

Usando a API de Persistência Java em EJBs

Em informática, o termo *Persistência de Dados* significa dizer que uma aplicação é capaz de executar o armazenamento, de modo confiável e coerente, das informações num sistema de armazenamento de dados, sistemas esses comumente chamados bancos de dados.

O modo mais tradicional de organizar dados em um banco é usando uma relação entre os diversos elementos de dados, organizados dentro de categorias (representadas sob a forma de tabelas), sem que haja redundância da informação.

O uso dos bancos de dados relacionais, no entanto, traz alguns inconvenientes aos desenvolvedores que usam linguagens para orientação a objetos, como é o caso de Java. As técnicas de modelagem orientada a objetos (OO) e relacional são bastante diferentes, e usá-las conjuntamente implica em enfatizar uma tecnologia em sacrifício da outra. Apesar de existirem bancos de dados OO, seu uso ainda é muito restrito, e não há sinais de que o mercado trocará os aplicativos de banco de dados atuais para bancos de dados orientados a objetos nos próximos anos.

Nas versões anteriores à versão 5 da plataforma corporativa do Java, todo o trabalho de persistência era feito usando comandos SQL, normalmente sendo executados através da API JDBC de Java, que é uma API de baixo nível, muitas vezes exigindo do desenvolvedor o conhecimento das “nuances” do banco de dados que será usado pela aplicação. Apesar de ser uma maneira bastante eficiente de acessar os dados em bancos de dados relacionais, a API JDBC faz com que o desenvolvedor deixe de usar os benefícios da programação orientada a objetos, pois o desenvolvedor precisa lidar com tabelas, registros e listas de resultados (resultsets), e não com objetos.

Para resolver esta situação, é necessário mapear a ligação entre os objetos e as entidades, e seus relacionamentos, no banco de dados. Isso é conhecido como mapeamento objeto-relacional (mapeamento O/R). Nesse mapeamento, classes e atributos do sistema são mapeados na forma de tabelas e colunas, respectivamente, e a persistência ocorre de modo transparente para a aplicação. Desse modo, objetos que existentes na memória podem ser gravados em bancos de dados, enquanto que os dados do banco são trazidos para a memória, e mantidos ali sob a forma de objetos.

Entretanto, esse trabalho, apesar de relativamente simples, consome muito tempo dos desenvolvedores, podendo levar várias horas para se criar uma camada completa de persistência para bancos com poucas tabelas. Para agilizar esse processo, a comunidade dos desenvolvedores Java criou alguns frameworks, como Hibernate, Struts, JSF, etc..., que não seguem nenhum padrão de implementação.

O exercício de hoje é montar manualmente um framework desse tipo, para entender como ele funciona. Assim, aproveitando o exemplo do Sistema de Vídeo Locadora, vamos criar uma

camada de persistência que pode ser usada por qualquer tipo de interface gráfica de usuário, que ficará alojada dentro de um componente do tipo EJB, ao invés de acessar diretamente uma classe de entidade (Entity Class).

Nesta demonstração, criaremos uma classe de entidade para o cliente, que usaremos para inserir dados relacionados aos clientes dentro do banco de dados, usando a API de persistência, além de atualizar e remover alguns dados.

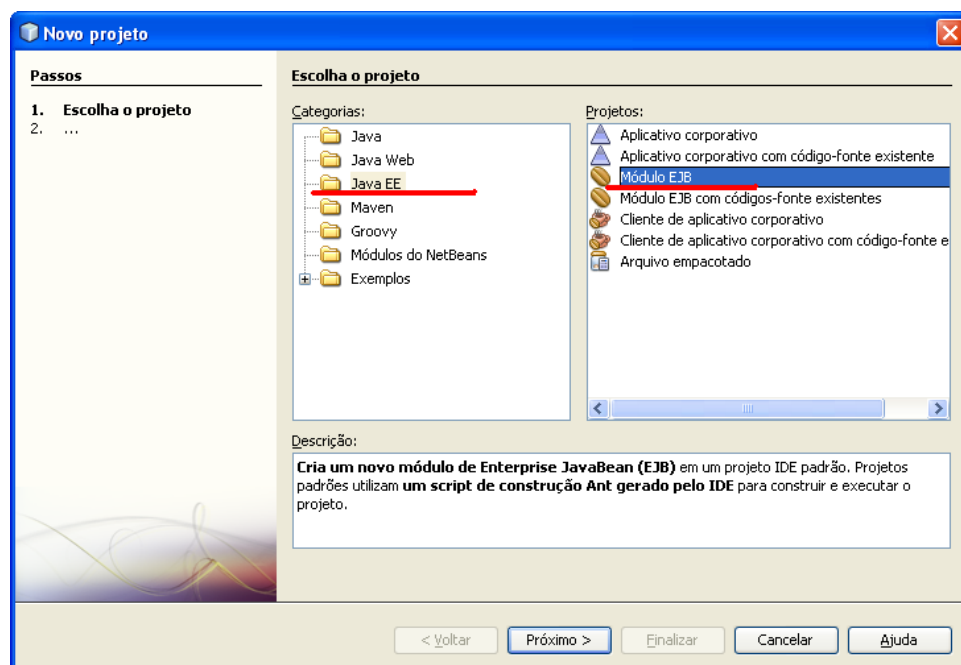
Criando a Entidade Cliente e Seus Relacionamentos

Num sistema de vídeo locadora, os atributos principais de um cliente são: nome, código do cliente, endereço, telefone e a data de cadastramento do cliente no sistema e a relação de vídeos que o cliente está alugando. Do mesmo modo, os principais atributos do vídeo são: código de cadastramento, número da cópia, título, gênero, e o cliente que está alugando uma determinada cópia.

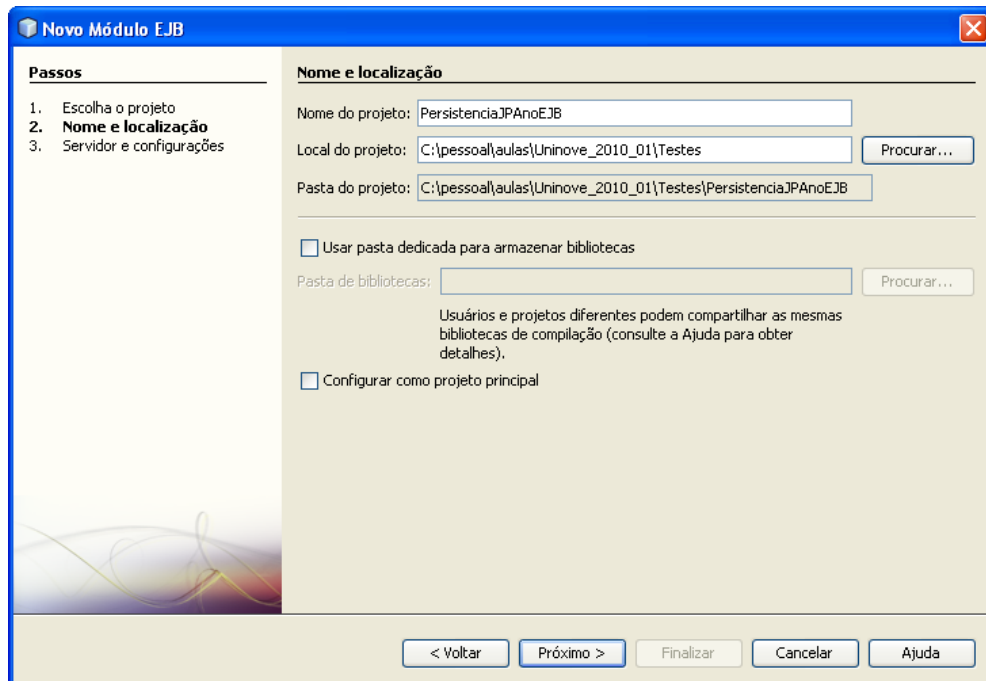
O primeiro passo é criar um projeto EJB, que terá o código da classe de entidade. Mas deve-se ter em mente que apesar da entidade ser uma classe, se essa entidade representa uma tabela no banco de dados, devem ser respeitadas as relações entre ela e as demais tabelas que se relacionam com ela, bem como as definições dos campos da tabela.

Para isso, o primeiro passo é criar um projeto EJB, que contém a classe de entidade. Uma vez que o NetBeans esteja aberto, siga os seguintes passos:

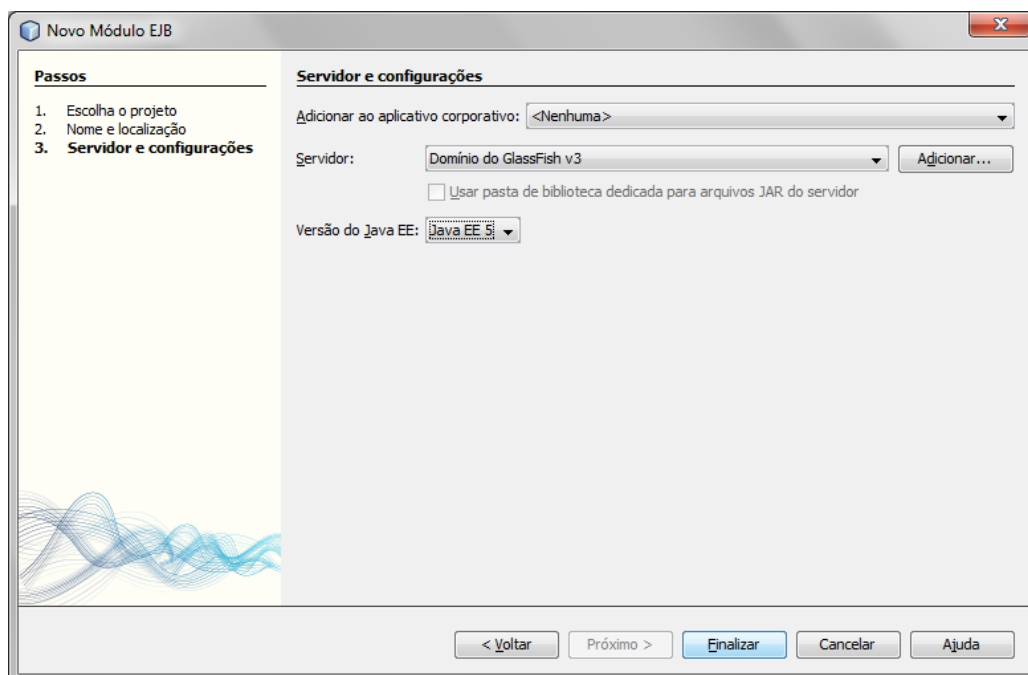
1. Clique em “Novo Projeto”, e selecione a categoria “Java EE”, e um projeto “Módulo EJB”, e depois clique em Próximo.



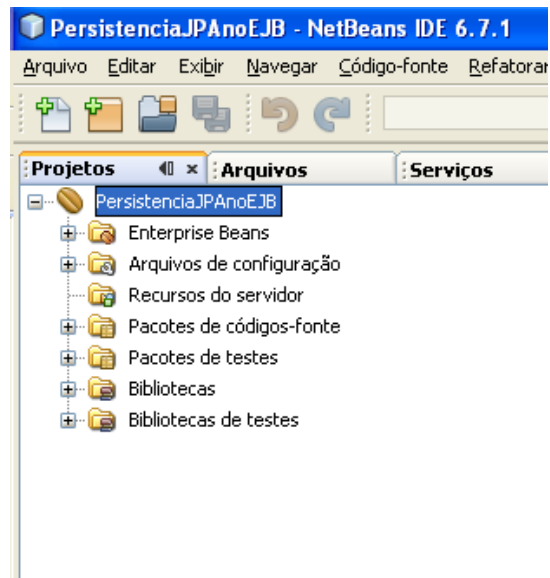
2. Atribua um nome para o projeto, eu usei o nome PersistenciaJPAnoEJB no exemplo. Atribua uma localização para o projeto e depois, clique em Próximo



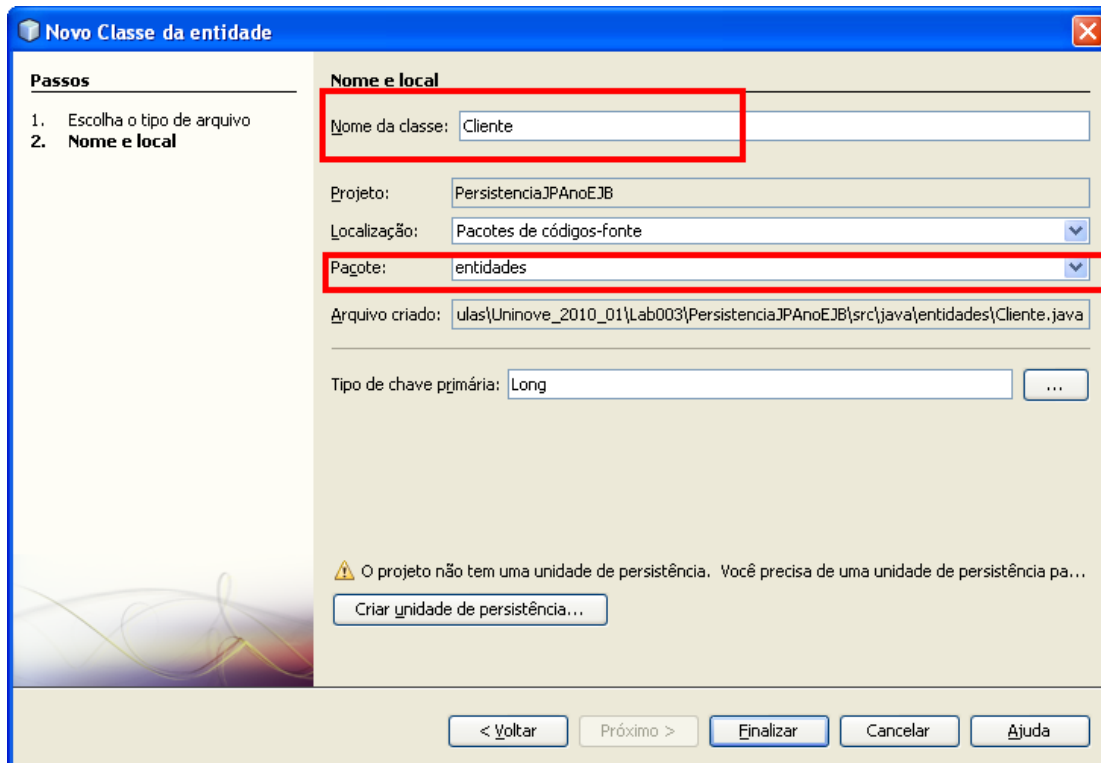
3. Especifique a versão do GlassFish como V2.1, V2 ou V3, conforme a opção que aparece no combo box, mas mantenha a versão do Java EE sempre como Java EE 5 ou Java EE 6, e clique em Finalizar



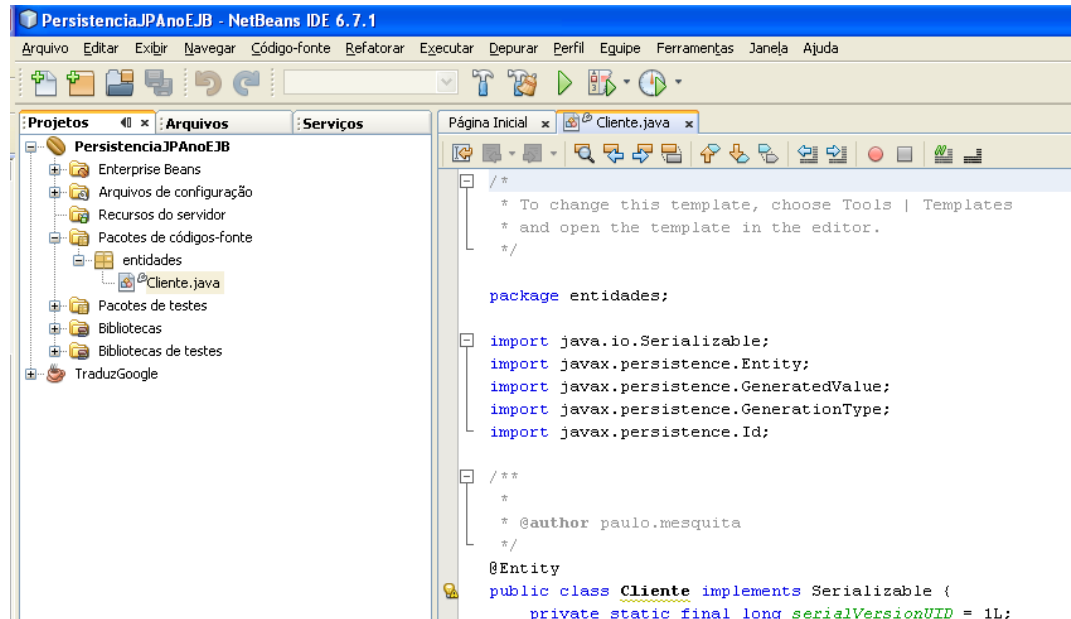
4. Depois de clicar em Finalizar, a área do projeto deve ficar com a seguinte aparência:



5. Até este momento, foi configurado o projeto do EJB. Agora, será criada a classe de entidade, e para isso, clique com o botão direito no mouse no nome do projeto e selecione a opção Novo -> Classe de Entidade:



6. No exemplo, foi atribuído o nome “Cliente” para nome da classe, e para o nome do pacote onde ela ficará armazenada, foi atribuído o nome “entidades”. Feito isso, basta clicar em Finalizar. A tela final deverá ficar com a seguinte aparência:



Nessa tela, pode-se notar o bloco de código a seguir, que é a importação das classes e anotações da API de Persistência do Java (disponível no pacote javax.persistence). Os imports chamam as classes disponíveis no JPA que serão usadas pela classe Cliente:

```
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;
```

Seguindo o código, há a definição da classe Cliente como sendo uma classe de entidade, definido pelo uso da anotação **@Entity** uma linha antes da definição do nome da classe, e que será mapeado no banco de dados para a tabela CLIENTE. Depois, há a implementação dessa classe como serializável, pois ela pode ser acessada a partir de aplicações remotas, que usam a bufferização de dados para transmissões pelas redes.

```
@Entity  
public class Cliente implements Serializable {
```

Depois, serão definidos os atributos da classe Cliente, que serão mapeados nas tabelas na forma de colunas da tabela CLIENTE. Automaticamente, quando a classe de entidade é criada, o seguinte pedaço de código aparece:

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

Essa parte do código contém as anotações @Id, indicando à API de Persistência que a o atributo

```
private Long id
```

se refere ao código do cliente, e

```
@GeneratedValue(strategy=GenerationType.AUTO)
```

indicando que esse valor será criado, no momento em que o cliente for adicionado ao banco. Caso eu mantenha este código, a coluna do código do cliente na tabela **CLIENTE** se chamará **ID**. Entretanto, para facilitar a leitura do modelo de dados, é melhor alterar o nome da coluna para um nome do tipo “**CD_CLIENTE**”, uma abreviação indicando que essa coluna é o código do cliente. Para isso, eu uso a anotação

```
@Column (name="CD_CLIENTE")
```

na linha acima da criação do atributo, para mapear esse atributo com a coluna **CD_CLIENTE** na tabela. Após fazer a alteração, o código ficará como o pedaço a seguir:

Agora, basta adicionar os atributos nome e endereço do cliente à classe. Para isso, depois da linha **private Long id**, adiciona-se os demais atributos do **Cliente**, para que a classe de entidade fique com o código semelhante ao que é apresentado a seguir:

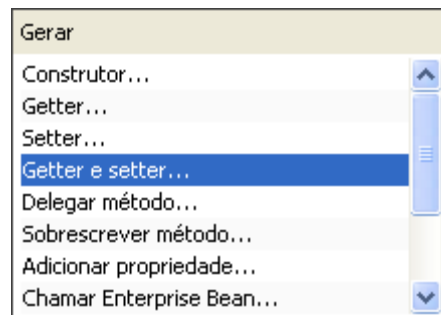
```

10  @Entity
11  public class Cliente implements Serializable {
12
13      private static final long serialVersionUID = 1L;
14      @Id
15      @GeneratedValue(strategy = GenerationType.AUTO)
16      private Long id;
17      private String nome;
18      private String endereco;
19      private Integer telefone;
20      private Integer CEP;
21      @Temporal(javax.persistence.TemporalType.DATE)
22      private java.util.Date dataCadastramento;
23
24      public Long getId() {
25          return id;
26      }
27
28      public void setId(Long id) {
29          this.id = id;
30      }
31
32      @Override
33      public int hashCode() {
34          int hash = 0;
35          hash += (id != null ? id.hashCode() : 0);
36          return hash;
37      }
38
39      @Override
40      public boolean equals(Object object) {

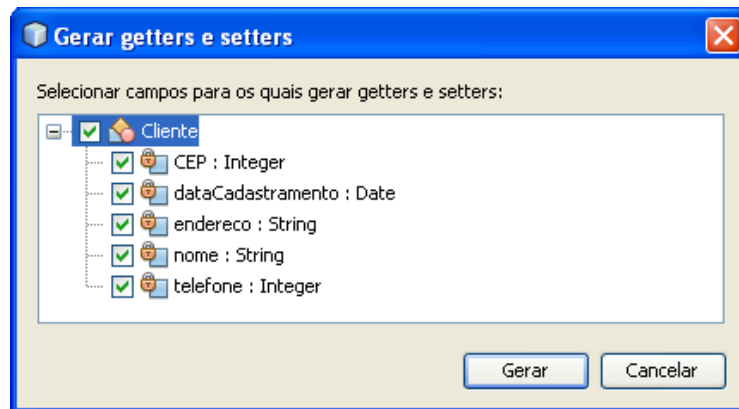
```

Pode-se notar a anotação **@Temporal** acima da definição do atributo “**private java.util.Date dataCadastramento;**”. Essa anotação relaciona o formato de uma data usado na classe, com o formato esperado pelo banco de dados, que nem sempre são iguais.

Depois, deve-se adicionar os métodos getters e setters relacionados aos atributos da classe **Cliente**. Para isso, na linha acima do método **public Long getId()**, clique com o botão direito do mouse, e selecione “**Inserir Código**”, e selecione a opção **Getter** e **Setter** no menu que se abre.



Na tela seguinte, indique que a criação dos métodos getter e setter para os demais atributos da classe Cliente, e clique em Gerar.



Depois de clicar no botão Gerar, será completada a criação da classe de entidade Cliente, cujo código completo deve ficar parecido com o código a seguir:

```
package entidades;

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;

@Entity
public class Cliente implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nome;
    private String endereco;
    private Integer telefone;
    private Integer CEP;
    @Temporal(javax.persistence.TemporalType.DATE)
    private java.util.Date dataCadastramento;

    public Integer getCEP() {
        return CEP;
    }

    public void setCEP(Integer CEP) {
```



```
        this.CEP = CEP;
    }

    public Date getDataCadastramento() {
        return dataCadastramento;
    }

    public void setDataCadastramento(Date dataCadastramento) {
        this.dataCadastramento = dataCadastramento;
    }

    public String getEndereco() {
        return endereco;
    }

    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }

    public String getNome() {
        return nome;
    }

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

    public Integer getTelefone() {
        return telefone;
    }

    public void setTelefone(Integer telefone) {
        this.telefone = telefone;
    }

    public Long getId() {
        return id;
    }

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

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
}
```

```

    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Cliente)) {
            return false;
        }
        Cliente other = (Cliente) object;
        if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }

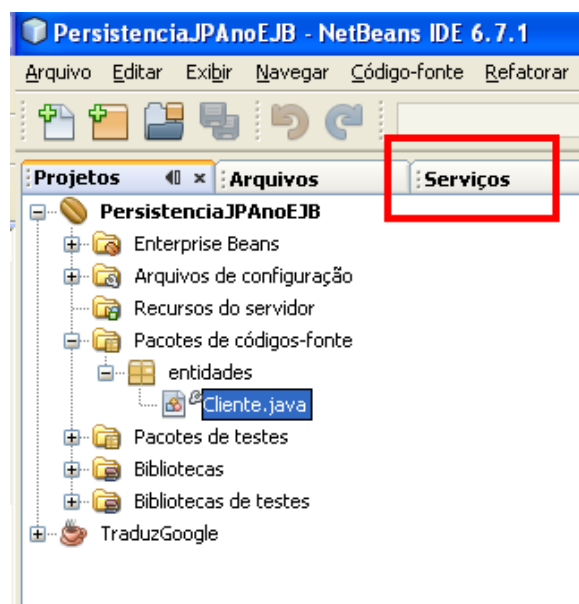
    @Override
    public String toString() {
        return "ents.Cliente[ id=" + id + " ]";
    }
}

```

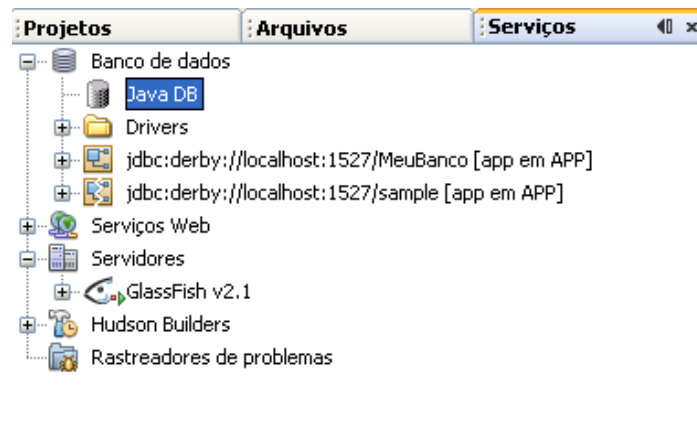
Criação do Banco de Dados

O passo seguinte para o nosso projeto é a criação de uma base de dados e o driver de conexão a essa base. Como nos exemplos anteriores, o banco de dados será o Java DB. Neste momento, o banco de dados será criado vazio, e as tabelas serão criadas durante a implantação do projeto no servidor de aplicações.

1. Para isso, selecione a aba Serviços no netbeans.

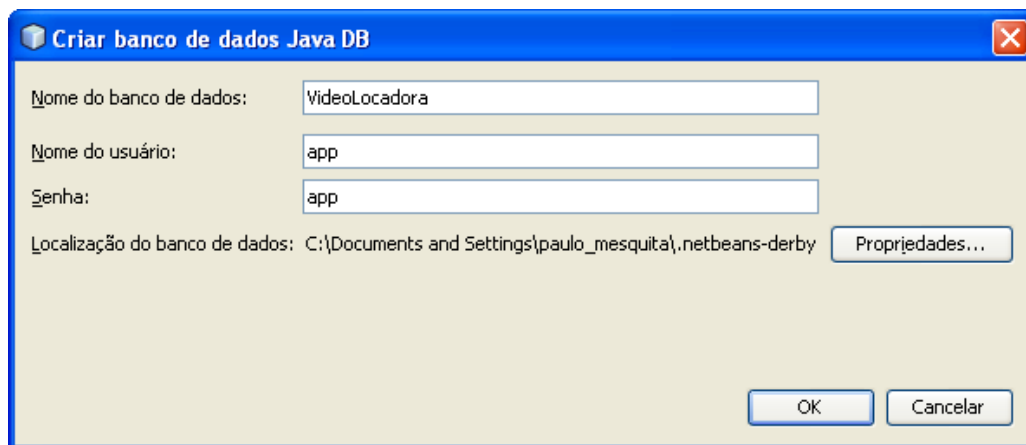


2. Isso fará com que seja aberta a aba de Serviços, com a seguinte aparência:

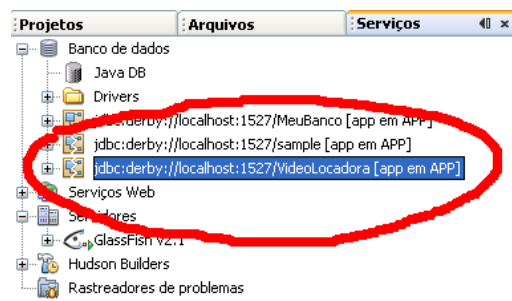


Para criar o banco de dados, execute os seguintes passos:

1. Clique como o botão direito do mouse em Java DB, e selecione Criar banco de dados;
2. Na janela Criar banco de dados Java DB, atribua um nome para o banco de dados, e defina o nome do usuário do banco e a senha como app. Depois disso, clique em OK



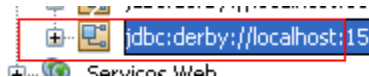
3. Após a criação do banco, a tela criará uma nova ponte para o banco de dados criado.



4. Neste momento, ainda não existe uma conexão ao bando, como atestado pela linha branca que quebra o ícone da conexão.



5. Para acessar o banco de dados, é necessário conectar-se a ele. Para isso, com o botão direito do mouse, clique na linha correspondente ao banco que foi criado, e selecione Conectar. Depois disso, a linha branca deve sumir, indicando que o banco de dados está conectado ao projeto.

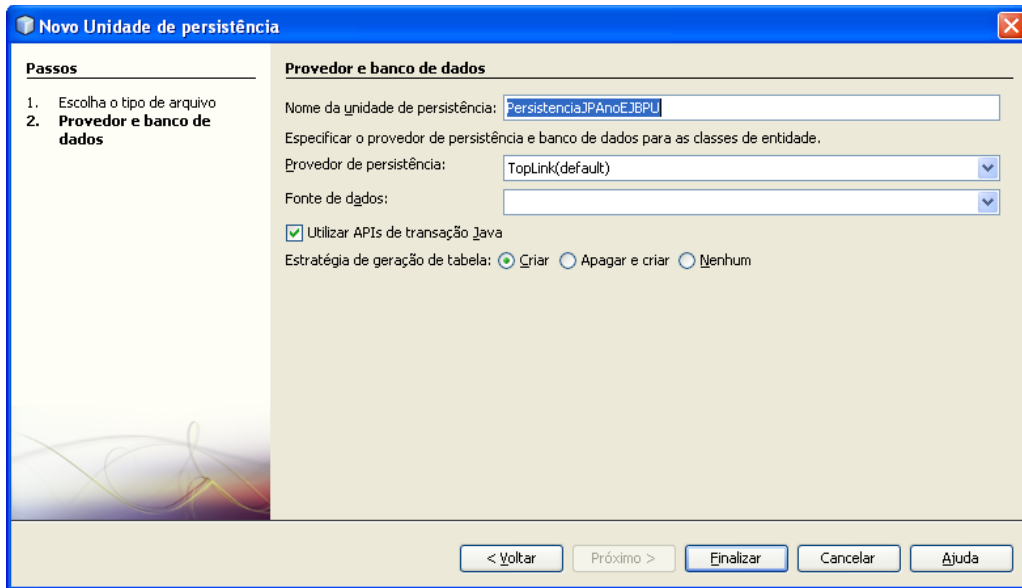


Neste momento, o banco de dados já pode ser usado. Entretanto, Para poder persistir as entidades do projeto, a JPA precisa de um “contexto de persistência” (persistence context), que faz a conexão entre as instâncias das entidades e o banco de dados.

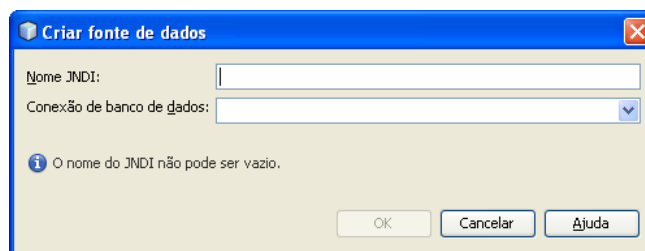
Criação da unidade de persistência

Quando a unidade de persistência é criada, as configurações definidas pelo desenvolvedor ficam armazenadas no arquivo “persistence.xml”. A unidade de persistência será referenciada através de um nome, e estará ligada ao provedor de persistência que a aplicação usa para gerenciar as instâncias de suas entidades. Também é possível especificar as fontes de dados e a estratégia para gerar as tabelas no banco de dados, quando a aplicação for instalada no servidor de aplicações. Para criar a unidade de persistência, execute os seguintes passos:

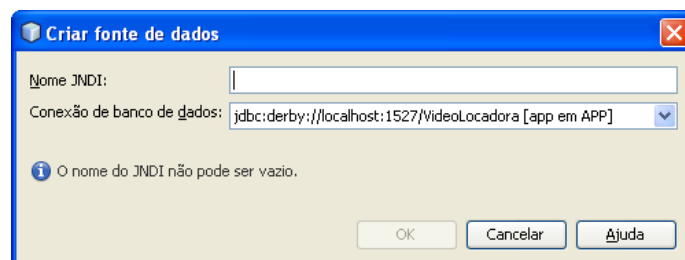
1. Clique com o botão direito do mouse no nome do projeto e escolha Novo > “Unidade de persistência” para abrir o assistente “Nova unidade de persistência”. Deverá aparecer uma tela semelhante à figura abaixo:



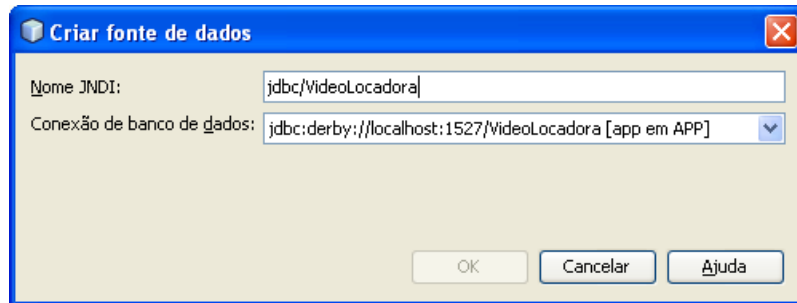
2. O valor do campo **“Nome da unidade de persistência”** não precisa ser alterado.
3. O campo **“Provedor de persistência”** deve ser preenchido com o valor TopLink, ou EclipseLink;
4. No campo **“Fonte de dados”**, selecione a opção **“Nova fonte de dados”**, costuma ser a última da lista. Ao selecionar essa opção, deverá aparecer uma tela semelhante à tela a seguir:



5. Nessa tela, ao clicar no campo **“Conexão de banco de dados”**, selecione o banco de dados criado anteriormente, e a tela ficará com a seguinte aparência:

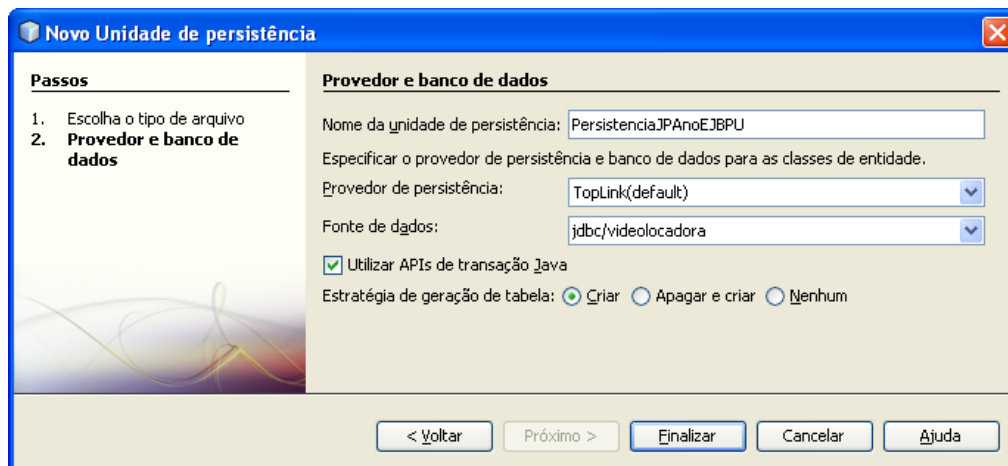


6. No campo **“Nome JNDI”**, crie um nome que servirá de link para o endereço do servidor de dados e para a própria base em si. Esse nome deve ser no formato **“nome do tipo de conexão/nome do banco”**, seguindo a mesma seqüência de maiúsculas e minúsculas, pois a linguagem Java é Case Sensitive. Assim, a aparência final dessa tela deve ser:



The screenshot shows a dialog box titled "Criar fonte de dados". It has two input fields: "Nome JNDI:" with the text "jdbc/VideoLocadora" and "Conexão de banco de dados:" with a dropdown menu showing "jdbc:derby://localhost:1527/VideoLocadora [app em APP]". At the bottom, there are three buttons: "OK", "Cancelar", and "Ajuda".

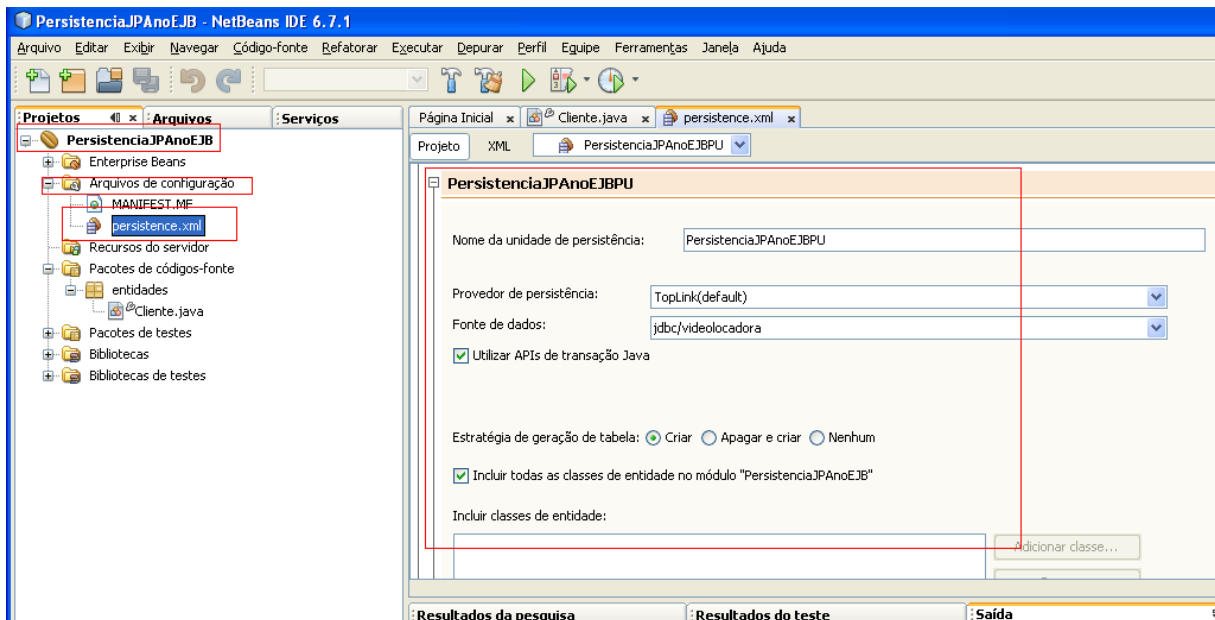
7. Uma vez definidos os valores dos campos, clique em Ok. A aplicação voltará à tela anterior, mostrando o nome da **“Fonte de dados”** criada pelo usuário. Nesta tela, caso apareça uma caixa de checagem com o nome **“Utilizar APIs de transação Java”**, mantenha a caixa selecionada, caso ela não apareça selecionada. Fazendo isso, a aparência final da tela será semelhante a:



The screenshot shows a dialog box titled "Novo Unidade de persistência". It has a "Passos" section on the left with two steps: "1. Escolha o tipo de arquivo" and "2. Provedor e banco de dados". The "Provedor e banco de dados" section on the right contains the following fields: "Nome da unidade de persistência:" with the text "PersistenciaJPAnoEJBPU", "Especificar o provedor de persistência e banco de dados para as classes de entidade.", "Provedor de persistência:" with a dropdown menu showing "TopLink(default)", "Fonte de dados:" with a dropdown menu showing "jdbc/videolocadora", a checked checkbox for "Utilizar APIs de transação Java", and "Estratégia de geração de tabela:" with three radio buttons: "Criar" (selected), "Apagar e criar", and "Nenhum". At the bottom, there are five buttons: "< Voltar", "Próximo >", "Finalizar", "Cancelar", and "Ajuda".

8. Se a tela estiver assim, clique em Finalizar.

Ao término desta operação, será criado um arquivo chamado **“persistence.xml”** dentro de **“Arquivos de Configuração”** do módulo **“PersistenciaJPAnoEJB”**, como na tela a seguir:

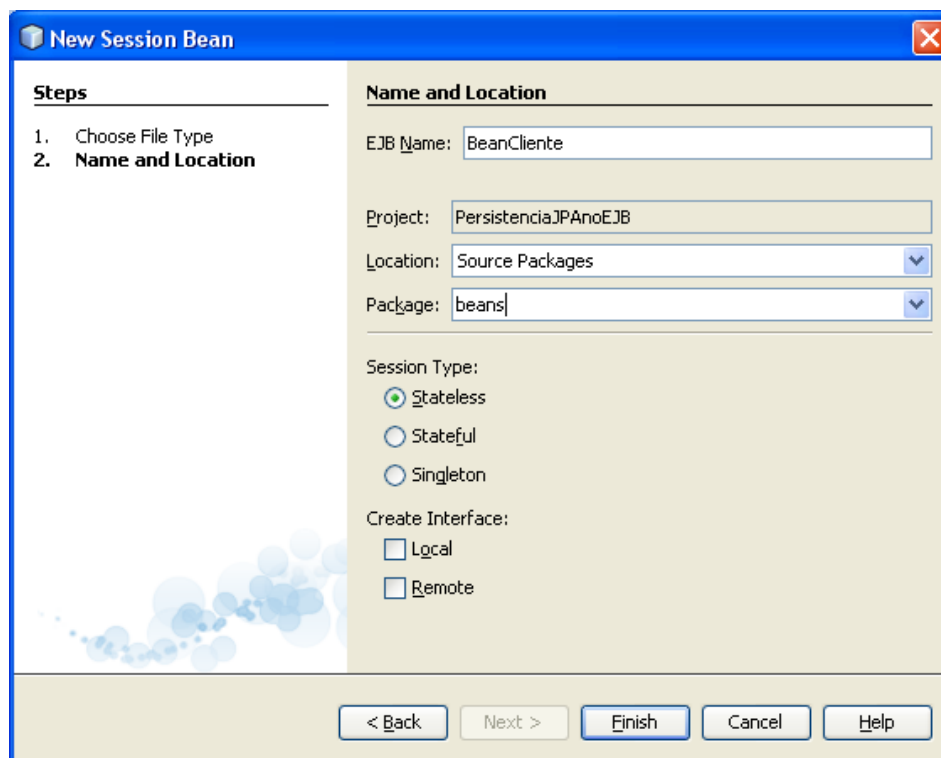


Automaticamente, após criar o arquivo **“persistence.xml”**, será aberta uma tela com as informações de configuração definidas pelo usuário. Esse arquivo armazena essas configurações do provedor de dados usado por uma determinada aplicação, como conexão com o banco de dados, gerenciamento das entidades, tipos de transação, e outros.

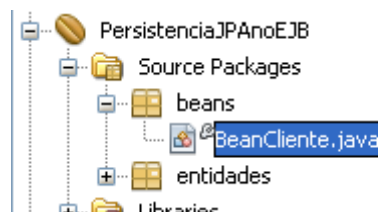
Usando a Entidade a partir de um Session Bean

Nesta aplicação, o Session Bean será o responsável por controlar o uso das classes de entidade para as tarefas de criação, atualização e remoção de clientes da base de dados, através da API de persistência, que é a verdadeira responsável por controlar os acessos ao banco de dados. Para criar o bean de sessão, siga os seguintes passos:

1. Com o botão direito do mouse no nome do projeto e selecione Novo -> Bean de Sessão. Na tela que abrir, atribuo o nome para o bean de sessão no campo “Nome EJB”, o pacote onde ele ficará armazenado no campo “Pacote”, e seleciono o campo “Tipo de Sessão” como “Sem estado (Stateless)”.



2. Definidos esses valores, clique em Finalizar, e a área de “Projetos” deverá ficar parecido com a seguinte tela:



Perceba que dentro de “Pacotes de códigos-fonte”, serão criados dois pacotes (área enquadada em vermelho), um chamado “beans”, que armazenará os beans de sessão e suas respectivas interfaces, que estão sendo criados, e outro pacote chamado “entidades”, onde ficarão armazenadas as classes de entidade criadas para a aplicação.

Quando usamos o JSF para criar os mapeamentos O/R a partir das classes de entidade, automaticamente são criados todos os componentes que gerenciam o acesso tanto ao servidor de dados, quanto às bases de dados propriamente ditas.

Mas, como agora essa responsabilidade está sendo atribuída ao bean de sessão, será necessário criar um “contexto de persistência” (persistence context), que conecta as instâncias das entidades e o banco de dados. Esse contexto é usado pelo bean para gerir as entidades que são usadas pela aplicação e pertencentes a esse contexto.

Para isso, é usada a anotação **@PersistenceContext**, que armazena o contexto de persistência a ser usado pelo bean. Em Java, isso é feito pelas seguintes linhas de código, logo abaixo da definição do nome da classe do bean:

```
@PersistenceContext(name="PersistenciaJPAnoEJBPU")
```

O contexto de persistência usa um “gerenciador de entidades” (entity manager), que é responsável por executar as operações básicas relacionadas à entidade (criação de dados, atualização, exclusão, localização, consulta dos dados, etc.). Esse gerenciador de entidades é criado do seguinte modo de modo similar à instanciação de classes:

EntityManager em;

Além das operações de banco de dados, o bean de sessão deve ser capaz de tratar as informações recebidas de quem chamou o bean. A primeira operação a ser implementada para o cliente é o cadastramento de seus dados na base de dados. Por isso, é criado um método que recebe os dados por quem chamou o bean, e os repassa para o gerenciador de entidades, usando os métodos “set” definidos na classe de entidade do cliente. Para esse fim, é criado um método chamado `cadastrarCliente`, similar ao seguinte pedaço de código:

```
public void cadastrarCliente(String nome, String endereco, Integer telefone,Integer CEP,  
                             Date dataCadastramento){  
  
    Cliente cli = new Cliente();  
    cli.setNome(nome);  
    cli.setEndereco(endereco);  
    cli.setTelefone(telefone);  
    cli.setCEP(CEP);  
    cli.setDataCadastramento(dataCadastramento);  
  
    // pedaço restante do código omitido
```

Juntando todas estas informações, e corrigindo as importações necessárias ao código, depois de completo, ele deve ficar igual a este:

```
package beans;

import entidades.Cliente;
import javax.ejb.Stateless;
import java.util.Date;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class BeanClienteBean implements BeanClienteRemote {

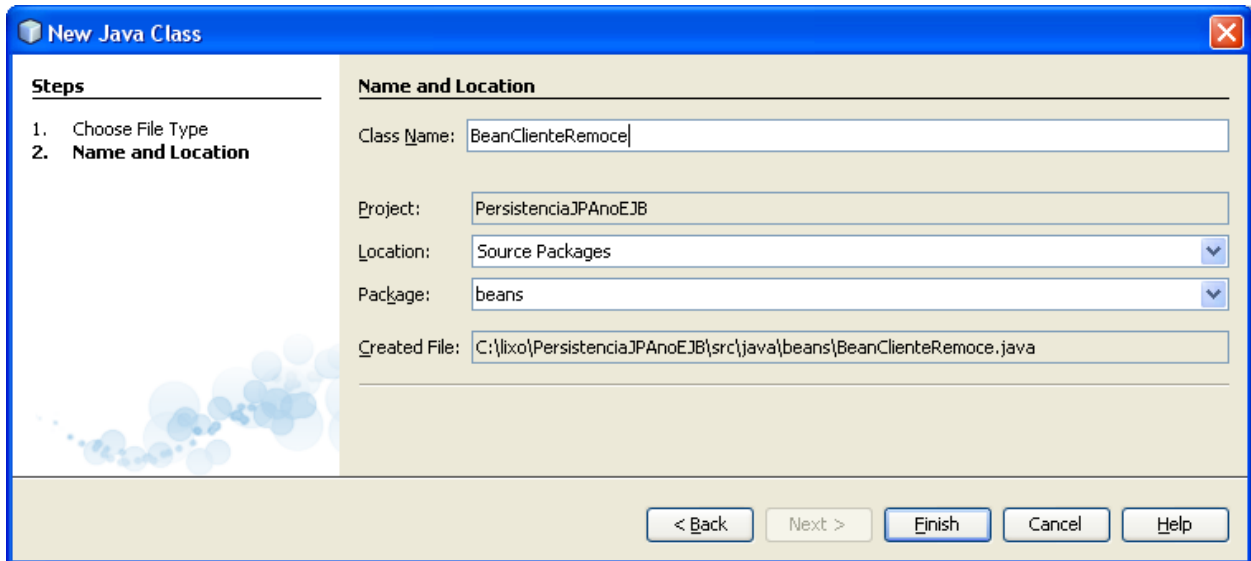
    @PersistenceContext(name="PersistenciaJPAnoEJBPU")
    EntityManager em;

    public void cadastrarCliente(String nome, String endereco,
                                  Integer telefone,Integer CEP,Date dataCadastramento){

        Cliente cli = new Cliente();
        cli.setNome(nome);
        cli.setEndereco(endereco);
        cli.setTelefone(telefone);
        cli.setCEP(CEP);
        cli.setDataCadastramento(dataCadastramento);
        em.persist(cli);
    }
}
```

Depois, deve ser criado o método `cadastrarCliente` na interface remota. Fazendo essa alteração, o código da classe remota deve ficar igual ao pedaço de código a seguir:

1. Para criar a interface remota, clique no nome do projeto, e selecione as opções Novo -> Classe Java, para abrir a tela a seguir:

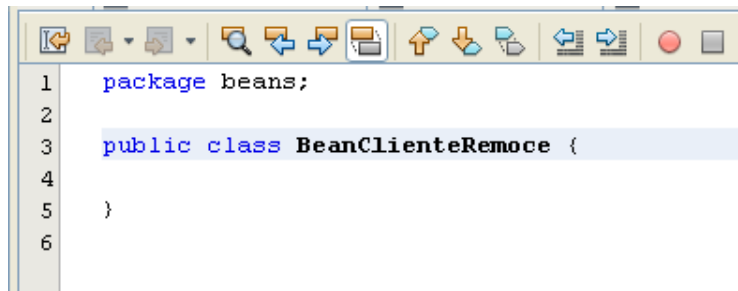


The image shows the 'New Java Class' dialog box in an IDE. It has a title bar 'New Java Class' with a close button. On the left, under 'Steps', step 2 'Name and Location' is selected. The main area is titled 'Name and Location' and contains the following fields:

- Class Name: BeanClienteRemoce
- Project: PersistenciaJPAnoEJB
- Location: Source Packages (dropdown menu)
- Package: beans (dropdown menu)
- Created File: C:\lixo\PersistenciaJPAnoEJB\src\java\beans\BeanClienteRemoce.java

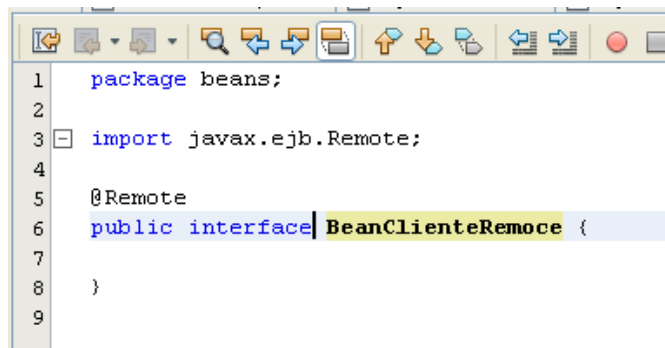
At the bottom, there are five buttons: '< Back', 'Next >', 'Finish', 'Cancel', and 'Help'.

2. Ao clicar em Finalizar, será aberto o código para a interface remota, como apresentado a seguir:



```
1 package beans;
2
3 public class BeanClienteRemoce {
4
5 }
6
```

3. Para criar a interface, altere o código da classe para que fique igual à listagem à seguir:



```
1 package beans;
2
3 import javax.ejb.Remote;
4
5 @Remote
6 public interface BeanClienteRemoce {
7
8 }
9
```

4. Para que a interface mapeia o método `cadastrarCliente` definido na classe do bean, o código deve ser alterado para a listagem a seguir:

```
1 package beans;
2
3 import java.util.Date;
4 import javax.ejb.Remote;
5
6 @Remote
7 public interface BeanClienteRemoce {
8
9     public void cadastrarCliente(String nome, String endereco, Integer telefone,
10                                Integer CEP, Date dataCadastramento);
11
12 }
13
```

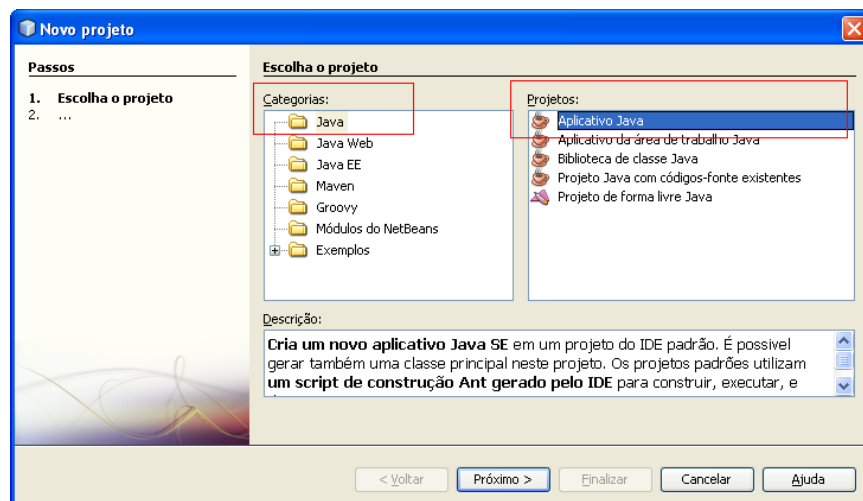
Neste momento, o EJB está pronto para ser compilado e instalado no servidor de aplicações para uma primeira bateria de testes. Para compilar o EJB, clique no nome do módulo EJB, e clique com o botão direito do mouse nas opções “Construir” ou “Limpar e Construir”. Se não houver erros de compilação, selecione novamente o nome do módulo EJB, e clique com o botão direito do mouse na opção “Implantar”. Ao fim da operação, o EJB estará instalado no servidor de aplicações, e pronto para ser testado.

Para acelerar o processo de teste, será usada uma aplicação Desktop, que executará o bean.

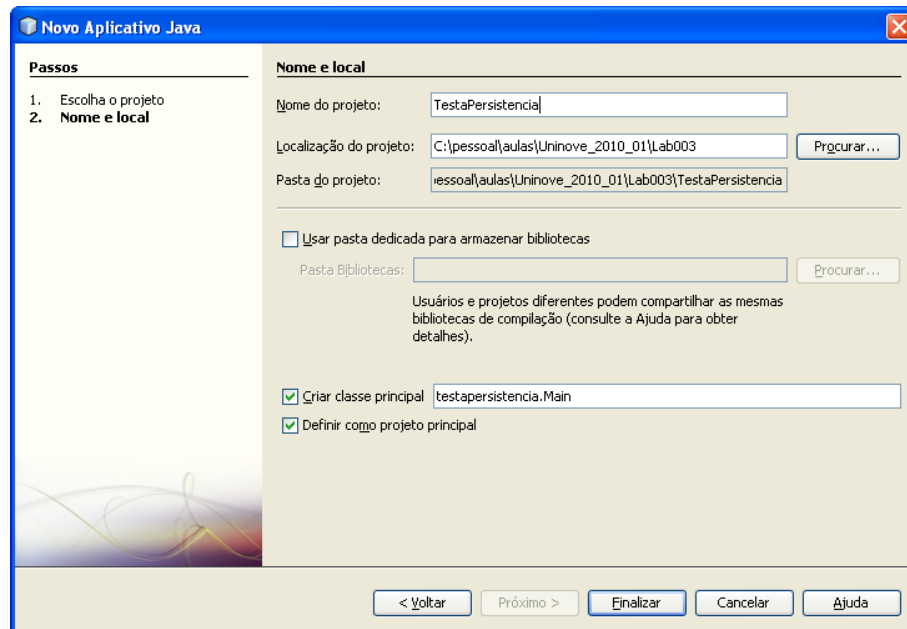
Testando o Bean Através de um Cliente Desktop

Para criar a aplicação desktop, clique no ícone “Novo Projeto, e siga os passos a seguir:

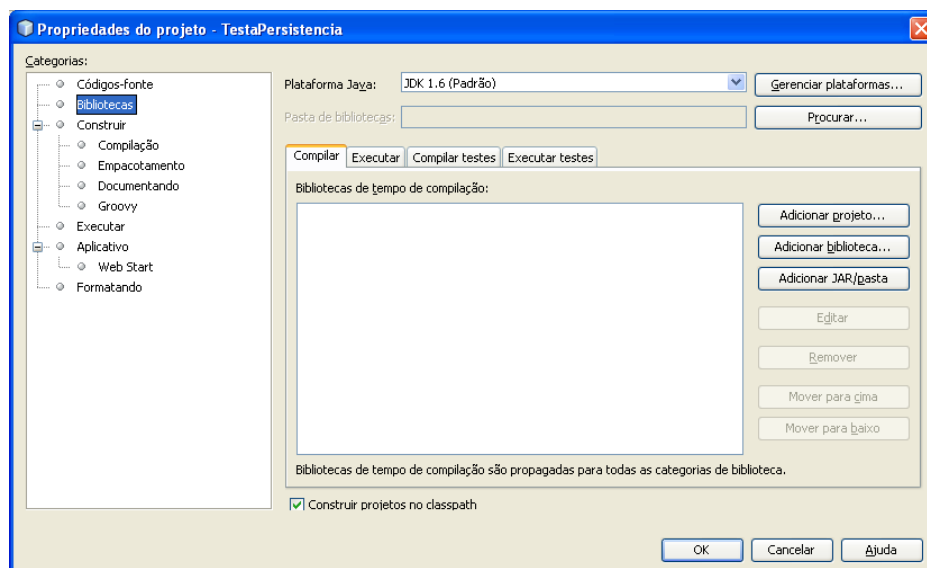
1. Na tela que abrir, selecione a “Categoria” Java, e em Projetos, selecione a opção “Aplicativo Java”, como mostrado na tela a seguir:



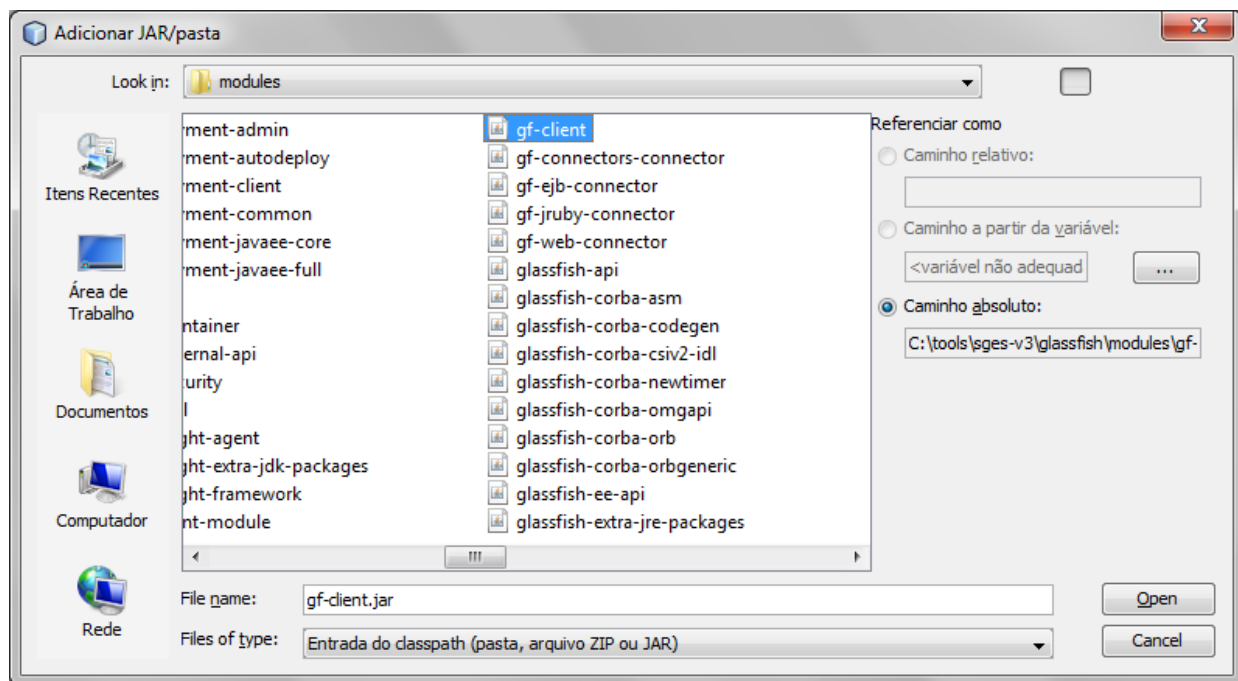
2. Clique em Próximo, e na próxima tela (semelhante à próxima tela), atribua um nome para a aplicação, e clique em Finalizar.



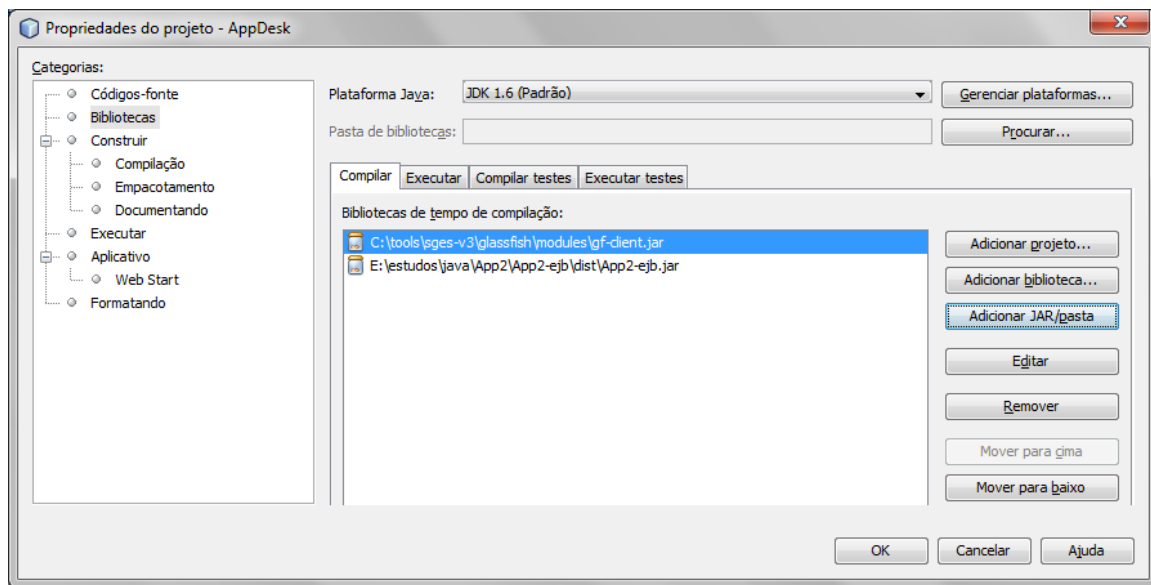
3. Para poder desenvolver um aplicativo que execute o bean, é necessário adicionar algumas bibliotecas Java (arquivos jar) ao projeto stand alone para que ele possa fazer o endereçamento ao servidor de aplicações, bem como o arquivo jar relativo ao bean que acabou de ser implementado. Para adicionar o bean às bibliotecas do projeto dektop, clique com o botão direito do mouse sobre o nome do projeto, e selecione propriedades, e será aberta uma tela semelhante à próxima:



4. Para adicionar o jar relativo ao bean, clique em “Adicionar JAR\Pasta”, e vá para o diretório dist, localizado dentro do diretório do projeto do bean. Selecione o nome o bean, e clique no botão “Open” ou “Abrir” para adicioná-lo ao projeto da aplicação desktop.
5. Depois, é necessário adicionar as libs do GlassFish que permitem a conexão remota ao servidor de aplicações. Para isso, clique no botão “Adicionar JAR\Pasta”, e vá até o diretório lib que se encontra dentro do diretório de instalação do servidor de aplicações. No caso do meu ambiente de desenvolvimento, elas estão em: “C:\tools\sges-v3\glassfish\modules”.
6. A biblioteca selecionada deve ser:



7. Ao término, a tela final deve ficar com a seguinte aparência:



8. O próximo passo é a codificação da aplicação desktop, que será iniciada a partir do método “main()”, que deverá conter um código igual à próxima listagem:

```
package testapersistencia;
import beans.BeanClienteRemote;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Date;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Main {

public static void main(String[] args) {

try {
    Properties props = new Properties();
    props.load(new FileInputStream("jndi.properties"));
    InitialContext ctx = new InitialContext(props);
    BeanClienteRemote testeEJB = (BeanClienteRemote) ctx.lookup("beans.BeanClienteRemote");
    testeEJB.cadastrarCliente("Anselmo", "R. dos Arapanés, 135", 8675389, 4521000, new
    Date(System.currentTimeMillis()));
    System.out.println("Cadastrou Anselmo");
    testeEJB.cadastrarCliente("Duarte", "R. dos Apinajés, 35", 887679, 4590700, new
    Date(System.currentTimeMillis()));
    System.out.println("Cadastrou Duarte");
    testeEJB.cadastrarCliente("Maria", "R. da Fofoca, 536", 93848592, 4520900, new
    Date(System.currentTimeMillis()));
    System.out.println("Cadastrou Maria");
    testeEJB.cadastrarCliente("Roberta", "R. do Barulho, 333", 1111111, 9888800, new
    Date(System.currentTimeMillis()));
    System.out.println("Cadastrou Roberta");
    testeEJB.cadastrarCliente("Marcos", "R. dos ?????, 123", 8111389, 4989800, new
    Date(System.currentTimeMillis()));
    System.out.println("Cadastrou Marcos");
}
```

```

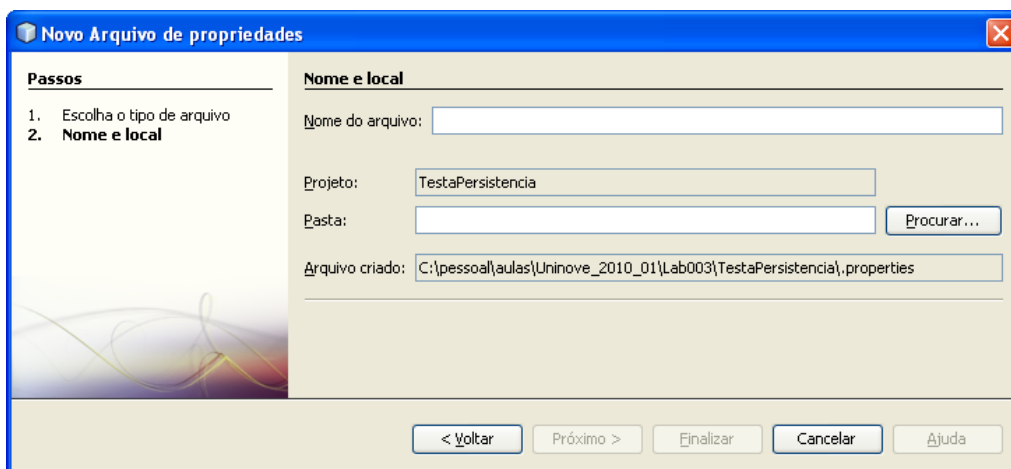
        testeEJB.cadastrarCliente("Aurélia", "R.    Aurélia,    S/N",    8643219,    4512000,    new
        Date(System.currentTimeMillis()));
        System.out.println("Cadastrou Anselmo");
    } catch (NamingException nex) {
        nex.printStackTrace();
    } catch (FileNotFoundException fnfex) {
        fnfex.printStackTrace();
    } catch (IOException ioex) {
        ioex.printStackTrace();
    }
}
}
}

```

Para que a aplicação desktop encontre o bean no servidor de aplicações, ele precisa de um arquivo de mapeamento, que nesta aplicação será nomeado como “jndi.properties”. Ele é o responsável por indicar para a aplicação desktop onde está o arquivo EJB que deve ser usado, e como ele pode ser acessado.

Para criar esse arquivo no projeto, siga os próximos passos:

8. Clique com o botão direito do mouse no nome do projeto, e selecione a opção Novo -> Arquivo de propriedades, para que apareça a seguinte tela:



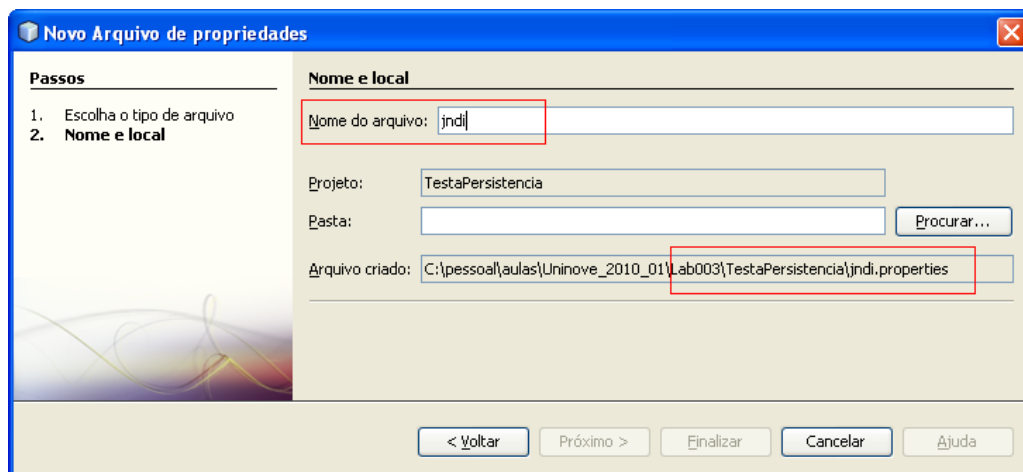
9. No campo “Nome do arquivo” atribua um nome para o arquivo de propriedades. Não esqueça que o nome do arquivo de propriedades deve ser o mesmo nome definido na seguinte linha de código, que no caso, é “jndi.properties”:

```

props.load(new FileInputStream("jndi.properties"));

```


10. Depois que o nome do arquivo for atribuído, a tela ficará parecida com esta:



11. Perceba nos enquadramentos em vermelho que não é necessário incluir a extensão “properties” ao nome do arquivo, a ferramenta já faz isso automaticamente. Clique no botão “Finalizar” e escreva o seguinte conteúdo no arquivo “jndi.properties”:

```
java.naming.factory.initial = com.sun.enterprise.naming.SerialInitContextFactory
java.naming.factory.url.pkgs = com.sun.enterprise.naming
java.naming.factory.state = com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl
org.omg.CORBA.ORBInitialHost = localhost
org.omg.CORBA.ORBInitialPort = 3700
```

OBSERVAÇÃO: Esse arquivo sempre é armazenado na pasta raiz do projeto.

Isso finaliza a aplicação desktop. Para testá-la, mande compilar o projeto e execute-o. A saída deve ser semelhante à tela a seguir:

