ATYPON

# Collaborative Code Editor

Capstone Project

**Done By:**
Lana Alnimreen
**Supervisor:**
Dr.Motasem Aldiab

## Thanks and Appreciation

I want to express my deep gratitude to our supervisor, Dr. Motasem Aldiab, for his patient guidance, motivated support, and important feedback during this training. I really appreciate the time you spent coaching us and providing important feedback, he has been a great supervisor.

## Abstract

This project describes the building of a Collaborative Code Editor, a web-based platform that enables several users to edit and run code together in real time. The platform is built with Java on the back end and React JavaScript on the front end, and it supports real-time communication via WebSocket, allowing for immediate changes among clients. The server component handles user authentication using OAuth 2.0, role-based access control, and complex file management with version control, allowing users to return to previous code states. The system allows code execution in separate Docker containers, allowing for multi-language compatibility across a variety of development environments. Key challenges addressed in including concurrency management, thread safety, secure communication (SSL), and the integration of DevOps practices like CI/CD pipelines for automatic deployment using tools like GitHub Actions. This project demonstrates the application of clean code principles, scalability, and software engineering best practices, offering a powerful solution for collaborative coding environments.

# Contents

## 1. Introduction

The Collaborative Code Editor is an advanced web-based platform that enables real-time, multi-user coding sessions, making it perfect for collaborative programming, code reviews, and collaborative programming settings. The major goal of this project is to create a full-stack application that enables users to write, modify, and execute code collaboratively while using real-time communication technologies such as WebSocket and HTTP. This platform includes advanced software engineering concepts including concurrency, security, version control, and DevOps processes. The project focuses on back-end development in Java (Spring Boot framework), with a front-end created in React JavaScript. The project shows a thorough use of modern web technologies and scalable software solutions by supporting features like role-based access control (RBAC), version control, and multi-language code execution using Docker containers.

## 2. The Assignment

Develop a collaborative code editor with a web interface that allows multiple users to write, edit, and execute code in real time. The project will be developed primarily using Java, with emphasis on back-end development and incorporating advanced software engineering principles. The web interface will support both live updates via WebSockets and manual refresh on request via HTTP. Additionally, students will implement features focusing on clean code, concurrency, scalability, security, and DevOps practices.

## 3. Backend Implementation

I will be talking about my backend in depth of each class and how they work together.

### 3.1 Config

1. `SecurityConfig` class: This class is defining the security settings of Spring Boot application.
    - **CORS Configuration**: Allows requests from the frontend and supports various HTTP methods.
    - **OAuth2 Login**: The application redirects users to a success or failure URL based on their login attempts, facilitating GitHub OAuth authentication.
2. `WebConfig` class: Implements WebMvcConfigurer to customize CORS settings.
    - **CORS Mapping**: Specifies allowed origins, methods, headers, and allows credentials to ensure smooth communication between the frontend and backend.
3. `WebSocketConfig` class:
    - **WebSocket Handlers**: Configures handlers for WebSocket connections at the specified endpoint, allowing real-time messaging and collaboration features.

### 3.2 Controller

1. `AuthController` class:
    - **Login Success**: After a successful login via GitHub, the user details (user ID, username, email) are logged, and the user is saved in the MongoDB database if they are new.
    - **Email Fetching**: If the email is not provided by GitHub, the application fetches it using the GitHub API.

    - **Login Failure**: Captures and logs login failures, returning a response to inform the user.
2. `FileController` class: manages file operations.
    - **File Upload**: Validates user roles (only admins can upload), saves the file to a designated directory, and updates MongoDB with file details and versions.
    - **File Download**: Retrieves the file by its ID and serves it as a downloadable response.
    - **File Deletion**: Checks user permissions before deleting files from both the server and MongoDB, ensuring that unauthorized deletions are prevented.
3. `WebSocketController` class:
    - **Message Handling**: Listens for incoming messages and processes actions such as EDIT, SAVE, EXECUTE, and COMMENT.
    - **Action Handlers**: Each action (edit, save, etc.) has its dedicated method to manage the respective functionality, ensuring that real-time updates are communicated to all connected clients.
4. `RoomController` class: It handles the rooms and files management in the collaborative code editor application and applied multi processes on them.

## 3.3 DTO

`RoleAssignmentRequest` Class: This class serves as a data transfer object (DTO) for assigning roles to users.

- `userId`: The ID of the user to whom a role is being assigned.
- `role`: The role itself (Admin, Editor, Viewer).

## 3.4 Handler

`WebSocketHandler` Class: This class manages WebSocket connections for real-time communication in the application.

## 3.5 Models

The project has several models represent documents in the editor database such as `User`, `Room`, `Comment`, `File`.

## 3.6 Repositories

I have defined several repositories for interacting with MongoDB, which encapsulate various database operations related to comments, execution results, files, rooms, versions, and users.

## 3.7 Services

1. `CodeExecutorImpl` class: It is well-structured for executing code snippets based on the language specified.
2. `ContainerManagement` class: It manages Docker containers for code execution.
3. `PythonExecutor` and `JavaExecutor` classes implement `CodeExecutionStrategy`, each responsible for executing code snippets in their respective languages.

## 4. Frontend Implementation

I will be talking about my frontend in depth of each class and how they work together.

### 4.1 Authentication and Authorization (AuthPage.js):

- handles user registration and login, allowing users to log in via GitHub. Credentials are passed to the backend for authentication using `api.loginUser` or `api.registerUser`.
- Roles like `ADMIN`, `EDITOR`, and `VIEWER` are fetched from the server and dictate what actions users can perform in a room.

### 4.2 Room Management (Rooms.js):

- Users can create new rooms or join existing ones. Each room contains files that can be collaboratively edited.
- The interface separates owned rooms from rooms shared with the user.

### 4.3 File Management and Version Control (RoomDetails.js):

- It allows users to view files in a room, download or delete them, and upload new files.
- Admins can create or fork rooms, clone rooms, or merge files from different rooms.
- Participants and their roles are listed, helping manage collaboration.

### 4.4 Code Editor (CodeEditor.js):

- It provides an interface for writing and editing code. Real-time collaboration is handled via WebSocket, which broadcasts code changes and comments to all participants in the room.
- It also supports file versioning and execution of code, with built-in support for Python and Java.
- Users can add comments to specific lines of code, which are stored and shared via WebSocket updates.

### 4.5 Real-time Collaboration:

Web Sockets are used to handle real-time updates for code editing, comments, and participants. Messages related to file changes, comments, or user actions are broadcasted to all users in the room.

## 5. Multithreading

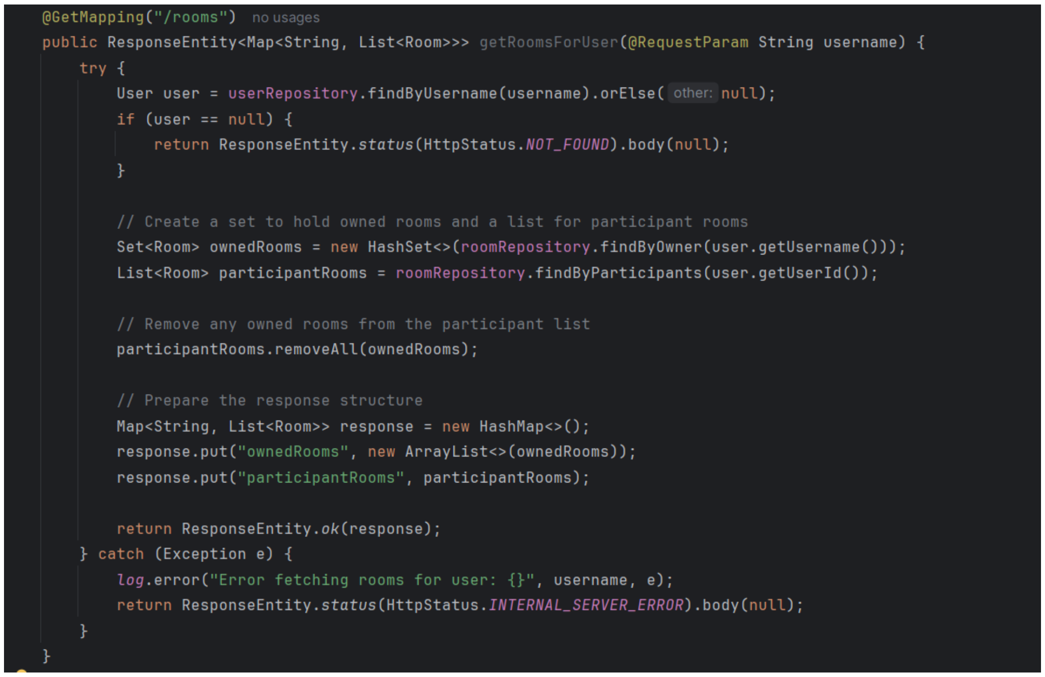
The application supports the execution of code snippets in multiple programming languages (e.g., Python and Java) within isolated Docker containers.

The `JavaExecutor` and `PythonExecutor` services make use of Docker's resource-limiting features (memory and CPU limits), which prevent one user's code execution from hogging all resources, ensuring that multiple users can execute code concurrently without affecting the system's performance.

## 6. Clean Code Principles (Uncle Bob)

My code follows clean code principles, emphasizing readability, simplicity, and maintainability.

- **Meaningful Names**: Classes, methods, and variables are named clearly to convey their goal, such as `userRole`, `participants`, `fileName`, and `executionOutput`. This makes the code self-explanatory and easy to follow.
- **Single Responsibility Principle:** The code follows the Single Responsibility Principle, each class and method are designed to handle a specific responsibility. For example, `FileController` handles file-related operations, `RoomController` manages room-related logic, and `AuthController` is responsible for authentication.
- **Error Handling:** The code uses `try-catch` blocks effectively, ensuring that any exceptions are caught and logged, preventing the program from crashing abruptly. For instance, in the file upload and execution services, errors are properly managed, and user-friendly messages are provided. Here is a sample of using error handling in my code:



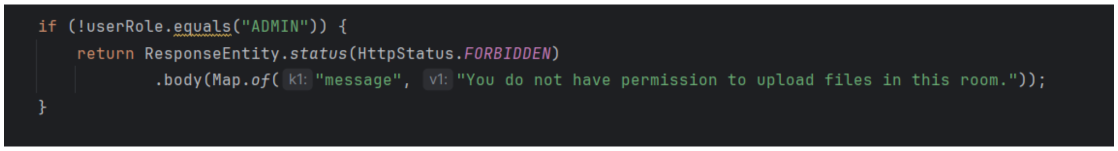
```java
@GetMapping("/rooms")  no usages
public ResponseEntity<Map<String, List<Room>>> getRoomsForUser(@RequestParam String username) {
    try {
        User user = userRepository.findByUsername(username).orElse( other: null);
        if (user == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(null);
        }

        // Create a set to hold owned rooms and a list for participant rooms
        Set<Room> ownedRooms = new HashSet<>(roomRepository.findByOwner(user.getUsername()));
        List<Room> participantRooms = roomRepository.findByParticipants(user.getUserId());

        // Remove any owned rooms from the participant list
        participantRooms.removeAll(ownedRooms);

        // Prepare the response structure
        Map<String, List<Room>> response = new HashMap<>();
        response.put("ownedRooms", new ArrayList<>(ownedRooms));
        response.put("participantRooms", participantRooms);

        return ResponseEntity.ok(response);
    } catch (Exception e) {
        log.error("Error fetching rooms for user: {}", username, e);
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
    }
}
```

- **Consistent formatting**: The code is consistently indented, with logical grouping of related functions. For example, in `CodeEditor.js`, state variables are defined together at the top, followed by lifecycle methods, and then event handlers.
- **DRY (Don't Repeat Yourself):**
  The code follows the DRY principle by avoiding code duplication. For instance, the logic for fetching user roles, participants, and file content is encapsulated in reusable methods, and services like `UserService` and `RoomService` centralize common business logic.

- **Clear Output Messages**: My code provides clear output messages to users in various scenarios, such as success messages for file uploads ("File uploaded and version saved successfully") or error messages when users lack permission ("You do not have permission to upload files in this room"). That could improves the user experience by providing immediate feedback.
- **Initialization in Constructor:** Objects like `Room`, `User`, and `File` initialize their collections (e.g., `participants`, `fileIds`, and `comments`) in their constructors. which ensures the fields are always properly initialized and ready to use without additional setup.

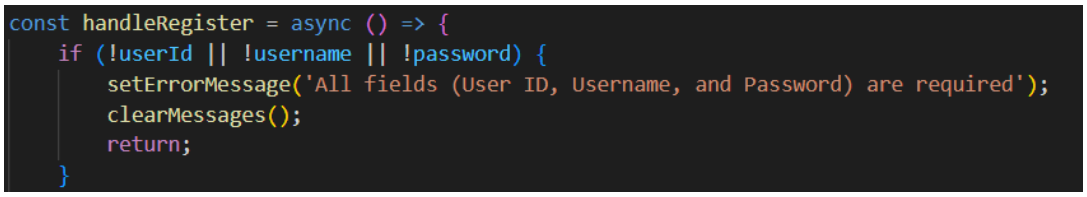
## 7. Efective Java (Joshua Bloch)

- **Item 1: Consider static factory methods instead of constructors:** In the `ContainerManager`, static methods like `terminateContainer` and `readOut` follow the pattern of using utility methods without the need to instantiate the class, making the utility class more flexible and efficient.
- **Item 6: Avoid creating unnecessary objects:** The code avoids redundant object creation by reusing beans via dependency injection. For instance, `FileRepository`, `RoomRepository`, and `UserRepository` are autowired rather than created manually, following the dependency injection pattern.
- **Item 16: In public classes, use accessor methods, not public fields:** I am using accessor methods (getters and setters) in public classes to access private fields.
- **Item 17: Minimize mutability:** Objects like `Room` and `File` are kept mutable, but practices like controlled setters (e.g., `setName`, `setOwner`) help maintain state consistency. This minimizes bugs caused by unexpected changes in object states.
- **Item 47: Prefer Collection to Stream as a return type:** The use of `List` and `Set` to handle collections of files, users, and room participants is effective. For instance, the `room.getFileIds().add(savedFile.getId())` ensures that new files are appended efficiently.
- **Item 49: Check parameters for validity:** I am checking the parameter validity. Like checking if the user has the ADMIN role before allowing them to upload files. This ensures that only authorized users can upload files, validating user permissions.



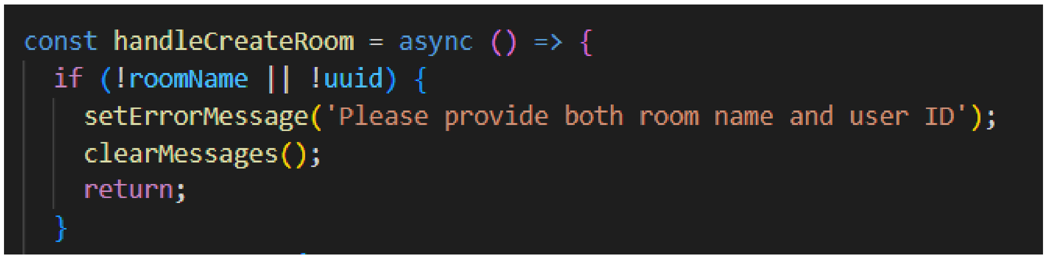
```
if (!userRole.equals("ADMIN")) {
    return ResponseEntity.status(HttpStatus.FORBIDDEN)
            .body(Map.of( k1: "message",  v1: "You do not have permission to upload files in this room."));
}
```

Checks if the username and password fields are populated before attempting to log in.



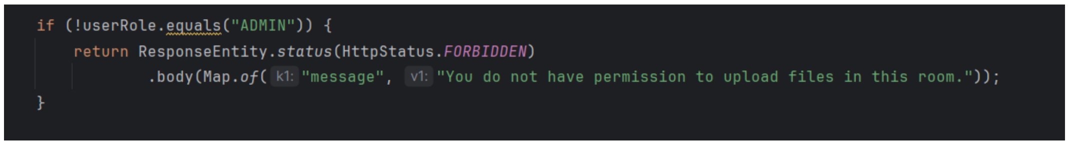
```
const handleRegister = async () => {
    if (!userId || !username || !password) {
        setErrorMessage('All fields (User ID, Username, and Password) are required');
        clearMessages();
        return;
    }
```

The roomName and uuid parameters are validated to ensure they are not empty before creating a room:



```
const handleCreateRoom = async () => {
  if (!roomName || !uuid) {
    setErrorMessage('Please provide both room name and user ID');
    clearMessages();
    return;
  }
```

- **Item 59: Know and use the libraries:** In the frontend, `useEffect`, `useState`, and `useContext` hooks are used effectively to manage component lifecycle events and state, instead of writing custom logic to manage these features.
- **Item 75: Include failure-capture information in detail messages:** error messages are clear and provide context for failures. For example:



```
if (!userRole.equals("ADMIN")) {
    return ResponseEntity.status(HttpStatus.FORBIDDEN)
            .body(Map.of( k1: "message",  v1: "You do not have permission to upload files in this room."));
}
```

## 8. SOLID Principles:

- **Single Responsibility Principle (SRP)**: Each class (such as `CodeEditor.js` handles code editing features, while `RoomDetails.js` manages room and file-related operations.) and service (like `CommentService`, `RoomService`) is responsible for a single part of the application logic, promoting modularity and making the code easier to maintain.
- **Open/Closed Principle (OCP)**: The code is designed to be easily extended without modifying existing code. For example, `FileController` could easily support additional file operations (like renaming) without altering the existing methods.
- **Liskov Substitution Principle (LSP)**: The use of interfaces like `CodeExecutionStrategy` ensures that different executors (Java and Python) can be substituted without affecting the client code.

- **Interface Segregation Principle (ISP)**: The services and handlers are separated logically. For instance, `AuthController` is responsible for handling OAuth logic, while `WebSocketHandler` manages real-time communication, this ensures that no components rely on functionality they don't use.
- **Dependency Inversion Principle (DIP)**: It can be seen in how the service classes depend on interfaces (repositories) rather than concrete implementations.

## 9. Design Patterns:

Several design patterns are implemented across the project:

- **Strategy Pattern**: In the backend, the `CodeExecutorImpl` service uses a strategy pattern by selecting the appropriate `CodeExecutionStrategy` implementation based on the language (`java` or `python`).
- **Observer Pattern**: The WebSocket implementation in `CodeEditor.js` and `RoomDetails.js` follows the observer pattern, where the frontend components listen for updates on code changes, comments, and participants, reacting accordingly.
- **Factory Pattern**: The use of Spring's `@Autowired` and the `@Component` annotations in services and repositories can be considered a factory pattern, as the Spring framework automatically creates instances and injects them into classes.
- **Repository Pattern**: The use of `RoomRepository`, `FileRepository`, and `UserRepository` follows the repository design pattern, abstracting the data access layer and allowing the business logic to remain decoupled from data storage concerns.

## 10.DevOps:

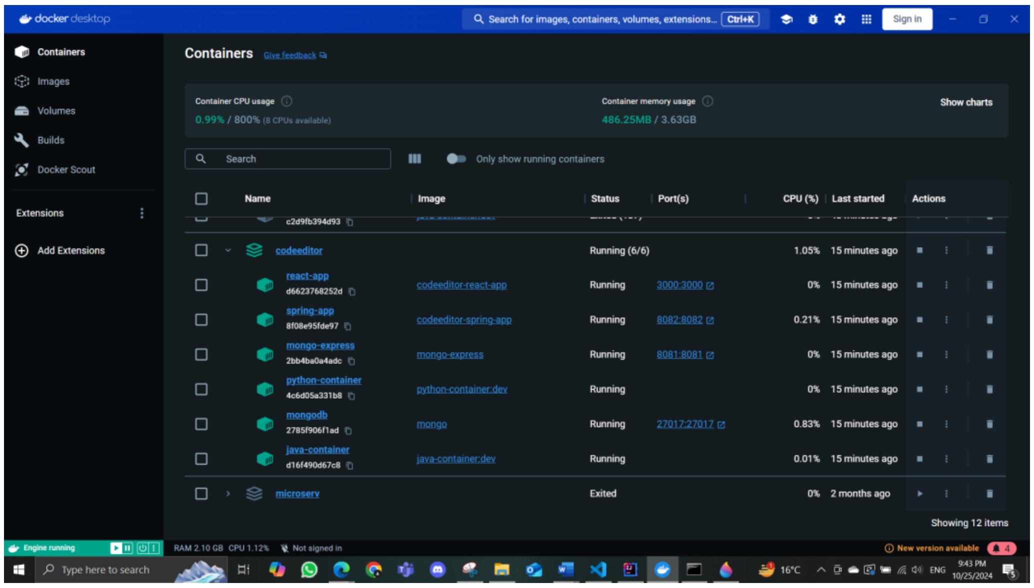
### 10.1 Dockerization for Application Scalability

Dockerizing each component allows the application to run in isolated environments, ensuring that dependencies and configurations for each service do not conflict.

**Docker Compose Configuration:**

- **Java and Python Execution Containers**: Separate containers handle Java and Python code execution, providing the flexibility to scale independently based on usage.
- **MongoDB and Mongo-Express**: MongoDB serves as the main database, and Mongo-Express offers an administrative interface.
- **Spring Backend**: The Spring Boot application handles API requests, running on port 8082.
- **React Frontend**: Serves the frontend application on port 3000.

Docker Compose ensures that each service is defined with dependencies, port mappings, and health checks, maintaining smooth communication between containers.



Containers

Container CPU usage 0.99% / 800% (8 CPUs available)

Container memory usage 486.25MB / 3.63GB

| Name | Image | Status | Port(s) | CPU (%) | Last started |
|---|---|---|---|---|---|
| c2d9fb394d93 | | | | | |
| codeeditor | | Running (6/6) | | 1.05% | 15 minutes ago |
| react-app d66237682 52d | codeeditor-react-app | Running | 3000:3000 | 0% | 15 minutes ago |
| spring-app 8f08e95fde97 | codeeditor-spring-app | Running | 8082:8082 | 0.21% | 15 minutes ago |
| mongo-express 2bb4ba0a4adc | mongo-express | Running | 8081:8081 | 0% | 15 minutes ago |
| python-container 4c6d05a331b8 | python-container:dev | Running | | 0% | 15 minutes ago |
| mongodb 2785f906f1ad | mongo | Running | 27017:27017 | 0.83% | 15 minutes ago |
| java-container d16f490d67c8 | java-container:dev | Running | | 0.01% | 15 minutes ago |
| microserv | | Exited | | 0% | 2 months ago |

Showing 12 items

RAM 2.10 GB CPU 1.12% Not signed in New version available 4

16°C ENG 9:43 PM 10/25/2024

Figure 1: Containers

## 10.2 CI/CD Pipeline with GitHub Actions

This CI/CD pipeline automates building, testing, containerizing, and deploying the application, making the entire process streamlined and consistent.

1. Triggering Events

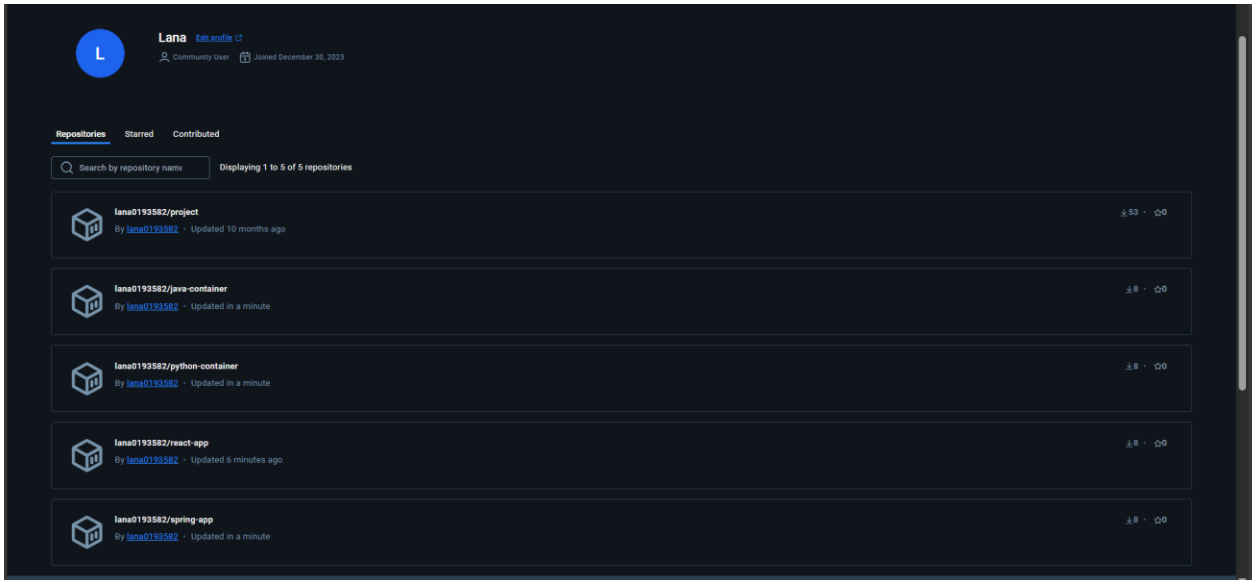
- The pipeline runs on any push or pull request to the main branch, ensuring that only tested code goes into production.

2. Setup and Services

- **MongoDB Service**: The MongoDB service runs on port 27017 and stores data persistently via a Docker volume (mongo-data). This setup mimics the production database environment during tests.
- **Cleanup**: Before building, unnecessary packages and Docker containers are removed to free up disk space.

3. Steps

- **Code Checkout**: Fetches the latest code changes.
- **Java Environment Setup**: Configures Java 17 with the Temurin distribution to support the Spring backend.
- **Build and Verification**: Maven builds the backend (CollabCode), followed by a verification of the .jar file in target to ensure successful packaging.
- **Docker Images**:
  - **Java, Python, Spring, and React Docker Images**: Docker images are built and pushed to Docker Hub using docker/build-push-action. The Spring and React applications are built separately from the backend to maintain modularity.

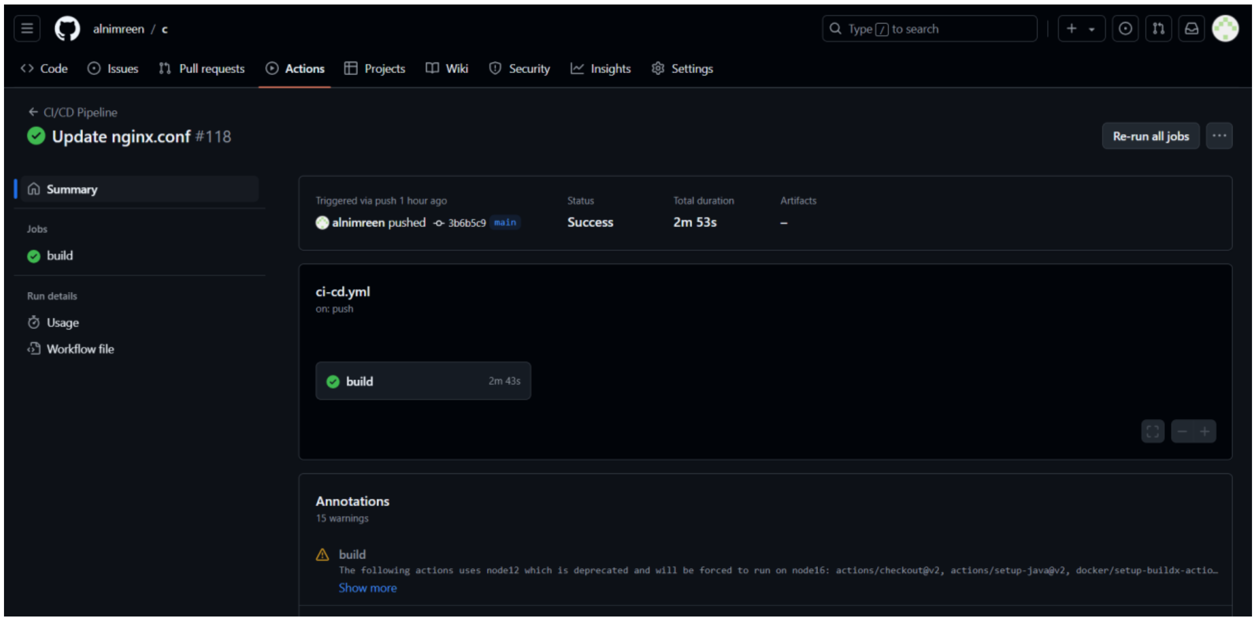


Lana
Community User · Joined December 30, 2023

Displaying 1 to 5 of 5 repositories

| | | |
|---|---|---|
| lana0193582/project | By lana0193582 · Updated 10 months ago | 53 · 0 |
| lana0193582/java-container | By lana0193582 · Updated in a minute | 8 · 0 |
| lana0193582/python-container | By lana0193582 · Updated in a minute | 8 · 0 |
| lana0193582/react-app | By lana0193582 · Updated 6 minutes ago | 8 · 0 |
| lana0193582/spring-app | By lana0193582 · Updated in a minute | 8 · 0 |

Figure 2: Images Pushed to DockerHub in CI/CD



alnimreen / c

CI/CD Pipeline
Update nginx.conf #118

Triggered via push 1 hour ago: alnimreen pushed 3b6b5c9 main
Status: Success
Total duration: 2m 53s
Artifacts: –

ci-cd.yml
on: push

build 2m 43s

Annotations
15 warnings

build
```
The following actions uses node12 which is deprecated and will be forced to run on node16: actions/checkout@v2, actions/setup-java@v2, docker/setup-buildx-actio…
```

Figure 3: Deployment Done Successfully

## 4. Deployment to AWS EC2

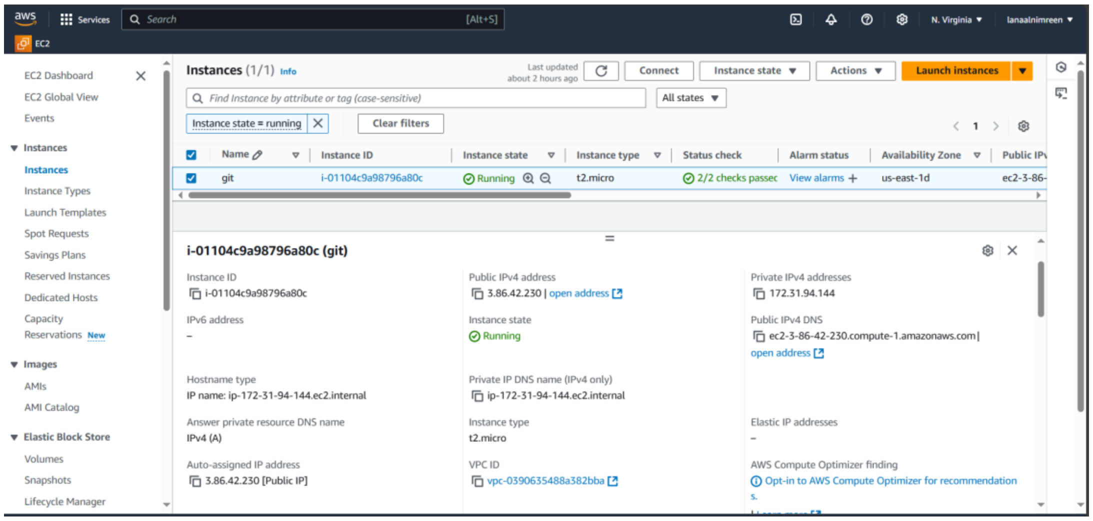
Using SSH, the pipeline connects to the AWS EC2 instance and performs the following actions:

- Clones the project if it doesn't exist or pulls the latest changes.
- Updates Docker images and runs docker-compose up -d to restart containers with the latest version of the application.

## 10.3 Deployment on AWS EC2

Using an AWS EC2 instance, the application is deployed to ensure accessibility and scalability.

1. **EC2 and Docker**:
   - The application is deployed to an EC2 instance for continuous availability.
   - The instance pulls the latest images from Docker Hub and uses docker-compose to orchestrate the containers.
2. **Environment Variables**:
   - AWS credentials for the deployment are securely stored in GitHub Secrets, enabling secure deployment actions.



Instances (1/1) Info

Last updated about 2 hours ago

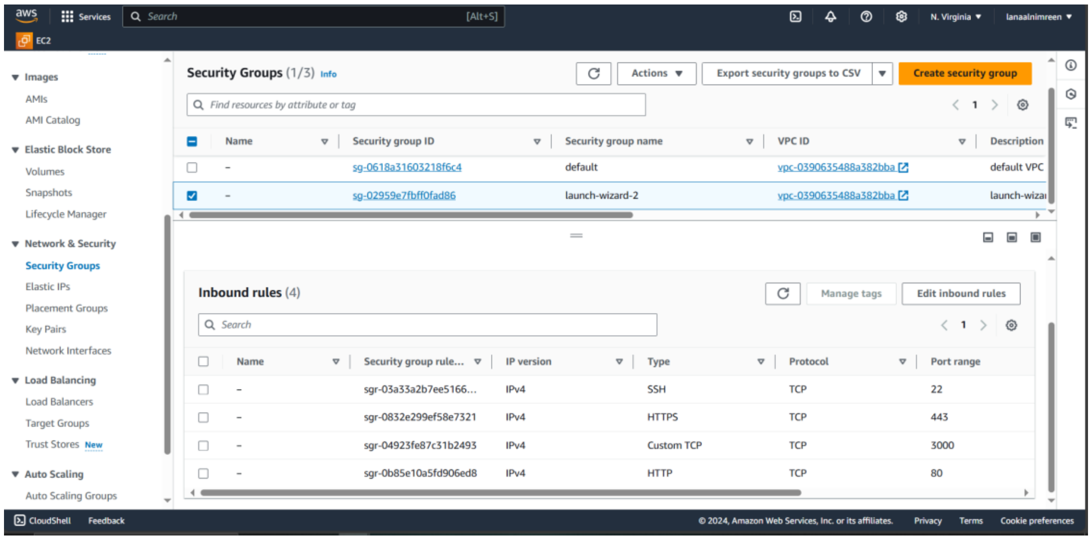
Instance state = running

| Name | Instance ID | Instance state | Instance type | Status check | Alarm status | Availability Zone | Public IPv |
|---|---|---|---|---|---|---|---|
| git | i-01104c9a98796a80c | Running | t2.micro | 2/2 checks passec | View alarms + | us-east-1d | ec2-3-86- |

i-01104c9a98796a80c (git)

| | | |
|---|---|---|
| Instance ID<br>i-01104c9a98796a80c | Public IPv4 address<br>3.86.42.230 \| open address | Private IPv4 addresses<br>172.31.94.144 |
| IPv6 address<br>– | Instance state<br>Running | Public IPv4 DNS<br>ec2-3-86-42-230.compute-1.amazonaws.com \| open address |
| Hostname type<br>IP name: ip-172-31-94-144.ec2.internal | Private IP DNS name (IPv4 only)<br>ip-172-31-94-144.ec2.internal | |
| Answer private resource DNS name<br>IPv4 (A) | Instance type<br>t2.micro | Elastic IP addresses<br>– |
| Auto-assigned IP address<br>3.86.42.230 [Public IP] | VPC ID<br>vpc-0390635488a382bba | AWS Compute Optimizer finding<br>Opt-in to AWS Compute Optimizer for recommendations. |

Figure 4: EC2 Instance in AWS



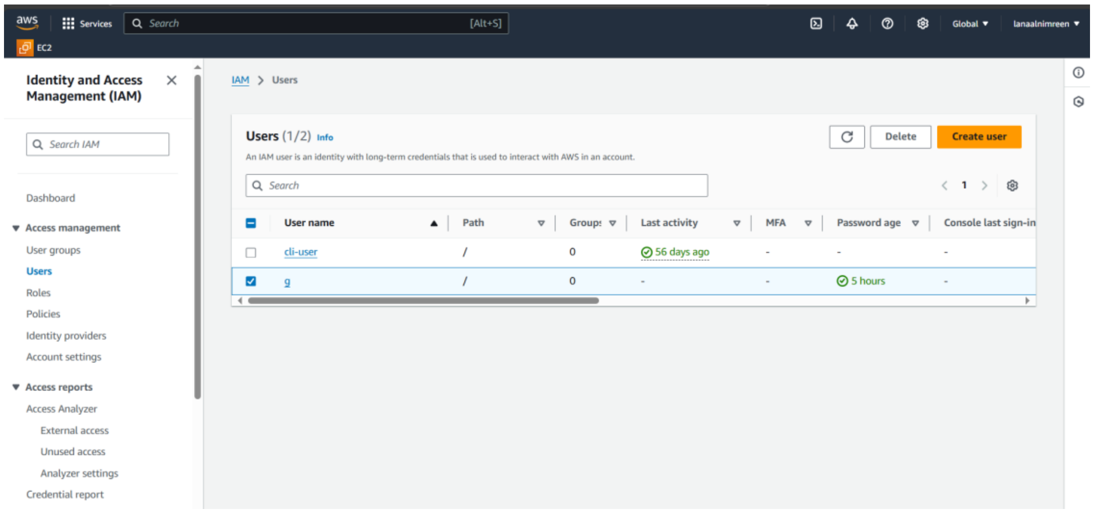
Security Groups (1/3) Info

| Name | Security group ID | Security group name | VPC ID | Description |
|---|---|---|---|---|
| - | sg-0618a31603218f6c4 | default | vpc-0390635488a382bba | default VPC |
| - | sg-02959e7fbff0fad86 | launch-wizard-2 | vpc-0390635488a382bba | launch-wiza |

Inbound rules (4)

| Name | Security group rule... | IP version | Type | Protocol | Port range |
|---|---|---|---|---|---|
| - | sgr-03a33a2b7ee5166... | IPv4 | SSH | TCP | 22 |
| - | sgr-0832e299ef58e7321 | IPv4 | HTTPS | TCP | 443 |
| - | sgr-04923fe87c31b2493 | IPv4 | Custom TCP | TCP | 3000 |
| - | sgr-0b85e10a5fd906ed8 | IPv4 | HTTP | TCP | 80 |

Figure 5: Security Groups used in EC2

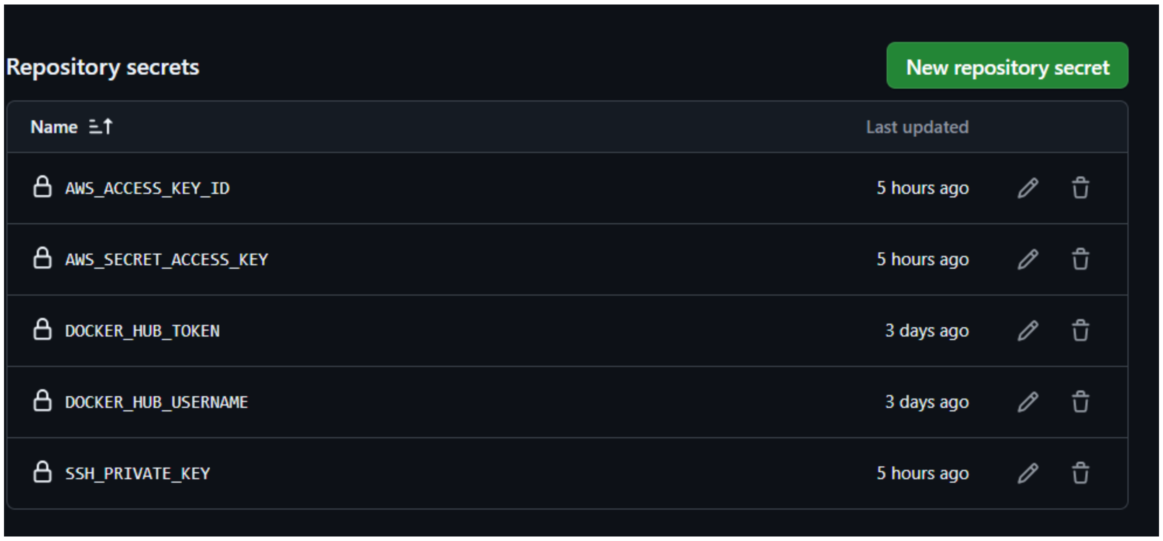


Users (1/2) Info

An IAM user is an identity with long-term credentials that is used to interact with AWS in an account.

| | User name | Path | Groups | Last activity | MFA | Password age | Console last sign-in |
|---|---|---|---|---|---|---|---|
| ☐ | cli-user | / | 0 | 56 days ago | - | - | - |
| ☑ | g | / | 0 | - | - | 5 hours | - |

Figure 6: User used to connect to AWS



Repository secrets

| Name | Last updated |
|---|---|
| AWS_ACCESS_KEY_ID | 5 hours ago |
| AWS_SECRET_ACCESS_KEY | 5 hours ago |
| DOCKER_HUB_TOKEN | 3 days ago |
| DOCKER_HUB_USERNAME | 3 days ago |
| SSH_PRIVATE_KEY | 5 hours ago |

Figure 7: Secrets Variables used on CI/CD

## 11. Testing

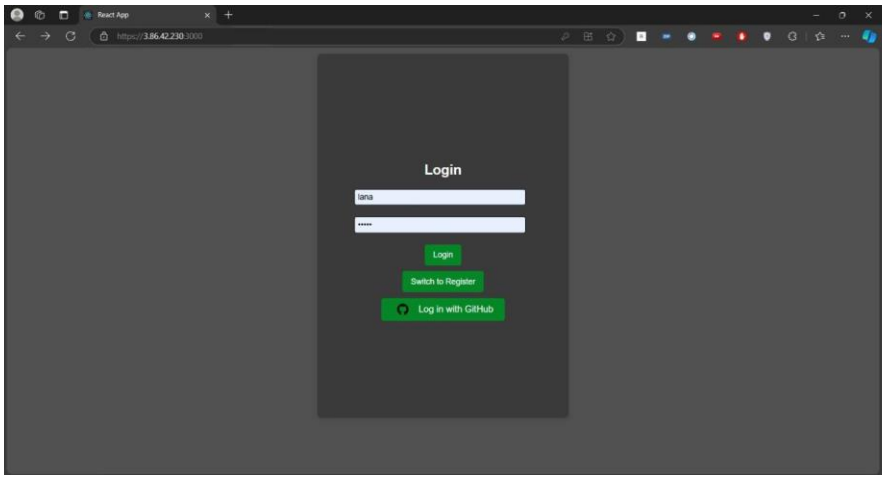


Figure 8:Login Page

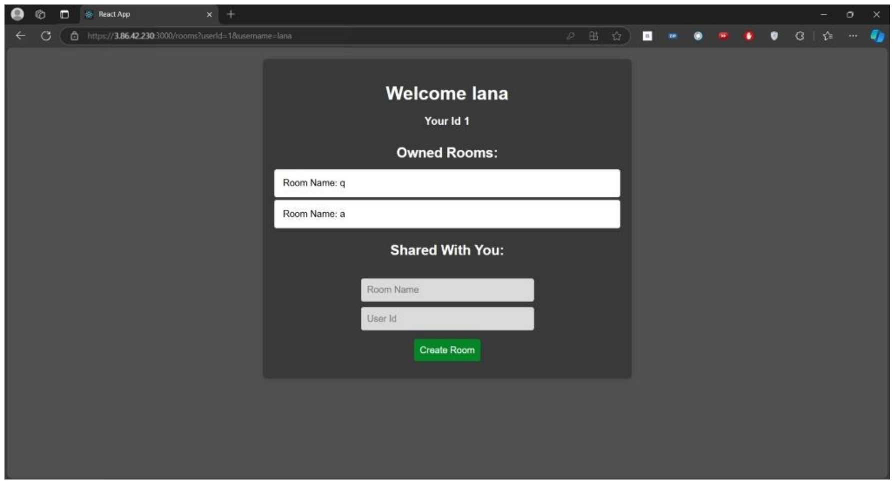


Figure 9:Rooms Page

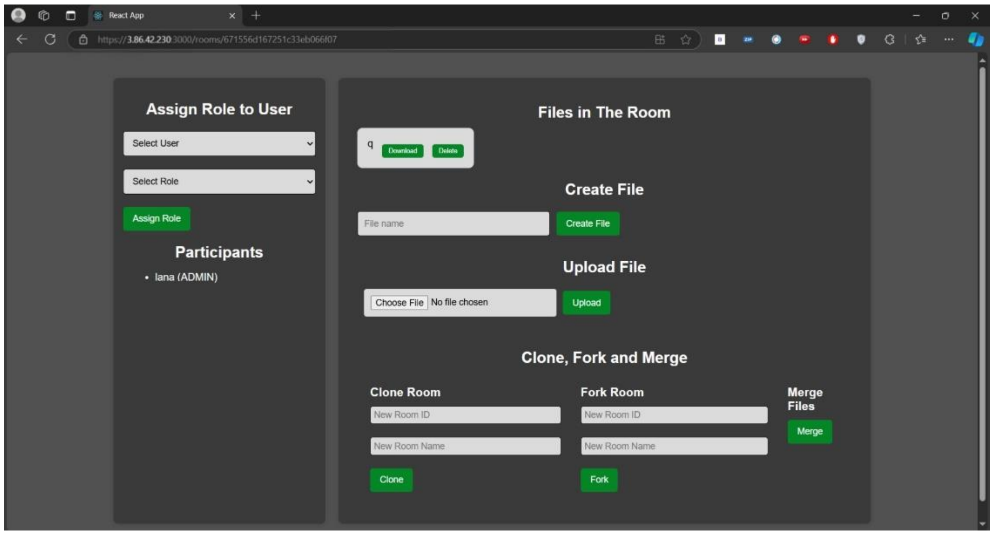
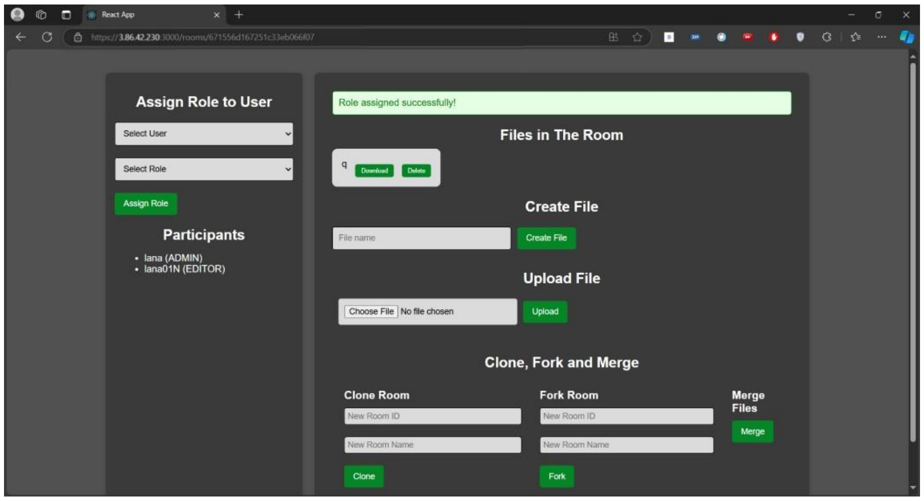
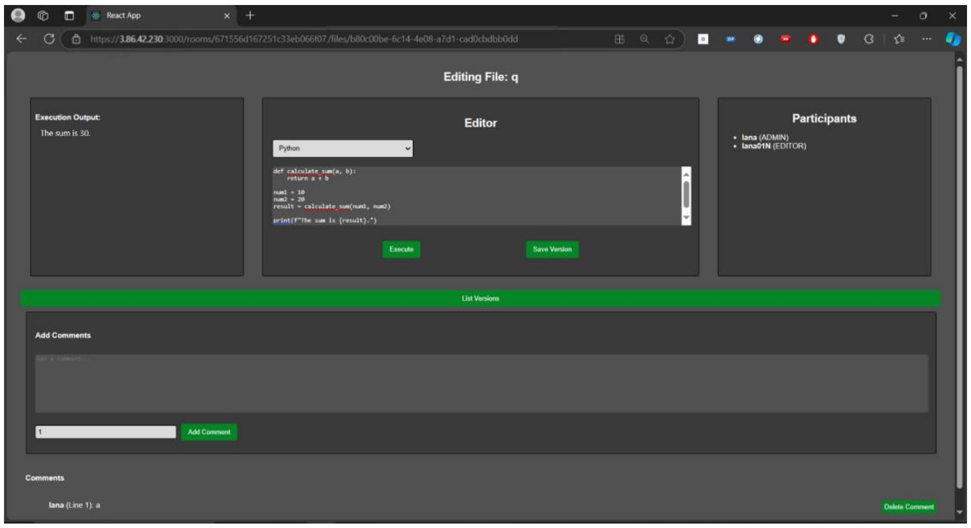


Figure 10:RoomDetails Page



Role assigned successfully!

Figure 11:After Assigning Role



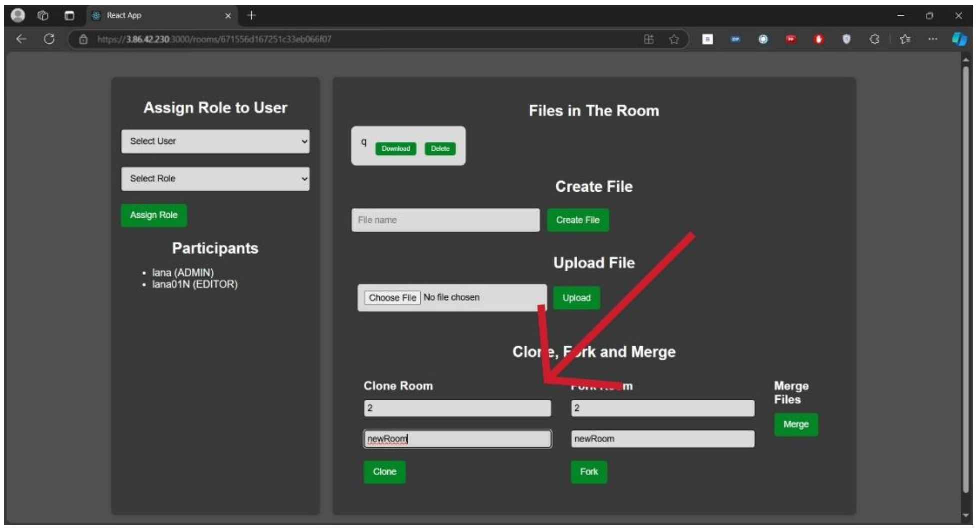
```
The sum is 30.
```

```python
def calculate_sum(a, b):
    return a + b

num1 = 10
num2 = 20
result = calculate_sum(num1, num2)

print(f"The sum is {result}.")
```

- lana (ADMIN)
- lana01N (EDITOR)

lana (Line 1): a

Figure 12: Code Editor Page

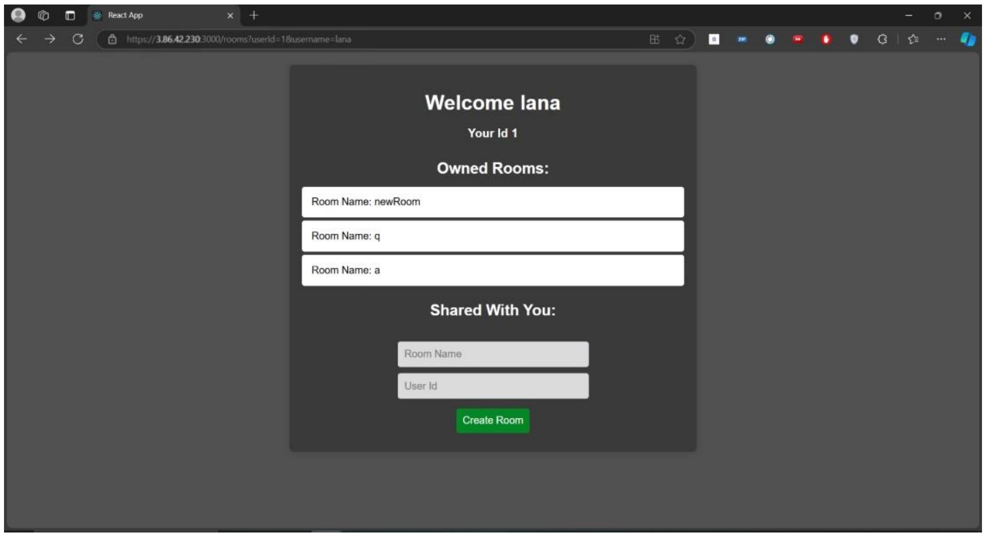


- lana (ADMIN)
- lana01N (EDITOR)

Clone Room: 2, newRoom

Fork Room: 2, newRoom

Figure 13:Clone Room


Welcome lana

Your Id 1

Owned Rooms:

Room Name: newRoom

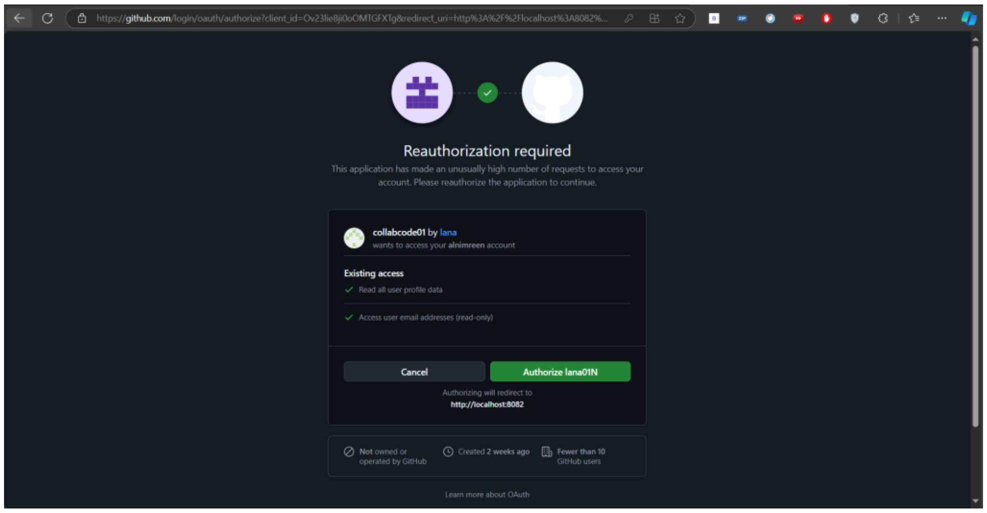
Room Name: q

Room Name: a

Shared With You:

Figure 14:Clone Result

## Login using GitHub


Reauthorization required

This application has made an unusually high number of requests to access your account. Please reauthorize the application to continue.

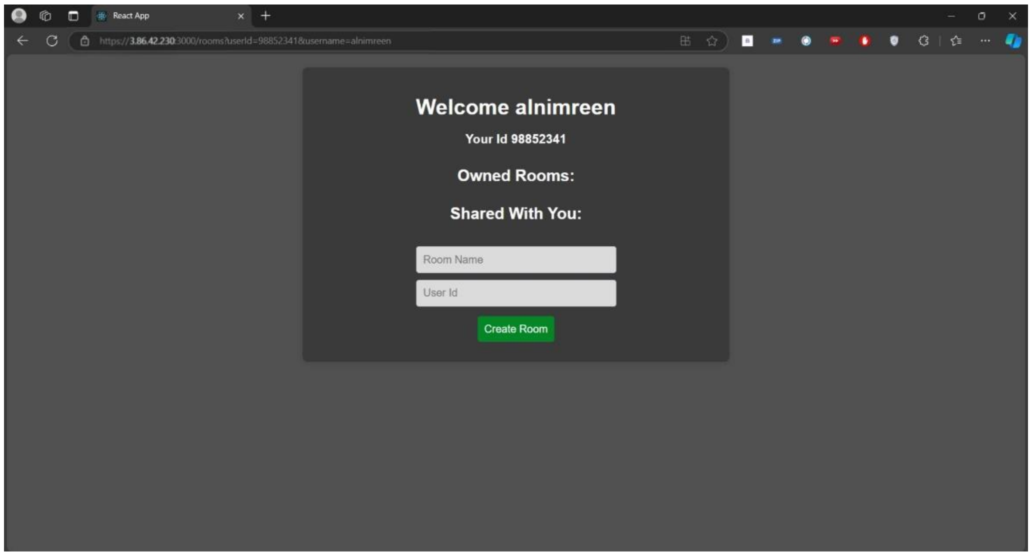
collabcode01 by lana
wants to access your alnimreen account

Existing access
- ✓ Read all user profile data
- ✓ Access user email addresses (read-only)

Authorizing will redirect to
http://localhost:8082

Not owned or operated by GitHub · Created 2 weeks ago · Fewer than 10 GitHub users

Figure 15: Auth Page for GitHub


React App
https://3.86.42.230:3000/rooms?userId=98852341&username=alnimreen

Welcome alnimreen

Your Id 98852341

Owned Rooms:

Shared With You:

Room Name
User Id
Create Room

Figure 16:Login with GitHub