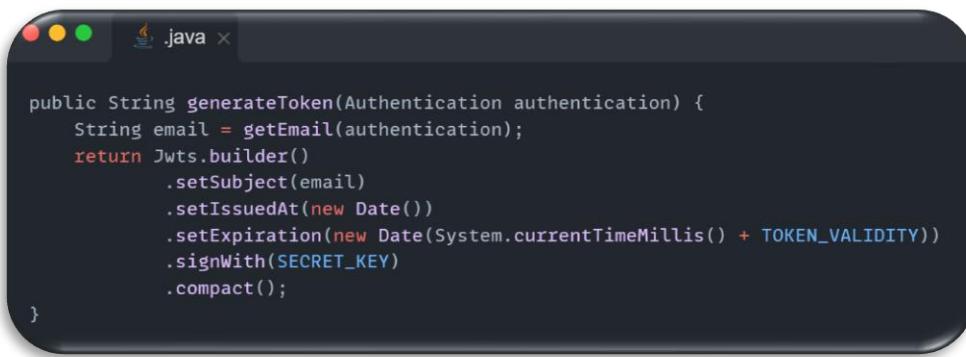
7.4 Formatting

- ❖ [Principle] Group related code vertically and use consistent horizontal/vertical formatting.

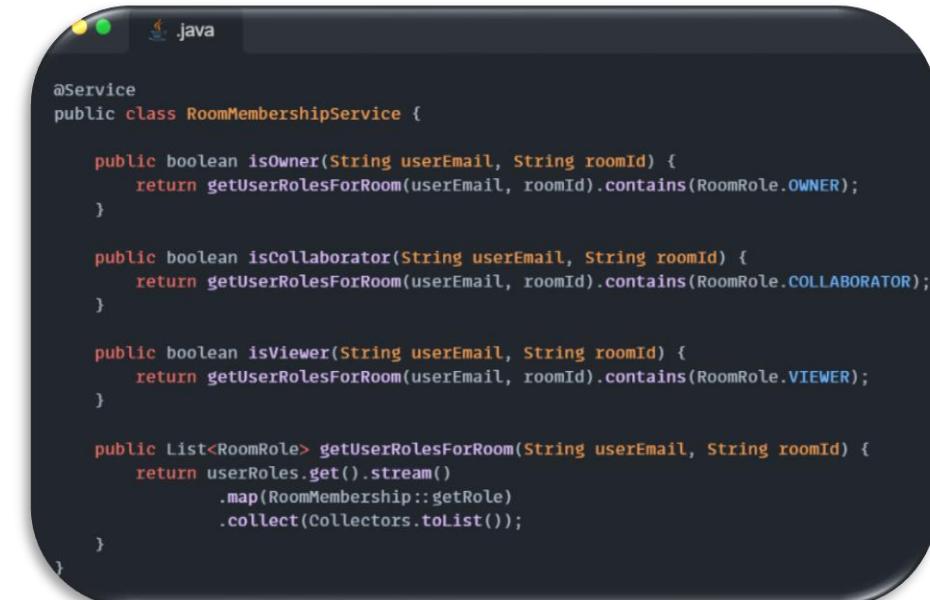
- **Defend:** Methods are grouped logically, and spacing is used effectively to separate logical blocks of code.

The logical flow of token generation is easy to read, with clear spacing between method components.

Example:



```
public String generateToken(Authentication authentication) {
    String email = getEmail(authentication);
    return Jwts.builder()
        .setSubject(email)
        .setIssuedAt(new Date())
        .setExpiration(new Date(System.currentTimeMillis() + TOKEN_VALIDITY))
        .signWith(SECRET_KEY)
        .compact();
}
```



```
@Service
public class RoomMembershipService {

    public boolean isOwner(String userEmail, String roomId) {
        return getUserRolesForRoom(userEmail, roomId).contains(RoomRole.OWNER);
    }

    public boolean isCollaborator(String userEmail, String roomId) {
        return getUserRolesForRoom(userEmail, roomId).contains(RoomRole.COLLABORATOR);
    }

    public boolean isViewer(String userEmail, String roomId) {
        return getUserRolesForRoom(userEmail, roomId).contains(RoomRole.VIEWER);
    }

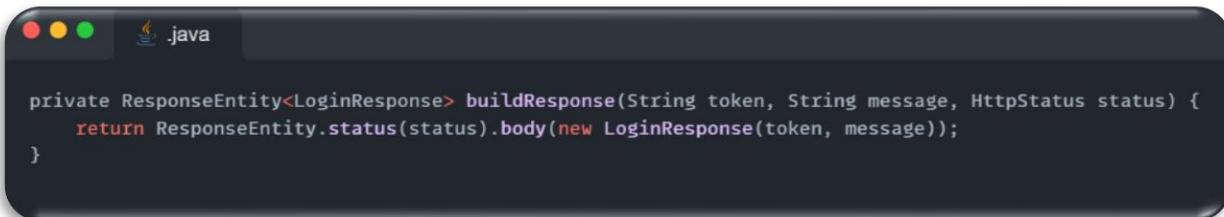
    public List<RoomRole> getUserRolesForRoom(String userEmail, String roomId) {
        return userRoles.get().stream()
            .map(RoomMembership::getRole)
            .collect(Collectors.toList());
    }
}
```

7.5 Don't Repeat Yourself

❖ [Principle] Avoiding Duplication – DRY Principle.

- **Defend:** Instead of having each controller or service build its response separately, this common logic is centralized into a single function that can be reused across different controllers or services. This ensures consistency and reduces code duplication.

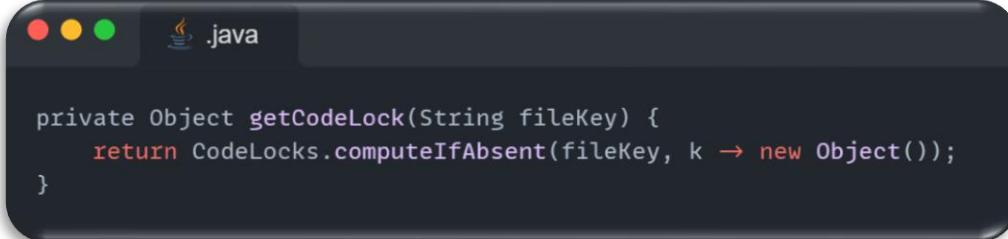
Example:



```
private ResponseEntity<LoginResponse> buildResponse(String token, String message, HttpStatus status) {
    return ResponseEntity.status(status).body(new LoginResponse(token, message));
}
```

- **Defend:** The logic for locking files during concurrent edits is abstracted into the getCodeLock(fileKey) method to avoid repeating the locking logic in multiple places. The method is reused in both insertComment and saveCodeUpdate methods.

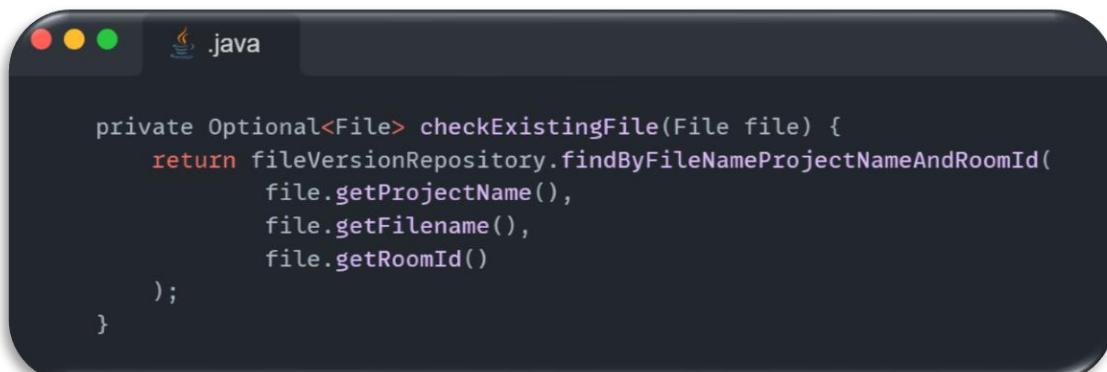
Example:



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

- **Defend:** The checkExistingFile(File file) method to avoid repeating the check Existing File logic in multiple places. The method is reused in different places inside the class **FileServiceImpl**.

Example:



```
private Optional<File> checkExistingFile(File file) {
    return fileVersionRepository.findByFileNameProjectNameAndRoomId(
        file.getProjectName(),
        file.getFilename(),
        file.getRoomId()
    );
}
```

7.6 Don't Repeat Yourself

- **Defend:** Logic for handling file content (e.g., pushing, pulling, and merging file content) is placed in the FileService, by centralizing the file operations in FileService, the code ensures that file management logic is not duplicated across different parts of the codebase, such as various controllers.

Example:



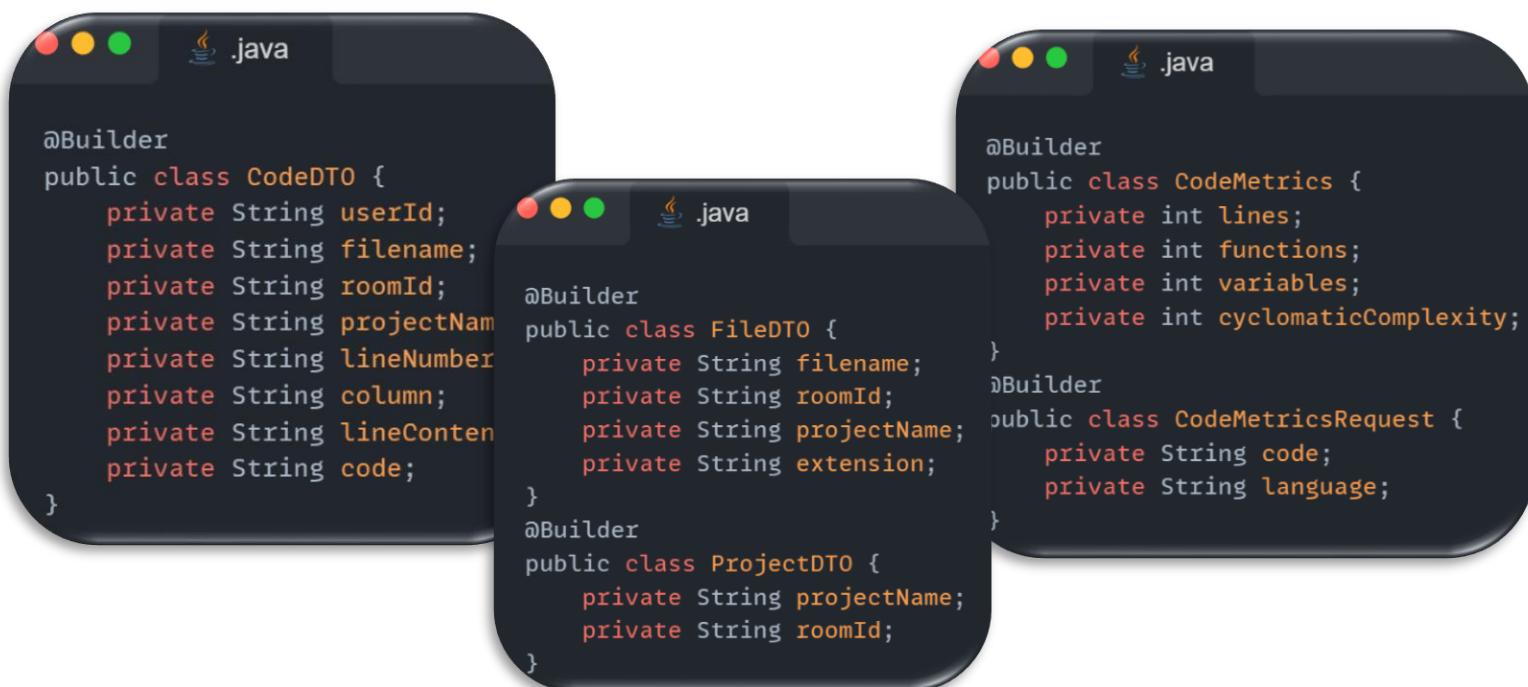
```
@PostMapping("/pull-file-content")
public ResponseEntity<Map<String, File>> pullFileContent(@RequestBody FileDTO fileDTO) {
    try {
        File pulledFile = fileService.pullFileContent(fileDTO);
        return ResponseEntity.ok(Map.of("file", pulledFile));
    } catch (FileNotFoundException e) {
        return ResponseEntity.notFound().build();
    }
}
```

7.7 Use of DTOs

❖ [Principle] Avoiding Duplication – DRY Principle.

- **Defend:** DTOs (Data Transfer Objects) are used extensively in the code to abstract data representations and facilitate communication between layers (controllers, services, etc.) so I used it most of the time when coding.

Examples:



```
@Builder
public class CodeDTO {
    private String userId;
    private String filename;
    private String roomId;
    private String projectName;
    private String lineNumber;
    private String column;
    private String lineContent;
    private String code;
}
```

```
@Builder
public class FileDTO {
    private String filename;
    private String roomId;
    private String projectName;
    private String extension;
}
```

```
@Builder
public class ProjectDTO {
    private String projectName;
    private String roomId;
}
```

```
@Builder
public class CodeMetrics {
    private int lines;
    private int functions;
    private int variables;
    private int cyclomaticComplexity;
}

@Builder
public class CodeMetricsRequest {
    private String code;
    private String language;
}
```

Effective Java

❖ [Item - 1] Consider static factory methods instead of constructors

- **Defense:** JwtUtil, SecurityConfig, and RoomController 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.

The image shows two separate Java code editors. The left editor contains code for `AuthenticationConfig.java` with annotations `@Bean` and `@Component`. The right editor contains code for `JwtUtil.java` with an annotation `@Component`.

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration config) {
    return config.getAuthenticationManager();
}

@Bean
public DaoAuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
    provider.setPasswordEncoder(new BCryptPasswordEncoder());
    provider.setUserDetailsService(userDetailsService);
    return provider;
}
```

```
@Component
public class JwtUtil { }
```

- **Defense:** The code does use constructors, but in appropriate places. For example, constructors in the **Project**, **MessageLog**, **Room**, **RoomMembership**, **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.

The image shows a Java code editor with a single file named `Room.java`. It contains annotations `@NoArgsConstructor`, `@AllArgsConstructor`, and `@Builder`, followed by a class definition for `Room`.

```
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Room { ... }
```

- **Defense:** 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.

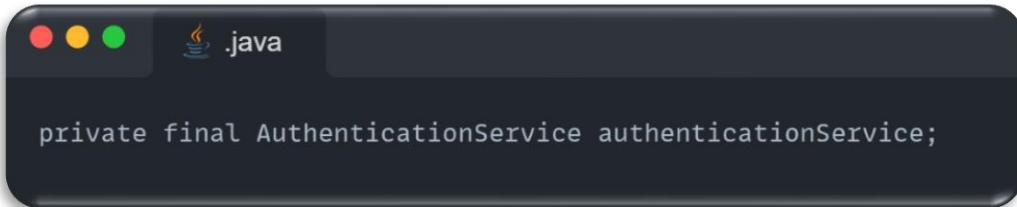
The image shows a Java code editor with a single file named `SignInController.java`. It contains a constructor with a `@Qualifier` annotation for `AuthenticationServiceImpl`.

```
public SignInController(@Qualifier("AuthenticationServiceImpl") AuthenticationService authenticationService) {
    this.authenticationService = authenticationService;
}
```

8.1 Effective Java

❖ [Item - 3] Minimize the accessibility of classes and members

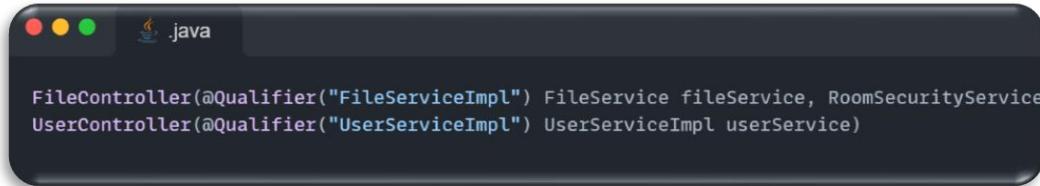
- **Defense:** the controllers are appropriately marked as private, such as the authenticationService in the SignInController. This is in line with the principle of encapsulation and keeping the class's internal representation hidden.



```
private final AuthenticationService authenticationService;
```

❖ [Item - 5] Prefer Dependency Injection to Hardwiring Resources

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



```
FileController(@Qualifier("FileServiceImpl") FileService fileService, RoomSecurityService
UserController(@Qualifier("UserServiceImpl") UserServiceImpl userService)
```

❖ [Item - 9] Try-with-Resources to try-finally

- **Defense:** 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.



```
@PostConstruct
public void loadCommandTemplates() throws Exception {
    ObjectMapper objectMapper = new ObjectMapper();
    try (InputStream templateStream = new ClassPathResource("data/commandTemplates.json").getInputStream()) {
        commandTemplates = objectMapper.readValue(templateStream, new TypeReference<>() {});
    }
}
```

❖ [Item - 10] In public classes, use accessor methods, not public fields

- **Defense:** the entire classes I used private fields with @Getter @Setter annotations.

8.1 Effective Java

❖ [Item - 17] Favor Composition Over Inheritance

- **Defense:** entity relationships (e.g., RoomMembership to Room and User) are represented with composition, not inheritance. The relationships are modeled with @ManyToOne and @OneToMany, which reflects a correct design choice.

❖ [Item - 20] Prefer Interfaces over abstract classes

- **Defense:** Interfaces are preferable than the abstract classes, that's because Java only allows single inheritance, so I used the interfaces in most of the services.

```
public interface FileService {  
    List<FileDTO> getFiles(ProjectDTO project);  
  
    void createFile(FileDTO fileDTO);  
  
    void pushFileContent(File file);  
  
    File mergeFileContent(File newVersion);  
  
    File pullFileContent(FileDTO fileDTO);  
}  
  
public interface ProjectService {  
    void createProject(ProjectDTO project);  
  
    List<ProjectDTO> getProjects(String roomId);  
  
    void deleteProject(ProjectDTO projectDTO);  
  
    void downloadProject(ProjectDTO projectDTO, String downloadPath);  
}
```

❖ [Item - 30] Use enums instead of int constants

- **Defense:** the **RoomRole** enum in methods like addMember and removeMember is excellent. It enhances code clarity and type safety.

```
public enum AuthConstants {  
    INVALID_CREDENTIALS,  
    USER_NOT_FOUND,  
    WRONG_PASSWORD,  
    SESSION_EXPIRED,  
    USER_ALREADY_EXISTS,  
    USER_NOT_ACTIVE,  
    USER_ALREADY_REGISTERED,  
    EMAIL_ALREADY_EXISTS,  
    USER_NOT_VERIFIED,  
    USER_ALREADY_BLOCKED,  
    USER_ROLE_NOT_ALLOWED,  
    USER_EMAIL_NOT_VERIFIED,  
    USER_EMAIL_ALREADY_EXISTS,  
    USER_PASSWORD_NOT_COMPLEX  
}  
  
public enum RoomRole {  
    OWNER,  
    COLLABORATOR,  
    VIEWER  
}  
  
public enum AccountSource {  
    GITHUB,  
    GOOGLE,  
    CODE_EDITOR  
}
```

8.1 Effective Java

❖ [Item - 39] Use annotations

- **Defense:** Annotations such as @Test, @BeforeEach, @Mock, and @InjectMocks are used correctly in your test cases. They make tests cleaner and easier to maintain.

```
class LogsServiceImplTest {  
    @InjectMocks  
    private LogsServiceImpl logsService;  
  
    @Mock  
    private LogsRepository logsRepository;  
  
    @BeforeEach  
    void setUp() {  
        MockitoAnnotations.openMocks(this);  
    }  
  
    @Test  
    void testSaveLog() {  
    }  
}
```

❖ [Item - 16] Know and Use Libraries

- **Defense:** In this project I researched libraries to help me in this project and made an effective use of them, I used
 - Spring boot: for building the API and for networking aspects
 - Lombok: for implementing builder classes and implementing getters and setters
 - Jason: for parsing and handling JSON processing
 - Junit: for writing automated tests
 - JBcrypt: for handling hashing passwords and comparing hashes with plain passwords

❖ [Item - 18] Prefer lambdas to anonymous classes:

- **Defense:** I used them quite often especially when working with streams.

```
return files.stream()  
    .map(file →  
        FileDTO  
            .builder()  
            .filename(file.getFilename())  
            .roomId(file.getRoomId())  
            .projectId(file.getProjectName())  
            .extension(file.getExtension())  
            .build()  
    )  
    .collect(Collectors.toList());
```

8.1 Effective Java

❖ [Item - 19] Document all exceptions thrown by each method:

- **Defense:** I used them quite often especially when working with the Centralized Exceptions.

```
@ExceptionHandler(UserNotFoundException.class)
@ResponseBody(HttpStatus.NOT_FOUND)
public Map<String, String> handleUserNotFoundException(UserNotFoundException ex) {
    logger.warn("User not found: {}", ex.getMessage(), ex);
    return createErrorResponse("User Not Found", ex.getMessage(), "USER_NOT_FOUND");
}
```

❖ [Item - 49] Check parameters for validity

- **Defense:** 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.

```
if (loginRequest == null || loginRequest.getUsername() == null) {
    throw new IllegalArgumentException("Invalid login request");
}
```

❖ [Item - 67] Optimize method signatures

- **Defense:** with minimal parameters. This is especially true for the methods handling simple requests like signIn, createAccount, and runCode.

```
@PostMapping("/sign-in")
public ResponseEntity<LoginResponse> signIn(@RequestBody LoginRequest loginRequest) {
    ...
}
```

❖ [Item - 77+85] Synchronize Access to Shared Mutable Data && Prefer Executors and Tasks to Threads

- **Defense:** using the fileLocks map and the synchronized keyword in methods like pushFileContent and mergeFileContent

```
private final Map<String, Object> fileLocks = new ConcurrentHashMap<>();
private Object getFileLock(String roomId) {
    return fileLocks.computeIfAbsent(roomId, k -> new Object());
}

String fileKey = newVersion.getRoomId() + "-" + new
synchronized (getFileLock(fileKey)) {
    ...
}
```

```
private final ExecutorService executorService;
@Bean
@Qualifier("webSocketExecutorService")
public ExecutorService webSocketExecutorService() {
    return Executors.newFixedThreadPool(100);
}
```

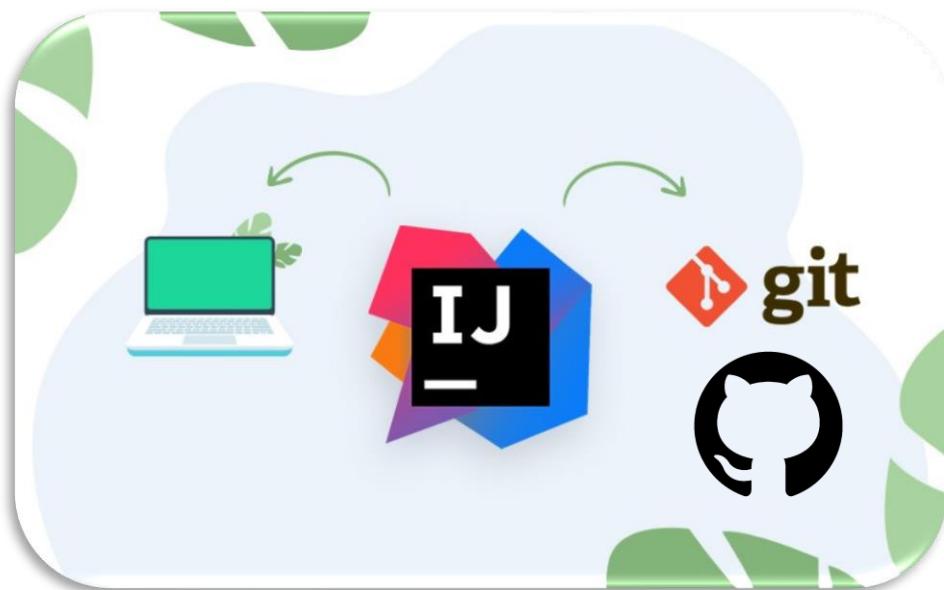
DevOps Practices

In the project, I implemented several key DevOps practices that significantly enhanced the workflow, scalability, and maintainability. These include **GitHub** for version control, **Docker** for containerization, **AWS** for deployment and scalability, and a **CI/CD** pipeline setup using **GitHub Actions** and **DockerHub**. I will explain why these choices were critical for the project and how they contributed to its success.

9.1 Git & GitHub

I chose **GitHub** as the version control system for this project due to its robust collaboration features.

Additionally, GitHub's integration with other DevOps tools, such as **CI/CD** pipelines (like **GitHub Actions**), **DockerHub**, and **AWS**, streamlined the entire development and deployment process. Its issue tracking system and project boards also helped manage tasks, track progress, and maintain documentation all in one place. By using GitHub, the project benefited from a version control system that facilitated collaboration and continuous integration with other tools.



In this project, I used the **Git GUI** integrated with **IntelliJ IDEA**, which streamlined the process of managing version control directly within the development environment. After completing each feature, I pushed the changes to a dedicated branch, ensuring that each feature was isolated from the main codebase to prevent conflicts. **Once the feature was ready**, I created a **pull request** to review how well the feature integrated with the entire application before merging it into the main branch.

9.1 Git & GitHub

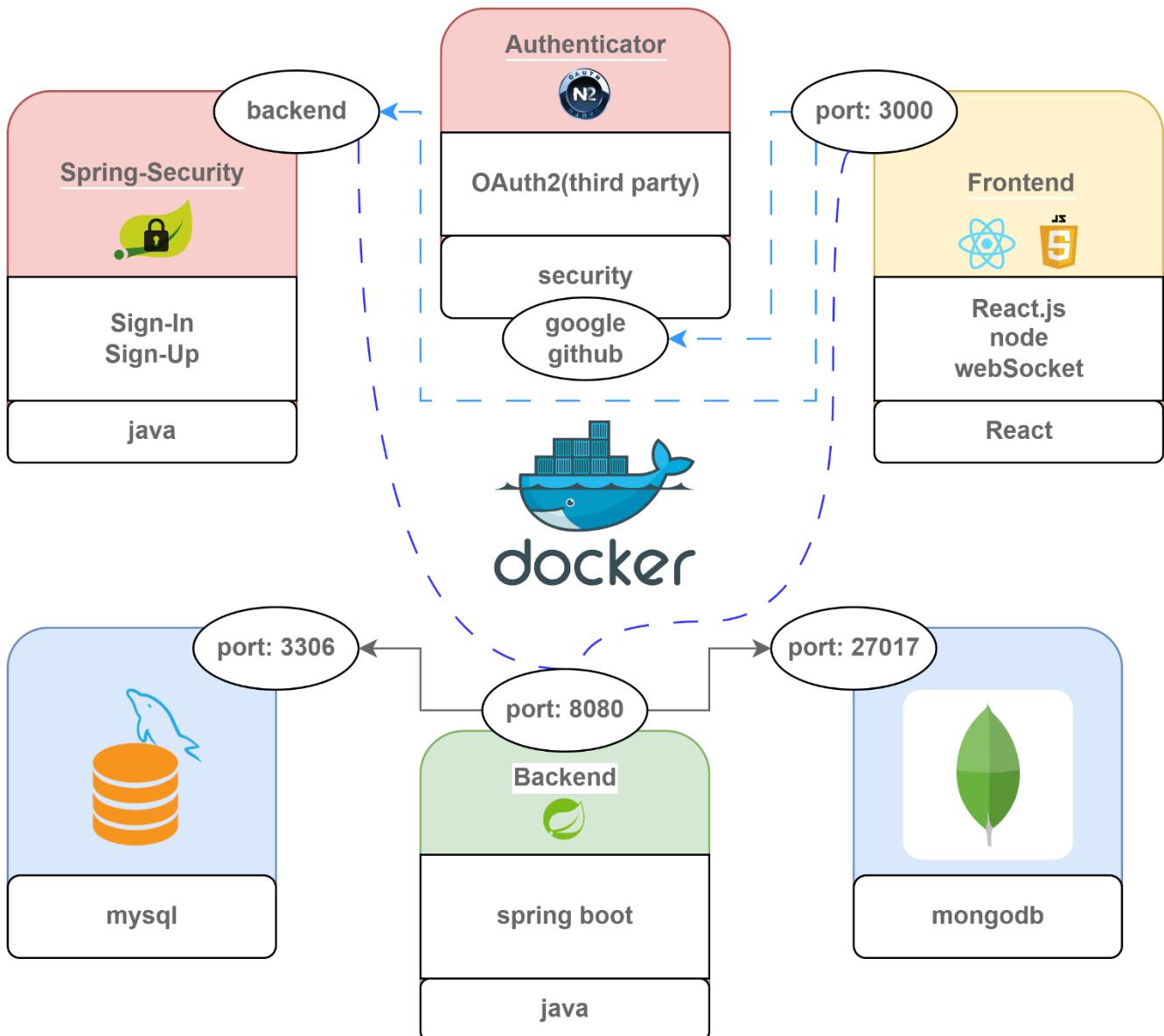
The screenshot shows a GitHub repository interface. At the top, it displays the repository name "feature-chat-message", 5 branches, and 2 tags. Below this, a message states "This branch is 31 commits ahead of, 2 commits behind main". A "Contribute" button is also present. The main area shows a list of 31 commits from AbdullahAymanII, all made 25 minutes ago. The commits include changes to .github/workflows, .idea, docker, editor, frontend, logs, .dockerignore, .gitignore, README.md, and docker-compose.yaml. The "README" file is also listed. On the left, there's a detailed commit history for the "origin & feature-chat-message" branch, showing numerous commits from Abdullah Ayman across various dates. On the right, a file tree shows the directory structure: .\.\editor (10 files), editor\src\main\java\com\collaborative\editor (7 files), and configuration\websocket (1 file). The "WebSocketConfig.java" file is highlighted.

In the project's **.gitignore** file, I ensured that certain directories and files were excluded from version control to maintain a clean repository and prevent unnecessary files from being tracked.

Node.js/React (Frontend):

- This directory contains the dependencies installed by npm. there's no need to version them, which would otherwise lead to large commits.

Docker



In this project, I used Docker to streamline both **development and deployment** environments. For development, I created Dockerfiles and a Docker Compose setup that supports a multi-container architecture. This architecture includes services for the backend, frontend, MySQL, and MongoDB, all of which work together.

10.1 Docker for Development

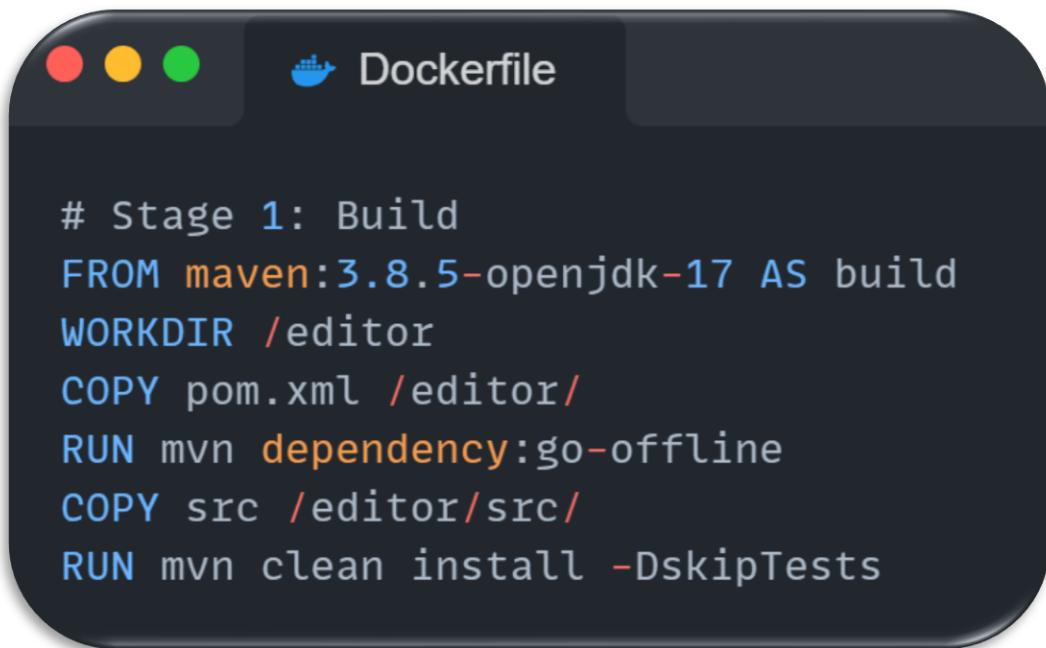
The backend is built using Maven with a multi-stage Dockerfile, while the frontend uses Node.js with live reloading via Nodemon. MySQL and MongoDB containers are included, along with their management UIs, phpMyAdmin and Mongo Express, to simplify database management during development. The Docker Compose file ensures all services run together in isolated containers, making the development process more efficient and consistent.

1. Backend (Dockerfile and Docker Compose)

- **Dockerfile for Backend (Spring Boot):** The backend is built using a two-stage multi-stage build process to optimize the container size and isolate the build from the run environment.

- **Stage 1: Build**

I used a Maven Docker image to compile the project. It installs dependencies in offline mode to reduce build times and ensures that all necessary libraries are available. The `pom.xml` and source code are copied, then the application is built using `mvn clean install -DskipTests`, skipping the tests **to save time during development**.



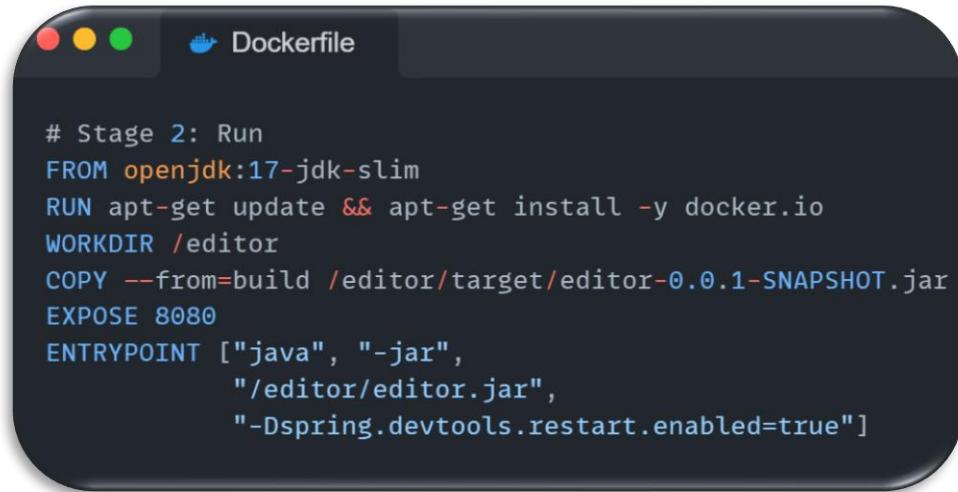
A screenshot of a terminal window titled "Dockerfile". The window has a dark theme with red, yellow, and green close buttons. The content of the terminal shows a Dockerfile with the following code:

```
# 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
```

10.1 Docker for Development

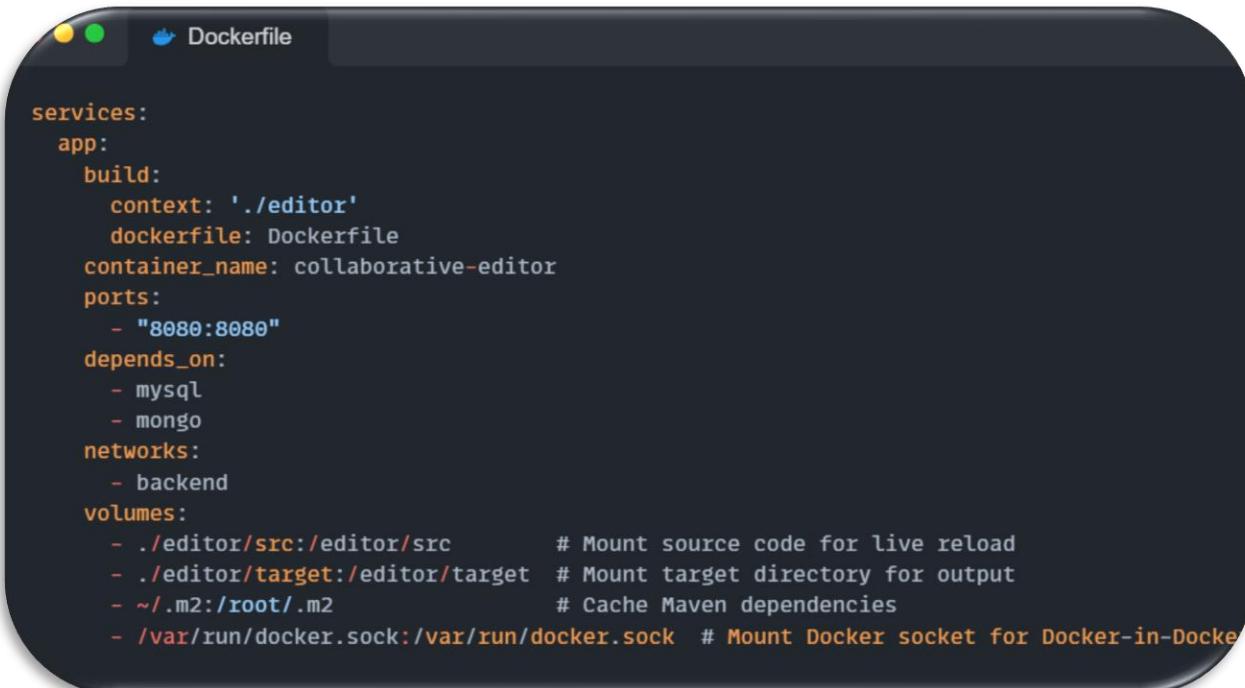
- Stage 2: Run

The application is run in a slim OpenJDK 17 image. I installed Docker CLI within the container to support Docker-in-Docker functionality, allowing commands to be executed inside the container (**for RUN the code of the editor**). The JAR file from the build stage is copied and executed with **Spring Boot's DevTools enabled for live reload (only for development)**.



```
# Stage 2: Run
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
EXPOSE 8080
ENTRYPOINT ["java", "-jar",
            "/editor/editor.jar",
            "-Dspring.devtools.restart.enabled=true"]
```

- **Docker Compose for Backend:** In the docker-compose.yml file, the backend service is defined with a build context and specific ports exposed. To ensure the backend works seamlessly in the development environment, I used volumes to mount the source code and Maven dependencies to enable live reload. Additionally, I mounted the **Docker socket to allow the backend to interact with Docker (Docker-in-Docker)**.



```
services:
  app:
    build:
      context: './editor'
      dockerfile: Dockerfile
    container_name: collaborative-editor
    ports:
      - "8080:8080"
    depends_on:
      - mysql
      - mongo
    networks:
      - backend
    volumes:
      - ./editor/src:/editor/src      # Mount source code for live reload
      - ./editor/target:/editor/target # Mount target directory for output
      - ~/.m2:/root/.m2              # Cache Maven dependencies
      - /var/run/docker.sock:/var/run/docker.sock # Mount Docker socket for Docker-in-Docker
```

10.1 Docker for Development

2. Frontend (Dockerfile and Docker Compose)

- **Dockerfile for Frontend (React):** The frontend uses a Node.js base image, with Yarn as the package manager. I installed the dependencies and **Nodemon for live reloading during development**. The source code is copied into the container, and the container exposes port 3000 for the React application.

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

- **Docker Compose for Frontend:** The frontend service in the Docker Compose file builds from its Dockerfile, exposing port 3000. Like the backend, volumes are used to mount the source code to allow live reloading of changes during development.

```
frontend:
  build:
    context: './frontend'
    dockerfile: Dockerfile
  container_name: collaborative-frontend
  ports:
    - "3000:3000"
  depends_on:
    - app
  networks:
    - backend
  volumes:
    - ./frontend/src:/app/src # Mount frontend code for live reload
```

10.1 Docker for Development

3. MySQL (Docker Compose)

- **For the MySQL database**, I used the official MySQL 8.0 image. Environment variables were set for the root user, password, and the initial database. Port 3306 is exposed to connect to the database from the backend, PHP MyAdmin service is included for a simple web interface to view and manage mysql data.

```
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:
    - "3306:3306"
  volumes:
    - mysql_data:/var/lib/mysql
  networks:
    - backend
```

```
# phpMyAdmin Service
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"
  networks:
    - backend
  depends_on:
    - mysql
```

4. MongoDB (Docker Compose)

- **For MongoDB**, I used the official MongoDB image with environment variables for the root username, password, and database. Mongo Express is included for a simple web interface to view and manage MongoDB data.

```
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/mongo_db
    - ./mongo_init:/docker-entrypoint-initdb.d
  networks:
    - backend
```

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

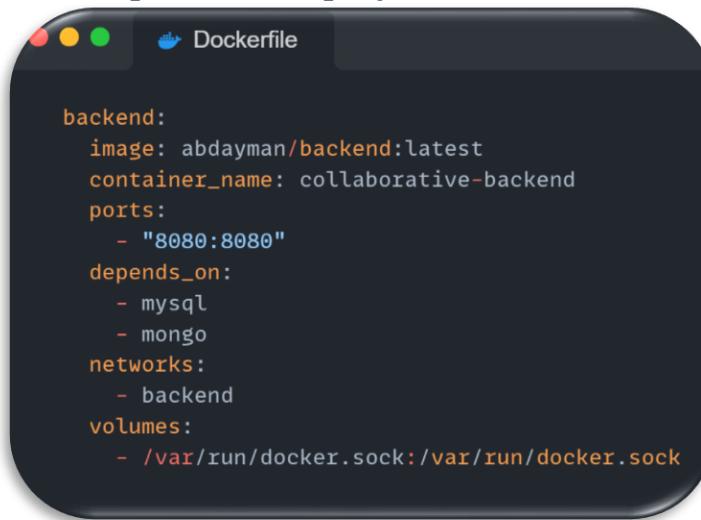
10.2 Docker for Deployment

By using the images which created in dockerHub, I used them in deployment at docker instead of AWS because the free tier and cost constraint in AWS.

Here no need to use the docker file, only using the docker compose will be enough.

Name	Image	Status	Port(s)
docker		Running (4/4)	
collaborative-mysql	mysql:8.0	Running	3306:3306
collaborative-mongo	mongo:<none>	Running	27017:27017
collaborative-backend	abdayman/backend:latest	Running	8080:8080
collaborative-frontend	abdayman/frontend:latest	Running	3000:3000

1. Backend in Docker Compose for Deployment



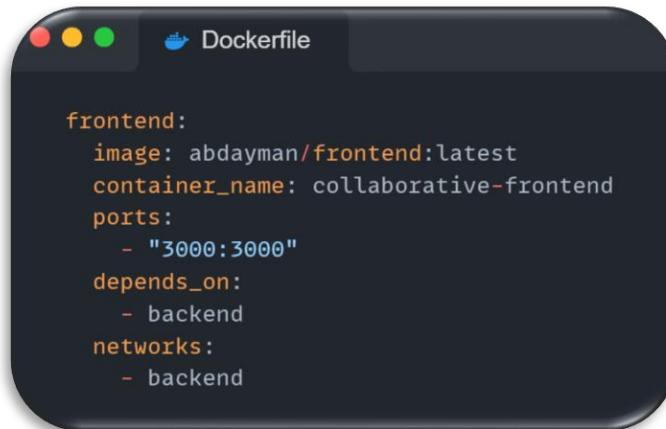
```
backend:
  image: abdayman/backend:latest
  container_name: collaborative-backend
  ports:
    - "8080:8080"
  depends_on:
    - mysql
    - mongo
  networks:
    - backend
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
```

Explanation:

- **Image:** Specifies a custom image (**abdayman/backend:latest**), which should already be built and available.
- **Ports:** Maps port 8080 of the host to port 8080 of the container, allowing access to the backend application.
- **Depends On:** Indicates that this service should start after the mysql and mongo services are up and running, ensuring the backend has access to its database dependencies.
- **Networks:** Connects this service to the backend network.
- **Volumes:**
- **Mounts the Docker socket** (**/var/run/docker.sock**) into the container to enable Docker-in-Docker functionality, allowing the backend service to run the codes.

10.2 Docker for Deployment

2. Frontend in Docker Compose for Deployment

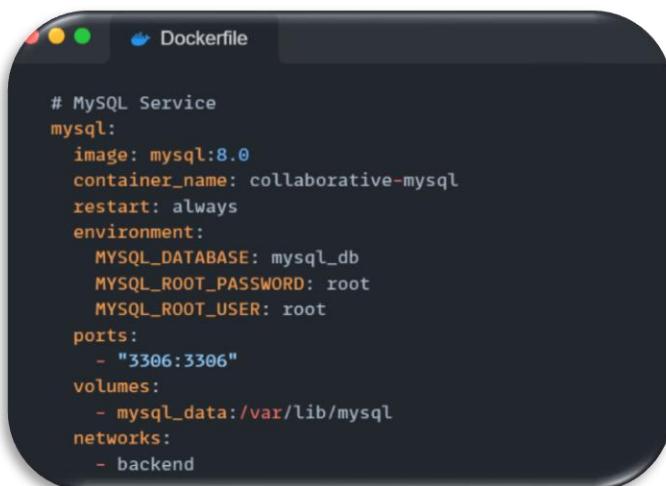


```
frontend:
  image: abdayman/frontend:latest
  container_name: collaborative-frontend
  ports:
    - "3000:3000"
  depends_on:
    - backend
  networks:
    - backend
```

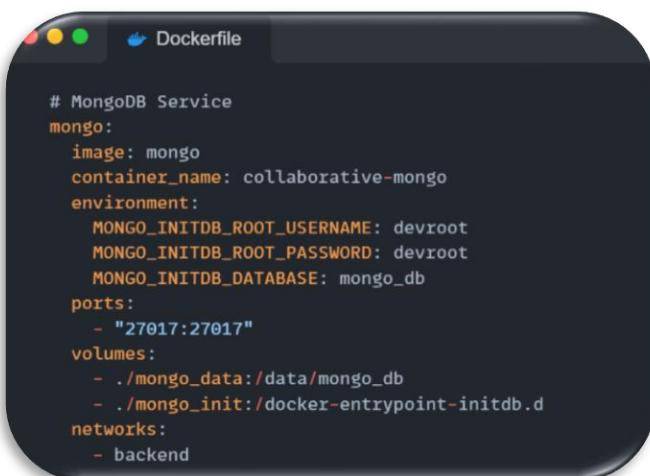
Explanation:

- Image: Specifies a custom image (abdayman/frontend:latest) for the frontend application.
- Ports: Maps port 3000 of the host to port 3000 of the container, allowing access to the frontend application.
- Depends On: Indicates that this service should start after the backend service is up and running, ensuring the frontend can communicate with the backend.
- Networks: Connects this service to the backend network.

3. Mysql and MongoDB Docker Compose for Deployment



```
# MySQL Service
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:
    - "3306:3306"
  volumes:
    - mysql_data:/var/lib/mysql
  networks:
    - backend
```



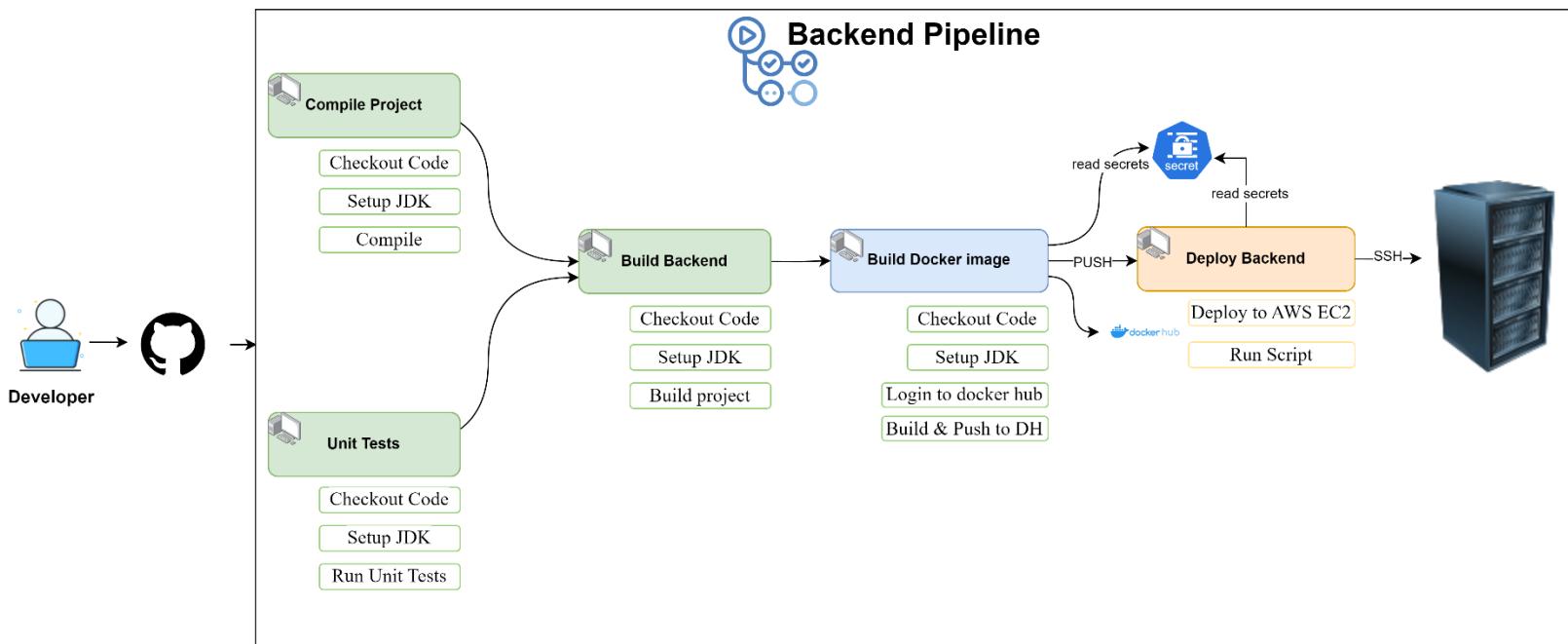
```
# MongoDB Service
mongo:
  image: mongo
  container_name: collaborative-mongo
  environment:
    MONGO_INITDB_ROOT_USERNAME: devroot
    MONGO_INITDB_ROOT_PASSWORD: devroot
    MONGO_INITDB_DATABASE: mongo_db
  ports:
    - "27017:27017"
  volumes:
    - ./mongo_data:/data/mongo_db
    - ./mongo_init:/docker-entrypoint-initdb.d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

CI/CD Pipeline



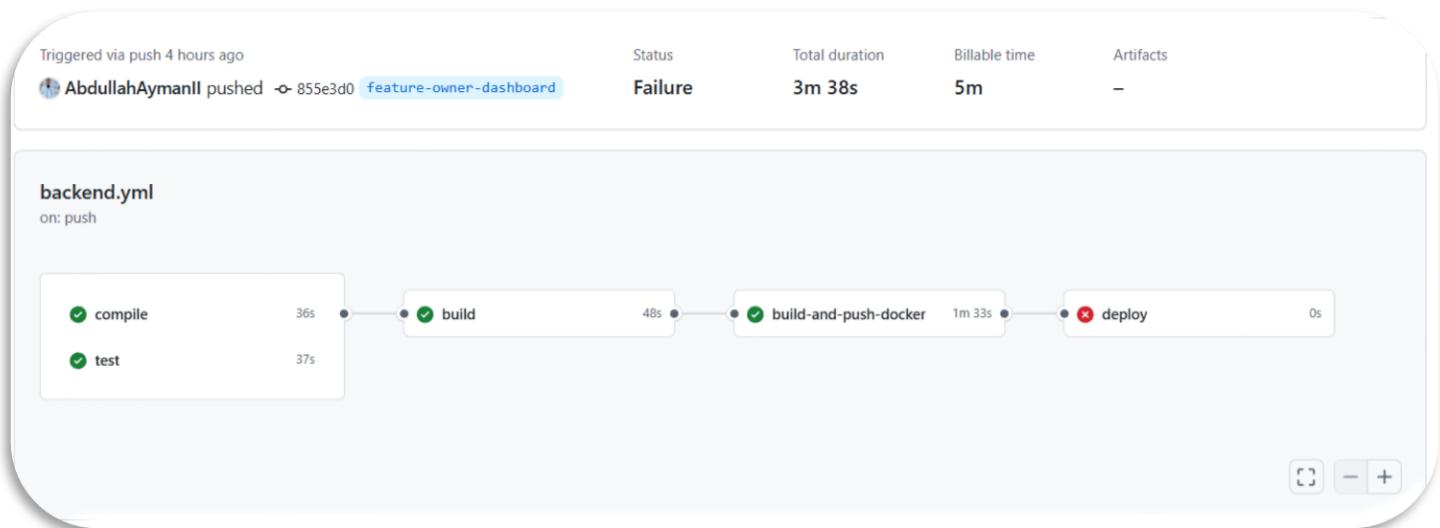
In my project, I used **GitHub Actions** for CI/CD. I created **two separate pipelines**: one for the backend and one for the frontend. Each pipeline has different stages for testing, building, and deploying the code. However, [due to cost constraints and limitations with the AWS free tier, the deployment to AWS is not functioning \(I did the deployment in the previous part\)](#). Below is an explanation of the CI/CD pipeline for both the backend and frontend.

11.1 Backend Pipeline



This pipeline is responsible for compiling, testing, building, pushing Docker images, and deploying the backend code.

11.1 Backend Pipeline



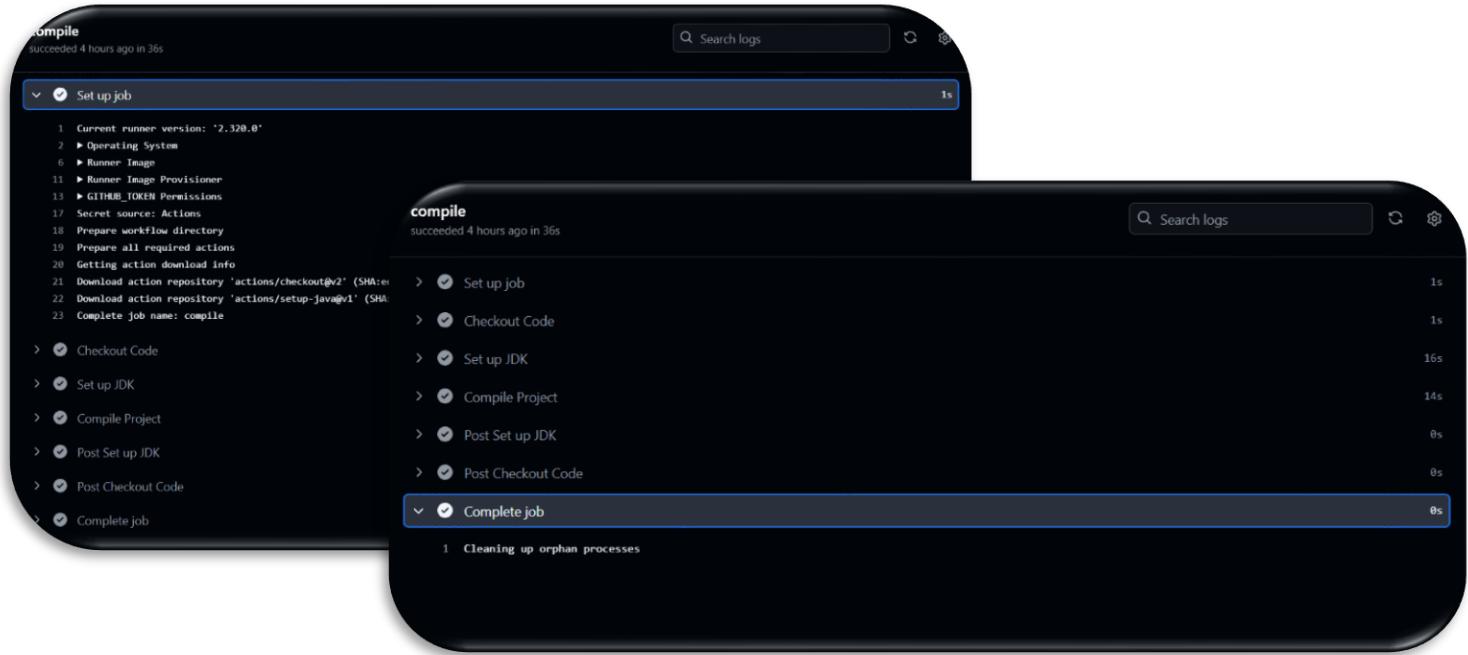
1. Compile Job:

This stage compiles the backend project to ensure the codebase is in good condition.

```
compile:
  runs-on: ubuntu-latest
  steps:
    - name: Checkout Code
      uses: actions/checkout@v2
    - name: Set up JDK
      uses: actions/setup-java@v1
      with:
        java-version: '17'
    - name: Compile Project
      run: mvn clean compile
      working-directory: ./editor
```

- **Checkout Code:** Retrieves the latest code from the repository.
- **Set up JDK:** Configures the environment with JDK 17 to match the backend's Java requirements.
- **Compile Project:** Runs `mvn clean compile` to compile the project in the editor directory.

11.1 Backend Pipeline



2. Test Job:

In this stage, unit tests are executed to validate the code.

```
test:  
  runs-on: ubuntu-latest  
  steps:  
    - name: Checkout Code  
      uses: actions/checkout@v2  
    - name: Set up JDK  
      uses: actions/setup-java@v1  
      with:  
        java-version: '17'  
    - name: Run Unit Tests  
      run: mvn test -DskipTests=false  
      working-directory: ./editor
```

- Run Unit Tests: Executes unit tests using Maven (mvn test). The -DskipTests=false ensures that tests are not skipped.

11.1 Backend Pipeline

The screenshot shows a CI pipeline interface with three main stages:

- Set up job:** Includes steps like Current runner version, Operating System, Runner Image, GitHub TOKEN Permissions, Secret source: Actions, Prepare workflow directory, Prepare all required actions, Getting action download info, Download action repository, Download action repository, and Complete job name: test.
- Run Unit Tests:** Shows log output from 12115 to 12133, indicating tests run, failures, errors, and skipped, along with build success and completion details.
- Post Set up JDK:** Includes steps like Post job cleanup, Post Checkout Code, and Complete job.

❖ Junit & Mockito (Testing)

In my project, I've implemented unit tests for various controller and service classes using JUnit and Mockito.



11.1 Backend Pipeline

- **SignInControllerTest** and **SignUpControllerTest**

Tests:

- **signInSuccess**: Verifies successful login.
- **signInInvalidCredentials**: Tests login failure due to invalid credentials.
- **signInUserNotFound**: Tests failure when the user is not found.
- **createAccountSuccess**: Tests successful account creation.
- **createAccountFailureDueToExistingEmail**: Tests failure due to an already existing email.
- **createAccountFailureDueToInvalidEmail**: Tests failure due to invalid email format.
- **createAccountFailureDueToWeakPassword**: Tests failure due to weak password.
- **providerSignIn**: Tests OAuth provider sign-in for Google and GitHub.

```
① @Test
void createAccountSuccess() {
    ResponseEntity<Map<String, String>> response = signUpController.createAccount(user);

    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals("Account created successfully", response.getBody().get("message"));
}

② @Test
void createRoom_success() {
    when(userService.findUserByEmail(anyString())).thenReturn(Optional.of(owner));
    when(roomService.createRoom(any(User.class), anyString())).thenReturn("roomId123");

    ResponseEntity<Map<String, String>> response = roomController.createRoom(createRoomRequest);

    assertEquals(200, response.getStatusCodeValue());
    assertNotNull(response.getBody());
    assertEquals("roomId123", response.getBody().get("room"));
}

③ public class SignUpControllerTest {

    @InjectMocks
    private SignUpController signUpController;

    @Mock
    private UserServiceImpl userService;

    private User user;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
        user = new User();
        user.setEmail("test@example.com");
        user.setPassword("testUserPassword");
        user.setUsername("testUser");
    }
}
```

11.1 Backend Pipeline

- RoomControllerTest

Tests:

- **createRoom_success:** Verifies room creation success.
- **deleteRoom_success:** Verifies successful room deletion.
- **addMember_success:** Tests successful addition of a member.
- **getRoomDetails_success:** Verifies successful retrieval of room details.
- **renameRoom_success:** Verifies room renaming.



```
@Test
void createRoom_success() {
    when(userService.findUserByEmail(anyString()).thenReturn(Optional.of(owner));
    when(roomService.createRoom(any(User.class), anyString()).thenReturn("roomId123");

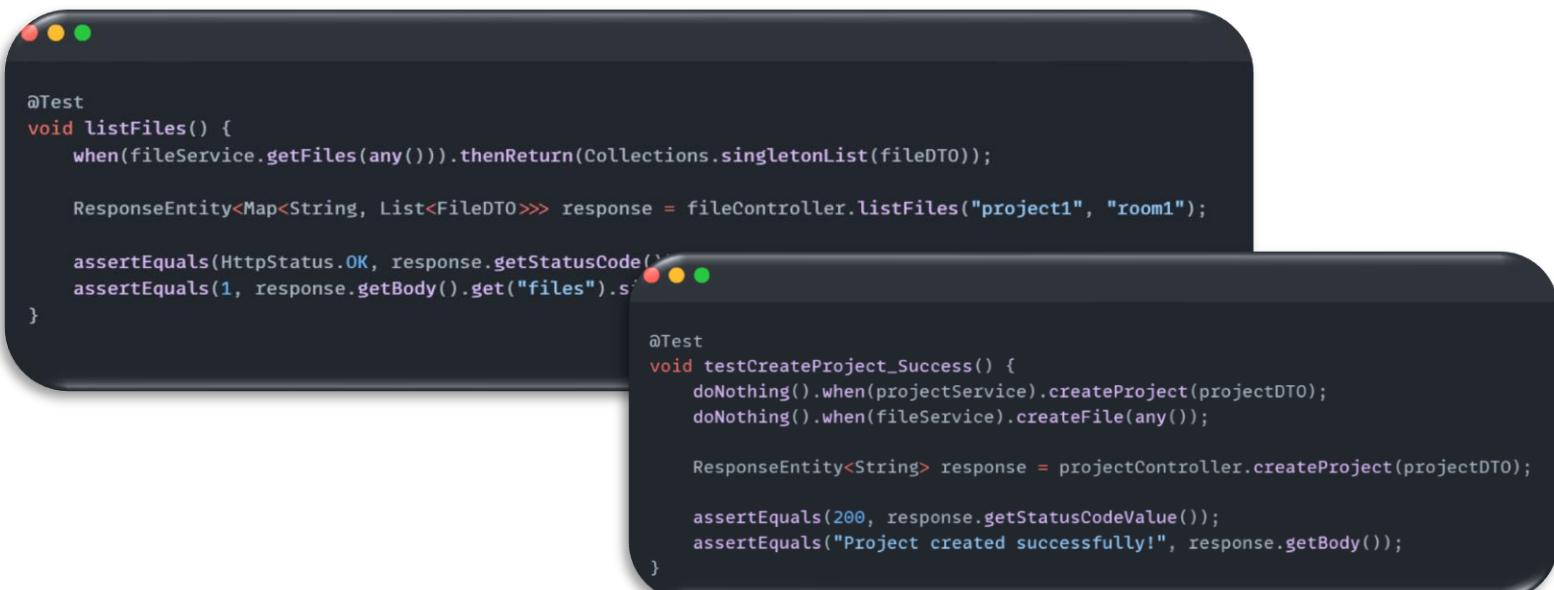
    ResponseEntity<Map<String, String>> response = roomController.createRoom(createRoomRequest);

    assertEquals(200, response.getStatusCodeValue());
    assertNotNull(response.getBody());
    assertEquals("roomId123", response.getBody().get("roomId"));
}
```

- FileControllerTest and ProjectControllerTest

Tests:

- **listFiles:** Tests listing of files in a project.
- **createFile:** Tests file creation.
- **pullFileContent:** Tests pulling file content.
- **mergeFileContent:** Tests merging file content.
- **testCreateProject_Success:** Tests successful project creation.
- **testCreateProject_Failure:** Tests project creation failure.
- **testListProjects_Success:** Verifies successful listing of projects.
- **testDeleteProject_Success:** Verifies successful project deletion.



```
@Test
void listFiles() {
    when(fileService.getFiles(any())).thenReturn(Collections.singletonList(fileDTO));

    ResponseEntity<Map<String, List<FileDTO>>> response = fileController.listFiles("project1", "room1");

    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals(1, response.getBody().get("files").size());
}

@Test
void testCreateProject_Success() {
    doNothing().when(projectService).createProject(projectDTO);
    doNothing().when(fileService).createFile(any());

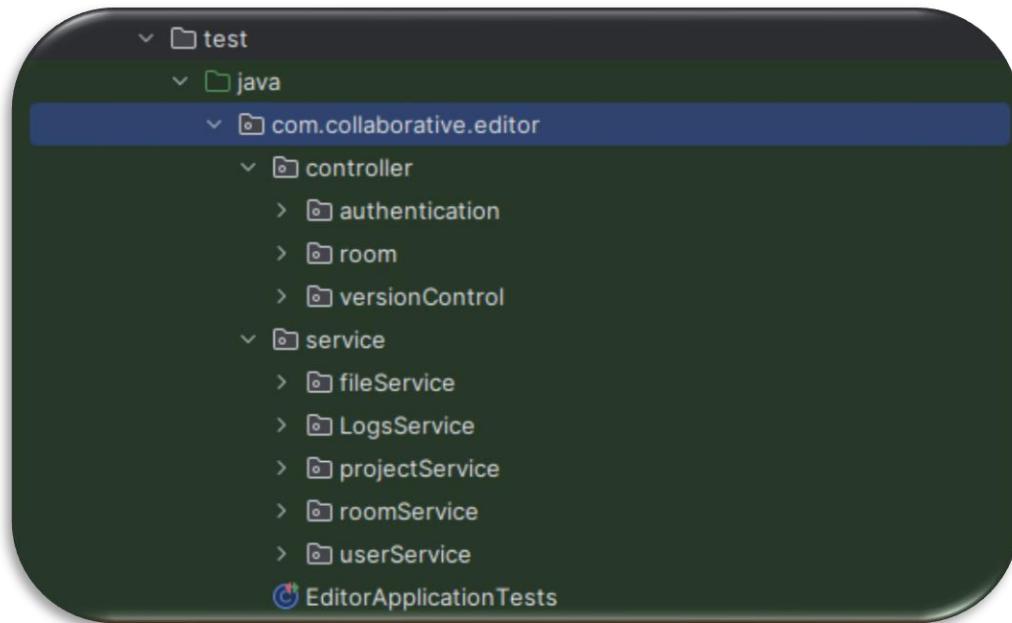
    ResponseEntity<String> response = projectController.createProject(projectDTO);

    assertEquals(200, response.getStatusCodeValue());
    assertEquals("Project created successfully!", response.getBody());
}
```

11.1 Backend Pipeline

Note

The other Unit tests you can find them inside the codes.



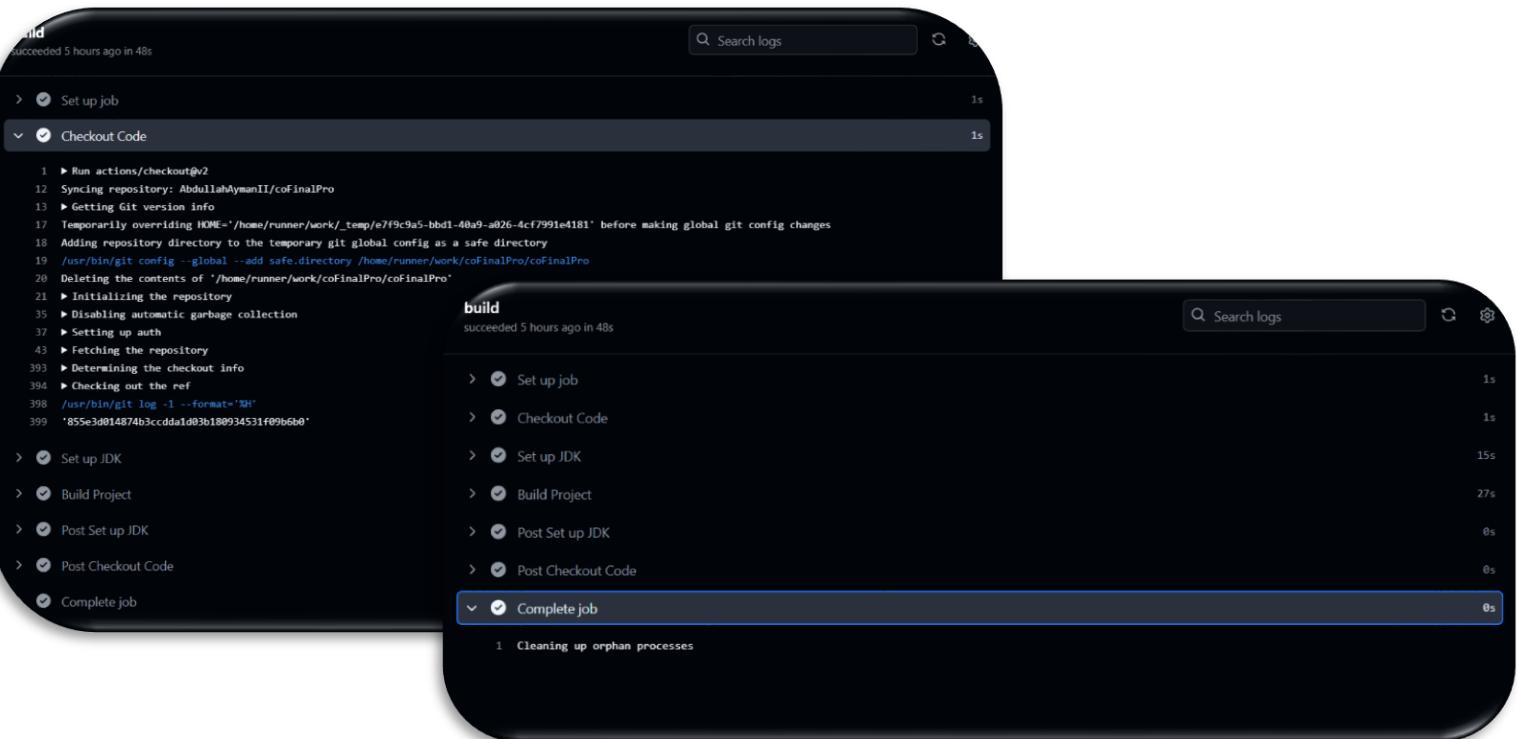
3. Build Job:

This stage builds the final JAR file for deployment after both **compilation and testing are completed**.

```
build:
  runs-on: ubuntu-latest
  needs: [test, compile]
  steps:
    - name: Checkout Code
      uses: actions/checkout@v2
    - name: Set up JDK
      uses: actions/setup-java@v1
      with:
        java-version: '17'
    - name: Build Project
      run: mvn clean package -DskipTests
      working-directory: ./editor
```

- **Needs:** This job depends on both compile and test jobs being successful.
- **Build Project:** Uses mvn clean package to package the backend project, creating the JAR file for later use.

11.1 Backend Pipeline



4. Build and Push Docker Image:

This job builds the Docker image for the backend and pushes it to Docker Hub.

```
build-and-push-docker:
  runs-on: ubuntu-latest
  needs: build
  steps:
    - name: Checkout Code
      uses: actions/checkout@v2
    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2
    - name: Log in to Docker Hub
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USERNAME }}" --password-stdin
    - name: Build Docker Image
      run: docker build -t ${{ secrets.DOCKER_USERNAME }}/backend:latest ./editor
    - name: Push Docker Image
      run: docker push ${{ secrets.DOCKER_USERNAME }}/backend:latest
```

- **Build Docker Image:** Uses the Dockerfile in the ./editor directory to create a Docker image.
- **Push Docker Image:** Pushes the newly built image to Docker Hub.

11.1 Backend Pipeline

abdayman / backend
Contains: Image • Last pushed: about 5 hours ago

abdayman / frontend
Contains: Image • Last pushed: about 5 hours ago

abdayman/backend (Public)
Last pushed about 5 hours ago
backend (Edit)
WEB SERVERS (Edit)

Docker commands
To push a new tag to this repository:
docker push abdayman/backend:tagname

Tags
This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
latest	🐧	Image	---	5 hours ago

[See all](#)

Automated Builds
Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.
Available with Pro, Team and Business subscriptions. [Read more about automated builds](#).

[Upgrade](#)

build-and-push-docker
succeeded 5 hours ago in 1m 33s

▶ Set up job 1s
▶ Checkout Code 1s
▶ Set up Docker Buildx 9s
Log in to Docker Hub 1s

Run echo "****" | docker login -u "****" --password-stdin
Login Succeeded
WARNING! Your password will be stored unencrypted in /home/runn.../config.json.
Configure a credential helper to remove this warning. See
<https://docs.docker.com/engine/reference/comm...>

Build Docker Image
Push Docker Image
Post Set up Docker Buildx
Post Checkout Code
Complete job

Repository secrets

New repository secret

Name	Last updated
AWS_BACKEND_HOST	2 weeks ago
AWS_FRONTEND_HOST	2 weeks ago
AWS_SSH_KEY	2 weeks ago
DOCKER_PASSWORD	2 weeks ago
DOCKER_USERNAME	2 weeks ago

11.1 Backend Pipeline

5. Deploy Job (Not Working Due to AWS Constraints) Attempts to deploy the Docker image to an AWS EC2 instance.

```
deploy:
  runs-on: ubuntu-latest
  needs: build-and-push-docker # Run after Docker image is pushed
  steps:
    - name: Deploy to AWS EC2
      run: |
        ssh -o StrictHostKeyChecking=no -i ${{ secrets.AWS_EC2_KEY }} ubuntu@${{ secrets.AWS_EC2_IP }}
        'docker pull ${{ secrets.DOCKER_USERNAME }}/backend:latest && docker-compose up -d'
    - name: SSH into Backend EC2
      uses: appleboy/ssh-action@master
      with:
        host: ${{ secrets.AWS_BACKEND_HOST }}
        username: i-0adabf6ef59b693e9
        key: ${{ secrets.AWS_SSH_KEY }}
        script: |
          # Pull the latest backend image
          docker pull ${{ secrets.DOCKER_USERNAME }}/backend:latest

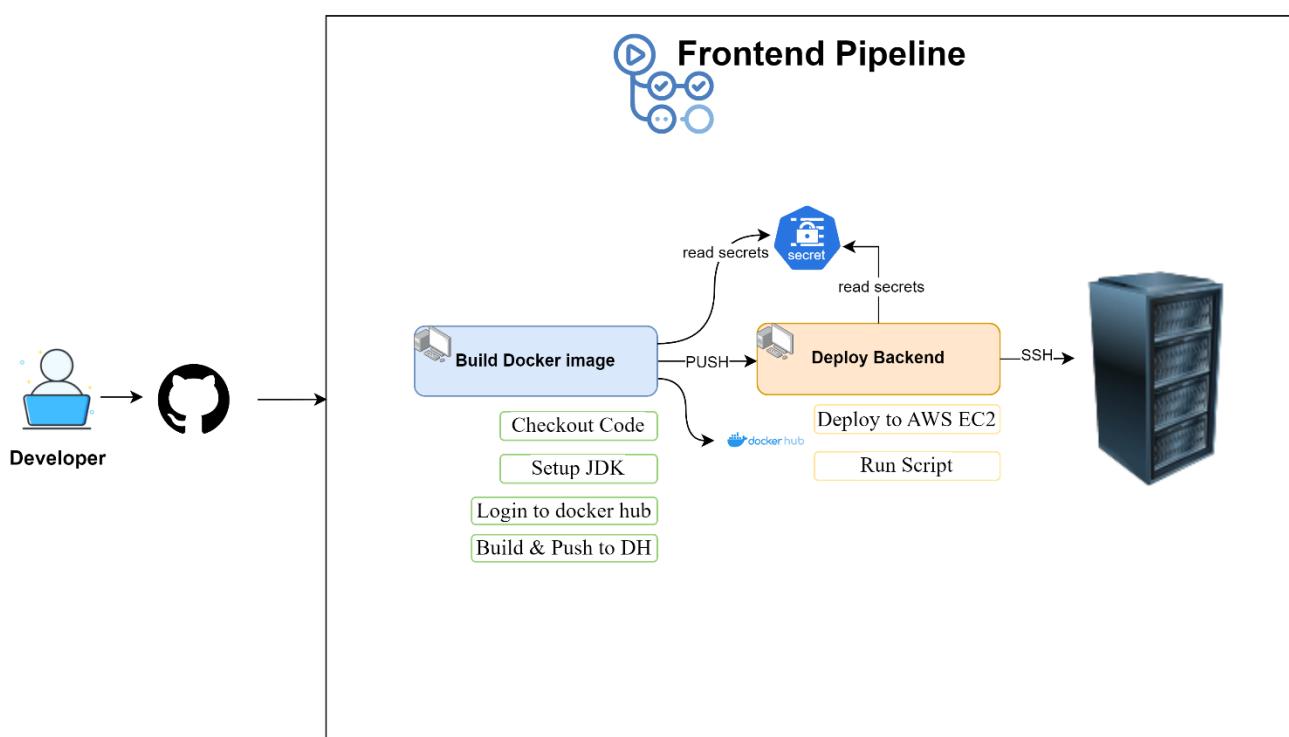
          # Stop and remove existing backend container
          docker stop backend || true
          docker rm backend || true

          # Run MySQL container
          docker run --name mysql -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=mysql_db -p 3306:3306 -d mysql:8.0 || docker start mysql

          # Run MongoDB container
          docker run --name mongo -e MONGO_INITDB_ROOT_USERNAME=devroot -e MONGO_INITDB_ROOT_PASSWORD=devroot -p 27017:27017 -d mongo || docker start mongo

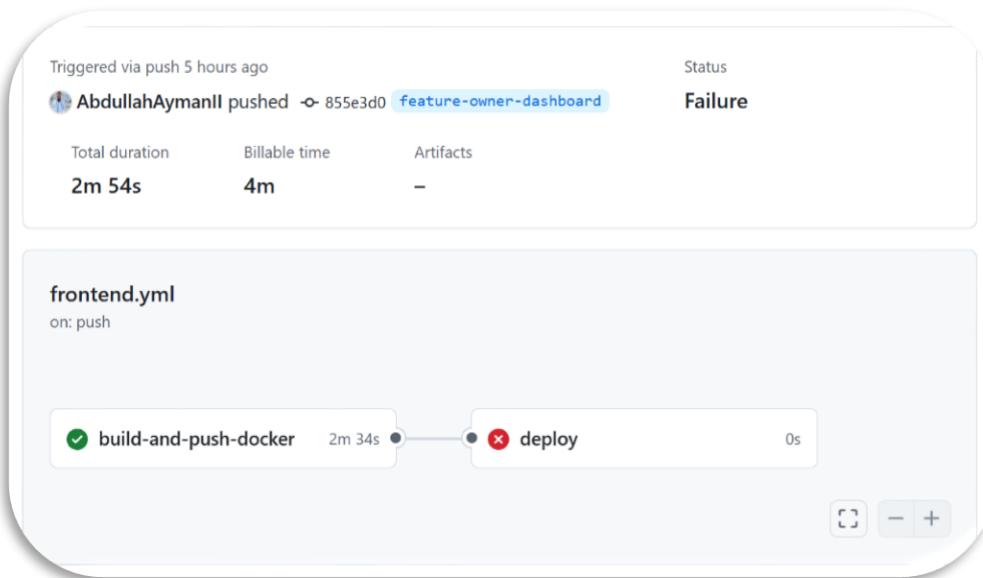
          # Run the backend container
          docker run -d -p 8080:8080 --name backend --link mysql:mysql --link mongo:mongo ${{ secrets.DOCKER_USERNAME }}/backend:latest
```

11.2 Frontend Pipeline



11.2 Frontend Pipeline

This pipeline handles the process for building and deploying the frontend.



1. Build and Push Docker Image

Like the backend, this stage builds the frontend Docker image and pushes it to Docker Hub.

```
build-and-push-docker:
  runs-on: ubuntu-latest
  steps:
    - name: Checkout Code
      uses: actions/checkout@v2
    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2
    - name: Log in to Docker Hub
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "{{ secrets.DOCKER_USERNAME }}" --password-stdin
    - name: Build Docker Image
      run: docker build -t ${{ secrets.DOCKER_USERNAME }}/frontend:latest ./frontend
    - name: Push Docker Image
      run: docker push ${{ secrets.DOCKER_USERNAME }}/frontend:latest
```

- **Build Docker Image:** Creates the frontend Docker image using the ./frontend directory.
- **Push Docker Image:** Pushes the frontend image to Docker Hub.

11.2 Frontend Pipeline

The screenshot shows a GitHub Actions pipeline named "build-and-push-docker" that has succeeded 5 hours ago in 2m 34s. The steps completed are: Set up job, Checkout Code, Set up Docker Buildx, Log in to Docker Hub, Build Docker Image, Push Docker Image, Post Set up Docker Buildx, Post Checkout Code, and Complete job. The Docker Hub repository "abdayman/frontend" contains one tag, "latest". The Docker commands section shows the command "docker push abdayman/frontend:tagname".

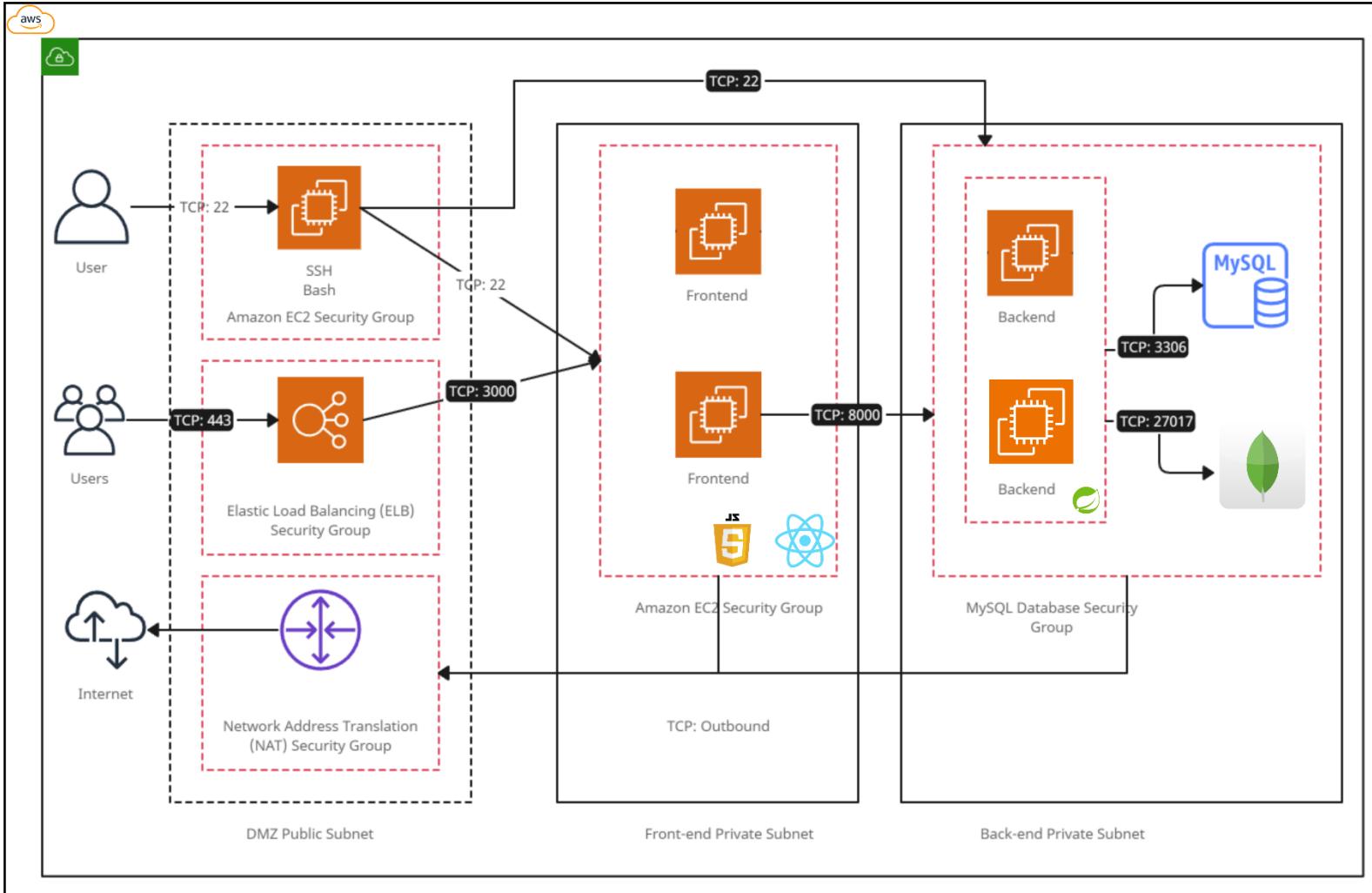
5. Deploy Job (Not Working Due to AWS Constraints) Attempts to deploy the frontend Docker image to an AWS EC2 instance.

```
deploy:
  runs-on: ubuntu-latest
  needs: build-and-push-docker
  steps:
    - name: Deploy to AWS EC2
      run: |
        ssh -o StrictHostKeyChecking=no -i ${{ secrets.AWS_EC2_KEY }} ubuntu@${{ secrets.AWS_EC2_IP }}
        'docker pull ${secrets.DOCKER_USERNAME}/frontend:latest && docker-compose up -d'
    - name: SSH into Frontend EC2
      uses: appleboy/ssh-action@master
      with:
        host: ${secrets.AWS_FRONTEND_HOST}
        username: i-013c7f2dc6a0b6e68
        key: ${secrets.AWS_SSH_KEY}
        script: |
          # Pull the latest frontend image
          docker pull ${secrets.DOCKER_USERNAME}/frontend:latest

          # Stop and remove existing frontend container
          docker stop frontend || true
          docker rm frontend || true

          # Run the frontend container
          docker run -d -p 3000:3000 --name frontend ${secrets.DOCKER_USERNAME}/frontend:latest
```

- **Deploy to AWS:** Pulls the latest frontend image from Docker Hub and runs it on AWS EC2 using docker-compose up. Again, this step doesn't work because of the AWS free tier constraints.



In the AWS network I used the same scalable network infrastructure as that is one in the AWS assignment, and I used **Terraform** to automate the creation process. The architecture includes a Virtual Private Cloud (VPC) with both public and private subnets, designed to host a multi-tier web application. Key components include an Internet Gateway (IGW), NAT Gateway, Security Groups for access control, EC2 instances for compute resources, and an Application Load Balancer (ALB) for traffic distribution. The following sections provide a detailed breakdown of each component and its

12.1 Terraform

I used the terraform to automate the network creation, and the explanations of the scripts used.

1. AWS Provider and VPC

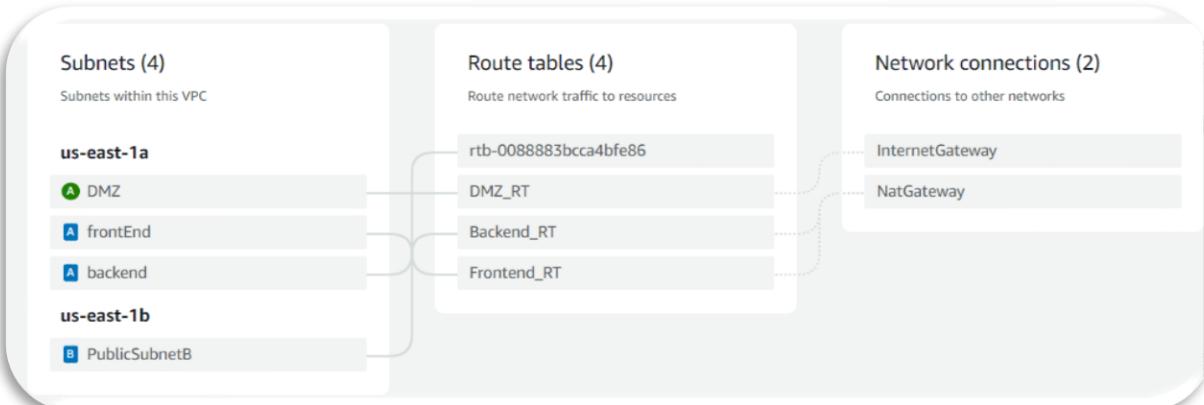
```
provider "aws" {
    region = "us-east-1"
}
resource "aws_vpc" "vpc" {
    cidr_block = "10.0.0.0/16"
    enable_dns_support = true
    enable_dns_hostnames = true
    tags = {
        Name = "collaborativeCodeEditor"
    }
}
```

- **Provider:** Specifies AWS as the cloud provider and sets the region to us-east-1.
- **VPC:** Creates a Virtual Private Cloud (VPC) with the specified CIDR range and DNS support.

2. Subnets

```
resource "aws_subnet" "publicB" { ... }
resource "aws_subnet" "frontend" { ... }
resource "aws_subnet" "backend" { ... }
resource "aws_subnet" "DMZ" {
    vpc_id = aws_vpc.vpc.id
    cidr_block = "10.0.1.0/24"
    map_public_ip_on_launch = true
    availability_zone = "us-east-1a"
    tags = {
        Name = "DMZ"
    }
}
```

- **Subnets:** Defines different subnets for DMZ, public, frontend, and backend within the VPC, each with unique CIDR blocks and availability zones.



12.1 Terraform

3. Internet Gateway and Route Tables



```
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.vpc.id
  tags = {
    Name = "InternetGateway"
  }
}

resource "aws_route_table" "DMZ_RT" {
  vpc_id = aws_vpc.vpc.id
  route {
    cidr_block      = "0.0.0.0/0"
    gateway_id     = aws_internet_gateway.igw.id
  }
  tags = {
    Name = "DMZ_RT"
  }
}
resource "aws_route_table" "Frontend_RT" { ... }
resource "aws_route_table" "Backend_RT" { ... }
```

- **Internet Gateway:** Connects the VPC to the internet.
- **Route Tables:** Defines routes for public (DMZ) and private (Frontend/Backend) traffic, connecting via either the internet gateway or NAT gateway.

4. NAT Gateway and Elastic IP



```
resource "aws_eip" "nat_eip" {
  domain = "vpc"
  tags = {
    Name = "Elastic IP"
  }
}

resource "aws_nat_gateway" "my_nat" {
  allocation_id = aws_eip.nat_eip.id
  subnet_id     = aws_subnet.DMZ.id
  tags = {
    Name = "NatGateway"
  }
}
```

- **Elastic IP:** Allocates a static IP for the NAT gateway.
- **NAT Gateway:** Allows instances in private subnets (Frontend/Backend) to access the internet.

12.1 Terraform

5. Route Table Associations



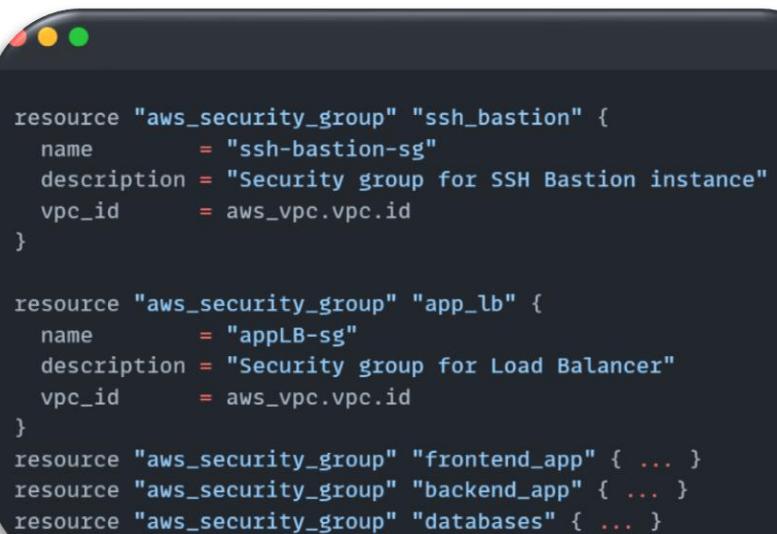
```
resource "aws_route_table_association" "DMZ_subnet_association" {
    subnet_id      = aws_subnet.DMZ.id
    route_table_id = aws_route_table.DMZ_RT.id
}

resource "aws_route_table_association" "Frontend_association" {
    subnet_id      = aws_subnet.frontend.id
    route_table_id = aws_route_table.Frontend_RT.id
}

resource "aws_route_table_association" "Backend_association" {
    subnet_id      = aws_subnet.backend.id
    route_table_id = aws_route_table.Backend_RT.id
}
```

- **Associations:** Links subnets with their corresponding route tables.

6. Security Groups



```
resource "aws_security_group" "ssh_bastion" {
    name          = "ssh-bastion-sg"
    description   = "Security group for SSH Bastion instance"
    vpc_id        = aws_vpc.vpc.id
}

resource "aws_security_group" "app_lb" {
    name          = "appLB-sg"
    description   = "Security group for Load Balancer"
    vpc_id        = aws_vpc.vpc.id
}
resource "aws_security_group" "frontend_app" { ... }
resource "aws_security_group" "backend_app" { ... }
resource "aws_security_group" "databases" { ... }
```

- **Security Groups:** Defines access control for SSH bastion, load balancer, frontend, backend, and database layers.

7. Security Group Rules

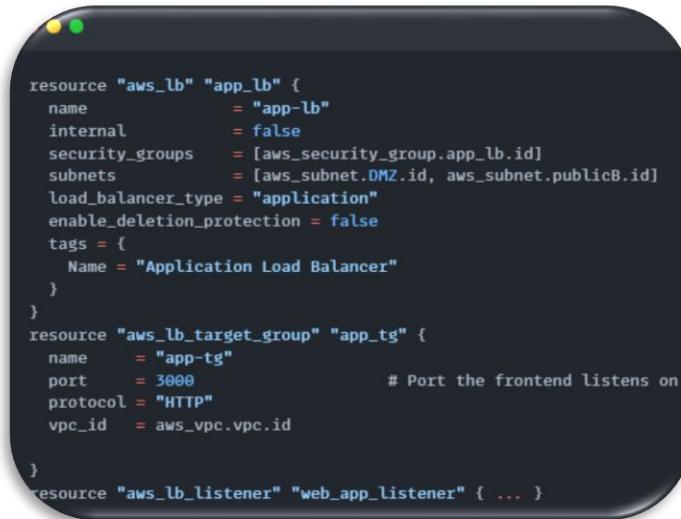


```
resource "aws_security_group_rule" "ssh_bastion_ingress" {
    type          = "ingress"
    from_port     = 22
    to_port       = 22
    protocol      = "tcp"
    cidr_blocks   = ["0.0.0.0/0"]
    security_group_id = aws_security_group.ssh_bastion.id
}
resource "aws_security_group_rule" "app_lb_ingress_http" { ... }
resource "aws_security_group_rule" "Frontend_app_ingress_http" { ... }
```

- **Ingress and Egress Rules:** Control inbound/outbound traffic to instances (SSH, HTTP, DB connections, etc.).

12.1 Terraform

8. Load Balancer and Target Group



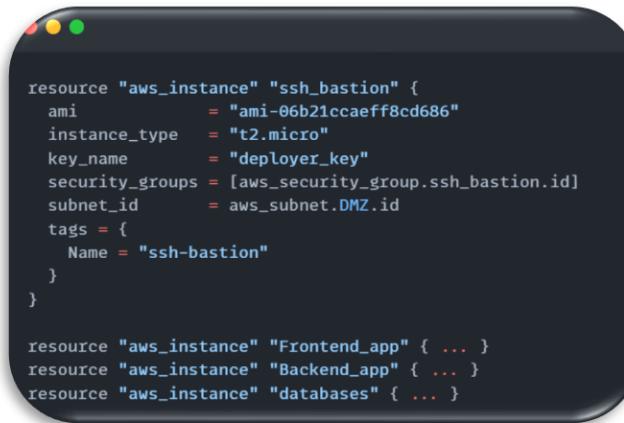
```
resource "aws_lb" "app_lb" {
  name            = "app-lb"
  internal        = false
  security_groups = [aws_security_group.app_lb.id]
  subnets         = [aws_subnet.DMZ.id, aws_subnet.publicB.id]
  load_balancer_type = "application"
  enable_deletion_protection = false
  tags = {
    Name = "Application Load Balancer"
  }
}

resource "aws_lb_target_group" "app_tg" {
  name      = "app-tg"
  port      = 3000          # Port the frontend listens on
  protocol = "HTTP"
  vpc_id   = aws_vpc.vpc.id
}

resource "aws_lb_listener" "web_app_listener" { ... }
```

- **Load Balancer:** Distributes incoming traffic across multiple frontend instances.
- **Target Group:** Groups instances for the load balancer to forward traffic to.
- **Listener:** Listens on port 80 for HTTP requests and forwards them to the target group.

9. EC2 Instances



```
resource "aws_instance" "ssh_bastion" {
  ami           = "ami-06b21ccaeff8cd686"
  instance_type = "t2.micro"
  key_name      = "deployer_key"
  security_groups = [aws_security_group.ssh_bastion.id]
  subnet_id     = aws_subnet.DMZ.id
  tags = {
    Name = "ssh-bastion"
  }
}

resource "aws_instance" "Frontend_app" { ... }
resource "aws_instance" "Backend_app" { ... }
resource "aws_instance" "databases" { ... }
```

- **EC2 Instances:** Defines instances for the SSH bastion, frontend, backend, and database services. Each runs Docker containers for the respective services.

	Name	Instance ID	Instance state	Instance type	Status check	Alarm status
	frontend-inst...	i-0d2b8845f5cebaa77	Running	t2.micro	2/2 checks passed	View alarms +
	databases	i-05eb0ee750eae8270	Running	t2.micro	2/2 checks passed	View alarms +
	ssh-bastion	i-06720b5c30d476a2c	Running	t2.micro	2/2 checks passed	View alarms +
	backend-inst...	i-01a9e4ccd7a2f2b50	Running	t2.micro	2/2 checks passed	View alarms +

- The **t2.micro** is **not enough** to deploy our collaborative code editor that is why I deploy it in docker environment.

Future Work

- Implement SSL for secure WebSocket and HTTP communication to enhance security.
- Improve AWS deployment by creating an S3 bucket for HTTPS certificates, ensuring encrypted data transmission.
- Integrate Kubernetes for better orchestration of Docker containers, enhancing scalability and management of backend and frontend applications.
- Add Pull Request functionality in the collaborative code editor for version control, allowing team collaboration and code reviews.
- Introduce a notification system to keep users updated on room activities and actions.
- Develop a more powerful dashboard for room owners, offering detailed insights and control over the collaborative environment.

ATYPON