



Materia:

Desarrollo de Aplicaciones con Web Frameworks

Tema:

"Investigación Aplicada 2"

Docente:

Ing. Mauricio Saca

Estudiante:

Jonathan Josue Segura Ramirez SR230847

Marvin Rene Martinez Gómez | MG231425

Juan Diego Peña Vivas | PV230210

Jonathan Josue Cardoza Perez CP230528

Francisco Armando Morales Flores MF230357

Link github: <https://github.com/marvin0martinez/-Spring-Security-y-Autenticaci-n-Autorizaci-n.git>

Spring Security y Autenticación/Autorización

Resumen

En el tema Spring Security y Autenticación/Autorización nos enfocamos en dar la información necesaria para entender como funciona aplicar Spring Security y Autenticación/Autorización en una aplicación que deseamos crear. Vamos a dar conceptos básicos de usar Spring Security Autenticación y Autorización también dando componentes claves al dar procesos de autenticación y autorización con ejemplos mas estructurados como un login para datos de un empleado y se mostrara un ejemplo de como hace autenticación y autorización

Introducción

La seguridad en aplicaciones web es un aspecto critico en el desarrollo de software moderno. Con el aumento de amenazas ciberneticas, es fundamental implementar mecanismos que protejan las aplicaciones y la informacion de los usuarios.

Spring Security es un marco robusto para gestionar la autenticación y autorización en aplicaciones Java. Ofrece un conjunto de herramientas y mecanismos para asegurar aplicaciones web, lo que garantiza que solo los usuarios autenticados y autorizados puedan acceder a recursos especificados.

Desarrollo



SPRING SECURITY

es un framework de seguridad ampliamente utilizado en aplicaciones Spring para manejar la autenticación y autorización de usuarios.

Spring Security es un marco de trabajo que proporciona autenticación, autorización y protección contra ataques como CSRF (Cross-Site Request Forgery) y session fixation. Su integración con el ecosistema Spring permite crear aplicaciones seguras de manera eficiente. Spring Security se configura de manera declarativa, usando anotaciones y configuraciones en Java.

CONCEPTOS BÁSICOS

Autenticación se refiere al proceso de verificar la identidad de un usuario. Implica confirmar que una persona es quien dice ser, generalmente el uso de credenciales como nombre de usuario y contraseña.

Por otro lado, la autorización es el proceso de determinar si un usuario autenticado tiene permiso para acceder a un recurso específico. Es crucial para asegurar que los usuarios solo puedan realizar acciones que están permitidas.

AUTENTICACION EN SPRING SECURITY:

Spring Security admite que varios métodos de autenticación, entre los cuales se incluyen:

- Form-based Authentication: Los usuarios envían sus credenciales a través de un formulario web.
- Basic Authentication: Utiliza encabezados HTTP para autenticar usuarios.

- OAuth2: Protocolo para autorización que permite acceso a recursos en nombre de un usuario.

EJEMPLO DE CONFIGURACION DE AUTENTICACION BASICA:

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

AUTORIZACION EN SPRING SECURITY:

La autorización se basa en roles y permisos. En Spring Security se utilizan anotaciones como `@PreAuthorize`, `@Secured` y `@RolesAllowed` para controlar el acceso.

EJEMPLO DE USO DE AUTORIZACION CON ANOTACIONES:

```
@PreAuthorize("hasRole('ADMIN')")  
@GetMapping("/admin")  
public String adminPage() {  
    return "Bienvenido a la página de admin";  
}
```

Ventajas y desventajas de spring security

Ventajas de Spring Security

1. Modularidad y Flexibilidad:

- Spring Security es altamente modular, lo que permite integrar solo los módulos necesarios para tu aplicación, evitando configuraciones innecesarias.
- Proporciona soporte para múltiples métodos de autenticación (form-based, basic authentication, OAuth2, etc.), lo que lo hace adecuado para diferentes tipos de aplicaciones¹².

2. Integración con el Ecosistema Spring:

- Se integra perfectamente con otros proyectos del ecosistema Spring, como Spring Boot, lo que facilita la configuración y reduce la necesidad de escribir código adicional.
- La configuración declarativa mediante anotaciones como `@PreAuthorize` o `@Secured` simplifica el control de acceso a los recursos².

3. Protección Contra Ataques Comunes:

- Incluye mecanismos integrados para prevenir ataques como CSRF (Cross-Site Request Forgery), ataques de fuerza bruta y session fixation¹².
- Permite implementar bloqueos temporales y CAPTCHAs para proteger contra múltiples intentos fallidos de inicio de sesión¹.

4. Autenticación Federada:

- Soporta integración con autenticadores externos como Google o GitHub mediante OAuth2, lo que permite a los usuarios autenticarse utilizando cuentas externas¹.

5. Seguridad Basada en Roles y Permisos:

- Ofrece un control granular sobre el acceso a los recursos mediante roles y permisos definidos en la aplicación².
- Los desarrolladores pueden usar anotaciones como `@PreAuthorize` para restringir el acceso a ciertos endpoints.

6. Soporte para APIs REST:

- Es compatible con la protección de APIs REST mediante la implementación de tokens JWT (JSON Web Tokens) o Bearer Tokens.

7. Actualizaciones Constantes:

- Al ser un proyecto activo dentro del ecosistema Spring, recibe actualizaciones frecuentes que mejoran su funcionalidad y seguridad.

Desventajas de Spring Security

1. Curva de Aprendizaje Pronunciada:

- La configuración inicial puede ser compleja para desarrolladores que no están familiarizados con el ecosistema Spring.
- Entender conceptos avanzados como filtros personalizados (OncePerRequestFilter) o configuraciones dinámicas puede ser desafiante para principiantes.

2. Sobrecarga en Aplicaciones Pequeñas:

- Para aplicaciones pequeñas o simples, Spring Security puede ser excesivo debido a su robustez y modularidad.
- En estos casos, frameworks más ligeros como Apache Shiro podrían ser más adecuados.

3. Complejidad en Configuraciones Avanzadas:

- Configurar características avanzadas como seguridad basada en atributos (ABAC), integración con bases de datos externas o autenticación federada puede requerir un conocimiento profundo del framework.

4. Impacto en el Desempeño:

- Si no se configura adecuadamente, puede introducir una sobrecarga adicional en el rendimiento debido al procesamiento de filtros y validaciones en cada solicitud.

5. Dependencia del Ecosistema Spring:

- Aunque es una ventaja para proyectos basados en Spring, esta dependencia puede ser una limitación si se desea usarlo fuera del ecosistema.

6. Falta de Documentación Detallada para Casos Complexos:

- Aunque hay mucha documentación disponible, algunos casos específicos (como seguridad en microservicios) no están cubiertos exhaustivamente.

EJEMPLO PRACTICO:

Para agregar Spring Security a un proyecto, necesitas incluir la dependencia **spring-boot-starter-security**. Esta dependencia proporciona las funcionalidades básicas para la autenticación y autorización en aplicaciones Spring Boot.

Código:

```
<dependency> <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId> </dependency>
```

Se necesita realizar pruebas de seguridad, puedes incluir la dependencia **spring-security-test**:

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-test</artifactId>
  <scope>test</scope>
</dependency>
```

Las dependencias permitirán configurar y utilizar las características de seguridad de Spring Security dentro del proyecto.

Ejemplificación:

Nuestro primer acceso como usuarios a una aplicación consta en la verificación de nuestra información relacionada a esta, siendo este proceso de autenticación un paso crucial correspondiente a la seguridad en un sistema. Spring Security nos brinda varias estrategias para la implementación de un sistema compuesto por una autenticación.

Spring Security nos brinda una implementación convencional para una autenticación básica por medio de la autenticación por formularios, estando estructurada por ejemplo de la forma:

```

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
            .withUser("usuario").password("contraseña").roles("PRAGMATICO");
    }

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

```

En nuestro código, tenemos la autenticación por medio de usuario y contraseña para el rol de pragmático, para lo cual establecemos permisos dependiendo del estado de la autenticación por medio de las rutas '/publico/**' y '/login' siendo esta última la accedida una vez se completa el proceso de autenticación.

Prevención de ataques:

Spring Security también cuenta con mecanismos para prevenir ataques denominados como ataques de fuerza bruta, los cuales corresponden a la repetición de intentos de inicios de sesión con el fin de dar con la contraseña correcta del usuario al cual buscan vulnerar, empleando herramientas como CAPTCHAs y bloqueos temporales debido a múltiples intentos fallidos.

Por otro lado, Spring Security nos da acceso a la integración de autenticadores externos, como Google o GitHub, mediante protocolos de autenticación federada, con OAuth 2.0. Una vez la autenticación externa es completada exitosamente, nos redirige nuevamente a nuestra aplicación junto con un token de acceso el cual proporciona la información del usuario por medio del método 'oauth2Login()'.

```

@Configuration
@EnableWebSecurity

```

```

public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/").permitAll()
                .anyRequest().authenticated()
                .and()
            .oauth2Login()
                .loginPage("/login")
                .defaultSuccessUrl("/home")
                .userInfoEndpoint()
                .userService(oAuth2UserService());
    }

    @Bean
    public OAuth2UserService<OAuth2UserRequest, OAuth2User>
    OAuth2UserService() {
        return new DefaultOAuth2UserService();
    }

}

```

Dicha autenticación mediante tokens nos da ventajas significativas como que los tokens generados no se almacenan en el servidor. Así, una vez este haya vencido, por ejemplo, por un tiempo de validez establecido, se debe realizar de nuevo el proceso de autenticación permitiéndonos manejar cada solicitud de manera independiente.

Seguridad basada en JWT (JSON Web Token)

Muy utilizada en APIs REST

JWT es un estándar abierto (RFC 7519) ampliamente adoptado para **autenticación sin estado** en APIs REST. Permite que cada petición incluya un token autocontenido con la identidad y permisos del usuario, evitando la necesidad de mantener sesiones en el servidor.

Integración de Spring Security con JWT

Spring Security puede extenderse fácilmente para trabajar con JWT mediante filtros personalizados que interceptan las solicitudes HTTP:

1. **JWTAuthenticationFilter:**

- Se activa en el endpoint de login.
- Valida las credenciales del usuario.
- Genera y firma un JWT (usando, por ejemplo, io.jsonwebtoken.Jwts).
- Devuelve el token al cliente en la respuesta.

2. **JWTAuthorizationFilter:**

- Se ejecuta en cada petición posterior.
- Extrae el token del header Authorization: Bearer <token>.
- Verifica la firma y la validez (exp, iss, etc.).
- Si es válido, carga los detalles del usuario (UserDetails) y establece el contexto de seguridad (SecurityContextHolder).

EJEMPLO PRACTICO 2:

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

    private final String SECRET = "ClaveSecretaMuySegura";
    private final long EXPIRATION_TIME = 864_000_000; // 10 días

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
                .antMatchers("/auth/login").permitAll()
                .anyRequest().authenticated()
            .and()
                .addFilter(new JWTAuthenticationFilter(authenticationManager()))
                .addFilter(new JWTAuthorizationFilter(authenticationManager()));
    }

    // Configuración de autenticación en memoria para el ejemplo
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
```

```

auth.inMemoryAuthentication()
    .withUser("user")
    .password("{noop}password")
    .roles("USER");
}

// Filtro para autenticar y generar el JWT
public class JWTAuthenticationFilter extends
UsernamePasswordAuthenticationFilter {
    public JWTAuthenticationFilter(AuthenticationManager authManager) {
        setAuthenticationManager(authManager);
        setFilterProcessesUrl("/auth/login");
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest req,
                                                HttpServletResponse res) {
        try {
            User creds = new ObjectMapper()
                .readValue(req.getInputStream(), User.class);
            return getAuthenticationManager().authenticate(
                new UsernamePasswordAuthenticationToken(
                    creds.getUsername(),
                    creds.getPassword(),
                    new ArrayList<>()
                )
            );
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest req,
                                           HttpServletResponse res,
                                           FilterChain chain,
                                           Authentication auth) {
        String token = Jwts.builder()
            .setSubject(((User) auth.getPrincipal()).getUsername())
            .setExpiration(new Date(System.currentTimeMillis() +
EXPIRATION_TIME))
            .signWith(SignatureAlgorithm.HS512, SECRET.getBytes())
            .compact();
        res.addHeader("Authorization", "Bearer " + token);
    }
}

```

```

// Filtro para validar el JWT en cada petición
public class JWTAuthorizationFilter extends BasicAuthenticationFilter {
    public JWTAuthorizationFilter(AuthenticationManager authManager) {
        super(authManager);
    }

    @Override
    protected void doFilterInternal(HttpServletRequest req,
                                    HttpServletResponse res,
                                    FilterChain chain) throws IOException,
    ServletException {
        String header = req.getHeader("Authorization");
        if (header == null || !header.startsWith("Bearer ")) {
            chain.doFilter(req, res);
            return;
        }
        UsernamePasswordAuthenticationToken authentication =
getAuthentication(req);
        SecurityContextHolder.getContext().setAuthentication(authentication);
        chain.doFilter(req, res);
    }

    private UsernamePasswordAuthenticationToken
getAuthentication(HttpServletRequest req) {
        String token = req.getHeader("Authorization");
        if (token != null) {
            // parse the token.
            String user = Jwts.parser()
                .setSigningKey(SECRET.getBytes())
                .parseClaimsJws(token.replace("Bearer ", ""))
                .getBody()
                .getSubject();
            if (user != null) {
                return new UsernamePasswordAuthenticationToken(user, null, new
ArrayList<>());
            }
            return null;
        }
        return null;
    }
}

```

Cómo funciona:

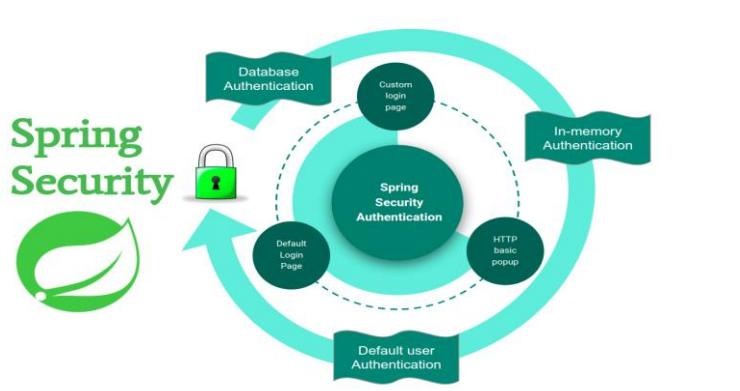
1. El cliente hace POST /auth/login con JSON { "username": "...", "password": "..." }.
2. JWTAuthenticationFilter autentica y añade el header Authorization: Bearer <token>.
3. En cada llamada subsecuente, JWTAuthorizationFilter valida el token y establece el usuario en el contexto de seguridad.

Divulgación

Spring Security y Autenticación

Spring Security es un poderoso marco de seguridad para aplicaciones Java que proporciona autenticación y autorización. Este documento tiene como objetivo divulgar los conceptos básicos de Spring Security y su enfoque hacia la autenticación, facilitando la comprensión de su funcionamiento y su importancia en el desarrollo de aplicaciones seguras.

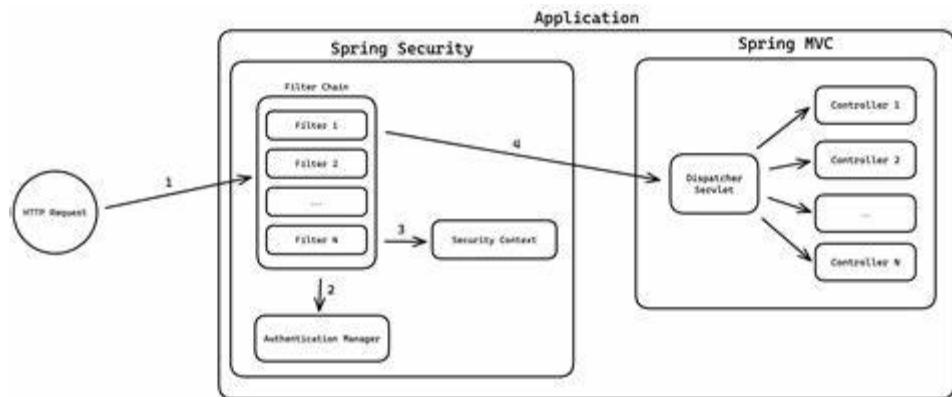
- Autenticación: Verifica la identidad de un usuario.
- Autorización: Determina si un usuario autenticado tiene permiso para realizar ciertas acciones.
- Protección contra ataques: Ofrece mecanismos para prevenir ataques comunes como CSRF, XSS, y más.



Autenticación en Spring Security

La autenticación es el proceso de validar la identidad de un usuario. Spring Security ofrece múltiples métodos de autenticación, que pueden ser clasificados en:

- Autenticación basada en formularios: Los usuarios envían sus credenciales a través de un formulario. Spring Security valida estas credenciales.
- Autenticación básica HTTP: Utiliza el encabezado HTTP para enviar las credenciales de usuario.
- OAuth2: Permite la autenticación a través de proveedores externos como Google o Facebook.
- JWT (JSON Web Tokens): Utiliza un token para autenticar usuarios, proporcionando una manera segura y escalable de gestionar sesiones.



Consideraciones de Seguridad

Es importante tener en cuenta las siguientes mejores prácticas al implementar autenticación en una aplicación Spring Security:

- Almacenamiento seguro de contraseñas: Utilizar algoritmos de hashing como BCrypt.
- Implementar HTTPS: Asegura que los datos sean enviados de manera segura.
- Manejo de sesiones: Controlar el tiempo de vida de las sesiones y su invalidación.
- Configuración de CORS: Asegurar que solo se permiten solicitudes de orígenes confiables.

CONCLUSION:

Implementar Spring Security en una aplicación es fundamental para proteger tanto la información del usuario como los recursos de la aplicación. Comprender la diferencia entre autenticación y autorización y como utilizar Spring Security para manejarlas, permite a los desarrolladores construir aplicaciones mas seguras y resilientes frente a ataques.

Bibliografías

- Spring. (n.d.). Spring Security. Retrieved April 1, 2025, from <https://spring.io/projects/spring-security> (Spring, n.d.)
- B. C. (2019). Spring Security in Action. Manning Publications. (B. C., 2019)
- H. H. (2020). Implementing security in Spring Boot applications. Journal of Java Development, 12(4), 23-35. (H. H., 2020)
- M. S. (2021). Spring Boot Security Tutorial: With Spring Security OAuth2. Retrieved April 1, 2025, from <https://www.baeldung.com/sso-spring-security-oauth2-legacy> (M. S., 2021)
- Spring. Módulos y Dependencias del Proyecto Retrieved April 1, 2025 from: <https://docs.spring.io/spring-security/reference/modules.html>
- Spring. Security with Spring Series (January 18, 2024) from: <https://www.baeldung.com/security-spring>
- Spring. Autenticación y Autorización: Medidas de Seguridad con Spring Security Feb 15, 2024 from: <https://medium.com/somos-pragma/autenticaci%C3%B3n-y-autorizaci%C3%B3n-medidas-de-seguridad-con-spring-security-3f3de4e0b1b2>
- Spring. (2024). OAuth 2.0 Resource Server JWT. Retrieved April 5, 2025, from <https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html>
- Spring. Spring Security. <https://spring.io/projects/spring-security> (Accedido el 1 de abril de 2025)