

Task Scheduling System Documentation

Project Author Details:

Dulanji Cooray

Emp No: 105999

Email: dulanji.cooray@axiatadigitallabs.com

Contact: 0772693890

Project Repository Links:

Git Repository Link:

https://github.com/Dulanji1/java-Assessment-SE_CEG_JAV-0004.git

Google Drive Link:

https://drive.google.com/drive/folders/1znZYwvxaiKkZOcA7sjgaN2_LMlx0inzr?usp=sharing

Project Overview

Develop a task automation and scheduling system that allows users to automate repetitive tasks and schedule them for execution. The system should focus on modularity, concurrency, and integration with external APIs.

A Task Automation and Scheduling System is a software solution designed to automatically execute repetitive tasks at specified times or intervals without manual intervention. It allows users to define, manage, and schedule tasks, automating workflows and integrating external services to complete those tasks.

Objective:

Develop a task automation and scheduling system that allows users to automate repetitive tasks and schedule them for execution. The system should focus on modularity, concurrency, and integration with external APIs.

Features:

1. **User Authentication:** Implement secure user login and registration.
2. **Task Management:** Create, edit, and delete tasks with specific actions.
3. **Scheduling:** Schedule tasks to run at specified times or intervals.
4. **Integration:** Connect with external APIs to perform actions (e.g., send emails, fetch data).
5. **Notifications:** Alert users upon task completion or failure.

Technical Requirements:

- **Java 17:** Utilize new features and enhancements in Java 17.
- **Spring Boot:** Use Spring Boot for building the application.
- **Quartz Scheduler:** Implement task scheduling.
- **RESTful APIs:** Create APIs for task management and user interactions.
- **Concurrency:** Use Java's concurrency utilities for handling multiple tasks.
- **Database:** Use a relational database like PostgreSQL for data persistence.
- **Docker:** Containerize the application for easy deployment.

Steps to Implement:

1. Set Up the Project:
 - Create a Spring Boot project using Maven or Gradle.
2. Develop Core Features:
 - **User Authentication:** Implement using Spring Security.
 - **Task Management:** Create APIs to manage tasks and their schedules.
 - **Scheduling:** Use Quartz Scheduler to automate task execution.
3. Integration with External APIs:
 - Implement connectors to interact with external services (e.g., email, data APIs).
4. Implement Concurrency:
 - Use Java's concurrency features to handle multiple task executions efficiently.
5. Database Integration:
 - Use JPA/Hibernate for ORM with PostgreSQL.
 - Ensure proper indexing and query optimization.
6. Containerization:
 - Create a Docker file to containerize the application.
7. Testing:
 - Write unit and integration tests for key components.
8. Documentation:
 - Provide comprehensive

Project Setup Instructions

Prerequisites

- Java 17 (Ensure Java 17 is installed)
- Maven (Project is built using Maven)
- PostgreSQL/MySQL (For database integration, H2 for in-memory)
- SMTP Server for email notifications (e.g., Gmail)

Features Overview

1. User Authentication:

- **Description:**
Secure user login and registration were implemented using **Spring Security**. Passwords are encrypted using industry-standard encryption algorithms, and users are authenticated via RESTful API calls.
- **Key Technologies:** Spring Security, JWT Authentication.

2. Task Management:

- **Description:**
Users can create, edit, and delete tasks. Tasks include specific actions like API calls, sending emails, or internal system processes. Each task has associated metadata, such as scheduled time, task status, and user information.
- **Key Technologies:** Spring Boot, Hibernate (JPA) for data management, PostgreSQL.

3. Scheduling:

- **Description:**
The system uses **Quartz Scheduler** to manage and execute tasks at specified times or intervals. The scheduling module supports both one-time and recurring tasks, allowing for flexibility in task automation.
- **Key Technologies:** Quartz Scheduler, Spring Boot.

4. Integration with External APIs:

- **Description:**
Tasks can be integrated with external APIs, such as sending emails through third-party email services (e.g., SendGrid) or fetching data from other services like weather APIs or stock market data.
- **Key Technologies:** RESTful API, Spring RestTemplate.

5. Notifications:

- **Description:**
The system sends notifications to users upon task completion or failure. Notifications are delivered via email or displayed on the user interface, depending on user preferences. This ensures users are always informed about the status of their tasks.
- **Key Technologies:** Email integration, real-time notification

6. Concurrency:

Definition: Concurrency is the ability of a system to perform multiple tasks at the same time. It allows different parts of a program to run simultaneously, improving efficiency and performance, especially on multi-core processors.

Example: In a chat application, you might send messages while at the same time receiving new messages from others, both happening concurrently.

The system is designed to handle multiple tasks running concurrently using Java's concurrency utilities. Each task operates in its own thread, enabling efficient and simultaneous execution of various scheduled jobs. By leveraging Java 17's advanced concurrency tools, the system can manage task execution states such as running, failed, and completed effectively.

In addition, asynchronous task execution is implemented using Spring's `@Async` annotation, allowing certain tasks to run independently without blocking the main thread. This ensures optimal performance for tasks that involve waiting for external resources (e.g., API calls or database transactions).

Transactional management is used to ensure that tasks requiring database updates are handled in an atomic way, guaranteeing data consistency even when multiple tasks are processed concurrently. In the case of failure, the transaction is rolled back to maintain data integrity.

- Key Technologies:
 - Java 17 Concurrency Utilities:
Utilized features like Executor Service and CompletableFuture to manage and run tasks asynchronously and concurrently.
 - Threads:
Each task runs in its own thread, which allows parallel execution of multiple tasks.
 - Spring's `@Async` Annotation:
Used to perform asynchronous operations, ensuring that long-running tasks do not block other tasks.
 - Transaction Management:
Integrated Spring's `@Transactional` annotation to handle database operations within tasks, ensuring ACID properties and atomic updates.

Transactional refers to a way of managing database operations in a way that ensures they are completed successfully or not at all. In Spring, you use the `@Transactional` annotation to group these operations, making sure they follow the ACID properties

(Atomicity, Consistency, Isolation, Durability). This means that if one part of the operation fails, all changes can be rolled back, keeping the database in a consistent state.

- **Error Handling:**
In case of failures, proper exception handling mechanisms are employed to log errors and trigger notifications.

7. Modularity:

Modularity refers to breaking down a system into separate, interchangeable components or modules. Each module has its own specific functionality, making the code easier to maintain, update, and understand.

- **Example:** In a web application, one module might handle user authentication, another might manage payments, and another might handle data storage.

8. Integration with External APIs:

- **Definition:** Integration with external APIs means connecting your software to other external services or systems via APIs (Application Programming Interfaces). APIs allow different software systems to communicate and exchange data.

Project Structure

A **Spring Boot project structure** is organized in a way that separates concerns, making it easy to manage and scale the application. Here's a simple breakdown of the common folders and their purposes:

1. `src/main/java`:

- Contains the main Java source code for your application.
- **Key components:**
 - **Controller:** Handles HTTP requests (API endpoints).
 - **Service:** Contains business logic.
 - **Repository:** Interacts with the database (using JPA or other methods).
 - **Model/Entity:** Represents the data structure or database tables.
 - **Config:** Custom configurations (e.g., security, scheduling).

2. `src/main/resources`:

- Stores non-code resources like configurations.
- **application.properties** or **application.yml**: Configuration for database, logging, security, etc.
- **static/**: Static files like HTML, CSS, JavaScript (for web apps).
- **templates/**: Thymeleaf or other template engine files (for server-rendered views).

3. `src/test/java`:

- Contains test code for unit and integration tests.

4. `pom.xml` (or `build.gradle`):

- Configuration file for Maven (or Gradle), managing project dependencies, plugins, and build processes.

```

task-automation-scheduling-system/
├─ src/
│  ├─ main/
│  │  ├─ java/
│  │  │  └─ com/
│  │  │     └─ example/
│  │  │        └─ scheduling/
│  │  │           ├─ config/           // Configuration classes (e.g., SecurityConfig)
│  │  │           ├─ controller/      // REST controllers
│  │  │           ├─ dto/             // Data Transfer Objects (DTOs)
│  │  │           ├─ entity/          // Entity classes for database tables
│  │  │           ├─ exception/       // Custom exceptions and error handling
│  │  │           ├─ repository/      // Repository interfaces for database access
│  │  │           ├─ scheduler/       // Classes for task scheduling (Quartz)
│  │  │           ├─ service/         // Service layer for business logic
│  │  │           └─ util/            // Utility classes (e.g., JWT utility)
│  │  └─ resources/
│  │     ├─ application.properties    // Application configuration properties
│  │     └─ static/                   // Static resources (if any, e.g., CSS, JS)
│  └─ Dockerfile                      // Dockerfile for containerization
├─ test/
│  └─ java/
│     └─ com/
│        └─ example/
│           └─ scheduling/
│              ├─ controller/         // Tests for controllers
│              ├─ service/            // Tests for services
│              └─ repository/         // Tests for repositories
└─ resources/                        // Test resources (if needed)
└─ pom.xml or build.gradle           // Maven or Gradle build file

```

The screenshot shows an IDE with the following components:

- Project Explorer:** Displays the project structure:
 - src
 - main
 - java
 - com.example.scheduling.system
 - config
 - controller
 - dto
 - entity
 - repository
 - scheduler
 - service
 - util
 - SchedulingSystemApplication
 - resources
 - static
 - templates

- Code Editor:** Shows the code for `SchedulingSystemApplication.java`:


```

11 import java.util.concurrent.Executor;
12
13 @SpringBootApplication(scanBasePackages = "com.example.scheduling.system")
14 @EnableAsync
15 @EnableScheduling
16 public class SchedulingSystemApplication {
17     public static void main(String[] args) {
18         SpringApplication.run(SchedulingSystemApplication.class, args);
19     }
20
21     @Bean
22     public ModelMapper modelMapper() {
23         return new ModelMapper();
24     }
25
26     @Bean
27     public Executor taskExecutor() {
28         ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();

```
- Run Console:** Shows the application startup logs:


```

2024-09-20T16:00:41.083+05:30 DEBUG 19376 --- [scheduling-system] [ restartedMain] o.s.boot.devtools.restart.Restarter : Starting applica
2024-09-20T16:00:41.084+05:30 DEBUG 19376 --- [scheduling-system] [ restartedMain] o.s.b.a.ApplicationAvailabilityBean : Application avai
2024-09-20T16:00:46.023+05:30 DEBUG 19376 --- [scheduling-system] [ _ClusterManager] o.s.s.quartz.LocalDataSourceJobStore : ClusterManager:
2024-09-20T16:00:51.023+05:30 DEBUG 19376 --- [scheduling-system] [ _ClusterManager] o.s.s.quartz.LocalDataSourceJobStore : ClusterManager:
2024-09-20T16:00:56.023+05:30 DEBUG 19376 --- [scheduling-system] [ _ClusterManager] o.s.s.quartz.LocalDataSourceJobStore : ClusterManager:

```


API Documentation

Task API Endpoints

Ex: Create a Task

Endpoint: POST /api/tasks

Request Body:

```
{  
  "name": "Sample Task",  
  "description": "This is a sample task.",  
  "scheduledTime": "2024-10-01T15:00:00",  
  "status": "PENDING",  
  "userId": 1  
}
```

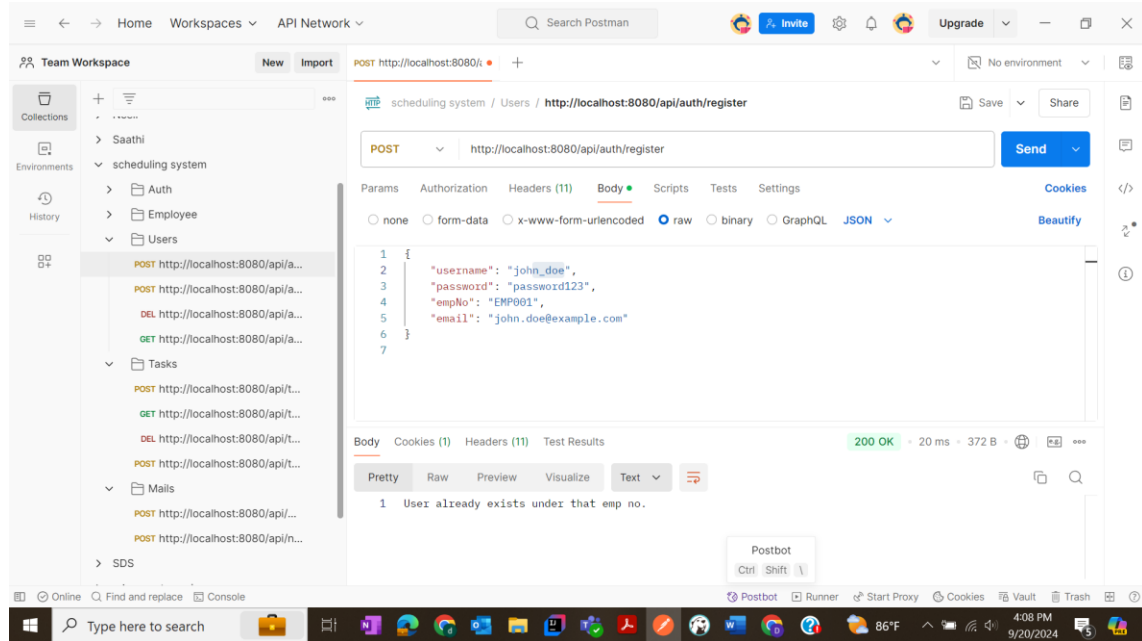
Response:

- 200 OK: Task created successfully.
- 400 Bad Request: Invalid input data.

Postman API Collection Access Link:

<https://orange-escape-442612.postman.co/workspace/Team-Workspace~6efaddfd-2c57-44a5-8301-24939465f0a8/collection/23512729-c47c92ca-c97c-4086-b7df-52c6d7bf0281?action=share&creator=23512729>

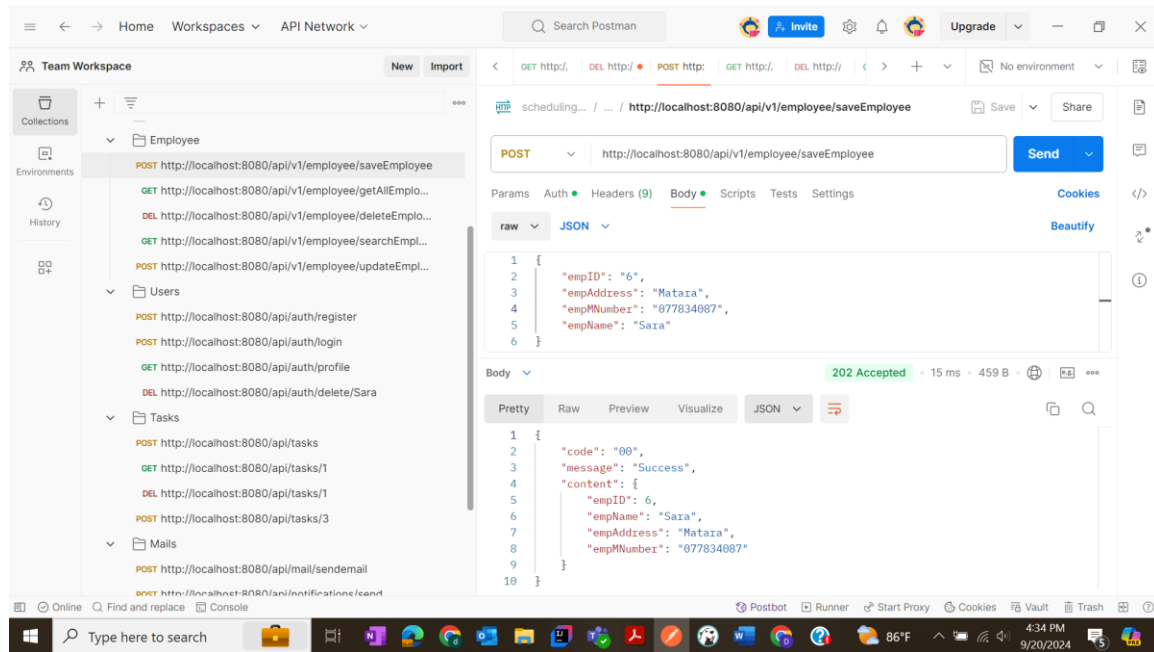
Postman API Samples



Postman interface showing a POST request to `http://localhost:8080/api/auth/register`. The request body is a JSON object:

```
1 {
2   "username": "john_doe",
3   "password": "password123",
4   "empNo": "EMP001",
5   "email": "john.doe@example.com"
6 }
7
```

The response is `200 OK` with a message: `1 User already exists under that emp no.`



Postman interface showing a POST request to `http://localhost:8080/api/v1/employee/saveEmployee`. The request body is a JSON object:

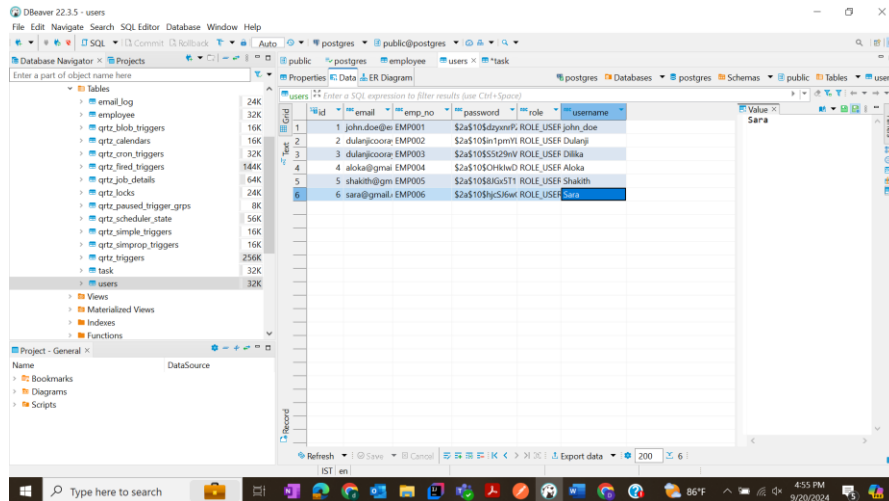
```
1 {
2   "empID": "6",
3   "empAddress": "Matara",
4   "empMNumber": "077834087",
5   "empName": "Sara"
6 }
```

The response is `202 Accepted` with a detailed JSON object:

```
1 {
2   "code": "00",
3   "message": "Success",
4   "content": {
5     "empID": 6,
6     "empName": "Sara",
7     "empAddress": "Matara",
8     "empMNumber": "077834087"
9   }
10 }
```

Data Base

The system's database is designed using **Progress SQL** to store and manage all critical data related to tasks, users, scheduling, and task execution history. Progress SQL is chosen for its robustness in handling high-volume transactional operations, its scalability, and strong support for real-time data processing. The database is optimized for task scheduling and automation, ensuring fast data retrieval and efficient transaction handling during task executions.



Key Database Features:

- Progress SQL Transactions:

Transactions ensure that database operations, especially task creation, scheduling, and execution logging, are atomic, consistent, isolated, and durable (ACID properties). This prevents partial task executions from being recorded, maintaining the integrity of the scheduling system.

- Indexing:

Indexes are created on key fields like taskID, userID, and scheduledTime to enhance query performance, particularly for fetching tasks that are due for execution or retrieving user-specific task histories.

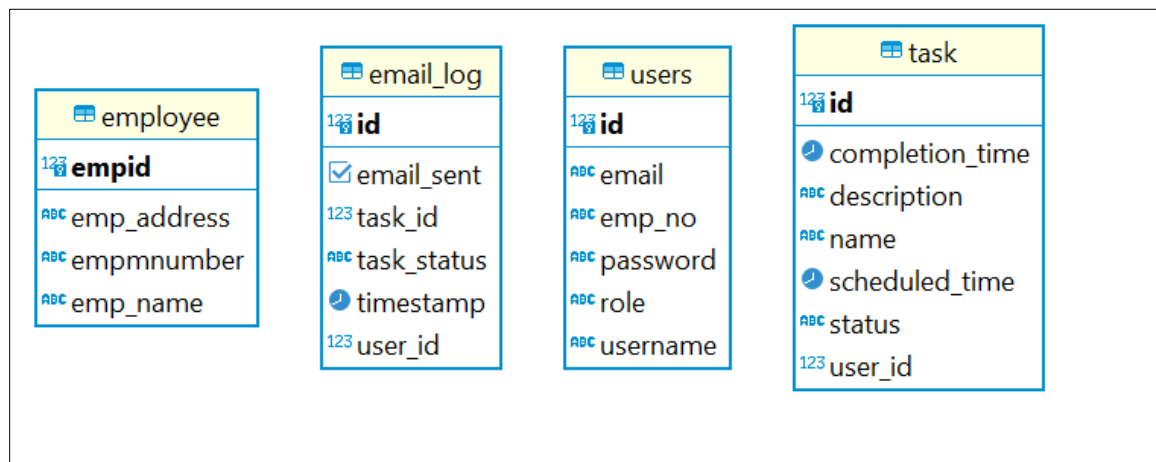
- Data Consistency:

Referential integrity is enforced using foreign key constraints between tables, ensuring consistent relationships between users, tasks, and notifications.

- Optimized Queries:

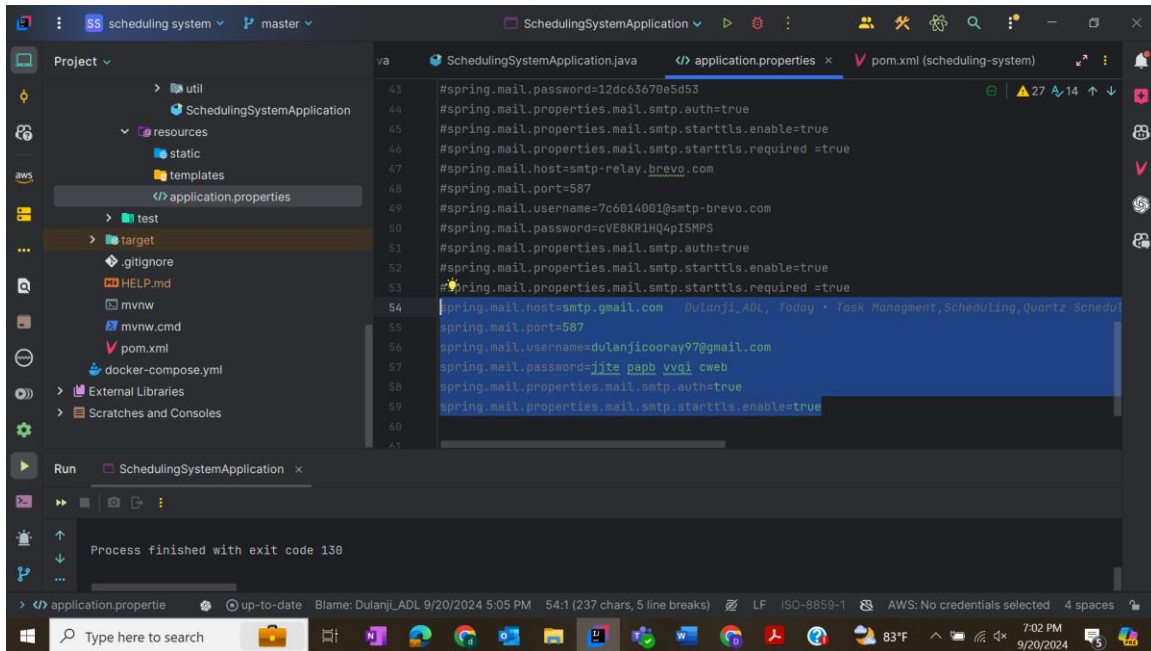
Queries are optimized to ensure fast and efficient task retrieval, especially for concurrent task executions. The system handles high-load scenarios, ensuring that multiple tasks can be managed and executed without bottlenecks.

ER Diagram



Notification

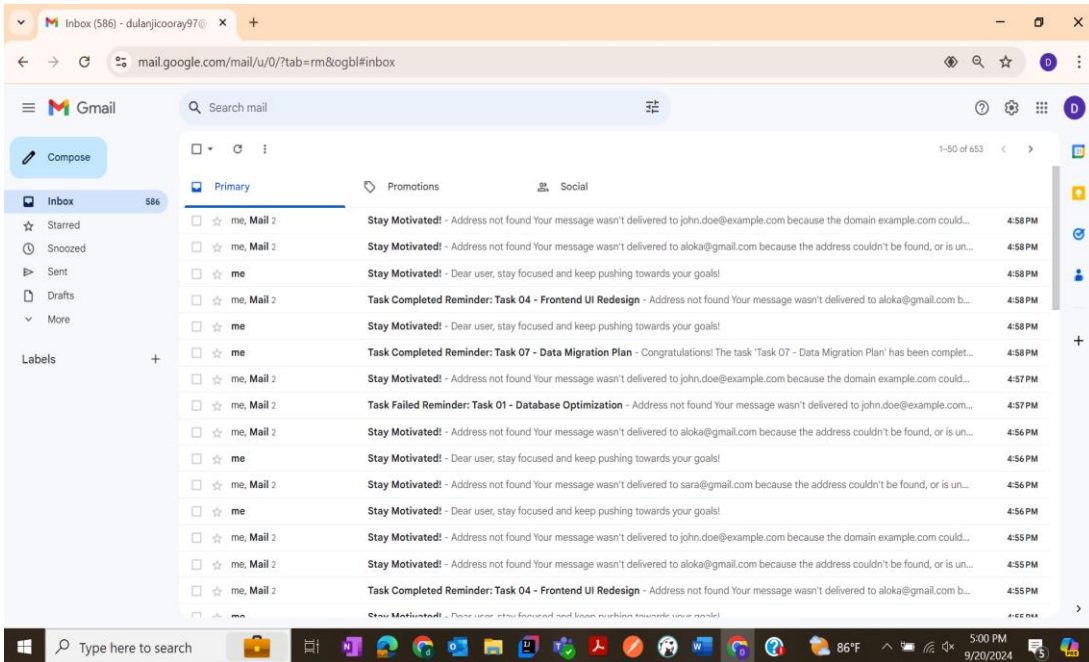
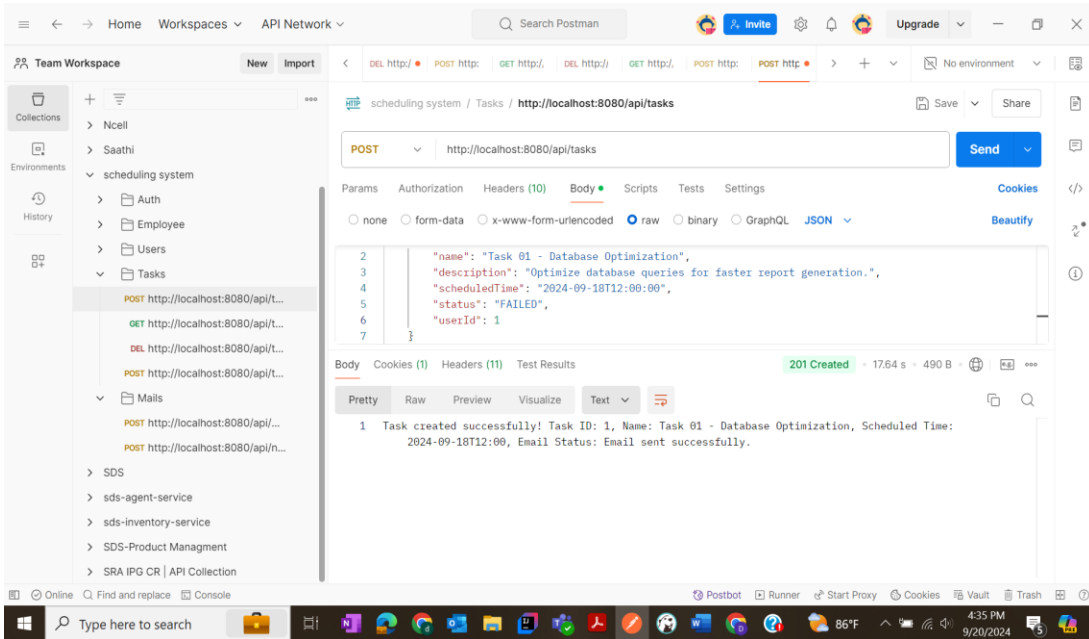
When tasks are completed (either successfully or with failure), the system automatically sends email notifications to users using Gmail. This keeps users informed about the status of their scheduled tasks.

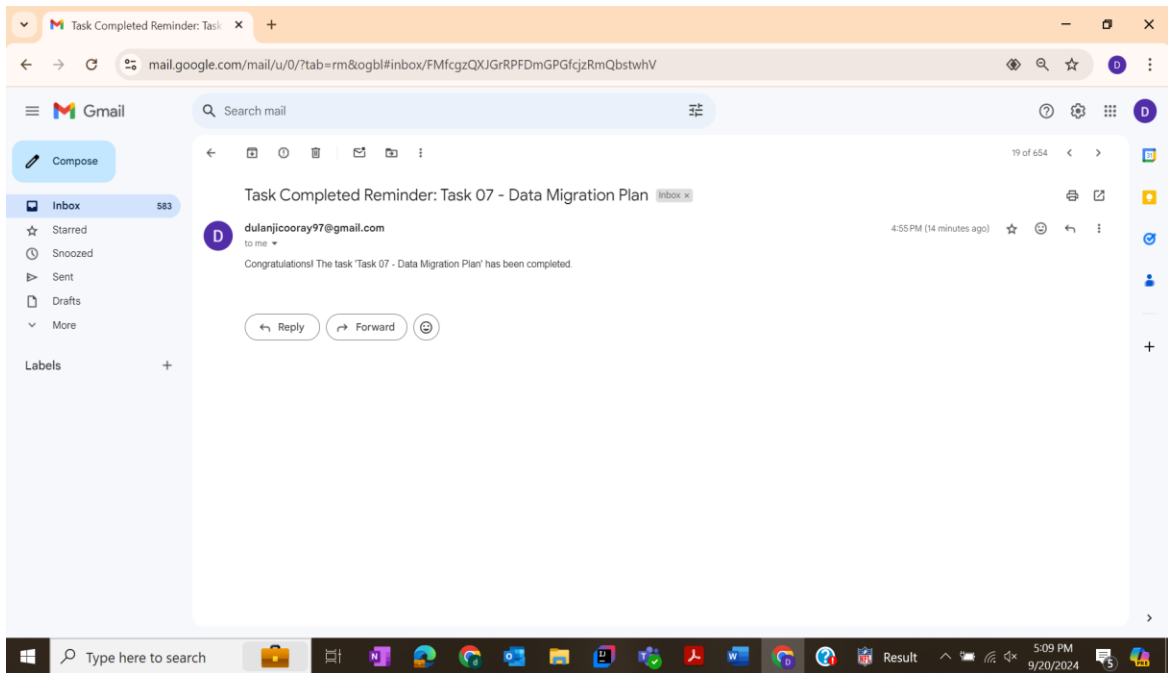
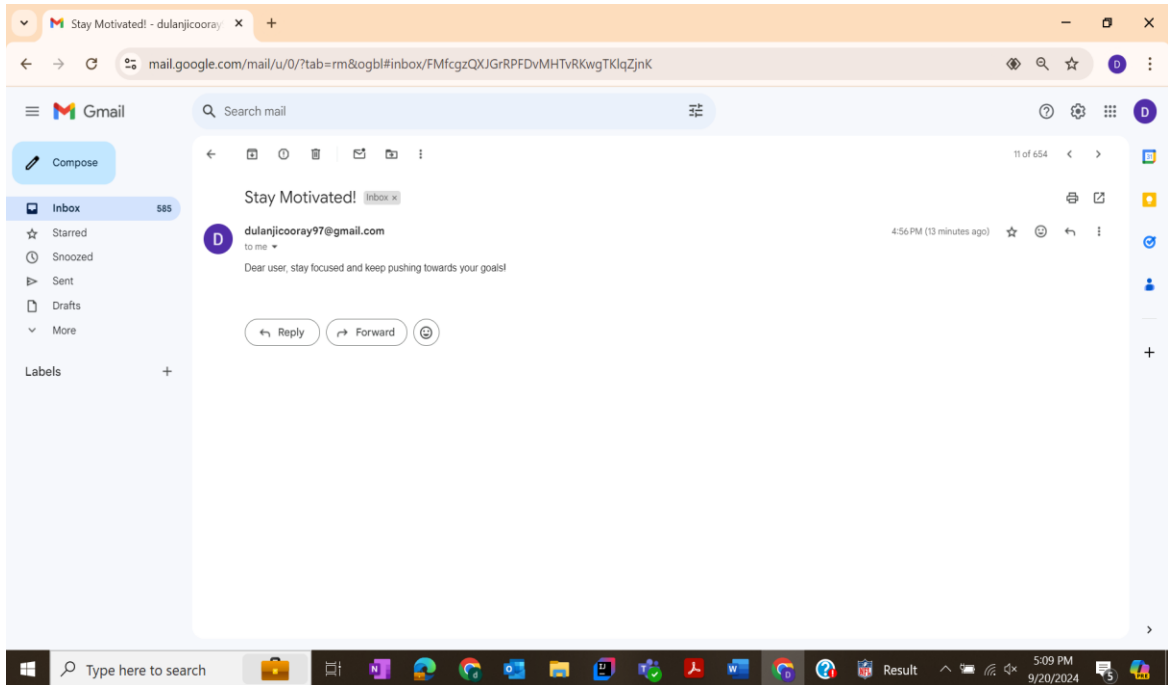


Implementation:

- **Spring Boot Mail API:** The system uses Spring Boot's **JavaMailSender** to send emails. The email service is configured to work with Gmail's SMTP server.
- **Gmail SMTP Configuration:** The system is set up to connect with Gmail's SMTP server by providing the necessary Gmail account credentials and SMTP settings.
- **HTML Email Templates:** The emails sent to users are formatted using HTML templates, making the notifications more visually appealing.
- **Automatic Triggers:** The system triggers an email notification whenever a task changes status (e.g., from pending to completed or failed).

Postman API And Send Email Samples





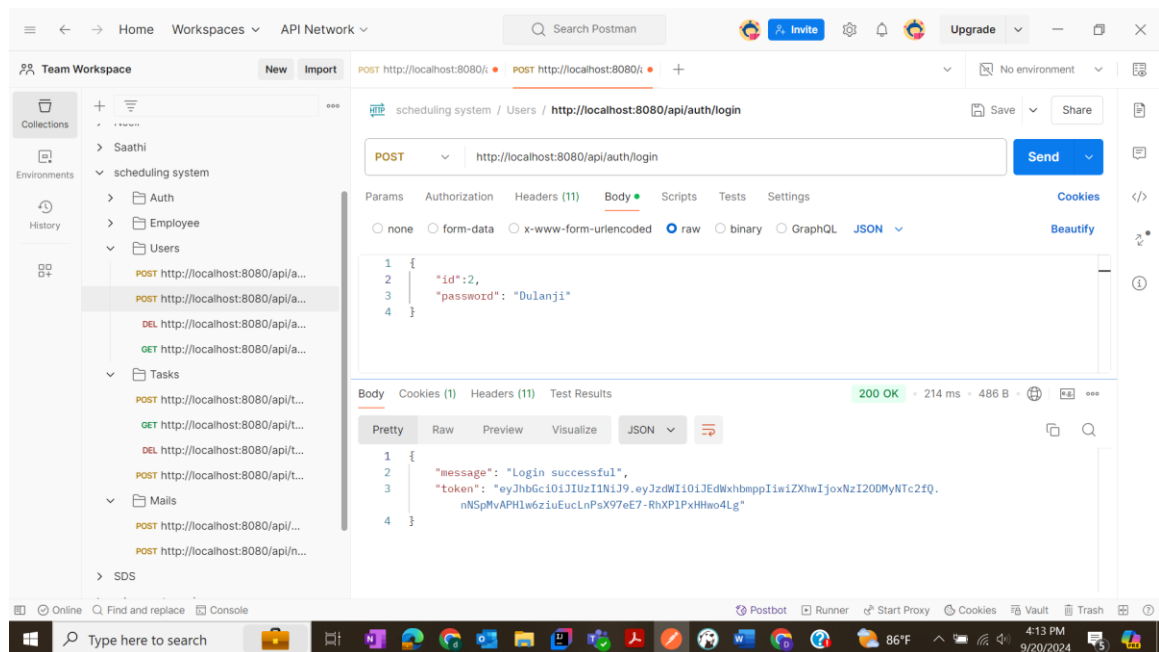
Security Features

In the Task Automation and Scheduling System, security is a critical aspect. The system ensures secure user authentication and session management through password hashing and JWT tokens.

Password Hashing

Password hashing ensures that user passwords are securely stored in the database in an irreversible format. Instead of storing plain-text passwords, the system uses a secure hashing algorithm to transform the password into a hash.

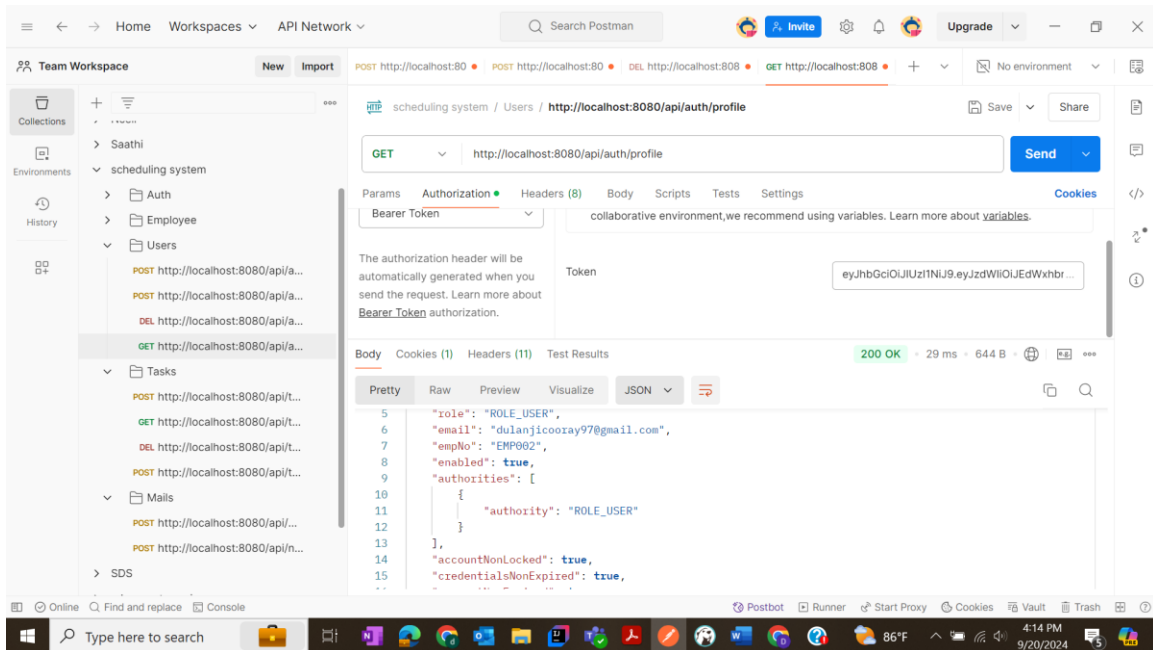
- **Hashing Algorithm:** The system uses BCrypt, a widely-used algorithm for password hashing that includes a salt to prevent dictionary attacks and rainbow table attacks.



JWT Token Authentication

After a successful login, the system generates a JWT (JSON Web Token) to manage user sessions securely. The JWT token contains user-specific information (e.g., user ID, roles) and is used to authenticate the user for subsequent API requests.

JWT tokens are stateless and provide a scalable way to handle authentication across distributed systems. They eliminate the need for server-side session storage.



Spring Security

Spring Security is integrated into the Task Automation and Scheduling System to provide robust authentication and authorization mechanisms. It secures user login, protects APIs, and manages user sessions.

Key Features:

1. User Authentication:

- Implements secure login using **BCrypt** for password hashing and verifies user credentials against stored hashes.
- Upon successful authentication, a **JWT token** is issued to maintain session state.

2. API Authorization:

- Uses role-based access control (RBAC) to restrict access to certain endpoints.
- Only authenticated users can create, modify, or delete tasks, with specific roles (e.g., admin, user) determining access levels.

3. Security Configuration:

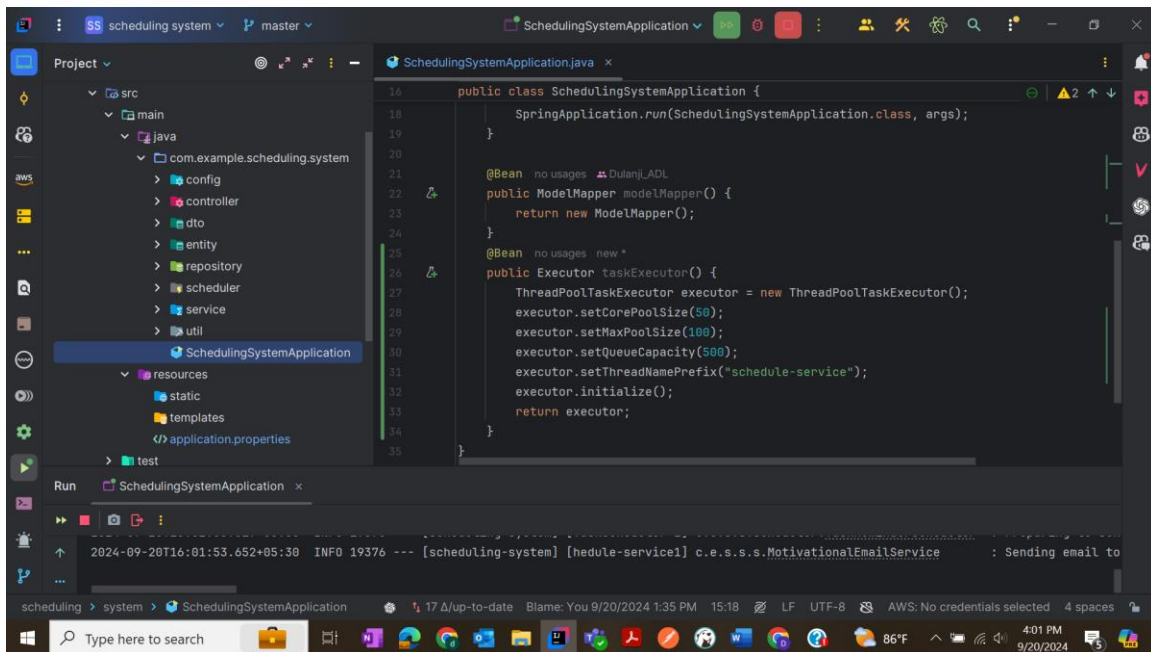
- Configures endpoints to require authentication while allowing public access to specific URLs (e.g., /login, /register).

Concurrency

- **Threads Using Java Concurrency Utilities**

Java provides several ways to work with concurrency, but starting with Threads is one of the basic approaches. Here's how you can use Threads in your system:

Using Threads: A Thread in Java allows you to execute a task asynchronously, meaning multiple tasks can run in parallel without blocking each other.



- Threads are the foundation for concurrency. You can create tasks and run them in separate threads.
- Executor Service offers better thread management by utilizing thread pools, improving performance and scalability.
- **CompletableFuture** allows **asynchronous**, non-blocking task handling, suitable for chaining dependent tasks.
- **@Async** and **@Transactional** used for handling asynchronous tasks with database operations.

Test Cases

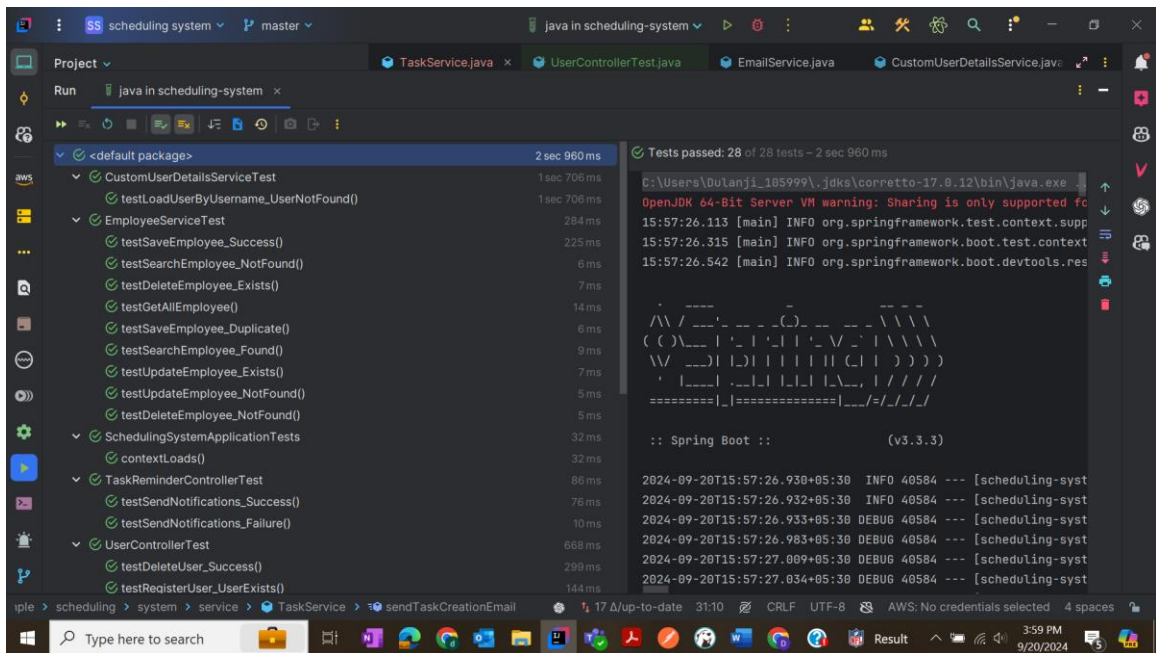
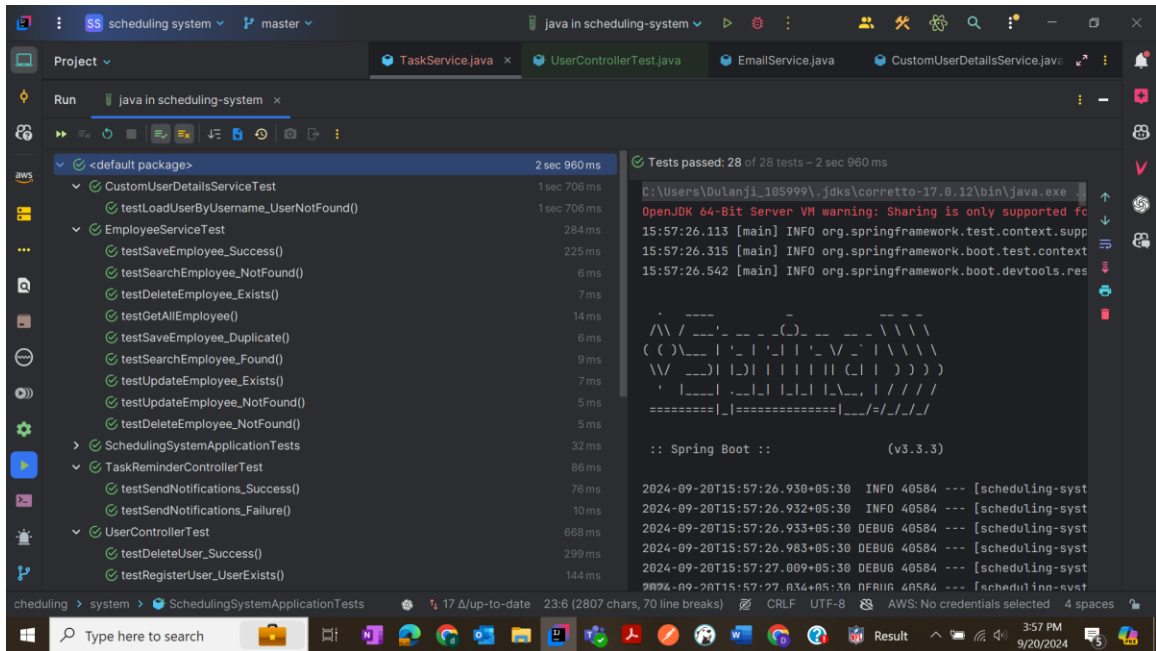
test cases cover the core functionality of your Task Automation and Scheduling System, ensuring proper task management, scheduling, API integration, concurrency, and notification functionality.

1. User Authentication

Test Case ID	Test Case Description	Input	Expected Output	Status
TC_01	Verify valid user login	Username: validUser, Password: validPwd	User is authenticated, and token/session is generated	Pass/Fail
TC_02	Verify invalid user login	Username: validUser, Password: wrongPwd	User is denied access, and error message is returned	Pass/Fail
TC_03	Verify registration of a new user	Valid user details	User is successfully registered	Pass/Fail
TC_04	Verify registration of a user with existing email	Email already exists	Error message stating "Email already exists"	Pass/Fail

2. Task Management

Test Case ID	Test Case Description	Input	Expected Output	Status
TC_05	Verify task creation	Task name, description, scheduled time, user ID	Task is successfully created	Pass/Fail
TC_06	Verify task update	Existing task ID, new details	Task details are updated successfully	Pass/Fail
TC_07	Verify task deletion	Valid task ID	Task is deleted	Pass/Fail
TC_08	Verify task deletion with invalid task ID	Invalid task ID	Error message "Task not found"	Pass/Fail
TC_09	Verify fetching all tasks for a user	Valid user ID	List of all tasks for the user is returned	Pass/Fail



Docker

Docker is a platform that helps developers package applications into containers. These containers hold all the components (code, libraries, dependencies) needed to run an application, ensuring it runs the same way in any environment. It simplifies development, deployment, and scaling of apps.

Common Docker Commands (with simple meanings):

- **docker run:**
Runs a new container based on a specified image.
 - Example: `docker run hello-world`
- **docker pull:**
Downloads a Docker image from Docker Hub (or another registry).
 - Example: `docker pull nginx`
- **docker build:**
Builds an image from a Dockerfile.
 - Example: `docker build -t my-app .`
- **docker ps:**
Lists all running containers.
 - Example: `docker ps`
- **docker stop:**
Stops a running container.
 - Example: `docker stop container_id`
- **docker rm:**
Removes a stopped container.
 - Example: `docker rm container_id`
- **docker rmi:**
Removes an image.
 - Example: `docker rmi image_id`
- **docker exec:**
Runs a command inside a running container.
 - Example: `docker exec -it container_id bash`

- **docker-compose up**: This command starts up all the services defined in the `docker-compose.yml` file. It pulls any necessary images, creates containers, networks, and volumes, and starts the services in the foreground, displaying their logs in the terminal.
- **docker-compose up -d**: This does the same thing as `docker-compose up`, but in **detached mode**. The services run in the background, so you won't see the logs in the terminal, allowing you to continue using the terminal for other tasks.

The screenshot shows a Visual Studio Code editor with a file named `docker-compose.yml` open. The file content is as follows:

```

1 version: '3.8'
2
3 services:
4   scheduling-system:
5     build: .
6     ports:
7       - "8080:8080"
8     environment:
9       SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/postgres
10      SPRING_DATASOURCE_USERNAME: postgres
11      SPRING_DATASOURCE_PASSWORD: root
12     depends_on:
13       - db
14
15 db:
16   image: postgres:latest
17   container_name: container-pg
18   hostname: db
19   environment:
20     POSTGRES_DB: postgres
21     POSTGRES_USER: postgres
22     POSTGRES_PASSWORD: root

```

The terminal at the bottom shows the output of the command `docker-compose up -d`:

```

PS D:\Java-Assessment-SE_CEG_JAV-0004\scheduling system> docker-compose up -d
times="2024-09-23T11:50:44+05:30" level=warning msg="D:\Java-Assessment-SE_CEG_JAV-0004\scheduling system\docker-compose.yml:
the attribute 'version' is obsolete, it will be ignored, please remove it to avoid potential confusion"
service "db" refers to undefined volume postgres-data: invalid compose project
PS D:\Java-Assessment-SE_CEG_JAV-0004\scheduling system>

```

