**Tutorial : Securing a REST API with Spring Security**

- Turcut Stefan-Adrian -

This tutorial explores how to secure a REST API using Spring Security in a Spring Boot application. It covers the concept of authentication and authorization, as well as demonstrates a working example implemented in Java.

**1. Introduction to Security in REST APIs**

Security is a critical aspect of any web application, especially when dealing with sensitive data or performing actions on behalf of users. REST APIs, being a popular way to interact with web services, require robust security measures to protect against unauthorized access and potential threats.

Spring Security provides a comprehensive framework for securing Java-based applications, including RESTful services. It offers features like authentication, authorization, and protection against common security vulnerabilities such as CSRF (Cross-Site Request Forgery) and XSS (Cross-Site Scripting).

**2. Understanding Authentication and Authorization**

Authentication: The process of verifying the identity of a user or system attempting to access the API. It involves validating credentials (e.g., username/password, tokens) provided by the client.

Authorization: Determining what actions a user or system is allowed to perform after successful authentication. It involves checking permissions and roles associated with the authenticated user.

**3. Implementation Overview**

A brief overview of the components involved:

* **Dependency**:
  + Inclusion of spring-boot-starter-security dependency in our project to leverage Spring Security features seamlessly.
* **User Roles and Permissions:** 
  + Definition of roles and permissions using enums. Each role may have multiple permissions associated with it.
* **Security Configuration:**
  + Configuration of Spring Security by extending WebSecurityConfigurerAdapter, defining authentication filters, session management policies, and URL access permissions.
* **JWT (JSON Web Token) Integration:**
  + Usage of JWT for stateless authentication. JWTs are self-contained tokens containing user information and are signed to ensure integrity.
* **Authentication Filter:**
  + Implementation of filters to extract JWT from incoming requests, validate it, and set the authenticated user's context.
* **JWT Issuer:**
  + Utility class to generate JWTs for authenticated users upon successful login.

A screenshot of a computer program

Description automatically generated**4. Step-by-Step Implementation**

* Step 0: Add required dependencies
  + Add most recent spring-boot-starter-security dependency.
* Step 1: Define User Roles and Permissions
  + Define roles (e.g., USER, ADMIN) and corresponding permissions (e.g., USER\_CREATE, ADMIN\_READ) using enums.
* Step 2: Configure Spring Security
  + A screen shot of a computer program

    Description automatically generatedAdd a Web Security Configuration class to customize security settings. Configure authentication filters, session management, and URL access permissions using method chaining.
* Step 3: Implement JWT Integration
  + Create utility classes for JWT generation and parsing. Use a secret key to sign and verify tokens.

A screen shot of a computer program

Description automatically generated

* Step 4: Implement Authentication Filter
  + Create a UserPrincipal class implementing the already existing UserDetails(from dependency) , set all boolean methods to true and update the rest as needed:
  + A screen shot of a computer program

    Description automatically generated
  + Create your custom token class extending AbstractAuthenticationToken and having the recently created UserPrincipal
  + A screen shot of a computer program

    Description automatically generated
  + Create a filter to intercept incoming requests, extract JWT, validate it, and set the authenticated user's context using Spring Security's SecurityContextHolder.

**A screenshot of a computer program

Description automatically generated**

* Step 5: Secure API Endpoints
  + Use annotations like @PreAuthorize to enforce role-based access control on API endpoints. Specify required roles or permissions for each endpoint.
  + To access those apis you now need a generated token of a user with those specific roles and authorities. (In headers there needs to be an “Authorization”: “Bearer {token}”)

**A screen shot of a computer program

Description automatically generated** **A screenshot of a computer

Description automatically generated**

**Conclusion**

Securing a REST API can be crucial in protecting sensitive data and ensure proper access control. Spring Security, with its robust features and easy integration, provides an excellent solution for implementing security in Java-based applications. Following this tutorial and leveraging the provided example code shows how to effectively secure RESTful services using Spring Security.