Sistemas Web Seguros

UNIDADE 06

Controle de acesso em serviços REST

Nesta unidade, conheceremos o conceito de filtros (*servlet filter*), especificação da linguagem Java para sistemas *web*. Por meio de filtros, construiremos a camada de segurança no SISRH, que garantirá que todas as requisições vindas ao sistema precisarão estar devidamente autorizadas pelo modelo RBAC, que também será construído. Além disso, criaremos o sistema de *login* único (*single sign-on*), por meio de JWT.

CONTROLE DE ACESSO EM SERVIÇOS REST

Vimos que o serviço de gestão de identidade e acesso tem por finalidade gerenciar com segurança o acesso aos serviços e recursos de uma solução de TI. Essa é uma solução chamada transversal, pois envolve todas as operações do sistema. Por esse motivo, não recomendamos implementar essa gestão manualmente em cada um dos serviços REST, pois corremos o risco de falha humana. Já pensou se o desenvolvedor se esquecer de chamar a gestão de identidade e acesso em um serviço importante? Por isso, vamos conhecer um conceito importante em aplicações Java *web* para garantir que todos os serviços sejam verificados automaticamente por meio de filtros.

Filtros (*servlet filter*)

Em qualquer sistema, há requisitos que não estão diretamente ligados a regras de negócio, como, por exemplo: *logs* de auditoria, quando desejamos registrar as chamadas da nossa aplicação; autorização de acesso; tratamento de erro; criptografia ou compactação de dados, entre outros. Todas essas funcionalidades não fazem parte diretamente das funções de negócio do sistema, mas são essenciais para o bom funcionamento delas.

Os filtros em Java são usados para interceptar a solicitação do cliente e fazer algum pré-processamento. Eles também podem interceptar a resposta e fazer o pós-processamento antes de enviar ao cliente no aplicativo da *web*.

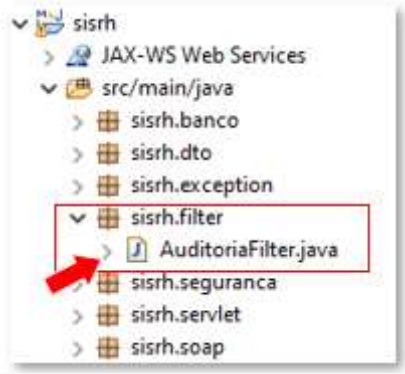
Algumas tarefas comuns que podemos realizar com filtros são:

- gestão automatizada de *log* para auditoria;
- autenticação e autorização de recursos;
- formatação de dados;
- compressão de dados para envio ao cliente;
- inclusão de *cookies* e informações de cabeçalho nas respostas.

Exemplo: filtro de auditoria

Para exemplificar, vamos apresentar como seria ter um filtro no SISRH para gerar *logs* de auditoria de todas as requisições HTTP feitas com o respectivo IP do cliente que as originou.

Podemos criar uma classe Java chamada **AuditoriaFilter** e, para manter a organização do projeto, incluir no pacote **sisrh.filter**.



Veja a estrutura inicial de uma classe do tipo **Filter**:

```
package sisrh.filter;

import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;


@WebFilter("*")
public class AuditoriaFilter implements Filter {

	@Override
	public void init(final FilterConfig filterConfig) throws
ServletException {
	}

	@Override
	public void doFilter(ServletRequest request,
ServletResponse response, FilterChain chain)
				throws IOException, ServletException {
	}

	@Override
	public void destroy() {
	}
}
```

A anotação **@WebFilter** informa o padrão de requisição que fará o filtro ser acionado. Em nosso exemplo, vamos auditar todas as requisições; assim, foi utilizado @WebFilter("*"). Se quiséssemos filtrar apenas as requisições dos serviços REST do

SISRH, usaríamos: @WebFilter("/rest/*").

As classes **Filter** possuem três métodos:

- **init**: quando o sistema é inicializado, este método é chamado. Ele é chamado apenas uma vez e serve para inicializar qualquer recurso que seja necessário. O parâmetro **FilterConfig** é usado para fornecer o objeto de contexto do *servlet* para o filtro.
- **doFilter**: este método é invocado todas as vezes em que é preciso aplicar o filtro a um recurso. Ele recebe o objeto da requisição (**ServletRequest**) e o objetivo de resposta (**ServletResponse**). O parâmetro **FilterChain** é usado para chamar o próximo filtro da cadeia.
- **destroy**: é acionado antes de o sistema ser desligado. Nesse momento, podemos fechar quaisquer recursos abertos pelo filtro. Ele é chamado apenas uma vez durante a vida útil do filtro.

Veja como seria a implementação da nossa auditoria utilizando o método **doFilter**:

```
@Override
public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain)
                throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest)
request;
        String ipAddress = request.getLocalAddr();

        System.out.println(ipAddress + ":" +
httpRequest.getRequestURI().toString());

        chain.doFilter(httpRequest, response);
}
```

Observe que, a partir do objeto **request**, obtivemos os dois dados desejados: a URL chamada (`httpRequest.getRequestURI().toString()`) e o IP do cliente (`request.getLocalAddr()`).

Veja o exemplo dos *logs* que seriam gerados:

```
192.168.0.3:/sisrh/soap/basico
192.168.0.5:/sisrh/soap/basico
```

```
192.168.0.3:/sisrh/soap/basico
127.0.0.1:/sisrh/soap/basico
127.0.0.1:/sisrh/soap/basico
192.168.0.3:/sisrh/rest/swagger.json
192.168.0.8:/sisrh/rest/
192.168.0.8:/sisrh/rest/empregado
```

Agora que conhecemos a estrutura essencial dos filtros, vamos para a atividade da semana, em que construiremos juntos o filtro para garantir que todas as requisições REST passaram pelo filtro de segurança que validará os *tokens* de acesso com o padrão de acesso RBAC.

CONCLUSÃO

- Filtros em Java são importantes para construir funcionalidades transversais que podem atuar em todas as funcionalidades de um sistema *web*.
- Filtros são representados por classes Java com a anotação **@WebFilter**.
- O filtro possui três principais métodos: **init**, **doFilter** e **destroy**.
- Vimos na atividade como desenvolver com filtros uma camada de segurança para aplicações *web*.