02

Primeiro filtro

Transcrição

Bem vindo de volta! Agora vamos falar sobre um recurso muito importante do mundo Servlet.

Anteriormente, queríamos melhorar o código do nosso controlador, já que misturamos um pouco as responsabilidades. Mas o que isso quer dizer?

Quando criamos nosso controlador, fizemos uma implementação bem genérica para que ele decidisse qual ação executar. Posteriormente, como todas as requisições vão passar por esse controlador, adicionamos uma regra de segurança - verificamos, em todas as requisições, se existe o atributo `usuarioLogado` na `Httpsession` ou não. Foi nesse momento que começamos a misturar as coisas.

Vamos estabelecer mais uma motivação. Imagine que nossa aplicação já está rodando, mas nosso chefe ou gerente reclama que ela está lenta - um problema clássico. Algo na rede, ou mesmo no computador dele, pode não estar funcionando. Talvez mesmo o banco de dados ou a nossa aplicação não estejam funcionando muito bem.

Ou seja, precisamos filtrar e descobrir onde está o problema, já que uma aplicação WEB é complicada, pois tem várias camadas físicas, redes e servidores envolvidos.

Com essa intenção, vamos abrir `ListaEmpresas` e medir o tempo de execução.

Lembre-se de apagar, nessa classe, o código que verificava se o usuário está logado, já que o implementamos de maneira mais elegante no `UnicaEntradaServlet`.

Para medirmos o tempo de execução, existe uma classe fachada chamada `System` que possui vários métodos estáticos, dentre eles o `currentTimeMillis()`, que devolve quantos milisegundos se passaram desde 1970, considerado o ano 0 da computação. Quando esse valor é positivo, significa que estamos em um momento do tempo após 1970; se esse valor fosse negativo, estaríamos antes de 1970.

Criaremos uma variável para representar os milisegundos antes da execução de `ListaEmpresas` (" `antes` "), e repetiremos essa lógica após essa ação, criando a variável " `depois` ". Para mostrarmos quanto tempo se deu entre a criação dessas duas variáveis, vamos printar a subtração delas:

```
public String executa(HttpServletRequest request, HttpServletR

    long antes = System.currentTimeMillis();

    System.out.println("listando empresas");

    Banco banco = new Banco();
    List<Empresa> lista = banco.getEmpresas();

    request.setAttribute("empresas", lista);

    long depois = System.currentTimeMillis();

    System.out.println("Tempo de execução " + (depois - antes)

    return "forward:listaEmpresas.jsp";
}
```

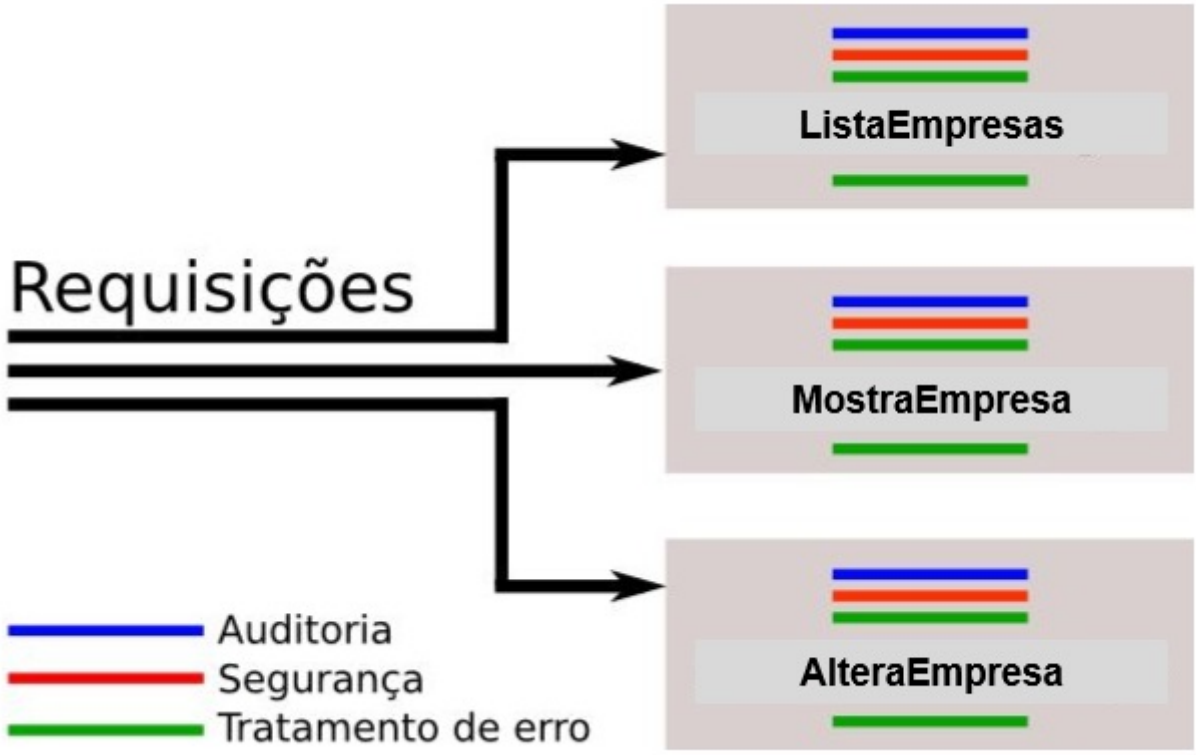
COPIAR CÓDIGO

Agora basta logarmos na nossa aplicação, o que automaticamente executará a ação `ListaEmpresas`. No console, poderemos verificar o tempo de execução (que deve ser aproximadamente `1ms`).

Se quiséssemos medir todas as aplicações, teríamos que copiar essas três linhas de código em cada ação. Portanto, podemos perceber que esse monitoramento é um código que não combina muito bem com as nossas ações.

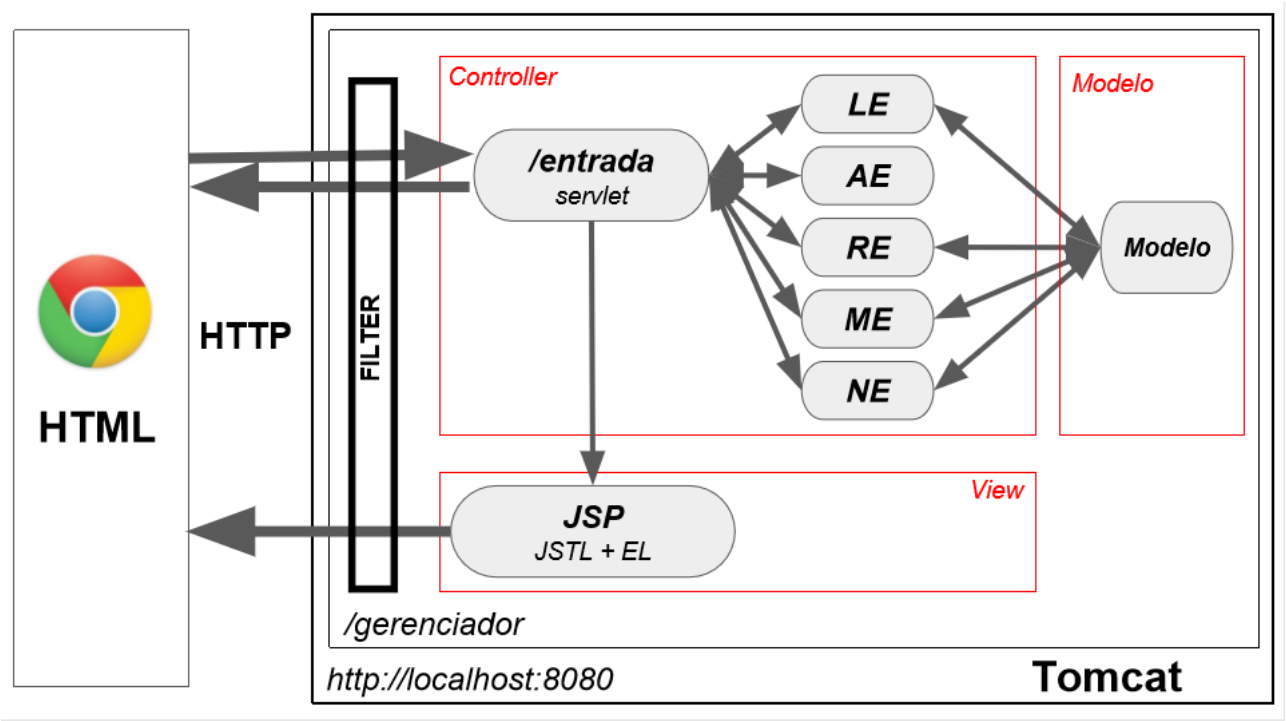
Existem outros exemplos de códigos que se espalham pelas ações - por exemplo, uma lógica de auditoria, que verifica qual usuário está executando qual ação, o que é muito importante para alguns sistemas.

Para exemplificar, se tivéssemos um código relacionado à auditoria, ele ocuparia somente uma linha no início de cada uma das ações. Outras preocupações básicas, como a segurança (fazer um teste de permissões mais detalhada) ou o tratamento de erros - o clássico "*Try/catch*" ou as transações `begin` (que inicia os comandos) e `commit` (que consolida os dados em um banco de dados) - também se espalham pelas ações.



Entretanto, nosso objetivo é centralizar o nosso código, pois dessa forma precisaríamos fazer alterações em um único lugar. Até agora definimos esse lugar como o nosso controlador `UnicaEntradaServlet`, mas já estabelecemos que o controlador não deveria testar as permissões ou fazer auditoria, tratamento de erros, transações, etc.

Existe um recurso fantástico do mundo Servlet que serve justamente para esses tipos de preocupações: o **filtro** (*"filter"*).



O filtro é como uma porta que é colocada entre o navegador e o Servlet, e ele permite filtrar as requisições. Da mesma forma que conseguimos mapear uma requisição para um Servlet, conseguimos mapear uma requisição para um filtro - no entanto, o filtro tem uma responsabilidade a mais: ele consegue parar uma requisição.

Começaremos então a fazer uma primeira implementação de filtro. De início, faremos a medição do tempo de resposta da nossa aplicação através desse filtro.

No pacote `servlet`, criaremos uma classe clicando com o botão direito e em seguida em "*New > Class*". Se repararmos bem, existe um item específico para criação de filtros ("Filter"), mas primeiro criaremos a classe manualmente para no futuro usarmos esse diálogo.

Chamaremos nossa classe de `MonitoramentoFilter`. No nosso `OiMundoServlet`, era necessário extender uma classe específica do mundo Servlet. No filtro é parecido, porém não existe uma classe para extender, mas sim uma interface para implementar. Essa interface se chama `implements Filter`.

Devemos tomar cuidado ao importar o tipo `Filter`, pois o conceito de filtro é utilizado não só no mundo Servlet, como também no mundo EJB, Spring, VRaptor e até mesmo no Java Runtime, no qual existem outras implementações de filtros para resolver determinados problemas. Nesse caso, devemos importar `javax.servlet`.

```
package br.com.alura.gerenciador.servlet;

import javax.servlet.Filter;

public class MonitoramentoFilter implements Filter {

}
```

COPIAR CÓDIGO

O Eclipse irá apontar um erro, pois precisamos implementar três métodos. Clicando no ícone de erro no canto esquerdo, selecionaremos a opção "*Add unimplemented methods*". A partir do Java 8, temos um novo recurso com o qual as interfaces conseguem fornecer uma implementação padrão de um método. Na verdade são mais dois métodos, mas não somos obrigados a implementá-los. Dessa forma, restou apenas um método para implementarmos - o mesmo que acontece no Servlet, no qual somos responsáveis por imlementar o método `Service()`. No filtro, esse método se chama `doFilter`:

```
public class MonitoramentoFilter implements Filter {

    @Override
    public void doFilter(ServletRequest arg0, ServletResponse a
            throws IOException, ServletException {
        // TODO Auto-generated method stub

    }
```

```
}
```

COPIAR CÓDIGO

Como parâmetros, atribuiremos `request` para `ServletRequest`, `response` para `ServletResponse` e `chain` para `FilterChain`:

```
public class MonitoramentoFilter implements Filter {

    @Override
    public void doFilter(ServletRequest request, ServletRespons
            throws IOException, ServletException {

    }

}
```

COPIAR CÓDIGO

Quando reparamos que o filtro também recebe requisição e resposta como parâmetros, percebemos quão parecido ele é com o Servlet. Porém, o filtro recebe uma terceira referência, além de receber `ServletRequest` ao invés de `HttpServletRequest` - ou seja, ele é uma interface.

Nosso objetivo é utilizar o filtro para medirmos o tempo de execução. Portanto, vamos recortar o código que criamos em `ListaEmpresas` e colar na classe `MonitoramentoFilter`:

```
public class MonitoramentoFilter implements Filter {

    @Override
    public void doFilter(ServletRequest request, ServletRespons
            throws IOException, ServletException {
```

```
        long antes = System.currentTimeMillis();

        //executa a ação

        long depois = System.currentTimeMillis();

        System.out.println("Tempo de execução " + (depois - ant

    }

}
```

COPIAR CÓDIGO

Queremos que a requisição continue sendo executada após passar pelo filtro. Isso é feito por meio do objeto `chain` ("cadeia").

```
long antes = System.currentTimeMillis();

//executa a ação
chain.doFilter(request, response);

long depois = System.currentTimeMillis();

System.out.println("Tempo de execução " + (depois - antes));
```

COPIAR CÓDIGO

Se não chamarmos o método `chain.doFilter()` , a requisição irá parar. Essa é justamente a diferença entre um filtro e um Servlet - a capacidade de pararmos a requisição nesse método, por exemplo, se o usuário não tem permissão de executá-la.

Nosso método `doFilter()` está pronto: estamos medindo o tempo antes da requisição, o Servlet continua sendo executado e depois medimos o tempo novamente. Mas ainda falta uma coisa: o mapeamento. O filtro e o Servlet são

objetos que são chamados a partir de uma requisição Http, portanto precisaremos mapear a requisição para execução desse objeto.

Nesse caso, o mapeamento é `WebFilter()`, para o qual passaremos o mesmo objeto do nosso Servlet: `/entrada`.

> Dica: o uso de `urlPatterns=` é opcional, e poderia ter sido implementado também no nosso Servlet.

```
@WebFilter(urlPatterns="/entrada")
public class MonitoramentoFilter implements Filter {

    @Override
    public void doFilter(ServletRequest request, ServletRespons
            throws IOException, ServletException {

        long antes = System.currentTimeMillis();

        //executa a ação

        chain.doFilter(request, response);

        long depois = System.currentTimeMillis();

        System.out.println("Tempo de execução " + (depois - ant

    }

}
```

COPIAR CÓDIGO

Dessa forma, todas as requisições que chegarem no nosso `UnicaEntradaServlet` também chegarão, antes, no `MonitoramentoFilter`.

Recapitulando: a requisição passa pelo filtro, que faz a medição do tempo e delega a requisição para o próximo passo, que é o Servlet. O Servlet cumpre o seu papel de controlador, passando a requisição para o JSP, que então volta pelo filtro, fazendo a medição do tempo novamente e imprimindo ela no console.

Vamos testar? Quando efetuamos o login na aplicação, chamando `ListaEmpresas`, nosso filtro já recebe duas requisições, e os respectivos tempos de resposta são impressos no console.

Lembrando que tudo que podemos fazer em um Servlet, também podemos fazer em um filtro. Por exemplo, como temos `request` e `response`, podemos chamar o método `request.getParameter()` para lermos o parâmetro `acao`, que define o nome da ação que a aplicação está executando.

A ideia é imprimir a ação atual. Faremos isso escrevendo:

```
long antes = System.currentTimeMillis();

        String acao = request.getParameter("acao");
        //executa a ação

        chain.doFilter(request, response);

        long depois = System.currentTimeMillis();

        System.out.println("Tempo de execução da ação "+ acao
```

COPIAR CÓDIGO

Testaremos novamente efetuando o login. O console retornará algo como:

```
Logando nico
```

Usuario existe

Tempo de execução da ação Login -> 32

listando empresas

Tempo de execução da ação ListaEmpresas -> 248

Repare que cada ação nos devolve uma mensagem de que o código foi executado, mas tivemos que mexer em um único lugar. Agora, se quiséssemos alterar o monitoramento, poderíamos fazer isso facilmente. Portanto, os filtros nos ajudam a centralizar as preocupações do nosso código ao invés de espalharmos pelas diferentes ações.

Também é muito simples habilitar e desabilitar esse filtro, sem que seja necessário alterar as outras ações. Até poderíamos ter uma configuração na aplicação que lesse dinamicamente se o usuário deseja ou não fazer o monitoramento.

Nós ainda não resolvemos a questão da autenticação. O próximo passo é extrairmos o código que escrevemos para essa funcionalidade e criarmos um filtro separado que fará a verificação do login do usuário. No próximo vídeo, implementatemos esse filtro e revisaremos os conceitos desse recurso importante do mundo Servlet. Até lá!