**Lab Assignment: User Management & Role Assignment System**

**1. Objective**

Design and implement a **User Management System** following **Robert C. Martin’s Clean Architecture principles** using **Spring Boot**.

This system will allow:

* Creating **Users** and **Roles**
* Assigning **Roles** to Users
* Retrieving User details along with assigned Roles

You must separate **Entities**, **Use Cases**, **Adapters**, and **Frameworks/Drivers** clearly in your code.

**2. Functional Requirements**

| **#** | **Requirement** | **Description** |
| --- | --- | --- |
| FR1 | Create User | Create a new user with name and email. |
| FR2 | Create Role | Create a new role with roleName. |
| FR3 | Assign Role to User | Assign an existing role to an existing user. |
| FR4 | Fetch User Details | Retrieve user info along with the list of assigned roles. |

**Entity Attributes**

**User:**

* id: UUID
* name: String
* email: String

**Role:**

* id: UUID
* roleName: String

**REST API Endpoints**

|  |  |  |  |
| --- | --- | --- | --- |
| **HTTP Method** | **URL** | **Request Body** | **Response** |
| POST | /users | { "name": "John Doe", "email": "john@example.com" } | Created User ID |
| POST | /roles | { "roleName": "ADMIN" } | Created Role ID |
| POST | /users/{userId}/assign-role/{roleId} | (Empty body) | Success Message |
| GET | /users/{id} | - | User Details with Roles |

**3. Non-Functional Requirements**

* **Follow Clean Architecture structure strictly**.
* **Spring Boot** version 3.x or higher.
* Use **H2 database** (in-memory).
* Validation:
  + Email should be properly formatted.
  + Name and role name must not be blank.
* Errors should return **meaningful HTTP status codes** (e.g., 404 if user not found).
* **Unit test** the **application layer** (use case services).

**4. Architecture Expectations (Mandatory Structure)**

**a) Domain Layer (Entities)**

* User.java
* Role.java

Pure business objects. No annotations like @Entity inside them.  
(If you want, you can have a separate persistence entity for DB.)

**b) Application Layer (Use Cases)**

* UserService.java — contains all user-related operations.
* RoleService.java — contains all role-related operations.
* interfaces/
  + UserRepository.java
  + RoleRepository.java

This layer **depends only on domain** and **interfaces**.

**c) Infrastructure Layer (Adapters)**

* **Persistence**:
  + UserJpaRepository.java (extends Spring Data JpaRepository internally)
  + RoleJpaRepository.java
* **Controller**:
  + UserController.java
  + RoleController.java

Only this layer will use Spring Boot, JPA, etc.

**d) Configuration Layer**

* BeanConfig.java for Bean wiring if needed.

**e) Folder Structure Example**

swift

CopyEdit

src/main/java/com/example/usermanagement/

├── domain/

│ ├── User.java

│ └── Role.java

├── application/

│ ├── UserService.java

│ ├── RoleService.java

│ └── interfaces/

│ ├── UserRepository.java

│ └── RoleRepository.java

├── infrastructure/

│ ├── controller/

│ │ ├── UserController.java

│ │ └── RoleController.java

│ └── persistence/

│ ├── UserJpaEntity.java

│ ├── RoleJpaEntity.java

│ ├── UserJpaRepository.java

│ └── RoleJpaRepository.java

└── config/

└── BeanConfig.java

**5. Submission Checklist**

Before submitting, make sure:

* Codebase follows Clean Architecture layers.
* **Domain layer** has no Spring Boot / JPA annotations.
* **Repositories** in Application layer are interfaces (ports).
* **Persistence** classes adapt JpaRepository to Application Interfaces.
* **Controllers** adapt REST APIs to Services.
* **Dependency Rule** is maintained (dependencies point inward).
* **Basic Validation** is implemented (e.g., blank name/invalid email).
* **Error Handling** returns proper HTTP status and message.
* **H2 database console** is enabled (/h2-console).
* **Unit Tests** exist for Service Layer.
* README.md with setup instructions (run and test).

**6. Bonus (Optional Advanced Tasks)**

* Add Pagination to GET /users list API.
* Add "remove role from user" API.
* Add createdDate/updatedDate audit fields.
* Use **DTOs** for request/response instead of exposing domain entities directly.
* Create Swagger/OpenAPI documentation using SpringDoc.
* Add integration tests for full end-to-end testing.

**7. Hints**

* **UUID** generation: Use UUID.randomUUID() inside constructors.
* **Validation**: Use @Valid, @NotBlank, @Email annotations.
* **Error Handling**: Use @ControllerAdvice to catch exceptions globally.
* **Separation**: Never directly inject JpaRepository into the service layer.
* **Testing**: Mock the Repository Interfaces (e.g., Mockito).

**Final Tip:**

***"Your entities and use cases should never know if they are backed by JPA, exposed through REST, or consumed by a mobile app."***  
That's the spirit of **Clean Architecture**.