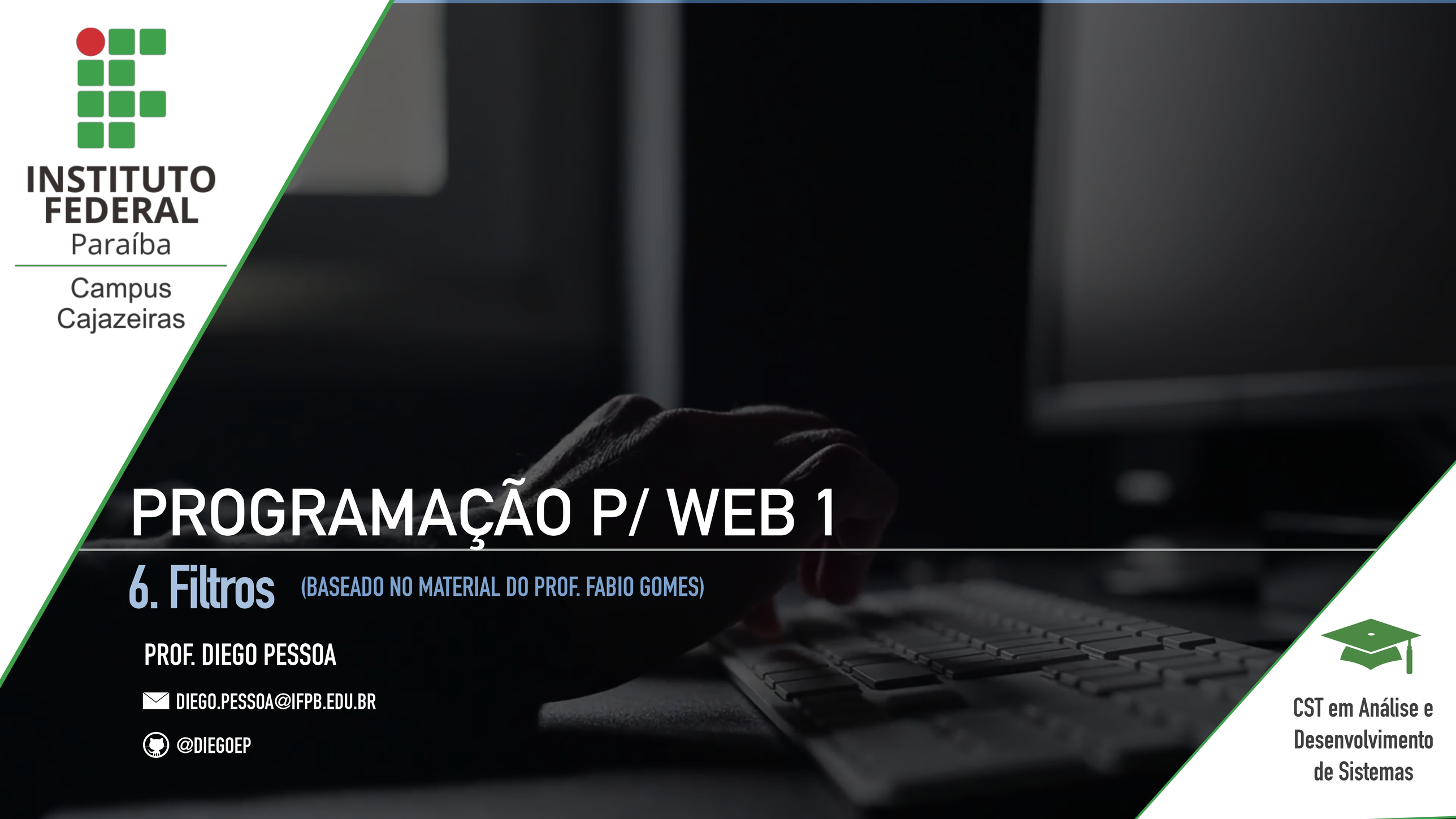


INSTITUTO FEDERAL
Paraíba

Campus Cajazeiras

PROGRAMAÇÃO P/ WEB 1

6. Filtros

(BASEADO NO MATERIAL DO PROF. FABIO GOMES)

PROF. DIEGO PESSOA

DIEGO.PESSOA@IFPB.EDU.BR

@DIEGOEP

CST em Análise e Desenvolvimento de Sistemas

Introdução

- Em muitas situações, precisamos executar ações que são inerentes à aplicação como um todo ou parte dela;

- Exemplos: realizar logs, auditorias, verificar restrições de segurança, etc;

Introdução

- Os filtros correspondem à solução para realizar estas tarefas de forma simples e flexível;

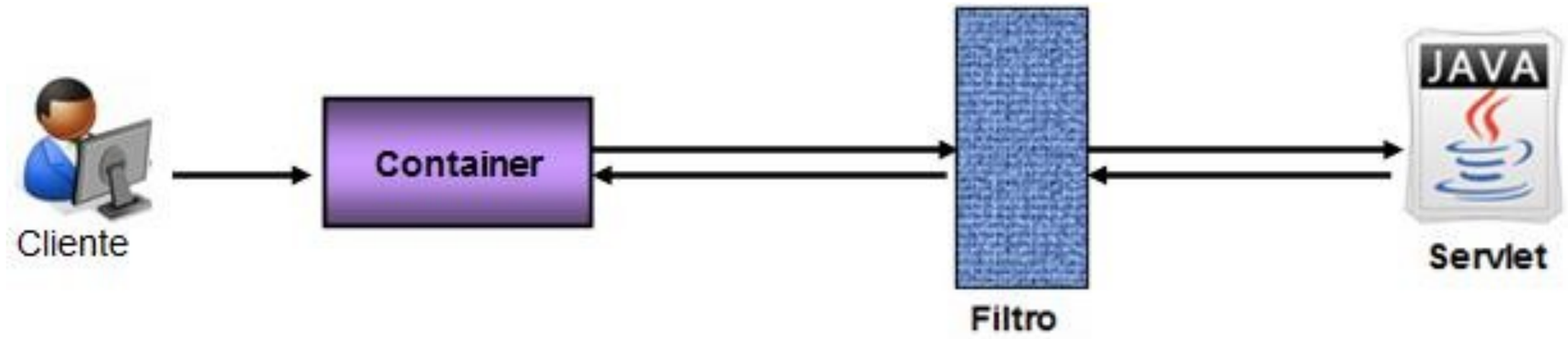
- Eles permitem realizar estas tarefas de forma transparente para os recursos da aplicação;

Filtros

- O que são filtros?
  - São componentes que nos permitem interceptar e processar requisições antes que as mesmas sejam enviadas ao recurso solicitado pelo cliente;
  - Também podem ser usados para processar a resposta produzida pelo recurso antes que a mesma seja enviada de volta ao cliente;

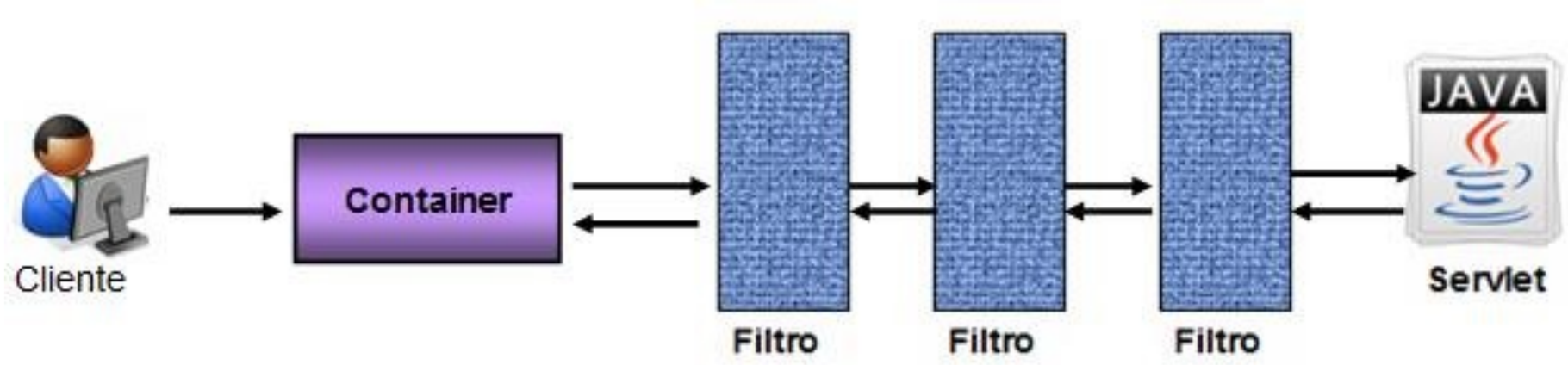
Filtros

- Os filtros atuam como um moderador entre o container e o servlet:



Filtros

- Os filtros também podem ser usados em cadeia:



Filtros

- Mais detalhes:
  - O uso do filtro é transparente para o recurso solicitado;
  - A configuração sobre a utilização dos filtros pode ser definida no descritor da aplicação;

Filtros

- Mais detalhes:
  - Os filtros são independentes, ou seja, eles desconhecem a existência e a ação de outros filtros;
  - Em tempo de execução, o container é o responsável por invocar o (s) filtro (s) necessário (s) para cada tipo de requisição;

Filtros

- Exemplos de ações que podemos fazer com filtros de requisição:
  - Realizar verificações de segurança;
  - Reformatar os cabeçalhos ou o corpo da requisição;
  - Auditar solicitações para registrá-las em log;
  - Comprimir o stream da resposta;
  - Anexar algo ou alterar o stream da resposta;
  - Criar uma resposta totalmente diferente;

A Interface Filter

- Interface principal que descreve o comportamento de um filtro;
  - Todos os filtros da aplicação web devem implementar esta interface;

- Esta interface descreve os métodos do ciclo de vida de um filtro;

A Interface Filter

- A interface Filter define três métodos:
  - init(filterConfig);
  - doFilter(request, response, filterChain);
  - destroy();

A Interface Filter

- O método init:
  - Usamos este método para definir as ações que devem ser executadas durante a inicialização do filtro;
  - Este método recebe como parâmetro de entrada uma instância de uma classe que implementa a interface FilterConfig;
    - ✓ Falaremos mais desta interface daqui a pouco;

A Interface Filter

- O método doFilter:
  - É o principal método da interface Filter;
  - Usamos este método para definir as ações correspondentes ao trabalho do filtro;
    - ✓ Ele tem a mesma função que os métodos doGet, do Post e doTag;

A Interface Filter

- O método doFilter:
  - O método doFilter recebe como parâmetros de entrada três tipos de objeto:
    - ✓ A requisição que o cliente enviou para a aplicação;
    - ✓ A resposta que a aplicação enviará de volta para o cliente;
    - ✓ A cadeia de filtros na qual o filtro está inserido;

A Interface Filter

- O método destroy:
  - Usamos este método para programar as ações que devem ser executadas antes que o filtro seja finalizado;
  - É normalmente usado para liberar os objetos e demais recursos usados pelo filtro;

A Interface FilterConfig

- Especifica a interface de um objeto que encapsula as informações de configuração do filtro;
- Tais informações podem ser definidas no descritor da aplicação ou mediante uma anotação;

A Interface FilterConfig

- O container é responsável por buscar as informações de configuração e instanciar o objeto;

A Interface FilterConfig

- Os métodos da interface FilterConfig:
  - getFilterName():
    - ✓ Retorna o nome do filtro;
  - getInitParameterNames():
    - ✓ Retorna uma enumeração contendo os nomes de todos os parâmetros de inicialização definidos para o filtro;

A Interface FilterConfig

- Os métodos da interface FilterConfig:
  - getInitParameter(String parameterName):
    - ✓ Retorna o valor de um determinado parâmetro de inciialização;
  - getServletContext():
    - ✓ Retorna uma referência para o contexto da aplicação na qual o filtro está sendo executado;

A Interface FilterChain

- Define a interface de um objeto que representa a cadeia de filtros pela qual a requisição passará até chegar ao recurso;

- Os seus objetos permitem que o filtro possa encaminhar a requisição (ou a resposta) para o próximo filtro da cadeia;

A Interface FilterChain

- Caso o filtro seja o último filtro da cadeia, ele encaminha a requisição para o recurso solicitado;

- Ou, no caso da volta, para o cliente que fez a requisição;

A Interface FilterChain

- A interface FilterChain define apenas o método doFilter;

- Este método é usado para encaminhar os objetos da solicitação para o próximo filtro da cadeia;

A Interface FilterChain

- O método doFilter recebe como parâmetros de entrada uma referência para requisição e outra para a resposta;

O Ciclo de Vida de um Filtro

- A execução de um filtro acontece da seguinte forma:
  1. O container instancia o filtro;
  2. O container chama o método init;
  3. O container invoca o método doFilter;
  4. O container invoca o método destroy:

A Estrutura do Método doFilter

- Como já vimos, cada requisição passa pelo filtro duas vezes;
  - Quando a requisição é enviada para a aplicação e quando a resposta é enviada de volta para o cliente;

- Entretanto, todas as ações programadas para o filtro são definidas no método doFilter;

A Estrutura do Método doFilter

- Desta forma, como podemos separar o que deve ser feito na ida e o que deve ser feito na volta?

A Estrutura do Método doFilter

- Resposta: Isto é feito por meio da invocação do método doFilter do objeto que implementa o FilterChain;

A Estrutura do Método doFilter

- O método doFilter diz ao container que a requisição do cliente já pode ser encaminhada;

- Feito isto, o método para e aguarda até que a resposta retornada pelo cliente passe novamente pelo filtro;

A Estrutura do Método doFilter

- Quando o método é invocado, o container é responsável por verificar o próximo filtro a ser aplicado;
  - O filtro não sabe nada sobre a existência de outros filtros;

- Caso não haja mais filtros, o container encaminha a requisição para o recurso solicitado;

A Estrutura do Método doFilter

- Assim, podemos dizer que a execução do método doFilter de um filtro acontece da seguinte forma:
  - O filtro processa a requisição;
  - O filtro invoca o método doFilter do objeto FilterChain;
  - O filtro processa a resposta;

A Estrutura do Método doFilter

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

    // comandos

    chain.doFilter(request, response);

    // comandos
}
```

Estes comandos são executados durante a ida

Estes comandos são executados durante a volta

Implementando um Filtro

- Vamos agora ver a implementação de um filtro que faz o registro de requisições feitas para uma aplicação;

- O filtro deste exemplo faz o log da requisição e do tempo que o servidor demorou para gerar a resposta;

Exemplo de implementação de um filtro

```
public class LogFilter implements Filter{

    private FilterConfig config;
    private PrintWriter out;

    @Override
    public void init(FilterConfig config) throws ServletException {
        this.config = config;
        try {
            out = new PrintWriter(new File("c:/dados/logs/FilterLog.txt"));
        }
        catch (FileNotFoundException ex) {}
    }

    @Override
    public void doFilter(ServletRequest sr, ServletResponse sr1, FilterChain fc)
            throws IOException, ServletException {...9 linhas }

    @Override
    public void destroy() {
        config = null;
        out.close();
    }

}
```

Implementando um Filtro

- Implementação do método doFilter:

```
@Override
public void doFilter(ServletRequest sr, ServletResponse sr1, FilterChain fc)
        throws IOException, ServletException {
    long requestTime = System.currentTimeMillis();
    out.println("A requisição passou pelo filtro de log");
    fc.doFilter(sr, sr1);
    long responseTime = System.currentTimeMillis();
    out.println("A resposta passou pelo filtro de log");
    long total = responseTime - requestTime;
    out.println("A requisição foi processada em "+total+" milissegundos");
}
```

Declarando o Filtro

- Uma vez implementado, o filtro precisa ser declarado na aplicação;
- Essa declaração pode ser feita de duas formas:
  - No descritor da aplicação ou mediante uma anotação;

Declarando o Filtro no Descritor da Aplicação

- No descritor da aplicação, filtros são descritos por meio de dois elementos: filter e filter-mapping;

- Os dois elementos são obrigatórios para cada filtro usado pela aplicação;

Declarando o Filtro no Descritor da Aplicação

- O elemento filter:
  - É o principal elemento para a descrição do filtro;
  - É neste elemento onde declaramos a classe correspondente à implementação do filtro;
  - É composto por dois subelementos obrigatórios: filter-name e filter-class;

Declarando o Filtro no Descritor da Aplicação

- O elemento filter-mapping:
  - Usamos este elemento para indicar as situações nas quais o filtro deve ser aplicado;
  - Este elemento pode ser composto por três tipos de subelementos: filter-name, url-pattern, servlet-name;

Declarando o Filtro no Descritor da Aplicação

- O elemento filter-mapping:
  - O subelemento filter-name corresponde à identificação do filtro no descritor da aplicação;
  - O seu valor deve casar com o valor do elemento filter-name do elemento filter usado para descrever o filtro;

Declarando o Filtro no Descritor da Aplicação

- O elemento filter-mapping:
  - Os subelementos url-pattern e servlet-name são usados para definir as situações nas quais o filtro deve ser aplicado;
  - O filtro é aplicado sempre que houver uma requisição para um recurso cujo nome case com os valores definidos para estes subelementos;

Declarando o Filtro no Descritor da Aplicação

- O sub-elemento url-pattern:
  - Neste subelemento, a aplicação do filtro é definida através da análise da URL do recurso solicitado pela requisição;
  - O filtro é aplicado nos casos em que a URL requisitada casa com o valor definido para este atributo;

Declarando o Filtro no Descritor da Aplicação

- O sub-elemento url-pattern:
  - Exemplo 1: <url-pattern>*.jsp </url-pattern>;
    - ✓ O filtro deve ser aplicado sempre que houver uma requisição a qualquer página JSP da aplicação;
  - Exemplo 2: <url-pattern>app.jsp </url-pattern>;
    - ✓ O filtro só deve ser aplicado quando a página app.jsp for requisitada;

Declarando o Filtro no Descritor da Aplicação

- O sub-elemento url-pattern:
  - Exemplo 3: <url-pattern> /admin/* </url-pattern>;
    - ✓ O filtro deve ser aplicado sempre que for solicitado um recurso da subpasta admin;
  - Exemplo 4: <url-pattern>/* </url-pattern>;
    - ✓ O filtro deve ser aplicado a todas as requisições realizadas na aplicação;

Declarando o Filtro no Descritor da Aplicação

- O sub-elemento url-pattern:
  - Exemplo 5: <url-pattern>/secret/* </url-pattern>;
    - ✓ O filtro deve ser aplicado quando houver uma requisição para qualquer recurso do diretório secret;

Declarando o Filtro no Descritor da Aplicação

- O sub-elemento servlet-name:
  - Usamos este elemento para indicar que o filtro deve ser aplicado quando um determinado servlet for executado;
  - O seu valor corresponde ao identificador do servlet no descritor, que é o seu atributo servlet- name;

Declarando o Filtro no Descritor da Aplicação

- Exemplo de declaração de filtro no descritor da aplicação:

```
<filter>
    <filter-name>log</filter-name>
    <filter-class>br.ifpb.psd.LogFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>log</filter-name>
    <url-pattern>/logs.htm</url-pattern>
</filter-mapping>
```

Declarando o Filtro no Descritor da Aplicação

- Por default, filtros são aplicados apenas a requisições recebidas diretamente do cliente;
- Podemos usar o elemento dispatcher para aplicá-los em outras situações;
- Este atributo é usado dentro do elemento filter-mapping;

Declarando o Filtro no Descritor da Aplicação

- O elemento dispatcher tem quatro valores possíveis:
  - REQUEST;
  - INCLUDE;
  - FORWARD;
  - ERROR;

Declarando o Filtro no Descritor da Aplicação

- Podemos escolher mais de um valor para o elemento dispatcher;
- A opção REQUEST é usada como o valor default para este elemento;

Declarando o Filtro no Descritor da Aplicação

- Exemplo de uso do atributo dispatcher:

```
<filter>
    <filter-name>myFilter</filter-name>
    <filter-class>br.ifpb.psd.MyFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>myFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher> REQUEST </dispatcher>
    <dispatcher> INCLUDE </dispatcher>
    <dispatcher> FORWARD </dispatcher>
</filter>
```

Declarando o Filtro no Descritor da Aplicação

- Cada recurso da aplicação pode casar com o padrão de URL (ou nome do servlet) de mais de um filtro;

- Nestes casos, todos os filtros que casam com o recurso da aplicação são aplicados;

Declarando o Filtro no Web.Xml

- A ordem de aplicação dos filtros é definida com base na ordem de declaração dos elementos filter-mapping;

- Os elementos declarados primeiro são acionados antes pelo container;

Declarando o Filtro no Descritor da Aplicação

- Vamos ver um exemplo em que três filtros são aplicados a uma requisição;
- Cada filtro altera um atributo de log indicando que a requisição passou pelo mesmo antes de chegar ao recurso;

Exemplo de uso de uma cadeia de filtros

```
public class FilterChain1 implements Filter{

    private FilterConfig config;

    @Override
    public void init(FilterConfig fc) {...3 linhas }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain fc)
            throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpSession session = httpRequest.getSession();
        String log = "";
        if(session.getAttribute("log")!=null){
            log = session.getAttribute("log").toString();
        }
        log+="Passei pelo Filtro 1";
        session.setAttribute("log", log);
        fc.doFilter(request, response);
    }


    @Override
    public void destroy() {...3 linhas }
}
```

Exemplo de uso de uma cadeia de filtros

```
public class FilterChain2 implements Filter{

    private FilterConfig config;

    @Override
    public void init(FilterConfig fc) {...3 linhas }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain fc)
            throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpSession session = httpRequest.getSession();
        String log = "";
        if(session.getAttribute("log")!=null){
            log = session.getAttribute("log").toString();
        }
        log+="Passei pelo Filtro 2";
        session.setAttribute("log", log);
        fc.doFilter(request, response);
    }

    @Override
    public void destroy() {...3 linhas }

}
```

Exemplo de uso de uma cadeia de filtros

```
public class FilterChain3 implements Filter{

    private FilterConfig config;

    @Override
    public void init(FilterConfig fc) {...3 linhas }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain fc)
            throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpSession session = httpRequest.getSession();
        String log = "";
        if(session.getAttribute("log")!=null){
            log = session.getAttribute("log").toString();
        }
        log+="Passei pelo Filtro 3";
        session.setAttribute("log", log);
        fc.doFilter(request, response);
    }


    @Override
    public void destroy() {...3 linhas }

}
```

Declarando o Filtro no Descritor da Aplicação

- Declaração dos elementos filter-mapping no descritor da aplicação:

```
<filter-mapping>
    <filter-name>fc1</filter-name>
    <url-pattern>/filterChain.jsp</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>fc2</filter-name>
    <url-pattern>/filterChain.jsp</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>fc3</filter-name>
    <url-pattern>/filterChain.jsp</url-pattern>
</filter-mapping>
```

Declarando o Filtro no Descritor da Aplicação

- Página JSP que mostra o atributo de log:

```
<html>
<head>
<title>Exemplo de aplicação de filtros</title>
</head>
<body>
    <p> Mostrando a ordem de log dos filtros! </p>
    <p>${log}</p>
</body>
</html>
```

Declarando o Filtro Através de Anotações

- Filtros também podem ser declarados através de anotações;
- Para isso, usamos a anotação @WebFilter;
- Os atributos filterName e urlPatterns podem ser usados para passar as informações de configuração;

Declarando o Filtro Através de Anotações

- O atributo servletNames pode ser usado para definir os nomes dos servlets aos quais o filtro deve ser aplicado;

- O atributo dispatcherTypes pode ser usado para definir os tipos de despacho aos quais o flitro deve ser aplicado;

Declarando um filtro mediante uma anotação

```java
@WebFilter(filterName="annotationLogFilter", urlPatterns= {"/log.htm"})
public class AnnotationLogFilter implements Filter{

    private FilterConfig config;
    private PrintWriter out;

    @Override
    public void init(FilterConfig config) throws ServletException {...7 linhas }

    @Override
    public void doFilter(ServletRequest sr, ServletResponse sr1, FilterChain fc)
            throws IOException, ServletException {
        long requestTime = System.currentTimeMillis();
        out.println("A requisição passou pelo filtro de log");
        fc.doFilter(sr, sr1);
        long responseTime = System.currentTimeMillis();
        out.println("A resposta passou pelo filtro de log");
        long total = responseTime - requestTime;
        out.println("A requisição foi processada em "+total+" milissegundos");
    }

    @Override
    public void destroy() {...4 linhas }

}
```

```java
@WebFilter(
        servletNames = {"UserLogServlet", "AdminLogServlet"},
        dispatcherTypes = {DispatcherType.REQUEST, DispatcherType.FORWARD})
public class AnnotationLogFilter implements Filter{

    private FilterConfig config;
    private PrintWriter out;

    @Override
    public void init(FilterConfig config) throws ServletException {...7 linhas }

    @Override
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain fc)
            throws IOException, ServletException {
        long requestTime = System.currentTimeMillis();
        out.println("A requisição passou pelo filtro de log");
        fc.doFilter(req, resp);
        long responseTime = System.currentTimeMillis();
        out.println("A resposta passou pelo filtro de log");
        long total = responseTime - requestTime;
        out.println("A requisição foi processada em "+total+" milissegundos");
    }

    @Override
    public void destroy() {...4 linhas }

}
```

Desviando o Acesso ao Recurso

- Podemos desviar o acesso ao recurso não invocando o método doFilter da cadeia de filtro;

- Nestes casos, podemos despachar a requisição para outro ponto da aplicação;

Desviando o Acesso ao Recurso

- Vamos ver um filtro que verifica se o usuário está com uma sessão aberta e se ele tem permissão para acessar o recurso;

- Caso estas condições não sejam satisfeitas, o cliente é encaminhado para outro ponto da aplicação;

Trecho do filtro que faz o desvio do acesso ao recurso

```
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain fc)
        throws IOException, ServletException {
    HttpServletRequest httpRequest = (HttpServletRequest) request;
    HttpSession session = httpRequest.getSession();
    if(session.getAttribute("user")!=null){
        User user = (User) session.getAttribute("user");
        if(user.getType()==User.ADMIN){
            fc.doFilter(request, response);
        }
        else dispatch(request,response,"/permissionError.html");
    }
    else dispatch(request,response,"/login.html");
}

private void dispatch(ServletRequest sr, ServletResponse sr1, String resource)
        throws ServletException, IOException{
    ServletContext context = config.getServletContext();
    RequestDispatcher dispatcher = context.getRequestDispatcher(resource);
    dispatcher.forward(sr, sr1);
}
```

Definindo Parâmetros de Inicialização

- Como nos servlets, podemos usar o elemento init-param para definir parâmetros de inicialização para um filtro;

- Cada parâmetro é descrito por dois atributos: param-name e param-value;
  - O valor do parâmetro é sempre tratado como um string;

Definindo Parâmetros de Inicialização

- O container é responsável por obter informações sobre os parâmetros de inicialização do filtro;

- Estas informações são encapsuladas no objeto que implementa a interface FilterConfig;

Definindo Parâmetros de Inicialização

- No FilterConfig, as informações sobre os parâmetros de incialização podem ser acessadas através de dois métodos;
  - getInitParameterNames();
    - ✓ Recupera os nomes dos parâmetros;
  - getInitParameter(String paramteterName):
    - ✓ Recupera o valor de um determinado parâmetro;

Recuperando os Parâmetros de Inicialização

- Podemos obter os parâmetros de inicialização de um filtro através do objeto FilterConfig;

- O método getInitParameterNames pode ser usado para recuperar os nomes dos parâmetros;

Recuperando os Parâmetros de Inicialização

- O método `getInitParameter` pode ser usado para recuperar o valor de um parâmetro específico;

- O nome do parâmetro deve ser passado como argumento;

Trabalhando com Parâmetros de Inicialização

- Vamos implementar um filtro que só permite o acesso ao módulo de administração da aplicação a partir de endereços previamente cadastrados;

- Os endereços permitidos são configurados como parâmetros de inicialização do filtro;

Trabalhando com Parâmetros de Inicialização

- Declarando o filtro com parâmetros de inicialização:

```
<filter>
    <filter-name>address</filter-name>
    <filter-class>br.ifpb.psd.filters.AddressFilter</filter-class>
    <init-param>
        <param-name>allowedAddresses</param-name>
        <param-value>
            127.0.0.1
            125.15.40.128
            125.15.40.129
        </param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>address</filter-name>
    <url-pattern>/admin/*</url-pattern>
</filter-mapping>
```

Trabalhando com Parâmetros de Inicialização

- O filtro obtém o endereço da requisição e verifica se o mesmo está cadastrado entre os endereços permitidos;
- Se sim, o filtro encaminha a requisição ao recurso solicitado;
- Se não, o filtro despacha a requisição para uma página de erro de permissão;

Filtro que manipula parâmetros de inicialização

```java
public class AddressFilter implements Filter{

    private FilterConfig config;
    private String[] allowedAddresses;

    @Override
    public void init(FilterConfig fc) {
        config = fc;
        String addressList = config.getInitParameter("allowedAddresses");
        allowedAddresses = addressList.split("\n");
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain fc)
            throws IOException, ServletException {...19 linhas }

    @Override
    public void destroy() {
        config = null;
        allowedAddresses = null;
    }

}
```

```java
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain fc)
        throws IOException, ServletException {
    String remoteAddress = request.getRemoteAddr();
    boolean allowed = false;
    for(String address : allowedAddresses){
        if(remoteAddress.equals(address)){
            allowed = true;
            break;
        }
    }
    if(allowed){
        fc.doFilter(request, response);
    }
    else{
        ServletContext context = config.getServletContext();
        context.log("Tentativa de acesso não autorizado a partir de "+remoteAddress);
        RequestDispatcher dispatcher = context.getRequestDispatcher("/permissionError.jsp");
        dispatcher.forward(request, response);
    }
}
```

Implementação do método doFilter do filtro que manipula parâmetros de inicialização

Trabalhando com Parâmetros de Inicialização

- Parâmetros de inicialização do filtro também podem ser definidos através de anotações;

- Nesse caso, os parâmetros são definidos através do atributo initParams;

Trabalhando com Parâmetros de Inicialização

- Cada parâmetro de inicialização é definido através da anotação @WebInitParam;

- As informações de configuração são passadas através dos atributos name e value;

Usando anotações para declarar o filtro que manipula parâmetros de inicialização

```
@WebFilter(
    filterName = "address",
    urlPatterns = {"/admin/*"},
    initParams = {
        @WebInitParam(
            name="allowedAddresses",
            value="127.0.0.1\n25.15.40.128\n125.15.40.129"
        )
    }
)
public class AddressFilter implements Filter{

    private FilterConfig config;
    private String[] allowedAddresses;

    @Override
    public void init(FilterConfig fc) {...5 linhas }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain fc)
            throws IOException, ServletException {...19 linhas }

    @Override
    public void destroy() {...4 linhas }

}
```

Usando Requisições e Respostas Personalizadas

- Com filtros, podemos usar interceptar a requisição e a resposta originais e substitui-las por outros personalizados;

- Fazemos isto invocando o método doFilter do objeto FilterChain com uma requisição ou resposta diferente;

Usando Requisições e Respostas Personalizadas

- Entretanto, criar uma nova implementação destes objetos não é uma tarefa simples;
  - Devido à grande quantidade de métodos que precisam ser implementados;

Usando Requisições e Respostas Personalizadas

- Para não ter que lidar com esta dificuldade, usamos duas classes disponibilizadas pela API:
  - HttpServletRequestWrapper;
  - HttpServletResponseWrapper;

Usando Requisições e Respostas Personalizadas

- Estes dois objetos implementam o padrão de projeto Decorator (ou Wrapper);

- Para criar uma requisição ou resposta personalizada, estendemos a classe apropriada;

Usando Requisições e Respostas Personalizadas

- Na nova classe, adicionamos as novas características que devem ser incorporadas ao objeto;

- E sobrescrevemos a implementação apenas dos métodos que precisam ser alterados;

Usando Requisições e Respostas Personalizadas

- Chamadas a métodos inalterados são mapeadas para o objeto original, que é encapsulado dentro do novo objeto;

- Vamos ver a implementação de uma resposta personalizada, que mantém uma lista de cookies;

Implementando um wrapper para o objeto de resposta

```
public class ResponseCookiesWrapper extends HttpServletResponseWrapper{

    private List<Cookie> cookies;

    public ResponseCookiesWrapper(HttpServletResponse response) {
        super(response);
        cookies = new ArrayList();
    }

    public List<Cookie> getCookies(){
        return cookies;
    }

    public void addCookie(Cookie cookie){
        cookies.add(cookie);
        HttpServletResponse httpResponse = (HttpServletResponse) getResponse();
        httpResponse.addCookie(cookie);
    }


}
```

Usando Requisições e Respostas Personalizadas

- Agora, vamos implementar um filtro que usa a resposta personalizada;

- O filtro faz o log de todos os cookies contidos na resposta antes de encaminhá-la para o cliente;

Filtro que usa uma resposta personalizada

```
public class LogResponseCookiesFilter implements Filter{

    private FilterConfig config = null;

    @Override
    public void init(FilterConfig fc) {...3 linhas }

    @Override
    public void doFilter(ServletRequest sr, ServletResponse sr1, FilterChain fc)
            throws IOException, ServletException {
        HttpServletResponse httpResponse = (HttpServletResponse) sr1;
        ResponseCookiesWrapper wrappedResponse = new ResponseCookiesWrapper(httpResponse);
        fc.doFilter(sr, wrappedResponse);
        ServletContext context = config.getServletContext();
        context.log("Cookies gerados na resposta");
        List<Cookie> cookies = wrappedResponse.getCookies();
        for(Cookie cookie : cookies){
            context.log(cookie.getName() +" = "+cookie.getValue());
        }
    }

    @Override
    public void destroy() {...3 linhas }

}
```