

1. Arquitetura distribuída
2. Padrões atuais na indústria
3. O padrão EJB: passado, presente e futuro
4. Servidores de objetos proprietários com Java
5. Protocolo RMI e servidor RMI com Java
6. Vantagens corporativas do uso de EJB
7. Tipos de Enterprise Java Beans
8. Serviço JNDI e configurações de recursos no JBOSS
9. Session Bean Stateless
10. Session Bean Stateful
11. Entity Bean CMP
12. Entity Bean BMP
13. Message Driven Bean e JMS
14. Transações
15. Segurança e Deployment Descriptor

Enterprise Java Beans

Enterprise JavaBeans é uma arquitetura padrão de componentização para objetos distribuídos. Objetos distribuídos são aqueles que são instalados em um servidor e podem ser acessados remotamente.

A base da implementação de Enterprise Java Beans é o protocolo RMI que especifica mecanismos de transporte de informação e de chamada a métodos remotos na rede. RMI também é responsável pelas operações de Marshall e Unmarshall (encapsulamento de argumentos de entrada e saída de um método).

EJB e o interesse corporativo

Com o padrão Enterprise Java Beans empresas conseguem desenvolver softwares com alta escalabilidade, confiabilidade e disponibilidade sem gastar tempo com programação de infra-estrutura básica (sistemas de cluster, segurança, transação, persistência e outros).

Porque utilizar EJB?

- Especificação separada da implementação
- Interoperabilidade
- Desenvolvedores mais focados na lógica de negócio
- Compatibilidade com CORBA / IIOP

O que é um Enterprise Bean?

Enterprise Beans são componentes que fazem parte de uma aplicação distribuída e orientada a transações. Algumas características típicas de EJBs são:

- O ciclo de vida destes componentes é dependente do Container.
- Contém lógica de negócios
- As instâncias são criadas e mantidas pelo container
- Podem ser customizadas em tempo de deploy através do deployment descriptor.

- Segurança e gerenciamento de transação e outros serviços como estes não são responsabilidades do EJB.
- Um cliente nunca acessa um EJB diretamente

Os três tipos de Enterprise Java Beans encontrados na especificação EJB 2.0 são:

- Entity Bean
- Session Bean
- Message-driven bean


Message-driven Beans

São os mais novos tipos de EJBs, eles foram criados na especificação EJB2.0. Este tipo de EJB foi criado para trabalhar com mensagens assíncronas, utilizando Java Message Service (JMS).

Message-driven Beans não são expostos diretamente aos clientes. Um message-driven bean recebe as mensagens enviadas utilizando JMS e processa estas mensagens anonimamente.

Quando o container recebe uma mensagem ele entrega a mensagem para uma instância de Message-driven Beans já criada, ou se necessário cria uma nova instância. Ou seja, message-driven beans são stateless.

Quais são os "concorrentes" do EJB?

	Consórcio / Empresa
	Java Community Process (www.jcp.org)
	Protocolo utilizado
	RMI - Remote Method Invocation
	Vantagens
	<ul style="list-style-type: none">• Padrão aberto, com opções de servidores pagos e gratuitos.• Muitas opções de ambientes de desenvolvimento.• Alta escalabilidade.
	Desvantagens
	<ul style="list-style-type: none">• Falta de recursos.• Custo de aquisição de conhecimento um pouco alto.

Corba	Consórcio / Empresa
	Object Management Group (www.omg.org)
	Protocolo utilizado
	IIOP
	Vantagens
	<ul style="list-style-type: none"> • Grande número de componentes de negócio • Independência de linguagem de desenvolvimento
	Desvantagens
	<ul style="list-style-type: none"> • Alto custo de mão de obra • Alto TCO (Total Cost of Ownership)

COM / DCOM	Consórcio / Empresa
	Microsoft
	Protocolo utilizado
	Remote Procedure Call
	Vantagens
	<ul style="list-style-type: none"> • Desenvolvimento e deploy simplificados com a utilização do Visual Basic. • Facilidade de aprendizado.
	Desvantagens
	<ul style="list-style-type: none"> • Para componentes mais robustos e escaláveis é necessário programarmos em Visual C++ que apresenta um TCO (Total Cost of Ownership) muito grande para as empresas. • Padrão proprietário e aparentemente descontinuado na nova plataforma Microsoft.net

Servidores de Objetos Customizados



Exemplo: Protocolo.java

```
1  import java.util.*;
2  import java.io.*;
3
4  public class Protocolo implements Serializable{
5
6      public static final int BUSCA = 0;
7      public static final int GUARDA = 1;
8      public static final int REMOVE = 2;
9
10     int operacao;
11     String parametro = null;
12     Object resultado = null;
13
14     /***** Contrutores *****/
15     public Protocolo(int operacao, String parametro){
16         this.operacao = operacao ;
17         this.parametro= parametro;
18     }
19
20     public Protocolo(int operacao){
21         this.operacao = operacao;
22     }
23 }
```

```
24  /***** Getters & Setters *****/
25  public int getOperacao(){
26      return this.operacao;
27  }
28
29  public void setOperacao( int operacao){
30      this.operacao = operacao;
31  }
32
33  public String getParametro(){
34      return this.parametro ;
35  }
36
37  public void setParametro(String parametro){
38      this.parametro = parametro;
39  }
40
41  public Object getResultado(){
42      return this.resultado;
43  }
44
45  public void setResultado(Object resultado){
46      this.resultado= resultado;
47  }
48  }
```

Exemplo: Cliente.java

```
1  import java.io.*;
2
3  public class Cliente implements Serializable{
4      String nome;
5      public Cliente(String nome){
6          this.nome = nome;
7      }
8
9      public String toString(){
10         return nome;
11     }
12 }
```

Exemplo: Server.java

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.*;
```

```
4
5 public class Server{
6
7     public static void main(String args[]) throws Exception{
8
9         ObjectOutputStream objectWriter;
10        ObjectInputStream objectReader;
11
12        ServerSocket server = new ServerSocket(7777);
13
14        System.out.println("SERVER SOCKET INCICIADO, PORTA :7777\n");
15
16        while (true){
17
18            Socket s = server.accept();
19
20            System.out.print("CONEXAO ACEITA: ");
21
22            objectReader = new ObjectInputStream(s.getInputStream());
23
24            Protocolo protocolo = (Protocolo)objectReader.readObject();
25
26            int operacao = protocolo.getOperacao();
27
28            objectWriter = new ObjectOutputStream(s.getOutputStream());
29
30            if (operacao==protocolo.BUSCA){
31                System.out.println("Realizando busca");
32
33                // Criando um novo objeto Cliente com o nome recebido
34                Cliente c = new Cliente(protocolo.getParametro());
35
36                // serializando o objeto para enviar para o cliente
37                protocolo.setResultado(c);
38
39            }
40            if (operacao==protocolo.GUARDA){
41                System.out.println("Realizando busca");
42            }
43            if (operacao==protocolo.REMOVE){
44                System.out.println("Removendo objeto");
45            }
46
47            objectWriter.writeObject(protocolo);
48
49            objectWriter.flush();
50
51            s.close();
52        }
53    }
```

Exemplo: ClientSocket.java

R. Alexandre Dumas, 1268 cj. 134 – Chac. Sto. Antonio – São Paulo – SP
55 11 5183 2999 – www.globalcode.com.br


```
1 import java.net.*;
2 import java.io.*;
3 import java.util.*;
4
5 public class ClientSocket{
6
7     public static void main(String args[]) throws Exception{
8
9         if (args.length < 1) {
10             System.out.println("uso ClientSocket <parametro busca>");
11             System.exit(1);
12         }
13         String parametro = args[0];
14
15         InputStreamReader isr = new InputStreamReader(System.in);
16         BufferedReader leitorTeclado = new BufferedReader (isr);
17
18         Socket s=new Socket("localhost",7777);
19
20         System.out.println("CONECTANDO A PORTA 7777");
21
22         // obtendo o canal de envio de dados para o Server socket
23         ObjectOutputStream objectOutput;
24         objectOutput = new ObjectOutputStream(s.getOutputStream());
25
26         // Criando um objeto da classe protocolo, indicando
27         // operação de busca com parâmetro especificado
28         Protocolo p = new Protocolo(Protocolo.BUSCA, parametro);
29         objectOutput.writeObject(p);
30         objectOutput.flush();
31
32         // obtendo o canal de leitura com o server socket
33         ObjectInputStream objectInput;
34         objectInput = new ObjectInputStream(s.getInputStream());
35
36         // recebendo a resposta através do objeto Protocolo
37         Protocolo resposta = (Protocolo)objectInput.readObject();
38         Cliente cliente = (Cliente)resposta.getResultado();
39
40         System.out.println("Cliente : " + cliente);
41         System.out.println("hora : " + new Date() );
42
43         s.close();
44     }
45 }
```

RMI

Um dos aspectos mais importantes da arquitetura EJB é justamente ser uma arquitetura distribuída. Isto quer dizer que nem todos os objetos serão executados na mesma JVM. Isto é possível através de **Remote Method Invocation (RMI)**.

RMI foi utilizado em diversas outras tecnologias distribuídas, como CORBA e DCOM. Existe uma versão específica de RMI para o Java, chamado Java RMI, através deste protocolo objetos Java podem trocar mensagens de maneira transparente, mesmo estando em diferentes JVM.

Para que possamos chamar métodos de objetos em difrentes JVM de maneira transparência o protocolo JVM utiliza dois componentes chamados *stub* e *skeleton*.

O que são Stubs e Skeletons?

Para que um objeto possa fazer uma chamada um método de um objeto que está em outra JVM ele precisa primeiro localizar este objeto e então fazer a chamada passando os parâmetros necessários via rede.

Já o objeto que recebe a chamada precisa conseguir receber a chamada a métodos e de alguma maneira tratar da segurança e sincronização, bem como garantir que as respostas sejam enviadas para os clientes certos.

O stub é um remote Proxy, responsável por processar as chamadas remotas a métodos. Ou seja, ele deixa transparente o fato de o cliente estar se comunicando com um objeto remoto.

O stub funciona como um Proxy para o RMI da seguinte forma:

1. Faz a conexão com o objeto em outra JVM.
2. Faz o marshal (escrita e transmissão) dos parâmetros necessários para a outra JVM.
3. Espera o resultado da invocação ao método.
4. Unmarshal (leitura) dos parâmetros ou exceptions retornados.
5. Retorna o valor ou exceção para o objeto (cleinte) que fez a chamada ao método.

O Stub disponibiliza os mesmos métodos (com as mesmas assinaturas) apresentados no objeto que deverá ser chamado remotamente.

Todos os objetos remotos devem ter um Skeleton correspondente. O skeleton é responsável por redirecionar todas as chamadas recebidas para a instância remota correta. Normalmente temos um skeleton para todos os objetos remotos.

Quando o skeleton recebe uma chamada a um método, ele faz o seguinte:

1. Unmarshal (leitura) dos parâmetros recebidos
2. Faz a chamada ao método
3. Marshal (escrita e transmissão) do resultado ou exception para o objeto que chamou o método.

Para todos os enterprise beans que são disponibilizados para objetos remotos um stub e possivelmente um skeleton são criados. No entanto, a forma como isto acontece depende do container utilizado.

Não é necessário criarmos as classes stub e skeleton pois elas são criadas pelo compilador RMI.

Hello World RMI

- 1 - Desenvolver uma interface que defina o objeto remoto (remote interface);

Esta interface é o protocolo de comunicação de comportamento entre o cliente-servidor.

Exemplo: Database.java

```
1  /*
2   * 1. REMOTE INTERFACE
3   */
4
5  import java.rmi.Remote;
6  import java.rmi.RemoteException;
7
8  public interface Database extends Remote{
9
10     public String getCliente(int clientId) throws RemoteException;
11
12 }
```

- 2 - Criar a implementação da classe (o servant) para a interface definida no item 1;

Esta classe tem a implementação da interface, ou seja, é o "recheio" do objeto remoto.

Exemplo: DatabaseImpl.java

```
1  /*
2   * 2. SERVANT
3   */
4
5  import java.rmi.RemoteException;
6  import java.rmi.server.UnicastRemoteObject;
7
8  public class DatabaseImpl extends UnicastRemoteObject
9                          implements Database{
10
11      public DatabaseImpl() throws RemoteException{
12      }
13
14      public String getCliente(int clientId){
15
16          return "\n Objeto String representando o cliente " + clientId;
17          //poderia ser qualquer objeto Serializable
18      }
19  }
```

3 - Criar um servidor do qual gerencie as instâncias dos servidores (servant);

O servidor deve pré-instanciar os objetos a registrá-los no serviço RMI de catálogos de objetos.

Exemplo: DatabaseServer.java

```
1  /*
2   * 3. Servidor de Objetos
3   */
4
5  import java.rmi.Naming;
6
7  public class DatabaseServer{
8
9      public static void main (String args[]) throws Exception{
10
11          // Criando o servidor
12          // instância para registro
13          DatabaseImpl dbRef = new DatabaseImpl();
14
15          // Preparando o objeto para o registro rmi
16          Naming.rebind ("DB", dbRef);
17          System.out.println(" Objeto Database pronto e");
18          System.out.println(" registrado no naming service como DB");
19      }
20  }
```

4 - Criar um client que utilize objetos remotos;

O cliente utiliza o serviço TCP/IP de catálogo de objetos para obter a referência remota ao objeto.

Exemplo: DatabaseCliente.java

```
1  /*
2   * 4. Cliente
3   */
4
5  import java.rmi.Naming;
6
7  public class DatabaseClient{
8      public static void main(String args[]) throws Exception{
9
10         //Checando argumento do contador
11
12         if(args.length != 2){
13             System.out.println("Usage: DatabaseClient <server><seu
14
15             nome>");
16         }
17         else{
18             String url = new String ("rmi: //" + args[0] + "/DB");
19             Database db = (Database) Naming.lookup(url);
20
21             String cliente = db.getCliente(Integer.parseInt(args[1]));
22             System.out.println (cliente);
23         }
24     }
25 }
```

Última Revisão: 1/10/2002 Data Criação:9/8/2002 Revisão 1

5 - Usar o compilador Java para criar classes de arquivos;

Processo de compilação padrão com javac.

```

C:\teste\rmi>javac *.java

C:\teste\rmi>dir
O volume na unidade C é Disco local
O número de série do volume é 647B-C535

Pasta de C:\teste\rmi
13/02/2003  17:17      <DIR>          .
13/02/2003  17:17      <DIR>          ..
13/02/2003  17:17                218 Database.class
13/02/2003  16:51                212 Database.java
13/02/2003  17:17            1.006 DatabaseClient.class
13/02/2003  17:13            534 DatabaseClient.java
13/02/2003  17:17            614 DatabaseImpl.class
13/02/2003  16:51            441 DatabaseImpl.java
13/02/2003  17:17            684 DatabaseServer.class
13/02/2003  17:06            479 DatabaseServer.java
                8 arquivo(s)          4.188 bytes
                2 pasta(s) 1.654.292.480 bytes disponíveis

```

6 - Rodar o compilador RMI (rmic) para criar as classes de apoio (stubs e skeletons);

rmic é um compilador especial responsável por gerar os Stubs e Skeletons, classes do Presentation layer do ISO-OSI, responsável por trabalhar argumentos de entrada e saída de um objeto.

```

C:\teste\rmi>rmic DatabaseImpl

C:\teste\rmi>dir
O volume na unidade C é Disco local
O número de série do volume é 647B-C535

Pasta de C:\teste\rmi
13/02/2003  17:17      <DIR>          .
13/02/2003  17:17      <DIR>          ..
13/02/2003  17:17                218 Database.class
13/02/2003  16:51                212 Database.java
13/02/2003  17:17            1.006 DatabaseClient.class
13/02/2003  17:13            534 DatabaseClient.java
13/02/2003  17:17            614 DatabaseImpl.class
13/02/2003  16:51            441 DatabaseImpl.java
13/02/2003  17:17            1.644 DatabaseImpl_Skel.class
13/02/2003  17:17            3.189 DatabaseImpl_Stub.class
13/02/2003  17:17            684 DatabaseServer.class
13/02/2003  17:06            479 DatabaseServer.java

```

Repare que as classes DatabaseImpl_Skel e DatabaseImpl_Stub foram criadas.

Para executar a aplicação:

1. Inicie o serviço de catálogo de objetos: rmiregistry.
2. Execute a classe DatabaseServer
3. Execute o programa DatabaseClient

JNDI

Diversos serviços são oferecidos aos EJBs pelo seu container, entre eles: persistência, pool de objetos, transação, acessibilidade via IIOP e outros mais.

Todos os recursos consumidos pelos componentes (ejb, servlet e outros) podem ser gerenciados através de uma API chamada JNDI.

JNDI é uma API utilizada para acessar diversos recursos oferecidos por um servidor de aplicação. Podemos obter via JNDI uma conexão de um pooling, obter uma variável de configuração da aplicação, ou ainda obter uma referência para um EJB.

Servlets também podem utilizar JNDI para acessar recursos de um servidor de aplicação.

Exemplo: Código Servlet utilizando JNDI Service

```
1  import java.io.IOException;
2  import java.io.PrintWriter;
3  import java.util.Enumeration;
4  import javax.servlet.ServletException;
5  import javax.servlet.http.HttpServlet;
6  import javax.servlet.http.HttpServletRequest;
7  import javax.servlet.http.HttpServletResponse;
8  import javax.naming.*;
9  import javax.ejb.*;
10 import java.rmi.RemoteException;
11 import javax.rmi.PortableRemoteObject;
12 import javax.sql.*;
13 import java.sql.*;
14
15 public class UsarJDBC extends HttpServlet {
16     public void doGet(HttpServletRequest request,
17                       HttpServletResponse response)
18         throws IOException, ServletException
19     {
20         response.setContentType("text/html");
21         PrintWriter writer = response.getWriter();
22         writer.println("<html>");
23         writer.println("<head>");
24         writer.println("</head>");
25         writer.println("<body bgcolor=white>");
26
27         try {
28             Context initial = new InitialContext ();
29             DataSource ds;
30             ds=(DataSource)initial.lookup("java:comp/env/db/J2EEDS");
31             Connection con = ds.getConnection();
32             writer.println("Conexão obtida com sucesso, devolvendo..");
33             con.close();
34         }
35         catch(Exception e) {
36             writer.println("<p>Erro ao acessar pool jdbc</p>");
37             writer.println("<p>" + e.getMessage() + "</p>");
38         }
39         writer.println("</body>");
40         writer.println("</html>");
41     }
42 }
```


web.xml

Deployment descriptor

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
  2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <display-name>CursoJ2EE</display-name>
  <description>Aplicacao para Labs </description>

  <servlet>
    <servlet-name>UsarJDBC</servlet-name>
    <servlet-class>UsarJDBC</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>UsarJDBC</servlet-name>
    <url-pattern>/UsarJDBC</url-pattern>
  </servlet-mapping>

  <resource-ref>
    <res-ref-name>db/J2EEDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>

</web-app>
```

jboss-web.xmlDeployment Descriptor
de Assembling

```
<?xml version="1.0" encoding="UTF-8"?>

<jboss-web>

  <resource-ref>

    <res-ref-name>db/J2EEDS</res-ref-name>
    <jndi-name></jndi-name>
    <default-resource-principal></default-resource-principal>

  </resource-ref>

</jboss-web>
```

Configurações Tomcat

1- Criar um novo context

Para adicionarmos um context (nova aplicação Web) no Tomcat fora do diretório "webapps" é necessário configurarmos tal aplicação no web.xml:

```
<Context path="/uri" docBase="c:/seuprojeto" debug="0" reloadable="true"/>
```

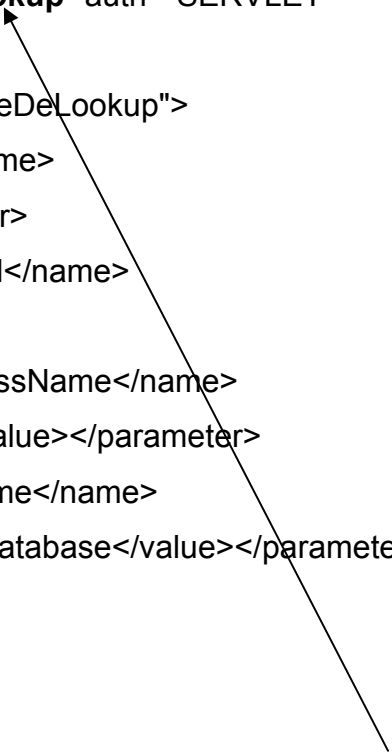
Onde reloadable indica se o Tomcat deve recarregar classes Java quando alteradas.

2- Configurando um pool de conexão

Podemos também configurar um pool de conexões JDBC no serviço JNDI do Tomcat, para isso coloque o seguinte TAG XML dentro do Context como indicado abaixo:

```
<Context path="/suaURI" docBase="c:/seuprojeto" debug="0" reloadable="true">
```

```
<Resource name="nomeDeLookup" auth="SERVLET" type="javax.sql.DataSource"/>
<ResourceParams name="nomeDeLookup">
  <parameter><name>user</name>
  <value>sa</value></parameter>
  <parameter><name>password</name>
  <value></value></parameter>
  <parameter><name>driverClassName</name>
  <value>org.hsqldb.jdbcDriver</value></parameter>
  <parameter><name>driverName</name>
  <value>jdbc:HypersonicSQL:database</value></parameter>
</ResourceParams>
</Context>
```



Para pegarmos um conexão deste pool no código Java, devemos utilizar:

```
Connection conn = null;
DataSource ds = null;
try {
    Context initCtx = new InitialContext();
    ds =
        (DataSource) initCtx.lookup("java:comp/env/nomeDeLookup");
    conn = ds.getConnection();
}
catch(NamingException e) {
    throw new ServletException(e);
}
```

Session Beans

Um Session Bean normalmente representa um processo (lógica de negócios), onde um conjunto de tarefas deve ser concluído para que o processo seja de fato concluído. Este tipo de processo que depende do sucesso de um conjunto de operações é chamada operação atômica.

Neste cenário, caso uma das tarefas não tenha sido bem sucedida as tarefas anteriores devem desfazer seu trabalho (roll-back), ou seja, ou todas realizam seu trabalho, ou nenhuma realizará.

Session Beans também podem ser utilizados para gerenciamento de dados da seção do usuário.

Sesion Beans podem acessar diretamente o banco de dados, ou então acessar Entity Beans.

Stateless

Como o próprio nome diz, (stateless = sem estado) um Stateless Session Beans não mantém o estado conversacional.

Quando utilizamos Stateless Session Bean não temos a garantia que estaremos sempre utilizando a mesma instância de Session Bean em duas chamadas consecutivas a métodos, ou seja, podemos chamar o método setNome("Globalcode") na instância 1 e em seguida o método getNome() ser chamado na instância 2, retornando um nome diferente de "Globalcode".

Todas as instâncias de Stateless Session Bean são consideradas idênticas do ponto de vista do cliente.

Por isso o Stateless Session Bean tem maior performance e escalabilidade.

O desenvolvimento de um stateless session beans é relativamente simples e eles são muito eficientes. Este tipo de bean consome pouco recurso do servidor, pois não são persistentes e não são dedicados a um cliente específico, ou seja, não mantém o estado conversacional de um cliente.

Um stateless session bean possui tres partes:

- Home Interface
- Remote Interface
- Implementação

A **home interface** de um EJB é utilizada para que o cliente possa obter uma instancia do objeto em questão no servidor. A home interface representa um tipo de "factory" EJB.

Exemplo: AnoBisextoHome.java

```
1 package session;
2
3 import java.rmi.RemoteException;
4 import javax.ejb.CreateException;
5 import javax.ejb.EJBHome;
6
7 public interface AnoBisextoHome extends EJBHome {
8
9     public AnoBisexto create() throws RemoteException, CreateException;
10
11 }
```

Uma **remote interface** possui os métodos que serão implementados pela classe de implementação do EJB. Um client EJB não possui a classe de implementação do EJB pois esta é sempre executada no servidor, portanto todo client se referencia a um EJB através de uma interface Remote. A interface remote age como um Proxy Pattern.

Exemplo: AnoBisexto.java

```
1 package session;
2
3 import javax.ejb.EJBObject;
4 import java.rmi.RemoteException;
5
6 public interface AnoBisexto extends EJBObject {
7
```

```
8      public boolean anoBisexto(int ano) throws RemoteException;
9
10     }
```

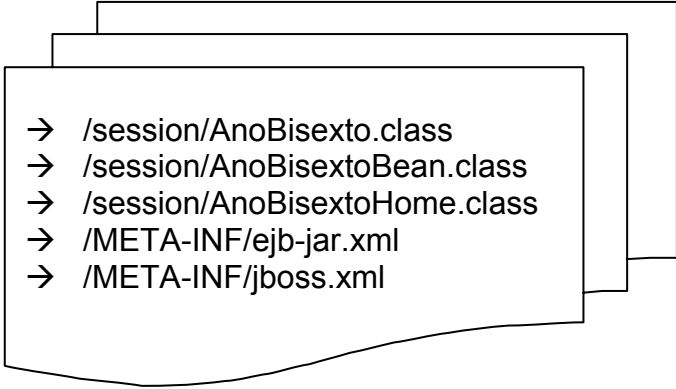
A **Bean class** é a implementação concreta do EJB. Nela faremos a codificação da funcionalidade do componente.

Exemplo: AnoBisexto.java

```
1  package session;
2
3  import javax.ejb.*;
4  import javax.naming.Context;
5  import javax.naming.InitialContext;
6  import java.util.*;
7  import java.io.*;
8  import java.net.*;
9  import java.rmi.RemoteException;
10
11  public class AnoBisextoBean implements SessionBean {
12
13      public void ejbCreate() throws CreateException {}
14
15      public boolean anoBisexto(int ano) {
16          boolean b=false;
17          if(ano%4==0 && ano%100==0 || ano%400==0)
18              b=true;
19          return b;
20      }
21
22      public AnoBisextoBean() {}
23      public void ejbRemove() {}
24      public void ejbActivate() {}
25      public void ejbPassivate() {}
26      public void setSessionContext(SessionContext sc) {}
```

27 }

Após compilarmos as classes acima devemos "empacotar" todo o conteúdo em um ejb jar:



- /session/AnoBisexto.class
- /session/AnoBisextoBean.class
- /session/AnoBisextoHome.class
- /META-INF/ejb-jar.xml
- /META-INF/jboss.xml

ejb-jar.xml

Deployment Descriptor do EJB

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar>

  <description>Exemplo de EJB Session</description>
  <display-name>EjbSession</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>AnoBisexto</ejb-name>
      <home>session.AnoBisextoHome</home>
      <remote>session.AnoBisexto</remote>
      <ejb-class>session.AnoBisextoBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>

</ejb-jar>
```


jboss.xmlDeployment
Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>

  <enterprise-beans>
    <session>
      <ejb-name>AnoBisexto</ejb-name>
      <jndi-name>ejb/AnoBisexto</jndi-name>
    </session>

  </enterprise-beans>
</jboss>
```

Temos 3 tipos freqüentes de clientes de um EJB:

- Servlet
- Aplicação stand-alone
- Outro EJB

Para que um Servlet possa acessar um EJB precisamos do seguinte código na classe servlet e dos seguintes deployment descriptors:

Exemplo: Servlet acessando EJB

```
1 package servletejb;
2
3 import java.io.*
4 import java.util.Enumeration;
5 import javax.servlet.ServletException;
6 import javax.servlet.http.*;
7 import javax.naming.*;
8 import javax.ejb.*;
9 import java.rmi.RemoteException;
10 import javax.rmi.PortableRemoteObject;
11 import session.*;
12
13 public class UsarAnoBisexto extends HttpServlet {
14
15     public void doGet(HttpServletRequest request,
```

```
16         HttpServletResponse response)
17         throws IOException, ServletException
18     {
19         response.setContentType("text/html");
20         PrintWriter writer = response.getWriter();
21         writer.println("<html>");
22         writer.println("<head>");
23         writer.println("<title>J2EE - Ano Bisexto Remoto</title>");
24         writer.println("</head>");
25         boolean anobisexto=false;
26
27         String stringAno=request.getParameter("textAno");
28
29         int ano=2000;
30
31         if(stringAno!=null && !stringAno.equals(""))
32             ano = Integer.parseInt(stringAno);
33
34         writer.println("<body bgcolor=white>");
35
36         try {
37
38             AnoBisexto ab = null;
39             Context initial = new InitialContext();
40             Object objref;
41             Objref = initial.lookup("java:comp/env/ejb/AnoBisexto");
42             AnoBisextoHome anoHome = null;
43             objref = (AnoBisextoHome) PortableRemoteObject.narrow(objref,
44                                                                     AnoBisextoHome.class);
45
46             if (objref instanceof AnoBisextoHome) {
47                 anoHome = (AnoBisextoHome) objref;
48             }
49             else {
50                 writer.println("<p>Erro de narrowing</p>");
51             }
52             ab = anoHome.create();
53             anobisexto=ab.anoBisexto(ano);
54             writer.println("Ano "+ano+(anobisexto ? "é bisexto" : " nao
55                                                                    é bisexto"));
56         }
57         catch(Exception e){
58             writer.println("<p>Erro ao acessar ejb</p>");
59             writer.println("<p>" + e.getMessage() + "</p>");
60         }
61
62         writer.println("</body>");
63         writer.println("</html>");
64     }
65 }
```

web.xml

Deployment descriptor

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
  2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <display-name>CursoJ2EE</display-name>
  <description>Aplicacao para Labs </description>

  <servlet>
    <servlet-name>AnoBisextoServlet</servlet-name>
    <servlet-class>servletejb.UsarAnoBisexto</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>AnoBisextoServlet</servlet-name>
    <url-pattern>/AnoBisexto</url-pattern>
  </servlet-mapping>

  <ejb-ref>
    <ejb-ref-name>ejb/AnoBisexto</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>session.AnoBisextoHome</home>
    <remote>session.AnoBisexto</remote>
  </ejb-ref>

</web-app>
```

jboss-web.xml

 Deployment Descriptor
de Assembling

```
<?xml version="1.0" encoding="UTF-8"?>

<jboss-web>

    <ejb-ref>

        <ejb-ref-name>ejb/AnoBisexto</ejb-ref-name>
        <jndi-name>ejb/AnoBisexto</jndi-name>

    </ejb-ref>

</jboss-web>
```

Clientes stand-alone também podem se conectar a um EJB, vejamos uma classe acessa um EJB:

```
Hashtable props = new Hashtable();
props.put(Context.INITIAL_CONTEXT_FACTORY,
           "org.jnp.interfaces.NamingContextFactory");
props.put(Context.PROVIDER_URL, "localhost:1099");
props.put("java.naming.rmi.security.manager", "yes");
props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming");
Context context = new InitialContext(props);
// deste ponto em diante é o mesmo código do servlet
```

Stateful Session Bean

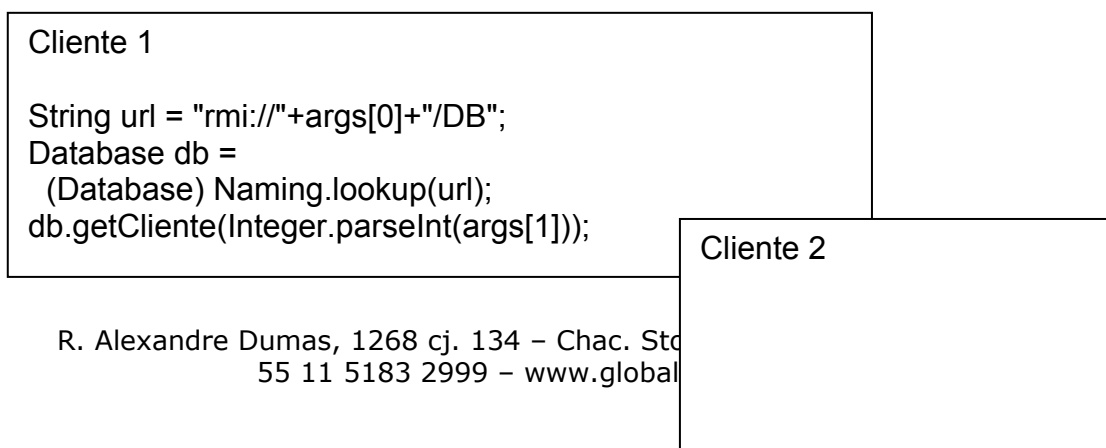
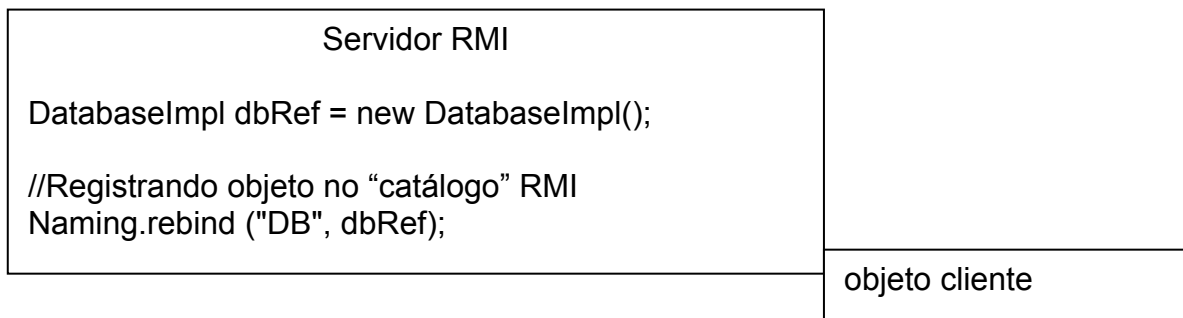
Um Stateful Session Bean é feito para atender apenas um cliente por vez, ou seja, ele mantém o *estado conversacional** entre chamadas a métodos. Neste caso, um cliente que chama um método setNome("Globalcode") tem certeza que na próxima chamada

a um método do Session Bean estará chamando um método na mesma instância de objeto chamado anteriormente.

Entity Bean

Vamos imaginar a situação do primeiro servidor de objetos RMI Java que montamos no curso que não continha um factory de objetos. O que aconteceria se diversos usuários utilizassem este mesmo objeto ao mesmo tempo?

O servidor abriria uma thread para cada usuário conectado (socket server), porém ambos utilizariam a mesma área de memória e o desenvolvedor ficará responsável por sincronizar o acesso ao dado compartilhado.



Um Entity Bean normalmente representa uma linha em um banco de dados relacional. No entanto, um Entity Bean também pode ser mapeado para um conjunto de dados encontrados em mais que uma tabela.

Isto não quer dizer que se a tabela tem 100.000 linhas o container irá criar 100.000 Entity Beans e deixá-los disponíveis, isto quer dizer que existe uma instância diferente para cada linha da tabela que deva ser acessada, ou seja, se dois clientes acessarem a mesma linha de uma tabela, estarão usando a mesma instância de Entity Bean.

A arquitetura EJB provê a sincronização necessária para que o entity bean suporte este acesso concorrente.

Entity Bean é um tipo de objeto que fornece suporte a acesso remoto sincronizado e podem ser persistidos no banco de dados automaticamente ou pelo container.

O desenvolvimento é muito semelhante ao de Session bean, ou seja, temos uma home, uma remote a classe de implementação.

Os entity beans com persistência automática são chamados de CMP (container managed persistence):

Exemplo de CMP entity:

Home Interface

```
public interface CorretoraHome extends EJBHome  
{
```

```
    public Corretora create(String id)  
        throws RemoteException, CreateException;  
    public Corretora findByPrimaryKey (String id)  
        throws RemoteException, FinderException;
```

R. Alexandre Dumas, 1268 cj. 134 – Chac. Sto. Antonio – São Paulo – SP
55 11 5183 2999 – www.globalcode.com.br

Última Revisão: 1/10/2002 Data Criação:9/8/2002 Revisão 1

```
public Collection findByType (String mne)
    throws RemoteException, FinderException;
public Collection findAll()
    throws RemoteException, FinderException;
}
```

A implementação do método findByType em CMP na espec 1.1 é feita de acordo com o vendor deployment descriptor.

findAll e findByPrimaryKey é padrão J2EE

Bean Class

```

public class CorretoraBean extends CorretoraValue implements EntityBean {
    EntityContext ctx;
    public String ejbCreate (String id)
    {
        cdCorretora = id;
        return null;
    }
    public void ejbPostCreate(String id) {}
    public void setEntityContext(EntityContext ctx) {
        this.ctx = ctx; }
    public void unsetEntityContext() { ctx = null; }
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}
    public CorretoraValue getLocalCorretora() {
        CorretoraValue value=new CorretoraValue(mneCorretora,
        mneUsuarioInclusao, mneUsuarioOcorrencia, cdCorretora, nomCorretora,
        ocorrencia, dtInclusao, dtOcorrencia);
        return value;
    }
}

```



Em CMP's utilizamos os métodos de call-back da interface entity bean somente para otimizações finas no trabalho de persistência do container.

Remote Interface

```
package ejb.entity;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.util.Date;

public interface Corretora extends EJBObject
{
    public void setMneCorretora(String mne) throws RemoteException;
    public String getMneCorretora() throws RemoteException;
    public void setMneUsuarioInclusao(String mne) throws RemoteException;
    public String getMneUsuarioInclusao() throws RemoteException;
    public void setMneUsuarioOcorrencia(String mne) throws RemoteException;
    public String getMneUsuarioOcorrencia() throws RemoteException;
    public void setCdCorretora(String cod) throws RemoteException;
    public String getCdCorretora() throws RemoteException;
    public void setNomCorretora(String nome) throws RemoteException;
    public String getNomCorretora() throws RemoteException;
    public void setDtInclusao(Date data) throws RemoteException;
    public Date getDtInclusao() throws RemoteException;
    public void setDtOcorrencia(Date data) throws RemoteException;
    public Date getDtOcorrencia() throws RemoteException;
    public void setOcorrencia(String flag) throws RemoteException;
    public String getOcorrencia() throws RemoteException;
    public CorretoraValue getLocalCorretora() throws RemoteException;
}
```

Deployment Descript do EJB**ejb-jar.xml**

```
<?xml version="1.0"?>
```

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise  
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

```
<ejb-jar>
```

```
  <display-name>Corretora</display-name>
```

```
  <enterprise-beans>
```

```
    <entity>
```

```
      <description>Entidade de Corretora</description>
```

```
      <ejb-name>CorretoraBean</ejb-name>
```

```
      <home>ejb.entity.CorretoraHome</home>
```

```
      <remote>ejb.entity.Corretora</remote>
```

```
      <ejb-class>ejb.entity.CorretoraBean</ejb-class>
```

```
      <persistence-type>Container</persistence-type>
```

```
        <prim-key-class>java.lang.String</prim-key-class>
```

```
        <primkey-field>cdCorretora</primkey-field>
```

```
    <reentrant>False</reentrant>
```

```
      <cmp-field><field-name>cdCorretora</field-name></cmp-field>
```

```
      <cmp-field><field-name>mneCorretora</field-name></cmp-field>
```

```
    ...
```

```
  </entity>
```

```
</enterprise-beans>
```

```
<assembly-descriptor>
```

```
  <container-transaction>
```

```
    <method>
```

R. Alexandre Dumas, 1268 cj. 134 – Chac. Sto. Antonio – São Paulo – SP
55 11 5183 2999 – www.globalcode.com.br

Última Revisão: 1/10/2002 Data Criação:9/8/2002 Revisão 1

```
<ejb-name>CorretoraBean</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Supports</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

jboss.xml

```
<?xml version="1.0" encoding="Cp1252"?>

<jboss>
  <secure>>false</secure>
  <container-configurations />
  <resource-managers />

  <enterprise-beans>
    <entity>
      <ejb-name>CorretoraBean</ejb-name>
      <jndi-name>globalcode/ejb/Corretora</jndi-name>
      <configuration-name></configuration-name>
    </entity>
  </enterprise-beans>
</jboss>
```

Configurando a camada de persistência do Jboss

A camada persistente do jboss é chamada de Jaws, e configuramos ela no seguinte arquivo xml que se encontra no diretório conf:

standardjaws.xml

```
<jaws>
  <datasource>java:/poolcurso</datasource>
  <type-mapping>mySQL</type-mapping>
  <debug>>false</debug>

  <default-entity>
    <create-table>>true</create-table>
    <remove-table>>false</remove-table>
    <tuned-updates>>true</tuned-updates>
    <read-only>>false</read-only>
    <time-out>300</time-out>
    <select-for-update>>false</select-for-update>
  </default-entity>

  ...
</jaws>
```

DataSource configurado jboss.jcml

Escolha um Type-Mapping
de tipos JAVA - RDBMS

Configuração homologada no JBOSS 2.4.4

Client-side

Percorre a lista de corretoras:

```
Iterator I = ClientUtil.getCorretoraHome().findByType( "99").iterator();
listCorretoras.removeAllItems();
Corretora corretora;
while(i.hasNext()) {
    corretora = (Corretora) PortableRemoteObject.narrow(i.next(),
Corretora.class);
    CorretoraValue value=corretora.getLocalCorretora();
    listCorretoras.addItem(value);
}
```

Delete em uma entidade:

```
ClientUtil.getCorretoraHome().remove(corretora.getCdCorretora());
```

Inserindo dados em uma entidade:

```
corretora =
ClientUtil.getCorretoraHome().create(this.textCodigoCorretora.getText());
corretora.setMneCorretora(this.textApelido.getText());
corretora.setNomCorretora(this.textNomeCorretora.getText());
```

Message Driven Beans e JMS

]Atualmente podemos trocar mensagens entre computadores Através de protocolos diversos como FTP, HTTP, POP etc.

Existem sistemas mais especializados de mensagens que tentam solucionar através de servidores, protocolos e outros problemas de confiabilidade na conexão, empacotamento e falhas da rede.

Contamos na indústria com servidores de mensagens como IBM MQSeries, TIBCO Rendezvous e outros. Tais produtos prevêm formas de trocar pacotes de informações na rede de maneira segura, com camada de transporte com garantia de entrega e outros. Isso é muito interessante no ponto de vista de integração de software e também na confiabilidade para transações e-commerce em geral.

JMS é uma API do **Java Community Process** (www.jcp.org) que prevê uma maneira padrão de comunicar aplicação Java com servidores de mensagens.

Existem servidores que são compatíveis com tal padrão, como o formato de driver definidos por interface como JDBC e JNDI para diretórios.

Estes servidores contam com **dois** padrões populares na distribuição de mensagens:

Point-to-point

- Baseado em filas FIFO (First In First Out).
- Uma mensagem tem um só consumidor.
- **Independente de tempo.**
- O destino deve sinalizar o recebimento da mensagem.

Podemos contar com múltiplos servidores consumindo uma mesma fila com o objetivo de nos beneficiarmos de sistemas de cluster de consumo ativo com balanceamento de carga entre os diversos servidores.

Publish/Subscribe

- Múltiplos recipientes por mensagens.
- Baseado em IP Multicast (ips delasse D).
- Relativamente dependente de tempo.
- Conceito de tópicos e assinantes.

Configurando filas no JBoss

Para filas Publish/Subscribe:

```
<mbean code = "org.jboss.mq.server.TopicManager" name =  
JBossMQ:service=Topic,name=noticiasEleicao"/>
```

Para files point-to-point:

```
<mbean code = "org.jboss.mq.server.QueueManager" name =  
JBossMQ:service=Queue,name=SPB"/>
```

Exemplo de programa “Publisher”

```
String msg="mensagem para publisher";  
Hashtable props = new Hashtable();  
props.put(Context.INITIAL_CONTEXT_FACTORY,  
    "org.jnp.interfaces.NamingContextFactory");  
props.put(Context.PROVIDER_URL, "localhost:1099");  
props.put("java.naming.rmi.security.manager", "yes");  
props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming");
```

```
Context context = new InitialContext(props);  
TopicConnectionFactory topicFactory =  
    (TopicConnectionFactory)context.lookup("ConnectionFactory");  
TopicConnection topicConnection =  
    topicFactory.createTopicConnection();  
TopicSession topicSession =  
    topicConnection.createTopicSession(  
        // No transaction  
        false,
```

R. Alexandre Dumas, 1268 cj. 134 – Chac. Sto. Antonio – São Paulo – SP
55 11 5183 2999 – www.globalcode.com.br

```
// Auto ack
Session.AUTO_ACKNOWLEDGE);
Topic topic=(Topic) context.lookup("topic/noticiasEleicao");
// Create a publisher
TopicPublisher topicPublisher=topicSession.createPublisher(topic);
TextMessage message = topicSession.createTextMessage();
message.setText(msg);
topicPublisher.publish(topic, message);
```

Exemplo de programa enviando mensagem para fila

```
QueueConnectionFactory queueFactory =  
(QueueConnectionFactory)context.lookup("ConnectionFactory");  
// Create the connection  
QueueConnection queueConnection =  
queueFactory.createQueueConnection();  
  
// Create the session  
QueueSession queueSession=queueConnection.createQueueSession(  
    // No transaction  
    false,  
    // Auto ack  
    Session.AUTO_ACKNOWLEDGE);  
// Look up the destination  
Queue queue = (Queue)context.lookup("queue/SPB");  
QueueSender queueSender = queueSession.createSender(queue);  
message = queueSession.createTextMessage();  
message.setText(msg);  
queueSender.send(queue, message);
```

Observações:

Para executar o exemplos acima com JBOSS 2.4.4 é necessário configurar o CLASSPATH com os seguintes JARs:

jboss-client.jar, jbossmq-client.jar, log4j.jar

Message Driven Bean

Message Driven Bean é um EJB que é acionado mensagens, ou seja, não possui interface Remote e não pode ser chamado “sincronamente”. Ele pode consumir mensagens de uma fila ou então ser assinante de um tópico.

Ele é o tipo mais simples de EJB pois não possui Home nem Remote interface. Seu empacotamento é semelhante ao de um Session bean, onde devemos colocar os Deployment Descriptors no diretório META-INF.

O comportamento de um MDB é semelhante ao de um Session Bean stateless. O servidor de aplicações pode criar um pool de objetos para consumo e processamento das mensagens da fila.

A ordem de chamada dos call-backs é a seguinte:

1. **newInstance()** – método da classes class.
2. **setMessageDrivenContext()** – “Construtor Enterprise”
3. **onMessage(msg)** – método chamado a cada nova mensagem

A classe bean deve seguir as regras abaixo:

1. Deve ser public
2. Não pode ser declarada como final ou abstract
3. Deve conter um construtor “no-arg” / padrão
4. Não pode implementar métodos com modificador finalize

Vamos fazer o deploy do exemplo abaixo:

ProcessMDB.java

```
package ejb.mdb;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.ejb.EJBException;

import javax.jms.MessageListener;
import javax.jms.Message;

public class ProcessMDB implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx = null;

    public ProcessMDB() {
    }

    //--- MessageDrivenBean
    public void setMessageDrivenContext(MessageDrivenContext ctx)
        throws EJBException {
        this.ctx = ctx;
    }

    public void ejbCreate() {}

    public void ejbRemove() {ctx=null;}

    //--- MessageListener
    public void onMessage(Message message) {
        System.err.println("Solicitação de Processamento :" + message.toString());
    }
}
```

ejb-jar.xml

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar>
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>ProcessMDB</ejb-name>
      <ejb-class>ejb.mdb.ProcessMDB</ejb-class>
      <message-selector></message-selector>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <destination-type>javax.jms.Queue</destination-type>
        <subscription-durability>NonDurable</subscription-durability>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>ProcessMDB</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

jboss.xml

```
<?xml version="1.0" encoding="Cp1252"?>
<jboss>
  <enterprise-beans>
```

```
<message-driven>
  <ejb-name>ProcessMDB</ejb-name>
  <configuration-name>Standard Message Driven Bean</configuration-
name>
  <destination-jndi-name>queue/ProcessMDB</destination-jndi-name>
</message-driven>
</enterprise-beans>
</jboss>
```

Transações

NotSupported = não criará contexto de transação e se o EJB for chamado de um outro EJB com contexto de transação, a transação sera interrompida temporariamente.

Requires New = sempre cria um contexto de transação para executar o método.

Required = se não tem contexto cria, caso contrário se une ao contexto que foi chamado.

Supports = tanto faz, se tiver um contexto criado se une a ele, caso contrário se comporta como NotSupported

Mandatory = exige que o método seja chamado de um método que tenha criado o contexto, caso contrário dispara um Exception.

Never = se for chamado de um método com contexto de transação dispara uma exception.

Indicamos o tipo de comportamento de transação no deployment descriptor do EJB como vemos no próximo capítulo.

O nível de isolamento bem como lock otimista ou pessimista é essencialmente dependente do recurso em questão. A API JDBC através do `Connection.setTransactionIsolation` permite configurarmos níveis de isolamento para:

Dirty Read = Permite que uma linha modificada por uma transação seja lida por outra transação antes do comitt.

Non-repetable read = Quando uma transação faz a leitura de uma linha e esta é alterada por outra transação.

Phantom read = Quando uma transação faz leituras de uma tabela com uma determinada clausula WHERE e outra transação adiciona ou altera registros que afetariam a consulta executada pela 1ª transação.

Como cada RDBMS implementa estratégia para tais garantias é específico de cada fabricante.

Segurança e Deployment Descriptor

Podemos configurar segurança em EJBs como fazendo em servlets: definindo roles e indicando que role o usuário precisa pertencer para executar um determinado método:

```
<assembly-descriptor>
  <security-role>
    <role-name>admin</role-name>
  </security-role>
  <method-permission>
    <role-name>admin</role-name>
    <method>
      <description />
      <ejb-name>Enterprise1</ejb-name>
      <method-intf>Home</method-intf>
      <method-name>create</method-name>
    </method>
  </method-permission>
  <container-transaction>
    <method>
      <ejb-name>Enterprise1</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

Podemos definir variáveis de ambiente de EJBs, basta colocarmos o TAG abaixo na descrição do EJB e fazer o lookup via Context:

<entity> ou <session>

...

<env-entry>

<env-entry-name>valorMinimo</env-entry-name>

<env-entry-type>java.lang.Double</env-entry-type>

<env-entry-value>10.0</env-entry-value>

</env-entry>

</entity> ou </session>

Exemplo de Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise  
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

```
<ejb-jar>
```

```
<enterprise-beans>
```

```
<session>
```

```
<ejb-name>Enterprise1</ejb-name>
```

```
<home>tejb.Enterprise1Home</home>
```

```
<remote>tejb.Enterprise1</remote>
```

```
<ejb-class>tejb.Enterprise1Bean</ejb-class>
```

```
<session-type>Stateless</session-type>
```

```
<transaction-type>Container</transaction-type>
```

```
<env-entry>
```

```
<description />
```

```
<env-entry-name>servidorSPB</env-entry-name>
```

```
<env-entry-type>java.lang.String</env-entry-type>
```

```
<env-entry-value>192.168.5.12</env-entry-value>
```

```
</env-entry>
```

```
<resource-ref>
```

```
<description />
```

```
<res-ref-name>db/J2EEDS1</res-ref-name>
```

Última Revisão: 1/10/2002 Data Criação:9/8/2002 Revisão 1

```
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
</resource-ref>
</session>
<session>
  <ejb-name>Enterprise2</ejb-name>
  <home>tejb.Enterprise2Home</home>
  <remote>tejb.Enterprise2</remote>
  <ejb-class>tejb.Enterprise2Bean</ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
</session>
<entity>
  <ejb-name>Enterprise3</ejb-name>
  <home>tejb.Enterprise3Home</home>
  <remote>tejb.Enterprise3</remote>
  <ejb-class>tejb.Enterprise3Bean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>
</entity>
</enterprise-beans>
<assembly-descriptor>
  <security-role>
    <role-name>padmin</role-name>
  </security-role>
  <method-permission>
    <role-name>admin</role-name>
    <method>
      <description />
      <ejb-name>Enterprise1</ejb-name>
      <method-intf>Home</method-intf>
```

```
<method-name>create</method-name>
</method>
</method-permission>
<container-transaction>
  <method>
    <ejb-name>Enterprise1</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
    <ejb-name>Enterprise2</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
    <ejb-name>Enterprise3</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```