

GITHUB LINK:

https://github.com/BhavanaPerecharla/Mini_Project_Builder_Portfolio_Management_System

Builder Portfolio Management System – CodeLogic.pdf

Name: PERECHARLA BHAVANA

Project: Builder Portfolio Management System (Spring Boot)

Project Objective: -

The goal is to digitize portfolio tracking for the local Builders' Association to prevent data loss, reduce miscommunication, and improve client engagement. This backend system is built using **Spring Boot** and provides APIs for user and project management with proper layered architecture, validation, and unit testing.

Architecture

1. Controller Layer

- Handles incoming HTTP requests (@RestController)
- Maps endpoints like /users, /projects, /auth/login
- Delegates logic to the Service layer
- Returns appropriate ResponseEntity<>

2. Service Layer (Interface)

- Defines **contract** of business operations (createUser(), getAllProjects())
- Promotes abstraction and testability

3. ServiceImpl Layer

- Implements service interface
- Contains **actual business logic**
- Interacts with repository
- Handles mapping from DTOs to entities and vice versa using Mapper

4. Repository Layer

- Interfaces extending JpaRepository<T, ID>
- Uses Spring Data JPA to communicate with the **PostgreSQL** database
- Provides CRUD operations and custom queries

5. Model Layer

- Annotated with @Entity
- Maps Java classes to PostgreSQL tables using **ORM (Hibernate)**
- Example: User, Project

6. DTO Layer

- Contains **UserRequest, UserResponse, ProjectRequest, ProjectResponse**
- Used for clean data exchange between client and server
- Prevents exposing entity internals directly to external users

7. Mapper Layer

- Converts between Entity and DTO using manual mapping
- Promotes separation of concerns and cleaner service logic

8. Constants Layer

- Enums such as:
 - Role {ADMIN, CLIENT, BUILDER}
 - Project Status {INPROGRESS, COMPLETED, UPCOMING}

- Used for clarity and type safety

Security

- **Password encoding** is done using PasswordEncoder
- Secure login logic through AuthController and AuthService

Modules Implemented

1. User Management Module

- **Entity:** User
- **Fields:** id, name, email, role (ADMIN, BUILDER, CLIENT)
- **Endpoints:**
 - POST /users – Create user
 - GET /users – Fetch all users
- **Validation:** Ensured required fields and enum-based role restriction

1. API Endpoint: POST /users – Create User

Purpose

This endpoint is used to register a new user in the system. It accepts a UserRequest DTO and saves the user in the database after validation.

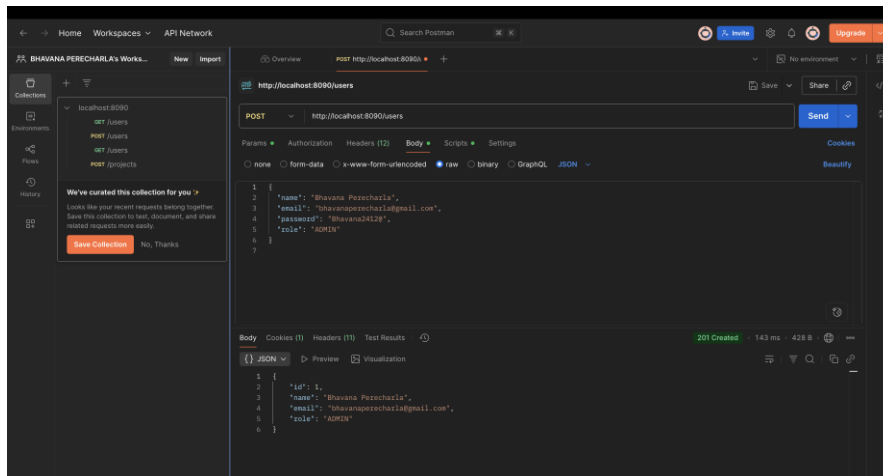
Validation Rules

- name: must not be blank
- email: must be a valid email and unique
- role: must be one of the allowed enum values – ADMIN, BUILDER, CLIENT

Backend Logic

- **Controller** (UserController) receives the request.
- **Service** (UserServiceImpl) performs:
 - Email uniqueness check
 - Role validation via Enum.valueOf
 - Saves user using UserRepository
- Returns a UserResponse DTO containing saved user data.

ADMIN:



CLIENT:

← → Home Workspaces ▾ API Network 🔍 Search Postman

BHAVANA PERECHARLA's Works... New Import

Collections + ▾

- localhost:8090
 - GET /users
 - POST /users
 - GET /users
 - POST /projects

We've curated this collection for you 🌟

Looks like your recent requests belong together. Save this collection to test, document, and share related requests more easily.

Save Collection No, Thanks

Overview POST http://localhost:8090/users

Save Share </>

POST http://localhost:8090/users

Params Authorization Headers (12) Body Scripts Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

Cookies Beautify

```
1 {
2   "name": "Varalakshmi Perecharla",
3   "email": "varalakshmi@gmail.com",
4   "password": "Varaa2412@",
5   "role": "CLIENT"
6 }
7
```

Body Cookies (1) Headers (11) Test Results

201 Created · 88 ms · 427 B

JSON Preview Visualization

```
1 {
2   "id": 3,
3   "name": "Varalakshmi Perecharla",
4   "email": "varalakshmi@gmail.com",
5   "role": "CLIENT"
6 }
```

← → Home Workspaces ▾ API Network 🔍 Search Postman

BHAVANA PERECHARLA's Works... New Import

Collections + ▾

- localhost:8090
 - GET /users
 - POST /users
 - GET /users
 - POST /projects

We've curated this collection for you 🌟

Looks like your recent requests belong together. Save this collection to test, document, and share related requests more easily.

Save Collection No, Thanks

Overview POST http://localhost:8090/users

Save Share </>

POST http://localhost:8090/users

Params Authorization Headers (12) Body Scripts Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

Cookies Beautify

```
1 {
2   "name": "Srinivasa Rao Perecharla",
3   "email": "srinivasarao@gmail.com",
4   "password": "Srinivasa2412@",
5   "role": "CLIENT"
6 }
7
```

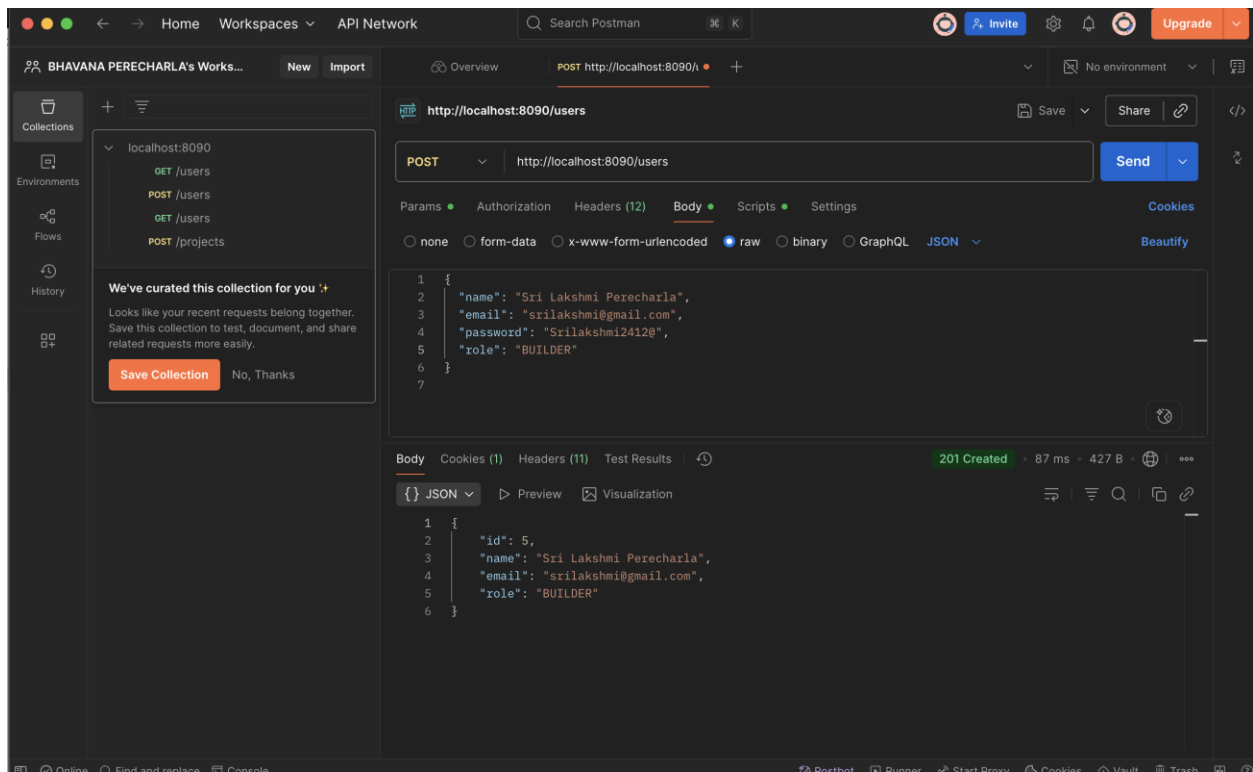
Body Cookies (1) Headers (11) Test Results

201 Created · 86 ms · 430 B

JSON Preview Visualization

```
1 {
2   "id": 4,
3   "name": "Srinivasa Rao Perecharla",
4   "email": "srinivasarao@gmail.com",
5   "role": "CLIENT"
6 }
```

BUILDER:



DATABASE

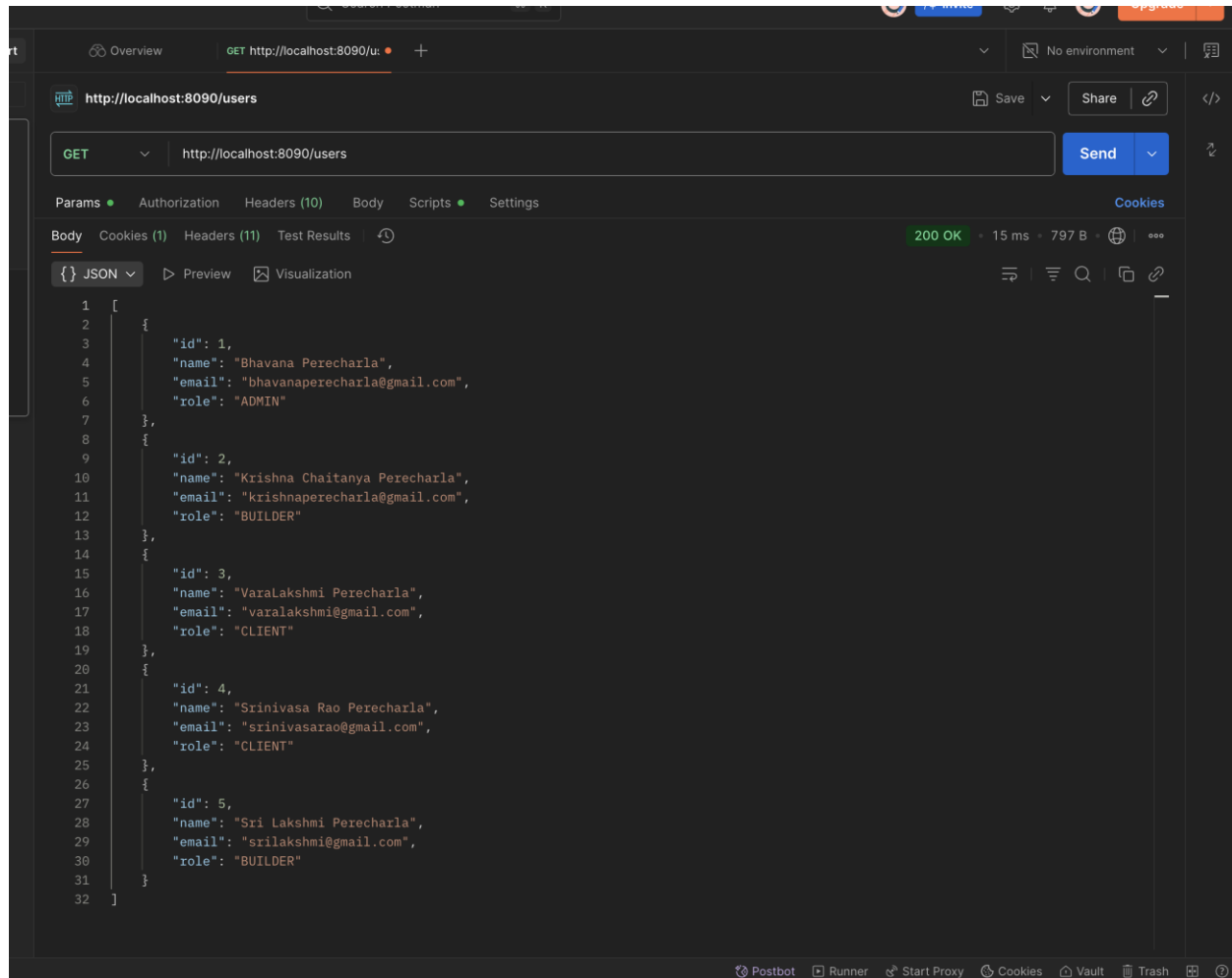
Data Output Messages Notifications						
Showing rows: 1 to 5 Page No:						
	id [PK] bigint	email character varying (255)	name character varying (255)	password character varying (255)	role character varying (255)	
1	1	bhavanaperecharla@gmail.c...	Bhavana Perecharla	\$2a\$10\$58BjoPA.N1wrubizQ.cr6epBB8YjdCLDVk96djsaYRVBHVHNxb...	ADMIN	
2	2	krishnaperecharla@gmail.com	Krishna Chaitanya Perecharla	\$2a\$10\$wzKiLFmfTT197qlXPbML5uiFMKFYOy.hjVq/qVqmPP.sgcdt5go...	BUILDER	
3	3	varalakshmi@gmail.com	VaraLakshmi Perecharla	\$2a\$10\$GKWnenaTGisCgn9.SnrcbOYM5izm4ZXw5c975/t.7JLNIIBh5ET...	CLIENT	
4	4	srinivasarao@gmail.com	Srinivasa Rao Perecharla	\$2a\$10\$cccZrPSEb9uXhFLx.jd7yuGe4HBkd9P1sVa/cR7bHxC6XHUGR...	CLIENT	
5	5	srilakshmi@gmail.com	Sri Lakshmi Perecharla	\$2a\$10\$ZKeo5XTGySrics11IjYL8uDVthoWKRQwRxUbELr9xYOLnZwV.Vi1i	BUILDER	

2.GET/Users:

Process Logic:

1. Request is received by the `UserController.getAllUsers()` method.
2. The controller calls `UserService.getAllUsers()`.
3. The service fetches all User entities using `UserRepository.findAll()`.
4. Each entity is mapped to a `UserResponse` DTO using `UserMapper`.

5. The list is returned to the controller and sent as the HTTP response.



2. Project Management Module

- **Entity:** Project
- **Fields:** id, title, description, status (UPCOMING, IN_PROGRESS, COMPLETED), clientId, builderId
- **Endpoints:**
 - POST /projects – Add project
 - GET /projects – List all projects
 - GET /projects/{id} – Fetch specific project
 - PUT /projects/{id} – Update existing project

- DELETE /projects/{id} – Remove project
- **Relationships:** ManyToOne between Project and User (for builder and client)

1. POST /projects – Add new project

Purpose:

To allow the creation of a new project in the system by specifying details like title, description, status, and associated client and builder

Validations:

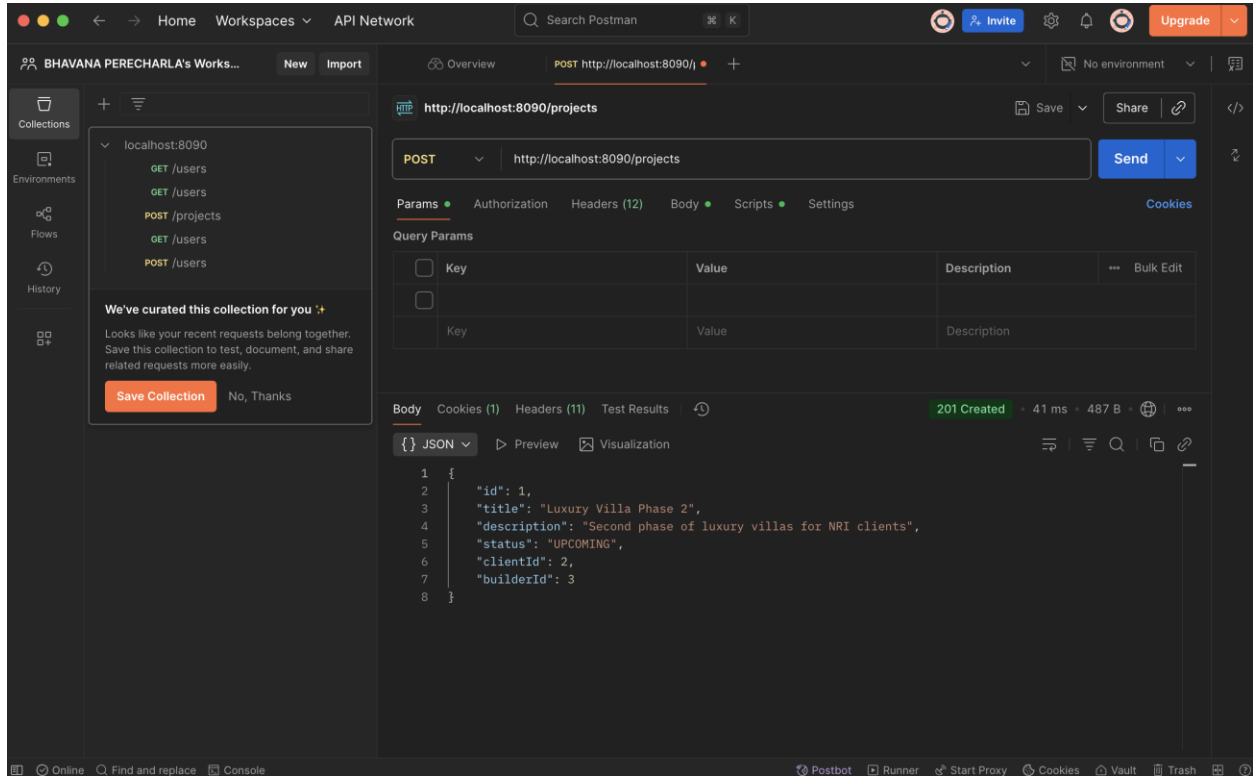
- clientId and builderId must exist in DB.
- Enum values must match exactly (case-sensitive).
- Title and description should not be null or empty (if validated in DTO).

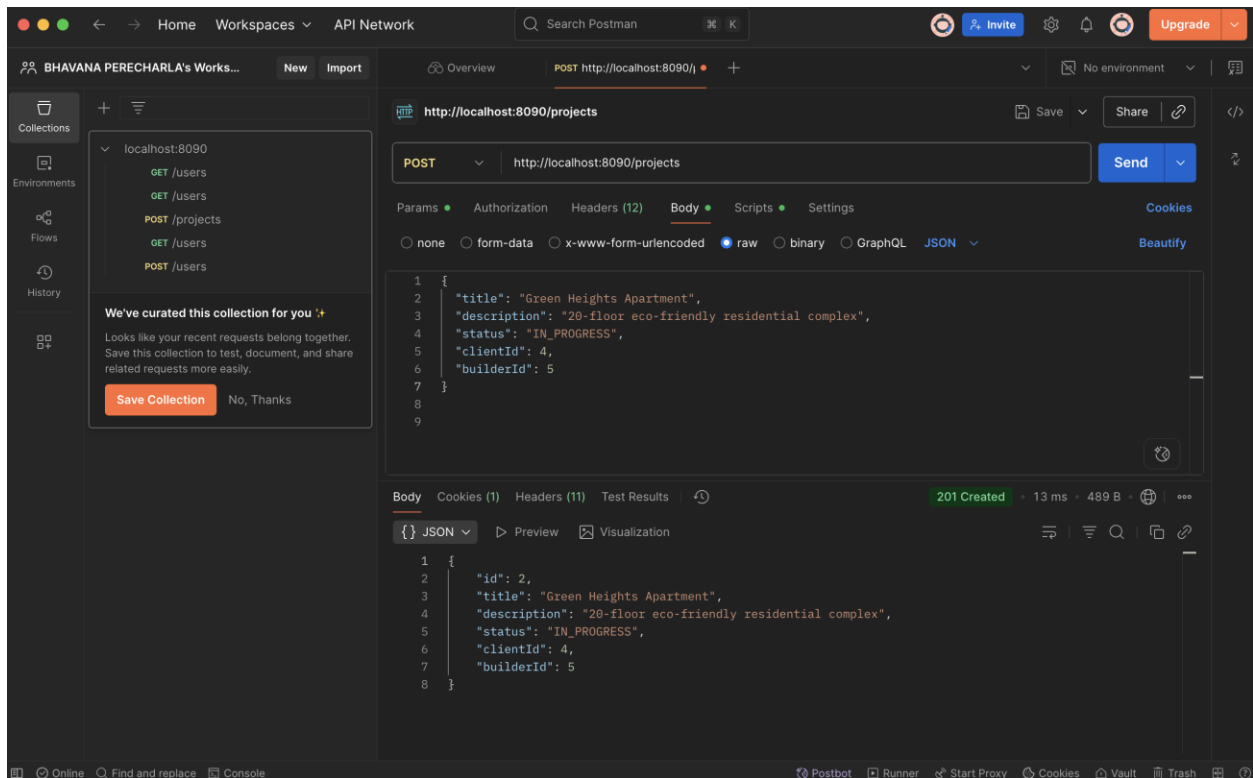
Step-by-Step Logic Flow:

- 1. Receive Request**
 - a. API receives a JSON payload with fields: title, description, status, clientId, and builderId.
- 2. Validation (Optional, but Recommended)**
 - a. The input is validated using annotations in ProjectRequest DTO (e.g., @NotBlank, @NotNull).
 - b. Ensures no field is empty and status matches one of the allowed enums: UPCOMING, IN_PROGRESS, COMPLETED.
- 3. Controller Layer (ProjectController)**
 - a. Calls projectService.createProject(projectRequest) with the request DTO.
- 4. Service Layer (ProjectServiceImpl)**
 - a. Validates if both clientId and builderId exist in the database via userRepository.findById().
 - b. Optionally checks if those users have correct roles (CLIENT, BUILDER).
 - c. Converts ProjectRequest DTO to a Project entity using ProjectMapper.toEntity(request).
- 5. Repository Layer**
 - a. Calls projectRepository.save(project) to persist the entity into the database.
 - b. JPA handles ID generation and table insertion.

6. Return Response

- Converts the saved entity to a ProjectResponse DTO.
- Returns it with HTTP 201 Created.





DATABASE

Data Output Messages Notifications							
Showing results							
	id [PK] bigint	description character varying (1000)	status character varying (255)	title character varying (255)	builder_id bigint	client_id bigint	
1	1	Second phase of luxury villas for NRI clients	UPCOMING	Luxury Villa Phase 2	3	2	
2	2	20-floor eco-friendly residential complex	IN_PROGRESS	Green Heights Apartment	5	4	
3	3	Sky Towers High Buildings	IN_PROGRESS	Luxury Homes	3	2	

2.GET /projects – List all projects

Purpose: To retrieve a list of all projects stored in the database. This helps Admins, Clients, or Builders to view available or assigned projects.

Step-by-Step Logic:

1. Request Initiated:

- a. A client (Postman/browser/another service) makes a GET request to the /projects endpoint.

2. API Controller Receives Request:

- a. The controller layer is responsible for handling HTTP requests. It receives the GET request and calls the service layer to get the data.

3. Service Layer Processes Logic:

- a. The service layer interacts with the repository (database layer) to fetch all project records.
- b. It ensures any necessary business logic (e.g., sorting, filtering, transformations) is applied.

4. Repository Fetches Data:

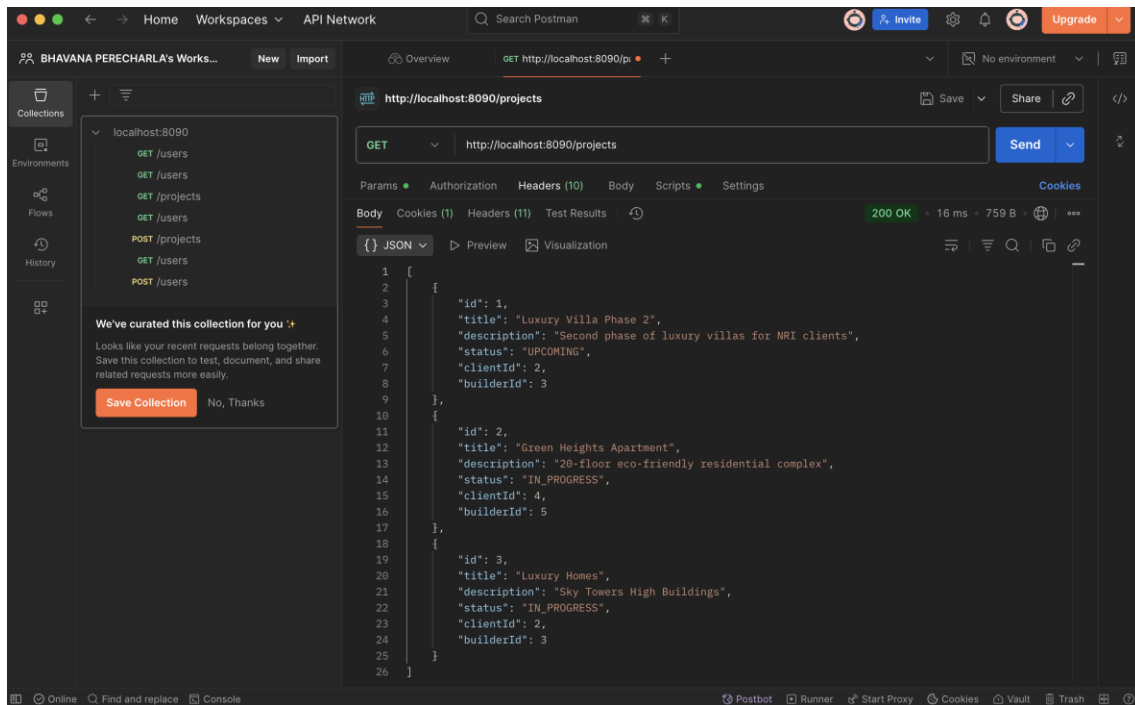
- a. The repository connects to the database and retrieves all rows from the projects table.
- b. This includes each project's id, title, description, status, clientId, and builderId.

5. Data Transformation:

- a. Each project entity is converted into a clean response format (DTO – Data Transfer Object) that is safe and structured for clients.

6. Response Sent Back:

- a. The controller sends back the list of all projects in JSON format along with a **200 OK** status.
- b. If the list is empty, it still returns an empty array with **200 OK**.



3.GET /projects/{id} – Fetch specific project

The **primary purpose** of this endpoint is to: **Retrieve detailed information of a specific project by its unique ID.**

Step-by-Step Logic

1. Controller Layer:

- URL pattern `/projects/{id}` maps to `getProjectById()` method.
- Extracts the `{id}` path variable.
- Calls `projectService.getProjectById(id)`.

2. Service Layer:

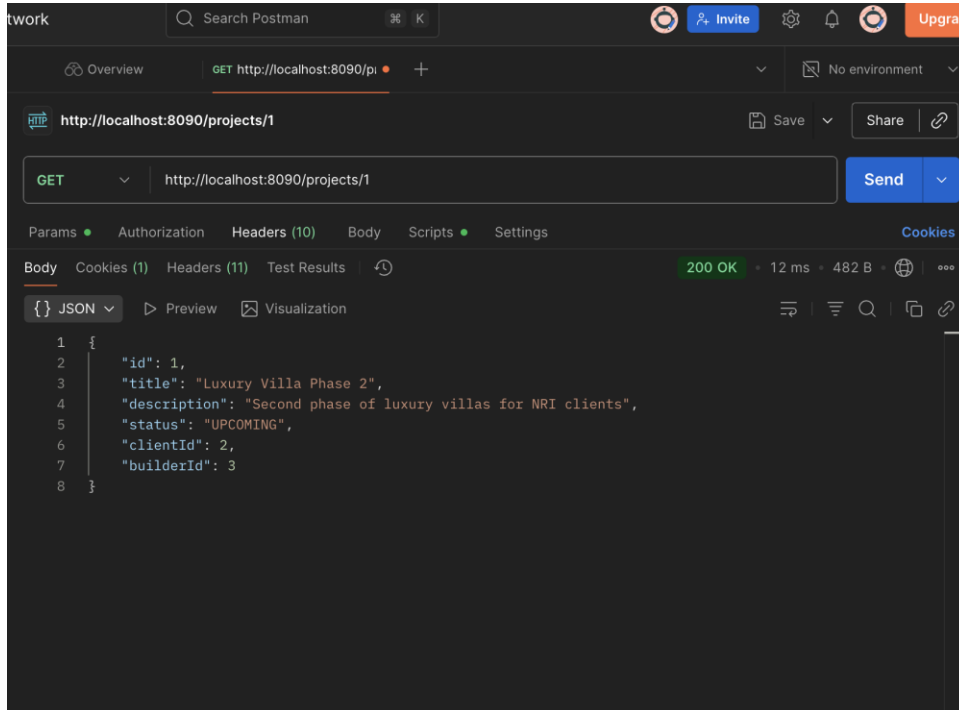
- `projectRepository.findById(id)` is called.
- If the project is present, it's returned.
- If not present, throws a custom `ResourceNotFoundException`.

3. Mapper Layer:

- Entity is mapped to a clean response DTO (`ProjectResponse`).

4. Response:

- a. Controller returns `ResponseEntity.ok(responseDto)`.
- b. If exception, it's handled by `@ControllerAdvice`.



4.PUT /projects/{id} – Update existing project

To **modify the details** of an already existing project in the system.

This is used when project information such as title, description, current status, or associated client/builder changes and needs to be updated.

Code Logic

1. Extract the id from the request URL

- a. The controller takes the projectId from the path variable.

2. Read new project data from the request body

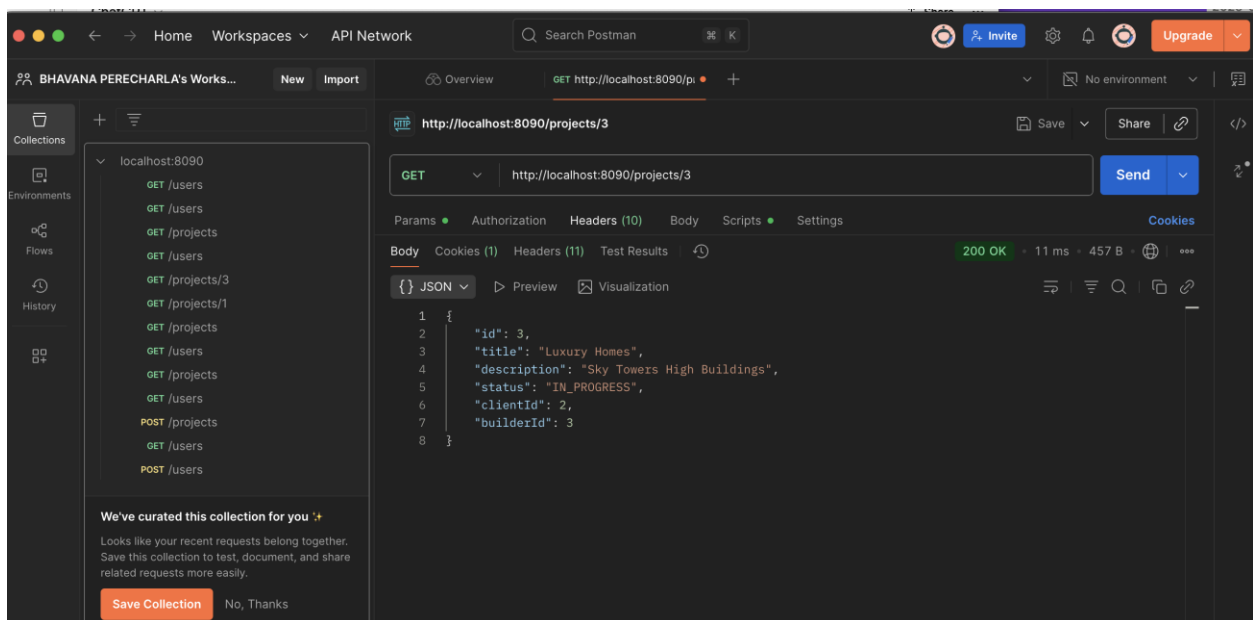
- a. The updated title, description, status, clientId, and builderId are sent in JSON format.

3. Check if the project exists in the database

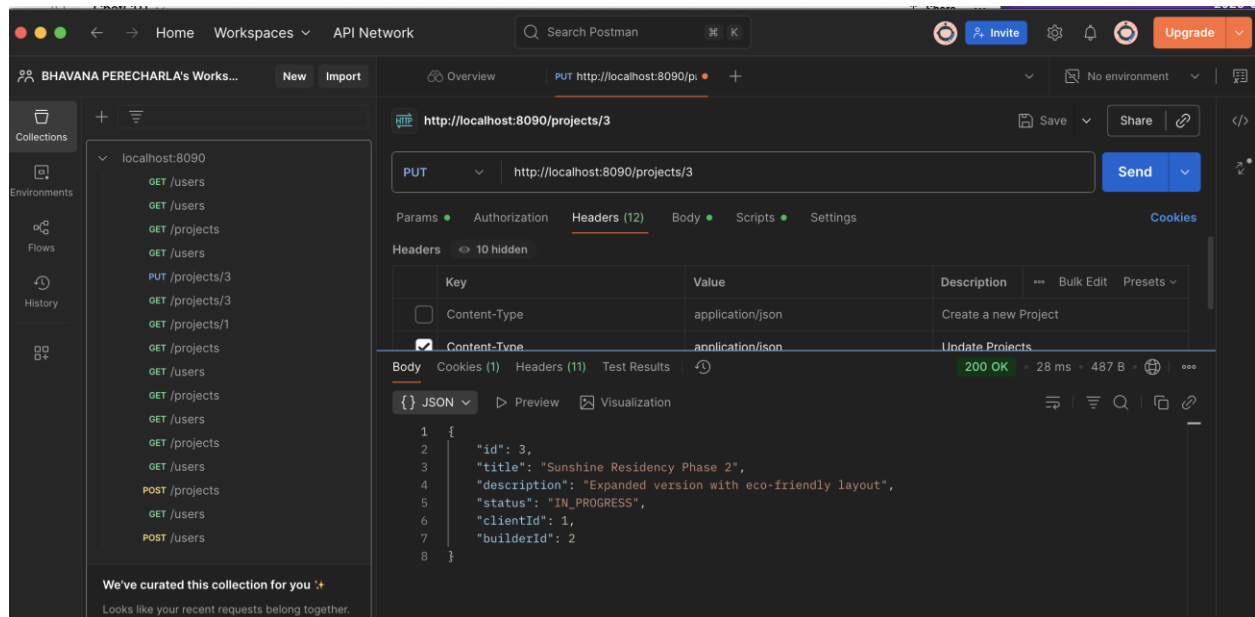
- a. Use the repository to find the project by ID.

- b. If not found, throw a `ProjectNotFoundException`.
- 4. Validate referenced `clientId` and `builderId`**
 - a. Ensure the given `clientId` corresponds to a user with the role `CLIENT`.
 - b. Ensure the given `builderId` corresponds to a user with the role `BUILDER`.
 - c. If either is invalid or missing, throw an appropriate exception.
- 5. Update the fields of the existing project**
 - a. Replace existing title, description, status, etc., with new values from the request.
- 6. Save the updated project**
 - a. Call the repository save method to persist the changes to the database.
- 7. Return the updated project details**
 - a. Send a 200 OK response with the updated project as confirmation.

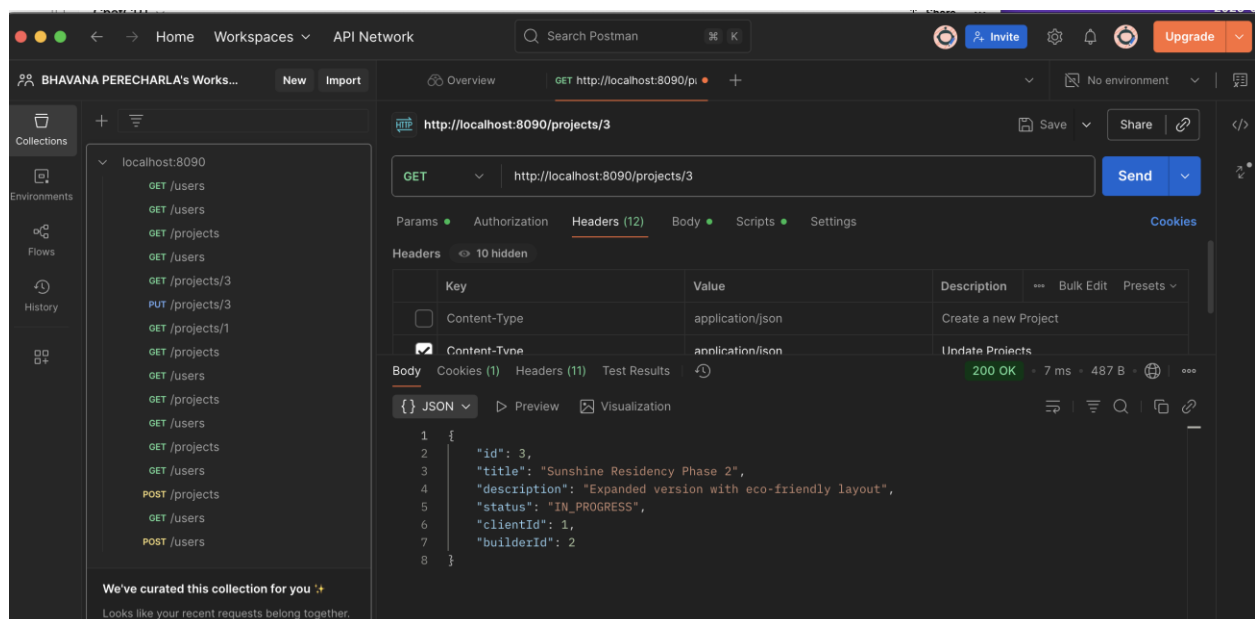
Before Updating the project-id: 3



UPDATING PROJECT with project id -3:



AFTER-GET PROJECT ID WITH 3



DATABASE BEFORE AND AFTER:

Data Output Messages Notifications						
Showing row						
	id [PK] bigint	description character varying (1000)	status character varying (255)	title character varying (255)	builder_id bigint	client_id bigint
1	1	Second phase of luxury villas for NRI clients	UPCOMING	Luxury Villa Phase 2	3	2
2	2	20-floor eco-friendly residential complex	IN_PROGRESS	Green Heights Apartment	5	4
3	3	Sky Towers High Buildings	IN_PROGRESS	Luxury Homes	3	2

Data Output Messages Notifications						
Showing row						
	id [PK] bigint	description character varying (1000)	status character varying (255)	title character varying (255)	builder_id bigint	client_id bigint
1	1	Second phase of luxury villas for NRI clients	UPCOMING	Luxury Villa Phase 2	3	2
2	2	20-floor eco-friendly residential complex	IN_PROGRESS	Green Heights Apartment	5	4
3	3	Expanded version with eco-friendly layout	IN_PROGRESS	Sunshine Residency Phase 2	2	1

5. DELETE /projects/{id} – Remove project

Purpose

To **permanently remove a project** from the system using its unique ID.

1.Receive Request

- A DELETE request is made to the URL /projects/{id} (e.g., /projects/5).
- The {id} in the URL is the unique identifier of the project to be deleted.

2.Extract the Project ID

- The controller method reads the projectId from the path variable in the request URL.

3.Check if the Project Exists

- Call the project repository (projectRepository.findById(id)) to check whether a project with the given ID exists.
- If not found, throw a custom exception (e.g., ProjectNotFoundException) and return a 404 Not Found.

4.Delete the Project

- If the project is found, use the repository method (projectRepository.deleteById(id)) to delete it from the database.

5. Return a Success Response

- After deletion, return an HTTP 204 No Content

BEFORE DELETION:

Data Output Messages Notifications							
Showing rows							
	id [PK] bigint	description character varying (1000)	status character varying (255)	title character varying (255)	builder_id bigint	client_id bigint	
1	1	Second phase of luxury villas for NRI clients	UPCOMING	Luxury Villa Phase 2	3	2	
2	2	20-floor eco-friendly residential complex	IN_PROGRESS	Green Heights Apartment	5	4	
3	3	Expanded version with eco-friendly layout	IN_PROGRESS	Sunshine Residency Phase 2	2	1	

DELETION OF PROJECT ID:2

HTTP <http://localhost:8034/projects/2> 3:40 PM X


DELETE [Send](#)

Params • Authorization Headers (10) Body Scripts • Settings Cookies

Query Params

<input type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input type="checkbox"/>					
	Key	Value	Description		

Body Cookies Headers Test Results 3:40 PM 204 No Content • 227 ms • 282 B • ...



Response not stored

[Save response setting](#) must have been off when this request was sent.

DELETION AFTER PROJECT ID:2

```

3
4 select * from projects
5

```

Data Output Messages Notifications

Showing rows: 1 to 2 Page No: 1

	id [PK] bigint	description character varying (1000)	status character varying (255)	title character varying (255)	builder_id bigint	client_id bigint
1	1	Second phase of luxury villas for NRI clients	UPCOMING	Luxury Villa Phase 2	3	2
2	3	Expanded version with eco-friendly layout	IN_PROGRESS	Sunshine Residency Phase 2	2	1

Functional Validations

- Input validations using @NotBlank, @NotNull
- Enum constraints for role and status
- Error handling for invalid/missing data

Service Layer

1. createUser() → should save and return the user
2. getAllProjects() → should return list of projects

Controller Layer

3. GET /users → should return HTTP 200 and list of users
4. POST /projects → should return HTTP 201 when project is created

UNIT TESTING – UserControllerTest.java

This section documents the unit tests implemented for the UserController class in the Builder Portfolio Management System project. These tests ensure that the controller layer behaves as expected when creating users and fetching all users.

Test Case Summary

1. testCreateUser_ShouldReturnCreatedUser()

Purpose:

Tests the POST /users endpoint by simulating the creation of a user. It ensures:

- The controller accepts a UserRequest object.
- Delegates it to the service correctly.
- Returns HTTP 201 Created with correct user data.

Key Assertions:

- HTTP status code is 201.
- Response body is not null.
- Response fields match the input request (name, email, role).

2. testGetAllUsers_ShouldReturnUserList()

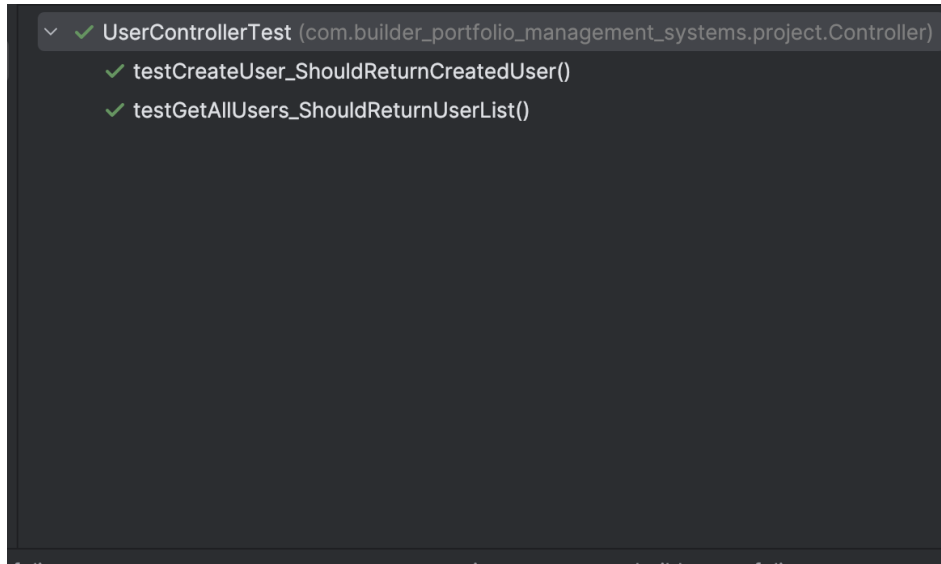
Purpose:

Tests the GET /users endpoint by simulating retrieval of all users. It ensures:

- The controller calls the service to fetch all users.
- Returns a list of users with HTTP 200 OK.

Key Assertions:

- HTTP status code is 200.
- Response body is a list of users.
- Each user has the correct name, email, and role.



ProjectControllerTest.java – Unit Testing Description

The purpose of this unit test class is to test the behavior of the ProjectController layer independently by mocking the ProjectService layer. These tests ensure that the controller endpoints for handling project creation and retrieval are functioning as expected under both normal and exceptional conditions.

Test Methods Explained

1. testCreateProject_ShouldReturnCreatedProject()

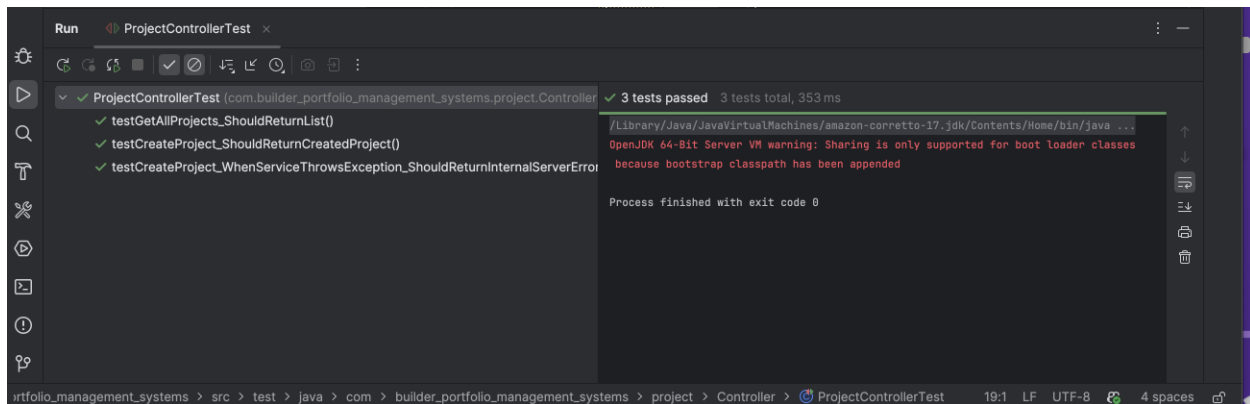
- **Objective:** Verifies that the controller successfully returns a 201 Created response when a valid ProjectRequest is submitted.
- **Setup:**
 - A mock ProjectRequest is passed to the controller.
 - The mocked service returns a corresponding ProjectResponse.
- **Assertions:**
 - HTTP status code is 201.
 - Response body is not null and matches the expected project title.
- **Significance:** Validates correct behavior for POST /projects.

2. testGetAllProjects_ShouldReturnList()

- **Objective:** Checks that the controller returns all existing projects in a list.
- **Setup:**
 - The mock service returns a list containing one ProjectResponse.
- **Assertions:**
 - HTTP status code is 200.
 - Response contains a non-empty list.
 - Project details match the expected data.
- **Significance:** Validates correct behavior for GET /projects.

3. testCreateProject_WhenServiceThrowsException_ShouldReturnInternalServerError()

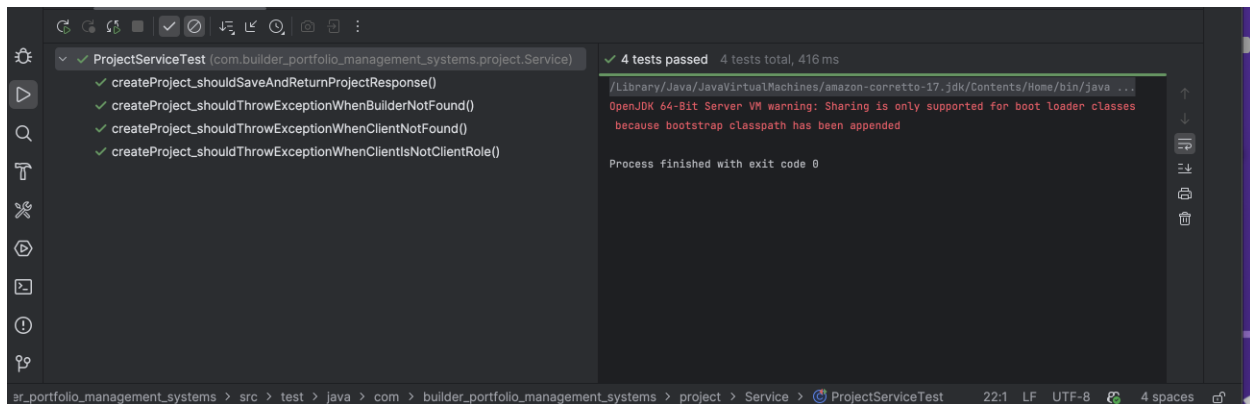
- **Objective:** Tests the controller's behavior when the service layer throws an exception during project creation.
- **Setup:**
 - The mock service throws a RuntimeException with message "DB error".
- **Assertions:**
 - Confirms the exception is propagated and message is accurate.
- **Significance:** Validates controller response under failure conditions (helps ensure exception handling is robust in production code).



Unit Testing - ProjectServiceTest.java

This particular test class is designed to **validate the core business logic** of the ProjectServiceImpl- createProject() method, ensuring it handles various scenarios correctly, including success and multiple failure paths.

Test Method	Scenario	Outcome
createProject_shouldSaveAndReturnProjectResponse()	Valid client & builder with correct roles	Returns ProjectResponse
createProject_shouldThrowExceptionWhenClientNotFound()	Client ID does not exist	Throws IllegalArgumentException
createProject_shouldThrowExceptionWhenBuilderNotFound()	Builder ID does not exist	Throws IllegalArgumentException
createProject_shouldThrowExceptionWhenClientIsNotClientRole()	Client has wrong role (e.g., ADMIN)	Throws IllegalArgumentException



4. UserServiceTest – Unit Testing

This unit test class verifies the correctness of the service layer responsible for managing users in the **Builder Portfolio Management System**. It ensures the UserServiceImpl logic behaves as expected for various scenarios using **JUnit 5** and **Mockito**.

. Test Case: createUser_shouldSaveAndReturnUserResponse()

► Description:

This test checks that when a new user is created:

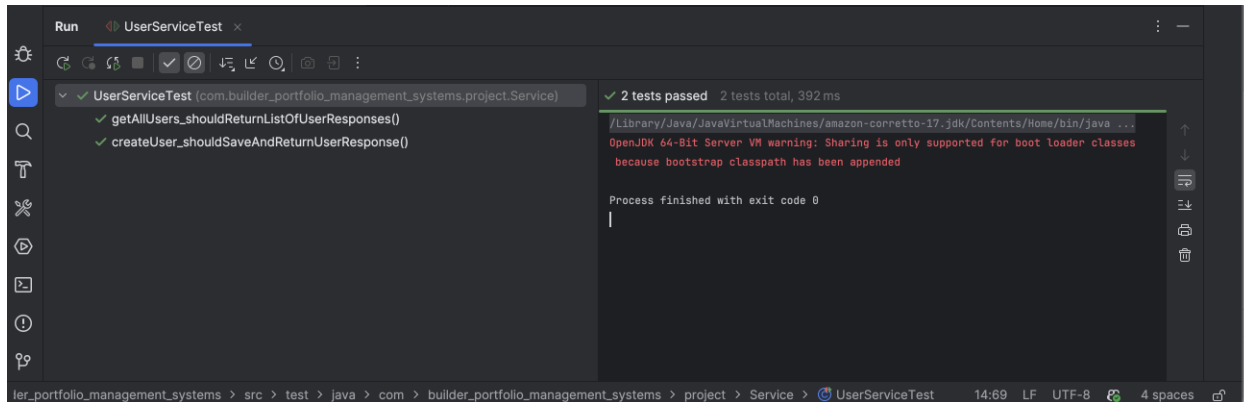
- The input DTO is correctly mapped to a User entity.
- The password is encoded using PasswordEncoder.
- The user is saved via the UserRepository.
- The saved entity is converted back to a UserResponse DTO.
- All fields are verified for correctness after creation.

2. Test Case: getAllUsers_shouldReturnListOfUserResponses()

► Description:

This test verifies that:

- The system retrieves a list of users from the database.
- Each user is mapped to its corresponding UserResponse DTO.
- The resulting list is correctly sized and contains expected values.



Instructions to Run the Project

This project is built using **Spring Boot** , **Maven**, and **PostgreSQL**, following a layered RESTful architecture.

Please follow the steps below to run and test the complete project:

1. Prerequisites

Ensure the following software is installed on your system:

Tool	Version
Java SDK	17 or above
Maven	3.8+
PostgreSQL	12 or above
Postman	(for API testing)
IDE	IntelliJ IDEA / Eclipse / VS Code (Recommended)

2. PostgreSQL Database Setup

1. **Create a database** named:

Sql : CREATE DATABASE builder_portfolio_db;

2. **(Optional)** You can modify the DB name or credentials in the application file:

src/main/resources/application.properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/builder_portfolio_db
spring.datasource.username=your_username
spring.datasource.password=your_password
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

3. Run the Application

You can run the application using:

Option A: Using Maven

```
mvn spring-boot:run
```

Option B: From Your IDE

- Open the project in your IDE.
- Run the main class:

```
com.builder_portfolio_management_systems.project.BuilderPortfolioManagementSystemsApplication
```

4. Test the APIs

Use **Postman** or similar tools to test the following endpoints:

Method	Endpoint	Description
POST	/users	Register new user
GET	/users	Get all users
POST	/projects	Create new project
GET	/projects	Get all projects
POST	/auth/login	User login

All request and response formats are defined in the DTO classes inside:

src/main/java/com/builder_portfolio_management_systems/project/DTO/

5. Run Unit Tests

To run the unit tests for controller and service layers:

```
mvn test
```

The following test classes are included:

- UserServiceTest
- UserControllerTest
- ProjectServiceTest
- ProjectControllerTest

6. Project Structure Overview

```
bash
```

```
CopyEdit
```

```
|— Controller    # REST API endpoints
|— Service       # Interfaces
|— ServiceImpl  # Business logic
|— Repository    # JPA Repositories
|— Model         # Entity models
|— DTO           # Request and response objects
|— Mapper        # Entity-DTO converters
|— Constants     # Enums like Role, Status
|— Config        # Global config
|— tests/        # JUnit + Mockito test cases
```

Note for Evaluation

- All validation, exception handling, and business logic are implemented using best practices.
- Passwords are securely encoded.
- Enum-based fields (Role, ProjectStatus) are used for clarity and type safety.
- Clean architecture with DTOs, service/repo separation, and meaningful unit tests is followed.
- The project runs independently and can be tested via Postman.