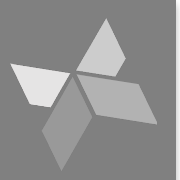
argonavis
tecnologia e arte

WEB
servlets & webapps

JAVA EE 7

Helder da Rocha (helder@acm.org)

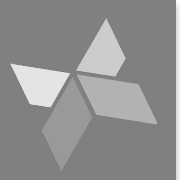
Atualizado em maio de 2015



Sobre este tutorial

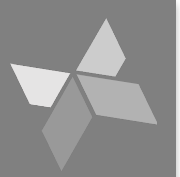
Este é um tutorial sobre tecnologia de WebServlets (de acordo com a especificação Java EE 7) criado para cursos presenciais e práticos de programação por Helder da Rocha

O código dos exemplos está disponível em repositório público no GitHub e é software livre sob os termos da licença Apache 2.0.



Conteúdo

1. Aplicações Web
2. WebServlets
3. Requisição e resposta HTTP
4. URLs e redirecionamento
5. Contexto da aplicação
6. Sessão do cliente
7. Filtros interceptadores
8. Arquitetura MVC
9. Integração com bancos de dados
10. Segurança

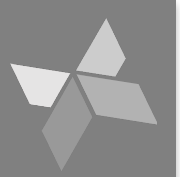


JAVAEE7

WEB

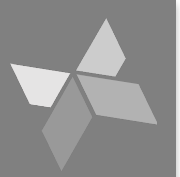
servlets & webapps

aplicações web



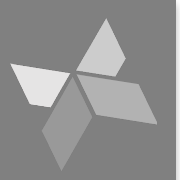
WebServlets

- WebServlets são a base das **aplicações Web** em Java EE
- São componentes que executam **métodos HTTP**
- Embora a principal API para criação de aplicações Web seja JavaServer Faces, WebServlets é a **plataforma básica** sobre a qual são construídos o JSF e frameworks de outros fabricantes (Spring MVC, Wicket, GWT, etc.)
- WebServlets servem de suporte a vários outros serviços Java EE como WebServices REST e SOAP, e WebSockets



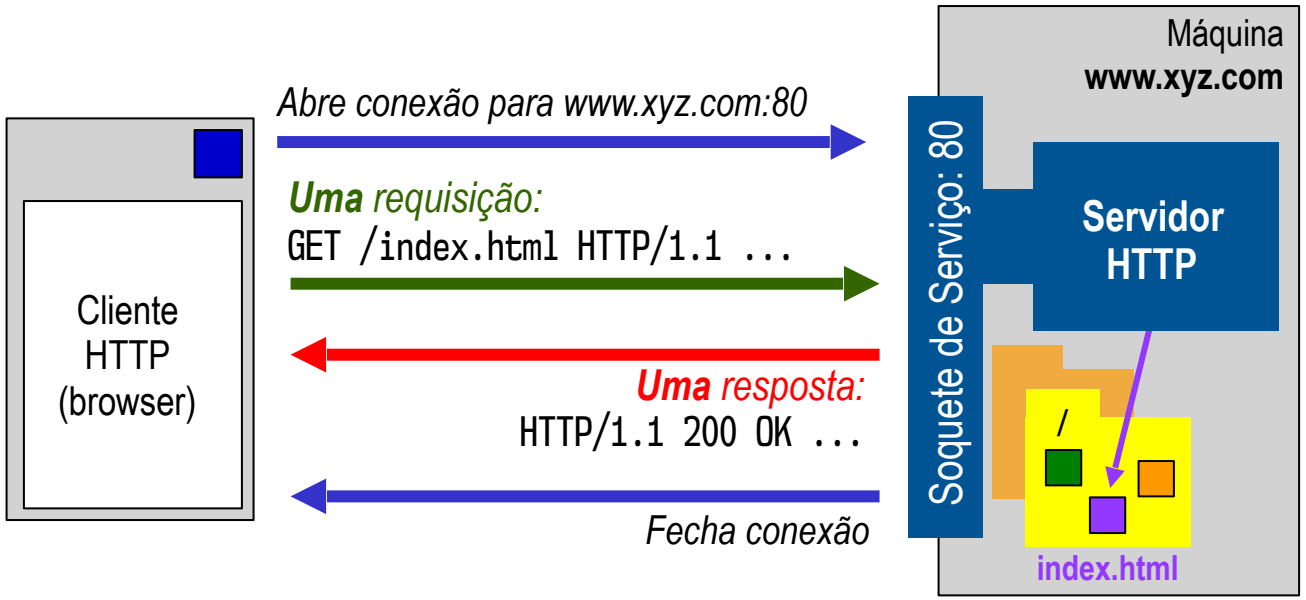
Arquitetura Web

- A arquitetura Web é baseada em
  - **cliente** (geralmente um browser ou um script)
  - **protocolo HTTP/1** (padrão Internet, RFC 2068) stateless
  - **servidor** Web (que pode estar sob controle de um servidor de aplicações)
- O protocolo HTTP consiste de **requisições** e **respostas**, que transportam **meta-informação** (cabeçalhos) e **dados**
- O servidor representa um **sistema de arquivos virtual** e responde a requisições que contém URLs para cada recurso disponibilizado.



Arquitetura Web

Requisição HTTP iniciada por um browser e
Resposta HTTP retornada por um servidor Web



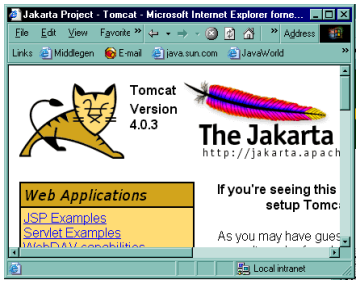
Só é garantida uma requisição/resposta por conexão

Requisição e resposta HTTP/1.x

- 1. Página HTML

```
<img src="tomcat.gif"/>
```

Interpreta HTML



- 2. Requisição: browser solicita imagem

Linha em branco termina cabeçalhos

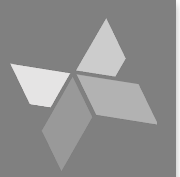
```
GET /tomcat.gif HTTP/1.1
User-Agent: Mozilla 6.0 [en] (Windows 95; I)
Cookies: querty=uiop; SessionID=D236S11943245
```

Envia requisição ***GET***

- 3. Resposta: servidor devolve cabeçalho + stream

```
HTTP 1.1 200 OK
Server: Apache 1.32
Date: Friday, August 13, 2003 03:12:56 GMT-03
Content-type: image/gif
Content-length: 23779

!#GIF89~¾7
.55.a6¤Ü4 ...
```

tomcat.gif



Aplicações Web

- A especificação de WebServlets descreve como empacotar **aplicações Web** para implantação em containers Web Java EE.
- Uma aplicação Web Java EE **mínima** (Java EE 7 / Servlet 3.1) consiste de pelo menos um arquivo HTML empacotado em um **arquivo WAR** (ZIP), contendo uma pasta WEB-INF

- Uma aplicação Web Java EE **típica** consiste de: arquivos XHTML, JSP ou HTML, imagens, arquivos JavaScript e CSS, arquivos XML de configuração, arquivo **web.xml**, arquivos .properties, JARs e classes Java

Arquivo WAR

- **ZIP** com estrutura de arquivos e diretórios.
- A raiz do arquivo é a **raiz do contexto** da aplicação: quaisquer arquivos ou pastas lá contidas serão por default **acessíveis** aos clientes Web.
- A pasta **WEB-INF** é privativa e **inacessível**
- Dentro de WEB-INF fica o arquivo **web.xml**, usado para configuração
- Se houver classes Java (servlets, managed beans) elas ou estarão em **WEB-INF/classes** (o classpath) ou empacotadas em um JAR dentro de **WEB-INF/lib**

biblioteca.war

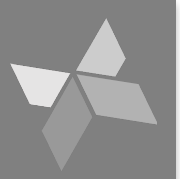
```
index.xhtml
login.xhtml
logo.png
css/
  biblioteca.css
js/
  jquery.js
  app.js
WEB-INF/
  web.xml
  classes/
    br/com/unijava/biblioteca/
      CadastroServlet.class
      EmprestimoServlet.class
  lib/
    ojdbc7.jar
```

Contexto de um arquivo WAR

- O contexto da aplicação geralmente é mapeado como um **subcontexto da raiz de documentos** (DOCUMENT_ROOT) do servidor (geralmente **/**)
- Por default, o **nome do contexto** é o nome do WAR (mas é comum que esse nome seja alterado na configuração da aplicação)
- Por exemplo, a aplicação **biblioteca.war**, publicada em um servidor Tomcat em **http://localhost:8080** será acessível, por default, em

`http://localhost:8080/biblioteca`

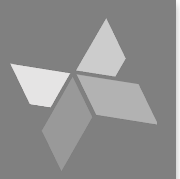
(a raiz de documentos é **/** e o contexto é **biblioteca**)



Configuração: web.xml

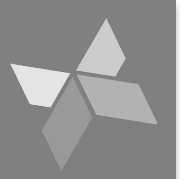
- O arquivo **web.xml** é o **Web Deployment Descriptor**
- **Opcional** (em aplicações Java EE 7 muito simples). Se estiver presente deve aparecer dentro de **WEB-INF/**
- A maior parte das configurações feitas no web.xml podem também ser feitas via **anotações** (web.xml tem precedência)
- Estrutura **mínima**:

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                             http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

</web-app>
```



Exercícios

- 1. Instale a aplicação **orbitas.war** no servidor (Tomcat ou Glassfish ou WildFly) e abra em **http://localhost:8080/orbitas**
- 2. Configure seu ambiente para o desenvolvimento de aplicações Web
  - Eclipse/NetBeans + Maven
  - Tomcat ou WildFly ou Glassfish
  - Abra o projeto **orbitas2**, verifique se há configurações a fazer (ex: configurar o servidor no Eclipse) e execute pelo IDE. A execução deve abrir uma janela do browser com a aplicação
  - Opcionalmente exporte o WAR pelo Eclipse e instale no TOMCAT através do Tomcat Manager ou copiando o WAR para a pasta WebApps/



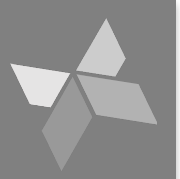
JAVAEE7

WEB

servlets & webapps

2

webservlets

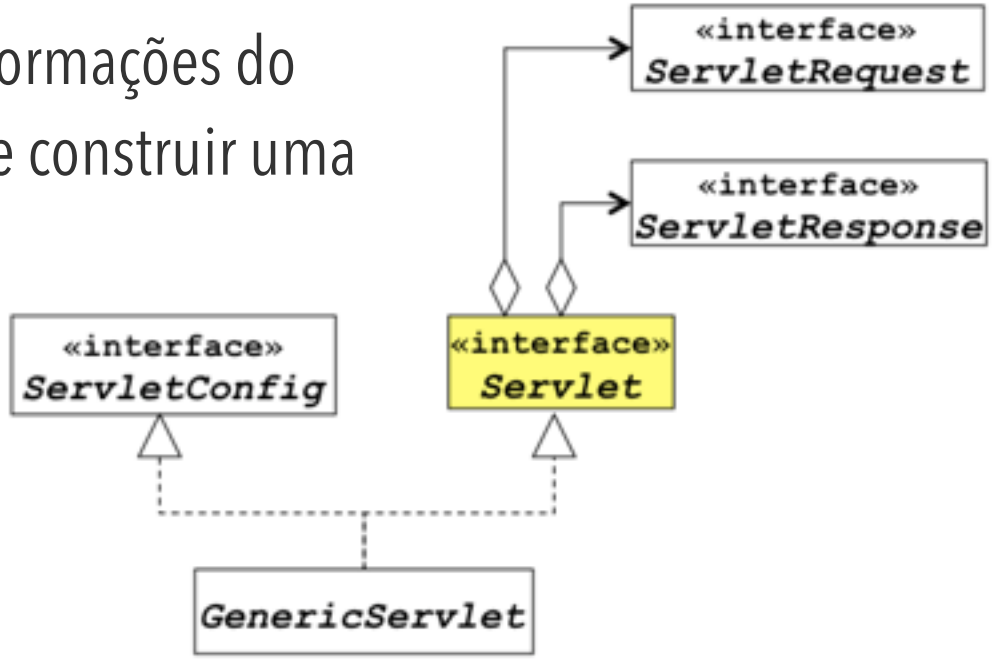


Construção de WebServlets

- **WebServlets** são classes Java que representam uma **requisição** seguida de **resposta HTTP**
- WebServlets são um tipo de **Servlet** (javax.servlet.Servlet) construído especialmente para executar serviços HTTP
- WebServlets são o componente fundamental para qualquer tipo de aplicação Java baseada em HTTP (aplicações Web comuns, aplicações JSF, Spring MVC, Google GWT, Web Services SOAP e RESTful, WebSocket handshakes, etc)

Servlets (javax.servlet)

- A interface **java.servlet.Servlet** representa um **executor** que executa um **serviço**
- O seu método **service(ServletRequest, ServletResponse)** encapsula o serviço realizado
- Um serviço pode obter informações do objeto **ServletRequest**, e construir uma resposta preenchendo um **ServletResponse**.

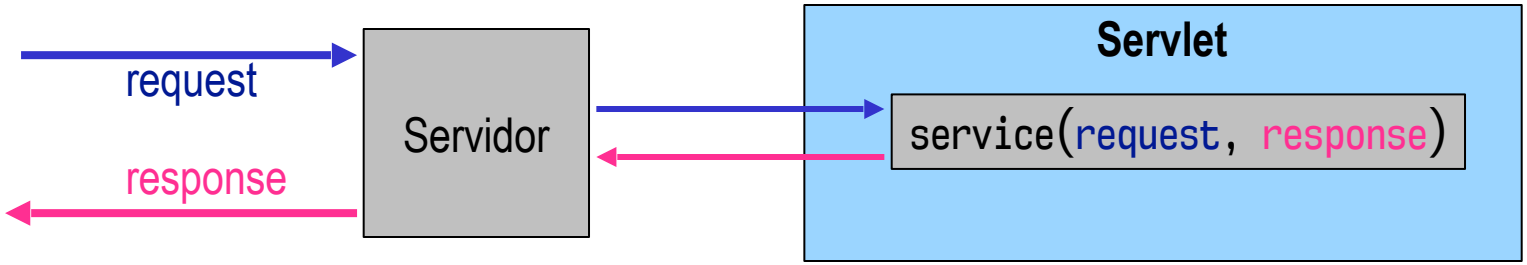


Método service(request, response)

- O **método de serviço** de um servlet genérico é:

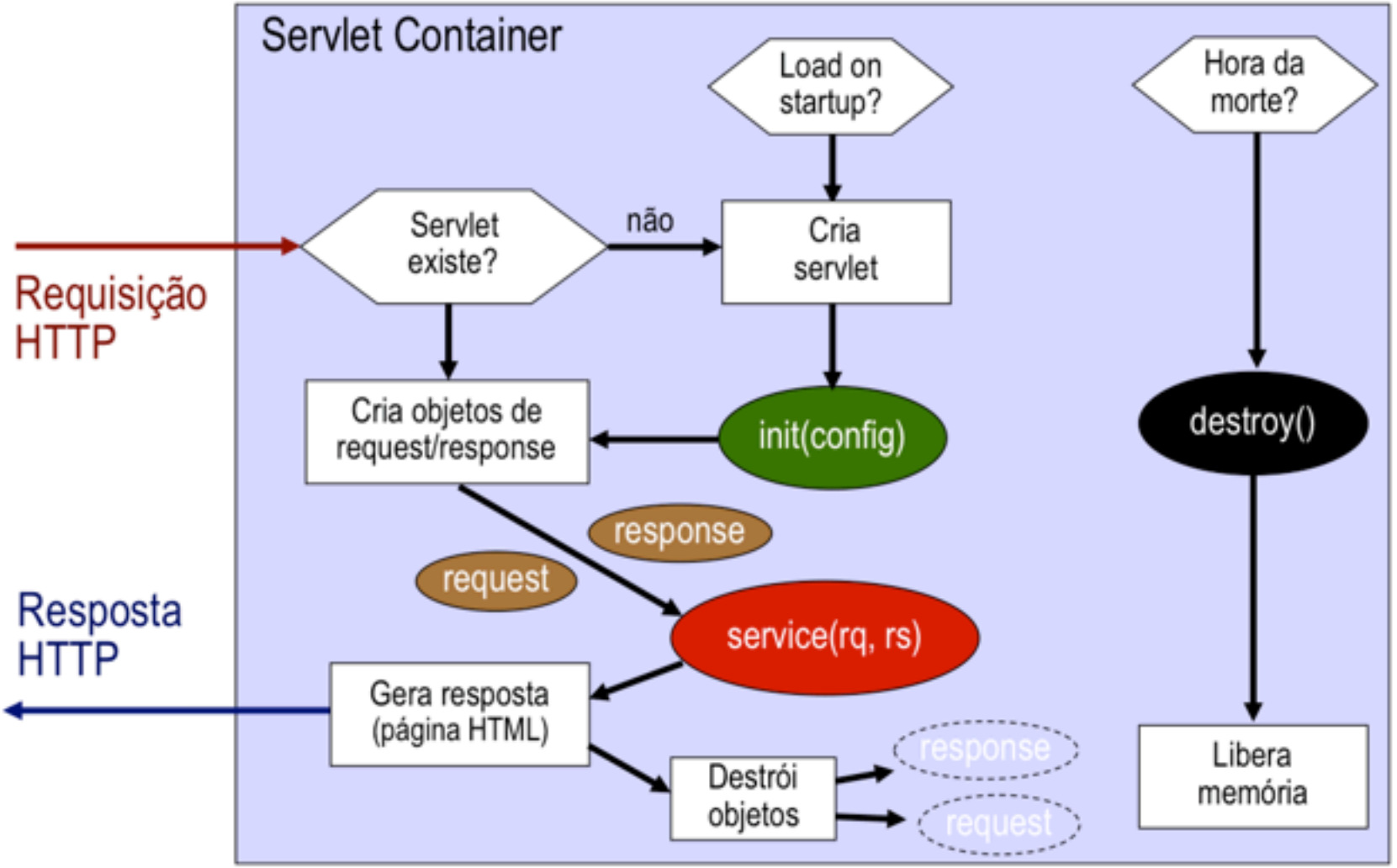
```
public void service(ServletRequest, ServletResponse)
```

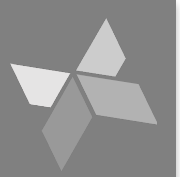
- Sempre que um servidor repassar uma requisição a um servlet, ele executará este método



- Um servlet genérico que sobrepõe este método poderá utilizar o objetos **ServletRequest** para ler os dados da requisição e **ServletResponse** para compor os dados da resposta

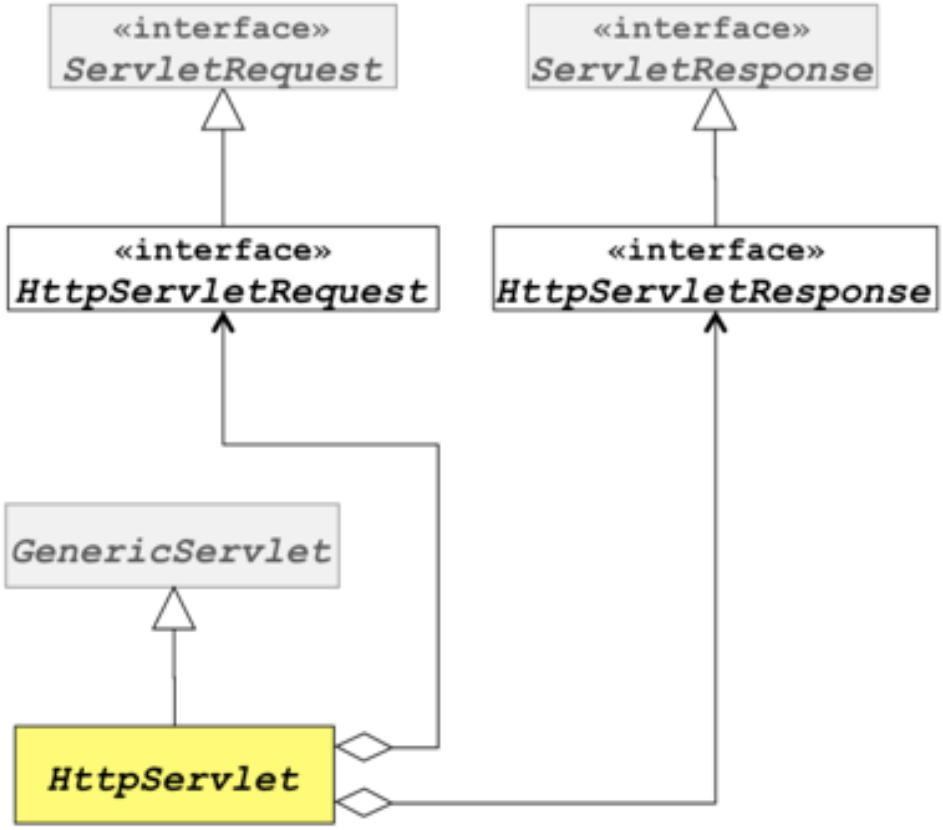
Ciclo de vida de um servlet

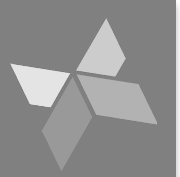




WebServlet (javax.servlet.http.HttpServlet)

- Um WebServlet **implementa** a interface Servlet
- O método service() de um WebServlet **delega** para métodos que lidam com cada método do protocolo HTTP (GET, POST, OPTIONS, PUT, etc.)
- Cada método recebe um **HttpServletRequest** e um **HttpServletResponse**



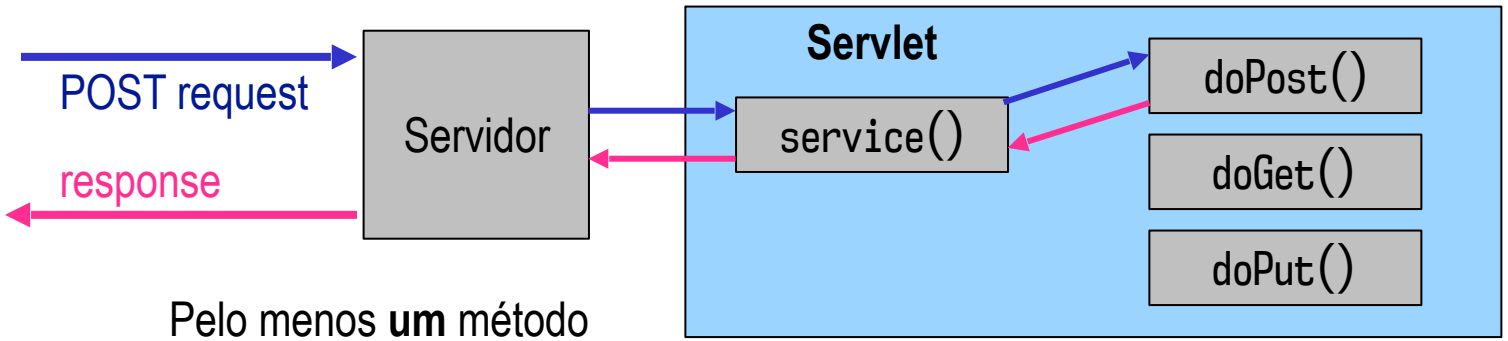


Métodos service() de um WebServlet

- A classe HttpServlet implementa métodos que refletem os métodos da plataforma HTTP/1: GET, POST, HEAD, PUT, DELETE, OPTIONS,TRACE

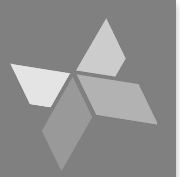
```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {...}
public void doPost(HttpServletRequest request,
                   HttpServletResponse response) {...}
...
```

- Pedidos recebidos por service() são **redirecionados** para métodos **doXxx()** correspondentes



Criando um WebServlet

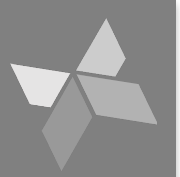
- Para criar um WebServlet é preciso **estender** a classe HttpServlet e implementar **pelo menos um** método de serviço (ex: doGet())
- O método deve ser compatível com a requisição HTTP (ex: GET) e tem a forma **doXxx(HttpServletRequest, HttpServletResponse)** onde Xxx é um método HTTP com a primeira letra maiúscula)
- O **request** vem preenchido com dados da requisição e cabeçalhos, possíveis parâmetros e cookies enviados pelo cliente
- O **response** deve ser configurado e preenchido, podendo-se obter um stream de saída para produzir uma pagina de resposta



Exemplo: WebServlet

- Um WebServlet mínimo que responde a comandos HTTP **GET**
- Este servlet ignora o objeto request; usa o **response** para obter um stream e imprimir HTML

```
public class ServletWeb extends HttpServlet {

    public void doGet (HttpServletRequest  request,
                       HttpServletResponse response)
                                           throws IOException {

        Writer out = response.getWriter();
        response.setContentType("text/html");
        out.println("<h1>Hello</h1>");
        out.close();
    }
}
```



Como implementar doGet() e doPost()

- Use **doGet()** para receber requisições GET
  - Links clicados ou URL digitadas diretamente
- Use **doPost()** para receber dados de formulários
  - Se quiser executar código independente do método delegue doGet() e doPost() para um método comum (não sobreponha service)

```
public class ServletWeb extends HttpServlet {
   public void doGet (HttpServletRequest request, HttpServletResponse response) {
         processarRequisicao(request, response);
   }
   public void doPost (HttpServletRequest request, HttpServletResponse response) {
         processarRequisicao(request, response);
   }
   public void processarRequisicao(HttpServletRequest request,
                                   HttpServletResponse response) {
         ...
   }
}
```

Mapeamento de um webservlet

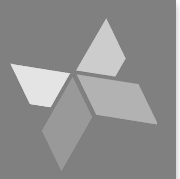
- Para ser acessado, o WebServlet precisa ser **mapeado** a um **caminho** acessível publicamente através do **contexto**

- Uma das maneiras de mapear um servlet é através de **anotações**:

```
@WebServlet("/HelloServlet")
public class ServletWeb extends HttpServlet { ... }
```

- Compile e copie a classe para uma pasta **WEB-INF/classes** e empacote em um ZIP chamado **contexto.war**. Depois instale em um servidor **Tomcat** que serve em **localhost:8080**, e será possível executar o servlet através de um browser chamando a URL:

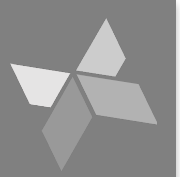
```
http://localhost:8080/contexto/HelloServlet
```



Mapeamento em web.xml

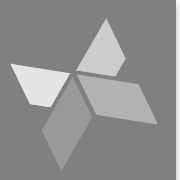
- Um servlet também pode ser mapeado no arquivo **web.xml**, usando dois conjuntos de tags: um para associar a **classe** do servlet a uma **variável**, e o outro para associar essa **variável** a um padrão de **URL**:

```
<web-app ...>
    ...
    <servlet>
        <servlet-name>umServlet</servlet-name>
        <servlet-class>pacote.ServletWeb</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>umServlet</servlet-name>
        <url-pattern>/HelloServlet</url-pattern>
    </servlet-mapping>
</web-app>
```



Padrão de URL

- Pode ter vários formatos (os caminhos são relativos ao contexto):
  - **/nome** – mapeamento exato (nome do servlet é /nome). Outro exemplo: **/nome/subnome** (nome do servlet é /nome/subnome)
  - **/nome/*** – mapeamento com parâmetro. Aceita texto depois do nome fixo. Ex: /nome/a (nome do servlet é /nome, path-info é /a). Outro exemplo: **/nome/subnome/*** (nome do servlet é /nome/subnome). Ex: /nome/subnome/a/b/c (path-info é /a/b/c)
  - **/** – default: este servlet recebe requisições com URLs que não combinam com nenhum mapeamento configurado (nome do servlet é /)
  - ***.ext** - mapeamento de extensão (absoluto, dentro do contexto). Qualquer URL terminando nesta extensão redireciona a requisição para o servlet.



Mapeamentos com anotação

- Um mapeamento pode ser usado para passar parâmetros

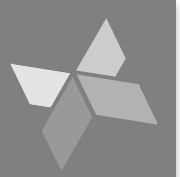
```
@WebServlet("/inserir/*")
public class InserirProdutoServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, ... ) {
        String parametro = request.getPathInfo();
    }
}
```

Se URL for /inserir/**123**, parametro será **/123**
Se URL for /inserir/**a/b/c**, parametro será **/a/b/c**

- Um servlet pode ser mapeado a várias URLs.

```
@WebServlet(urlPatterns = {"/listar", "/detalhar"})
public class ProdutoServlet extends HttpServlet { … }
```

Se URL for /listar ou /detalhar este servlet receberá a requisição
(nome /listar ou /detalhar poderá ser lido com **request.getServletName()**)



Parâmetros de inicialização

- Podem ser definidos para cada servlet no **web.xml**:

```
<servlet>
    <servlet-name>umServlet</servlet-name>
    <servlet-class>pacote.subp.MeuServlet</servlet-class>
    <init-param>
      <param-name>dir-imagens</param-name>
      <param-value>c:/imagens</param-value>
    </init-param>
    <init-param> ... </init-param>
</servlet>
```

- Podem ser recuperados dentro do servlet:

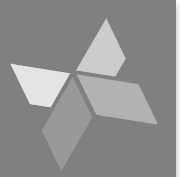
```
String dirImagensStr = getInitParameter("dir-imagens");
```

- Também podem ser definidos via anotações no servlet:

```
@WebServlet( urlPatterns = "/HelloServlet",
    initParams = @WebInitParam(name = "dir-imagens", value = "C:/imagens"))
public class ServletWeb extends HttpServlet { … }
```

Exercícios

- 1. Escreva um servlet GET que imprima a data e a hora atuais em um bloco <h1> do HTML
  - Mapeie o servlet ao nome "/datahora"
  - Instale e teste a aplicação
- 2. Escreva um servlet GET que imprima uma tabela HTML relacionando um número n com o seu quadrado (n*n)
  - Mapeie o servlet como "/quadrado/*" e obtenha o número como extra-path-info (use getPathInfo())
  - Instale e teste a aplicação



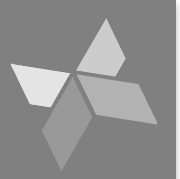
JAVAEE7

WEB

servlets & webapps

3

requisição e resposta http

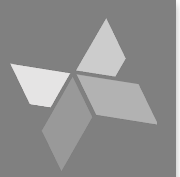


Requisição HTTP

- Uma requisição HTTP feita pelo browser tipicamente contém vários cabeçalhos RFC822 (padrão de cabeçalhos de email)

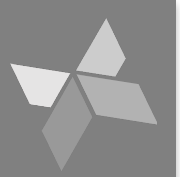
```
GET /docs/index.html HTTP/1.0
Connection: Keep-Alive
Host: localhost:8080
User-Agent: Mozilla 6.0 [en] (Windows 95; I)
Accept: image/gif, image/x-bitmap, image/jpg, image/png, */*
Accept-Charset: iso-8859-1, *
Cookies: jsessionid=G3472TS9382903
```

- Métodos de HttpServletRequest permitem extrair informações de qualquer um deles e informações de método e URL



Alguns métodos de HttpServletRequest

- Enumeration **getHeaderNames**() - obtém nomes dos cabeçalhos
- String **getHeader**("nome") - obtém primeiro valor do cabeçalho
- Enumeration **getHeaders**("nome") - todos os valores do cabeçalho
- String **getParameter**(param) - obtém parâmetro HTTP
- String[] **getParameterValues**(param) - obtém parâmetros repetidos
- Enumeration **getParameterNames**() - obtém nomes dos parâmetros
- Cookie[] **getCookies**() - recebe cookies do cliente
- HttpSession **getSession**() - retorna a sessão
- void **setAttribute**("nome", obj) - define um atributo obj chamado "nome".
- Object **getAttribute**("nome") - recupera atributo chamado nome



Parâmetros de uma requisição HTTP

- Pares **nome=valor** enviados pelo cliente em strings separados por **&**:

```
nome=Jo%E3o+Grand%E3o&id=agente007&acesso=3
```

- Se método for **GET**, parâmetros são passados no **Query String** (depois da **?**)

```
GET /servlet/Teste?id=nome=Jo%E3o+Grand%E3o&agente007&acesso=3 HTTP/1.0
```

- Se método for **POST**, parâmetros são stream no corpo da mensagem:

```
POST /servlet/Teste HTTP/1.0
Content-length: 33
Content-type: x-www-form-urlencoded

nome=Jo%E3o+Grand%E3o&id=agente007&acesso=3
```

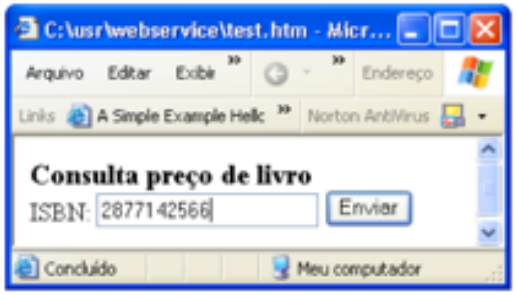
- Caracteres reservados e maiores que ASCII-7bit são codificados seguindo o padrão usado em URLs: (ex: ã = %E3)

- Os valores dos parâmetros podem ser recuperados pelo método getParameter():

```
String nome = request.getParameter("nome");
String id = request.getParameter("id");
String acesso = request.getParameter("acesso"); // deve ser convertido em int
```

Exemplo: POST

- Formato **text/x-www-form-urlencoded** representa dados lidos e codificados por um formulário.




```
<form action="/aplicacao/Consulta"
      method="post">
 <h3>Consulta preço de livro</h3>
 <p>ISBN: <input type="text" name="isbn">
 <input type="submit" VALUE="Enviar">
</form>
```

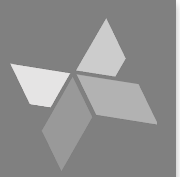
- Requisição **POST** gerada pelo envio do formulário

Cabeçalho HTTP

Linha em branco

Mensagem (corpo da requisição)

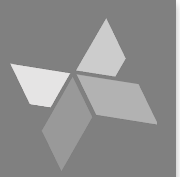
```
POST /aplicacao/Consulta HTTP/1.1
Content-type: text/x-www-form-urlencoded
Content-length: 15

isbn=2877142566
```



Exemplo: doPost

- Servlet (de aplicacao.war) que recebe requisição POST

```
@WebServlet("/Consulta")
public class ServletWeb extends HttpServlet {
   @Inject LivroDAO dao;

   public void doPost (HttpServletRequest request,
                       HttpServletResponse response) throws IOException {

        String isbn = request.getParameter("isbn");
        Livro livro = dao.getLivro(isbn);
        Writer out = response.getWriter();
        response.setContentType("text/html");
        out.println("<h1>"+ livro.getTitulo() +"</h1>");
   }
}
```



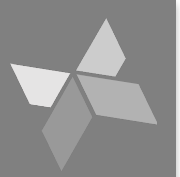
Parâmetros repetidos

- Parâmetros de **mesmo nome** podem estar repetidos.
  - Ocorre em checkboxes ou menus de seleção múltipla
  - A URL gerada concatena os valores.
  - **String getParameter()** retorna apenas a primeira ocorrência.
  - **String[] getParameterValues()** retorna todas
- Por exemplo os parâmetros da URL:

```
http://servidor/aplicacao?nome=Fulano&nome=Sicrano
```

- Podem ser recuperados usando:

```
String[] params = request.getParameterValues("nome");
```

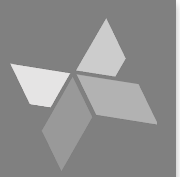


Resposta HTTP

- Uma **resposta HTTP** é enviada pelo servidor ao browser e contém informações sobre os dados anexados.

```
HTTP/1.0 200 OK
Content-type: text/html
Date: Mon, 7 Apr 2003 04:33:59 GMT-03
Server: Apache Tomcat/4.0.4 (HTTP/1.1 Connector)
Connection: close
Set-Cookie: jsessionid=G3472TS9382903

<HTML>
   <h1>Hello World!</h1>
</HTML>
```

- Os métodos de HttpServletResponse permitem construir **cabeçalhos** e anexar **dados** no corpo da resposta



Alguns métodos de HttpServletResponse

- void **addHeader**(String nome, String valor) - adiciona cabeçalho HTTP
- void **setContentType**(String tipo) - define o tipo MIME que será usado para gerar a saída (text/html, image/gif, etc.); mesmo que cabeçalho **Content-type: tipo**
- void **sendRedirect**(String url) - envia informação de redirecionamento para o cliente; mesmo que enviar o cabeçalho **Location: url**
- Writer **getWriter**() - obtém um Writer para gerar a saída. Ideal para saída de texto.
- OutputStream **getOutputStream**() - obtém um OutputStream. Ideal para gerar formatos diferentes de texto (imagens, etc.)
- void **addCookie**(Cookie c) - adiciona um novo cookie; mesmo que **Set-Cookie: c**
- void **encodeURL**(String url) - envia como anexo da URL a informação de identificador de sessão (sessionid)
- void **reset**() - limpa toda a saída inclusive os cabeçalhos
- void **resetBuffer**() - limpa toda a saída, exceto cabeçalhos

Como construir uma resposta

- Para construir uma resposta, é necessário obter um **fluxo de saída**, de caracteres (Writer) ou de bytes (OutputStream)

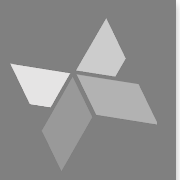
```
Writer out = response.getWriter();    // ou
OutputStream out = response.getOutputStream();
```

- Deve-se também definir o cabeçalho HTTP **Content-type** para que o browser saiba exibir as informações corretamente

```
response.setContentType("image/png");
```

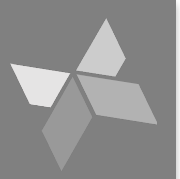
- Depois, pode-se **imprimir** os dados no objeto de saída (out)

```
byte[] image = gerarImagemPNG();
out.write(buffer);
```



Exercícios

- 1. Escreva um servlet que receba um número n e imprima uma tabela contendo uma sequência de 1 a n, relacionando n e n*n. O número deve ser passado através de um parâmetro da requisição.
- 2. Escreva um servlet POST e formulário HTML integrados
  - O formulário deverá receber um número e um nome
  - O servlet deverá imprimir o nome dentro de itens <li> um bloco <ul>, a quantidade de vezes representada pelo número



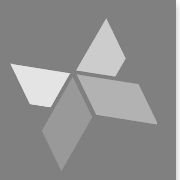
JAVAEE7

WEB

servlets & webapps

4

urls, redirecionamento e repasse



Componentes da URL

- Por default, o nome do **contexto do WAR** aparece na URL absoluta após o **nome/porta do servidor**:

  http://serv:8080/contexto/subdir/pagina.html
  http://serv:8080/contexto/servletMapeado

- Links relativos em páginas **estáticas** (HTML) são relativas a **DOCUMENT_ROOT** (ex: ao "/" em http://servidor:8080/) e não ao contexto do WAR

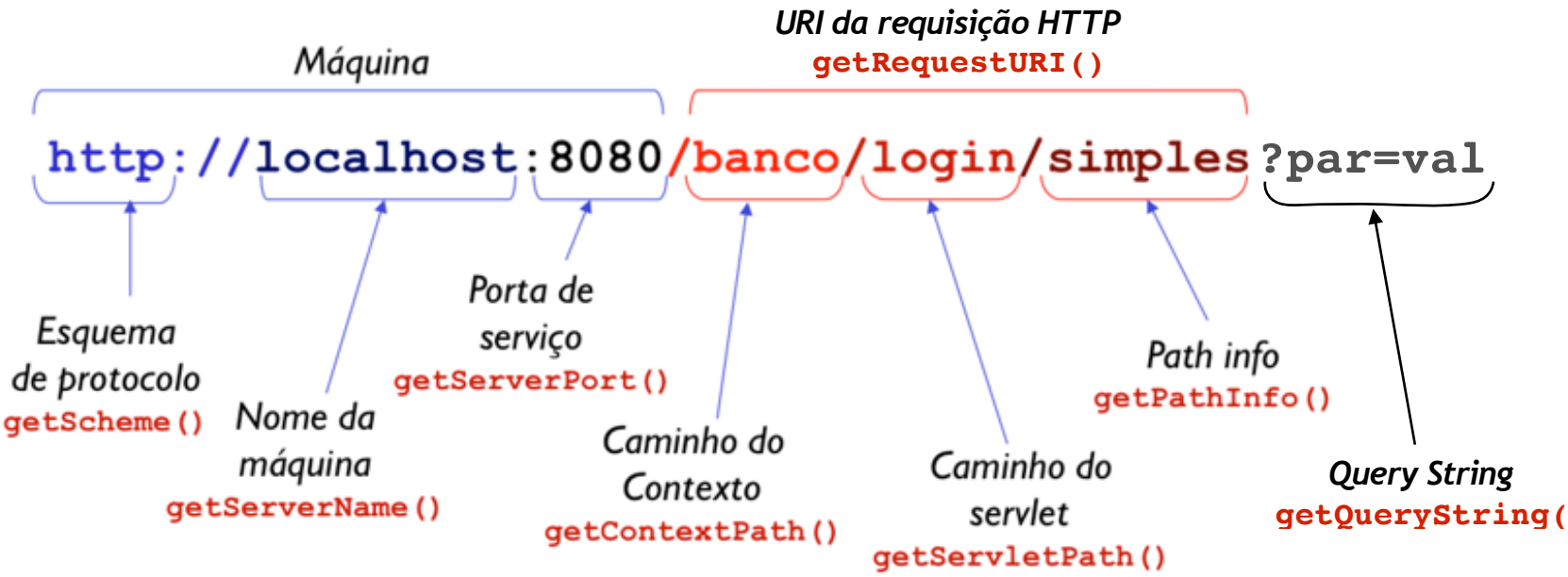
```
<a href="/contexto/subdir/pagina.html"> ...
<img src="/contexto/servletMapeado">
```

- Dentro do **web.xml** e em páginas **dinâmicas** (JSF/EL, JSP, servlets), URLs são relativas ao **contexto do WAR** (ex: http://servidor:8080/**contexto**)

```
dispatcher = request.getRequestDispatcher("/subdir/pagina.html");
<h:form action="/servletMapeado"> ...
```

Acesso a componentes da URL

- Diferentes partes de uma URL usada na requisição podem ser extraídas usando métodos de **HttpServletRequest** listados na figura abaixo



Processamento de URLs e mapeamentos

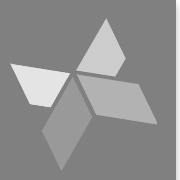
- Web container compara mapeamentos (web.xml/anotações) com URL recebida
  - 1. Procura primeiro um **mapeamento exato**
  - 2. Se não achar, procura entre os **caminhos que terminam em ***
  - 3. Por último, procura pelo mapeamento da **extensão do arquivo**, se houver
  - 4. Se comparação falhar, redireciona ao **servlet default** (mostra erro, se não existir)
- Texto adicional à direita é **path-info** (recupere com `request.getPathInfo()`)
- Por exemplo, o mapeamento **/um/*** e a URL
  - `http://localhost:8080/contexto/um/dois/tres/abc.txt`
- Mesmo existindo, mapeamento `*.txt` não será considerado pois `/um/*` tem precedência e `/dois/tres/abc.txt` será considerado path info!

Passagem de atributos na requisição

- Para compartilhar dados entre métodos, não use variáveis estáticas ou de instância. Em vez disso use atributos de requisição (HttpServletRequest)

```
void setAttribute("nome", objeto)
Object getAttribute("nome")
```

- Atributos são destruídos com a requisição (quando service termina) e não são compartilhados entre clientes (são thread-safe!)
  - Forma recomendada de comunicação entre métodos na mesma requisição
  - Se desejar que o valor dure mais que uma requisição, copie-os para um objeto de persistência maior (por exemplo, um atributo de contexto)

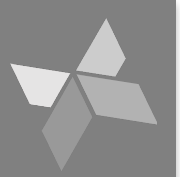


Repasse de requisição

- Um **RequestDispatcher** repassa **requisições** para outra página ou servlet

```
HttpRequestDispatcher dispatcher =
   request.getRequestDispatcher("url");
dispatcher.forward(request, response);
```

- Seus dois principais métodos são
  - **include(request, response)** - despacha a requisição, mas mantém responsabilidade pela resposta (o resultado da URL é incluída)
  - **forward(request, response)** - despacha a requisição e a responsabilidade pela resposta (quem gera a resposta é o servlet que receber a requisição)
- No repasse, o controle não passa pelo browser (não é um redirecionamento) e todos os parâmetros e atributos da requisição são preservados



Redirecionamento x Repasse

- Pode-se enviar um cabeçalho de **redirecionamento** para o browser usando

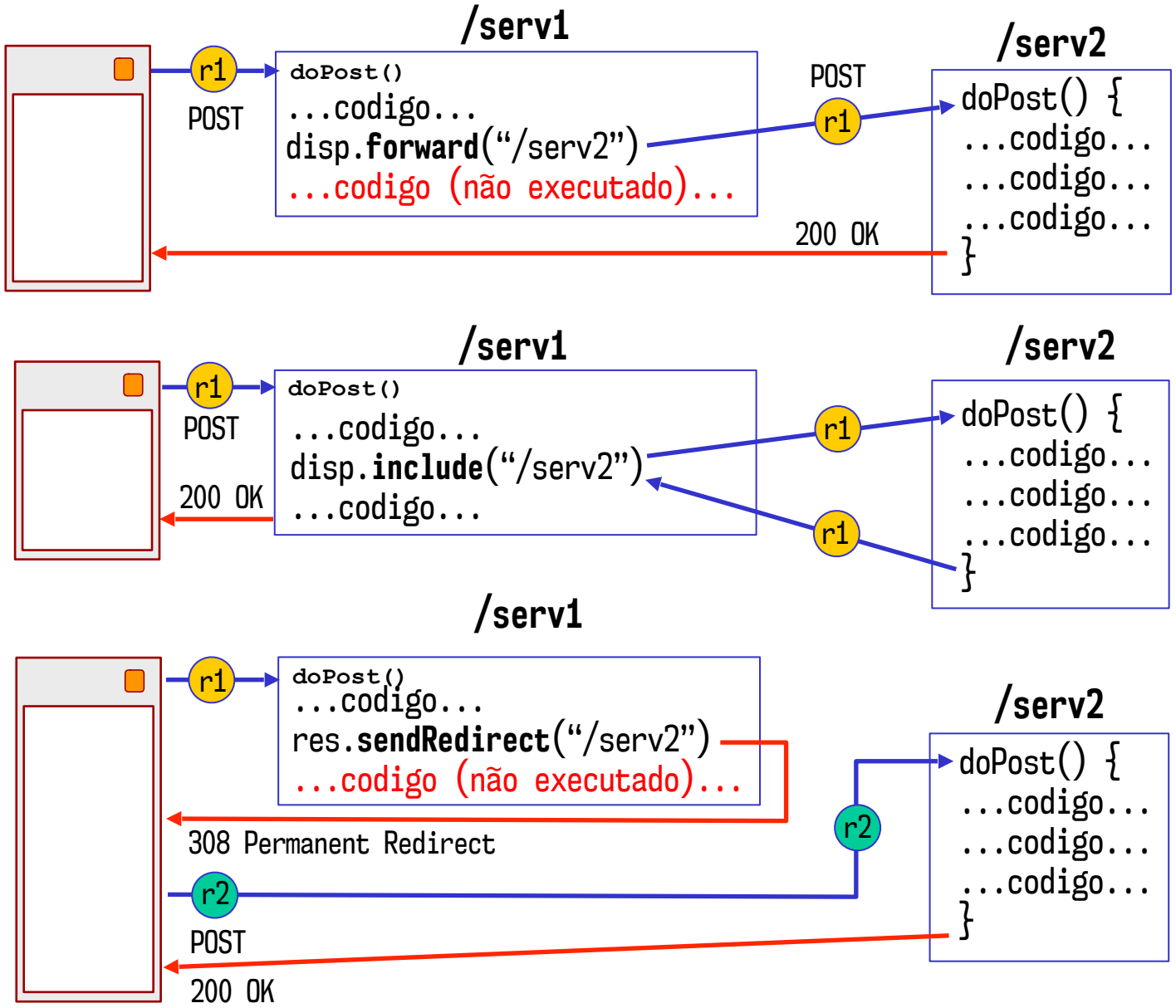
```
response.sendRedirect("url");
```

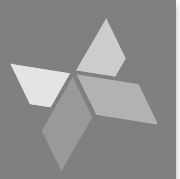
- Isto é o mesmo que

```
response.setHeader("Location", "url");
```

- Location instrui o browser para redirecionar para outro lugar
- Sempre que o controle volta ao browser, a primeira requisição terminou e outra requisição foi iniciada (os objetos HttpServletResponse e HttpServletRequest e todos seus atributos e parâmetros foram destruídos)
- Com **repasse** de requisições, usando um RequestDispatcher obtido do objeto request, o controle não volta ao browser mas continua em outro servlet (forward()) ou no mesmo servlet (include())

Redirecionamento x Repasse





Erros em servlets

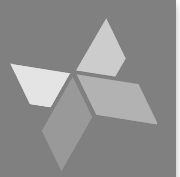
- Dois tipos de erros podem ocorrer em servlets
  - Erros HTTP
  - Exceções (HTTP 500)
- Erros HTTP são identificados pelo servidor através de um código de status
  - Códigos de status que começam com **1**, **2** ou **3** não indicam erro
  - Códigos que começam com **4** (404, 401, etc.) indicam erro originado no cliente (que, em tese, poderia ser corrigido pelo browser - ex: desviar para outra página)
  - Códigos que começam com **5** (500, 501, etc.) indicam erro no servidor (não adianta o browser tentar evitá-lo)

Repasse para páginas de erro

- Quando acontece um erro do servidor, causado ou não por uma exceção, o servidor normalmente mostra uma página default
- No web.xml, pode-se mudar esse comportamento definindo o repasse da requisição a um componente, de acordo com o código de erro HTTP ou exceção
- No exemplo abaixo <error-page> repassa a requisição para uma página HTML se houver erro 404 e a um servlet em caso de FileNotFoundException

```
<error-page>
  <error-code>404</error-code>
  <location>/notFound.html</location>
</error-page>

<error-page>
  <exception-type>java.io.FileNotFoundException</exception-type>
  <location>/servlet/j550.error.IOExceptionServlet</location>
</error-page>
```

Pode haver qualquer número de elementos <error-page> mas apenas um para cada tipo de exceção específica ou código de erro HTTP



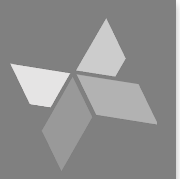
Atributos especiais

- Destinos de erro recebem dois **atributos de requisição** que podem ser lidas em componentes dinâmicos (servlet / JSP / Facelets)
  - Atributo: javax.servlet.error.request_uri. Tipo: String
  - Atributo: javax.servlet.error.exception. Tipo: Throwable
- A exceção corresponde à embutida dentro de ServletException (a causa)

```
public void doGet(... request, ... response) {
  Throwable exception = (Throwable)
    request.getAttribute("javax.servlet.error.exception");
  String urlCausadora = (String)
    request.getAttribute("javax.servlet.error.request_uri");

  // Faça alguma coisa com os dados recebidos

}
```

Exercícios

- 1. Escreva um servlet que receba um texto e um número (exercício anterior) mas que, se o texto for "quadrado" repasse a requisição para o servlet que imprime uma tabela de quadrados
  - Crie um método doPost() se necessário
- 2. Altere o servlet para que ele imprima o nome repetidas vezes e também inclua o resultado da tabela de quadrados
- 3. Altere o servlet para que ele redirecione para o servlet que mostra data e hora, se o texto recebido for "hora"
- 4. Escolha um servlet que imprima a) todos os componentes de sua URL, b) todos os cabeçalhos da requisição, e c) o método da requisição



JAVAEE7

WEB servlets & webapps

5

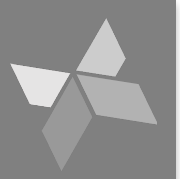
contexto da aplicação

3 O Contexto

- A interface **ServletContext** encapsula informações sobre o contexto ou aplicação
- Cada servlet possui um método **getServletContext()** que devolve o contexto atual
- A partir de uma referência ao contexto pode-se interagir com a **aplicação inteira** e compartilhar informações entre servlets

Componentes de um contexto

- **{Contexto}**
  - É a raiz da aplicação (internamente corresponde a "/")
  - Na raiz ficam páginas, imagens, applets e objetos para download via HTTP
- **{Contexto}/WEB-INF/web.xml**
  - Arquivo de configuração da aplicação (deployment descriptor)
  - Define parâmetros iniciais, mapeamentos, configurações de servlets e filtros
- **{Contexto}/WEB-INF/classes/**
  - Classpath da aplicação
- **{Contexto}/WEB_INF/lib/**
  - Qualquer JAR incluído terá o conteúdo carregado no CLASSPATH



Tipos e fragmentos de URL

- URL **absoluta**: identifica recurso na Internet. Usada no campo de entrada de localidade no browser, em páginas fora do servidor, etc.

  `http://serv:8080/ctx/servlet/cmd/um`

- **Relativa** ao **servidor** (Request URI): identifica o recurso no servidor. Usada no código processado pelo browser e é relativa a DOCUMENT_ROOT

  `/ctx/servlet/cmd/um`

- **Relativa** ao **contexto**: identifica o recurso dentro do contexto. Usada no código de processado no servidor (servlets, beans) e web.xml

  `/servlet/cmd/um`

- **Relativa** ao **componente** (extra path info): texto na URL após a identificação do componente (mapeamentos que terminam em *)

  `/cmd/um`

Exemplo de web.xml (1/2)

```
<web-app>
  <context-param>
    <param-name>tempdir</param-name>
    <param-value>/tmp</param-value>
  </context-param>

  <servlet>
    <servlet-name>myServlet</servlet-name>
    <servlet-class>exemplo.pacote.MyServlet</servlet-class>
    <init-param>
      <param-name>datafile</param-name>
      <param-value>data/data.txt</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>myServlet</servlet-name>
    <url-pattern>/myservlet</url-pattern>
  </servlet-mapping>
                          ...
```

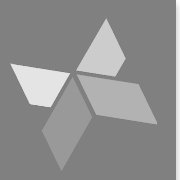
Parâmetro que pode ser lido por todos os componentes

Instância de um Servlet

Parâmetro que pode ser lido pelo servlet

Ordem para carga prévia do servlet

myServlet mapeado à URL /myservlet



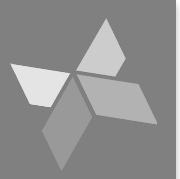
Exemplo de web.xml (2/2)

```
...

  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.xhtml</welcome-file>
  </welcome-file-list>

  <error-page>
    <error-code>404</error-code>
    <location>/notFound.jsp</location>
  </error-page>

</web-app>
```

Sessão do usuário expira expira em 60 minutos

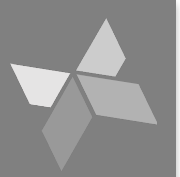
Lista de arquivos que serão carregados automaticamente em URLs terminadas em diretório

Redirecionar para esta página em caso de erro 404



Principais métodos de ServletContext

- String **getInitParameter**(String): obtém parâmetros de inicialização do contexto (não confunda com o método similar de ServletConfig)
- Enumeration **getInitParameterNames**(): obtém lista de parâmetros
- InputStream **getResourceAsStream**(): obtém um stream para recurso localizado dentro do contexto
- void **setAttribute**(String nome, Object): grava um atributo no contexto
- Object **getAttribute**(String nome): lê um atributo do contexto
- void **log**(String mensagem): escreve mensagem no log do contexto



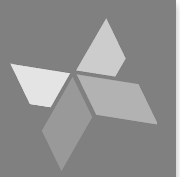
Inicialização do contexto

- E possível configurar parâmetros de configuração para o contexto no web.xml. Parâmetros têm nome e valor e são definidos em **<context-param>**:

```
<context-param>
    <param-name>tempdir</param-name>
    <param-value>/tmp</param-value>
</context-param>
```

- Para ler um parâmetro no servlet, é preciso obter acesso à instância de ServletContext usando o método **getServletContext()**:

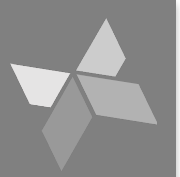
```
ServletContext ctx = this.getServletContext();
String tempDir = ctx.getInitParameter("tempdir");

if (tempDir == null) {
   throw new UnavailableException("Configuração errada");
}
```



Resources

- Arquivos armazenados dentro do contexto do WAR
- Devem ser arquivos somente leitura: arquivos de configuração (XML, properties), imagens, etc.
- Método **getResourceAsStream()** localiza resource e carrega o stream

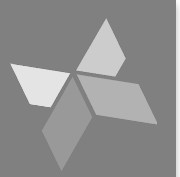
```
ServletContext ctx = getServletContext();
String arquivo = "/WEB-INF/usuarios.xml";
InputStream stream = ctx.getResourceAsStream(arquivo);
InputStreamReader reader = new InputStreamReader(stream);
BufferedReader in = new BufferedReader(reader);
String linha = "";
while ( (linha = in.readLine()) != null) {
   // Faz alguma coisa com linha de texto
}
```



Logging

- Para cada contexto criado, o servidor fornece um **arquivo de log**, onde mensagens serão gravadas
  - O arquivo será criado quando a primeira mensagem for gravada
- Há dois métodos disponíveis
  - **log**(String mensagem): grava uma mensagem
  - **log**(String mensagem, Throwable exception): grava uma mensagem e o stack-trace de uma exceção. Use nos blocos try-catch.
- Log é implementado em GenericServlet também. Pode ser chamado tanto do contexto como do servlet

```
this.log("Arquivo carregado com sucesso.");
ServletContext ctx = getServletContext();
ctx.log("Contexto obtido!");
```



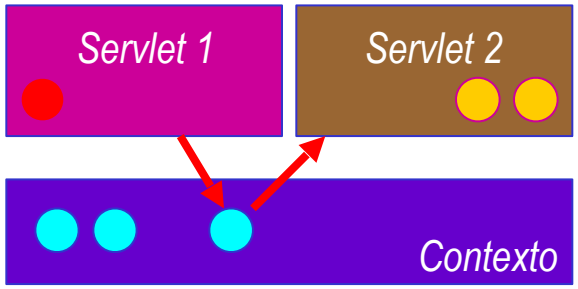
Atributos de contexto

- Objetos podem ser armazenados no contexto. Isto permite que sejam **compartilhados** entre servlets.

- O exemplo abaixo mostra como gravar um objeto no contexto com **setAttribute(chave, objeto)**:

SERVLET 1

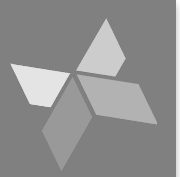
```
String[] vetor = {"um", "dois", "tres"};
ServletContext ctx =
    this.getServletContext();
ctx.setAttribute("dados", vetor);
```



- Para recuperar o objeto utilize **getAttribute(chave)**:

SERVLET 2

```
ServletContext ctx =
    this.getServletContext();
String[] dados = (String[])ctx.getAttribute("dados");
```



Listeners

- Um **ServletContextListener** pode controlar o ciclo de vida de um contexto implementando dois métodos:

```
public void contextInitialized(ServletContextEvent e);
public void contextDestroyed(ServletContextEvent e);
```

- O objeto **ServletContextEvent**, recebido em ambos os métodos, possui um método **getServletContext()** que permite obter o contexto associado.

- O listener precisa ser **registrado**. Isto pode ser feito com **@WebListener**:

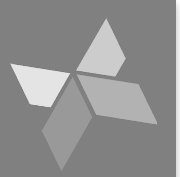
```
@WebListener
public class OuvinteDeContexto implements ServletContextListener {...}
```

- Ou no **web.xml** usando o elemento **<listener>**

```
<listener>
   <listener-class>ex01.OuvinteDeContexto</listener-class>
</listener>
```

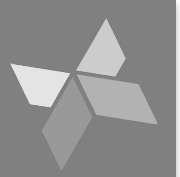
Outros listeners de contexto

- É possível saber **quando** um atributo foi adicionado a um contexto usando **ServletContextAttributeListener** e **ServletContextAttributeEvent**
- Métodos a implementar do Listener
  - attributeAdded(ServletContextAttributeEvent e)
  - attributeRemoved(ServletContextAttributeEvent e)
  - attributeReplaced(ServletContextAttributeEvent e)
- **ServletContextAttributeEvent** possui métodos para recuperar **nome** e **valor** dos atributos adicionados, removidos ou substituidos (valores que mudaram)
  - String getName()
  - String getValue()
- É preciso **registrar** o listener com **@WebListener** ou no **web.xml**



Exercícios

- 1. Escreva um listener que guarde no contexto um Map de filmes (código IMDB e nome - use o arquivo disponível)
  - Escreva um servlet que, quando chamado, imprima uma tabela contendo a lista de filmes obtida do contexto
- 2. Altere o listener para que ele carregue o arquivo WEB-INF/filmes.csv que contém os dados de filmes e use-o para montar o Map.



JAVAEE7

WEB

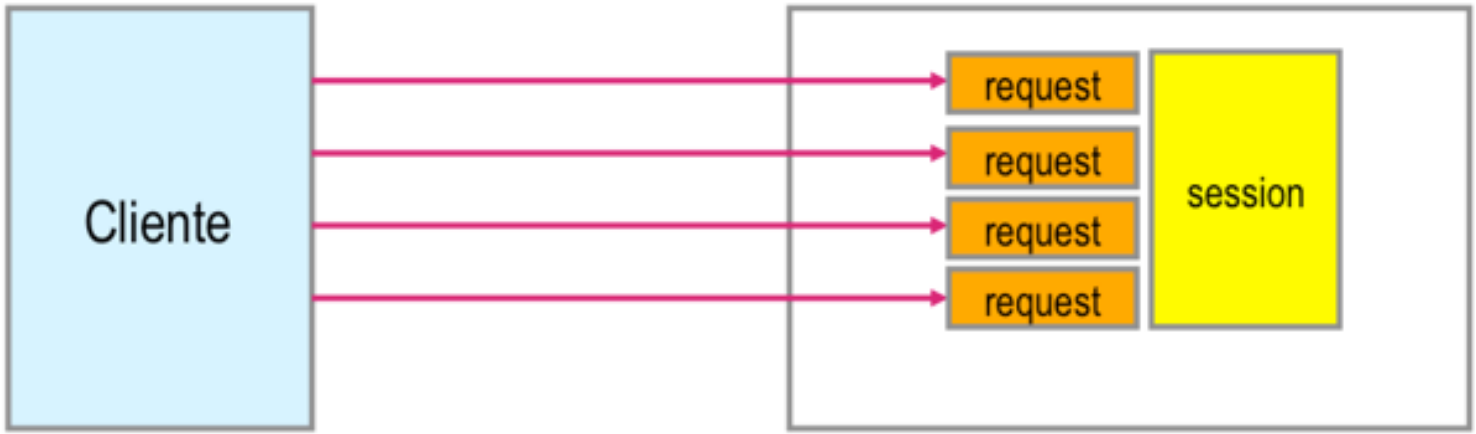
servlets & webapps

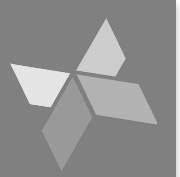
6

sessão do cliente

Sessões HTTP

- Uma sessão representa o tempo que um **cliente** acessa uma pagina ou domínio
- Como **HTTP/1 não mantém estado de sessão**, as aplicações Web precisam cuidar de mantê-lo quando necessário
- **Cookies** são o mecanismo mais usado para manter sessão em HTTP
- Uma sessão é **iniciada com uma requisição**, é única para cada cliente e persiste através de várias requisições.





HttpSession

- Sessões são representadas por objetos **HttpSession** e são obtidas a partir de uma requisição. O objeto pode ser usado para armazenar objetos que serão recuperados posteriormente pelo mesmo cliente.

- Para **criar** uma nova sessão ou para **recuperar** uma referência para uma sessão existente usa-se o mesmo método:

```
HttpSession session = request.getSession();
```

- Para saber se uma sessão é nova, use o método **isNew()**:

```
BusinessObject myObject = null;
if (session.isNew()) {
    myObject = new BusinessObject();
    session.setAttribute("obj", myObject);
}
myObject = (BusinessObject)session.getAttribute("obj");
```

Atributos de sessão

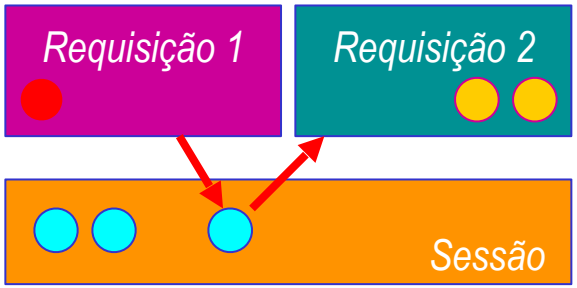
- Dois métodos de **HttpSession** permitem compartilhamento de objetos

```
void   setAttribute("nome", objeto)
Object getAttribute("nome")
```

- Por exemplo, o código abaixo grava um array em uma sessão:

REQUISIÇÃO 1

```
String[] vetor = {"um", "dois", "tres"};
HttpSession session = request.getSession();
session.setAttribute("dados", vetor);
```



- Outra requisição da mesma sessão (mesmo cliente), pode obter o array:

REQUISIÇÃO 2

```
HttpSession session = request.getSession();
String[] dados = (String[])session.getAttribute("dados");
```

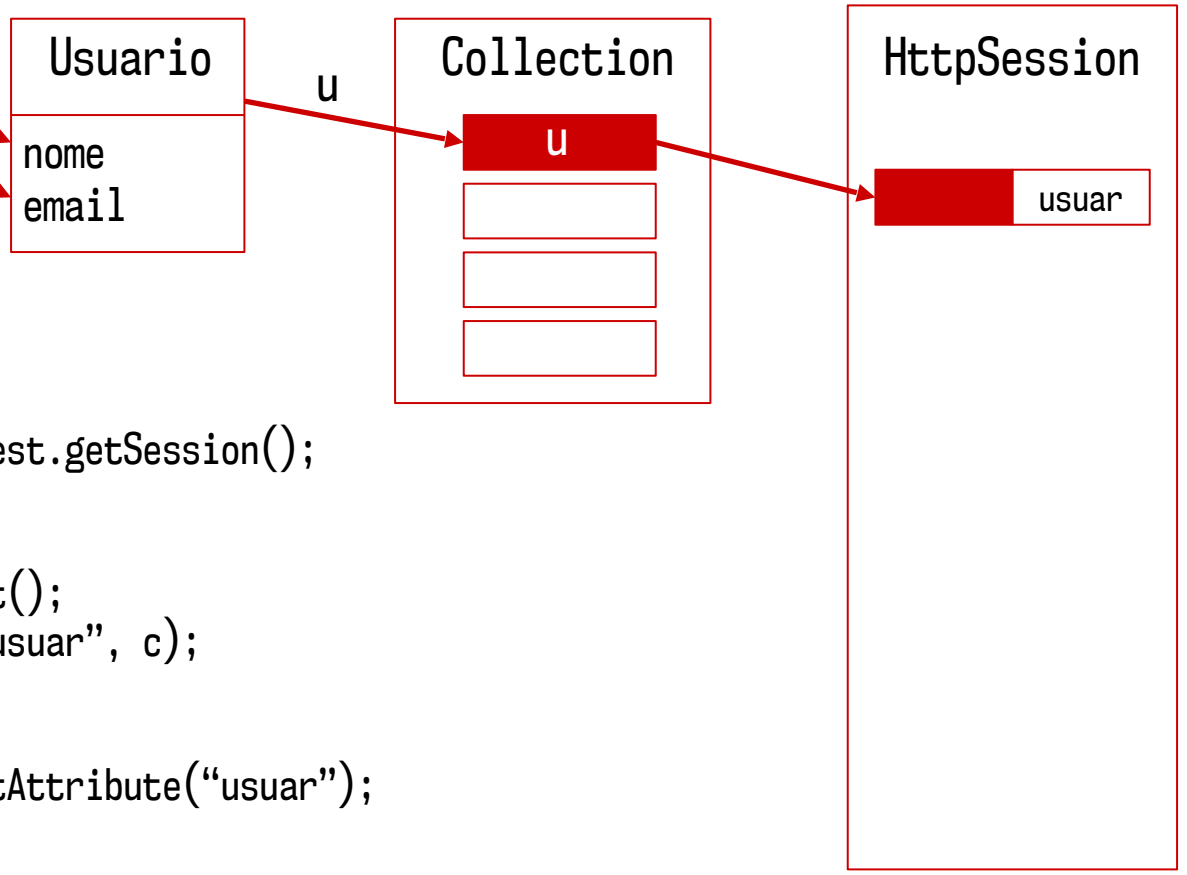
- Como a sessão pode durar várias requisições, é possível que a persistência de alguns objetos não sejam desejáveis. Eles podem ser removidos usando:

```
session.removeAttribute("nome")
```

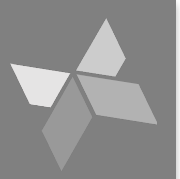
Exemplo de uso de uma sessão

- Este exemplo usa uma sessão para guardar uma coleção de objetos

```
Usuario u = new Usuario();
u.setNome(nome)
u.setEmail(email)
```




```
Collection c = null;
HttpSession s = request.getSession();

if (s.isNew()) {
    c = new ArrayList();
    s.setAttribute("usuar", c);
}

c = (Collection)s.getAttribute("usuar");
c.add(u);
```

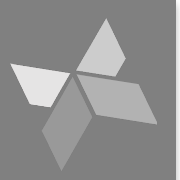


Listener para eventos em atributos

- É possível saber **quando** um atributo foi adicionado a uma sessão usando **HttpSessionAttributeListener** e **HttpSessionBindingEvent**
- Métodos a implementar do Listener
  - **attributeAdded** (HttpSessionBindingEvent e)
  - **attributeRemoved** (HttpSessionBindingEvent e)
  - **attributeReplaced** (HttpSessionBindingEvent e)
- **HttpSessionBindingEvent** possui três métodos para recuperar **sessão**, **nome** e **valor** dos atributos (valores novos)
  - String **getName**()
  - String **getValue**()
  - HttpSession **getSession**()
- É preciso registrar o listener com **@WebListener** ou no **web.xml**

Listeners para eventos do ciclo de vida

- Pode-se saber quando uma sessão foi **criada**, **invalidada** ou **expirada** usando **HttpSessionListener**:
  - Métodos sessionCreated() e sessionDestroyed()
- Para saber quando uma sessão existente foi **ativada** ou está para ser **passivada** usa-se **HttpSessionActivationListener**:
  - Métodos sessionDidActivate() e sessionWillPassivate()
- Para controlar quando objetos são **associados** a uma sessão e quando deixam a sessão (por qualquer razão) deve-se implementar um **HttpSessionBindingListener**
  - Métodos valueBound() e valueUnbound()
- Cada listener tem um evento correspondente, que é recebido em cada método (consulte a documentação para mais detalhes)



Sessão à prova de clientes

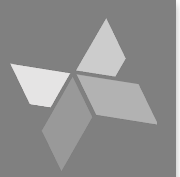
- A sessão é implementada com cookies **se o cliente suportá-los**
  - Caso não suporte, o servidor pode usar outro meio de manter a sessão
- O método **encodeURL()** de HttpServletResponse permite guardar informações de sessão na URL

```
out.print("<a href='" + response.encodeURL("Teste") + "'>");
```

- Se cliente suportar cookies, URL passa inalterada (o identificador da sessão será guardado em um cookie)
- Se cliente não suportar cookies, o identificador será passado como parâmetro da requisição

```
http://localhost:8080/servlet/Teste;jsessionid=A424JX08S99
```

- Esta solução é mais complexa, menos eficiente e menos segura que cookies



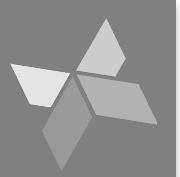
Validade e duração de uma sessão

- Há quatro métodos em HttpSession para lidar com duração de uma sessão:
  - void **setMaxInactiveInterval**(int) – define novo valor para timeout
  - int **getMaxInactiveInterval**() – recupera valor de timeout
  - long **getLastAccessedTime**() – recupera data em UTC
  - long **getCreationTime**() – recupera data em UTC
- Timeout default pode ser definido no **web.xml** para todas as sessões. Por exemplo, para que dure ate 15 minutos:

```
<session-config>
    <session-timeout>15</session-timeout>
</session-config>
```

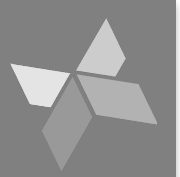
- Para destruir imediatamente uma sessão usa-se:

```
session.invalidate();
```



Cookies

- Usamos cookies **persistentes** para definir **preferências** associados ao cliente que duram mais do tempo da sessão
  - Servidor irá criar cabeçalho que irá instruir o browser a criar um arquivo guardando as informações do cookie
- Para criar cookies cookies persistentes é preciso
  - Criar um novo objeto **Cookie**
  - Definir a duração do cookie com o método **setMaxAge()**
  - Definir outros **atributos** do cookie (domínio, caminho, segurança) se necessário
  - Adicionar o cookie à **resposta**



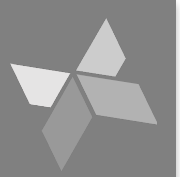
Como usar cookies

- **HttpServletResponse**: criação de um cookie "usuario" contendo valor recebido como parâmetro na requisição, e que irá expirar em 60 dias:

```
String nome = request.getParameter("nome");
Cookie cookie = new Cookie("usuario", nome);
cookie.setMaxAge(1000 * 24 * 3600 * 60); // 60 dias
response.addCookie(cookie); // adiciona o cookie na resposta
```

- **HttpServletRequest**: leitura do cookie em outra requisição:

```
String usuario = null;
for (Cookie cookie : request.getCookies()) {
   if (cookie.getName().equals("nome") {
        usuario = cookie.getValue();
        break;
   }
}
```



Compartilhamento usando objetos de escopo

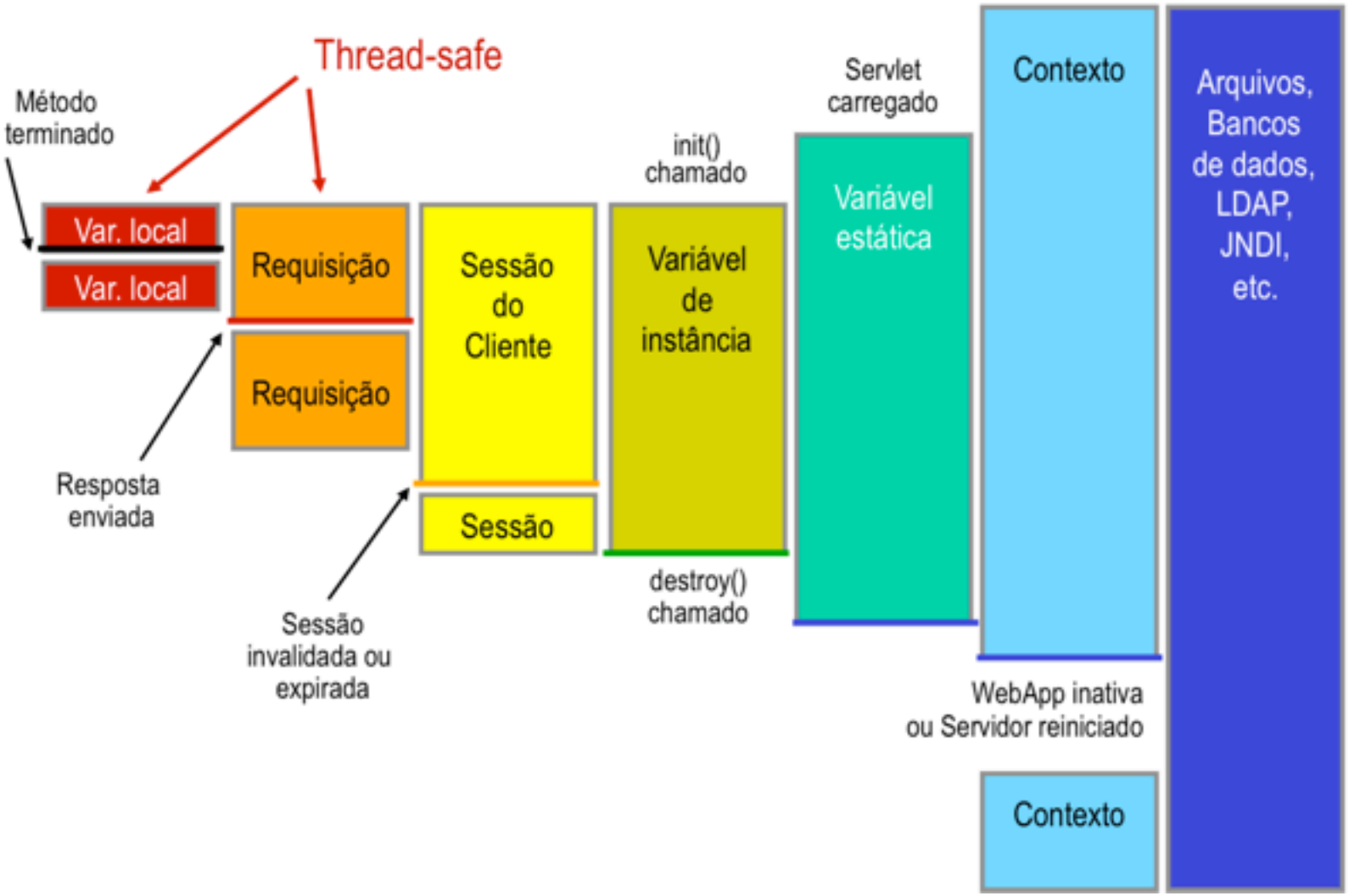
- Devido à natureza dos servlets e sua forma de execução concorrente, não é recomendado o compartilhamento usando variáveis **estáticas** e de **instancia**
- A forma recomendada é usar os métodos **get/setAttribute** dos três objetos de escopo **ServletContext**, **HttpSession** e **HttpServletRequest**
- Para **gravar** dados em um objeto de persistência na memória, deve-se usar:

```
objeto.setAttribute("nome", dados);
```

- E para **recuperar** ou remover os dados:

```
Object dados = objeto.getAttribute("nome");
objeto.removeAttribute("nome");
```

Escopos e compartilhamento

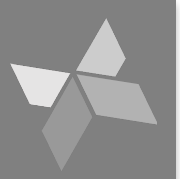


Lidando com recursos compartilhados

- Servlets podem ser acessados por muitos **threads** simultaneamente (cada cliente é um thread)
- Se operar apenas com variáveis locais ao método de serviço (recomendado) ou compartilhar apenas objetos imutáveis o servlet é **thread-safe**
- Alterações em **contextos** (e potencialmente em sessões) **podem** gerar condições de concorrência que **devem** ser sincronizadas

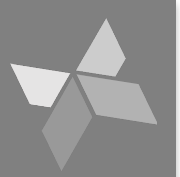
```
synchronized(this) {
    context.setAttribute("nome", objeto);
}
```

- A interface **SingleThreadModel** impede acesso simultâneo de múltiplos clientes ao servlet que a implementa. Mas é um **anti-pattern** (não use).



Exercícios

- 1. Escreva um formulário com 4 páginas:
  - O primeiro deve receber nome e email
  - O segundo deve receber um endereço
  - O terceiro deve receber 3 números
  - O quarto deve listar todos os dados digitados anteriormente
- 2. Crie um "carrinho de compras" e uma página com 4 botões para adicionar produtos. Crie uma página que liste os produtos armazenados no carrinho.



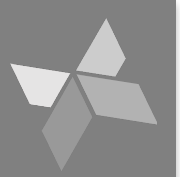
JAVAEE7

WEB

servlets & webapps

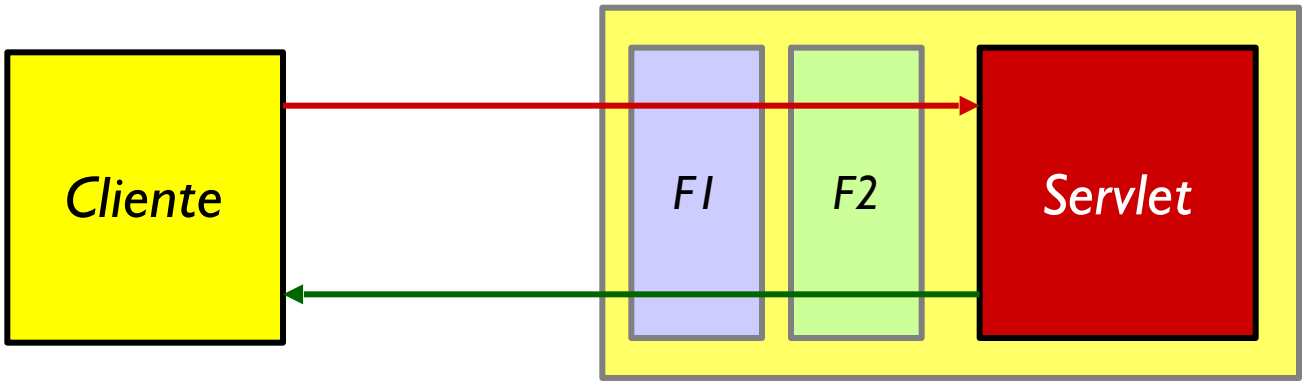
7

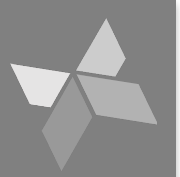
filtros interceptadores



WebFilter

- Componente Web que **intercepta** requisições e respostas no seu caminho até um servlet, e de volta ao cliente. Sua existência é ignorada por ambos.
- Criado implementando a interface **javax.servlet.Filter** e mapeado a servlets através de anotações **@WebFilter** ou via **web.xml**
- Pode ser concatenado em **corrente**: as requisições são interceptadas em uma ordem e as respostas em ordem inversa
- Aplicações típicas: autenticação, tradução, conversão de caracteres, encoding, tokenizing, conversão de imagens, compressão/descompressão e criptografia.





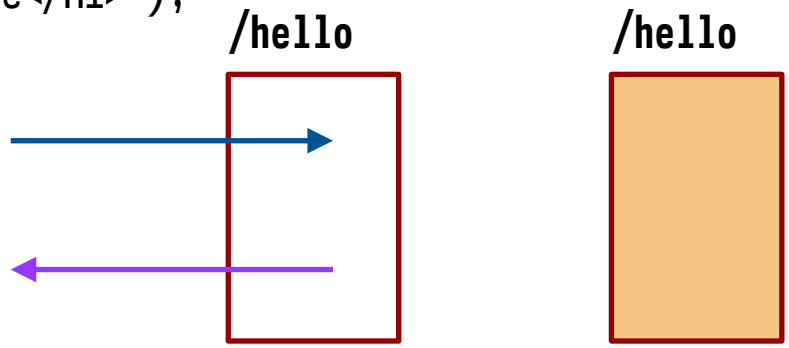
Como funcionam?

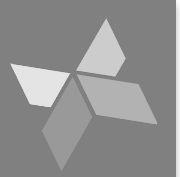
- Quando o container recebe uma requisição, ele verifica se há um filtro associado à URL solicitada. Se houver, a requisição é roteada ao filtro
- O filtro, então, pode
  - 1. Gerar sua própria resposta para o cliente (bloqueando a passagem)
  - 2. Repassar a requisição, modificada ou não, ao próximo filtro da corrente, (se houver), ou ao recurso final, se ele for o último filtro
  - 3. Rotear a requisição para outro recurso
- Na volta para o cliente, a resposta passa pelo mesmo conjunto de filtros em ordem inversa

Filtro que substitui servlet

Requisições **/hello** serão redirecionadas para este "filtro", que simplesmente retorna uma página

```
@WebFilter("/hello")
public class HelloFilter implements Filter {
  public void init(FilterConfig config) {}

  public void doFilter(ServletRequest request, ServletResponse response,
                       FilterChain filterChain)
                        throws ServletException, IOException {
     PrintWriter out = response.getWriter();
     out.println("<HTML><HEAD><TITLE>Filter Response");
     out.println("</TITLE></HEAD><BODY>");
     out.println("<H1>Filter Response</H1>");
     out.println("</BODY></HTML>");
     out.close();
   }

   public void destroy() {}
}
```





Configuração

- Pode-se mapear via anotações **@WebFilter** ou **web.xml**

```
<filter>
    <filter-name>umFiltro</filter-name>
    <filter-class>filtros.HelloFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>umFiltro</filter-name>
    <url-pattern>/hello</url-pattern>
</filter-mapping>
```

- Se filtros forem configurados via anotações (com filterName e urlPatterns), podem ser incluído em web.xml apenas para **ordenação**

```
@WebFilter(filterName="umFiltro", urlPatterns="/hello")
public class HelloFilter implements Filter {}
```

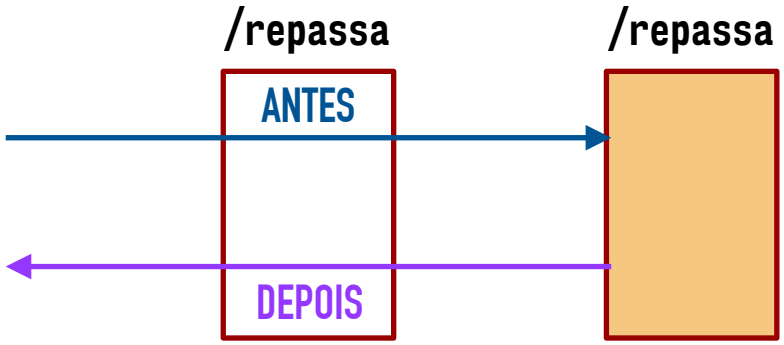
umFiltro será chamado **antes** de outroFiltro

```
<filter-mapping>
    <filter-name>umFiltro</filter-name><url-pattern/>
</filter-mapping>
<filter-mapping>
    <filter-name>outroFiltro</filter-name><url-pattern>/hello</url-pattern>
</filter-mapping>
```

Filtro que repassa requisição

Requisições **/repassa** serão redirecionadas para este filtro, que apenas registra a chegada da requisição e a resposta em log

```
@WebFilter("/repassa")
public class RepasseFilter implements Filter {
  public void init(FilterConfig config) {}

  public void doFilter(ServletRequest request, ServletResponse response,
                       FilterChain filterChain)
                        throws ServletException, IOException {
     getServletContext().log("Antes da requisição");
     filterChain.doFilter(request, response);
     getServletContext().log("Depois da resposta");
   }

   public void destroy() {}
}
```

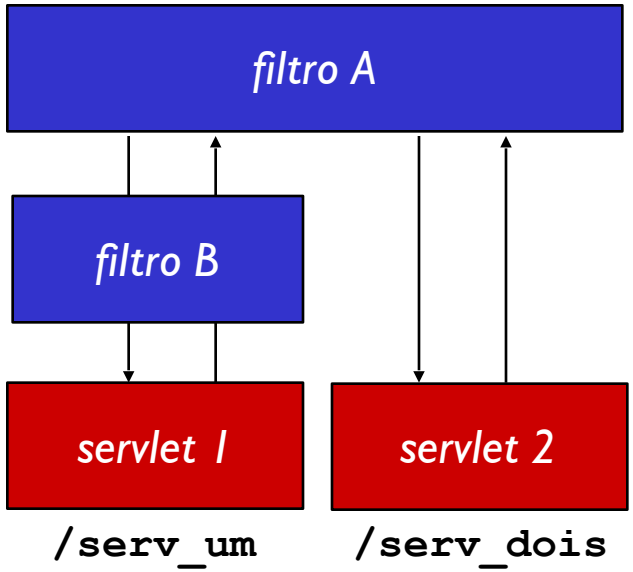


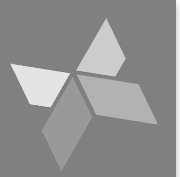
Configuração da corrente

- A corrente pode ser configurada com definição das instâncias de filtros e mapeamentos em ordem

```
@WebFilter("filtroA")
public class FilterA implements Filter { ... }
```

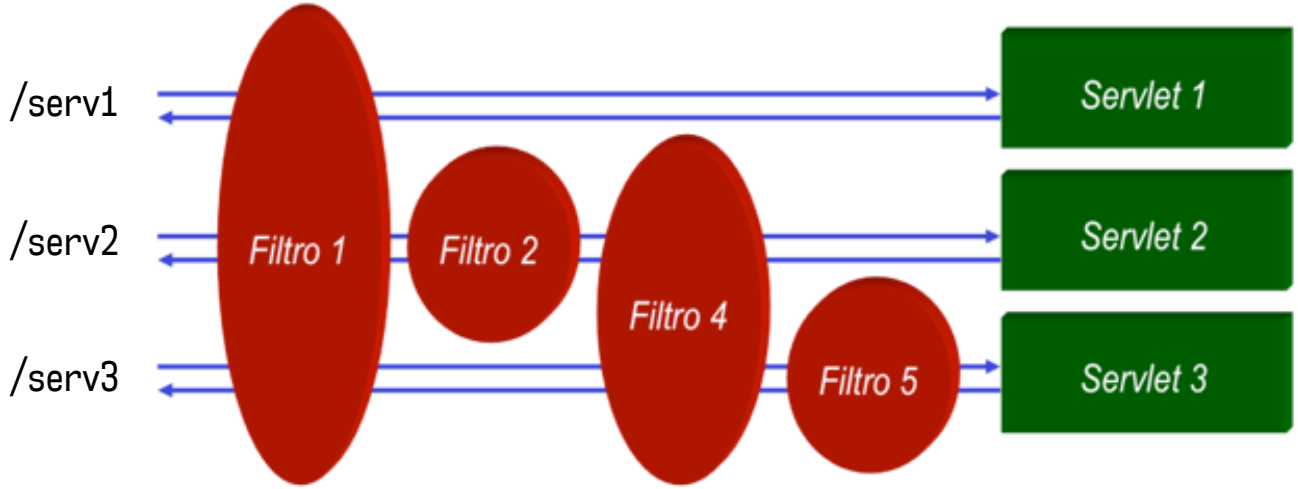
```
<filter>
    <filter-name>filtroB</filter-name>
    <filter-class>j550.filtros.FilterB</filter-class>
</filter>
<filter-mapping>
    <filter-name>filtroA</filter-name>
    <url-pattern>/serv_um</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>filtroA</filter-name>
    <servlet-name>/serv_dois</servlet-name>
</filter-mapping>
<filter-mapping>
    <filter-name>filtroB</filter-name>
    <servlet-name>serv_um</servlet-name>
</filter-mapping>
```





Exemplo: múltiplos filtros em cascata

- Caminho seguido pelas requisições
  - **Filtro1**: /serv1, /serv2, /serv3, **Filtro2**: /serv2, **Filtro4**: /serv2, /serv3, **Filtro5**: /serv3



  - **Requisição** para **/serv1**: passa por Filtro 1, depois Servlet 1; **Resposta** passa por Filtro 1
  - **Requisição** para **/serv2**: passa por Filtro 1, depois Filtro 2, depois Filtro 4, depois Servlet 2
    Resposta passa por Filtro 4, depois Filtro 2, depois Filtro1
  - **Requisição** para **/serv3**: passa por Filtro 1, depois Filtro 4, depois Filtro 5, depois Servlet 3
    Resposta passa por Filtro 5, depois Filtro 4, depois Filtro1

Wrappers de requisições/respostas

- Permitem que filtros **transformem** requisições ou respostas
- Uma subclasse de um Wrapper pode ser passada em uma corrente de filtros **em substituição** à requisição ou resposta original
- Métodos como getParameter() e getHeader() podem ser sobrepostos para alterar parâmetros e cabeçalhos
- **HttpServletRequestWrapper** e **HttpServletResponseWrapper**: implementam todos os métodos das interfaces correspondentes, facilitando a sobreposição para alteração de cabeçalhos, etc.
- Antes da chamada ao **doFilter()**, o filtro pode substituir os objetos ServletRequest e ServletResponse ao passá-los adiante

```
ServletRequest newReq = new ModifiedRequest(...);
chain.doFilter(newReq, res);
```

- Na volta, pode alterar a resposta antes de devolver para o cliente

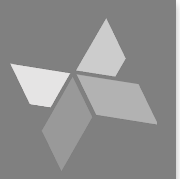
Wrappers

- Sobrepondo um **HttpServletRequest**

```
public class MyServletWrapper extends HttpServletRequestWrapper {
   public MyServletWrapper(HttpServletRequest req) {
      super(req);
   }
   public String getParameter(String name) {
      return super.getParameter(name).toUpperCase();
   }
}
```

- Usando wrappers dentro de **doFilter(request, response, chain)**

```
HttpServletRequest req = (HttpServletRequest)request;
HttpServletResponse res = (HttpServletResponse)response;
HttpServletRequest fakeReq = new MyServletWrapper(req);
HttpServletResponse fakeRes = new TestResponse(res);

chain.doFilter(fakeReq, fakeRes);
```



Filtros que tomam decisões

- Um filtro pode ler a requisição e tomar decisões como transformá-la, passá-la adiante ou retorná-la

```
public void doFilter(.. request, ... response, ... chain) {
   String param = request.getParameter("saudacao");
   if (param == null) {
      PrintWriter out = response.getWriter();
      response.setContentType("text/html");
      out.println("<h1>Erro!</h1>");
   } else if (param.equals("Bonjour!")) {
      class MyWrapper extends ServletRequestWrapper { // ...
        public String getParameter(String name) {
          if (name.equals("saudacao")) {
             return "Bom Dia";
          }
        }
      }
      ServletRequest myRequest = new MyWrapper(request);
      chain.doFilter(myRequest, response);
   } else { chain.doFilter(request, response); }
}
```

Se parâmetro for null, retorna página HTML com mensagem

Se parâmetro lido for "Bonjour", criar um request falso contendo outro valor

Substituir request que será passado à etapa seguinte

Filtros e repasse de requisições

- Filtros são normalmente **ignorados** durante **repasse de requisições** (**RequestDispatcher** ou **páginas de erro**): o recurso é acessado diretamente sem filtragem

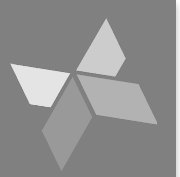
- A política de redirecionamento pode ser configurada para aceitar essas condições usando o tag **<dispatcher>** em web.xml:

```
<filter-mapping>
    ...
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

  - Um mapeamento pode ter zero ou mais dispatchers. Os valores podem ser **INCLUDE**, **FORWARD**, **ERROR**, **REQUEST**. Default é **REQUEST**

Exercícios

- 1. Escreva um filtro para interceptar o servlet que imprime quadrados para que o servlet só receba o número se ele for menor que 10
- 2. Altere o filtro do exercício anterior para que ele permita números até 100, mas divida por 2 se for maior que 10
- 3. Escreva um filtro para interceptar o servlet que imprime a data e hora, para que a resposta imprima apenas a data
- 4. Escreva um filtro que transforme os parâmetros "nome" em caixa alta.
- 5. Escreva um filtro que transforme os parâmetros "nome" concatenando um "" antes e um "" depois. Escreva outro filtro que faça o mesmo mas com um tag "" e "". Configure para que o filtro do exercício anterior e esses três filtros sejam chamados em cascata (verifique o resultado e o efeito da ordenação)



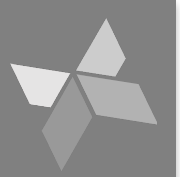
JAVAEE7

WEB

servlets & webapps

8

arquitetura mvc



Arquitetura MVC

- Aplicações Web devem ser construídas em **camadas**, separando **responsabilidades**
  - Uma camada de apresentação (**V**iew): para exibir informações e controles para interação com o usuário
  - Uma camada de controle (**C**ontroller), para capturar eventos, selecionar informações a exibir e delegar tarefas para componentes de negócio
  - Uma camada de dados (**M**odel), contendo representações das informações exibidas e manipuladas pela aplicação
- Frameworks como JavaServer Faces (JSF) impõem uma estrutura própria e abstraem os detalhes dessa arquitetura (isto será abordado em outro tutorial)
- Servlets, páginas HTML, JSP ou XML e objetos Java comuns podem ser usados para criar aplicações Web em Java com separação de responsabilidades

Views

- Um **View** é qualquer componente da aplicação que assume a responsabilidade de proporcionar uma interface com o usuário
- Pode ser um servlet que gera HTML e JavaScript, uma página HTML dinâmica com JQuery+Bootstrap), mas geralmente é um **template**
- A especificação Java EE define dois tipos de templates de página: **JavaServer Pages** (JSP) e **Facelets** (XHTML)
  - **JSP** gera servlets automaticamente mapeados ao nome do arquivo e permite incluir variáveis dinâmicas em scriptlets, tags ou expressões
  - **Facelets** são escritos em XHTML e usam tags e expressões para comunicação com objetos Java; são usados dentro da arquitetura do JSF

JavaServer Pages (JSP)

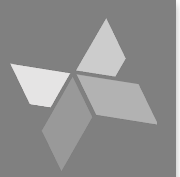
- **JavaServer Pages** são desenvolvidos como páginas Web comuns e **transformadas em servlets** quando instaladas
- O acesso ao servlet é feito pelo nome do arquivo JSP (**o servlet é mapeado ao nome do arquivo**)
- Uma página JSP simples pode ser criada a partir de uma página HTML simplesmente mudando a extensão **.html** para **.jsp**
- Páginas JSP podem conter estruturas que **geram HTML dinâmico**, como **scritplets**

/data.jsp

```
<body>
<p>A data de hoje é
   <%=new Date() %>.</p>
<body>
```

http://localhost:8080/contexto/data.jsp

A data de hoje é Thu Aug 22 2009 11:51:33 GMT-0300



Scriptlets (anti-pattern)

- **Scriptlets** são trechos de código Java (no contexto do método service) incluídos em uma página JSP em blocos delimitados por **<% ... %>**:

```
<p>Texto repetido:
<% for (int i = 0; i < 10; i++) { %>
   <p>Esta é a linha <%=i %>
<% } %>
```

- Dentro dos scriptlets pode-se ter acesso a objetos implícitos do servlet (request, response, etc.) e programar a geração dinâmica da página

- Mas scriptlets misturam responsabilidades e por esse motivo são hoje considerados um **anti-pattern**. É possível substitui-los por **taglibs** e Expression Language (**EL**)

Taglibs

- **Bibliotecas de tags** XML vinculadas a objetos que interagem com os objetos implícitos de um servlet (request, response, beans, HTML, etc.)
- Criadas implementando interfaces de **javax.servlet.jsp** configuradas em XML (**TLD** - Tag Library Descriptor) que associa **tags** a um **namespace**
- O namespace é **declarado** nas páginas de templates (JSP ou XML) que usam a biblioteca, e associado a um **prefixo** usado nos tags

```
<%@taglib uri="http://abc.com/ex" prefix="exemplo"%>
```

ou

```
<html xmlns="exemplo:http://abc.com/ex"> ... </html>
```

Sintaxe JSP usando diretiva @taglib

ou

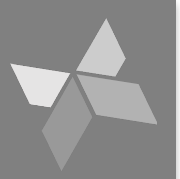
Sintaxe XML usando namespaces

/data.xhtml

```
<body>
<p>A data de hoje é
   <exemplo:dataHoje>.</p>
<body>
```

http://localhost:8080/contexto/data.jsp

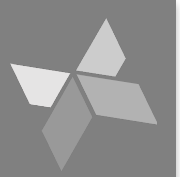
A data de hoje é Thu Aug 22 2009 11:51:33 GMT-0300



JSTL (JSP Standard Tag Library)

- **JSTL** é um conjunto de **taglibs padrão** que fornece controles básicos como blocos condicionais, repetições, transformações, etc.
- O uso de JSTL (junto com **EL**) estimula a separação de apresentação e lógica promovendo a arquitetura **MVC** e são a forma recomendada de usar JSP
- Para usar os tags é preciso **declarar o taglib** desejado em cada página (usando a sintaxe JSP ou a sintaxe XML). Exemplo com sintaxe JSP:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <body>
  <c:if test='${not empty param.email}'>
    <p>E-mail: <c:set value="${param.email}"/></p>
  </c:if>
  </body>
</html>
```



EL (Expression Language)

- **Expression Language** (**EL**) e uma linguagem declarativa criada para viabilizar a comunicação entre **views** (páginas JSP e XHTML) e **controllers** (beans e outros objetos Java) em aplicações Web que usam arquitetura MVC

- Expressões são incluídas na página dentro de delimitadores **${...}** ou **#{...}**

- Permite acesso a objetos implícitos, beans e seus atributos, fazendo conversão automática de tipos, com suporte a valores default

- Em vez de usar os scriptlets

```
<% request.getAttribute("nome")      %>
<% bean.getPessoa().getNome() %>
```

- Use as expressões

```
${nome}
${bean.pessoa.nome}
```

EXEMPLOS

```
${pageContext.request.contextPath }
${contato.id + 1 }
${not empty biblioteca.livros}
${livro.preco * pedido.quantidade}
${livro.autor}
${param.turno}
${paramValues.turno[0]}
${cookie['nome']}
```

Facelets

- Tags mapeados a componentes usados em Views do JSF2

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
     <title>Introducao a JSF</title>
    </h:head>
    <h:body>
      <p>#{introBean.mensagem}</p>
    </h:body>
</html>
```

Mapeamento XML do namespace da biblioteca padrão XHTML a tags sem prefixo (tags sem prefixo são XHTML)

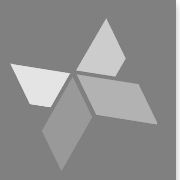
```
@Named
public class IntroBean {
  public String getMensagem() {
     return "Hello!";
  }
}
```

Tags XHTML

Expressão EL
Id (nome) do bean derivado do nome da classe

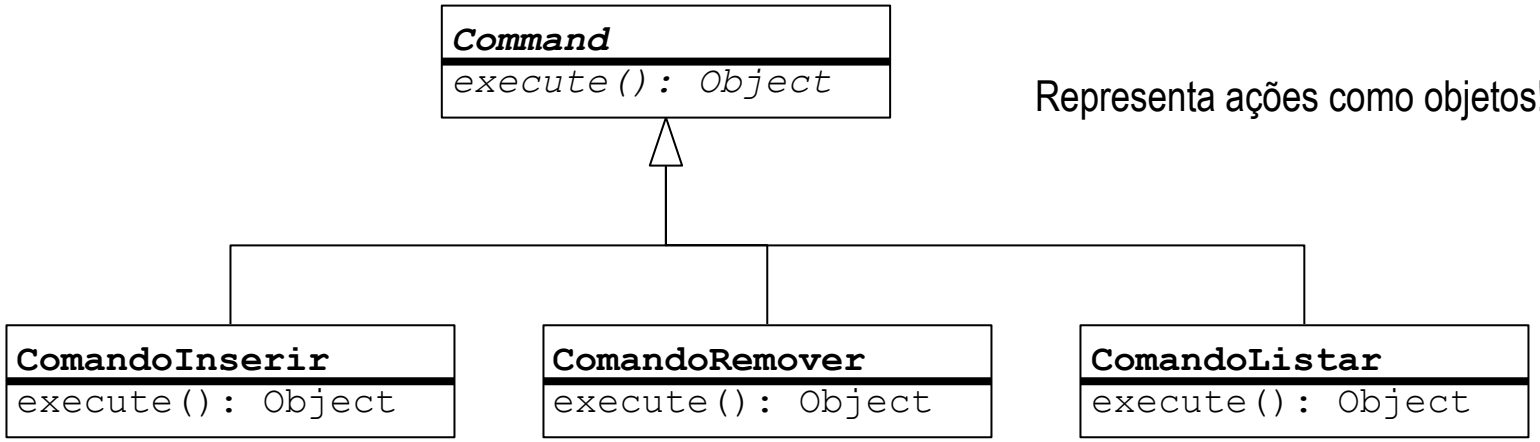
Facelets: tags mapeados a componentes HTML nativos do JSF

Mapeamento XML de namespace de biblioteca de tags JSF HTML a prefixo padrão (tags com prefixo "h" fazem parte desta biblioteca)

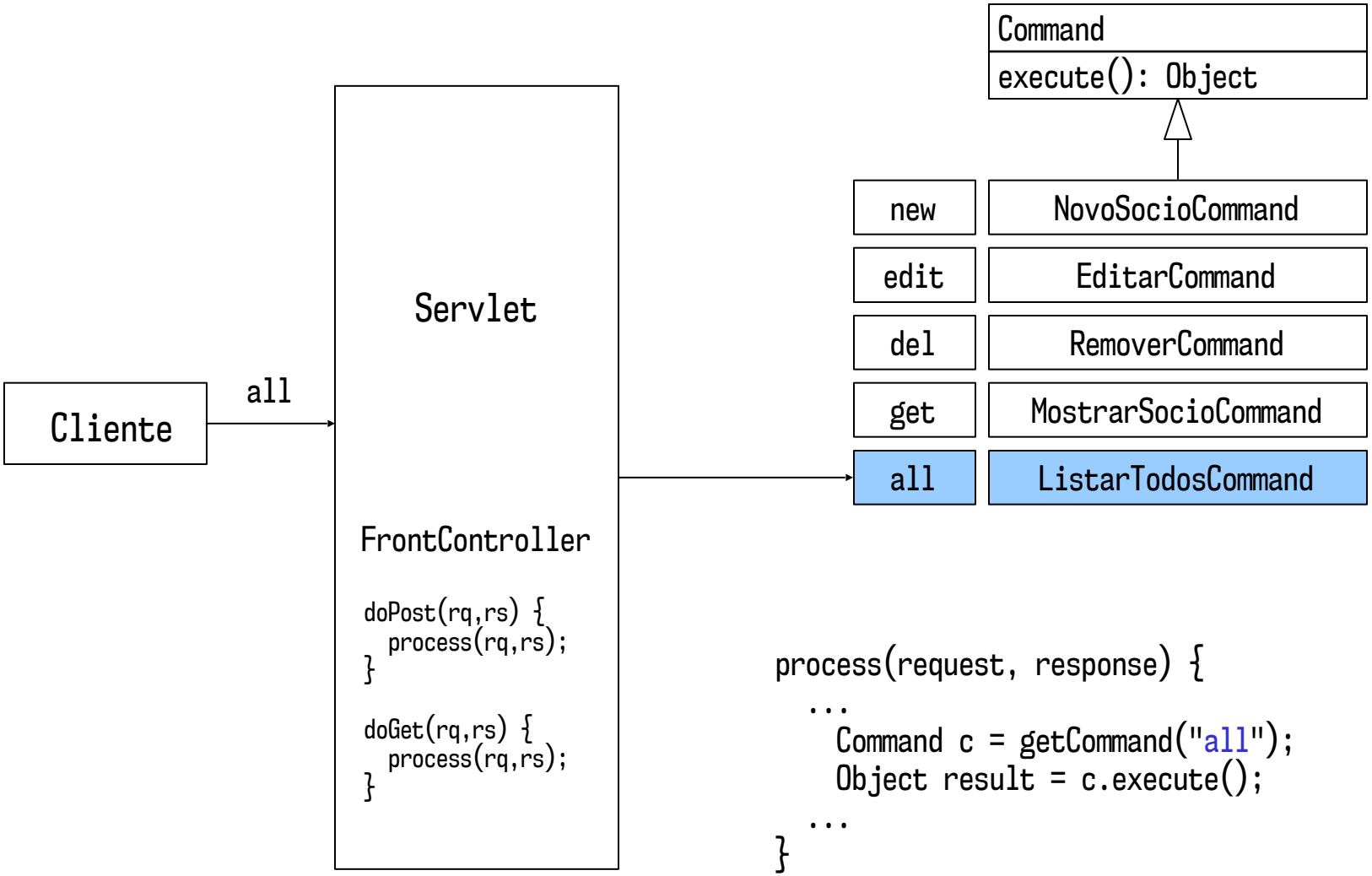


Controllers

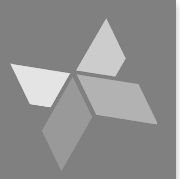
- **Controladores** são componentes que redirecionam para views, processam dados para exibição, recebem dados para transformação, processam ou delegam eventos, delegam chamadas para ler e alterar dados na aplicação
- Um **servlet** pode ser um controlador
- Uma arquitetura comum é **Front Controller**, onde um único servlet intercepta todas as requisições e delega para executores individuais, ou **Actions** (padrão Command)



Front Controller Pattern

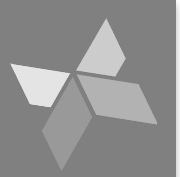



```
process(request, response) {
  ...
    Command c = getCommand("all");
    Object result = c.execute();
  ...
}
```



Models

- Modelos são **representações** dos dados usados pela aplicação
- Tipicamente a aplicação interage com dados na memória, em um banco de dados relacional (via DAO ou ORM), NoSQL, etc.
- Podem ser objetos recebidos de outras camadas da aplicação (ex: **entidades** ou value objects recebidos de camada EJB, representações XML/JSON, RESTful resources, etc.)
- Controladores podem fazer cópias de dados para uso nos views (objetos de contexto, backing beans, etc.)

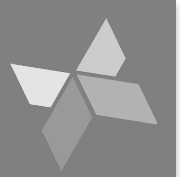


Frameworks para aplicações Web

- Há vários frameworks que facilitam a construção de aplicações Web em Java. Entre os mais populares estão:
  - **JavaServer Faces** é o principal framework do Java EE 7 e baseia-se em uma arquitetura de componentes com rigorosa separação de responsabilidades;
  - **Spring MVC** é uma arquitetura MVC baseada em servlets; é mais flexível que JSF e facilita a integração com front-ends em HTML5, Bootstrap, Angular
  - **Google Web Toolkit** (GWT) é outra arquitetura baseada em servlets que gera interfaces do usuário em HTML e JavaScript. Usada no GMail e Google Docs
- Outros frameworks incluem: Struts (já foi o mais popular), Wicket, Grails (baseado em Groovy), VRaptor, Play
- A maior parte desses frameworks utiliza estruturas padrão como templates JSP ou XML, taglibs, JSTL e EL, vistos neste capítulo

Introdução a JSF ou Spring MVC

- Esta seção consiste de uma demonstração prática que será realizada de forma interativa no ambiente local usando JSF ou Spring MVC
- Veja **http://www.argonavis.com.br/download/jsf2minicurso.html** para uma introdução a JSF 2.2
- Uma referência de Spring MVC pode ser encontrada em **https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html**



JAVAEE7

WEB

servlets & webapps

integração com bancos de dados

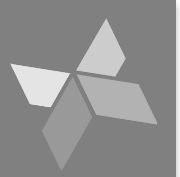
Acesso a BDs em Aplicações Web

- Servlets são aplicações Java e, como qualquer outra aplicação Java, podem usar **JDBC** e integrar-se com um banco de dados relacional
- Pode-se usar o **java.sql.DriverManager** e obter a conexão da forma tradicional

```
Class.forName("nome.do.Driver");
Connection con = DriverManager.getConnection("url", "nm", "ps");
```

- Em servidores J2EE, pode-se obter as conexões de um pool de conexões nativo através de **javax.sql.DataSource** via JNDI

```
DataSource ds = (DataSource)ctx.lookup("java:/comp/env/jbdc/Banco");
Connection con = ds.getConnection();
```



Servlet que faz SELECT em Banco

- Uma vez obtida a conexão, pode-se usar o resto da API JDBC para realizar a comunicação com o banco

```
import java.sql.*; // ... outros pacotes
public class JDBCServlet extends HttpServlet {
   String jdbcURL; String driverClass;
   String nome = ""; String senha = ""; Connection con;
   public void init() {
     try {
        Class.forName(driverClass);
        con = DriverManager.getConnection(jdbcURL, nome, senha);
     } catch (Exception e) { throw new UnavailableException(e); }
   }
   public void doGet(... request, ... response) ... {
    try {
      // ... codigo de inicializacao do response
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT * FROM nomes");
      out.println("<table border>");
      while(rs.next()) {
        out.println("<tr><td>"+rs.getString("nome")+"</tr></td>");
      }
      // ... resto do codigo
     } catch (SQLException e) {throw new ServletException(e);}
  }
}
```

Servlet que usa DAO

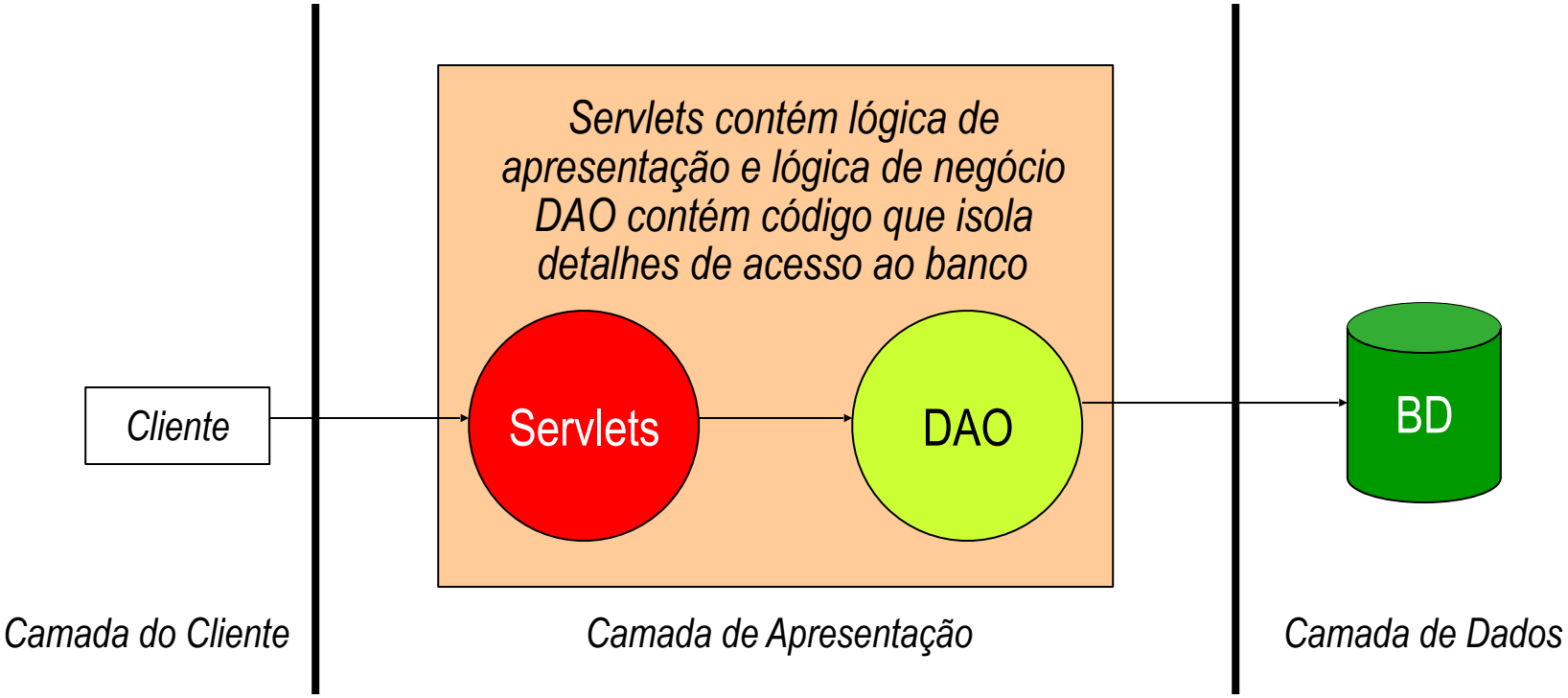
- Mas misturar código de servlet com banco de dados não é uma boa idéia. O melhor é usar um **Data Access Object** (**DAO**)

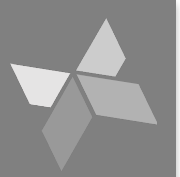
```
public class DataAccessServlet extends HttpServlet {
  DataAccessDAO dao;
  public void init() {
    dao = JDBCDataAccessDAO.getInstance();
  }
  public void doGet(... request, ... response) ... {
    try {
      // ... codigo de inicializacao do response
      String[] nomes = dao.listarTodosOsNomes();
      out.println("<table border>");
      for(int i = 0; i < nomes.length; i++) {
        out.println("<tr><td>"+nomes[i]+"</tr></td>");
      }
      // ... resto do codigo
    } catch (AcessoException e) {
      throw new ServletException(e);
    }
  }
}
```

Método da API do DAO

O que é um DAO?

- DAO oferece uma API para acesso ao banco e permite que o servlet não se preocupe com a forma de persistência que está sendo usada (facilita o reuso e manutenção)

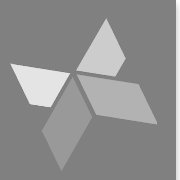




Exemplo de DAO

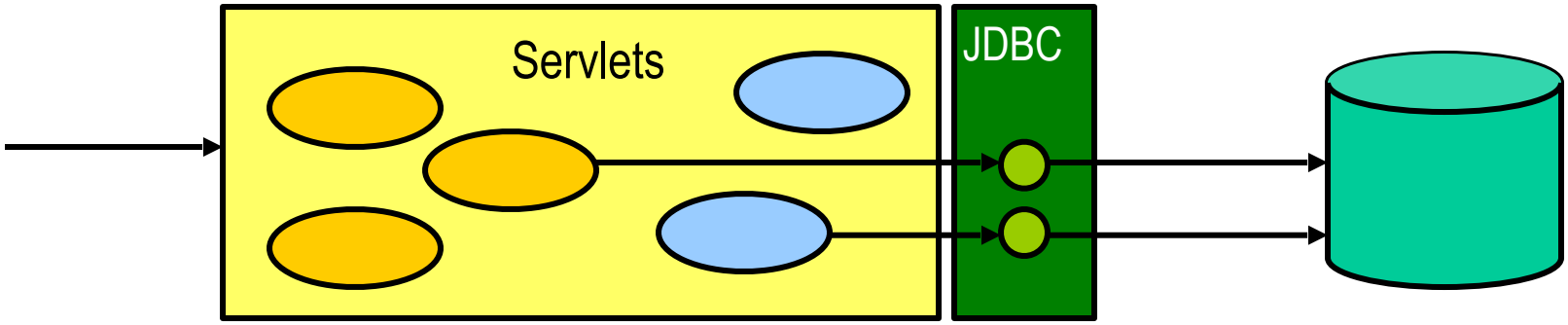
- O DAO **isola** o servlet da **camada de dados**, deixando-o à vontade para mudar a implementação

```
java.sql.*; // ... outros pacotes
public class JDBCDataAccessDAO implements DataAccessDAO {
   // ...
   public void JDBCDataAccessDAO() {
      try {
         Class.forName(driverClass);
         con = DriverManager.getConnection(jdbcURL, nome, senha);
      } catch (Exception e) { ... }
   }
   public String[] listarTodosOsNomes() throws AcessoException {
      try {
         Statement stmt = con.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT * FROM nomes");
         List nomesList = new ArrayList();
         while(rs.next()) {
            nomesList.add(rs.getString("nome"));
         }
         return (String[]) nomesList.toArray(new String[nomesList.size()];
      } catch (SQLException e) { throw new AcessoException(e); }
   }
}
```

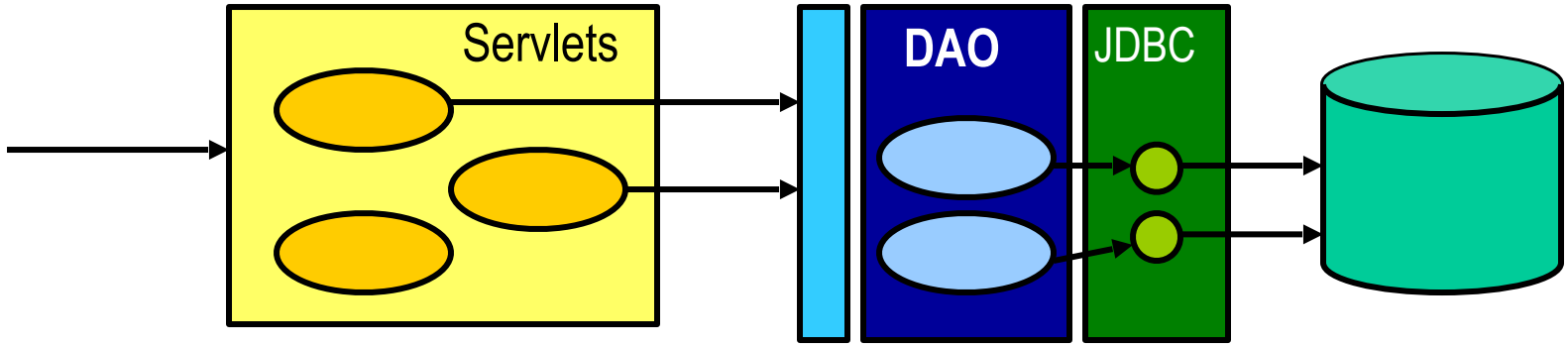


Arquitetura: com DAO ou sem DAO

- Sem DAO: código JSP e servlet misturado com código JDBC

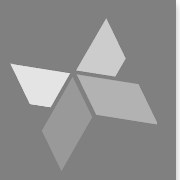


- Com DAO: camadas são independentes!



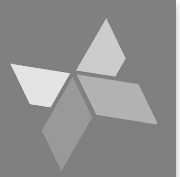
Outros bancos

- O padrão DAO pode ser usado para isolar qualquer tipo de camada de dados do servlet
  - Bancos de dados diferentes usando JDBC
  - Soluções ORM diferentes
  - Bancos de dados NoSQL
- Em alguns casos DAOs podem ser gerados através de scripts (ex: soluções de ORM com JPA)



Acesso via ORM

- Pode-se abstrair a camada SQL mais ainda usando uma solução **ORM - Object Relational Mapping,** que permite interagir com objetos, e não tabelas
- **JPA - Java Persistence API** é a interface padrão do Java EE para implementações ORM
- Usando JPA, pode-se configurar objetos persistentes (**entidades**) mapeados a registros de um banco de dados, e a aplicação Web pode trabalhar apenas com os objetos, ignorando os detalhes do banco
- Um servlet ou controlador precisa usar uma **API de persistência** (como **EntityManager** do JPA) para persistir e recuperar objetos, mas a implementação é totalmente transparente



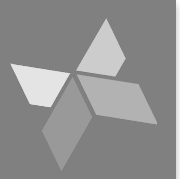
Aplicação Web + JPA

- Uma aplicação JPA precisa primeiro configurar o **mapeamento** classe-esquema
- Uma vez configurado, um servlet poderá usar o **EntityManager** (injetável via CDI) para interagir com objetos persistentes

```
public class ListarProdutoServlet extends HttpServlet {
    @Inject EntityManager em;

    public void doPost(...) {
        em.getTransaction().begin();

        List<Produto> produtos = em.createQuery("select p from Produto p");
        out.println("<p>"+produtos.size() + " produtos encontrados!</p>");
        out.println("<table border>");
        for (Produto p: produtos) {
            out.println("<tr><td>"+p.getNome()+"</td><td>$"
                                    +p.getPreco()+"</td></tr>");
        }
        out.println("</table>");
        em.getTransaction().commit();
    }
}
```

Exercícios

- 1. Configure o banco de dados no seu servidor e execute o script CreateTables (é uma aplicação Java standalone) que irá criar tabelas para os próximos exercícios
- 2. Escreva um servlet que liste os filmes disponíveis na tabela Filme (imdb, titulo, diretor, duração, ano)
- 3. Escreva um servlet e um formulário que permita inserir um novo filme no banco
- 4. Escreva um servlet que receba como parâmetro o imdb de um filme e remova do banco



JAVAEE7

WEB

servlets & webapps

10

segurança

Autenticação e autorização

- **Autenticação** é o processo de verificação de uma identidade, através da avaliação de credenciais (ex: senha)
  - Java EE 7 não oferece uma solução completa: é preciso construir as bases de usuário/senha no servidor usando ferramentas proprietárias
  - Java EE oferece mecanismos para **selecionar** e **configurar** a obtenção de informações de usuário e credenciais (**login**) via HTTP
- **Autorização** é o controle de acesso a recursos protegidos
  - Java EE oferece uma **API declarativa** para restringir acesso através de permissões (roles) filtrados por URLs e métodos HTTP
  - Uma **API programática** também permite restringir com base no usuário logado

Autenticação e autorização

Browser

Requisição 1: Pede página restrita

GET /app/faces/biblioteca.xhtml

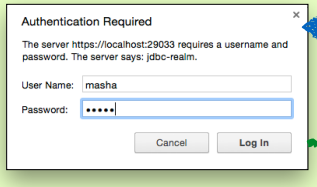
HTTP

Resposta 1: Pede credenciais

401 Unauthorized

WWW-Authenticate: Basic realm="jdbc-realm"

HTTPS



HTTPS

Requisição 2: Envia credenciais

GET /app/faces/biblioteca.xhtml

Authorization: Basic bWFzaGE6MTIzNDU=

HTTPS

Resposta 2: Envia página restrita

200 OK

Web Container

Configuração (web.xml)
- HTTPS 2-way (CONFIDENTIAL)
- Autenticação: jdbc-realm
- Métodos: GET, POST
- Roles: todos os logados (**)

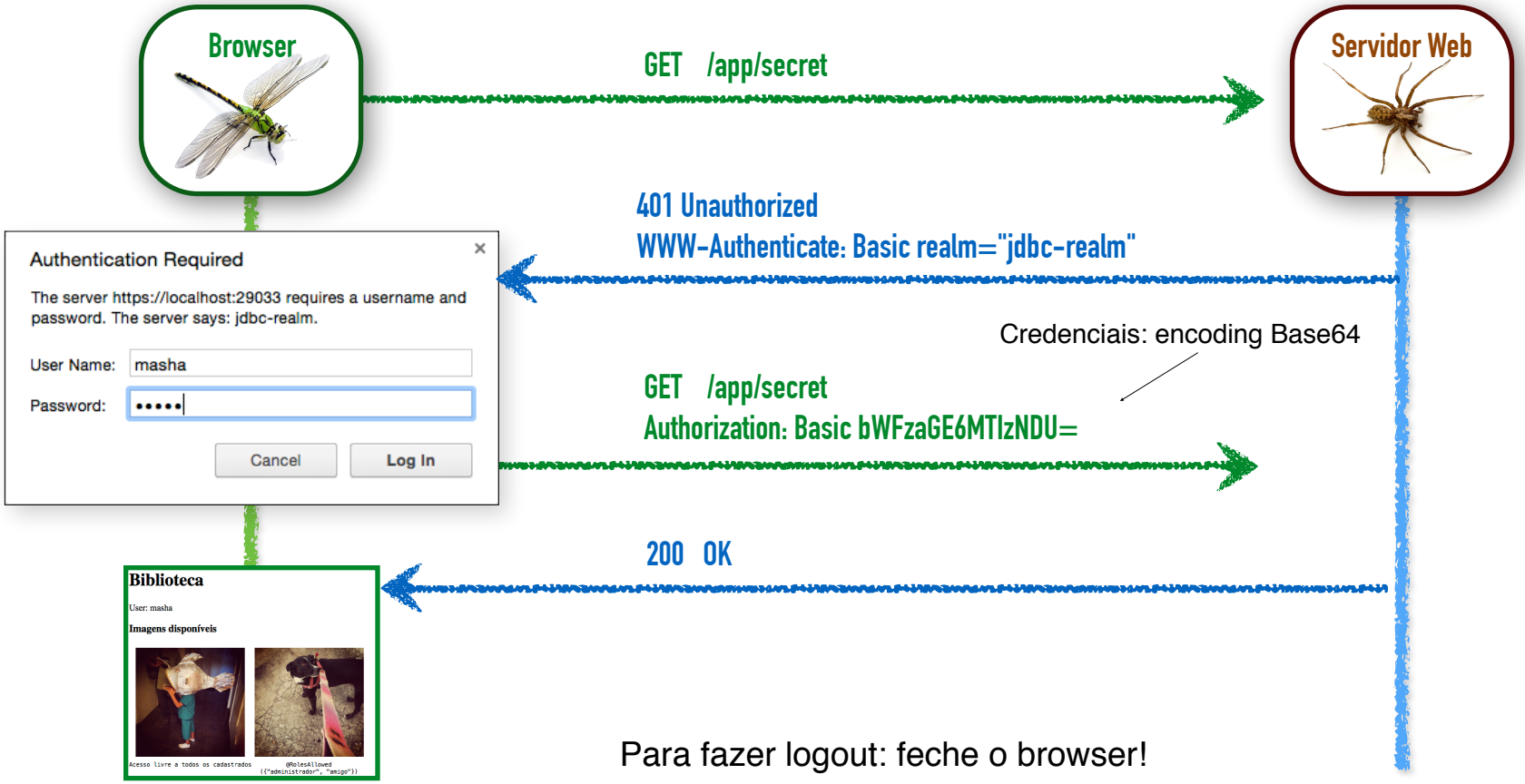
Autenticação
- Realizar autenticação
- Autenticado? Cria credencial.
  - Ir para Autorização
- Não autenticado?
  - Retorna 401 Unauthorized

Autorização
- Verificar permissões
- Autorizado?
  - Retorna recurso pedido
- Não autorizado?
  - Retorna 403 Forbidden

BASIC (RFC 2069 / 2617)

```
<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>jdbc-realm</realm-name>
</login-config>
```

configuração
web.xml

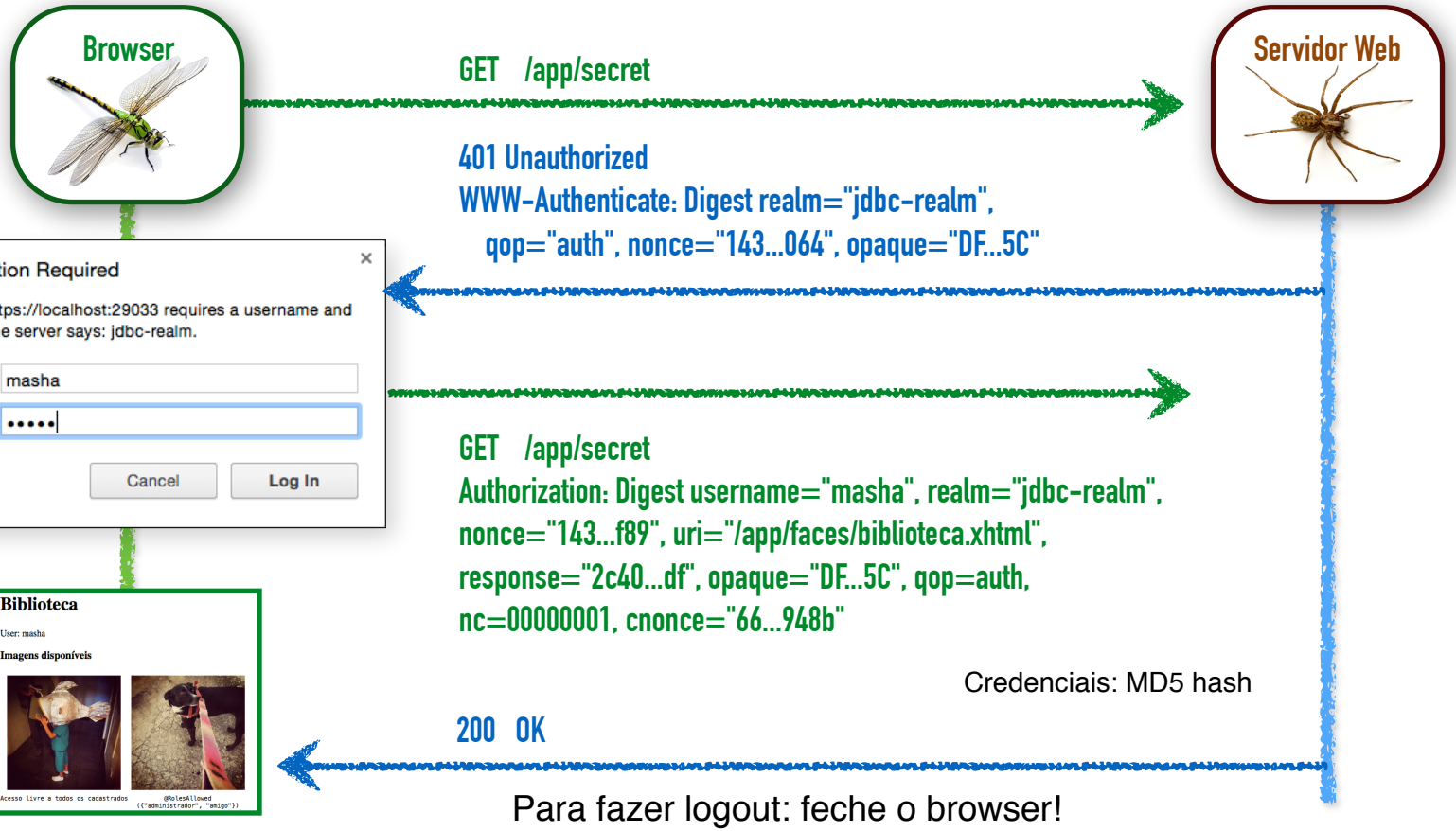


Para fazer logout: feche o browser!

DIGEST (RFC 2069 / 2617)

```
<login-config>
   <auth-method>DIGEST</auth-method>
   <realm-name>jdbc-realm</realm-name>
</login-config>
```

configuração
web.xml



Para fazer logout: feche o browser!

FORM

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/form.html</form-login-page>
    <form-error-page>/erro.html</form-error-page>
  </form-login-config>
</login-config>
```

configuração **web.xml**

GET /app/secret

200 OK

Set-Cookie: JSESSIONID=aac...44; Path=/app; Secure; HttpOnly

```
<form action="j_security_check"
      method="POST">
 <input type="text" name="j_username">
 <input type="password" name="j_password">
  <input type="submit">
</form>
```

Login:

User name: masha

Password: •••••

Submit Reset

POST https://localhost:29033/app/faces/j_security_check

Referer: https://localhost:29033/app/faces/biblioteca.xhtml

Cookie: JSESSIONID=aaab5...b1f6

Authorization:Basic Y2VyZWJyb2xoMjM0NQ==

Encoding Base64

j_username:masha

j_password:12345

Proteção depende da camada SSL/TLS

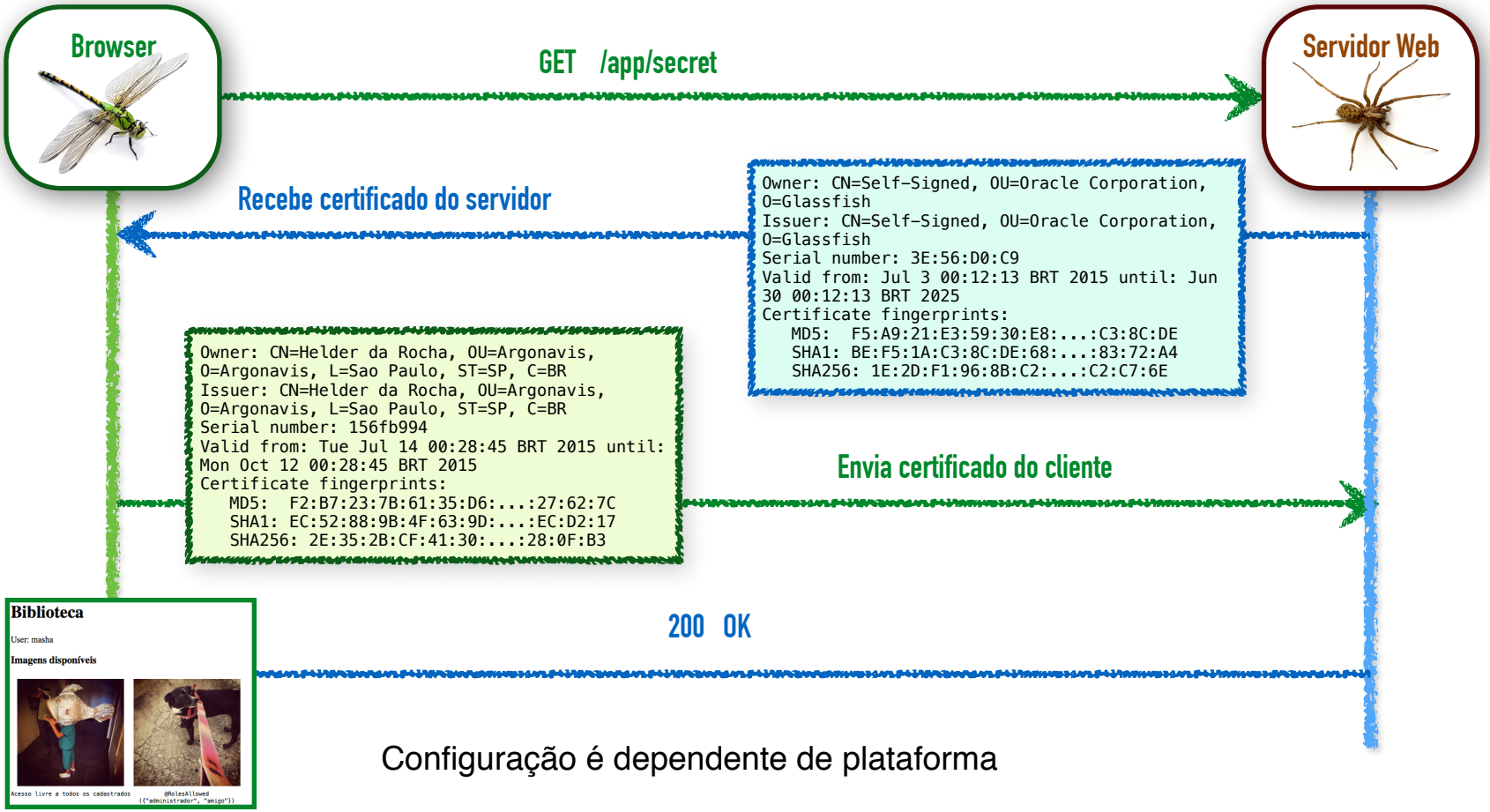
200 OK

Para fazer logout: **HttpSession#invalidate()**

CLIENT-CERT

```
<login-config>
   <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

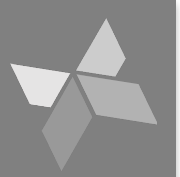
configuração
web.xml



Configuração é dependente de plataforma

Autenticação via HttpServletRequest

- API programática (permite autenticação local ou via serviço externo)
- **getAuthType**(): String
  - Retorna string com mecanismo local de autenticação (BASIC, DIGEST, FORM, CLIENT-CERT) ou null (se usar um mecanismo não-declarativo como JASPIC)
- **authenticate**(HttpServletResponse response)
  - Autenticar usando mecanismo configurado
- **login**(String nome, String senha)
  - Autenticar usando login e senha
- **logout**()
  - Faz com que getRemoteUser(), getCallerPrincipal() e getAuthType() retorne null

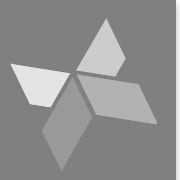


Associação de perfis com usuário no Tomcat

- A configuração de Identity Stores é proprietário. No Tomcat eles são chamados de domínios de segurança (security realms) (http://localhost:8080/tomcat-docs/realm-howto.html)
  - Há três níveis diferentes de configuração. Um grava os dados em banco relacional (JDBC Realm), outro em LDAP via JNDI (JNDI Realm) e o mais simples usa um par de arquivos (Memory Realm)
- Para usar Memory Realm, localize o arquivo **tomcat-users.xml** no diretório conf/ em **$TOMCAT_HOME** e acrescente os usuários, senhas e perfis desejados

```
<tomcat-users>
  <user name="einstein" password="e=mc2" roles="visitante" />
  <user name="caesar"   password="rubicon"
                        roles="administrador, membro"  />
  <user name="brutus"   password="tomcat" roles="membro" />
</tomcat-users>
```

tomcat-users.xml



Autorização em web.xml

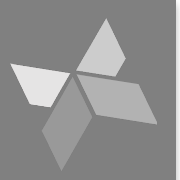
- É preciso declarar os roles no web.xml usando **<security-role>** (em servlets pode ser via anotações).
  - Mapeamento dos roles com usuários/grupos é dependente de servidor (glassfish-web.xml, jboss-web.xml, etc.)

```
<web-app> ...
        <security-role>
                <role-name>administrador</role-name>
        </security-role>

        <security-role>
                <role-name>amigo</role-name>
        </security-role>

        <security-role>
                <role-name>especial</role-name>
        </security-role>
   ...
</web-app>
```

Security constraints

- Bloco <security-constraint> contém três partes
  - **Um ou mais** <web-resource-collection>
  - **Pode** conter **um** <auth-constraint> (lista de roles)
  - **Pode** conter **um** <user-data-constraint> (SSL/TLS)

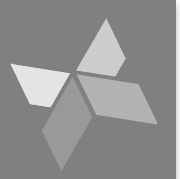
```
<security-constraint>
	<web-resource-collection>
		<web-resource-name>Área restrita</web-resource-name>
		<url-pattern>/secreto/*</url-pattern>
	</web-resource-collection>
	<auth-constraint>
		<role-name>administrador</role-name>
	</auth-constraint>
</security-constraint>
```



Web resource collection

- Agrupa recursos e operações controladas através de padrões de URL + métodos HTTP
  - Métodos HTTP não informados são restritos, mas métodos não informados estão descobertos
  - Use <deny-uncovered-http-methods/> ou outro bloco negando acesso a todos exceto <http-method-omission>

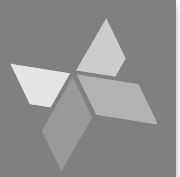
```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>Área restrita</web-resource-name>
                <url-pattern>/secreto/*</url-pattern>
                <url-pattern>/faces/castelo.xhtml</url-pattern>
                <http-method>GET</http-method>
                <http-method>POST</http-method>
        </web-resource-collection>
    ...
</security-constraint>
```



Authorization constraint

- Com **<auth-constraint>** as **<web-resource-collection>** são acessíveis apenas aos **<role-name>** declarados
  - Sem **<auth-constraint>**: sem controle de acesso (livre)
  - **<auth-constraint />** vazio: ninguém tem acesso

```
<web-app>
        ...
        <security-constraint>
                <web-resource-collection> ... </web-resource-collection>
                <web-resource-collection> ... </web-resource-collection>
                <auth-constraint>
                        <role-name>administrador</role-name>
                        <role-name>especial</role-name>
                </auth-constraint>
        </security-constraint>
                ...
        <security-constraint> ....</security-constraint>
</web-app>
```



Transport guarantee

- Garantias mínimas para comunicação segura SSL/TLS
  - **NONE**: (ou ausente) não garante comunicação segura
  - **INTEGRAL**: proteção integral, autenticação no servidor
  - **CONFIDENTIAL**: proteção integral, autenticação: cliente e servidor

```
<web-app> ...
        <security-constraint>
                ...
                <user-data-constraint>
                        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
                </user-data-constraint>
        </security-constraint>
        <security-constraint>
                ...
                <user-data-constraint>
                        <transport-guarantee>INTEGRAL</transport-guarantee>
                </user-data-constraint>
        </security-constraint>
</web-app>
```

Usando anotações @ServletSecurity

- Alternativa a web.xml

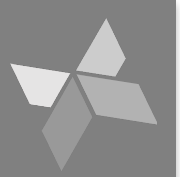
```
@WebServlet(name = "ServletSecreto", urlPatterns = {"/secreto"})
@ServletSecurity(
    @HttpConstraint(
            transportGuarantee = TransportGuarantee.CONFIDENTIAL,
            rolesAllowed = {"amigo", "administrador"}
    )
)
public class ServletSecreto extends HttpServlet { ... }
```

```
@WebServlet(name = "LabirintoServlet", urlPatterns = {"/labirinto"})
@ServletSecurity(httpMethodConstraints={
    @HttpMethodConstraint("GET"),
    @HttpMethodConstraint(value="POST", rolesAllowed={"outro"}),
    @HttpMethodConstraint(
            value="TRACE",
            transportGuarantee = TransportGuarantee.NONE,
            rolesAllowed = {"amigo", "administrador"}
    )
})
public class LabirintoServlet extends HttpServlet { ... }
```

Autorização com HTTPServletRequest

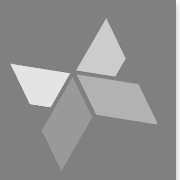
- Acesso via API a **principal** e **roles** permitem controle programático da autorização
- **getUserPrincipal()**
  - Obtém java.security.Principal (getName() obtém nome)
- **isUserInRole("role")**
  - Testa se o usuário autenticado faz parte do role

```
String loggedUser = request.getUserPrincipal().getName();

if( !request.isUserInRole("administrador")) {
        if (!loggedUser.equals("cerebro")) {
                throw new SecurityException("...");
        }
}
```



Exercícios

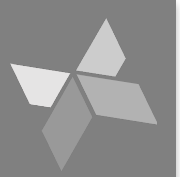
- 1. Configure um MemoryRealm com o Tomcat e autenticação BASIC ou FORM para restringir acesso a alguns servlets
  - Crie um usuário e roles associados
  - Configure acesso a diferentes URLs e métodos HTTP
  - Configure acesso usando SSL



Exercícios de revisão

- Faça os primeiros três exercícios* da aplicação Biblioteca:
  - 1. Acesso a uma base de dados na memória (implementar uma listagem de livros a partir de uma classe existente)
  - 2. Aplicação JDBC FrontController com comandos para listar e inserir livros na Biblioteca
  - 3. Refatorar a aplicação para que a camada Web seja isolada da camada de dados através de um DAO

*Exercicios: http://www.argonavis.com.br/download/exercicios_javaee.html



Referências

- **Especificações Java EE**
  - Java EE 7 https://java.net/projects/javaee-spec/pages/Home
  - Java™ Servlet Specification. Version 3.1. Oracle. 2013.
- **Tutorial Java EE**
  - Java EE 6 http://docs.oracle.com/javaee/6/tutorial/doc/
  - Java EE 7 http://docs.oracle.com/javaee/7/tutorial/doc/
- **Livros**
  - Arun Gupta. *Java EE 7 Essentials*. O'Reilly and Associates. 2014