**Exercise 12: Online Bookstore - Securing RESTful Endpoints with Spring Security**

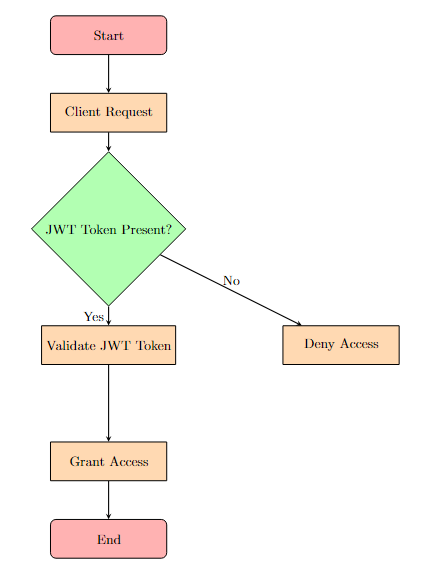
Business Scenario:

Secure your bookstore's RESTful endpoints using Spring Security with JWT-based authentication.

**Key Points of the Implementation:**

1. **Integrating Spring Security:**
   * Add the necessary Spring Security dependency to the pom.xml.
   * Create a security configuration class that extends WebSecurityConfigurerAdapter to customize security settings.
   * In the configuration, disable default CSRF protection since JWT is being used, and specify which endpoints require authentication.
2. **JWT Authentication:**
   * **JWT Utility Class:**
     + A utility class is created for generating, parsing, and validating JWT tokens.
     + The generateToken() method generates a JWT token using user details.
     + The validateToken() method ensures that the token is valid and hasn't expired.
   * **Authentication Filter:**
     + A custom filter (JwtAuthenticationFilter) is created to intercept requests and validate the JWT token.
     + If the token is valid, the user's authentication is set in the security context.
   * **UserDetailsService:**
     + Implement a custom UserDetailsService to load user-specific data, such as username, password, and roles from the database.
3. **CORS Configuration:**
   * Implement a global CORS configuration to allow cross-origin requests to the REST API.
   * Use the @CrossOrigin annotation on specific controllers or configure CORS globally through a CorsConfigurationSource bean.
4. **Securing Endpoints:**
   * In the security configuration, use .antMatchers() to specify which endpoints require authentication and which are publicly accessible (e.g., login, registration).
   * Use role-based access control by specifying .hasRole() for specific endpoints.
5. **Authentication and Authorization Flow:**
   * Upon login, the user provides credentials, which are authenticated by Spring Security.
   * A JWT token is generated and returned in the response.
   * For subsequent requests, the client includes the JWT token in the Authorization header. The custom filter validates the token, and if valid, the request is allowed to proceed.
6. **Exception Handling:**
   * Implement custom exception handling for authentication failures and unauthorized access, returning appropriate HTTP status codes and error messages.

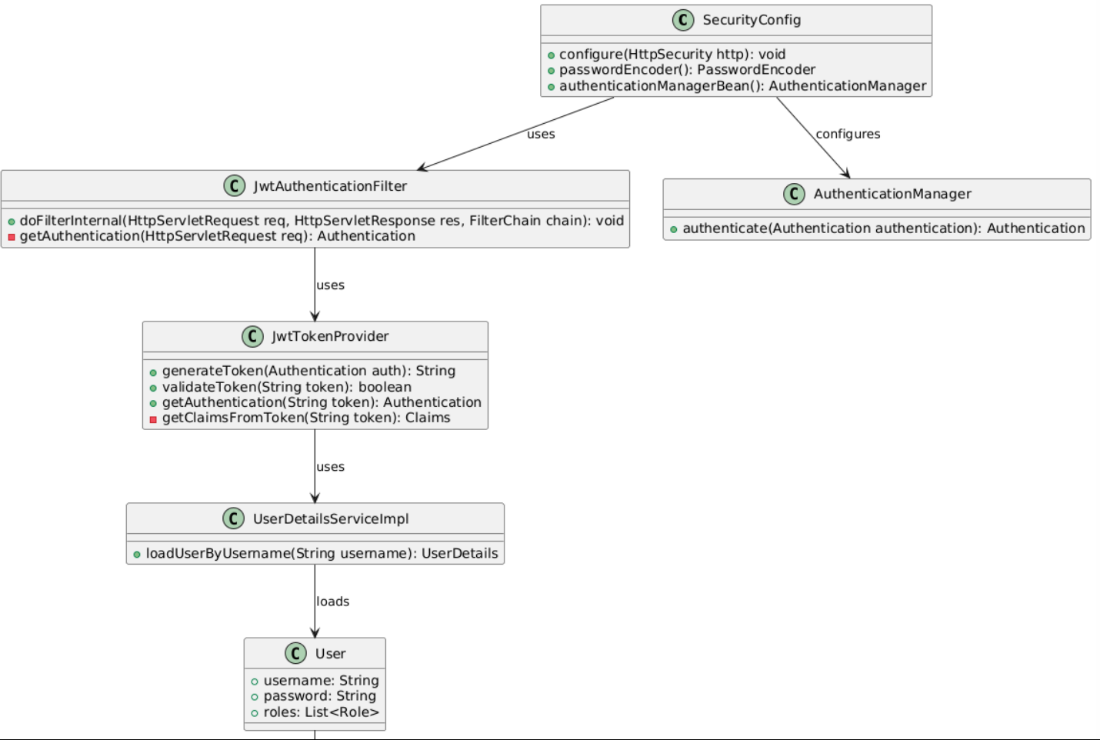
**FLOWCHART :**



**Explanation of Flowchart**

1. **Start**: The flowchart begins with the start node.
2. **Client Request**: This represents the initial HTTP request made by the client to access a resource in the online bookstore.
3. **JWT Token Present?**: A decision node checks if the JWT (JSON Web Token) is present in the Authorization header of the request.
   * **Yes**: If a JWT token is present, the flow proceeds to the "Validate JWT Token" step.
   * **No**: If a JWT token is not present, access is denied, and the flow proceeds to the "Deny Access" node.
4. **Validate JWT Token**: If the token is present, it is validated to ensure it is correctly signed and has not expired.
5. **Grant Access**: If the JWT token is valid, access is granted to the requested resource.
6. **Deny Access**: If the JWT token is missing or invalid, access is denied.
7. **End**: The flowchart concludes after either granting or denying access.

**CLASS DIAGRAM :**



**Explanation of Class Diagram:**

1. **SecurityConfig**: The main configuration class that sets up security configurations, such as HTTP security and authentication. It uses JwtAuthenticationFilter to apply JWT-based security and configures the AuthenticationManager.
2. **JwtAuthenticationFilter**: A filter that intercepts HTTP requests to check for a valid JWT token. It uses JwtTokenProvider to validate and extract authentication information from the token.
3. **JwtTokenProvider**: This class is responsible for generating and validating JWT tokens. It also extracts user information from the token to create an Authentication object.
4. **UserDetailsServiceImpl**: A custom implementation of UserDetailsService that loads user-specific data. It interacts with the User entity to retrieve user details based on the username.
5. **User**: Represents the user entity, containing the username, password, and a list of roles.
6. **Role**: Represents the roles assigned to users (e.g., ROLE\_USER, ROLE\_ADMIN).
7. **AuthenticationManager**: Manages the authentication process, typically used by SecurityConfig to handle authentication.