Módulo 6

Programação WEB

Lição 3

Classes Servlets Avançadas

Versão 1.0 - Nov/2007

Autor
Daniel Villafuerte

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Allan Souza Nunes
Allan Wojcik da Silva
Angelo de Oliveira
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Denis Mitsuo Nakasaki
Emanoel Tadeu da Silva Freitas
Felipe Gaúcho
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Luciana Rocha de Oliveira
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauro Cardoso Mortoni
Paulo Oliveira Sampaio Reis
Pedro Henrique Pereira de Andrade
Ronie Dotzlaw
Sergio Terzella
Thiago Magela Rodrigues Dias
Vanessa dos Santos Almeida
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Na lição anterior, reparamos como as classes *Servlets* podem ser utilizadas pelos desenvolvedores Java para receber requisições de clientes e produzir respostas dinamicamente. Também vimos como distribuir *Servlets* empacotando-as em arquivos WAR e carregando-as no contêiner WEB.

Ao final desta lição, o estudante será capaz de:

- Redirecionar respostas
- Identificar o escopo dos objetos
- Utilizar sessões e rastreamento de sessão
- Utilizar Filtros

2. Redirecionamento de Resposta

Existem casos onde queremos que a servlet tenha somente a responsabilidade de fazer algum processamento inicial e deixe a geração do conteúdo a alguma outra entidade.

Suponhamos que seja obtido do usuário algum valor e o apresentemos com diversas visões baseadas no valor. Digamos também que tenhamos um site que, após processar a entrada do usuário, apresente diferentes páginas dependendo do perfil do usuário no sistema. Colocar toda a geração de conteúdo para todas as páginas em uma única servlet pode torná-la não gerenciável. Nestes casos, seria melhor a servlet redirecionar a geração da saída.

Existem dois modos que podemos utilizar para realizar esse redirecionamento.

- Através do objeto *RequestDispatcher*
- Através do método *sendRedirect*() encontrado no objeto *HttpServletResponse*

2.1. RequestDispatcher

É possível obter uma instância do objeto *RequestDispatcher* invocando o seguinte método:

```
public RequestDispatcher getRequestDispatcher(String path)
```

Este método pode ser encontrado no objeto *HttpServletRequest*.

O parâmetro String que este método recebe é a localização da página HTML, JSP ou a servlet para a qual se queria disparar a requisição. Uma vez com a instância do objeto *RequestDispatcher*, existe a opção de se invocar um dos seguintes métodos:

```
public void include(ServletRequest req, ServletResponse res)
public void forward(ServletRequest req, ServletResponse res)
```

Ambos métodos pegam o conteúdo produzido no local especificado e torna-o parte da resposta da servlet para o usuário. A principal diferença entre eles é que, o método *forward* faz com que a página alvo tenha toda a responsabilidade de gerar a resposta, enquanto que o include só incorpora o conteúdo da página alvo. Utilizando o método *include*, pode-se adicionar outro conteúdo a reposta ou mesmo incluir outra página alvo.

Para ilustrar a diferença entre os dois, são apresentadas duas *Servlets* abaixo. O código é virtualmente idêntico. A primeira faz o uso do método *include* enquanto a segunda faz uso do método *forward*.

```
public DispatchServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("Error occurred during login request processing");
    RequestDispatcher rd = request.getRequestDispatcher("/login.html");
    rd.include(request, response);
  }
}

public DispatchServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("Error occurred during login request processing");
    RequestDispatcher rd = request.getRequestDispatcher("/login.html");
    rd.forward(request, response);
  }
}
```

Ambas *servlets* definidas acima usam uma página denominada login.html para formar a base da

saída dos dados. A definição desta página segue abaixo:

```
<H1>Login Page</H1>
<form action="DispatchServlet">
 User name : <input type="text" name="loginName"><br/>
 Password : <input type="password" name="password"><br/>
 <input type="submit"/>
</form>
```

Ao executar a primeira *Servlet*, a que utiliza o método *include*, a seguinte saída é gerada:

Figura 1: Saída de DispatchServlet utilizando o método include

Executando a segunda *Servlet* temos o seguinte resultado:

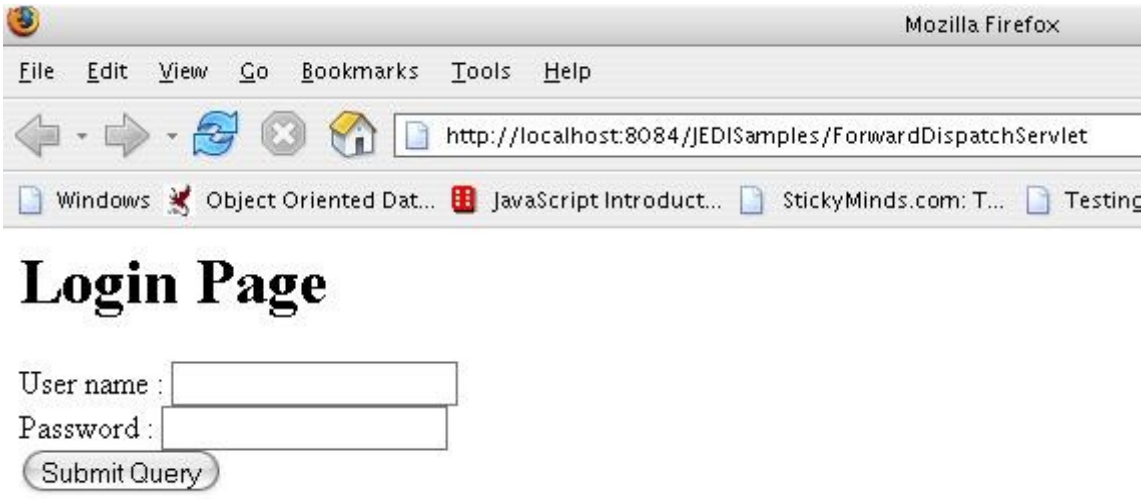

Figura 2: Saída de DispatchServlet utilizado método forward

Dos códigos praticamente idênticos, temos duas saídas diferentes. Ao utilizar o método include, a mensagem que é enviada para o usuário antes de se chamar o método. Este não é o caso para a saída do método *forward* onde a mensagem é apresentada antes da chamada do método não se tornar parte da saída.

Com o método *forward*, todo o conteúdo do buffer da resposta é limpo antes da chamada do método, e depois a resposta é imediatamente enviada. Nenhum conteúdo pode ser adicionado. Com o método *include*, todo o conteúdo colocado no buffer de resposta é mantido antes e depois da chamada do método.

Utilizando-se o método include, é possível que a servlet apresente a saída de diferentes fontes de uma só vez. É também permitido concatenar mensagens a outro conteúdo estático. Tome-se o seguinte exemplo:

```
public LoginServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    RequestDispatcher rd = null;

    // recupera os parâmetros do formulário do usuário
    String loginName = request.getParameter("loginName");
    String password = request.getParameter("password");
    User user = null;

    // cria o JavaBean implementando a funcionalidade de autenticação
    UserService service = new UserService();
    user = service.authenticateUser(loginName, password);

    if (user == null) {
      // gera a mensagem de erro
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("User does not exist with given login and/or password");

      // retorna para a página de login do usuário
      rd = request.getRequestDispatcher("/login.html");
      rd.include(request, response);
      out.close();
    } else {
      // armazena o objeto User na sessão
      HttpSession session = request.getSession();
      session.setAttribute(ApplicationConstants.USER_OBJECT, user);

      // constrói a resposta a partir de múltiplos componentes HTML
      rd = request.getRequestDispatcher("/header.html");
      rd.include(request, response);

      rd = request.getRequestDispatcher("/mainContent.html");
      rd.include(request, response);

      rd = request.getRequestDispatcher("/footer.html");
      rd.include(request, response);
    }
}
```

Como o método include somente adiciona conteúdo de outra fonte e não envia a resposta após sua chamada, podemos utilizá-lo repetidamente. Utilizando este princípio, a classe *LoginServlet* apresentada acima é capaz de criar uma resposta contendo três páginas diferentes.

2.2. sendRedirect

A outra forma de redirecionar a saída de uma entidade fora de uma servlet é o método *sendRedirect*. Este método tem a seguinte assinatura e pode ser encontrado no objeto *HttpServletResponse*:

```
public void sendRedirect(String relativePath)
```

O parâmetro que o método recebe é um String que representa o caminho para a página alvo a ser redirecionada para o usuário. A chamada a este método instrui o *browser* para enviar outra requisição HTTP para a página alvo especificada.

Isto torna a página alvo a única entidade responsável por gerar o conteúdo que, de certa forma, é de resultado similar ao se utilizar o método *forward* do objeto *RequestDispatcher* discutido anteriormente. Contudo, a requisição reenviada apresenta diversas diferenças práticas comparadas ao método *forward*:

- A URL no *browser* reflete o alvo especificado. Isto torna do método *sendRedirect* não desejável caso não se queira que o usuário esteja ciente do redirecionamento.
- Como efetivamente é uma nova requisição, os dados armazenados no objeto da requisição são descartados. Os parâmetros fornecidos pelo usuário, se existirem, devem ser resubmetidos caso a página alvo necessite destes. Os dados que estiverem armazenados no objeto *request* devem ser mantidos de alguma forma. Caso contrário, serão perdidos.

É recomendado então que o método include seja utilizado para permitir saída a partir de diversas fontes. O método *forward* deveria ser utilizado no redirecionamento para componentes cujo conteúdo seja gerado de forma dinâmica (ex: *Servlets* e *JSPs*), enquanto que o método *sendRedirect* deveria ser utilizado no redirecionamento para componentes que gerem conteúdo estático.

3. Objetos de Escopo

A especificação servlet nos apresenta quatro escopos onde os dados podem ser armazenados, de forma que possam ser compartilhados entre os componentes. Eles estão listados abaixo em ordem crescente de visibilidade:

- Página – O escopo de página é definido para ser o escopo de uma página JSP simples. As variáveis declaradas dentro da página são visíveis somente nela e deixaram de ser visíveis uma vez que o serviço da página for finalizado. O objeto associado a esse escopo é o objeto *PageContext*.
- Requisição – O escopo de requisição é definido pela requisição simples proveniente do cliente. Os dados que são armazenados neste escopo são visíveis a todos componentes WEB que manipulam a requisição do cliente. Após a requisição do cliente ter sido finalizada, ou seja, o cliente recebeu uma resposta HTTP e finalizou a conexão com o servidor, todos os dados armazenados dentro deste escopo não são mais visíveis.

O objeto associado a este escopo é o objeto *HttpServletRequest*. As instâncias deste objeto estão prontamente disponíveis às *Servlets* como parâmetros que podem ser obtidos nos métodos de serviço invocados pelo contêiner através da requisição do cliente.

- Sessão – Este é o escopo que abrange uma "sessão" com o cliente. Esta sessão se inicia quando o cliente acessa a aplicação pela primeira vez e finaliza quando o usuário realiza o *logout* do sistema ou a partir de um *timeout* definido no servidor. Os dados deste escopo estão visíveis a todos componentes WEB que o cliente utilizar durante este intervalo. Os dados de uma sessão de usuário, contudo, não estão visíveis para outros usuários.

O objeto associado com este escopo é o objeto *HttpSession*. Uma instância deste objeto pode ser obtida utilizando o método *getSession*() do objeto *HttpServletRequest*.

- Aplicação – Este escopo abrange toda a aplicação. Os dados armazenados neste escopo são visíveis a todos os componentes, independente de requisição de usuário ou sessão que os estejam manipulando, e dura até a aplicação ser finalizada.

O objeto associado com esse escopo é o objeto *ServletContext*. Como citado anteriormente, ele pode ser obtido através da chamada do método *getServletContext*() do um objeto *ServletConfig* válido.

3.1. Armazenando e Recuperando Dados do Escopo

Todos os objetos do escopo mencionados acima contém métodos comuns para recuperar e armazenar dados neles. Para armazenar os dados, utilize o método:

```
public void setAttribute(String chave, Object valor);
```

O parâmetro String que o método recebe armazenará o Objeto associado a a este valor. Para recuperar os dados posteriormente, passar a mesma chave como parâmetro para o método:

```
public Object getAttribute(String chave);
```

Se não houver objeto a ser recuperado para uma dada chave, o valor null é retornado pelo método. Além disso, como o tipo de retorno é Object, fica a cargo do desenvolvedor fazer o *casting* para a classe apropriado e realizar checagem de tipo.

Para remover um atributo do escopo, simplesmente invoque o método *removeAttribute*() e passe a chave do atributo como parâmetro.

3.2. Cenário de Exemplo

Vamos analisar o seguinte cenário: a aplicação deverá exibir os detalhes de uma pessoa a partir de seu *personalID*. Este *personalID* será fornecido pelo usuário. Para promover a manutenibilidade, o componente que recuperará os detalhes pessoais será separado do componente que exibirá as informações para o usuário. Estes dois componentes farão a

comunicação utilizando o armazenamento de dados e as estruturas disponíveis no escopo de requisição.

```
public PersonalDataRetrievalServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {

    // recupera o parâmetro personalID informado pelo usuário
    String personalID = request.getParameter("personalID");

    // cria o objeto de negócios que manipula uma implementação para obtenção de
    // dados para um dado id.
    DataService service = new DataService();
    // recupera o objeto person que contém os detalhes necessários
    Person person = service.retrievePersonalDetails(personalID);

    // armazena os dados no escopo requisição utilizando uma chave
    request.setAttribute(ApplicationConstants.PERSON, person);

    // forward da requisição para o componente que exibirá os detalhes
    RequestDispatcher dispatcher =
      request.getRequestDispatcher("/DisplayServlet");
    dispatcher.forward(request, response);
  }
}

public DisplayServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws
    ServletException, IOException {

    // recupera o objeto Person que contém os detalhes
    Person person = (Person) request.getAttribute(ApplicationConstants.PERSON);

    // constrói a resposta baseado nas informações
    StringBuffer buffer = new StringBuffer();
    buffer.append("<HTML><TITLE>Personal Info</TITLE>");
    buffer.append("<BODY><H1>Details : </H1><br/>");
    buffer.append("Name : ");
    buffer.append(person.getName());
    buffer.append("<br> Address : ");
    buffer.append(person.getAddress());
    buffer.append("</BODY>");

    // exibe a resposta
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println(buffer.toString());
    out.close();
  }
}
```

O código para o objeto *DataService* e o objeto *Person* não são listados aqui com suas implementações porquê não são relevantes para nosso exemplo. O que o exemplo acima mostra é como armazenar os dados de um objeto em uma servlet e como recuperar os mesmos dados a partir de outra servlet que possui acesso ao mesmo escopo.

Seguem algumas notas sobre o exemplo. Primeiro, a segunda servlet foi capaz de recuperar os dados da requisição porquê ela é parte da mesma requisição. Se fosse usado o método *sendRedirect*() como meio de redirecionamento de controle da servlet, o método *getAttribute*() retornaria null já que o método *sendRedirect*() teria feito com que o *browser* enviasse outra requisição. Isto efetivamente acabaria com o tempo de vida da requisição, e com os objetos armazenados dentro dela.

Segundo, é recomendado que as chaves utilizadas para armazenar e recuperar os dados estejam

disponíveis para a aplicação como constantes, conforme o exemplo acima. Isto assegura que a mesma chave seja utilizada para armazenamento e recuperação, reduzindo a possibilidade de erros que possam ser encontrados durante o desenvolvimento.

4. Rastreabilidade e Gerência de Sessão

Recordemos que o protocolo HTTP foi projetado para ser um protocolo conectado, sem estado. Quando um *browser* enviar uma requisição para o servidor, ele abre uma conexão, envia uma requisição HTTP, consome a resposta HTTP, e então fecha a conexão. Como cada requisição de um *browser* é efetivamente enviada por diferentes conexões a cada vez, e o servidor não sabe dos clientes que o estão acessando, o problema da manutenção do estado para uma transação de um usuário em particular é um problema a ser enfrentado.

Uma solução para este problema é o servidor manter o conceito de uma "sessão de usuário". Com uma sessão, o servidor é capaz de reconhecer um cliente no meio de múltiplas requisições. Com este característica, o servidor consegue agora ser capaz de manter o estado para um cliente.

Para tal sessão existir, o *browser* cliente deve ser capaz de enviar dados para o servidor além da requisição HTTP padrão. Estes dados permitem ao servidor identificar qual usuário está realizando a requisição, e manter seu estado conciso. Existem 3 soluções típicas para esse problema: *cookies*, URL-*rewriting* (reescrita de URL) e campos ocultos no formulário.

4.1. Métodos Tradicionais para manter o Estado da Sessão

4.1.1. Cookies

Cookies são pequenas estruturas de dados utilizados pelo servidor WEB que entregam dados para o *browser* cliente. Estes dados são armazenados pelo *browser* e, e em algumas circunstâncias, o *browser* retorna os dados de volta ao servidor WEB.

Ao utilizar *cookies*, os *Servlets* podem armazenar "identificadores das sessões" que podem ser utilizados para identificar o usuário como um participante de uma sessão em particular. Após o identificador ser gerado, ele é armazenado num objeto *Cookie*, e é enviado de volta ao *browser* cliente para ser armazenado. Este objeto *Cookie* pode então ser obtido toda vez a partir do objeto da requisição para determinar se o usuário está em determinada sessão.

Abaixo segue um trecho de código sobre como utilizar *Cookies* para rastreamento de sessões nas *Servlets*:

```
...
// gerar um novo session ID para o usuário
String sessionID = generateSessionID();

// criar novo mapa que será utilizado para armazenar os dados a serem mantidos
na sessão
HashMap map = new HashMap();

// recuperar um mapa utilizado como contêiner para as informações do usuário
HashMap containerMap = retrieveSessionMaps();

// adicionar o novo mapa criado no mapa contendo todas as informações de sessão
containerMap.put(sessionID, map);

// criar o cookie que será armazenado no browser
Cookie sessionCookie = new Cookie("JSESSIONID", sessionID);

// adicionar o cookie à resposta e pedir ao browser para armazená-lo
response.addCookie(sessionCookie);
...
```

Para obter o MAP contendo os dados da sessão, as *Servlets* obtêm o *Cookie* contendo o ID da sessão, e então obtêm o *HashMap* apropriado utilizando o identificador da sessão como uma chave.

Embora os *Cookies* sejam uma boa solução para o rastreamento, seu uso requer que os desenvolvedores manipulem diversos detalhes, como:

- gerar um id único para a sessão de cada usuário,

- recuperar o *Cookie* apropriado do *Browser* que contém o *Session* ID e
- definir um tempo apropriado para a expiração do *Cookie*.

Além dos detalhes acima, um problema com a utilização de *Cookies* é que alguns usuários desabilitam o suporte a *Cookies* no *Browser* por questões de segurança. São necessárias abordagens alternativas.

4.1.2. URL Rewriting (Reescrita de URL)

Nesta abordagem, o *Browser* cliente concatena um *Session* ID único ao fim de cada requisição que ele faz ao servidor. Este *Session* ID pode então ser lido, e a informação apropriada ao usuário é recuperada.

Esta é outra boa solução que funciona mesmo se o usuário desabilitar o uso de *Cookies*. Contudo, existem dois problemas associados com essa abordagem. Primeiro, o servidor deve assegurar que cada *Session* ID que ele fornece seja único. Segundo, a aplicação completa deve ser escrita de forma que o *Session* ID concatenado a todos os *Links*/URLs apontem para a aplicação. Qualquer requisição que não inclua o *Session* ID não seria considerada parte da sessão e não teria acesso às informações específicas da sessão.

4.1.3. Campos ocultos no fomulário

Nesta abordagem, um campo oculto no formulário é introduzido nos formulários HTML com o valor sendo definido a uma sessão particular. Entretanto, este método é muito limitado pelo fato de que somente pode ser utilizando quando há um formulário na página que o cliente esteja utilizando.

4.2. Rastreabilidade de Sessão em Servlets

A especificação de *Servlets* oferece uma API de alto nível para fornecer acesso ao rastreamento de sessão: a API *HttpSession*. Ao utilizar esta API, o desenvolvedor não precisa se preocupar com muitos dos detalhes mencionados acima: reconhecimento do *Session* ID, detalhes da manipulação de *Cookies* e detalhes da extração de informações da URL. Estes detalhes são transparentes ao desenvolvedor. E, além disso, o desenvolvedor conta com um local apropriado conveniente para armazenar os dados de uma sessão para o usuário. O uso correto da API *HttpSession* também permite à aplicação trocar para o método de reescrita de URL caso seja detectado que o suporte a *Cookies* no *Browser* cliente esteja desabilitado.

4.2.1. Obtendo uma instância do objeto HttpSession

O objeto *HttpSession* representa os dados da sessão associados a uma dada requisição por um cliente, este pode ser obtido chamando o método *getSession*() do objeto *HttpServletRequest*. O contêiner é então responsável por ler os dados do cliente (ou dos *Cookies* ou da reescrita de URL), e criar uma instância do objeto *HttpSession*.

Passando um valor booleano para o método *getSession*() (ex., *getSession*(true)), é possível especificar ao servidor se um novo objeto *HttpSession* deve ser automaticamente criado no caso de um usuário não ter uma sessão.

Um código de exemplo mostrando como obter uma instância do objeto HttpSession é apresentado a seguir:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
  ....
  HttpSession session = request.getSession();
```

4.2.2. Armazenando e recuperando dados em uma sessão

Através da classe javax.servlet.http.HttpSession, os desenvolvedores não precisam mais gerenciar objetos explicitamente para armazenar dados em uma sessão de usuário, basta utilizar um dos seguintes métodos:

```
public void setAttribute(String key, Object value)
public Object getAttribute(String key)
```

Este são os mesmos métodos que encontramos durante nossa discussão sobre diferentes escopos de objetos. Na verdade, os objetos HttpSession que estamos trabalhando agora representam o escopo de sessão discutido previamente.

Outro exemplo de como salvar e recuperar objetos no escopo de sessões é apresentado abaixo:

```
public class LoginServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    User user = authenticateUser();
    ...
    HttpSession session = request.getSession();
    session.setAttribute("userObject", user);
    ...
  }
}
```

O trecho de código acima mostra o tratamento de uma entrada do usuário através de uma *Servlet*. Uma atividade comum realizada por tais tratamentos, além da autenticação de usuários, é a armazenagem de um Objeto tipo *user* que representa o usuário no escopo da sessão. Este objeto pode ser recuperado por outras *Servlets* que precisarem de informações sobre o usuário atual, tais como:

```
public class InfoServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    HttpSession session = request.getSession();
    User user = (User)session.getAttribute("userObject");
    // implementar aqui as operações necessárias sobre o objeto "user"
  }
}
```

4.2.3. Removendo dados armazenados na sessão do usuário

Para remover dados armazenados no escopo de sessão, chame o método *removeAttribute*() e passe a chave (tipo String) associada ao valor em questão.

Um exemplo de servlet realizando tal remoção é apresentado abaixo. Neste cenário imaginário, a aplicação armazenou um valor temporário dentro da sessão associado à chave "tempData". Após ter realizado todas as operações necessárias com os valores associados a esta chave, o código remove o dado da sessão.

```
public class InfoProcessingServlet extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
    HttpSession session = request.getSession();
    session.removeAttribute("tempData");
    // redirecionar a resposta para a próxima página web
    response.sendRedirect("next.html");
  }
}
```

4.2.4. Encerrando a sessão

Encerramento Automático através de *timeout*

Sessões são automaticamente encerradas após um intervalo pré-determinado de tempo (diz-se que "a sessão expira"). Este intervalo pode ser inspecionado e configurado no descritor de distribuição da aplicação (deployment descriptor).

O descritor de distribuição para o exemplo *FirstServlet* é apresentado abaixo para que possamos revisá-lo novamente.

```
...
```

```
</servlet-mapping>
 <session-config>
  <session-timeout>
   30
  </session-timeout>
 </session-config>
<welcome-file-list>
...
```

O valor 30 do elemento <session-timeout> indica que o servidor deve esperar pelo menos 30 minutos de inatividade do usuário antes de encerrar uma sessão. Após este período, todos os objetos contidos no escopo de sessão serão removidos e o objeto HttpSession se tornará inválido.

Forçar o encerramento da sessão através de código (encerramento programático)

Além do período de time-out, uma sessão pode ser terminada programaticamente através do método *invalidate*() da classe *HttpSession*. O uso deste método é recomendado para situações onde o usuário não precisa mais fazer parte de uma sessão, como por exemplo quando realizamos a saída da aplicação.

Encerrar uma sessão programaticamente, ao invés de esperar o *time-out*, é também uma forma de liberar os recursos mais rapidamente. Isso também garante que qualquer informação sigilosa armazenada na sessão seja descarregada do escopo de sessão imediatamente, dificultando que tais informações sejam acessadas por outras pessoas.

Um exemplo de tratamento de *logout* por uma *Servlet* é apresentado abaixo:

```
public class LogoutServlet extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
    HttpSession session = request.getSession();
    // realizar aqui outras tarefas do logout
    // invalidar a sessão do usuário
    session.invalidate();
    // redirecionar o usuário para a página de login
    response.sendRedirect("login.jsp");
  }
}
```

4.2.5. Realizando Redirecionamento de URL (URL-Rewriting)

Por default, objetos *HttpSession* usam *Cookies* para manter o escopo de sessões. Entretanto, devemos desenvolver nossas aplicações adicionando suporte para redirecionamento de URLs, permitindo que a aplicação rode igualmente bem em navegadores que não suportam *Cookies*.

Podemos adicionar suporte a redirecionamento de URLs usando o método *encodeURL*() encontrado na interface *javax.servlet.http.HttpServletResponse*. Esse método recebe como parâmetro uma String representando um caminho (path) ou uma URL. O método, então, verifica se o suporte a *Cookies* está habilitado no navegador do usuário. Se o suporte a *Cookies* está habilitado, o método retorna a String recebida como parâmetro. Caso contrário, habilita o redirecionamento de URLs incluindo a identificação da sessão (*Session* ID) como um dos parâmetros da URL.

O código abaixo é um exemplo de como usar o método *encodeURL*:

```
...
String encodedURL = response.encodeURL("/welcome.jsp");
out.println("<A HREF='" + encodedURL + "'>Clique aqui para continuar</A>");
...
```

Para fornecer a funcionalidade de redirecionamento de URLs em nossa aplicação WEB, devemos substituir todas as URLs exibidas ao usuário, como *hyperlinks*, por alguma página passada com o método encodeURL. Caminhos (*paths*) repassados ao método sendRedirect, discutido anteriormente, também devem ser novamente codificados – desta vez, usando o método encodeRedirectURL().

5. Filtros (Filter)

Filtros são componentes WEB avançados, introduzidos desde a especificação *Serlvet* 2.3. São componentes que se interpõem entre uma requisição do cliente e um determinado recurso. Qualquer tentativa de recuperar o recurso deve passar pelo filtro. Um recurso pode ser qualquer conteúdo estático ou dinâmico (HTML, JSP, GIF, etc.).

Filtros funcionam interceptando as requisições do cliente ao servidor através de um encadeamento de filtros. Ou seja, existe uma série de filtros, através dos quais uma requisição passa, antes de, finalmente, acessar um determinado recurso. Quando a requisição passa pelo filtro, esse filtro pode processar os dados contidos na requisição e também decidir sobre o próximo passo da requisição – que pode ser encaminhada para um outro filtro (caso o filtro atual não seja o último da cadeia de filtros), acessar o recurso diretamente ou impedir que o usuário acesse o recurso desejado.

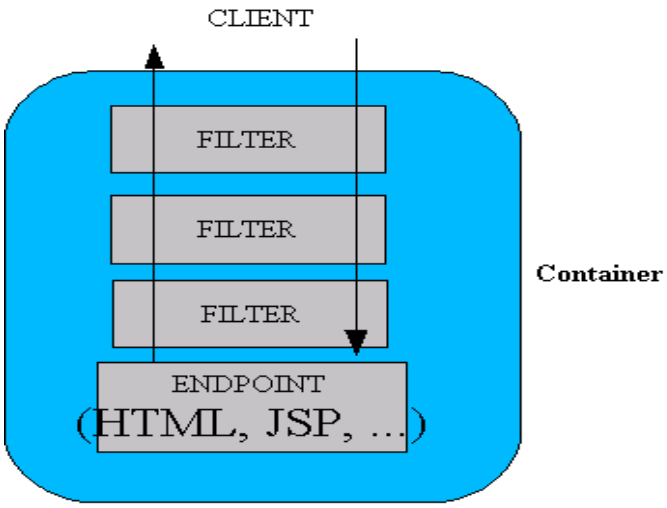

Figura 3: Ilustração de requisição de passagem por filtros

A figura acima apresenta a maneira pela qual as requisições passam através de filtros, antes de acessar seu ponto final (*EndPoint*) – um recurso.

Filtros são componentes úteis, visto que fornecem ao desenvolvedor uma forma simples de incluir processamento adicional, antes que os recursos de uma aplicação WEB sejam acessados. Esse tipo de funcionalidade também é possível de ser implementado através do redirecionamento de *Servlets* e requisições. Entretanto, o uso de redirecionamento é bem menos elegante, porque exige que o encadeamento dos processos seja programado diretamente no código das *Servlets* que compõem a aplicação – e cada vez que esse encadeamento mudar, serão necessárias modificações no código das *Servlets* para reconfigurar o encadeamento. Filtros, por sua vez, não sofrem essa limitação, pois é o contêiner que gerencia a seqüência de componentes a serem chamados, antes que uma determinada requisição acesse um recurso. Eventualmente, o contêiner pode ser reconfigurado alterando-se a quantidade e a ordem dos filtros, sem que haja necessidade de alterações no código-fonte da aplicação.

5.1. Criando um Filtro

Para criar um filtro, os desenvolvedores devem criar uma classe que implemente a interface javax.servlet.Filter. Essa interface define os seguintes métodos:

- **void init(FilterConfig config) throws ServletException** – esse método é chamado pelo contêiner durante a primeira vez que o filtro é carregado na memória. Códigos de inicialização devem ser colocados aqui, incluindo o uso de parâmetros de inicialização contidos no arquivo **web.xml** – recebidos no parâmetro *FilterConfig*.
- **void destroy** – esse método é chamado pelo contêiner quando o filtro é descarregado da memória. Isso ocorre normalmente quando a aplicação WEB é encerrada. O código de

liberação de recursos utilizados pelo filtro deve ser inseridos aqui – o encerramento de uma conexão ao banco de dados, por exemplo.

- void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException – esse método contém toda a funcionalidade do filtro, ou seja, as regras que devem ser verificadas, antes que uma requisição possa continuar seu caminho até um recurso. Esse método é chamado pelo contêiner quando o servidor decide que o filtro deve interceptar uma determinada requisição ou resposta ao usuário. Os parâmetros passados para esse método são instâncias das classes *ServletRequest* e *ServletResponse* do pacote *javax.servlet.http* e da classe *FilterChain* do pacote *javax.servlet*. Se o filtro estiver sendo usado por uma aplicação web (o caso mais comum), o desenvolvedor pode realizar o casting entre esses objetos para instâncias das classes *HttpServletRequest* e *HttpServletResponse* do pacote *javax.servlet.http*, respectivamente. Isso permite a recuperação de informações específicas do protocolo HTTP.

Assim como um servlet, o contêiner irá criar uma única instância da classe Filter e usar processamento multi-tarefa (*multi-threading*), para lidar com as requisições concorrentes de vários clientes. Isso significa que esse método deve ser programado no modo *thread-safe*.

Um exemplo de classe implementando um filtro é apresentado abaixo. Esse filtro usa o recurso de entrada disponível na classe *ServletContext*, para registrar todas as requisições do usuário que passam pelo filtro.

```
import java.io.IOException;
import java.util.logging.Filter;
import java.util.logging.LogRecord;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class LoggingFilter implements Filter {
  private FilterConfig config;
  public void init(FilterConfig config) {
    this.config = config;
  }
  public void doFilter(
    ServletRequest request, ServletResponse response, FilterChain chain)
    throws ServletException, IOException {
    // recupera o objeto ServletContext que sera usado para realizar o logging
    ServletContext context = config.getServletContext();

    // cria um registro da URL acessada pelo usuário
    String logEntry = request.getServerName() + ":" + request.getServerPort();
    logEntry += "/" + request.getLocalAddr() + "/" + request.getLocalName();
    logEntry += "--> acessado pelo usuário em " + new java.util.Date();

    // utilizando o recurso de logging disponível na classe ServletContext
    context.log(logEntry);
    // chama o próximo filtro na cadeia de filtros
    chain.doFilter(request, response);
  }
  public void destroy() {}
  public boolean isLoggable(LogRecord arg0) {
    throw new UnsupportedOperationException("Not supported yet.");
  }
}
```

5.2. Encadeamento de Filtros (Filter Chains)

O encadeamento de filtros permite que filtros sejam aplicados a um recurso obedecendo a uma seqüência pré-determinada. A característica de encadeamento dos filtros permite uma lógica diferenciada no uso destes componentes. O encadeamento é a diferença básica entre os filtros e

os *servlets*.

O encadeamento de filtros permite uma separação clara entre as diversas camadas de processamento de requisições. Se, por exemplo, novas funcionalidades precisarem ser implementadas antes do processamento de uma requisição, basta criar um novo filtro. Uma vez que o filtro estiver configurado, basta adicioná-lo à cadeia de filtros aplicados à requisição antes que ela atinja o seu ponto final (um recurso). Isso não interfere em nenhum processamento que já estivesse sendo realizado sobre a requisição, a menos que o código do filtro tenha sido projetado para isso.

A seqüência da cadeia de filtros é determinada pela localização da declaração de um filtro no descritor de distribuição (*deployment descriptor*) e obedece à ordem crescente dos elementos **<filter-mapping>**. Esses elementos representam o mapeamento entre cada filtro e um recurso.

Conforme dito anteriormente, a classe *javax.servlet.FilterChain* representa a seqüência de filtros que serão chamados antes de uma requisição atingir seu ponto final. O único controle que o desenvolvedor tem sobre essa seqüência é o método *doFilter* da classe *FilterChain*: esse método chama o próximo filtro da seqüência ou diretamente o recurso, se o filtro for o último da seqüência. Esse método requer objetos das classes *ServletRequest* e *ServletResponse*, do pacote *javax.servlet*, como parâmetros. Novamente, como não há necessidade do programador se preocupar com o próximo filtro do encadeamento, ele pode se concentrar apenas na lógica da classe que está sendo produzida.

Se um desenvolvedor esquecer de chamar o método *doFilter*, o resto da cadeia de filtros (e eventualmente o recurso final) nunca será acessada. Se essa não for a funcionalidade desejada, deve-se ter atenção no momento de programar o código do filtro para garantir a chamada do método. Entretanto, existem casos em que desejamos realmente interromper a cadeia de filtros. Um bom exemplo é um filtro de segurança que evita o acesso indevido a recursos de uma aplicação caso alguns critérios não sejam obedecidos (senha inválida, etc.).

Um exemplo desse filtro é apresentado abaixo. O código descreve um filtro de segurança que verifica a existência de informações sobre o usuário armazenadas no escopo da sessão. Se não encontrar tais informações o filtro assume que o usuário não está mais logado ou então que a sua sessão expirou. Em qualquer um desses casos, o filtro responde à requisição, redirecionando o usuário à página de *login*.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.logging.Filter;
import java.util.logging.LogRecord;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class SecurityFilter implements Filter {
  private FilterConfig config;

  public void init(FilterConfig config) {
    this.config = config;
  }
  public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain)
    throws ServletException, IOException {
    HttpServletRequest httpRequest = (HttpServletRequest)request;
    HttpServletResponse httpResponse = (HttpServletResponse)response;
    HttpSession session = httpRequest.getSession();

    User user = (User) session.getAttribute("userObject");
    if (user != null) {
```

```
        // se o usuário atender a certas condições, permitir o acesso aos recursos
        chain.doFilter(request, response);
      } else {
        PrintWriter out = response.getWriter();
        out.println("Bem-vindo, por favor, " +
          "faça o login antes de continuar utilizando a aplicação.");
        RequestDispatcher rd = request.getRequestDispatcher("login.jsp");
        rd.include(request, response);
      }
    }
    public void destroy() {}
    public boolean isLoggable(LogRecord arg0) {
      throw new UnsupportedOperationException("Not supported yet.");
    }
  }
```

5.3. Configuração de filtros

A configuração de filtros é muito parecida com a configuração de *Servlets*. Existe uma sessão necessária para configurar cada filtro utilizado na aplicação bem como sessões para configurar o mapeamento entre os filtros e os padrões de URLs aos quais as cadeias de filtros serão aplicadas.

Um exemplo de configuração de filtro é apresentado abaixo:

```
  ...
</context-param>
<filter>
 <filter-name>LoggingFilter</filter-name>
 <filter-class>jedi.filters.LoggingFilter</filter-class>
</filter>
<filter-mapping>
 <filter-name>LoggingFilter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

Dado que a interpretação do arquivo **web.xml** sofre influência da ordem em que os elementos são declarados, recomendamos a declaração dos filtros após os elementos <context-param> e antes de qualquer definição sobre servlets. Também cuide para colocar os elementos <filter-mapping> após a definição dos filtros.

Por default, filtros não são aplicados a componentes web (outros servlets, JSP, etc.) acessados pela classe RequestDispatcher através de chamadas include ou forward. Filtros são aplicados apenas a requisições feitas diretamente pelo cliente. Esse comportamento, entretanto, pode ser modificado, incluindo-se um ou mais elementos <dispatch> ao mapeamento do filtro.

```
...
</context-param>
<filter>
 <filter-name>LoggingFilter</filter-name>
 <filter-class>jedi.filters.LoggingFilter</filter-class>
</filter>
<filter-mapping>
 <filter-name>LoggingFilter</filter-name>
 <url-pattern>/*</url-pattern>
 <dispatch>REQUEST</dispatch>
 <dispatch>INCLUDE</dispatch>
</filter-mapping>
```

Elementos <dispatch> possuem um dos seguintes valores: REQUEST, INCLUDE, FORWARD e ERROR. Isso indica quando um filtro deve ser aplicado apenas a requisições do cliente, apenas em includes, apenas em requisições, apenas em caso de erro ou na combinação desses quatro valores.

6. Exercícios

6.1. Exercícios sobre redirecionamento de respostas

1) Quais são as duas formas de se realizar redirecionamento na API *Servlet*?
2) Qual é a diferença entre utilizar o método include e o método *forward* no objeto *RequestDispatcher*?
3) Qual é a diferença entre utilizar o método *forward* do objeto *RequestDispatcher* e o método *sendRedirect* utilizando o objeto *HttpServletResponse*?

6.2. Exercícios sobre escopo

1) Quais são os diferentes escopos disponíveis em uma aplicação WEB nos quais um objeto pode ser armazenado?
2) Qual objeto é associado ao escopo de aplicação? E ao escopo de sessão?
3) Qual o tempo de vida/visibilidade dos objetos colocados no escopo de sessão? E no escopo de requisição?
4) Dado o código abaixo:

```
public class SenderServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    String message = "Can you hear me?";
    request.setAttribute("messageKey", message);
    response.sendRedirect("ReceiverServlet");
  }
}
```

Se houvesse uma *ReceiverServlet* mapeada para o local indicado dentro do método *sendRedirect*, seria possível recuperar a mensagem? Justifique?

5) Crie uma implementação de *ReceiverServlet* que tente recuperar a mensagem armazenada dentro do escopo de requisição pela classe *SenderServlet* dada no exercício anterior.

6.3. Exercícios sobre rastreabilidade

1) A classe *HttpSession* fornecida pela *Servlet* API utiliza *Cookies* para controlar o escopo de sessão de usuário. Qual é a desvantagem dessa forma de controlar sessões de usuários?
2) Como um desenvolvedor pode recuperar uma instância da classe *HttpSession*?
3) Considere o seguinte código:

```
public class InfoBean {
  private String nome;
  private String numero;
  // métodos get(...) e set(...)
}
```

Se uma instância dessa classe foi armazenada em uma sessão sob a chave "infoBean", crie uma *Servlet* que recupere a instância e exiba os valores dos atributos nome e número.

6.4. Exercícios sobre filtros

1) Quais são os três métodos definidos na interface Filter que devem ser implementados em classes de filtro?
2) Quando configuramos um filtro no descritor de aplicações WEB, essa configuração deve vir antes ou depois da definição das *Servlets*?
3) Após um filtro ter realizado sua funcionalidade, como ele chama o próximo filtro da cadeia de filtros ou o recurso desejado?

4) Considere o seguinte cenário:

Temos uma aplicação WEB com funcionalidades administrativas que podem ser disponibilizadas aos seus usuários comuns. Se todos os recursos necessários a essas funcionalidades administrativas estão localizados no diretório **admin** de nossa aplicação, como configurar um filtro chamado *AdminFilter* tal que todas as requisições a recursos administrativos passem por esse filtro?

5) Crie uma classe que implementa o *AdminFilter* descrito acima. O filtro deve ser capaz de determinar quando um usuário é autorizado ou não a recuperar um valor lógico armazenado no escopo de sessão. O nome da chave usada para armazenar o valor lógico é **isAdmin**. Se o valor for *true*, o usuário está autorizado a acessar os recursos administrativos. Caso contrário, o usuário deve ser redirecionado à página de entrada, com uma mensagem informando que ele não tem as credenciais necessárias para acessar a parte administrativa do sistema.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.