

Tabela de conteúdos

Introdução	1.1
Java Persistence API	1.2
Entity	1.2.1
EntityManager	1.2.2
Exemplo de DAO (CRUD) utilizando JPA	1.2.3
Estratégia de SEQUENCE para gerar ID	1.2.4
Utilizando chave composta	1.2.5
Exercícios	1.2.6
Relacionamento entre entidades	1.3
Os quatro tipos de cardinalidade	1.3.1
CascadeType	1.3.2
FetchType	1.3.3
Exercícios	1.3.4
Consultas com JPAQL	1.4
Interface Query	1.4.1
Exercícios	1.4.2
Enterprise JavaBeans	1.5
Session Bean	1.5.1
Criando um projeto EJB no NetBeans	1.6
Exercícios	1.6.1
Usando a anotação @EJB	1.7
Criando uma aplicação EJB + JPA	1.8
Interceptando os EJBs para criar objetos DAOs	1.9
Web Services SOAP	1.10
Web Service REST	1.11
GET, POST, PUT e DELETE	1.11.1
Integração via REST com OpenWS	1.11.2

Java Naming and Directory Interface	1.12
Remote Method Invocation	1.13

Desenvolvimento Distribuído com Java EE

Bem vindo ao **Desenvolvimento Distribuído com Java EE!** Se você está lendo esse livro é porque tem interesse em aprender como criar aplicações distribuídas em Java.

O livro está dividido em três partes para explicar persistencia de dados, criação de componentes distribuídos e criação de web services.

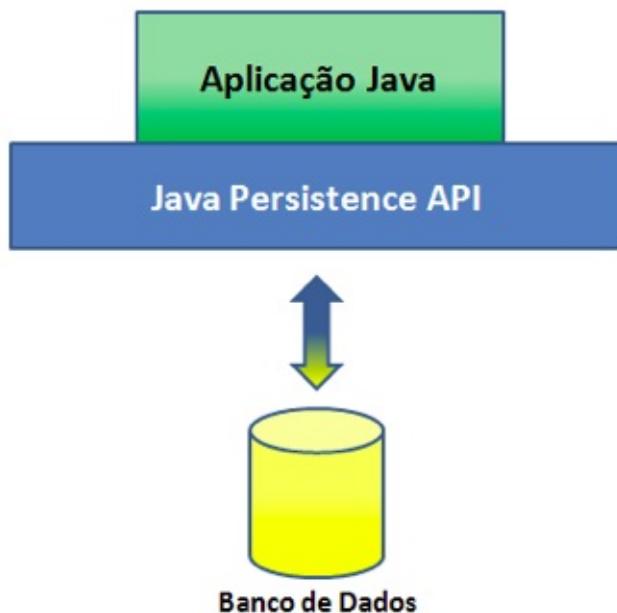
Durante o desenrolar do livro, apresento alguns exemplos no estilo passo a passo e também alguns exercícios de fixação.

Boa leitura.

Rafael Guimarães Sakurai

Java Persistence API

O Java Persistence API é um framework para camada de persistência dos dados, que fornece uma camada de comunicação entre a aplicação escrita em Java e o banco de dados.



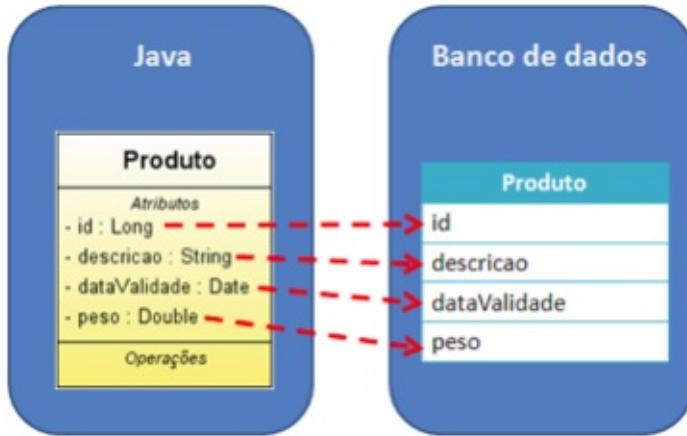
Algumas facilidades que o JPA oferece são:

- Conversão de registros do banco de dados em objetos Java;
- Não precisa criar códigos SQL para salvar, alterar ou remover registros do banco de dados;
- A aplicação não fica presa a um banco de dados sendo simples a troca.

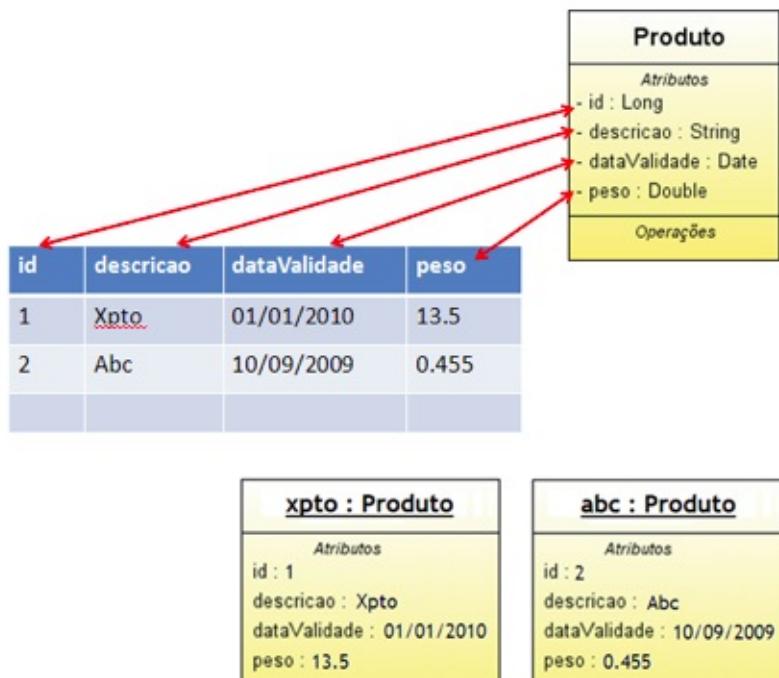
Também aumenta a produtividade dos desenvolvedores que utilizam o banco de dados, deixando de forma transparente a sua utilização, principalmente por não deixar a programação Java vinculada a um tipo específico de banco de dados.

Para trazer as informações do banco de dados e convertê-las em classes Java, acaba sendo um pouco trabalhoso. Quando é usado JDBC puro há a necessidade de realizar o mapeamento entre os atributos e colunas do banco de dados, às vezes é necessário fazer uma conversão do tipo de dado declarado no banco de dados com o tipo de dado utilizado na classe. O mesmo processo ocorre quando os objetos Java são salvos no banco de dados.

O JPA utiliza o conceito de **mapeamento objeto / relacional (ORM – Object / Relational Mapping)** para fazer ponte entre a base de dados relacional e os objetos Java. A figura a seguir mostra o próprio framework faz o relacionamento entre os atributos das classes Java com a tabela do banco de dados.



O JPA cria uma instância da classe Produto para cada linha da tabela Produto, como mostrado na figura a seguir, e também atribui os valores das propriedades da classe Produto de acordo com os valores das colunas da tabela. Por padrão o JPA realizará o mapeamento da classe e atributos com o mesmo nome.



Atualmente o JPA está na versão 2.1 e pode ser utilizado tanto no contexto *Java EE (Java Enterprise Edition)* como *Java SE (Java Standard Edition)*.

Entity

Uma Entity (Entidade) é um objeto leve de domínio persistente utilizado para representar uma tabela da base de dados, sendo que cada instância da entidade corresponde a uma linha da tabela.

A Entity é baseada em uma simples classe Java do tipo *Plain Old Java Object (POJO)*, portanto uma classe Java comum com anotações para fornecer informações mais específica ao gerenciador das entidades.

Exemplo de entity que representa um produto:

```
package pbc.jpa.exemplo1.entity;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

/**
 * Classe utilizada para representar a tabela Produto.
 */
@Entity
public class Produto implements Serializable {
    private static final long serialVersionUID
        = 4185059514364687794L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String descricao;
    @Temporal(TemporalType.DATE)
    private Date dataValidade;
    private Double peso;
```

```
public Date getDataValidade() { return dataValidade; }
public void setDataValidade(Date dataValidade) {
    this.dataValidade = dataValidade;
}

public String getDescricao() { return descricao; }
public void setDescricao(String descricao) {
    this.descricao = descricao;
}

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

public Double getPeso() { return peso; }
public void setPeso(Double peso) { this.peso = peso; }
}
```

Neste exemplo é utilizada a anotação **@Entity** para definir esta classe como uma entidade do banco de dados. Utilizamos a anotação **@Id** define que o atributo **Long id** é chave primaria da tabela **Produto**; também é definido que o atributo **Long id** tem seu valor gerado automaticamente através da anotação **@GeneratedValue**. Quando têm atributos que representam datas ou tempos é necessário adicionar a anotação **@Temporal** para definir que o atributo **Date dataValidade** é um campo **Date** no banco de dados.

O relacionamento feito entre a classe Java e a tabela do Banco de Dados, ocorre automaticamente quando o nome da classe é igual ao nome da tabela; o mesmo vale para os atributos que correspondem às colunas da tabela. Desta forma quando solicitar ao JPA para salvar, consultar, alterar ou excluir uma entidade automaticamente será criado o Script SQL para executar a operação.

Esta classe está implementando a interface **java.io.Serializable** para informar que pode ser serializada e trafegada pela rede de computadores. Quando as entidades fazem parte de uma aplicação console ou desktop não há necessidade de implementar esta interface, a não ser que a aplicação precise trafegar o objeto desta classe pela rede ou em algum HD. Quando implementada esta interface

também é preciso definir o atributo **private static final long serialVersionUID** com um número longo, isto serve para identificar a classe quando o objeto for trafegar via rede.

Observação: o nome das tabelas no banco de dados não é case sensitive quando o banco de dados é instalado no Windows, no caso do Linux já é case sensitive.

Anotações para Entity

As entidades podem ser melhor detalhadas adicionando mais algumas anotações e suas propriedades, estas anotações podem informar ao JPA que por exemplo, uma entidade não segue o padrão de nome igual ao da tabela, ou que sua tabela no banco tem um relacionamento de Um-Para-Muitos com outra tabela, que a tabela utiliza um gerador de ID do tipo SEQUENCE para definir o número da chave primaria e outras informações que veremos a seguir:

Obrigatoriamente toda entidade do JPA precisa ter pelo menos as anotações **javax.persistence.Entity** que informa que é uma tabela do banco de dados e **javax.persistence.Id** que informa qual o atributo é chave primaria da tabela.

javax.persistence.Entity

Usado para definir que a classe é uma Entity, por padrão quando o nome da Entity é igual ao nome da tabela o relacionamento é feito automaticamente pelo JPA. As propriedades da anotação **@Entity** são listadas na tabela a seguir:

Propriedade	Descrição
name	Informa o nome da entidade, por padrão o nome da entidade é nome da classe. Este nome é utilizado para referenciar a entidade na consulta.

javax.persistence.Table

Define o nome da tabela no banco de dados. As propriedades da anotação **@Table** são listadas na tabela a seguir:

Propriedade	Descrição
catalog	O catalogo da tabela.
name	O nome da tabela.
schema	O esquema da tabela.
uniqueConstraints	Regras que podem ser adicionadas na tabela.

javax.persistence.Id

Informa o atributo da Entity que representa a chave primaria.

javax.persistence.Column

Informa as configurações de coluna da tabela, por padrão quando o nome do atributo da Entity é igual ao nome da coluna da tabela, o relacionamento é feito automaticamente pelo JPA. As propriedades da anotação @Column são listadas na tabela a seguir:

Propriedade	Descrição
columnDefinition	Definição do tipo da coluna.
insertable	Informa se a tabela deve ser incluída no SQL de insert, por padrão é true.
length	Tamanho da coluna, por padrão é 255.
name	Nome da tabela que contém está coluna, se não for informado a coluna assume o nome da tabela da entity.
nullable	Informa se o valor pode ser null.
precision	Precisão da coluna decimal.
scale	Número de casas decimais, usado somente em coluna com número decimal.
table	Nome da tabela que contém está coluna, se não for informado assume o nome da tabela da entity.
unique	Informa se a coluna é chave única.
updatable	Informa se a coluna deve ser incluída no SQL de update, por padrão é true.

javax.persistence.SequenceGenerator

Utilizado para representar uma sequência numérica gerada através do banco de dados. As propriedades da anotação `@SequenceGenerator` são listadas na tabela a seguir:

Propriedade	Descrição
name	Nome único para o gerador que pode ser referenciado por uma ou mais classes que pode ser utilizado para gerar valores de chave primária.
allocationSize	A quantidade que será incrementada na sequence, o padrão é 50.
initialValue	Valor inicial da sequence.
sequenceName	Nome da sequence do banco de dados.

javax.persistenceGeneratedValue

Define a estratégia para criar o ID, pode ser tipo AUTO (incrementa automaticamente 1, 2, 3 em sequência) ou utilizando uma SEQUENCE. As propriedades da anotação `@GeneratedValue` são listadas na tabela a seguir:

Propriedade	Descrição
generator	Nome do gerador da chave primária que é especificado na anotação <code>@SequenceGenerator</code> ou <code>@TableGenerator</code> .
strategy	Estratégia de geração de chave primária que o serviço de persistência precisa usar para gerar a chave primária. Seu valor pode ser obtido através da enum <code>javax.persistence.GenerationType</code> , os valores podem ser AUTO, IDENTITY, SEQUENCE ou TABLE.

javax.persistence.Temporal

Utilizado para representar campos de Data e Hora, nesta anotação podemos definir o tipo de dado DATE, TIME e TIMESTAMP. As propriedades da anotação `@Temporal` são listadas na tabela a seguir:

Propriedade	Descrição
value	O tipo usado para mapear java.util.Date e java.util.Calendar. Seu valor pode ser obtido através da enum javax.persistence.TemporalType, os valores podem ser DATE, TIME e TIMESTAMP.

javax.persistence.Transient

Informa que o atributo não é persistente.

Exemplo de Entity

Nesse exemplo será criado uma entity para tabela Usuario a seguir:

```
CREATE TABLE Usuario (
    id          NUMBER(10)      NOT NULL PRIMARY KEY,
    nome        VARCHAR2(100)   NOT NULL,
    dataNasc    DATE           NOT NULL,
    email       VARCHAR2(150)   NOT NULL,
    ativo       NUMBER(1)       NOT NULL,
    comentario VARCHAR2(200)
);
```

Crie a Entity para representar a tabela Usuario:

```
package pbc.jpa.exemplo1.entity;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
```

```
/**  
 * Entidade utilizada para representar um Usuario.  
 */  
  
@Entity  
public class Usuario implements Serializable {  
    private static final long serialVersionUID  
        = -8762515448728066246L;  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
    @Column(nullable = false)  
    private String nome;  
    @Temporal(TemporalType.DATE)  
    @Column(name="dataNasc", nullable = false)  
    private Date dataNascimento;  
    @Column(nullable = false)  
    private String email;  
    @Column(nullable = false)  
    private Boolean ativo;  
    private String comentario;  
  
    public Boolean getAtivo() { return ativo; }  
    public void setAtivo(Boolean ativo) {  
        this.ativo = ativo;  
    }  
  
    public String getComentario() { return comentario; }  
    public void setComentario(String comentario) {  
        this.comentario = comentario;  
    }  
  
    public Date getDataNascimento() {  
        return dataNascimento;  
    }  
    public void setDataNascimento(Date dataNascimento) {  
        this.dataNascimento = dataNascimento;  
    }  
  
    public String getEmail() { return email; }
```

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

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

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

Foi utilizada a anotação **@Entity** para informar que a classe **Usuario** é uma entidade do banco de dados; também foi definido que a propriedade **Long id** será o **ID** da tabela através da anotação **@Id** e é informado que seu valor será gerado automaticamente com a anotação **@GeneratedValue**. Por meio da anotação **@Column** é especificado quais os atributos não podem ser **null** e que o atributo **Date dataNascimento** está mapeado para a coluna **dataNasc** da tabela **Usuario**.

EntityManager

O EntityManager é um serviço responsável por gerenciar as entidades. Por meio dele é possível gerenciar o ciclo de vida das entidades, a operação de sincronização com a base de dados (inserir, atualizar ou remover), a consulta de entidades, entre outras operações.

Quando uma entidade está associada a um EntityManager, esta entidade está no contexto em que pode ser persistida, em que todas as operações realizadas no objeto da entidade é refletido no banco de dados. Todas as identidades das entidades são únicas, portanto para cada registro no banco de dados haverá apenas uma referência no contexto do EntityManager.

O EntityManager pode ser gerenciado de duas formas:

- Gerenciado pelo Container;
- Gerenciado pela Aplicação.

Unidade de Persistência

A unidade de persistência é utilizada para configurar as informações referentes ao provedor do JPA (implementação da especificação JPA) e ao banco de dados; também é possível identificar as classes que serão mapeadas como entidades do banco de dados.

Para definir a unidade de persistência é necessário um arquivo XML chamado **persistence.xml**, que deve ser criado na pasta **META-INF** do projeto. Por meio deste arquivo é possível definir quantas unidades de persistência for necessárias para o projeto. Exemplo de unidade de persistência mapeado para um banco de dados Oracle:

```
<?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="ExemploJPAPU" transaction-type
        ="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>pbc.jpa.exercicio1.modelo.Livro</class>
        <properties>
            <property name="hibernate.connection.username"
                value="usuario"/>
            <property name="hibernate.connection.password"
                value="senha"/>
            <property name="hibernate.connection.driver_class"
                value="oracle.jdbc.driver.OracleDriver"/>
            <property name="hibernate.connection.url"
                value="jdbc:oracle:thin:@localhost:1521:XE"/>
            <property name="hibernate.cache.provider_class"
                value="org.hibernate.cache.NoCacheProvider"/>
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.Oracle9Dialect"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Neste arquivo **persistence.xml** a tag `persistence-unit` define a unidade de persistência, e na propriedade **name** qual seu nome (utilizado quando um EntityManager é criado por meio do EntityManagerFactory no contexto Java SE ou quando é realizado注入 de dependência através da anotação **javax.persistence.PersistenceUnit** no contexto Java EE). A propriedade `transaction-type` informa qual o tipo de transação (RESOURCE_LOCAL ou JTA). Se a aplicação é Java SE, então, utilize o tipo de transação **RESOURCE_LOCAL**, assim programaticamente são criadas as transações com o banco de dados e na aplicação Java EE utilize o **JTA** que acessa um pool de conexões em um servidor web.

Em uma unidade de persistência utilize a tag `provider` para informar qual a API que fornecerá uma implementação do JPA.

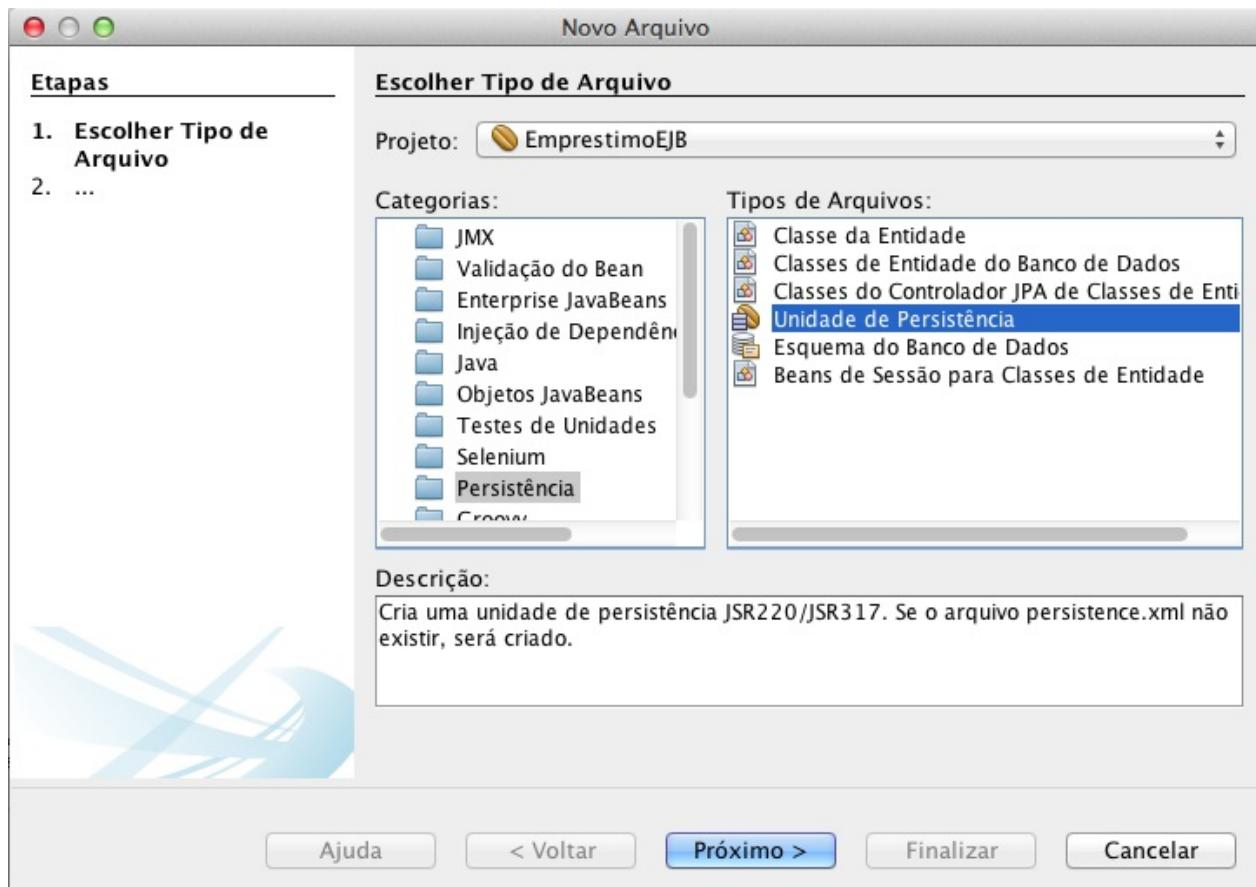
Em uma aplicação Java SE é necessário informar quais as classes são entidades do banco de dados através da tag `class` e também informar quais as propriedades necessárias para encontrar o banco de dados através da tag `properties`.

Nas aplicações Java EE é possível criar um pool de conexões com o banco de dados no servidor web; neste caso é aconselhado utilizar o tipo de transação **Java Transaction API (JTA)** que é fornecida pelo container EJB. Também é utilizada a tag `jta-data-source` para informar a fonte do pool de conexões (nome JNDI). Exemplo de **persistence.xml** para aplicações Java EE:

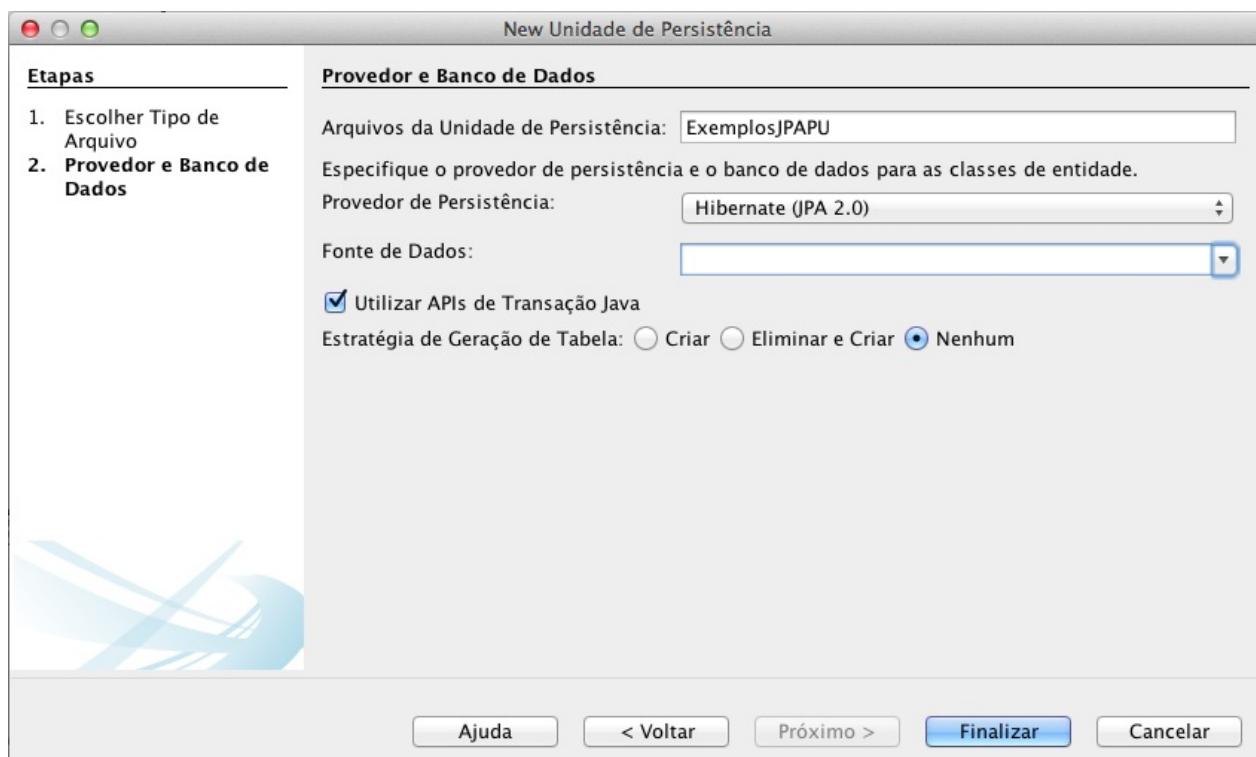
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="ExemploJPAPU" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>jdbc/ExemplosJPA</jta-data-source>
        <properties>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Criando uma unidade de persistência no NetBeans

O NetBeans possui uma forma mais simples de criação da unidade de persistência. Clique com o botão direito no projeto e selecione a opção **Novo -> Outro...**, na tela de **Novo** arquivo selecione a categoria **Persistence** e o tipo de arquivo **Unidade de Persistência**, conforme a seguir:

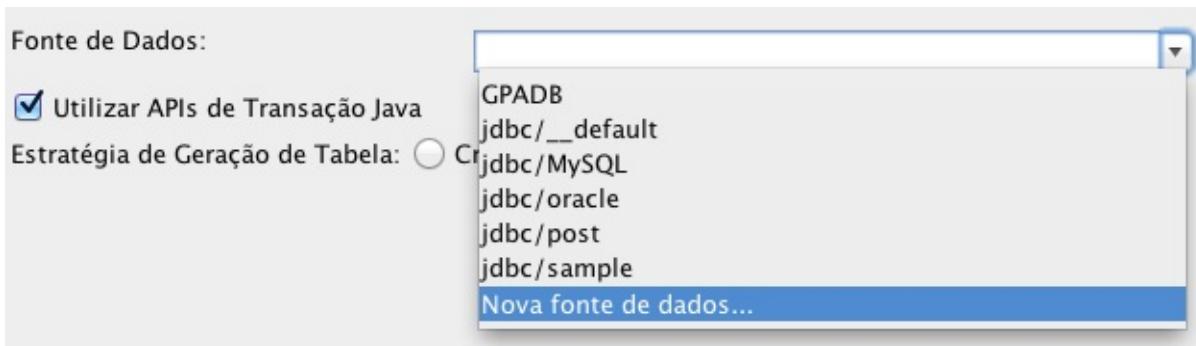


Clique em **Próximo >** para definir as propriedades do banco de dados. Na tela de **Provedor e banco de dados** digite um nome para a unidade de persistência, escolha qual a biblioteca de persistência, defina a conexão com o banco de dados e qual a estratégia para geração de tabelas, conforme a figura a seguir:

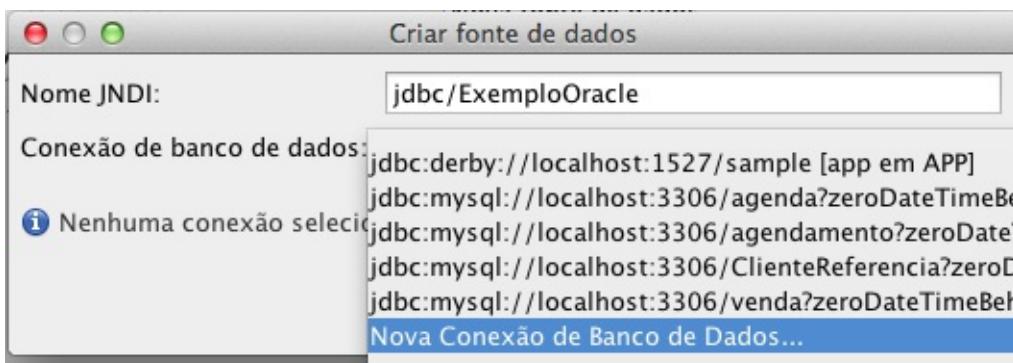


- O **Arquivos da Unidade de Persistência** é utilizado quando a EntityManager é criada. Por meio deste nome o EntityManager encontrará as configurações do banco de dados.
- O **Provedor de Persistência** é a implementação do JPA utilizada para acessar o banco de dados; neste exemplo é utilizado o framework **Hibernate (JPA 2.0)**.
- **Fonte de dados** é a forma como será realizada a conexão com o banco de dados. Pode ser feito de duas maneiras, criada uma conexão direta na aplicação ou criado um pool de conexões no servidor.

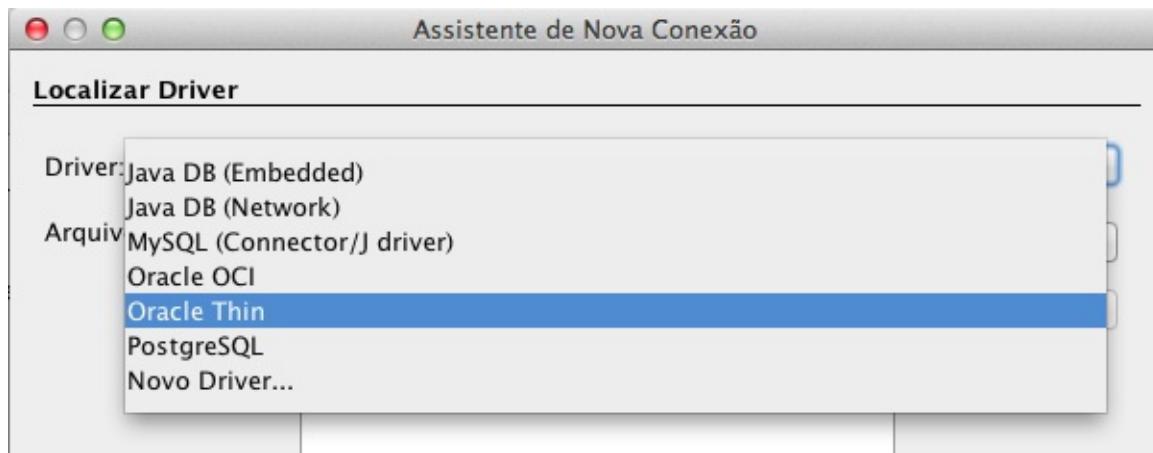
Para criar uma nova conexão direta na aplicação clique na caixa de seleção **Fonte de Dados** e escolha a opção **Nova fonte de dados...**, conforme apresentado na figura a seguir:



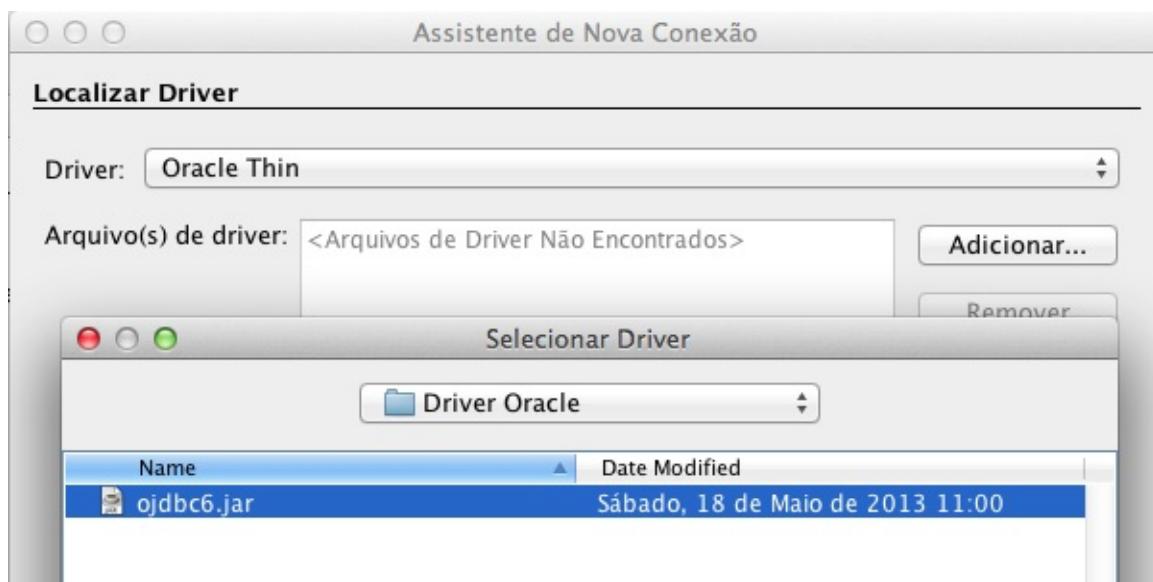
Na tela de **Criar fonte de dados** informe um **Nome JNDI**. Esse nome é usado para referenciar a conexão e também uma **Conexão de banco de dados**. Caso ainda não tenha uma conexão criada, escolha na caixa de seleção Conexão de banco de dados a opção **Nova Conexão de Banco de Dados...**, conforme apresentado na figura a seguir:



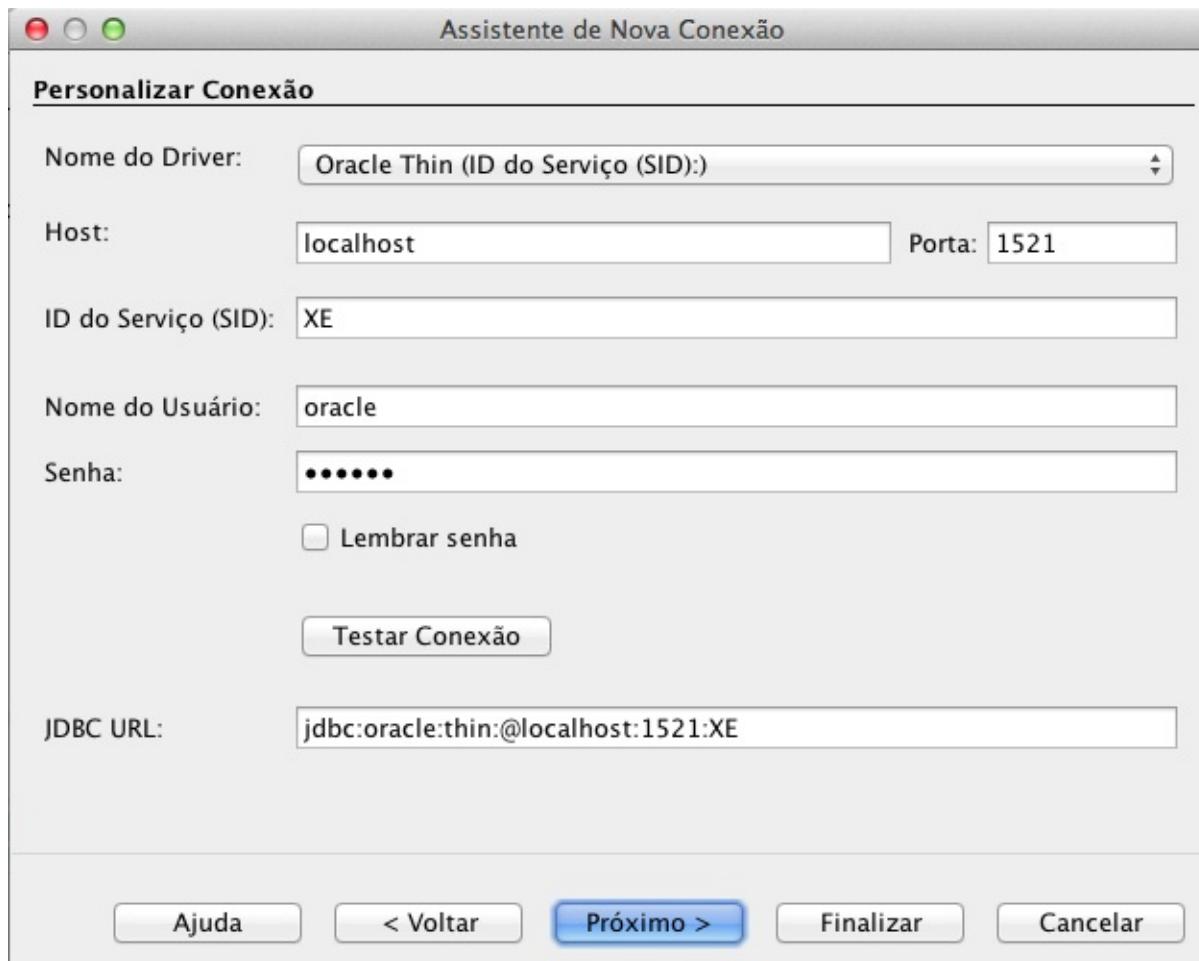
Na tela **Assistente de Nova Conexão** escolha o **Driver** chamado **Oracle Thin**, conforme apresentado na figura a seguir:



Na primeira vez que ele é usado é preciso especificar o driver do banco de dados Oracle. Clique em Adicionar e escolha o driver **ojdbc6.jar**, conforme a figura a seguir:

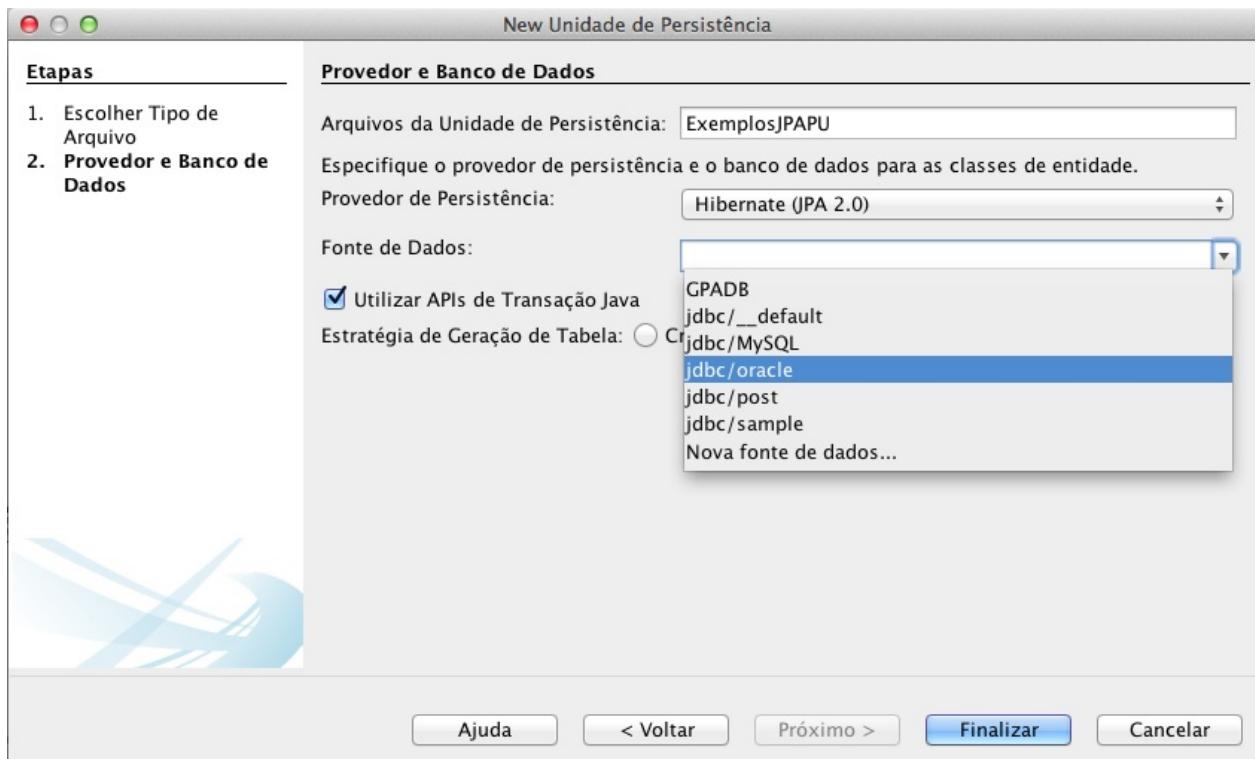


Clique em Próximo para continuar a criação da conexão. Continuando a criação da Nova Conexão escolha o **Nome do Driver**, defina o **Host (IP)**, **Porta**, **ID do Serviço**, **Nome do usuário** e **Senha**, conforme o exemplo da figura a seguir:



Clique em **Testar Conexão** caso queira validar se as informações da conexão estão corretas, depois clique em **Próximo** para continuar e depois clique em **Finalizar** para terminar a criação da conexão.

Durante o desenvolvimento de uma aplicação web que acessa banco de dados, pode ser utilizado um pool de conexões que é criado pelo próprio servidor de aplicações web, ao invés de criar uma **conexão com o banco de dados** direto na aplicação é necessário criar um pool de conexões no servidor e depois escolher na **Fonte de dados**, conforme a figura a seguir:

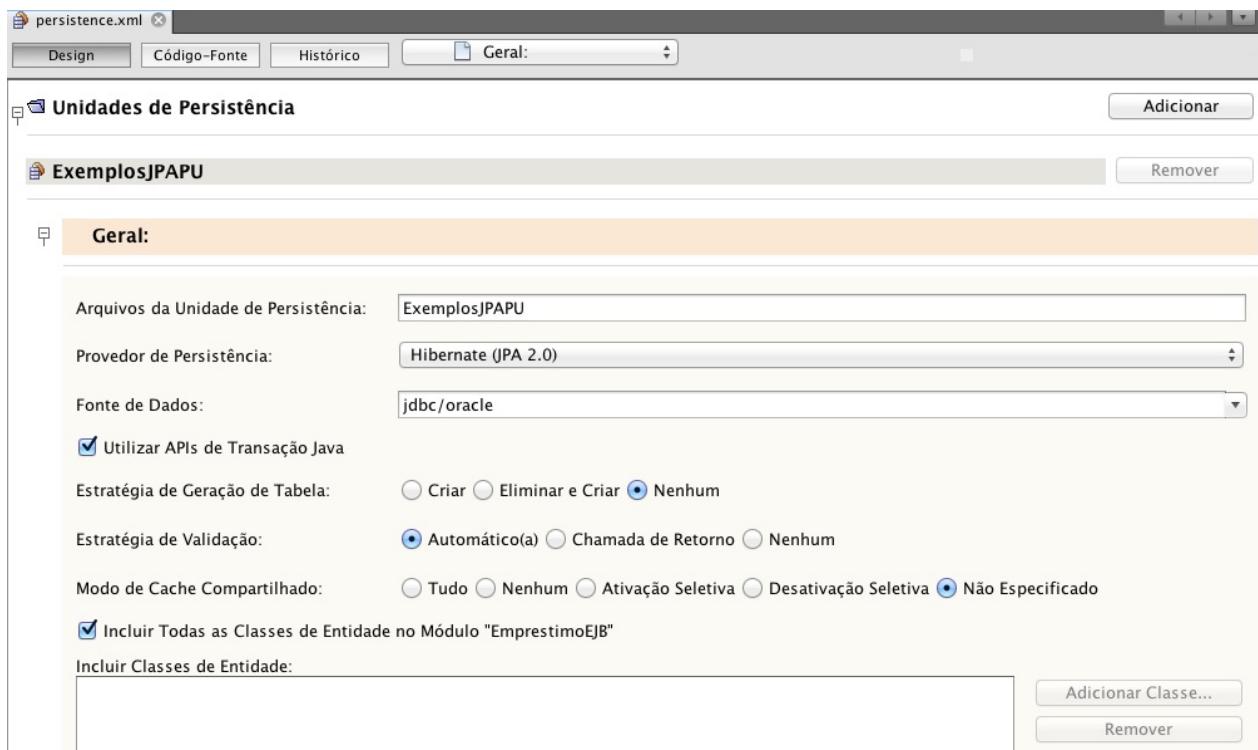


Neste exemplo será escolhida uma Fonte de Dados que foi criada no servidor GlassFish.

A Estratégia de geração de tabelas permite:

- Criar - o JPA cria a estrutura de tabelas no banco de dados;
- Apagar e criar – o JPA apaga a estrutura existente e cria uma estrutura nova das tabelas do banco de dados;
- Nenhum – criar manualmente as tabelas do banco de dados.

Depois de criada a unidade de persistência, conforme apresentado na figura a seguir, note que esta é uma versão visual do arquivo persistence.xml.



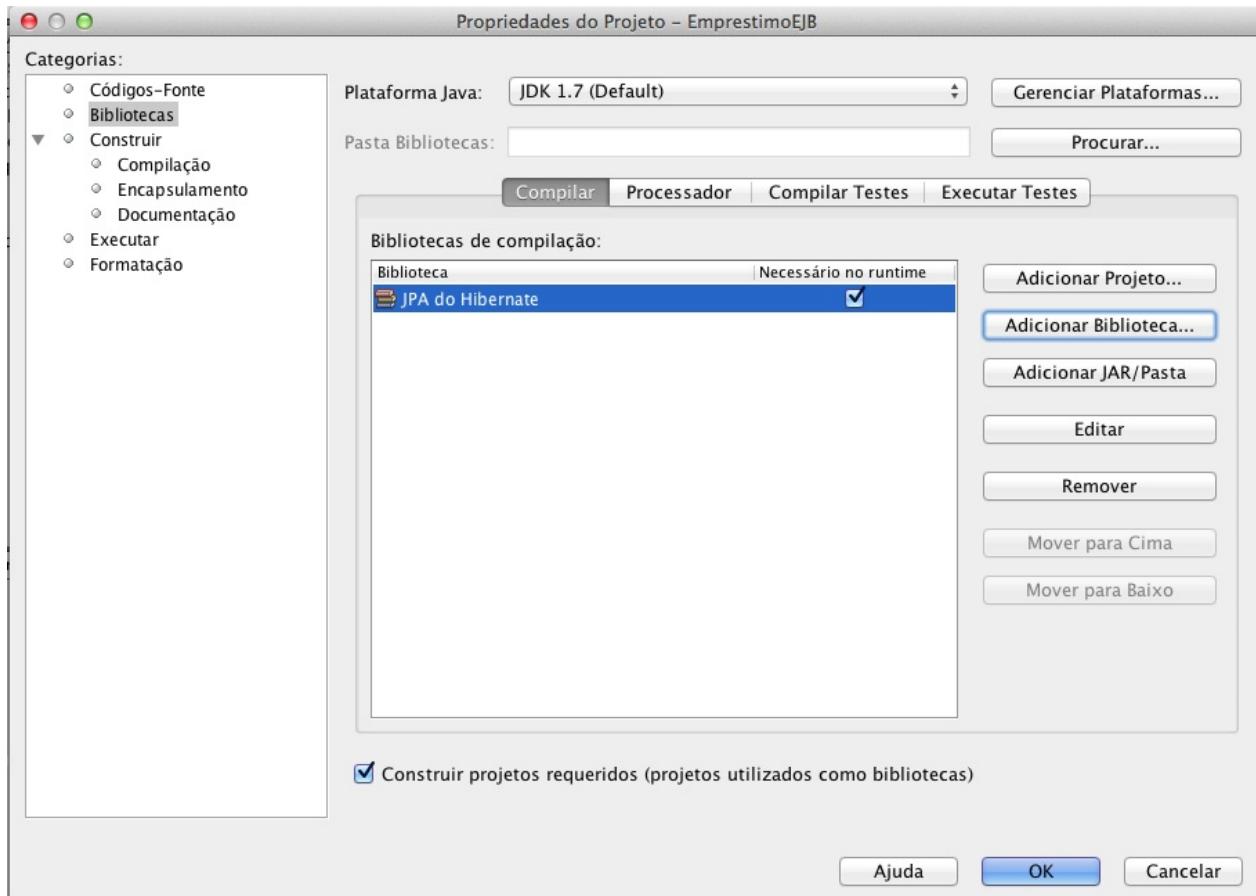
Este é o arquivo **persistence.xml** criado na pasta **META-INF** do projeto:

```
<?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="ExemplosJPAPU" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>jdbc/Oracle</jta-data-source>
        <properties>
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.Oracle9Dialect"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Para utilizar o Hibernate é necessário adicionar suas bibliotecas no projeto. Para fazer isso clique com o botão direito no projeto e selecione a opção

Propriedades, nesta tela selecione a categoria **Bibliotecas**, após isso clique no

botão **Adicionar biblioteca...** e adicione as bibliotecas JPA do Hibernate, conforme a figura a seguir:



EntityManager gerenciado pela Aplicação

Nas aplicações Java SE é necessário controlar pela aplicação como deve ser criado o EntityManager, para isso precisamos fazer os seguintes passos:

1) Utilizando a classe **javax.persistence.Persistence** podemos utilizar o método **createEntityManagerFactory()** que recebe como parâmetro o nome da unidade de persistência (que veremos mais adiante) que contem as informações sobre o banco de dados para criar uma **javax.persistence.EntityManagerFactory**.

```
EntityManagerFactory factory =  
    Persistence.createEntityManagerFactory("UnitName");
```

2) Através do **javax.persistence.EntityManagerFactory** podemos utilizar o método **createEntityManager()** para obtermos uma EntityManager.

```
EntityManager entityManager = factory.createEntityManager();
```

Exemplo:

```
package pbc.jpa.exemplo.dao;

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

/**
 * Classe utilizada para fazer realizar as operações
 * de banco de dados sobre a entity Livro.
 */
public class DAO {
    private EntityManager entityManager;

    public EntityManager getEntityManager() {
        EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("ExemplosJPAPU");

        entityManager = factory.createEntityManager();
        factory.close();
    }
}
```

No exemplo dentro do método **getEntityManager()** criamos um **EntityManagerFactory** da unidade de persistência **ExemplosJPAPU** (criada no item 1.2.3) através da classe **Persistence**, depois criamos um **EntityManager** a partir da factory. Quando terminamos de utilizar a **EntityManagerFactory** é importante chamar o método **close()** para liberar os recursos da factory.

EntityManager gerenciado pelo Container

No desenvolvimento de aplicações Java EE é possível deixar o contêiner EJB injetar a unidade de persistência através da anotação **javax.persistence.PersistenceContext**.

```
package pbc.jpa.exemplo.ejb;

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

/**
 * EJB utilizado para demonstrar o uso da injeção de
 * dependência do EntityManager.
 */
@Stateless
public class ExemploBean implements ExemploRemote {

    @PersistenceContext(unitName = "ExercicioJPA1PU")
    private EntityManager entityManager;

}
```

Este exemplo mostra um componente Session Bean Stateless que utiliza a anotação **@PersistenceContext** para adicionar uma unidade de persistência no EntityManager. Repare que não é preciso criar manualmente a EntityManager, pois o container EJB se encarrega de fazer isso e também atribui seu objeto para a variável com a anotação. A anotação **@PersistenceContext** possui o atributo **unitName** para informar que o nome da unidade de persistência é **ExemplosJPAPU**, definido na tag `persistence-unit` do arquivo **persistence.xml**.

Interface EntityManager

A interface **javax.persistence.EntityManager** possui a assinatura de métodos manipular as entidades, executar consultas e outros:

```
public void persist(Object entity);
```

Faz uma nova instância gerenciável e persistível.

```
entityManager.persist(pessoa);
```

Neste exemplo o método **persist** da EntityManager salva a entidade Pessoa no banco de dados.

```
public <T> T persist(T entity);
```

Junta o estado da Entity com o estado persistido, por exemplo:

```
entityManager.merge(pessoa);
```

Neste exemplo o método **merge** da EntityManager atualiza a entidade Pessoa no banco de dados.

```
public void merge(Object entity);
```

Remove a instância da Entity do banco de dados, por exemplo:

```
entityManager.remove(pessoa);
```

Neste exemplo o método **remove** da EntityManager exclui a entidade Pessoa no banco de dados.

```
public boolean remove(Object entity);
```

Verifica se a instância da Entity está em um estado persistível.

```
public <T> T find(Class<T> entityClass, Object primaryKey);
```

Procura um registro no banco de dados através da Entity e id (chave primária) da tabela, caso não encontre retorna null, por exemplo:

```
Pessoa pessoa = entityManager.find(Pessoa.class, id);
```

Neste exemplo o método **find** da EntityManager pesquisa uma entidade pela sua classe e a chave primária.

EntityManager

```
public <T> T getReference(Class<T> entityClass, Object primaryKey);
```

Procura um registro no banco de dados através da Entity e id da tabela, caso não encontre retorna uma exceção **javax.persistence.EntityNotFoundException**.

```
public void flush();
```

Os métodos persist(), merge() e remove() aguardam a finalização da transação para sincronizar as entidades com o banco de dados, o método **flush()** força esta sincronização no momento em que é chamado.

```
public void refresh(Object entity);
```

Verifica se houve alguma alteração no banco de dados para sincronizar com a entidade.

```
public void clear();
```

Remove as entidades que estão no estado gerenciável dentro da EntityManager.

```
public void close();
```

Fecha a conexão do EntityManager.

```
public boolean isOpen();
```

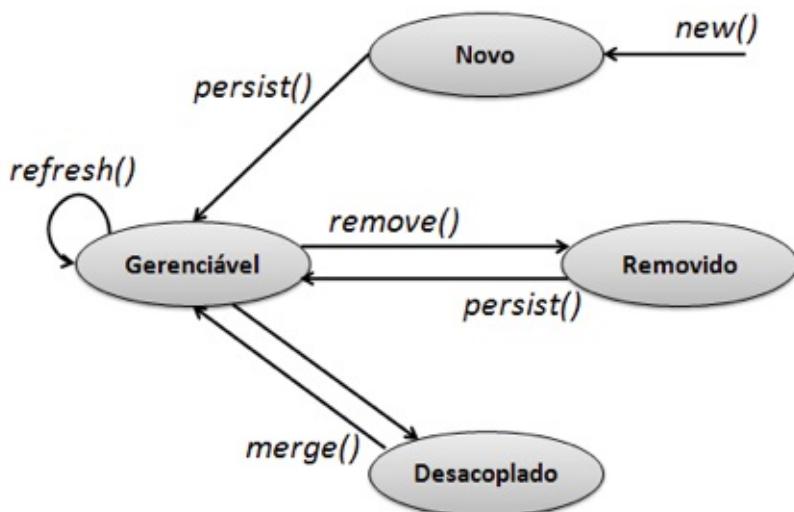
Verifica se o EntityManager está com a conexão aberta.

```
public EntityTransaction getTransaction();
```

Obtém uma **javax.persistence.EntityTransaction** que é uma transação com o banco de dados.

Ciclo de vida da Entity

Uma entidade do banco de dados pode passar por quatro estados diferentes que fazem parte do seu ciclo de vida: novo, gerenciável, desacoplado e removido, conforme a figura a seguir:



Ciclo de vida da entidade adaptado de BROSE, G.; SILVERMAN, M.; SRIGANESH, R. P. (2006, p.180)

Novo (new)

A Entity foi criada, mas ainda não foi persistida no banco de dados. Mudanças no estado da Entity não são sincronizadas com o banco de dados.

Gerenciado (managed)

A Entity foi persistida no banco de dados e encontra-se em um estado gerenciável. Mudanças no estado da Entity são sincronizadas com o banco de dados assim que uma transação for finalizada com sucesso.

Removido (removed)

A Entity foi agendada para ser removida da base de dados, esta entidade será removida fisicamente do banco de dados quando a transação for finalizada com sucesso.

Desacoplado (detached)

A Entity foi persistida no banco de dados, mas encontra-se em um estado que não está associada ao contexto persistível, portanto as alterações em seu estado não são refletidas na base de dados.

Exemplo de DAO (CRUD) utilizando JPA

Nesse exemplo, vamos criar uma aplicativo Java (console) chamado **Exemplo**, para salvar, alterar, consultar por id e apagar um registro de pessoa, utilizaremos o banco de dados Oracle.

Observação: Lembre de adicionar as bibliotecas do Hibernate JPA e Oracle (ojdbc6.jar) no seu projeto.

Vamos criar uma Entity para representar a tabela Pessoa a seguir:

```
CREATE SEQUENCE PESSOA_SEQ INCREMENT BY 1
    START WITH 1 NOCACHE NOCYCLE;

CREATE TABLE Pessoa (
    id      NUMBER(5)      NOT NULL,
    nome   VARCHAR2(100) NOT NULL,
    dataNasc DATE          NOT NULL,
    email   VARCHAR2(150),
    PRIMARY KEY(ID)
);
```

Criando a entity para representar a tabela Pessoa:

```
package pbc.jpa.exemplo1.modelo;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
```

```
/**  
 * Classe utilizada para representar uma pessoa.  
 */  
  
@Entity  
@SequenceGenerator(name="PES_SEQ",  
    sequenceName="PESSOA_SEQ", allocationSize=1,  
    initialValue=1)  
public class Pessoa implements Serializable {  
    private static final long serialVersionUID = 1L;  
  
    @Id  
    @GeneratedValue(strategy=GenerationType.SEQUENCE,  
        generator="PES_SEQ")  
    private Long id;  
    @Column(nullable = false)  
    private String nome;  
    @Temporal(TemporalType.DATE)  
    @Column(name = "dataNasc", nullable = false)  
    private Date dataNascimento;  
    private String email;  
  
    public Date getDataNascimento() {  
        return dataNascimento;  
    }  
    public void setDataNascimento(Date dataNascimento) {  
        this.dataNascimento = dataNascimento;  
    }  
  
    public String getEmail() { return email; }  
    public void setEmail(String email) {  
        this.email = email;  
    }  
  
    public Long getId() { return id; }  
    public void setId(Long id) { this.id = id; }  
  
    public String getNome() { return nome; }  
    public void setNome(String nome) { this.nome = nome; }  
}
```

Vamos criar o arquivo persistence.xml, O NetBeans possui um wizard que permite de forma fácil a criação desta unidade de persistência, para isto, clique com o botão direito do mouse no nome do projeto, selecione a opção **Novo -> Outro...**, na tela de **Novo arquivo** selecione a categoria **Persistence** e o tipo de arquivo **Unidade de Persistência**.

Na tela de Novo Unidade de persistência, informe:

- **Nome da unidade de persistência:** ExemplosJPAPU
- **Biblioteca de persistência:** Hibernate JPA (1.0)
- **Conexão com o banco de dados:** Aqui vamos criar uma nova conexão com o Oracle, então selecione a opção **Nova conexão com banco de dados...**

Na tela **Nova Conexão do Banco de Dados**, selecione no campo **Nome do driver** a opção **Novo driver...**

Na tela de **Novo driver JDBC**, clique em **Adicionar...** e procure pelo arquivo com o driver do Oracle, neste caso o arquivo ojdbc6.jar. Após isso clique em **OK** para prosseguir.

Voltando na tela **Nova Conexão do Banco de Dados**, configure as seguintes informações:

- **Modo de entrada de dados:** Entrada direta de URL
- **Nome do usuário:** Informe seu usuário
- **Senha do usuário:** Informe sua senha
- **URL JDBC:** Para conectar no banco de dados utilize a URL
`jdbc:oracle:thin:@localhost:1521:XE` ou outra de acordo com o local que foi instalado o seu banco de dados.

Clique em **OK** para prosseguir, a conexão com o banco de dados será testada e apresentará a mensagem na tela “Conexão estabelecida”. Clique em **OK** para prosseguir.

Ao clicar em **Finalizar** será gerado 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="ExemplosJPAPU"
        transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>pbc.jpa.exemplo1.modelo.Pessoa</class>
        <properties>
            <property name="hibernate.connection.username"
                value="usuario"/>
            <property name="hibernate.connection.password"
                value="senha"/>
            <property name="hibernate.connection.driver_class"
                value="oracle.jdbc.driver.OracleDriver"/>
            <property name="hibernate.connection.url"
                value="jdbc:oracle:thin:@localhost:1521:XE"/>
            <property name="hibernate.cache.provider_class"
                value="org.hibernate.cache.NoCacheProvider"/>
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.Oracle9Dialect"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Vamos criar uma classe PessoaDAO que possui os métodos para manipular (salvar, atualizar, apagar e consultar por id) um objeto Pessoa.

```
package pbc.jpa.exemplo.dao;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import pbc.jpa.exemplo.modelo.Pessoa;

/**
```

```
* Classe utilizada para fazer realizar as operações de
* banco de dados sobre a entity Pessoa.
*/
public class PessoaDAO {
    /**
     * Método utilizado para obter o entity manager.
     * @return
     */
    private EntityManager getEntityManager() {
        EntityManagerFactory factory = null;
        EntityManager entityManager = null;
        try {
            //Obtém o factory a partir da unidade de persistência.
            factory = Persistence.createEntityManagerFactory
                ("ExemplosJPAPU");
            //Cria um entity manager.
            entityManager = factory.createEntityManager();
            //Fecha o factory para liberar os recursos utilizado.
        } finally {
            factory.close();
        }
        return entityManager;
    }

    /**
     * Método utilizado para salvar ou atualizar as
     * informações de uma pessoa.
     * @param pessoa
     * @return
     * @throws java.lang.Exception
     */
    public Pessoa salvar(Pessoa pessoa) throws Exception {
        EntityManager entityManager = getEntityManager();
        try {
            // Inicia uma transação com o banco de dados.
            entityManager.getTransaction().begin();
            System.out.println("Salvando a pessoa.");
            /* Verifica se a pessoa ainda não está salva
             no banco de dados. */
            if(pessoa.getId() == null) {
```

```
        entityManager.persist(pessoa);
    } else {
        pessoa = entityManager.merge(pessoa);
    }
    // Finaliza a transação.
    entityManager.getTransaction().commit();
} finally {
    entityManager.close();
}
return pessoa;
}

/**
 * Método que apaga a pessoa do banco de dados.
 * @param id
 */
public void excluir(Long id) {
    EntityManager entityManager = getEntityManager();
    try {
        // Inicia uma transação com o banco de dados.
        entityManager.getTransaction().begin();
        /* Consulta a pessoa na base de dados através
         do seu ID. */
        Pessoa pessoa = consultarPorId(id);
        System.out.println("Excluindo a pessoa: "
            + pessoa.getNome());

        // Remove a pessoa da base de dados.
        entityManager.remove(pessoa);
        // Finaliza a transação.
        entityManager.getTransaction().commit();
    } finally {
        entityManager.close();
    }
}

/**
 * Consulta o pessoa pelo ID.
 * @param id
 * @return o objeto Pessoa.

```

```
/*
public Pessoa consultarPorId(Long id) {
    EntityManager entityManager = getEntityManager();
    Pessoa pessoa = null;
    try {
        pessoa = entityManager.find(Pessoa.class, id);
    } finally {
        entityManager.close();
    }
    return pessoa;
}
```

O método **salvar** recebe o objeto **Pessoa** que será salvo, neste exemplo usaremos este método para salvar uma nova pessoa ou atualizar os dados de uma pessoa.

Mas como sabemos quando temos que salvar e quando tem que atualizar, basta olhar o atributo **id** da classe **Pessoa**, se o **id** for **null** significa que é um novo objeto que ainda não foi salvo no banco de dados, então utilizaremos o método **persist** da **EntityManager** para salva-lo, caso o **id** tenha algum valor então significa que o objeto já foi salvo anteriormente portanto ele deve ser atualizado então utilizaremos o método **merge** da **EntityManager** para atualiza-lo.

Note que como vamos salvar ou atualizar os dados, precisamos criar uma transação, com o método **getTransaction()** do **EntityManager** obtemos um objeto **EntityTransaction** com ele podemos iniciar a transação através do método **begin()**, finalizar a transação com sucesso através do método **commit()** ou desfazer as alterações em caso de erro com o método **rollback()**. Este mesmo conceito de transação será utilizado no método excluir.

O método **excluir** não precisa receber todos os dados da **Pessoa**, recebendo apenas o seu ID através do parâmetro **Long id**, podemos utilizar o método **find** do **EntityManager** para consultar os dados da **Pessoa**, depois com o objeto **Pessoa** consultado podemos usar o método **remove** do **EntityManager** para apagar os dados da **Pessoa**.

O método **consultarPorId** recebe um objeto **Long** chamado **id**, com o ID da tabela Pessoa, utilizando o método **find** do **EntityManager** passamos a classe da entidade **Pessoa.class** e seu **id** para que possamos consultar os dados da Pessoa.

Vamos criar uma classe **PessoaDAOTeste** para testarmos os métodos da classe **PessoaDAO**:

```
package pbc.jpa.exemplo1.teste;

import java.util.Calendar;
import java.util.GregorianCalendar;
import pbc.jpa.exemplo1.dao.PessoaDAO;
import pbc.jpa.exemplo1.modelo.Pessoa;

/**
 * Classe utilizada para testar os métodos do PessoaDAO.
 */
public class PessoaDAOTeste {
    public static void main(String[] args) throws Exception {
        Pessoa pessoa = new Pessoa();
        pessoa.setId(1L);
        pessoa.setNome("Rafael Sakurai");
        Calendar data = new GregorianCalendar();
        data.set(Calendar.YEAR, 1983);
        data.set(Calendar.MONTH, 11);
        data.set(Calendar.DAY_OF_MONTH, 26);
        pessoa.setDataNascimento(data.getTime());
        pessoa.setEmail("rafael.sakurai@metodista.br");

        PessoaDAO dao = new PessoaDAO();
        System.out.println("Salvando a pessoa: "
            + pessoa.getNome());
        pessoa = dao.salvar(pessoa);

        pessoa.setNome("Rafael Guimarães Sakurai");
        pessoa = dao.salvar(pessoa);
        System.out.println("Alterando a pessoa: "
            + pessoa.getNome());
    }
}
```

```
Pessoa pessoa2 = dao.consultarPorId(pessoa.getId());
System.out.println("Consultando: " + pessoa2.getNome());

System.out.println("Removendo a pessoa: "
    + pessoa.getId());
dao.excluir(pessoa.getId());
}

}
```

Neste teste vamos criar um objeto pessoa e salva-lo, depois vamos altera o nome da pessoa, vamos consultar a pessoa pelo id e no final vamos apagar o registro da pessoa.

Estratégia de SEQUENCE para gerar ID

O banco de dados Oracle possui um tipo de objeto que pode ser criado internamente chamado SEQUENCE (sequência) e que pode ser associado a um contador. Cada vez que o seu próximo valor é solicitado ocorre um incremento deste valor. Para gerar o ID da entidade é possível utilizar a estratégia de SEQUENCE também, associando a sequência criada no banco de dados com o gerador de ID.

No exemplo, a seguir, será criada uma entidade que utiliza uma SEQUENCE para gerar o id referente a chave primária da tabela. Primeiro será criada a sequência USUARIO_SEQ no banco de dados:

```
CREATE SEQUENCE USUARIO_SEQ INCREMENT BY 1 START WITH 1 NOCACHE  
NOCYCLE;
```

A seguir, crie a entidade Usuario que utilizará esta sequência para gerar o ID.

```
package pbc.jpa.exemplo.sequence.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

/**
 * Classe utilizada para demonstrar o uso da SEQUENCE.
 */
@Entity
@Table(name = "USUARIO")
@SequenceGenerator(name="USU_SEQ", sequenceName="USUARIO_SEQ",
    initialValue=1, allocationSize=1)
public class Usuario implements Serializable {
    private static final long serialVersionUID
        = -4023522856316087762L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "USU_SEQ")
    private Long id;
    private String nome;

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

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

A anotação **@SequenceGenerator** vai indicar qual a sequência do banco de dados, qual o valor inicial e a quantidade que esta sequência é incrementada.

- **name** = informa o apelido da sequence exemplo "USU_SEQ";
- **sequenceName** = informa o nome da sequência criada no banco de dados,

por exemplo: "USUARIO_SEQ";

- **initialValue** = informa o valor inicial da sequência, por exemplo: 1;
- **allocationSize** = informa a quantidade que será incrementada na sequência, o padrão (default) é 50, neste exemplo foi definido que será incrementado de 1 em 1.

A anotação **@GeneratedValue** nesse caso está usando a estratégia de SEQUENCE então ele vai procurar a sequência que tem o apelido "USU_SEQ" e faz um consulta no banco para pegar seu resultado:

```
SELECT USUARIO_SEQ.NEXTVAL FROM DUAL;
```

- **strategy** = informa o tipo de estratégia que será utilizado nesse exemplo GenerationType.SEQUENCE;
- **generator** = informa o apelido da sequence nesse exemplo "USU_SEQ".

Utilizando chave composta

A chave composta define que vários atributos serão utilizados para definir a chave de uma entidade, com isso, acabamos tendo uma restrição onde os atributos da chave composta não podem ser repetidos.

No JPA quando precisamos definir uma chave composta precisamos criar uma classe separada apenas com os atributos que fazem parte da chave composta e precisamos utilizar a anotação **javax.persistence.Embeddable**.

Neste exemplo vamos criar uma entidade Telefone que possui uma chave composta pelos atributos ddd e número do telefone, pois não pode ter o mesmo número de telefone para mais de 1 cliente.

Para declarar a chave composta vamos criar uma classe chamada TelefonePK com os atributos ddd e numero:

```
package pbc.jpa.exemplo.chave.composta.modelo;

import java.io.Serializable;
import javax.persistence.Embeddable;

/**
 * Esta classe representa a composição da chave do
 * telefone, esta chave é composta por ddd + numero
 * do telefone.
 */
@Embeddable
public class TelefonePK implements Serializable {
    private static final long serialVersionUID
        = -637018809489152388L;

    private Short ddd;
    private String numero;

    public Short getDdd() { return ddd; }
    public void setDdd(Short ddd) { this.ddd = ddd; }

    public String getNumero() { return numero; }
    public void setNumero(String numero) {
        this.numero = numero;
    }

    @Override
    public String toString() {
        return "("+ getDdd() + ") " + getNumero();
    }
}
```

Observação: Sobrescrevi o método `toString()` para imprimir de forma mais amigável a chave composta.

Note que adicionamos a anotação **@Embeddable** na classe `TelefonePK` para informar que está classe será adicionado em outra entidade.

Para adicionarmos a chave composta na entidade, vamos criar um atributo do tipo da classe que possui a anotação `@Embeddable` e vamos adicionar a anotação `javax.persistence.EmbeddedId` neste atributo.

Na entidade Telefone vamos declarar um atributo chamado `id` do tipo `TelefonePK` para representar a chave composta:

```
package pbc.jpa.exemplo.chave.composta.modelo;

import java.io.Serializable;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;

/**
 * Classe utilizada para representar o telefone de um cliente.
 * Nesta classe abordamos o uso da chave composta através
 * da anotação @EmbeddedId.
 */
@Entity
public class Telefone implements Serializable {
    private static final long serialVersionUID
        = 5999236902534007386L;

    @EmbeddedId
    private TelefonePK id;
    private String cliente;

    public String getCliente() { return cliente; }
    public void setCliente(String cliente) {
        this.cliente = cliente;
    }

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

Agora vamos criar a classe `TelefoneDAO` para executar as operações de Salvar, Alterar, Consultar por chave composta e apagar o telefone:

```
package pbc.jpa.exemplo.chave.composta.dao;

import javax.persistence.EntityExistsException;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import pbc.jpa.exemplo.chave.composta.modelo.Telefone;
import pbc.jpa.exemplo.chave.composta.modelo.TelefonePK;

/**
 * Classe utilizada para testar as operações Salvar,
 * Altera, Consultar por Id e Apagar o registro de
 * um telefone.
 */
public class TelefoneDAO {
    public EntityManager getEntityManager() {
        EntityManagerFactory factory = null;
        EntityManager entityManager = null;

        try {
            factory = Persistence.createEntityManagerFactory
                ("ExemplosJPAPU");
            entityManager = factory.createEntityManager();
        } finally {
            factory.close();
        }

        return entityManager;
    }

    public Telefone consultarPorId(TelefonePK id) {
        EntityManager entityManager = getEntityManager();
        Telefone telefone = null;

        try {
            telefone = entityManager.find(Telefone.class, id);
        } finally {
            entityManager.close();
        }
    }
}
```

```
        return telefone;
    }

public Telefone salvar(Telefone telefone)
    throws Exception {
    EntityManager entityManager = getEntityManager();

    try {
        entityManager.getTransaction().begin();
        entityManager.persist(telefone);
        entityManager.flush();
        entityManager.getTransaction().commit();
        /*Esta exceção pode ser lançada caso já exista um registro
         com a mesma chave composta. */
    } catch (EntityExistsException ex) {
        entityManager.getTransaction().rollback();
        throw new Exception("Este telefone já está registrado
            para outro cliente.");
    } catch (Exception ex) {
        entityManager.getTransaction().rollback();
    } finally {
        entityManager.close();
    }

    return telefone;
}

public Telefone atualizar(Telefone telefone)
    throws Exception {
    EntityManager entityManager = getEntityManager();

    try {
        entityManager.getTransaction().begin();
        entityManager.merge(telefone);
        entityManager.flush();
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        entityManager.getTransaction().rollback();
    } finally {
```

```
        entityManager.close();
    }

    return telefone;
}

public void apagar(TelefonePK id) {
    EntityManager entityManager = getEntityManager();

    try {
        entityManager.getTransaction().begin();
        Telefone telefone = entityManager.find(Telefone.class,
            id);
        entityManager.remove(telefone);
        entityManager.flush();
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        entityManager.getTransaction().rollback();
    } finally {
        entityManager.close();
    }
}
```

No método **consultarPorId** precisamos agora passar um atributo da chave composta **TelefonePK** para que possamos localizar um telefone.

No método **salvar** vamos receber o objeto Telefone que será salvo, note que estamos tratando a exceção **javax.persistence.EntityExistsException** que será lançada caso tentamos salvar duas entidades com o mesmo id, neste caso com o mesmo ddd e número de telefone.

Criamos agora um método **atualizar** separado apenas para atualizar o telefone, pois podemos criar um Telefone e alterar o nome do cliente, mas não podemos criar dois telefones com o mesmo ddd e número de telefone.

No método **apagar** precisamos receber um objeto que representa a chave composta TelefonePK, desta forma podemos localizar o objeto Telefone para que possamos apagá-lo.

Nesta classe **TelefoneDAO**Teste vamos testar todas as operações da classe TelefoneDAO declarada acima:

```
package pbc.jpa.exemplo.chave.composta.dao;

import pbc.jpa.exemplo.chave.composta.modelo.Telefone;
import pbc.jpa.exemplo.chave.composta.modelo.TelefonePK;

/**
 * Classe utilizada para testar as operações da
 * classe TelefoneDAO.
 */
public class TelefoneDAOTeste {
    public static void main(String[] args) throws Exception {
        /* Cria uma chave composta. */
        TelefonePK pk = new TelefonePK();
        pk.setDdd((short) 11);
        pk.setNumero("1111-1111");

        /* Cria um telefone. */
        Telefone tel = new Telefone();
        tel.setId(pk);
        tel.setCliente("Sakurai");

        TelefoneDAO dao = new TelefoneDAO();
        System.out.println("Salvando o telefone: " + pk);
        dao.salvar(tel);

        System.out.println("Consultando o telefone: " + pk);
        Telefone tel2 = dao.consultarPorId(pk);
        System.out.println("Cliente " + tel2.getCliente());

        try {
            System.out.println("Tentando salvar o mesmo número
                de telefone para outro cliente.");
            Telefone tel3 = new Telefone();
            tel3.setId(pk);
            tel3.setCliente("Rafael");
            dao.salvar(tel3);
        } catch (Exception ex) {
```

Utilizando chave composta

```
        System.out.println(ex.getMessage());
    }

    System.out.println("Alterando o cliente:");
    tel.setCliente("Rafael");
    dao.atualizar(tel);
    System.out.println("Apagando o registro do telefone:");
    dao.apagar(pk);
}

}
```

Exercício 1

Neste exercício vamos abordar como criar uma aplicação CRUD (salvar, alterar, consultar e excluir) do Livro utilizando o Java Persistence API.

Crie o seguinte banco de dados:

```
CREATE SEQUENCE LIVRO_SEQ INCREMENT BY 1
    START WITH 1 NOCACHE NOCYCLE;

CREATE TABLE Livro (
    id      number(5)      NOT NULL,
    titulo  varchar2(200)   NOT NULL,
    autor   varchar2(200)   NOT NULL,
    isbn    varchar2(50)    NOT NULL,
    paginas number(5)      NOT NULL,
    preco   number(10,2)    NOT NULL,
    PRIMARY KEY(id)
);
```

Crie um Projeto Java chamado **ExercicioJPA1**, adicione as bibliotecas **EclipseLink JPA** e Driver da Oracle **ojdbc7.jar** e crie:

- Uma classe entity para representar um Livro com os atributos id, titulo, autor, isbn, paginas e preco.

```
package pbc.jpa.exercicio1.modelo;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.SequenceGenerator;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
```

```
/**  
 * Classe utilizada para representar uma Entity Livro.  
 * @author Rafael Guimarães Sakurai  
 */  
  
@Entity  
@SequenceGenerator(name = "LIVRO_SEQ",  
    sequenceName = "LIVRO_SEQ", initialValue = 1,  
    allocationSize = 1)  
public class Livro implements Serializable {  
    private static final long serialVersionUID  
        = 2405106626392673061L;  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.SEQUENCE,  
        generator = "LIVRO_SEQ")  
    @Column(nullable = false)  
    private Long id;  
    @Column(nullable = false)  
    private String titulo;  
    @Column(nullable = false)  
    private String autor;  
    @Column(nullable = false)  
    private String isbn;  
    @Column(nullable = false)  
    private Integer paginas;  
    @Column(nullable = false)  
    private Double preco;  
  
    public String getAutor() { return autor; }  
    public void setAutor(String autor) {  
        this.autor = autor;  
    }  
  
    public Long getId() { return id; }  
    public void setId(Long id) { this.id = id; }  
  
    public String getIsbn() { return isbn; }  
    public void setIsbn(String isbn) { this.isbn = isbn; }  
  
    public Integer getPaginas() { return paginas; }
```

```
public void setPaginas(Integer paginas) {
    this.paginas = paginas;
}

public Double getPreco() { return preco; }
public void setPreco(Double preco) {
    this.preco = preco;
}

public String getTitulo() { return titulo; }
public void setTitulo(String titulo) {
    this.titulo = titulo;
}
}
```

- Crie uma classe LivroDAO com os seguintes métodos:
 - **getEntityManager()** – que crie um EntityManager
 - **salvar()** – chame o método do EntityManager que realize a operação de salvar ou alterar um livro.
 - **consultarPorId()** – chame o método do EntityManager que realize a operação de consultar passando o atributo da chave primária.
 - **excluir()** – chame o método do EntityManager que realize a operação de remover um livro.

```
package pbc.jpa.exercicio1.dao;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import pbc.jpa.exercicio1.modelo.Livro;

/**
 * Classe utilizada para fazer realizar as operações de
 * banco de dados sobre a entity Livro.
 */
public class LivroDAO {
```

```
/*
 * Método utilizado para obter o entity manager.
 * @return
 */
private EntityManager getEntityManager() {
    EntityManagerFactory factory = null;
    EntityManager entityManager = null;
    try {
        //Obtém o factory a partir da unidade de persistência.
        factory = Persistence.createEntityManagerFactory
            ("ExercicioJPA1PU");
        //Cria um entity manager.
        entityManager = factory.createEntityManager();
        //Fecha o factory para liberar os recursos utilizado.
    } catch(Exception e) {
        e.printStackTrace();
    }
    return entityManager;
}

/*
 * Método utilizado para salvar ou atualizar as
 * informações de um livro.
 * @param livro
 * @return
 * @throws java.lang.Exception
 */
public Livro salvar(Livro livro) throws Exception {
    EntityManager entityManager = getEntityManager();
    try {
        // Inicia uma transação com o banco de dados.
        entityManager.getTransaction().begin();
        System.out.println("Salvando o livro.");
        /* Verifica se o livro ainda não está salvo
         no banco de dados. */
        if(livro.getId() == null) {
            entityManager.persist(livro);
        } else {
            livro = entityManager.merge(livro);
        }
    } catch(Exception e) {
        e.printStackTrace();
    } finally {
        entityManager.close();
    }
}
```

```
    }
    // Finaliza a transação.
    entityManager.getTransaction().commit();
} catch(Exception e) {
    e.printStackTrace();
} finally {
    entityManager.close();
}
return livro;
}

/**
 * Método que exclui o livro do banco de dados.
 * @param id
 */
public void excluir(Long id) {
    EntityManager entityManager = getEntityManager();
    try {
        // Inicia uma transação com o banco de dados.
        entityManager.getTransaction().begin();
        // Consulta o livro na base de dados através do seu ID.
        Livro livro = entityManager.find(Livro.class, id);
        System.out.println("Excluindo o livro: "
            + livro.getTitulo());

        // Remove o livro da base de dados.
        entityManager.remove(livro);
        // Finaliza a transação.
        entityManager.getTransaction().commit();
    } catch(Exception e) {
        e.printStackTrace();
    } finally {
        entityManager.close();
    }
}

/**
 * Consulta o livro pelo ID.
 * @param id
 * @return
```

```
/*
public Livro consultarPorId(Long id) {
    EntityManager entityManager = getEntityManager();
    Livro livro = null;
    try {
        livro = entityManager.find(Livro.class, id);
    } catch(Exception e) {
        e.printStackTrace();
    } finally {
        entityManager.close();
    }
    return livro;
}
}
```

Vamos criar um arquivo **persistence.xml** dentro da pasta **META-INF** para guardar as configurações do banco de dados. Neste arquivo também vamos informar a Entity Livro.

```
<?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="ExercicioJPA1PU" transaction-type="RES
OURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</p
rovider>
        <class>pbc.jpa.exercicio1.modelo.Livro</class>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:or
acle:thin:<IP>:1521:<SERVICE_NAME>"/>
            <property name="javax.persistence.jdbc.password" value="se
nha"/>
            <property name="javax.persistence.jdbc.driver" value="orac
le.jdbc.OracleDriver"/>
            <property name="javax.persistence.jdbc.user" value="usuari
o"/>
            <property name="javax.persistence.schema-generation.databa
se.action" value="create"/>
        </properties>
    </persistence-unit>
</persistence>
```

Para testar as operações sobre a entidade Livro, vamos criar a classe **LivroTeste** que utiliza o LivroDAO para fazer uso da EntityManager para gerenciar a entity Livro.

```
package pbc.jpa.exercicio1.teste;

import pbc.jpa.exercicio1.dao.LivroDAO;
import pbc.jpa.exercicio1.modelo.Livro;

/**
 * Classe utilizada para testar a persistência da
 * entity Livro.
 */
public class LivroTeste {
    public static void main(String[] args) {
        try {
            Livro livro = new Livro();
            livro.setAutor("Rafael Guimarães Sakurai");
            livro.setIsbn("111-11-1111-111-1");
            livro.setPaginas(439);
            livro.setPreco(30.90);
            livro.setTitulo("Guia de estudos SCJA.");

            LivroDAO dao = new LivroDAO();
            livro = dao.salvar(livro);
            System.out.println("ID do livro salvo: "
                + livro.getId());

            /*
             * PARA FAZER - Teste a consulta, alteração
             * e exclusão do livro.
             */
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Agora só testar e terminar de implementar.

Exercício 2

Crie uma aplicação Swing ou Console utilizando JPA para fazer o CRUD (salvar, alterar, consultar e excluir) da tabela Produto a seguir:

```
CREATE TABLE Produto (
    id          number(5)      NOT NULL PRIMARY KEY,
    nome        varchar2(200)   NOT NULL,
    preco       number(10,2)    NOT NULL,
    dataValidade date,
    qtdEstoque  number(5)
);
```

Exercício 3

Crie uma aplicação Swing ou Console utilizando o JPA para fazer o CRUD (salvar, consultar, alterar e excluir) de duas entidades a sua escolha.

Relacionamento entre entidades

Os relacionamentos também podem ser:

Unidirecional

A partir de uma entidade, é possível encontrar outra entidade, mas o contrario não acontece, nesse exemplo a Entidade A conhece a Entidade B, mas o contrario não acontece.



Bidirecional

Ambas entidades se conhecem, nesse exemplo a Entidade A conhece a Entidade B e vice-versa.



Esses relacionamentos são definidos de acordo com a necessidade do negocio, não existe uma regra que sempre tem que ser unidirecional ou bidirecional.

Os quatro tipos de cardinalidade

Um-para-Um (OneToOne)

Este relacionamento informa que há apenas um registro da entidade relacionado com um registro de outra entidade.

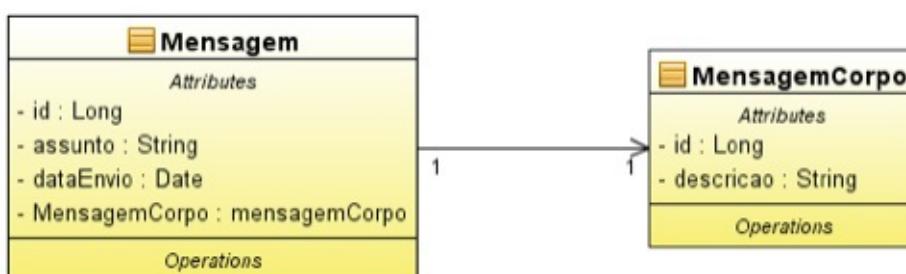
Exemplo de relacionamento Um-para-Um unidirecional:

Script do banco de dados:

```
CREATE TABLE Mensagem (
    id number(5) NOT NULL PRIMARY_KEY,
    assunto varchar2(200) NOT NULL,
    dataEnvio date NOT NULL,
    mensagemcorpo_id number(5)
);

CREATE TABLE MensagemCorpo (
    id number(5) NOT NULL PRIMARY KEY,
    descricao varchar2(200)
);
```

Modelo UML:



Neste exemplo definimos que uma **Mensagem** possui uma **MensagemCorpo**, então desta forma a Mensagem sabe qual é seu MensagemCorpo, mas o contrario não existe, a MensagemCorpo não tem a necessidade de conhecer qual a Mensagem está associado a ele, ou seja, temos um relacionamento **unidirecional**.

Primeiramente podemos mostrar apenas uma listagem de Mensagens, mas não tem necessidade por enquanto de mostrar o conteúdo de todas as mensagens e depois caso eu queira ler o conteúdo da mensagem podemos através dela chegar até seu corpo utilizando o atributo do tipo MensagemCorpo.

Código fonte das classes com o relacionamento:

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

/**
 * Classe utilizada para representar uma Mensagem.
 */
@Entity
public class Mensagem implements Serializable {
    private static final long serialVersionUID
        = 1912492882356572322L;

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

    private String assunto;

    @Temporal(TemporalType.DATE)
    private Date dataEnvio;

    @OneToOne(cascade=CCascadeType.ALL)
    private MensagemCorpo mensagemCorpo;
```

```
public String getAssunto() { return assunto; }
public void setAssunto(String assunto) {
    this.assunto = assunto;
}

public Date getDataEnvio() { return dataEnvio; }
public void setDataEnvio(Date dataEnvio) {
    this.dataEnvio = dataEnvio;
}

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

public MensagemCorpo getMensagemCorpo() {
    return mensagemCorpo;
}
public void setMensagemCorpo(MensagemCorpo mensagemCorpo) {
    this.mensagemCorpo = mensagemCorpo;
}
```

Na classe Mensagem utilizamos a anotação **javax.persistence.OneToOne** para definir o relacionamento de um-para-um entre as classes Mensagem e MensagemCorpo.

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

/**
 * Classe utilizada para representar o corpo de uma
 * mensagem.
 */
@Entity
public class MensagemCorpo implements Serializable {
    private static final long serialVersionUID
        = 986589124772488369L;

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

    private String descricao;

    public String getDescricao() { return descricao; }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

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

A classe MensagemCorpo é uma entidade normal que não conhece a classe Mensagem.

Note que não precisamos criar nenhuma referência para informar que o atributo mensagemCorpo da classe **Mensagem** referencia a coluna **mensagemcorpo_id** da tabela Mensagem.

O JPA possui alguns padrões para facilitar o mapeamento entre a classe Java e a tabela do banco de dados, quando criamos uma coluna de chave estrangeira seguindo o padrão **nometabela_chaveprimaria**, o mapeamento é feito automaticamente pelo JPA, ou seja, o atributo `MensagemCorpo mensagemCorpo` é automaticamente associado com a coluna `mensagemcorpo_id`.

javax.persistence.OneToOne

Esta anotação define uma associação com outra entidade que tenha a multiplicidade de um-para-um. A tabela a seguir mostra as propriedades da anotação `@OneToOne`.

Propriedade	Descrição
cascade	As operações que precisam ser refletidas no alvo da associação.
fetch	Informa se o alvo da associação precisa ser obtido apenas quando for necessário ou se sempre deve trazer.
mappedBy	Informa o atributo que é dono do relacionamento.
optional	Informa se a associação é opcional.
targetEntity	A classe entity que é alvo da associação.

Quando precisamos especificar um mapeamento que não é padrão do JPA, podemos utilizar a anotação **javax.persistence.JoinColumn**, por exemplo se a tabela Mensagem e MensagemCorpo fossem:

```
CREATE TABLE Mensagem (
    id number(5) NOT NULL PRIMARY KEY,
    assunto varchar2(200),
    dataEnvio date,
    ID_MENSAGEMCORPO number(5)
);

CREATE TABLE MensagemCorpo (
    MC_ID number(5) NOT NULL PRIMARY_KEY,
    descricao varchar2(200)
);
```

Poderíamos utilizar o JoinColumn para criar a associação:

```
@OneToOne(cascade=CascadeType.ALL)
@JoinColumn(name="ID_MENSAGEMCORPO", referencedColumnName="MC_ID")
private MensagemCorpo mensagemCorpo;
```

javax.persistence.JoinColumn

Esta anotação é utilizada para especificar a coluna utilizada na associação com outra entity. A tabela a seguir apresenta as propriedades da anotação `@JoinColumn`.

Propriedade	Descrição
columnDefinition	Definição do tipo da coluna.
insertable	Informa se a coluna é incluída no SQL de INSERT.
name	Informa o nome da coluna de chave estrangeira.
nullable	Informa se a coluna pode ser null.
referencedColumnName	Nome da coluna que é referenciada pela coluna da chave estrangeira.
table	Nome da tabela que contém a coluna.
unique	Informa se a propriedade é chave única.
updatable	Informa se a coluna é incluída no SQL de UPDATE.

Um-para-Muitos (OneToMany)

Este relacionamento informa que o registro de uma entidade está relacionado com vários registros de outra entidade.

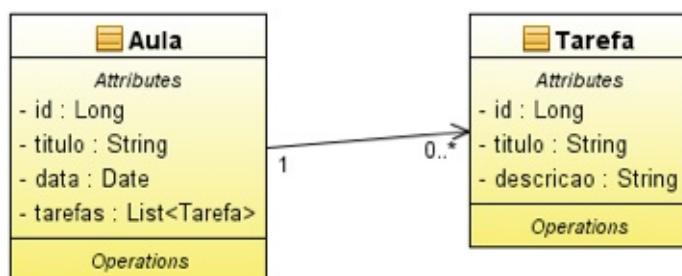
Exemplo de relacionamento Um-para-Muitos unidirecional:

Script do banco de dados:

```
CREATE TABLE Aula (
    id number(5) NOT NULL PRIMARY KEY,
    titulo varchar2(45) NOT NULL,
    data date NOT NULL
);

CREATE TABLE Tarefa (
    id number(5) NOT NULL PRIMARY KEY,
    titulo varchar2(45) NOT NULL,
    descricao varchar2(45) NOT NULL,
    aula_id number(5)
);
```

Modelo UML:



Neste exemplo definimos que uma **Aula** possui uma lista de **Tarefa**, portanto a aula pode não ter tarefa, pode ter apenas uma tarefa ou pode ter varias tarefas, uma Tarefa não precisa saber de qual Aula ela está associada, portanto temos um relacionamento unidirecional.

Código fonte das classes com o relacionamento:

Na classe Aula utilizamos a anotação **javax.persistence.OneToMany** no atributo **tarefas**, para informar que uma Aula está associada com várias tarefas.

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import java.util.Date;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

/**
 * Classe utilizada para representar uma aula.
 */
@Entity
public class Aula implements Serializable {
    private static final long serialVersionUID
        = -6745032908099856302L;

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

    private String titulo;

    @Temporal(TemporalType.DATE)
    private Date data;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name="aula_id")
    private List<Tarefa> tarefas;

    public Date getData() { return data; }
    public void setData(Date data) { this.data = data; }

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

    public String getTitulo() { return titulo; }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public List<Tarefa> getTarefas() { return tarefas; }
```

```
public void setTarefas(List<Tarefa> tarefas) {  
    this.tarefas = tarefas;  
}  
}
```

javax.persistence.OneToMany

Esta anotação define uma associação com outra entidade que tenha a multiplicidade de um-para-muitos.

Propriedade	Descrição
cascade	As operações que precisam ser refletidas no alvo da associação.
fetch	Informa se o alvo da associação precisa ser obtido apenas quando for necessário ou se sempre deve trazer.
mappedBy	Informa o atributo que é dono do relacionamento.
targetEntity	A classe entity que é alvo da associação.

A classe Tarefa é uma entidade normal que não conhece a classe Aula.

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

/**
 * Classe utilizada para representar as tarefas
 * aplicadas em uma aula.
 */
@Entity
public class Tarefa implements Serializable {
    private static final long serialVersionUID = 29526300171271739
88L;

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

    private String titulo;
    private String descricao;

    public String getDescricao() { return descricao; }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

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

    public String getTitulo() { return titulo; }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
}
```

Muitos-para-Um (ManyToOne)

Este relacionamento informa que existem muitos registros de uma entidade associados a um registro de outra entidade.

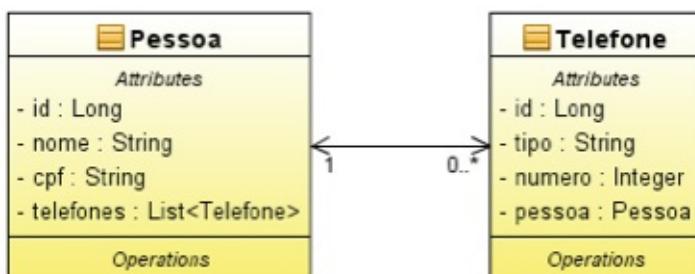
Exemplo de relacionamento Um-para-Muitos e Muitos-para-Um bidirecional:

Script do banco de dados:

```
CREATE TABLE Pessoa (
    id number(5) NOT NULL PRIMARY KEY,
    nome varchar2(200) NOT NULL,
    cpf varchar2(11) NOT NULL
);

CREATE TABLE Telefone (
    id number(5) NOT NULL PRIMARY KEY,
    tipo varchar2(200) NOT NULL,
    numero number(8) NOT NULL,
    pessoa_id number(5)
);
```

Modelo UML:



Neste exemplo definimos que uma Pessoa possui uma lista de Telefones e um Telefone está associado a uma Pessoa, portanto temos um relacionamento bidirecional.

Código fonte das classes com o relacionamento:

Na entidade **Pessoa** definimos que uma pessoa possui vários telefones através do atributo `List<Telefone> telefones` e adicionamos a anotação `javax.persistence.OneToMany` para informar que o relacionamento de Pessoa

para Telefone é de Um-para-Muitos, note que nesta anotação definimos a propriedade **mappedBy** como "pessoa" que é para informar que o atributo com o nome **pessoa** na entity **Telefone** que é dona do relacionamento.

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

/**
 * Classe utilizada para representar uma Pessoa.
 */
@Entity
public class Pessoa implements Serializable {
    private static final long serialVersionUID
        = -1905907502453138175L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nome;
    private String cpf;
    @OneToMany(mappedBy="pessoa", cascade=CascadeType.ALL)
    private List<Telefone> telefones;

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

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

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

```
public List<Telefone> getTelefones() { return telefones; }
public void setTelefones(List<Telefone> telefones) {
    this.telefones = telefones;
}
```

Na entidade Telefone definimos o atributo `Pessoa pessoa` e adicionamos a anotação `javax.persistence.ManyToOne` para definir que o relacionamento de Telefone para Pessoa é de Muitos-para-Um.

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

/**
 * Classe utilizada para representar um Telefone.
 */
@Entity
public class Telefone implements Serializable {
    private static final long serialVersionUID
        = 7526502149208345058L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String tipo;
    private Integer numero;
    @ManyToOne
    private Pessoa pessoa;

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

    public Integer getNumero() { return numero; }
```

```
public void setNumero(Integer numero) {
    this.numero = numero;
}

public Pessoa getPessoa() { return pessoa; }
public void setPessoa(Pessoa pessoa) {
    this.pessoa = pessoa;
}

public String getTipo() { return tipo; }
public void setTipo(String tipo) { this.tipo = tipo; }
}
```

javax.persistence.ManyToOne

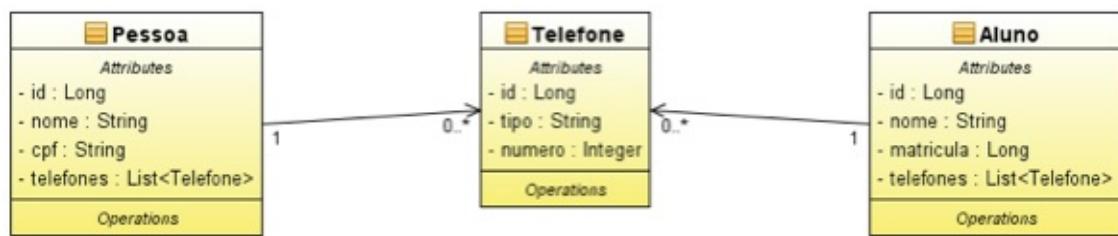
Esta anotação define uma associação com outra entidade que tenha a multiplicidade de muitos-para-um.

Propriedade	Descrição
cascade	As operações que precisam ser refletidas no alvo da associação.
fetch	Informa se o alvo da associação precisa ser obtido apenas quando for necessário ou se sempre deve trazer.
optional	Informa se a associação é opcional.
targetEntity	A classe entity que é alvo da associação.

Também podemos adicionar uma tabela para realizar o relacionamento unidirecional de um-para-muitos e o relacionamento muitos-para-muitos, normalmente utilizamos esta alternativa como uma forma de normalizar os dados evitando duplicar o conteúdo dos registros.

Nesse exemplo queremos utilizar a entidade Telefone com as entidades Pessoa e Aluno, dessa forma Pessoa possui uma lista de Telefones e Aluno possui uma lista de Telefones, mas o telefone não sabe para quem ele está associado. Este tipo de relacionamento é unidirecional de um-para-muitos.

Os quatro tipos de cardinalidade



Na base de dados iremos criar as tabelas Pessoa, Telefone e Aluno, também iremos criar duas tabelas de associação chamadas Pessoa_Telefone e Aluno_Telefone:

```
CREATE TABLE Pessoa (
    id number(5) NOT NULL PRIMARY KEY,
    nome varchar2(200) NOT NULL,
    cpf varchar2(11) NOT NULL
);

CREATE TABLE Aluno (
    id number(5) NOT NULL PRIMARY_KEY,
    nome varchar2(200) NOT NULL,
    matricula number(5) NOT NULL
);

CREATE TABLE Telefone (
    id number(5) NOT NULL PRIMARY KEY,
    tipo varchar2(200) NOT NULL,
    numero number(5) NOT NULL
);

CREATE TABLE Pessoa_Telefone (
    pessoa_id number(5),
    telefone_id number(5)
);

CREATE TABLE Aluno_Telefone (
    aluno_id number(5),
    telefone_id number(5)
);
```

Na entidade **Pessoa** definimos que uma pessoa possui vários telefones através do atributo `List<Telefone> telefones` e adicionamos a anotação **javax.persistence.OneToMany** para informar que o relacionamento de Pessoa para Telefone é de Um-para-Muitos.

Para informar que vamos utilizar a tabela PESSOA_TELEFONE para realizar a associação entre as tabelas PESSOA e TELEFONE utilizamos a anotação **javax.persistence.JoinTable**.

Para informar que a coluna PESSOA_ID da tabela PESSOA_TELEFONE é a coluna chave estrangeira para a tabela PESSOA e para informar que a coluna TELEFONE_ID da tabela PESSOA_TELEFONE é a chave estrangeira para a tabela TELEFONE utilizamos a anotação **javax.persistenceJoinColumn**.

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import java.util.List;
import javax.persistence.CascadeType;
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.JoinTable;
import javax.persistence.OneToMany;

/**
 * Classe utilizada para representar uma Pessoa.
 */
@Entity
public class Pessoa implements Serializable {
    private static final long serialVersionUID
        = -1905907502453138175L;

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

```

private String nome;
private String cpf;
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinTable(name="PESSOA_TELEFONE",
joinColumns={@JoinColumn(name = "PESSOA_ID")},
inverseJoinColumns={@JoinColumn(name = "TELEFONE_ID")})
private List<Telefone> telefones;

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

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

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

public List<Telefone> getTelefones() { return telefones; }
public void setTelefones(List<Telefone> telefones) {
    this.telefones = telefones;
}
}

```

javax.persistence.JoinTable

Esta anotação é utilizada para definir uma tabela que será utilizada na associação de um-para-muitos ou de muitos-para-muitos.

Propriedade	Descrição
catalog	O catalogo da tabela.
inverseJoinColumns	Chave estrangeira para realizar a associação com a tabela que não é dona do relacionamento.
joinColumns	Chave estrangeira para realizar a associação com a tabela que é dona do relacionamento.
name	Nome da tabela de associação.
schema	Esquema da tabela.
uniqueConstraints	Regras que podem ser adicionadas na tabela.

Na entidade **Aluno** definimos que um aluno possui vários telefones através do atributo `List<Telefone> telefones` e adicionamos a anotação **javax.persistence.OneToMany** para informar que o relacionamento de Aluno para Telefone é de Um-para-Muitos.

Para informar que vamos utilizar a tabela ALUNO_TELEFONE para realizar a associação entre as tabelas ALUNO e TELEFONE utilizamos a anotação **javax.persistence.JoinTable**.

Para informar que a coluna ALUNO_ID da tabela ALUNO_TELEFONE é a coluna chave estrangeira para a tabela ALUNO e para informar que a coluna TELEFONE_ID da tabela ALUNO_TELEFONE é a chave estrangeira para a tabela TELEFONE utilizamos a anotação **javax.persistenceJoinColumn**.

```
package pbc.jpa.exemplo.modelo;

import java.util.List;
import javax.persistence.CascadeType;
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.JoinTable;
import javax.persistence.OneToMany;

/**
 * Classe utilizada para representar uma entidade Aluno.
 */
@Entity
public class Aluno {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nome;
    private Long matricula;
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinTable(name="ALUNO_TELEFONE",
```

```
joinColumns={@JoinColumn(name = "ALUNO_ID")},
inverseJoinColumns={@JoinColumn(name = "TELEFONE_ID"))}
private List<Telefone> telefones;

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

public Long getMatricula() { return matricula; }
public void setMatricula(Long matricula) {
    this.matricula = matricula;
}

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

public List<Telefone> getTelefones() { return telefones; }
public void setTelefones(List<Telefone> telefones) {
    this.telefones = telefones;
}
```

Agora vamos declarar a entidade **Telefone**, note que esta entidade não conhece as associações que são criadas para ela.

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

/**
 * Classe utilizada para representar um Telefone.
 */
@Entity
public class Telefone implements Serializable {
    private static final long serialVersionUID
        = 7526502149208345058L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String tipo;
    private Integer numero;

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

    public Integer getNumero() { return numero; }
    public void setNumero(Integer numero) {
        this.numero = numero;
    }

    public String getTipo() { return tipo; }
    public void setTipo(String tipo) { this.tipo = tipo; }
}
```

Para testar o cadastro de Aluno e Telefone vamos criar a classe **AlunoDAO** para salvar, alterar, consultar por id e apagar os registro do aluno e telefone.

```
package pbc.jpa.exemplo.dao;
```

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import pbc.jpa.exemplo.modelo.Aluno;

/**
 * Classe DAO para manipular as informações do
 * Aluno no banco de dados.
 */
public class AlunoDAO {
    private EntityManager getEntityManager() {
        EntityManagerFactory factory = null;
        EntityManager entityManager = null;
        try {
            //Obtem o factory a partir da unidade de persistência.
            factory = Persistence.createEntityManagerFactory
                ("UnidadeDePersistencia");
            //Cria um entity manager.
            entityManager = factory.createEntityManager();
        } finally {
            factory.close();
        }
        return entityManager;
    }

    public Aluno consultarPorId(Long id) {
        EntityManager entityManager = getEntityManager();
        Aluno aluno = null;
        try {
            aluno = entityManager.find(Aluno.class, id);
        } finally {
            entityManager.close();
        }
        return aluno;
    }

    public Aluno salvar(Aluno aluno) {
        EntityManager entityManager = getEntityManager();
        try {
```

```
// Inicia uma transação com o banco de dados.
entityManager.getTransaction().begin();
System.out.println("Salvando as informações do aluno.");
/* Verifica se o aluno ainda não está salvo no
banco de dados. */
if(aluno.getId() == null) {
    entityManager.persist(aluno);
} else {
    aluno = entityManager.merge(aluno);
}
// Finaliza a transação.
entityManager.getTransaction().commit();
} catch(Exception ex) {
    entityManager.getTransaction().rollback();
} finally {
    entityManager.close();
}
// Retorna o aluno salvo.
return aluno;
}

public void apagar(Long id) {
    EntityManager entityManager = getEntityManager();
    try {
        // Inicia uma transação com o banco de dados.
        entityManager.getTransaction().begin();
        // Consulta o aluno na base de dados através do seu ID.
        Aluno aluno = entityManager.find(Aluno.class, id);
        System.out.println("Excluindo o aluno: "
            + aluno.getNome());

        // Remove o aluno da base de dados.
        entityManager.remove(aluno);
        // Finaliza a transação.
        entityManager.getTransaction().commit();
    } catch(Exception ex) {
        entityManager.getTransaction().rollback();
    } finally {
        entityManager.close();
    }
}
```

```
}
```

```
}
```

Vamos criar a classe **AlunoDAOTeste** que utiliza a classe AlunoDAO para salvar, alterar, consultar por id e apagar os registros de aluno e telefone:

```
package pbc.jpa.exemplo.dao;

import java.util.ArrayList;
import java.util.List;
import pbc.jpa.exemplo.modelo.Aluno;
import pbc.jpa.exemplo.modelo.Telefone;

/**
 * Classe utilizada para testar as operações do
 * banco de dados referente ao Aluno.
 */
public class AlunoDAOTeste {
    public static void main(String[] args) {
        AlunoDAO dao = new AlunoDAO();

        //Cria uma aluno.
        Aluno aluno = new Aluno();
        aluno.setNome("Rafael");
        aluno.setMatricula(123456L);

        //Cria o telefone residencial do aluno.
        Telefone telefone = new Telefone();
        telefone.setTipo("RES");
        telefone.setNumero(12345678);
        //Cria o telefone celular do aluno.
        Telefone telefone2 = new Telefone();
        telefone2.setTipo("CEL");
        telefone2.setNumero(87654321);

        //Cria uma lista de telefones e guarda dentro do aluno.
        List<Telefone> telefones = new ArrayList<Telefone>();
        telefones.add(telefone);
        telefones.add(telefone2);
```

```
aluno.setTelefones(telefones);

System.out.println("Salva as informações do aluno.");
aluno = dao.salvar(aluno);

System.out.println("Consulta o aluno que foi salvo.");
Aluno alunoConsultado = dao.consultarPorId(aluno.getId());
System.out.println(aluno.getNome());
for(Telefone tel : aluno.getTelefones()) {
    System.out.println(tel.getTipo() + " - "
        + tel.getNumero());
}

//Cria o telefone comercial do aluno.
Telefone telefone3 = new Telefone();
telefone3.setTipo("COM");
telefone3.setNumero(55554444);
//Adiciona o novo telefone a lista de telefone do aluno.
alunoConsultado.getTelefones().add(telefone3);

System.out.println("Atualiza as informações do aluno.");
alunoConsultado = dao.salvar(alunoConsultado);
System.out.println(alunoConsultado.getNome());
for(Telefone tel : alunoConsultado.getTelefones()) {
    System.out.println(tel.getTipo() + " - "
        + tel.getNumero());
}

System.out.println("Apaga o registro do aluno.");
dao.apagar(alunoConsultado.getId());
}
```

Quando executamos a classe **AlunoDAOTeste** temos a seguinte saída no console:

```
Salva as informações do aluno:
```

```
Hibernate: insert into Aluno (matricula, nome) values (?, ?)
```

```
Hibernate: insert into Telefone (numero, tipo) values (?, ?)
Hibernate: insert into Telefone (numero, tipo) values (?, ?)
Hibernate: insert into ALUNO_TELEFONE (ALUNO_ID, TELEFONE_ID)
values (?, ?)
Hibernate: insert into ALUNO_TELEFONE (ALUNO_ID, TELEFONE_ID)
values (?, ?)
```

Consulta o aluno que foi salvo:

```
Hibernate: select aluno0_.id as id21_1_, aluno0_.matricula as
matricula21_1_, aluno0_.nome as nome21_1_, telefones1_.ALUNO_ID
as ALUNO1_3_, telefone2_.id as TELEFONE2_3_, telefone2_.id as
id18_0_, telefone2_.numero as numero18_0_, telefone2_.tipo as
tipo18_0_ from Aluno aluno0_ left outer join ALUNO_TELEFONE
telefones1_ on aluno0_.id=telefones1_.ALUNO_ID left outer join
Telefone telefone2_ on telefones1_.TELEFONE_ID=telefone2_.id
where aluno0_.id=?Rafael
RES - 12345678
CEL - 87654321
```

Atualiza as informações do aluno:

```
Hibernate: select aluno0_.id as id38_1_, aluno0_.matricula as
matricula38_1_, aluno0_.nome as nome38_1_, telefones1_.ALUNO_ID
as ALUNO1_3_, telefone2_.id as TELEFONE2_3_, telefone2_.id as
id35_0_, telefone2_.numero as numero35_0_, telefone2_.tipo as
tipo35_0_ from Aluno aluno0_ left outer join ALUNO_TELEFONE
telefones1_ on aluno0_.id=telefones1_.ALUNO_ID left outer join
Telefone telefone2_ on telefones1_.TELEFONE_ID=telefone2_.id
where aluno0_.id=?
Hibernate: insert into Telefone (numero, tipo) values (?, ?)
Hibernate: delete from ALUNO_TELEFONE where ALUNO_ID=?
Hibernate: insert into ALUNO_TELEFONE (ALUNO_ID, TELEFONE_ID)
values (?, ?)
Hibernate: insert into ALUNO_TELEFONE (ALUNO_ID, TELEFONE_ID)
values (?, ?)
Hibernate: insert into ALUNO_TELEFONE (ALUNO_ID, TELEFONE_ID)
values (?, ?)
Rafael
RES - 12345678
```

```
CEL - 87654321  
COM - 55554444
```

Apaga o registro do aluno:

```
Hibernate: select aluno0_.id as id55_1_, aluno0_.matricula as matricula55_1_, aluno0_.nome as nome55_1_, telefones1_.ALUNO_ID as ALUNO1_3_, telefone2_.id as TELEFONE2_3_, telefone2_.id as id52_0_, telefone2_.numero as numero52_0_, telefone2_.tipo as tipo52_0_ from Aluno aluno0_ left outer join ALUNO_TELEFONE telefones1_ on aluno0_.id=telefones1_.ALUNO_ID left outer join Telefone telefone2_ on telefones1_.TELEFONE_ID=telefone2_.id where aluno0_.id=?  
Hibernate: delete from ALUNO_TELEFONE where ALUNO_ID=?  
Hibernate: delete from Telefone where id=?  
Hibernate: delete from Telefone where id=?  
Hibernate: delete from Telefone where id=?  
Hibernate: delete from Aluno where id=?
```

Muitos-para-Muitos (ManyToMany)

Este relacionamento informa que muitos registros de uma entidade estão relacionados com muitos registros de outra entidade:

Script do banco de dados:

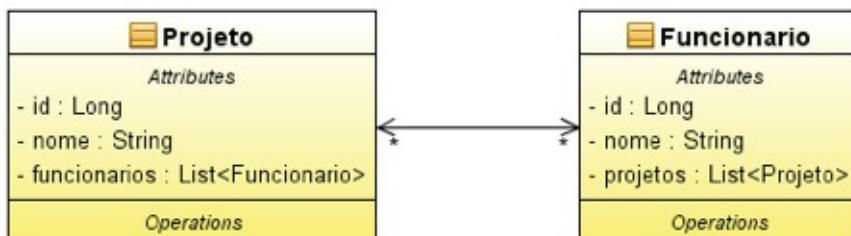
```
CREATE TABLE Projeto (
    id number(5) NOT NULL PRIMARY_KEY,
    nome varchar2(200) NOT NULL
);

CREATE TABLE Funcionario (
    id number(5) NOT NULL PRIMARY_KEY,
    nome varchar2(200) NOT NULL
);

CREATE TABLE Projeto_Funcionario (
    projeto_id number(5),
    funcionario_id number(5)
);
```

Note que nesse caso precisamos utilizar uma tabela intermediária entre Projeto e Funcionario, para evitar duplicar dados (normalização) desnecessários no banco de dados. Através da tabela Projeto_Funcionario criamos o relacionamento entre Projeto e Funcionario.

Modelo UML:



Neste exemplo definimos que um **Projeto** tem vários funcionários, e um **Funcionario** participa de vários projetos, portanto temos um relacionamento **bidirecional** de muitos-para-muitos.

Código fonte das classes com o relacionamento:

A entidade **Projeto** possui um relacionamento de **muitos-para-muitos** com a entidade **Funcionario**, para definir esta associação utilizamos a anotação **javax.persistence.ManyToMany**, note que utilizamos a propriedade **mappedBy** da anotação ManyToMany para informar que o Projeto é o dono do relacionamento.

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

/**
 * Classe utilizada para representar um projeto.
 */
@Entity
public class Projeto implements Serializable {
    private static final long serialVersionUID
        = 1081869386060246794L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nome;
    @ManyToOne(mappedBy="projetos", cascade=CascadeType.ALL)
    private List<Funcionario> desenvolvedores;

    public List<Funcionario> getDesenvolvedores() {
        return desenvolvedores;
    }
    public void setDesenvolvedores(List<Funcionario>
        desenvolvedores) {
        this.desenvolvedores = desenvolvedores;
    }

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

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

javax.persistence.ManyToMany

Esta anotação define uma associação com outra entidade que tenha a multiplicidade de muitos-para-muitos.

Propriedade	Descrição
cascade	As operações que precisam ser refletidas no alvo da associação.
fetch	Informa se o alvo da associação precisa ser obtido apenas quando for necessário ou se sempre deve trazer.
mappedBy	Informa o atributo que é dono do relacionamento.
targetEntity	A classe entity que é alvo da associação.

A entidade Funcionario possui um relacionamento de **muitos-para-muitos** com a entidade Projeto, para definir esta associação utilizamos a anotação **javax.persistence.ManyToMany**.

Para informar que vamos utilizar a tabela PROJETO_FUNCIONARIO para realizar a associação entre PROJETO e FUNCIONARIO utilizamos a anotação **javax.persistence.JoinTable**.

Para informar que a coluna PROJETO_ID da tabela PROJETO_FUNCIONARIO é a coluna chave estrangeira para a tabela PROJETO e para informar que a coluna FUNCIONARIO_ID da tabela PROJETO_FUNCIONARIO é a chave estrangeira para a tabela FUNCIONARIO utilizamos a anotação

javax.persistenceJoinColumn.

```
package pbc.jpa.exemplo.modelo;

import java.io.Serializable;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
```

```
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;

/**
 * Classe utilizada para representar um Funcionário.
 */
@Entity
public class Funcionario implements Serializable {
    private static final long serialVersionUID
        = -9109414221418128481L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nome;
    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name="PROJETO_FUNCIONARIO",
               joinColumns={@JoinColumn(name="PROJETO_ID")},
               inverseJoinColumns={@JoinColumn(name="FUNCIONARIO_ID")})
    private List<Projeto> projetos;

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

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

    public List<Projeto> getProjetos() { return projetos; }
    public void setProjetos(List<Projeto> projetos) {
        this.projetos = projetos;
    }
}
```

CascadeType

Com o CascadeType podemos definir a forma como serão propagadas as operações em cascata de uma Entity para suas referencias.



- **PERSIST** – Quando salvar a Entidade A, também será salvo todas as Entidades B associadas.
- **MERGE** – Quando atual as informações da Entidade A, também será atualizado no banco de dados todas as informações das Entidades B associadas.
- **REMOVE** – Quando remover a Entidade A, também será removida todas as entidades B associadas.
- **REFRESH** – Quando houver atualização no banco de dados na Entidade A, todas as entidades B associadas serão atualizadas.
- **ALL** – Corresponde a todas as operações acima (MERGE, PERSIST, REFRESH e REMOVE).

EntityType

Com o FetchType podemos definir a forma como serão trazidos os relacionamentos, podemos fazer de duas formas:



- **EAGER** - Traz todas as entidades que estão relacionadas, ou seja, se a Entidade A possui um relacionamento com a Entidade B, então quando consultar a Entidade A, também será consultado suas referencias na Entidade B.
- **LAZY** - Não traz as entidades que estão relacionadas, ou seja, se a Entidade A possui um relacionamento com a Entidade B, então quando consultar a Entidade A só serão retornadas as informações referentes a esta Entidade.

Exemplo de relacionamento LAZY, desse modo o corpo da mensagem é consultado apenas quando houver a necessidade:

```
@Entity
public class Mensagem {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String assunto;
    @Temporal(TemporalType.DATE)
    private Date dataEnvio;

    @OneToOne(cascade=CascadeType.ALL, fetch=FetchType.LAZY)
    private MensagemCorpo mensagemCorpo;
}
```

Nesse exemplo utilizamos a enum **FetchType.LAZY** no relacionamento um-para-um para informar que quando consultarmos a entidade Mensagem, não queremos consultar a entidade MensagemCorpo ao mesmo tempo.

Exercícios

Exercício 1

Neste exercício vamos abordar como funciona os relacionamentos entre as entidades, vamos utilizar o relacionamento entre as entidades **Cliente** e **Endereco**, quando salvar o cliente também deve salvar o endereço e quando o cliente for consultado deve trazer também as informações do endereço.

Crie o seguinte banco de dados:

```
CREATE SEQUENCE CLI_SEQ INCREMENT BY 1
    START WITH 1 NOCACHE NOCYCLE;
CREATE SEQUENCE END_SEQ INCREMENT BY 1
    START WITH 1 NOCACHE NOCYCLE;

CREATE TABLE ENDERECO (
    id number(5) NOT NULL PRIMARY KEY,
    estado VARCHAR2(50) NOT NULL,
    cidade VARCHAR2(50) NOT NULL,
    bairro VARCHAR2(50) NOT NULL,
    logradouro VARCHAR2(50) NOT NULL,
    complemento VARCHAR2(50) NOT NULL
);

CREATE TABLE CLIENTE (
    id number(5) NOT NULL PRIMARY KEY,
    nome VARCHAR2(100) NOT NULL,
    endereco_id number(5) NOT NULL
);
```

Crie um projeto Java chamado **ExercicioJPA4**, adicione as bibliotecas **EclipseLink JPA** e Driver da Oracle **ojdbc7.jar** e crie:

- Classe entity para representar um endereço com os atributos id, estado, cidade, bairro, logradouro e complemento.

```
package pbc.jpa.exercicio4.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;

/**
 * Classe utilizada para representar um Endereço.
 */
@Entity
@SequenceGenerator(name = "ENDERECO_SEQ",
    sequenceName = "END_SEQ", initialValue = 1,
    allocationSize = 1)
public class Endereco implements Serializable {
    private static final long serialVersionUID
        = 5331450149454053703L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "ENDERECO_SEQ")
    private Long id;
    private String estado;
    private String cidade;
    private String bairro;
    private String logradouro;
    private String complemento;

    public String getBairro() { return bairro; }
    public void setBairro(String bairro) {
        this.bairro = bairro;
    }

    public String getCidade() { return cidade; }
    public void setCidade(String cidade) {
        this.cidade = cidade;
    }
```

```
public String getComplemento() { return complemento; }
public void setComplemento(String complemento) {
    this.complemento = complemento;
}

public String getEstado() { return estado; }
public void setEstado(String estado) {
    this.estado = estado;
}

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

public String getLogradouro() { return logradouro; }
public void setLogradouro(String logradouro) {
    this.logradouro = logradouro;
}
}
```

- Classe entity para representar um cliente com id, nome e endereço, note que vamos utilizar a anotação **javax.persistence.OneToOne** para definir o relacionamento de um-para-um entre as entidades Cliente e Endereco.

```
package pbc.jpa.exercicio4.modelo;

import java.io.Serializable;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.SequenceGenerator;

/**
 * Classe utilizada para representar um Cliente.
 */
@Entity
@SequenceGenerator(name = "CLIENTE_SEQ",
```

```
sequenceName = "CLI_SEQ", initialValue = 1,
allocationSize = 1)
public class Cliente implements Serializable {
    private static final long serialVersionUID
    = 4521490124826140567L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "CLIENTE_SEQ")
    private Long id;
    private String nome;
    @OneToOne(cascade=CascadeType.ALL)
    private Endereco endereco;

    public Endereco getEndereco() { return endereco; }
    public void setEndereco(Endereco endereco) {
        this.endereco = endereco;
    }

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

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

- Crie o arquivo **persistence.xml** dentro da pasta META-INF do projeto, note que neste arquivo vamos informar qual o banco de dados iremos utilizar e quais classes são entidades do banco:

```
<?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="ExercicioJPA4PU" transaction-type="RES
OURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</p
rovider>
        <class>pbc.jpa.exercicio3.modelo.Endereco</class>
        <class>pbc.jpa.exercicio3.modelo.Cliente</class>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:or
acle:thin:<IP>:1521:<SERVICE_NAME>"/>
            <property name="javax.persistence.jdbc.password" value="se
nha"/>
            <property name="javax.persistence.jdbc.driver" value="orac
le.jdbc.OracleDriver"/>
            <property name="javax.persistence.jdbc.user" value="usuari
o"/>
            <property name="javax.persistence.schema-generation.databa
se.action" value="create"/>
        </properties>
    </persistence-unit>
</persistence>
```

- Crie a classe **ClienteDAO** que será responsável por utilizar o **EntityManager** para manipular (salvar, alterar, remover e consultar por id) as informações referentes ao Cliente.

```
package pbc.jpa.exercicio4.dao;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import pbc.jpa.exercicio4.modelo.Cliente;
import pbc.jpa.exercicio4.modelo.Endereco;
```

```
/***
 * Classe utilizada para fazer as operações de
 * banco de dados sobre a entity Cliente.
 */
public class ClienteDAO {

    /**
     * Método utilizado para obter o entity manager.
     * @return
     */
    private EntityManager getEntityManager() {
        EntityManagerFactory factory = null;
        EntityManager entityManager = null;
        try {
            //Obtém o factory a partir da unidade de persistencia.
            factory = Persistence.createEntityManagerFactory
                ("ExercicioJPA4PU");
            //Cria um entity manager.
            entityManager = factory.createEntityManager();
        } catch(Exception e){
            e.printStackTrace();
        }
        return entityManager;
    }

    /**
     * Método que salva ou atualiza as informações do cliente.
     * @param cliente
     * @return
     * @throws java.lang.Exception
     */
    public Cliente salvar(Cliente cliente) throws Exception {
        EntityManager entityManager = getEntityManager();
        try {
            // Inicia uma transação com o banco de dados.
            entityManager.getTransaction().begin();
            System.out.println("Salvando as informações do cliente.");
            /* Verifica se o cliente ainda não está salvo
             no banco de dados. */
        }
    }
}
```

```
if(cliente.getId() == null) {
    entityManager.persist(cliente);
} else {
    cliente = entityManager.merge(cliente);
}
// Finaliza a transação.
entityManager.getTransaction().commit();
} catch(Exception ex) {
    entityManager.getTransaction().rollback();
} finally {
    entityManager.close();
}
// Retorna o cliente salvo.
return cliente;
}

/**
 * Método que apaga as informações do cliente
 * do banco de dados.
 * @param id
 */
public void apagar(Long id) {
    EntityManager entityManager = getEntityManager();
    try {
        // Inicia uma transação com o banco de dados.
        entityManager.getTransaction().begin();
        // Consulta o cliente na base de dados através do seu ID.
        Cliente cliente = entityManager.find(Cliente.class, id);
        System.out.println("Excluindo o cliente: "
            + cliente.getNome());

        // Remove o cliente da base de dados.
        entityManager.remove(cliente);
        // Finaliza a transação.
        entityManager.getTransaction().commit();
    } catch(Exception ex) {
        entityManager.getTransaction().rollback();
    } finally {
        entityManager.close();
    }
}
```

```
}

/**
 * Consulta o cliente pelo ID.
 * @param id
 * @return
 */
public Cliente consultarPorId(Long id) {
    EntityManager entityManager = getEntityManager();
    Cliente cliente = null;
    try {
        //Consulta o cliente pelo ID.
        cliente = entityManager.find(Cliente.class, id);
    } finally {
        entityManager.close();
    }
    //Retorna o cliente consultado.
    return cliente;
}
}
```

Vamos desenvolver a interface gráfica em SWING que será responsável por chamar o ClienteDAO para executar as operações no banco de dados.



Código fonte da tela de cadastro do cliente.

Observação: este código não está completo, ele possui apenas implementado o código dos botões.

```
package pbc.jpa.exercicio4.tela;

import javax.swing.JOptionPane;
import pbc.jpa.exercicio4.dao.ClienteDAO;
import pbc.jpa.exercicio4.modelo.Cliente;
import pbc.jpa.exercicio4.modelo.Endereco;

/**
 * Classe utilizada para representar o cadastro do Cliente.
 */
public class CadastroCliente extends javax.swing.JFrame {

    private static final long serialVersionUID
        = -6011351657657723638L;

    public CadastroCliente() {
        initComponents();
    }

    /**
     * Código para montar a tela.
     */
    @SuppressWarnings("unchecked")
    private void initComponents() {
        //Código que monta a tela mostrada na imagem anterior.
    }

    /**
     * Botão que salva as informações do cliente.
     *
     * @param evt
     */
    private void botaoSalvarActionPerformed(
        java.awt.event.ActionEvent evt) {

        try {
            //Cria um objeto endereço;
            Endereco e = new Endereco();
            e.setEstado(this.estado.getText());
            e.setCidade(this.cidade.getText());
        }
    }
}
```

```
e.setBairro(this.bairro.getText());
e.setLogradouro(this.logradouro.getText());
e.setComplemento(this.complemento.getText());

//Cria um objeto cliente.
Cliente c = new Cliente();
c.setNome(this.nome.getText());
c.setEndereco(e);

//Salva o cliente.
ClienteDAO dao = new ClienteDAO();
c = dao.salvar(c);

JOptionPane.showMessageDialog(this, "Cliente "
+ c.getId() + " - " + c.getNome(), "INFORMAÇÃO",
 JOptionPane.INFORMATION_MESSAGE);
limparDados();
} catch (Exception ex) {
    JOptionPane.showMessageDialog(this, ex.getMessage(),
    "ERRO", JOptionPane.ERROR_MESSAGE);
}
}

/**
 * Botão que consulta as informações do cliente.
 *
 * @param evt
 */
private void botaoConsultarActionPerformed(
    java.awt.event.ActionEvent evt) {

try {
    ClienteDAO dao = new ClienteDAO();
    Cliente c = dao.consultarPorId(Long.valueOf(this.id.getText()));

    if(c != null) {
        this.nome.setText(c.getNome());
        this.estado.setText(c.getEndereco().getEstado());
        this.cidade.setText(c.getEndereco().getCidade());
    }
}
}
```

```
        this.bairro.setText(c.getEndereco().getBairro());
        this.logradouro.setText(c.getEndereco().getLogradouro())
    ;
        this.complemento.setText(c.getEndereco().getComplemento());
    } else {
        limparDados();
        JOptionPane.showMessageDialog(this,
            "Cliente não foi encontrado!",
            "ERRO", JOptionPane.ERROR_MESSAGE);
    }
} catch (NumberFormatException ex) {
    JOptionPane.showMessageDialog(this,
        "O campo código precisa ser um número inteiro",
        "ERRO", JOptionPane.ERROR_MESSAGE);
}
}

/**
 * Botão para limpar as informações do formulário
 * para cadastrar um novo cliente.
 *
 * @param evt
 */
private void botaoNovoActionPerformed(
    java.awt.event.ActionEvent evt) {

    limparDados();
}

/**
 * Botão para remover as informações referentes a um cliente.
 *
 * @param evt
 */
private void botaoApagarActionPerformed(
    java.awt.event.ActionEvent evt) {

    try {
        ClienteDAO dao = new ClienteDAO();
```

```
        dao.apagar(Long.valueOf(this.id.getText()));
        limparDados();
        JOptionPane.showMessageDialog(this,
            "As informações do cliente foram apagadas do sistema.",
            "INFORMAÇÃO", JOptionPane.INFORMATION_MESSAGE);
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this,
            "O campo código precisa ser um número inteiro",
            "ERRO", JOptionPane.ERROR_MESSAGE);
    }
}

/**
 * Limpa os dados do formulario.
 */
private void limparDados() {
    this.id.setText(null);
    this.nome.setText(null);
    this.estado.setText(null);
    this.cidade.setText(null);
    this.bairro.setText(null);
    this.logradouro.setText(null);
    this.complemento.setText(null);
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new CadastroCliente().setVisible(true);
        }
    });
}

private javax.swing.JTextField bairro;
private javax.swing.JButton botaoApagar;
private javax.swing.JButton botaoConsultar;
private javax.swing.JButton botaoNovo;
```

```
private javax.swing.JButton botaoSalvar;
private javax.swing.JTextField cidade;
private javax.swing.JTextField complemento;
private javax.swing.JTextField estado;
private javax.swing.JTextField id;
private javax.swing.JLabel labelBairro;
private javax.swing.JLabel labelCadastroCliente;
private javax.swing.JLabel labelCidade;
private javax.swing.JLabel labelCodigo;
private javax.swing.JLabel labelComplemento;
private javax.swing.JLabel labelEstado;
private javax.swing.JLabel labelLogradouro;
private javax.swing.JLabel labelNome;
private javax.swing.JTextField logradouro;
private javax.swing.JTextField nome;
}
```

Exercício 2

Neste exercício vamos desenvolver uma aplicação swing para implementar o seguinte requisito de sistema:

Precisamos controlar as vendas de instrumentos musicais, desenvolva uma aplicação para cadastrar, alterar, consultar pelo código e remover os instrumentos musicais (marca, modelo e preço). Também devemos cadastrar as vendas feitas para um cliente, onde o cliente (nome, cpf e telefone) pode comprar diversos instrumentos musicais. Não temos a necessidade de controlar o estoque dos produtos, pois apenas será vendido os itens que estão nas prateleiras.

Consultas com JPAQL

Utilizando JPA podemos também adicionar consultas personalizadas, para criação das consultas utilizamos a Java Persistence API Query Language (JPAQL) que é uma linguagem muito similar ao Structured Query Language (SQL) com a diferença que é mais voltado para orientação a objetos, onde facilita a navegação pelos objetos.

Nesse exemplo vamos abordar o seguinte cenário, empréstimo de livros para os clientes da biblioteca, temos as seguintes tabelas:

LIVRO

id	titulo	autor	isbn	paginas
1	Almoçando com Java	Sakurai	111-11-1111-111-1	325
2	Classes Java em fila indiana	Cristiano	222-22-2222-222-2	120
3	Java em todo lugar	Sakurai	333-33-3333-333-3	543
4	Viajando no Java	Cristiano	444-44-4444-444-4	210

EMPRESTIMO

id	livro_id	cliente_id	dataEmprestimo	dataDevolucao
1	1	1	10/08/2009	20/08/2009
2	3	2	15/08/2009	30/08/2009
3	3	1	01/09/2009	

CLIENTE

id	nome	cpf	telefone
1	Marcelo	333.333.333-33	9999-8888
2	Ana	222.222.222-22	7777-6666

Script do banco de dados:

```
CREATE TABLE Livro (
    id number(5) NOT NULL,
    titulo varchar2(200) NOT NULL,
    autor varchar2(200) NOT NULL,
    isbn varchar2(20) NOT NULL,
    paginas number(5) NOT NULL,
    PRIMARY KEY(id)
);
```

```

CREATE TABLE Cliente (
    id number(5) NOT NULL,
    nome varchar2(200) NOT NULL,
    cpf varchar2(14) NOT NULL,
    telefone varchar2(9) NOT NULL,
    PRIMARY KEY(id)
);

CREATE TABLE Emprestimo (
    id number(5) NOT NULL,
    livro_id number(5) NOT NULL,
    cliente_id number(5) NOT NULL,
    dataEmprestimo date NOT NULL,
    dataDevolucao date,
    PRIMARY KEY(id)
);

INSERT INTO Livro (id, titulo, autor, isbn, paginas)
VALUES (1, 'Almoçando com Java', 'Sakurai',
'111-11-1111-111-1', 325);
INSERT INTO Livro (id, titulo, autor, isbn, paginas)
VALUES (2, 'Classes Java em fila Indiana', 'Cristiano',
'222-22-2222-222-2', 120);
INSERT INTO Livro (id, titulo, autor, isbn, paginas)
VALUES (3, 'Java em todo lugar', 'Sakurai',
'333-33-3333-333-3', 543);
INSERT INTO Livro (id, titulo, autor, isbn, paginas)
VALUES (4, 'Viajando no Java', 'Cristiano',
'444-44-4444-444-4', 210);

INSERT INTO Cliente (id, nome, cpf, telefone)
VALUES (1, 'Marcelo', '333.333.333-33', '9999-8888');
INSERT INTO Cliente (id, nome, cpf, telefone)
VALUES (2, 'Ana', '222.222.222-22', '7777-6666');

```

Utilizando o EJB-QL podemos criar diversas formas de consultas, onde podemos especificar as projeções, associações, restrições e outros. Exemplos de consultas:

Consultar todos os empréstimos:

```
SELECT e  
FROM Emprestimo e
```

Consultar a quantidade de empréstimo por livro:

```
SELECT count(e)  
FROM Emprestimo e  
WHERE e.livro.id = :id
```

Consultar os empréstimos por título do livro:

```
SELECT e  
FROM Emprestimo e, Livro l  
WHERE e.livro.id = l.id  
AND l.titulo LIKE :titulo
```

As consultas podem ser criadas junto com as entidades:

```
package pbc.jpa.exemplo.modelo;  
  
import java.io.Serializable;  
import java.util.Date;  
import javax.persistence.CascadeType;  
import javax.persistence.Entity;  
import javax.persistence.FetchType;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
import javax.persistence.ManyToOne;  
import javax.persistence.NamedQueries;  
import javax.persistence.NamedQuery;  
import javax.persistence.Temporal;  
import javax.persistence.TemporalType;  
  
/**
```

```

 * Classe utilizada para representar o emprestimo
 * de um livro para um cliente.
 */

@NamedQueries({
    @NamedQuery(name = "Emprestimo.consultarTodos",
        query= "SELECT e FROM Emprestimo e"),
    @NamedQuery(name = "Emprestimo.qtdEmprestimosPorLivro",
        query = " SELECT count(e) " +
            " FROM Emprestimo e " +
            " WHERE e.livro.id = :id "),
    @NamedQuery(name = "Emprestimo.consultarTodosPorTituloLivro",
        query = " SELECT e " +
            " FROM Emprestimo e, Livro l " +
            " WHERE e.livro.id = l.id " +
            " AND l.titulo LIKE :titulo ")
})
@Entity
public class Emprestimo implements Serializable {
    private static final long serialVersionUID
        = 7516813189218268079L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @ManyToOne(cascade=CascadeType.REFRESH, fetch=FetchType.EAGER)
    private Livro livro;
    @ManyToOne(cascade=CascadeType.REFRESH, fetch=FetchType.EAGER)
    private Cliente cliente;
    @Temporal(TemporalType.DATE)
    private Date dataEmprestimo;
    @Temporal(TemporalType.DATE)
    private Date dataDevolucao;

    public Cliente getCliente() { return cliente; }
    public void setCliente(Cliente cliente) {
        this.cliente = cliente;
    }

    public Date getDataDevolucao() { return dataDevolucao; }
    public void setDataDevolucao(Date dataDevolucao) {

```

```
    this.dataDevolucao = dataDevolucao;
}

public Date getDataEmprestimo() { return dataEmprestimo; }
public void setDataEmprestimo(Date dataEmprestimo) {
    this.dataEmprestimo = dataEmprestimo;
}

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

public Livro getLivro() { return livro; }
public void setLivro(Livro livro) { this.livro = livro; }
}
```

Interface Query

A interface **Query** é responsável por:

- Fazer as consultas;
- Executar updates;
- Passar parâmetros para consulta;
- Pegar um simples resultado;
- Pegar uma lista de resultados.

Para executar uma consulta utilizamos a interface javax.persistence.Query e criamos uma Query a partir do EntityManager, exemplo:

```
public List<Emprestimo> consultarTodos() {  
    EntityManager em = getEntityManager();  
    Query query = em.createNamedQuery("Emprestimo.consultarTodos")  
;  
    return query.getResultList();  
}
```

Nesse exemplo estamos utilizando a EntityManager para criar uma consulta nomeada através do método **getNamedQuery** e passamos o nome da consulta "Emprestimo.consultarTodos" declarado na entidade.

Através da EntityManager podemos utilizar o método **createQuery** para criar a consulta e já declarar todo o código dela, podendo criar uma consulta um pouco mais personalizada:

```
Query query = em.createQuery("SELECT c FROM Cliente c");
```

Também podemos criar uma consultas nativa para uma base de dados específica:

```
Query query = em.createNativeQuery("SELECT * FROM Cliente");
```

Para executar uma consulta podemos ter como resposta um simples objeto ou uma lista de objetos.

Neste exemplo vamos executar uma consulta que retorna uma lista de objetos:

```
Query query = em.createNamedQuery("Cliente.consultarTodosClientes");
List<Cliente> clientes = (List<Cliente>) query.getResultList();
```

Quando utilizamos o método **getResultSet** da interface Query é retornado uma lista de entidades.

Consulta que retorna um único objeto:

```
Query query = em.createNamedQuery("Emprestimo.qtdEmprestimosPorLivro");
Long quantidade = (Long) query.getSingleResult();
```

Quando utilizamos o método **getSingleResult** da interface Query é retornado apenas um objeto. Este método pode lançar também a exceção **javax.persistence.NoResultException** quando a consulta não retorna nenhum resultado ou também uma exceção **javax.persistence.NonUniqueResultException** quando você espera retornar apenas um objeto e a consulta acaba retornando mais de um objeto.

Através da interface Query podemos passando parâmetros para a consulta através do método **setParameter**:

```
public List<Emprestimo> consultarTodosPorTituloLivro
(String tituloLivro) {

    EntityManager entityManager = getEntityManager();
    Query query = entityManager.createNamedQuery
        ("Emprestimo.consultarTodosPorTituloLivro");

    query.setParameter("titulo", tituloLivro);

    return query.getResultList();
}
```

Exemplo de classe DAO que executa as operações: salvar, alterar, remover, consultar por id, consultar todos os empréstimos, consultar a quantidade de empréstimos de um livro e consultar todos os empréstimos de um livro.

```
package pbc.jpa.exemplo.dao;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import pbc.jpa.exemplo.modelo.Emprestimo;

/**
 * Classe utilizada para representar as
 * operações sobre a entidade emprestimo.
 */

public class EmprestimoDAO {
    public EntityManager getEntityManager() {
        EntityManagerFactory factory = null;
        EntityManager entityManager = null;
        try {
            factory = Persistence.createEntityManagerFactory
                ("UnidadeDePersistencia");
            entityManager = factory.createEntityManager();
        } finally {
```

```
        factory.close();
    }
    return entityManager;
}

public Emprestimo consultarPorId(Long id) {
    EntityManager entityManager = getEntityManager();
    Emprestimo emprestimo = null;

    try {
        emprestimo = entityManager.find(Emprestimo.class, id);
    } finally {
        entityManager.close();
    }

    return emprestimo;
}

public Emprestimo salvar(Emprestimo emprestimo) {
    EntityManager entityManager = getEntityManager();
    try {
        entityManager.getTransaction().begin();
        if(emprestimo.getId() == null) {
            entityManager.persist(emprestimo);
        } else {
            entityManager.merge(emprestimo);
        }
        entityManager.flush();
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        entityManager.getTransaction().rollback();
    } finally {
        entityManager.close();
    }

    return emprestimo;
}

public void apagar(Long id) {
    EntityManager entityManager = getEntityManager();
```

```
try {
    entityManager.getTransaction().begin();
    Emprestimo emprestimo = entityManager.
        find(Emprestimo.class, id);
    entityManager.remove(emprestimo);
    entityManager.flush();
    entityManager.getTransaction().commit();
} catch (Exception ex) {
    entityManager.getTransaction().rollback();
} finally {
    entityManager.close();
}

public List<Emprestimo> consultarTodos() {
    EntityManager entityManager = getEntityManager();
    Query query = entityManager.createNamedQuery
        ("Emprestimo.consultarTodos");
    return query.getResultList();
}

public Long getQtdEmprestimosPorLivro(Long id) {
    EntityManager entityManager = getEntityManager();
    Query query = entityManager.createNamedQuery
        ("Emprestimo.qtdEmprestimosPorLivro");
    query.setParameter("id", id);
    return (Long) query.getSingleResult();
}

public List<Emprestimo> consultarTodosPorTituloLivro
    (String tituloLivro) {
    EntityManager entityManager = getEntityManager();
    Query query = entityManager.createNamedQuery
        ("Emprestimo.consultarTodosPorTituloLivro");
    query.setParameter("titulo", tituloLivro);
    return query.getResultList();
}
```


Exercícios

Exercício 1

Seguindo como base o exercício anterior, agora vamos criar uma aplicação para realizar a venda de um produto para um cliente.

Utilize as tabelas da figura a seguir para modelar o sistema.

Tabela Produto

id	descricao	preco
1	Mochila p/ notebook	130,90
2	Mouse bluetooth	99,50
3	Cabo de segurança	25,00
4	Pen drive 16Gb	110,99

Tabela Venda

*	id	cliente_id	produto_id	quantidade
1	1		1	1
2	1		2	1
3	2		4	10
4	3		3	3

Tabela Cliente

id	nome	cpf
1	Ana Maria	111.111.111-11
2	Paulo da Silva	222.222.222-22
3	Carlos Luis	333.333.333-33

Crie as entidades Produto, Cliente e Venda, DAOs e uma classe para teste.

Observação: Se quiser pode começar com uma carga prévia de clientes e produtos fazendo assim apenas a parte das vendas.

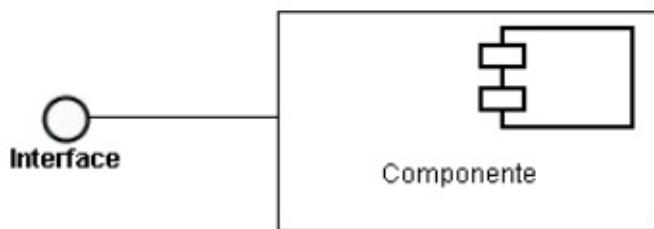
Após isso amplie a aplicação para que seja possível vender vários produtos para um cliente.

Enterprise JavaBeans

O Enterprise JavaBeans 3.0 (EJB) é um framework que simplifica o processo de desenvolvimento de componentes para aplicações distribuídas em Java.

Utilizando EJB o desenvolvedor pode dedicar mais tempo ao desenvolvimento da lógica de negócio, preocupando-se menos com a transação, a persistência de dados, os serviços de rede, entre outros.

Componente



Um componente é um pedaço reutilizável de software, normalmente criado de forma genérica e independente de sistemas podendo ser um único arquivo ou um conjunto de arquivos com um objetivo em comum.

A reutilização de código tem alguns benefícios muito significantes como redução do custo e tempo de desenvolvimento, uma vez que o componente está pronto e testado, caso você precise em outro lugar do software ou em software diferente utilizar a mesma funcionalidade, basta reutilizá-lo novamente.

"Um componente é uma parte do sistema não trivial, quase independente e substituível, desempenhando funções claras no contexto de uma arquitetura bem definida. Um componente obedece e provê a realização física de um conjunto de interfaces que especificam alguma abstração lógica." - **Definição de Componente por Grady Booch**

Prós e contras da programação baseado em componentes

Prós

- Reutilização do componente, uma vez criado e homologado ele pode ser utilizado em vários sistemas diferentes;
- Redução de custo, depois que o componente já foi criado ele é reaproveitado em outros sistemas;
- Desenvolvimento em paralelo, no projeto pode ter diversas equipes trabalhando no mesmo projeto e criando componentes separados.

Contras

- Encontrar o componente certo, encontrar qual parte do sistema que pode ser desenvolvido de forma a ser reaproveitado em outros sistemas ou procurar um componente que tenha qualidade para ser utilizado no seu sistema;
- Qualidade do componente, quando você resolve utilizar um componente que não foi desenvolvido pela sua empresa, é preciso testar esse componente para saber se ele vai ser compatível com o seu sistema;
- Custo de tempo de desenvolvimento, na criação do componente ele precisa ser feito de forma genérica, para que possa ser reaproveitado e isso nem sempre é fácil de fazer.

Container EJB

Para utilizar o EJB é necessário um Servidor de Aplicações Web que possua um Container EJB, pois é este container que gerenciará todo o ciclo de vida de um componente EJB. Existem diversos servidores de aplicações web que suportam EJB 3.0 como o JBoss, GlassFish, WebSphere, entre outros.

Quando a aplicação utiliza o container EJB, diversos serviços são disponibilizados para a aplicação EJB como, por exemplo:

- Gerenciamento de transação
- Segurança
- Controle de concorrência
- Serviço de rede
- Gerenciamento de recursos
- Persistência de dados
- Serviço de mensageria

Tipos de EJB

Existe três tipos de EJB que podemos utilizar, cada um com sua finalidade:

- **Session Beans** – Utilizado para guardar a lógica de negocio da aplicação;
- **Message-Driven Bean** – Utilizado para troca de mensagens;
- **Entity Bean (até versão EJB 2.x)** – Utilizado para representar as tabelas do banco de dados.

Session Bean

O Session Bean é um tipo de componente que executa a lógica de negócio da aplicação; por meio dele podem ser criados componentes que podem ser acessados por várias aplicações diferentes escritas em Java. E os session beans podem ser criados de duas formas: stateless e stateful, cada um com suas características que serão detalhadas a seguir.

Como o ciclo de vida de um componente EJB é controlado pelo Container EJB que existe no servidor web, não é possível criar um novo EJB através da palavra reservada new da linguagem Java, por exemplo:

```
MeuEJB ejb = new MeuEJB(); // ERRADO!!!
```

O código anterior compila e executa, mas não funciona corretamente, pois não foi criado pelo Container EJB, então não ganha todos os benefícios que o framework EJB disponibiliza para os componentes distribuídos. A forma correta é solicitar ao servidor web a criação de um componente, por exemplo:

```
InitialContext ctx = new InitialContext();
MeuEJB ejb = (MeuEJB) ctx.lookup("nomeComponente");
```

Deste modo, o componente EJB é instanciado dentro do Container EJB e assim ganhará todos os benefícios como uma conexão com o banco de dados, acesso a uma fila de mensageria, entre outros.

Stateless Session Bean

O Stateless Session Bean tem o ciclo de vida que dura apenas o tempo de uma simples chamada de método, sendo assim ao solicitar um componente deste tipo para o servidor web e executar o método desejado, este componente já pode ser destruído.

Quando é solicitado um Stateless Session Bean ao Container EJB, por meio do `lookup()`, ocorrem os seguintes passos:

- O Container EJB cria uma instância do Stateless Session Bean;
- Executa o método chamado;
- Se houver retorno no método, este objeto retornado é enviado para quem chamou o método do EJB;
- Depois o Container EJB destrói o objeto do componente EJB.

O Stateless também não mantém o estado entre as chamadas de método, deste modo, caso tenha algum atributo dentro do EJB este atributo é perdido toda vez que o EJB é chamado.

O problema do Stateless Session Bean é que a cada requisição um novo bean pode ser criado, por isso normalmente é configurado no container EJB, para gerar um pool de instâncias do EJB.

Um Stateless Session Bean é formado por uma interface **Remota**, **Local**, ou ambas, e mais uma implementação destas interfaces.

Quando a aplicação cliente do componente EJB está em outro servidor web ou é uma aplicação desktop, faz-se necessário a criação de uma interface **Remota**. Para isto, é necessário criar uma interface para o componente EJB e adicionar a anotação **javax.ejb.Remote**.

No exemplo, a seguir, será criada uma interface chamada **ServicoRemote** e adicionado a anotação **@Remote** para informar que será a interface remota de um componente EJB. Na interface de um EJB é declarada apenas a assinatura dos métodos que a implementação deste EJB precisa ter.

```
package metodista.pbc.ejb.exemplo;

import javax.ejb.Remote;

/**
 * Interface remota para o EJB de Serviço.
 */
@Remote
public interface ServicoRemote {
    /**
     * Método utilizado para abrir uma solicitação
     * para um aluno.
     * @param tipoServico - Serviço que será aberto.
     * @param aluno - Aluno que solicitou o serviço.
     */
    public abstract void abrirServico(String tipoServico,
        Aluno aluno);
}
```

Também há a opção de criar uma interface **Local** que é utilizada quando for necessário fazer uma chamada a um EJB que está no mesmo servidor web da aplicação cliente; para isto é necessário adicionar a anotação **javax.ejb.Local** na interface local.

No exemplo, a seguir, será criada uma interface chamada **ServicoLocal** e adicionado a anotação **@Local** para informar que ela será a interface local de um EJB. Na interface de um EJB é declarada a assinatura dos métodos que a implementação deste EJB precisa ter.

```
package metodista.pbc.ejb.exemplo;

import java.util.List;
import javax.ejb.Local;

/**
 * Interface local para o EJB de serviço.
 */
@Local
public interface ServicoLocal {
    /**
     * Método utilizado para buscar todos os serviços
     * de um aluno.
     * @param aluno - Aluno que abriu os serviços.
     * @return Lista de serviços de um aluno.
     * @throws Exception
     */
    public abstract List<Servico> buscarServicos(Aluno aluno)
        throws Exception;
}
```

Mesmo criando uma interface remota o componente EJB pode ser acessado localmente, então escolher entre Remoto e Local vai muito da situação. Se precisar acessar o componente apenas dentro do mesmo servidor web vale a pena deixar Local, mas se houver a necessidade de utilizar em outras aplicações distribuídas em vários servidores então é preciso criar a interface remota.

Também é necessário criar uma classe que implemente uma ou ambas interfaces. Para que esta classe seja reconhecida pelo Container EJB, como um EJB Session Bean do tipo Stateless, é necessário adicionar a anotação **javax.ejb.Stateless**.

No exemplo, a seguir, foi criada uma classe **ServicoBean** que implementa a interface **ServicoRemote**; note a anotação **@Stateless** que define o tipo deste EJB como Stateless Session Bean:

```
package metodista.pbc.ejb.exemplo;

import javax.ejb.Stateless;

/**
 * Stateless session bean que implementa as funcionalidades
 * referentes a interface remota.
 */
@Stateless
public class ServicoBean implements ServicoRemote {
    /**
     * Método utilizado para abrir uma solicitação para um aluno.
     * @param tipoServiço - Serviço que será aberto.
     * @param aluno - Aluno que solicitou o serviço.
     */
    public void abrirServiço(String tipoServiço, Aluno aluno) {
        /* Código que abre a solicitação de serviço para um aluno. */

    }
}
```

Stateful Session Bean

O Stateful session bean é bem parecido com o Stateless. A diferença é que ele pode viver no container EJB enquanto a sessão do usuário estiver ativa, portanto a partir do momento em que o componente é chamado pelo cliente, ele passa a ser exclusivamente do cliente que o chamou até que o cliente deixe a aplicação.

Outra diferença fundamental é que ele mantém o estado durante as chamadas dos métodos preservando os valores das variáveis.

O problema do Stateful Session Bean é que pode haver um número muito grande de instâncias na memória do servidor por ser associado a um único cliente, portanto cada vez que um cliente conecta a aplicação e precisa usar este tipo de EJB, um novo objeto EJB é criado para atendê-lo; desta forma se for necessário é possível manter o estado dos atributos do EJB para diversas chamadas de métodos feitas nele.

A implementação do Stateful também é muito parecida com a implementação do Stateless, a única diferença é que a classe do tipo Stateful utiliza a anotação **javax.ejb.Stateful**.

No exemplo, a seguir, é criada uma classe chamada **ServicoBean** que implementa a interface ServicoRemote, note a anotação **@Stateful** que define o tipo deste EJB como Stateful Session Bean:

```
package metodista.pbc.ejb.exemplo;

import javax.ejb.Stateful;

/**
 * Stateful session bean que implementa as funcionalidades
 * referentes a interface remota.
 */
@Stateful
public class ServicoBean implements ServicoRemote {
    /**
     * Método utilizado para abrir uma solicitação para um aluno.
     * @param tipoServiço - Serviço que será aberto.
     * @param aluno - Aluno que solicitou o serviço.
     */
    public void abrirServiço(String tipoServiço, Aluno aluno) {
        /* Código que abre a solicitação de serviço para um aluno. */

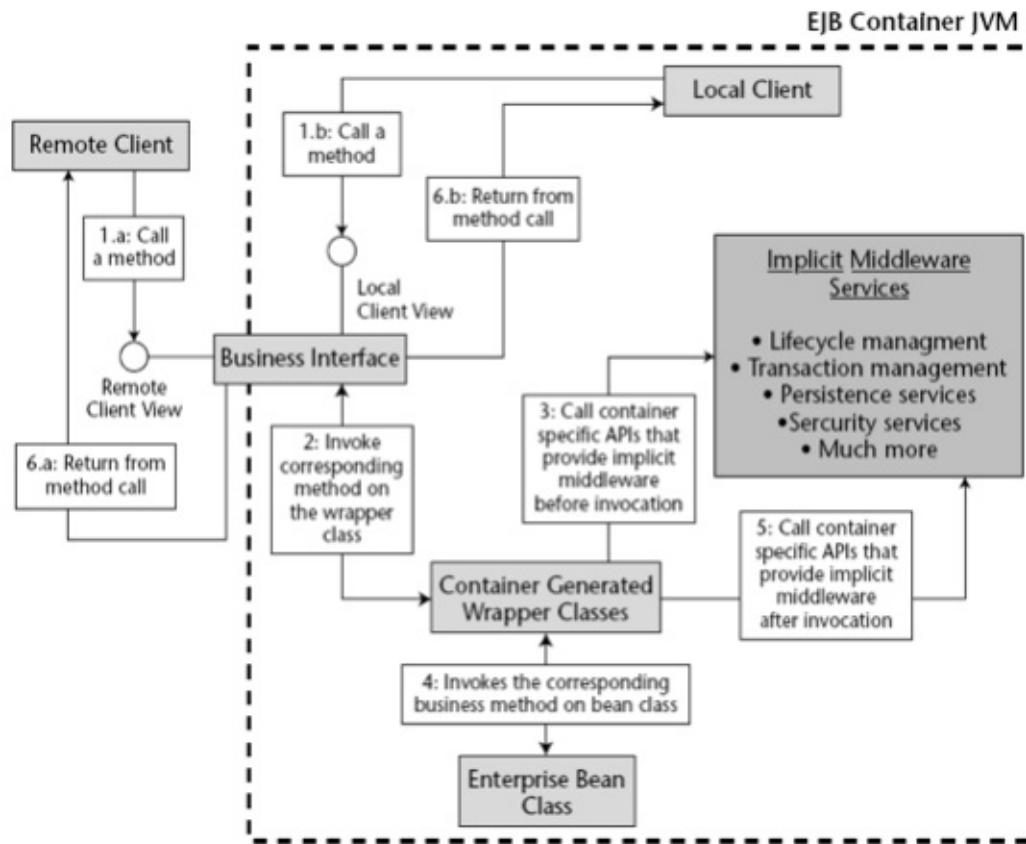
    }
}
```

Também é possível fazer um pool de Stateful Session Bean, mas alguns problemas podem ocorrer, como, por exemplo, acabar a memória do servidor. Para limitar a quantidade de instâncias na memória o container EJB pode guardar o estado da conversa (valor dos atributos do EJB) no HD ou em base de dados, isto é chamado de passivation.

Quando o cliente original do bean fizer uma requisição para um componente que foi guardado, o Container EJB traz o objeto EJB de volta para a memória, isto é chamado de activation.

Uma coisa importante é que quando o bean faz parte de uma transação, este bean não pode passar por passivation até terminar a transação.

A figura a seguir mostra o fluxo de uma chamada feita por um cliente remoto ou cliente local a um componente EJB.



Modelo de programação EJB 3.0 (BROSE, G.; SILVERMAN, M.; SRIGANESH, R. P., 2006, p.106)

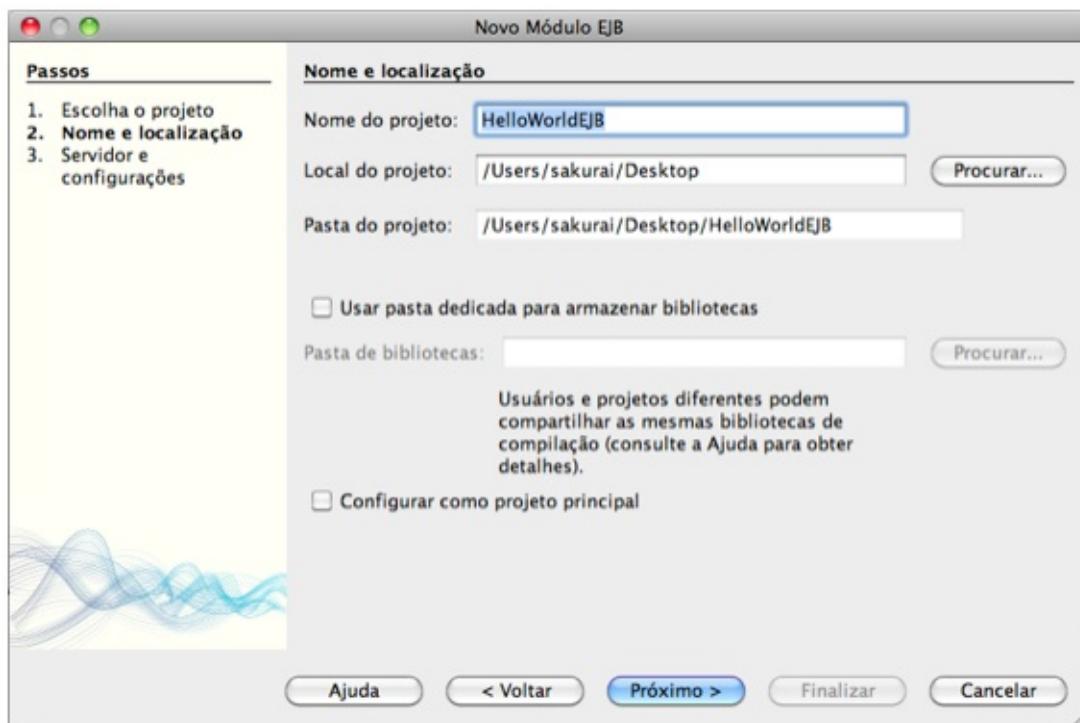
Criando um projeto EJB no NetBeans

Criando o módulo EJB

Para começar crie um **Novo Projeto**, na aba **Categorias** selecione **Java EE** e na aba **Projetos** selecione **Módulo EJB**, conforme a figura **ProjetoEJB01**.



Clique em **Próximo >** para continuar a criação do projeto. Na tela de **Novo Módulo EJB** apresentada na figura a seguir, no campo **Nome do projeto** digite **HelloWorldEJB** e informe o **Local do projeto** de sua preferência.



Clique em **Próximo >** para continuar a criação do projeto. Agora selecione o **Servidor** que será usado para publicar o projeto EJB nesse exemplo usaremos o **Glassfish Server 3.1** e a **Versão do Java EE** usada será a **Java EE 6**, conforme a figura a seguir:



Clique em **Finalizar** para terminar a criação do projeto. A Figura a seguir apresenta como é a estrutura inicial do projeto gerado pelo NetBeans:



Criando um EJB Stateless Session Bean

Crie a interface **HelloWorldRemote** no pacote **pbc.ejb** e adicione a assinatura do método **public abstract String ola()**.

```
package pbc.ejb;

import javax.ejb.Remote;

@Remote
public interface HelloWorldRemote {
    public abstract String ola();
}
```

Nesta interface adicione a anotação **@Remote** para informar que ela será acessada remotamente.

Agora vamos implementar esta interface, crie uma classe chamada **HelloWorldBean** no mesmo pacote e implemente o método **ola()**.

```

package pbc.ejb;

import javax.ejb.Stateless;

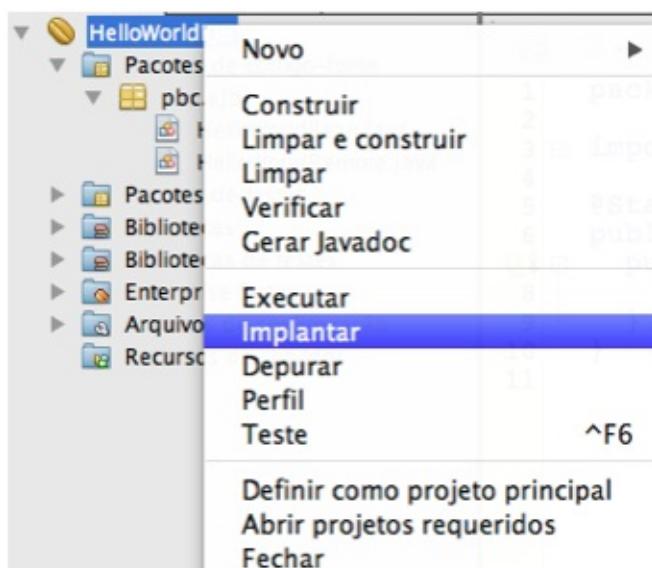
@Stateless
public class HelloWorldBean implements HelloWorldRemote {
    public String ola() {
        return "Ola mundo com EJB.";
    }
}

```

Nesta classe adicione a anotação **@Stateless** para informar que ela será um componente EJB do tipo Stateless Session Bean.

Publicando o projeto EJB no Glassfish

Após criar o projeto EJB é necessário publicá-lo em um servidor web para poder acessar os componentes EJB. Para isso clique com o botão direito sobre o projeto **HelloWorldEJB** e escolha a opção **Implantar**, conforme a Figura **ProjetoEJB05**; se o servidor web GlassFish não estiver iniciado, o NetBeans irá iniciá-lo e publicará o projeto EJB nele.



Depois que terminar de iniciar o Glassfish, aparecerá a seguinte mensagem no console de saída:

```
INFO: Portable JNDI names for EJB HelloWorldBean :  
[java:global/HelloWorldEJB>HelloWorldBean!pbc.ejb.HelloWorldRemote,  
java:global/HelloWorldEJB>HelloWorldBean]  
INFO: Glassfish-specific (Non-portable) JNDI names for EJB  
HelloWorldBean : [pbc.ejb.HelloWorldRemote#pbc.ejb.HelloWorldRemote,  
pbc.ejb.HelloWorldRemote]  
INFO: HelloWorldEJB was successfully deployed in 11.478  
milliseconds.
```

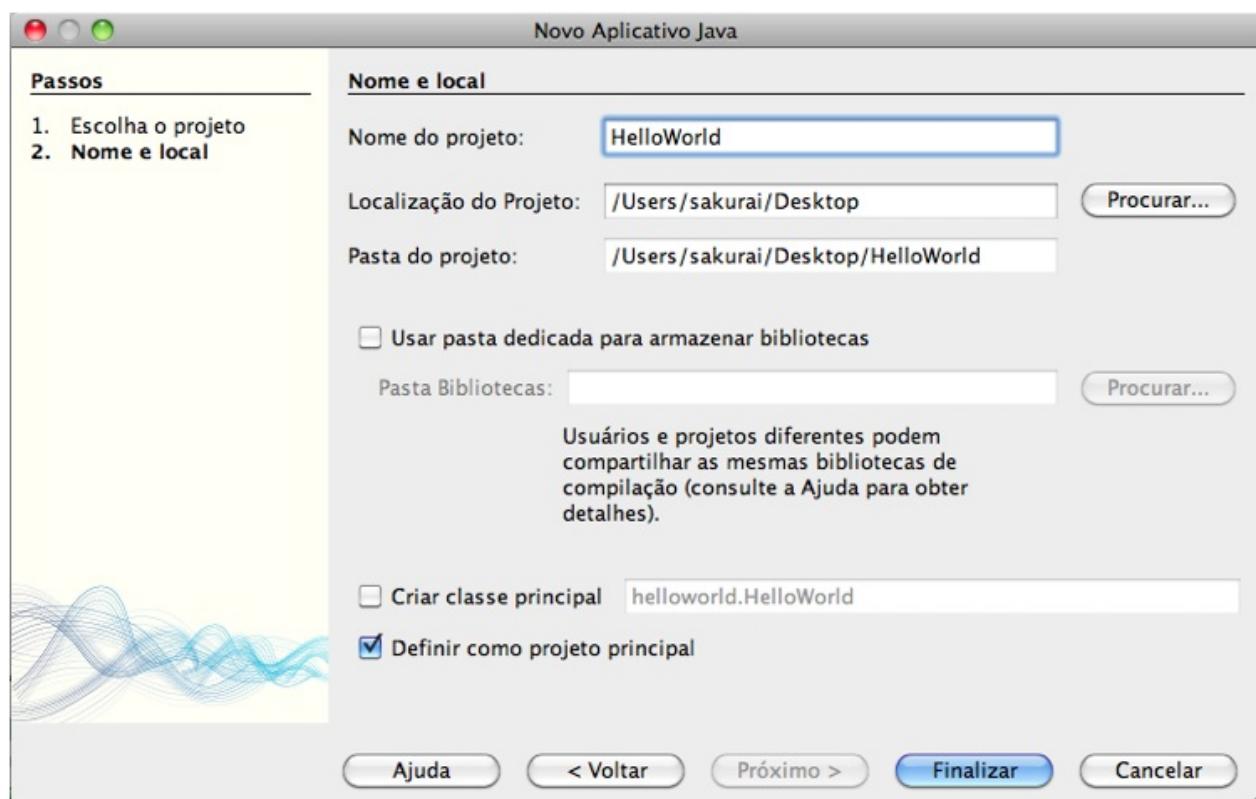
Caso ocorra algum problema ao tentar publicar a aplicação no servidor web GlassFish, será apresentado neste console as mensagens dos erros.

Criando um Projeto Java Console para testar o EJB

Para testar o componente EJB crie um projeto console. No NetBeans crie um **Novo projeto**, na aba **Categorias** selecione **Java**, e na aba **Projetos** selecione **Aplicativo Java** e clique em **Próximo**, conforme a figura a seguir:

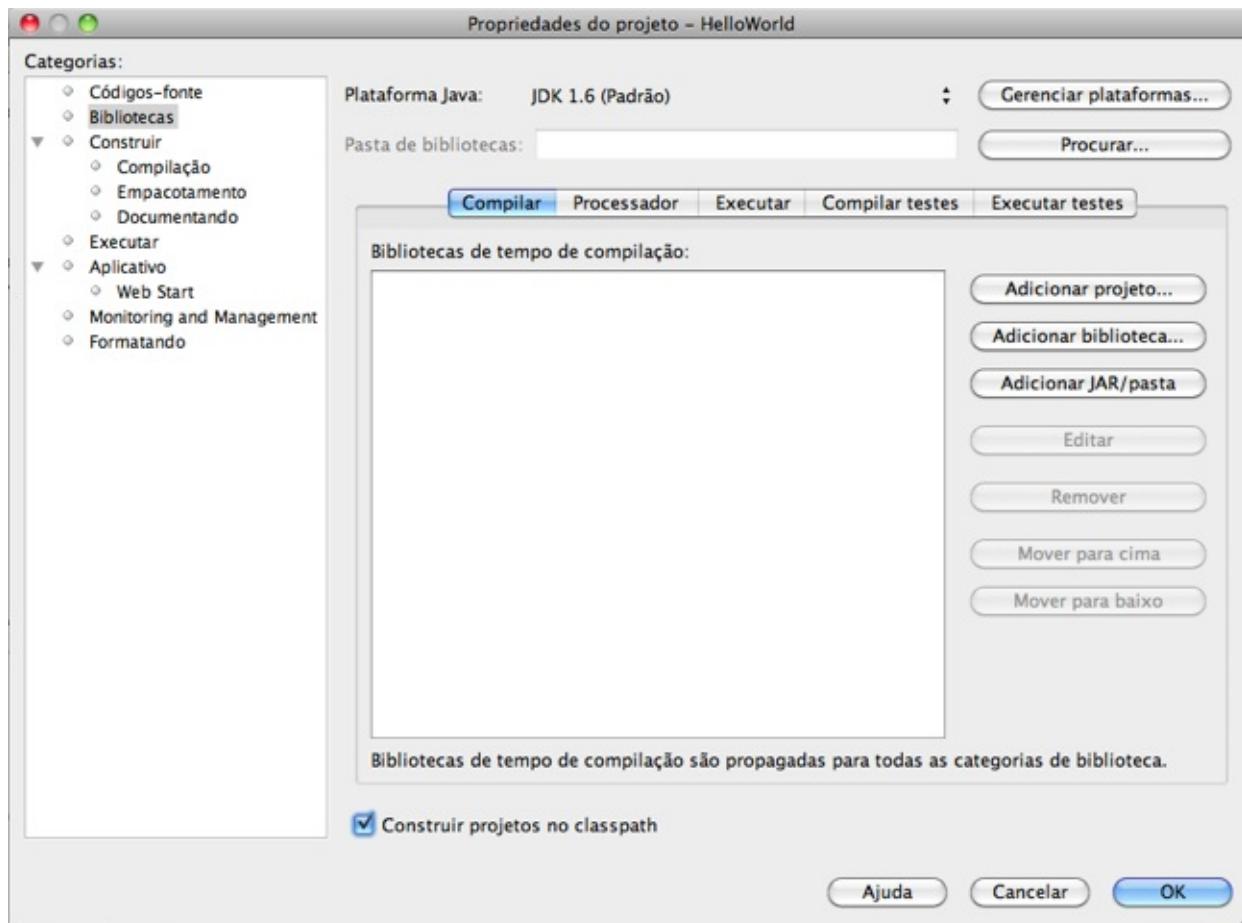


Na janela **Novo Aplicativo Java**, deixe o **Nome do projeto** como **HelloWorld** e na **Localização do projeto**, defina a mesma pasta que foi criado o projeto EJB **HelloWorldEJB**, conforme a figura a seguir:



Antes de começar a desenvolver o cliente do EJB, é necessário adicionar algumas bibliotecas (jar) no projeto, estas bibliotecas serão utilizadas para fazer o lookup do EJB.

Clique com o botão direito no projeto **HelloWorld** e escolha o item **Propriedades**. Na tela de **Propriedades do projeto**, apresentada na figura a seguir, selecione a categoria **Bibliotecas** e clique no botão **Adicionar JAR / Pasta**.



Na janela **Adicionar JAR / Pasta** navegue até a pasta de instalação do Glassfish e entre no diretório **/glassfish-vX/glassfish/modules/**, conforme apresentado na figura a seguir:



Selecione **gf-client.jar** ou **gf-client-module.jar** e clique em **Selecionar**.

Também é preciso adicionar uma referência entre os projetos. Na mesma tela de **Biblioteca**, clique no botão **Adicionar projeto...** e selecione o projeto **HelloWorldEJB** e clique em **OK**. Esta referência entre os projetos serve para referenciar as classes e interfaces, como, por exemplo, a interface **HelloWorldRemote**.

Na pasta **Pacotes de códigos-fontes**, crie a classe **TesteEJB** no pacote **pbc.teste**, que será utilizada para testar o EJB **HelloWorldRemote**.

```
package pbc.teste;

import pbc.ejb.HelloWorldRemote;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * Classe utilizada para testar o EJB de HelloWorldRemote.
 */
public class TesteEJB {
    public static void main(String[] args) {
        try {
            //Método que faz o lookup para encontrar o EJB de HelloWorldRemote.
            InitialContext ctx = new InitialContext();
            HelloWorldRemote ejb = (HelloWorldRemote) ctx.lookup
                ("pbc.ejb.HelloWorldRemote");

            System.out.println(ejb.ola());

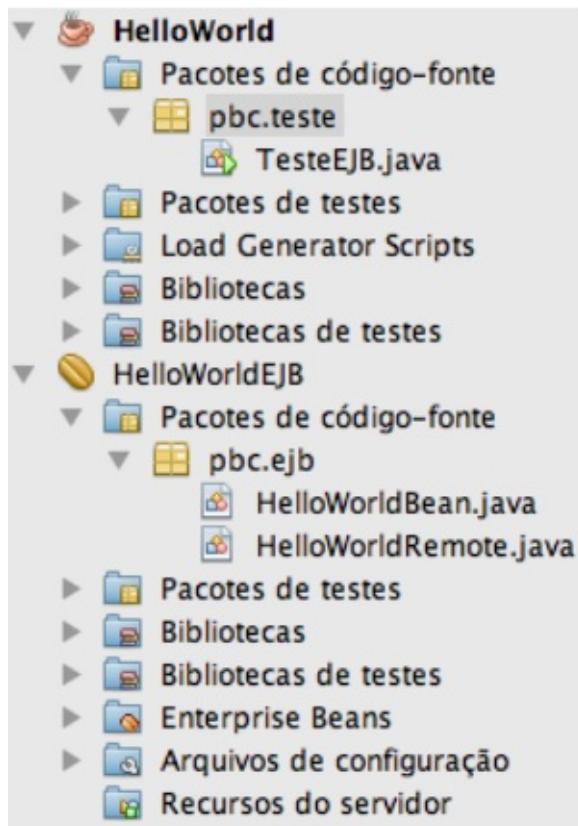
        } catch (NamingException ex) {
            ex.printStackTrace();
            System.out.println("Não encontrou o EJB.");
        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println(ex.getMessage());
        }
    }
}
```

No método **main()** foi criado um objeto do tipo **javax.naming.InitialContext** que possui o método **lookup** utilizado para obter uma instância do EJB, após pedir uma instância do EJB `ctx.lookup("pbc.ejb.HelloWorldRemote")` é chamado o método **ola()**, que devolve a mensagem definida no componente EJB.

Repare que para pedir um EJB para o GlassFish é utilizado o método **lookup** passando o **nome completo da classe remota** do EJB, também é possível fazer o **lookup** especificando o projeto e módulo, por exemplo: `java:global[/<app-`

```
name>] /<module-name>/<bean-name> .
```

A figura a seguir apresenta a estrutura atual do projeto.



Testando o EJB

Para testar o EJB, execute a classe **TesteEJB** do projeto **HelloWorld**. Será apresentada a seguinte saída no console.

```
Ola mundo com EJB.
```

Se o Glassfish estiver instalado em outra máquina ou a porta padrão **3700** for alterada é possível criar um arquivo de propriedades através da classe **java.util.Properties** para definir estas informações, como por exemplo:

```
package pbc.teste;

import pbc.ejb.HelloWorldRemote;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * Classe utilizada para testar o EJB de HelloWorldRemote.
 */
public class TesteEJB {
    public static void main(String[] args) {
        try {
            /* Monta um objeto Properties com as informações
               para localizar o Glassfish. */
            Properties props = new Properties();
            props.put("org.omg.CORBA.ORBInitialHost", "localhost");
            props.put("org.omg.CORBA.ORBInitialPort", "3700");

            /* Método que faz o lookup para encontrar o EJB
               de HelloWorldRemote. */
            InitialContext ctx = new InitialContext(getProperties());
            HelloWorldRemote ejb = (HelloWorldRemote) ctx.lookup
                ("pbc.ejb.HelloWorldRemote");

            System.out.println(ejb.ola());
        } catch (NamingException ex) {
            ex.printStackTrace();
            System.out.println("Não encontrou o EJB.");
        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println(ex.getMessage());
        }
    }
}
```

Melhorando o lookup

Para melhorar a forma de fazer lookup existe um padrão de projeto chamado **Service Locator**, que consiste em deixar genérica a forma como é feito o lookup do EJB.

O padrão **Service Locator** serve para separar a lógica envolvida no lookup das classes que precisam utilizar o EJB.

A seguir, será criada uma classe chamada **ServiceLocator**, esta classe será responsável por fazer um lookup de forma genérica, note que a natureza desta classe é altamente coesa, ou seja seu objetivo é bem definido: fazer o lookup do EJB.

```
package pbc.util;

import java.util.Properties;
import javax.naming.InitialContext;

/**
 * Classe utilizada para fazer um lookup genérico do EJB.
 */
public class ServiceLocator {

    /**
     * Propriedades do Glassfish.
     */
    private static Properties properties = null;

    /**
     * Monta um objeto Properties com as informações para
     * localizar o glassfish.
     * @return
     */
    private static Properties getProperties() {
        if (properties == null) {
            properties = new Properties();
            properties.put("org.omg.CORBA.ORBInitialHost", "localhost");
            properties.put("org.omg.CORBA.ORBInitialPort", "3700");
        }
        return properties;
    }
}
```

```
}

/**
 * Método que faz o lookup generico de um EJB.
 *
 * @param <T>
 * @param clazz - Classe que representa a interface do
 *   EJB que será feito o lookup.
 * @return
 * @throws Exception
 */
public static <T> T buscarEJB(Class<T> clazz) throws Exception
{
    /* Cria o initial context que faz via JNDI a procura do EJB.
 */
    InitialContext ctx = new InitialContext(getProperties());

    /* Pega o nome completo da interface utilizando
       reflection, faz o lookup do EJB e o retorno do EJB. */
    return (T) ctx.lookup(clazz.getName());
}
```

Agora a classe que utiliza o EJB, não precisa mais saber como fazer o **lookup** do EJB, precisa apenas chamar o método **buscarEJB** da classe **ServiceLocator** passando como parâmetro a **Class** da interface do EJB.

```
package pbc.teste;

import pbc.ejb.UsuarioRemote;
import aula.util.ServiceLocator;

/**
 * Classe utilizada para testar o EJB de HelloWorldRemote.
 */
public class TesteEJB {
    public static void main(String[] args) {
        try {
            HelloWorldRemote ejb = (HelloWorldRemote)
                ServiceLocator.buscarEJB(HelloWorldRemote.class);

            System.out.println(ejb.ola());
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

Existem outras vantagens de utilizar o **Service Locator** para separar a lógica do lookup, por exemplo: um Proxy para caso seja necessário gravar algum log quando criar o EJB ou quando seus métodos são chamados.

Exercícios

Exercício 1

Crie um projeto EJB chamado CalculadoraEJB, dentro dele crie o seguinte componente:

- Crie um componente EJB para realizar as operações de uma calculadora (soma, subtração, divisão e multiplicação).

Crie um projeto Java chamado CalculadoraCliente que dever ter uma interface Swing para utilizar as operações do Calculadora EJB.

Exercício 2

Crie um projeto EJB chamado LivrariaEjb, dentro dele crie o seguinte componente EJB:

- Crie um EJB para gerenciar (salvar, alterar, consultar (por autor, título ou isbn) e excluir) livros.
- Também crie um método para efetuar a venda de livros.

Crie um projeto chamado Livraria para testar o EJB criado, este projeto pode ser Console, Swing ou Web (utilizando Servlet).

Observação: Se a aplicação de teste for Console ou Desktop, utilize o Service Locator para separar a lógica que busca os EJBs.

Exercício 3

O restaurante JavaFood tem um cardápio, dentro do cardápio existem diversos itens como, por exemplo: entradas, pratos quentes, bebidas, sobremesas, etc. Para cada item eu tenho as informações nome, descrição, preço.

Quando o cliente vem ao restaurante ele pode fazer o pedido de um ou muitos itens do cardápio, enquanto o pedido não estiver pronto, o cliente pode cancelá-lo a qualquer momento.

OBS: Não é necessário representar o cardápio. Os itens do cardápio podem ser previamente carregados na base de dados.

Crie as classes de negocio para representar o pedido com os itens do cardápio, adicione as anotações de JPA.

Crie uma classe DAO onde temos as operações de fazer pedido ou cancelamento.

Crie um EJB que possui a lógica de negocio para fazer o pedido ou cancelamento de algum item.

Usando a anotação @EJB

Quando queremos acessar um **EJB A** a partir de outro **EJB B** podemos utilizar a anotação **@EJB**, através desta anotação o **Container EJB** faz o lookup do **EJB A** e guarda sua referencia no **EJB B**, isso é chamado de Injeção de Dependência.

O Servidor Web Glassfish estende um pouco mais a utilização desta anotação, se temos um cliente que é uma Aplicação Web, ao utilizarmos Servlets podemos também usar a anotação **@EJB** para que seja feito o lookup do EJB.

No exemplo a seguir, crie uma **Servlet** e utilize a anotação **@EJB** para que o servidor web atribua uma referência do EJB **AlunoRemote**, depois chame um método do EJB e deixe a servlet retornar um HTML.

Exemplo:

A seguir, crie a interface remota do EJB, que possui a assinatura do método **consultarMediaFinal**.

```
package aula.ejb;

import javax.ejb.Remote;

/**
 * Interface remota para o EJB de Aluno.
 */
@Remote
public interface AlunoRemote {
    /**
     * Método utilizado para consultar a média final
     * de um aluno.
     *
     * @param matricula - Matricula do aluno utilizado
     *     para a consulta.
     * @return Media final do aluno.
     */
    public Double consultarMediaFinal(String matricula);
}
```

A seguir crie o EJB do tipo Stateless, que é responsável por implementar o método **consultarMediaFinal** que retorna um número randômico.

```
package aula.ejb;

import javax.ejb.Stateless;

/**
 * Session Beans Stateless que implementa as
 * funcionalidades referentes a aluno.
 */
@Stateless
public class AlunoBean implements AlunoRemote {
    /**
     * Método utilizado para consultar a média final
     * de um aluno.
     *
     * @param matricula - Matricula do aluno utilizado
     * na consulta.
     * @return Média final do aluno.
     */
    public Double consultarMediaFinal(String matricula) {
        /* Implementação de exemplo
         * Retorna um número randomico entre 0 e 10. */
        return Math.random() * 10;
    }
}
```

A seguir, declare uma Servlet para testar o EJB de Aluno.

```
package aula.servlet;

import aula.ejb.AlunoRemote;
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
```

```
* Servlet utilizada para testar o EJB de Aluno.  
*/  
public class AlunoServlet extends HttpServlet {  
    private static final long serialVersionUID =  
        -1359012378400126336L;  
  
    @EJB  
    private AlunoRemote ejb;  
  
    @Override  
    protected void doGet(HttpServletRequest request,  
        HttpServletResponse response) throws ServletException,  
        IOException {  
  
        response.setContentType("text/html;charset=UTF-8");  
        PrintWriter out = response.getWriter();  
        try {  
            out.println("<html>");  
            out.println("  <head>");  
            out.println("    <title>Teste @EJB + Servlet</title>");  
            out.println("  </head>");  
            out.println("  <body>");  
            out.println("    <h1>Consultando a média do aluno.</h1>");  
            out.println("    <br>");  
            out.println(ejb.consultarMediaFinal("123456"));  
            out.println("  </body>");  
            out.println("</html>");  
        } finally {  
            out.close();  
        }  
    }  
}
```

Está é a Servlet que usa o EJB de Aluno, note que foi declarado um atributo do tipo **AlunoRemote** que é a interface remota do EJB, e foi utilizado a anotação **@EJB** nesse atributo.

Quando chamar esta Servlet o próprio servidor web coloca uma instância do EJB **AlunoBean** no atributo ejb.

Depois de publicar o EJB e o Servlet, quando chamar a URL do Servlet temos a seguinte saída:

Criando uma aplicação EJB + JPA

Vamos desenvolver uma aplicação utilizando EJB + JPA, o projeto em questão possui um cadastro de livro, um cadastro de pessoa e podemos emprestar livros para as pessoas.

Tendo este escopo vamos criar um Projeto EJB no NetBeans:

Criando o Projeto EJB

Na área **Projetos**, clique com o botão direito do mouse, e em seguida clique em **Novo projeto....**

Na tela **Novo projeto**, selecione na aba **Categorias** a opção **Java EE**, na aba **Projetos** a opção **Módulo EJB** e clique em **Próximo**.

Na tela de **Novo Módulo EJB**, vamos definir os seguintes campos:

- **Nome do projeto:** EmprestimoEJB
- **Localização do projeto:** (Escolha o local para salvar o projeto no micro)

Clique em **Próximo**.

Na tela de Novo Módulo EJB, vamos definir os seguintes campos:

- **Servidor:** GlassFish Server 3.1
- **Versão do Java EE:** JavaEE 6

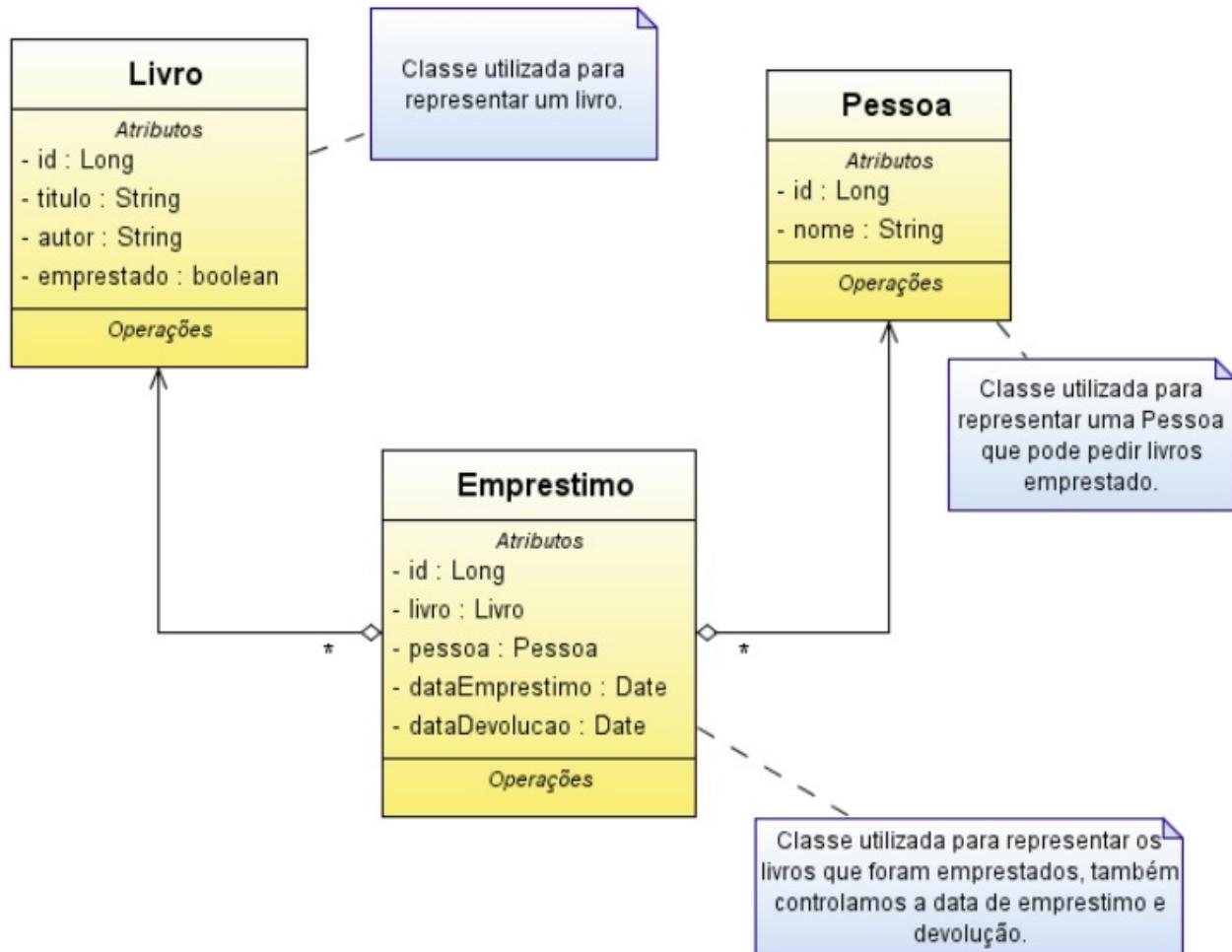
Clique em **Finalizar**.

Desta forma criamos um Projeto EJB chamado **EmprestimoEJB** que será publicado dentro do servidor de aplicação web **GlassFish**.

Criando as classes de negócio

Com base no escopo do projeto teremos inicialmente três classes **Livro**, **Pessoa** e **Emprestimo**.

A figura a seguir apresenta o diagrama de classes de negócio do projeto.



Com base no diagrama de classes, vamos criar essas classes dentro do nosso projeto EmprestimoEJB:

Clique com o botão direito do mouse sobre **Pacotes de código fonte**, depois selecione a opção **Novo**, depois selecione **Classe Java....**

Na tela de **Novo Classe Java**, vamos definir os seguintes valores:

- **Nome da classe:** Livro
- **Pacote:** pbc.emprestimo.modelo (utilizamos o pacote para separar os arquivos Java dentro da aplicação).

Clique em **Finalizar**.

Repita este mesmo processo para as classes **Pessoa** e **Emprestimo**.

Seguindo o modelo UML, vamos adicionar nas classes os atributos, métodos get / set e anotações referentes ao JPA, depois nossas classes ficaram da seguinte forma:

Classe Livro:

```
package pbc.emprestimo.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

/**
 * Classe utilizada para representar um Livro.
 */
@Entity
@SequenceGenerator(name="LIVRO_SEQ", sequenceName="LIVRO_SEQ",
    initialValue=1, allocationSize=1)
public class Livro implements Serializable {
    /* Serial Version UID. */
    private static final long serialVersionUID =
        6775900088322385451L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "LIVRO_SEQ")
    private Long id;
    private String titulo;
    private String autor;
    private boolean emprestado;

    public Livro() {
        this.emprestado = false;
    }

    public String getAutor() { return autor; }
    public void setAutor(String autor) { this.autor = autor; }
```

```
public boolean isEmprestado() { return emprestado; }
public void setEmprestado(boolean emprestado) {
    this.emprestado = emprestado;
}

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

public String getTitulo() { return titulo; }
public void setTitulo(String titulo) {
    this.titulo = titulo;
}
```

Classe Pessoa:

```

package pbc.emprestimo.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

/**
 * Classe utilizada para representar uma Pessoa.
 */
@Entity
@SequenceGenerator(name="PESSOA_SEQ", sequenceName="PESSOA_SEQ",

    initialValue = 1, allocationSize = 1)
public class Pessoa implements Serializable {
    /* Serial Version UID */
    private static final long serialVersionUID =
        5486103235574819424L;

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
        generator="PESSOA_SEQ")
    private Long id;
    private String nome;

    public Pessoa() { }

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

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

```

Classe Emprestimo:

```

package pbc.emprestimo.modelo;

```

```
import java.io.Serializable;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.Temporal;

/**
 * Classe utilizada para representar o Emprestimo
 * de um Livro feito por uma Pessoa.
 */
@Entity
@SequenceGenerator(name = "EMPRESTIMO_SEQ", sequenceName =
    "EMPRESTIMO_SEQ", initialValue = 1, allocationSize = 1)
public class Emprestimo implements Serializable {
    /* Serial Version UID */
    private static final long serialVersionUID =
        4621324705834774752L;

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
        generator="EMPRESTIMO_SEQ")
    private Long id;
    @ManyToOne
    private Livro livro;
    @ManyToOne
    private Pessoa pessoa;
    @Temporal(javax.persistence.TemporalType.DATE)
    private Date dataEmprestimo;
    @Temporal(javax.persistence.TemporalType.DATE)
    private Date dataDevolucao;

    public Emprestimo() { }

    public Date getDataDevolucao() { return dataDevolucao; }
    public void setDataDevolucao(Date dataDevolucao) {
        this.dataDevolucao = dataDevolucao;
    }
}
```

```

public Date getDataEmprestimo() { return dataEmprestimo; }
public void setDataEmprestimo(Date dataEmprestimo) {
    this.dataEmprestimo = dataEmprestimo;
}

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

public Livro getLivro() { return livro; }
public void setLivro(Livro livro) { this.livro = livro; }

public Pessoa getPessoa() { return pessoa; }
public void setPessoa(Pessoa pessoa) {
    this.pessoa = pessoa;
}
}

```

Classes DAO

Vamos criar a camada de persistência dos dados, para isso criaremos as classes **DAO**, que utilizam o **EntityManager** para executar as operações no Banco de Dados. Estas classes devem ser criadas no pacote **pbc.emprestimo.dao**.

Classe **EmprestimoDAO**:

```

package pbc.emprestimo.dao;

import javax.persistence.EntityManager;
import pbc.emprestimo.modelo.Emprestimo;

/**
 * Classe utilizada para realizar as operações com
 * o bando de dados.
 */
public class EmprestimoDAO {
    private EntityManager entityManager;

    /**

```

```
* Construtor da classe DAO que chama os métodos do
* EntityManager.
* @param entityManager
*/
public EmprestimoDAO(EntityManager entityManager) {
    this.entityManager = entityManager;
}

/**
 * Método para salvar ou atualizar o empréstimo.
* @param emprestimo
* @return
* @throws java.lang.Exception
*/
public Emprestimo salvar(Emprestimo emprestimo)
throws Exception {
System.out.println("Emprestando o livro " +
emprestimo.getLivro().getTitulo() +
" para a pessoa " + emprestimo.getPessoa().getNome());

/* Verifica se o emprestimo ainda não está salvo no
banco de dados. */
if(emprestimo.getId() == null) {
    /* Salva o emprestimo no banco de dados. */
    this.entityManager.persist(emprestimo);
} else {
    /* Verifica se o emprestimo não está no estado managed. */
    if(!this.entityManager.contains(emprestimo)) {
        /* Se o emprestimo não está no estado managed
verifica se ele existe na base. */
        if (consultarPorId(emprestimo.getId()) == null) {
            throw new Exception("Livro não existe!");
        }
    }
    /* Faz uma atualização do empréstimo. */
    return entityManager.merge(emprestimo);
}

/* Retorna o empréstimo que foi salvo, este retorno ocorre
para modemos ter o id que foi salvo. */
```

```

        return emprestimo;
    }

    /**
     * Método que exclui o Emprestimo do banco de dados.
     * @param id
     */
    public void excluir(Long id) {
        /* Consulta o emprestimo na base de dados através de
         seu ID. */
        Emprestimo emprestimo = consultarPorId(id);
        System.out.println("Excluindo o emprestimo: " +
            emprestimo.getId());

        /* Remove o emprestimo da base de dados. */
        entityManager.remove(emprestimo);
    }

    /**
     * Método que consulta um Emprestimo através do Id.
     * @param id
     * @return
     */
    public Emprestimo consultarPorId(Long id) {
        return entityManager.find(Emprestimo.class, id);
    }
}

```

Note que agora não precisamos mais criar a **EntityManager** manualmente, podemos deixar para quem for utilizar o DAO (no nosso caso o EJB) passar uma referência para o EntityManager.

Os métodos do DAO que fazem manipulação no banco de dados agora não precisam mais iniciar e finalizar uma transação, pois como serão chamados a partir de um EJB, os métodos do EJB que chamam os métodos do DAO já possuem transação por padrão.

Classe **LivroDAO**:

```
package pbc.emprestimo.dao;

import javax.persistence.EntityManager;
import pbc.emprestimo.modelo.Livro;

/**
 * Classe utilizada para realizar as operações com o
 * bando de dados.
 */
public class LivroDAO {
    private EntityManager entityManager;

    /**
     * Construtor da classe DAO que chama os métodos
     * do EntityManager.
     * @param entityManager
     */
    public LivroDAO(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    /**
     * Método para salvar ou atualizar o livro.
     * @param livro
     * @return
     * @throws java.lang.Exception
     */
    public Livro salvar(Livro livro) throws Exception{
        System.out.println("Salvando o livro: "
            + livro.getTitulo());

        /* Verifica se o livro ainda não está salvo no banco
         * de dados. */
        if(livro.getId() == null) {
            /* Salva o livro no banco de dados. */
            this.entityManager.persist(livro);
        } else {
            /* Verifica se o livro não está no estado managed. */
            if(!this.entityManager.contains(livro)) {
```

```
    /* Se o livro não está no estado managed verifica se
       ele existe na base. */
    if (entityManager.find(Livro.class, livro.getId())
        == null) {
        throw new Exception("Livro não existe!");
    }
}

/* Faz uma atualização do livro que estava gravado na base
   de dados. */
return entityManager.merge(livro);
}

/* Retorna o livro que foi salvo, este retorno ocorre para
   podermos ter o id que foi salvo. */
return livro;
}

/**
 * Método que exclui o livro do banco de dados.
 *
 * @param id
 */
public void excluir(Long id) {
    /* Consulta o livro na base de dados através
       de seu ID. */
    Livro livro = entityManager.find(Livro.class, id);
    System.out.println("Excluindo o livro: "
        + livro.getTitulo());

    /* Remove o livro da base de dados. */
    entityManager.remove(livro);
}

/**
 * Método que consulta o livro pelo ID.
 *
 * @param id
 * @return
 */
public Livro consultarPorId(Long id) {
```

```
    return entityManager.find(Livro.class, id);
}
}
```

Classe PessoaDAO:

```
package pbc.emprestimo.dao;

import javax.persistence.EntityManager;
import pbc.emprestimo.modelo.Pessoa;

/**
 * Classe utilizada para realizar as operações com o
 * banco de dados.
 */
public class PessoaDAO {
    private EntityManager entityManager;

    /**
     * Construtor da classe DAO que chama os métodos do
     * EntityManager.
     * @param entityManager
     */
    public PessoaDAO(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    /**
     * Método para salvar ou atualizar a pessoa.
     * @param pessoa
     * @return
     * @throws java.lang.Exception
     */
    public Pessoa salvar(Pessoa pessoa) throws Exception{
        System.out.println("Salvando o pessoa: "
            + pessoa.getNome());

        /* Verifica se a pessoa ainda não está salva no
        banco de dados. */
    }
}
```

```
if(pessoa.getId() == null) {
    /* Salva a pessoa no banco de dados. */
    this.entityManager.persist(pessoa);
} else {
    /* Verifica se a pessoa não está no estado managed. */
    if(!this.entityManager.contains(pessoa)) {
        /* Se a pessoa não está no estado managed verifica
         * se ele existe na base. */
        if(entityManager.find(Pessoa.class, pessoa.getId())
            == null) {
            throw new Exception("Livro não existe!");
        }
    }
    /* Faz uma atualização da pessoa que estava gravado na
     * base de dados. */
    return entityManager.merge(pessoa);
}

/* Retorna a pessoa que foi salva, este retorno ocorre para
 * podermos ter o id que foi salvo. */
return pessoa;
}

/**
 * Método que exclui a pessoa do banco de dados.
 * @param id
 */
public void excluir(Long id) {
    /* Consulta a pessoa na base de dados através de seu ID. */
    Pessoa pessoa = entityManager.find(Pessoa.class, id);
    System.out.println("Excluindo a pessoa: " + pessoa.getNome());
};

/* Remove a pessoa da base de dados. */
entityManager.remove(pessoa);
}

/**
 * Consulta a pessoa por ID.
 * @param id
*/
```

```
* @return
*/
public Pessoa consultarPorId(Long id) {
    return entityManager.find(Pessoa.class, id);
}
```

Componentes EJB

Vamos criar os componentes EJB que possuem a lógica de negócio. Lembrando que cada EJB é formado por uma interface e uma classe.

Crie uma interface chamada **EmprestimoRemote** e uma classe chamada **EmprestimoBean** no pacote **pbc.emprestimo.ejb**.

Interface **EmprestimoRemote**:

```
package pbc.emprestimo.ejb;

import javax.ejb.Remote;
import pbc.emprestimo.modelo.Emprestimo;

/**
 * Interface que possui os métodos que o EJB de Emprestimo
 * precisa implementar.
 */
@Remote
public interface EmprestimoRemote {
    public Emprestimo salvar(Emprestimo emprestimo)
        throws Exception;

    public void excluir(Long id);

    public Emprestimo consultarPorId(Long id);
}
```

Estamos declarando uma interface chamada **EmprestimoRemote** e adicionamos a anotação **@Remote** que irá informar que este componente EJB pode ser acessado remotamente ou forá do seu container.

Na interface nós declaramos todos os métodos que o componente EJB poderá disponibilizar.

Classe **EmprestimoBean**:

```
package pbc.emprestimo.ejb;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import pbc.emprestimo.dao.EmprestimoDAO;
import pbc.emprestimo.modelo.Emprestimo;

/**
 * EJB de Emprestimo com a lógica de negocio.
 */
@Stateless
public class EmprestimoBean implements EmprestimoRemote {
    @PersistenceContext(unitName = "EmprestimoPU")
    private EntityManager em;

    public Emprestimo salvar(Emprestimo emprestimo)
        throws Exception {
        EmprestimoDAO dao = new EmprestimoDAO(em);
        return dao.salvar(emprestimo);
    }

    public void excluir(Long id) {
        EmprestimoDAO dao = new EmprestimoDAO(em);
        dao.excluir(id);
    }

    public Emprestimo consultarPorId(Long id) {
        EmprestimoDAO dao = new EmprestimoDAO(em);
        return dao.consultarPorId(id);
    }
}
```

A classe **EmprestimoBean** implementa a interface `EmprestimoRemote`, portanto ela implementa todos os métodos declarados na interface. Também adicionamos a anotação **@Stateless** para informar que este componente é do tipo **Stateless Session Bean** e que não irá manter o valor dos seus atributos (estado).

Note que declaramos um atributo chamado em do tipo EntityManager a adicionamos a anotação **@PersistenceContext**, dessa forma o componente EJB recebe do Container EJB uma instancia do tipo EntityManager com a conexão com o banco de dados.

Quando utilizamos **@PersistenceContext** especificamos qual o nome da unidade de persistência queremos obter (O nome da unidade de persistência é especificado no arquivo persistence.xml).

Vamos agora criar o EJB para o Livro e Pessoa:

Interface **LivroRemote**:

```
package pbc.emprestimo.ejb;

import javax.ejb.Remote;
import pbc.emprestimo.modelo.Livro;

/**
 * Interface que possui os métodos que o EJB de Livro
 * precisa implementar.
 */
@Remote
public interface LivroRemote {
    public Livro salvar(Livro livro) throws Exception;

    public void excluir(Long id);

    public Livro consultarPorId(Long id);
}
```

Classe **LivroBean**:

```
package pbc.emprestimo.ejb;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import pbc.emprestimo.dao.LivroDAO;
import pbc.emprestimo.modelo.Livro;

/**
 * EJB de Livro com a lógica de negocio.
 */
@Stateless
public class LivroBean implements LivroRemote {
    @PersistenceContext(unitName = "EmprestimoPU")
    private EntityManager em;

    public Livro salvar(Livro livro) throws Exception {
        LivroDAO dao = new LivroDAO(em);
        return dao.salvar(livro);
    }

    public void excluir(Long id) {
        LivroDAO dao = new LivroDAO(em);
        dao.excluir(id);
    }

    public Livro consultarPorId(Long id) {
        LivroDAO dao = new LivroDAO(em);
        return dao.consultarPorId(id);
    }
}
```

Interface **PessoaRemote**:

```
package pbc.emprestimo.ejb;

import javax.ejb.Remote;
import pbc.emprestimo.modelo.Pessoa;

/**
 * Interface que possui os métodos que o EJB de Pessoa
 * precisa implementar.
 */
@Remote
public interface PessoaRemote {
    public Pessoa salvar(Pessoa pessoa) throws Exception;

    public void excluir(Long id);

    public Pessoa consultarPorId(Long id);
}
```

Classe PessoaBean:

```
package pbc.emprestimo.ejb;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import pbc.emprestimo.dao.PessoaDAO;
import pbc.emprestimo.modelo.Pessoa;

/**
 * EJB de Pessoa com a lógica de negocio.
 * @author Rafael Guimarães Sakurai
 */
@Stateless
public class PessoaBean implements PessoaRemote {
    @PersistenceContext(unitName = "EmprestimoPU")
    private EntityManager em;

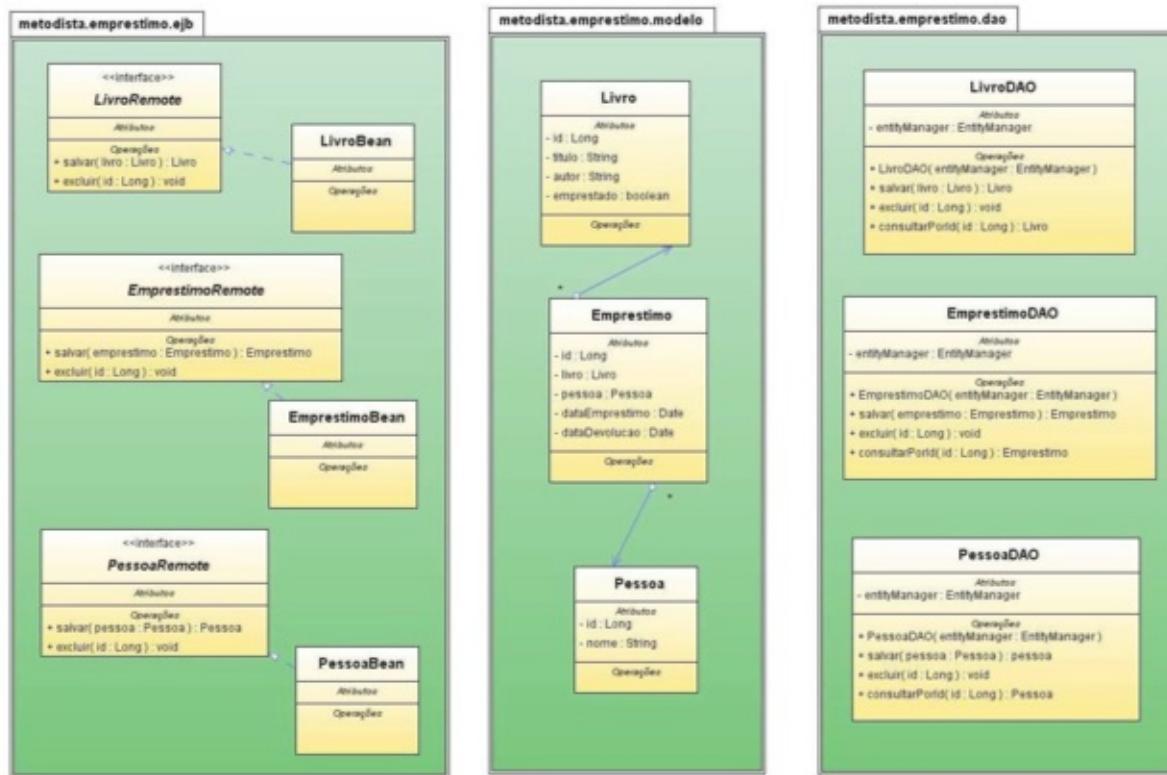
    public Pessoa salvar(Pessoa pessoa) throws Exception {
        PessoaDAO dao = new PessoaDAO(em);
        return dao.salvar(pessoa);
    }

    public void excluir(Long id) {
        PessoaDAO dao = new PessoaDAO(em);
        dao.excluir(id);
    }

    public Pessoa consultarPorId(Long id) {
        PessoaDAO dao = new PessoaDAO(em);
        return dao.consultarPorId(id);
    }
}
```

Até o momento criamos a seguinte estrutura, note que as interfaces e classes são bem coesas, ou seja, todas elas tem um propósito específico, na camada de EJB temos apenas lógica de negocio, na camada de persistência temos apenas classes que fazem o trabalho com o banco de dados.

A figura a seguir mostra as classes e interfaces criadas até o momento:



Criando a base de dados

A seguir temos o script para o banco de dados Oracle para criar as sequences e tabelas:

```
CREATE SEQUENCE LIVRO_SEQ INCREMENT BY 1
    START WITH 1 NOCACHE NOCYCLE;

CREATE SEQUENCE PESSOA_SEQ INCREMENT BY 1
    START WITH 1 NOCACHE NOCYCLE;

CREATE SEQUENCE EMPRESTIMO_SEQ INCREMENT BY 1
    START WITH 1 NOCACHE NOCYCLE;

CREATE TABLE livro (
    id NUMBER(5) NOT NULL PRIMARY KEY,
    titulo VARCHAR2(100) NOT NULL,
    autor VARCHAR2(100) NOT NULL,
    emprestado NUMBER(1) NOT NULL
);

CREATE TABLE pessoa (
    id NUMBER(5) NOT NULL PRIMARY KEY,
    nome VARCHAR2(100) NOT NULL
);

CREATE TABLE emprestimo (
    id NUMBER(5) NOT NULL PRIMARY KEY,
    livro_id NUMBER(5) NOT NULL,
    pessoa_id NUMBER(5) NOT NULL,
    dataemprestimo DATE,
    datadevolucao DATE
);
```

Configuração da camada de persistência

Quando utilizamos o JPA para fazer a persistência com o banco de dados, precisamos fazer as seguintes configurações:

- Adicionar driver do banco de dados Oracle no Glassfish;
- Criar um data source com pool de conexão no Glassfish;
- Criar arquivo persistence.xml.

Adicionar do driver do banco de dados

Coloque o driver do banco de dados na pasta de instalação do Glassfish.

```
..\glassfish-vx\domains\domain1\lib
```

Como estamos usando o Oracle nesse exemplo, devemos colocar na pasta de bibliotecas do Glassfish o jar ojdbc6.jar.

Criar um data source com pool de conexão no Glassfish

Inicie o Glassfish e entre na url: <http://localhost:4848/login.jsf> que abrirá a tela de administração.

Na tela de administração do Glassfish, selecione o menu **Recursos**, dentro dele escolha **JDBC** e depois escolha o menu **JDBC Connection Pools**.

Na tela de **JDBC Connection Pools**, clique no botão **Novo....**

Na tela **Novo grupo de conexões JDBC (Etapa 1 de 2)** (Novo Pool de Conexão JDBC), configure as seguintes opções:

- **Pool Name:** Emprestimo
- **Tipo de Recurso:** javax.sql.ConnectionPoolDataSource
- **Fornecedor do Banco de Dados:** Oracle

Se estivermos trabalhando com outra base de dados é só escolher no combo de **Fornecedor do Banco de Dados**.

Depois clique em **Avançar**.

Na tela de **Novo grupo de conexões JDBC (Etapa 2 de 2)**, desça até a parte de **Adicionar Propriedades**.

Clique no botão **Selecionar todos**, depois clique em **Excluir Propriedades**.

Depois adicione as seguintes propriedades:

- **user:** nome do usuário no banco de dados

- **password:** senha do usuário no banco de dados
- **url:** caminho para encontrar o banco de dados.

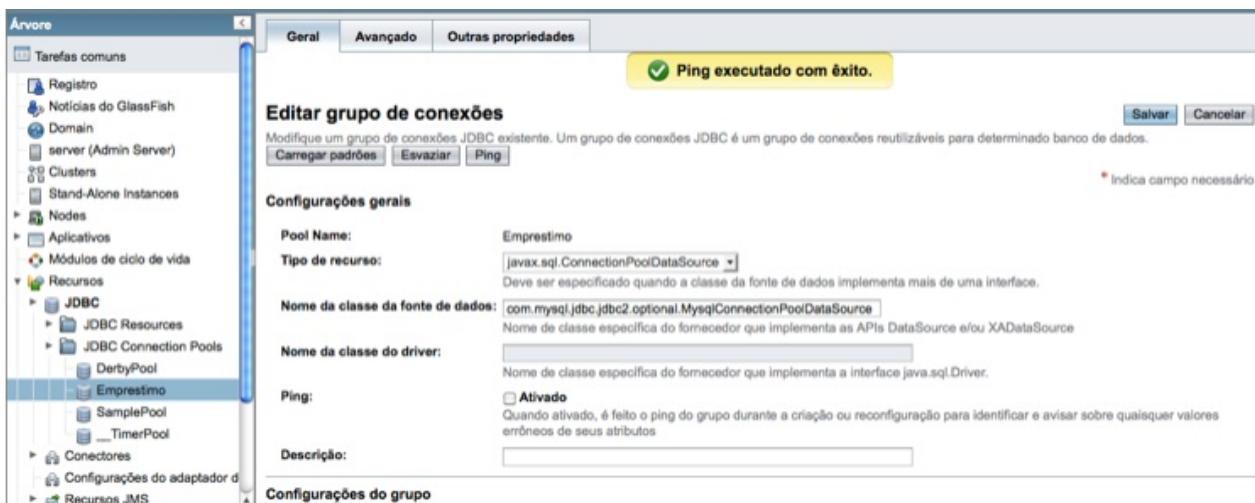
No Oracle utilizamos a URL:

```
jdbc:oracle:thin:@localhost:1521:XE
```

ou de acordo com sua instalação.

Volte para o topo da página e clique em **Finalizar**.

Clique no pool de conexão que acabamos de criar **Emprestimo**, e clique no botão **Ping** para verificar se o Glassfish encontra o banco de dados como apresentado na figura a seguir:



Deve aparecer na tela a mensagem **Ping executado com êxito**, informando que conseguiu encontrar no banco de dados.

Agora vamos criar o recurso JDBC que será utilizado por nosso projeto EJB:

Clique na menu **Recursos**, dentro dele escolha **JDBC** e depois escolha **JDBC Resources**.

Na tela de **Recursos JDBC** clique no botão **Novo....**

Na tela de **Novo Recuros JDBC** configure os itens:

- **Nome JNDI:** jdbc/Emprestimo
- **Nome do grupo:** Emprestimo

Depois clique no botão **OK**. Pronto, criamos o pool de conexão no Glassfish.

Criar arquivo persistence.xml

Agora vamos criar uma **Unidade de Persistência**, que utiliza o **Pool de Conexão** que acabamos de criar no Glassfish:

Clique com o botão direito sobre o projeto **EmprestimoEJB**, selecione a opção **Novo** e depois selecione **Unidade de persistência....**

Na tela de Novo Unidade de persistência, configure os seguintes itens:

- **Nome da unidade de persistência:** EmprestimoPU
- **Provedor de persistência:** Hibernate JPA (1.0)
- **Fontes de dados:** jdbc/Emprestimo
- **Estratégia de geração de tabela:** Nenhum

Depois clique no botão **Finalizar**.

Abra o arquivo **persistence.xml** dentro da pasta **Arquivos de configuração** e escreva o seguinte código:

```
<?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="EmprestimoPU" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>jdbc/Emprestimo</jta-data-source>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Note que o nome da unidade de persistência **EmprestimoPU** que acabamos de criar, é o mesmo nome que utilizamos na anotação **@PersistenceContext** como mostrado na figura a seguir, pois é através dessa unidade de persistência que o **Container EJB** cria um **EntityManager**.

```

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_1_0.xsd">
    <persistence-unit name="EmprestimoPU" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>jdbc/Emprestimo</jta-data-source>
        <properties>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>

```

```

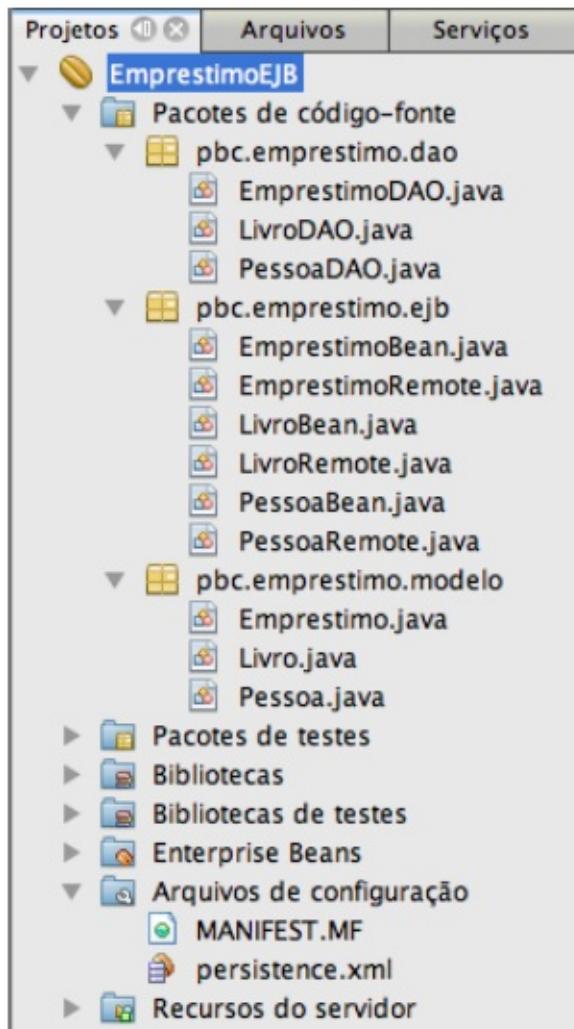
EmprestimoBean.java
package metodista.emprestimo.ejb;
import ...
/*
 * ...
 */
@Stateless
public class EmprestimoBean implements EmprestimoRemote {
    @PersistenceContext(unitName = "EmprestimoPU")
    private EntityManager em;

    public Emprestimo salvar(Emprestimo emprestimo) throws Exception {
        ...
    }

    public void excluir(Long id) {
        ...
    }
}

```

Feito isto nossa aplicação EmprestimoEJB já está pronta, com a estrutura igual a da figura a seguir:



A figura a seguir mostra como publicar a aplicação EmprestimoEJB no Glassfish.



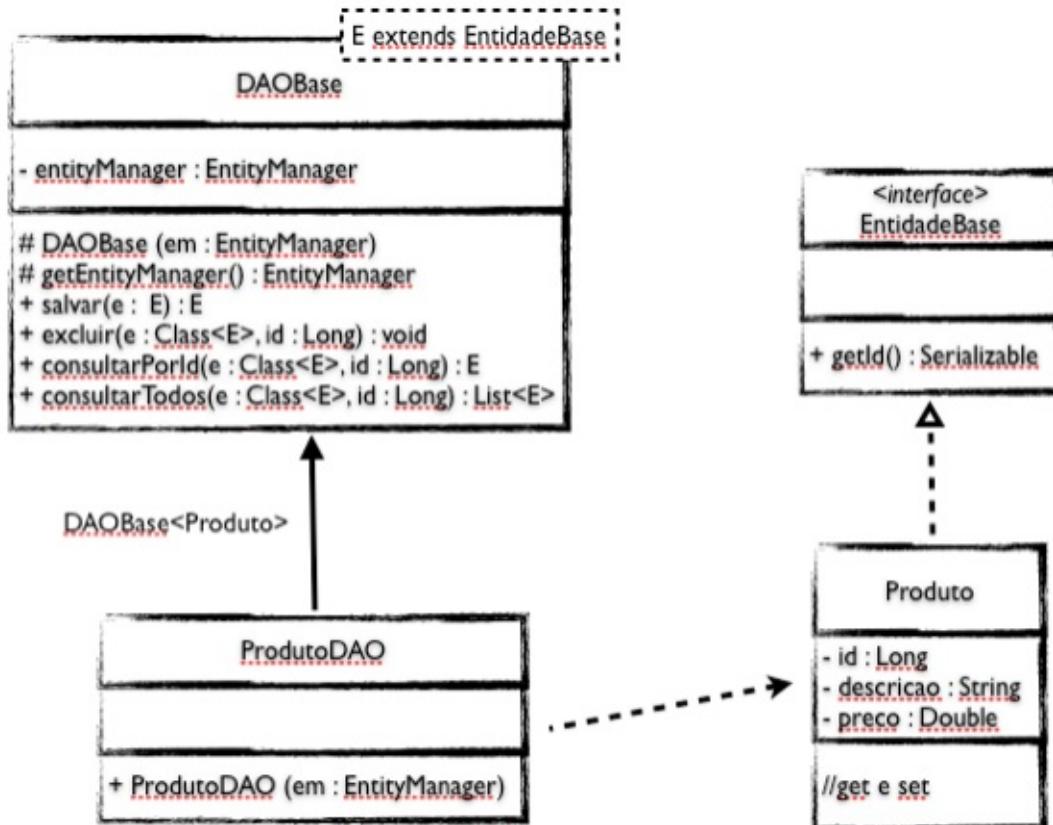
Interceptando os EJBs para criar objetos DAOs

Quando utilizamos EJBs estamos utilizando componentes que possuem seu ciclo de vida gerenciados pelo Container EJB, as vezes desejamos executar algo antes da criação ou depois da criação do EJB, como por exemplo quando terminar de criar o objeto do EJB, gostaríamos de criar os objetos das classes DAOs de forma genérica para qualquer EJB, sem ter que ficar declarando em cada EJB como criar os DAOs.

Os interceptadores são objetos que executam durante o ciclo de vida do EJB, neste exemplo vamos criar um interceptador para servir como uma fabrica de objetos DAO nos EJBs.

As operações salvar, atualizar, apagar, consultar por id e consultar todos de um DAO são muito similares, o que muda é apenas a entidade utilizada pelo DAO, para facilitar a criação das classes DAO vamos criar uma classe DAOBase que possui essas funcionalidades criadas de forma genérica, ficando assim independente da entidade as operações básicas serão executadas.

A figura a seguir mostra como fica a estrutura de classes do DAO e entidade de forma genérica.



A primeira coisa que precisamos fazer é criar uma interface base para as entidades onde definimos que toda entidade deve ter uma método para obter o objeto ID da entidade. Portanto todas as entidades precisam implementar esta interface.

```

package metodista.infra.dao;

import java.io.Serializable;

/**
 * Classe basica para criação das Entity's.
 */
public interface EntidadeBase {
    /**
     * @return o Id da Entity.
     */
    public Serializable getId();
}
  
```

Exemplo de entidade:

```
package metodista.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import metodista.infra.dao.EntidadeBase;

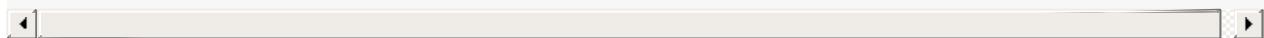
/**
 * Entidade que representa um Produto e demonstra o
 * uso da interface EntidadeBase.
 */
@Entity
@SequenceGenerator(name="PRODUTO_SEQ", sequenceName="PRODUTO_SEQ"
,
    initialValue = 1, allocationSize = 1)
public class Produto implements Serializable, EntidadeBase {
    private static final long serialVersionUID =
        4772286290044296438L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "PRODUTO_SEQ")
    private Long id;
    private String descricao;
    private Double preco;

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

    public String getDescricao() { return descricao; }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public Double getPreco() { return preco; }
    public void setPreco(Double preco) { this.preco = preco; }
}
```



Vamos criar uma classe DAOBase que possui os métodos genéricos já que as operações de salvar, atualizar, remover, consultar por id e consultar todos são iguais para as entidades:

```
package metodista.infra.dao;

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

/**
 * Classe base para todas as demais classes de acesso aos
 * dados, possui os métodos genéricos para todas os demais DAO's
 *
 * @param <E>
 */
public class DAOBase<E extends EntidadeBase> {

    /**
     * EntityManager que controla as Entity's da aplicação.
     */
    private EntityManager entityManager;

    /**
     * Construtor.
     * @param em - EntityManager que controla a conexão com
     * o banco de dados.
     */
    protected DAOBase(final EntityManager em) {
        this.entityManager = em;
    }

    /**
     * @return the entityManager
     */
    protected EntityManager getEntityManager() {
        return this.entityManager;
    }
}
```

```

    /**
     * Persiste um Entity na base de dados.
     *
     * @param e - Entity que será persistido.
     * @return o Entity persistido.
     * @throws Exception caso ocorra algum erro.
     */
    public E salvar(final E e) throws Exception {
        try {
            //Verifica se a Entity já existe para fazer Merge.
            if (e.getId() != null) {
                /* Verifica se a Entity não está gerenciavel pelo
                   EntityManager. */
                if (!this.entityManager.contains(e)) {
                    //Busca a Entity da base de dados, baseado no Id.
                    if (this.entityManager.find(e.getClass(), e.getId())
                        == null) {
                        throw new Exception("Objeto não existe!");
                    }
                }
                return this.entityManager.merge(e);
            } else { // Se a Entity não existir persiste ela.
                this.entityManager.persist(e);
            }

            //Retorna a Entity persistida.
            return e;
        } catch (PersistenceException pe) {
            pe.printStackTrace();
            throw pe;
        }
    }

    /**
     * Exclui um Entity da base de dados.
     *
     * @param e - Entity que será excluída.
     * @param k - Id da Entity que será excluída.
     * @throws Exception caso ocorra algum erro.
     */

```

```

public void excluir(final Class<E> e, final Long k)
    throws Exception {
    try {
        E entity = this.consultarPorId(e, k);
        //Remove a Entity da base de dados.
        this.entityManager.remove(entity);
    } catch (PersistenceException pe) {
        pe.printStackTrace();
        throw pe;
    }
}

/**
 * Consulta um Entity pelo seu Id.
 *
 * @param e - Classe da Entity.
 * @param l - Id da Entity.
 * @return - a Entity da classe.
 */
public E consultarPorId(final Class<E> e, final Long l) {
    /* Procura uma Entity na base de dados a partir da classe
     * e do seu ID. */
    return this.entityManager.find(e, l);
}

public List<E> consultarTodos(final Class<E> e) {
    Query query = this.entityManager.createQuery("SELECT e FROM "
        + e.getSimpleName() + " e");

    return (List<E>) query.getResultList();
}

```

Todas as classes DAO agora devem extender a classe DAOBase dessa forma elas receberam a implementação genérica desses métodos.

Exemplo de DAO que é uma subclasse da DAOBase:

```

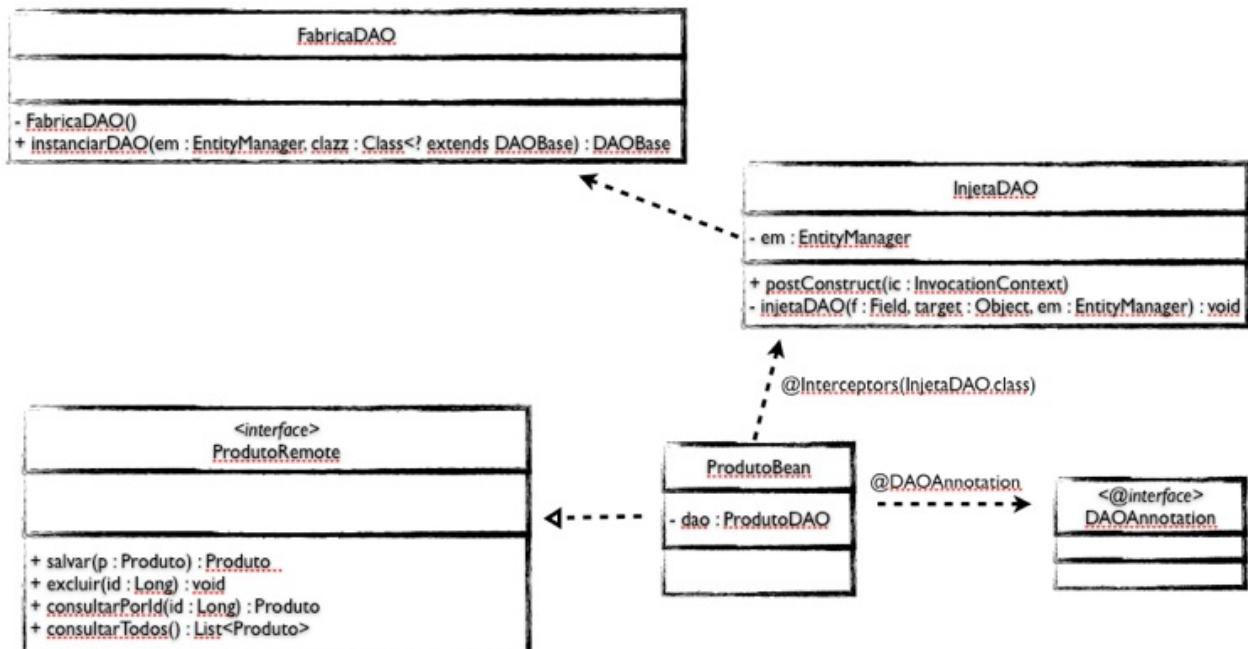
package metodista.dao;

import javax.persistence.EntityManager;
import metodista.infra.dao.DAOBase;
import metodista.modelo.Produto;

/**
 * Classe DAO que é uma subclasse da classe DAOBase.
 */
public class ProdutoDAO extends DAOBase<Produto> {
    public ProdutoDAO(EntityManager em) {
        super(em);
    }
}

```

Agora que montamos um DAO genérico para qualquer entidade, vamos criar o interceptador para o EJB como mostrado na figura a seguir:



Vamos criar uma anotação chamada `DAOAnnotation`, esta anotação será utilizada pelo interceptador para verificar quais os atributos que são DAO devem ser instanciados, iremos utilizar esta anotação na declaração dos atributos DAO dentro do EJB.

```
package metodista.infra.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Annotation responsável por criar uma instancia do DAO.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface DAOAnnotation {
}
```

Vamos criar agora uma fabrica de DAO, está fabrica cria uma instancia do DAO via Reflection.

```
package metodista.infra.dao;

import java.lang.reflect.Constructor;
import javax.persistence.EntityManager;

/**
 * Essa Factory gera instancias automática do <code>DAO</code>.
 * Utilizando o <code>DAOAnnotation</code> nos atributos dos
 * EJBs, pega qual a classe do <code>DAO</code> gera uma nova
 * instancia.
 */
public final class FabricaDAO {

    /**
     * Construtor privado.
     */
    private FabricaDAO() {
    }

    /**
     * Método responsável por criar uma instancia do

```

```
* <code>DAO</code>.
*
* @param entityManager - <code>EntityManager</code> que é
* usado no construtor do <code>DAO</code>.
* @param clazz - Classe do DAO que será criado.
* @return Instancia do DAO.
* @throws Exception - Exceção lançada caso ocorra algum
* problema ao instanciar o <code>DAO</code>.
*/
public static DAOBase instanciarDAO(final EntityManager
    entityManager, final Class<? extends DAOBase> clazz)
    throws Exception {
    /* Usando Reflection para pegar o construtor do DAO que
       recebe um EntityManager como parametro. */
    Constructor construtor =
        clazz.getConstructor(EntityManager.class);

    /* Cria uma instancia do DAO passando o EntityManager
       como parâmetro. */
    return (DAOBase) construtor.newInstance(entityManager);
}
}
```

Agora vamos criar o interceptador de EJB chamado InjetaDAO, este interceptador possui um método que é executado após o objeto EJB ser criado e antes dele ser entregue para quem fez seu lookup, este método vai verificar via Reflection quais atributos declarados no EJB possuem a anotação `@DAOAnnotation`, esses atributos serão instanciados e um objeto EntityManager será passado para eles via construtor.

```
package metodista.infra.dao;

import java.lang.reflect.Field;
import javax.annotation.PostConstruct;
import javax.interceptor.InvocationContext;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import metodista.infra.annotation.DAOAnnotation;
```

```
/**
 * Classe usada para injetar o <code>DAO</code> nos
 * atributos dos EJBs.
 */
public class InjetaDAO {

    /**
     * Construtor.
     */
    public InjetaDAO() {
    }

    /**
     * EntityManager será repassando ao <code>DAO</code>.
     */
    @PersistenceContext(unitName = "ExemploPU")
    private EntityManager entityManager;

    /**
     * Este método é usado após o EJB ser criado, e dentro
     * do EJB procura os <code>DAO</code>s que precisa instanciar.
     * @param invocationContext - Alvo que será adicionado
     *           o <code>DAO</code>.
     * @throws Exception - Exceção lançada caso ocorra algum
     *           problema quando
     *           adicionar o <code>DAO</code>.
     */
    @PostConstruct
    public void postConstruct(final InvocationContext
        invocationContext) throws Exception {
        //Pega o alvo
        Object target = invocationContext.getTarget();

        //Pega a classe alvo.
        Class classe = target.getClass();
        //Procura os atributos da classe.
        Field[] fields = classe.getDeclaredFields();

        /* Verifica se algum dos campos da classe possui
         * o DAOAnnotation. */
    }
}
```

```

        for (Field field : fields) {
            if (field.isAnnotationPresent(DAOAnnotation.class)) {
                /* Quando encontrar algum atributo, com DAOAnnotation,
                   gera uma instancia do DAO.*/
                this.injetaDAO(field, target, this.entityManager);
            }
        }
    }

    /**
     * Método usado para gerar uma instancia do <code>DAO</code>
     * e atribui-la ao atributo.
     *
     * @param field - Atributo que vai receber o <code>DAO</code>.
     * @param target - Classe alvo.
     * @param entityManager - <code>EntityManager</code> que será
     *                       usado na instancia do <code>DAO</code>.
     * @throws Exception - Exceção lançada caso ocorra algum
     *                     problema quando adicionar o <code>DAO</code>.
     */
    private void injetaDAO(final Field field, final Object
        target, final EntityManager entityManager) throws Exception
    {
        //Pega a classe do DAO que sera instanciado.
        Class clazz = field.getType();

        //Gera uma instancia do DAO.
        DAOBase dao = FabricaDAO.instanciarDAO(entityManager, clazz)
        ;

        //Verifica se o atributo esta acessível.
        boolean acessivel = field.isAccessible();

        //Se o atributo nao e acessível, deixa ele como acessível.
        if (!acessivel) {
            field.setAccessible(true);
        }

        //Seta o DAO no atributo.
        field.set(target, dao);
    }
}

```

```
//Se o atributo não é acessível, volta ao valor original.  
if (!acessivel) {  
    field.setAccessible(acessivel);  
}  
}  
}
```

Agora precisamos adicionar as anotações no EJB para que o interceptador possa ser chamado:

```
package metodista.ejb;  
  
import java.util.List;  
import javax.ejb.Remote;  
import metodista.modelo.Produto;  
  
/**  
 * Interface Remota do EJB com operações básicas de  
 * um CRUD de Produto.  
 */  
@Remote  
public interface ProdutoRemote {  
    public Produto salvar(Produto produto) throws Exception;  
  
    public void excluir(Long id) throws Exception;  
  
    public Produto consultarPorId(Long id);  
  
    public List<Produto> consultarTodos();  
}
```

Implementação do EJB:

```
package metodista.ejb;

import java.util.List;
import javax.ejb.Stateless;
import javax.interceptor.Interceptors;
import metodista.dao.ProdutoDAO;
import metodista.infra.annotation.DAOAnnotation;
import metodista.infra.dao.InjetaDAO;
import metodista.modelo.Produto;

/**
 * EJB do tipo Stateless que utiliza a injeção do DAO.
 */
@Stateless
@Interceptors(InjetaDAO.class)
public class ProdutoBean implements ProdutoRemote {

    @DAOAnnotation
    private ProdutoDAO dao;

    public Produto salvar(Produto produto) throws Exception {
        return dao.salvar(produto);
    }

    public void excluir(Long id) throws Exception {
        dao.excluir(Produto.class, id);
    }

    public Produto consultarPorId(Long id) {
        return dao.consultarPorId(Produto.class, id);
    }

    public List<Produto> consultarTodos() {
        return dao.consultarTodos(Produto.class);
    }
}
```

Note que desta forma na implementação do EJB não é mais necessário criar o DAO manualmente, também não precisa obter o EntityManager, pois ele é adicionado ao DAO no interceptador.

Web Services SOAP

Introdução

Serviço Web (Web Service) é uma forma padronizada para integração entre aplicações diferentes.

Através do Web Service (WS) é possível integrar aplicações desenvolvidas com linguagens de programação diferente, pois o Web Service utiliza a troca de arquivos XML para realizar a comunicação.

Quando desenvolvemos uma aplicação com WS precisamos informar para quem for utilizar (consumir) o WS como ele funciona, para isso criamos um arquivo WSDL e enviamos este arquivo para quem for criar o cliente para o WS, após o cliente ser criado a comunicação feita entre as aplicações utiliza envelopes SOAP no formato XML como mostrado na figura a seguir:



Web Service Description Language

Web Service Description Language (WSDL) é um documento no padrão XML que descreve o funcionamento de um Web Service.

Neste documento podemos descrever:

- como encontrar o Web Service;
- quais métodos possuem esse Web Service;
- quais parâmetros os métodos recebem;

- o que os métodos retornam;

Exemplo de WSDL:

Neste exemplo temos o WSDL de um Web Service chamado ExemploWS que possui o método olaMundo que não recebe parâmetro e retorna uma String.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net.
RI's version is JAX-WS RI 2.1.3.1-hudson-749-SNAPSHOT. -->
<definitions targetNamespace="http://ws.exemplo.pbc/"
name="ExemploWSService" xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://ws.exemplo.pbc/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <ns1:Policy wsu:Id="ExemploWSServiceBinding_olaMundo_WSAT_Policy"
        xmlns:ns1="http://www.w3.org/ns/ws-policy">
        <ns1:ExactlyOne>
            <ns1:All>
                <ns2:ATAlwaysCapability
                    xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/10/wsat"
                />
                <ns3:ATAssertion ns1:Optional="true" ns4:Optional="true"
                    xmlns:ns4="http://schemas.xmlsoap.org/ws/2002/12/policy"
                    xmlns:ns3="http://schemas.xmlsoap.org/ws/2004/10/wsat"/>
            </ns1:All>
        </ns1:ExactlyOne>
    </ns1:Policy>
    <types>
        <xsd:schema>
            <xsd:import namespace="http://ws.exemplo.pbc/"
                schemaLocation="ExemploWSService_schema1.xsd"/>
        </xsd:schema>
    </types>
    <message name="olaMundo">
        <part name="parameters" element="tns:olaMundo"/>
    </message>
    <message name="olaMundoResponse">
```

```
<part name="parameters" element="tns:olaMundoResponse"/>
</message>
<portType name="ExemploWS">
    <operation name="olaMundo">
        <input message="tns:olaMundo"/>
        <output message="tns:olaMundoResponse"/>
    </operation>
</portType>
<binding name="ExemploWSPortBinding" type="tns:ExemploWS">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/
        http" style="document"/>
    <operation name="olaMundo">
        <ns5:PolicyReference
            URI="#ExemploWSPortBinding_olaMundo_WSAT_Policy"
            xmlns:ns5="http://www.w3.org/ns/ws-policy"/>
        <soap:operation soapAction="" />
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>
<service name="ExemploSService">
    <port name="ExemploWSPort" binding="tns:ExemploWSPortBinding
">
        <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
</service>
</definitions>
```

Simple Object Access Protocol

Simple Object Access Protocol (SOAP) é um padrão com tags XML utilizado para enviar requisições e receber respostas de um Web Service.

Exemplo de mensagem SOAP:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Header/>
<S:Body>
    <ns2:olaMundo xmlns:ns2="http://ws.exemplo.pbc/">
</S:Body>
</S:Envelope>
```

Exemplo de resposta SOAP:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
    <ns2:olaMundoResponse xmlns:ns2="http://ws.exemplo.pbc/">
        <return>Ola Mundo!!!</return>
    </ns2:olaMundoResponse>
</S:Body>
</S:Envelope>
```

Criando um Web Service com EJB 3.0

Para criar um serviço web baseado no EJB 3.0, podemos utilizar a Java API for XML-based Web Services (JAX-WS), através dessa API podemos adicionar anotações ao EJB para que ele possa ser utilizado como um serviço web.

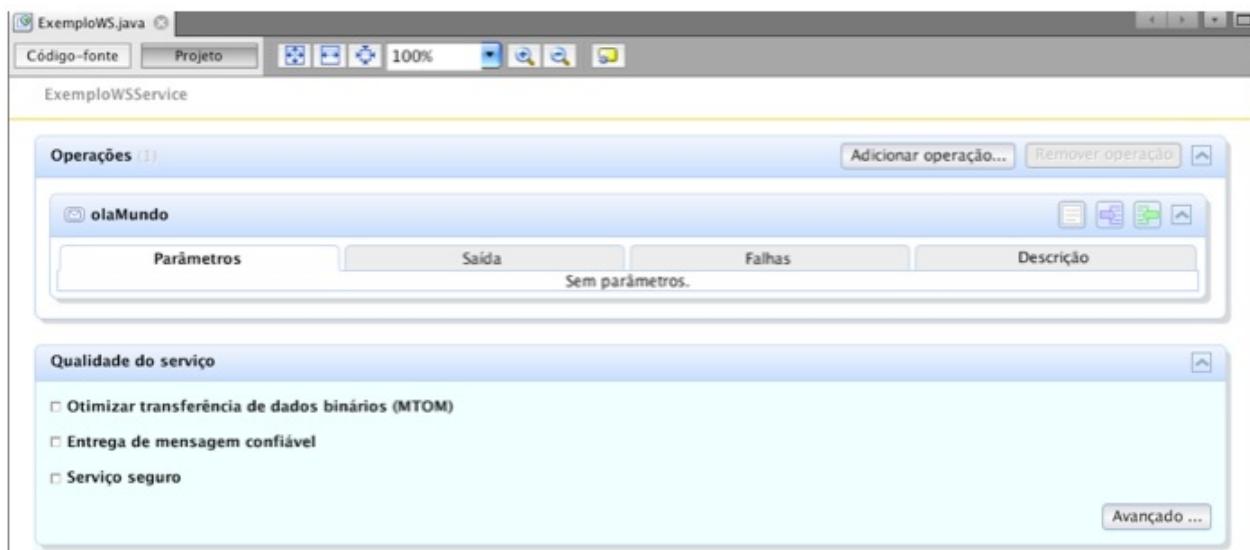
Algumas das anotações que utilizamos para definir um web service:

- **javax.jws.WebService** - Define uma classe como um WS (Está classe precisa ser um Stateless Session Bean).
- **javax.jws.WebMethod** - Define que um método será disponibilizado via WS.
- **javax.jws.WebParam** - Define os parâmetros que serão recebidos pelo WS.
- **javax.jws.WebResult** - Define o nome da tag XML com o conteúdo do retorno, por padrão a tag de retorno do método chama .

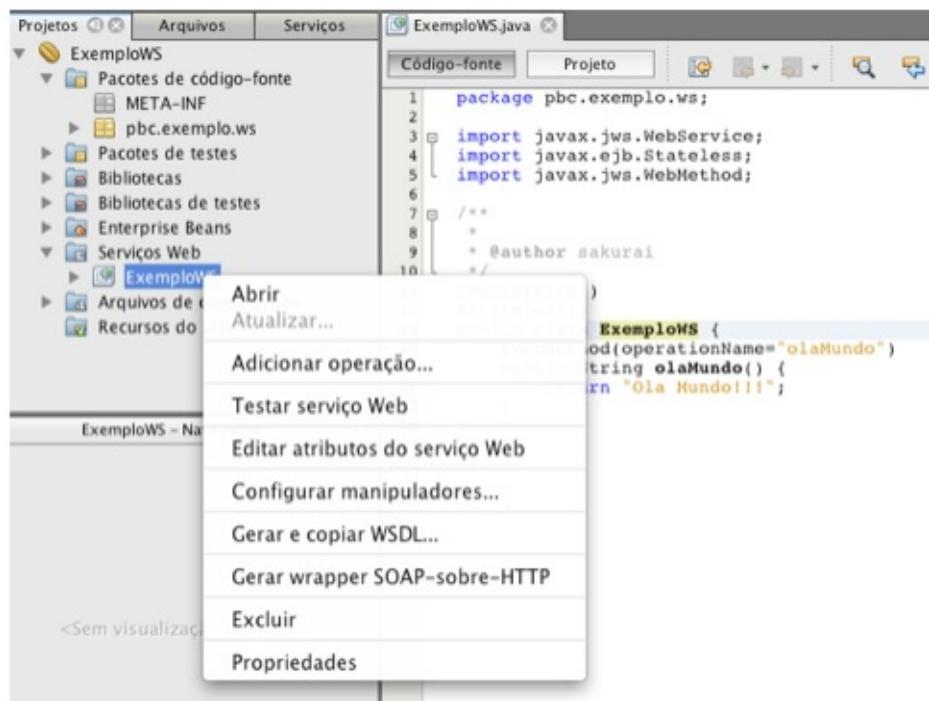
Exemplo de Web Service:

```
@WebService()  
@Stateless()  
public class ExemploWS {  
  
    @WebMethod(operationName = "olaMundo")  
    public String olaMundo() {  
        return "Ola Mundo!!!!";  
    }  
  
}
```

A figura a seguir apresenta um exemplo visual de um web service no NetBeans:



No NetBeans após criar um Web Service e publica-lo no Glassfish, podemos testa-lo clicando com o botão direito do mouse no nome do Web Service (dentro da guia **Serviços Web**) e selecionando a opção **Testar serviço Web** como mostrado na figura a seguir:



Após clicar no item **Testar serviço web** será aberto no navegador uma página para testar os métodos do web service como a figura a seguir, se clicarmos em **WSDL File (arquivo WSDL)** podemos ver o XML do WSDL gerado e se clicarmos na operação **olaMundo** veremos a execução de seu método:

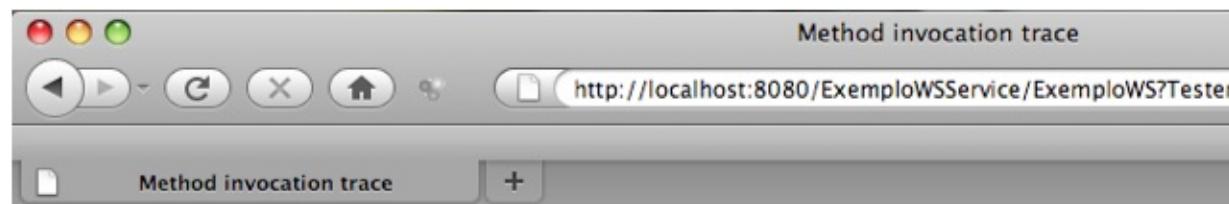
This form will allow you to test your web service implementation ([WSDL File](#))

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

Methods :

```
public abstract java.lang.String pbc.exemplo.ws.ExemploWS.olaMundo()
    olaMundo()
```

Quando clicamos no método **olaMundo** podemos ver a requisição e a resposta no formato SOAP como mostrado na figura a seguir:



Outro exemplo de Web Service

Neste exemplo temos um Web Service que faz uma conexão com o banco de dados para consultar as vendes de livros realizados em um período recebido via parâmetro e retorna um simples texto com as informações das vendas.

```

@WebService()
@Stateless()
public class VendasWS {
    @PersistenceContext(unitName = "LivrariaPU")
    private EntityManager em;

    @WebMethod(operationName = "consultarVendasPorPeriodo")
    public String consultarVendasPorPeriodo(
        @WebParam(name="inicioPeriodo") String inicioPeriodo,
        @WebParam(name="fimPeriodo") String fimPeriodo) {
        String retorno = "";
        try {
            DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy")
;
            VendaDAO dao = new VendaDAO(em);
            List<Venda> vendas =
                dao.procurarPorPeriodo(dateFormat.parse(inicioPeriodo),
                dateFormat.parse(fimPeriodo));

            for(Venda venda : vendas) {
                retorno += venda.getCliente().getNome() + " - "
                    + dateFormat.format(venda.getDataVenda());
                retorno += "\n-----Livros-----";
                for(Livro livro : venda.getLivros()) {
                    retorno += "\nTitulo: " + livro.getTitulo()
                        + " - R$" + livro.getPreco();
                }
                retorno += "\nTotal: R$" + venda.getValorTotal()
                    + "\n-----\n";
            }
        } catch (ParseException ex) {
            ex.printStackTrace(); //Erro ao converter as datas.
        }
        return retorno;
    }
}

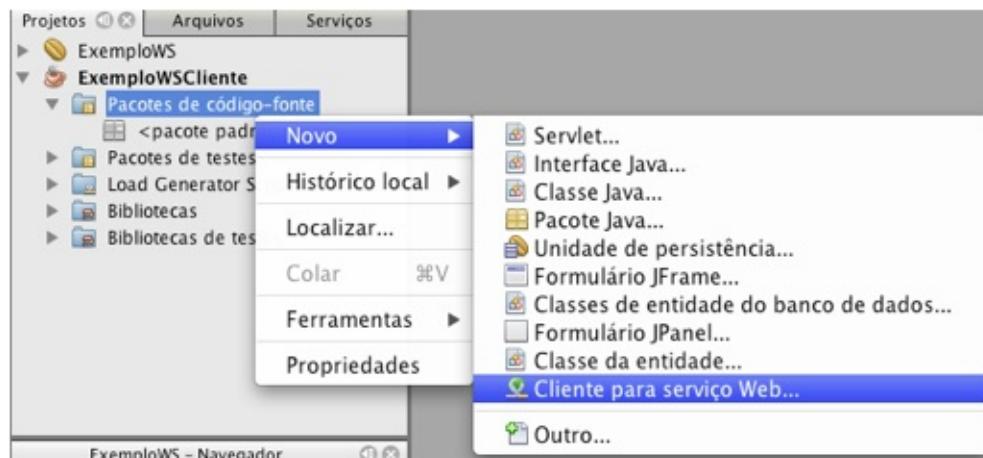
```

Criando um cliente para um Web Service

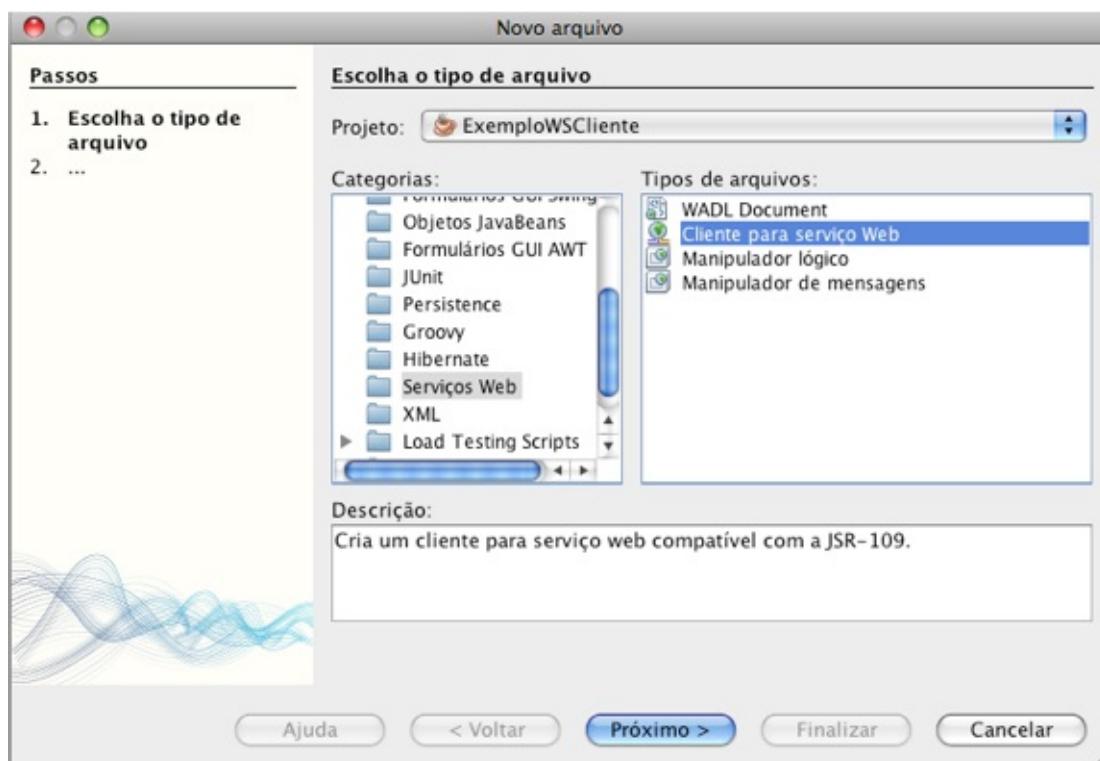
O NetBeans possui uma maneira automática para geração de clientes de web services, para isto basta termos o WSDL do WS, no exemplo abaixo vamos criar um cliente para o WS de Ola Mundo.

Crie um projeto de Aplicativo Java chamado **ExemploWSCliente**.

Para gerar um cliente para o web service, clique com o botão direito em cima de Pacotes de código-fonte, selecione **Novo** e clique em **Cliente para serviço Web...** como mostrado na figura a seguir:



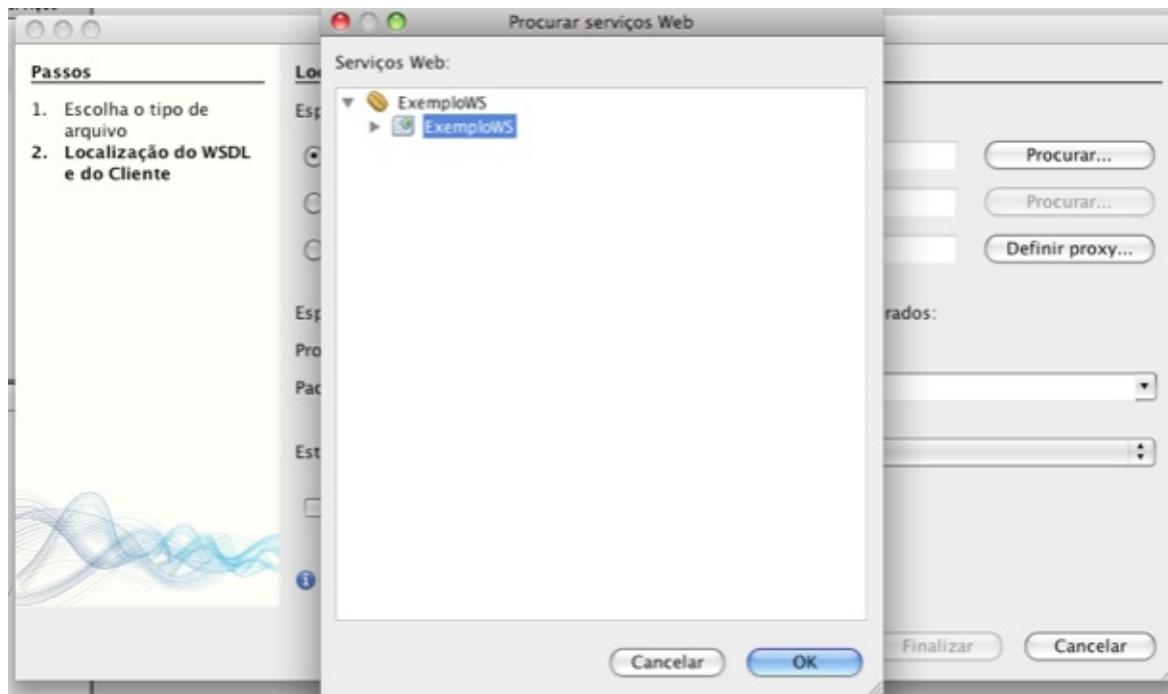
Ou clique em **Novo** e selecione **Outro....**, na tela de Novo arquivo selecione a categoria **Serviços Web** e marque a opção **Cliente para serviço web**.



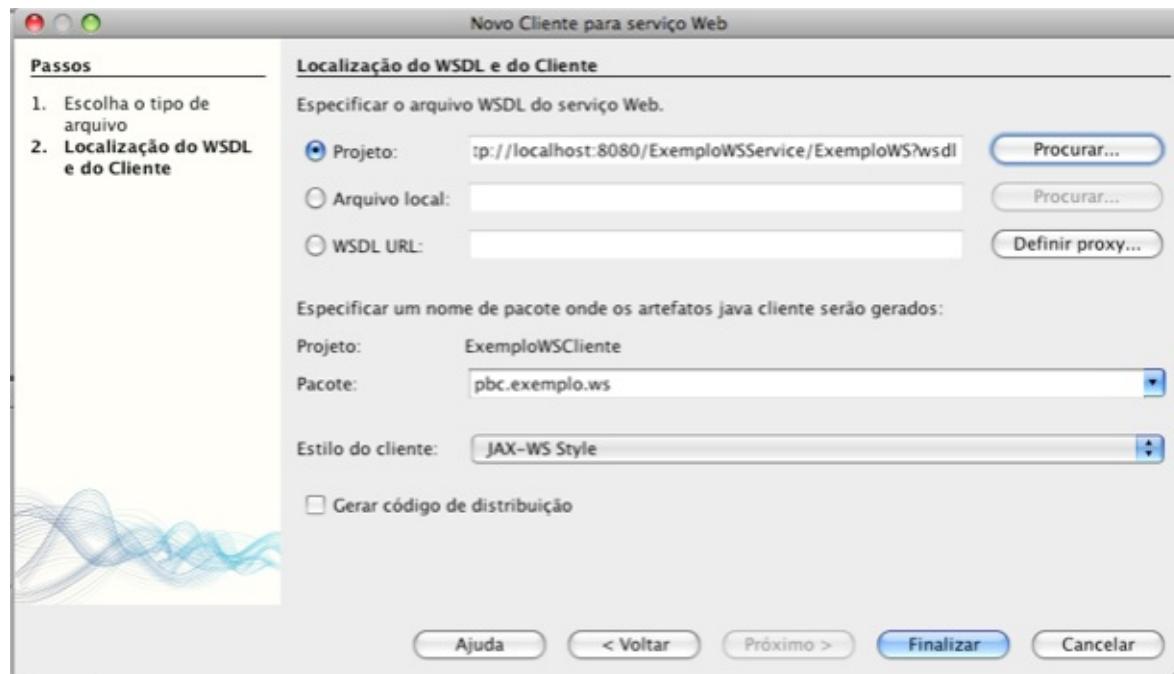
Para especificar o Web Service que será gerado o cliente, podemos fazer isso de duas formas:

- Quando conhecemos o projeto do Web Service:

Na tela de **Novo Cliente para Serviço Web** selecione a opção **Projeto** e clique em **Procurar...**, irá aparecer uma tela onde podemos encontrar os web services criados nos projetos que temos aberto no NetBeans, selecione o Web Service desejado e clique em **OK** como mostrado na figura a seguir:

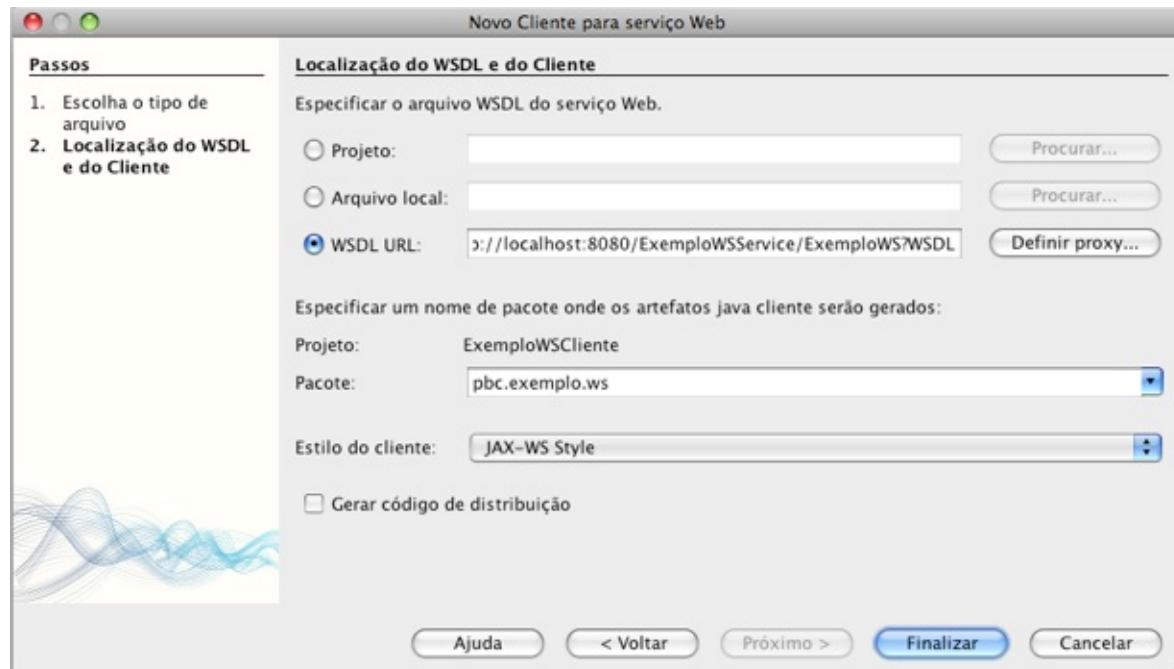


Informe o **Pacote** que será gerado o código do cliente do serviço web e clique em **Finalizar**.



- Quando o Web Service é desenvolvido por outra pessoa e temos apenas o URL do WSDL:

Na tela de Novo Cliente para Serviço Web selecione a opção **WSDL URL** e informe a url do **WSDL**. Informe o **Pacote** que será gerado o código do cliente do serviço web e clique em **Finalizar**.



Após criar o cliente para o serviço web podemos testá-lo, para isto vamos criar uma classe para chamar o Web Service:

```
package pbc.exemplo.ws.cliente;

import pbc.exemplo.ws.ExemploWS;
import pbc.exemplo.ws.ExemploSService;

/**
 * Classe utilizada para testar o web service.
 */
public class TesteWS {
    public static void main(String[] args) {
        ExemploSService service = new ExemploSService();
        ExemploWS port = service.getExemploWSPort();

        System.out.println(port.olaMundo());
    }
}
```

Quando executarmos esta classe teremos a saída no console:

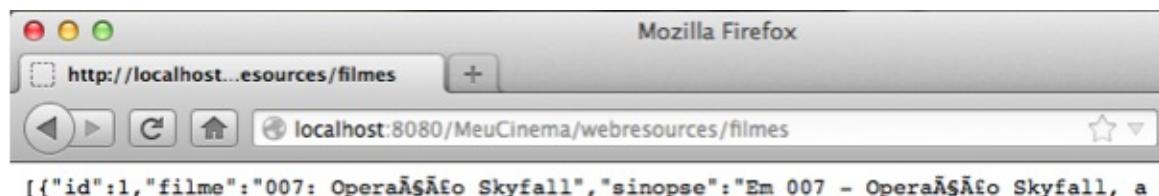
```
Ola Mundo!!!
```

Web Service REST

O serviço web (web service) é uma forma de integração bem utilizada atualmente para realizar a integração entre aplicações. O Web Service REST é uma das formas de criar um serviço web, que utiliza muito o protocolo HTTP para realizar essa integração entre as aplicações.

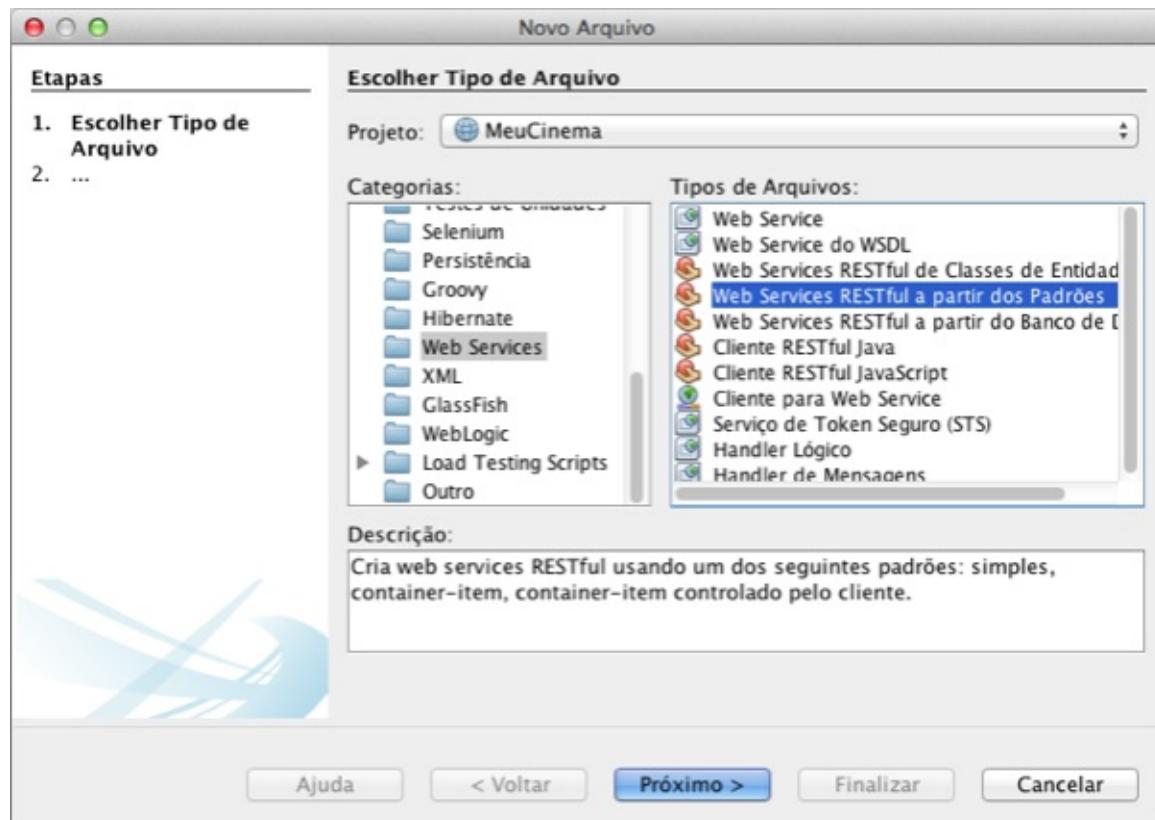
O Web Service REST pode disponibilizar através do protocolo HTTP métodos para manipulação de informações, por exemplo: na figura a seguir mostra uma requisição HTTP através do método GET para a URL

<http://localhost:8080/MeuCinema/webresources/filmes> para obter um recurso, que nesse caso é uma lista de filmes. Se chamar essa URL através de um navegador podemos verificar o retorno desse Web Service REST.

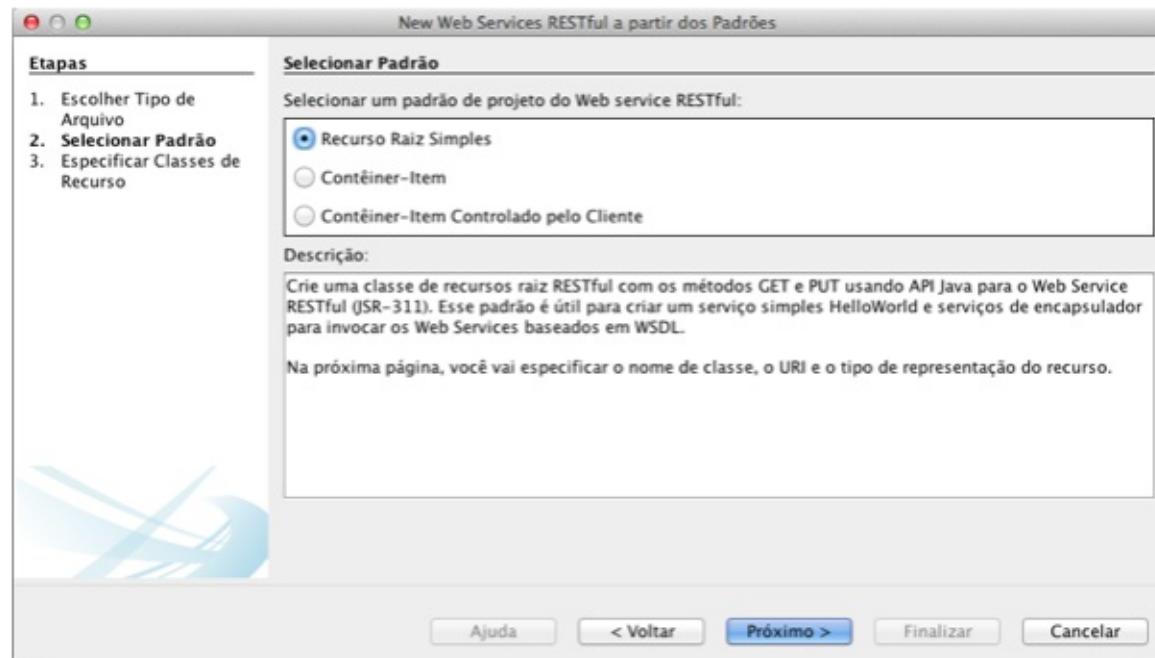


Para exemplificar crie uma Aplicação Web chamada **MeuCinema**. Dentro desse projeto vamos criar um **Web Service REST**, para isso clique com o botão direito do mouse sobre o **Pacotes de Códigos-Fonte** e escolha no menu a opção **Novo** e depois **Outros**.

Na tela de **Novo Arquivo** apresentada na figura a seguir, escolha a categoria **Web Services**, depois escolha o tipo de arquivo **Web Services RESTful a partir dos Padrões** e clique em **Próximo** para continuar.



Na tela de **Novo Web Services RESTful a partir dos Padrões** apresentado na figura a seguir, escolha a opção **Recurso Raiz Simples** e clique em **Próximo**.

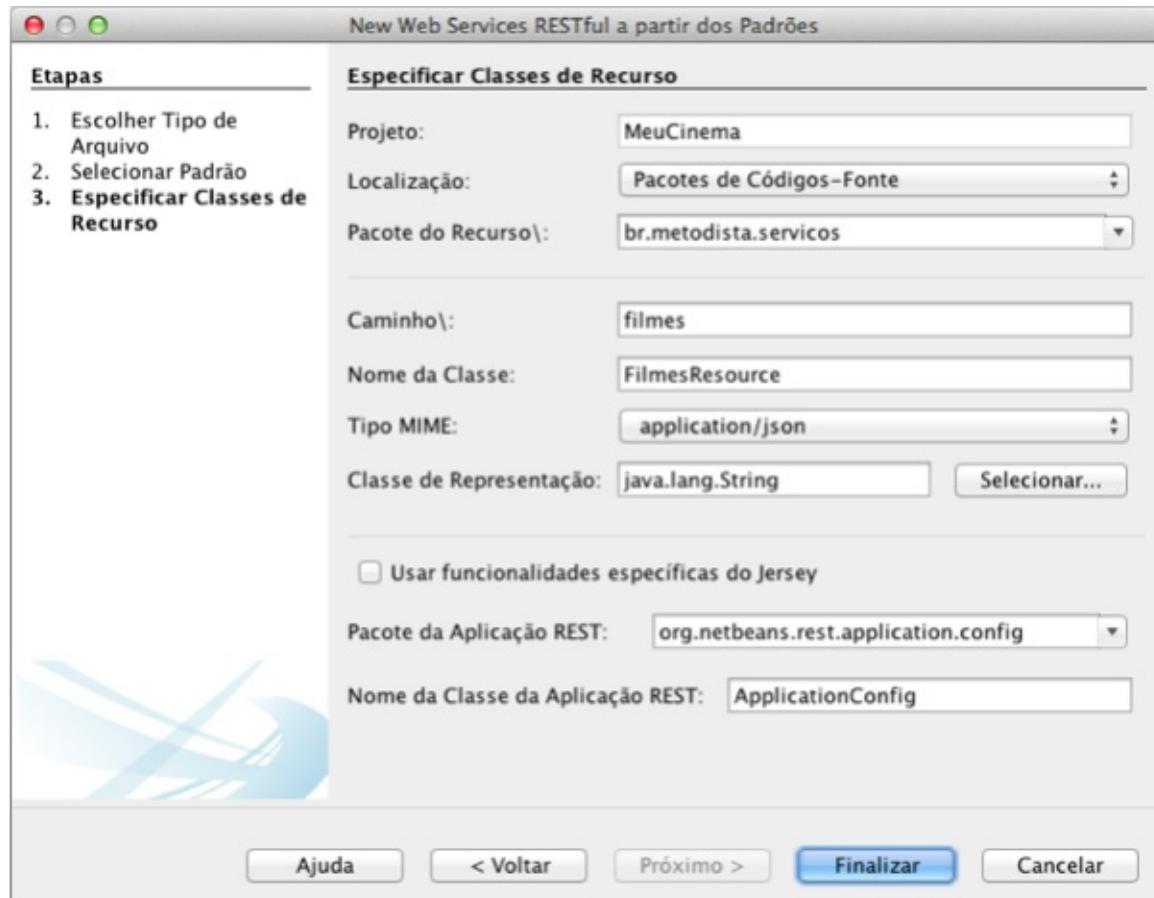


Continuando a criação do Web Service RESTful, defina as opções:

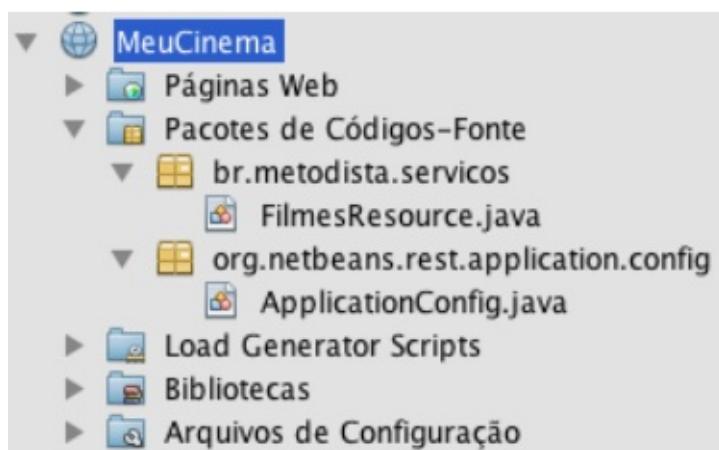
- **Pacotes do Recurso:** br.metodista.servicos
- **Caminho:** filmes
- **Nome da Classe:** FilmesResource

- **Tipo MIME:** application/json

E clique em **Finalizar** para terminar a criação do Web Service RESTful.



A estrutura do projeto ficará com a seguinte aparência:



A classe **FilmesResource** é o Web Service REST, para interagir com ele será utilizado os métodos do HTTP, o NetBeans gera essa classe com os métodos GET e PUT:

```
@GET  
@Produces("application/json")  
public String getJson() {  
    //TODO return proper representation object  
    throw new UnsupportedOperationException();  
}  
  
@PUT  
@Consumes("application/json")  
public void putJson(String content) {  
}
```

Nesses métodos vamos trocar informações no formato JSON.

Para converter objeto Java em JSON, podemos usar uma API simples do Google chamada **GSon** (<https://code.google.com/p/google-gson>) que realiza essa conversão de maneira fácil. Faça o download da biblioteca gson e adicione na biblioteca do projeto.

Vamos criar uma classe chamada **Filme** no pacote **br.metodista.modelo** para representar o recurso que desejamos compartilhar através do Web Service REST.

```

package br.metodista.modelo;

public class Filme {
    private Long id;
    private String filme;
    private String sinopse;
    private String genero;
    private Integer duracao;
    private String trailer;

    public Filme(Long id, String filme, String sinopse,
                String genero, Integer duracao, String trailer) {

        this.id = id;
        this.filme = filme;
        this.sinopse = sinopse;
        this.genero = genero;
        this.duracao = duracao;
        this.trailer = trailer;
    }

    // get e set dos atributos.
}

```

Vamos agora alterar a classe **FilmesResource**, dentro dela crie uma lista de filmes e altere o construtor para inicializar alguns filmes para teste:

```

private static List<Filme> filmes;

public FilmeResource() {
    filmes = new ArrayList<Filme>();
    filmes.add(new Filme(1L, "007: Operação Skyfall",
                        "Em 007 - Operação Skyfall, a lealdade de Bond a M é testada quando o seu passado volta a atormentá-la. Com a MI6 sendo atacada, o 007 precisa rastrear e destruir a ameaça, não importando o quanto pessoal será o custo disto.", "Ação", 145, ""));
    filmes.add(new Filme(2L, "Atividade Paranormal 4",
                        "Atividade Paranormal 4 se passa em 2011, cinco

```

anos depois de Katie matar seu namorado Micah, sua irmã Kristi e seu marido Daniel e levar seu bebê, Hunter. A história centra-se em Alice e sua mãe, experimentando atividades estranhas quando os novos vizinhos mudam-se para a casa ao lado.", "Suspense", 89, ""));
filmes.add(new Filme(3L, "Até Que A Sorte Nos Separe", "Livremente inspirado no best-seller Casais Inteligentes Enriquecem Juntos, Até que a sorte nos separe é uma comédia romântica sobre as aventuras de um casal que consegue, 2 vezes, o quase impossível: Ganhar na loteria e gastar tudo em dez anos...o filme fará com que o público se divirta e se identifique com os segredos e trapalhadas de uma família descobrindo que uma boa conta bancária até ajuda, mas desde que você saiba o que fazer com ela.", "Comédia", 104, ""));
filmes.add(new Filme(4L, "Busca implacável 2", "Em Istambul, agente aposentado da CIA, Bryan Mills, e sua mulher são feitos reféns pelo pai de um sequestrador que Mills matou durante o resgate de sua filha no primeiro filme.", "Ação", 94, ""));
filmes.add(new Filme(5L, "Gonzaga de Pai para Filho", "Um pai e um filho, dois artistas, dois sucessos. Um do sertão nordestino, o outro carioca do Morro de São Carlos; um de direita, o outro de esquerda. Encontros, desencontros e uma trilha sonora que emocionou o Brasil. Esta é a história de Luiz Gonzaga e Gonzaguinha, e de um amor que venceu o medo e o preconceito e resistiu à distância e ao esquecimento.", "Drama", 130, ""));
filmes.add(new Filme(6L, "Hotel Transilvânia 3D", "Bem-vindos ao Hotel Transilvânia, o luxuoso resort 'cinco estacas' de Drácula, onde monstros e suas famílias podem viver livres da intromissão do mundo humano. Mas há um fato pouco conhecido sobre Drácula: ele não é apenas o príncipe das trevas, mas também é um pai super-protetor de uma filha adolescente, Mavis, e inventa contos de perigo para dissuadir seu espírito aventureiro.", "Infantil", 93, ""));
filmes.add(new Filme(7L, "Possessão", "Uma jovem compra uma caixa antiga sem saber que dentro do

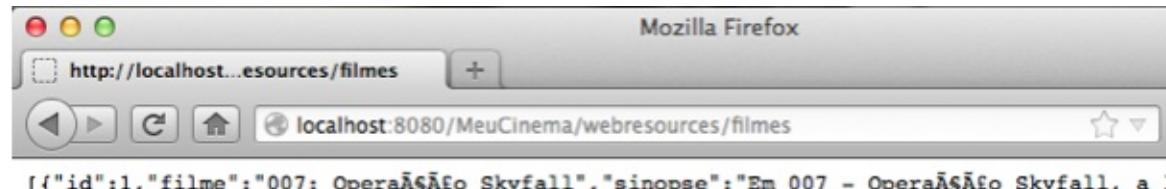
```
    objeto existe um espírito malicioso. Os pais da  
    menina tentam encontrar uma maneira de acabar com  
    a maldição que domina sua filha.", "Terror", 92,  
    ""));  
}
```

Também vamos alterar o método **getJson()** para devolver uma String com contendo todos os filmes no formato um JSON:

```
@GET  
@Produces("application/json")  
public String getFilmes() {  
    Gson gson = new Gson();  
    return gson.toJson(filmes);  
}
```

Para testas se o método GET do Web Service REST está funcionando, publique a aplicação **MeuCinema** no GlassFish e acesse a URL:

<http://localhost:8080/MeuCinema/webresources/filmes>.



Deverá apresentar a lista de filmes no formato JSON.

Vamos adicionar também um método para obter apenas um filme através de seu ID, na classe **FilmesResource** adicione o método:

```

@GET
@Path("{filmeId}")
@Produces("application/json")
public String getFilme(@PathParam("filmeId")
String filmeId) {

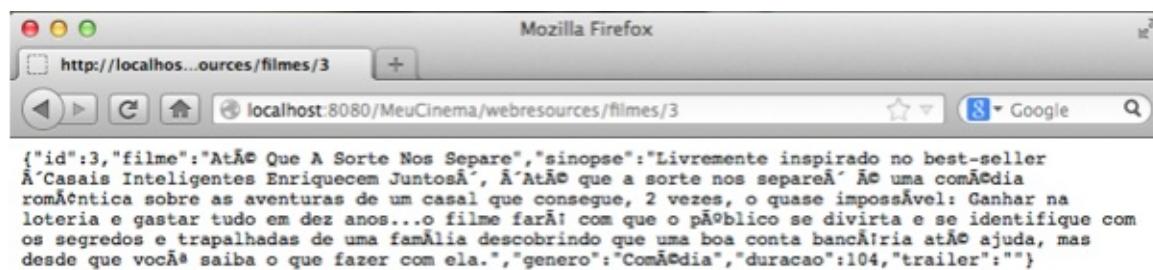
    for(Filme f : filmes) {
        if(f.getId() == Long.valueOf(filmeId)) {
            Gson gson = new Gson();
            return gson.toJson(f);
        }
    }

    return null;
}

```

Para chamar esse método acesse a URL

[http://localhost:8080/MeuCinema/webresources/filmes/3.](http://localhost:8080/MeuCinema/webresources/filmes/3)



Consumindo um web service REST

Para consumir um Web Service REST, existem diversas implementações possíveis, uma delas é através da API Jersey, que é a implementação de referência do JavaEE.

Crie uma aplicação web chamada **MeuCinemaJSF**, que utiliza o framework do JavaServer Faces para criação das páginas WEB.

Vamos alterar uma página inicial **index.xhtml**, para que ela utilize um managed bean para consumir esse web serviço, a página ficará assim:

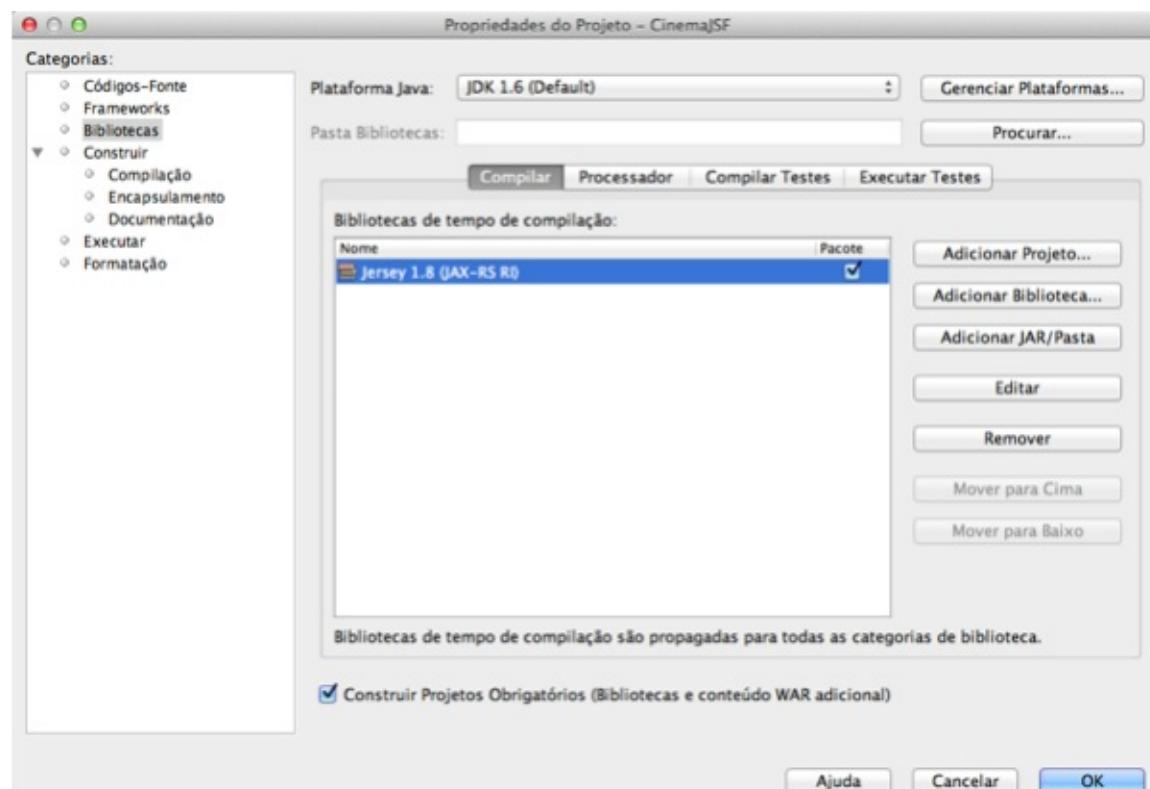
```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
    <title>Meu Cinema</title>
</h:head>
<h:body>
    <h1>Filmes em cartaz</h1>
    <h:outputText value="#{cinemaMB.filmesEmCartaz}" />
</h:body>
</html>

```

O outputText irá chamar o método **getFilmesEmCartaz()** da managed bean **CinemaMB**, que chama o web service REST que traz todos os filmes em cartaz.

Para utilizar a API do Jersey dentro da aplicação, clique com o botão direito do mouse em cima do nome do projeto e escolha o item Propriedades. Na tela de propriedades acesse a categoria Bibliotecas e adicione a biblioteca Jersey 1.8 (JAX-RS RI) através do menu Adicionar Biblioteca como mostrado na figura a seguir:



Vamos criar a managed bean **CinemaMB** com a implementação do método:

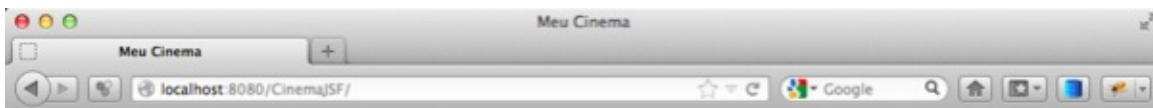
```
package br.metodista.managedbean;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import javax.faces.bean.ManagedBean;

@ManagedBean
public class CinemaMB {
    public String getFilmesEmCartaz() {
        Client c = Client.create();
        WebResource wr = c.resource(
            "http://localhost:8080/MeuCinema/webresources/filmes");
        return wr.get(String.class);
    }
}
```

Com a classe Client é possível obter um resource web (recurso web) através da URL do web service REST, e com esse recurso é possível chamar os métodos que o web service REST suporta, como: get, post, put, delete, etc.

Ao chamar o método `wr.get(String.class)`, estamos esperando que a chamada para esse serviço devolva uma String, nesse exemplo essa String vem no formato JSON (JavaScript Object Notation), mas poderia ser uma String simples, um formato XML, etc. Ao executar a aplicação **CinemaJSF** teremos a tela da figura a seguir:



[{"id":1,"filme":"007: Operação Skyfall","sinopse":"Em 007 - Operação Skyfall, a lealdade de Bond a M é testada quando o seu passado volta a atormentá-la. Com a MI6 sendo atacada, o 007 precisa rastrear e destruir a ameaça, não importando o quanto pessoal será o custo disto.", "genero":"Ação","duracao":145,"trailer":""}, {"id":2,"filme":"Atividade Paranormal 4","sinopse":"Atividade Paranormal 4 se passa em 2011, cinco anos depois de Katie matar seu namorado Micah, sua irmã Kristi e seu marido Daniel e levar seu bebê, Hunter. A história centra-se em Alice e sua mãe, experimentando atividades estranhas quando os novos vizinhos mudam-se para a casa ao lado.", "genero":"Suspense","duracao":89,"trailer":""}, {"id":3,"filme":"Até Que A Sorte Nos Separe","sinopse":"Livremente inspirado no best-seller 'Casais Inteligentes Enriquecem Juntos', 'Até que a sorte nos separe' é uma comédia romântica sobre as aventuras de um casal que consegue, 2 vezes, o quase impossível: Ganhar na loteria e gastar tudo em dez anos...o filme fará com que o público se divirta e se identifique com os segredos e trapalhadas de uma família descobrindo que uma boa conta bancária até ajuda, mas desde que você saiba o que fazer com ela.", "genero":"Comédia","duracao":104,"trailer":""}, {"id":4,"filme":"Busca Impiedosa 2","sinopse":"Em Istambul, agente aposentado da CIA, Bryan Mills, e sua mulher são feitos reféns pelo pai de um sequestrador que Mills matou durante o resgate de sua filha no primeiro filme.", "genero":"Ação","duracao":94,"trailer":""}, {"id":5,"filme":"Gonzaga do Pai para Filho","sinopse":"Um pai e um filho, dois artistas, dois sucessos. Um do sertão nordestino, o outro carioca do Morro de São Carlos; um de direita, o outro de esquerda. Encontros, desencontros e uma trilha sonora que emocionou o Brasil. Esta é a história de Luiz Gonzaga e Gonzaguinha, e de um amor que venceu o medo e o preconceito e resistiu à distância e ao esquecimento.", "genero":"Drama","duracao":130,"trailer":""}, {"id":6,"filme":"Hotel Transilvânia 3D","sinopse":"Bem-vindos ao Hotel Transilvânia, o luxuoso resort 'cinco estrelas' de Drácula, onde monstros e suas famílias podem viver livres da intromissão do mundo humano. Mas há um fato pouco conhecido sobre Drácula: ele não é apenas o príncipe das trevas, mas também é um pai super-protetor de uma filha adolescente, Mavis, que inventa contos de perigo para dissuadir seu espírito aventureiro.", "genero":"Infantil","duracao":93,"trailer":""}, {"id":7,"filme":"Possessão","sinopse":"Uma jovem compra uma caixa antiga sem saber que dentro do objeto existe um espírito malicioso. Os pais da menina tentam encontrar uma maneira de acabar com a maldição que domina sua filha.", "genero":"Terror","duracao":92,"trailer":""}]

Para converter esse JSON em objeto Java, podemos usar uma API simples do Google chamada **GSon** (<https://code.google.com/p/google-gson>) que realiza essa conversão de maneira fácil. Primeiro vamos criar a classe Filme que irá representar cada filme da lista:

```
package br.metodista.modelo;

public class Filme {
    private Long id;
    private String filme;
    private String sinopse;
    private String genero;
    private Integer duracao;
    private String trailer;

    public Filme() {
    }

    public Long getId() {
        return id;
    }

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

```
public String getFilme() {
    return filme;
}

public void setFilme(String filme) {
    this.filme = filme;
}

public String getSinopse() {
    return sinopse;
}

public void setSinopse(String sinopse) {
    this.sinopse = sinopse;
}

public String getGenero() {
    return genero;
}

public void setGenero(String genero) {
    this.genero = genero;
}

public Integer getDuracao() {
    return duracao;
}

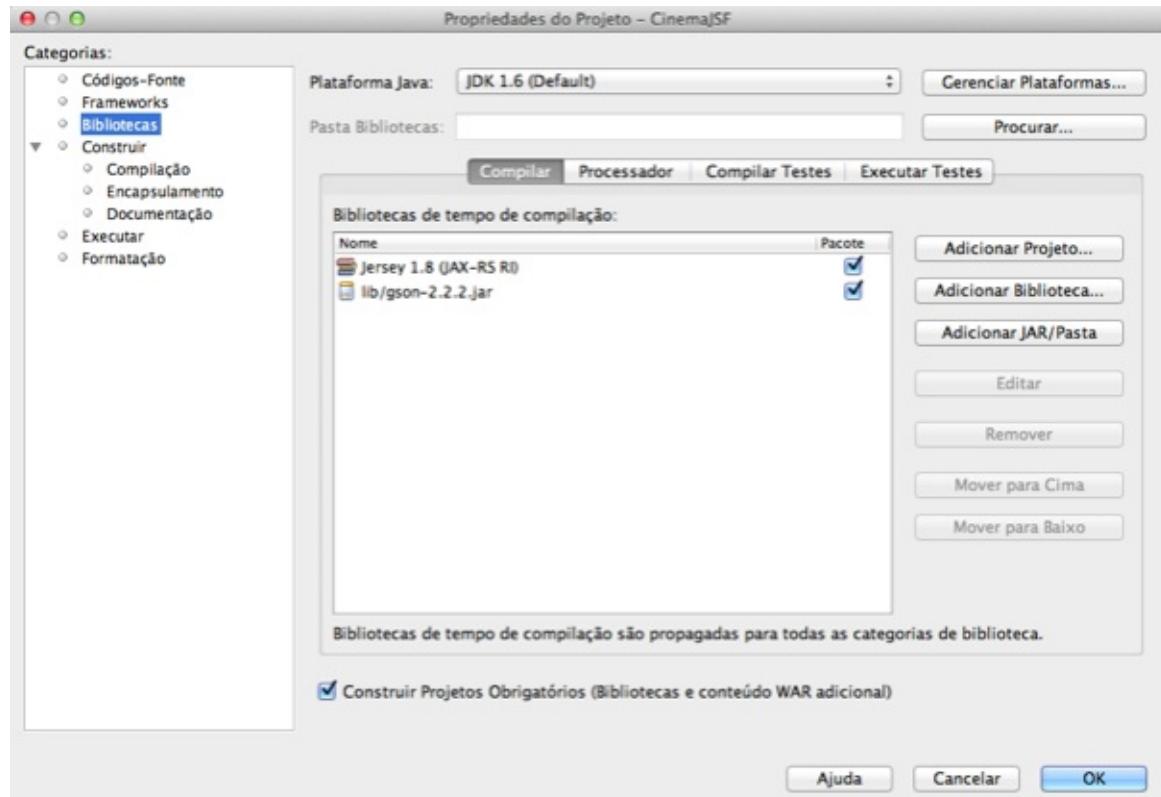
public void setDuracao(Integer duracao) {
    this.duracao = duracao;
}

public String getTrailer() {
    return trailer;
}

public void setTrailer(String trailer) {
    this.trailer = trailer;
}
```

}

Adicione a biblioteca do gson no projeto, clique com o botão direito do mouse em cima do nome do projeto e escolha o item Propriedades. Na tela de propriedades acesse a categoria Bibliotecas e adicione a biblioteca **gson-2.2.2.jar** através do menu Adicionar JAR / Pasta.



Vamos alterar o CinemaMB para utilizar a API do gson e converter o JSON em uma lista de filmes:

```

package br.metodista.managedbean;

import br.metodista.modelo.Filme;
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import java.util.List;
import javax.faces.bean.ManagedBean;

@ManagedBean
public class CinemaMB {
    public List<Filme> getFilmesEmCartaz() {
        Client c = Client.create();
        WebResource wr = c.resource
            ("http://localhost:8080/MeuCinema/webresources/filmes");
        String json = wr.get(String.class);

        Gson gson = new Gson();
        return gson.fromJson(json,
            new TypeToken<List<Filme>>() {}.getType());
    }
}

```

Agora vamos mudar a página index.xhtml para mostrar uma tabela com os filmes:

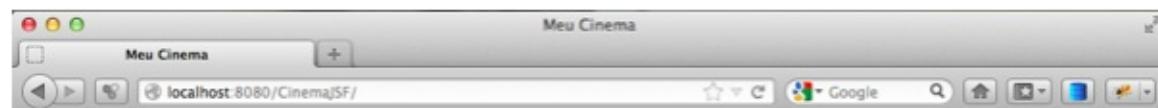
```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<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>Meu Cinema</title>
</h:head>
<h:body>
    <h1>Filmes em cartaz</h1>
    <h: dataTable value="#{cinemaMB.filmesEmCartaz}"
                  var="f" width="100%">

```

```
<h:column>
    <f:facet name="header">
        <h:outputText value="Titulo"/>
    </f:facet>
    <h:outputText value="#{f.filme}"/>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Genero"/>
    </f:facet>
    <h:outputText value="#{f.genero}"/>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Duração"/>
    </f:facet>
    <h:outputText value="#{f.duracao} min"/>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Sinopse"/>
    </f:facet>
    <h:outputText value="#{f.sinopse}"/>
</h:column>
</h:datatable>
</h:body>
</html>
```

A tela ficará com a seguinte aparência:



Filmes em cartaz

Titulo	Genero	Duração	Sinopse
007: Operação Skyfall	Ação	145 min	Em 007 - Operação Skyfall, a lealdade de Bond a M é testada quando o seu passado volta a atormentá-la. Com a MI6 sendo atacada, o 007 precisa rastrear e destruir a ameaça, não importando o quão pessoal será o custo disto.
Atividade Paranormal 4	Suspense	89 min	Atividade Paranormal 4 se passa em 2011, cinco anos depois de Katie matar seu namorado Micah, sua irmã Kristi e seu marido Daniel e levar seu bebê, Hunter. A história centra-se em Alice e sua mãe, experimentando atividades estranhas quando os novos vizinhos mudam-se para a casa ao lado.
Até Que A Sorte Nos Separe	Comédia	104 min	Livremente inspirado no best-seller 'Casais Inteligentes Enriquecem Juntos', 'Até que a sorte nos separe' é uma comédia romântica sobre as aventuras de um casal que consegue, 2 vezes, o quase impossível: Ganhar na loteria e gastar tudo em dez anos...o filme fará com que o público se divirta e se identifique com os segredos e trapalhadas de uma família descobrindo que uma boa conta bancária até ajuda, mas desde que você saiba o que fazer com ela.
Busca implacável 2	Ação	94 min	Em Istambul, agente aposentado da CIA, Bryan Mills, e sua mulher são feitos reféns pelo pai de um sequestrador que Mills matou durante o resgate de sua filha no primeiro filme.
Gonzaga de Pai para Filho	Drama	130 min	Um pai e um filho, dois artistas, dois sucessos. Um do sertão nordestino, o outro carioca do Morro de São Carlos; um de direita, o outro de esquerda. Encontros, desencontros e uma trilha sonora que emocionou o Brasil. Esta é a história de Luiz Gonzaga e Gonzaguinha, e de um amor que venceu o medo e o preconceito e resistiu à distância e ao esquecimento.
Hotel Transilvânia 3D	Infantil	93 min	Bem-vindos ao Hotel Transilvânia, o luxuoso resort 'cinco estacas' de Drácula, onde monstros e suas famílias podem viver livres da intromissão do mundo humano. Mas há um fato pouco conhecido sobre Drácula: ele não é apenas o príncipe das trevas, mas também é um pai super-protetor de uma filha adolescente, Mavis, e inventa contos de perigo para dissuadir seu espírito aventureiro.
Possessão	Terror	92 min	Uma jovem compra uma caixa antiga sem saber que dentro do objeto existe um espírito malicioso. Os pais da menina tentam encontrar uma maneira de acabar com a maldição que domina sua filha.

GET, POST, PUT e DELETE

Nesse exemplo vamos criar um web service REST que permite consultar, inserir, alterar e remover (CRUD) palestras. Para deixar o exemplo mais real, vamos implementar desde o banco de dados até o web service REST.

Criando o Web Service REST

Para começar vamos criar a sequência e tabela no banco de dados, nesse exemplo estou utilizando Oracle, mas se preferir é só adaptar essa parte de persistência para o banco de dados que você for usar.

Script

```
CREATE SEQUENCE palestra_seq INCREMENT BY 1 START WITH 1 NOCACHE
  NOCYCLE;

CREATE TABLE palestra (
    id NUMBER(8) NOT NULL,
    titulo VARCHAR2(200) NOT NULL,
    descricao VARCHAR2(1000) NOT NULL,
    palestrante VARCHAR2(200) NOT NULL,
    PRIMARY KEY (id)
)
```

Crie uma aplicação web chamada **PalestraWSREST** no NetBeans, essa aplicação posteriormente será publicada no servidor de aplicações web Glassfish.

Dentro da aplicação vamos criar a entidade **Palestra**:

```
package evento.modelo;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity
@NamedQueries({
    @NamedQuery(name = "Palestra.consultarTodos",
        query = "SELECT p FROM Palestra p ORDER BY p.titulo"
    )
})
@SequenceGenerator(name = "palestra_seq", sequenceName = "palestra_seq",
    allocationSize = 1, initialValue = 1)
public class Palestra implements Serializable {
    private static final long serialVersionUID = -2806694270931063283L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "palestra_seq")
    private Long id;
    private String titulo;
    private String descricao;
    private String palestrante;

    public Long getId() {
        return id;
    }

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

    public String getTitulo() {
        return titulo;
    }
```

```

public void setTitulo(String titulo) {
    this.titulo = titulo;
}

public String getDescricao() {
    return descricao;
}

public void setDescricao(String descricao) {
    this.descricao = descricao;
}

public String getPalestrante() {
    return palestrante;
}

public void setPalestrante(String palestrante) {
    this.palestrante = palestrante;
}
}

```

Na entidade Palestra, declarei uma consulta nomeada chamada "Palestra.consultarTodos" que será usada para consultar todas as palestras cadastradas.

Agora vamos criar o DAO para salvar, alterar, remover, consultar palestra por id e consultar todas palestras.

```

package evento.dao;

import evento.modelo.Palestra;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.Query;

public class PalestraDAO {
    private EntityManager em;

    public PalestraDAO(EntityManager em) {

```

```
        this.em = em;
    }

    public Palestra salvar(Palestra p) throws Exception {
        if(p.getId() == null) {
            em.persist(p);
        } else {
            if(!em.contains(p)) {
                if(em.find(Palestra.class, p.getId()) == null) {
                    throw new Exception("Erro ao atualizar a palestra!");
                }
            }
            p = em.merge(p);
        }

        return p;
    }

    public void remover(Long id) {
        Palestra p = em.find(Palestra.class, id);
        em.remove(p);
    }

    public Palestra consultarPorId(Long id) {
        return em.find(Palestra.class, id);
    }

    public List<Palestra> consultarTodos() {
        Query q = em.createNamedQuery("Palestra.consultarTodos")
        ;
        return q.getResultList();
    }
}
```

No Glassfish, crie um pool de conexões com o banco de dados, nesse exemplo eu criei um recurso JDBC chamado `jdbc/palestras`.

Com o pool criado e funcionando, cria a unidade de persistência. Note que o nome da unidade é `name="PalestraPU"` se preferir pode criar com outro nome, mas lembre de usar o nome que você criou no EJB.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
</persistence>
<persistence-unit name="PalestraPU" transaction-type="JTA">
  <jta-data-source>jdbc/palestras</jta-data-source>
  <exclude-unlisted-classes>false</exclude-unlisted-classes>
  <properties/>
</persistence-unit>
</persistence>
```

Agora vamos criar a interface remota do EJB:

```
package evento.ejb;

import evento.modelo.Palestra;
import java.util.List;
import javax.ejb.Remote;

@Remote
public interface PalestraRemote {
    public Palestra salvar(Palestra p) throws Exception;
    public void remover(Long id);
    public Palestra consultarPorId(Long id);
    public List<Palestra> consultarTodos();
}
```

E também a sua implementação:

```
package evento.ejb;

import evento.dao.PalestraDAO;
import evento.modelo.Palestra;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class PalestraBean implements PalestraRemote {
    @PersistenceContext(unitName = "PalestraPU")
    private EntityManager em;

    @Override
    public Palestra salvar(Palestra p) throws Exception {
        PalestraDAO dao = new PalestraDAO(em);
        return dao.salvar(p);
    }

    @Override
    public void remover(Long id) {
        PalestraDAO dao = new PalestraDAO(em);
        dao.remover(id);
    }

    @Override
    public Palestra consultarPorId(Long id) {
        PalestraDAO dao = new PalestraDAO(em);
        return dao.consultarPorId(id);
    }

    @Override
    public List<Palestra> consultarTodos() {
        PalestraDAO dao = new PalestraDAO(em);
        return dao.consultarTodos();
    }
}
```

Na aplicação crie um novo Web Service REST com o nome **PalestraResource**.

```
package evento.ws;

import com.google.gson.Gson;
import evento.ejb.PalestraRemote;
import evento.modelo.Palestra;
import javax.ejb.EJB;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Produces;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PUT;
import javax.ws.rs.PathParam;

/**
 * REST Web Service
 */
@Path("palestra")
public class PalestraResource {
    private Gson gson = new Gson();

    @EJB
    private PalestraRemote ejb;

    public PalestraResource() {
    }

    @GET
    @Produces("application/json")
    public String getPalestras() {
        return gson.toJson(ejb.consultarTodos());
    }

    @Path("{id}")
    @GET
```

```

    @Produces("application/json")
    public String getPalestra(@PathParam("id") String palestraId
) {
    return gson.toJson(ejb.consultarPorId(Long.valueOf(palestraId)));
}

@POST
@Produces("application/json")
@Consumes("application/json")
public String salvarPalestra(String palestra) {
    try {
        Palestra p = gson.fromJson(palestra, Palestra.class);
;
        return gson.toJson(ejb.salvar(p));
    } catch (Exception e) {
        return null;
    }
}

@DELETE
@Path("/{id}")
public void removerPalestra(final @PathParam("id") String id
) {
    ejb.remover(Long.valueOf(id));
}

@PUT
@Consumes("application/json")
public void putPalestra(String palestra) {
    salvarPalestra(palestra);
}
}

```

Nesse Web Service REST estou usando o GSon para fazer as conversões de Java para JSON. Nela estamos disponibilizando a consulta de todas as palestras e consulta da palestra pelo seu ID, permitimos salvar, alterar e remover uma palestra.

Note como cada método HTTP está sendo usado com um propósito.

Método HTTP	Uso
GET	Obter algum ou alguns recursos, nesse caso para consultar todas palestras e consultar uma palestra por id
POST	Inserir uma nova palestra.
PUT	Atualizar uma palestra.
DELETE	Remover uma palestra.

Altere também a anotação **ApplicationPath** da classe **ApplicationConfig** só para facilitar o nome de acesso aos recursos, nesse exemplo alterei para "recursos" :

```
package evento.ws;

import java.util.Set;
import javax.ws.rs.core.Application;

@javax.ws.rs.ApplicationPath("recursos")
public class ApplicationConfig extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        addRestResourceClasses(resources);
        return resources;
    }

    /**
     * Do not modify addRestResourceClasses() method.
     * It is automatically populated with
     * all resources defined in the project.
     * If required, comment out calling this method in getClasses().
     */
    private void addRestResourceClasses(Set<Class<?>> resources)
    {
        resources.add(evento.ws.PalestraResource.class);
    }
}
```

Pronto, a implementação do web service REST está criada, agora falta implementar o cliente desse serviço.

Consumindo um Web Service REST

Vamos criar uma nova aplicação web para consumir o web service REST que criamos.

Nessa aplicação web vamos criar um modelo para representar a **Palestra**:

```
package evento.modelo;

public class Palestra {
    private Long id;
    private String titulo;
    private String descricao;
    private String palestrante;

    public Long getId() {
        return id;
    }

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

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public String getPalestrante() {
        return palestrante;
    }

    public void setPalestrante(String palestrante) {
        this.palestrante = palestrante;
    }
}
```

Vamos criar uma tela simples usando JSF para fazer o CRUD da palestra e mostrar todas as palestras cadastradas como mostrado na figura a seguir:

Palestras

Título:

Descrição:

Palestrante:

JPA e a festa na piscina

Carlos Faces
Criando pool de conexões com o JPA.

No meio do caminho tinha um bug, tinha um bug no meio do caminho

Lana Lang
Caça aos bugs, aprenda as técnicas milenares de depuração de código.

Ops! O servidor caiu

Claudia String
Você não sabe porque seu servidor de produção cai bem na hora que mais vende? Vejas as principais causas e soluções para esse problema.

O código da página **index.xhtml** fica assim:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Palestras</title>
  </h:head>
  <h:body>
    <div style="text-align: center; width: 100%">
      <h1>Palestras</h1>
    </div>
```

```

<h:form id="formulario">
    <h:panelGrid id="dados" columns="2" width="100%">
        <h:outputText value="Título: "/>
        <h:inputText value="#{palestraMB.palestra.titulo}" style="width: 100%"/>
        <h:outputText value="Descrição: "/>
        <h:inputTextarea value="#{palestraMB.palestra.descricao}" cols="40" rows="5" style="width: 100%"/>
        <h:outputText value="Palestrante: "/>
        <h:inputText value="#{palestraMB.palestra.palestrante}" style="width: 100%"/>
        <h:commandButton value="Salvar" action="#{palestraMB.salvar}"/>
    </h:panelGrid>
    <h:panelGroup id="groupPalestras">
        <h:dataTable id="palestras" value="#{palestraMB.palestras}" var="p" width="100%">
            <h:column>
                <h2><h:outputText value="#{p.titulo}"/></h2>
                <i><b><h:outputText value="#{p.palestrante}"/></b></i>
            <br/>
                <h:outputText value="#{p.descricao}"/><br/>
                <h:commandButton value="Editar" action="#{palestraMB.editar(p)}">
                    <f:ajax render=":formulario:dados"/>
                </h:commandButton>
                <h:commandButton value="Remover" action="#{palestraMB.remover(p)}">
                    <f:ajax render=":formulario:groupPalestras"/>
                </h:commandButton>
            </h:column>
        </h:dataTable>
    </h:panelGroup>
</h:form>
</h:body>
</html>

```

No ManagedBean, estamos fazendo as chamadas para web service REST

```
package evento.mb;

import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;
import evento.modelo.Palestra;
import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class PalestraMB {
    private Palestra palestra = new Palestra();
    private List<Palestra> palestras = new ArrayList();
    private Client c = Client.create();
    private Gson gson = new Gson();

    public PalestraMB() {
        recarregar();
    }

    public void recarregar() {
        WebResource wr = c.resource(
            "http://localhost:8080/PalestraWSREST/recursos/palestra");
        String json = wr.get(String.class);
        palestras = gson.fromJson(json, new TypeToken<List<Palestra>>() {}.getType());
    }

    public String salvar() {
        WebResource wr = c.resource(
            "http://localhost:8080/PalestraWSREST/recursos/palestra");
        if(palestra.getId() == null) {
```

```
        wr.type("application/json").accept("application/json")
    )
        .post(gson.toJson(palestra));
    } else {
        wr.type("application/json").accept("application/json")
    )
        .put(gson.toJson(palestra));
    }
palestra = new Palestra();
recarregar();
return "index";
}

public void remover(Palestra p) {
    WebResource wr = c.resource(
        "http://localhost:8080/PalestrawsREST/recursos/palestra/"
        +p.getId());
    wr.delete();
    recarregar();
}

public void editar(Palestra p) {
    this.palestra = p;
}

public Palestra getPalestra() {
    return palestra;
}

public void setPalestra(Palestra palestra) {
    this.palestra = palestra;
}

public List<Palestra> getPalestras() {
    return palestras;
}

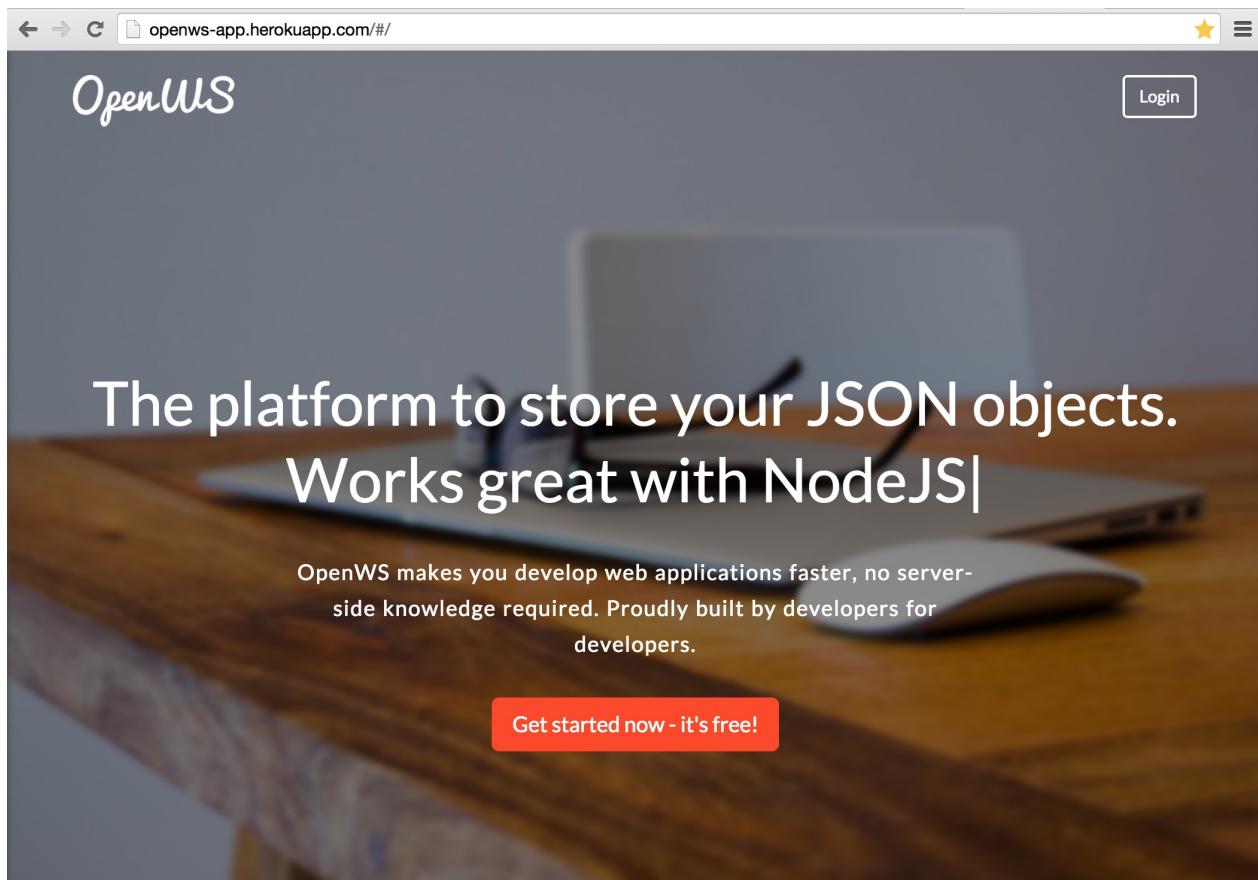
public void setPalestras(List<Palestra> palestras) {
    this.palestras = palestras;
}
```

```
}
```

Integração via REST com OpenWS

Nesse exemplo quero mostrar como podemos usar o OpenWS como repositório de objetos ao invés de termos que criar toda a parte dos web services REST dentro da aplicação.

Para quem não conhece, o OpenWS (<http://openws-app.herokuapp.com>) é uma plataforma para armazenar objetos JSON. Após se cadastrar gratuitamente temos acesso a um tutorial sobre como utilizá-lo e também acesso as coleções de objetos que criamos.



Algo bem transparente do OpenWS é que não precisamos criar previamente estruturas de objetos ou as coleções que queremos usar, isso é feito automaticamente pelo próprio OpenWS, e também utilizando os métodos HTTP (GET, POST, PUT e DELETE) para acessar os objetos.

Nesse exemplo vamos criar um CRUD de Abastecimentos (quando você abastece o carro com álcool ou gasolina), estou usando JSF + Bootstrap (<http://getbootstrap.com/>) para fazer a parte visual, a tela da aplicação vai ficar

assim:

The screenshot shows a web application titled "Abastecimentos" running on localhost:8080/OpenWSClient. The interface includes a form for entering new fuel purchases and a table listing existing ones.

Form Fields:

- Tipo combustivel: Álcool
- Valor litro: 0.0
- Quantidade: 0.0
- Data: dd/mm/aaaa
- Local: (empty)

Buttons:

- Novo (New)
- Salvar (Save) - highlighted in green

Table Headers:

Tipo combustivel	Valor litro	Quantidade	Data	Local		
------------------	-------------	------------	------	-------	--	--

Table Data:

Gasolina	3.069	39.72	09/04/2015	Extra Anchieta	<button>Editar</button>	<button>Remover</button>
Álcool	2.099	40.0	14/04/2015	Posto Ipiranga - Vergueiro	<button>Editar</button>	<button>Remover</button>
Gasolina	3.099	39.0	27/04/2015	Extra Anchieta	<button>Editar</button>	<button>Remover</button>

Para começar crie um novo projeto web, nesse exemplo chamei o projeto de **OpenWSCiente**.

Vamos criar uma classe que representa um **Abastecimento**:

```
package exemplo.modelo;

import java.util.Date;

public class Abastecimento {
    private String _id;
    private String tipoCombustivel;
    private double valorLitro;
    private double quantidade;
    private Date dataAbastecimento;
    private String local;

    public String getId() {
        return _id;
    }

    public void setId(String id) {
        this._id = id;
    }
}
```

```
}

public String getTipoCombustivel() {
    return tipoCombustivel;
}

public void setTipoCombustivel(String tipoCombustivel) {
    this.tipoCombustivel = tipoCombustivel;
}

public double getValorLitro() {
    return valorLitro;
}

public void setValorLitro(double valorLitro) {
    this.valorLitro = valorLitro;
}

public double getQuantidade() {
    return quantidade;
}

public void setQuantidade(double quantidade) {
    this.quantidade = quantidade;
}

public Date getDataAbastecimento() {
    return dataAbastecimento;
}

public void setDataAbastecimento(Date dataAbastecimento) {
    this.dataAbastecimento = dataAbastecimento;
}

public String getLocal() {
    return local;
}

public void setLocal(String local) {
    this.local = local;
```

```
    }
}
```

A seguir temos o código da página index.xhtml com o cadastro e tabela para apresentar os abastecimentos salvos.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:pt="http://xmlns.jcp.org/jsf/passthrough">
<h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <h:outputStylesheet library="css" name="bootstrap.min.css"/>
    <h:outputStylesheet library="css" name="bootstrap-theme.min.css"/>
    <title>Abastecimentos</title>
</h:head>
<h:body>
    <div style="width: 100%; text-align: center">
        <h1>Abastecimentos</h1>
    </div>
    <br/>
    <h:form id="formulario">
        <h:panelGroup id="dados">
            <div class="form-group">
                <div class="row">
                    <div class="col-md-2">
                        <h:outputText value="Tipo combustivel: " class="control-label"/>
                    </div>
                    <div class="col-md-3">
                        <h:selectOneMenu id="tipoCombustivel"
                            value="#{abastecimentoMB.abastecimento.tipoCombustivel}"
                            class="form-control">
```

```
<f:selectItem itemLabel="Álcool" itemValue="Álco  
ol"/>  
    <f:selectItem itemLabel="Gasolina" itemValue="Ga  
solina"/>  
        </h:selectOneMenu>  
    </div>  
</div>  
<div class="row">  
    <div class="col-md-2">  
        <h:outputText value="Valor litro: " class="control  
-label"/>  
    </div>  
    <div class="col-md-3">  
        <h:inputText id="valorLitro"  
            value="#{abastecimentoMB.abastecimento.valorLitr  
o}"  
            class="form-control"/>  
    </div>  
</div>  
<div class="row">  
    <div class="col-md-2">  
        <h:outputText value="Quantidade: " class="control-  
label"/>  
    </div>  
    <div class="col-md-3">  
        <h:inputText id="quantidade"  
            value="#{abastecimentoMB.abastecimento.quantidad  
e}"  
            class="form-control"/>  
    </div>  
</div>  
<div class="row">  
    <div class="col-md-2">  
        <h:outputText value="Data: " class="control-label"  
/>  
    </div>  
    <div class="col-md-3">  
        <h:inputText id="dataAbastecimento"  
            value="#{abastecimentoMB.abastecimento.dataAbast  
ecimento}"
```

```

        pt:placeholder="dd/mm/aaaa" class="form-control">

        <f:convertDateTime for="dataAbastecimento"
            timeZone="America/Sao_Paulo" pattern="dd/MM/yy
yy" />
        </h:inputText>
    </div>
</div>
<div class="row">
    <div class="col-md-2">
        <h:outputText value="Local " class="control-label">
    </div>
    <div class="col-md-9">
        <h:inputText id="local" value="#{abastecimentoMB.a
bastecimento.local}"
            class="form-control"/>
    </div>
    </div>
    <h:commandButton value="Novo" action="#{abastecimentoM
B.novo}"
        style="width: 80px;" class="btn btn-default"/>
    <h:commandButton value="Salvar" action="#{abasteciment
oMB.salvar}"
        style="width: 80px;" class="btn btn-success"/>
    </div>
</h:panelGroup>
<h:panelGroup id="groupAbastecimento">
    <div class="table-responsive">
        <h:dataTable value="#{abastecimentoMB.abastecimentos}"
var="a"
            width="100%" class="table table-hover">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Tipo combustivel"/>
                </f:facet>
                <h:outputText value="#{a.tipoCombustivel}"/>
            </h:column>
            <h:column>
                <f:facet name="header">

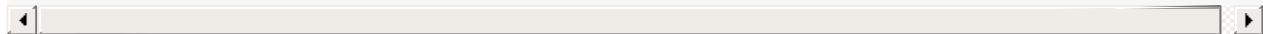
```

```

        <h:outputText value="Valor livro"/>
    </f:facet>
    <h:outputText value="#{a.valorLitro}"/>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Quantidade"/>
    </f:facet>
    <h:outputText value="#{a.quantidade}"/>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Data"/>
    </f:facet>
    <h:outputText value="#{a.dataAbastecimento}">
        <f:convertDateTime for="dataAbastecimento"
            timeZone="America/Sao_Paulo" pattern="dd/MM/yy
yy" />
        </h:outputText>
    </h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Local"/>
    </f:facet>
    <h:outputText value="#{a.local}"/>
</h:column>
<h:column>
    <h:commandButton value="Editar" action="#{abastecimentoMB.editar(a)}"
        style="width: 80px;" class="btn btn-primary">
        <f:ajax render=":formulario:dados"/>
    </h:commandButton>
</h:column>
<h:column>
    <h:commandButton value="Remover" action="#{abastecimentoMB.remover(a)}"
        style="width: 80px;" class="btn btn-danger">
        <f:ajax render=":formulario:groupAbastecimento"/>
    </h:commandButton>

```

```
</h:column>
</h:dataTable>
</div>
</h:panelGroup>
</h:form>
</h:body>
</html>
```



Observação: Os arquivos do bootstrap: bootstrap.min.css e bootstrap-theme.min.css devem ser colocados dentro de Páginas Web / resources / css.

Agora vamos implementar o ManagedBean:

```
package exemplo.mb;

import com.google.common.reflect.TypeToken;
import com.google.gson.Gson;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import exemplo.modelo.Abastecimento;
import java.io.Serializable;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class AbastecimentoMB implements Serializable {

    private static final long serialVersionUID = -83406950083467
67777L;
    private Abastecimento abastecimento = new Abastecimento();
    private List<Abastecimento> abastecimentos;
    private Client c = Client.create();
    private Gson gson = new Gson();

    private static final String URL = "https://openws.herokuapp.com/abastecimento";
```

```
//Coloque sua API Key aqui.
private static final String MINHA_CHAVE = "?apiKey=xxxxxxxxxxxx
xxxxx";

public AbastecimentoMB() {
    recarregar();
}

public void recarregar() {
    WebResource wr = c.resource(URL + MINHA_CHAVE);
    String json = wr.get(String.class);
    abastecimentos = gson.fromJson(json, new TypeToken<List<
Abastecimento>>() {
        .getType());
}

public String novo() {
    this.abastecimento = new Abastecimento();
    return "index";
}

public String salvar() {
    if (abastecimento.getId() == null) {
        WebResource wr = c.resource(URL + MINHA_CHAVE);
        wr.type("application/json").accept("application/json")
)
        .post(gson.toJson(abastecimento));
    } else {
        WebResource wr = c.resource(URL + "/" + abasteciment
o.getId()
        + MINHA_CHAVE);
        wr.type("application/json").accept("application/json"
)
        .put(gson.toJson(abastecimento));
    }
    abastecimento = new Abastecimento();
    recarregar();
    return "index";
}
```

```
public void remover(Abastecimento a) {
    WebResource wr = c.resource(URL + "/" + a.getId() + MINHA_CHAVE);
    wr.delete();
    recarregar();
}

public void editar(Abastecimento a) {
    System.out.println(a);
    this.abastecimento = a;
}

public Abastecimento getAbastecimento() {
    return abastecimento;
}

public void setAbastecimento(Abastecimento abastecimento) {
    this.abastecimento = abastecimento;
}

public List<Abastecimento> getAbastecimentos() {
    return abastecimentos;
}

public void setAbastecimentos(List<Abastecimento> abastecimentos) {
    this.abastecimentos = abastecimentos;
}
```

No ManagedBean informe qual a sua chave (API Key) obtida no OpenWS:

```
//Coloque sua API Key aqui.
private static final String MINHA_CHAVE = "?apiKey=xxxxxxxxxxxxxx"
;
```

Criei uma URL com o endereço:

```
private static final String URL = "https://openws.herokuapp.com/  
abastecimento";
```

Isso informa para o OpenWS criar uma coleção de objetos chamado `abastecimento` e dentro dela ele vai guardar os objetos.

Agora só testar a aplicação, veja que não precisamos criar o Web Service, o próprio OpenWS faz isso, precisamos apenas criar o cliente.

Java Naming and Directory Interface

O Java Naming and Directory Interface (JNDI) é uma API para acesso ao serviço de nomes e diretórios, utilizado para pedir ao servidor de aplicações web (Glassfish) um componente EJB.

Para criar uma conexão com o serviço de nomes e diretórios utilizamos a classe **java.naming.InitialContext**.

No exemplo a seguir, criamos um método **getProperties()** que configura alguns parâmetros informando de onde iremos utilizar o serviço de nomes e diretórios, depois criamos o método **buscaEJB()** que irá utilizar o **InicialContext** para pedir um objeto através de seu nome "br.exemplo.ejb.UsuarioRemote".

```
package br.exemplo.jndi;

import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * Classe utilizada para demonstrar como localizar
 * um objeto através do JNDI.
 */
public class ExemploJNDI {
    public static UsuarioRemote buscarEJB() throws Exception {
        try {
            InitialContext ctx = new InitialContext();
            UsuarioRemote usuarioEjb = (UsuarioRemote)
                ctx.lookup("br.exemplo.ejb.UsuarioRemote");

            return usuarioEjb;
        } catch (NamingException ex) {
            ex.printStackTrace();
            throw new Exception("Não encontrou o EJB");
        }
    }
}
```

Remote Method Invocation

Quando temos uma aplicação distribuída, onde cada parte do sistema está em locais (maquinas) diferentes, é necessário fazer a comunicação entre as partes, e na grande maioria é necessário trafegar informações, chamar remotamente o método de uma classe Java pela rede, enviar e receber objetos Java, etc.

Para fazermos essa comunicação entre sistemas pela rede, podemos utilizar o Remote Method Invocation (RMI) que implementa toda a camada de rede necessária para fazer uma chamada remota a um método.

Quando trabalhamos com EJB, toda parte de comunicação entre os componentes EJB é feito automaticamente pelo container EJB, utilizando RMI.

A aplicação cliente conhece apenas a interface do componente EJB, através do JNDI é solicitado ao servidor um objeto que representa o componente EJB, esse objeto criado na verdade é um objeto que simula o componente solicitado mas possui a lógica para fazer a chamada remota a outro objeto pela rede de computadores, é criado um objeto desse tipo no lado do servidor (chamado Skeleton) e outro no lado cliente (chamado Stub).

A chamada RMI funciona da seguinte forma:

- O objeto da aplicação cliente chama o método do componente a partir do objeto Stub;
- O Stub por sua vez irá chamar o método do objeto Skeleton que fica no lado do servidor;
- O Skeleton chama o objeto real do componente no servidor.

