CAPÍTULO 19

Apêndice - Tópicos da Servlet API

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."
— Bill Gates

Este capítulo aborda vários outros pequenos assuntos da Servlet API ainda não tratados, muitos deles importantes para a certificação SCWCD.

19.1 - INIT-PARAMS E CONTEXT-PARAMS

Podemos configurar no web.xml alguns parâmetros que depois vamos ler em nossa aplicação. Uma forma é passarmos parâmetros especificamente para uma Servlet ou um filtro usando a tag `<init-param>` como nos exemplos abaixo:

```
<!-- em servlet -->
<servlet>
  <servlet-name>MinhaServlet</servlet-name>
  <servlet-class>pacote.MinhaServlet</servlet-class>
  <init-param>
    <param-name>nome</param-name>
    <param-value>valor</param-value>
  </init-param>
</servlet>

<!-- em filter -->
<filter>
  <filter-name>MeuFiltro</filter-name>
  <filter-class>pacote.MeuFiltro</filter-class>
  <init-param>
    <param-name>nome</param-name>
    <param-value>valor</param-value>
  </init-param>
</filter>
```

Podemos, inclusive, ter vários parâmetros na mesma servlet ou filtro. Depois, no código Java da Servlet ou do filtro específico, podemos recuperar esses parâmetros usando:

```
// em servlet
String valor = getServletConfig().getInitParameter("nome");

// em filtro, no init
String valor = filterConfig.getInitParameter("nome")
```

Outra possibilidade é configurar parâmetros para o contexto inteiro e não apenas uma servlet específica. Podemos fazer isso com a tag `<context-param>`, como abaixo:

```
<context-param>
  <param-name>nome</param-name>
  <param-value>param</param-value>
</context-param>
```

E, no código Java de uma Servlet, por exemplo:

```
String valor = getServletContext().getInitParameter("nome");
```

Muitos frameworks usam parâmetros no web.xml para configurar. O VRaptor e o Spring são exemplos de uso desse recurso. Mas podemos usar isso em nossas aplicações também para retirar do código Java certas configurações parametrizáveis.

19.2 - WELCOME-FILE-LIST

É possível configurar no **web.xml** qual arquivo deve ser chamado quando alguém acessar uma URL raiz no servidor, como por exemplo:

```
http://localhost:8080/fj-21-agenda/
http://localhost:8080/fj-21-agenda/uma-pasta/
```

São os arquivos index que normalmente usamos em outras plataformas. Mas no Java EE podemos listar os nomes de arquivos que desejamos que sejam os *welcome files*. Basta definí-los no XML e o servidor vai tentá-los na ordem de declaração:

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

Tire suas dúvidas no novo GUJ Respostas

O GUJ é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra

ajudar você a esclarecer suas dúvidas.

Faça sua pergunta.

19.3 - Propriedades de páginas JSP

Como dizer qual o encoding de nossos arquivos jsp de uma maneira global? Como nos proteger de programadores iniciantes em nossa equipe e desabilitar o código `scriptlet`? Como adicionar um arquivo antes e/ou depois de todos os arquivos JSPs? Ou de todos os JSPs dentro de determinado diretório?

Para responder essas e outras perguntas, a API de jsp resolveu possibilitar definir algumas tags no nosso arquivo `web.xml`.

Por exemplo, para desativar *scripting* (os scriptlets):

```
<scripting-invalid>true</scripting-invalid>
```

Ativar *expression language* (que já vem ativado):

```
<el-ignored>false</el-ignored>
```

Determinar o encoding dos arquivos de uma maneira genérica:

```
<page-encoding>UTF-8</page-encoding>
```

Incluir arquivos estaticamente antes e depois de seus JSPs:

```
<include-prelude>/antes.jspf</include-prelude>
<include-coda>/depois.jspf</include-coda>
```

O código a seguir mostra como aplicar tais características para todos os JSPs, repare que a tag `url-pattern` determina o grupo de arquivos cujos atributos serão alterados:

```
<jsp-config>
  <jsp-property-group>
    <display-name>todos os jsps</display-name>
    <description>configuracoes de todos os jsps</description>
    <url-pattern>*.jsp</url-pattern>
    <page-encoding>UTF-8</page-encoding>
    <scripting-invalid>true</scripting-invalid>
    <el-ignored>false</el-ignored>
    <include-prelude>/antes.jspf</include-prelude>
    <include-coda>/depois.jspf</include-coda>
  </jsp-property-group>
</jsp-config>
```

19.4 - Inclusão estática de arquivos

Existe uma maneira em um arquivo JSP de incluir um outro arquivo estaticamente. Isto faz com que o arquivo a ser incluído seja literalmente copiado e colado dentro do seu arquivo antes da primeira interpretação (compilação) do seu jsp.

A vantagem é que como a inclusão é feita uma única vez antes do arquivo ser compilado, essa inclusão é extremamente rápida, porém vale lembrar que o arquivo incluído pode ou não funcionar separadamente.

```
<%@ include file="outra_pagina.jsp" %>
```

19.5 - Tratamento de erro em JSP

Como tratar possíveis exceptions em nossa página JSP? Nossos exercícios de listagem de contatos tanto com scriptlets quanto com JSTL usam o `ContatoDao` que pode lançar uma exceção se o banco de dados estiver fora do ar, por exemplo. Como tratar?

Se nosso JSP é um imenso scriptlet de código Java, o tratamento é o mesmo de códigos Java normais: `try catch`:

```
<html>
  <%
  try {
    ContatoDao dao = new ContatoDao();
    // ... etc ...
  } catch(Exception ex) {
  %>
    Ocorreu algum erro ao acessar o banco de dados.
  <%
  }
  %>
</html>
```

Não parece muito elegante. Mas e quando usamos tags, há uma forma melhor? Poderíamos usar a tag `c:catch`, com o mesmo tipo de problema da solução anterior:

```
<c:catch var="error">
  <jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDao"/>
  <c:forEach var="contato" items="${dao.lista}">
    ....
  </c:forEach>
</c:catch>
<c:if test="${not empty error}">
  Ocorreu algum erro ao acessar o banco de dados.
</c:if>
```

Repare que a própria JSTL nos apresenta uma solução que não se mostra boa para esse tipo de erro que queremos tratar. É importante deixar claro que desejamos tratar o tipo de erro que não tem volta, devemos mostrar uma mensagem de erro para o cliente e pronto, por exemplo quando a conexão com o banco cai ou quando ocorre algum erro no servidor.

Quando estávamos trabalhando com Servlets, havia uma solução simples e elegante: não tratar as exceções de forma espalhada mas sim criar uma página centralizada de tratamento de erros. Naquele caso, conseguimos isso com o `<error-page>`.

Com JSPs, conseguimos o mesmo resultado mas sem XML. Usamos uma diretiva no topo do JSP que indica qual é a página central de tratamento de erro. E nesse caso não precisamos nem de `try/catch` nem de `<c:catch>`:

```
<%@ page errorPage="/erro.html" %>
...
<jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDao"/>
...
```

Nova editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

19.6 - Descobrindo todos os parâmetros do request

Para ler todos os parâmetros do `request` basta acessar o método `getParameterMap` do `request`.

```
Map<String,Object> parametros = request.getParameterMap();
for(String parametro:parametros.keySet()) {
  // faca algo com o parametro
}
```

19.7 - Trabalhando com links com a c:url

Às vezes não é simples trabalhar com links pois temos que pensar na URL que o cliente acessa ao visualizar a nossa página.

A JSTL resolve esse problema: supondo que a sua aplicação se chame `jspteste`, o código abaixo gera a string `/jspteste/imagem/banner.jpg`.

```
<c:url value="/imagem/banner.jpg"/>
```

É bastante útil ao montar menus únicos incluídos em várias páginas e que precisam lidar com links absolutos.

19.8 - Context listener

Sabemos que podemos executar código no momento que uma Servlet ou um filtro são inicializados através dos métodos `init` de cada um deles. Mas e se quisermos executar algo no início da aplicação Web (do contexto Web), independente de termos ou não Servlet e filtros e do número deles?

Para isso existem os **context listeners**. Você pode escrever uma classe Java com métodos que serão chamados automaticamente no momento que seu contexto for iniciado e depois desligado. Basta implementar a interface `ServletContextListener` e usar a tag `<listener>` no web.xml para configurá-la.

Por exemplo:

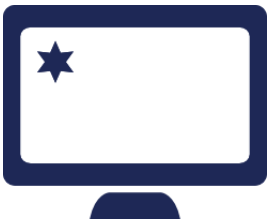
```
public class MeuListener implements ServletContextAttributeListener {
    public void contextInitialized(ServletContextEvent event) {
    System.out.println("Contexto iniciado...");
  }

    public void contextDestroyed(ServletContextEvent event) {
    System.out.println("Contexto desligado...");
    }
}
```

E depois no XML:

```
<listener>
    <listener-class>pacote.MeuListener</listener-class>
</listener>
```

Já conhece os cursos online Alura?

A **Alura** oferece dezenas de **cursos online** em sua plataforma exclusiva de ensino



que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Java, Ruby, Web, Mobile, .NET e outros, com uma **assinatura** que dá acesso a todos os cursos.

Conheça os cursos online Alura.

19.9 - O ServletContext e o escopo de aplicação

As aplicações Web em Java têm 3 escopos possíveis. Já vimos e usamos dois deles: o de request e o de sessão. Podemos colocar um atributo no request com `request.setAttribute(..,..)` e ele estará disponível para todo o request (desde a Action até o JSP, os filtros etc).

Da mesma forma, podemos pegar a `HttpSession` e colocar um atributo com `session.setAttribute(..,..)` e ela estará disponível na sessão daquele usuário através de vários requests.

O terceiro escopo é um escopo global, onde os objetos são compartilhados na aplicação inteira, por todos os usuários em todos os requests. É o chamado **escopo de aplicação**, acessível pelo `ServletContext`.

Podemos, em uma Servlet, setar algum atributo usando:

```
getServletContext().setAttribute("nomeGlobal", "valor");
```

Depois, podemos recuperar esse valor com:

```
Object valor = getServletContext().getAttribute("nomeGlobal");
```

Um bom uso é compartilhar configurações globais da aplicação, como por exemplo usuário e senha de um banco de dados, ou algum objeto de cache compartilhado etc. Você pode, por exemplo, inicializar algum objeto global usando um `ServletContextListener` e depois disponibilizá-lo no `ServletContext` para o resto da aplicação acessar.

E como fazemos para acessar o escopo de aplicação no nosso JSP? Simples, uma das variáveis que já existe em um JSP se chama `application`, algo como:

```
ServletContext application = getServletContext();
```

Portanto podemos utilizá-la através de scriptlet:

```
<%= application.getAttribute("nomeGlobal") %><br/>
```

Como já vimos anteriormente, o código do tipo scriptlet pode ser maléfico para nossa aplicação, sendo assim vamos utilizar Expression Language para acessar um atributo do escopo aplicação:

```
Acessando com EL: ${nomeGlobal}<br/>
```

Repare que a Expression Language procurará tal atributo não só no escopo do `application`, como veremos mais a frente. Para deixar claro que você procura uma variável do escopo de aplicação, usamos a variável implícita chamada `applicationScope`:

Acessando escopo application: `${applicationScope['nomeGlobal']}<br/>`

Métodos no ServletContext

Além da característica de escopo global com os métodos `getAttribute` e `setAttribute`, outros métodos úteis existem na `ServletContext`. Consulte o Javadoc para mais informações.

19.10 - Outros listeners

Há ainda outros listeners disponíveis na API de Servlets para capturar outros tipos de eventos:

- `HttpSessionListener` - provê métodos que executam quando uma sessão é criada (`sessionCreated`), destruída (`sessionDestroyed`);
- `ServletContextAttributeListener` - permite descobrir quando atributos são manipulados no `ServletContext` com os métodos `attributeAdded`, `attributeRemoved` e `attributeReplaced`;
- `ServletRequestAttributeListener` - tem os mesmos métodos que o `ServletContextAttributeListener` mas executa quando os atributos do *request* são manipulados;
- `HttpSessionAttributeListener` - tem os mesmos métodos que o `ServletContextAttributeListener` mas executa quando os atributos da `HttpSession` são manipulados;
- `ServletRequestListener` - permite executar códigos nos momentos que um request chega e quando ele acaba de ser processado (métodos `requestDestroyed` e `requestInitialized`);
- Outros menos usados: `HttpSessionActivationListener` e `HttpSessionBindingListener`.

A configuração de qualquer um desses listeners é feita com a tag `<listener>` como vimos acima. É possível inclusive que uma mesma classe implemente várias das interfaces de listeners mas seja configurada apenas uma vez o web.xml.

CAPÍTULO ANTERIOR:

Apêndice - Java EE 6

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter