Session Beans

"O caminho do inferno está pavimentado de boas intenções." – Marx.

Finalmente implementando um EJB

A especificação da Sun

Vamos agora utilizar a tecnologia especificada pela sun, chamada **Enterprise Java Beans**, para transformar o esboço do capítulo anterior em classes que sigam essa especificação.

A classe do EJB

Vamos começar pela classe concreta, que contém a nossa lógica de negócios:

```
package livraria;

import java.util.*;
import javax.ejb.*;

public class CarrinhoBean implements SessionBean {
	private List livros;
	private double total;

	public void ejbCreate() throws CreateException{
		this.livros = new ArrayList();
		this.total = 0;
	}

	public void addLivro(Livro livro) {
		this.livros.add(livro);
		this.total += livro.getPreco();
	}

	public List getLivros() {
		return this.livros;
	}

	public double getTotal() {
		return this.total;
	}

	public void ejbActivate() {}

	public void ejbPassivate() {}

	public void ejbRemove() {}

	public void setSessionContext(SessionContext sessionContext) {}
}
```

O nosso método reciclador é o **ejbCreate**!

Alguns métodos temos de escrever pois estão definidos na interface SessionBean. São os 4 últimos métodos. São métodos de **callback**: o servidor te avisa quando determinado evento ocorre.

A interface remota

A interface remota é muito simples:

```
package livraria;

import java.util.List;
import java.rmi.RemoteException;

import javax.ejb.EJBObject;

public interface Carrinho extends EJBObject {
```

```
        void addLivro(Livro livro) throws RemoteException;
        List getLivros() throws RemoteException;
        double getTotal() throws RemoteException;
}
```

Somos obrigados a declarar que esta interface lança **RemoteException** em todos os objetos, pois o EJB é acessado remotamente, e um objeto **RMI** será criado em cima dessa interface.

Interfaces locais

Um EJB pode ser acessado remotamente, o que nos traz muitas vantagens. Mas existem ocasiões em que não temos essa necessidade, e ficar acessando um objeto local de maneira remota (isto é, abrindo sockets, etc...) é muito estressante para o servidor.

Existe a possibilidade de criar **interfaces locais** (Local interface) para um EJB. O processo é muito similar a este, mas a nossa superinterface seria outra, assim como o arquivo de configuração mudaria um pouco.

A interface da fábrica

Precisamos definir a interface pela qual o cliente vai criar um objeto do tipo Carrinho. O servidor ira implementar essa interface, e não necessariamente estará criando um novo objeto a cada requisição, como visto no capítulo anterior.

```
package livraria;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface CarrinhoHome extends EJBHome {
        Carrinho create() throws RemoteException, CreateException;
}
```

Recebendo argumentos no momento da criação

O seu método **create** poderia estar recebendo argumentos, desde que isso se reflita no **ejbCreate** do seu bean. Você pode ter mais de um método de fabricação.

Classes auxiliares

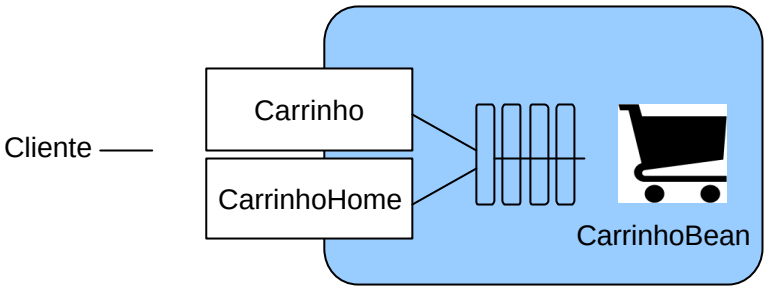
Além do nosso EJB, precisamos das classes referidas por ele que não fazem parte do J2SE ou EE. O nosso **Livro**, por exemplo.

```
package livraria;

public class Livro implements Serializable {

        private double preco;
        private String nome;

        public Livro(String nome, double preco) {
                this.nome = nome;
                this.preco = preco;
        }

        public String getNome() {
                        return this.nome;
        }

        public double getPreco() {
                        return this.preco;
        }
}
```

Como **Livro** não é um EJB, sempre será passado uma cópia dele. Por isso ele tem de implementar **Serializable**.

Unindo tudo isso

Por enquanto, temos o seguinte quadro:



Além das classes, precisamos de que em algum lugar seja definido quem é quem. Precisamos amarrar essas duas interfaces e a classe do EJB. Isso vai em um arquivo xml, que ficará dentro do diretório META-INF. É o ejb-jar.xml.

```
<ejb-jar>
   <description>Exemplo de Session Bean</description>
   <display-name>JCE session bean</display-name>
   <enterprise-beans>
          <session>
                    <ejb-name>CarrinhoEJB</ejb-name>
                    <home>livraria.CarrinhoHome</home>
                    <remote>livraria.Carrinho</remote>
                    <ejb-class>livraria.CarrinhoBean</ejb-class>
                    <session-type>Stateful</session-type>
                    <transaction-type>Container</transaction-type>
          </session>
   </enterprise-beans>
</ejb-jar>
```

Repare que estamos dando um nome ao nosso EJB: **CarrinhoEJB**. Depois citamos as duas interfaces e a classe que é o **Bean** de verdade. **session-type** é um atributo importante que será explicado mais adiante.

Todo servidor de aplicação é obrigado a entender este xml. Existe uma grande quantidade de configurações que poderiam ser definidas aqui.

Algumas outras configurações que não são especificadas pelo J2EE, podem ser configuradas no servidor de aplicação. Essas configurações são totalmente opcionais e variam de servidor para servidor, e não são compatíveis. Normalmente elas vão em outro xml de configuração dentro do META-INF, que só determinador servidor sabe entender.

Teremos então dois subdiretórios, o livraria, com todos os .class, e o META-INF, com esse XML. Crie um **carrinhoEJB.jar** com esses dois diretórios. Utilize a ferramenta **jar** par tal.

Ant

A ferramente Ant pode ser usada para gerar esse .jar. Procure sobre o Ant no JCE-U.

O Servidor de aplicação

Precisamos ter um servidor de aplicação para colocar o nosso EJB. Poderíamos usar a Reference Implementatio que vem no J2EE, porém ela é de difícil manuseio. O **JBoss** é um servidor de aplicação gratuito, sob licença LGPL.

Cada servidor de aplicação tem suas características próprias no momento de configurarmos e colocarmos o EJB dentro de seu **container**.

Instalando o JBoss

Você primeiramente ira baixar o JBoss no site http://www.jboss.org, e para a instalação basta descompactar o seu conteúdo. Pegue a última versão da série 3.2.x, que implementa a especificação EJB 2.0, que é uma das especificações do J2EE 1.3.

Você precisa ter duas variáveis de ambiente configuradas: a JBOSS_HOME, com o diretório que você descompactou o JBoss, e o JAVA_HOME, para o diretório de instalação do seu J2SDK (não o EE, o SE).

O deploy

Primeiramente rode o JBoss. Para isso, execute o run.bat (ou .sh, dependendo do seu sistema operacional), que encontra-se no diretório bin do JBoss. Uma mensagem parecida com essa deverá aparecer, indicando que o servidor está incializado.

```
[Server] JBoss (MX MicroKernel) [3.2.3RC1 (build: CVSTag=JBoss_3_2_3_RC1 date=200311101720)] Started
in 21s:711ms
```

Portas e configurações

O JBoss inicializa uma grande quantidade de serviços. Se alguma das portas que um desses serviços for utilizar não estiver desocupada, o JBoss inicializará mas com uma série de warnings e exceptions lançadas.

Isto acontece, por exemplo, quando você já tem um Tomcat ouvindo na porta 8080. O JBoss já vem com um servidor web, que escuta nessa porta. Você pode configurar essas portas e inúmeras outras opções nos xmls dentro do diretório server/default/conf.

Agora que você tem o jar do seu EJB, você pode colocar esse jar dentro do diretório server/default/deploy. Em poucos segundos o JBoss perceberá a presença de um novo jar, e fará o **deploy**. Deploy é o momento em que você coloca seus EJBs para o servidor poder oferecer esse novo serviço. Nesse momento o servidor é responsável por fazer uma série de checagens, como validar seu ejb-jar.xml, verificando se não há nenhum problema.

Na janela em que o JBoss abriu seu console, verifique as mensagens emitidas. Uma mensagem de sucesso é algo como:

```
[MainDeployer] Starting deployment of package: file:/D:/javaprograms/jboss-
       3.2.3RC1/server/default/deploy/carrinhoEJB.jar
[EjbModule] Deploying CarrinhoEJB
[StatefulSessionInstancePool] Started jboss.j2ee:jndiName=CarrinhoEJB,plugin=pool,service=EJB
[StatefulSessionFilePersistenceManager] Started null
[StatefulSessionContainer] Started jboss.j2ee:jndiName=CarrinhoEJB,service=EJB
[EjbModule] Started jboss.j2ee:module=carrinhoEJB.jar,service=EjbModule
[EJBDeployer] Deployed: file:/D:/javaprograms/jboss-3.2.3RC1/server/default/deploy/carrinhoEJB.jar
[MainDeployer] Deployed package: file:/D:/javaprograms/jboss-
       3.2.3RC1/server/default/deploy/carrinhoEJB.jar
```

Criando um cliente

Agora precisamos rodar uma aplicação java que utilize-se da **CarrinhoHome** para pegar um **Carrinho** e operá-lo:

```
import javax.naming.InitialContext;
import livraria.*;

public class Cliente {
       public static void main(String[] args) throws Exception {
                     InitialContext ic = new InitialContext();
```

```
                        CarrinhoHome home = (CarrinhoHome) ic.lookup("CarrinhoEJB");

                        Carrinho carrinho = home.create();

                        Livro livro1 = new Livro("Crime e Castigo", 45);
                        Livro livro2 = new Livro("O idiota", 30);

                        carrinho.addLivro(livro1);
                        carrinho.addLivro(livro2);

                        System.out.println(carrinho.getTotal());
        }
}
```

O **InitialContext** é o responsável por fazer o lookup do objeto, assim como o **Naming** fazia para o RMI. O **InitialContext** faz parte de um serviço de nomes mais robusto que o **Naming**, chamado JNDI e que faz parte da especificação J2EE.

Onde o JNDI irá se conectar para pegar o "CarrinhoEJB"? Precisamos falar isso para ele. Definimos então um arquivo jndi.properties, com a seguinte informação:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

No `java.naming.provider.url` você pode definir qual o ip do jboss.

Rodando o cliente

Basta executarmos a classe Cliente. É necessário que o **j2ee.jar** esteja no **classpath**, assim como o **jbossall-client.jar** (vem com o jboss), que é responsável por gerar os stubs a medida do necessário, assim como outras funcionalidades.

Problemas no começo

É muito comum a nossa primeira aplicação com EJB não funcionar. Fique atento as mensagens de erros que o JBoss lança no momento do deploy.

Stateless e Stateful session beans

Definimos no nosso **ejb-jar.xml** que nosso session bean será **Stateful**. Isso da a garantia que, depois de ter chamado o create no **CarrinhoHome**, o **Carrinho** que temos como referência será sempre o mesmo.

Em outras palavras, sempre que adicionarmos um novo **Livro** nesse carrinho, os outros Livros estarão lá. E era exatamente esse o nosso objetivo.

Em vez de **Stateful** (que mantém o estado), um session bean pode ser **Stateless**. Nesse caso, apesar de você ter uma referência para um EJB, ninguém garante, entre uma chamada de método e outra, que o mesmo EJB ira atende-lo. O **SessionBean Stateless** tem uma característica de **serviço**, já que a cada momento ele esta atendendo requisição de uma pessoa diferente.

Um **SessionBean Stateless** é muito mais leve para o servidor de aplicação, já que ele não precisa ficar isolando aquele EJB dos outros clientes, e o seu pool pode ser bem pequeno. É comum que um **Stateless** não tenha atributos, mas nada impede que ele tenha, o que não funciona é guardar alguma coisa de um cliente nesses atributos. Você pode pensar que o efeito é o mesmo de uma **Servlet**: em cada momento a **Servlet** pode estar atendendo um cliente diferente. Mas no EJB você não precisa se preocupar com o acesso multi threaded.

Exercícios

1-) Crie o componente **CarrinhoEJB** e teste-o.

2-) Crie outra maneira para que um **Carrinho** seja fabricado. Isto é, adicione um create na **CarrinhoHome** que receba uma **String** com o nome do dono do **Carrinho**, reflita isso no **ejbCreate** do **CarrinhoBean**.

3-) Crie um **SessionBean Stateless** que é responsável por gerenciar os **Livros**. Um Cliente deve sempre pesquisar nesse **GerenciadorDeLivros** antes de compra-lo.

Sobre o curso

A **Caelum** (http://www.caelum.com.br) oferece os cursos e a apostila "*Falando em Java*", que aborda o ensino dessa linguagem e tecnologia de forma mais simples e prática do que em outros cursos, poupando o aluno de assuntos que não são de seu interesse em determinadas fases do seu aprendizado.

As apostilas "*Falando em Java*" estão disponíveis para download no site http://www.caelum.com.br/fj.jsp. Para mais informações sobre as apostilas, os cursos e os direitos de uso, assim como a licença de uso, leia as informações do site.

Se você possui alguma colaboração, como correção de erros, sugestões, novos exercícios e outros, envie-nos um email!

Sobre os autores

Guilherme Silveira (*guilherme@guj.com.br*, http://www.caelum.com.br) é programador e web developer Java certificado pela Sun e trabalha com essa tecnologia desde 2000. Ensinou Java e Tibco na Alemanha onde programou e arquiteturou pequenos e grandes projetos. Cofundador do GUJ, escreve para a revista Mundo Java e o site GUJ. Estuda Matemática Aplicada na USP e ministra aulas periodicamente em São Paulo através da Caelum.

Paulo Silveira (*paulo@paulo.com.br*, http://www.caelum.com.br/) é programador e desenvolvedor certificado Java. Possui grande experiência com servlets, que utilizou na Alemanha, e vários outros projetos Java, onde trabalhou como consultor sênior. É instrutor Java pela Sun, cofundador do GUJ e criador do framework vRaptor. É formado em ciência da computação pela USP, onde realiza seu mestrado.