ព្រះរាជាណាចក្រកម្ពុជា

ជាតិ សាសនា ព្រះមហាក្សត្រ

Institute of Technology of Cambodia

Department of Information of Technology

Group: I4-GIC-C

Course : Software Engineering

Topic : Report week 2 Library Management tracking System

Name Student	ID of student	Score
RIN NAIRITH	e20221557	…….
THOU LAIHENG	e20220843	…….

Lecturer: **Mr. ROUEN Pacharoth**

Academic year 2025-2026

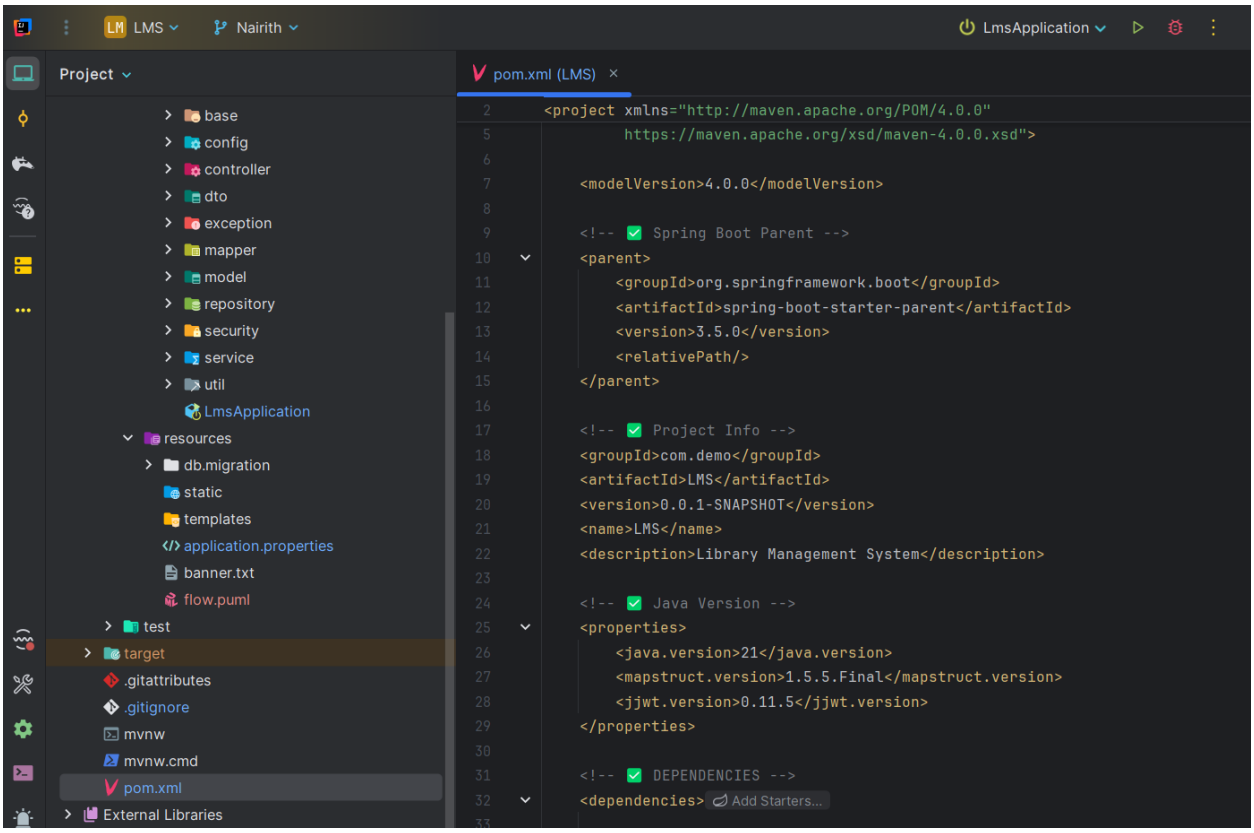
1. Project Setup and Environment Configuration

During Week 1, the initial setup of the **Library Management System (LMS)** backend was completed using **Spring Boot**. The project was created as a **Maven-based Spring Boot application**, ensuring modularity and ease of dependency management.

The following technologies and tools were configured:

- **Spring Boot 3.x** for application development
- **Java (JDK 21/25)** as the programming language
- **MySQL** as the relational database
- **Spring Data JPA (Hibernate)** for ORM and persistence
- **Flyway** for database version control and migration
- **Spring Security** for password encryption and future authentication
- **Swagger (OpenAPI)** for API documentation

The project follows a **layered and feature-based architecture**, separating concerns into controller, service, repository, DTO, mapper, entity, and configuration layers. This structure improves maintainability and scalability.



```
<!-- ✅ Java Version -->
<properties>
    <java.version>21</java.version>
    <mapstruct.version>1.5.5.Final</mapstruct.version>
    <jjwt.version>0.11.5</jjwt.version>
</properties>
```

2. Database Design and Table Definition

A conceptual database design was created to support a Library Management System.
For Week 2, the focus was placed on **user and role management**, which is a core foundation for authentication and authorization.

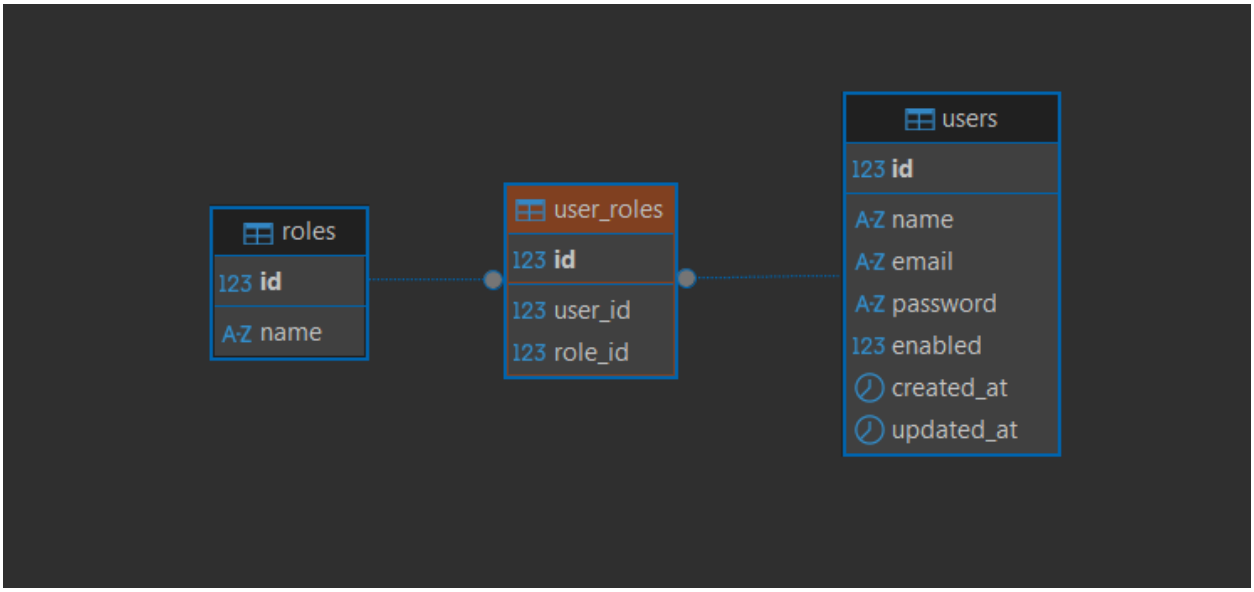
The following tables were defined and implemented:

- **Users** : Stores system user information such as name, email, and password.
- **Roles :** Stores predefined roles used for authorization.
- **user_roles :** A junction table implementing a many-to-many relationship between users and roles.

Relationships:

- One user can have multiple roles.
- One role can be assigned to multiple users.

Database constraints such as **primary keys**, **foreign keys**, **unique email constraint**, and **non-null role assignment** were applied to maintain data integrity.



3. CRUD Operations for User Management

Completed **CRUD (Create, Read, Update, Delete)** functionality was implemented for user management following RESTful API principles.

Implemented Operations:

- **Create User**
  - Validates input data using DTO validation.
  - Encrypts passwords using BCrypt.
  - Automatically assigns a default role (STUDENT).
- **Read Users**
  - Retrieve all users with their assigned roles.
  - Retrieve a single user by ID.
- **Update User**
  - Update name, email, and password.
  - Password is re-encrypted if changed.
- **Delete User** *(optional extension)*
  - Remove a user safely from the system.

The service layer handles business logic, repositories manage database access, and mappers convert entities into response DTOs. Transactions were managed correctly to prevent lazy-loading and data consistency issues.

4. Database Migration and Role Initialization

Flyway migrations were used to version-control the database schema.

Implemented Migrations:

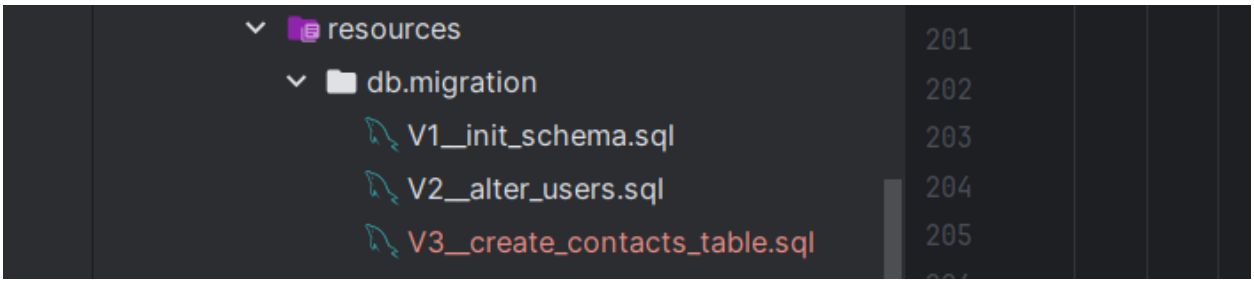
- **V1** – Initial schema creation (users, roles, user_roles)
- **V2** – Role initialization and adjustment

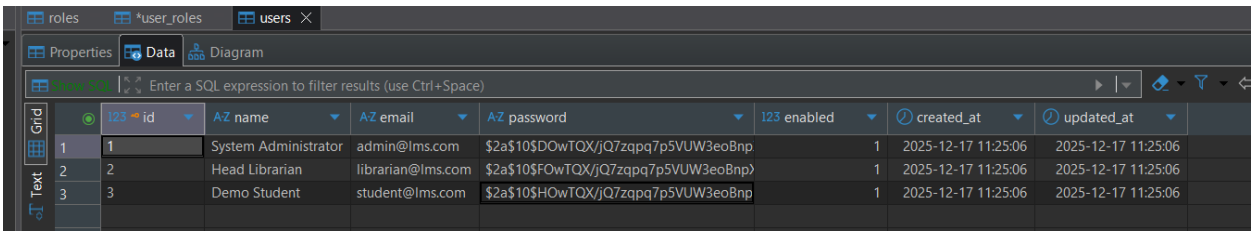
In **V2**, default roles were inserted:

- **ADMIN**

- **LIBRARIAN**
- **STUDENT**

This ensures that all environments (development, testing, production) have consistent role data without manual database setup. Role values were synchronized with Java enums using EnumType.STRING for safety and clarity.





5. Swagger API Documentation Integration

Swagger (OpenAPI) was integrated to provide **interactive API documentation**.

Features:

- Automatic documentation of all REST endpoints
- Ability to test APIs directly from the browser
- Clear visualization of request/response models

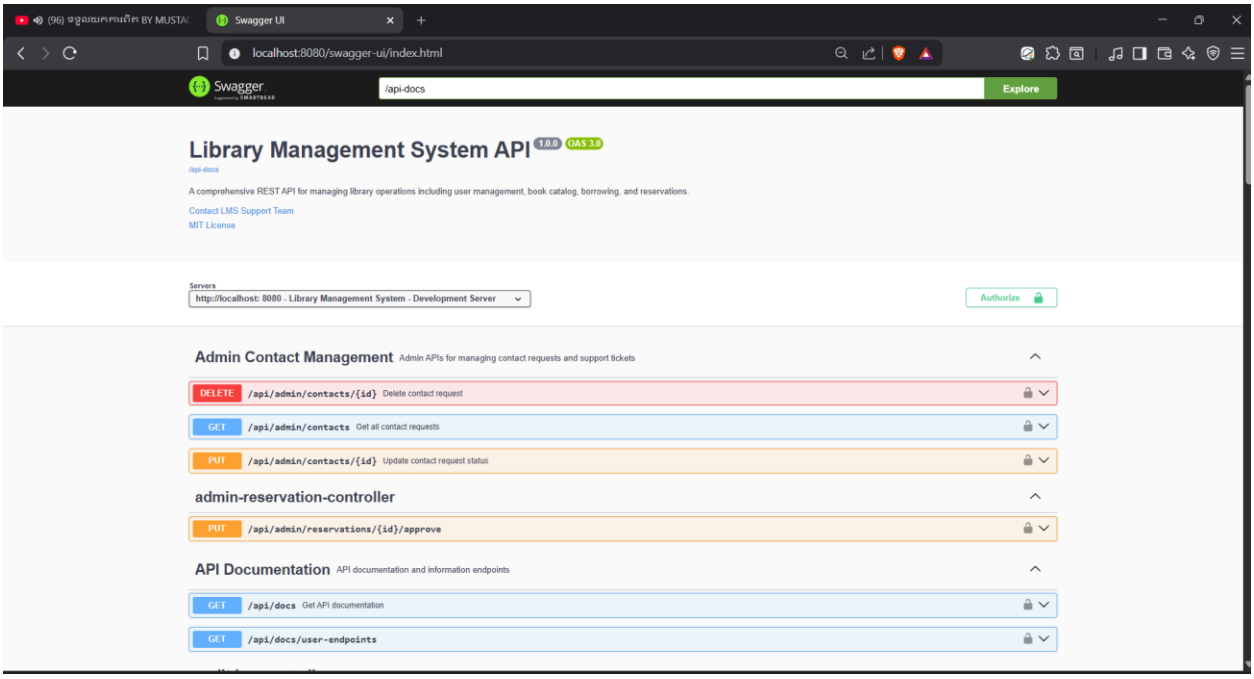
Swagger was configured to work alongside Spring Security in development mode, allowing unrestricted access to API documentation.

Access URL:

http://localhost:8080/swagger-ui.html

http://localhost:8080/api/-docs

This improves collaboration, testing efficiency, and project presentation quality.



Laiheng Part :

6. Add Dependencies:

- Add dependency spring security:
  For apply the protection in your web application like authentication, authorization, sessions, password, hashing, CORS, CSRF, JWT and more.
- Add dependency jwt(JSON web token): (secure and authentication (Suitable for REST API protection)):
  - jjwt-impl it is used jjwt-api like algorithm, parser,claim handling logic so that is why we need for runtime
  - jjwt-jackson use for serialize or deserialize
  - jjwt-api the one which use as public API

7. Configure the SecurityConfig:

- Add password method by using bcrypt

- @Configuration use for load the methods inside which has @Bean so all method that use annotation @Bean will be recognize in Spring Boot (Singleton register)
- @EnableMethodSecurity configure in SecurityConfig for enabling the annotation @PreAuthorize, @PostAuthorize, @Secure, @RolesAllowed (those annotation refer to the authentication and authorization to use specific method in our controller

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    private final JwtFilter jwtFilter;

    public SecurityConfig(JwtFilter jwtFilter) {
        this.jwtFilter = jwtFilter;
    }
```

```
 @Bean
    public PasswordEncoder passwordEncoder() {
        SecureRandom secureRandom;
        try{
            secureRandom = SecureRandom.getInstanceStrong();
        } catch (Exception e) {
            secureRandom = new SecureRandom();
        }
        return new BCryptPasswordEncoder(12, secureRandom);
    }
```

8. Testing password:

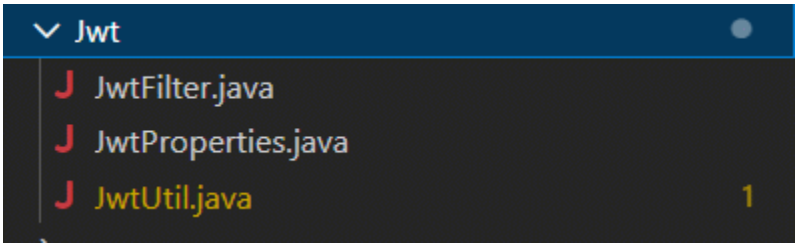
- Testing does the spring recognize the password method that we plugin in our SpringSecurity or not?
- Create AuthController and use a route for /api/login as testing password json by passing the request body with password
- But we need to configure the filter security as well when you want to make sure your route authorize.

```
@PostMapping("/api/login")
        public ResponseEntity<?> loginAPI(@RequestBody LoginRequest loginRequest)
{
        String email = loginRequest.getEmail() == null ? null :
loginRequest.getEmail().trim().toLowerCase();
```

```
        return userRepository.findByEmail(email)
            .filter(u -> encoder.matches(loginRequest.getPassword(), u.password))
            .<ResponseEntity<?>>map(u -> ResponseEntity.ok(new
AuthResponse(u.email, jwtUtil.generateToken(u.email))))
            .orElseGet(() -> ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                .body(Map.of("error", "Invalid credentials")));
        }
```

9. Create JwtUtils

- To generate token and extract username from the token and using @Component to make your spring boot recognize that util class.
- Generate SecretKey from secretkey of JwtProperties and then load the properties and create the expiry date for generate token.
- To verify the token, we need the key existing and parse with the token that user apply, and then extract the email




```
@Component
public class JwtUtil {
    private JwtProperties jwtProperties;
    private SecretKey key;

    public JwtUtil(JwtProperties jwtProperties) {
        this.jwtProperties = jwtProperties;
        this.key =
Keys.hmacShaKeyFor(Base64.getDecoder().decode(jwtProperties.getSecretKey()));
    }

    public String generateToken(String email){
        Date now = new Date();
        Date expiryDate = new Date(now.getTime() +
jwtProperties.getExpirationTime());
        return Jwts.builder()
                .subject(email)
                .issuedAt(now)
                .expiration(expiryDate)
```

```
                .signWith(key)
                .compact();
    }

    public String extractEmail(String token){
        return extractAllClaims(token).getSubject();
    }

    private Claims extractAllClaims(String token){
        return Jwts.parser()
                .verifyWith(key)
                .build()
                .parseSignedClaims(token)
                .getPayload();
    }
}
```

10. Modify the SecurityConfig

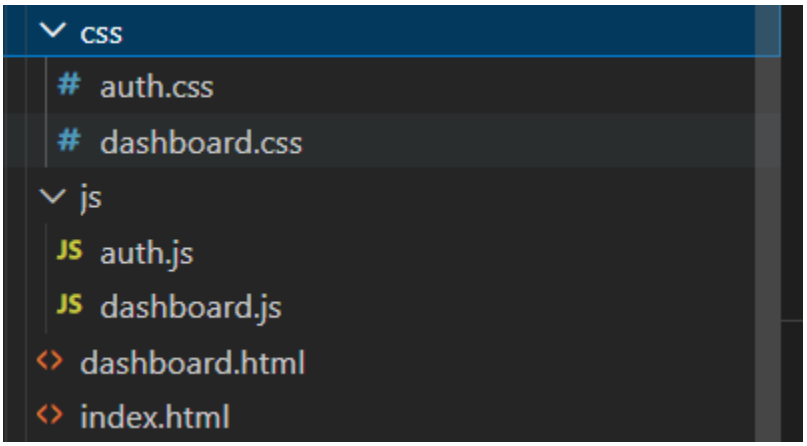
- Add authProvider into authenticationProvider method in httpSecurity that we created.
- Mapped the login page into that request
- Enable cors
- PermitAll route related to login, /api/login, /register
- /products protected authenticated
- Pass the filter jwt that we create to filter header if they use API

```
@Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .csrf(csrf -> csrf.disable()) // Disable CSRF for API development
            .authorizeHttpRequests(auth -> auth
                // Static resources & root - public access
                .requestMatchers(PathRequest.toStaticResources().atCommonLocation
s()).permitAll()
                // Swagger/OpenAPI endpoints - public access
                .requestMatchers("/api/login", "/login",
                "/register","/api/register").permitAll()
                // Actuator endpoints - public access
                .anyRequest().authenticated()
            )
            .addFilterBefore(jwtFilter,
```

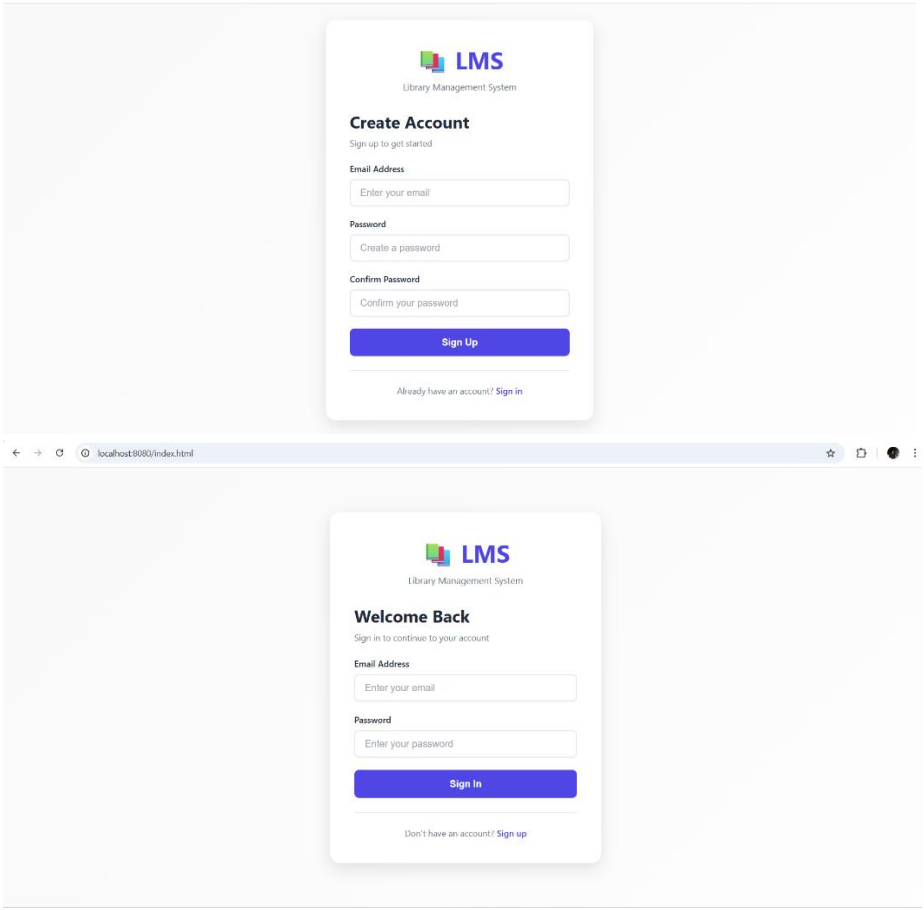
```
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFil
ter.class);

        return http.build();
    }
```

11. Create the UI for SignUp and LogIn



User Interface:



12. Conclusion

By the end of Week 2, the project achieved the following milestones:

- Successfully set up a Spring Boot backend project
- Designed and implemented core database tables with relationships
- Implemented full CRUD operations for user management
- Initialized roles using Flyway database migration
- Integrated Swagger for API documentation and testing
- Configure the SecurityConfig
- Create JwtUtils
- Modify the SecurityConfig
- Create UI for SignUp and Login

This week established a **strong and scalable foundation** for future features such as authentication (JWT), book management, borrowing workflows, and authorization control.

Planned Work for Week 3

- Integrating access control logic on the frontend to display elements relevant to the authenticated user's role
- Managing user session state
- Implementing the borrowing/returning logic and the overdue detection mechanism (calculating due dates and flagging late returns).
- Integrating access control logic on the frontend to display elements relevant to the authenticated user's role
- Handling book search, filtering, and inventory updates**.**