WEEK 10 COURSE CONTENT: SPRING SECURITY

TOPICS:

- Introduction to Spring Security
- User Authentication and Authorization
- **Securing RESTful APIs with Spring Security**
- Implementing Role-Based Access Control
- **Token-Based Authentication (JWT)**
- Securing Web Applications using Spring Security

LEARNING OUTCOME:

- Understand the concept of Spring Security
- Know the difference types of Authentication and Authorization strategy
- Securing Restful APIs with Spring Security
- Understanding and implementation of Role Based Access Control in Spring Security
- Securing applications, authenticating and authorizing users with JWT

DAY 1

Overview of Spring Security

Spring Security is a powerful framework that is widely used to enhance the security of web applications developed using the Spring framework. Its primary purpose is to provide a robust and flexible security solution for both authentication and authorization aspects of web applications.

Importance of Spring Security

Web application security is a critical concern in today's digital landscape. Spring Security plays a vital role in safeguarding your applications against common security threats such as unauthorized access, session management, cross-site request forgery (CSRF), and more. It helps you build secure and reliable applications, giving users peace of mind when interacting with your systems.

Core Concepts and Features of Spring Security

Spring Security is built around several core concepts and features:

- **Authentication**: This is the process of verifying the identity of a user. Spring Security supports various authentication mechanisms, including username/password, OAuth, and LDAP.
- **Authorization**: Once a user is authenticated, authorization determines what actions they are allowed to perform within the application. It enforces access control rules.
- **Security Filters**: Spring Security uses filters to intercept incoming requests and apply security checks. These filters can be customized to meet specific requirements.
- **User Details Service**: Spring Security interacts with a UserDetailsService to retrieve user information for authentication. Users and their roles can be stored in databases, LDAP, or other sources.
- **Session Management**: Spring Security manages user sessions, allowing you to configure session timeouts, concurrent session control, and more.
- **CSRF Protection**: Cross-Site Request Forgery attacks can be prevented using Spring Security's built-in CSRF protection.

Benefits of Using Spring Security in Spring Boot Applications

Spring Security seamlessly integrates with Spring Boot, providing several benefits:

- **Ease of Configuration**: Spring Boot's auto-configuration simplifies the setup of Spring Security, reducing the need for extensive XML or Java configuration.
- **Pre-built Security Features**: Spring Security comes with many pre-built security features, such as password hashing, session management, and authentication providers, saving development time and effort.
- **Customization**: While Spring Security offers sensible defaults, it is highly customizable to meet your application's specific security requirements.
- **Integration with Other Spring Projects**: Spring Security integrates smoothly with other Spring projects like Spring Data, Spring MVC, and Spring Cloud, creating a unified development environment.

Topic: User Authentication and Authorization

User Authentication and Authorization Concepts

User Authentication

User authentication is the process of verifying the identity of a user. It ensures that the person accessing the application is who they claim to be. Spring Security provides several authentication mechanisms:

- **Form-Based Authentication**: Users provide their credentials (username and password) via a login form.
- **OAuth Authentication**: Integrates with OAuth providers like Google, Facebook, or your custom OAuth server.
- **LDAP Authentication**: Authenticates users against an LDAP directory.
- **Custom Authentication Providers**: Allows you to implement custom authentication logic.

User Authorization

User authorization defines what actions authenticated users are allowed to perform within the application. Spring Security handles this by defining access control rules using expressions or annotations. You can specify who can access specific URLs, methods, or resources based on user roles and permissions.

Implementing User Authentication with Spring Security

To implement user authentication with Spring Security in a Spring Boot application, follow these steps:

- **Add Spring Security Dependency**: Include the Spring Security dependency in your project's **pom.xml** or **build.gradle** file.
- **Configure Security Settings**: Create a security configuration class that extends **WebSecurityConfigurerAdapter**. Override its methods to configure authentication and authorization settings.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
```

```
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .inMemoryAuthentication()
        .withUser("user").password("{noop}password").roles("USER");
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
            .and()
        .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
        .logout()
            .permitAll();
}
```

- **Create User Entities**: Define your user entities and roles. These can be stored in a database, LDAP, or any other data source.
- **User Interface**: Create login and logout pages or endpoints as needed for user interaction.
- **Secure Resources**: Use Spring Security annotations or expressions to secure specific endpoints or resources.

With Spring Security, you can achieve robust user authentication and authorization for your Spring Boot applications, ensuring the protection of sensitive data and features. Remember to continuously update and test your security measures to stay ahead of potential threats.

DAY 2

Securing RESTful APIs with Spring Security

Importance of Securing RESTful APIs

Securing RESTful APIs is crucial for several reasons:

- **Protecting Sensitive Data**: RESTful APIs often expose sensitive data and functionalities. Without proper security measures, this data can be vulnerable to unauthorized access or manipulation.
- **Compliance**: Many industries and applications must adhere to strict compliance standards (e.g., GDPR, HIPAA). Secure APIs help ensure compliance by safeguarding user data.
- **Preventing Unauthorized Access**: Unauthorized access to APIs can lead to data breaches, fraud, and other security issues. Proper security mechanisms help prevent these incidents.
- **Maintaining Trust**: Users and clients trust your API to keep their data safe. A breach can lead to a loss of trust and reputation damage.

Implementing Security Measures with Spring Security

Spring Security can be used to secure RESTful APIs effectively. Here's a high-level overview of the steps involved:

- **Add Spring Security Dependency**: Include the Spring Security dependency in your project.
- **Configure Security**: Create a security configuration class that extends **WebSecurityConfigurerAdapter**. Configure authentication and authorization rules, specifying which endpoints are protected and who can access them.

@Configuration

@EnableWebSecurity

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
                .and()
            .httpBasic(); // or use other authentication methods like JWT
    }
}
```

Secure Endpoints: Use method-level security annotations or expressions to secure individual endpoints within your REST API controllers.

```
@RestController
public class MyController {
    @GetMapping("/secure")
    @PreAuthorize("hasRole('ROLE_ADMIN')")
    public ResponseEntity<String> secureEndpoint() {
        return ResponseEntity.ok("This is a secure endpoint");
    }
}
```

- **Handle Authentication**: Configure authentication mechanisms, such as Basic Authentication or JWT, to verify the identity of users or clients.

- **Handle Authorization**: Define access control rules using annotations like **@PreAuthorize** or **@PostAuthorize** to control who can access specific API endpoints.

Topic: Implementing Role-Based Access Control

Role-Based Access Control (RBAC) and Its Significance

Role-Based Access Control (RBAC) is a security model that assigns roles to users or entities and restricts access based on those roles. It's significant in web application security because it:

- Simplifies Access Control: RBAC provides a structured way to manage access, making it easier to enforce security policies.
- Minimizes Errors: By grouping permissions into roles, RBAC reduces the chance of overlooking access control requirements.
- Enhances Scalability: As your application grows, RBAC scales well, making it manageable to add or remove users and their permissions.

Implementing RBAC with Spring Security

To implement RBAC using Spring Security, follow these steps:

- **Define Roles**: Define roles that represent different user access levels. For example, "ROLE_ADMIN" and "ROLE_USER."
- **Assign Roles to Users**: Associate roles with users during user registration or through an administrative interface.

```
@Entity
public class User {
   // ...
   @ManyToMany(fetch = FetchType.EAGER)
   @JoinTable(name = "user_roles", joinColumns = @JoinColumn(name = "user_id"), inverseJoinColumns = @JoinColumn(name = "role_id"))
   private Set<Role> roles;
```

}

Restrict Access: Use annotations like **@PreAuthorize** or **@PostAuthorize** to control access to specific methods or endpoints based on user roles.

```
@RestController
public class MyController {
   @GetMapping("/admin")
   @PreAuthorize("hasRole('ROLE_ADMIN')")
   public ResponseEntity<String> adminEndpoint() {
      return ResponseEntity.ok("This is an admin endpoint");
   }
}
```

- **Customize RBAC Logic**: Spring Security allows you to customize RBAC logic, such as handling role hierarchies or implementing custom access decision logic.

By implementing RBAC with Spring Security, you can effectively manage user access control in your web application, ensuring that users only have access to the resources and functionality appropriate to their roles.

DAY 3

Introduction to Token-Based Authentication

Token-based authentication is a popular method for securing stateless applications like RESTful APIs. Unlike traditional session-based authentication, token-based authentication does not require storing session state on the server, making it highly scalable and suitable for microservices architectures.

JSON Web Tokens (JWT) for Authentication

JSON Web Tokens (JWT) are a compact, URL-safe means of representing claims to be transferred between two parties. In the context of authentication, JWTs are used to securely transmit information between a client and a server.

A JWT typically consists of three parts:

- **Header**: Contains metadata about the token, such as the type of token and the signing algorithm being used.
- **Payload**: Contains claims, which are statements about an entity (typically, the user) and additional data. Claims can include user information, roles, and expiration date.
- **Signature**: Ensures the integrity of the token and verifies that it wasn't tampered with during transmission.

Using JWT for Authentication

JWTs can be used for authentication as follows:

- **User Authentication**: When a user logs in, the server generates a JWT containing user claims and signs it with a secret key. This JWT is sent to the client.
- **Token Storage**: The client stores the JWT, typically in local storage or a cookie.
- **Authorization**: For each subsequent request, the client includes the JWT in the request header. The server validates the JWT's signature and uses the claims to authorize the request.

Implementing Token-Based Authentication with Spring Security, SecurityFilterChain, and OncePerRequestFilter

To implement token-based authentication using Spring Security with the **SecurityFilterChain** and **OncePerRequestFilter**, follow these steps:

- **Add Dependencies**: Include Spring Security and JWT dependencies in your project.

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
```

```
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.5</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.5</version>
</dependency>
```

Configure SecurityFilterChain: Create a **SecurityFilterChain** bean in your Spring Security configuration to specify the security filter chain, including the **OncePerRequestFilter** for JWT authentication.

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {
   private final UsersAuthenticationFilter usersAuthenticationFilter;
   private final AuthenticationProvider authenticationProvider;

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws
Exception {
      httpSecurity.csrf().disable()
            .authorizeHttpRequests(auth -> auth
                  .antMatchers("/login/**")
                  .permitAll()
                  .anyRequest()
                  .authenticated())
            .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
            .authenticationProvider(authenticationProvider)
```

```
            .addFilterBefore(usersAuthenticationFilter,
UsernamePasswordAuthenticationFilter.class)
            .httpBasic(Customizer.withDefaults());

      return httpSecurity.build();

   }

}
```

Create JwtAuthenticationFilter: Implement a custom **OncePerRequestFilter** for JWT authentication. This filter will extract and validate JWT tokens from incoming requests.

```
@Component
@RequiredArgsConstructor
public class UsersAuthenticationFilter extends OncePerRequestFilter {
   @Autowired
   private final JWTService jwtService;

   private final UserDetailsService userDetailsService;



   @Override
   protected void doFilterInternal(@NonNull HttpServletRequest request,
                           @NonNull HttpServletResponse response,
                           @NonNull FilterChain filterChain) throws ServletException,
IOException {
      //String to hold the request header
      final String authenticationHeader = request.getHeader("Authorization");

      //String to hold the jwt token
      final String jwtToken;

      if( authenticationHeader == null || !authenticationHeader.startsWith("Bearer ") ){
         filterChain.doFilter(request, response);
         return;
```

```
        }
        //Extract token from header
        jwtToken = authenticationHeader.substring(7);

        //Extract username/email from token
        String username = jwtService.extractUsername(jwtToken);

        //Validate the username
        if( username != null && SecurityContextHolder.getContext().getAuthentication() ==
null){

            //Load UserDetails using the userDetailsService.loadUserByUsername(email)
method
            UserDetails userDetails = userDetailsService.loadUserByUsername(username);

            if( jwtService.isTokenValid(jwtToken, userDetails)){

                //Create a UsernamePasswordAuthenticationToken
                UsernamePasswordAuthenticationToken
usernamePasswordAuthenticationToken =
                        new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());

                //SetDetails on usernamePasswordAuthenticationToken with a
WebAuthenticationDetailsSource().buildDetails(HttpServletRequest)
                usernamePasswordAuthenticationToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

                //Set SecurityContextHolder with the credentials

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticatio
nToken);

            }

            filterChain.doFilter(request, response);
        }

    }
```

```
}
```

Protect Endpoints: Use method-level security annotations to protect specific endpoints.

```
@GetMapping("/decrypted/{id}")
@ResponseBody
@PreAuthorize("USER")
public List<UserActivity> getAllDecryptionActivities(@PathVariable Long id){
   Users user = userRepository.findById(id).orElseThrow();
   return userActivityService.findByUser(user).stream()
         .filter(userActivity -> userActivity.getActivity().equals(Activity.DECRYPT))
         .sorted((a, b) -> b.getDate().compareTo(a.getDate()))
         .collect(Collectors.toList());
}

@PostMapping("/encrypt/{id}")
@PreAuthorize("USER")
public ResponseEntity<UserActivity> postEncryptionActivity(@RequestBody @Valid
UserActivity userActivity,
                                          @PathVariable Long id ) {
   Users user = userRepository.findById(id).orElseThrow();
   userActivity.setUser(user);
   userActivity.setActivity(Activity.ENCRYPT);
   userActivity.setDate(LocalDate.now());
   return userActivityService.postActivity(userActivity);
}
```

By implementing token-based authentication with JWT and Spring Security using **`SecurityFilterChain`** and **`OncePerRequestFilter`**, you can secure your stateless applications efficiently while providing a seamless authentication experience for your users. This approach allows you to customize the authentication process and integrate it smoothly into your application's security pipeline.

DAY 4

- Provide hands-on coding exercises and examples to reinforce understanding.
- Encourage students to work on a sample project that incorporates Spring Security.

- Discuss real-world scenarios and challenges related to web application security.
- Conduct interactive discussions and Q&A sessions to address students' questions and concerns.