caelum

APOSTILA JAVA PARA DESENVOLVIMENTO WEB

CAPÍTULO 18

# Apêndice - Java EE 6

*"Nove pessoas não fazem um bebê em 1 mês"*
*— Fred Brooks*

## 18.1 - JAVA EE 6 E AS NOVIDADES

O Java EE, desde seu lançamento, é considerado uma maneira de desenvolvermos aplicativos Java com suporte a escalabilidade, flexibilidade e segurança. Em sua última versão, a 6, um dos principais focos foi simplificar e facilitar a codificação de aplicativos por parte dos desenvolvedores.

Lançada oficialmente no dia 10 de dezembro de 2009, a nova especificação Java EE 6 foi liberada para download juntamente com o Glassfish v3, que é a sua implementação de referência. O Java EE 6 vem com mudanças significativas que foram em parte baseadas na grande comunidade de desenvolvedores Java, e que visam facilitar de verdade o uso das principais tecnologias do Java EE.

Um dos principais problemas da antiga especificação Java EE era que, na maioria das vezes que desenvolvíamos algum aplicativo, não era necessário fazer uso de todas as tecnologias disponíveis nela. Mesmo usando apenas parte dessas tecnologias, tínhamos que um lidar com todo o restante de tecnologias da especificação. Por exemplo, quando desenvolvemos Web Services, precisamos utilizar apenas as especificações JAX-WS e JAXB, mas precisamos lidar com todo o restante das especificações mesmo sem precisarmos delas.

Para resolver esse problema, no Java EE 6 foi introduzido o conceito de profiles. Um profile é uma configuração onde podemos criar um subconjunto de tecnologias presentes nas especificações Java EE 6 ou até mesmo adicionar novas tecnologias definidas pela JCP (Java Community Process) que não fazem parte da especificação. A própria especificação Java EE já incluiu um profile chamado **Web Profile**.

O Web Profile reúne apenas as tecnologias usadas pela maioria dos desenvolvedores Web.

Ela inclui as principais tecnologias vista durante este curso (Servlets, JSP, JSTL) e outras tecnologias como, por exemplo, JPA e JSF que podem ser vistas no curso FJ-26 da Caelum.

No Java EE 5, foram feitas mudanças que já facilitaram muito a vida do desenvolvedor, como, por exemplo, o uso de POJOs (Plain Old Java Object) e anotações e preterindo o uso de XML's como forma de configuração. Além da criação da JPA (Java Persistence API) para facilitar o mapeamento objeto relacional em nossas aplicações cuja principal implementação é o famoso Hibernate.

Seguindo a onda de melhorias, o Java EE 6 introduziu melhorias em praticamente todas as tecnologias envolvidas no desenvolvimento de aplicativos enterprise ou Web. Por exemplo:

- **DI (Dependency Injection)** - Usar anotações para criarmos classes que podem ser injetadas como dependência em outras classes;
- **JPA 2.0** - Melhorias na Java Persistence Query Language e a nova API de Criteria;
- **EJB 3.1** - Facilidades no desenvolvimento de Enterprise Java Beans usando a nova API de EJB 3.1;
- **Bean Validation** - Validar nossos POJOs de maneira fácil utilizando anotações;
- **JAX-RS** - Especificação sobre como criar Web Services de maneira RESTful.

## 18.2 - Processamento assíncrono

Muitas vezes em nossas aplicações temos alguns servlets que, para finalizarem a escrita da resposta ao cliente, dependem de recursos externos como, por exemplo, uma conexão JDBC, um Web Service ou uma mensagem JMS. Esses são exemplos de recurso que não temos controle quanto ao tempo de resposta.

Não tínhamos muita saída a não ser esperar pelo recurso para que nosso `Servlet` terminasse sua execução. Porém, essa solução nunca foi a mais apropriada, pois bloqueávamos uma thread que estava à espera de um recurso intermitente.

Na nova API de Servlets 3.0, esse problema foi resolvido. Podemos processar nossos servlets de maneira **assíncrona**, de modo que a thread não fica mais à espera de recursos. A thread é liberada para executar outras tarefas em nosso servidor de aplicação.

Quando o recurso que estávamos esperando se torna disponível, outra thread pode escrever a resposta e enviá-la ao cliente ou podemos, como foi visto durante o curso e vimos que é uma ótima prática, delegar a escrita da resposta para outras tecnologias como JSP.

Tanto servlets quanto filtros podem ser escritos de maneira assíncrona. Para isso, basta colocarmos o atributo `asyncSupported` como `true` nas anotações `@WebServlet` ou `@WebFilter`. Aqui vemos um exemplo de como fazer nosso servlet ter suporte à requisições assíncronas:

```
@WebServlet(asyncSupported=true, urlPatterns={"/adiciona/contato"})
public class AdicionaContatoServlet extends HttpServlet{
  public void doGet(HttpServletRequest req, HttpServletResponse res) {
    //Aguardando conexão JDBC
    ContatoDao dao = new ContatoDao();
      }
}
```

No exemplo acima, habilitamos suporte assíncrono em nossa servlet, porém ainda não tornamos o código realmente assíncrono. Na interface `javax.servlet.ServletRequest`, temos um método chamado `startAsync` que recebe como parâmetros o `ServletRequest` e o `ServletResponse` que recebemos em nosso servlet. Uma chamada a este método torna a requisição assíncrona. Agora sim tornamos nosso código assíncrono:

```
@WebServlet(asyncSupported=true, urlPatterns={"/adiciona/contato"})
public class AdicionaContatoServlet extends HttpServlet{
  public void doGet(HttpServletRequest req, HttpServletResponse res) {
    AsyncContext context = req.startAsync(req, res);

    //Aguardando conexão JDBC
    ContatoDao dao = new ContatoDao();
      }
}
```

Note que, ao fazer uma chamada ao método `startAsync`, é retornado um objeto do tipo `AsyncContext`. Esse objeto guarda os objetos `request` e `response` passados ao método `startAsync`. A partir dessa chamada de método, a thread que tratava nossa requisição foi liberada para executar outras tarefas. Na classe `AsyncContext`, temos o método `complete` para confirmar o envio da resposta ao cliente.

```
@WebServlet(asyncSupported=true, urlPatterns={"/adiciona/contato"})
public class AdicionaContatoServlet extends HttpServlet{
  public void doGet(HttpServletRequest req, HttpServletResponse res) {
    final AsyncContext context = req.startAsync(req, res);
    final PrintWriter out = res.getWriter();

    context.addListener(new AsyncListener() {
      @Override
      public void onComplete(AsyncEvent event) throws IOException {
        out.println("<html>");
        out.println("Olá mundo!");
        out.println("</html>");
      }
    });

    //Aguardando conexão JDBC
```

```
    ContatoDao dao = new ContatoDao();
    //Nossa logica

    //Lógica completa, chama listener onComplete
    context.complete();
      }
}
```

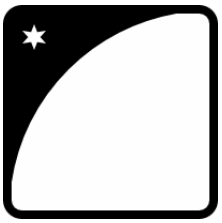
Também podemos delegar a escrita da resposta para um JSP por exemplo. Neste caso, faremos uso do método `dispatch`:

```
@WebServlet(asyncSupported=true, urlPatterns={"/adiciona/contato"})
public class AdicionaContatoServlet extends HttpServlet{
  public void doGet(HttpServletRequest req, HttpServletResponse res) {
    final AsyncContext context = req.startAsync(req, res);

    context.addListener(new AsyncListener() {
      @Override
      public void onComplete(AsyncEvent event) throws IOException {
        context.dispatch("/adicionado.jsp");
      }
    });

    //Aguardando conexão JDBC
    ContatoDao dao = new ContatoDao();
    //Nossa logica

    //Lógica completa, chama listener onComplete
    context.complete();
      }
}
```

**AsyncListener e seus outros métodos**

A classe `AsyncListener` suporta outros eventos. Quando a chamada assíncrona excedeu o tempo máximo que ela tinha para ser executada sobrescrevemos o método `onTimeout`. Para tratarmos um erro podemos sobrescrever o método `onError`.

Podemos prosseguir configurando nossa Servlets e Filtros no arquivo web.xml. Para configurarmos um servlet ou filtro como possivelmente assíncrono devemos especificar a tag `<async-supported>true</async-supported>` no mapeamento da servlet ou do filtro.



dos exercícios? Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o **curso FJ-21** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Java para Desenvolvimento Web*.

## 18.3 - Plugabilidade e Web fragments

Com a criação das novas anotações disponíveis na API Servlets 3.0, o uso do `web.xml` não é mais obrigatório. O web.xml é usado quando desejamos sobrescrever algumas configurações definidas em nossas anotações.

Percebemos que a grande ideia da nova API de Servlets é criar a menor quantidade de XML possível e colocar nossas configurações em anotações. Se pensarmos nos frameworks que utilizamos para nos auxiliar no desenvolvimento Web, por exemplo, o `VRaptor` ou o `Struts`, sabemos que ele exige que seja feita a configuração de um filtro no arquivo `web.xml` de nossa aplicação. Temos um fato que vai contra as premissas da nova API de Servlets, que é criar um arquivo web.xml quando o mesmo deveria ser opcional.

Com o intuito de resolver esse problema, foi introduzido o conceito de **web fragment**, que são módulos de um `web.xml`. Podemos ter vários `web fragment` que, em conjunto, podem ser visto como se fossem um `web.xml` completo. O framework que utilizamos pode criar seu próprio `web fragment` e colocar nele todas as configurações que teríamos que fazer manualmente em nosso `web.xml`. Por exemplo, no caso do `VRaptor` teríamos a configuração do filtro obrigatório do `VRaptor` dentro do próprio framework. Com isso, seríamos poupados de termos que colocar qualquer configuração em nossa aplicação. O próprio container conseguirá buscar por essas configurações e aplicá-las em nossa aplicação.

No `web fragment` podemos adicionar quase todos os elementos que estão disponíveis para o arquivo web.xml. As únicas restrições que devemos seguir são:

- Arquivo **deve** ser chamado de `web-fragment.xml`
- Root tag do arquivo **deve** ser `<web-fragment>`

No caso do `VRaptor` o arquivo `web-fragment.xml` ficaria assim:

```
<web-fragment>
  <filter>
    <filter-name>vraptor</filter-name>
```

```
    <filter-class>br.com.caelum.vraptor.VRaptor</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>vraptor</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
  </filter-mapping>
</web-fragment>
```

De preferência o framework que utilizamos devem colocar seus `web fragments` na pasta META-INF do arquivo `.jar`, que está normalmente localizado na pasta WEB-INF/lib da nossa aplicação.

O VRaptor 3.1 já possui essa configuração de Servlets 3.0 pronta.

> **Tag** `metadata-complete`
>
> Neste caso se a tag `metadata-complete` for setada como `true` o container não analisará qualquer `web fragment` presente em nossa aplicação, até mesmo os `web fragments` de algum framework que utilizamos.

Essa funcionalidade permite modularizarmos nosso `web.xml`. Podemos ter o tradicional arquivo descritor `web.xml`, ou podemos modularizá-lo em um ou mais `web fragment`.

Devido a essa possível modularização, pode ser importante a ordem em que esses fragmentos são processados. Por exemplo, a ordem em que estes fragmentos são processados pode alterar a ordem de execução dos filtros em nossa aplicação.

Servlets 3.0 nos permite configurar a ordem que estes fragmentos são processados de maneira absoluta ou relativa. Se quisermos configurar nossos fragmentos para serem processados em ordem absoluta, devemos colocar a tag `<absolute-ordering>` em nosso `web.xml`. Também podemos configurar uma ordem relativa colocando a tag `<ordering>` no arquivo `web-fragment.xml`.

Nossos fragmentos podem ter identificadores, nomes que os diferenciam. Sendo assim, se quisermos configurar uma ordem absoluta para nossos fragmentos, teríamos a seguinte configuração em nosso `web.xml`:

```
<web-app>
     <name>Minha Aplicação</name>
     <absolute-ordering>
        <name>Fragmento1</name>
        <name>Fragmento2</name>
```

```
    </absolute-ordering>
</web-app>
```

No código acima, eles seriam processados na seguinte ordem:

1. `web.xml` - Sempre será processado primeiro

2. Fragmento1

3. Fragmento2

A API de Servlets 3.0 tem uma nova interface chamada `ServletContainerInitializer` que nos permite plugar frameworks em nossa aplicação. Para isso, basta que o framework crie uma classe que implemente esta interface.

Por exemplo, na implementação da API JAX-WS (Web Services RESTful, que vemos no curso FJ-31) seria criado uma classe que implemente esta interface:

```
@HandlesTypes(WebService.class)
public class JAXWSServletContainerInitializer
    implements ServletContainerInitializer {
  public void onStartup(Set<Class<?>> c, ServletContext ctx)
      throws ServletException {
    ServletRegistration reg =
      ctx.addServlet("JAXWSServlet",
          "com.sun.webservice.JAXWSServlet");
    reg.addServletMapping("/jaxws");
  }
}
```

Os frameworks que implementam esta interface devem colocá-las na pasta META-INF/services em um arquivo chamado `javax.servlet.ServletContainerInitializer` que faça referência para a classe de implementação. Quando nosso container for iniciado, ele vai procurar por essas implementações quando ele for iniciado.

A anotação @HandlesTypes serve para especificar quais classes podem ser manipuladas pela implementação de `ServletContainerInitializer`

## 18.4 - Registro dinâmico de Servlets

Na nova API de Servlets 3.0 temos a possibilidade de adicionar um servlet em nossa aplicação de 3 maneiras. Duas delas já foram vistas durante este capítulo (XML e anotações). A terceira delas pode ser feita de uma maneira programática ou dinâmica.

Dentro do Java EE podemos classificar os objetos principais que habitam nossa aplicação em 3 escopos. O escopo de requisição que tem uma duração curta, o escopo de sessão que é

muito utilizado quando queremos guardar informações referentes a algum usuário específico e o terceiro deles que não abordamos durante o curso que é o escopo de aplicação.

São objetos que permanecem em memória desde o momento que nossa aplicação é iniciada até o momento que a mesma é finalizada. Os servidores de aplicação nos fornecem um objeto que é uma implementação da interface `ServletContext` que é um objeto de escopo de aplicação.

Com esse objeto podemos fazer coisas relativas a aplicação, como por exemplo, fazer logs de acesso, criar atributos que podem ser compartilhados por toda a aplicação, etc.

A interface `ServletContext` nos fornece alguns métodos adicionais que nos permitem fazer o registro dinâmico de servlets e filtros. A única restrição para chamarmos esses métodos é que devemos fazer isso no momento em que nossa aplicação está sendo iniciada em nosso servidor de aplicação.

Podemos fazer isso de duas maneiras distintas. A primeira delas e a mais conhecida, seria criarmos um listener do tipo `ServletContextListener` e adicionarmos o servlet no método `contextInitialized` como no código abaixo:

```
public class ServletListener implements ServletContextListener {
  public void contextInitialized(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    context.addServlet("MeuServlet", MeuServlet.class);
  }

  public void contextDestroyed(ServletContextEvent event) {

  }
}
```

Para adicionarmos um filtro, faríamos algo bem similar: só mudaríamos a chamada de `addServlet` para `addFilter` e passaríamos como parâmetro o filtro que desejamos adicionar.

A outra maneira de fazermos isso, seria criarmos uma implementação de `ServletContainerInitializer` e no método `onStartup`. Por exemplo:

```
public class MeuServletContainerInitializer
    implements ServletContainerInitializer {
  public void onStartup(Set<Class<?>> c,
      ServletContext context) throws ServletException {
    ServletRegistration reg = context
        .addServlet("MeuServlet", MeuServlet.class);
    reg.addServletMapping("/jaxws");
  }
}
```

Este segundo exemplo seria mais usado se fossemos distribuir nossa aplicação como um framework e nos beneficiarmos da plugabilidade como foi visto na seção anterior.

CAPÍTULO ANTERIOR:

Apêndice - VRaptor3 e produtividade na Web

PRÓXIMO CAPÍTULO:

Apêndice - Tópicos da Servlet API

Casa do Código

Twitter