

**Persistência Java – Hibernate e
JPA
On-Line**

Todos os direitos reservados para Alfamídia Prow

AVISO DE RESPONSABILIDADE

As informações contidas neste material de treinamento são distribuídas “NO ESTADO EM QUE SE ENCONTRAM”, sem qualquer garantia, expressa ou implícita. Embora todas as precauções tenham sido tomadas na preparação deste material, a Alfamídia Prow não tem qualquer responsabilidade sobre qualquer pessoa ou entidade com respeito à responsabilidade, perda ou danos causados, ou alegadamente causados, direta ou indiretamente, pelas instruções contidas neste material ou pelo software de computador e produtos de hardware aqui descritos.

03/2012 – Versão 1.0.73

Alfamídia Prow
<http://www.alfamidia.com.br>

A Alfamídia dá Boas Vindas aos seus clientes e deseja um excelente treinamento.

Benefícios ao aluno

- Suporte pós-treinamento via e-mail (3 consultas por 90 dias após o término do curso) para tirar dúvidas do conteúdo ministrado em aula, através do e-mail matricula@alfamidia.com.br ;
- Acesso a extranet www.alunoalfamidia.com.br para verificação de agenda e pontos do **PROGRAMA FIDELIDADE**;
- Convênio com o estacionamento do prédio (desconto);
- Avaliações de acompanhamento e final de curso (em cada módulo) durante todo o treinamento, garantindo a qualidade do curso.

Observações Importantes

- É obrigatório que sejam salvos todos os trabalhos efetuados durante a aula, no servidor indicado pelo instrutor.
- Não é permitido entrar em sala de aula com alimentos ou bebidas de qualquer espécie ou fumar nas dependências da Alfamídia;
- Não é permitida a instalação de outros Hardwares ou Softwares que não sejam os utilizados em treinamento;
- O preenchimento da avaliação final de curso/módulo é condição obrigatória para que o aluno possa acionar a garantia do curso, conforme previsto na ficha de matrícula;
- Somente será fornecido certificado ao aluno com assiduidade igual ou superior a 75% do treinamento;
- Qualquer necessidade de alteração na agenda ou cursos contratados inicialmente, consulte os Termos de Contratação na Ficha de Matrícula;
- Contatos com a Alfamídia podem ser feitos através dos e-mails:

matricula@alfamidia.com.br – dúvidas após contratação

info@alfamidia.com.br – novas contratações

Sumário

Capítulo 1 – Java Persistence API	6
Mapeamento Objeto/Relacional.....	6
Antes da JPA	7
Enterprise JavaBeans.....	7
Persistência Leve.....	8
EJB 3	9
JPA.....	10
Os Provedores JPA.....	11
Capítulo 2 – Preparando Seu Ambiente	12
Fazendo o Download do Hibernate	12
Adicionando o Hibernate ao Seu Projeto	14
Capítulo 3 – Entidades	19
A Classe de Bean	20
Campos Persistentes	21
Propriedades Persistentes	22
Anotações Básicas	23
A Anotação @Entity	23
A Anotação @Table.....	23
A Anotação @Id	24
A Anotação @GeneratedValue.....	24
A Anotação @Column	25
A Anotação @Temporal.....	26
A Anotação @Lob	27
A Anotação @Embedded	27
A Anotação @Embeddable	27
A Anotação @Transient	27
Anotando Chaves Primárias Compostas	28
A Anotação @IdClass	28
A Anotação @EmbeddedId.....	29
Capítulo 4 – Unidade de Persistência	31
Criando Uma Unidade de Persistência para Uma Aplicação Java SE	31
Criando Uma Unidade de Persistência para Uma Aplicação Java EE	38
Capítulo 5 – Gerenciando Entidades com o EntityManager.....	45

Entidades Gerenciadas e Não-Gerenciadas	46
O Ciclo de Vida de uma Entidade	46
Classes Utilizadas no Gerenciamento de Entidades	46
Obtendo o EntityManager	47
Operações Básicas com Dados	47
Criando Uma Entidade	48
Alterando uma Entidade	49
Excluindo uma Entidade.....	50
Recuperando uma Entidade	51
Capítulo 6 – Mapeando Relacionamentos.....	53
Relacionamentos Um para Um	53
A Anotação @OneToOne.....	55
A Anotação @JoinColumn	56
Relacionamentos Um para Muitos e Muitos para Um	56
A Anotação @OneToMany	58
A Anotação @ManyToOne	58
Relacionamentos Muitos para Muitos	59
A Anotação @ManyToMany	61
A Anotação @JoinTable	61
Um Exemplo Completo.....	62
Capítulo 7 – Fazendo Consultas	72
A Interface Query	72
Usando Queries	73
Utilizando Parâmetros	75
Parâmetros Nomeados	75
Parâmetros Posicionais	76
Parâmetros do Tipo Data	76
Queries Nomeadas	77
A Anotação @NamedQuery	77
A Anotação @NamedQueries.....	78

Capítulo 1 – Java Persistence API

Persistência é um dos principais conceitos no desenvolvimento de aplicações: um sistema informatizado é de pouca valia se não for capaz de preservar os seus dados quando for desligado.

Quando falamos de persistência em Java estamos provavelmente falando sobre armazenar dados em bancos de dados relacionais utilizando SQL.

O Modelo Relacional é um modelo de banco de dados que se baseia no princípio de que todos os dados estão guardados em tabelas (ou, matematicamente falando, relações). Toda sua definição teórica é baseada na lógica de predicados e na teoria dos conjuntos. O conceito foi criado por Edgar Frank Codd em 1969, tendo sido descrito no artigo "Relational Model of Data for Large Shared Data Banks".

Tecnologias como o JDBC, por exemplo, oferecem à linguagem Java a capacidade de persistir dados em um banco relacional.

Mas para aplicações de maior porte a tecnologia JDBC isoladamente não é suficiente, pois deixa muitas tarefas a cargo do programador:

- Pool de conexões;
- Transações;
- Concorrência;
- Segurança;
- etc ...

Mapeamento Objeto/Relacional

Em essência, mapeamento objeto/relacional (no inglês ORM, Object/Relational Mapping) é a persistência automática e transparente de objetos em uma aplicação Java em tabelas em um banco de dados relacional. Hoje em dia esta persistência normalmente é conseguida por meio da utilização de metadados que descrevem o mapeamento entre os objetos e o banco de dados, mas nos primórdios da tecnologia Java este mapeamento era feito quase manualmente, por meio da geração programática de queries SQL para a persistência de objetos.

Mas o mapeamento objeto/relacional também precisa fazer o reverso: ler dados do banco de dados e criar os objetos que os representam (diz-se, então, que o mapeamento objeto/relacional é um processo reversível, capaz de transformar objetos em dados relacionais e dados relacionais em objetos).

Obviamente este mapeamento implica em certos custos em termos de performance, mas se o mapeamento for implementado em uma camada de middleware há uma infinidade de oportunidades de otimização que não estariam disponíveis se o mapeamento fosse feito programaticamente em uma camada específica.

A criação dos metadados de mapeamento também implica em uma carga de trabalho adicional para os programadores, mas esta carga de trabalho normalmente é igual ou menor do que aquela que seria necessária para criar e manter uma solução utilizando apenas JDBC.

As bibliotecas de mapeamento objeto/relacional normalmente consistem dos seguintes componentes:

- Uma API para executar operações básicas como criar, editar e excluir objetos de classes persistidas;
- Uma API para executar buscar por classes persistidas, de maneira semelhante ao que o SQL faz para dados relacionais;
- Uma maneira de especificar os metadados de mapeamento;
- Uma maneira de interagir com objetos transacionais para verificar se eles estão sincronizados com o banco de dados, recuperação de associações e funções de otimização.

Este é exatamente o objetivo da JPA, objeto de nosso estudo. Mas antes de mergulharmos nas maravilhas que a nova tecnologia oferece vamos dar uma olhada na evolução de tecnologias que levaram à ela.

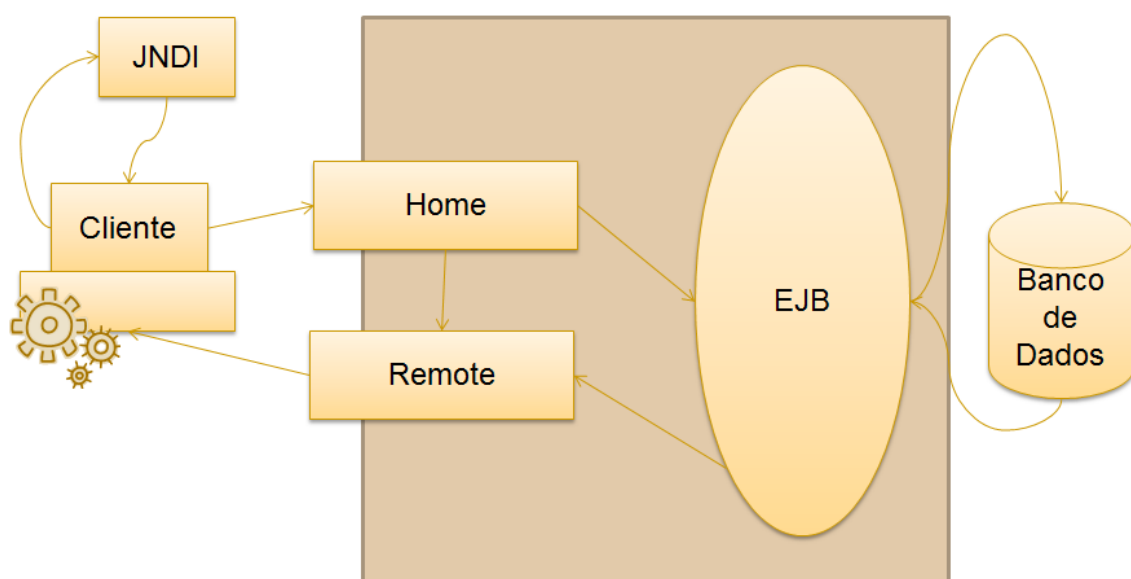
Antes da JPA

Enterprise JavaBeans

A primeira tentativa de resolver todos estes problemas surgiu em 1997, quando a IBM começou a desenvolver o que seria mais tarde conhecido como a Especificação EJB (de Enterprise JavaBeans).

A Sun adotou a ideia, e em 1999 lançou uma nova versão da especificação, a EJB 1.1.

A especificação acrescentou um componente importante às arquiteturas, o Container EJB. É ele quem desempenha aquelas tarefas repetitivas e complexas de que toda aplicação precisa.



Como vimos, a especificação EJB oferece uma série de recursos, mas o que mais nos interessa neste curso são os Entity Beans.

Um Entity Bean (Bean de Entidade) é um tipo de Enterprise JavaBean (EJB) que representa dados persistentes mantidos em um banco de dados.

Um bean de entidade pode gerenciar sua própria persistência (Bean de Persistência Gerenciada, ou simplesmente Bean Gerenciado) ou pode delegar esta função ao container (persistência gerenciada pelo container).

Por ter players grandes por trás de si, a especificação EJB rapidamente ganhou aceitação. Mas as críticas também não demoraram muito a surgir:

- A API era considerada desnecessariamente complexa;
- Existia a necessidade de se criar muito código repetitivo;
- Existiam gargalos importantes de performance decorrentes da arquitetura baseada em objetos distribuídos, mesmo que a maior parte das aplicações não necessitassem desta arquitetura.

Persistência Leve

O surgimento da especificação EJB 2.0 e seus beans locais não foi suficiente para dirimir as críticas àquela arquitetura, e o mercado começou a buscar soluções menos onerosas.

Nesta época turbulenta surgiram arquiteturas leves como o Hibernate e o Spring. A própria Sun criou uma nova API para persistência completamente separada da EJB, a Java Data Objects (JDO).

Hibernate

Hibernate é uma biblioteca de mapeamento objeto/relacional criada em 2001 por Gavin King como uma alternativa para os entity beans.

Hibernate busca solucionar os problemas do mapeamento objeto-relacional ao substituir o acesso direto ao banco de dados por funções de mais alto nível através do mapeamento transparente de classes Java para o banco de dados (e dos tipos de dados SQL para tipos Java).

Pode-se mapear classes Java para tabelas do banco de dados por meio de um arquivo de configuração XML ou, mais recentemente, por meio de annotations.

Existem ferramentas para criar relacionamentos de um para um, um para muitos e muitos para muitos. Além de gerenciar a associação entre objetos, a biblioteca ainda é capaz de criar associações reflexivas, onde um objeto tem uma relação de um para muitos objetos do seu próprio tipo.

Hibernate provê persistência transparente para objetos Java comuns (“Plain Old Java Objects”, POJOs). A única regra que eles devem seguir é oferecer um construtor sem argumentos, não necessariamente público.

Objetos relacionados podem ser configurados para executarem operações em cascata, de maneira a garantir a integridade referencial.

Coleções relacionadas são implementadas por meio das interfaces da API de `Collections` `Set` e `List`, e normalmente fazem uso do lazy-loading para evitar impactos no desempenho.

Spring

Spring é um framework para a plataforma Java criado por Rod Johnson, quando da publicação do seu livro “Expert One-on-One J2EE Design and Development” em Outubro de 2002.

Na verdade Spring, muito mais do que oferecer um mecanismo de persistência Java, constitui-se em um framework completo para aplicações Java, oferecendo recursos como inversão de controle, MVC, programação baseada em aspectos e muito mais.

Entre os recursos que o framework oferece para persistência podemos citar:

- Gerenciamento de recursos, como a obtenção e liberação automática de conexões com o banco de dados;
- Manipulação de exceções, convertendo exceções relacionadas à manipulação dos dados à hierarquia do framework;
- Gerenciamento de transações;
- Desempacotamento de recursos, como por exemplo a recuperação de objetos do banco de dados a partir de pools de conexão;
- Abstrações para lidar com campos do tipo BLOB e CLOB.

JDO

Java Data Objects (JDO) é uma especificação de persistência em Java criada em 2001 por meio da JSR-12.

Uma das suas principais características é a transparência dos serviços de persistência para o modelo de domínio: objetos persistentes JDO são objetos Java comuns, não existe a necessidade de que eles implementem ou estendam classes específicas do framework, como ocorria com os EJBs.

A persistência é definida por meio de arquivos de configuração XML, e é alcançada por meio dos "enhancers", que modificam o código compilado de forma a adicionar as funções de persistência.

EJB 3

Gradualmente surgiu um consenso de que a principal virtude da especificação EJB original – a integridade referencial em aplicações distribuídas – tinha pouca ou nenhuma utilidade para a maioria das aplicações comerciais atuais. As funcionalidades e o processo de desenvolvimento proporcionadas por frameworks mais simples como Spring e Hibernate pareciam muito mais interessantes. Desta forma, quando a especificação EJB 3 foi criada houve uma mudança radical de objetivos em favor deste novo paradigma.

A especificação do EJB 3 (JSR-220) mostrou uma clara influência do Spring no seu uso de POJOs e no seu suporte a injeção de dependências de forma a simplificar a configuração e integração de sistemas heterogêneos. Da mesma forma, Gavin King, o criador do Hibernate, participou ativamente na criação da especificação, e muitos dos recursos da biblioteca foram incorporados à especificação JPA, substituta dos entity beans no EJB 3.

A especificação EJB 3.0 pode ser decomposta em pelo menos duas partes principais. A primeira delas define o modelo de programação para session beans e message-driven beans, as regras de implantação e assim por diante. Já a segunda parte da especificação lida exclusivamente com persistência: entidades, mapeamento objeto-relacional, metadados, etc. Esta segunda parte da especificação é chamada de Java Persistence API (JPA), provavelmente porque todas as suas interfaces estão no pacote `javax.persistence`.

Esta separação também existe nos produtos EJB: alguns implementam um container EJB completo, que suporta todas as partes da especificação, enquanto que outros podem implementar apenas a parte de persistência. Dois princípios importantes foram criados ao se desenvolver a especificação:

- Os engines JPA devem ser plugáveis, o que significa que você deve ser capaz de remover um engine e substituí-lo por outro se não estiver satisfeito, mesmo se quiser continuar com o mesmo container EJB;
- Os engines JPA devem ser capazes de serem executados fora de um ambiente EJB 3.0, tornando-se disponíveis para aplicações Java comuns.

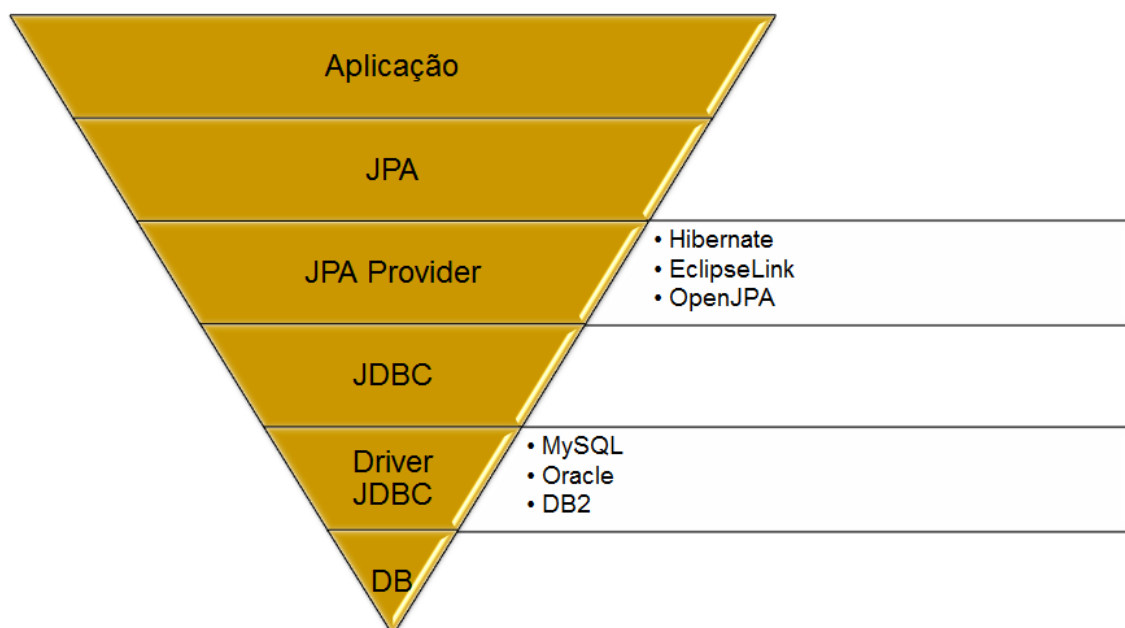
JPA

Java Persistence API (JPA) é um framework usado para lidar com dados armazenados em bancos de dados relacionais por meio das plataformas Java SE e Java EE. A primeira versão foi definida por meio da JSR-220, enquanto que a versão mais atual foi definida por meio da JSR-317, aprovada em 10 de Dezembro de 2009.

Esta API define uma maneira de mapear objetos comuns (POJOs) a um banco de dados. A API toma para si a tarefa de persistir estes objetos no banco, de forma que o programador não precise escrever qualquer código para conectar ao banco ou ler e salvar os dados. A especificação também define uma linguagem de consulta bastante semelhante à SQL, mas especializada para lidar com objetos Java ao invés de dados relacionais.

Os objetos que mapeiam dados ao banco são chamados de Entity Beans (beans de entidades), e ao contrário do que acontecia na especificação EJB 2.1, são objetos leves, ou seja, não precisam implementar qualquer interface ou classe em especial. Justamente por serem objetos comuns, os novos Entity Beans são portáteis não só entre servidores de aplicação, mas podendo inclusive serem transferidos entre um cliente e um servidor, tornando as aplicações muito mais simples e compactas.

Grosso modo podemos entender JPA como uma nova camada de abstração que nos permite desenvolver código que acessa o banco de dados de maneira muito mais otimizada e transparente do que aquela que costumávamos usar quando escrevíamos diretamente o código diretamente em JDBC.



Como o diagrama acima demonstra, a tecnologia JDBC continua fazendo parte essencial da nossa aplicação. Ocorre, entretanto, que ela fica abstraída por debaixo de uma camada adicional. Assim, é a JPA quem fica com a responsabilidade de gerar as queries SQL de que precisamos

para salvar ou recuperar os dados do banco: JPA pega seus beans e os persiste, e também pega os dados do banco e gera os beans correspondentes.

Os Provedores JPA

Assim como a tecnologia JDBC precisa que os fornecedores implementem os drivers de banco de dados, a tecnologia JPA também precisa que os fornecedores ofereçam implementações do padrão, os chamados Provedores JPA (JPA Providers).

Há vários deles, o EclipseLink, TopLink Essentials, OpenJPA e Hibernate. Neste material nos concentraremos no Hibernate, mas uma vez que a especificação determina como as coisas devem funcionar independentemente do fornecedor das bibliotecas, você não deve ter dificuldades se escolher usar outro provider. Tipicamente o que basta é acrescentar os arquivos `.jar` do provider ao classpath da sua aplicação, exatamente da maneira que faria com qualquer outra biblioteca (e exatamente da maneira que você fez para adicionar o driver JDBC do seu banco de dados).

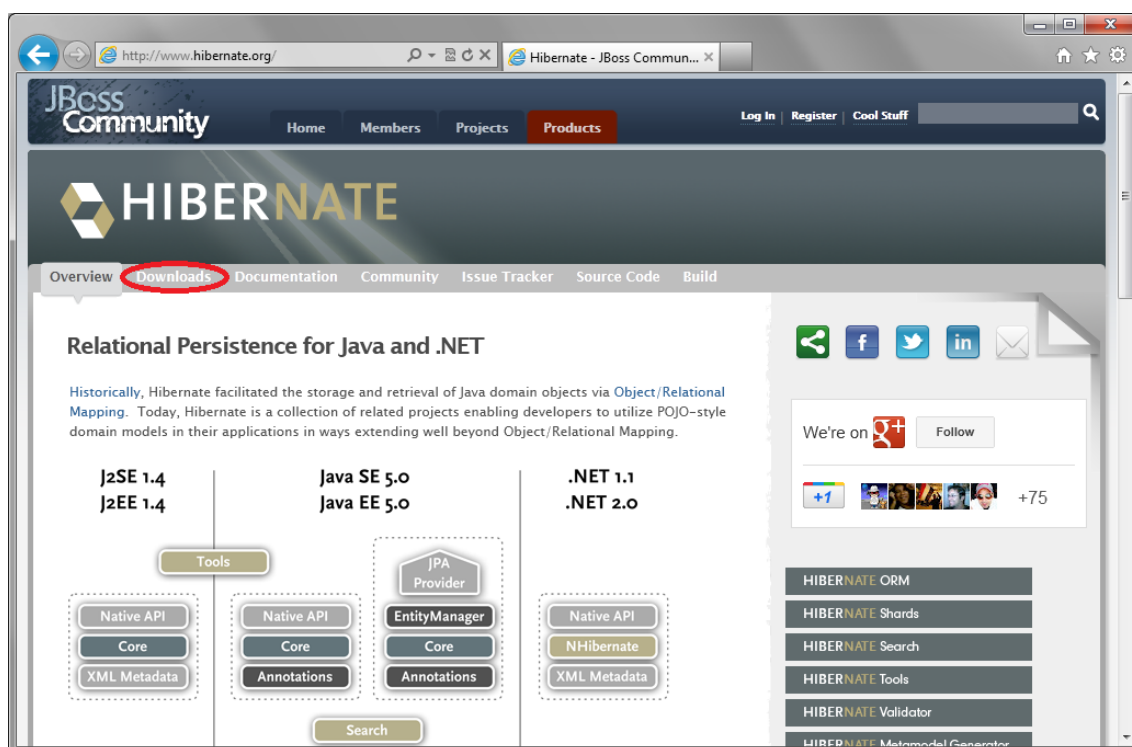
Capítulo 2 – Preparando Seu Ambiente

Uma vez que estamos decididos a utilizar o Hibernate como nosso provedor JPA, precisaremos fazer os downloads das bibliotecas do framework e adicioná-las ao nosso projeto. Vamos ver como fazer isso usando o NetBeans.

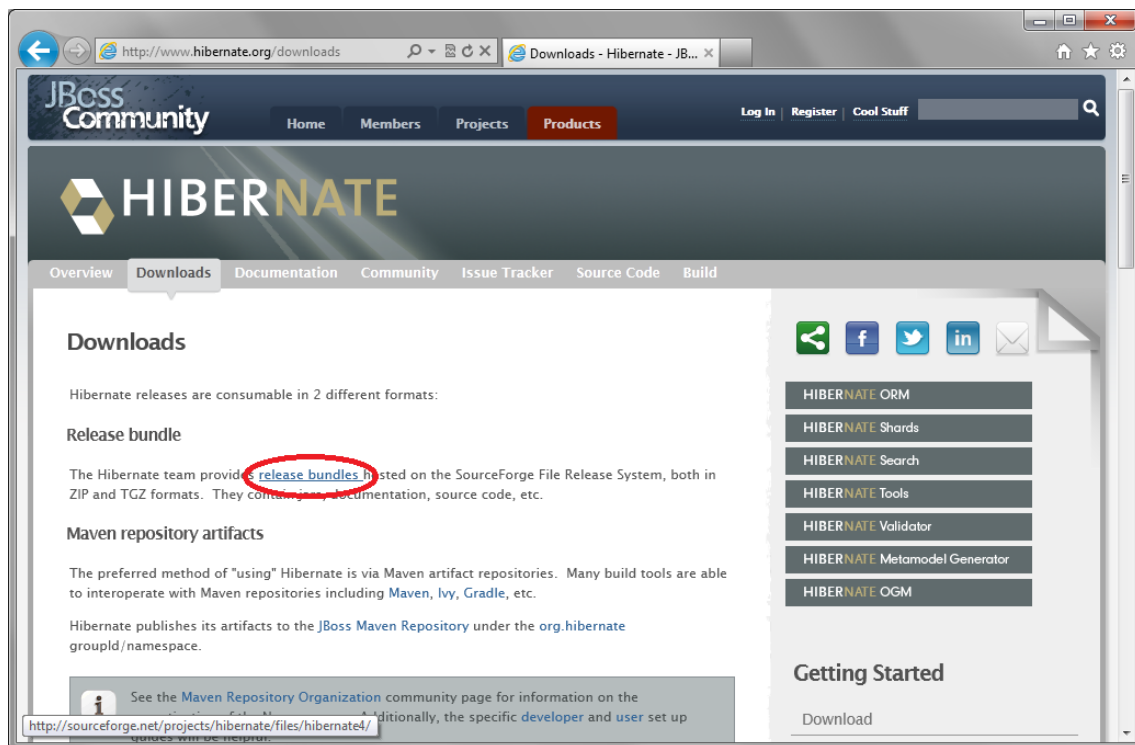
Fazendo o Download do Hibernate

O Hibernate pode se encontrado no site <http://www.hibernate.org>.

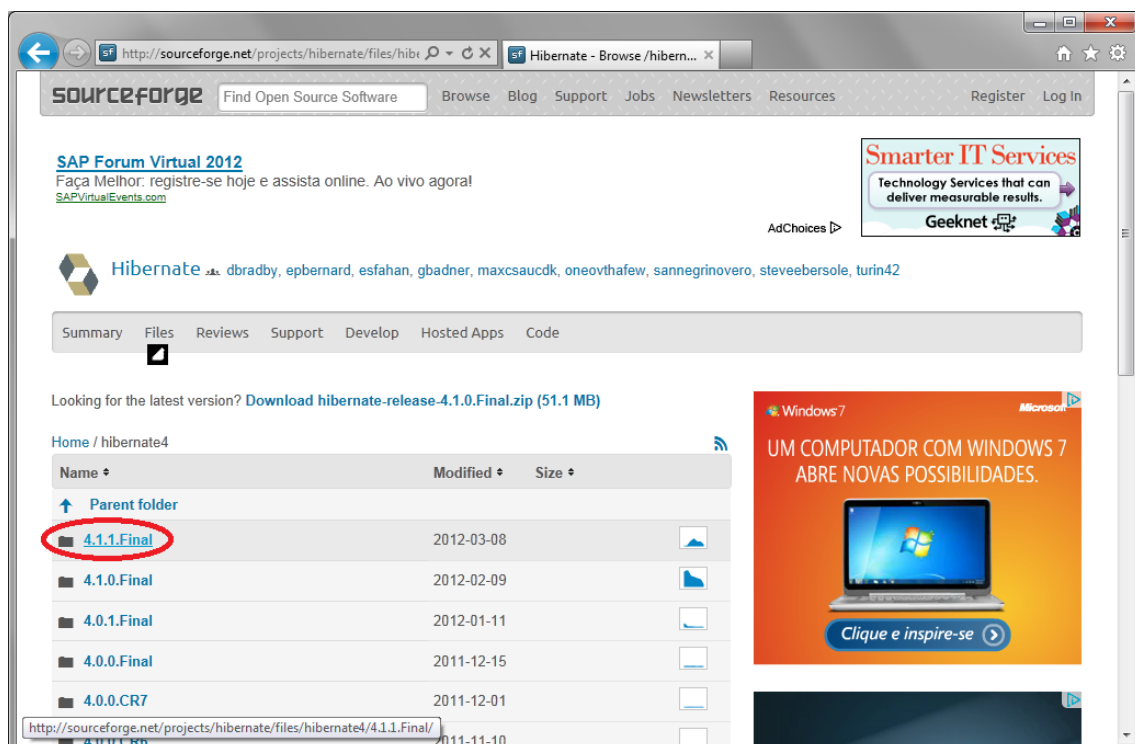
Quando acessar esta URL você verá a página mostrada abaixo:



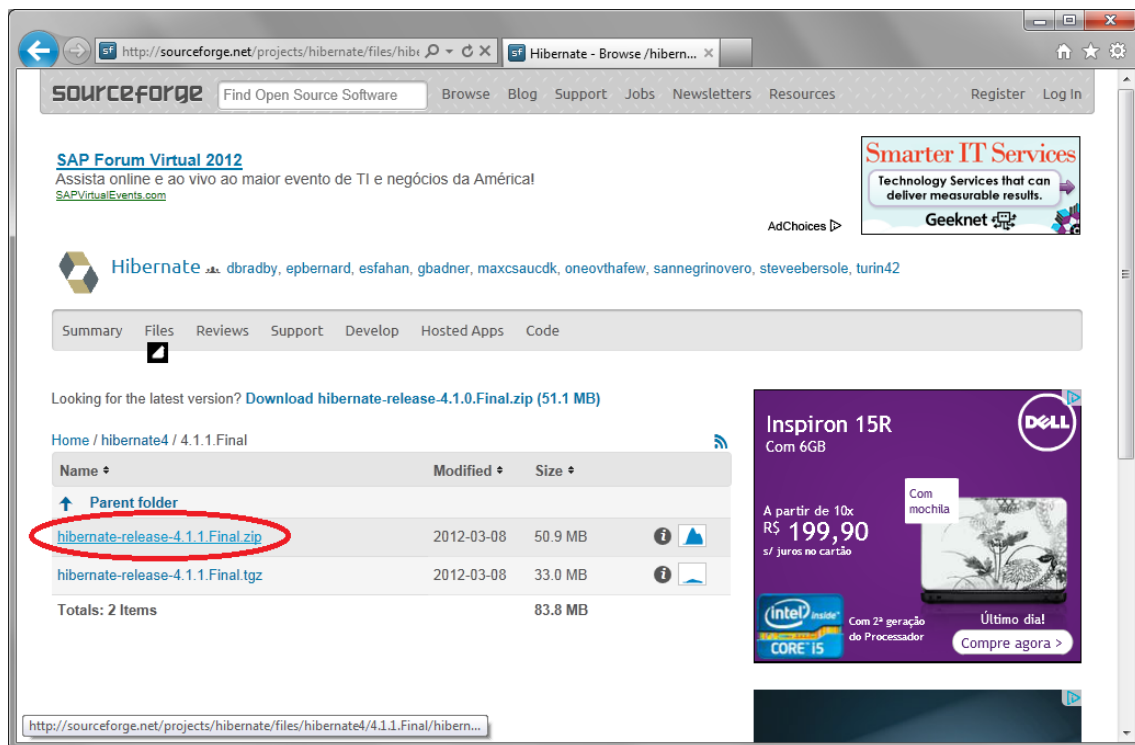
A homepage do projeto Hibernate apresenta as diversas versões da biblioteca. Comece por clicar em Download no topo da página.



O projeto Hibernate oferece muitas alternativas para fazer o download das bibliotecas. A mais conveniente para nós é baixar os Release Bundles por meio do SourceForge, um dos maiores repositórios de projetos open-source. Para tanto, clique em Release Bundles conforme mostra a imagem acima.



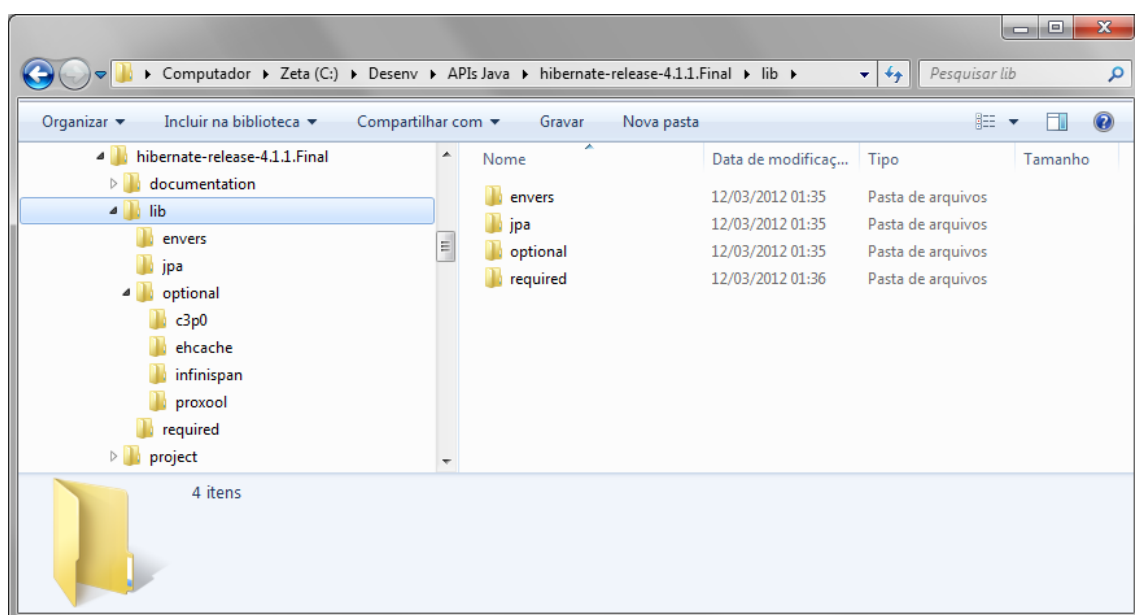
Agora selecione a versão mais recente disponível.



Por fim selecione o formato de arquivo que lhe parece mais conveniente (se você está usando o Windows o formato mais apropriado é provavelmente o `.zip`, mas se estiver usando o Unix/Linux então provavelmente o formato `.tgz` lhe será mais prático).

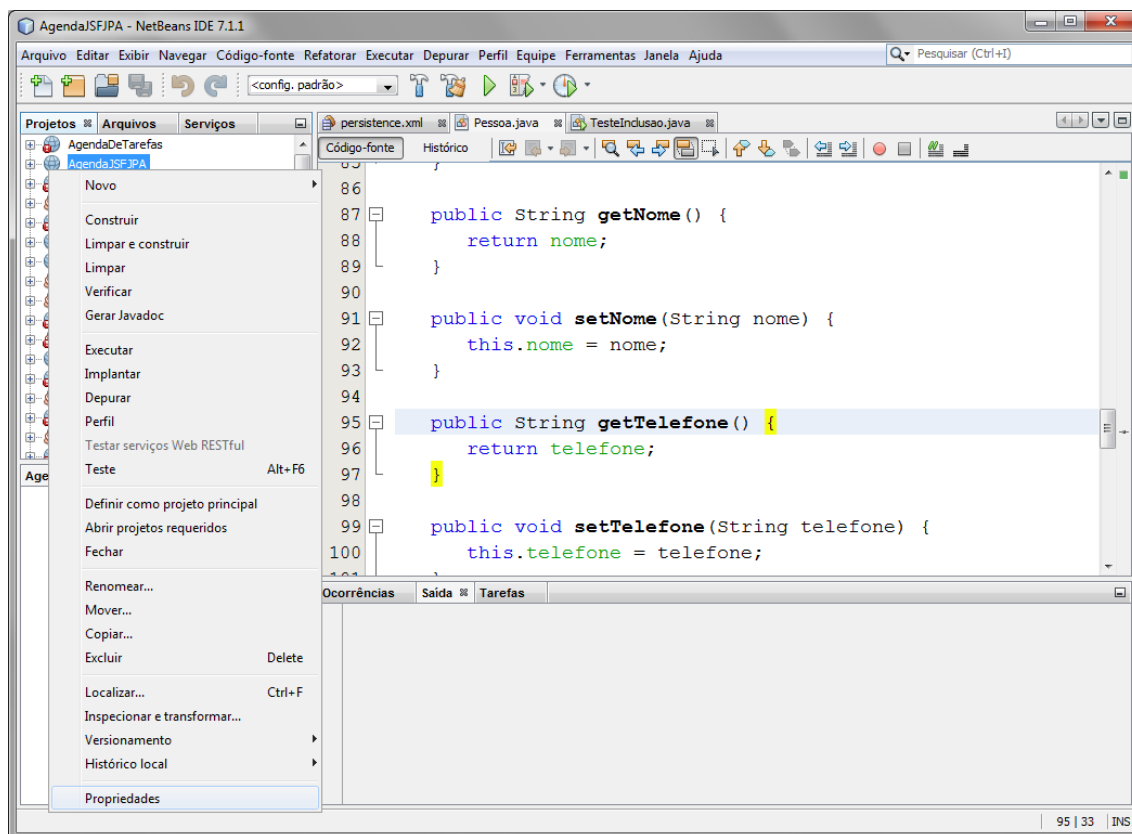
Adicionando o Hibernate ao Seu Projeto

Uma vez completado o download do Hibernate, agora você precisará adicioná-lo ao seu projeto. Para isso primeiro descompacte o arquivo baixado em algum diretório do seu computador. Você deverá terminar com uma estrutura parecida com a demonstrada abaixo:



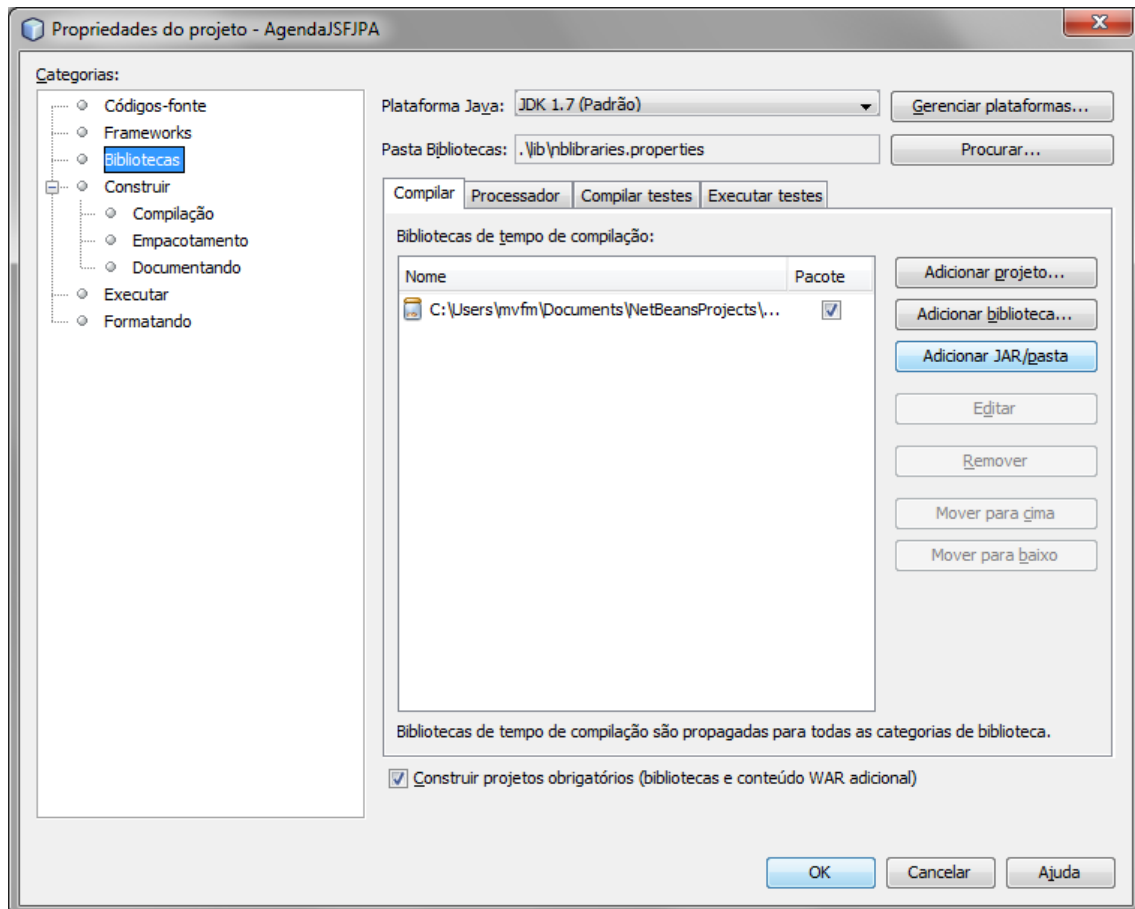
A figura mostrada acima apresenta a estrutura de diretórios criada depois que você extrai o arquivo .zip do Hibernate em uma máquina Windows. A partir do diretório de instalação você encontrará os subdiretórios `documentation`, `lib` e `project`. O mais importante, para nós, é o `lib`, que é onde ficam os diversos arquivos .jar que precisaremos incluir aos nossos projetos.

Mas não precisamos de todas as bibliotecas, apenas as que aparecem dentro dos diretórios `required` e `jpa`, e são estes arquivos que acrescentaremos aos nossos projetos, conforme mostram as imagens abaixo:

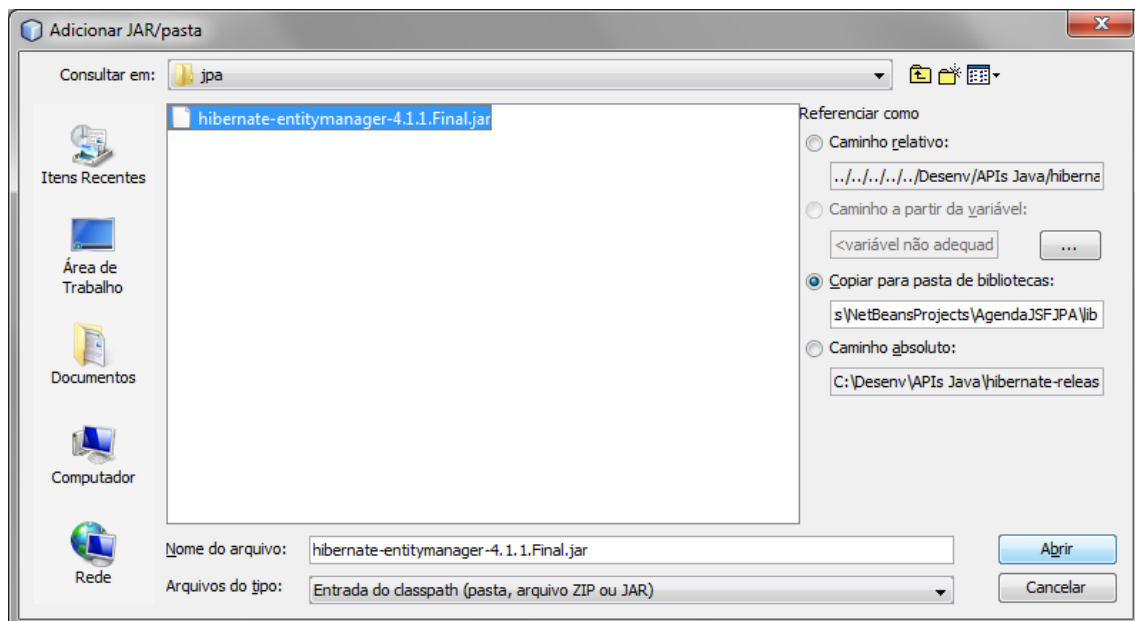


Para adicionar as bibliotecas do Hibernate ao seu projeto, primeiro crie seu projeto normalmente (tanto faz se ele será um projeto Java Web ou um projeto Java Desktop, os procedimentos e as bibliotecas serão exatamente os mesmos).

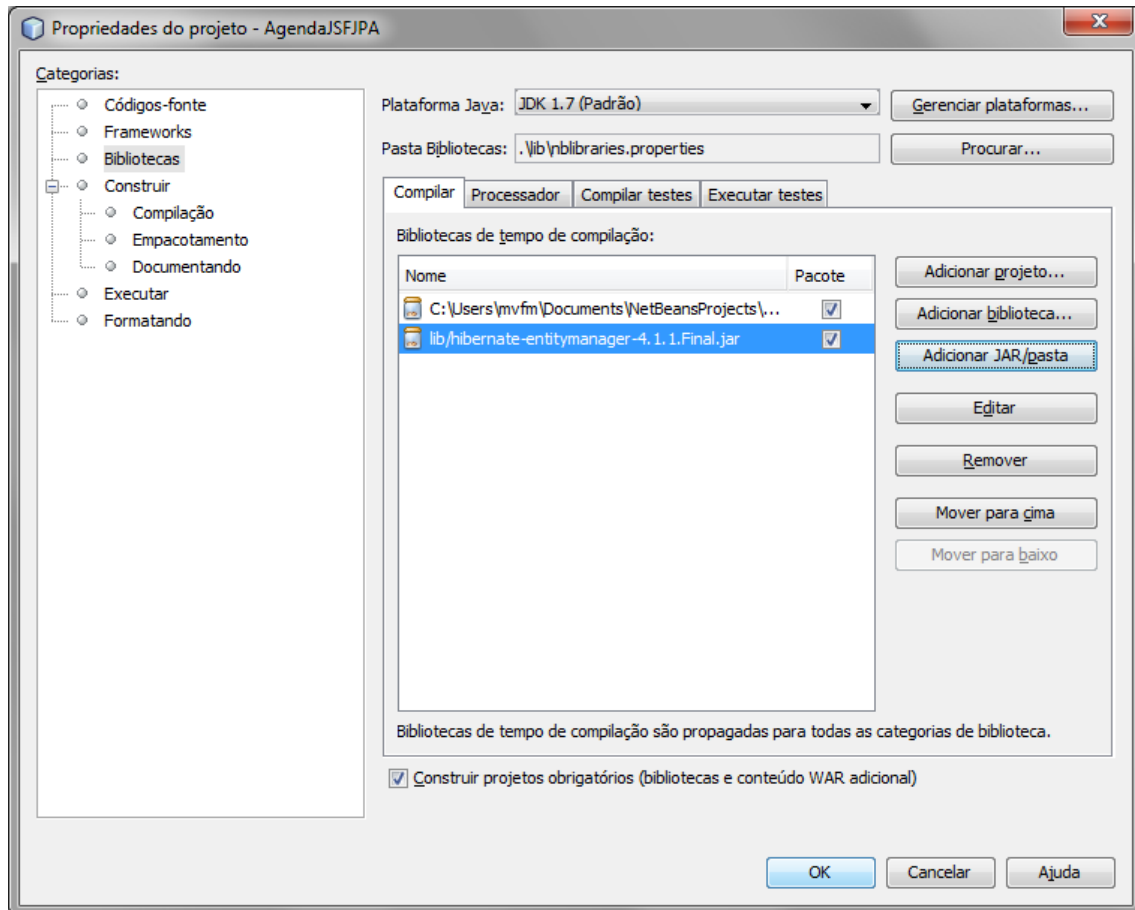
Uma vez que o projeto foi criado, agora clique com o botão direito sobre o nome do projeto no painel de Projetos e selecione a opção `Propriedades` para ver a janela de “Propriedades do projeto”.



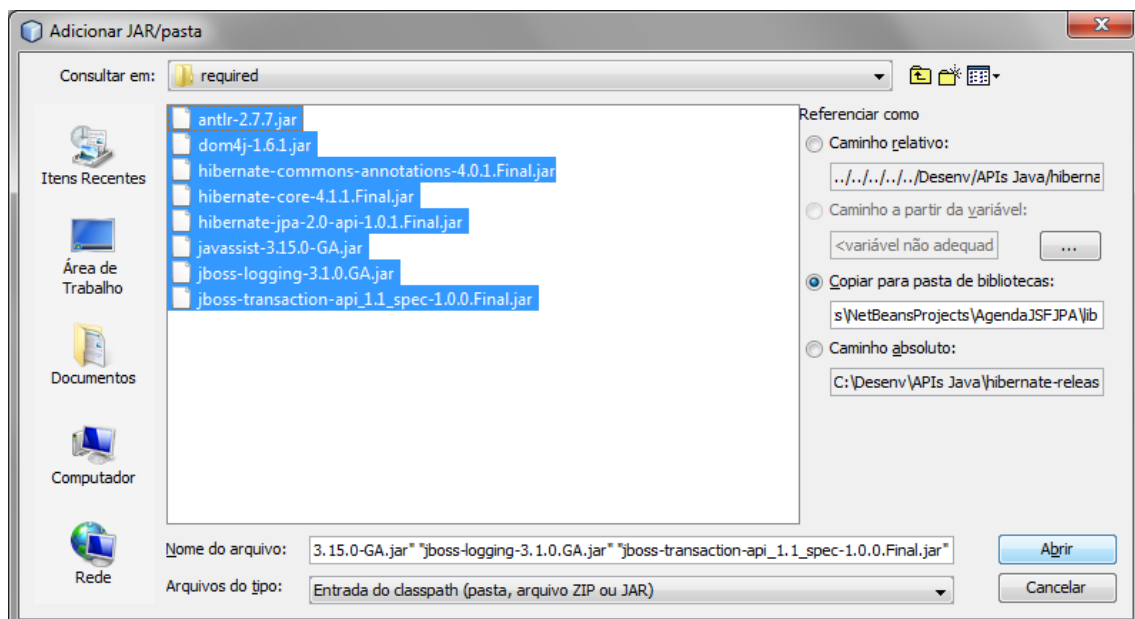
Nesta tela selecione a opção “Bibliotecas” para ver a tela mostrada acima. Uma vez lá, clique no botão “Adicionar JAR/pasta” para selecionar os arquivos `.jar` que você deseja adicionar ao classpath do projeto.



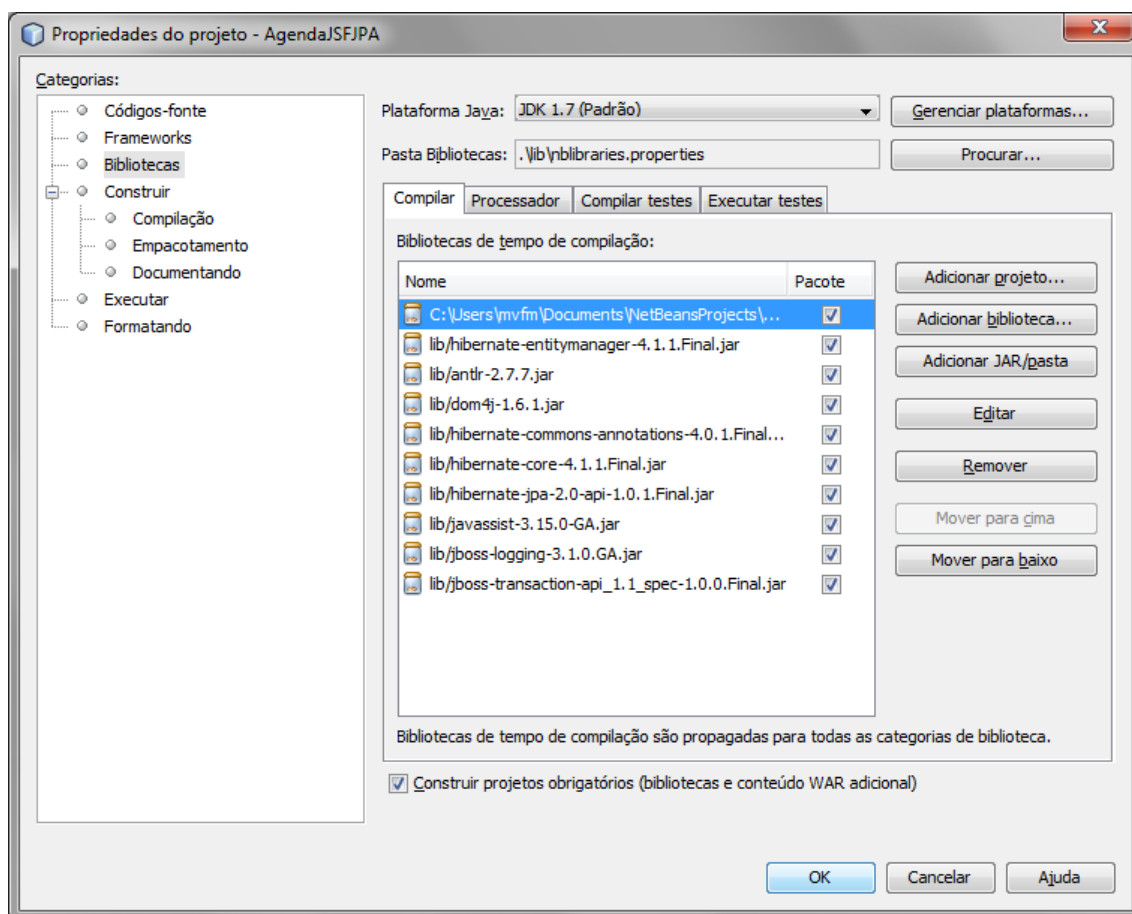
Comece navegando até o diretório `lib\jpa` do seu Hibernate e selecione o arquivo `.jar` que você encontrar lá dentro. Depois clique em “Abrir” para efetivamente adicioná-lo ao seu projeto.



Você será levado de volta à janela “Propriedades do projeto”. Lá, clique novamente em “Adicionar JAR/pasta” para adicionar as bibliotecas restantes.



Desta vez navegue até o diretório `lib\required` do seu Hibernate e selecione todos os arquivos lá presentes, clicando em seguida no botão “Abrir”.



Pronto, todas as bibliotecas necessárias foram acrescentadas. Clique em “OK” para concluir o processo.

Capítulo 3 – Entidades

Uma Entidade é um objeto de negócio capaz de ser persistido.

Normalmente uma entidade representa uma tabela em um banco de dados relacional, e cada instância da entidade representa um registro naquela tabela. O estado persistente de uma entidade é representado por seus atributos persistentes. Estes atributos fazem uso das anotações de mapeamento objeto/relacional para mapear as entidades e os relacionamentos com as tabelas no banco de dados subjacente.

Na especificação JPA, beans de entidade são objetos Java comuns que foram mapeados para tabelas em um banco de dados relacional. Ao contrário dos outros tipos EJB, entidades podem ser alocadas, serializadas e enviadas através da rede como qualquer outro POJO.

É comum entendermos que beans de entidade modelam conceitos da lógica de negócio que possam ser expressos na forma de substantivos: por exemplo, um bean de entidade poderia representar um cliente, uma peça de equipamento, um item no estoque, ou até mesmo um lugar. Em outras palavras, beans de entidade modelam objetos do mundo real, e estes objetos são costumeiramente geram registros em algum tipo de banco de dados.

Uma boa maneira de entender o projeto de beans de entidade é tentar você mesmo. Para criar um bean de entidade você precisará criar uma classe para o bean e definir qual campo você quer utilizar como identificador (ou chave primária):

- Chave Primária
 - A chave primária representa aquele atributo ou série de atributos que permitem que se identifique unicamente um registro no banco de dados. No caso dos entity beans, a chave primária serve para identificar um bean tanto em memória quanto na tabela. Em JPA, chaves primárias podem ser representadas tanto por classes quanto por tipos primitivos.
- Classe de Bean
 - Uma classe de bean é a representação de um registro do banco de dados na forma de um objeto. Normalmente os beans de entidade não apresentam qualquer lógica de negócio, embora alguns programadores gostem de implementar ao menos a lógica de validação nestes objetos.
 - Em JPA o bean de entidade é uma classe Java comum a qual, ao contrário do que acontecia nas versões anteriores da tecnologia EJB, não precisa implementar qualquer interface especial.
 - Classes de bean precisam ser marcadas com a anotação `@Entity`, e também precisam ter ao menos um atributo selecionado como sua chave primária (este por sua vez marcado com a anotação `@Id`). Há ainda uma série de outras anotações que podem se utilizadas para definir os parâmetros do mapeamento objeto-relacional.

Na tecnologia JPA, beans de entidades não são componentes, como costumavam ser na época dos EJB 2.1. A sua lógica de negócios interage diretamente com os beans de entidade, e não através de interfaces de componentes como você era obrigado quando utilizava os session beans.

Para que você possa interagir com os beans de entidade a JPA oferece um novo serviço chamado `EntityManager`. Todo acesso às entidades ocorre através deste serviço. Ele provê uma API de consultas e demais métodos de ciclo de vida para as entidades (incluir, alterar, excluir). Nenhuma mágica ou manipulação de bytecode, nenhum proxy especial, apenas código Java comum.

Ao contrário das versões anteriores da especificação EJB, com a JPA, beans de entidade e o `EntityManager` não precisam de um Servidor de Aplicação para serem usados. Você pode usar JPA em aplicações Java stand-alone ou mesmo em testes unitários.

A Classe de Bean

Teoricamente qualquer classe Java comum pode atuar como uma entidade para o JPA, porém algumas características se fazem necessárias:

- A classe deve ser marcada com a anotação `javax.persistence.Entity`;
- A classe deve ter um construtor público ou protegido sem parâmetros;
- A classe e seus atributos persistentes não podem ser finais;
- Se uma entidade às vezes é usada como um objeto desconectado e trafegada entre servidores, então deve ser marcada com a interface `java.io.Serializable`;
- Entidades podem estender tanto outras entidades quanto classes normais, e classes normais podem estender entidades;
- Os atributos persistentes devem ser declarados como `private`, `protected`, ou `default`, e só podem ser acessados diretamente pelos métodos da própria classe. Clientes sempre devem acessar os atributos por meio dos seus métodos getters e setters.

Entidades podem usar Campos Persistentes, Propriedades Persistentes ou uma combinação dos dois. Se as anotações de mapeamento são aplicadas às variáveis de instância, diz-se que a entidade usa campos persistentes. Se as anotações de mapeamento são aplicadas aos métodos getter, diz-se que a entidade usa propriedades persistentes.

Pessoa.java

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

@Entity
@Table(name="Pessoa")
public class Pessoa implements Serializable {

    @Id
    @GeneratedValue
    @Column(name="id", nullable=false)
    private int id;

    @Column(name="nome", nullable=false, length=45)
```

```
private String nome;

public int getId() {
    return id;
}

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

public String getNome() {
    return nome;
}

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

Campos Persistentes

Se a classe de entidade usa campos persistentes, o run-time de persistência acessa as variáveis de instância da classe diretamente. Todos os campos que não foram marcados com a anotação `@Transient` ou com a palavra-chave `transient` serão persistidos para o banco de dados.

As anotações de mapeamento objeto-relacional devem ser aplicadas às variáveis de instância.

Pessoa.java

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

@Entity
@Table(name="Pessoa")
public class Pessoa implements Serializable {

    @Id
    @GeneratedValue
    @Column(name="id", nullable=false)
    private int id;

    @Column(name="nome", nullable=false, length=45)
    private String nome;

    public int getId() {
        return id;
    }

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

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

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

Propriedades Persistentes

Se a entidade usa propriedades persistentes, a entidade deve seguir as convenções dos componentes JavaBeans.

Propriedades JavaBeans usam métodos getter e setter que tipicamente são nomeados a partir das variáveis de instância da classe. Para cada propriedade persistente do tipo Tipo haverá um método getter chamado `getPropriedade()` e um método setter chamado `setPropriedade()`. Se a propriedade for booleana você poderá chamar o método getter de `isPropriedade()`.

Por exemplo, se uma classe tem uma propriedade chamada nome do tipo String, você teria os dois métodos mostrados a seguir:

- `void setNome(String nome)`
- `String getNome()`

As anotações de mapeamento objeto-relacional devem ser aplicadas aos métodos getter.

Anotações de mapeamento não podem ser aplicadas às propriedades marcadas com a anotação `@Transient` ou com a palavra-chave `transient`.

Pessoa.java

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

@Entity
@Table(name="Pessoa")
public class Pessoa implements Serializable {

    private int id;

    private String nome;

    @Id
    @GeneratedValue
    @Column(name="id", nullable=false)
    public int getId() {
        return id;
    }
}
```

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

@Column(name="nome", nullable=false, length=45)
public String getNome() {
    return nome;
}

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

Anotações Básicas

Como dito anteriormente, a mágica do JPA é conseguida através das anotações de mapeamento objeto/relacional. As anotações do JPA estão todas definidas dentro do pacote `javax.persistence`. Abaixo veremos algumas das mais importantes:

A Anotação @Entity

A anotação `@Entity` (`javax.persistence.Entity`) é a anotação mais importante da JPA e deve aparecer na linha anterior à declaração de cada classe a ser persistida. Na maioria das vezes uma entidade representa uma tabela no banco de dados, e por convenção o nome da tabela correspondente a uma entidade tem o mesmo nome que a entidade (embora possamos mudar isso com o uso da anotação `@Table`).

A anotação `@Entity` possui os seguintes atributos:

Atributo	Descrição
name	O nome desta entidade, a ser utilizado em queries. Tipicamente a entidade tem o mesmo nome não-qualificado da classe, mas você pode definir um outro.

```
@Entity (name="Pessoa")
public class Pessoa {
    // ...
}
```

A Anotação @Table

A anotação `@Table` (`javax.persistence.Table`) é utilizada para definir o nome da tabela que será utilizada para persistir uma entidade.

A anotação `@Table` possui os seguintes atributos:

Atributo	Descrição
name	O nome desta entidade, a ser utilizado em queries. Tipicamente a entidade tem o mesmo nome não-qualificado da classe, mas você pode definir um outro.

Atributo	Descrição
catalog	Determina o nome do catalog da tabela.
schema	Determina o nome do schema da tabela.
uniqueConstraints	Define as constraints de unique-key da tabela. Veja a anotação @UniqueContraint para mais detalhes.

```
@Entity
@Table (name="Pessoa" schema="agendatelefonica"
public class Pessoa {
    ...
}
```

A Anotação @Id

A anotação @Id (javax.persistence.Id) é utilizada para marcar o campo ou propriedade persistente que atua como chave primária da entidade. A anotação suporta campos ou propriedades de tipos primitivos, wrappers, String, java.util.Date, java.sql.Date, java.math.BigDecimal ou java.math.BigInteger.

```
@Entity
public class Pessoa {
    @Id
    private int id;
    // ...
}
```

A Anotação @GeneratedValue

A anotação @GeneratedValue (javax.persistence.GeneratedValue) é utilizada para marcar um campo ou propriedade persistente que foi marcado como auto-incremento na tabela (apenas elementos marcados com @Id podem ser marcados com @GeneratedValue, já que apenas uma chave-primária pode ser auto-incremento).

A anotação @GeneratedValue tem os seguintes atributos:

Atributo	Descrição
generator	Determina o nome do gerador de chaves primárias a ser utilizado com esta tabela.

Atributo	Descrição
strategy	<p>Determina a estratégia de geração de chaves primárias a ser utilizada pelo provider nesta entidade, conforme definidas no Enum <code>javax.persistence.GenerationType</code>:</p> <ul style="list-style-type: none"> • AUTO: indica que o provider deve definir a melhor estratégia para este banco de dados em particular (teoricamente esta é a melhor configuração, pois garante a portabilidade); • IDENTITY: indica que o provider deve assinalar chaves primárias usando uma coluna de identidade (suportadas por bancos DB2, MySQL, SQL Server, Sybase e outros); • SEQUENCE: indica que o provider deve assinalar chaves primárias usando uma sequence (suportadas por bancos como o Oracle, Sybase e DB2, entre outros); • TABLE: indica que o provider deve assinalar chaves primárias usando outra tabela para garantir unicidade.

```
@Entity
public class Pessoa {
    @Id
    @GeneratedValue (strategy=GenerationTypes.AUTO)
    private int id;
    // ...
}
```

A Anotação @Column

A Anotação `@Column` (`javax.persistence.Column`) deve ser utilizada para marcar campos ou propriedades persistentes em uma entidade.

Os seguintes atributos podem ser definidos:

Atributo	Descrição
columnDefinition	Define o fragmento de SQL que poderia ser utilizado para gerar o DDL da coluna.
insertable	Determina se a coluna deve aparecer nas instruções INSERT SQL geradas pelo provedor de persistência.
length	Define o número de caracteres da coluna.
nullable	Indica se esta coluna pode receber valores nulos ou não.

Atributo	Descrição
precision	Indica o número de dígitos na parte fracionária da coluna.
scale	Indica o número de dígitos na parte inteira da coluna.
table	Determina o nome da tabela que contém a coluna.
unique	Indica se esta coluna é uma chave única ou não.
updatable	Determina se esta coluna deve aparecer nas instruções UPDATE SQL geradas pelo provedor de persistência.

```
@Entity
public class Pessoa {
    @Id
    @Column (name="id" nullable=false)
    private int id;
    // ...
}
```

A Anotação @Temporal

Por padrão os campos ou atributos do tipo `java.util.Date` ou `java.util.Calendar` serão persistidos com o tipo SQL `TIMESTAMP`, cuja semântica permite armazenar uma data e hora. Se quisermos persistir apenas a data ou a hora precisamos marcar o campo ou atributo com a anotação `@Temporal` (`javax.persistence.Temporal`).

A anotação `@Temporal` tem apenas um atributo:

Atributo	Descrição
value	<p>Determina o tipo de dados SQL usado para persistir uma coluna, conforme definição no Enum <code>javax.persistence.TemporalType</code>:</p> <ul style="list-style-type: none"> • <code>DATE</code>: persiste apenas a data; • <code>TIME</code>: persiste apenas a hora; • <code>TIMESTAMP</code>: persiste data e hora simultaneamente.

```
@Entity
public class Pessoa {
    // ...
    @Column (name="data_nascto")
    @Temporal (value=TemporalType.DATE)
    private Date dataNascto;
}
```

A Anotação @Lob

A anotação `@Lob` (`javax.persistence.Lob`) indica que um campo ou atributo persistente deve ser armazenado como um LOB (Large OBject) no banco de dados. O tipo Lob efetivamente utilizado é inferido a partir do tipo do campo ou atributo persistente, e exceto para Strings e arranjos de char será BLOB.

A anotação `@Lob` pode ser usada em conjunto com a anotação `@Basic` para determinar a estratégia de recuperação de dados (atributo `fetch`, que pode receber `EAGER` ou `LAZY`).

```
@Entity
public class Pessoa {
    // ...
    @Lob
    private byte[] foto;
}
```

A Anotação @Embedded

A anotação `@Embedded` (`javax.persistence.Embedded`) especifica um campo ou propriedade persistente cujo valor é uma instância de uma classe a ser embutida.

Esta anotação é tipicamente utilizada para modelar os “atributos compostos” da modelagem entidade-relacionamento, aqueles atributos que podem ser decompostos em elementos menores (como endereço, por exemplo, pode ser decomposto em nome da rua, número, complemento, etc.).

```
@Entity
public class Pessoa {
    // ...
    @Embedded
    private Endereco endereco;
}
```

A Anotação @Embeddable

A anotação `@Embeddable` (`javax.persistence.Embeddable`) define uma classe cujas instâncias são armazenadas como uma parte intrínseca de uma entidade com a qual ela compartilha a identidade. Cada uma das propriedades ou campos persistentes do objeto embutido é associada a uma coluna da tabela à qual a entidade está vinculada.

```
@Embeddable
public class Endereco {
    String rua;
    int numero;
    String complemento;
}
```

A Anotação @Transient

A anotação `@Transient` (`javax.persistence.Transient`) deve ser utilizada para marcar campos ou atributos que não devem ser persistidos.

```
@Entity
public class Pessoa {
    // ...
    @Transient
    private boolean processado;
}
```

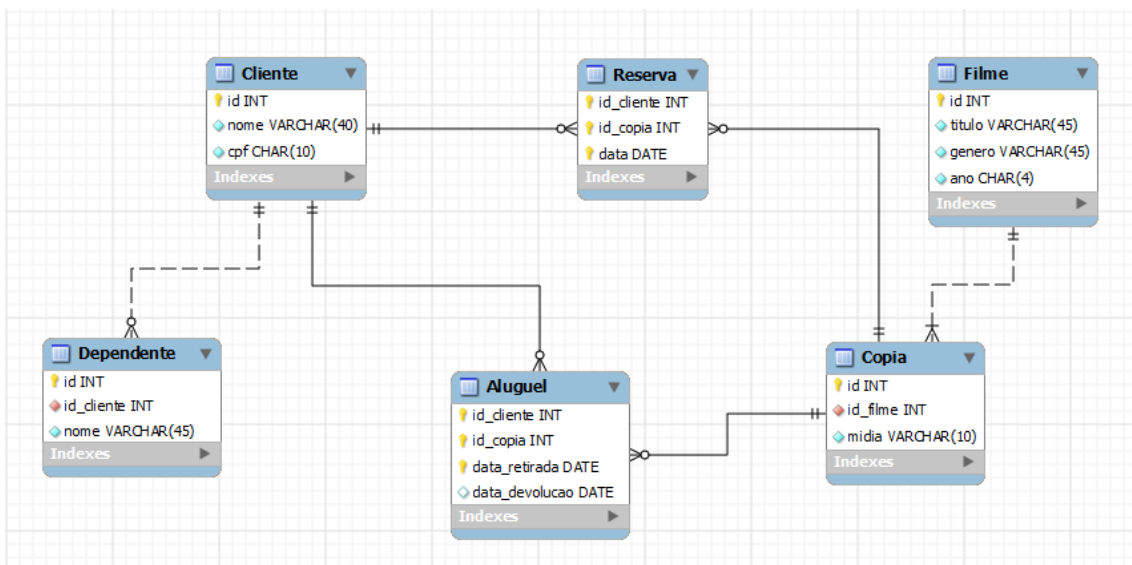
}

Anotando Chaves Primárias Compostas

Uma Chave Primária Composta (Composite Primary Key, em inglês) é uma chave formada a partir da combinação de mais de uma coluna. Estes tipos de chaves são utilizadas quando uma única coluna não é suficiente para identificar univocamente um registro.

Na JPA, chaves compostas precisam ser especificadas em uma classe criada especificamente para este fim. As chaves primárias compostas são indicadas por meio das anotações `@IdClass` ou `@EmbeddedId`.

Vejamos por exemplo o seguinte banco de dados:



Considere a tabela Aluguel, composta por quatro colunas, três delas fazendo parte da chave primária. Vejamos como podemos representá-la usando JPA.

A notação @IdClass

A anotação `@IdClass` (`javax.persistence.IdClass`) especifica uma classe de chave primária composta associada a múltiplos campos ou propriedades da entidade.

Aluguel.java

```

package br.com.alfamidia.videolocadora.bean;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@IdClass(AluguelPK.class)
public class Aluguel implements Serializable {

    @Id

```

```
@Column (name = "id_cliente")
int idCliente;

@Id
@Column (name = "id_copia")
int idCopia;

@Id
@Column (name = "data_retirada")
@Temporal (TemporalType.DATE)
Date dataRetirada;

@Column (name = "data_devolucao")
@Temporal (TemporalType.DATE)
private Date dataDevolucao;

// Construtores, getters, setters ...
}
```

Note que neste a anotação `@IdClass` indica que classe deve ser utilizada para representar a PK. Além disso, ainda precisamos de três campos ou propriedades marcados com a anotação `@Id`, um para cada coluna que compõe a PK da tabela. Estes campos ou propriedades não podem ser `private`.

AluguelPK.java

```
package br.com.alfamidia.videolocadora.bean;

import java.io.Serializable;
import java.util.Date;

public class AluguelPK implements Serializable {
    int idCliente;

    int idCopia;

    Date dataRetirada;
}
```

A classe de PK é uma classe Java comum, mas os seus campos não podem ser privados. Note que os nomes dos campos na classe de PK devem ser os mesmos dos campos ou propriedades de PK na entidade, e seus tipos de dados também devem ser iguais.

A Anotação @EmbeddedId

A anotação `@EmbeddedId` (`javax.persistence.EmbeddedId`) representa uma maneira alternativa de se definir uma chave primária composta em JPA. Neste caso, a classe de entidade contém apenas um único atributo que representa a sua chave primária, e este atributo é do tipo da classe de PK (claro que a classe pode ter outros atributos para as demais colunas da tabela):

Aluguel.java

```
package br.com.alfamidia.videolocadora.bean;

import java.io.Serializable;
```

```
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Aluguel implements Serializable {

    @EmbeddedId
    protected AluguelPK aluguelPK;

    @Column (name = "data_devolucao")
    @Temporal (TemporalType.DATE)
    private Date dataDevolucao;

    // Construtores, getters, setters ...
}
```

Note que ao invés de incluirmos atributos específicos para cada um dos campos da PK incluimos apenas um atributo contendo a classe de PK.

AluguelPK.java

```
package br.com.alfamidia.videolocadora.bean;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Embeddable;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Embeddable
public class AluguelPK implements Serializable {
    @Column (name = "id_cliente")
    private int idCliente;

    @Column (name = "id_copia")
    private int idCopia;

    @Column (name = "data_retirada")
    @Temporal (TemporalType.DATE)
    private Date dataRetirada;

    // Construtores, getters, setters ...
}
```

Note que agora nossa classe de PK é marcada com a anotação `@Embeddable` e demais anotações da JPA que porventura se fizerem necessárias (`@Column` e `@Temporal`, no nosso caso).

Capítulo 4 – Unidade de Persistência

A JPA trouxe uma novidade muito bem-vinda aos desenvolvedores Java: ao contrário dos EJBs, que eram restritos às aplicações Java EE, JPA é capaz de oferecer os serviços de persistência tanto para aplicações Java EE quanto para aplicações Java SE comuns.

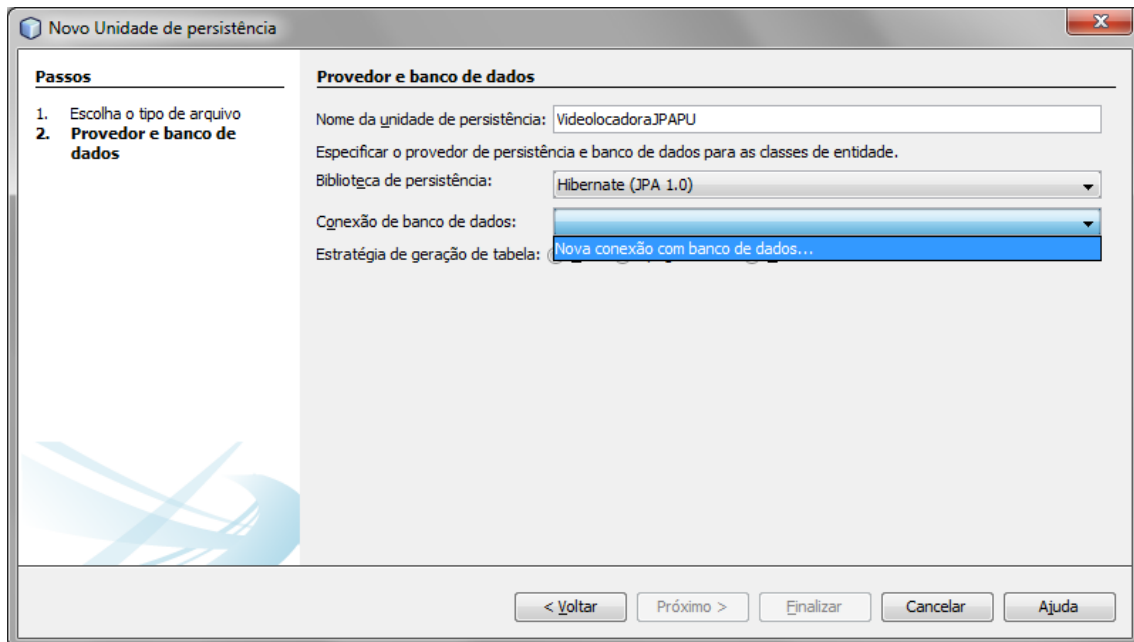
Mas para que possamos utilizar os serviços da JPA primeiro precisamos definir uma “unidade de persistência”, que é uma maneira de dizer ao ambiente a qual banco de dados queremos nos conectar e quais são as entidades que desejamos ver sincronizadas.

Para fazer isso precisamos criar um arquivo chamado `persistence.xml`, que funciona como se fosse o “deployment descriptor” para uma aplicação JPA. Um arquivo `persistence.xml` pode definir uma ou mais unidades de persistência, e é tipicamente colocado dentro do diretório META-INF na raiz do diretório de fontes da aplicação.

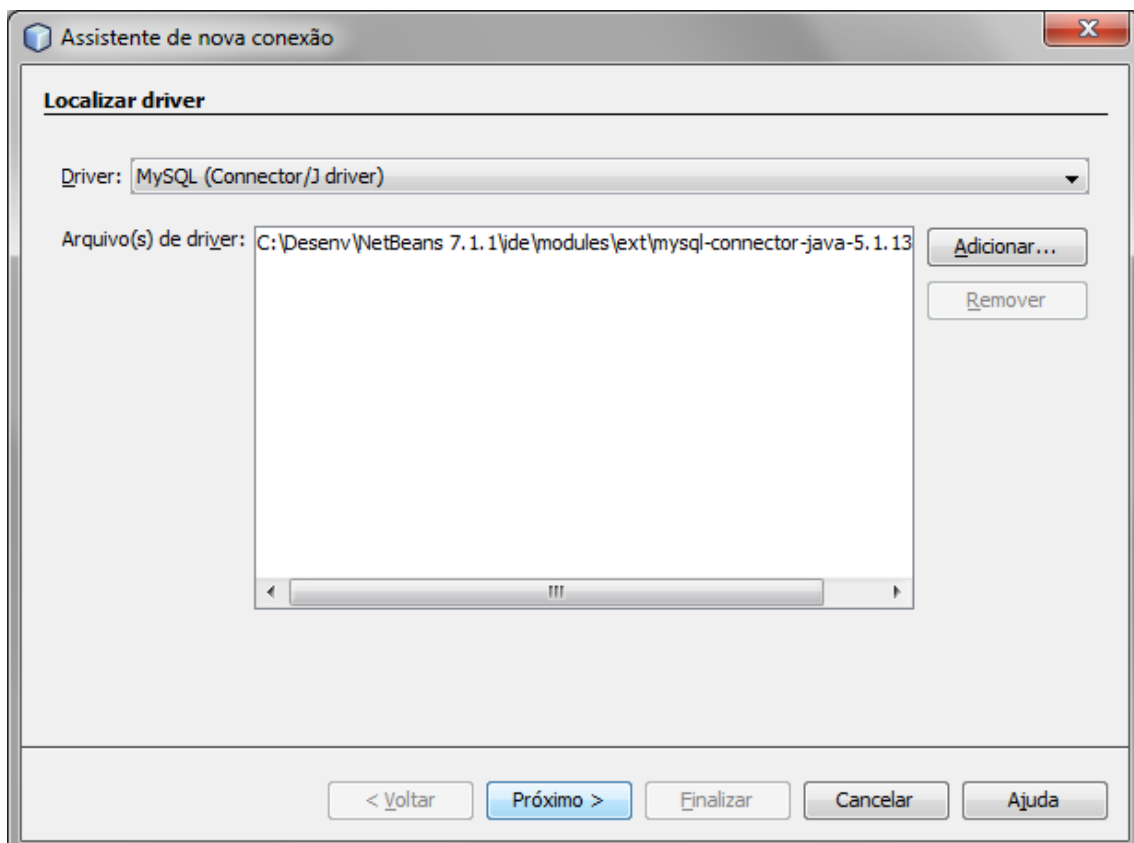
Obviamente você também pode usar o NetBeans para criar o arquivo para você, mas o processo será sutilmente diferente se você estiver criando o arquivo para uma aplicação Java SE ou Java EE.

Criando Uma Unidade de Persistência para Uma Aplicação Java SE

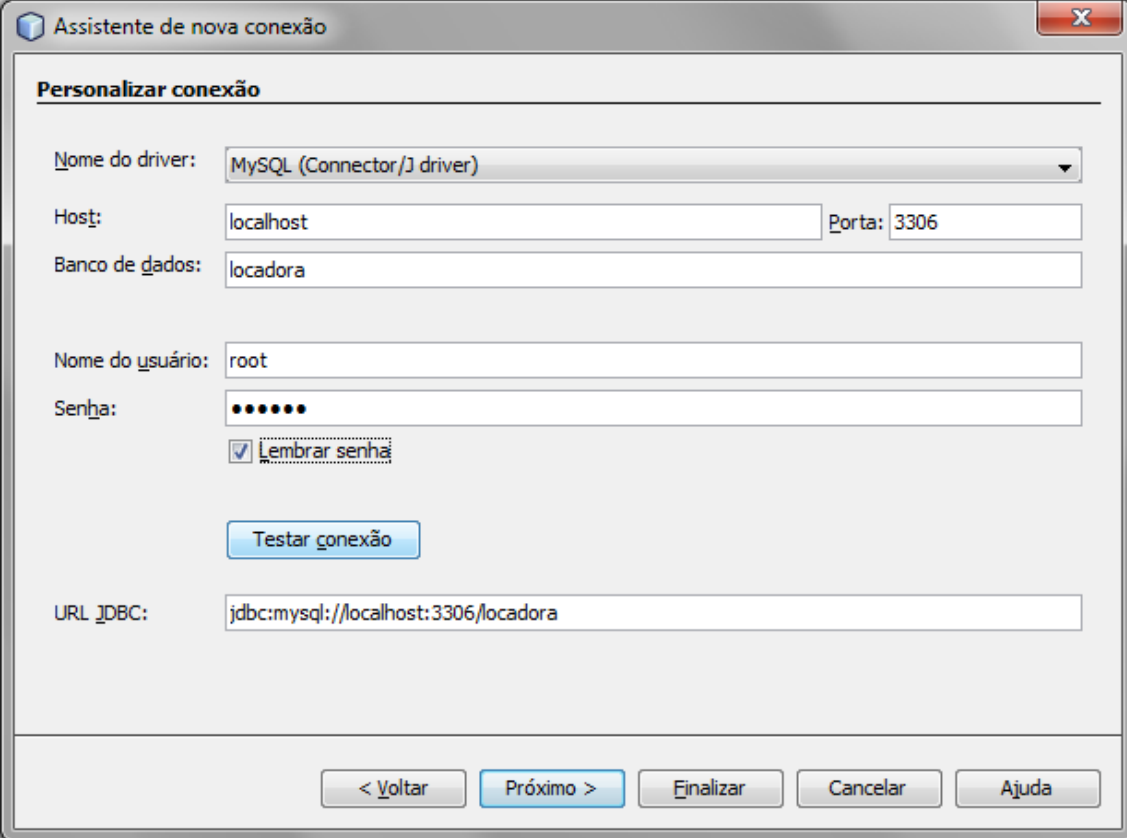
Primeiro abra o seu projeto e depois selecione Arquivo > Novo Arquivo... para ver a tela exibida abaixo:



A tela mostrada acima, a primeira do assistente de criação de novas unidades de persistência, serve para que você informe o nome da unidade de persistência, o seu provider e a conexão de banco de dados a utilizar. O nome é bastante arbitrário, e você pode até mesmo deixá-lo em branco se assim o desejar. Quanto ao provider, escolha Hibernate (já que foram as suas bibliotecas que adicionamos ao projeto), e em “Conexão de Banco de Dados” escolha alguma que você tenha criado anteriormente ou selecione “Nova conexão com banco de dados...” se deseja criar uma. É o que faremos nesta demonstração.



Quando você indica que deseja criar uma nova conexão de banco de dados uma nova janela se abre, o “Assistente de nova conexão”. Aqui informe o nome do driver a utilizar (no exemplo escolhemos MySQL).



A imagem mostra uma janela de software intitulada "Assistente de nova conexão". O título da janela é "Assistente de nova conexão" com um ícone de ajuda e um botão de fechar. O conteúdo da janela é dividido em seções. A primeira seção, intitulada "Personalizar conexão", contém os seguintes campos: "Nome do driver:" com uma lista suspensa selecionando "MySQL (Connector/J driver)"; "Host:" com o texto "localhost" e "Porta:" com o texto "3306"; "Banco de dados:" com o texto "locadora"; "Nome do usuário:" com o texto "root"; e "Senha:" com pontos e um checkbox "Lembrar senha" selecionado. Abaixo desses campos há um botão "Testar conexão". A segunda seção, intitulada "URL JDBC:", contém um campo de texto com o valor "jdbc:mysql://localhost:3306/locadora". Na base da janela, há uma barra com cinco botões: "< Voltar", "Próximo >", "Finalizar", "Cancelar" e "Ajuda".

A tela acima solicita as informações da conexão. Aqui informe o nome do host onde o banco de dados está instalado (e eventualmente sua porta, se o servidor não usa a default), o nome do banco de dados, usuário e senha (é bastante útil marcar o checkbox “Lembrar senha”). À medida que você vai fazendo isso o assistente vai populando o campo “URL JDBC” para você.

Depois de informar tudo você pode clicar no botão “Testar conexão” para verificar se os dados que informou realmente são capazes de obter uma conexão com o banco de dados.

Assistente de nova conexão

Personalizar conexão

Nome do driver: MySQL (Connector/J driver)

Host: localhost Porta: 3306

Banco de dados: locadora

Nome do usuário: root

Senha: ●●●●●●

☒ Lembrar senha

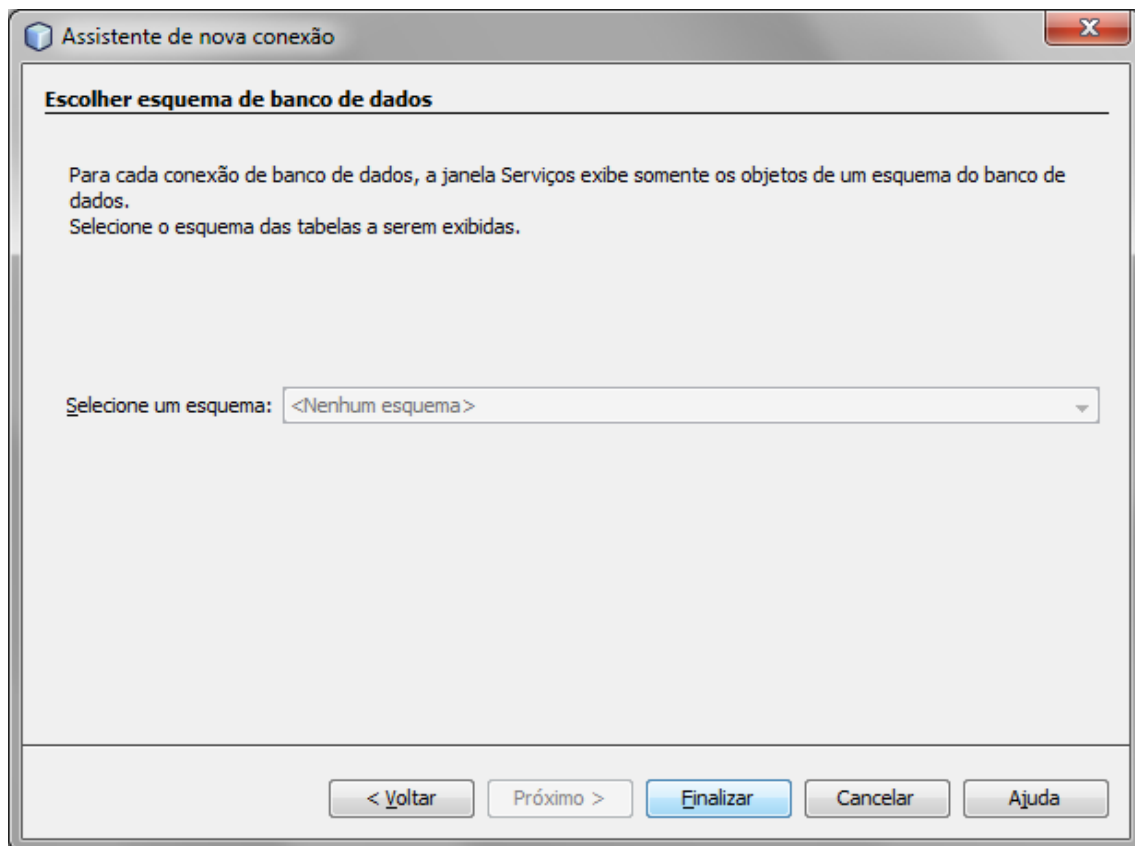
Testar conexão

URL JDBC: jdbc:mysql://localhost:3306/locadora

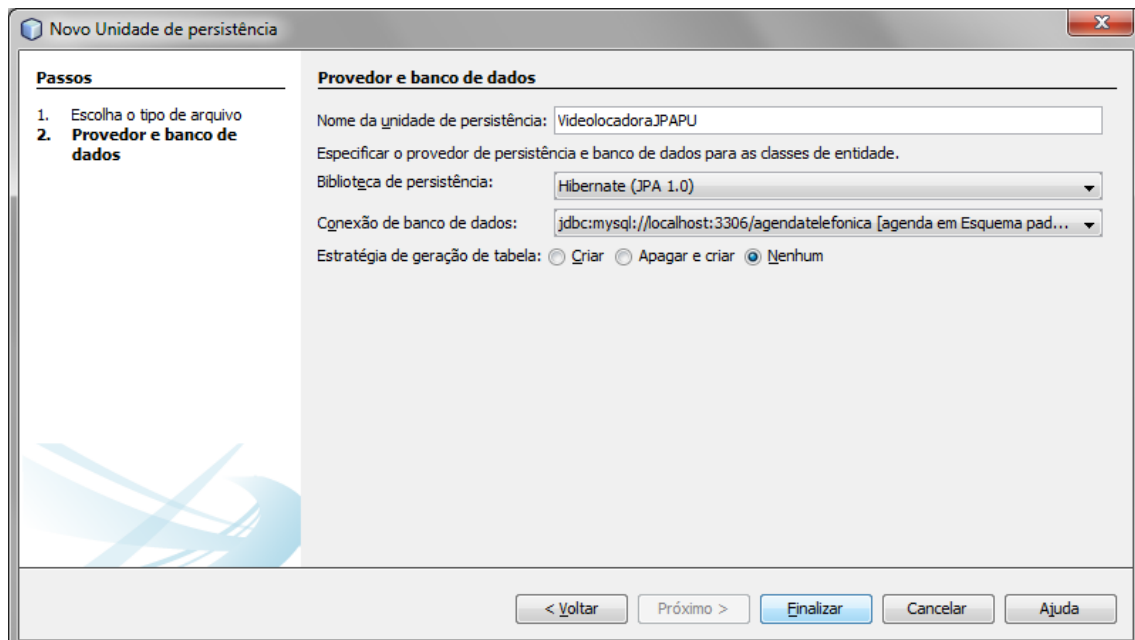
i Conexão bem-sucedida.

< Voltar Próximo > Finalizar Cancelar Ajuda

Se tudo funcionou você verá um texto indicativo avisando que a conexão foi bem-sucedida. Clique em “Próximo” para continuar.

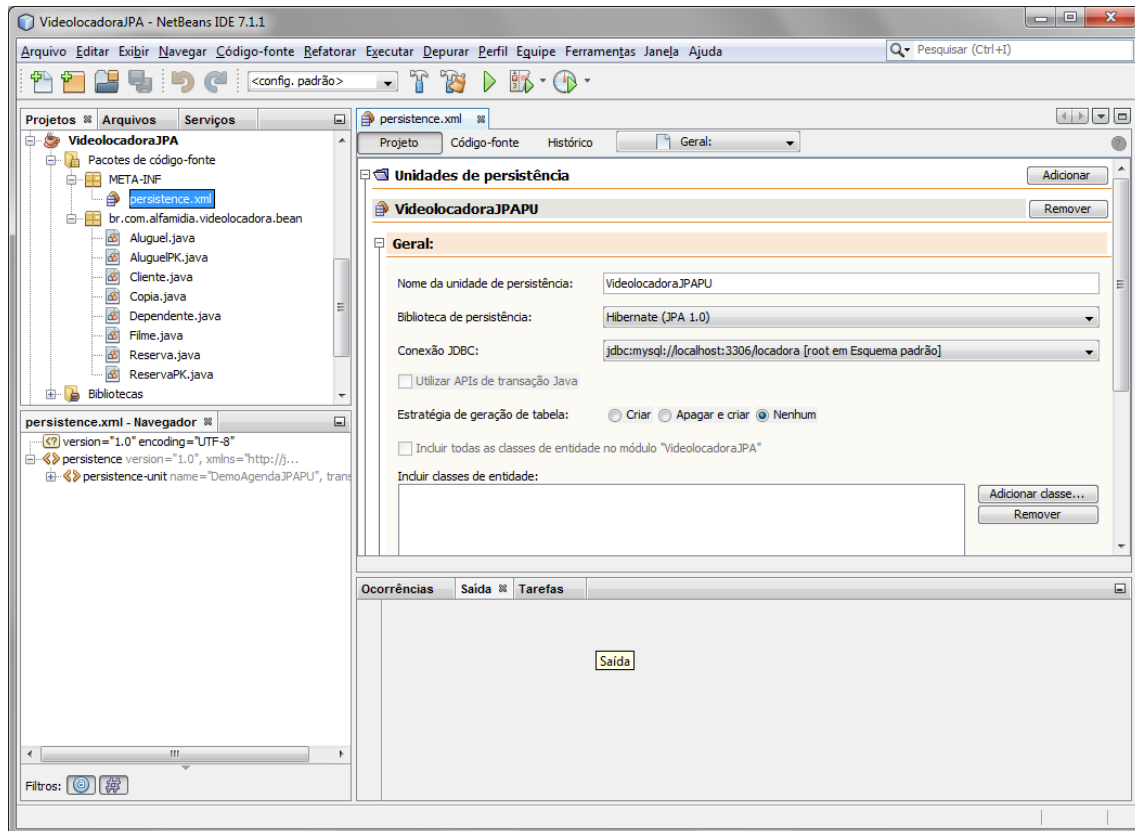


Nosso banco é o MySQL, então não há nada a informar aqui. Simplesmente clique em “Finalizar” para retornar ao assistente de unidade de persistência.



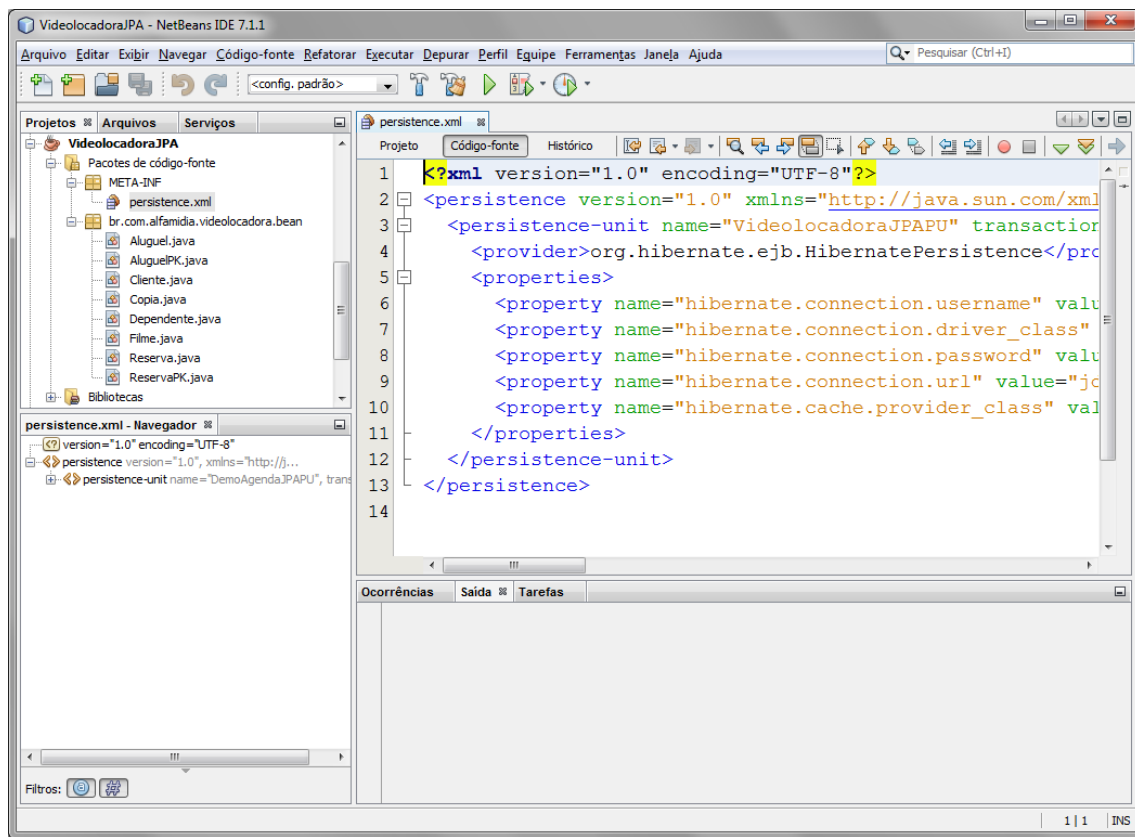
De volta ao assistente de unidade de persistência, certifique-se de ter marcado a opção “Nenhum” em “Estratégia de geração de tabelas” para evitar que seu banco de dados seja sobrescrito toda vez que iniciar o container.

Ao clicar em “Finalizar” você terá criado o seu arquivo `persistence.xml`. Você poderá editá-lo na tela a seguir.



A tela acima exibe o editor gráfico do arquivo `persistence.xml`, o qual apresenta uma maneira rápida e conveniente de atualizar o arquivo caso necessário.

Se clicar na aba “Código fonte” você poderá ver que este arquivo não passa de um arquivo XML comum, conforme mostrado na próxima imagem.



Na imagem acima você pode ver o fonte do arquivo persistence.xml. Note que os sub-elementos da tag <properties> são vendor-specific e serão diferentes se você escolher um outro JPA provider. No caso do Hibernate vemos as propriedades que definem o método de acesso ao banco, como por exemplo o nome da classe de driver JDBC, a URL do banco, nome de usuário e senha.

persistence.xml

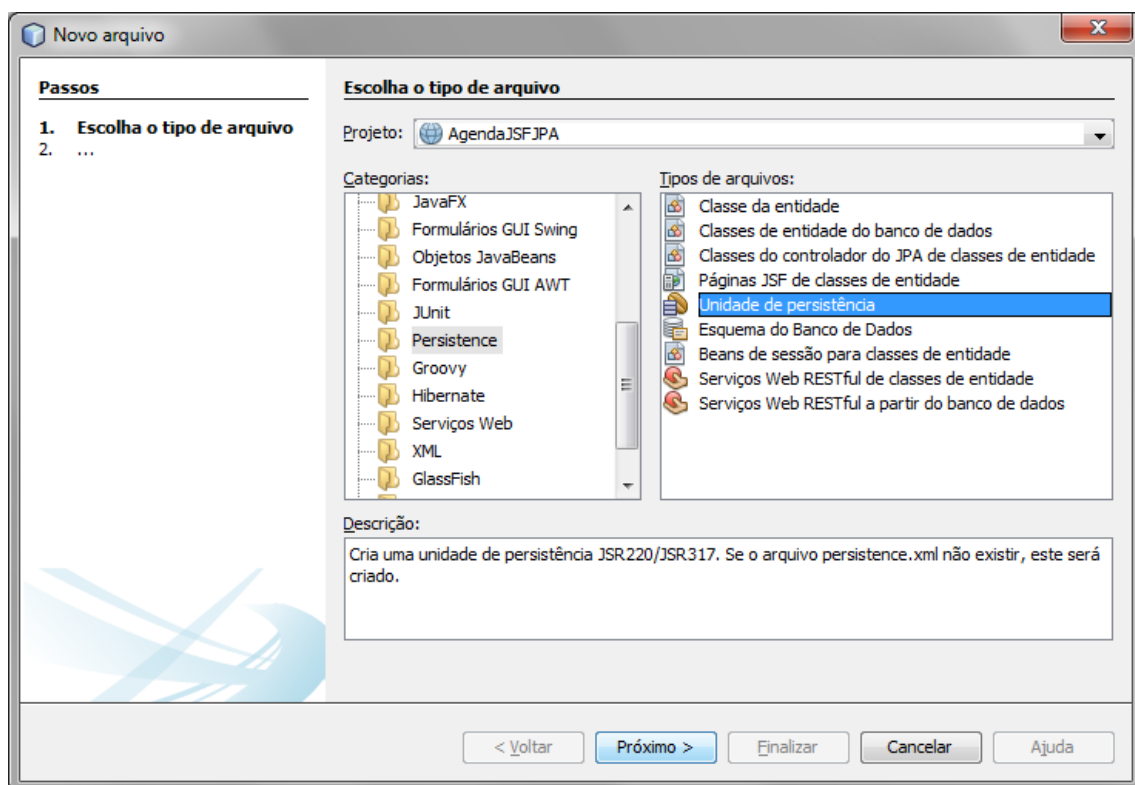
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="VideolocadoraJPAPU"
        transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="hibernate.connection.username"
                value="root"/>
            <property name="hibernate.connection.driver_class"
                value="com.mysql.jdbc.Driver"/>
            <property name="hibernate.connection.password"
                value="senha"/>
            <property name="hibernate.connection.url"
                value="jdbc:mysql://localhost:3306/videolocadora"/>
        </properties>
    </persistence-unit>
</persistence>
```

```
value="jdbc:mysql://localhost:3306/locadora"
/>
<property name="hibernate.cache.provider_class"
value="org.hibernate.cache.NoCacheProvider"/>
</properties>
</persistence-unit>
</persistence>
```

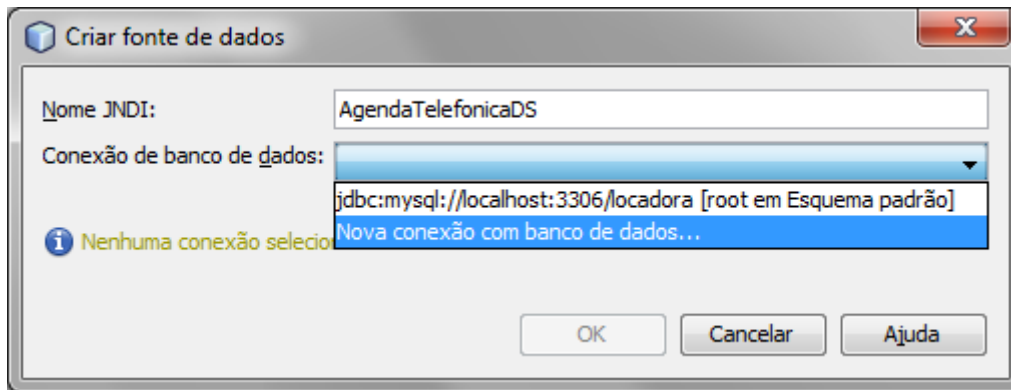
Para obter uma lista completa de propriedades de configuração do Hibernate visite <http://docs.jboss.org/hibernate/core/4.0/manual/en-US/html/session-configuration.html#configuration-hibernatejdbc>

Criando Uma Unidade de Persistência para Uma Aplicação Java EE

O processo de criação de uma unidade de persistência em uma aplicação Java EE é bastante semelhante ao mostrado anteriormente, com uma pequena diferença no momento em que se escolhe a maneira de se conectar ao banco de dados. Novamente você começa ao abrir o seu projeto e selecionar Arquivo > Novo Arquivo... para ver a tela exibida abaixo:

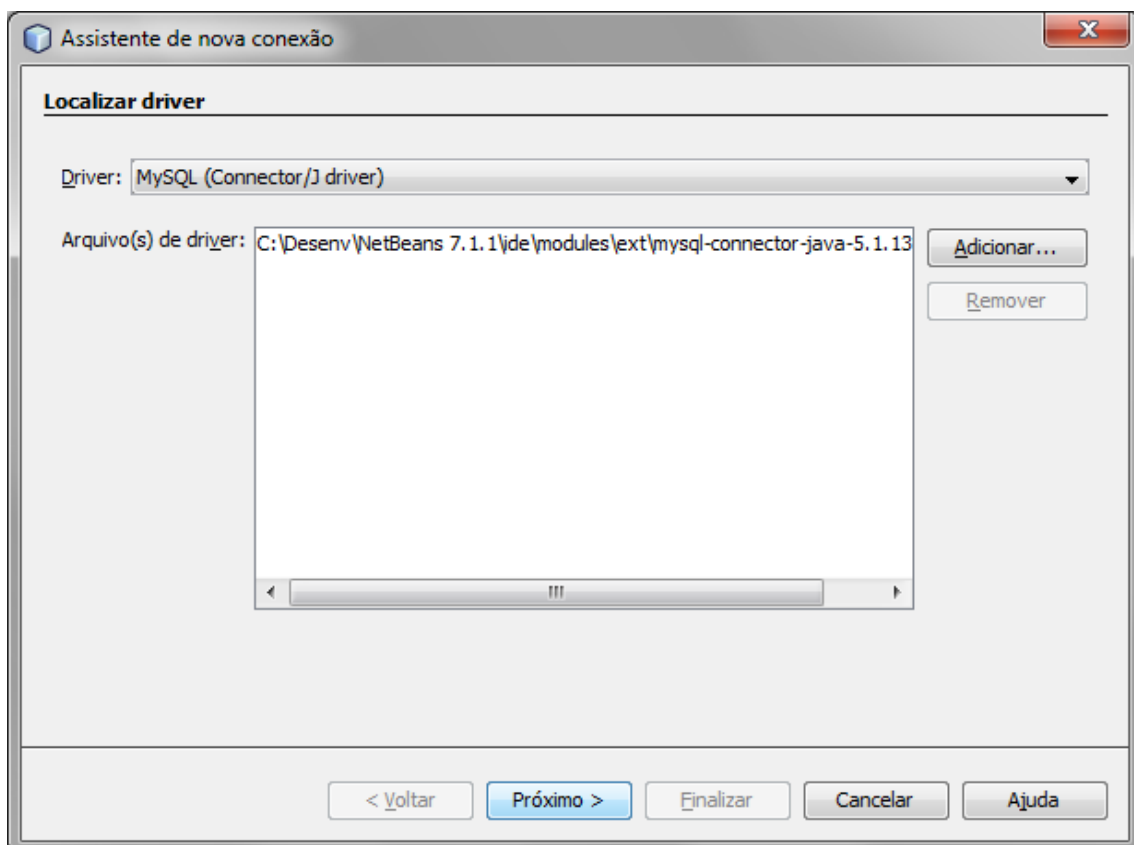


Selecione “Persistence” e “Unidade de persistência” para continuar, clicando no botão “Próximo” em seguida.



Aqui está a grande diferença. Ao invés de informar os dados de conexão diretamente, em aplicações Java EE você utilizará uma fonte de dados (“Data Source”, em inglês), uma conexão nomeada que lhe será entregue diretamente pelo container quando a sua aplicação Java EE for iniciada.

No nosso caso, escolha um nome JNDI para a sua fonte de dados e selecione “Nova conexão com banco de dados...” para continuar.



Voltamos para terreno familiar. No “Assistente de nova conexão”, escolha o driver do MySQL e clique em “Próximo”.

Assistente de nova conexão

Personalizar conexão

Nome do driver: MySQL (Connector/J driver)

Host: localhost Porta: 3306

Banco de dados: agendatelefonica

Nome do usuário: agenda

Senha:
☒ Lembrar senha

Testar conexão

URL JDBC: jdbc:mysql://localhost:3306/agendatelefonica

< Voltar Próximo > Finalizar Cancelar Ajuda

Use a tela acima para informar os dados da conexão, como nome do servidor, nome do banco de dados, usuário e senha. Clique no botão “Testar conexão” para ver se tudo funciona como esperado.

Assistente de nova conexão

Personalizar conexão

Nome do driver: MySQL (Connector/J driver)

Host: localhost Porta: 3306

Banco de dados: agendatelefonica

Nome do usuário: agenda

Senha: ●●●●●●

☒ Lembrar senha

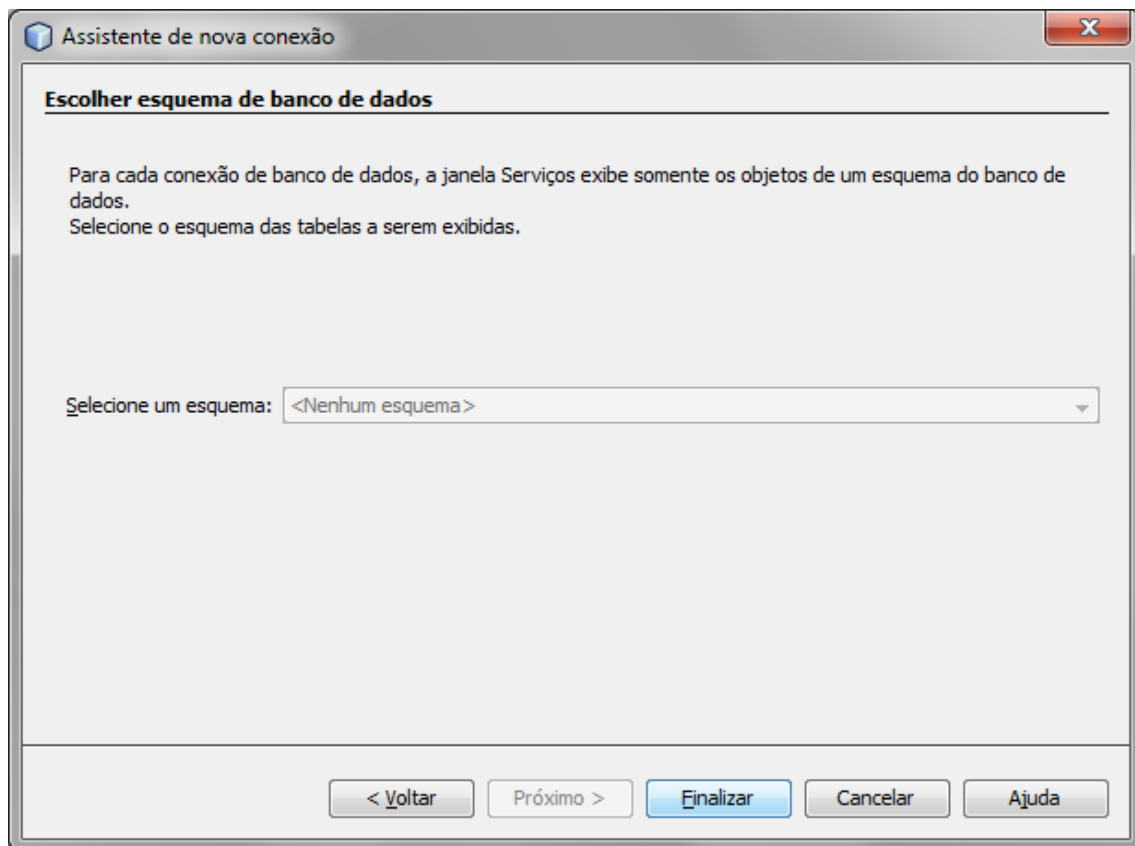
Testar conexão

URL JDBC: jdbc:mysql://localhost:3306/agendatelefonica

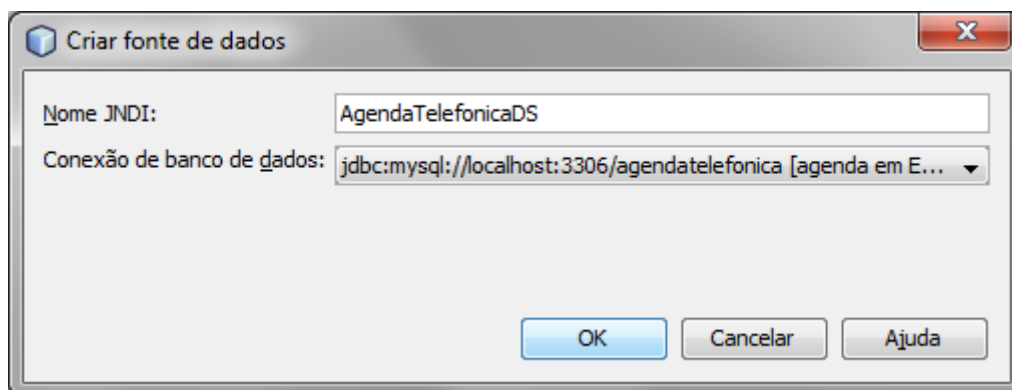
i Conexão bem-sucedida.

< Voltar Próximo > Finalizar Cancelar Ajuda

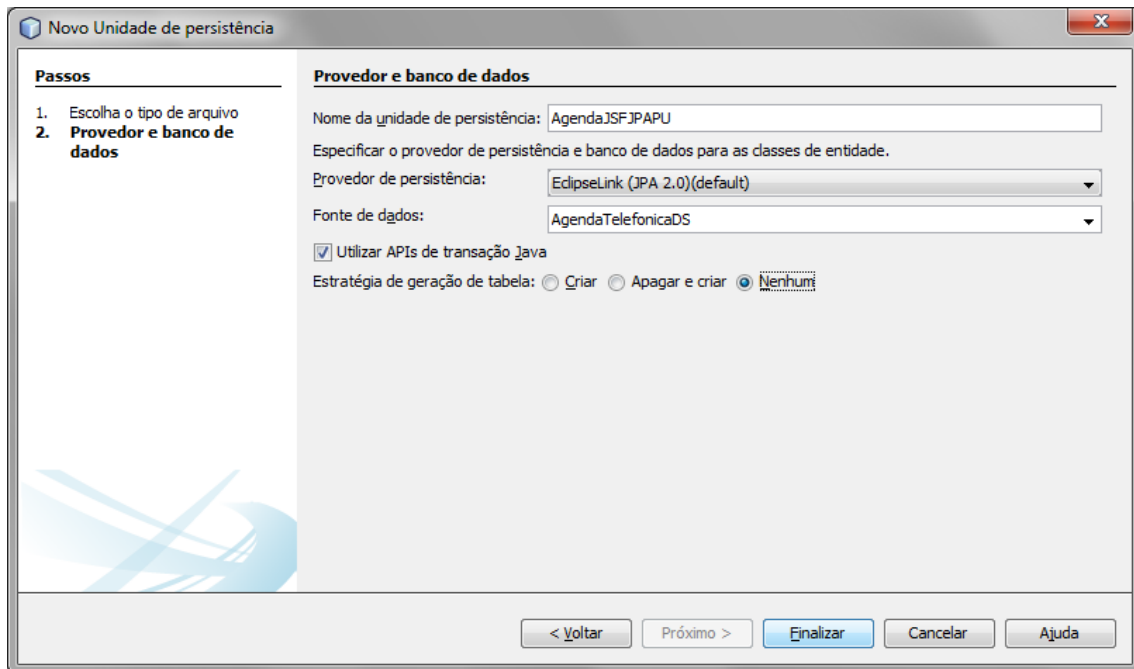
Se a conexão efetivamente funcionar você verá uma mensagem. Clique em “Próximo” para continuar.



Mais uma vez basta clicar em “Finalizar”.

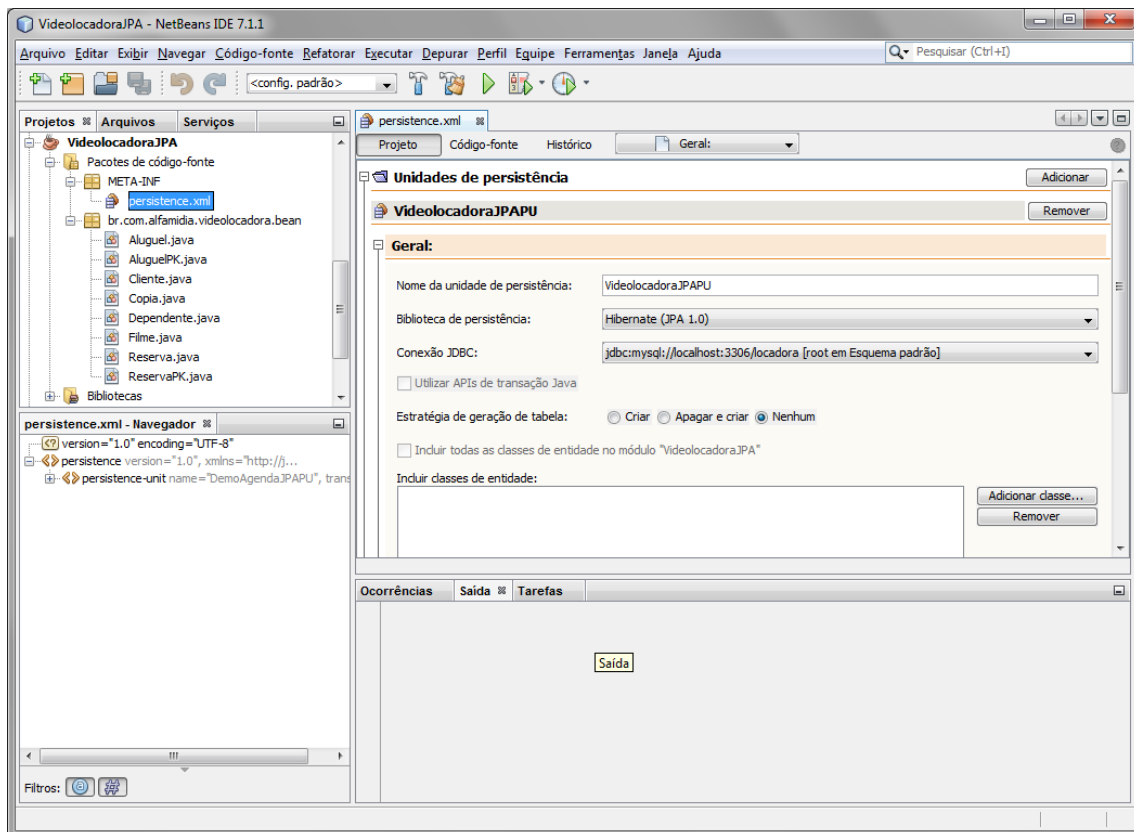


De volta à janela “Criar fonte de dados”, agora basta clicar em “OK” para encerrá-la.

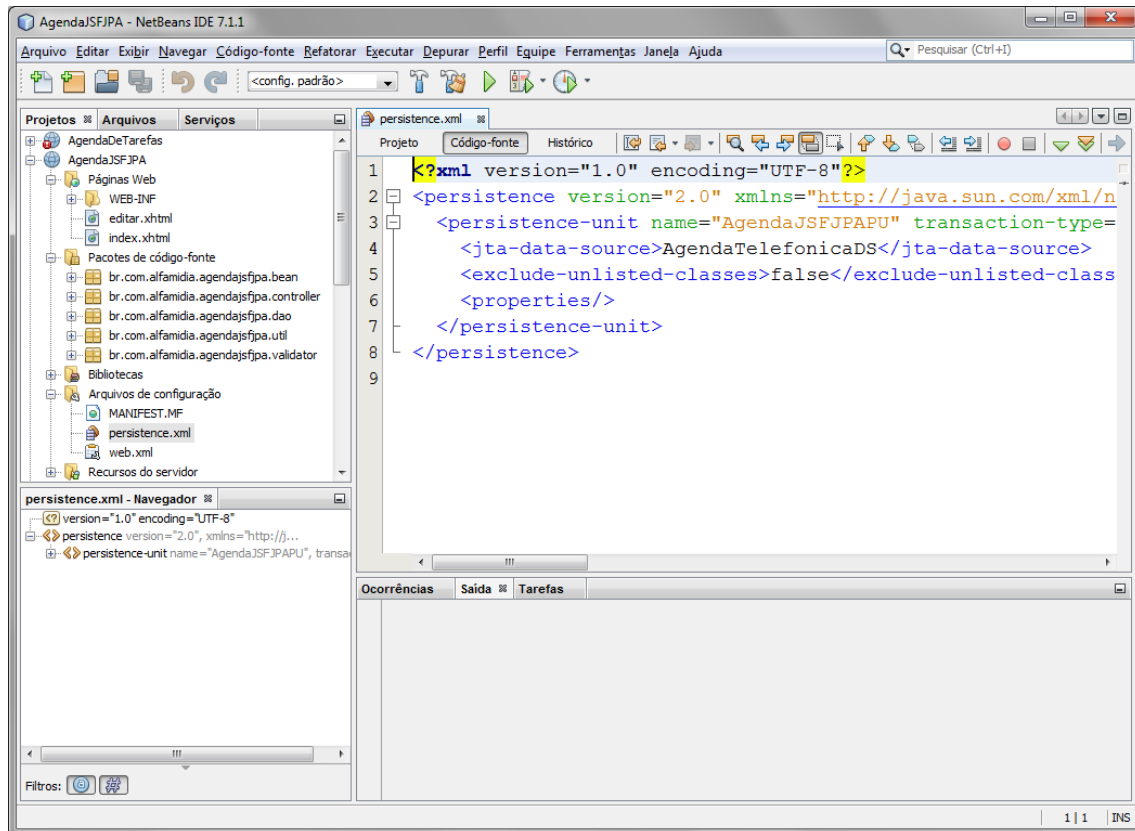


De volta à janela “Nova Unidade de persistência”, selecione “Nenhum” no campo “Estratégia de geração de tabelas” para evita que seu banco de dados seja apagado e recriado toda vez que você fizer o deployment de sua aplicação.

Clique em “Finalizar” para encerrar o assistente.



Agora o arquivo `persistence.xml` é aberto para você, no modo de edição de propriedades. Se quiser, clique na aba “Código-fonte” para ver o arquivo XML propriamente dito.



Note que desta vez o arquivo parece mais simples. Isso ocorre porque as informações de conexão não se fazem necessárias, basta indicar o nome da fonte de dados e pronto.

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="AgendaJSFJPAPU" transaction-
type="JTA">
    <jta-data-source>AgendaTelefonicaDS</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties/>
  </persistence-unit>
</persistence>
```

Capítulo 5 – Gerenciando Entidades com o EntityManager

Entidades, na especificação JPA, são objetos Java comuns: você as aloca usando o operador `new()` como faria com qualquer outra classe. Instâncias dos beans de entidade não se tornam persistentes até que sejam conectadas a um `EntityManager`. Vejamos novamente a nossa entidade `Pessoa`:

Pessoa.java

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

@Entity
public class Pessoa implements Serializable {

    @Id
    @GeneratedValue
    private int id;

    private String nome;

    public int getId() {
        return id;
    }

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

    public String getNome() {
        return nome;
    }

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

Quando você instancia um objeto do tipo `Pessoa` nenhuma mágica acontece para salvar este objeto no banco. Estes objetos permanecem classes Java comuns até que você utilize o `EntityManager` para sincronizar a entidade com o banco de dados.

Entidades Gerenciadas e Não-Gerenciadas

Para que possamos compreender mais profundamente os serviços de gerenciamento de entidades, primeiro precisamos conhecer o ciclo de vida das entidades. Um bean de entidade pode estar conectado ou não conectado a um gerenciador de entidades (em outras palavras, os beans podem ser gerenciados ou não-gerenciados). Quando uma entidade está conectada a um `EntityManager`, o gerenciador rastreia as mudanças de estado na entidade e as sincroniza com o banco de dados.

Quando uma entidade está desconectada, entretanto, isso não acontece. As modificações efetuadas à entidade não são rastreadas pelo `EntityManager`, então estas alterações não são propagadas até o banco de dados.

Trabalhar com uma entidade desconectada pode parecer um contrassenso, mas não é. Em diversas situações, como por exemplo quando você tem múltiplas alterações sucessivas ou uma conexão ruim até o banco de dados, pode ser que seja mais produtivo trabalhar desconectado e só sincronizar os dados com o banco de dados quando pertinente. E saber que a API é capaz de fazer a sincronização automaticamente deve servir para aplacar suas dúvidas.

O Ciclo de Vida de uma Entidade

Cada instância de uma entidade pode estar em um dos quatro estados mostrados abaixo:

- **Nova:** a instância é nova e ainda não foi associada a um contexto de persistência;
- **Gerenciada:** a instância de entidade está associada a um contexto de persistência;
- **Desconectada:** a instância tem uma identidade de persistência, mas não está mais associada a um contexto de persistência;
- **Removida:** a instância tem uma identidade de persistência e está associada a um contexto de persistência, mas foi agendada para remoção.

Classes Utilizadas no Gerenciamento de Entidades

Algumas classes estão envolvidas no processo de gerenciamento dos seus beans. Veja as principais abaixo:

Classe	Descrição
<code>javax.persistence.Persistence</code>	Cria uma <code>EntityManagerFactory</code> a partir do nome de uma unidade de persistência
<code>javax.persistence.EntityManagerFactory</code>	Usada para se obter instâncias de <code>EntityManager</code> .
<code>javax.persistence.EntityManager</code>	Usada para gerenciar a persistência de seus beans.
<code>javax.persistence.EntityTransaction</code>	Usada para iniciar e encerrar transações.

Obtendo o EntityManager

O `EntityManager` é o serviço central para todas as ações de persistência, provendo mecanismos para criar, alterar, excluir e buscar por beans de entidade.

Para isso você primeiro precisa obter acesso ao `EntityManager`. Em aplicações Java SE, instâncias de `javax.persistence.EntityManager` são obtidas por meio da classe `javax.persistence.EntityManagerFactory`. Embora você possa usar o mesmo approach em aplicações Java EE, esta plataforma oferece alguns serviços adicionais que reduzem o código necessário para se obter um `EntityManager`.

Obtendo o EntityManager com o Java SE

Em Java SE precisamos explicitamente recuperar o `EntityManagerFactory` e, a partir dele, criarmos um `EntityManager`. Veja o exemplo a seguir:

TesteJPAJavaSE.java

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

public class TesteJPAJavaSE {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("PU");
        EntityManager em = emf.createEntityManager();
    }
}
```

Obtendo o EntityManager com o Java EE

Já em Java EE nossa vida é facilitada pelo container de injeção de dependência, que automaticamente nos disponibiliza o `EntityManager` quando encontra a anotação `@PersistenceContext`. Veja o exemplo:

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

public class TesteJPAJavaEE {

    @PersistenceContext(unitName="PU")
    private EntityManager em;
    public void fazAlgo() {
        // Aqui, o objeto "em" está inicializado e pronto.
    }
}
```

Operações Básicas com Dados

Na tabela abaixo mostramos aqueles que são os principais métodos da classe `EntityManager`.

Método	Descrição
<code>void clear()</code>	Limpa o contexto de persistência, desconectando todas as entidades.

<code>void close()</code>	Encerra o EntityManager.
<code>boolean contains(Object entity)</code>	Verifica se o objeto passado como parâmetro é um bean gerenciado associado ao contexto de persistência atual.
<code>void detach(Object entity)</code>	Libera a entidade passada como parâmetro do contexto de persistência atual, fazendo com que ela mude para o estado desconectado.
<code><T> T find(Class<T> entityClass, Object primaryKey)</code>	Recupera um objeto por meio da sua chave primária.
<code>void flush()</code>	Sincroniza o contexto de persistência com o banco de dados subjacente.
<code><T> T getReference(Class<T> entityClass, Object primaryKey)</code>	Retorna uma instância, cujo estado pode ser recuperado de modo preguiçoso.
<code>EntityTransaction getTransaction()</code>	Retorna o objeto de transação.
<code>boolean isOpen()</code>	Determina se o gerenciador de entidades está aberto.
<code>void joinTransaction()</code>	Indica ao gerenciador de entidades que uma transação JTA já está ativa.
<code><T> T merge(T entity)</code>	Mescla o estado da entidade passada como parâmetro com o contexto de persistência.
<code>void persist(Object entity)</code>	Faz com que o objeto passado como parâmetro mude para o estado gerenciado.
<code>void refresh(Object entity)</code>	Atualiza o estado da instância a partir do banco de dados, sobrescrevendo qualquer alteração local.
<code>void remove(Object entity)</code>	Remove a instância da entidade passada como parâmetro, causando sua exclusão no banco de dados.

Criando Uma Entidade

Criar entidades é tão fácil quanto instanciar um novo objeto.

Mas no contexto de JPA, quando você fala de “criar uma entidade” está provavelmente se referenciando ao ato de inserir um registro no banco de dados a partir do seu bean.

Bom, isso é só um pouquinho mais complicado, conforme você pode ver no exemplo abaixo:

TesteInclusao.java

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

public class TesteInclusao {

    public static void main(String[] args) {
```



```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("PU");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();

Pessoa pessoa = new Pessoa();
Pessoa.setNome("João da Silveira");

em.persist(pessoa);

tx.commit();

System.out.println("Pessoa incluída com sucesso.");

em.close();
emf.close();

System.out.println("JPA encerrada com sucesso.");
}
}
```

Note que no exemplo obtivemos uma instância do `EntityManagerFactory` por meio a uma chamada a `Persistence.createEntityManagerFactory()`, passando como parâmetro o nome da nossa Persistence Unit.

Com o factory na mão podemos obter o `EntityManager`, e a partir dele criamos um objeto do tipo `EntityTransaction`, o qual pode ser usado para controlar as nossas transações.

Depois é só instanciar o bean e persistí-lo por meio da chamada a `EntityManager.persist()`, passando o bean como parâmetro. A partir deste momento o bean passa a ser um bean gerenciado.

Alterando uma Entidade

O mais importante que se precisa perceber é que, uma vez que o bean está no estado gerenciado, toda e qualquer alteração que ele sofrer vai ser automaticamente propagada para o banco de dados. Em resumo, você não precisa chamar nenhum método explicitamente para fazer com que o banco de dados seja atualizado. Veja o exemplo:

TesteAlteracao.java

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

public class TesteAlteracao {

    public static void main(String[] args) {

        EntityManagerFactory emf =
```

```

        Persistence.createEntityManagerFactory("PU");
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

        tx.begin();
        Pessoa pessoa = new Pessoa();
        pessoa.setNome("Marcelo Pereira Oliveira");
        em.persist(pessoa);
        tx.commit();

        System.out.println("Pessoa incluída com sucesso.");

        tx.begin();
        pessoa.setNome("Francisco Goulart");
        pessoa.setNome("Pedro Valério");
        tx.commit();

        System.out.println("Pessoa alterada com sucesso.");

        em.close();
        emf.close();

        System.out.println("JPA encerrada com sucesso.");
    }
}

```

Excluindo uma Entidade

Para excluir uma entidade basta chamar o método `EntityManager.remove()`, passando a entidade como parâmetro. Veja o exemplo:

TesteExclusao.java

```

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

public class TesteExclusao {

    public static void main(String[] args) {

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("PU");
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

        tx.begin();
        Pessoa pessoa = new Pessoa();
        pessoa.setNome("Pessoa a ser excluída");
        em.persist(pessoa);
        tx.commit();

        System.out.println("Pessoa incluída com sucesso, com id "

```

```
        + pessoa.getId());

    tx.begin();
    em.remove(pessoa);
    tx.commit();

    System.out.println("Pessoa excluída com sucesso");

    em.close();
    emf.close();

    System.out.println("JPA encerrada com sucesso.");
}
}
```

Recuperando uma Entidade

Para buscarmos por um registro no banco basta que usemos o método `EntityManager.find()`, passando dois argumentos:

A classe que serve como base para o bean que se deseja recuperar;

A chave primária do bean que se deseja recuperar.

Veja um exemplo:

TesteBusca.java

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

public class TesteBusca {

    public static void main(String[] args) {

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("PU");
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

        tx.begin();
        Pessoa pessoa = new Pessoa();
        pessoa.setNome("Bernardo Clementino");
        em.persist(pessoa);
        tx.commit();

        System.out.println("Pessoa incluída com sucesso, com id "
            + pessoa.getId());
        int id = pessoa.getId();

        Pessoa outraPessoa = em.find(Pessoa.class, id);
        System.out.println("Pessoa recuperada: "
            + outraPessoa.getNome());
    }
}
```

```
        em.close();
        emf.close();

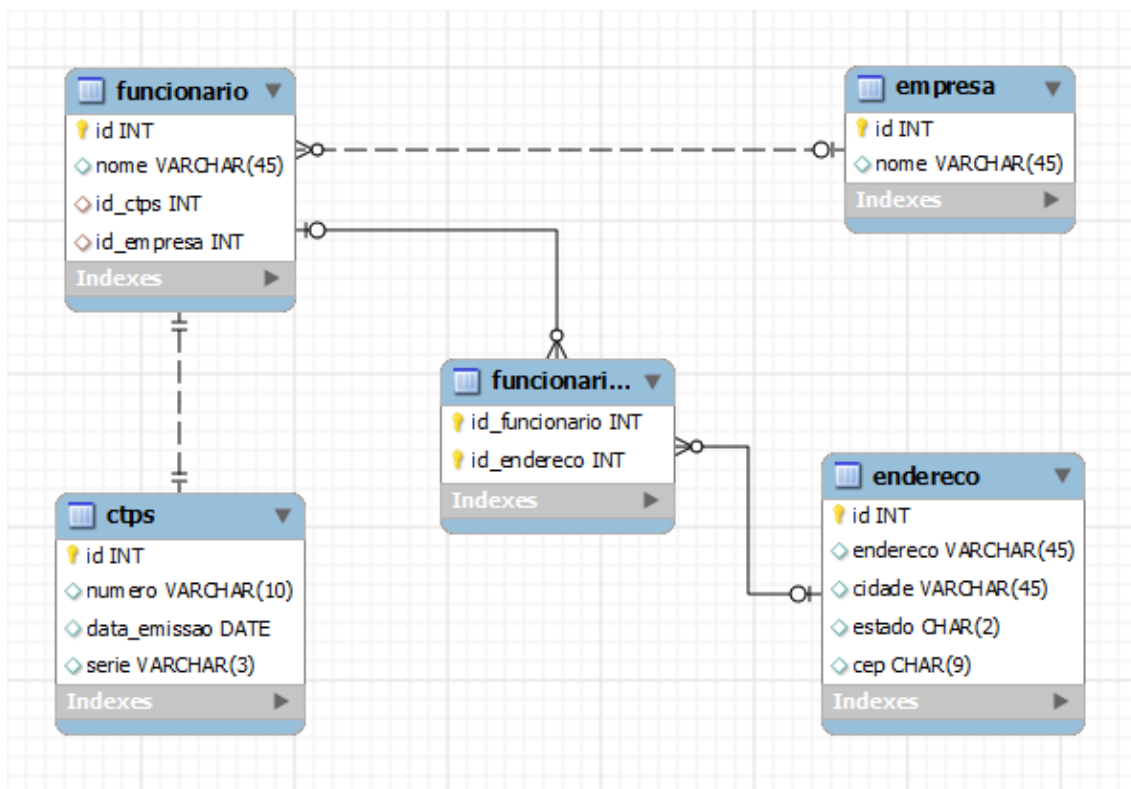
        System.out.println("JPA encerrada com sucesso.");
    }
}
```

Deixaremos as buscas mais complexas para mais tarde, primeiro vamos ver como se mapeiam relacionamentos.

Capítulo 6 – Mapeando Relacionamentos

Evidentemente que mapear entidades isoladas é mais fácil que mapear relacionamentos, e justamente por isso adiamos este capítulo até agora.

Vamos começar explorando nosso banco de dados de Empresa e ver que tipo de relacionamentos podemos encontrar.



Relacionamentos Um para Um

Se você examinar o nosso modelo de perto vai perceber que temos um relacionamento de um para um entre `funcionario` e `ctps`.

Este tipo de relacionamento é marcado com a anotação `@OneToOne`, e pode ser uni ou bi-direcional (conforme a sua necessidade de recuperar o dado inverso a partir de qualquer entidade). Veja o exemplo:

Funcionario.java

```
package br.com.alfamidia.jpa.bean;
```

```
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.JoinColumn;
import javax.persistence.OneToOne;

@Entity
public class Funcionario implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String nome;

    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="id_ctps")
    private CTPS ctps;

    // Demais relacionamentos não mostrados.

    // Getters e setters não mostrados.
}
```

CTPS.java

```
package br.com.alfamidia.jpa.bean;

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.OneToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class CTPS implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String numero;

    @Temporal(TemporalType.DATE)
    @Column(name="data_emissao")
    private Date dataEmissao;

    private String serie;
}
```

```
@OneToOne(mappedBy="ctps")
private Funcionario funcionario;

// Getters e setters não mostrados.
}
```

A Anotação @OneToOne

A anotação `@OneToOne` (`javax.persistence.OneToOne`) define um relacionamento de um para um com outra entidade. Geralmente não é necessário indicar explicitamente o tipo de dados da entidade alvo, uma vez que ele pode ser inferido a partir do tipo de dados do campo ou propriedade sendo mapeado.

Se o relacionamento é bi-direcional, o lado fraco precisa utilizar o elemento `mappedBy` da anotação `@OneToOne` para apontar para o campo ou propriedade que mapeou a relação na entidade forte.

Atributo	Descrição
cascade	Indica as operações que devem ser propagadas até a entidade alvo, conforme definidas no Enum <code>javax.persistence.CascadeType</code> <ul style="list-style-type: none"> • ALL: propaga todas as operações; • DETACH: propaga a operação de desconexão; • MERGE: propaga a operação de mescla; • PERSIST: propaga a operação de persistência; • REFRESH: propaga a operação de sincronização; • REMOVE: propaga a operação de remoção.
fetch	Define a estratégia de recuperação de dados a partir do banco de dados, conforme definidas no Enum <code>javax.persistence.FetchType</code> : <ul style="list-style-type: none"> • EAGER: os dados devem ser recuperados de maneira ansiosa; • LAZY: os dados devem ser recuperados de maneira preguiçosa.
mappedBy	Indica o nome do campo ou propriedade que definiu este relacionamento. Este elemento só deve ser utilizado no lado fraco do relacionamento.
optional	Determina se este relacionamento é opcional ou não.
orphanRemoval	Indica se o <code>EntityManager</code> deve desencadear uma operação de remoção para entidades que foram removidas de um relacionamento.

Atributo	Descrição
targetEntity	Determina a classe de entidade que é alvo deste relacionamento.

A Anotação @JoinColumn

A anotação @JoinColumn (javax.persistence.JoinColumn) pode ser usada para identificar a coluna de conexão de um relacionamento. Você pode usar esta anotação para indicar que uma Foreign Key não usa o nome default calculado pelo EntityManager (que é sempre o nome da tabela alvo mais o sufixo “_ID”).

Atributo	Descrição
columnDefinition	Define o fragmento de SQL que poderia ser utilizado para gerar o DDL da coluna.
insertable	Determina se a coluna deve aparecer nas instruções INSERT SQL geradas pelo provedor de persistência.
name	Indica o nome da coluna de FK.
nullable	Indica se a coluna de FK pode receber valores nulos ou não.
referencedColumnName	Indica o nome da coluna referenciada por esta coluna de FK.
table	Determina o nome da tabela que contém a coluna.
unique	Indica se esta coluna é uma chave única ou não.
updatable	Determina se esta coluna deve aparecer nas instruções UPDATE SQL geradas pelo provedor de persistência.

Relacionamentos Um para Muitos e Muitos para Um

No nosso banco de dados de exemplo temos um relacionamento de 1 para muitos entre as tabelas empresa e empregado.

Este tipo de relacionamento é marcado com as anotações @OneToMany e @ManyToOne, e pode ser uni ou bi-direcional (conforme a sua necessidade de recuperar o dado inverso a partir de qualquer entidade). Veja o exemplo:

Funcionario.java

```
package br.com.alfamidia.jpa.bean;

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



```
import javax.persistence.ManyToOne;

@Entity
public class Funcionario implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String nome;

    @ManyToOne(optional=false)
    @JoinColumn(name="id_empresa")
    private Empresa empresa;

    // Demais relacionamentos não mostrados.

    // Getters e setters não mostrados.
}
```

Empresa.java

```
package br.com.alfamidia.jpa.bean;

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

@Entity
public class Empresa implements Serializable {

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

    private String nome;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="id_empresa")
    private Collection<Funcionario> funcionarios =
        new ArrayList<>();

    // Getters e setters não mostrados.
}
```

A Anotação @OneToMany

A anotação `@OneToMany` (`javax.persistence.OneToMany`) define uma associação multivalorada com cardinalidade um para muitos. Geralmente não é necessário indicar explicitamente o tipo de dados da entidade alvo, uma vez que ele pode ser inferido a partir do tipo de dados do campo ou propriedade sendo mapeado, desde que você tenha usado Generics.

Atributo	Descrição
cascade	Indica as operações que devem ser propagadas até a entidade alvo, conforme definidas no Enum <code>javax.persistence.CascadeType</code> <ul style="list-style-type: none"> • ALL: propaga todas as operações; • DETACH: propaga a operação de desconexão; • MERGE: propaga a operação de mescla; • PERSIST: propaga a operação de persistência; • REFRESH: propaga a operação de sincronização; • REMOVE: propaga a operação de remoção.
fetch	Define a estratégia de recuperação de dados a partir do banco de dados, conforme definidas no Enum <code>javax.persistence.FetchType</code> : <ul style="list-style-type: none"> • EAGER: os dados devem ser recuperados de maneira ansiosa; • LAZY: os dados devem ser recuperados de maneira preguiçosa.
mappedBy	Indica o nome do campo ou propriedade que definiu este relacionamento. Este elemento só deve ser utilizado no lado fraco do relacionamento.
orphanRemoval	Indica se o <code>EntityManager</code> deve desencadear uma operação de remoção para entidades que foram removidas de um relacionamento.
targetEntity	Determina a classe de entidade que é alvo deste relacionamento.

A Anotação @ManyToOne

A anotação `@ManyToOne` (`javax.persistence.ManyToOne`) define uma associação de valor simples para uma entidade que tem cardinalidade muitos para um. Geralmente não é necessário indicar explicitamente o tipo de dados da entidade alvo, uma vez que ele pode ser inferido a partir do tipo de dados do campo ou propriedade sendo mapeado, desde que você tenha usado Generics.

Atributo	Descrição
cascade	<p>Indica as operações que devem ser propagadas até a entidade alvo, conforme definidas no Enum <code>javax.persistence.CascadeType</code></p> <ul style="list-style-type: none"> • ALL: propaga todas as operações; • DETACH: propaga a operação de desconexão; • MERGE: propaga a operação de mescla; • PERSIST: propaga a operação de persistência; • REFRESH: propaga a operação de sincronização; • REMOVE: propaga a operação de remoção.
fetch	<p>Define a estratégia de recuperação de dados a partir do banco de dados, conforme definidas no Enum <code>javax.persistence.FetchType</code>:</p> <ul style="list-style-type: none"> • EAGER: os dados devem ser recuperados de maneira ansiosa; • LAZY: os dados devem ser recuperados de maneira preguiçosa.
optional	Determina se este relacionamento é opcional ou não.
targetEntity	Determina a classe de entidade que é alvo deste relacionamento.

Relacionamentos Muitos para Muitos

No nosso banco de dados de exemplo temos uma relação de muitos para muitos entre as tabelas `funcionario` e `endereco`. Note que este tipo de relacionamento exige uma “tabela de junção”, que serve como união entre os dois lados da relação. Esta tabela não exige uma entidade, uma vez que seus atributos são meramente as chaves-primárias das entidades principais. Veja o exemplo:

Funcionario.java

```
package br.com.alfamidia.jpa.bean;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collection;
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;
```

```
@Entity
public class Funcionario implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String nome;

    @ManyToMany(cascade={CascadeType.ALL})
    @JoinTable(name="funcionario_endereco",
joinColumns=@JoinColumn(name="id_funcionario"),
inverseJoinColumns=@JoinColumn(name="id_endereco"))
    private Collection<Endereco> enderecos = new ArrayList<>();

    // Demais relacionamentos não mostrados.

    // Getters e setters não mostrados.
}
```

Endereco.java

```
package br.com.alfamidia.jpa.bean;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collection;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Endereco implements Serializable {

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

    private String endereco;

    private String cidade;

    private String estado;

    private String cep;

    @ManyToMany(mappedBy="enderecos")
    private Collection<Funcionario> residentes =
        new ArrayList<>();

    // Getters e setters não mostrados.
}
```

A Anotação @ManyToMany

A anotação `@ManyToMany` (`javax.persistence.ManyToMany`) define uma associação multivalorada com cardinalidade muitos para muitos.

Toda associação muitos para muitos tem dois lados, o lado forte e o fraco. A tabela de junção deve ser definida no lado forte. Se a relação foi bidirecional, o lado fraco deve utilizar o elemento `mappedBy` da anotação `@ManyToMany` para especificar o campo ou propriedade de relação no lado forte.

Atributo	Descrição
<code>cascade</code>	Indica as operações que devem ser propagadas até a entidade alvo, conforme definidas no Enum <code>javax.persistence.CascadeType</code> <ul style="list-style-type: none"> • ALL: propaga todas as operações; • DETACH: propaga a operação de desconexão; • MERGE: propaga a operação de mescla; • PERSIST: propaga a operação de persistência; • REFRESH: propaga a operação de sincronização; • REMOVE: propaga a operação de remoção.
<code>fetch</code>	Define a estratégia de recuperação de dados a partir do banco de dados, conforme definidas no Enum <code>javax.persistence.FetchType</code> : <ul style="list-style-type: none"> • EAGER: os dados devem ser recuperados de maneira ansiosa; • LAZY: os dados devem ser recuperados de maneira preguiçosa.
<code>mappedBy</code>	Indica o nome do campo ou propriedade que definiu este relacionamento. Este elemento só deve ser utilizado no lado fraco do relacionamento.
<code>targetEntity</code>	Determina a classe de entidade que é alvo deste relacionamento.

A Anotação @JoinTable

A anotação `@JoinTable` (`javax.persistence.JoinTable`) é tipicamente utilizada em relacionamentos muitos para muitos ou em relacionamentos unidirecionais de um para muitos, ou então em relacionamentos um para um tanto uni quanto bi-direcionais.

Se a anotação `@JoinTable` for omitida, assume-se que o nome da tabela de junção é composto pela concatenação dos nomes das tabelas primárias (o lado forte primeiro), separados por um sublinha.

Atributo	Descrição
catalog	Determina o nome do catalog da tabela.
inverseJoinColumn	As colunas de chave estrangeira que se referem à chave primária da tabela do lado fraco do relacionamento.
joinColumns	As colunas de chave estrangeira que se referem à chave primária da tabela do lado forte do relacionamento.
name	O nome desta entidade, a ser utilizado em queries. Tipicamente a entidade tem o mesmo nome não-qualificado da classe, mas você pode definir um outro.
schema	Determina o nome do schema da tabela.
uniqueConstraints	Define as constraints de unique-key da tabela. Veja a anotação @UniqueContraint para mais detalhes.

Um Exemplo Completo

Veja abaixo o código completo das entidades mostradas anteriormente, para sua referência.

Funcionario.java

```
package br.com.alfamidia.jpa.bean;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collection;
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;
import javax.persistence.ManyToOne;
import javax.persistence.OneToOne;

@Entity
public class Funcionario implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String nome;

    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="id_ctps")
    private CTPS ctps;
```

```
@ManyToOne(optional=false)
@JoinColumn(name="id_empresa")
private Empresa empresa;

@ManyToMany(cascade={CascadeType.ALL})
@JoinTable(name="funcionario_endereco",
joinColumns=@JoinColumn(name="id_funcionario"),
inverseJoinColumns=@JoinColumn(name="id_endereco"))
private Collection<Endereco> enderecos = new ArrayList<>();

public int getId() {
    return id;
}

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

public String getNome() {
    return nome;
}

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

public CTPS getCtps() {
    return ctps;
}

public void setCtps(CTPS ctps) {
    this.ctps = ctps;
}

public Empresa getEmpresa() {
    return empresa;
}

public void setEmpresa(Empresa empresa) {
    if (this.empresa != empresa) {
        if (this.empresa != null) {
            this.empresa.removeFuncionario(this);
        }

        this.empresa = empresa;

        if (empresa != null) {
            empresa.addFuncionario(this);
        }
    }
}
```

```
public Collection<Endereco> getEnderecos() {
    return enderecos;
}

public void setEnderecos(Collection<Endereco> enderecos) {
    this.enderecos = enderecos;
}

public void addEndereco(Endereco endereco) {
    if (!this.enderecos.contains(endereco)) {
        this.enderecos.add(endereco);
        endereco.addResidente(this);
    }
}

public void removeEndereco(Endereco endereco) {
    if (this.enderecos.contains(endereco)) {
        this.enderecos.remove(endereco);
        endereco.removeResidente(this);
    }
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Funcionario other = (Funcionario) obj;
    if (this.id != other.id) {
        return false;
    }
    return true;
}

@Override
public int hashCode() {
    int hash = 7;
    hash = 11 * hash + this.id;
    return hash;
}

@Override
public String toString() {
    return "Funcionario{" + "id=" + id + ", nome=" + nome
        + ", ctps=" + ctps + ", empresa=" + empresa
        + ", enderecos=" + enderecos + '}';
}
}
```


CTPS.java

```
package br.com.alfamidia.jpa.bean;

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.OneToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class CTPS implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String numero;

    @Temporal(TemporalType.DATE)
    @Column(name="data_emissao")
    private Date dataEmissao;

    private String serie;

    @OneToOne(mappedBy="ctps")
    private Funcionario funcionario;

    public int getId() {
        return id;
    }

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

    public String getNumero() {
        return numero;
    }

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

    public String getSerie() {
        return serie;
    }

    public void setSerie(String serie) {
```

```

        this.serie = serie;
    }

    public Date getDataEmissao() {
        return dataEmissao;
    }

    public void setDataEmissao(Date dataEmissao) {
        this.dataEmissao = dataEmissao;
    }

    public Funcionario getFuncionario() {
        return funcionario;
    }

    public void setFuncionario(Funcionario funcionario) {
        this.funcionario = funcionario;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final CTPS other = (CTPS) obj;
        if (this.id != other.id) {
            return false;
        }
        return true;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 11 * hash + this.id;
        return hash;
    }

    @Override
    public String toString() {
        return "CTPS{" + "id=" + id + ", dataEmissao="
            + dataEmissao + ", serie=" + serie + '}';
    }
}

```

Empresa.java

```

package br.com.alfamidia.jpa.bean;

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

```

```
import java.util.Collection;
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;

@Entity
public class Empresa implements Serializable {

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

    private String nome;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="id_empresa")
    private Collection<Funcionario> funcionarios = new
    ArrayList<>();

    public int getId() {
        return id;
    }

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

    public String getNome() {
        return nome;
    }

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

    public Collection<Funcionario> getFuncionarios() {
        return funcionarios;
    }

    public void setFuncionarios(
        Collection<Funcionario> funcionarios) {
        this.funcionarios = funcionarios;
    }

    public void addFuncionario(Funcionario funcionario) {
        if (!this.funcionarios.contains(funcionario)) {
            this.funcionarios.add(funcionario);
            funcionario.setEmpresa(this);
        }
    }
}
```

```

    }

    public void removeFuncionario(Funcionario funcionario) {
        if (this.funcionarios.contains(funcionario)) {
            this.funcionarios.remove(funcionario);
            funcionario.setEmpresa(null);
        }
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Empresa other = (Empresa) obj;
        if (this.id != other.id) {
            return false;
        }
        return true;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 97 * hash + this.id;
        return hash;
    }

    @Override
    public String toString() {
        return "Empresa{" + "id=" + id + ", nome=" + nome + '}';
    }
}

```

Endereco.java

```

package br.com.alfamidia.jpa.bean;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collection;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Endereco implements Serializable {

    @Id

```

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

private String endereco;

private String cidade;

private String estado;

private String cep;

@ManyToMany(mappedBy="enderecos")
private Collection<Funcionario> residentes =
    new ArrayList<>();

public String getCep() {
    return cep;
}

public void setCep(String cep) {
    this.cep = cep;
}

public String getCidade() {
    return cidade;
}

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

public String getEndereco() {
    return endereco;
}

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

public String getEstado() {
    return estado;
}

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

public int getId() {
    return id;
}

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

```
}

public Collection<Funcionario> getResidentes() {
    return residentes;
}

public void setResidentes(Collection<Funcionario> residentes)
{
    this.residentes = residentes;
}

public void addResidente(Funcionario residente) {
    if (!this.residentes.contains(residente)) {
        this.residentes.add(residente);
        residente.addEndereco(this);
    }
}

public void removeResidente(Funcionario residente) {
    if (this.residentes.contains(residente)) {
        this.residentes.remove(residente);
        residente.removeEndereco(this);
    }
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Endereco other = (Endereco) obj;
    if (this.id != other.id) {
        return false;
    }
    return true;
}

@Override
public int hashCode() {
    int hash = 5;
    hash = 37 * hash + this.id;
    return hash;
}

@Override
public String toString() {
    return "Endereco{" + "id=" + id + ", endereco="
        + endereco + ", cidade=" + cidade + ", estado="
        + estado + ", cep=" + cep + '}';
}
```

```
}
```

Capítulo 7 – Fazendo Consultas

Dados não são muito úteis se não existe uma maneira rápida e conveniente de pesquisá-los e recuperá-los. JPA leva esta necessidade tão a sério que oferece duas técnicas para tal: JP QL e SQL.

JP QL (Java Persistence Query Language, anteriormente conhecida como EJB QL, de Enterprise JavaBeans Query Language) é uma linguagem de consulta declarativa bastante similar à SQL, mas melhor adaptada para lidar com objetos Java. Para fazer uma consulta, ao invés de mencionar tabelas e colunas, você menciona as propriedades e relacionamentos dos seus beans de entidade. Quando a query é executada o `EntityManager` usa a informação que você proveu através dos metadados de mapeamento para gerar as queries SQL necessárias para recuperar aqueles objetos. Este SQL nativo é executado e o `EntityManager` toma para si a tarefa de decompor os resultados e gerar os objetos de maneira extremamente conveniente para você.

Algumas vezes, entretanto, a JP QL não é o suficiente. Já que ela precisa ser uma linguagem portátil, nem sempre consegue usar todas as features que seu banco de dados suporta. JP QL não suporta a execução de stored procedures, por exemplo, então para esse tipo de situação você vai ter que recorrer ao SQL diretamente, mas mesmo nestes casos o `EntityManager` vem em seu apoio, oferecendo as ferramentas para o mapeamento dos dados nativos para entity beans.

A Interface Query

Tanto as queries geradas com JP QL quanto aquelas geradas com SQL são executadas por meio da interface `javax.persistence.Query`, a qual funciona de uma maneira bastante parecida com a nossa velha conhecida `java.sql.PreparedStatement`, mas oferecendo ainda recursos avançados, como por exemplo a paginação de dados.

Método	Descrição
List getResultList()	Executa a query e retorna os seus resultados como um List sem tipo.
Object getSingleResult()	Executa a query e retorna um resultado único como um Object.
int executeUpdate()	Executa uma declaração de UPDATE ou DELETE, e retorna o número de entidades afetadas.
Query setMaxResults(int maxResult)	Indica o número máximo de resultados a serem retornados pela query.
Query setFirstResult(int startPosition)	Indica a posição do primeiro resultado a ser retornado.

Método	Descrição
Query setHint(String hintName, Object value)	Oferece uma maneira de utilizar recursos proprietários do JPA provider.
Query setParameter(String name, Object value)	Associa um valor a um parâmetro.
Query setParameter(String name, Date value, TemporalType temporalType)	Associa uma instância de java.util.Date a um parâmetro.
Query setParameter(String name, Calendar value, TemporalType temporalType)	Associa uma instância de java.util.Calendar a um parâmetro.
Query setParameter(int position, Object value)	Associa um valor a um parâmetro posicional.
Query setParameter(int position, Date value, TemporalType temporalType)	Associa uma instância de java.util.Date a um parâmetro posicional.
Query setParameter(int position, Calendar value, TemporalType temporalType)	Associa uma instância de java.util.Calendar a um parâmetro posicional.

Usando Queries

Instâncias de Query são obtidas por meio dos seguintes métodos de `javax.persistence.EntityManager`:

Método	Descrição
Query createQuery(String ejbqlString)	Cria uma instância de Query a partir de uma String que representa a consulta.
Query createNamedQuery(String name)	Cria uma instância de Query para executar uma consulta nomeada (tanto JP QL quanto SQL).
Query createNativeQuery(String sqlString)	Cria uma instância de Query para executar uma consulta SQL nativa, como por exemplo um UPDATE ou DELETE.
Query createNativeQuery(String sqlString, Class resultClass)	Cria uma instância de Query para executar uma consulta SQL nativa que retorna resultados do tipo indicado.
Query createNativeQuery(String sqlString, String resultSetMapping)	Cria uma instância de Query para executar uma consulta SQL nativa que retorna resultados de acordo com o mapeamento indicado.

Veja o exemplo abaixo:

TesteConsultaSimples.java

```
package br.com.alfamidia.demoagenda.jpa;

import br.com.alfamidia.demoagenda.jpa.bean.Pessoa;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
```

```
import javax.persistence.EntityNotFoundException;
import javax.persistence.NonUniqueResultException;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TesteConsultaSimples {

    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("PU");
        EntityManager em = emf.createEntityManager();

        try {
            Query query = em.createQuery("from Pessoa p "
                + "where p.nome = 'Marcelo Pereira Silveira'");
            Pessoa pessoa = (Pessoa) query.getSingleResult();
            System.out.println(pessoa);
        } catch (EntityNotFoundException enfe) {
            System.out.println("Incapaz de encontrar a entidade: "
                + enfe);
        } catch (NonUniqueResultException nure) {
            System.out.println("A query retornou mais de uma "
                + "entidade: " + nure);
        } finally {
            em.close();
            emf.close();
        }
    }
}
```

No exemplo acima pesquisamos por uma entidade individual (filtramos por nome e usamos). O método `getSingleResult()` espera que a query retorne uma única entidade. Se a query porventura retornar mais de uma entidade a exceção `javax.persistence.NonUniqueResultException` será lançada. Por outro lado, se a query não encontrar nenhuma entidade a exceção `javax.persistence.EntityNotFoundException`.

Alternativamente você poderia usar o método `getResultList()` para recuperar uma lista de resultados. Veja o exemplo abaixo:

TesteConsultaTotal.java

```
package br.com.alfamidia.demoagenda.jpa;

import br.com.alfamidia.demoagenda.jpa.bean.Pessoa;
import java.util.List;
import java.util.ListIterator;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TesteConsultaTotal {

    public static void main(String[] args) {
```

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

Query query = em.createQuery("from Pessoa p");
List pessoas = query.getResultList();
ListIterator pessoasIterator = pessoas.listIterator();
while (pessoasIterator.hasNext()) {
    Pessoa pessoa = (Pessoa) pessoasIterator.next();
    System.out.println(pessoa);
}
emf.close();
}
```

Utilizando Parâmetros

De maneira bastante parecida com o que acontece com `PreparedStatement`, a interface `Query` também suporta parâmetros. Mas diferente daquela, a interface JPA suporta parâmetros nomeados além dos parâmetros posicionais.

Na primeira forma você marca a sua query com o caractere `:`, que serve como prefixo para o nome do parâmetro. Na segunda forma, a posicional, você marca a sua query com `?`, de maneira bastante parecida com a que fazia com `PreparedStatement`.

Parâmetros Nomeados

Com parâmetros nomeados, você indica o nome dos parâmetros ao invés de meros marcadores, facilitando o processo de população dos mesmos mais tarde.

```
Query query = em.createQuery(
    "from Pessoa p where p.nome like :nome");
query.setParameter("nome", "%J%");
```

Veja o exemplo:

TesteConsultaParametros.java

```
package br.com.alfamidia.demoagenda.jpa;

import br.com.alfamidia.demoagenda.jpa.bean.Pessoa;
import java.util.List;
import java.util.ListIterator;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TesteConsultaParametros {

    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("PU");
        EntityManager em = emf.createEntityManager();

        Query query = em.createQuery(
```

```

        "from Pessoa p where p.nome like :nome");
query.setParameter("nome", "%J%");
List pessoas = query.getResultList();
ListIterator pessoasIterator = pessoas.listIterator();
while (pessoasIterator.hasNext()) {
    Pessoa pessoa = (Pessoa) pessoasIterator.next();
    System.out.println(pessoa);
}
emf.close();
}
}

```

Parâmetros Posicionais

Já os parâmetros posicionais funcionam de maneira bastante parecida com a que funcionavam na interface `PreparedStatement`: você marca seus parâmetros na query com sinais de interrogação (no caso de `Query` você coloca ainda um número depois da interrogação), e depois menciona a posição do parâmetro no momento em que quer populá-lo:

```

Query query = em.createQuery(
    "from Pessoa p where nome like ?1");
query.setParameter(1, "%J%");

```

Parâmetros do Tipo Data

Lidar com datas em Java sempre foi uma dor de cabeça: temos tantas classes semelhantes que sempre ficamos em dúvida sobre qual usar. Mas pior mesmo é quando temos que passar datas para outros componentes: como saber que tipo de data eles esperam.

Nos bancos de dados isso não é muito diferente: existem três grandes tipos de dados para data, `DATE`, `DATETIME` E `TIMESTAMP`, e alguns projetistas ainda usam datas como valores do tipo `long`, o que só aumenta a confusão.

Por isso, popular parâmetros do tipo data em queries JPA pode parecer meio assustador. Além da data propriamente dita ainda temos que indicar como o `EntityManager` deve lidar com a data na hora de enviá-la para o banco.

Veja abaixo os quatro métodos de `javax.persistence.Query` para lidar com parâmetros do tipo data:

Método	Descrição
<code>Query setParameter(String name, Date value, TemporalType temporalType)</code>	Associa uma instância de <code>java.util.Date</code> a um parâmetro.
<code>Query setParameter(String name, Calendar value, TemporalType temporalType)</code>	Associa uma instância de <code>java.util.Calendar</code> a um parâmetro.
<code>Query setParameter(int position, Date value, TemporalType temporalType)</code>	Associa uma instância de <code>java.util.Date</code> a um parâmetro posicional.
<code>Query setParameter(int position, Calendar value, TemporalType temporalType)</code>	Associa uma instância de <code>java.util.Calendar</code> a um parâmetro posicional.

Ou seja, no caso de datas, além de indicar o nome ou posição do parâmetro e o seu valor, você ainda precisa passar um terceiro parâmetro indicando o tipo temporal do parâmetro (conforme quer enviar data, hora ou timestamp). Para isso deve utilizar o Enum `javax.persistence.TemporalType`:

- DATE: envia apenas a data;
- TIME: envia apenas a hora;
- TIMESTAMP: envia data e hora.

Veja o exemplo abaixo:

```
Query query = em.createQuery(
    "from Funcionario f where f.dataContratacao = :data");
query.setParameter("data", new Date(), TemporalType.DATE);
```

Queries Nomeadas

Uma query nomeada é uma query definida estaticamente como uma anotação na classe de entidade. Sua utilização pode melhorar o desempenho e a segurança da sua aplicação, uma vez que a query é pré-compilada e, por isso mesmo, é capaz de evitar a injeção de SQL.

A Anotação @NamedQuery

A anotação @NamedQuery (javax.persistence.NamedQuery) pode ser utilizada para criar uma string JP QL associada a uma entidade. Esta query pode então ser recuperada e executada diretamente de outros contextos, se tornando reutilizável.

Atributo	Descrição
name	Indica o nome da query.
query	Determina o texto da query em JP QL.

Veja um exemplo:

Pessoa.java

```
package br.com.alfamidia.demoagenda.jpa.bean;

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

@Entity
@NamedQuery(name="Pessoa.achaTodas", query="from Pessoa p")
public class Pessoa implements Serializable {

    @Id
    @GeneratedValue
    @Column(name="id", nullable=false)
    private int id;

    // Demais atributos e métodos não mostrados.
}
```

TesteConsultaNomeada.java

```
package br.com.alfamidia.demoagenda.jpa.jpa;

import br.com.alfamidia.demoagenda.jpa.bean.Pessoa;
import java.util.List;
```

```
import java.util.ListIterator;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TesteConsultaNomeada {

    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("PU");
        EntityManager em = emf.createEntityManager();

        Query query = em.createNamedQuery("Pessoa.achaTodas");
        List pessoas = query.getResultList();
        ListIterator pessoasIterator = pessoas.listIterator();
        while (pessoasIterator.hasNext()) {
            Pessoa pessoa = (Pessoa) pessoasIterator.next();
            System.out.println(pessoa);
        }
        emf.close();
    }
}
```

A Anotação @NamedQueries

A anotação `@NamedQueries` (`javax.persistence.NamedQueries`) deve ser usada sempre que você deseja fornecer múltiplas queries em uma única entidade. Veja um exemplo abaixo:

Pessoa.java

```
package br.com.alfamidia.demoagenda.jpa.bean;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.NamedQuery;

@Entity
@NamedQueries({@NamedQuery(name="Pessoa.achaTodas",
                        query="from Pessoa p"),
               @NamedQuery(name="Pessoa.achaPorNome",
                        query="from Pessoa p where p.nome
like :nome")})
public class Pessoa implements Serializable {
    @Id
    @GeneratedValue
    @Column(name="id", nullable=false)
    private int id;

    // Demais atributos e métodos não mostrados.
}
```