Spring Security

¿Qué es Spring Security?

- Spring Security es un módulo de Spring que permite proteger aplicaciones web (aplicaciones monolíticas con JSP o Thymeleaf) o apis con mecanismos de autenticación y autorización.

- Spring Security nos proporciona una solución robusta y confiable para implementar medidas de seguridad a nivel de aplicación.

- Nos proporciona mecanismos de autenticación y autorización para implementar en nuestra aplicación, a través de diferentes estrategias, por ejemplo:
  - LDAP
  - Usuario y Contraseña (In Memory, JDBC)
  - OAUTH 2.0

Conceptos Principales

- Principal: representa al usuario, dispositivo o sistema que podría realizar una acción dentro de la aplicación
- Credenciales: son claves de identificación que un principal usa para confirmar su identidad
- Autenticación: es el proceso de verificar la validez de las credenciales del principal
- Autorización: es el proceso de decidir si un usuario autenticado puede realizar una determinada acción dentro de la aplicación
- Elemento protegido: describe cualquier recurso que se esté protegiendo.

Proceso general de Spring Security

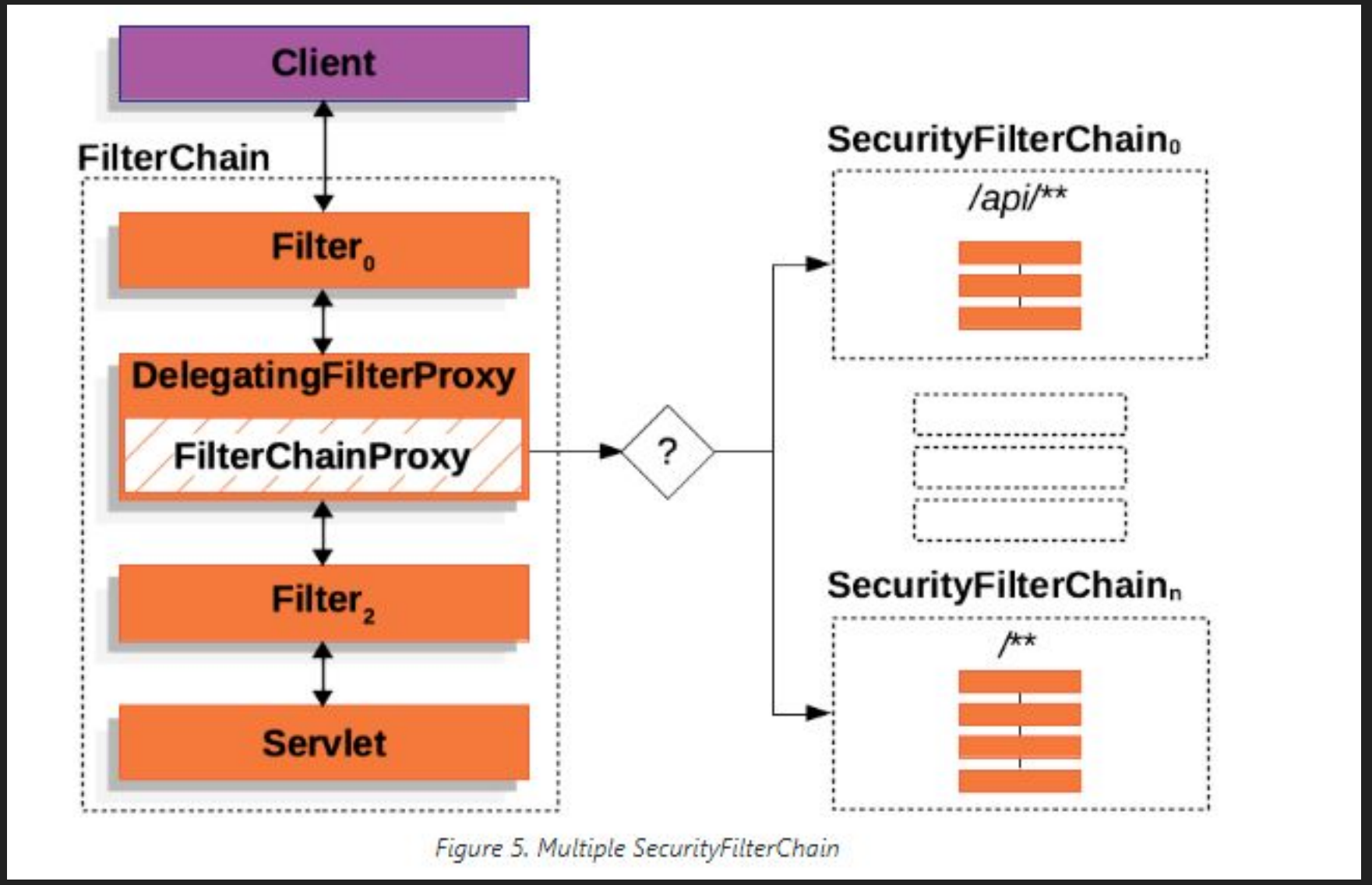


Figure 5. Multiple SecurityFilterChain

Filtros

- Los Filtros son interceptores de peticiones http que hacen algún tipo de comprobación sobre la misma.
- El orden de los filtros de seguridad es importante
- No es necesario conocer el orden los filtros de seguridad salvo en casos puntales.
- Algunos filtros importantes:
  - BasicAuthenticationFilter
  - UsernamePasswordAuthenticationFilter
  - DefaultLoginPageGeneratingFilter
  - DefaultLogoutPageGeneratingFilter
  - FilterSecurityInterceptor

Stateful vs Stateless

stateful

(Seguridad basada en sesiones)

Implica mantener un estado en el servidor para cada usuario que interactúa con la aplicación.

Eficaz para mantener información del usuario en el servidor y administrar su sesión.

Requiere el almacenamiento y la gestión de sesiones en el servidor, lo que puede generar problemas de escalabilidad y rendimiento en aplicaciones con muchos usuarios concurrentes.

stateless

(Seguridad basada en tokens de autenticación)

Se basa en la idea de que cada solicitud que realiza el cliente contiene toda la información necesaria para que el servidor la procese

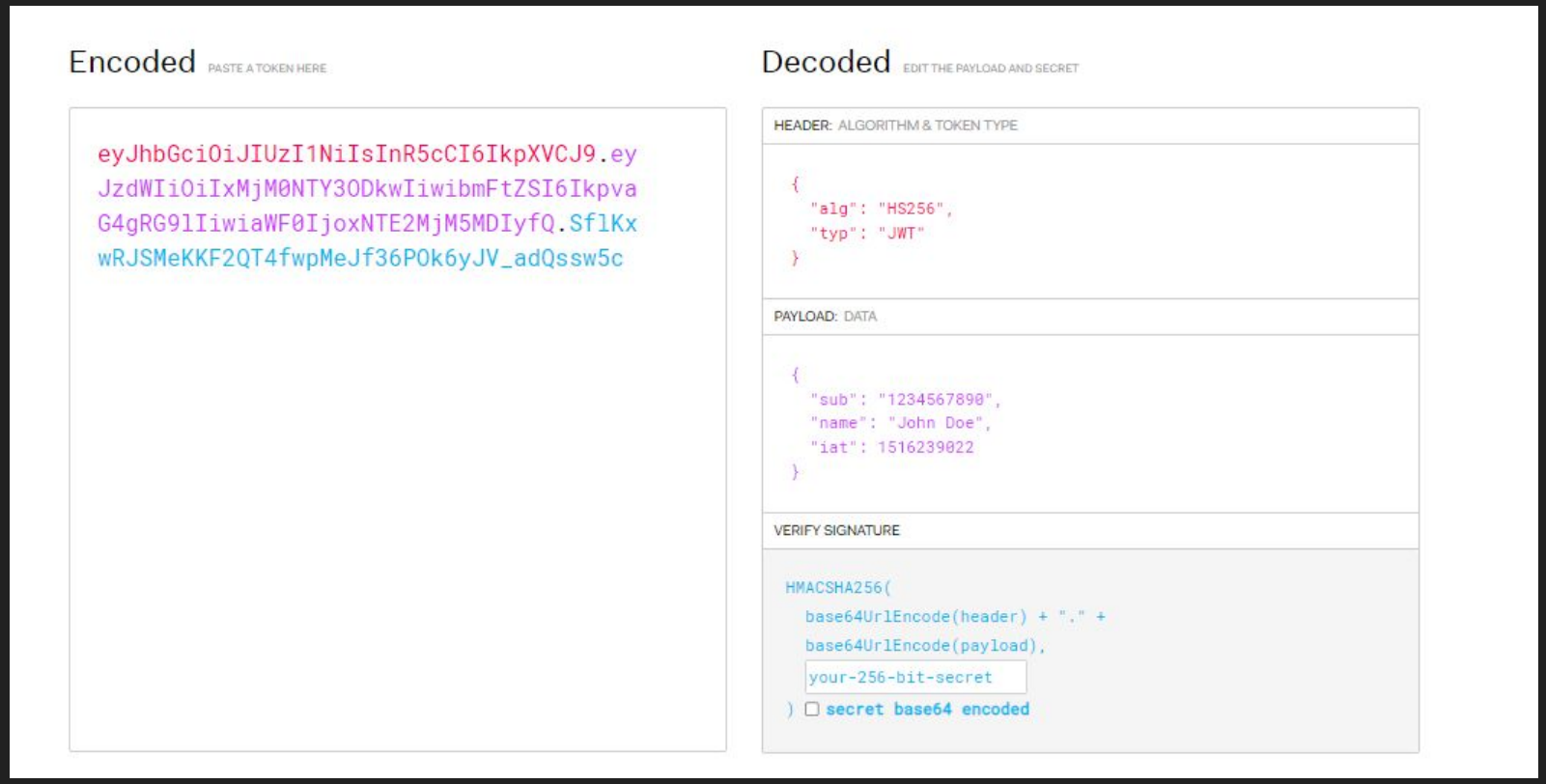
Altamente escalable y eficiente, ya que el servidor no necesita almacenar información del usuario.

Cada solicitud se procesa de manera independiente

Como toda la información necesaria se incluye en cada solicitud, los tokens o JWT deben ser protegidos adecuadamente para evitar accesos no autorizados

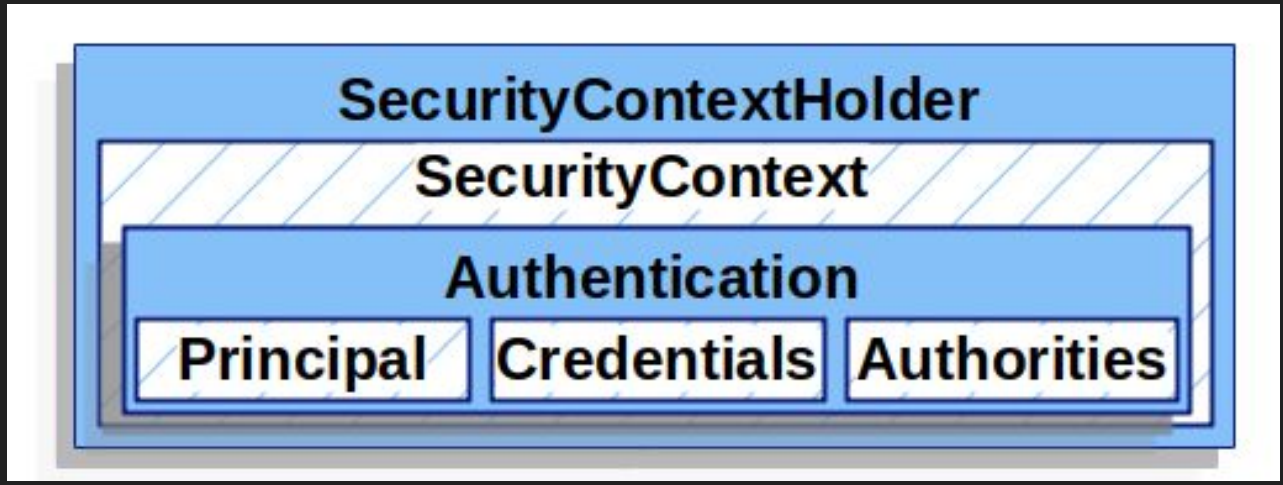
JWT (Jason Web Token)

- Es un estándar (RFC 7519) que define un método compacto y seguro de transmitir información en formato JSON y está diseñado para ser implementado en mecanismos de autenticación y autorización.
- Un JWT está formado por tres partes:
  - Encabezado
  - Carga útil
  - Firma



Objetos principales de Autenticación

- SecurityContextHolder: es un objeto que almacena el contexto de seguridad de una aplicación y lo asocia con la petición http que esta gestionando. Permite obtener información de autenticación y autorización de la solicitud http



- SecurityContext: representa el contexto de seguridad actual de una solicitud
- Credentials: representa la información de autenticación de un principal logueado
- Authorities: Es una lista de GrantedAuthorities, estos son permisos otorgados a un usuario.

AuthenticationManager

- Objeto principal que se utiliza a la hora de autenticar un usuario
- Es el encargado de llevar adelante el proceso de autenticación
- Recibe como parámetro el objeto que contine las credenciales del usuario

ProviderManager

- Implementación más común de AuthenticationManager
- Componente encargado de coordinar la autenticación
- Contiene una lista de AuthenticationProvider, cada uno representa una estrategia de autenticación
- Delega la autenticación al AuthenticationProvider correspondiente para la estrategia seleccionada, por ejemplo DaoAuthenticationProvider para autenticarnos con usuario y contraseña contra la base de datos.

Implementar autenticación contra base de datos

1. Creamos una clase para crear los beans de seguridad y le ponemos la anotación @Configuration
2. Inyectar AuthenticationConfiguration
3. Crear un bean que nos retorne el AuthenticationManager
4. Crear un bean que nos retorne el PasswordEncoder
5. Implementar el UserDetailsService
6. Crear un bean que nos retorne el AuthenticationProvider necesario, en este caso DaoAuthenticationProvider y pasarle el PasswordEncoder y el UserDetailsService

```java
@Configuration
public class SecurityBeansInjector {

    @Autowired
    private AuthenticationConfiguration authenticationConfiguration;

    @Autowired
    private UsuarioRepository usuarioRepository;

    @Bean
    public AuthenticationManager authenticationManager() throws Exception{

        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    public AuthenticationProvider authenticationProvider(){

        DaoAuthenticationProvider authenticationStrategy = new DaoAuthenticationProvider();
        authenticationStrategy.setPasswordEncoder(passwordEncoder());
        authenticationStrategy.setUserDetailsService(userDetailsService());

        return authenticationStrategy;
    }

    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
```

Dos formas de implemetnar UserDetailsServices

- A través de una función lambda

```java
@Bean
public UserDetailsService userDetailsService(){

    return (username) -> {
        return usuarioRepository.findByEmail(username)
            .orElseThrow(() -> new EntidadNoEncontradaException("No existe usuario "+ username));
    };
}
```

Dos formas de implemetnar UserDetailsServices

- Implementando la interfaz UserDetailsSerivce y sobreescribiendo el método loadUserByUsername

```java
@Service  no usages
public class UsuarioService implements UserDetailsService {

    @Autowired  1 usage
    UsuarioRepository usuarioRepository;

    @Override  no usages
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        return usuarioRepository.findByEmail(username)
                .orElseThrow(() -> new EntidadNoEncontradaException("No existe usuario "+ username));
    }
}
```

Implementamos UserDatails

```
public class User implements UserDetails
```

```
@Override  2 usages
public Collection<? extends GrantedAuthority> getAuthorities() {
    if(role == null) return null;

    List<SimpleGrantedAuthority> authorities = new ArrayList<>();

    authorities.add(new SimpleGrantedAuthority( role: "ROLE_" + this.role.name()));
    return authorities;
}

@Override
public String getPassword() { return password; }

@Override
public String getUsername() { return username; }

@Override  no usages
public boolean isAccountNonExpired() { return true; }

@Override  no usages
public boolean isAccountNonLocked() { return true; }

@Override  no usages
public boolean isCredentialsNonExpired() { return true; }

@Override
public boolean isEnabled() { return true; }
```

Ahora debemos configurar el Jwt y crear un servicio para crearlo

- Librerías Jwt (v 0.11.5) utilizadas
- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api
- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl
- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson

- Secret Key para cifrado y tiempo de expiración

```
security.jwt.expiration-in-minutes=30
security.jwt.secret-key=bWkgY2xhdmUgZXMgbXV5IHNlZ3VyYSAxMjM0NTY3OCBhYmNkZWZn
```

- Para la secret key se puede utilizar una clave de cifrado asimetrico (SSH Key)
- https://www.ssh.com/academy/ssh/keygen

Ahora debemos configurar el Jwt y crear un servicio para crearlo

```
@Value("${security.jwt.expiration-in-minutes}")  1 usage
private Long EXPIRATION_IN_MINUTES;

@Value("${security.jwt.secret-key}")  1 usage
private String SECRET_KEY;

public String generateToken(UserDetails user, Map<String, Object> extraClaims) {  2 usages

    Date issuedAt = new Date(System.currentTimeMillis());
    Date expiration = new Date( (EXPIRATION_IN_MINUTES * 60 * 1000) + issuedAt.getTime() );
    String jwt = Jwts.builder()
            .setClaims(extraClaims)
            .setSubject(user.getUsername())
            .setIssuedAt(issuedAt)
            .setExpiration(expiration)
            .setHeaderParam(Header.TYPE, Header.JWT_TYPE)
            .signWith(generateKey(), SignatureAlgorithm.HS256)
            .compact();

    return jwt;
}
```

```
private Key generateKey() {  2 usages
    byte[] passwordDecoded = Decoders.BASE64.decode(SECRET_KEY);
    System.out.println( new String(passwordDecoded) );
    return Keys.hmacShaKeyFor(passwordDecoded);
}

public String extractUsername(String jwt) { return extractAllClaims(jwt).getSubject(); }

private Claims extractAllClaims(String jwt) {  1 usage
    return Jwts.parserBuilder().setSigningKey( generateKey() ).build()
            .parseClaimsJws(jwt).getBody();
}
```

Ahora un método para crear el usuario

```java
@Autowired  1 usage
private PasswordEncoder passwordEncoder;

@Override  1 usage
public User registrOneCustomer(SaveUser newUser) {
    validatePassword(newUser);

    User user = new User();
    user.setPassword(passwordEncoder.encode(newUser.getPassword()));
    user.setUsername(newUser.getUsername());
    user.setName(newUser.getName());
    user.setRole(Role.CUSTOMER);

    return userRepository.save(user);
}
```

OncePerRequestFilter

- Es un filtro que extiende de GeneriFilterBean
- Sirve para asegurar una única ejecución del filtro por request (repetido)
- Si un usuario envía varios request por precionar varias veces un botón, este tipo de filtros se quedarán con el primer request y descartar el resto de los repetidos
- Util como un mecanismo para mitigar ataques DoS

Ahora creamos un filtro de Jwt para comprobar el token cada vez que lo envie

```java
@Component  2 usages
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Autowired  1 usage
    private JwtService jwtService;

    @Autowired  1 usage
    private UserService userService;

    @Override  no usages
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {

        System.out.println("ENTRO EN EL FILTRO JWT AUTHENTICATION FILTER");

        //1. Obtener encabezado http llamado Authorization
        String authorizationHeader = request.getHeader( s: "Authorization");//Bearer jwt
        if(!StringUtils.hasText(authorizationHeader) || !authorizationHeader.startsWith("Bearer ")){
            filterChain.doFilter(request, response);
            return;
        }

        //2. Obtener token JWT desde el encabezado
        String jwt = authorizationHeader.split( regex: " ")[1];

        //3. Obtener el subject/username desde el token
        // esta accion a su vez valida el formato del token, firma y fecha de expiración
        String username = jwtService.extractUsername(jwt);

        //4. Setear objeto authentication dentro de security context holder

        User user = userService.findOneByUsername(username)
                .orElseThrow(() -> new ObjectNotFoundException("User not found. Username: " + username));

        UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(
            username,  credentials: null, user.getAuthorities()
        );
        authToken.setDetails(new WebAuthenticationDetails(request));
        SecurityContextHolder.getContext().setAuthentication(authToken);
        System.out.println("Se acaba de setear el authentication");

        //5. Ejecutar el registro de filtros
        filterChain.doFilter(request, response);
    }
```

Ahora creamos un endpoint de login

```java
@PostMapping("/authenticate")  no usages
public ResponseEntity<AuthenticationResponse> authenticate(
        @RequestBody @Valid AuthenticationRequest authenticationRequest){

    AuthenticationResponse rsp = authenticationService.login(authenticationRequest);
    return ResponseEntity.ok(rsp);

}
```

```java
public AuthenticationResponse login(AuthenticationRequest autRequest) {  1 usage

    Authentication authentication = new UsernamePasswordAuthenticationToken(
            autRequest.getUsername(), autRequest.getPassword()
    );

    authenticationManager.authenticate(authentication);

    UserDetails user = userService.findOneByUsername(autRequest.getUsername()).get();
    String jwt = jwtService.generateToken(user, generateExtraClaims((User) user));

    AuthenticationResponse authRsp = new AuthenticationResponse();
    authRsp.setJwt(jwt);

    return authRsp;
}
```

Configuramos los endpoints que requieren autenticación

```java
@Configuration  no usages
@EnableWebSecurity
public class HttpSecurityConfig {

    @Autowired  1 usage
    private AuthenticationProvider daoAuthProvider;

    @Autowired  1 usage
    private JwtAuthenticationFilter jwtAuthenticationFilter;

    @Bean  no usages
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        SecurityFilterChain filterChain = http
                .csrf( csrfConfig -> csrfConfig.disable() )
                .sessionManagement( sessMagConfig -> sessMagConfig.sessionCreationPolicy(SessionCreationPolicy.STATELESS) )
                .authenticationProvider(daoAuthProvider)
                .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class)
                .authorizeHttpRequests( authReqConfig -> {
                    buildRequestMatchers(authReqConfig);
                } )
                .build();

        return filterChain;
    }
```

Configuramos los endpoints que requieren autenticación

```
private static void buildRequestMatchers(AuthorizeHttpRequestsConfigurer<HttpSecurity>.AuthorizationManagerRequestMatcherRegistry authReqConfig) {  1 usage
/*
Autorización de endpoints de products
 */
    authReqConfig.requestMatchers(HttpMethod.GET,  ...patterns: "/products")
            .hasAnyRole(Role.ADMINISTRATOR.name(), Role.ASSISTANT_ADMINISTRATOR.name());

      authReqConfig.requestMatchers(HttpMethod.GET, "/products/{productId}")
    authReqConfig.requestMatchers(RegexRequestMatcher.regexMatcher(HttpMethod.GET,  pattern: "/products/[0-9]*"))
            .hasAnyRole(Role.ADMINISTRATOR.name(), Role.ASSISTANT_ADMINISTRATOR.name());

    authReqConfig.requestMatchers(HttpMethod.POST,  ...patterns: "/products")
            .hasRole(Role.ADMINISTRATOR.name());

    authReqConfig.requestMatchers(HttpMethod.PUT,  ...patterns: "/products/{productId}")
            .hasAnyRole(Role.ADMINISTRATOR.name(), Role.ASSISTANT_ADMINISTRATOR.name());

    authReqConfig.requestMatchers(HttpMethod.PUT,  ...patterns: "/products/{productId}/disabled")
            .hasRole(Role.ADMINISTRATOR.name());
```

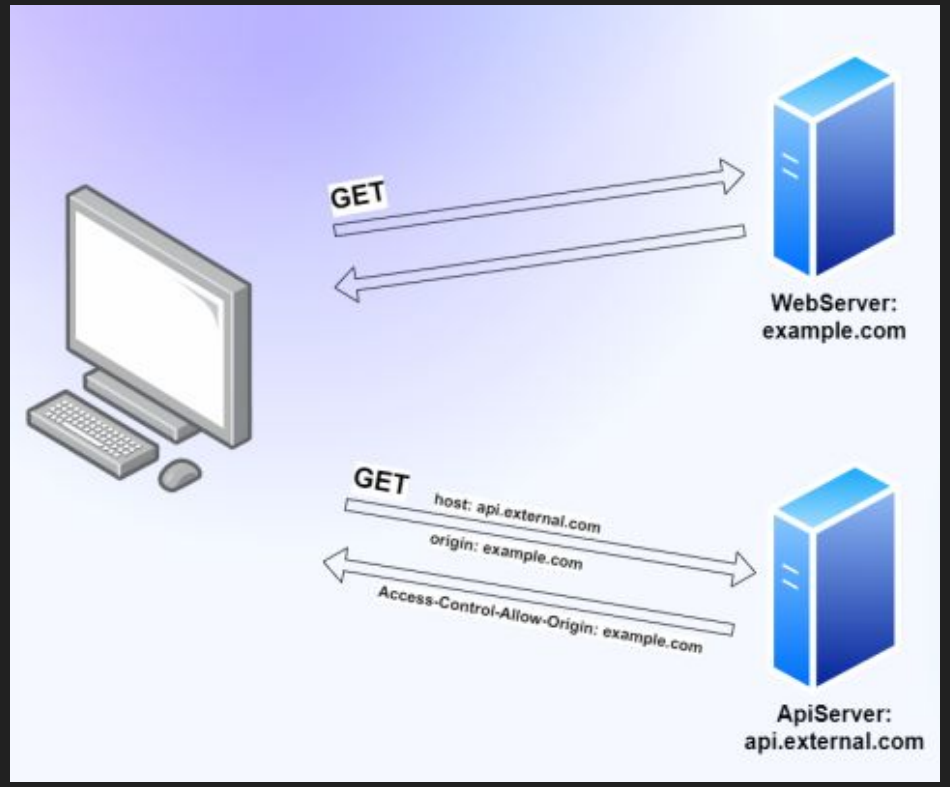
```
authReqConfig.requestMatchers(HttpMethod.POST,  ...patterns: "/customers").permitAll();
authReqConfig.requestMatchers(HttpMethod.POST,  ...patterns: "/auth/authenticate").permitAll();
authReqConfig.requestMatchers(HttpMethod.GET,  ...patterns: "/auth/validate-token").permitAll();

authReqConfig.anyRequest().authenticated();
```

Formas de implementar Autorización

- En el ejemplo anterior se implementó autorización a través de roles
- También podemos implementar autorización a través de authorities (permisos)
- Para Spring los roles y las atuhorities son lo mismo salvo que a los roles se les antepone el prefijo ROLE_
- Si queremos validar por permiso, en lugar de usar hasRole() o hasAnyRole() deberemos usar hasAuthorities() o hasAnyAuthorities()
- También se puede implementar autorización por aseguramiento de métodos, es decir, utilizando las anotaciones @PreAuthorize y @PostAuthorize

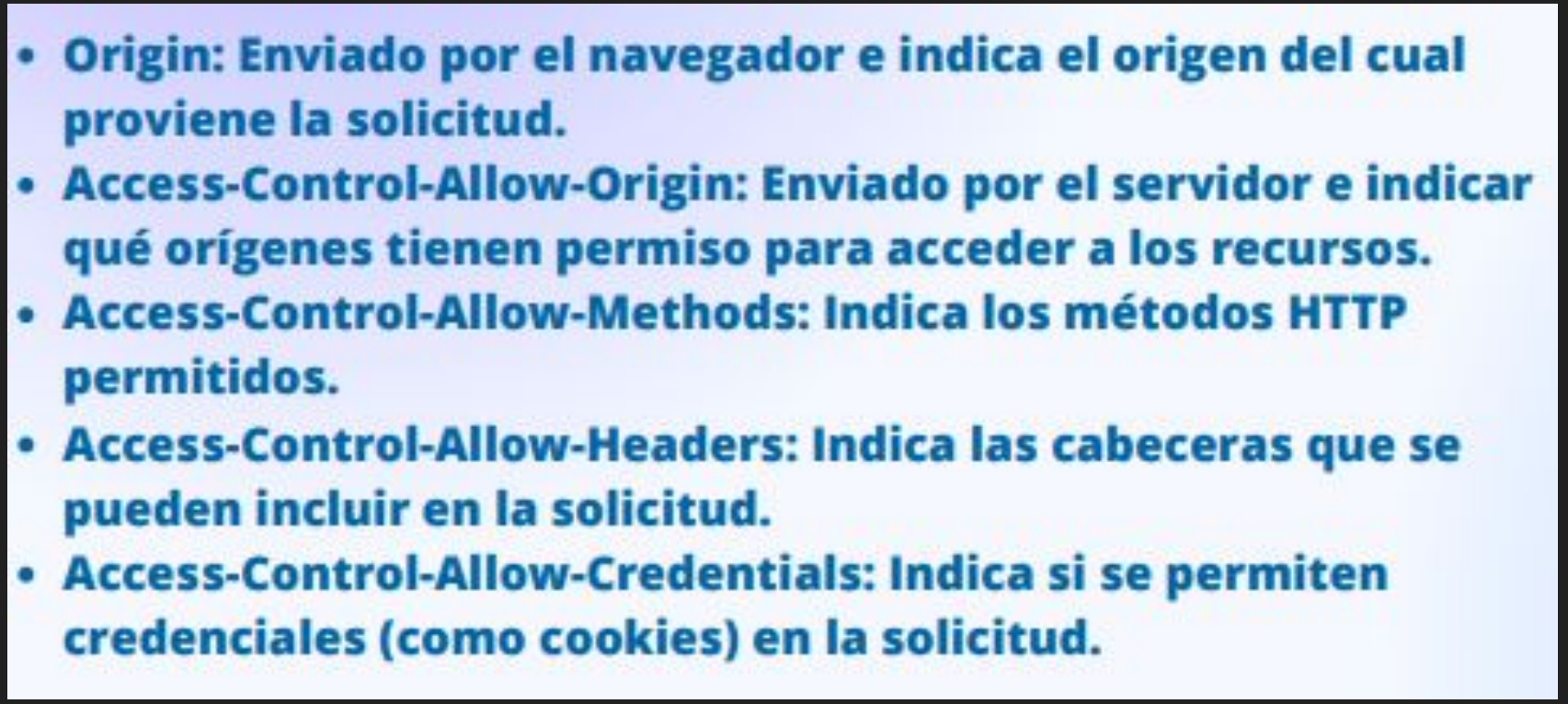
CORS (Cross Origin Resources Sharing)

- Es un mecanismo de seguridad implementado en los navegadores webs para permitir o restringir solicitudes de recursos entre diferentes dominios.
- CORS aborda la necesidad de permitir excepciones controladas a esta política, permitiendo que los servidores decidan a quién y cómo permitir solicitudes desde otros orígenes.



CORS (Cross Origin Resources Sharing)

- Por defecto los navegadores tiene configurada la política Same-Origin-Policy, es decir, solo pueden hacer peticiones del mismo origen.
- Los CORS funcionan a través del intercambio de encabezados Http entre el navegador y el servidor.
- Headers CORS:



Importancia de CORS

- Sin CORS los atacantes podrían acceder a recursos o realizar acciones en nombre de un usuario autenticado.

- Supongamos que me roban el token, entonces el atacante puede intentar hacer peticiones con ese token pero el origen ha cambiado, ahora el origen es la computadora del atacante, entonces CORS denegaría el acceso.

- Por esto debemos especificar correctamente los métodos y orígenes permitidos
- Evitar el uso de comodines como *, a menos que la api se a pública.
- Validar los orígenes permitidos ya que los encabezados CORS son fáciles de falsificar
- Enviar respuestas adecuadas a solicitudes OPTIONS preflight (los preflight se disparan con métodos HTTP sensibles como PUT o DELETE)
- Debemos configurar el CorsFilter en Spring para que se ejecute antes que nuestro JWTFilter

Anotación CrossOrigin

- @CrossOrigin nos permite configurar CORS a nivel de un método o de una clase
- Para que esto funcione debemos habilitar el CORS en la configuración de seguridad

```
@CrossOrigin(origins = "http://127.0.0.1")
@RestController
@RequestMapping("/products")
public class ProductController {
```

```
SecurityFilterChain filterChain = http
        .cors(Customizer.withDefaults())
        .csrf( csrfConfig -> csrfConfig.disable() )
        .sessionManagement( sessMagConfig -> sessMagCor
```

```
@CrossOrigin(origins = {"http://127.0.0.1", "www.google.com"}, methods = {RequestMethod.GET})
@GetMapping("/{productId}")
public ResponseEntity<Product> findOneById(@PathVariable Long productId){
```

Configuración de CORS global

- Vamos a crear un Bean que nos permita configurar el CORS para todos los controllers
- Para ello usaremos las clases de org.springframework.web.cors

- Como se puede ver en dicho bean seteamos todos los encabezados vistos anteriormente

```
@Bean  no usages
CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(Arrays.asList("https://www.google.com", "http://127.0.0.1:5500"));
    configuration.setAllowedMethods(Arrays.asList("*"));
    configuration.setAllowedHeaders(Arrays.asList("*"));
    configuration.setAllowCredentials(true);

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration( pattern: "/**", configuration);
    return source;
}
```