

Prof. Emílio Celso de Souza

Sumário

Introdução	4
Módulo 1 - Introdução ao Hibernate	5
Aula 1 – Conceitos fundamentais sobre o Hibernate	5
1.1 Definição, vantagens e desvantagens	5
1.2 Funcionamento e Arquitetura Básica	6
Aula 2 - Classes Persistentes	7
2.1 Definição de classes persistentes	7
2.2 Escolha de Chaves Primárias	8
2.3 Objetos Persistentes, Transientes e <i>Detached</i>	8
2.4 SQL Dialeto	8
Aula 3 - Conhecendo o HQL – Hibernate Query Language	9
3.1 Definição de HQL	9
Aula 4 - Desenvolvimento de uma Aplicação	10
4.1 Definição dos passos para o desenvolvimento de uma aplicação com Hibernate	10
4.2 Passo 1: Criando o Banco de Dados e o projeto	10
4.3 Passo 2: O arquivo hibernate.cfg.xml	11
4.4 Passo 3: Definindo a classe HibernateUtil.java	11
4.5 Passo 4: Criando as classes Java (POJO) e os arquivos de mapeamento do Hibernate	12
4.6 Passo 5: Definindo a classe para manipular os dados dos usuários	15
4.7 Passo 6: Inserindo um novo registro e Recuperando todos os usuários cadastrados	16
Aula 5 - Tornando um objeto persistente	17
5.1 Como tornar um objeto persistente	17
5.2 Considerações ao salvar um objeto	18
5.3 Carregando um objeto: métodos load() e get()	20
Aula 6 - Consultas	21
6.1 Executando consultas – métodos list() e uniqueResult()	21
6.2 Iterando resultados – método iterate()	22
6.3 Resultados escalares	23
6.4 Parâmetros nomeados	23
6.5 Paginação – métodos setFirstResult() e setMaxResults()	24
Aula 7 - Usando Criteria	24
7.1 A interface Criteria	24
7.2 Ordenando os resultados	26
7.3 Consultas em SQL nativo	27
Aula 8 – Outras operações com objetos persistentes	27
8.1 Alterando objetos persistentes	27
8.2 Removendo objetos persistentes	28
8.3 Replicando objetos entre dois bancos diferentes – método replicate()	28
8.4 Atualizando o Session: método flush()	29
Aula 9 - Mapeamento	29
9.1 Exemplo: as classes Person e Event	30
9.2 Associação baseada em conjunto - Set	32
9.3 Manipulando a associação	33

Exercícios do Módulo 1	36
Exercício 1	36
Exercício 2	45
Módulo 2 - Java Persistence API (JPA)	47
Caro aluno, JPA (Java Persistence API) é definida no padrão JSR-220 (<i>Enterprise JavaBeans</i> , versão 3.0), que padroniza o mapeamento objeto/relacional (ORM) na plataforma Java. A JPA é baseada no modelo POJO e os objetos persistentes são denominados entidades (Entity). Uma entidade é um POJO, que representa um conjunto de dados persistidos no banco.	47
Aula 10 - Annotations	47
10.1 A API de persistência	47
10.2 Exemplo de Aplicação	48
Aula 11 - Anotações JPA	55
11.1 @Entity	55
11.2 @Table	56
11.3 @Column	56
11.4 @Id	57
11.5 @GeneratedValue	57
11.6 @SequenceGenerator	57
11.7 @Transient	58
11.8 @Temporal	58
11.9 @PersistenceContext	59
11.10 @OneToMany	59
11.11 @ManyToOne	60
11.12 @ManyToMany	61
11.13 @Embeddable (Chave Composta)	61
Aula 12 - API Entity Manager	62
Exercícios do Módulo 2	64
Exercício 1	64
Exercício 2	71
Módulo 3 – Conceitos do Enterprise JavaBeans (EJB)	72
Aula 13 - Introdução ao EJB	72
13.1 Definição e Arquitetura do EJB	72
13.2 Componentes JEE – Camada Cliente	73
13.3 Características dos EJBs	74
13.4 Servidores de Aplicações para suporte EJB	74
13.5 A API JEE	75
Aula 14 – A plataforma <i>Enterprise JavaBeans</i>	75
14.1 Definição da Plataforma EJB	75
14.2 EJB como um componente	76
14.3 Entendendo os tipos EJB	77
14.4 Beans de sessão (<i>Session beans</i>)	77
14.5 Beans dirigidos por mensagens (<i>Message Driven Beans</i>)	78
14.6 Entidades e Java Persistence API (<i>Entity Beans</i>)	78
14.7 Resumo dos Componentes EJB	78
Aula 15 - Utilização completa de EJB	79

15.1 Exemplo de Utilização	79
15.2 Criando um Entity Bean	80
15.3 Adicionando anotações JPA	82
15.4 Criando interfaces local e remota	83
Aula 16 - Preparação do Eclipse para executar a versão 7.x do servidor JBoss	86
16.1 Instruções passo a passo	86
Exercícios do Módulo 3	94
Exercício 1	94
Exercício 2	100
Exercício 3	106
Exercício 4	111
Considerações Finais	120
Respostas Comentadas dos Exercícios	121
Exercícios do Módulo 1	121
Exercício 1	121
Exercício 2	129
Exercícios do Módulo 2	137
Exercício 1	137
Exercício 2	143
Exercícios do Módulo 3	149
Exercício 1: Componentes de servidor e cliente no mesmo projeto.	149
Exercício 2: Calculadora usando EJB e JSF	156
Exercício 3	162
Exercício 4	165
Referências	175

Introdução

Prezados alunos, é chegado o momento de apresentarmos um dos tópicos mais interessantes no mundo do desenvolvimento Java: os conceitos de persistência de dados em componentes locais ou remotos.

Nesta disciplina discorreremos sobre os mecanismos de acesso a dados sem necessariamente acessar um banco de dados. Parece estranho, não? Teremos a oportunidade de manipular objetos como se estivéssemos manipulando um banco de dados. Em outras palavras, teremos a oportunidade de alterar atributos de um objeto e essa alteração refletir no banco de dados. Neste ponto trataremos um objeto como uma entidade, ou seja, ele representará um registro no banco de dados.

Estamos falando do Hibernate! O Hibernate é um *framework* que permite quaisquer aplicações Java, seja *desktop*, *web* ou *webservices*, manipularem um banco de dados através de objetos devidamente mapeados para o banco de dados em questão.

Uma vez entendido o mecanismo de acesso a dados baseado no Hibernate, estudaremos o JPA (Java Persistence API), que na verdade é uma evolução do Hibernate no modo como os objetos são mapeados.

Outro tema que abordaremos nesta disciplina é o conceito dos *Enterprise JavaBeans* (EJB). Um EJB é um componente que permite acesso remoto por aplicações Java e não Java.

Suponha que estejamos desenvolvendo uma aplicação que deverá enviar solicitações a outra aplicação sem necessariamente ter suas classes copiadas para o projeto atual. Como exemplo, consideremos uma aplicação de faturamento acessando outra aplicação, em outro domínio, que lida com informações de cartões de crédito. A troca de informações entre as aplicações ocorre através de EJBs, e quando for necessário o acesso aos dados da aplicação remota, o acesso é realizado através do JPA.

Como você pode perceber, temos um tema muito importante a ser estudado! Sabemos que o acesso aos dados é uma tarefa imprescindível em qualquer aplicação e, quando o volume de informações aumenta, o Hibernate se torna indispensável na manipulação desses dados.

Espero que esta disciplina feche com chave de ouro o ciclo de desenvolvimento de aplicações em Java, e que aproveitem ao máximo os ensinamentos que passarei para vocês!

Prof. Emilio

Módulo 1 - Introdução ao Hibernate

Aula 1 – Conceitos fundamentais sobre o Hibernate

1.1 Definição, vantagens e desvantagens

O Hibernate é um *framework* de acesso a banco de dados escrito em Java. Além de realizar o mapeamento objeto-relacional, permite a consulta a banco de dados, contribuindo para uma redução considerável no tempo de desenvolvimento da aplicação. Sendo assim, ele foi criado para reduzir o tempo que o desenvolvedor consome nas atividades relacionadas à persistência de dados no desenvolvimento de um software orientado a objetos.



Fonte: <http://design.jboss.org/hibernate/logo/final/hibernate_logo_whitebkg_stacked_256px.png>. Acesso em: 07 nov. 2014.

Dentre suas vantagens, podemos citar:

- Utiliza o modelo natural de programação orientada a objetos (POO);
- Acelera o desenvolvimento;
- Permite uma transparência ao banco de dados;
- Boa performance (cache);
- Simples de implementar;
- É compatível com os principais bancos de dados de mercado.

Mas há, também, algumas desvantagens, como:

- Não é a melhor opção para todos os tipos de aplicação. Sistemas que fazem uso extensivo de *stored procedures*, *triggers* ou que implementam a maior parte da lógica de negócio no banco de dados não vão se beneficiar do Hibernate. Ele é indicado para sistemas que contam com um modelo rico,

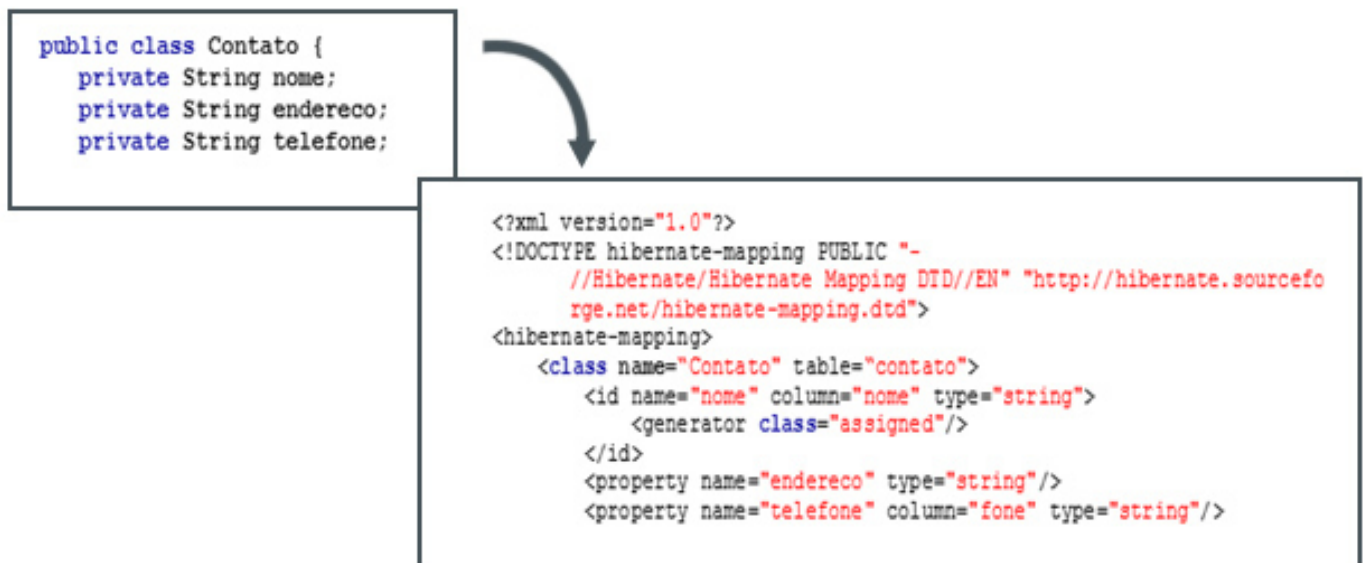
em que a maior parte da lógica de negócios fica na própria aplicação, dependendo pouco de funções específicas do banco de dados.

- Consultas muito complexas e extensas exigem grau avançado de conhecimento da API.

1.2 Funcionamento e Arquitetura Básica

Simplificadamente tem-se uma classe Java mapeada para uma tabela no banco de dados através de um arquivo XML ou de anotações. Veja o exemplo da Figura 1.2a.

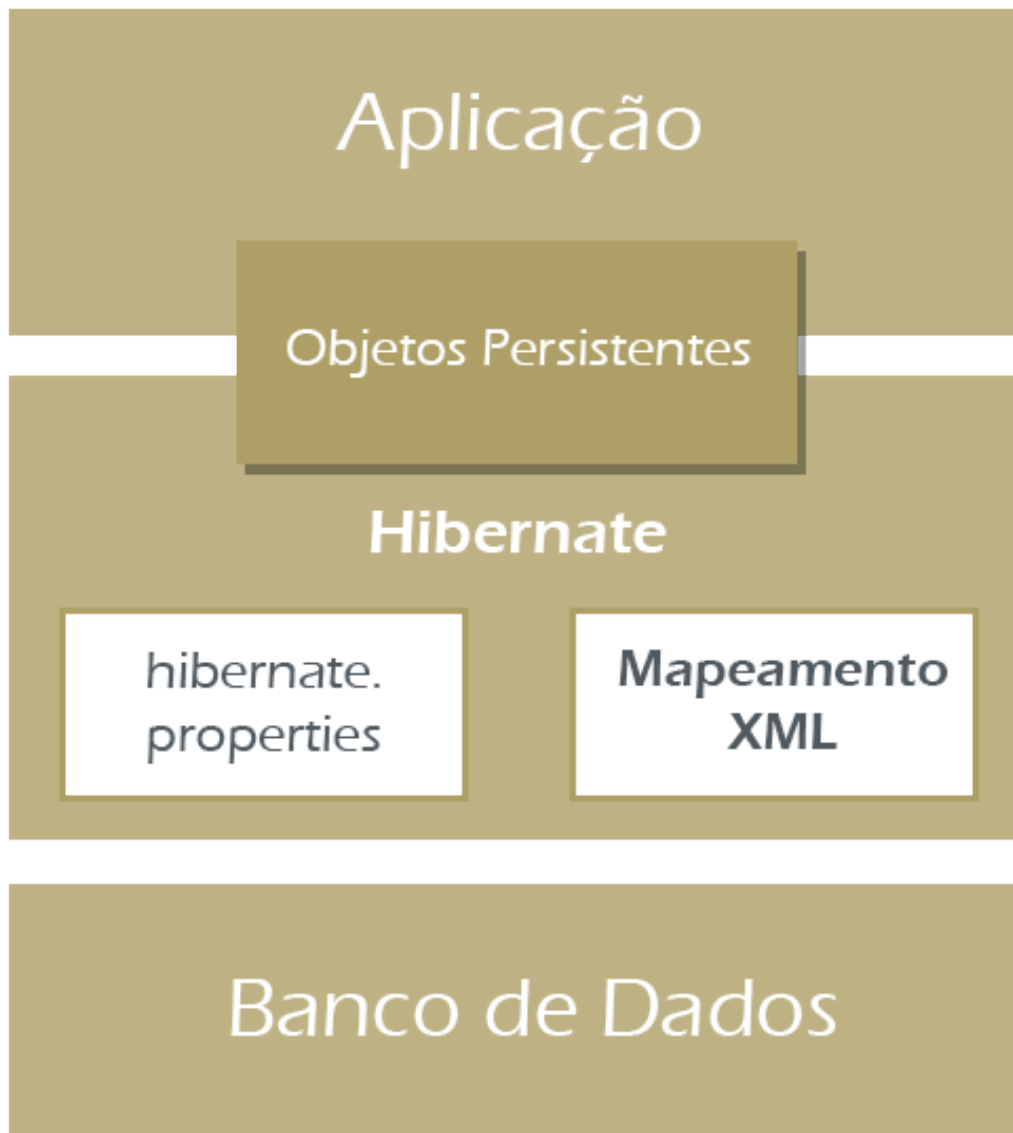
Figura 1.2a – Mapeamento objeto-relacional



A arquitetura do Hibernate é constituída por um conjunto de interfaces. A camada de negócio aparece acima da camada de persistência por atuar como uma cliente da camada de persistência. A Figura 1.2b apresenta a arquitetura básica do Hibernate.

As principais interfaces utilizadas no Hibernate são:

- **Session** (org.hibernate.Session): Possibilita a comunicação entre a aplicação e a camada de persistência, por meio de uma conexão JDBC. Com ela é possível criar, remover, atualizar e recuperar objetos persistentes.
- **SessionFactory** (org.hibernate.SessionFactory): Mantém o mapeamento objeto-relacional em memória. Permite a criação de objetos Session, a partir dos quais os objetos são acessados. Também denominado de "fábrica de objetos Session".
- **Configuration** (org.hibernate.Configuration): É utilizado para realizar as tarefas de inicialização do Hibernate.
- **Transaction** (org.hibernate.Transaction): É utilizada para representar uma unidade indivisível de uma operação de manipulação de dados.
- **Criteria** (org.hibernate.Criteria) e **Query** (org.hibernate.Query): utilizadas para realizar consultas ao banco de dados.

Figura 1.2b – Arquitetura do Hibernate

Fonte: <<https://docs.jboss.org/hibernate/orm/3.5/reference/pt-BR/html/images/overview.png>>. Acesso em: 04 nov. 2014.

Aula 2 - Classes Persistentes

2.1 Definição de classes persistentes

Segundo Fernandes e Lima (2007, p. 10-2):

As classes persistentes de uma aplicação são aquelas que implementam as entidades do domínio de negócio. O Hibernate trabalha associando cada tabela do banco de dados a um POJO (Plain Old Java Object). É importante considerar:

- O Hibernate requer que toda classe persistente possua um construtor padrão sem argumentos.
- O Hibernate persiste as propriedades no estilo *JavaBeans*, utilizando *getters* e *setters*.

- Ao utilizar Hibernate, todos os objetos persistentes devem possuir um identificador e que eles sejam independentes da lógica de negócio da aplicação.

2.2 Escolha de Chaves Primárias

Um passo importante ao utilizar o Hibernate é informá-lo sobre a estratégia utilizada para a geração de chaves primárias das tabelas.

Uma chave é candidata é uma coluna ou um conjunto de colunas que identifica unicamente uma linha de uma tabela do banco de dados. Ela deve satisfazer as seguintes propriedades:

- Única;
- Nunca ser nula;
- Constante.

Muitas aplicações utilizam chaves naturais como chaves primárias, ou seja, que têm significados de negócio.

2.3 Objetos Persistentes, Transientes e *Detached*

Vamos entender o mecanismo de persistência do Hibernate. Existem três tipos de objetos: objetos *transient* (transientes), *detached* (desligados) e *persistent* (persistentes). Estes objetos são assim descritos:

- **Objetos Transientes:** são aqueles que ainda não têm uma representação no banco de dados (ou que foram excluídos); eles ainda não estão sob o controle do *framework* e podem não ser mais referenciáveis a qualquer momento, como qualquer objeto normal em Java.
- **Objetos Persistentes:** são objetos que suas instâncias estão associadas a um contexto persistente, ou seja, têm uma identidade de banco de dados.
- **Objetos *detached*:** têm uma representação no banco de dados, mas não fazem mais parte de uma sessão do Hibernate, o que significa que o seu estado pode não estar mais sincronizado com o banco de dados.

É comum, para fins de testes ou de desenvolvimento, executar um programa no servidor, sendo o servidor o próprio computador cliente. Ou seja, não é necessário ter uma infraestrutura complexa para testar uma aplicação.

2.4 SQL Dialeto

Dialeto SQL possibilitam que o Hibernate tire proveito de características próprias do banco de dados.

Devem ser configurados utilizando o nome completo de uma subclasse de net.sf.hibernate.dialect.
Dialect

Exemplo: `hibernate.dialect=net.sf.hibernate.dialect.MySQLDialect`

(Essa configuração é feita dentro do arquivo hibernate.cfg.xml)

Aula 3 - Conhecendo o HQL – Hibernate Query Language

3.1 Definição de HQL

HQL - Hibernate Query Language é um mecanismo do Hibernate para definir instruções SQL em tempo de execução. Ao escrever HQL, podemos nos concentrar nos objetos e nas propriedades sem o conhecimento detalhado do banco de dados. Podemos tratar o HQL como uma variação orientada a objetos do SQL.

Como exemplo, podemos usar o comando HQL seguinte para buscar uma lista de livros. Observe o método **list()** utilizado no Quadro 1.

Quadro 1: Exemplo de HQL

```
Query query = session.createQuery("from Book");  
List books = query.list();
```

A interface Query fornece também dois métodos para recuperarmos somente um subconjunto do resultado, como mostra o Quadro 2.

Quadro 2: Exemplo de HQL com restrições

```
Query query = session.createQuery("from Book");  
query.setFirstResult(20);  
query.setMaxResults(10);  
List books = query.list();
```

Podemos, também, especificar parâmetros da mesma forma que o fazemos para instruções SQL. Se estivermos certos de que existe apenas um objeto retornado, podemos usar o método `uniqueResult()` para recuperá-lo. Se nenhuma informação for encontrada, o valor `null` é retornado. Veja o exemplo do Quadro 3.

Quadro 3: Exemplo de HQL com parâmetros

```
Query query = session.createQuery("from Book where isbn = ?");  
query.setString(0, "1932394419");  
Book book = (Book) query.uniqueResult();
```

No exemplo do Quadro 3, usamos "?" para representar um parâmetro de consulta e configurá-lo pelo índice, iniciando em zero, diferentemente do JDBC, que inicia com um. Esse tipo de parâmetro é chamado de "parâmetro posicional". Em HQL podemos usar também os "parâmetros nomeados". A vantagem é que eles são fáceis de entender e podem ser usados várias vezes. Veja o exemplo do Quadro 4.

Quadro 4: Exemplo de HQL com parâmetros nomeados

```
Query query = session.createQuery("from Book where isbn = :isbn");
query.setString("isbn", "1932394419");
Book book = (Book) query.uniqueResult();
```

Existe muito mais sobre HQL para ser explorado. Neste material é apresentada apenas sua visão geral. Mas há muito material disponível sobre o HQL, que pode ser solicitado ao professor ou obtido de <www.hibernate.org>.

Aula 4 - Desenvolvimento de uma Aplicação

4.1 Definição dos passos para o desenvolvimento de uma aplicação com Hibernate

O desenvolvimento de uma aplicação com Hibernate inclui os passos:

1. Criar o banco de dados;
2. Definir o arquivo de configurações do *framework*, com suas propriedades;
3. Gerar uma classe contendo os objetos *SessionFactory* e *Session*, dentre outros;
4. Criar as classes *JavaBeans* e os arquivos de mapeamento;
5. Desenvolver uma classe que crie os objetos do *framework*;
6. Desenvolver uma aplicação para testar as funcionalidades do *framework*;

A seguir estes passos são apresentados para a elaboração de uma aplicação. Podem ocorrer variações entre as aplicações, mas o desenvolvimento central é baseado na descrição aqui apresentada.

4.2 Passo 1: Criando o Banco de Dados e o projeto

Considere a tabela pessoa mostrada na Figura 4.2, do banco de dados chamado *introhibernate*, desenvolvido a partir do MySQL.

Figura 4.2 – Tabela PESSOA

PESSOA
ID
NOME
APELIDO
ENDEREÇO

Para criar um projeto usando Hibernate, é necessário obter a API correspondente. O *site* oficial é <<http://www.hibernate.org/>>.

Os arquivos .jar obtidos são inseridos no projeto, além do driver de acesso a dados (mysql, como mencionado anteriormente).

4.3 Passo 2: O arquivo hibernate.cfg.xml

O Hibernate possui um arquivo de configurações, normalmente incluído na raiz das classes (pasta src) chamado hibernate.cfg.xml. Este arquivo define os parâmetros de configuração do banco de dados a ser mapeado na aplicação. Para o nosso exemplo, o conteúdo do arquivo é o mostrado no Quadro 5.

Quadro 5: O arquivo hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://
hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/introhibernate</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">password</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.current_session_context_class">
      thread</property>
    <mapping resource="br/com/ead/Pessoa.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

4.4 Passo 3: Definindo a classe HibernateUtil.java

A utilização do Hibernate requer uma classe que promova a inicialização e que acesse o SessionFactory para obter um objeto Session. A classe executa o método configure(), que carrega as informações do arquivo hibernate.cfg.xml e, então, cria o SessionFactory para obter o objeto Session. Veja o Quadro 6.

Quadro 6: A classe HibernateUtil

```
package br.com.ead;

import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;

public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

4.5 Passo 4: Criando as classes Java (POJO) e os arquivos de mapeamento do Hibernate

As classes *JavaBeans* são usadas para representar os registros em uma tabela do banco de dados.

Os dados são obtidos e/ou lidos do banco de dados por meio de getters e setters. O mapeamento das classes para as tabelas é realizado por meio do arquivo de mapeamento do Hibernate. Em geral, tem-se um arquivo para cada classe e uma classe representando uma tabela. Como exemplo, veja a Classe Pessoa (POJO) mostrada no Quadro 7.

Quadro 7: Entidade Pessoa

```
package br.com.ead;

public class Pessoa implements java.io.Serializable {

    private int id;
    private String nome;
    private String apelido;
    private String endereco;

    public Pessoa() {
    }

    public Pessoa(int id) {
        this.id = id;
    }

    public Pessoa(int id, String nome, String apelido, String endereco) {
        this.id = id;
        this.nome = nome;
        this.apelido = apelido;
        this.endereco = endereco;
    }

    public int getId() {
        return this.id;
    }

    public void setId(int id) {
        this.id = id;
    }

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

```
public void setNome(String nome) {  
    this.nome = nome;  
}  
public String getApelido() {  
    return this.apelido;  
}  
  
public void setApelido(String apelido) {  
    this.apelido = apelido;  
}  
public String getEndereco() {  
    return this.endereco;  
}  
  
public void setEndereco(String endereco) {  
    this.endereco = endereco;  
}  
}
```

O mapeamento objeto-relacional (ORM) da classe Pessoa é apresentado no Quadro 8: **Pessoa.hbm.xml**.

Quadro 8: Arquivo de mapeamento Pessoa.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="br.com.ead.Pessoa" table="pessoa" catalog="introhibernate">
    <id name="id" type="int">
      <column name="ID" />
      <generator class="assigned" />
    </id>
    <property name="nome" type="string">
      <column name="NOME" length="45" />
    </property>
    <property name="apelido" type="string">
      <column name="APELIDO" length="45" />
    </property>
    <property name="endereco" type="string">
      <column name="ENDERECO" length="45" />
    </property>
  </class>
</hibernate-mapping>
```

4.6 Passo 5: Definindo a classe para manipular os dados dos usuários

Já temos a classe que nos fornece o **SessionFactory**, o arquivo de configurações do Hibernate, a classe POJO e o arquivo de mapeamento para esta classe.

Agora estamos prontos para definirmos a classe que de fato fará o trabalho. Essa classe é normalmente chamada de Classe **Helper**, e recomendamos nomeá-la desta forma. Por exemplo, no caso da tabela de usuários, chamaremos **PessoasHelper**. Veja o Quadro 9.

Quadro 9: A classe Helper para execução da transação no Hibernate

```
package br.com.ead;
import org.hibernate.*;
import java.util.*;

public class PessoasHelper {
    Session session = null;

    public PessoasHelper(){
        session = HibernateUtil.getSessionFactory().getCurrentSession();
    }

    public List<Pessoa> getPessoas(){
        List<Pessoa> lista = new ArrayList<Pessoa>();
        try{ Transaction tx = session.beginTransaction();
        Query q = session.createQuery("from Pessoa");
        lista = q.list();
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
        return lista;
    }

    public String salvarPessoa(Pessoa p){
        try{ Transaction tx = session.beginTransaction();
        session.save(p);
        tx.commit();
        return "Dados inseridos";
        }
        catch(Exception ex){
            return ex.getMessage();
        }
    }
}
```

4.7 Passo 6: Inserindo um novo registro e Recuperando todos os usuários cadastrados

Usando código Java, inserir um registro significa inserir um novo objeto no Hibernate. Ele se encarrega de persisti-lo no Banco de Dados. Veja o exemplo mostrado no Quadro 10.

Quadro 10: Inclusão de um novo registro

```
PessoasHelper helper = new PessoasHelper();
Pessoa p = new Pessoa();
p.setId(100);
p.setNome("Ze");
p.setApelido("ze");
p.setEndereco("aclimacao");

out.print(helper.salvarPessoa(p));
```

No exemplo do Quadro 11 mostraremos os nomes de todos os usuários. Para tanto, utilizaremos o método que retorne um List:

Quadro 11: Recuperando registros

```
PessoasHelper helper = new PessoasHelper();

List<Pessoa> lista = helper.getPessoas();
out.print("Lista de Usuários<br/>");
for(Pessoa pessoa:lista){
    out.print(pessoa.getNome() + "<br/>");
}
```

Aula 5 - Tornando um objeto persistente

5.1 Como tornar um objeto persistente

Uma nova instância de um objeto persistente é considerada transient pelo Hibernate, e ela se torna persistente quando associada a um Session, veja o Quadro 12.

Quadro 12: Persistindo uma entidade

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
sess.save(fritz);
```

Se a classe Cat possuir um identificador gerado automaticamente, ele é gerado e atribuído ao objeto quando o método `save()` é executado.

Alternativamente, pode-se atribuir o identificador usando-se uma versão sobrecarregada de `save()`, como no Quadro 13.

Quadro 13: Incluindo um elemento na coleção

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

Observe que o objeto persistente possui outros objetos associados (kittens, por exemplo).

5.2 Considerações ao salvar um objeto

O Hibernate possui um conjunto de métodos para incluir e atualizar valores no banco de dados. Os métodos se parecem bastante, mas possuem diferenças:

save() – armazena um objeto no banco de dados. Isso significa que, se o registro (chave primária) não existir, ele será incluído, caso contrário, lançará uma exceção.

update() – utilizado para atualizar o objeto através do seu identificador. Se não existir, o método lança uma exceção.

saveOrUpdate() – Este método chama `save()` ou `update()` dependendo da operação. Se o identificador existir, será chamado `update()`; senão será chamado o método `save()`. O Quadro 14 apresenta um exemplo.

Quadro 14: Exemplo de Hibernate

```
package hibernate;
import org.hibernate.*;

public class HibernateExample {
    public static void main(String args[]){
        Configuration configuration = new Configuration();
        SessionFactory sessionFactory = configuration
            .configure()
            .buildSessionFactory();
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();
        transaction.begin();
        EmployeeInfo employeeInfo = new EmployeeInfo();
        employeeInfo.setSno(1);
        employeeInfo.setName("HibernateTestSave");
        session.save(employeeInfo);
        transaction.commit();
        session.close();

        session = sessionFactory.openSession();
        transaction = session.beginTransaction();
        transaction.begin();
        employeeInfo = new EmployeeInfo();
        employeeInfo.setSno(1);
        employeeInfo.setName("HibernateTestUpdate");
        session.update(employeeInfo);
        transaction.commit();
        session.close();

        session = sessionFactory.openSession();
```

```

transaction = session.beginTransaction();
transaction.begin();
employeeInfo = new EmployeeInfo();
employeeInfo.setSno(1);
employeeInfo.setName("HibernateTestSaveOrUpdate");
session.saveOrUpdate(employeeInfo);
transaction.commit();

session.close();
}
}

```

5.3 Carregando um objeto: métodos load() e get()

O método load() fornece um meio de se recuperar uma instância persistente se o identificador é conhecido. load() toma a classe e carrega seu estado em uma nova instância desta classe, em um estado persistente, como mostra o Quadro 15a.

Quadro 15a: Recuperando uma entidade com o método load()

```

Cat fritz = (Cat) sess.load(Cat.class, generatedId);
// é necessário empacotar os dados primitivos
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );

```

Alternativamente, pode-se recuperar o estado de um objeto em uma dada instância, como no Quadro 15b.

Quadro 15b: Recuperando o estado de um objeto

```

Cat cat = new DomesticCat();
// carrega o estado de pk em um objeto Cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();

```

O método **load()** lançará uma exceção se não houver um registro válido com a chave informada. Se não houver a certeza de que exista o registro, é recomendável usar o método **get()**, que acessa o banco e retorna *null* se não houver um registro correspondente. Veja o Quadro 16.

Quadro 16: Alterando uma entidade

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

Aula 6 - Consultas

6.1 Executando consultas – métodos `list()` e `uniqueResult()`

Se o identificador do objeto não for conhecido, uma consulta (query) será necessária. Sabe-se que o Hibernate utiliza instruções HQL (Hibernate Query Language). Para a criação de consultas geradas programaticamente, é possível utilizar instruções SQL nativas do banco de dados utilizado, com o suporte adicional do Hibernate para a conversão de ResultSets em objetos.

A estrutura HQL e as consultas SQL são representadas por uma instância de Query. Essa interface oferece métodos para ações de consultas. Pode-se sempre obter uma Query usando-se Session atual, veja o Quadro 17.

Quadro 17: Listando entidades

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

Uma consulta é executada geralmente pela chamada ao método **list()**. Este resultado é atribuído a uma coleção. O método **uniqueResult()** fornece um atalho se o usuário souber que a consulta retorna um único resultado.

6.2 Iterando resultados – método **iterate()**

É possível obter uma melhor performance executando a *query* usando o método **iterate()**. Isso normalmente acontecerá quando se desejar que as instâncias retornadas pela *query* já estiverem em sessão ou em cache. Se ainda não estiverem em cache, o método será mais lento que **list()** e realizará múltiplos acessos ao banco de dados para uma simples consulta. Veja o Quadro 18.

Quadro 18: Iterando sobre entidades

```
// buscando ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeliness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next(); // busca o objeto
    // algo que não existe na query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // remove a instancia atual
        iter.remove();
        // continuação...
        break;
    }
}
```

6.3 Resultados escalares

Consultas podem especificar uma propriedade de uma classe na cláusula *select*, incluindo funções agregadas. Elas são chamadas propriedades escalares e são entidades com estados persistentes. Veja o Quadro 19.

Quadro 19: Retornado entidades em *arrays*

```
Iterator results = sess.createQuery("select cat.color,
    min(cat.birthdate),
    count(cat) from Cat cat group by cat.color").list().iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

6.4 Parâmetros nomeados

É possível usar parâmetros nomeados ao invés de numerados pelos seus índices. As vantagens dessa abordagem são:

- podem ocorrer em qualquer ordem na consulta;
- podem ocorrer várias vezes;
- são autoexplicativos;

Veja o exemplo do quadro 20.

Quadro 20: Usando parâmetros nomeados

```
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();

Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();

List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

6.5 Paginação – métodos `setFirstResult()` e `setMaxResults()`

Se for necessário especificar limites ao conjunto selecionado, ou seja, o máximo número de registros, ou o primeiro registro que se deseja recuperar, podem-se usar os métodos mostrados no Quadro 21.

Quadro 21: Filtrando registros

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Aula 7 - Usando Criteria

7.1 A interface Criteria

Instruções HQL são extremamente úteis, mas é possível construir *queries* dinamicamente. O Hibernate disponibiliza a interface Criteria para esses casos. Os exemplos dos Quadros 22 e 23 ilustram seu uso.

Quadro 22: Usando Criteria

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

Quadro 23: Usando Criteria com restrições

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Restrictions.eq( "color", Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

Um critério individual é uma instância de **org.hibernate.criterion.Criterion**. A classe **org.hibernate.criterion.Restrictions** define métodos para a obtenção de tipos **Criterion**. Veja o Quadro 24.

Quadro 24: Listando entidades com Criteria

```
List cats = sess.createCriteria(Cat.class)
.add( Restrictions.like("name", "Fritz%") )
.add( Restrictions.between("weight", minWeight, maxWeight) )
.list();
```

As restrições podem ser agrupadas logicamente, como mostra o Quadro 25.

Quadro 25: Encadeamento de restrições

```

List cats = sess.createCriteria(Cat.class)
.add( Restrictions.like("name", "Fritz%") )
.add( Restrictions.or(
Restrictions.eq( "age", new Integer(0) ),
Restrictions.isNull("age")
) )
.list();

List cats = sess.createCriteria(Cat.class)
.add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
.add( Restrictions.disjunction()
.add( Restrictions.isNull("age") )
.add( Restrictions.eq("age", new Integer(0) ) )
.add( Restrictions.eq("age", new Integer(1) ) )
.add( Restrictions.eq("age", new Integer(2) ) )
) )
.list();

```

Pode-se obter um critério a partir de uma instância de Property. Um **Property** pode ser criado através da chamada **Property.forName()**, como mostra o Quadro 26.

Quadro 26: Usando propriedades nomeadas

```

Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
.add( Restrictions.disjunction()
.add( age.isNull() )
.add( age.eq( new Integer(0) ) )
.add( age.eq( new Integer(1) ) )
.add( age.eq( new Integer(2) ) )
) )
.add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" } ) )
.list();

```

7.2 Ordenando os resultados

É possível ordenar os resultados através de **org.hibernate.criterion.Order**. Veja o Quadro 27.

Quadro 27: Ordenando entidades

```
List cats = sess.createCriteria(Cat.class)
.add( Restrictions.like("name", "F%") )
.addOrder( Order.asc("name") )
.addOrder( Order.desc("age") )
.setMaxResults(50)
.list();

List cats = sess.createCriteria(Cat.class)
.add( Property.forName("name").like("F%") )
.addOrder( Property.forName("name").asc() )
.addOrder( Property.forName("age").desc() )
.setMaxResults(50)
.list();
```

7.3 Consultas em SQL nativo

Pode-se expressar uma consulta usando-se o método **createSQLQuery()**. É possível também chamar **session.connection()** e usar conexão JDBC diretamente. Veja o Quadro 28.

Quadro 28: Usando SQL nativo

```
List cats = session.createSQLQuery("SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10")
.addEntity("cat", Cat.class)
.list();

List cats = session.createSQLQuery(
"SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
"{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
"FROM CAT {cat} WHERE ROWNUM<10")
.addEntity("cat", Cat.class)
.list();
```

Aula 8 – Outras operações com objetos persistentes**8.1 Alterando objetos persistentes**

Instâncias transacionais persistentes podem ser manipuladas pela aplicação e quaisquer alterações no estado persistente serão persistentes quando a Session for atualizada (método **flush()** chamado). Não há necessidade de se chamar um método em particular, como **update()**. O meio mais direto de se fazer essa tarefa é executar o método **load()** e manipulá-lo diretamente enquanto Session estiver ativo, como mostra o Quadro 29.

Quadro 29: Alterando uma entidade

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush(); // alterações são realizadas automaticamente
```

8.2 Removendo objetos persistentes

O **Session.delete()** removerá um objeto do banco de dados. A aplicação, entretanto, ainda pode manter uma instância do objeto removido, como mostra o Quadro 30.

Quadro 30: Removendo uma entidade

```
session.delete(cat);
```

8.3 Replicando objetos entre dois bancos diferentes – método replicate()

É possível replicar um objeto persistente em dois bancos de dados. Veja o Quadro 31.

Quadro 31: Replicando entidades

```
//recuperando um cat de um banco
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//inserindo em um Segundo banco
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

O ReplicationMode determina como **replicate()** lidará com conflitos entre linhas existentes no banco de dados:

ReplicationMode.IGNORE: ignora o objeto quando existir um registro no banco com o mesmo identificador;

ReplicationMode.OVERWRITE: sobrescreve qualquer linha existente no banco com o mesmo identificador;

ReplicationMode.EXCEPTION: lança uma exceção se existir um registro com o mesmo identificador

ReplicationMode.LATEST_VERSION: sobrescreve o registro se o número da versão for anterior ao da versão sendo utilizada. Se forem iguais, o objeto é ignorado.

8.4 Atualizando o Session: método flush()

Às vezes o Session executará instruções SQL necessitando sincronizar conexões JDBC com os objetos mantidos na memória. Esse processo ocorre por default antes de quaisquer execuções de query:

- Pelo método commit()
- Pelo método flush()

Exceto quando se usa explicitamente flush(), não existem garantias sobre quando o Session executa chamadas JDBC, somente a ordem em que são executadas.

É possível alterar o comportamento default de modo que flush() ocorra menos frequentemente. A classe FlushMode define três diferentes modos: somente atualiza em tempo de commit na transação, automaticamente, ou nunca, a menos que flush() seja chamado automaticamente. Este último é útil para longas tarefas. Veja o Quadro 32.

Quadro 32: Buscando entidade

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT);

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

sess.find("from Cat as cat left outer join cat.kittens kitten");

...
tx.commit();
sess.close();
```

Aula 9 - Mapeamento

Até o momento foi utilizado o mapeamento de uma tabela isolada. Nesta aula o conceito de mapeamento será ampliado, considerando associações entre tabelas. Os tópicos consideraram exemplos variados.

9.1 Exemplo: as classes Person e Event

Veja os exemplos mostrados nos Quadros 33 e 34.

Quadro 33: A entidade Person

```
package org.hibernate.tutorial.domain;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // setters e getters
}
```

Quadro 34: A entidade Event

```
package org.hibernate.tutorial.domain;
import java.util.*;
public class Event {

    private Long id;
    private Date date;
    private String title;

    public Event() {}

    // setters e getters
}
```

O arquivo **Person.hbm.xml** possui o aspecto mostrado no Quadro 35.

Quadro 35: O arquivo de mapeamento Person.hbm.xml

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

  <class name="Person" table="PERSON">
    <id name="id" column="PERSON_ID">
      <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>
  </class>

</hibernate-mapping>
```

E o arquivo **Event.hbm.xml**, possui o aspecto mostrado no Quadro 36.

Quadro 36: O arquivo de mapeamento Event.hbm.xml

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

  <class name="Event" table="EVENT">
    <id name="id" column="EVENT_ID">
      <generator class="native"/>
    </id>
    <property name="date"/>
    <property name="title"/>
  </class>

</hibernate-mapping>
```

Finalmente, no arquivo de configurações hibernate.cfg.xml, a configuração é como mostrado no Quadro 37.

Quadro 37: Configuração dos mapeamentos no arquivo hibernate.cfg.xml

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

Será criada uma associação entre estas duas entidades. Um Person pode participar de vários eventos (Event), e um evento possui participantes. O questionamento aqui se aplica a direcionamento, multiplicidade e comportamento do conjunto.

9.2 Associação baseada em conjunto - Set

Ao adicionar uma coleção de eventos na classe `Person`, é fácil navegar pelos eventos de uma pessoa, sem a necessidade de uma *query* – chamando **`Person#getEvent`**. Associações múltiplas em Hibernate são representadas por coleções e aqui foi selecionado `java.util.Set`, porque existe a garantia de não duplicidade e a ordem não é relevante nos nossos exemplos.

Cada caso utiliza uma coleção em particular, dependendo da finalidade: `List`, `Set` ou `Map`. Veja o Quadro 38.

Quadro 38: Associação de Eventos para Pessoa

```
public class Person {  
  
    private Set events = new HashSet();  
  
    public Set getEvents() {  
        return events;  
    }  
  
    public void setEvents(Set events) {  
        this.events = events;  
    }  
}
```

É possível executar uma consulta para recuperar os participantes de um evento em particular, mas a discussão aqui é com relação à multiplicidade da associação: “muitos para muitos”, que é chamada de associação *many-to-many* do ponto de vista do mapeamento Hibernate, como mostra o Quadro 39.

Quadro 39: Configurando a associação

```

<class name="Person" table="PERSON">
  <id name="id" column="PERSON_ID">
    <generator class="native"/>
  </id>
  <property name="age"/>
  <property name="firstname"/>
  <property name="lastname"/>

  <set name="events" table="PERSON_EVENT">
    <key column="PERSON_ID"/>
    <many-to-many column="EVENT_ID" class="Event"/>
  </set>

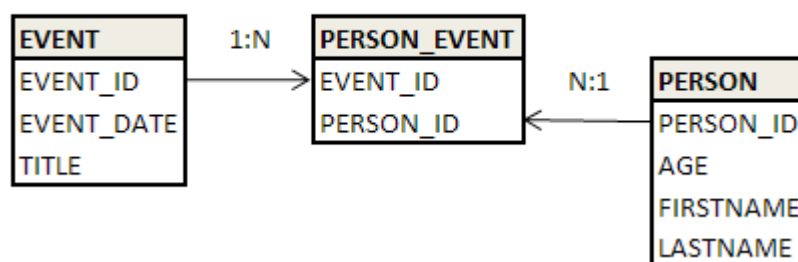
</class>

```

Para uma associação *many-to-many*, ou relacionamento *n:m*, uma tabela de associação se torna necessária, com cada linha representando um link entre uma Person e um Event. A tabela é declarada através do atributo **table** do **element set**. O campo identificador, do lado Person, é definido com o **element key**; do lado Event com o atributo **column** do relacionamento *many-to-many*.

O modelo para este mapeamento é mostrado na Figura 9.2.

Figura 9.2 – Modelo de dados



9.3 Manipulando a associação

Agora vamos mostrar um procedimento para adicionar um Event a um Person. Veja o Quadro 40.

Quadro 40: Manipulando a associação

```
private void addPersonToEvent(Long personId, Long eventId) {  
    Session session = HibernateUtil.  
        getSessionFactory().  
        getCurrentSession();  
    session.beginTransaction();  
  
    Person aPerson = (Person) session.  
        load(Person.class, personId);  
    Event anEvent = (Event) session.load(Event.class, eventId);  
    aPerson.getEvents().add(anEvent);  
  
    session.getTransaction().commit();  
}
```

Após carregar um Person e um Event, simplesmente a coleção é modificada usando os métodos tradicionais de Set. Não existe uma chamada explícita aos métodos update() ou save(); o Hibernate automaticamente detecta que a coleção foi modificada e necessita ser atualizada. Esse procedimento é conhecido como automatic dirty checking.

Como os objetos estão em um estado persistente, ou seja, fazendo parte de um Session, o Hibernate monitora quaisquer alterações e executa suas instruções SQL de forma transparente.

É possível carregar um Person e um Event em diferentes etapas, ou modificar um objeto fora de Session, quando não estiver no estado persistente (estado desacoplado, ou detached). Veja o Quadro 41.

Quadro 41: Adicionando elementos à associação

```
void addPersonToEvent(Long personId, Long eventId) {
    Session session = HibernateUtil
        .getSessionFactory()
        .getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session
        .createQuery("select p from Person p left join fetch p.events where p.id = :pid")
        .setParameter("pid", personId)
        .uniqueResult(); // estado desacoplado
    Event anEvent = (Event) session.load(Event.class, eventId);

    session.getTransaction().commit();

    // Fim da primeira etapa

    aPerson.getEvents().add(anEvent); // aPerson (e sua coleção)

    // Inicio da segunda etapa

    Session session2 = HibernateUtil
        .getSessionFactory()
        .getCurrentSession();
    session2.beginTransaction();
    session2.update(aPerson); // Novo vínculo de aPerson

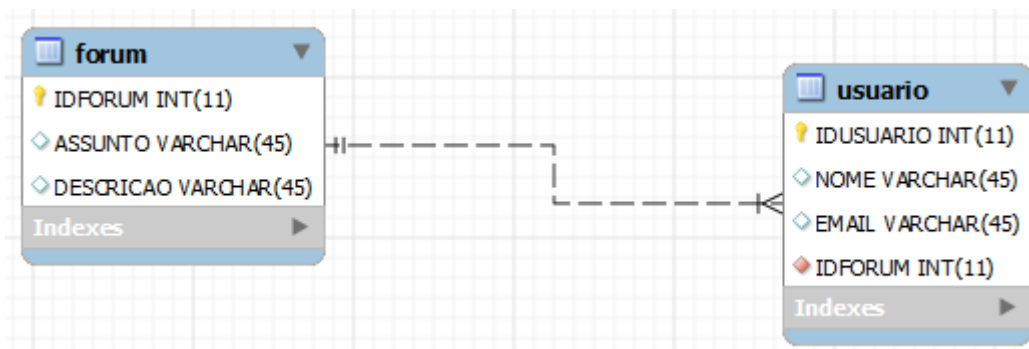
    session2.getTransaction().commit();
}
```


Exercícios do Módulo 1

Exercício 1

Neste exercício desenvolveremos uma aplicação baseada em Hibernate. Os passos são apresentados a seguir:

- Criar um projeto Java Project chamado ExemploHibernate.
- Importar a API do Hibernate.
- Criar o banco de dados chamado forum cujo modelo é dado a seguir:



- Com base neste modelo, criar as classes POJO **Forum** e **Usuario**:

```
package br.com.ead.entity;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

public class Forum implements Serializable{
    private static final long serialVersionUID = 1L;
    private int idforum;
    private String assunto;
    private String descricao;
    private Set<Usuario> usuarios = new HashSet<Usuario>(0);

    public int getIdforum() {
        return idforum;
    }

    public void setIdforum(int idforum) {
        this.idforum = idforum;
    }
}
```

```
public String getAssunto() {  
    return assunto;  
}  
public void setAssunto(String assunto) {  
    this.assunto = assunto;  
}  
public String getDescricao() {  
    return descricao;  
}  
public void setDescricao(String descricao) {  
    this.descricao = descricao;  
}  
public Set<Usuario> getUsuarios() {  
    return usuarios;  
}  
public void setUsuarios(Set<Usuario> usuarios) {  
    this.usuarios = usuarios;  
}  
}
```

```
package br.com.ead.entity;
import java.io.Serializable;

public class Usuario implements Serializable{

    private static final long serialVersionUID = 1L;
    private int idusuario;
    private Forum forum;
    private String nome;
    private String email;

    public int getIdusuario() {
        return idusuario;
    }
    public void setIdusuario(int idusuario) {
        this.idusuario = idusuario;
    }
    public Forum getForum() {
        return forum;
    }
    public void setForum(Forum forum) {
        this.forum = forum;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

- Definir os arquivos de mapeamento, no mesmo pacote que as classes POJO:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="br.com.ead.entity.Forum" table="forum" catalog="forum">
    <id name="idforum" type="int">
      <column name="IDFORUM" />
      <generator class="assigned" />
    </id>
    <property name="assunto" type="string">
      <column name="ASSUNTO" length="45" />
    </property>
    <property name="descricao" type="string">
      <column name="DESCRICAO" length="45" />
    </property>
    <set name="usuarios" inverse="false" cascade="save-update">
      <key>
        <column name="IDFORUM" not-null="true" />
      </key>
      <one-to-many class="br.com.ead.entity.Usuario" />
    </set>
  </class>
</hibernate-mapping>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
<class name="br.com.ead.entity.Usuario" table="usuario" catalog="forum">
  <id name="idusuario" type="int">
    <column name="IDUSUARIO" />
    <generator class="assigned" />
  </id>
  <many-to-one name="forum" class="br.com.ead.entity.Forum" fetch="select">
    <column name="IDFORUM" not-null="true" />
  </many-to-one>
  <property name="nome" type="string">
    <column name="NOME" length="45" />
  </property>
  <property name="email" type="string">
    <column name="EMAIL" length="45" />
  </property>
</class>
</hibernate-mapping>
```

- Definir o arquivo de configurações **hibernate.cfg.xml**, na raiz dos pacotes das classes:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/forum</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">password</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <mapping resource="br/com/ead/entity/Forum.hbm.xml"/>
    <mapping resource="br/com/ead/entity/Usuario.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

- Definir a classe **HibernateUtil**, responsável por realizar a leitura dos arquivos de configurações:

```
package br.com.ead.config;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Exception e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

- Escrever a classe **ForumHelper**, contendo métodos auxiliares para realizar a persistência:

```
package br.com.ead.helper;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.Session;
import org.hibernate.Transaction;
import br.com.ead.config.HibernateUtil;
import br.com.ead.entity.Forum;
import br.com.ead.entity.Usuario;

public class ForumHelper {
    Session session = null;
    Transaction transaction = null;
```

```
public String salvar(Forum forum){
    try{
        session = HibernateUtil.getSessionFactory().getCurrentSession();
        transaction = session.beginTransaction();
        session.save(forum);
        transaction.commit();
        return "Forum salvo";
    }catch(Exception e){
        return e.getMessage();
    }
}
```

```
public String salvar(Usuario forum){
    try{
        session = HibernateUtil.getSessionFactory().getCurrentSession();
        transaction = session.beginTransaction();
        session.save(forum);
        transaction.commit();
        return "Usuario salvo";
    }catch(Exception e){
        return e.getMessage();
    }
}
```

```
public String adicionarUsuario(int idForum, int idUsuario){
    try {
        session = HibernateUtil.getSessionFactory().getCurrentSession();
        transaction = session.beginTransaction();
        Forum f = (Forum)session.load(Forum.class, idForum);
        Usuario u = (Usuario)session.load(Usuario.class, idForum);

        f.getUsuarios().add(u);
        session.save(f);
        transaction.commit();
        return "Associação realizada";
    } catch(Exception e){
```

```

return e.getMessage();
}
}

public String adicionarUsuario(int idForum, Usuario usuario){
try {
session = HibernateUtil.getSessionFactory().getCurrentSession();
transaction = session.beginTransaction();
Forum f = (Forum)session.load(Forum.class, idForum);

usuario.setForum(f);

f.getUsuarios().add(usuario);
session.update(f);
transaction.commit();
return "Inclusao realizada";
} catch (Exception e){
return e.getMessage();
}
}

public Set<Usuario> listarUsuarios(int idForum){
Set<Usuario> usuarios = new HashSet<Usuario>();
try {
session = HibernateUtil.getSessionFactory().getCurrentSession();
transaction = session.beginTransaction();
Forum f = (Forum)session.load(Forum.class, idForum);
usuarios = f.getUsuarios();

} catch (Exception e) {

}

return usuarios;
}
}

```

- Para testar a aplicação, criar um fórum e três usuários, associando cada usuário ao fórum criado:


```
package br.com.ead.programa;

import java.util.Set;
import br.com.ead.entity.Forum;
import br.com.ead.entity.Usuario;
import br.com.ead.helper.ForumHelper;

public class TesteForum {
    public static void main(String[] args) {
        incluirForum();
        incluirUsuarionoForum();
        listarUsuariosPorForum();
    }

    private static void incluirForum(){
        Forum forum = new Forum();
        forum.setIdforum(10);
        forum.setAssunto("Avaliação");
        forum.setDescricao("Avaliação da disciplina Persistência");

        ForumHelper helper = new ForumHelper();
        System.out.println(helper.salvar(forum));
    }

    private static void incluirUsuarionoForum(){
        ForumHelper helper = new ForumHelper();
        Usuario u1 = new Usuario();
        u1.setNome("teresa");
        u1.setEmail("teresa@mail.com");
        u1.setIdusuario(1);

        Usuario u2 = new Usuario();
        u2.setNome("jonas");
        u2.setEmail("joas@mail.com");
        u2.setIdusuario(2);
    }
}
```

```

        Usuario u3 = new Usuario();
u3.setNome("abilio");
u3.setEmail("abilio@mail.com");
u3.setIdusuario(3);

System.out.println(helper.adicionarUsuario(10, u1));
System.out.println(helper.adicionarUsuario(10, u2));
System.out.println(helper.adicionarUsuario(10, u3));
    }

    private static void listarUsuariosPorForum(){
        ForumHelper helper = new ForumHelper();
        Set<Usuario> usuarios = helper.listarUsuarios(10);
for(Usuario usuario: usuarios){
            System.out.println("ID Usuario: " + usuario.getIdusuario());
            System.out.println("Nome Usuario: " + usuario.getNome());
            System.out.println("Email Usuario: " + usuario.getEmail());
            System.out.println("-----");
        }
    }
}

```

Exercício 2

Orientações:

1. Leia atentamente e siga os passos descritos neste documento para a elaboração desta atividade.
2. Criar os arquivos que forem necessários. Tomar como base o Exercício 1 deste módulo.

Cenário para elaboração do banco de dados:

Em uma aplicação bancária, um cliente correntista pode ter uma ou mais contas. Cada conta pertence a apenas um cliente. A sugestão para elaboração das tabelas é a seguinte:

CLIENTE
CPF (texto) - PK
NOME (texto)
DATANASC (data)

CONTA
AGENCIA (int)
CONTACORRENTE (texto)
CPF (texto) - FK
SALDO (double)

Passos para o desenvolvimento a atividade.

1. Criar esse banco de dados, considerando como chave de relacionamento o CPF do cliente.
2. O banco de dados deverá se chamar **banco**.
3. Criar um projeto "**Java Project**" chamado **Modulo1_Exercicio2**.
4. Definir as entidades para **Cliente** e **Conta**.
5. Escrever todas as classes necessárias para elaborar a persistência. Considerar as operações: **INCLUIR, ALTERAR, LISTAR e REMOVER** (CRUD)
6. Escrever um programa consistente para testar todos os itens que você definiu.

Módulo 2 - Java Persistence API (JPA)

Caro aluno, JPA (Java Persistence API) é definida no padrão JSR-220 (*Enterprise JavaBeans*, versão 3.0), que padroniza o mapeamento objeto/relacional (ORM) na plataforma Java. A JPA é baseada no modelo POJO e os objetos persistentes são denominados entidades (Entity). Uma entidade é um POJO, que representa um conjunto de dados persistidos no banco.

Aula 10 - Annotations

10.1 A API de persistência



Fonte: <<http://2.bp.blogspot.com/-2iVw6sLVsdY/T3BICGgvxQI/AAAAAAAAAnU/tPs-TxPD-qw/s320/JabatanPerkhidmatanAwamBiasiswaJPAPSDScholarships.png>>. Acesso em: 07 nov. 2014.

Java Annotations são tipos especialmente definidos com o intuito de simplificar tarefas em Java com uma simples anotação, colocada em frente ou acima dos elementos de programa como classes, métodos, campos ou variáveis.

Quando um elemento de um programa é anotado, o compilador lê a informação contida nessa anotação e pode reter esta informação nos arquivos de classe ou dispor disso de acordo com o que foi especificado na definição de tipo de anotação.

Quando estiverem nos arquivos de classe, os elementos contidos na anotação podem ser examinados em tempo de execução por uma API baseada em reflexão. Dessa forma, a JVM pode olhar esses metadados para determinar como interagir com os elementos do programa ou alterar seus comportamentos.

Uma anotação é precedida por um símbolo '@' seguida de uma meta-anotação. As anotações foram introduzidas a partir do Java 5 e fazem parte do Java EE 5.

Para que uma entidade se torne persistente, é necessário associá-la a um contexto de persistência. Esse contexto fornece a conexão entre as instâncias e o banco de dados.

As classes e interfaces da JPA estão localizadas no pacote `javax.persistence`. Com isso, pode-se fazer o mapeamento da aplicação utilizando anotações. Esse procedimento dispensa o uso de XML para cada uma das entidades da aplicação.

Por isso, uma entidade é rotulada com a anotação **@Entity** sendo ela uma classe Java comum. Uma tabela é representada pela anotação **@Table** e a chave primária pela anotação **@Id**. Cada coluna é especificada pela anotação **@Column** e assim por diante.

Caro aluno, você pode ver a documentação disponível em: <http://hibernate.org/> para conhecer todas as anotações.

10.2 Exemplo de Aplicação

Para criarmos uma aplicação, vamos considerar o banco de dados com as tabelas:

1. Schema: forum
2. Tabelas: FÓRUM (IDFORUM, ASSUNTO, DESCRICAO) e USUARIO (IDUSUARIO, NOME, EMAIL).
Primeiro vamos inserir o arquivo de persistência (persistence.xml) mostrado no Quadro 42.

Quadro 42: O arquivo persistence.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="ExemploJPA">
        <class>br.com.ead.Forum</class>
        <class>br.com.ead.Usuario</class>
        <exclude-unlisted-classes>>false</exclude-unlisted-classes>
        <properties>
            <property name="hibernate.connection.username" value="root"/>
            <property name="hibernate.connection.password" value="password"/>
            <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Dri-
ver"/>
            <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/
forum"/>
            <property name="hibernate.cache.provider_class" value="org.hibernate.cache.
NoCacheProvider"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>

```

Agora devemos mapear duas entidades **Forum** e **Usuario** de acordo com os Quadros 43 e 44.

Quadro 43: A entidade Fórum com annotations

```

package br.com.ead;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.*;

@Entity
@Table (name="FORUM",schema="forum")
public class Forum implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @SequenceGenerator(name="FORUM_IDFORUM_GENERATOR" )
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="FORUM_IDFORUM_GENERATOR")
    private int id;
    private String assunto, descricao;

    @OneToMany(cascade=CascadeType.ALL,fetch=FetchType.LAZY)
    private List<Usuario> usuarios;

    public Forum() {
        usuarios = new ArrayList<Usuario>();
    }

    public List<Usuario> getUsuarios() {
        return usuarios;
    }

    public void setUsuarios(List<Usuario> usuarios) {
        this.usuarios = usuarios;
    }
}

```

```
public int getId() {  
    return this.id;  
}  
  
public void setId(int id) {  
    this.id = id;  
}  
  
public String getAssunto() {  
    return this.assunto;  
}  
  
public void setAssunto(String assunto) {  
    this.assunto = assunto;  
}  
  
public String getDescricao() {  
    return this.descricao;  
}  
  
public void setDescricao(String descricao) {  
    this.descricao = descricao;  
}  
}
```

Quadro 44: A entidade Usuario com *annotations*

```
package br.com.ead;  
  
import java.io.Serializable;  
import javax.persistence.*;  
  
import java.util.List;  
  
@Entity  
@Table(name="USUARIO",schema="forum")  
public class Usuario implements Serializable {  
    private static final long serialVersionUID = 1L;
```



```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int id;

private String nome, email;

public Usuario() {
}

public int getId() {
    return this.id;
}

public void setId(int id) {
    this.id = id;
}

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

public void setNome(String nome) {
    this.nome = nome;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}
}
```

Agora será criada a classe Helper, responsável pela persistência e pela recuperação dos dados, como no Quadro 45.

Quadro 45: Utilização do objeto EntityManager

```
package br.com.ead;

import java.util.List;
import javax.persistence.EntityManager;

public class ForumJavaPersistenceDAO {

    private EntityManager em;

    public ForumJavaPersistenceDAO(EntityManager em) {
        this.em = em;
    }

    public Forum createForum(String assunto, String descricao){
        Forum forum = new Forum();
        forum.setDescricao(descricao);
        forum.setAssunto(assunto);
        em.getTransaction().begin();
        em.persist(forum);
        em.getTransaction().commit();
        return forum;
    }

    public Usuario createUsuario(String nome,String email){
        Usuario usuario = new Usuario();
        usuario.setNome(nome);
        usuario.setEmail(email);
        em.getTransaction().begin();
        em.persist(usuario);
        em.getTransaction().commit();
        return usuario;
    }

    public Forum findForum(int id){
        return em.find(Forum.class, id);
    }
}
```

```

public Forum changeDescricaoForum(int id, String descricao){
    Forum forum = this.findForum(id);
    forum.setDescricao(descricao);
    return forum;
}

public void deleteForum(int id){
    Forum forum = this.findForum(id);
    em.remove(forum);
}

public Forum addUsuarioToForum(int id,Usuario usuario){
    Forum forum = this.findForum(id);
    forum.getUsuarios().add(usuario);
    return forum;
}

public List<Usuario> listUsuariosFromForum(int id){
    List<Usuario> usuarios = this.findForum(id).getUsuarios();
    return usuarios;
}

public void closeEntityManager() {
    this.em.close();
}

}

```

Para testar a aplicação, usamos o programa do Quadro 46.

Quadro 46: Testando a aplicação

```

package br.com.ead;

import javax.persistence.*;
import java.util.*;

public class TesteJPA {

```

```

public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("ExemploJPA");
    EntityManager em = emf.createEntityManager();
    ForumJavaPersistenceDAO forumDAO = new ForumJavaPersistenceDAO(em);
    Forum forum = forumDAO.createForum("Java Persistence API",
        "JPA é utilizado para tratamento ORM");

    for (int i = 0; i < 10; i++) {
        Usuario usuario = forumDAO.createUsuario("Usuario"+i, "usuario"+i+"@ead.com.br");

        forumDAO.addUsuarioToForum(forum.getId(), usuario);
    }

    List<Usuario> usuarios = forumDAO.listUsuariosFromForum(forum.getId());
    System.out.println("Forum: " + forumDAO.findForum(forum.getId()).getAssunto());
    for (Usuario usuario: usuarios) {
        System.out.println(usuario.getNome() + " - " + usuario.getEmail());
    }

    forumDAO.closeEntityManager();
}

```

Aula 11 - Anotações JPA

As anotações JPA estão localizadas no pacote **javax.persistence** e são apresentadas nas seções que se seguem.

11.1 @Entity

Especifica que uma classe é uma entidade.

Quadro 47: A anotação @Entity

```

@Entity
public class ClienteEntity {
    private int id;
    private String nome;
    // métodos get e set
}

```

11.2 @Table

Especifica a tabela associada à entidade no banco de dados. Caso não definido, assume-se que a tabela terá o mesmo nome da classe da entidade.

Quadro 48: A anotação @Table

```
@Entity
@Table (name="TAB_CLIENTE")
public class ClienteEntity {
    private int id;
    private String nome;
    // métodos get e set
}
```

11.3 @Column

Especifica o campo associada ao atributo da entidade. Caso não definido assume-se que o campo terá o mesmo nome do atributo.

Parâmetros:

- Name → nome do campo;
- Unique (default false) → não permite duplicidade;
- Nullable (default false) → não permite valores nulos;
- Insertable (default true) → atributo utilizado em operações de INSERT;
- Updatable (default false) → atributo utilizado em operações de UPDATE.

Quadro 49: A anotação @Column

```
@Entity
@Table(name="TAB_CLIENTE")
public class ClienteEntity {
    @Column(name="COD_CLIENTE")
    private int id;
    @Column(name="NOM_CLIENTE", nullable=false)
    private String nome;
    // métodos get e set
}
```

11.4 @Id

Atributo que identificará unicamente as instâncias da entidade. Deve-se sempre definir o atributo que representará a chave primária.

Quadro 50: A anotação @Id

```
@Entity
@Table(name="TAB_CLIENTE")
public class ClienteEntity {
    @Id
    @Column(name="COD_CLIENTE")
    private int id;
    @Column(name="NOM_CLIENTE", nullable=false)
    private String nome;
    // métodos get e set
}
```

11.5 @GeneratedValue

Especifica a estratégia de geração de valores para atributos que são chave primária.

Parâmetros:

- Strategy → indica o tipo de estratégia utilizada;
- Generator → nome do gerador de chaves;

Tipos mais comuns:

- GenerationType.SEQUENCE → baseado em sequence;
- GenerationType.IDENTITY → campos identidade;

11.6 @SequenceGenerator

Define um gerador de chave primária baseado em sequence de banco de dados. Possui uma associação com o **@GeneratedValue**.

Parâmetros:

- Name → nome a ser referenciado pelo **@GeneratedValue**;
- sequenceName → nome da sequence de banco de dados;
- allocationSize (default 50) → increment.

Quadro 51: A anotação @GeneratedValue e @SequenceGenerator

```
@Entity
@SequeceGenerator(name="cliente", sequenceName="SEQ_CLIENTE",
    allocationSize=1)
@Table(name="TAB_CLIENTE")
public class ClienteEntity {
    @Id
    @GeneratedValue(strategy=GeneratorType.SEQUENCE,
        generator="cliente")
    @Column(name="COD_CLIENTE")
    private int id;
    @Column(name="NOM_CLIENTE", nullable=false)
    private String nome;
    // métodos get e set
}
```

11.7 @Transient

Indica que determinado atributo não deve ser persistido.

Quadro 52: A anotação @Transient

```
@Entity
@Table(name="TAB_CLIENTE")
public class ClienteEntity {
    @Id
    @Column(name="COD_CLIENTE")
    private int id;
    @Column(name="NOM_CLIENTE", nullable=false)
    private String nome;
    @Transient
    private int chaveAcesso;
}
```

11.8 @Temporal

Especifica o tipo de dado a ser armazenado em atributos do tipo *Date* e *Calendar*.

Parâmetros:

- TemporalType.TIMESTAMP → data e hora;
- TemporalType.DATE → somente data;

- `TemporalType.TIME` → somente hora.

Quadro 53: A anotação `@Temporal`

```
@Entity
@Table(name="TAB_CLIENTE")
public class Cliente {
    @Id
    @Column(name="COD_CLIENTE")
    private int id;
    @Column(name="DAT_NASCIMENTO")
    @Temporal(value=TemporalType.DATE)
    private Date dataNascimento;
    // ... Métodos get / set
}
```

11.9 @PersistenceContext

Utilizar a anotação **@PersistenceContext** para obter, via injeção de dependência, uma instância que implemente a interface **EntityManager**.

Veja um exemplo no Quadro 52.

Quadro 54: A anotação `@PersistenceContext`

```
public class ClienteBean ... {
    @PersistenceContext
    EntityManager manager = null;
}
```

11.10 @OneToMany

Utilizar a anotação **@OneToMany** no atributo que representa a associação.

Utilizar o atributo **cascade** para indicar:

- **CascadeType.ALL** → todas as operações na entidade pai serão refletidas na(s) filho(s);
- **CascadeType.MERGE** → somente operação de merge será refletida;
- **CascadeType.PERSIST** → somente operação de persist será refletida;
- **CascadeType.REFRESH** → somente operação refresh será refletida;
- **CascadeType.REMOVE** → somente operação remove será refletida;

Podem-se combinar vários tipos: **@OneToMany(cascade={CascadeType.MERGE, CascadeType.REMOVE})**

Utilizar a anotação **@JoinColumn** em conjunto com **@OneToMany** para indicar o nome da coluna que representa a chave estrangeira na tabela filho. Pode ser utilizada em uma associação unidirecional.

Quadro 55: A anotação @OneToMany

```
public class NFEntity {
    ...
    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="COD_NF")
    private Collection<NFItemEntity> itens;
    ...
}
public class NFItemEntity {
    ...
}
```

11.11 @ManyToOne

A anotação **@ManyToOne** pode ser utilizada para indicar que uma associação é bidirecional. Nas associações bidirecionais existe uma referência tanto do pai para o filho quanto do filho para o pai. Na entidade pai, utilizar o atributo **mappedBy** para indicar o nome do atributo no filho que representa a ligação com o pai.

Dispensa o uso de @JoinColumn na entidade pai, uma vez que a mesma é definida na entidade filho.

Quadro 56: A anotação @ManyToOne

```
public class NFEntity {
    ...
    @OneToMany(cascade=CascadeType.ALL, mappedBy="nf")
    private Collection<NFItemEntity> itens;
    ...
}
public class NFItemEntity {
    ...
    @ManyToOne
    @JoinColumn(name="COD_NF")
    private NFEntity nf;
    ...
}
```

11.12 @ManyToMany

Utilizada para representar associações M:N. Possui o mesmo atributo **cascade** da anotação **@OneToMany**.

Utilizar a anotação **@JoinTable** associada para referenciar a tabela associativa e os campos de chave estrangeira. Parâmetros:

- **name** → nome da tabela associativa;
- **joinColumns** → colunas de chave estrangeira que referenciam a entidade diretamente;
- **inverseJoinColumns** → colunas de chave estrangeira que referenciam a entidade no outro lado da relação.

Utilizar a anotação **@JoinColumn** para definir as referências diretas e inversas. Parâmetro:

- **name** → nome da coluna de chave estrangeira.

Quadro 57: A anotação **@ManyToMany**

```
public class CursoEntity {  
    ...  
    @ManyToMany  
    @JoinTable(name="TAB_ALUNO_CURSO",  
        joinColumns=@JoinColumn(name="COD_CURSO"),  
        inverseJoinColumns=@JoinColumn(name="COD_ALUNO"))  
    private Collection<AlunoEntity> alunos;  
    ...  
}
```

11.13 @Embeddable (Chave Composta)

Chaves compostas podem ser representadas através de uma classe com a anotação **@Embeddable**. A classe de chave primária deve ser utilizada como se fosse um **@Id**, só que com a anotação **@EmbeddedId**;

Essa classe deve ser *serializable*. Deve implementar os métodos **equals(Object)** e **hashCode()**.

Quadro 58: A anotação @Embeddable

```

@Embeddable
public class MatriculaID implements Serializable {
    @ManyToOne
    @JoinColumn(name="COD_ALUNO")
    private Aluno aluno;
    @ManyToOne
    @JoinColumn(name="COD_CURSO")
    private Curso curso;
    public MatriculaID(){ }
    public MatriculaID(Aluno aluno, Curso curso){
        this.aluno = aluno;
        this.curso = curso;
    }
    // get e set
    @Override
    public boolean equals(Object arg0) {
        return super.equals(arg0);
    }
    @Override
    public int hashCode() {
        return super.hashCode();
    }
}

```

```

@Entity
@Table(name="TAB_ALUNO_CURSO")
public class Matricula {
    @EmbeddedId
    private MatriculaID id;
    @column(name="DAT_MATRICULA")
    private Date data;
    // get e set
}

```

Aula 12 - API Entity Manager

Caro aluno, no JPA é possível manipular objetos persistentes usando-se apenas o Hibernate, ou generalizando-a através de um objeto da interface EntityManager. A vantagem dessa interface é sua integração com outros mecanismos de acesso como o EJB, por exemplo. Os métodos são diferentes daqueles usados pelo

Hibernate. Podemos até dizer que o Entity Manager permite trabalharmos como que podemos chamar de JPA Puro.

A seguir são apresentados os métodos de Entity Manager: *persist*, *merge*, *refresh* e *find*.

- **persist (Object entity)**: enfileira entidade para ser inserida uma entidade no banco de dados e a torna gerenciada. Veja um exemplo no Quadro 59.

Quadro 59: Incluindo um registro

```
VeiculoEntity veiculo = new VeiculoEntity();
veiculo.setPlaca("DHZ-5678");
veiculo.setModelo("Fusca");
manager.persist(veiculo);
```

- **merge (Object entidade)**: atualiza uma entidade não gerenciada no contexto de persistência. Caso já exista uma entidade gerenciada com o mesmo id à realiza uma operação de UPDATE; Caso contrário à realiza uma operação de INSERT. A entidade torna-se gerenciada. Veja um exemplo no Quadro 60.

Quadro 60: Alterando um registro

```
VeiculoEntity veiculo = new VeiculoEntity();
veiculo.setPlaca("DHZ-5678");
veiculo.setModelo("Fusca");
manager.merge(veiculo);
```

- **refresh (Object entidade)**: Atualiza dados da entidade com base no banco de dados. Valores não persistidos são descartados. A entidade torna-se gerenciada. Veja um exemplo no Quadro 61.

Quadro 61: Alterando um registro com base no banco de dados

```
VeiculoEntity veiculo = new VeiculoEntity();
veiculo = manager.find(VeiculoEntity.class, 1);
veiculo.setPlaca("DHZ-5678");
veiculo.setModelo("Fusca");
manager.refresh(veiculo);
```

- **find (Class classeEntidade, Object PK)**: localiza uma entidade através de sua chave primária (PK). Retorna *null* caso a entidade não seja localizada. Uma entidade localizada torna-se automaticamente gerenciada. Veja um exemplo no Quadro 62.

Quadro 62: Buscando e alterando um registro

```
// Busca veiculo com id igual a 10
VeiculoEntity veiculo = manager.find(VeiculoEntity.class, 10);
veiculo.setPlaca("HHH-7777");
```

O remove (Object entidade): Remove uma entidade acoplada. Para excluir entidades desacopladas, primeiro deve-se localizá-la através do método **find** ou **getReference**. Veja um exemplo no Quadro 63.

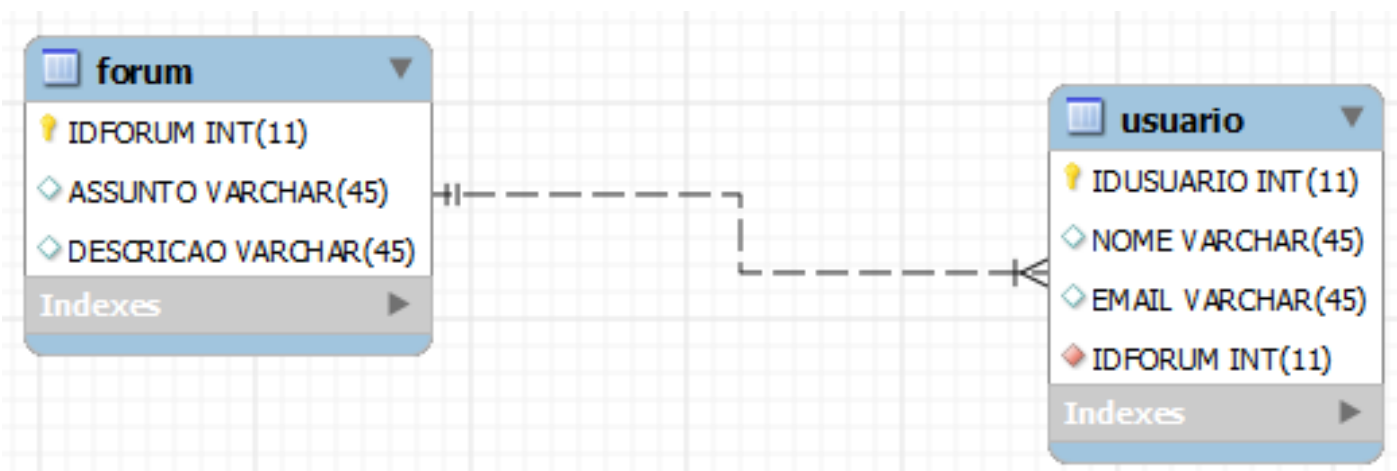
Quadro 63: Buscando e removendo um registro

```
VeiculoEntity veiculo = manager.find(VeiculoEntity.class, 10);
manager.remove(veiculo);
```

Exercícios do Módulo 2**Exercício 1**

Neste roteiro desenvolveremos uma aplicação baseada em anotações JPA. Os passos são apresentados a seguir:

- Criar um projeto Java Project chamado **ExemploJPA**.
- Importar a API do **Hibernate**.
- Importar o banco de dados **forum** do Exercício 1 do Módulo 1:



- Copiar este banco de dados para outro chamado **forum01**, e tornar as chaves **IDFORUM** e **IDUSUARIO** como autoincremento.
- Com base neste modelo, criar as entidades **Forum** e **Usuario** (as mesmas do módulo anterior):

```
package br.com.ead.entity;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name="FORUM", schema = "forum01")
public class Forum implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "IDFORUM")
    private int id;

    @Column(name = "ASSUNTO", length = 45)
    private String assunto;

    @Column(name = "DESCRICAO", length = 45)
    private String descricao;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "forum")
    private Set<Usuario> usuarios = new HashSet<Usuario>();

    public int getId() {
        return id;
    }
}
```

```
public void setId(int id) {  
    this.id = id;  
}  
public String getAssunto() {  
    return assunto;  
}  
public void setAssunto(String assunto) {  
    this.assunto = assunto;  
}  
public String getDescricao() {  
    return descricao;  
}  
public void setDescricao(String descricao) {  
    this.descricao = descricao;  
}  
public Set<Usuario> getUsuarios() {  
    return usuarios;  
}  
public void setUsuarios(Set<Usuario> usuarios) {  
    this.usuarios = usuarios;  
}  
}
```

```
package br.com.ead.entity;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table (name = "USUARIO", schema = "forum01")
public class Usuario implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "IDUSUARIO")
    private int id;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "IDFORUM")
    private Forum forum;

    @Column(name = "NOME")
    private String nome;

    @Column(name = "EMAIL")
    private String email;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```



```

public Forum getForum() {
return forum;
}
public void setForum(Forum forum) {
this.forum = forum;
}
public String getNome() {
return nome;
}
public void setNome(String nome) {
this.nome = nome;
}
public String getEmail() {
return email;
}
public void setEmail(String email) {
this.email = email;
}
}

```

- Definir, na pasta src/META-INF, o arquivo **persistence.xml**:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

  <persistence-unit name="Forum">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>br.com.ead.entity.Forum</class>
    <class>br.com.ead.entity.Usuario</class>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.format_sql" value="true" />
      <property
        name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLDialect" />
      <property
        name="hibernate.connection.driver_class"
        value="com.mysql.jdbc.Driver" />
    </properties>
  </persistence-unit>
</persistence>

```

```
<property
    name="hibernate.connection.url"
    value="jdbc:mysql://localhost/forum01" />
<property name="hibernate.connection.username" value="root" />
<property name="hibernate.connection.password" value="password" />
</properties>
</persistence-unit>
</persistence>
```

- Escrever a classe **ForumHelper**, contendo métodos auxiliares para realizar a persistência:

```
package br.com.ead.helper;

import javax.persistence.EntityManager;
import br.com.ead.entity.Forum;
import br.com.ead.entity.Usuario;

public class ForumHelper {
    private EntityManager em;

    public ForumHelper(EntityManager em){
        this.em = em;
    }

    public String salvar(Forum forum){
        try{
            em.getTransaction().begin();
            em.persist(forum);
            em.getTransaction().commit();
        } catch (Exception e){
            return "Forum salvo";
        }
        return e.getMessage();
    }
}
```

```

public String adicionarUsuario(int idForum, Usuario usuario){
try {
    Forum f = em.find(Forum.class, idForum);
    usuario.setForum(f);
    f.getUsuarios().add(usuario);

    em.getTransaction().begin();
    em.persist(f);
    em.getTransaction().commit();
return "Inclusao realizada";
} catch(Exception e){
return e.getMessage();
}
}
}

```

- Para testar a aplicação, criar um fórum e três usuários, associando cada usuário ao fórum criado:

```

package br.com.ead.programa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import br.com.ead.entity.Forum;
import br.com.ead.entity.Usuario;
import br.com.ead.helper.ForumHelper;

public class TesteForum {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("Forum");
        EntityManager em = emf.createEntityManager();
        ForumHelper dao = new ForumHelper(em);

        Forum forum = new Forum();
        forum.setAssunto("JPA");
        forum.setDescricao("Java Persistence API");
    }
}

```

```
System.out.println(dao.salvar(forum));

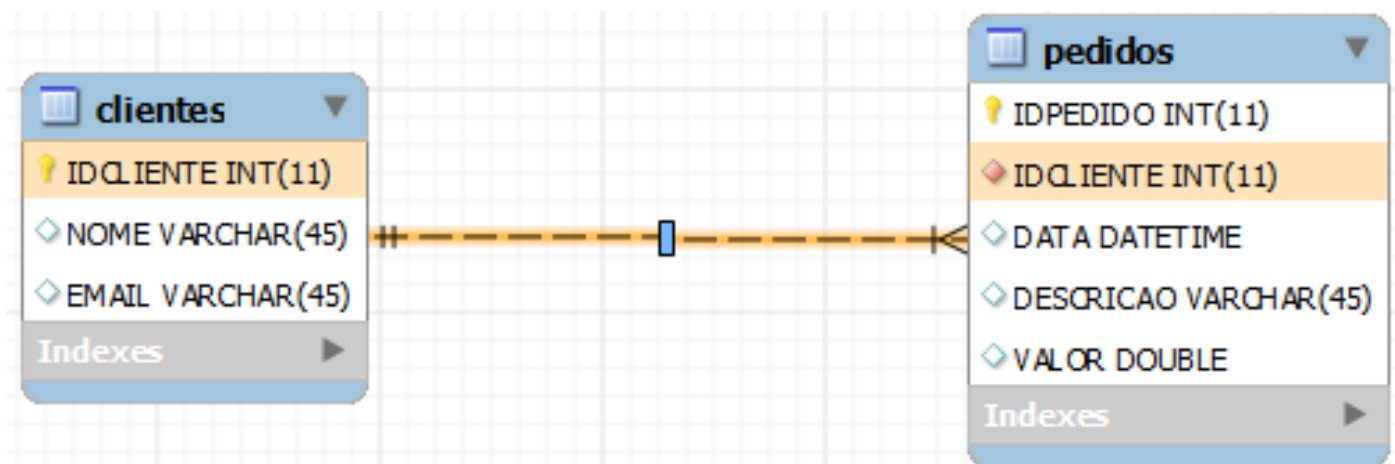
    Usuario usuario = new Usuario();
    usuario.setNome("Joaquim");
    usuario.setEmail("joaquim@ead.com.br");

    System.out.println(dao.adicionarUsuario(forum.getId(), usuario));

}
```

Exercício 2

Com base no exercício 1, criar uma aplicação para a persistência de clientes e pedidos, usando anotações JPA. O modelo é dado a seguir. É necessário criar a aplicação completa!



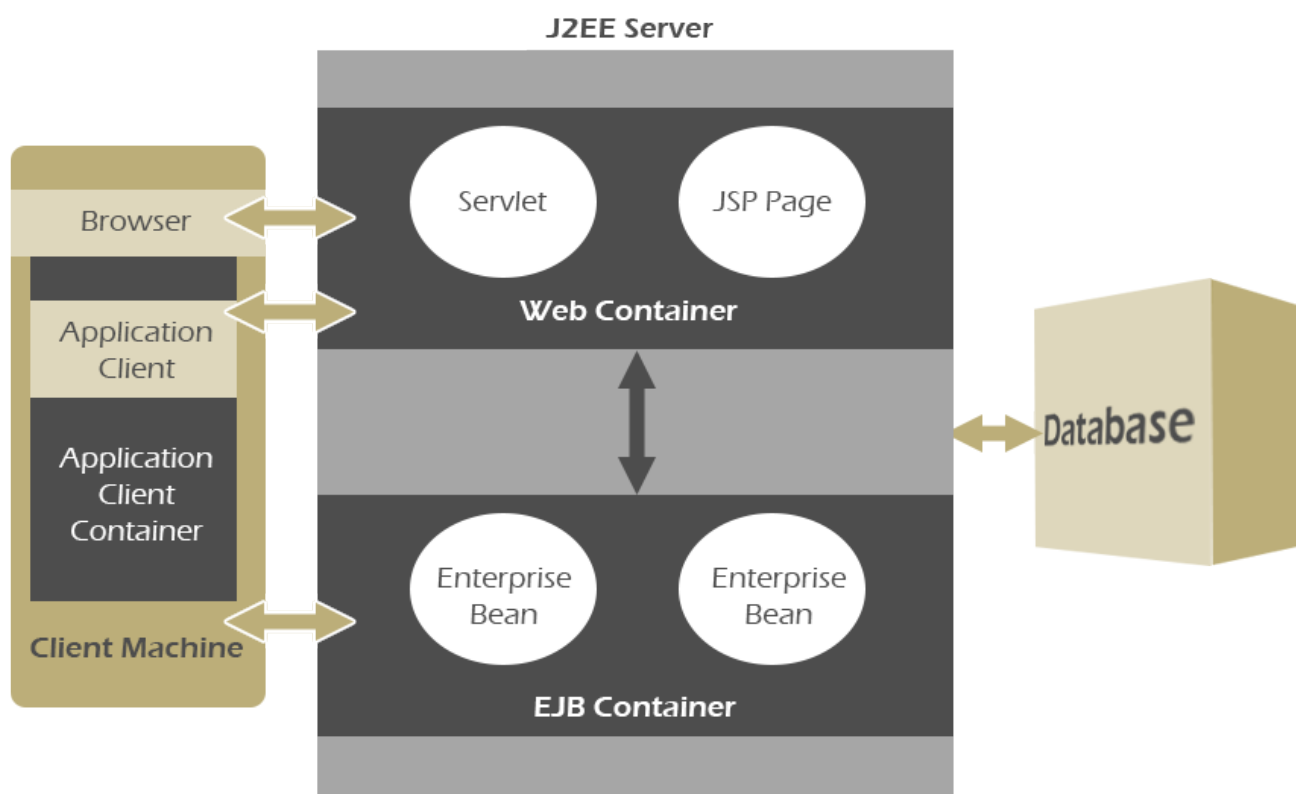
Módulo 3 – Conceitos do *Enterprise JavaBeans* (EJB)

Aula 13 - Introdução ao EJB

13.1 Definição e Arquitetura do EJB

O *Enterprise JavaBeans* (EJB) é uma arquitetura de desenvolvimento para a elaboração de aplicações altamente robustas e escaláveis, implantadas em servidores de aplicações compatíveis com JEE (Java Enterprise Edition), como JBoss, Web Logic, GlassFish, etc. A figura 13.1 apresenta a arquitetura JEE contemplando os componentes EJB.

Figura 13.1 – Mecanismo de requisição e resposta no EJB



Fonte: <http://www.linhadecodigo.com.br/artigos/img_artigos/eric_oliveira/containers.gif>. Acesso em: 05 nov. 2014.

Muitos sistemas corporativos são desenvolvidos seguindo a arquitetura definida pelo padrão *Enterprise JavaBeans*. Ao utilizar essa arquitetura, diversos recursos são disponibilizados a esses sistemas, como transações, segurança, remotabilidade, concorrência, persistência, gerenciamento de objetos e integração. Vejamos a que se referem:

Transações: A arquitetura EJB define um suporte sofisticado para utilização de transações. Esse suporte é integrado com a Java Transaction API (JTA) e oferece, inclusive, a possibilidade de realizar transações distribuídas.

Segurança: Oferece suporte para realizar autenticação e autorização de forma transparente. Os desenvolvedores das aplicações não precisam implementar a lógica de segurança, pois ela faz parte da arquitetura EJB.

Remotabilidade: Aplicações EJB podem ser acessadas remotamente através de diversos protocolos de comunicação. Consequentemente, é possível desenvolver aplicações de clientes de diversos tipos. Por exemplo, aplicações EJB podem funcionar como *Web Services*.

Multithreading e Concorrência: A arquitetura EJB permite que as aplicações sejam acessadas por múltiplos usuários simultaneamente, de maneira controlada, para evitar problemas de concorrência.

Persistência: Oferece facilidades para utilizar os serviços dos provedores de persistência que seguem a especificação JPA.

Gerenciamento de Objetos: Mecanismos de injeção de dependências e controle de ciclo de vida são oferecidos aos objetos de uma aplicação EJB. O mecanismo de controle de ciclo de vida pode garantir a escalabilidade de uma aplicação.

Integração: A arquitetura EJB é fortemente integrada com os componentes da plataforma Java EE. Podemos facilmente integrar os recursos do JSF em uma aplicação EJB.

Fonte: <<http://pt.slideshare.net/RodolfoSilva9/desenvolvimento-web-avancado-com-jsf-2-ejb-3-1-ecdi>>. Acesso em: 07 nov. 2014.

13.2 Componentes JEE – Camada Cliente

Na camada cliente de um EJB podemos destacar os seguintes itens pertinentes à camada de apresentação:

- Clientes *web* (*browser* ou *web services*);
- Acesso indireto a EJBs via camada *Web*;
- Mídia *web* estática (HTML, XML, GIF etc) geradas por componentes *web* no servidor;
- Clientes sempre “magros” (não contêm regra de negócio).

Os clientes de aplicação:

- Podem ter acesso direto a EJBs na camada de negócios;
- Consistem em aplicações de linha de comando, aplicações gráficas AWT ou Swing e permitem uma interface do usuário mais sofisticada.

Outros clientes também podem acessar componentes EJB. Alguns exemplos são apresentados a seguir:

Applets

- Podem ser clientes *Web* ou de aplicação;
- Têm acesso direto a EJBs via IIOP – Internet Inter-ORB Protocol (pertence ao Java RMI).

Servlets

- Classes pré-compiladas que processam requisições HTTP e devolvem respostas HTTP de qualquer tipo.
- Ideais para o gerenciamento de conteúdo dinâmico que não é enviado para o browser como texto (imagens, vídeos, arquivos *zip*, *flash* etc).
- Usados como controladores em aplicações JSP.

Java Server Pages (JSP)

- Páginas de texto contendo Java embutido que operam como servlets;
- Compiladas após a instalação ou durante a execução;
- Ideais para gerar páginas de texto, HTML e XML (porta de comunicação para *Web Services*).

13.3 Características dos EJBs

EJBs são escritos em Java, seu funcionamento é semelhante a um Servlet. A principal diferença é que eles podem comportar acesso remoto via *Web Services*, além de serem autossuficientes como componente, ou seja, eles se comportam como se eles próprios fossem programas isolados. Podemos destacar as seguintes características que tornam um EJB tão atraente:

- Formam o núcleo de uma aplicação distribuída;
 - Recebem e processam dados de clientes e enviam dados (de forma transparente) à camada de armazenamento;
 - Recuperam dados da camada de dados, processam e enviam para clientes.
- São desenvolvidos usando RMI sobre IIOP: modelo de programação Java que gera objetos CORBA.

13.4 Servidores de Aplicações para suporte EJB

Servidores de aplicação OO, também chamados de Component Transaction Monitors, oferecem ambiente para a operação de componentes (rodando em containers) e diversos serviços de *middleware*.

Servidores de aplicação que suportam EJB fornecem diversos serviços de *middleware*. Entre eles, destacamos:

- Controle de transações;
 - Autenticação e autorização;
 - Gerenciamento de recursos;
 - Persistência;
 - Acesso remoto (programação distribuída);
 - Nos servidores EJB esses serviços são configurados através de arquivos XML e de anotações;
 - Os serviços também podem ser usados de forma explícita.
- Implementações: JBoss, Weblogic, Websphere, Glassfish etc.

13.5 A API JEE

Na API JEE existem diversos pacotes que podemos usar para definir nossos componentes EJB. Essa API está disponível nos servidores de aplicações compatíveis com EJB, como o JBoss, WebLogic, entre outros. As classes desta API estão distribuídas em pacotes. São eles:

- **javax.activation:** Javax Activation *framework* usado pelo JavaMail;
- **javax.ejb:** classes e interfaces para construir EJBs;
- **javax.jms:** classes e interfaces para construir aplicações JMS;
- **javax.mail:** classes que modelam um cliente de *e-mail*;
- **javax.resource:** JCA: API para desenvolvimento de conectores (RARs);
- **javax.security.auth:** JAAS: API de autenticação e autorização;
- **javax.servlet:** classes e interfaces para construir servlets e páginas JSP;
- **javax.sql:** pacote JDBC estendido;
- **javax.transaction:** JTA: API para controle de transações;
- **javax.xml.parsers:** JAXP: classes para processamento de XML;
- **javax.xml.transform:** classes para processamento de transformações XSLT;
- **org.w3c.dom:** implementação de DOM, componente do JAXP;
- **org.xml.sax:** implementação do SAX, componente do JAXP.

Aula 14 – A plataforma *Enterprise JavaBeans*

14.1 Definição da Plataforma EJB

Enterprise JavaBeans é uma plataforma para construção de aplicações corporativas portátil, reutilizável e escalável, que utiliza a linguagem Java. Desde seu início, o EJB tem sido considerado um modelo de componente ou *framework* que permite que se construam aplicações corporativas Java sem ter de reinventar serviços necessários para a construção de uma aplicação, como as transações, segurança, persistência automatizada e assim por diante.

O EJB permite que os desenvolvedores de aplicação se foquem na construção corporativa lógica, sem ter de perder tempo na construção do código de infraestrutura.

Do ponto de vista do desenvolvedor, um EJB é um pedaço do código Java que executa em um ambiente de tempo de execução especializado, chamado **container EJB**, que fornece um certo número de compo-

nentes de serviços. Os serviços de persistência são fornecidos por um *framework* especializado chamado de fornecedor de persistência.

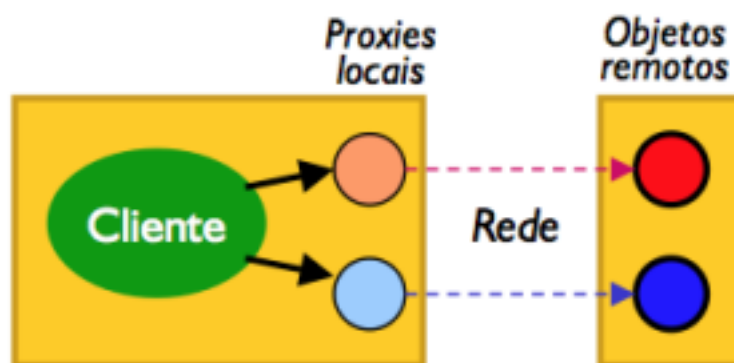
Resumindo, o EJB consiste em um sistema de objetos distribuídos que têm em comum uma arquitetura que consiste de:

- o objeto remoto;
- um stub/proxy que representa o objeto remoto no cliente;
- um esqueleto (skeleton) que representa o cliente no servidor.

O cliente conversa com o stub pensando se tratar do próprio objeto. O esqueleto conversa com o objeto remoto que pensa ser o próprio cliente que fala com ele.

O stub e o objeto remoto implementam a mesma interface! A Figura 14.1 mostra a conversa entre o stub (local) e o objeto remoto.

Figura 14.1 – Remote Method Invocation (RMI)



Fonte: <<http://digitalroad.files.wordpress.com/2011/07/capturadetela2011-07-06a25cc2580s21-34-12.png?w=300>>. Acesso em: 05 nov. 2014.

14.2 EJB como um componente

Quando falamos sobre EJBs, estamos nos referindo aos componentes server-side que podem ser utilizados para construir partes de sua aplicação, como a lógica corporativa ou o código de persistência.

No mundo do EJB 3, um componente é o que deve ser: um POJO com alguns poderes especiais. Esses poderes permanecem invisíveis até que sejam necessários e não se descuidam do propósito real do componente.

A ideia real por trás de um componente é que ele encapsule com eficácia o comportamento da aplicação. Os usuários de um componente não são necessários para conhecer suas operações internas. Tudo o que precisa saber é como ir além e o que esperar de volta.

14.3 Entendendo os tipos EJB

Enquanto muitos *frameworks* de fonte aberta e comerciais disponíveis podem simplificar o desenvolvimento da aplicação, o EJB é um *framework* atrativo que tem muito a oferecer. Vamos ver como utilizar o EJB como um *framework* para construir sua lógica de negócios e camadas de persistência de suas aplicações, começando com os *beans*.



Fonte: <<http://biiteducation.com/images/ejb3logo1.png>>. Acesso em: 07 nov. 2014.

Na linguagem EJB, um componente é um “*bean*”. O EJB classifica os *beans* em três tipos, baseados em sua utilidade:

- *Beans* de Sessão;
- *Beans* de Entidade;
- Message Driven Beans.

Cada tipo de *bean* tem um propósito e pode utilizar um subconjunto específico de serviços EJB.

A classificação do *bean* também ajuda você a entender e organizar uma aplicação de uma maneira sensata: por exemplo, os tipos de *beans* ajudam você a desenvolver aplicações baseadas em uma arquitetura multi-camada.

Os *beans* de sessão e os *beans* dirigidos por mensagens (Message Driven Beans- MDBs) são usados para construir lógica de negócios e residem em um container, que gerencia esses *beans* e fornece serviço a eles. As entidades são usadas para modelar a persistência de uma aplicação e, como o container, são um fornecedor de persistência que gerencia as entidades. Um fornecedor de persistência é plugável dentro do container e extraído no *Java Persistence API* (JPA).

14.4 Beans de sessão (Session beans)

Os session *beans* são *beans* com o propósito de realizar alguma operação de negócio específica, como consultar o histórico de crédito para um cliente.

Um *session bean* pode ser definido tanto com estado ou sem estado. Um ***session bean com estado (stateful)*** é típico para um carrinho de compras, que vai adicionado itens ao carrinho passo a passo, por isso o *bean* precisa manter o estado. Já um ***session bean sem estado (stateless)*** é um *bean* típico para consultar o histórico de crédito para um cliente, operação que não precisa guardar estado.

Um *session bean* pode ser executado localmente ou de forma remota por RMI. O *session bean* sem estado pode ser exposto como um *WebService*. Um *session bean* não sobrevive à queda ou parada temporária do servidor. Os *session beans* são gerenciados pelo container EJB.

14.5 Beans dirigidos por mensagens (Message Driven Beans)

Assim como os *session beans*, *Message Driven Beans* (MDBs) têm como propósito realizar operações de negócio, só que não são executados diretamente. Os MDBs são invocados através de mensagens enviadas ao servidor de mensagens que permitem o envio de mensagens assíncronas. Exemplos de servidores de mensagens IBM *WebSphere MQ*, *SonicMQ*, *Oracle Advanced Queueing* e *TIBCO*.

Sua utilização fundamental é para o envio de mensagem assíncrona, como uma solicitação para reabastecer o estoque de uma loja.

Os *session beans* são gerenciados pelo container EJB.

14.6 Entidades e Java Persistence API (Entity Beans)

Os *Entities Beans*, acredito que sejam os *beans* mais conhecidos e mais utilizados. Referem-se às entidades JPA que não necessitam de um container EJB para serem executadas.

JPA é uma especificação para os *frameworks* de ORM. As implementações mais conhecidas são pelos *frameworks* *Hibernate* e *Oracle TopLink*.

Entity bean é a forma que EJB 3 controla a persistência de dados, definindo o padrão para:

- Criação de metadados: mapeamento de entidades para tabelas do modelo relacional;
- API *EntityManager*: API para realização de CRUD (*Create, Retrieve, Update and Delete*);
- *Java Persistence Query Language (JPQL)*: linguagem que se parece com o *SQL ANSI* para pesquisa de dados persistidos.

Os *entities beans* não necessitam de um container EJB para serem executados.

14.7 Resumo dos Componentes EJB

Session beans

- Modelam processos de negócio. São ações, verbos.
- Fazem coisas: acessam um banco de dados, fazem contas.
- Podem manter ou não estado não-persistente.

- Exemplos: processar informação, comprar produto, validar cartão.

Entity Beans

- Modelam dados de negócios. São coisas, substantivos.
- Representam informações em bancos de dados.
- Mantêm estado persistente.
- Exemplos: um produto, um empregado, um pedido.

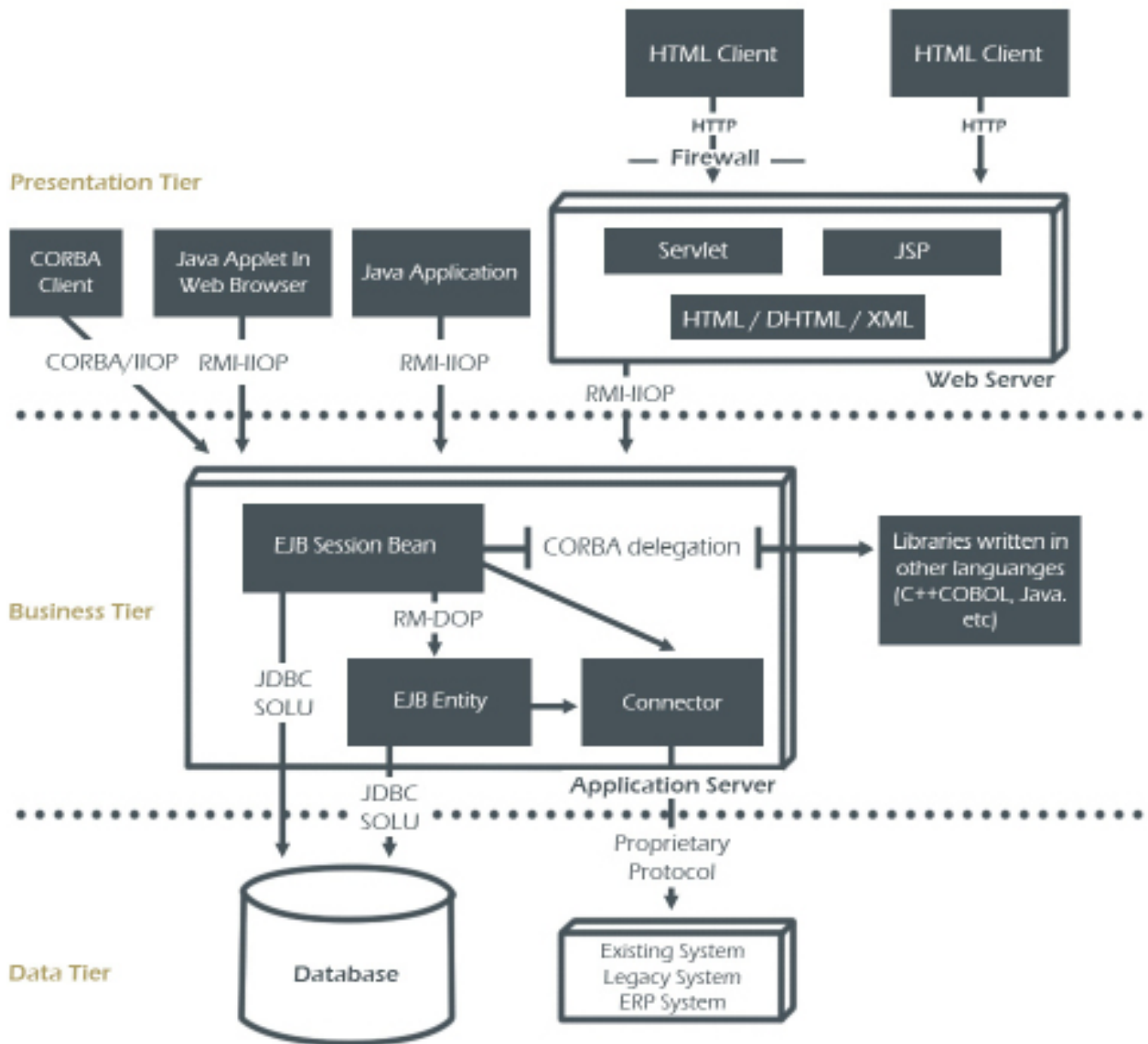
Message Driven Beans

- Modelam processos assíncronos. Respondem a eventos.
- Agem somente quando recebem uma mensagem.
- Não mantêm estado.

Aula 15 - Utilização completa de EJB

15.1 Exemplo de Utilização

A Figura 15.1 representa a arquitetura completa de uma aplicação envolvendo cliente HTML, páginas JSP, Servlets e os componentes EJB capazes de se comunicarem com outros componentes. A aplicação de exemplo tratará da persistência de uma entidade referente à classe Book.

Figura 15.1 – Arquitetura EJB

Fonte: <http://s3.amazonaws.com/magoo/ABAAfV_0AI-3.jpg>. Acesso em: 05 nov. 2014.

15.2 Criando um Entity Bean

O Quadro 64 apresenta a classe Book, a ser usada como elemento de persistência, representando assim um Entity Bean.

Quadro 64: Definindo um Entity Bean

```
import java.io.Serializable;

public class Book implements Serializable {
    private Integer id;
    private String title;
    private String author;

    public Book() {
        super();
    }

    public Book(Integer id, String title, String author) {
        super();
        this.id = id;
        this.title = title;
        this.author = author;
    }

    @Override
    public String toString() {
        return "Book: " + getId() + " Title " + getTitle() + " Author "
            + getAuthor();
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public Integer getId() {
        return id;
    }
}
```

```
public void setId(Integer id) {  
    this.id = id;  
}  
  
public String getTitle() {  
    return title;  
}  
  
public void setTitle(String title) {  
    this.title = title;  
}  
}
```

15.3 Adicionando anotações JPA

O Quadro 65 apresenta a inclusão das anotações JPA à classe `Book`, tornando-a um *Entity Bean*.

Quadro 65: Anotações em Entity Beans

```
@Entity  
@Table(name="book")  
@SequenceGenerator(name = "book_sequence", sequenceName = "book_id_seq")  
public class Book implements Serializable {  
    ...  
}
```

@Entity define um objeto desta classe como um *bean* de entidade.

@Table define o nome da tabela.

@SequenceGenerator define o gerador de sequência.

As chaves primárias podem ser criadas de diferentes formas. É possível atribuir seus valores, ou gerá-los através de uma sequência numérica, como pode ser visto no Quadro 66.

Quadro 66: Anotações em Entity Beans - @Id

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "book_sequence")
public Integer getId() {
    return id;
}
```

15.4 Criando interfaces local e remota

A interface local deve ser usada por default, por ser bem mais rápida. A interface remota deve ser usada somente quando o cliente não está sendo executado na mesma máquina virtual. O acesso remoto normalmente produz sobrecarga.

Vamos criar uma interface chamada **BookTestBeanLocal**. Ela será marcada com a anotação **@Local**. Veja o Quadro 67.

Quadro 67: A interface EJB local

```
import javax.ejb.Local;
@Local
public interface BookTestBeanLocal {
    public void test();
}
```

Vamos criar, agora uma interface chamada **BookTestBeanRemote**. Veja o Quadro 68.

Quadro 68: A interface EJB remota

```
import javax.ejb.Remote;
@Remote
public interface BookTestBeanRemote {
    public void test();
}
```

Agora será criado o *Stateless Session bean*.

Criamos uma nova classe no mesmo pacote das interfaces e fazemos com que ela implemente tanto a interface local como a interface remota. A classe é definida como um "*Stateless Session bean*" através da anotação **@Stateless**. Veja o Quadro 69.

Quadro 69: *Session bean* sem estado

```
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class BookTestBean implements BookTestBeanLocal, BookTestBeanRemote {
    @PersistenceContext
    EntityManager em;
    public static final String RemoteJNDIName = BookTestBean.class.getSimpleName() + "/remote";

    public static final String LocalJNDIName = BookTestBean.class.getSimpleName() + "/local";
}
```

Estamos interessados agora em acessar o *bean book* e, para tanto, necessitaremos de um *EntityManager*, que fornece todos os métodos necessários para manipular as entidades (select, update etc) e para criar as queries SQL e EJB-QL queries. Veja o Quadro 70.

Quadro 70: Definindo o Entity Manager

```
@PersistenceContext
EntityManager em;
```

A anotação *@PersistenceContext* instrui o servidor de aplicações a injetar uma entidade durante a criação da aplicação (deployment). Injetar significa que a entidade é atribuída pelo servidor de aplicações.

O método do Quadro 71 cria uma entrada, seleciona e remove alguns dados. Tudo é feito através do objeto *entity manager*.

Quadro 71: O Entity Manager completo

```
import java.util.Iterator;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class BookTestBean implements BookTestBeanLocal, BookTestBeanRemote {
    @PersistenceContext
    EntityManager em;
```

```

public static final String RemoteJNDIName = BookTestBean.class.getSimpleName() + "/"
remote";

public static final String LocalJNDIName = BookTestBean.class.getSimpleName() + "/"
local";

public void test() {
    Book book = new Book(null, "Primeiro Livro", "Adalberto Santos");
    em.persist(book);
    Book book2 = new Book(null, "Java na WEB", "Emilio");
    em.persist(book2);
    Book book3 = new Book(null, "EJB 3", "Cristiana Barbosa");
    em.persist(book3);

    System.out.println("Listando alguns livros ");

    List someBooks = em.createQuery("from Book b where b.author=:name")
.setParameter("name", "Emilio").getResultList();

    for (Iterator iter = someBooks.iterator(); iter.hasNext();)
    {
        Book element = (Book) iter.next();
        System.out.println(element);
    }

    System.out.println("Listando todos os livros ");
    List allBooks = em.createQuery("from Book").getResultList();

    for (Iterator iter = allBooks.iterator(); iter.hasNext();)
    {
        Book element = (Book) iter.next();
        System.out.println(element);
    }
}

```

```
System.out.println("removendo um livro");
em.remove(book2);

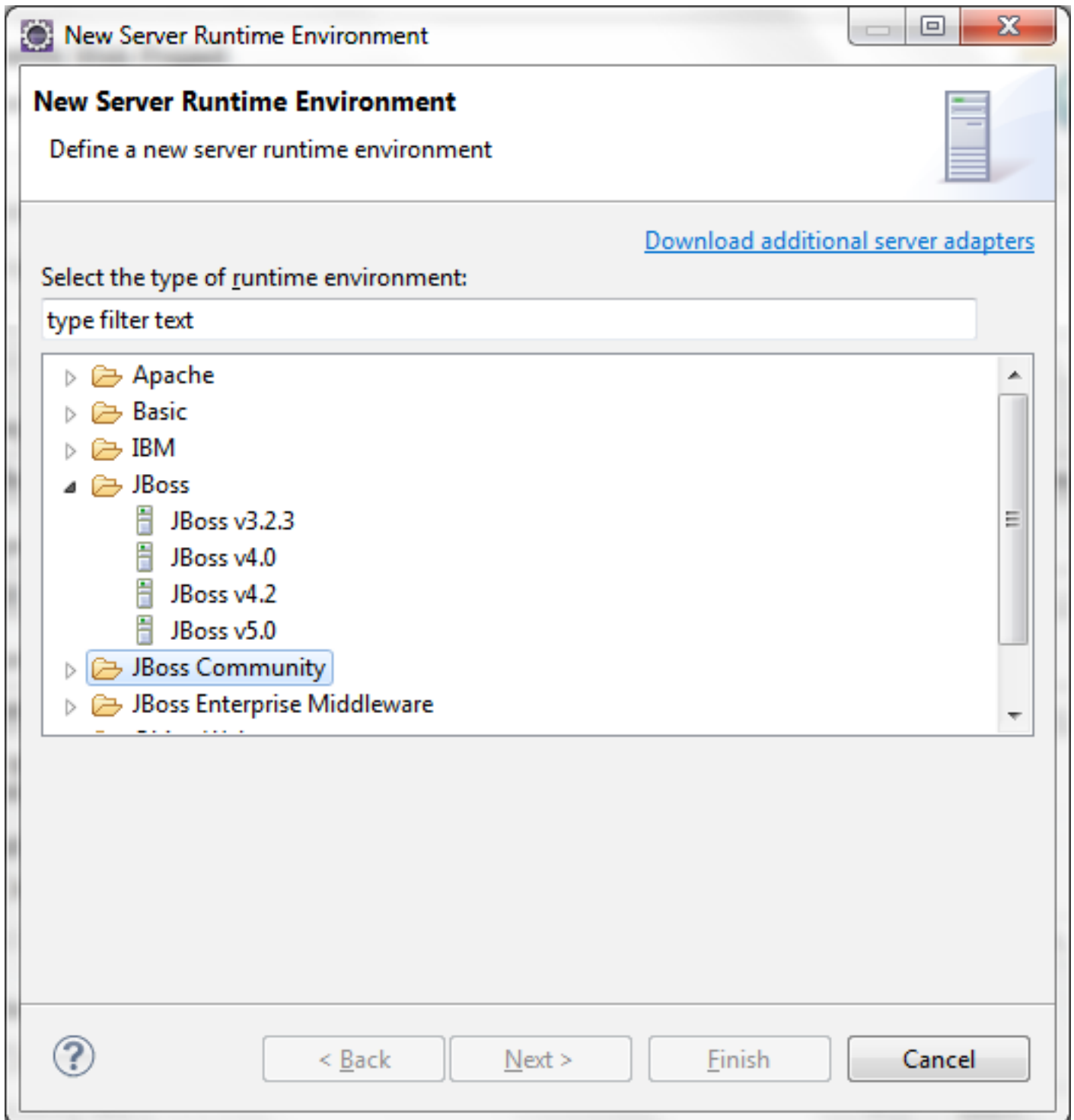
System.out.println("Listando todos os livros ");
allBooks = em.createQuery("from Book").getResultList();

for (Iterator iter = allBooks.iterator(); iter.hasNext();)
{
    Book element = (Book) iter.next();
    System.out.println(element);
}
}
```

Aula 16 - Preparação do Eclipse para executar a versão 7.x do servidor JBoss

16.1 Instruções passo a passo

Quando instalamos o JDK e o Eclipse (versão Kepler) temos a seguinte lista de servidores, mostrada na Figura 16.1a, disponível para inclusão do servidor JBoss, ao criarmos um projeto "**Dynamic Web Project**":

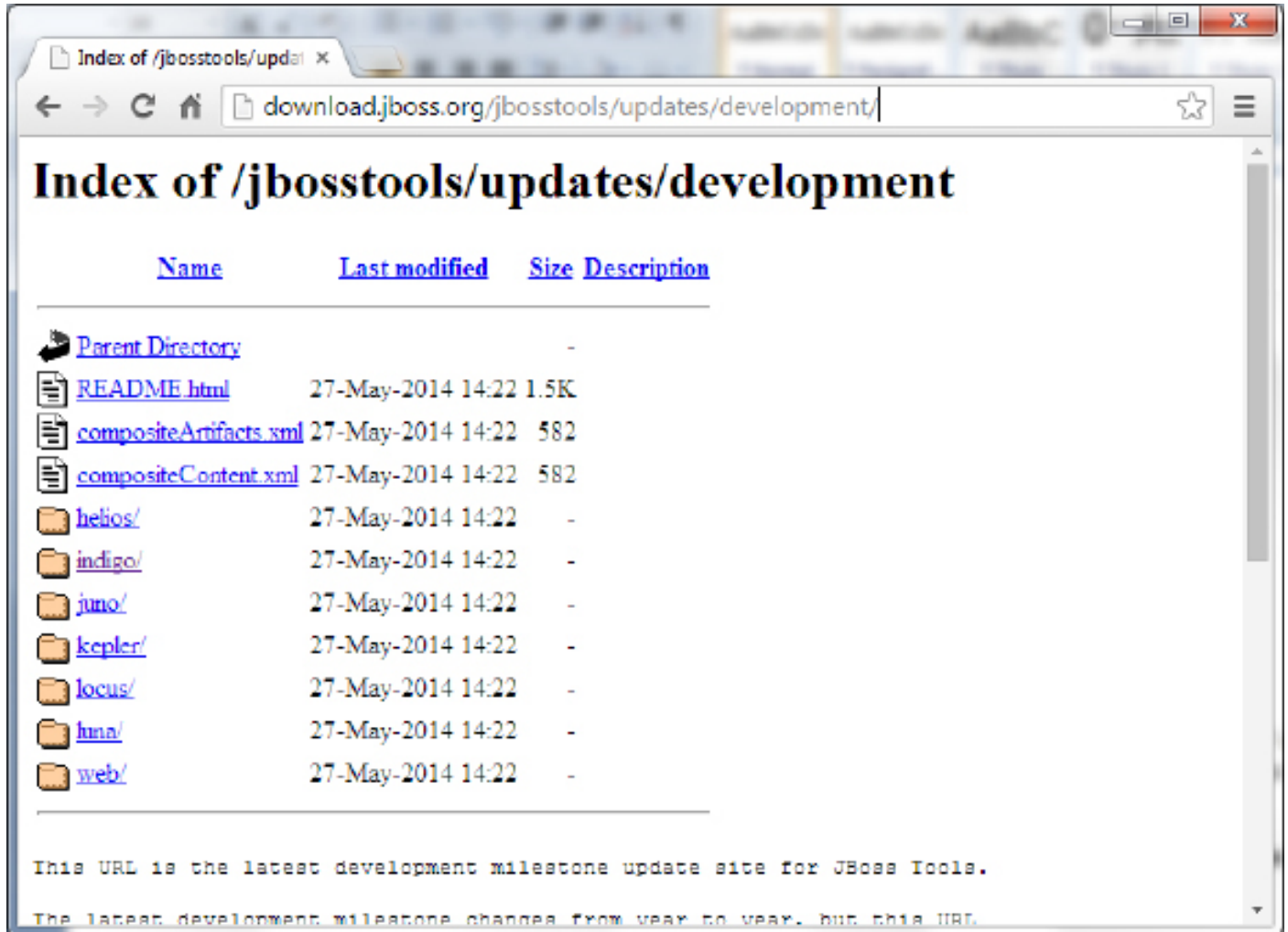
Figura 16.1a – Escolha do servidor no Eclipse

A lista de servidores é proveniente dos *plugins* disponíveis. Para incluirmos o JBoss 7 é necessário configurar o eclipse com os *plugins* necessários. Para tanto, devemos seguir os passos a seguir:

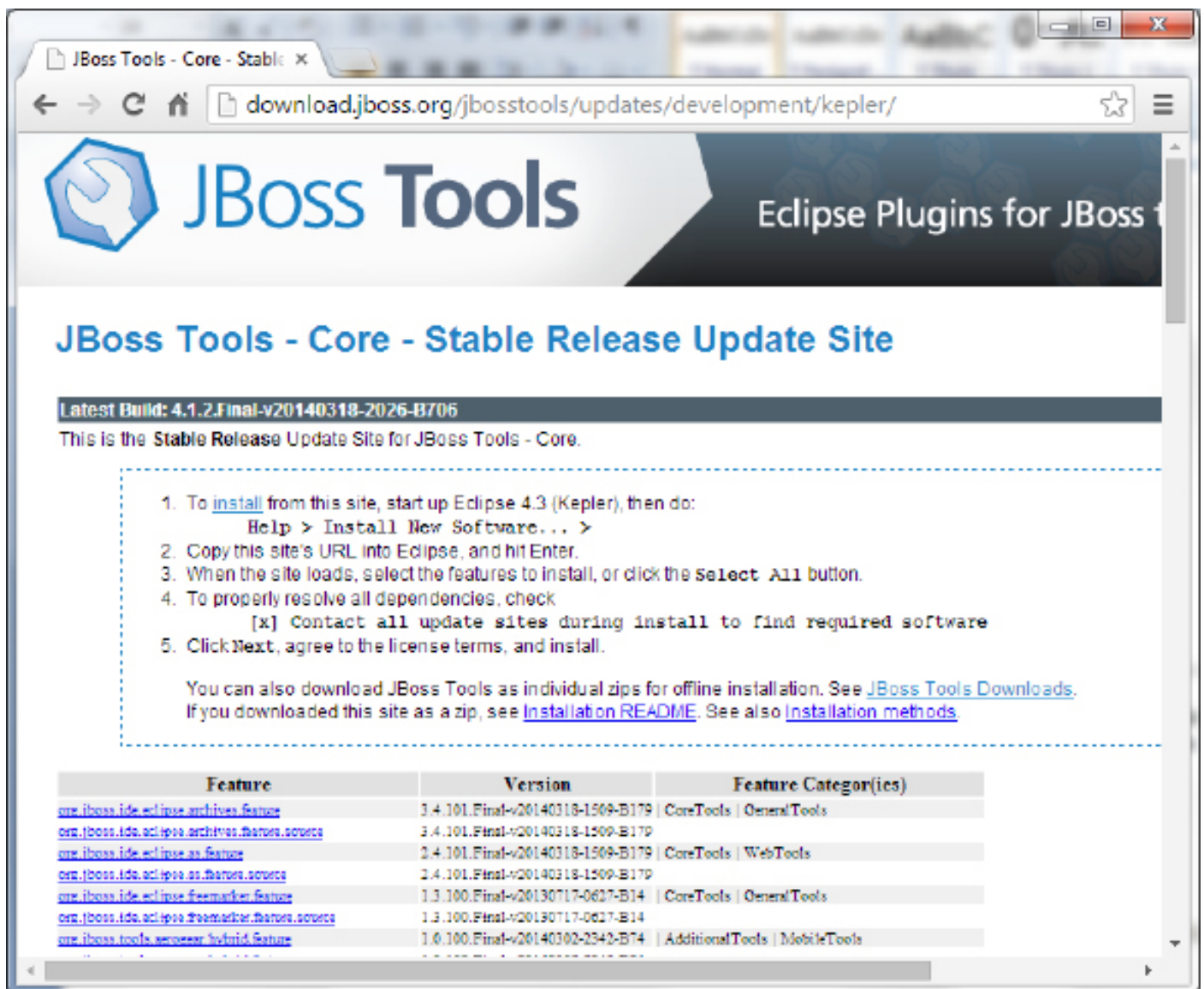
1. Acessar o *link*: <http://download.jboss.org/jbosstools/updates/development/>. (Na data da elaboração deste material, o *link* apresentou a tela mostrada na sequência deste exemplo. Devido

à natureza dinâmica da *web*, é possível que haja algumas alterações no futuro). Obteremos a tela mostrada na Figura 16.1b:

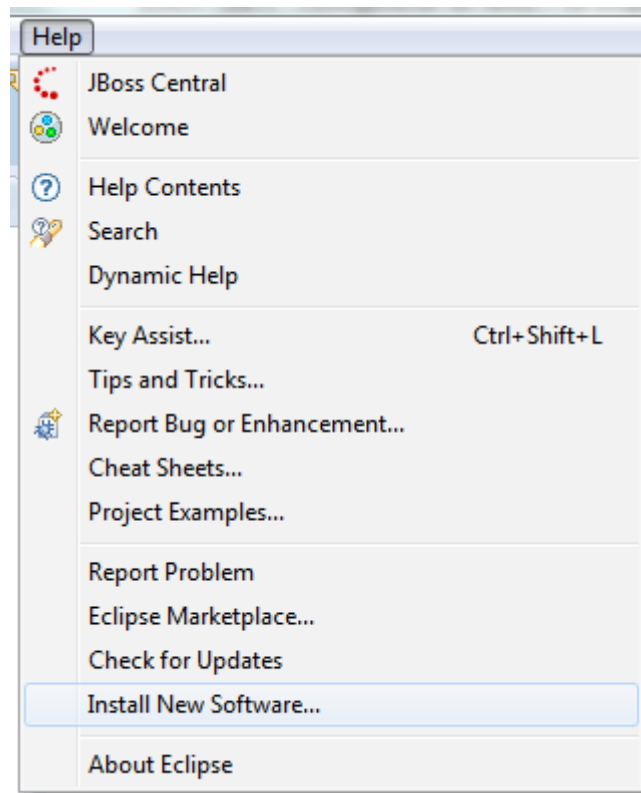
Figura 16.1b – Busca pela versão do Eclipse em uso



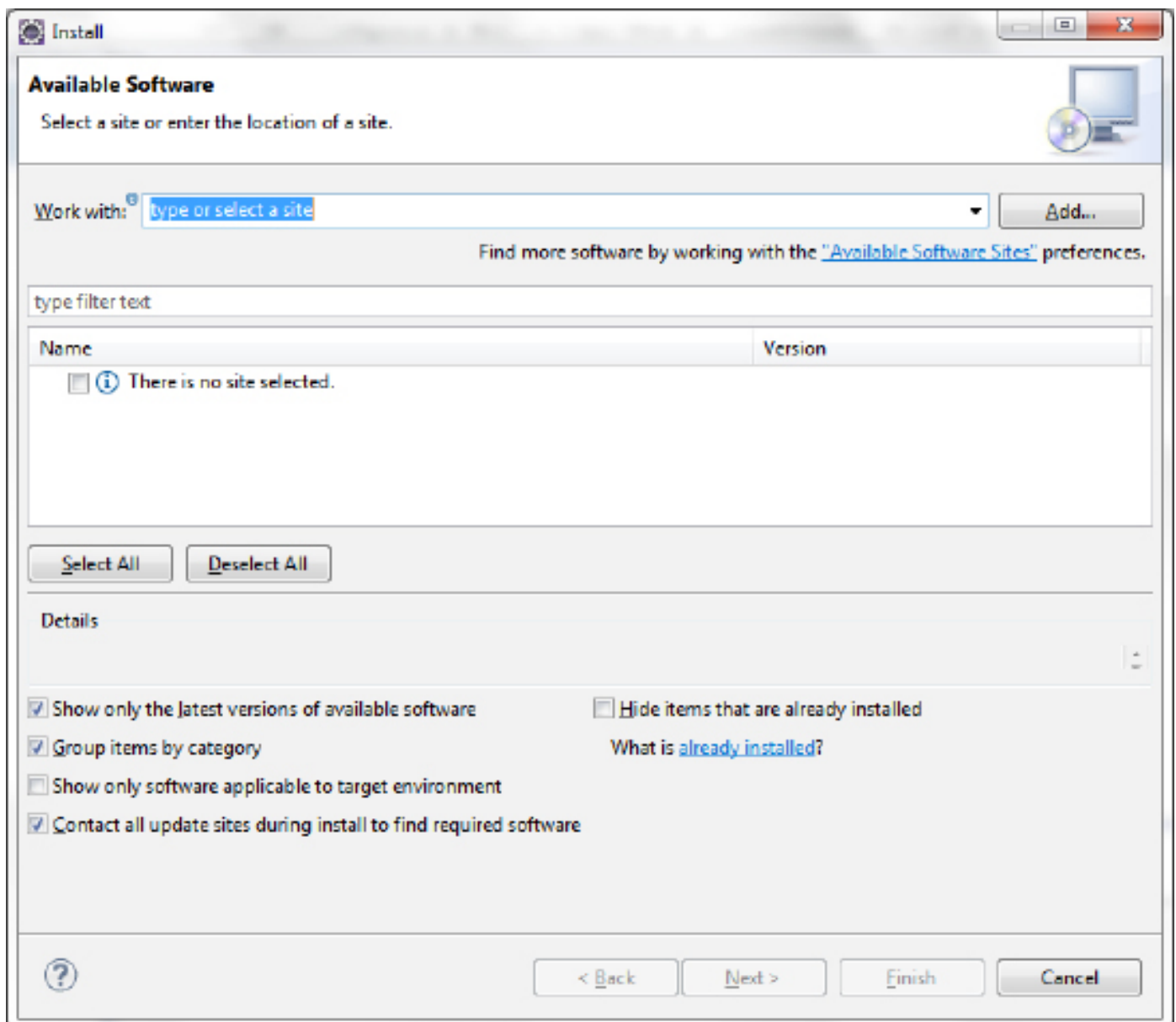
- Na lista apresentada, selecione a versão do Eclipse que você está utilizando. Se estiver utilizando a versão Kepler, selecione-a na lista ou acrescente a palavra "Kepler" ao *link* do item 1. Você obterá a lista mostrada na Figura 16.1c:

Figura 16.1c – Escolha do *plugin* do JBoss

3. No eclipse, selecione o menu: **Help -> Install New Software...** como mostra a Figura 16.1d:

Figura 16.1d – Atalho para instalar novo software

4. Na janela clique em Add... (Figura 16.1e):

Figura 16.1e – Tela para inclusão do *link* para instalação do novo software

5. Fornecer um nome para o novo software, e em Location, informar o *link* do item 3, como mostram as Figuras 16.1f e 16.1g.

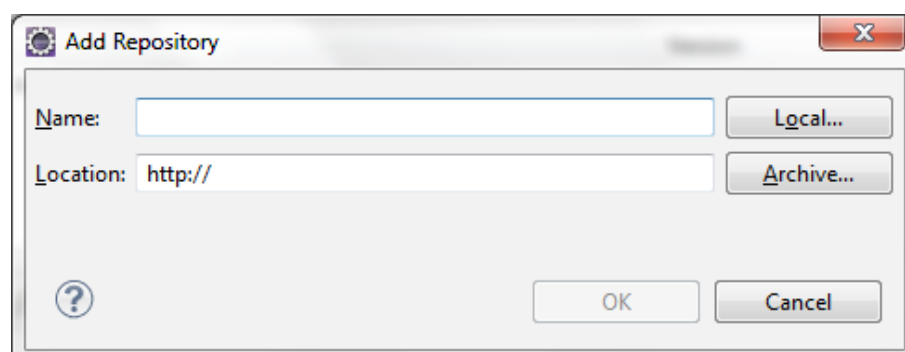
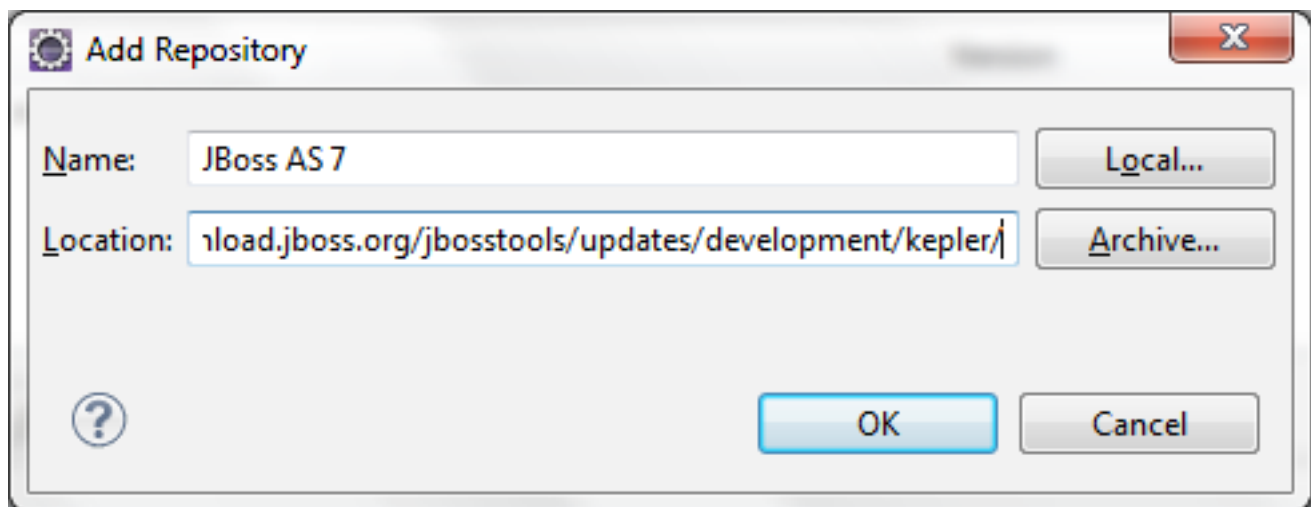
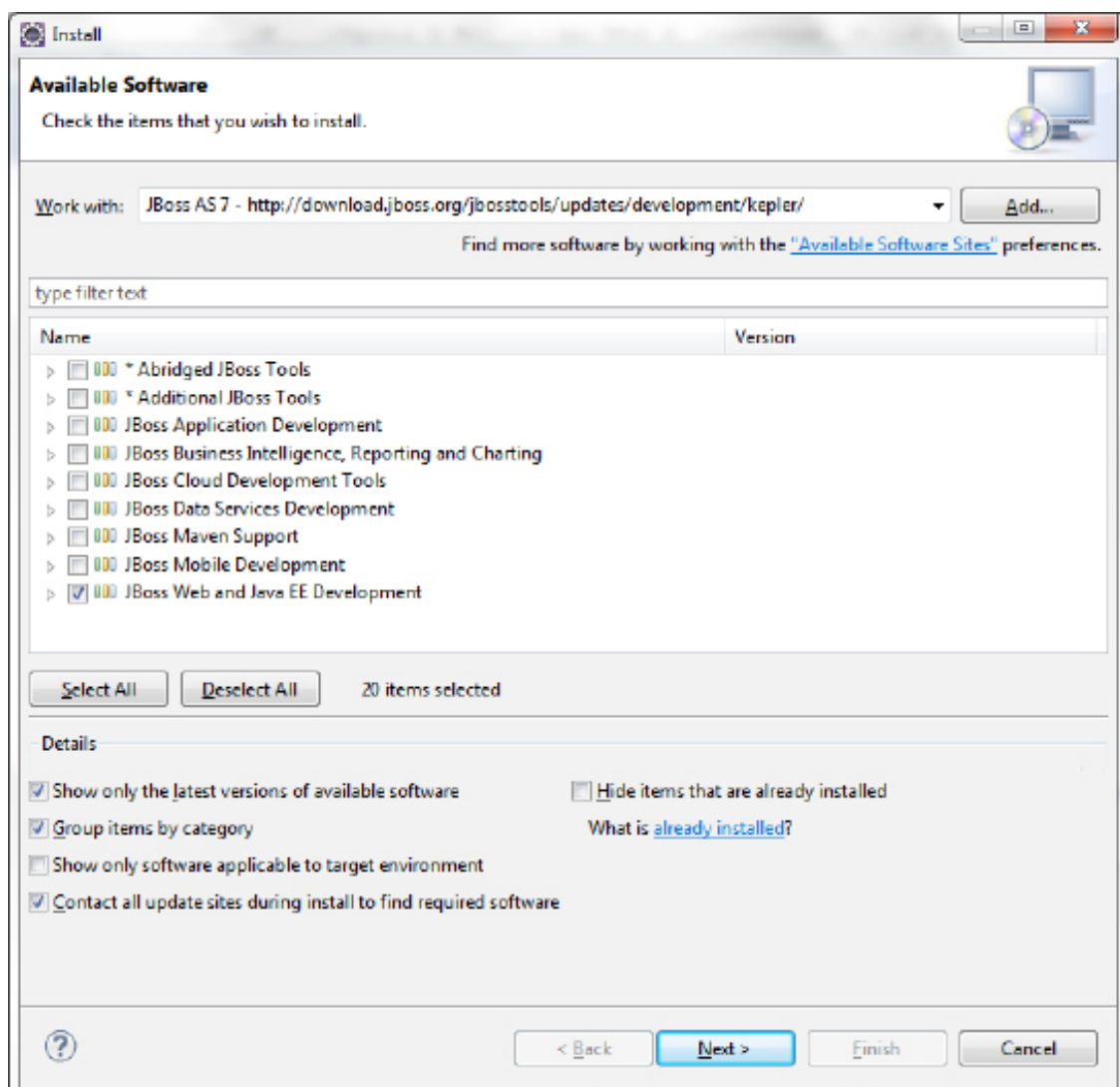
Figuras 16.1f

Figura 16.1g – Configuração da instalação

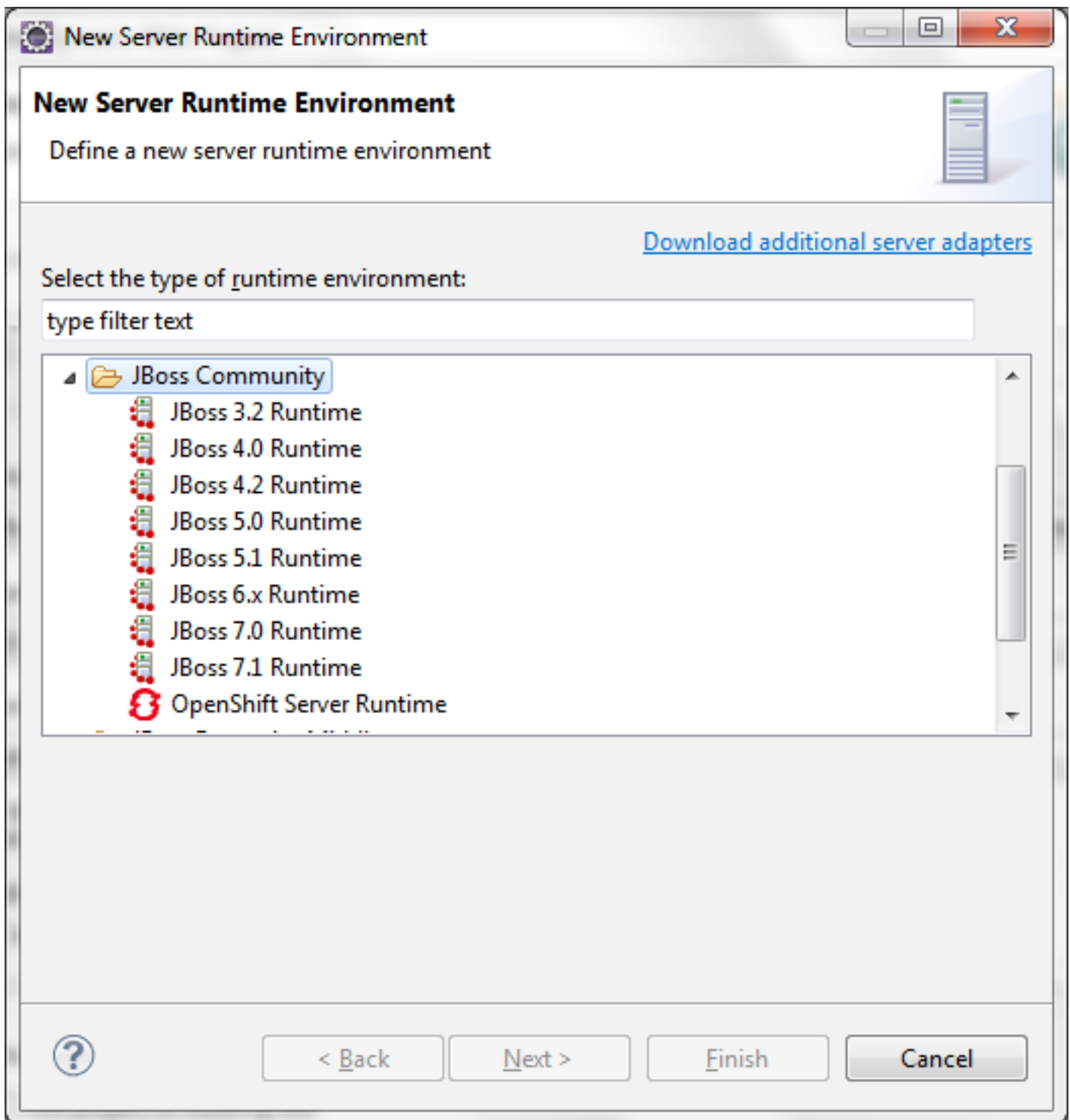
6. Aceite a opção (OK) e aguarde a lista de opções.
7. Na lista seleccionar a opção indicada na Figura 16.1h

Figura 16.1h – Finalização da instalação

Na próxima etapa aceite os termos de licença e prossiga.

8. Ao finalizar você deve reiniciar o Eclipse. A partir deste ponto você poderá utilizar as versões do JBoss mostradas na Figura 16.1i.

Figura 16.1i – Seleção do JBoss após instalado seu plugin no Eclipse



Pronto! Aproveite os benefícios do JBoss!

Exercícios do Módulo 3

Exercício 1

Componentes de servidor e cliente no mesmo projeto.

- No Eclipse, criar um projeto do tipo **EJB Project**. Nomeá-lo como **01_FolhaPagamento**. Incluir o servidor **JBoss AS 7.1**, e manter as configurações mostradas a seguir:

New EJB Project

Create an EJB Project and add it to a new or existing Enterprise Application.

Project name: 01_FolhaPagamento

Project location

☒ Use default location

Location: C:\Documentos\projetos\Java\EBJ_Aulas\01_FolhaPagamer Browse...

Target runtime

JBoss 7.1 Runtime New Runtime...

EJB module version

3.0

Configuration

<custom> Modify...

Hint: Get started quickly by selecting one of the pre-defined project configurations.

EAR membership

☒ Add project to an EAR

EAR project name: 01_FolhaPagamentoEAR New Project...

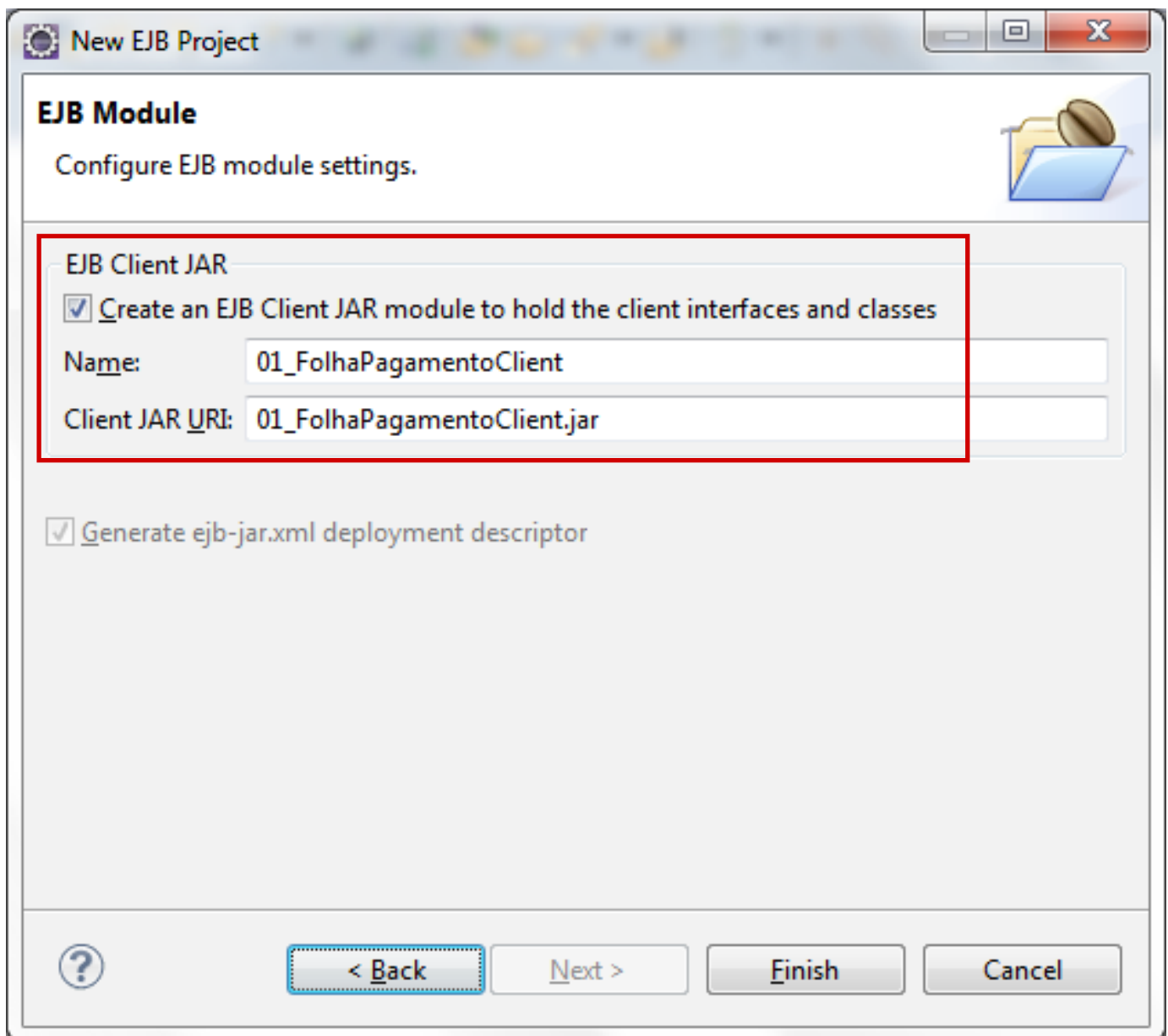
Working sets

☐ Add project to working sets

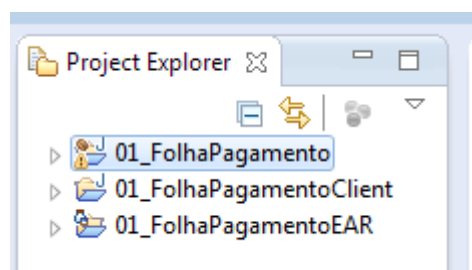
Working sets: Select...

Buttons: ? < Back Next > Finish Cancel

- Avance duas vezes, e mantenha a seleção a seguir:



- Ao finalizar, devemos ter os três projetos listados a seguir:



Os projetos possuirão os seguintes elementos:

- **01_FolhaPagamento** – Classes que implementam os *Stateless Session Beans*.
- **01_FolhaPagamentoClient** – Interfaces que definem as operações dos *Stateless Session Beans*.

- **01_FolhaPagamentoEAR** – Responsável por empacotar todos os módulos da aplicação.
- Criar um novo projeto *WEB* (Dynamic Web Project) para consumir o EJB. Nomeie-o como **01_FolhaPagamentoWeb**, mantendo as configurações mostradas a seguir:

New Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: 01_FolhaPagamentoWeb

Project location

☒ Use default location

Location: C:\Documentos\projetos\Java\EJB_Aulas\01_FolhaPagamer Browse...

Target runtime

JBoss 7.1 Runtime New Runtime...

Dynamic web module version

3.0

Configuration

Default Configuration for JBoss 7.1 Runtime Modify...

A good starting point for working with JBoss 7.1 Runtime runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership

☒ Add project to an EAR

EAR project name: 01_FolhaPagamentoEAR New Project...

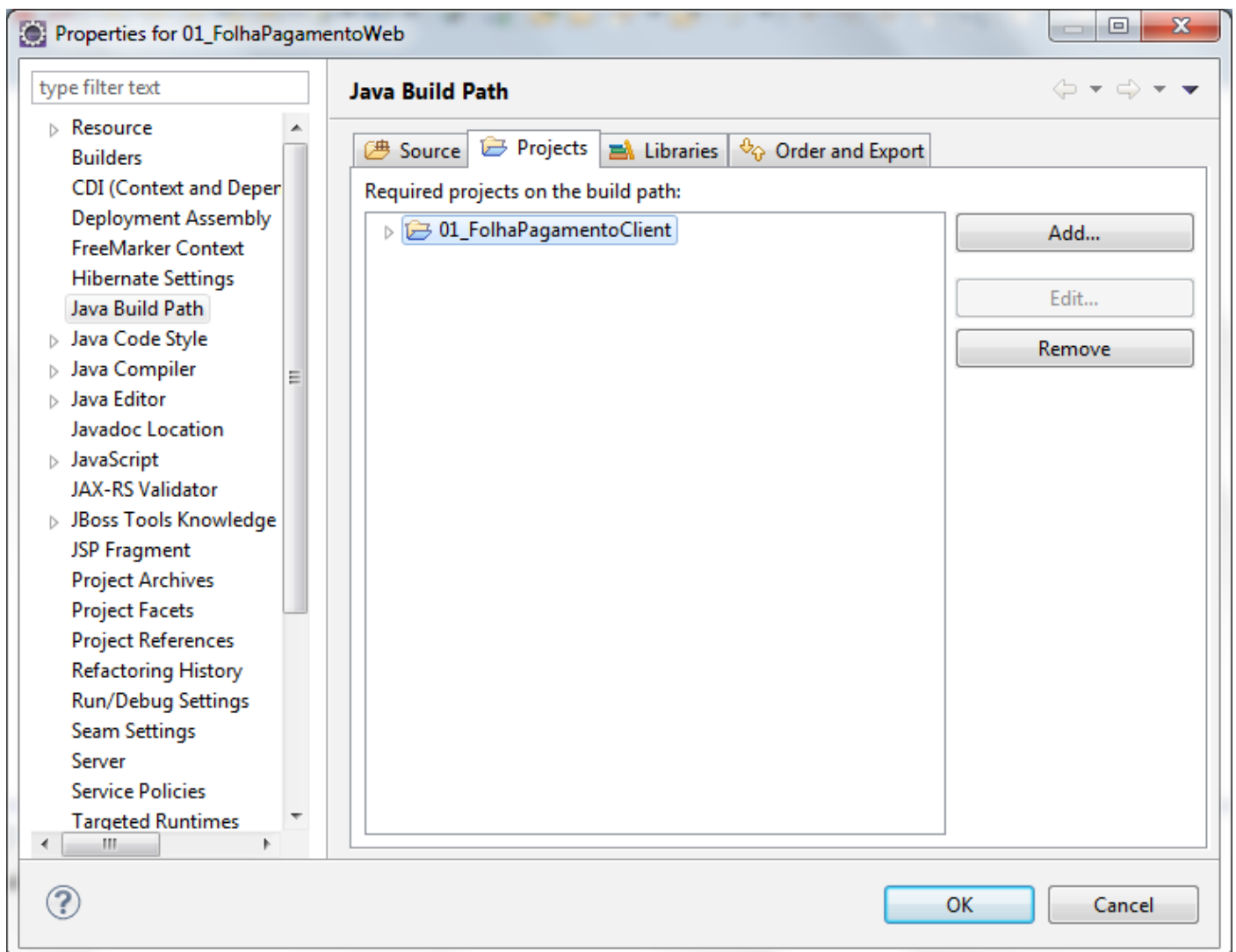
Working sets

☐ Add project to working sets

Working sets: Select...

? < Back Next > Finish Cancel

- Adicionar o projeto **01_FolhaPagamentoClient** como dependência do novo projeto *web* (Configuração do Build Path):



- No projeto **01_FolhaPagamentoClient** criar uma interface chamada **FolhaPagamento**, cujo modelo é ilustrado a seguir:

```
package br.com.ead.interfaces;

public interface FolhaPagamento {
    void setSalario(double salario);
    double calcularINSS(double taxa);
    double calcularSalarioLiquido();
}
```

- No projeto 01_FolhaPagamento criar a classe *FolhaPagamentoBean*, conforme ilustrado a seguir:

```

package br.com.ead.bean;

import javax.ejb.Local;
import javax.ejb.Stateless;

import br.com.ead.interfaces.FolhaPagamento;

@Stateless
@Local(FolhaPagamento.class)
public class FolhaPagamentoBean implements FolhaPagamento{

    private double salario;

    @Override
    public double calcularINSS(double taxa) {
        return salario * taxa / 100;
    }

    @Override
    public double calcularSalarioLiquido() {
        return salario - calcularINSS(10);
    }
}

```

- No projeto **01_FolhaPagamentoWeb**, criar um Servlet e uma página JSP, de acordo com os modelos dados a seguir:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Salarios</title>
</head>
<body>
    <form action="folha" method="post">
        <table>
            <tr>
                <td>Informe o valor do salário:</td>
                <td><input type="text" name="salario" size="10" /></td>
            </tr>
        </table>
    </form>

```

```

        </tr>
        <tr>
            <td colspan="2"><input type="submit" value="Enviar" /></td>
        </tr>
    </table>
</form>
</body>
</html>

```

```

package br.com.ead.web;

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;

import br.com.ead.interfaces.FolhaPagamento;

@WebServlet("/folha")
public class ServletFolha extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @EJB
    FolhaPagamento fp;

    public ServletFolha() {
        super();
        // TODO Auto-generated constructor stub
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // TODO Auto-generated method stub
    }
}

```



```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
```

```
    PrintWriter out = response.getWriter();
    response.setContentType("text/html");
    try {
        int salario = Integer.parseInt(request.getParameter("salario"));
        fp.setSalario(salario);
        out.print("Salario Bruto: " + salario);
        out.print("<br/>Salario Liquido: " + fp.calcularSalarioLiquido());

        out.print("<br/>");

        out.print(fp.getCartao());

    } catch (Exception e) {
        out.print(e.getMessage());
    }
}
```

- Para executar, selecione a opção Run As... -> Run On Server e certifique-se de que o arquivo EAR esteja na lista de execução.
- Executa a página JSP, e constatar o resultado.

Exercício 2

Calculadora usando EJB e JSF:

- Criar um novo projeto **EJB Project** chamado **02_Calculadora** (Não precisa marcar a opção "Add Project to na EAR").
- Criar uma interface chamada **CalculadoraLocal**, que defina as quatro operações de uma calculadora. Marcar esta interface com a anotação **@Local**:

```
package br.com.ead.interfaces;

import javax.ejb.Local;

@Local
public interface CalculadoraLocal {
    double somar(double x, double y);
    double subtrair(double x, double y);
    double multiplicar(double x, double y);
    double dividir(double x, double y);
}
```

- Definir a classe que representará o componente EJB. A classe deverá chamar **CalculadoraBean**, e deverá implementar a interface do item 2:

```
package br.com.ead.bean;

import javax.ejb.Stateless;
import br.com.ead.interfaces.CalculadoraLocal;

@Stateless
public class CalculadoraBean implements CalculadoraLocal {

    @Override
    public double somar(double x, double y) {
        return x + y;
    }

    @Override
    public double subtrair(double x, double y) {
        return x - y;
    }

    @Override
    public double multiplicar(double x, double y) {
        return x * y;
    }

    @Override
    public double dividir(double x, double y) {
        return x / y;
    }
}
```

- Criar um novo projeto *web* (Dynamic Web Project) baseado em JSF, chamado **02_Calculadora-Web**. Neste projeto, selecionar a opção *JavaServer Faces 2.1* como Configuration:

New Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: 02_CalculadoraWeb

Project location

☒ Use default location

Location: C:\Documentos\projetos\Java\EBJ_Aulas\02_CalculadoraWeb Browse...

Target runtime

JBoss 7.1 Runtime New Runtime...

Dynamic web module version

3.0

Configuration

JavaServer Faces v2.1 Project Modify...

Configures a Dynamic Web application to use JSF v2.1

EAR membership

☐ Add project to an EAR

EAR project name: EAR New Project...

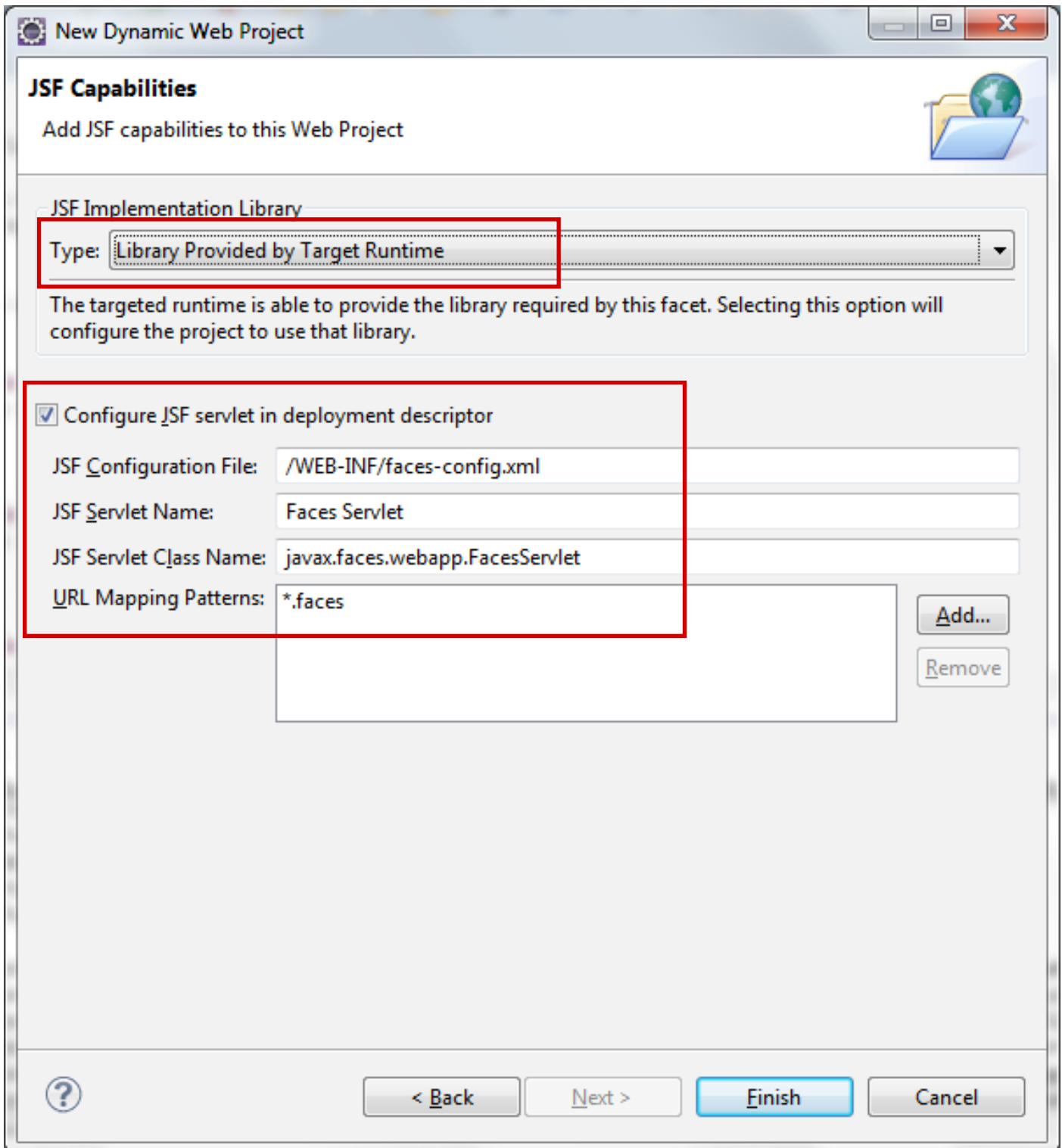
Working sets

☐ Add project to working sets

Working sets: Select...

? < Back Next > Finish Cancel

- Avançar, selecionar a opção que permite criar o arquivo web.xml, e manter a configuração a seguir:



New Dynamic Web Project

JSF Capabilities
Add JSF capabilities to this Web Project

JSF Implementation Library

Type: **Library Provided by Target Runtime**

The targeted runtime is able to provide the library required by this facet. Selecting this option will configure the project to use that library.

☒ Configure JSF servlet in deployment descriptor

JSF Configuration File: /WEB-INF/faces-config.xml

JSF Servlet Name: Faces Servlet

JSF Servlet Class Name: javax.faces.webapp.FacesServlet

URL Mapping Patterns: *.faces

Add...
Remove

? < Back Next > Finish Cancel

- Neste ponto, selecionar o pacote contendo a interface **CalculadoraLocal** e exportá-la para um arquivo **.jar**. Em seguida, importar este *jar* no seu projeto *web*. Como sugestão, chamar o arquivo de **calc.jar**.
- No projeto WEB criar um *Managed Bean* para conter os resultados:

```
package br.com.ead.mb;

import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;

import br.com.ead.interfaces.CalculadoraLocal;

@ManagedBean(name="calcMB")
public class CalculadoraManagedBean {

    //informamos o local do EJB via JNDI
    @EJB(lookup = "ejb:/02_Calculadora/CalculadoraBean!br.com.ead.interfaces.CalculadoraLocal")
    private CalculadoraLocal calc;

    private double x, y;
    private String resultado;

    public double getX() {
        return x;
    }
    public void setX(double x) {
        this.x = x;
    }
    public double getY() {
        return y;
    }
    public void setY(double y) {
        this.y = y;
    }

    public String getResultado() {
        return resultado;
    }
}
```

```

public void somar(){
    double result = calc.somar(x, y);
    resultado = "Soma = " + result;
}
public void subtrair() {
    double result = calc.subtrair(x, y);
    resultado = "Subtração = " + result;
}
public void multiplicar(){
    double result = calc.multiplicar(x, y);
    resultado = "Multiplicação = " + result;
}
public void dividir(){
    double result = calc.dividir(x, y);
    resultado = "Divisão = " + result;
}
}

```

Observe que incluímos o atributo "lookup" à anotação @EJB. Isso foi necessário porque o projeto *web* e o projeto EJB estão separados, não fazendo parte do mesmo EAR como ocorreu no exemplo 01. O valor desse atributo é proveniente de um mecanismo conhecido por JNDI (*Java Naming and Directory Interface*). O JBoss fornece uma facilidade para obter esse valor: basta executar o projeto EJB com a opção "Run on Server", que ela aparece no log do serviço.

- Definir, no projeto *web*, um arquivo xhtml para testar a calculadora remota!!!

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>Calculadora</title>
  </h:head>
  <h:body>
    <f:view>
      <h:form>

```

```

        <h:outputText value="Valor de X:" />
        <h:inputText value="#{calcMB.x}" />
        <br />
        <h:outputText value="Valor de Y:" />
        <h:inputText value="#{calcMB.y}" />
        <br />
        <h:commandButton value="Soma" action="#{calcMB.somar}" />
        <h:commandButton value="Subtração" action="#{calcMB.subtrair}" />
        <h:commandButton value="Multiplicação" action="#{calcMB.multiplicar}" />
        <h:commandButton value="Divisão" action="#{calcMB.dividir}" />
        <br />
        <h:outputText value="#{calcMB.resultado}" />
    </h:form>
</f:view>
</h:body>
</html>

```

- Para executar, manter os dois projetos na lista do servidor.

Exercício 3

Este exercício trata de um componente EJB que realiza persistência usando JPA. Para definir a camada de persistência, apresentaremos as etapas para sua realização. Como exercício o aluno definirá a camada de apresentação usando JSF.

- Definição do **datasource** (para várias aplicações, definir um datasource no servidor JBoss): na pasta **[JBoss]\modules\com\mysql\main** (criar a pasta):
 1. copiar o driver do mysql.
 2. criar o arquivo module.xml:

```

<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
    <resources>
        <resource-root path="mysql-connector-java-5.1.13-bin.jar" />
    </resources>
    <dependencies>
        <module name="javax.api" />
    </dependencies>
</module>

```

Localizar o arquivo standalone.xml na pasta **[JBoss]\standalone\configuration**. Dentro da tag `<datasources>` incluir o elemento:

```
<datasource jndi-name="java:/jdbc/ead" pool-name="ead" enabled="true" use-java-context="true">
  <connection-url>jdbc:mysql://localhost:3306/livrosejb</connection-url>
  <driver>com.mysql</driver>
  <security>
    <user-name>root</user-name>
    <password>ead</password>
  </security>
</datasource>
```

E dentro da tag `<drivers>` a seguinte configuração:

```
<driver name="com.mysql" module="com.mysql">
  <xa-datasource-class>com.mysql.jdbc.Driver</xa-datasource-class>
</driver>
```

- Criar um novo projeto EjbProject, chamado **03_LivrosEJB**. Selecionar JBoss 7.0 ou 7.1, mas manter a versão 3.0.
- Na pasta **ejbModule**, criar a classe para a entidade Livros:


```
package br.com.ead.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="livros", schema="livrosejb")
public class Livros {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="ID")
    private Integer id;

    @Column(name="TITULO")
    private String titulo;

    @Column(name="AUTOR")
    private String autor;

    @Column(name="PRECO")
    private Double preco;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getTitulo() {
        return titulo;
    }
}
```

```

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getAutor() {
        return autor;
    }
    public void setAutor(String autor) {
        this.autor = autor;
    }
    public Double getPreco() {
        return preco;
    }
    public void setPreco(Double preco) {
        this.preco = preco;
    }
}

```

Criar, na pasta \META-INF, o arquivo persistence.xml:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0 "
    xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="eadPU" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>jdbc/ead</jta-data-source>
        <properties>
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.
MySQL5InnoDBDialect" />
        </properties>
    </persistence-unit>
</persistence>

```

- Definir o banco de dados no MySQL.
- Criar o SessionBean e a interface local. Chamar a classe de **LivrosBean**.
- Codificar a classe e a interface:

```
package br.com.ead.bean;
```

```
import java.util.List;
```

```
import javax.ejb.Local;
```

```
import br.com.ead.entity.Livros;
```

```
@Local
```

```
public interface LivrosBeanLocal {
```

```
    void add(Livros livro);
```

```
    List<Livros> getAll();
```

```
}
```

```
package br.com.ead.bean;
```

```
import java.util.List;
```

```
import javax.ejb.Stateless;
```

```
import javax.persistence.EntityManager;
```

```
import javax.persistence.PersistenceContext;
```

```
import javax.persistence.TypedQuery;
```

```
import br.com.ead.entity.Livros;
```

```
@Stateless
```

```
public class LivrosBean implements LivrosBeanLocal {
```

```
    public LivrosBean() { }
```

```
    @PersistenceContext(unitName="eadPU")
```

```
    private EntityManager em;
```

```
    public void add(Livros livro){
```

```
        em.persist(livro);
```

```
    }
```

```
    public List<Livros> getAll(){
```

```
        TypedQuery<Livros> query = em.createQuery("select u from Livros u", Livros.class);
```

```
        return query.getResultList();
```

```
    }
```

```
}
```

- Iniciar o servidor e verificar se o datasource foi definido:

Bound data source [java:/jdbc/ead]

- Usando JSF, e baseado no exemplo da calculadora, criar a aplicação cliente, tanto para inserir como para listar livros.

Exercício 4

Este exercício trata de um componente **EJB MDB (*Message-Driven Beans*)**.

Message-Driven Bean é um componente que permite que aplicações Java EE processem mensagens assincronamente. Este tipo de *bean* normalmente age como um *listener* de mensagem JMS (Java Message Service), que é similar a um *listener* de eventos, como eventos em uma interface gráfica, mas recebe mensagens JMS ao invés de eventos propriamente ditos.

As mensagens podem ser enviadas por qualquer componente Java EE, como um *servlet*, outro componente EJB, aplicações JSF, ou até mesmo por aplicações não Java EE, como um cliente Java executando o método `main()`.

Diferenças entre Message-Driven Beans e Session beans:

A diferença de maior destaque entre **MDB** e **SB** é que clientes não acessam MDB através de interfaces. Diferente de SB, um MDB possui somente uma classe *bean*.

Em muitos aspectos, um MDB se assemelha a um Stateless Session bean.

As variáveis de instância de um MDB podem conter dados através da manipulação de mensagens de clientes, como uma conexão JMS, conexão com um banco de dados, ou uma referência para um objeto EJB.

Componentes cliente não localizam MDBs ou invocam seus métodos diretamente. Ao invés disso, um cliente acessa um MDB através de JMS enviando mensagens para o destino em que o MDB é um *MessageListener*. A configuração de um MDB ocorre durante o *deploy* no servidor.

Um MDB possui as seguintes características:

- Executam sobre uma simples mensagem do cliente.
- São invocados assincronamente.
- Possuem relativamente vida curta.
- Não representam dados em uma base de dados, mas acessam esta base.
- São *stateless*.

Quando usar MDBs:

Session beans permitem o envio e recebimento de mensagens sincronicamente. A fim de evitar bloqueio de recursos no servidor, use MDBs.

Nas etapas seguintes desenvolveremos um MDB.

- Criar um projeto EJB Project chamado **04_ProjetoMDB**.
- Neste projeto, incluir a classe **Cliente**, conforme modelo a seguir:

```
package br.com.ead.mdb.classes;

import java.io.Serializable;

public class Cliente implements Serializable {

    private static final long serialVersionUID = 1L;

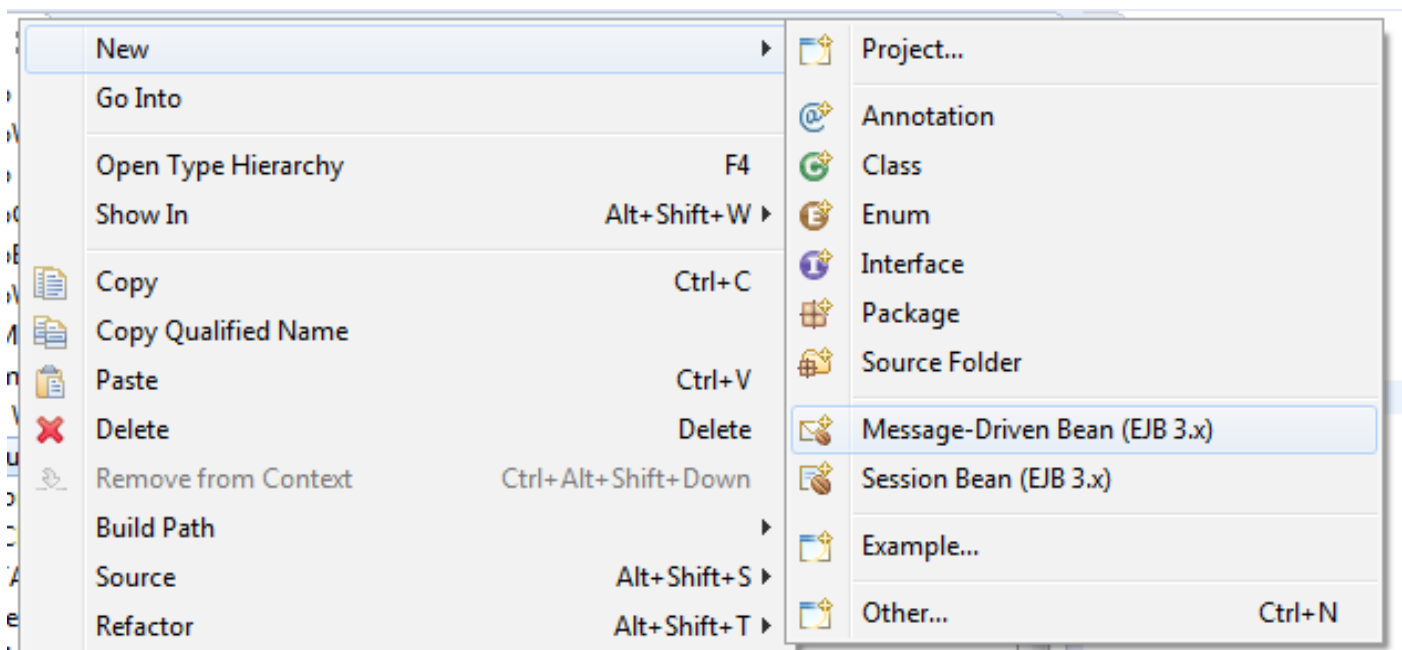
    private int id;
    private String nome, telefone, email;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getTelefone() {
        return telefone;
    }
    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

```

@Override
public String toString(){
    return "Cliente [id=" + this.getId() + ", nome=" +
        this.getNome() + ", telefone=" + this.getTelefone() +
        ", email=" + this.getEmail() + "]";
}
}

```

- Criar o consumidor Message-Driven Bean. Crie uma classe com as configurações apresentadas na interface a seguir:



- Acrescentar as alterações na classe resultante, conforme modelo:

```

package br.com.ead.mdb;

import java.util.Date;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;
import javax.jms.TextMessage;

```

```

import br.com.ead.mdb.classes.Cliente;

@MessageDriven(
    activationConfig = { @ActivationConfigProperty(
        propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
        @ActivationConfigProperty(
            propertyName= "destination", propertyValue = "queue/ExemploQueue")
    })

public class QueueListenerMDB implements MessageListener {

    public QueueListenerMDB() {
        // TODO Auto-generated constructor stub
    }

    public void onMessage(Message message) {
        try {
            if (message instanceof TextMessage) {
                System.out.println("Queue: TextMessage recebida em " + new Date());
                TextMessage msg = (TextMessage) message;
                System.out.println("Message is : " + msg.getText());
            } else if (message instanceof ObjectMessage) {
                System.out.println("Queue: ObjectMessage recebida em " + new Date());
                ObjectMessage msg = (ObjectMessage) message;
                Cliente cliente = (Cliente) msg.getObject();
                System.out.println("Detalhes do cliente: ");
                System.out.println(cliente.getId());
                System.out.println(cliente.getNome());
                System.out.println(cliente.getTelefone());
                System.out.println(cliente.getEmail());
            } else {
                System.out.println("Nenhuma mensagem válida!");
            }
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
}

```

- Localizar o arquivo **standalone.xml** na pasta **[JBoss]\standalone\configuration**. Dentro da tag `<extensions>` incluir o elemento:
`<extension module="org.jboss.as.messaging"/>`

- Dentro do elemento <profile>, adicionar o seguinte elemento <subsystem>:

```
<subsystem xmlns="urn:jboss:domain:messaging:1.1">
  <hornetq-server>
    <persistence-enabled>true</persistence-enabled>
    <journal-file-size>102400</journal-file-size>
    <journal-min-files>2</journal-min-files>

    <connectors>
      <netty-connector name="netty" socket-binding="messaging"/>
      <netty-connector name="netty-throughput" socket-binding="messaging-throughput">
        <param key="batch-delay" value="50"/>
      </netty-connector>
      <in-vm-connector name="in-vm" server-id="0"/>
    </connectors>

    <acceptors>
      <netty-acceptor name="netty" socket-binding="messaging"/>
      <netty-acceptor name="netty-throughput" socket-binding="messaging-throughput">
        <param key="batch-delay" value="50"/>
        <param key="direct-deliver" value="false"/>
      </netty-acceptor>
      <in-vm-acceptor name="in-vm" server-id="0"/>
    </acceptors>

    <security-settings>
      <security-setting match="#">
        <permission type="send" roles="guest"/>
        <permission type="consume" roles="guest"/>
        <permission type="createNonDurableQueue" roles="guest"/>
        <permission type="deleteNonDurableQueue" roles="guest"/>
      </security-setting>
    </security-settings>
```



```
<address-settings>
<address-setting match="#">
<dead-letter-address>jms.queue.DLQ</dead-letter-address>
<expiry-address>jms.queue.ExpiryQueue</expiry-address>
<redelivery-delay>0</redelivery-delay>
<max-size-bytes>10485760</max-size-bytes>
<address-full-policy>BLOCK</address-full-policy>
<message-counter-history-day-limit>10</message-counter-history-day-limit>
</address-setting>
</address-settings>

<jms-connection-factories>
<connection-factory name="InVmConnectionFactory">
<connectors>
<connector-ref connector-name="in-vm"/>
</connectors>
<entries>
<entry name="java:/ConnectionFactory"/>
</entries>
</connection-factory>
<connection-factory name="RemoteConnectionFactory">
<connectors>
<connector-ref connector-name="netty"/>
</connectors>
<entries>
<entry name="RemoteConnectionFactory"/>
<entry name="java:jboss/exported/jms/RemoteConnectionFactory"/>
</entries>
</connection-factory>
<pooled-connection-factory name="hornetq-ra">
<transaction mode="xa"/>
<connectors>
<connector-ref connector-name="in-vm"/>
</connectors>
<entries>
<entry name="java:/JmsXA"/>
```

```

</entries>
</pooled-connection-factory>
</jms-connection-factories>

<jms-destinations>
<jms-queue name="testQueue">
<entry name=" queue/ExemploQueue"/>
</jms-queue>
<jms-topic name="testTopic">
<entry name="topic/MyTopic"/>
</jms-topic>
</jms-destinations>
</hornetq-server>
</subsystem>

```

Observações:

- Hornetq é um projeto *opensource* que permite a construção de sistemas assíncronos multiplataforma. A partir da versão 7.0 do JBoss, um grupo de configurações no arquivo standalone.xml é suficiente. Em versões anteriores era necessário usar a API Hornetq, disponível em: <<http://hornetq.jboss.org/docs.html>>.
- Projetado pela Sun Microsystems com o apoio de empresas associadas, e lançado para o mercado em agosto de 1998, o JMS surgiu para permitir que os aplicativos escritos na linguagem Java pudessem criar, receber e enviar mensagens destinadas ou oriundas de outros aplicativos.

A principal característica desse tipo de processamento, classificado como fracamente acoplado, é que todas as operações que envolvem a troca de mensagens são feitas de forma assíncrona, fazendo com que as aplicações participantes não precisem ficar bloqueadas esperando o término de alguma computação remotamente solicitada, como ocorre naquelas aplicações que utilizam o Remote Procedure Call(RPC) ou mesmo o Remote Method Invocation(RMI)

- Dentro do elemento <socket-binding-group> adicionar os elementos:

```

<socket-binding name="messaging" port="5445"/>
<socket-binding name="messaging-throughput" port="5455"/>

```

- Localize o elemento <subsystem xmlns="urn:jboss:domain:ejb3:1.2">. Em seguida, adicione o elemento:

```
<mdb>
<resource-adapter-ref resource-adapter-name="hornetq-ra"/>
<bean-instance-pool-ref pool-name="mdb-strict-max-pool"/>
</mdb>
```

- Inicie o servidor. Após sua inicialização, verifique as mensagens:

```
Bound messaging object to jndi name java:/queue/ExemploQueue
Started message driven bean 'QueueListenerMDB' with 'hornetq-ra' resource adapter
```

- Para criar o cliente JMS, usaremos uma aplicação Web com um *servlet*. Definir um novo Dynamic Web Project com o nome **04_MDBCliente**. Usar o mesmo servidor de aplicações usado no MDB.
- Definir um arquivo **.jar** contendo a classe Cliente que você criou no primeiro projeto deste exercício.
- Adicionar o arquivo **.jar** criado na pasta **WEB-INF/lib** da aplicação Web.
- Definir, no projeto Web, um *servlet* chamado **ServletMdb**, com o mapeamento **/mdb**:
- No método doGet() escrever o código:

```
final String QUEUE_LOOKUP = "queue/ExemploQueue";
final String CONNECTION_FACTORY = "ConnectionFactory";

PrintWriter out = response.getWriter();
try{
Context context = new InitialContext();
QueueConnectionFactory factory =
(QueueConnectionFactory)context.lookup(CONNECTION_FACTORY);
QueueConnection connection = factory.createQueueConnection();
QueueSession session =
connection.createQueueSession(false,
QueueSession.AUTO_ACKNOWLEDGE);

Queue queue = (Queue)context.lookup(QUEUE_LOOKUP);
QueueSender sender = session.createSender(queue);

//1. Enviando objeto TextMessage
TextMessage message = session.createTextMessage();
message.setText("Exemplo EJB3 MDB Queue!!!");
sender.send(message);
out.println("1. Enviando mensagem tipo TextMessage");

//2. Enviando objeto ObjectMessage
ObjectMessage objMsg = session.createObjectMessage();
```

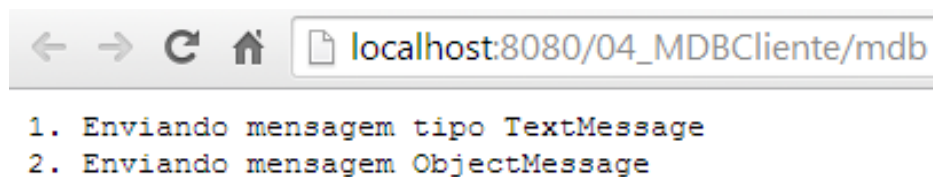
```

Cliente cliente = new Cliente();
cliente.setId(1500);
cliente.setNome("Ead Pós");
cliente.setTelefone("3385-8010");
cliente.setEmail("emilio.celso@ead.com.br");
objMsg.setObject(cliente);
sender.send(objMsg);
out.println("2. Enviando mensagem ObjectMessage");

session.close();
}
catch(Exception ex){
    ex.printStackTrace();
}

```

- Incluir o projeto Web ao servidor.
- Executar o servlet. O resultado no *browser* deverá ser semelhante a:



E no *log* do servidor, a informação:

21:33:05,558 INFO [stdout] (Thread-5 (HornetQ-client-global-threads-4967270)) Queue: TextMessage recebida em Sun Aug 24 21:33:05 BRT 2014

21:33:05,559 INFO [stdout] (Thread-5 (HornetQ-client-global-threads-4967270)) Message is : Exemplo EJB3 MDB Queue!!!

21:33:05,560 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) Queue: ObjectMessage recebida em Sun Aug 24 21:33:05 BRT 2014

21:33:05,563 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) Detalhes do cliente:

21:33:05,564 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) 1500

21:33:05,564 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) Ead Pós

21:33:05,565 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) 3385-8010

21:33:05,565 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) emilio.celso@ead.com.br

Considerações Finais

Caros alunos, chegamos ao final! Nesta etapa pudemos avançar no uso do Java aplicando-o a mecanismos de persistência de dados por meio do Hibernate.

Vimos, também, o quanto ele é poderoso, permitindo-nos usar simples métodos Java para acessarmos informações em bancos de dados, apenas mapeando as classes (chamadas de Entidades) para as tabelas nos bancos de dados, incluindo o relacionamento que possa ocorrer entre elas.

Neste momento posso garantir que você, prezado aluno, está preparado para desenvolver uma aplicação séria, profissional, contemplando tudo o que aprendemos até aqui.

Desejo-lhes um enorme sucesso e recomendo que jamais parem de estudar Java, pois sabemos que uma nova atualização é apresentada em curtos espaços de tempo. Novos recursos são inseridos na linguagem, e sei que vocês serão capazes de evoluir por conta própria daqui para frente.

Sucesso para vocês!

Prof. Emilio

Respostas Comentadas dos Exercícios

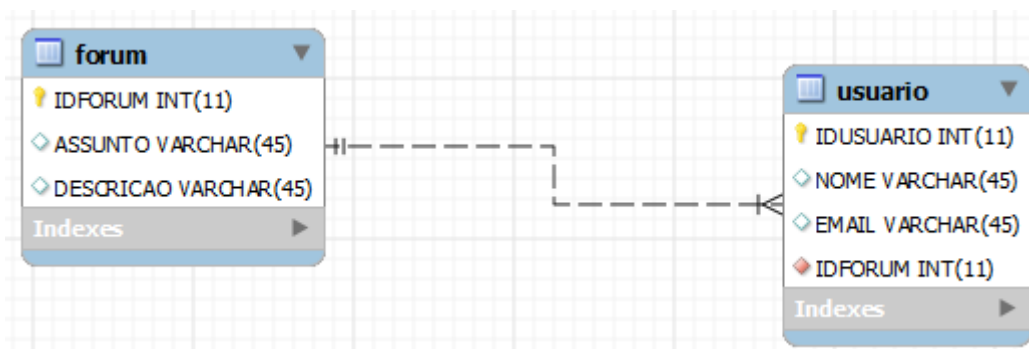
Exercícios do Módulo 1

Exercício 1

Caro aluno, o primeiro exercício apresenta uma solução completa e deve ser seguida na elaboração dos exercícios posteriores. Sugiro que você implemente o primeiro completamente, antes de passar ao próximo.

Neste exercício desenvolveremos uma aplicação baseada em Hibernate. Os passos são apresentados a seguir:

- Criar um projeto Java Project chamado **ExemploHibernate**.
- Importar a API do **Hibernate**.
- Criar o banco de dados chamado **forum** cujo modelo é dado abaixo:



- Com base neste modelo, criar as classes POJO **Forum** e **Usuario**:

```
package br.com.ead.entity;
```

```
import java.io.Serializable;
```

```
import java.util.HashSet;
```

```
import java.util.Set;
```

```
public class Forum implements Serializable{
```

```
    private static final long serialVersionUID = 1L;
```

```
    private int idforum;
```

```
    private String assunto;
```

```
    private String descricao;
```

```
    private Set<Usuario> usuarios = new HashSet<Usuario>(0);
```

```
    public int getIdforum() {
```

```
        return idforum;
    }
    public void setIdforum(int idforum) {
        this.idforum = idforum;
    }
    public String getAssunto() {
        return assunto;
    }

    public void setAssunto(String assunto) {
        this.assunto = assunto;
    }
    public String getDescricao() {
        return descricao;
    }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
    public Set<Usuario> getUsuarios() {
        return usuarios;
    }
    public void setUsuarios(Set<Usuario> usuarios) {
        this.usuarios = usuarios;
    }
}
```

```
package br.com.ead.entity;
import java.io.Serializable;
```

```
public class Usuario implements Serializable{

    private static final long serialVersionUID = 1L;
    private int idusuario;
    private Forum forum;
    private String nome;
    private String email;

    public int getIdusuario() {
        return idusuario;
    }
    public void setIdusuario(int idusuario) {
        this.idusuario = idusuario;
    }
}
```

```

    }
    public Forum getForum() {
        return forum;
    }
    public void setForum(Forum forum) {
        this.forum = forum;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}

```

- Definir os arquivos de mapeamento, no mesmo pacote que as classes POJO:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

```

```

<hibernate-mapping>
  <class name="br.com.ead.entity.Forum" table="forum" catalog="forum">
    <id name="idforum" type="int">
      <column name="IDFORUM" />
      <generator class="assigned" />
    </id>
    <property name="assunto" type="string">
      <column name="ASSUNTO" length="45" />
    </property>
    <property name="descricao" type="string">
      <column name="DESCRICAO" length="45" />
    </property>
    <set name="usuarios" inverse="false" cascade="save-update">
      <key>
        <column name="IDFORUM" not-null="true" />
      </key>
      <one-to-many class="br.com.ead.entity.Usuario" />
    </set>
  </class>
</hibernate-mapping>

```



```

    </set>
  </class>
</hibernate-mapping>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="br.com.ead.entity.Usuario" table="usuario" catalog="forum">
    <id name="idusuario" type="int">
      <column name="IDUSUARIO" />
      <generator class="assigned" />
    </id>
    <many-to-one name="forum" class="br.com.ead.entity.Forum" fetch="select">
      <column name="IDFORUM" not-null="true" />
    </many-to-one>
    <property name="nome" type="string">
      <column name="NOME" length="45" />
    </property>
    <property name="email" type="string">
      <column name="EMAIL" length="45" />
    </property>
  </class>
</hibernate-mapping>

```

- Definir o arquivo de configurações **hibernate.cfg.xml**, na raiz dos pacotes das classes:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/forum</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">password</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <mapping resource="br/com/ead/entity/Forum.hbm.xml"/>
    <mapping resource="br/com/ead/entity/Usuario.hbm.xml"/>
  </session-factory>
</hibernate-configuration>

```

```

    </session-factory>
</hibernate-configuration>

```

- Definir a classe **HibernateUtil**, responsável por realizar a leitura dos arquivos de configurações:

```
package br.com.ead.config;
```

```
import org.hibernate.SessionFactory;
```

```
import org.hibernate.cfg.Configuration;
```

```
public class HibernateUtil {
```

```
    private static final SessionFactory sessionFactory;
```

```
    static {
```

```
        try {
```

```
            sessionFactory = new Configuration().configure().buildSessionFactory();
```

```
        } catch (Exception e) {
```

```
            throw new ExceptionInInitializerError(e);
```

```
        }
```

```
    }
```

```
    public static SessionFactory getSessionFactory() {
```

```
        return sessionFactory;
```

```
    }
```

```
}
```

- Escrever a classe **ForumHelper**, contendo métodos auxiliares para realizar a persistência:

```
package br.com.ead.helper;
```

```
import java.util.HashSet;
```

```
import java.util.Set;
```

```
import org.hibernate.Session;
```

```
import org.hibernate.Transaction;
```

```
import br.com.ead.config.HibernateUtil;
```

```
import br.com.ead.entity.Forum;
```

```
import br.com.ead.entity.Usuario;
```

```
public class ForumHelper {
```

```
    Session session = null;
```

```
    Transaction transaction = null;
```

```
    public String salvar(Forum forum){
```

```
try{
    session = HibernateUtil.getSessionFactory().getCurrentSession();
    transaction = session.beginTransaction();
    session.save(forum);
    transaction.commit();
    return "Forum salvo";
}catch(Exception e){
    return e.getMessage();
}
}
```

```
public String salvar(Usuario forum){
    try{
        session = HibernateUtil.getSessionFactory().getCurrentSession();
        transaction = session.beginTransaction();
        session.save(forum);
        transaction.commit();
        return "Usuario salvo";
    }catch(Exception e){
        return e.getMessage();
    }
}
```

```
public String adicionarUsuario(int idForum, int idUsuario){
    try {
        session = HibernateUtil.getSessionFactory().getCurrentSession();
        transaction = session.beginTransaction();
        Forum f = (Forum)session.load(Forum.class, idForum);
        Usuario u = (Usuario)session.load(Usuario.class, idForum);

        f.getUsuarios().add(u);
        session.save(f);
        transaction.commit();
        return "Associação realizada";
    } catch(Exception e){
        return e.getMessage();
    }
}
```

```
public String adicionarUsuario(int idForum, Usuario usuario){
    try {
        session = HibernateUtil.getSessionFactory().getCurrentSession();
        transaction = session.beginTransaction();
```

```

        Forum f = (Forum)session.load(Forum.class, idForum);

        usuario.setForum(f);

        f.getUsuarios().add(usuario);
        session.update(f);
        transaction.commit();
        return "Inclusao realizada";
    } catch (Exception e){
        return e.getMessage();
    }
}

public Set<Usuario> listarUsuarios(int idForum){
    Set<Usuario> usuarios = new HashSet<Usuario>();
    try {
        session = HibernateUtil.getSessionFactory().getCurrentSession();
        transaction = session.beginTransaction();
        Forum f = (Forum)session.load(Forum.class, idForum);
        usuarios = f.getUsuarios();

        } catch (Exception e) {

        }

    return usuarios;
}
}

```

- Para testar a aplicação, criar um fórum e três usuários, associando cada usuário ao fórum criado:

```

package br.com.ead.programa;

import java.util.Set;
import br.com.ead.entity.Forum;
import br.com.ead.entity.Usuario;
import br.com.ead.helper.ForumHelper;

public class TesteForum {
    public static void main(String[] args) {
        incluirForum();
        incluirUsuarionoForum();
        listarUsuariosPorForum();
    }
}

```

```

private static void incluirForum(){
    Forum forum = new Forum();
    forum.setIdforum(10);
    forum.setAssunto("Avaliação");
    forum.setDescricao("Avaliação da disciplina Persistência");

    ForumHelper helper = new ForumHelper();
    System.out.println(helper.salvar(forum));
}

private static void incluirUsuarionoForum(){
    ForumHelper helper = new ForumHelper();
    Usuario u1 = new Usuario();
    u1.setNome("teresa");
    u1.setEmail("teresa@mail.com");
    u1.setIdusuario(1);

    Usuario u2 = new Usuario();
    u2.setNome("jonas");
    u2.setEmail("joas@mail.com");
    u2.setIdusuario(2);

    Usuario u3 = new Usuario();
    u3.setNome("abilio");
    u3.setEmail("abilio@mail.com");
    u3.setIdusuario(3);

    System.out.println(helper.adicionarUsuario(10, u1));
    System.out.println(helper.adicionarUsuario(10, u2));
    System.out.println(helper.adicionarUsuario(10, u3));
}

private static void listarUsuariosPorForum(){
    ForumHelper helper = new ForumHelper();
    Set<Usuario> usuarios = helper.listarUsuarios(10);
for(Usuario usuario: usuarios){
    System.out.println("ID Usuario: " + usuario.getIdusuario());
    System.out.println("Nome Usuario: " + usuario.getNome());
    System.out.println("Email Usuario: " + usuario.getEmail());
    System.out.println("-----");
}
}
}

```

Exercício 2

Orientações:

1. Leia atentamente e siga os passos descritos neste documento para a elaboração desta atividade.
2. Criar os arquivos que forem necessários. Tomar como base o Exercício 1 deste módulo.

Cenário para elaboração do banco de dados:

Em uma aplicação bancária, um cliente correntista pode ter uma ou mais contas. Cada conta pertence a apenas um cliente.

A sugestão para elaboração das tabelas é a seguinte:

CLIENTE	CONTA
CPF (texto) - PK	AGENCIA (int)
NOME (texto)	CONTACORRENTE (texto)
DATANASC (data)	CPF (texto) - FK
	SALDO (double)

Passos para o desenvolvimento a atividade.

1. Criar este banco de dados, considerando como chave de relacionamento o CPF do cliente.
O banco deverá ser criado no MySQL, seguindo os passos descritos nas disciplinas JavaSE – Programação Avançada e Java Web.
2. O banco de dados deverá se chamar **banco**.
Este deverá ser o nome do banco de dados, na ocasião da inclusão de um novo schema no MySQL.
3. Criar um projeto “**Java Project**” chamado **Modulo1_Exercicio2**.
4. Definir as entidades para **Cliente** e **Conta**.

Entidade Cliente:

```
package br.com.entity;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;
```

```
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

public class Cliente implements Serializable {

    private static final long serialVersionUID = 1L;

    private String cpf;

    private String nome;

    private String datanasc;

    private Set<Conta> contas = new HashSet<Conta>(0);

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getDatanasc() {
        return datanasc;
    }

    public void setDatanasc(String datanasc) {
        this.datanasc = datanasc;
    }
}
```

```
    public Set<Contas> getContas() {  
        return contas;  
    }  
  
    public void setContas(Set<Contas> contas) {  
        this.contas = contas;  
    }  
  
}
```

Entidade Conta:

```
package br.com.entity;  
  
import java.io.Serializable;  
  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.FetchType;  
import javax.persistence.Id;  
import javax.persistence.JoinColumn;  
import javax.persistence.ManyToOne;  
import javax.persistence.Table;  
  
public class Conta implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
  
    private int agencia;  
  
    private Clientes cpf;  
  
    private String contacorrente;  
  
    private double saldo;  
  
    public int getAgencia() {  
        return agencia;  
    }  
  
    public void setAgencia(int agencia) {  
        this.agencia = agencia;  
    }  
  
}
```



```

    public Clientes getCpf() {
        return cpf;
    }

    public void setCpf(Clientes cpf) {
        this.cpf = cpf;
    }

    public String getContacorrente() {
        return contacorrente;
    }

    public void setContacorrente(String contacorrente) {
        this.contacorrente = contacorrente;
    }

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
}

```

5. Escrever todas as classes necessárias para elaborar a persistência. Considerar as operações: **INCLUIR**, **ALTERAR**, **LISTAR** e **REMOVER** (CRUD).

Será apresentado um exemplo de utilização considerando classes genéricas:

- Definir os arquivos de mapeamento, no mesmo pacote que as classes POJO:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="br.com.entity.Cliente" table="CLIENTE" catalog="banco">
        <id name="id" type="int">
            <column name="ID" />
            <generator class="assigned" />
        </id>
        <property name="nome" type="string">

```

```

        <column name="NOME" length="45" />
    </property>
    <property name="datanasc" type="date">
        <column name="DATANASC" />
    </property>
    <set name="contas" inverse="false" cascade="save-update">
        <key>
            <column name="ID" not-null="true" />
        </key>
        <one-to-many class="br.com.entity.Contas" />
    </set>
</class>
</hibernate-mapping>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="br.com.entity.Conta" table="CONTA" catalog="banco">
        <id name="agencia" type="int">
            <column name="AGENCIA" />
            <generator class="assigned" />
        </id>
        <many-to-one name="cliente" class="br.com.entity.Forum" fetch="select">
            <column name="ID" not-null="true" />
        </many-to-one>
        <property name="cpf" type="string">
            <column name="CPF" length="11" />
        </property>
        <property name="contacorrente" type="string">
            <column name="CONTACORRENTE" length="10" />
        </property>
        <property name="saldo" type="double">
            <column name="SALDO" />
        </property>

    </class>
</hibernate-mapping>

```

- Definir o arquivo de configurações **hibernate.cfg.xml**, na raiz dos pacotes das classes:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC

```

```

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/banco</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">password</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <mapping resource="br/com/entity/Cliente.hbm.xml"/>
    <mapping resource="br/com/entity/Conta.hbm.xml"/>
  </session-factory>
</hibernate-configuration>

```

- Definir a classe **HibernateUtil**, responsável por realizar a leitura dos arquivos de configurações:

```
package br.com.config;
```

```
import org.hibernate.SessionFactory;
```

```
import org.hibernate.cfg.Configuration;
```

```
public class HibernateUtil {
```

```
    private static final SessionFactory sessionFactory;
```

```
    static {
```

```
        try {
```

```
            sessionFactory = new Configuration().configure().buildSessionFactory();
```

```
        } catch (Exception e) {
```

```
            throw new ExceptionInInitializerError(e);
```

```
        }
```

```
    }
```

```
    public static SessionFactory getSessionFactory() {
```

```
        return sessionFactory;
```

```
    }
```

```
}
```

- Escrever a classe **ClienteHelper**, contendo métodos auxiliares para realizar a persistência:

```
package br.com.ead.helper;
```

```
import java.util.HashSet;
import java.util.Set;
import org.hibernate.Session;
import org.hibernate.Transaction;
import br.com.ead.config.HibernateUtil;
import br.com.ead.entity.Forum;
import br.com.ead.entity.Usuario;

public class ClienteHelper {
    Session session = null;
    Transaction transaction = null;

    public String salvar(Cliente cliente){
        try{
            session = HibernateUtil.getSessionFactory().getCurrentSession();
            transaction = session.beginTransaction();
            session.save(cliente);
            transaction.commit();
            return "Cliente salvo";
        }catch(Exception e){
            return e.getMessage();
        }
    }

    public String salvar(Conta conta){
        try{
            session = HibernateUtil.getSessionFactory().getCurrentSession();
            transaction = session.beginTransaction();
            session.save(conta);
            transaction.commit();
            return "Conta salva";
        }catch(Exception e){
            return e.getMessage();
        }
    }

    public String adicionarUsuario(int idCliente, int agencia){
        try {
            session = HibernateUtil.getSessionFactory().getCurrentSession();
            transaction = session.beginTransaction();
            Cliente f = (Cliente)session.load(Conta.class, idCliente);
            Conta u = (Conta)session.load(Conta.class, agencia);

            f.getContas().add(u);
```

```

        session.save(f);
        transaction.commit();
        return "Associação realizada";
    } catch (Exception e) {
        return e.getMessage();
    }
}

public Set<Conta> listarContas(int idCliente){
    Set<Conta> contas = new HashSet<Conta>();
    try {
        session = HibernateUtil.getSessionFactory().getCurrentSession();
        transaction = session.beginTransaction();
        Cliente f = (Cliente)session.load(Cliente.class, idCliente);
        contas = f.getContas();

        } catch (Exception e) {

        }
    return contas;
}
}

```

6. Escrever um programa consistente para testar todos os itens que você definiu.

```

package br.com.programa;

import java.util.Set;
import br.com.entity.Cliente;
import br.com.entity.Conta;
import br.com.helper.ClienteHelper;

public class Teste Cliente {
    public static void main(String[] args) {
        incluirCliente ();
        incluirContanoClientem();
        listarContassPorCliente ();
    }

    private static void incluirCliente (){
        Cliente cliente = new Cliente ();
        cliente.setCpf("12345678900");
        forum.setNome("Jose");
        forum.setDadanascimento(new Date());
    }
}

```

```
        ClienteHelper helper = new ClienteHelper();
        System.out.println(helper.salvar(cliente));
    }

    private static void incluirContanoCliente(){
        ClienteHelper helper = new ClienteHelper();
        Conta u1 = new Conta();
        u1.setAgencia(1234);
        u1.setContacorrente("12345-6");
        u1.setCpf("12345678900");

        Conta u2 = new Conta();
        u2.setAgencia(1235);
        u2.setContacorrente("12355-6");
        u2.setCpf("12345678900");

        System.out.println(helper.adicionarCliente(10, u1));
        System.out.println(helper.adicionarCliente(10, u2));
    }
}
```

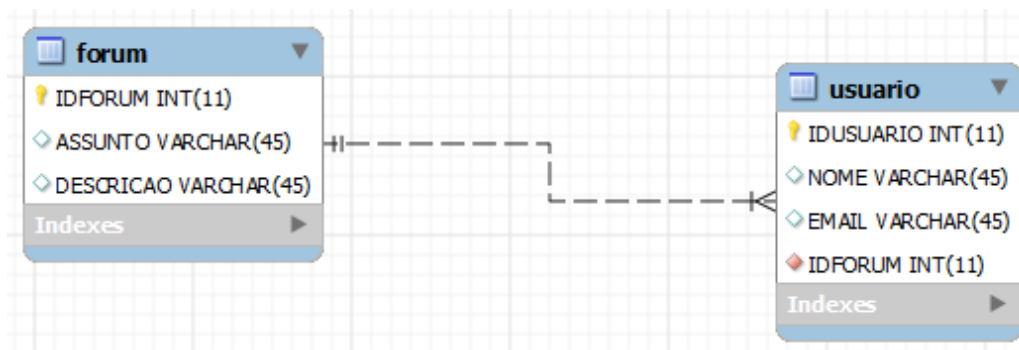
Exercícios do Módulo 2

Caro aluno, o primeiro exercício deste módulo apresenta uma solução completa e deve ser seguida na elaboração dos exercícios posteriores. Sugiro que este seja implementado completamente, antes de você passar ao próximo.

Exercício 1

Neste roteiro desenvolveremos uma aplicação baseada em anotações JPA. Os passos são apresentados a seguir:

- Criar um projeto Java Project chamado **ExemploJPA**.
- Importar a API do **Hibernate**.
- Importar o banco de dados **forum** do Exercício 1 do Módulo 1:



- Copiar este banco de dados para outro chamado **forum01**, e tornar as chaves **IDFORUM** e **IDUSUARIO** como auto-incremento.
- Com base neste modelo, criar as entidades **Forum** e **Usuario** (as mesmas do módulo anterior):

```
package br.com.ead.entity;
```

```
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
```

```
@Entity
@Table(name="FORUM", schema = "forum01")
```

```
public class Forum implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "IDFORUM")
    private int id;

    @Column(name = "ASSUNTO", length = 45)
    private String assunto;

    @Column(name = "DESCRICAO", length = 45)
    private String descricao;
```

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "forum")
```

```

private Set<Usuario> usuarios = new HashSet<Usuario>();

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getAssunto() {
    return assunto;
}
public void setAssunto(String assunto) {
    this.assunto = assunto;
}
public String getDescricao() {
    return descricao;
}
public void setDescricao(String descricao) {
    this.descricao = descricao;
}
public Set<Usuario> getUsuarios() {
    return usuarios;
}
public void setUsuarios(Set<Usuario> usuarios) {
    this.usuarios = usuarios;
}
}

```

```

package br.com.ead.entity;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

```

```

@Entity
@Table(name = "USUARIO", schema = "forum01")
public class Usuario implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)

```



```

@Column(name = "IDUSUARIO")
private int id;
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "IDFORUM")
private Forum forum;

@Column(name = "NOME")
private String nome;

@Column(name = "EMAIL")
private String email;

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public Forum getForum() {
    return forum;
}
public void setForum(Forum forum) {
    this.forum = forum;
}
public String getNome() {
    return nome;
}
public void setNome(String nome) {
    this.nome = nome;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
}

```

- Definir, na pasta src/META-INF, o arquivo **persistence.xml**:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

<persistence-unit name="Forum">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>br.com.ead.entity.Forum</class>
    <class>br.com.ead.entity.Usuario</class>
    <properties>
        <property name="hibernate.hbm2ddl.auto" value="update" />
        <property name="hibernate.format_sql" value="true" />
        <property
            name="hibernate.dialect"
            value="org.hibernate.dialect.MySQLDialect" />
        <property
            name="hibernate.connection.driver_class"
            value="com.mysql.jdbc.Driver" />
        <property
            name="hibernate.connection.url"
            value="jdbc:mysql://localhost/forum01" />
        <property name="hibernate.connection.username" value="root" />
        <property name="hibernate.connection.password" value="password" />
    </properties>
</persistence-unit>
</persistence>

```

- Escrever a classe **ForumHelper**, contendo métodos auxiliares para realizar a persistência:

```

package br.com.ead.helper;

import javax.persistence.EntityManager;
import br.com.ead.entity.Forum;
import br.com.ead.entity.Usuario;

public class ForumHelper {
    private EntityManager em;

    public ForumHelper(EntityManager em){
        this.em = em;
    }

    public String salvar(Forum forum){
        try{
            em.getTransaction().begin();
            em.persist(forum);

```

```

        em.getTransaction().commit();
        return "Forum salvo";
    } catch (Exception e) {
        return e.getMessage();
    }
}

public String adicionarUsuario(int idForum, Usuario usuario){
    try {
        Forum f = em.find(Forum.class, idForum);
        usuario.setForum(f);
        f.getUsuarios().add(usuario);
        em.getTransaction().begin();
        em.persist(f);
        em.getTransaction().commit();
        return "Inclusao realizada";
    } catch (Exception e) {
        return e.getMessage();
    }
}
}

```

- Para testar a aplicação, criar um fórum e três usuários, associando cada usuário ao fórum criado:

```

package br.com.ead.programa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import br.com.ead.entity.Forum;
import br.com.ead.entity.Usuario;
import br.com.ead.helper.ForumHelper;

public class TesteForum {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("Forum");
        EntityManager em = emf.createEntityManager();
        ForumHelper dao = new ForumHelper(em);

        Forum forum = new Forum();
        forum.setAssunto("JPA");
        forum.setDescricao("Java Persistence API");
    }
}

```

```

        System.out.println(dao.salvar(forum));

        Usuario usuario = new Usuario();
        usuario.setNome("Joaquim");
        usuario.setEmail("joaquim@ead.com.br");

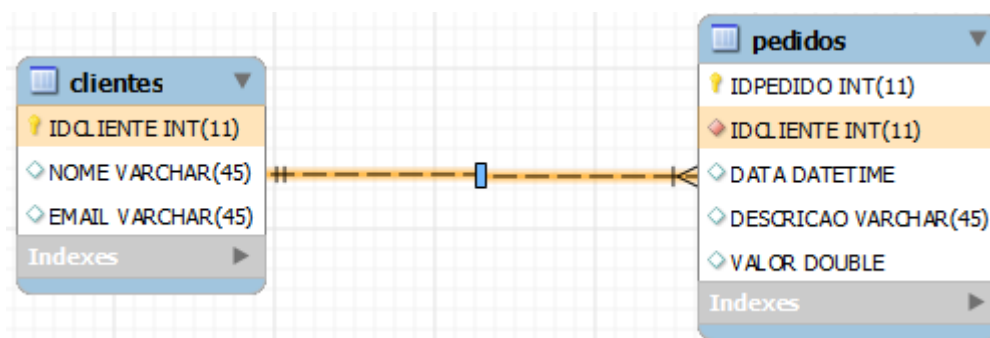
        System.out.println(dao.adicionarUsuario(forum.getId(), usuario));

    }
}

```

Exercício 2

Com base no Exercício1, criar uma aplicação para a persistência de clientes e pedidos, usando anotações JPA. O modelo é dado a seguir. É necessário criar a aplicação completa!



- Com base neste modelo, criar as entidades **Clientes e Pedidos**:

Entidade Clientes:

```
package br.com.entity;
```

```
import java.io.Serializable;
```

```
import java.util.HashSet;
```

```
import java.util.Set;
```

```
import javax.persistence.CascadeType;
```

```
import javax.persistence.Column;
```

```
import javax.persistence.Entity;
```

```
import javax.persistence.FetchType;
```

```
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
```

```
import javax.persistence.Id;
```

```
import javax.persistence.OneToMany;
```

```
import javax.persistence.Table;
```

```
@Entity
@Table(name="clientes")
public class Clientes implements Serializable{

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "IDCLIENTE")
    private int id;

    @Column(name = "NOME")
    private String nome;

    @Column(name = "EMAIL")
    private String email;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "cliente")
    private Set<Pedidos> pedidos = new HashSet<Pedidos>();

    public Set<Pedidos> getPedidos() {
        return pedidos;
    }

    public void setPedidos(Set<Pedidos> pedidos) {
        this.pedidos = pedidos;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

 }

Entidade Pedidos

```

package br.com.ead.entity;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

```

```

package br.com.entity;

```

```

import java.io.Serializable;
import java.util.Date;

```

```

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

```

```

@Entity
@Table(name="clientes")
public class Pedidos implements Serializable{

```

```

    @Id

```

```

    @GeneratedValue(strategy = GenerationType.AUTO)

```

```

    @Column(name = "IDUSUARIO")

```

```

    private int id;

```

```

    @Column(name = "DATA")

```

```

    private Date data;

```

```

    @Column(name = "DESCRICAO")

```

```

    private String descricao;

```

```
@Column(name = "VALOR")
private double valor;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "IDCLIENTE")
private Clientes cliente;

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public Date getData() {
    return data;
}

public void setData(Date data) {
    this.data = data;
}

public String getDescricao() {
    return descricao;
}

public void setDescricao(String descricao) {
    this.descricao = descricao;
}

public double getValor() {
    return valor;
}

public void setValor(double valor) {
    this.valor = valor;
}

public Clientes getCliente() {
    return cliente;
}
```

```

    public void setCliente(Clientes cliente) {
        this.cliente = cliente;
    }
}

```

- Definir, na pasta src/META-INF, o arquivo **persistence.xml**:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="clientePU">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>br.com.entity.Clientes</class>
    <class>br.com.entity.Pedidos</class>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.format_sql" value="true" />
      <property
        name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLDialect" />
      <property
        name="hibernate.connection.driver_class"
        value="com.mysql.jdbc.Driver" />
      <property
        name="hibernate.connection.url"
        value="jdbc:mysql://localhost/forum01" />
      <property name="hibernate.connection.username" value="root" />
      <property name="hibernate.connection.password" value="password" />
    </properties>
  </persistence-unit>
</persistence>

```

- Escrever a classe **Helper**, contendo métodos auxiliares para realizar a persistência:

```

package br.com.helper;

import javax.persistence.EntityManager;
import br.com.entity.Forum;
import br.com.entity.Usuario;

```



```

public class Helper {
    private EntityManager em;

    public Helper(EntityManager em){
        this.em = em;
    }

    public String salvar(Clientes cliente){
        try{
            em.getTransaction().begin();
            em.persist(cliente);
            em.getTransaction().commit();
            return "Cliente salvo";
        }catch(Exception e){
            return e.getMessage();
        }
    }

    public String adicionarPedido(int idCliente, Pedidos pedido){
        try {
            Cliente f = em.find(Cliente.class, idCliente);
            pedido.setCliente(f);
            f.getPedidos().add(pedido);

            em.getTransaction().begin();
            em.persist(f);
            em.getTransaction().commit();
            return "Inclusao realizada";
        } catch(Exception e){
            return e.getMessage();
        }
    }
}

```

- Para testar a aplicação, criar um fórum e três usuários, associando cada usuário ao fórum criado:

```

package br.com.ead.programa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import br.com.entity.Forum;

```

```
import br.com.entity.Usuario;
import br.com.helper.ForumHelper;

public class TesteCliente {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("clientePU");
        EntityManager em = emf.createEntityManager();
        Helper dao = new Helper(em);

        Cliente cliente = new Cliente();
        cliente.setNome("Estacio");
        cliente.setEmail("pos@estacio.com");

        System.out.println(dao.salvar(cliente));

        Pedido pedido = new Pedido();
        pedido.setData(new Date());
        pedido.setDescricao("Apostila");
        pedido.setValor(120);

        System.out.println(dao.adicionarCliente(cliente.getId(), pedido));
    }
}
```

Exercícios do Módulo 3

Caro aluno, as aplicações deste módulo são longas e possuem uma complexidade maior. Por isso, elas estão resolvidas mesmo no material da apostila. Siga cada um dos passos para implementar as aplicações.

Exercício 1: Componentes de servidor e cliente no mesmo projeto.

- No Eclipse, criar um projeto do tipo **EJB Project**. Nomeá-lo como **01_FolhaPagamento**. Incluir o servidor **JBoss AS 7.1** e manter as configurações mostradas abaixo:

New EJB Project

Create an EJB Project and add it to a new or existing Enterprise Application.

EJB Project

Project name: 01_FolhaPagamento

Project location

☒ Use default location

Location: C:\Documentos\projetos\Java\EBJ_Aulas\01_FolhaPagamer Browse...

Target runtime

JBoss 7.1 Runtime New Runtime...

EJB module version

3.0

Configuration

<custom> Modify...

Hint: Get started quickly by selecting one of the pre-defined project configurations.

EAR membership

☒ Add project to an EAR

EAR project name: 01_FolhaPagamentoEAR New Project...

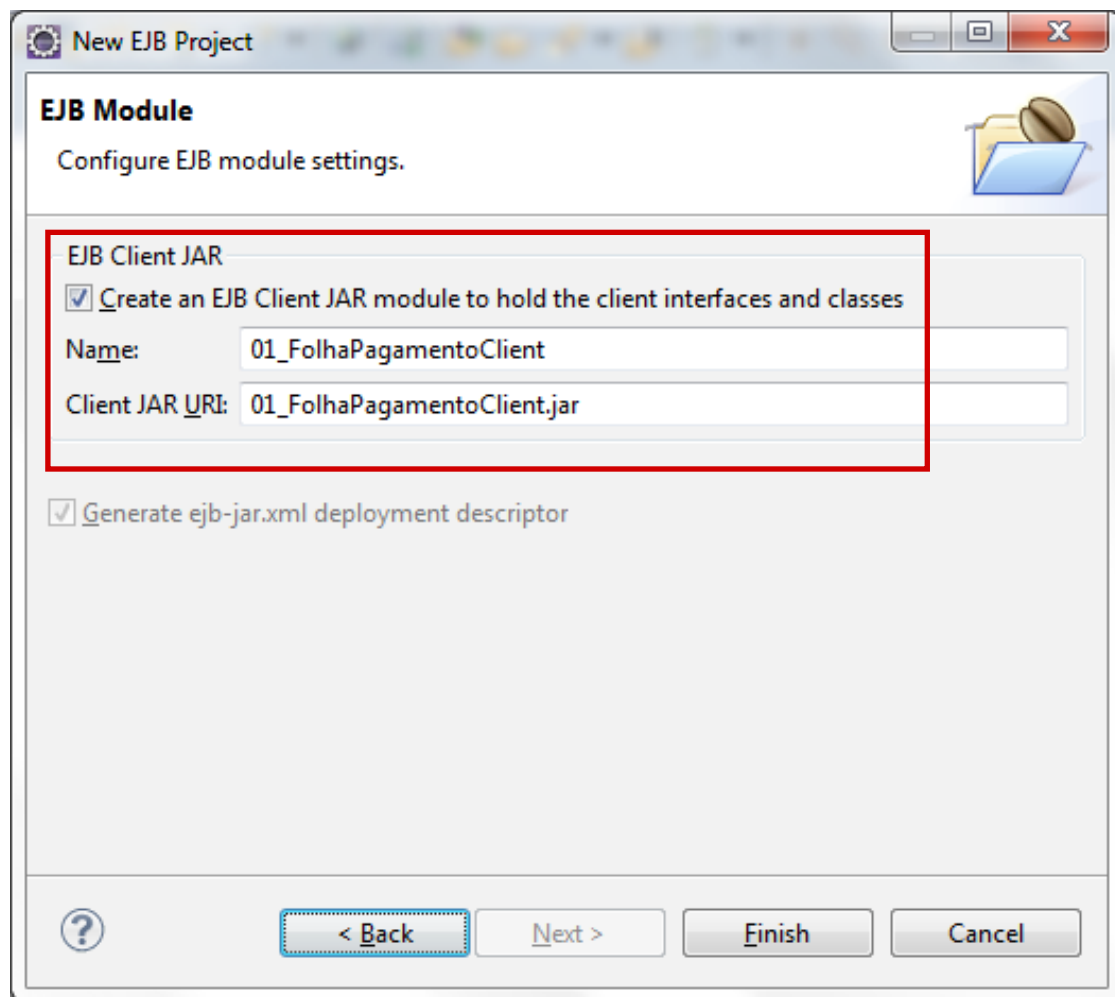
Working sets

☐ Add project to working sets

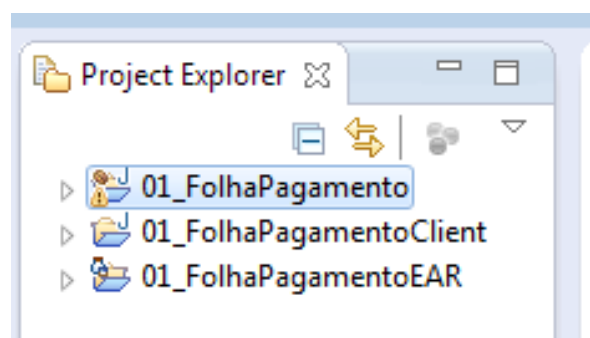
Working sets: Select...

? < Back Next > Finish Cancel

- Avance duas vezes e mantenha a seleção a seguir:



- Ao finalizar, devemos ter os três projetos listados abaixo:



Os projetos possuirão os seguintes elementos:

- **01_FolhaPagamento** – Classes que implementam os Stateless Session Beans.
- **01_FolhaPagamentoClient** – Interfaces que definem as operações dos Stateless Session Beans.
- **01_FolhaPagamentoEAR** – Responsável por empacotar todos os módulos da aplicação.

- Criar um novo projeto WEB (Dynamic Web Project) para consumir o EJB. Nomeie-o como **01_FolhaPagamentoWeb**, mantendo as configurações mostradas a seguir:

New Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: 01_FolhaPagamentoWeb

Project location

☒ Use default location

Location: C:\Documentos\projetos\Java\EBJ_Aulas\01_FolhaPagamer **Browse...**

Target runtime

JBoss 7.1 Runtime **New Runtime...**

Dynamic web module version

3.0

Configuration

Default Configuration for JBoss 7.1 Runtime **Modify...**

A good starting point for working with JBoss 7.1 Runtime runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership

☒ Add project to an EAR

EAR project name: 01_FolhaPagamentoEAR **New Project...**

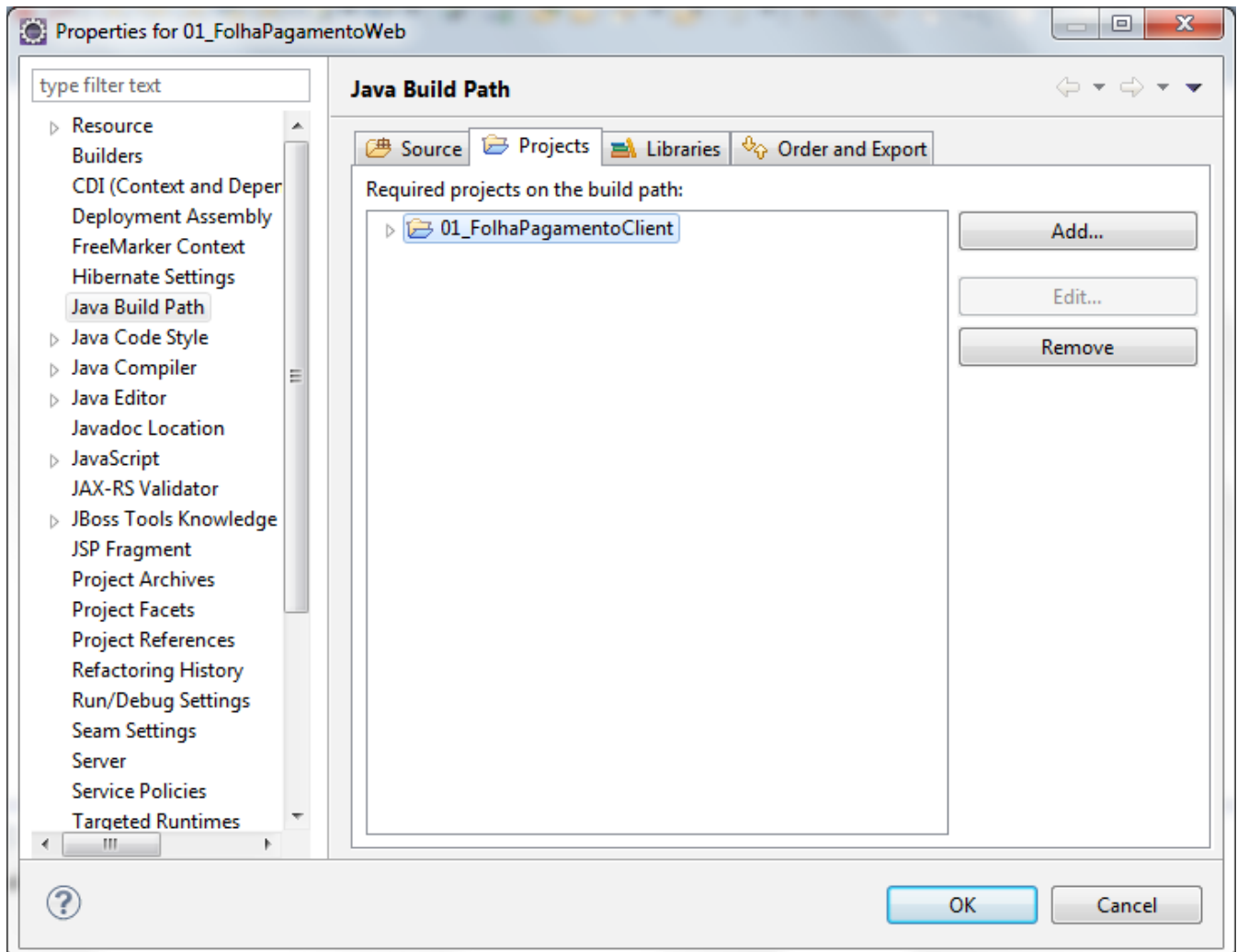
Working sets

☐ Add project to working sets

Working sets: **Select...**

Navigation: ? < Back Next > Finish Cancel

- Adicionar o projeto **01_FolhaPagamentoClient** como dependência do novo projeto web (Configuração do Build Path):



- No projeto **01_FolhaPagamentoClient** criar uma interface chamada **FolhaPagamento**, cujo modelo é ilustrado a seguir:

```
package br.com.ead.interfaces;
```

```
public interface FolhaPagamento {  
    void setSalario(double salario);  
    double calcularINSS(double taxa);  
    double calcularSalarioLiquido();  
}
```

- No projeto **01_FolhaPagamento** criar a classe **FolhaPagamentoBean**, conforme ilustrado a seguir:

```
package br.com.ead.bean;
```

```

import javax.ejb.Local;
import javax.ejb.Stateless;
import br.com.ead.interfaces.FolhaPagamento;

@Stateless
@Local(FolhaPagamento.class)
public class FolhaPagamentoBean implements FolhaPagamento{

    private double salario;

    @Override
    public double calcularINSS(double taxa) {
        return salario * taxa / 100;
    }

    @Override
    public double calcularSalarioLiquido() {
        return salario - calcularINSS(10);
    }
}

```

- No projeto **01_FolhaPagamentoWeb**, criar um Servlet e uma página JSP, de acordo com os modelos dados a seguir:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Salarios</title>
</head>
<body>
    <form action="folha" method="post">
        <table>
            <tr>
                <td>Informe o valor do salário:</td>
                <td><input type="text" name="salario" size="10" /></td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit" value="Enviar" /></td>
            </tr>
        </table>
    </form>

```

```

    </form>
</body>
</html>

```

```

package br.com.ead.web;

```

```

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;

```

```

import br.com.ead.interfaces.FolhaPagamento;

```

```

@WebServlet("/folha")

```

```

public class ServletFolha extends HttpServlet {
    private static final long serialVersionUID = 1L;

```

```

    @EJB
    FolhaPagamento fp;
    public ServletFolha() {
        super();
        // TODO Auto-generated constructor stub
    }

```

```

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // TODO Auto-generated method stub
    }

```

```

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

```

```

        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        try {
            int salario = Integer.parseInt(request.getParameter("salario"));
            fp.setSalario(salario);
            out.print("Salario Bruto: " + salario);
            out.print("<br/>Salario Liquido: " + fp.calcularSalarioLiquido());

```



```

        out.print("<br/>");

        out.print(fp.getCartao());

    } catch (Exception e) {
        out.print(e.getMessage());
    }
}

```

- Para executar, selecione a opção Run As... -> Run On Server e certifique-se que o arquivo EAR esteja na lista de execução.
- Executar a página JSP e constatar o resultado.

Exercício 2: Calculadora usando EJB e JSF

- Criar um novo projeto **EJB Project** chamado **02_Calculadora** (Não precisa marcar a opção "Add Project to" na EAR).
- Criar uma interface chamada **CalculadoraLocal**, que defina as quatro operações de uma calculadora. Marcar esta interface com a anotação **@Local**:

```

package br.com.ead.interfaces;

import javax.ejb.Local;

@Local
public interface CalculadoraLocal {
    double somar(double x, double y);
    double subtrair(double x, double y);
    double multiplicar(double x, double y);
    double dividir(double x, double y);
}

```

- Definir a classe que representará o componente EJB. A classe deverá se chamar **CalculadoraBean** e deverá implementar a interface do item 2:

```

package br.com.ead.bean;

import javax.ejb.Stateless;
import br.com.ead.interfaces.CalculadoraLocal;

@Stateless
public class CalculadoraBean implements CalculadoraLocal {

```

```
@Override
public double somar(double x, double y) {
    return x + y;
}

@Override
public double subtrair(double x, double y) {
    return x - y;
}
@Override
public double multiplicar(double x, double y) {
    return x * y;
}

@Override
public double dividir(double x, double y) {
    return x / y;
}
}
```

- Criar um novo projeto web (Dynamic Web Project) baseado em JSF, chamado **02_Calculadora-Web**. Neste projeto, selecionar a opção *JavaServer Faces 2.1* como Configuration:

Dynamic Web Project
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: 02_CalculadoraWeb

Project location
☒ Use default location
Location: C:\Documentos\projetos\Java\EBJ_Aulas\02_CalculadoraWeb

Target runtime
JBoss 7.1 Runtime

Dynamic web module version
3.0

Configuration
JavaServer Faces v2.1 Project
Configures a Dynamic Web application to use JSF v2.1

EAR membership
☐ Add project to an EAR
EAR project name: EAR

Working sets
☐ Add project to working sets
Working sets:

< Back Next > Finish Cancel

- Avançar, selecionar a opção que permite criar o arquivo web.xml e manter a configuração abaixo:

New Dynamic Web Project

JSF Capabilities
Add JSF capabilities to this Web Project

JSF Implementation Library

Type: **Library Provided by Target Runtime**

The targeted runtime is able to provide the library required by this facet. Selecting this option will configure the project to use that library.

☒ Configure JSF servlet in deployment descriptor

JSF Configuration File: /WEB-INF/faces-config.xml

JSF Servlet Name: Faces Servlet

JSF Servlet Class Name: javax.faces.webapp.FacesServlet

URL Mapping Patterns: *.faces

Add... Remove

< Back Next > Finish Cancel

- Neste ponto, selecionar o pacote contendo a interface **CalculadoraLocal** e exportá-la para um arquivo **.jar**. Em seguida, importar este *jar* no seu projeto web. Como sugestão, chamar o arquivo de **calc.jar**.
- No projeto WEB criar um *Managed Bean* para conter os resultados:

```
package br.com.ead.mb;
```

```
import javax.ejb.EJB;
```

```
import javax.faces.bean.ManagedBean;
```

```
import br.com.ead.interfaces.CalculadoraLocal;
```

```
@ManagedBean(name="calcMB")
public class CalculadoraManagedBean {

    //informamos o local do EJB via JNDI
    @EJB(lookup = "ejb:/02_Calculadora/CalculadoraBean!br.com.ead.interfaces.
CalculadoraLocal")
    private CalculadoraLocal calc;

    private double x, y;
    private String resultado;

    public double getX() {
        return x;
    }
    public void setX(double x) {
        this.x = x;
    }
    public double getY() {
        return y;
    }
    public void setY(double y) {
        this.y = y;
    }

    public String getResultado() {
        return resultado;
    }

    public void somar(){
        double result = calc.somar(x, y);
        resultado = "Soma = " + result;
    }
    public void subtrair() {
        double result = calc.subtrair(x, y);
        resultado = "Subtração = " + result;
    }
    public void multiplicar(){
        double result = calc.multiplicar(x, y);
        resultado = "Multiplicação = " + result;
    }
    public void dividir(){
        double result = calc.dividir(x, y);
        resultado = "Divisão = " + result;
    }
}
```

```
}
```

Observe que incluímos o atributo "lookup" à anotação @EJB. Isso foi necessário porque o projeto web e o projeto EJB estão separados, não fazendo parte do mesmo EAR como ocorreu no exemplo 1. O valor deste atributo é proveniente de um mecanismo conhecido por JNDI (*Java Naming and Directory Interface*). O JBoss fornece uma facilidade para obter este valor: basta executar o projeto EJB com a opção "Run on Server", que ela aparece no log do serviço.

- Definir, no projeto Web, um arquivo xhtml para testar a calculadora remota!!!

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>Calculadora</title>
  </h:head>
  <h:body>
    <f:view>
      <h:form>

        <h:outputText value="Valor de X:" />
        <h:inputText value="#{calcMB.x}" />
        <br />
        <h:outputText value="Valor de Y:" />
        <h:inputText value="#{calcMB.y}" />
        <br />
        <h:commandButton value="Soma" action="#{calcMB.somar}" />
        <h:commandButton value="Subtração" action="#{calcMB.subtrair}" />
        <h:commandButton value="Multiplicação" action="#{calcMB.multiplicar}" />
        <h:commandButton value="Divisão" action="#{calcMB.dividir}" />
        <br />
        <h:outputText value="#{calcMB.resultado}" />
      </h:form>
    </f:view>
  </h:body>
</html>
```

- Para executar, manter os dois projetos na lista do servidor.

Exercício 3

Caro aluno, este exercício trata de um componente EJB que realiza persistência usando JPA. Para definir a camada de persistência, apresentaremos as etapas para sua realização. Como exercício você deve definir a camada de apresentação usando JSF.

- Definição do datasource (para várias aplicações, definir um datasource no servidor JBoss): na pasta **[JBoss]\modules\com\mysql\main** (criar a pasta):
 - copiar o driver do mysql
 - criar o arquivo module.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.13-bin.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
  </dependencies>
</module>
```

- Localizar o arquivo standalone.xml na pasta **[JBoss]\standalone\configuration**. Dentro da tag <datasources> incluir o elemento:

```
<datasource jndi-name="java:/jdbc/ead" pool-name="ead"
enabled="true" use-java-context="true">
  <connection-url>jdbc:mysql://localhost:3306/livrosejb</connection-url>
  <driver>com.mysql</driver>
  <security>
    <user-name>root</user-name>
    <password>ead</password>
  </security>
</datasource>
```

e dentro da tag <drivers> a seguinte configuração:

```
<driver name="com.mysql" module="com.mysql">
  <xa-datasource-class>com.mysql.jdbc.Driver</xa-datasource-class>
</driver>
```

- Criar um novo projeto EjbProject, chamado **03_LivrosEJB**. Selecionar JBoss 7.0 ou 7.1, mas manter a versão 3.0.
- Na pasta **ejbModule**, criar a classe para a entidade Livros:

```
package br.com.ead.entity;
```

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="livros", schema="livrosejb")
public class Livros {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="ID")
    private Integer id;

    @Column(name="TITULO")
    private String titulo;

    @Column(name="AUTOR")
    private String autor;

    @Column(name="PRECO")
    private Double preco;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getAutor() {
        return autor;
    }
    public void setAutor(String autor) {
        this.autor = autor;
    }
}
```



```

    public Double getPreco() {
        return preco;
    }
    public void setPreco(Double preco) {
        this.preco = preco;
    }
}

```

- Criar, na pasta \META-INF, o arquivo persistence.xml:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0 "
    xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="eadPU" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>jdbc/ead</jta-data-source>
        <properties>
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.
MySQL5InnoDBDialect" />
        </properties>
    </persistence-unit>
</persistence>

```

- Definir o banco de dados no MySQL.
- Criar o SessionBean e a interface local. Chamar a classe de LivrosBean.
- Codificar a classe e a interface:

```

package br.com.ead.bean;

import java.util.List;
import javax.ejb.Local;
import br.com.ead.entity.Livros;

@Local
public interface LivrosBeanLocal {
    void add(Livros livro);
    List<Livros> getAll();
}

```

```

}

package br.com.ead.bean;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import br.com.ead.entity.Livros;

@Stateless
public class LivrosBean implements LivrosBeanLocal {

    public LivrosBean() { }

    @PersistenceContext(unitName="eadPU")
    private EntityManager em;

    public void add(Livros livro){
        em.persist(livro);
    }

    public List<Livros> getAll(){
        TypedQuery<Livros> query = em.createQuery("select u from Livros u", Livros.class);
        return query.getResultList();
    }
}

```

- Iniciar o servidor e verificar se o datasource foi definido:
Bound data source [java:/jdbc/ead]
- Usando JSF e baseado no exemplo da calculadora, criar a aplicação cliente, tanto para inserir como para listar livros.

Exercício 4

Este exercício trata de um componente **EJB MDB (Message-Driven Beans)**.

Message-Driven Bean é um componente que permite que aplicações Java EE processem mensagens assincronamente. Este tipo de bean normalmente age como um listener de mensagem JMS (Java Message Service), que é similar a um listener de eventos, como eventos em uma interface gráfica, mas recebe mensagens JMS ao invés de eventos propriamente ditos.

As mensagens podem ser enviadas por qualquer componente Java EE, como um servlet, outro componente EJB, aplicações JSF, ou até mesmo por aplicações não Java EE, como um cliente Java executando o método `main()`.

Diferenças entre Message-Driven Beans e Session Beans: a diferença de maior destaque entre **MDB** e **SB** é que clientes não acessam MDB através de interfaces. Diferente de SB, um MDB possui somente uma classe bean.

Em muitos aspectos, um MDB se assemelha a um Stateless Session Bean.

As variáveis de instância de um MDB podem conter dados através da manipulação de mensagens de clientes, como por exemplo, uma conexão JMS, conexão com um banco de dados ou uma referência para um objeto EJB.

Componentes cliente não localizam MDBs ou invocam seus métodos diretamente. Ao invés disso, um cliente acessa um MDB através de JMS enviando mensagens para o destino onde o MDB é um *MessageListener*. A configuração de um MDB ocorre durante o deploy no servidor.

Os MDBs possuem as seguintes características:

- Executam sobre uma simples mensagem do cliente.
- São invocados assincronamente.
- Possuem vida relativamente curta.
- Não representam dados em uma base de dados, mas acessam esta base.
- São *stateless*.

Quando usar MDBs

Session beans permitem o envio e recebimento de mensagens sincronicamente. A fim de evitar bloqueio de recursos no servidor, use MDBs.

Nas etapas seguintes desenvolveremos um MDB.

- Criar um projeto EJB Project chamado **04_ProjetoMDB**.
- Neste projeto, incluir a classe **Cliente**, conforme modelo abaixo:

```
package br.com.ead.mdb.classes;
```

```
import java.io.Serializable;
```

```
public class Cliente implements Serializable {
```

```
    private static final long serialVersionUID = 1L;
```

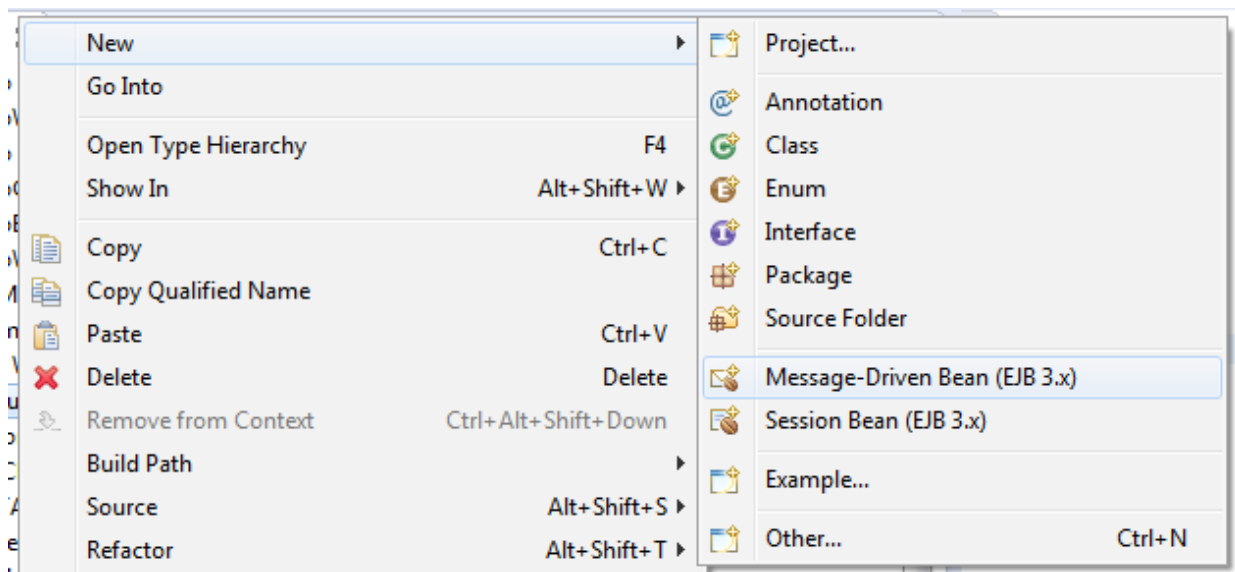
```
    private int id;
```

```
private String nome, telefone, email;
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getNome() {
    return nome;
}
public void setNome(String nome) {
    this.nome = nome;
}
public String getTelefone() {
    return telefone;
}

public void setTelefone(String telefone) {
    this.telefone = telefone;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}

@Override
public String toString(){
    return "Cliente [id=" + this.getId() + ", nome=" +
        this.getNome() + ", telefone=" + this.getTelefone() +
        ", email=" + this.getEmail() + "];"
}
}
```

Criar o consumidor Message-Driven Bean. Crie uma classe com as configurações apresentadas na interface abaixo:



- Acrescentar as alterações na classe resultante, conforme modelo:

```
package br.com.ead.mdb;
```

```
import java.util.Date;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;
import javax.jms.TextMessage;
import br.com.ead.mdb.classes.Cliente;
```

```
@MessageDriven(
    activationConfig = { @ActivationConfigProperty(
        propertyName = "destinationType", propertyValue = "javax.jms.
Queue"),
        @ActivationConfigProperty(
            propertyName="destination", propertyValue = "queue/ExemploQueue")
    })
```

```
public class QueueListenerMDB implements MessageListener {
```

```
    public QueueListenerMDB() {
        // TODO Auto-generated constructor stub
    }
}
```

```

public void onMessage(Message message) {
    try {
        if (message instanceof TextMessage) {
            System.out.println("Queue: TextMessage recebida em " + new Date());
            TextMessage msg = (TextMessage) message;
            System.out.println("Message is : " + msg.getText());
        } else if (message instanceof ObjectMessage) {
            System.out.println("Queue: ObjectMessage recebida em " + new Date());
            ObjectMessage msg = (ObjectMessage) message;
            Cliente cliente = (Cliente) msg.getObject();
            System.out.println("Detalhes do cliente: ");
            System.out.println(cliente.getId());
            System.out.println(cliente.getNome());
            System.out.println(cliente.getTelefone());
            System.out.println(cliente.getEmail());
        } else {
            System.out.println("Nenhuma mensagem válida!");
        }
    } catch (JMSException e) {
        e.printStackTrace();
    }
}

```

- Localizar o arquivo **standalone.xml** na pasta **[JBoss]\standalone\configuration**. Dentro da tag <extensions> incluir o elemento:
<extension module="org.jboss.as.messaging"/>
- Dentro do elemento <profile>, adicionar o seguinte elemento <subsystem>:

```

<subsystem xmlns="urn:jboss:domain:messaging:1.1">
    <hornetq-server>
        <persistence-enabled>true</persistence-enabled>
        <journal-file-size>102400</journal-file-size>
        <journal-min-files>2</journal-min-files>
    <connectors>
        <netty-connector name="netty" socket-binding="messaging"/>
        <netty-connector name="netty-throughput" socket-binding="messaging-throughput">
            <param key="batch-delay" value="50"/>
        </netty-connector>
        <in-vm-connector name="in-vm" server-id="0"/>
    </connectors>

    <acceptors>

```

```

<netty-acceptor name="netty" socket-binding="messaging"/>
<netty-acceptor name="netty-throughput" socket-binding="messaging-throughput">
  <param key="batch-delay" value="50"/>
  <param key="direct-deliver" value="false"/>
</netty-acceptor>
<in-vm-acceptor name="in-vm" server-id="0"/>
</acceptors>

<security-settings>
  <security-setting match="#">
    <permission type="send" roles="guest"/>
    <permission type="consume" roles="guest"/>
    <permission type="createNonDurableQueue" roles="guest"/>
    <permission type="deleteNonDurableQueue" roles="guest"/>
  </security-setting>
</security-settings>

<address-settings>
  <address-setting match="#">
    <dead-letter-address>jms.queue.DLQ</dead-letter-address>
    <expiry-address>jms.queue.ExpiryQueue</expiry-address>
    <redelivery-delay>0</redelivery-delay>
    <max-size-bytes>10485760</max-size-bytes>
    <address-full-policy>BLOCK</address-full-policy>
    <message-counter-history-day-limit>10</message-counter-history-day-limit>
  </address-setting>
</address-settings>

<jms-connection-factories>
  <connection-factory name="InVmConnectionFactory">
    <connectors>
      <connector-ref connector-name="in-vm"/>
    </connectors>
    <entries>
      <entry name="java:/ConnectionFactory"/>
    </entries>
  </connection-factory>
  <connection-factory name="RemoteConnectionFactory">

    <connectors>
      <connector-ref connector-name="netty"/>
    </connectors>
    <entries>
      <entry name="RemoteConnectionFactory"/>
    </entries>
  </connection-factory>

```

```

        <entry name="java:jboss/exported/jms/RemoteConnectionFactory"/>
    </entries>
</connection-factory>
<pooled-connection-factory name="hornetq-ra">
    <transaction mode="xa"/>
    <connectors>
        <connector-ref connector-name="in-vm"/>
    </connectors>
    <entries>
        <entry name="java:/JmsXA"/>
    </entries>
</pooled-connection-factory>
</jms-connection-factories>

<jms-destinations>
    <jms-queue name="testQueue">
        <entry name="queue/ExemploQueue"/>
    </jms-queue>
    <jms-topic name="testTopic">
        <entry name="topic/MyTopic"/>
    </jms-topic>
</jms-destinations>
</hornetq-server>
</subsystem>

```

Obs.:

- Hornetq ie um projeto opensource que permite a construção de sistemas assíncronos multiplataforma. A partir da versão 7.0 do JBoss, um grupo de configurações no arquivo standalone.xml é suficiente. Em versões anteriores era necessário usar a API Hornetq, disponível em **<http://hornetq.jboss.org/docs.html>**.
- Projetado pela Sun Microsystems, com o apoio de empresas associadas, e lançado para o mercado em agosto de 1998, o JMS surgiu para permitir que os aplicativos escritos na linguagem Java pudessem criar, receber e enviar mensagens destinadas ou oriundas de outros aplicativos. A principal característica deste tipo de processamento, classificado como fracamente acoplado, é que todas as operações que envolvem a troca de mensagens são feitas de forma assíncrona, fazendo com que as aplicações participantes não precisem ficar bloqueadas esperando o término de alguma computação remotamente solicitada, como ocorre naquelas aplicações que utilizam o Remote Procedure Call (RPC) ou mesmo o Remote Method Invocation (RMI).
- Dentro do elemento <socket-binding-group> adicionar os elementos:


```
<socket-binding name="messaging" port="5445"/>
<socket-binding name="messaging-throughput" port="5455"/>
```

- Localize o elemento `<subsystem xmlns="urn:jboss:domain:ejb3:1.2">`. Em seguida, adicione o elemento:

```
<mdb>
  <resource-adapter-ref resource-adapter-name="hornetq-ra"/>
  <bean-instance-pool-ref pool-name="mdb-strict-max-pool"/>
</mdb>
```

- Inicie o servidor. Após sua inicialização, verifique as mensagens:

```
Bound messaging object to jndi name java:/queue/ExemploQueue
Started message driven bean 'QueueListenerMDB' with 'hornetq-ra' resource adapter
```

- Para criar o cliente JMS, usaremos uma aplicação Web com um servlet. Definir um novo Dynamic Web Project com o nome **04_MDBCliente**. Usar o mesmo servidor de aplicações usado no MDB.
- Definir um arquivo .jar contendo a classe Cliente que você criou no primeiro projeto deste exercício.
- Adicionar o arquivo **.jar** criado na pasta **WEB-INF/lib** da aplicação Web.
- Definir, no projeto Web, um servlet chamado **ServletMdb**, com o mapeamento **/mdb**:
- No método doGet() escrever o código:

```
final String QUEUE_LOOKUP = "queue/ExemploQueue";
final String CONNECTION_FACTORY = "ConnectionFactory";

PrintWriter out = response.getWriter();
try{
    Context context = new InitialContext();
    QueueConnectionFactory factory =
        (QueueConnectionFactory)context.lookup(CONNECTION_FACTORY);
    QueueConnection connection = factory.createQueueConnection();
    QueueSession session =
        connection.createQueueSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);

    Queue queue = (Queue)context.lookup(QUEUE_LOOKUP);
    QueueSender sender = session.createSender(queue);

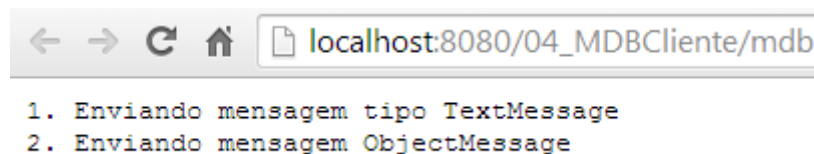
    //1. Enviando objeto TextMessage
    TextMessage message = session.createTextMessage();
    message.setText("Exemplo EJB3 MDB Queue!!!");
    sender.send(message);
    out.println("1. Enviando mensagem tipo TextMessage");
}
```

```
//2. Enviando objeto ObjectMessage
ObjectMessage objMsg = session.createObjectMessage();

Cliente cliente = new Cliente();
cliente.setId(1500);
cliente.setNome("Ead Pós");
cliente.setTelefone("3385-8010");
cliente.setEmail("emilio.celso@ead.com.br");
objMsg.setObject(cliente);
sender.send(objMsg);
out.println("2. Enviando mensagem ObjectMessage");

session.close();
}
catch(Exception ex){
    ex.printStackTrace();
}
```

- Incluir o projeto Web ao servidor.
- Executar o servlet. O resultado no browser deverá ser semelhante a:



e no log do servidor, a informação:

21:33:05,558 INFO [stdout] (Thread-5 (HornetQ-client-global-threads-4967270)) Queue: TextMessage recebida em Sun Aug 24 21:33:05 BRT 2014

21:33:05,559 INFO [stdout] (Thread-5 (HornetQ-client-global-threads-4967270)) Message is : Exemplo EJB3 MDB Queue!!!

21:33:05,560 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) Queue: ObjectMessage recebida em Sun Aug 24 21:33:05 BRT 2014

21:33:05,563 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) Detalhes do cliente:

21:33:05,564 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) 1500

21:33:05,564 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) Ead Pós

21:33:05,565 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) 3385-8010

21:33:05,565 INFO [stdout] (Thread-6 (HornetQ-client-global-threads-4967270)) emilio.celso@ead.com.br

Referências

BAUER, C.; KING, G. **Hibernate em Ação**. Rio de Janeiro: Ciência Moderna , 2005.

BOFF, M.; et al. **Artigo EJB**. Disponível em: <http://www.ebah.com.br/content/ABAAAfV_0AI/artigo-ejb>. Acesso em 23 out. 2014.

DIGITAL ROAD. **Projeto de sistemas distribuídos usando CORBA, parte 1: O projeto**. 2011. Disponível em: <<http://digitalroad.wordpress.com/category/rmi/>>. Acesso em 23 out. 2014.

FERNANDES, R. G.; LIMA, G. A. F. **Hibernate com Anotações**. Disponível em: <http://www.futurepages.org/wiki/lib/exe/fetch.php?media=quickstart:hibernate_annotacoes.pdf>. Acesso em: 06 nov. 2014.

GONÇALVES, E. **Desenvolvendo Aplicações Web com JSP , Servlets , Java Server Faces , Hibernate , EJB 3 Persistence**. Rio de Janeiro: Ciência Moderna , 2007.

GONÇALVES, A. **Beginning Java EE 6 Platform with GlassFish 3**. Apress, 2010.

HIBERNATE. **Hibernate search documentation guide**. Disponível em: <<http://hibernate.org/search/documentation/>>. Acesso em: 06 nov. 2014.

JBoss.ORG. **Hibernate Community Documentation- Capítulo 2- Arquitetura**. 2004. Disponível em: <<https://docs.jboss.org/hibernate/orm/3.5/reference/pt-BR/html/architecture.html>>. Acesso em 20 out. 2014.

K19 TREINAMENTOS. **Desenvolvimento Web avançado com JSF2, EJB3.1 e CDI**. Disponível em: <<http://pt.slideshare.net/RodolfoSilva9/desenvolvimento-web-avancadocomjsf2ejb31ecdi>>. Acesso em: 07 nov. 2014.

OLIVEIRA, E. C. M. **Conhecendo a plataforma J2EE: um breve overview**. Disponível em: <<http://www.linhadecodigo.com.br/artigo/333/conhecendo-a-plataforma-j2ee-um-breve-overview.aspx>>. Acesso em 22 out. 2014.