05. Controle de acesso

Date	@06/02/2023
Categoria	Java API
Curso	Spring Boot 3: aplique boas práticas e proteja uma API Rest

Tópicos

- Projeto da aula anterior
- Interceptando requisições
- Para saber mais: Filters
- Criando o filter de segurança
- Recuperando o token
- Validando o token recebido
- Autenticando o usuário
- authorizeRequests deprecated
- Filtrando requisições
- Testando o controle de acesso
- Para saber mais: controle de acesso por url
- Para saber mais: controle de acesso por anotações
- Faça como eu fiz: autorizando requisições
- Projeto final do curso
- O que aprendemos?
- Conclusão

Para saber mais: Filters

Filter é um dos recursos que fazem parte da especificação de Servlets, a qual padroniza o tratamento de requisições e respostas em aplicações Web no Java. Ou seja, tal recurso não é específico do Spring, podendo assim ser utilizado em qualquer aplicação Java.

É um recurso muito útil para isolar códigos de infraestrutura da aplicação, como, por exemplo, segurança, logs e auditoria, para que tais códigos não sejam duplicados e misturados aos códigos relacionados às regras de negócio da aplicação.

Para criar um *Filter*, basta criar uma classe e implementar nela a interface `Filter` (pacote jakarta.servlet). Por exemplo:

```
@WebFilter(urlPatterns = "/api/**")
public class LogFilter implements Filter {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException,
        System.out.println("Requisição recebida em: " + LocalDateTime.now());
        filterChain.doFilter(servletRequest, servletResponse);
    }

}
```

O método `doFilter` é chamado pelo servidor automaticamente, sempre que esse *filter* tiver que ser executado, e a chamada ao método `filterChain.doFilter` indica que os próximos *filters*, caso existam outros, podem ser executados. A anotação `@WebFilter`, adicionada na classe, indica ao servidor em quais requisições esse *filter* deve ser chamado, baseando-se na URL da requisição.

No curso, utilizaremos outra maneira de implementar um *filter*, usando recursos do Spring que facilitam sua implementação.

Criando o filter de segurança

- O `@Component` é utilizado para que o Spring carregue uma classe/componente genérico
- **FilterChain**: Representa a cadeia de filtros na aplicação

---

Recuperando o token

- Por padrão, o tipo de prefixo Bearer é utilizado para tokens JWT

Filtrando requisições

Em relação às classes `Filter`, conforme abordado ao longo dessa aula, escolha as opções que indicam os objetivos do ***Filter Chain***:

Selecione 2 alternativas

- Pode ser utilizado para bloquear uma requisição.
  - É possível interromper o fluxo de uma requisição com o objeto *Filter Chain*.
- Representa o conjunto de filtros responsáveis por interceptar requisições.
  - Esse é um dos objetivos do *Filter Chain*.
- Pode ser utilizado para validar *tokens JWT*.
  - O *Filter Chain*não tem relação direta com *tokens JWT*.
- Pode ser utilizado para autenticar o usuário.
  - O *Filter Chain* não tem relação direta com o processo de autenticação.

Testando o controle de acesso

- É importante determinar a ordem dos filtros aplicados

Para saber mais: controle de acesso por url

Na aplicação utilizada no curso não teremos perfis de acessos distintos para os usuários. Entretanto, esse recurso é utilizado em algumas aplicações e podemos indicar ao *Spring Security* que determinadas URLs somente podem ser acessadas por usuários que possuem um perfil específico.

Por exemplo, suponha que em nossa aplicação tenhamos um perfil de acesso chamado de **ADMIN**, sendo que somente usuários com esse perfil possam excluir médicos e pacientes. Podemos indicar ao *Spring Security* tal configuração alterando o método `securityFilterChain`, na classe `SecurityConfigurations`, da seguinte maneira:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http.csrf().disable()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and().authorizeRequests()
        .antMatchers(HttpMethod.POST, "/login").permitAll()
        .antMatchers(HttpMethod.DELETE, "/medicos").hasRole("ADMIN")
        .antMatchers(HttpMethod.DELETE, "/pacientes").hasRole("ADMIN")
        .anyRequest().authenticated()
        .and().addFilterBefore(securityFilter, UsernamePasswordAuthenticationFilter.class)
        .build();
}
```

Repare que no código anterior foram adicionadas duas linhas, indicando ao *Spring Security* que as requisições do tipo `DELETE` para as URLs `/medicos` e `/pacientes` somente podem ser executadas por usuários autenticados e cujo perfil de acesso seja **ADMIN**.

Para saber mais: controle de acesso por anotações

Outra maneira de restringir o acesso a determinadas funcionalidades, com base no perfil dos usuários, é com a utilização de um recurso do Spring Security conhecido como **Method Security**, que funciona com a utilização de anotações em métodos:

```
@GetMapping("/{id}")
@Secured("ROLE_ADMIN")
public ResponseEntity detalhar(@PathVariable Long id) {
    var medico = repository.getReferenceById(id);
    return ResponseEntity.ok(new DadosDetalhamentoMedico(medico));
}
```

No exemplo de código anterior o método foi anotado com `@Secured("ROLE_ADMIN")`, para que apenas usuários com o perfil **ADMIN** possam disparar requisições para detalhar um médico. A anotação `@Secured` pode ser adicionada em métodos individuais ou mesmo na classe, que seria o equivalente a adicioná-la em **todos** os métodos.

Atenção! Por padrão esse recurso vem desabilitado no spring Security, sendo que para o utilizar devemos adicionar a seguinte anotação na classe `Securityconfigurations` do projeto:

```
@EnableMethodSecurity(securedEnabled = true)
```

Você pode conhecer mais detalhes sobre o recurso de method security na documentação do Spring Security, disponível em: https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html

---

O que aprendemos?

Nessa aula, você aprendeu como:

- Funcionam os `Filters` em uma requisição;
- Implementar um *filter* criando uma classe que herda da classe `OncePerRequestFilter`, do Spring;
- Utilizar a biblioteca ***Auth0 java-jwt*** para realizar a validação dos *tokens* recebidos na API;
- Realizar o processo de autenticação da requisição, utilizando a classe `SecurityContextHolder`, do Spring;
- Liberar e restringir requisições, de acordo com a URL e o verbo do protocolo HTTP.
