Disciplina: Desenvolvimento de Software

Aula 9: JPA e EJB

Apresentação

Um problema comum nos sistemas atuais é a grande diferença existente entre o modelo de programação orientada a objetos e a forma relacional de armazenamento de dados.

Com a explosão do número de sistemas orientados a objeto, o mapeamento objeto-relacional se tornou uma prática comum e diversas ferramentas foram criadas para a automatização desse processo, o qual é efetuado via JPA no ambiente Java da atualidade.

Os sistemas corporativos também utilizam objetos distribuídos, como os Enterprise Java Beans, tirando proveito de toda uma arquitetura segura e transacional para efetuar diversos processos de negócios.

Objetivos

- Definir o conceito de mapeamento objeto-relacional;
- Explicar a tecnologia JPA;
- Analisar a arquitetura de objetos distribuídos do tipo EJB.

Mapeamento objeto-relacional

Os bancos de dados relacionais são a tecnologia de armazenamento mais consolidada no mercado, mas, como pudemos observar, a forma de tratar as operações sobre o banco, dentro de um ambiente orientado a objetos, envolve a criação de classes que viabilizem a organização e reúso do código.

Se observarmos bem, as tarefas envolvidas nessa transformação são bastante repetitivas, podendo ser facilmente padronizadas. Foi com base nessa premissa que surge a ideia por trás do mapeamento objeto-relacional.

Também chamado de ORM (do inglês Object-Relational Mapping), a técnica envolve uma reinterpretação dos bancos relacionais, fazendo com que as tabelas sejam representadas por classes e os registros dessas tabelas por instâncias dessas classes.

Além disso, é necessário desenvolver gestores de transação, que assumem o papel de uma classe DAO, mas que não exigem a criação de comandos SQL.

Esses gestores vão gerar os comandos necessários durante a execução de forma automática, eliminando a necessidade de utilizar mais de uma linguagem simultaneamente, como ocorria antes.

É claro que deverá ter alguma forma de mapear as tabelas e os respectivos campos para as classes e atributos, o que inicialmente era feito com o uso de **XML** em frameworks como **Hibernate.**

O uso de XML fornece uma solução bastante dinâmica, mas que não é tão formal quanto a utilização de anotações, como ocorre nos atuais mapeamentos feitos no ambiente Java.

Comentário

A plataforma Microsoft traz uma solução muito interessante, que inclusive não necessita desse tipo de mapeamento, denominada **LINQ.**

Com o uso da abordagem proporcionada pelo ORM, os sistemas apresentam um código mais conciso, e ainda a vantagem de possibilitar a modificação do servidor de banco de dados com grande facilidade, alterando apenas poucos arquivos de configuração, normalmente no formato XML.

Java Persistence API

Um dos maiores avanços do Java foi a definição do **JPA,** ou Java Persistence API, unificando os diversos frameworks de persistência em uma arquitetura padronizada, com base em código anotado e apenas um arquivo de configuração, o **persistence.xml.**

Quando nos referimos ao JPA, não estamos tratando apenas de um framework, mas de uma API de persistência, descrevendo uma interface comum que deve ser seguida pelos diversos frameworks de persistência do ambiente Java, como Hibernate, Eclipse Link e Oracle Toplink.

Para criar uma entidade no JPA, devemos criar um **POJO** (Plain Old Java Object), ou seja, uma classe sem métodos de negócios, mas com atributos definidos de forma privada e métodos de acesso públicos, além de um construtor padrão e alguns métodos utilitários, como **hash.**

Essa classe deve receber anotações que serão responsáveis pelo mapeamento efetuado entre a classe e a tabela, ou seja, o mapeamento objeto-relacional.

Podemos observar, a seguir, um exemplo simples de entidade JPA.

```
@Entity
@Table(name="TB_ALUNO)
public class Aluno implements Serializable{
  @Id
  @Column(name="PK_MATRICULA")
  private String matricula;
  @Column(name="TX_NOME")
  private String nome;
  public Aluno()
  {
  public Aluno(String matricula){
      this.matricula = matricula;
  }
  // getters e setters públicos
}
```

As duas primeiras anotações definem o POJO como uma entidade e fazem o mapeamento com a tabela que será utilizada para efetuar a persistência no banco de dados.

```
@Entity
@Table(name="TB_ALUNO)
public class Aluno {
```

A anotação Id define um atributo como chave primária, enquanto **Column** faz o mapeamento do atributo para o campo correto da tabela.

```
@Id
  @Column(name="PK_MATRICULA")
  private String matricula;
```

Tendo as entidades criadas, devemos definir a conexão com o banco de dados, com uso de um driver **JDBC,** um framework de persistência e o arquivo **persistence.xml.**

Esse arquivo será colocado na pasta META-INF do projeto e responsável pela configuração da conexão, indicando a URL do banco, driver JDBC e framework utilizados, entre diversas outras opções, como podemos observar no exemplo seguinte.

Exemplo:

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

< persistence version="2.1"
  xmlns="//xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="//www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="//xmlns.jcp.org/xml/ns/persistence
  //xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

 < persistence-unit name="TesteBasePU"
      transaction-type="RESOURCE_LOCAL">
  < provider>org.eclipse.persistence.jpa.PersistenceProvider
  < /provider>
  < class > modelo.Produto< /class >

  < properties>
      < property name="javax.persistence.jdbc.url"
value="jdbc:derby://localhost:1527/LojaEAD"/>
  < property name="javax.persistence.jdbc.user"
      value="LojaEAD"/>
  < property name="javax.persistence.jdbc.driver"
      value="org.apache.derby.jdbc.ClientDriver"/>
  < property name="javax.persistence.jdbc.password"
      value="LojaEAD"/>
 < /properties>

 < /persistence-unitv>
< /persistence>
```

A primeira informação relevante que encontramos nesse arquivo é o nome da unidade de persistência, juntamente com o tipo de transação que será utilizada.

As transações são de enorme importância para manter o nível de isolamento adequado entre tarefas executadas de forma concorrente, como no caso de múltiplos usuários acessando o mesmo banco de dados.

```
< persistence-unit name="TesteBasePU"

transaction-type="RESOURCE_LOCAL">
```

Em seguida, temos o framework de persistência que será utilizado, no caso o Eclipse Link, e as classes de entidade a ser consideradas.

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider
</provider>
<class>modelo.Produto</class>
```

Por fim, temos as configurações relacionadas à conexão JDBC, como URL, driver utilizado, usuário e senha.

```
<property name="javax.persistence.jdbc.url"
      value="jdbc:derby://localhost:1527/LojaEAD"/>
  <property name="javax.persistence.jdbc.user"
      value="LojaEAD"/>
  <property name="javax.persistence.jdbc.driver"
      value="org.apache.derby.jdbc.ClientDriver"/>
  <property name="javax.persistence.jdbc.password"
      value="LojaEAD"/>
```

Outro elemento importante, no ambiente do JPA, é o EntityManager, responsável por efetuar as operações de persistência no banco de dados, bem como a seleção a partir do mesmo.

Comentário

Note que o JPA não elimina o uso de JDBC, pois o que é feito é a geração dos comandos SQL de forma automatizada a partir das requisições efetuadas pelo EntityManager, sempre utilizando as informações transmitidas pelas classes anotadas.

Para gerar um objeto da classe EntityManager, precisamos de um **EntityManagerFactory,** o qual estará relacionado diretamente ao arquivo **persistence.xml** através da definição da unidade de persistência.

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("TesteBasePU");
<EntityManager em = emf.createEntityManager();
```

O uso do NetBeans facilita muito a criação das entidades JPA, efetuando uma inferência sobre o esquema existente no banco de dados com suas tabelas.

Para utilizar essa funcionalidade, devemos adicionar Novo Arquivo, a partir do menu, e seguir os seguintes passos:

Comentário

Como o banco conta com apenas uma tabela, ela será a única entidade gerada, mas poderíamos recuperar diversas tabelas e relacionamentos de uma só vez por meio desse processo de criação fornecido pelo NetBeans.

Podemos observar a entidade gerada a seguir.

```
@Entity
@Table(name = "PRODUTO")
@NamedQueries({
  @NamedQuery(name = "Produto.findAll", query = "SELECT p FROM Produto p")})
public class Produto implements Serializable {

  private static final long serialVersionUID = 1L;
  @Id
  @Basic(optional = false)
  @Column(name = "CODIGO")
  private Integer codigo;
  @Column(name = "NOME")
  private String nome;
  @Column(name = "QUANTIDADE")
  private Integer quantidade;

  public Produto() { }

  public Produto(Integer codigo) {
      this.codigo = codigo;
  }

  public Integer getCodigo(){return codigo;}
  public void setCodigo(Integer codigo){this.codigo=codigo;}
  public String getNome(){return nome; }
  public void setNome(String nome){this.nome=nome;}
  public Integer getQuantidade(){return quantidade;}
  public void setQuantidade(Integer quantidade)
{this.quantidade=quantidade;}

  @Override
  public int hashCode() {
return (codigo != null ? codigo.hashCode() : 0);
  }

  @Override
  public boolean equals(Object other) {
      return ((other==null) || !(other instanceof Produto))?false:
((this.codigo==null)||((Produto)other).codigo==null)
  ?false:this.codigo.equals(((Produto)other).codigo);
  }

  @Override
  public String toString() {
return "modelo.Produto[ codigo=" + codigo + " ]";
  }
  }
```

Foram utilizadas algumas anotações novas nessa entidade, a começar por **NamedQuery**, que define uma instrução de consulta na sintaxe **JPQL,** retornando todos os produtos a partir do banco de dados.

```
@NamedQueries({
      @NamedQuery(name = "Produto.findAll", query = "SELECT p FROM Produto p")})
```

Também temos a obrigatoriedade do campo-chave com o uso de uma das opções de Basic.

```
@Id
@Basic(optional = false)
@Column(name = "CODIGO")
private Integer codigo;
```

O arquivo de configuração **persistence.xml** também foi adicionado ao projeto, e o próximo passo é a alteração de nossa classe DAO para utilizar o EntityManager, como podemos observar no código seguinte.

As importações não foram transcritas, pois são adicionadas de forma automática pelo NetBeans.

```
public class ProdutoDAO implements Serializable {


 public ProdutoDAO() {
   emf = Persistence.createEntityManagerFactory("TesteBasePU");
 }
 private final EntityManagerFactory emf;

 public EntityManager getEntityManager() {
   return emf.createEntityManager();
}

 public void incluir(Produto produto) {
   EntityManager em = getEntityManager();
   try {
em.getTransaction().begin();
em.persist(produto);
em.getTransaction().commit();
   } finally {
em.close();
   }
 }

 public void excluir(Integer codigo) {
   EntityManager em = getEntityManager();
   try {
em.getTransaction().begin();
Produto produto = em.find(Produto.class, codigo);
em.remove(produto);
em.getTransaction().commit();
   } finally {
em.close();
   }
 }

 public List obterTodos() {
   EntityManager em = getEntityManager();
   try {
CriteriaQuery cq =
em.getCriteriaBuilder().createQuery();
cq.select(cq.from(Produto.class));
Query q = em.createQuery(cq);
return q.getResultList();
   } finally {
em.close();
 }
}
}
```

É fácil notar como o código fica muito mais simples com o uso do **EntityManager**, o qual é obtido a partir do **EntityManagerFactory**, instanciado no construtor da classe DAO.

```
public EntityManager getEntityManager() {
return emf.createEntityManager();
}
```

Com esse método utilitário em mãos, podemos efetuar as diversas operações de banco, como a inclusão, que utilizará o método **persist** dentro de uma transação JPA.

Quando esse método é executado, o JPA faz uma inferência sobre a classe da entidade (Produto): descobre o nome da tabela e dos campos, pega os valores do objeto corrente, e monta o SQL com o comando INSERT necessário para ser enviado ao banco via JDBC.

```
em.getTransaction().begin();
em.persist(produto);
em.getTransaction().commit();
```

O procedimento para exclusão segue um caminho similar, exigindo apenas a busca da entidade no banco para que possa ser excluída. O método **find** efetua a busca a partir do valor da chave primária, retornando o produto em questão, já associado ao banco, e o método **remove** permite excluir o produto encontrado.

```
em.getTransaction().begin();
Produto produto = em.find(Produto.class,
codigo);
em.remove(produto);
em.getTransaction().commit();
```

O uso do comando find implica no envio de um comando SELECT ao banco, enquanto remove enviará um comando DELETE.

Nem sempre o comando de seleção precisa ser executado, já que o JPA trabalha com um pool de entidades, atualizado de acordo com o nível de acesso, e, se a entidade já se encontra no pool, ela é utilizada diretamente.

Finalmente, podemos observar o método para carregar o conjunto de entidades a partir da tabela, o que é feito com a transformação de um comando JPQL para o equivalente SQL, que deverá ser enviado ao banco via JDBC.

```
CriteriaQuery cq =
em.getCriteriaBuilder().createQuery();
cq.select(cq.from(Produto.class));
Query q = em.createQuery(cq);
return q.getResultList();
```

Nesse caso, foi utilizado um objeto do tipo CriteriaQuery para a montagem do comando JPQL, mas poderíamos utilizar uma NameQuery, bastando alterar esse trecho de código para as linhas apresentadas a seguir.

Nesse segundo formato, seria utilizada a NamedQuery definida na classe Produto a partir das anotações relacionadas.

```
Query q =
em.createNamedQuery("Produto.findAll");
return q.getResultList();
```

Agora é só utilizar o DAO normalmente, como fazíamos antes, como no exemplo seguinte.

```
public class TesteBase2 {
 public static void main(String[] args) {
   ProdutoDAO dao = new ProdutoDAO();
   dao.obterTodos().forEach((p) -> {
System.out.println(p.getCodigo()+" - "+p.getNome()+
" :: "+p.getQuantidade());
   });
 }
}
```

A saída da execução desse arquivo é apresentada a seguir. Podemos observar algumas linhas com a assinatura **“EL Info”**, referente ao uso do **Eclipse Link** como framework de persistência.

Exemplo

```
[EL Info]: 2019-01-05
12:15:18.758--ServerSession(1821228886)--EclipseLink,
version: Eclipse Persistence Services - 2.5.2.v20140319-9ad6abd
[EL Info]: connection: 2019-01-05
12:15:19.656--ServerSession(1821228886)--
file:/C:/MeusTestes/TesteBase/build/classes/_TesteBasePU
login successful
1 - Banana :: 1000
2 - Morango :: 150
3 - Laranja :: 400
4 - Manga :: 350
[EL Info]: connection: 2019-01-05
12:15:20.019--ServerSession(1821228886)--
file:/C:/MeusTestes/TesteBase/build/classes/_TesteBasePU
logout successful
```

Atenção

As anotações utilizadas para configurar o mapeamento da entidade não são serializáveis, o que faz com que seja mantido o isolamento do acesso ao banco de dados em arquiteturas com múltiplas camadas.

Com o JPA, devemos utilizar uma linguagem de consulta própria, denominada **JPQL** (Java Persistence Query Language), que acaba sendo bastante semelhante ao **SQL**, mas trata com objetos e coleções, ao invés de conjuntos de tuplas.

Uma Query utiliza a sintaxe JPQL, podendo receber parâmetros, como podemos observar no exemplo seguinte, com a construção de um método que pode ser adicionado a ProdutoDAO.

Exemplo:

```
public List<Produto>obterNome(String nome){
   EntityManager em = getEntityManager();
   try {
Query q = em.createQuery(
"Select p from Produto p where p.nome like ?1");
q.setParameter(1, "%"+nome+"%");
return q.getResultList();
   } finally {
em.close();
   }
} (
```

O parâmetro é definido com o uso de ?**1** e seu valor é configurado com **setParameter**.

```
Query q = em.createQuery(
"Select p from Produto p where p.nome like ?1");
q.setParameter(1, "%"+nome+"%");
```

Enterprise Java Beans

Um Enterprise Java Bean **(EJB)** é um componente corporativo utilizado de forma indireta.

Dentro de um ambiente de objetos distribuídos, suporta transações locais e distribuídas, elementos de autenticação e segurança, acesso a banco de dados via pool de conexões, entre diversos outros elementos da plataforma Java Enterprise Edition **(JEE)**.

Inicialmente, todo componente EJB existe dentro de um **pool** de objetos do mesmo tipo. O número de instâncias presentes irá aumentar ou diminuir de acordo com a quantidade de solicitações de serviços efetuadas em paralelo a cada intervalo determinado.

O acesso aos serviços oferecidos por esse pool deve ser solicitado a partir de uma interface **local** ou **remota.** Essas interfaces são geradas a partir de “fábricas” registradas através do **JNDI**, sendo chamadas de **Home**, pra interfaces remotas, e **LocalHome,** para interfaces locais.

A programação em si, considerando o modelo adotado a partir do JEE5, é bastante simples, e precisaremos apenas das anotações corretas para que os Application Servers, como o JBoss ou o GlassFish, se encarreguem de montar toda a estrutura necessária.

Isso é bem diferente do processo de criação adotado pelo J2EE, o qual utilizava um modelo de programação baseado em contrato, envolvendo diversas classes, interfaces e arquivos XML.

O primeiro tipo de EJB a ser considerado é o de sessão, responsável por efetuar processos de negócios de forma síncrona.

Ele pode assumir três comportamentos diferentes:

Clique nos botões para ver as informações.

Stateless

Assume esse comportamento quando não guarda valores entre chamadas sucessivas.

Utilizamos Stateless quando não precisamos de nenhuma informação de processos anteriores ao corrente. Qualquer instância do pool de EJBs pode ser escolhida, e não há necessidade de carga de dados anteriores, o que faz com que seja o comportamento mais ágil para um Session Bean.

Sateful

Assume esse comportamento quando, ao contrário do anterior, guarda valores entre chamadas sucessivas.

O Sateful deve ser utilizado quando precisamos de informações anteriores, como em uma cesta de compras virtual, ou processos com acumuladores de valores em cálculos estatísticos, entre diversas outras situações.

Singleton

Assume esse comportamento quando utiliza apenas uma instância por JVM.
Ele é utilizado quando queremos compartilhar dados entre todos os usuários conectados ao aplicativo, mesmo que utilizando múltiplas JVMs, nos ambientes distribuídos.

Temos que lembrar sempre que essa é uma tecnologia corporativa, e que a execução de forma clusterizada não é uma exceção em sistemas de missão crítica.

Para trabalhar com EJBs no NetBeans devemos utilizar a opção de projeto corporativo, de acordo com os seguintes passos:

Atenção

Deixe marcadas as opções “Criar Módulo EJB” e “Criar Módulo de Aplicação Web”.

Ao final desses passos, teremos três projetos:

Quando trabalhamos com um projeto corporativo, devemos sempre implantar o projeto principal, com a extensão ear (Enterprise Archived), cujo ícone é um triângulo.

Qualquer tentativa de implantar os dois projetos secundários irá impedir a implantação correta do conjunto, exigindo que seja feita a remoção manual dos projetos implantados anteriormente pela aba de **Serviços**.

Agora que temos um projeto corporativo, podemos criar nosso primeiro **Session Bean**, que será do tipo **Stateless**. Ele será criado no projeto secundário **EJBTeste001-ejb**, adicionando novo arquivo e seguindo os seguintes passos:

Devemos alterar a interface local para o Session Bean, de forma a expressar os métodos que serão expostos para o usuário. No caso, vamos apenas definir os métodos **somar** e **subtrair**.

Exemplo:

```
package ejbs;

import javax.ejb.Local;

@Local
public interface CalculadoraLocal {
   int somar(int a, int b);
   int subtrair(int a, int b);
}
```

É importante observar a anotação **@Local**, do tipo **javax.ejb.Local**, imediatamente antes da definição da interface, pois é isso que a define como um acesso aos serviços do EJB de forma local. Se fosse uma interface remota, a anotação seria **@Remote**.

Após a definição dos métodos na interface, devemos implementá-los no EJB, como podemos observar a seguir.

Exemplo:

```
package ejbs;

import javax.ejb.Stateless;

@Stateless
public class Calculadora implements CalculadoraLocal {

   @Override
   public int somar(int a, int b) {
return a + b;
   }

   @Override
   public int subtrair(int a, int b) {
return a - b;
   }
}
```

Comentário

É possível notar a presença da anotação **@Stateless** na definição da classe, justamente o que faz com que o Application Server utilize Calculadora como um Session Bean do tipo Stateless.

Para testar nosso novo EJB, podemos utilizar um **Servlet** no projeto **EJBTeste001-war**, sendo o caminho mais simples para a utilização desse tipo de componente.

Vamos adicionar um Servlet ao projeto, com o nome **ServCalc** e pacote **servlets**, e modificar seu código para o que é apresentado a seguir.

```
package servlets;

import ejbs.CalculadoraLocal;
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "ServCalc", urlPatterns = {"/ServCalc"})
public class ServCalc extends HttpServlet {

 @EJB
 CalculadoraLocal calculadora;
 protected void doPost(HttpServletRequest request,
HttpServletResponse response)
 throws ServletException, IOException {
 response.setContentType("text/html;charset=UTF-8");
 try (PrintWriter out = response.getWriter()) {
  int a = new Integer(request.getParameter("A"));
  int b = new Integer(request.getParameter("B"));
  out.println("<!DOCTYPE html>");
  out.println("<html><body>");
  out.println("Soma: "+calculadora.somar(a, b));
  out.println("Subtração: "+calculadora.subtrair(a, b));
  out.println("</body>/htmlv");
 }
 }
}
```

Para efetuar a ligação com o pool de EJBs através da interface local, basta utilizar a anotação @EJB e definir um atributo para a recepção da interface.

```
@EJB
CalculadoraLocal calculadora;
```

Depois é só utilizar os métodos como se fossem chamadas comuns. Na verdade, a cada chamada é executada uma solicitação ao pool de objetos para que um deles efetue a operação solicitada e retorne o resultado.

```
out.println("Soma: "+calculadora.somar(a, b));
out.println("Subtração:
"+calculadora.subtrair(a, b));
```

Finalmente, devemos chamar o Servlet, o que pode ser feito com uma pequena modificação na página index, como pode ser observado a seguir.

Exemplo:

```
<html>
 <body>
  <form action="ServCalc" method="post">
   <input type="text" name="A"/>
   <input type="text" name="B"/>
   <input type="submit" value="Enviar"/>
  </form>
 </body>
 </html >
<html>
```

Para testar todos esses diversos passos, devemos selecionar o projeto principal (**EJBTeste001**), identificado pelo ícone do triângulo, e mandar executar, sendo apresentada a janela de index, como podemos observar a seguir.

Preenchendo os valores e clicando em Enviar, teremos o resultado das operações efetuadas por intermédio do EJB denominado Calculadora.

Existe outro tipo de EJB denominado Message Driven Bean (**MDB**), que tem como finalidade a recepção de dados a partir de mensagerias[1] através do middleware JMS, trazendo a possibilidade de processamento assíncrono ao ambiente JEE.

Podem apresentar dois tipos de comportamento:

Comentário

Com o uso de mensagerias, o cliente pode enviar a mensagem mesmo que o servidor não esteja ativo no momento, pois as mensagens ficam guardadas na mensageria até que seja processada pelo servidor.
Além disso, após a mensagem ser enviada para a mensageria, o cliente continua normalmente seu processamento, sem esperar a resposta do servidor, o qual tratará sequencialmente as mensagens recebidas, definindo um comportamento assíncrono.

Na figura abaixo, podemos ver um modelo muito importante para a rede bancária, onde transações como DOC e TED são enviadas de um banco para outro através de mensagerias.

O cliente recebe um comprovante da solicitação da transação, mas não tem que esperar pela compensação, que poderá ocorrer alguns minutos ou horas depois.

Há diversos fornecedores de mensagerias no mercado, como JBoss MQ, IBM MQ Series, Microsoft MQ e Open MQ. Para acessar todas elas de forma unívoca, utilizaremos o middleware Java Message Service (**JMS**).

Com a configuração da conexão com a fila ou tópico, através de JMS, o componente MDB estará apto a tratar as mensagens que chegam através de seu único método (**onMessage**), como podemos observar no exemplo seguinte.

Exemplo:

```
@JMSDestinationDefinition(name = "java:app/jms/Fila00001",
interfaceName = "javax.jms.Queue", resourceAdapter = "jmsra",
destinationName = "Fila00001")
@MessageDriven(activationConfig = {
@ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "java:app/jms/Fila00001"),
@ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue")
})
public class Mensageiro001 implements MessageListener {
 public Mensageiro001() {}
 @Override
 public void onMessage(Message message) {
   try {
   System.out.println(((TextMessage)message).getText());
  } catch (JMSException ex) {}
 }
}
```

Algo importante acerca do MDB é que ele foi projetado exclusivamente para receber mensagens a partir de filas ou tópicos, e não pode ser acionado diretamente, como os EJBs do tipo Session Bean.

No J2EE, existia um EJB para persistência denominado **EntityBean**, que seguia o padrão Active Record, o qual, no entanto, se mostrou inferior a alguns frameworks de persistência em termos de eficiência, sendo substituído pelo JPA no JEE5.

Atividades

1. Em termos de programação para banco de dados, em linguagens orientadas a objetos, é bastante aconselhável efetuar uma transformação dos registros para instâncias de classes de entidade. Esse processo ficou conhecido como:

a) ORM.
b) POO.
c) RUP.
d) UML.
e) SQL.

2. No uso de JPA, qual a classe que fica responsável por inserir os dados da entidade no banco de dados, através do método persist?

a) EntityManagerFactory
b) Query
c) CriteriaQuery
d) NamedQuery
e) EntityManager

3. Existem diversos tipos de EJBs, mas um deles é voltado exclusivamente para o tratamento de mensagens recebidas a partir de mensagerias. Qual é esse tipo de EJB?

a) Stateless Session Bean
b) Message Driven Bean
c) Stateful Session Bean
d) Singleton Bean
e) Entity Bean

Notas

Mensagerias[1]

As mensagerias são canais de comunicação seguros e robustos para envio de mensagens.

Referências

CASSATI, J. P. **Programação servidor em sistemas web.** Rio de Janeiro: Estácio, 2016.

DEITEL, P.; DEITEL, H. **Ajax**, rich internet applications e desenvolvimento web para programadores. São Paulo: Pearson Education, 2009.

_____. **Java**, como programar. 8. ed. São Paulo: Pearson, 2010.

MONSON-HAEFEL, R.; BURKE, B. **Enterprise Java Beans** 3.0. 5. ed. São Paulo: Pearson, 2007.

Próxima aula

- Mapeamento objeto-relacional com JPA.
- Componentes EJB como Facade.
- Sistema MVC completo com EJB, JPA e JSF.

Explore mais

Leia os textos:

- Tutorial: JPA; <https://www.tutorialspoint.com/jpa>
- Sintaxe JPQL; <https://thoughts-on-java.org/jpql/>
- Noções gerais de Enterprise Java Beans; <https://docs.oracle.com/javaee/6/tutorial/doc/gijsz.html>
- Message Oriented Middleware. <https://docs.oracle.com/cd/E19316-01/820-6424/aeraq/index.html>