Guia Prático com Spring Security

O **Spring Security** é um framework do Spring utilizado para autenticação, autorização e outras medidas de segurança em aplicações Spring e Spring Boot. Ele é altamente configurável e pode ser usado em conjunto com várias estratégias de segurança.

1. Configuração Padrão

Quando adicionamos a dependência spring-boot-starter-security, as configurações padrões do Spring Security são ativadas automaticamente.

- **Interface gráfica para login**: O Spring Security cria uma página de login básica.
- **Usuário padrão**:
    - O Spring gera automaticamente um nome de usuário (user) e uma senha, exibida no console na inicialização da aplicação.
    - **Exemplo no console**:

Using generated security password: e3a1f8be-742c-40bd-b9b2-78a27c45830c

- **Dependência Maven**:

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

2. Definindo Usuário e Senha no application.properties

Para criar um usuário e senha personalizados:

```
spring.security.user.name=admin
spring.security.user.password=admin123
```

3. Usando a Interface UserDetailsService

O Spring Security utiliza a interface **UserDetailsService** para buscar as credenciais e permissões dos usuários. Para configurações simples, a implementação padrão é a **InMemoryUserDetailsManager**, usada para armazenar usuários e senhas na memória.

Exemplo de implementação personalizada:

```
@Bean
public UserDetailsService userDetailsService() {
   var user = User.withUsername("customuser")
              .password("{noop}password") // {noop} indica que a senha não está encriptada
              .roles("USER")
              .build();
```

```
    return new InMemoryUserDetailsManager(user);
}
```

4. Cookies e Sessões

Após o login bem-sucedido, o Spring Security armazena um cookie chamado **JSESSIONID** no navegador, que é usado para gerenciar a sessão do usuário.

SERVLETS E FILTROS NO SPRING

1. Requisições HTTP no Spring

- As aplicações Java utilizam **Servlet Containers** como **Apache Tomcat** para processar requisições HTTP.
- O Servlet Container converte requisições HTTP em objetos Java:
    - **HttpServletRequest**: Representa a requisição HTTP.
    - **HttpServletResponse**: Representa a resposta HTTP.

2. Filtros

Os filtros permitem interceptar requisições e respostas antes ou depois do processamento pelos Servlets. No Spring Security, o **SecurityFilterChain** é o principal mecanismo usado para aplicar regras de autenticação e autorização.

3. DispatcherServlet no Spring MVC

- **Responsabilidade**: Gerenciar o fluxo de requisições em uma aplicação Spring.
- **Etapas**:
    - Captura todas as requisições mapeadas.
    - Determina qual controlador (controller) e qual método devem lidar com a requisição.
    - Retorna uma resposta ao cliente.

4. Exemplo de Fluxo Completo no Spring Boot

1. **Adicionar Filtro**:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
        .authorizeHttpRequests(auth -> auth
            .anyRequest().authenticated() // Todas as requisições devem ser autenticadas
        )
        .formLogin(Customizer.withDefaults()) // Página de login padrão
```

```
        .build();
}
```

2. **Criar um Controlador**:

```
@RestController
public class HomeController {
   @GetMapping("/")
   public String home() {
      return "LOGGED IN";
   }
}
```

3. **Comportamento**:
    a. Requisição não autenticada para /:
        i. O filtro redireciona para a página de login.
    b. Requisição autenticada:
        i. O DispatcherServlet encaminha a requisição para o método home().

FLUXO INTERNO DO SPRING SECURITY

O **Spring Security** possui um fluxo bem definido para autenticação e autorização, baseado em um conjunto de interfaces e classes extensíveis. Vamos detalhar cada componente e como eles interagem.

1. Requisição de Autenticação

Quando um usuário tenta acessar uma área protegida da aplicação:

1. O **Authentication Filter** intercepta a requisição.
2. Extrai as credenciais (por exemplo, do formulário de login ou do cabeçalho HTTP).
3. Cria um objeto **Authentication** (como UsernamePasswordAuthenticationToken).

2. Authentication Filter

Responsável por capturar as credenciais do usuário na requisição e iniciar o processo de autenticação.

Exemplos de filtros fornecidos pelo Spring:

- **UsernamePasswordAuthenticationFilter**: Lida com login baseado em formulário.
- **BasicAuthenticationFilter**: Para autenticação HTTP básica.
- **JwtAuthenticationFilter** (personalizado): Usado para validar JWTs.

Exemplo de uso manual do UsernamePasswordAuthenticationToken:

```
var usernamePassword = new UsernamePasswordAuthenticationToken(email, password);
var auth = authenticationManager.authenticate(usernamePassword);
```

3. AuthenticationManager

O **AuthenticationManager** gerencia a autenticação delegando-a a um ou mais **AuthenticationProviders**.

Funcionamento:

- Recebe o objeto **Authentication** do filtro.
- Verifica os **AuthenticationProviders** registrados para encontrar um que suporte o tipo de autenticação.
- Se um **AuthenticationProvider** for encontrado e as credenciais forem válidas, retorna um objeto **Authentication** preenchido.
- Caso contrário, lança uma exceção (ex.: **BadCredentialsException**).

Exemplo: Configurando manualmente um AuthenticationManager com provedores:

```
@Bean
public AuthenticationManager authenticationManager() {
   List<AuthenticationProvider> providers = new ArrayList<>();
   providers.add(new DaoAuthenticationProvider());
   return new ProviderManager(providers);
}
```

4. AuthenticationProvider

Interface que contém a lógica de autenticação. Cada implementação:

1. Verifica se suporta o tipo de autenticação.
2. Valida as credenciais do usuário.
3. Recupera o **UserDetails** usando o **UserDetailsService**.
4. Verifica a senha com o **PasswordEncoder** configurado.

DaoAuthenticationProvider é a implementação padrão usada com o banco de dados.

5. UserDetailsService

O **UserDetailsService** carrega os dados do usuário, como:

- Nome de usuário.
- Senha.
- Autoridades (permissões).

Exemplo de implementação personalizada:

```
@Service
public class AuthorizationService implements UserDetailsService {
   @Autowired
   private UserRepository userRepository;

   @Override
   public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
      return userRepository.findByEmail(username)
            .orElseThrow(() -> new UsernameNotFoundException("Usuário não encontrado"));
   }
}
```

6. PasswordEncoder

O **PasswordEncoder** é usado para codificar e verificar senhas, garantindo segurança.

Exemplo de configuração do PasswordEncoder:

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

7. SecurityContext

Após a autenticação:

1. O objeto **Authentication** é armazenado no **SecurityContext**.
2. Ele fica acessível em toda a aplicação enquanto a sessão ou o token JWT for válido.

Para manipular o SecurityContext manualmente:

```
SecurityContextHolder.getContext().setAuthentication(authentication);
```

RESUMO DO FLUXO

1. **Requisição**: O usuário envia uma solicitação para acessar um recurso protegido.
2. **Filtro de Autenticação**:
    a. Captura credenciais (ex.: nome de usuário e senha).
    b. Cria um objeto Authentication e chama o **AuthenticationManager**.
3. **AuthenticationManager**:
    a. Verifica qual **AuthenticationProvider** pode processar a autenticação.
    b. Delega ao **AuthenticationProvider** encontrado.

4. **AuthenticationProvider**:
    a. Usa o **UserDetailsService** para recuperar o usuário.
    b. Valida as credenciais com o **PasswordEncoder**.
    c. Retorna um objeto Authentication preenchido em caso de sucesso.
5. **SecurityContext**:
    a. O objeto Authentication autenticado é armazenado no SecurityContext.
6. **Autorização**:
    a. Verifica as permissões do usuário antes de permitir o acesso ao recurso.

JWT E FILTROS CUSTOMIZADOS

No caso de autenticação via **JWT**, o fluxo difere um pouco:

1. O cliente envia o token JWT no cabeçalho **Authorization**.
2. Um filtro customizado (JwtAuthenticationFilter) valida o token:
    a. Decodifica o JWT e extrai os dados do usuário.
    b. Cria um objeto **Authentication** manualmente.
    c. Armazena-o no **SecurityContext**.

Exemplo de filtro JWT personalizado:

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {
   @Override
   protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain) throws ServletException, IOException {
      String token = resolveToken(request); // Recupera o token do cabeçalho
      if (token != null && validateToken(token)) {
         Authentication authentication = getAuthentication(token); // Cria o objeto
Authentication
         SecurityContextHolder.getContext().setAuthentication(authentication); // Armazena
no SecurityContext
      }
      filterChain.doFilter(request, response); // Continua o fluxo
   }
}
```

Autenticação com Sucesso e Segurança no Spring Security

A autenticação bem-sucedida no Spring Security segue um fluxo padrão que finaliza com a criação de um objeto **Authentication** validado, que é armazenado no **SecurityContext** para uso contínuo durante a sessão ou na validação de tokens em cada requisição.

1. Autenticação com Sucesso

Quando o Spring Security valida as credenciais:

1. **O AuthenticationProvider transforma o UserDetails em um objeto Authentication** (como um UsernamePasswordAuthenticationToken autenticado).
2. Esse objeto é retornado ao **AuthenticationManager**.
3. O **SecurityContext** armazena o objeto para acessos futuros.

Observações importantes:

- Durante o processo, a senha **não é enviada ao banco de dados**. Primeiro, o Spring busca o usuário pelo nome de usuário e carrega as informações usando o **UserDetailsService**.
- A comparação da senha inserida com a armazenada é feita localmente com o **PasswordEncoder**.

2. Construção do SecurityContext

O **SecurityContext** é a principal estrutura de armazenamento das informações de autenticação. Ele:

- É gerenciado pelo **SecurityContextHolder**, que usa uma variável **thread-local** para armazenar os dados.
- Pode ser recriado em cada requisição (em aplicações stateless), geralmente com base em tokens (ex.: JWT).

Variáveis thread-local são específicas para cada thread, garantindo que múltiplas requisições simultâneas não compartilhem o mesmo contexto de autenticação.

3. Hashing e Segurança de Senhas

Existem três métodos principais para armazenar senhas:

1. **Encoder**: Apenas codifica os dados (ex.: Base64). **Não recomendado** para senhas.
2. **Encryptor**: Utiliza uma chave secreta para encriptar os dados, que podem ser descriptografados.
3. **Hashing**: Usa um algoritmo irreversível para gerar um hash a partir da senha. Este método é amplamente utilizado.

4. Salting

O **salting** adiciona um valor único (salt) à senha antes de aplicar o hash, garantindo que senhas iguais tenham hashes diferentes.

Exemplo de hash com BCrypt:

$2a$10$EixZaYVK1fsbw1ZfbX3OXePaWxn96p36yAHaP8zzhPRWFtoCzP4W

- **Prefixo**: Algoritmo e custo.

- **Salt**: Valor único.
- **Hash**: Resultado final.

5. Implementações de PasswordEncoder

Encoder	Características
NoOpPasswordEncoder	Armazena senhas como texto simples. **Não recomendado.**
BCryptPasswordEncoder	Seguro e amplamente usado; requer alto poder de computação para ataques de força bruta.
SCryptPasswordEncoder	Resistente a ataques de força bruta, permite ajustar uso de memória e processamento.
Argon2PasswordEncoder	Similar ao SCrypt, adiciona controle sobre o número de threads.
Pbkdf2PasswordEncoder	Mais vulnerável a ataques de força bruta devido ao baixo custo de memória.

Exemplo:

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

Registro de usuário:

- Ao registrar um usuário, a senha é codificada manualmente.
- Durante o login, o Spring Security usa o mesmo PasswordEncoder configurado para verificar a senha.

6. Autenticação Stateless com JWT

Em aplicações **stateless**, o SecurityContext é recriado em cada requisição com base nas informações contidas no token JWT.

- **Por que não é vulnerável a CSRF?**
- Em aplicações stateless, não há sessão no servidor. O cliente precisa enviar o token em cada requisição, eliminando o risco associado ao armazenamento automático de cookies pelo navegador.

Exemplo de filtro JWT personalizado:

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {
   @Override
   protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain)
```

```
        throws ServletException, IOException {
    String token = resolveToken(request); // Extrai o token do cabeçalho Authorization
    if (token != null && validateToken(token)) {
        Authentication auth = getAuthentication(token); // Cria o objeto Authentication
        SecurityContextHolder.getContext().setAuthentication(auth); // Armazena no
SecurityContext
    }
    filterChain.doFilter(request, response); // Continua o fluxo
  }
}
```

7. Segurança Adicional: CORS e CSRF

CORS (Cross-Origin Resource Sharing):

- Impede que navegadores acessem recursos de domínios diferentes sem permissão explícita.
- Configurado no servidor para permitir (ou restringir) origens específicas.

Aplicação Angular comunicando-se com Spring Boot:

```
@Bean
public WebMvcConfigurer corsConfigurer() {
  return new WebMvcConfigurer() {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
      registry.addMapping("/**")
          .allowedOrigins("http://localhost:4200")
          .allowedMethods("GET", "POST", "PUT", "DELETE");
    }
  };
}
```

CSRF (Cross-Site Request Forgery):

- Protege aplicações stateful contra requisições não autorizadas originadas de sites maliciosos.
- Não é necessário em aplicações **stateless** com JWT, mas **CORS ainda é essencial.**

Desabilitar CSRF:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.csrf().disable();
}
```

8. Como Funciona o Cookie de Sessão

Os navegadores enviam cookies automaticamente para o servidor correspondente em todas as requisições, sem distinção de origem.

Em ataques CSRF:

- Um site malicioso força o navegador da vítima a enviar requisições utilizando cookies válidos.
- Aplicações **stateless** mitigam isso ao exigir tokens explícitos em cada requisição.

Diferença no Envio de Cookies e Tokens JWT

Cookie de Sessão (Stateful)

- **Configuração**: O navegador armazena e envia cookies automaticamente em todas as requisições para o domínio de origem, configurados pelo cabeçalho **Set-Cookie** enviado pelo servidor.
- **Funcionamento**:
    - A aplicação backend gerencia as sessões no servidor, associando cada usuário autenticado a um cookie de sessão.
    - O Spring Security utiliza a configuração:

```
http.sessionManagement()
   .sessionCreationPolicy(SessionCreationPolicy.ALWAYS); // ou IF_REQUIRED
```

Token JWT (Stateless)

- **Configuração**: O token JWT não é gerenciado automaticamente pelo navegador. Deve ser explicitamente incluído nos cabeçalhos da requisição:

```
Authorization: Bearer <token>
```

- **Funcionamento**:
    - O backend não mantém estado, e cada requisição inclui o token JWT nos cabeçalhos para autenticação.
    - O Spring Security usa:

```
http.sessionManagement()
   .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
```

CSRF e Cookies

CSRF (Cross-Site Request Forgery)

- Um ataque CSRF aproveita o fato de que navegadores enviam cookies automaticamente com as requisições, permitindo que um site malicioso realize ações em nome do usuário autenticado.
- Aplicações **stateful** (com cookies de sessão) precisam habilitar CSRF para evitar esse tipo de ataque.
- **Como funciona no Spring Security**:
    - Um token CSRF é gerado para cada sessão e enviado ao cliente no corpo da resposta ou em um cookie.
    - O cliente (frontend) precisa enviar esse token em cada requisição que altera dados (POST, PUT, DELETE), seja no corpo ou nos cabeçalhos.

Exemplo de Configuração com CSRF:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
   http.csrf()
      .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
   return http.build();
}
```

- **withHttpOnlyFalse()**: Permite que o frontend leia o token CSRF do cookie para incluí-lo nas requisições.
- O Spring automaticamente insere o token nos cabeçalhos das requisições, como:

X-CSRF-Token: <token>

Cookies e Domínios

- Cookies armazenados pelo navegador são associados a um domínio específico. Outros domínios não podem acessar esses cookies por questões de segurança.
- O envio automático de cookies nas requisições é limitado ao domínio do servidor que os configurou.

BasicAuthenticationFilter

- Usado em autenticação básica via cabeçalhos HTTP:

Authorization: Basic <base64-encoded-credentials>

- O Spring converte a requisição em um **UsernamePasswordAuthenticationToken** automaticamente.

- Este filtro é aplicado no backend quando habilitamos autenticação básica no Spring Security:

```
http.httpBasic();
```

Diferenças: Authentication x Authorization

Conceito	Definição
Authentication	Processo de verificar a identidade do usuário (ex.: login com credenciais válidas).
Authorization	Processo de verificar os privilégios e permissões do usuário para acessar recursos específicos.

Diferenças: Authority x Role

Termo	Definição
Authority	Representa privilégios individuais, como a capacidade de acessar ou executar uma ação específica.
Role	Agrupamento lógico de **authorities**. Representa níveis de acesso, como "ROLE_ADMIN".

- O Spring adiciona automaticamente o prefixo ROLE_ para roles. Não é necessário incluir manualmente.

Filtros no Spring Security

- Todos os endpoints protegidos passam por filtros de segurança configurados no Spring Security.
- Endpoints com **.permitAll()**:
  - São processados pelos filtros, mas não requerem autenticação.
  - Uma autenticação anônima pode ser criada automaticamente.

Customizando Filtros

- Para criar filtros personalizados, estenda a classe **OncePerRequestFilter** e sobrescreva o método doFilterInternal:

```
public class CustomFilter extends OncePerRequestFilter {
   @Override
   protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain)
         throws ServletException, IOException {
      // Lógica personalizada aqui
      filterChain.doFilter(request, response); // Continua para o próximo filtro
   }
}
```

- **Adicionando Filtros ao Spring Security:**

```
http.addFilterBefore(new CustomFilter(), BasicAuthenticationFilter.class);
```

Método shouldNotFilter

- Define condições em que o filtro não deve ser aplicado:

```
@Override
protected boolean shouldNotFilter(HttpServletRequest request) throws ServletException {
   return request.getServletPath().startsWith("/public");
}
```

TOKENS

Por que usar Tokens?

- Tokens permitem invalidar acessos específicos sem que o usuário precise alterar suas credenciais.
- Reduz a necessidade de passar credenciais (como login e senha) a cada requisição.
- A validação do token é feita no servidor por meio de operações matemáticas e criptográficas, sem a necessidade de armazená-lo.

JWT (JSON Web Token)

Estrutura do JWT

1. **Header**:
    a. Contém informações como o algoritmo de assinatura e o tipo de token (geralmente JWT).
    b. Exemplo (em JSON, codificado em Base64):

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. **Payload**:
    a. Armazena os dados do usuário e informações como expiração do token (exp) e claims personalizadas.
    b. Exemplo:

```
{
  "sub": "user@example.com",
  "role": "ADMIN",
  "exp": 1695580800
}
```

3. **Signature**:
    a. Garante a integridade do token. É gerada combinando o header, payload, e uma chave secreta com o algoritmo especificado.
    b. Exemplo:

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

Stateful x Stateless

Stateful

- O servidor mantém o estado da sessão, identificada por um **cookie de sessão (JSESSIONID)**.
- Requisições subsequentes utilizam esse identificador para recuperar o estado do usuário.
- Sessões podem ser encerradas por:
    - **Expiração**: Tempo configurado sem atividade.
    - **Logout explícito**: Usuário realiza logout.
    - **Inatividade**: Tempo sem interação.
    - **Reinício do servidor**: Limpa todas as sessões ativas.

Configuração no Spring Security:

```
http.sessionManagement()
   .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED); // ou
SessionCreationPolicy.ALWAYS
```

Stateless

- O servidor não mantém estado. Todas as informações de autenticação são armazenadas no token JWT.
- O cliente armazena o token (em localStorage, sessionStorage ou cookies) e o envia no cabeçalho **Authorization**:

Authorization: Bearer <token>

- O Spring Security valida o token e extrai as informações necessárias, sem armazená-lo.

Configuração no Spring Security:

```
http.sessionManagement()
   .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
```

Exemplo de validação de token em um filtro:

```
public class JWTTokenValidatorFilter extends OncePerRequestFilter {
   @Override
```

```
   protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain)
          throws ServletException, IOException {
       String token = recoverToken(request); // Recupera o token do cabeçalho Authorization
       if (token != null && tokenService.validateToken(token)) {
          String email = tokenService.extractEmail(token);
          UserDetails userDetails = userRepository.findByEmail(email);

          UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(
              userDetails, null, userDetails.getAuthorities());
          SecurityContextHolder.getContext().setAuthentication(authentication);
       }
       filterChain.doFilter(request, response); // Continua com a cadeia de filtros
   }
}
```

SecurityContextHolder

- Mantém o contexto de segurança (autenticação e autorização) durante uma requisição.
- No modo **stateless**, o contexto é descartado automaticamente após a requisição:

```
SecurityContextHolder.clearContext();
```

Method Level Security

@PreAuthorize

- Define regras de autorização antes de executar o método:

```
@PreAuthorize("hasRole('ADMIN')")
public void deleteUser(Long id) { ... }
```

@PostAuthorize

- Define regras de autorização após a execução do método:

```
@PostAuthorize("returnObject.owner == authentication.name")
public User getUser(Long id) { ... }
```

Exemplo Completo: Gerar e Validar JWT

Gerar o Token

```
@Value("${api.gym.token.secret}")
private String secret;

public String generateToken(UserModel user) throws JWTCreationException {
```

```
    Algorithm algorithm = Algorithm.HMAC256(secret);
    return JWT.create()
            .withIssuer("gym-api")
            .withSubject(user.getEmail())
            .withExpiresAt(generateExpirationDate())
            .sign(algorithm);
}

private Instant generateExpirationDate() {
    return LocalDateTime.now().plusHours(2).toInstant(ZoneOffset.UTC);
}
```

Validar o Token

```
public String validateToken(String token) throws JWTVerificationException {
    Algorithm algorithm = Algorithm.HMAC256(secret);
    return JWT.require(algorithm)
            .withIssuer("gym-api")
            .build()
            .verify(token)
            .getSubject();
}
```