



Desenvolvimento de Sistemas

ORM (mapeadores objeto-relacional): conceitos, alternativas e aplicabilidade

Introdução e conceitos

ORM (*object relational mapper*, ou “mapeamento objeto-relacional”, em português) é uma técnica que permite fazer um paralelo entre os objetos do paradigma da orientação a objetos da linguagem de programação e os dados armazenados no banco de dados, os quais esses objetos representam.

Essa técnica tornou-se cada vez mais popular nos últimos anos e isso se deve à praticidade de não ser necessário escrever código SQL (*structured query language*) focando-se nas definições dentro da estrutura das linguagens.

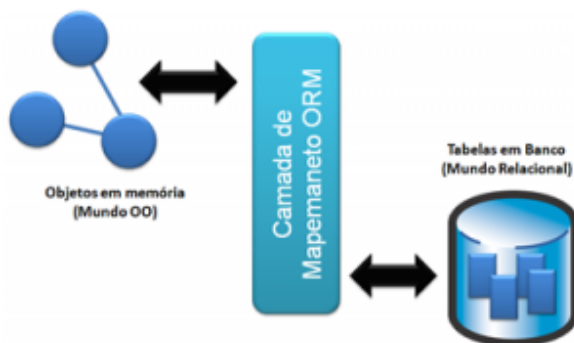


Figura 1 – Funcionamento de um ORM

Fonte: Cadu (2011)

Com o ORM constam três elementos:



Núcleo relacional

No núcleo relacional, a finalidade é armazenar e gerenciar corretamente os dados usando sistemas gerenciadores de bancos de dados e linguagens SQL e, às vezes, NoSQL. O princípio do núcleo relacional é informar para o sistema “o que”, ou seja, quais informações o sistema deve guardar e não “como” essas informações devem ser guardadas.

Núcleo orientado a objetos

No núcleo orientado a objetos estão as classes e os métodos fundamentados em engenharia de *software* e princípios da orientação a objetos. Em outras palavras, nesse núcleo é dito ao sistema “como” lidar com esses dados, enviando ou lendo e exibindo os dados para os usuários do sistema.

ORM

Entre esses dois núcleos está a camada do ORM, que possibilita o “armazenamento” de seus objetos de banco de dados por meio do mapeamento dos objetos.

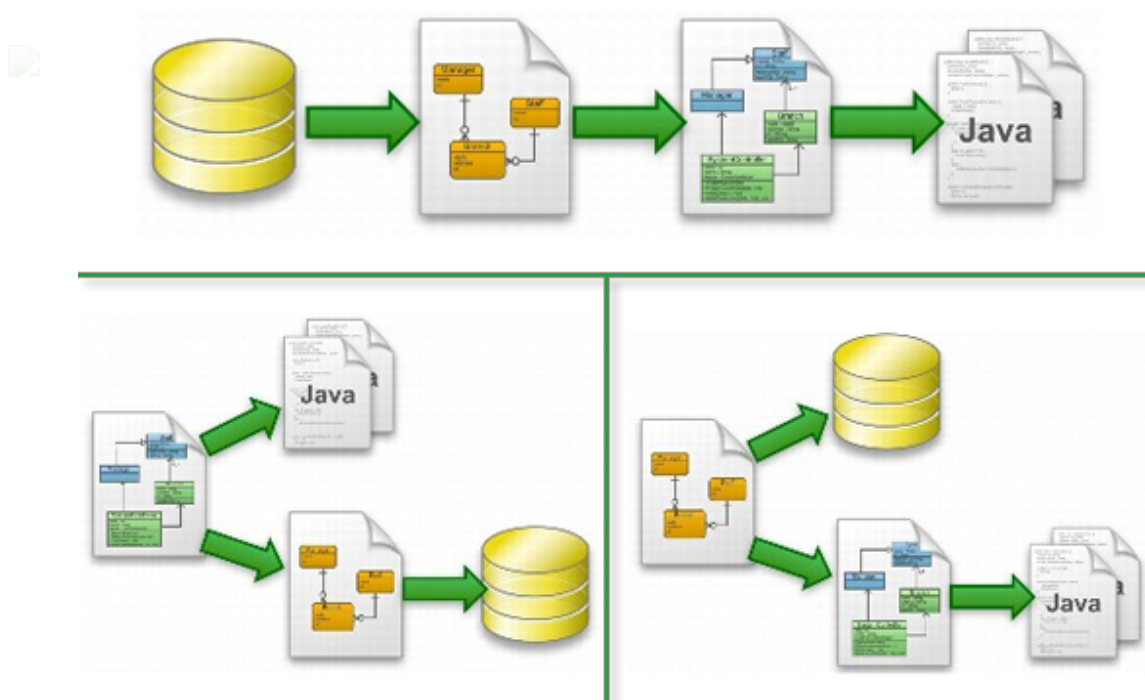


Figura 2 – Como um ORM trabalha

Fonte: Cadu (2011)

O sistema ORM faz o mapeamento da classe Java para o banco de dados criando uma tabela no banco, espelhando o modelo das classes Java, sem a necessidade de escrever e rodar manualmente os códigos SQL. Além de ganhar produtividade, você tem garantia de que os dados são compatíveis, evitando erros de tipos entre as duas linguagens e criando um tipo de validação inicial.

Em resumo, um ORM permite que você programe interação com o banco de dados sem uso de SQL e de manipulação direta do banco, podendo trabalhar diretamente com objetos em vez de trabalhar com várias informações separadas. Isso beneficia também a independência do sistema, pois, se for necessário trocar de SGBD (Sistema de Gerenciamento de Banco de Dados), como você está lidando com uma camada intermediária, essa troca poderá acontecer sem muitas mudanças no projeto Java, ao contrário do que poderia acontecer se você usasse apenas JDBC (Java Database Connectivity).

JPA



JPA (em inglês Java Persistence API, que significa “Interface de Programação de Aplicação de Persistência do Java” em português) é um *framework* genérico voltado à organização dos conceitos de ORM dentro de projetos Java.

A versão atual da JPA trouxe muitas melhorias em relação à versão anterior. Entre essas melhorias estão as extensões para as linguagens de consultas dinâmicas no banco de dados com Java, as chamadas Java Persistence Query Language (JPQL) ou Criteria.

JPQL

Java Persistence Query Language é uma maneira de fazer consultas em um banco de dados relacional utilizando Java. A vantagem da JPQL é que a sintaxe dela é muito semelhante à SQL.

Após uma consulta ao banco de dados com a JPQL, ele retornará um objeto/entidade, ao invés de um campo, como é na SQL convencional.

Criteria

Criteria é uma API (*application programming interface*) do Java desenvolvida para definir uma forma alternativa de criar consultas para entidades. Essas consultas são seguras, tipadas, portáteis e fáceis de modificar alterando sua sintaxe.

Outra vantagem da JPA foi a facilitação das configurações a partir de arquivos XML ou de anotações, ou os dois, o que simplifica bastante o processo, que pode iniciar rapidamente.

Segundo Melo (2020), do *site* Tecnoblog, XML é a sigla para “Extensible Markup Language”, uma linguagem de marcação com regras para formatar documentos de forma que eles sejam facilmente lidos tanto por humanos quanto

por máquinas. No Brasil, ficou popularizado por ser o tipo de arquivo mais utilizado na emissão de notas fiscais digitais.

Além disso, um dos maiores benefícios da JPA é a possibilidade de delegar para o *framework* a escrita do código SQL, que é diferente para cada SGBD. Isso significa que o código usado para “criar” o banco e as tabelas no MySQL será o mesmo no SQLServer, no Oracle, no SQLite, no PostgreSQL e em qualquer outro. Entretanto, se isso for resolvido com SQL puro (necessário no JDBC), existem diferenças sutis entre um e outro, que criam incompatibilidades que impedem a utilização do mesmo código. JPA, assim, é uma solução mais prática e mais inteligente.

Pode-se destacar, ainda, velocidade para começar e terminar um projeto, validação e garantia de compatibilidade, que são características importantes para aplicações de qualquer porte, afinal “tempo é dinheiro”.

Alternativas de implementações JPA (OpenJPA, Hibernate, e EclipseLink)

A JPA tem uma série de especificações que definem as regras gerais de como ela deve ser implementada em seus projetos. Você pode até mesmo criar suas próprias implementações sem prejuízo para nenhum projeto, mas isso é trabalhoso e demanda muito tempo para ser executado. É mais inteligente utilizar uma das implementações criadas dentro das especificações.

Se você usa no seu projeto uma implementação que está nas especificações, isso dá a você a liberdade de trocar essa implementação por outra sem precisar reescrever o projeto inteiro. Por isso, é importante conhecer as principais implementações JPA:

OpenJPA, da Apache Foundation

O OpenJPA é uma implementação criada pela fundação Apache, a qual mantém diversos projetos, como OpenJDK, OpenJRE, NetBeans, WebServer Apache e muitos outros.

Alguns pontos positivos do OpenJPA é o fato de já estar tudo “empacotado” em um arquivo **.jar**, o que reduz o número de dependências a instalar para o bom funcionamento da implementação. Por ser uma implementação simples e de rápida configuração, pode ser a escolha ideal para pequenos projetos e iniciantes em Java e em JPA. No entanto, talvez devido a essa característica, são poucas empresas e projetos grandes conhecidos.

Hibernate, da Red Hat

O Hibernate é um dos primeiros *frameworks* da implementação JPA, que surgiu juntamente com o próprio conceito. Foi criado por um grupo “voluntário” de desenvolvedores espalhados pelo mundo inteiro e foi rapidamente muito bem recebido pelos desenvolvedores Java, criando uma comunidade ativa e colaborativa. Os principais desenvolvedores foram contratados pela JBoss para trabalhar no projeto e dar suporte à ferramenta. A própria JBoss foi comprada pela Red Hat, que é conhecida pela distribuição Linux para servidores, além de serviços de suporte, treinamentos e nuvem de dados.

O Hibernate é uma das principais implementações JPA exatamente por ter uma adoção muito grande pela comunidade. Ele contém muita documentação, tutoriais e conteúdos, até mesmo livros, em português. Esse *framework* é utilizado em muitas empresas e em grandes projetos de *software*, principalmente aqueles voltados para *web*, sendo nativo em *frameworks* como o Spring e outros.

Uma desvantagem do Hibernate é que ele tem uma necessidade de muitas dependências, o que pode causar conflitos, principalmente após atualizações, e ainda dificultar as configurações iniciais.

EclipseLink, da Fundação Eclipse



O EclipseLink, além de implementar o padrão JPA, implementa outros padrões e vários tipos de serviços, como *web services*, XML, EIS, JAXB, JCA, SDO, entre outros. O EclipseLink foi implementado com base no TopLink, da Oracle, que liberou o código-fonte para outros projetos *open source*. Atualmente, o EclipseLink evoluiu bastante e é uma opção melhor que o TopLink.

Hoje, o EclipseLink é uma implementação de referência, ou seja, quando há alterações nas especificações da JPA, o EclipseLink é um dos primeiros a implementar mudanças e validar as novas ideias.

Metadados

JPA, sendo uma especificação de ORM, permite mapeamento de tabelas de banco de dados relacional em classes e objetos e vice-versa. Para facilitar esse mapeamento, são usadas **anotações** junto às classes e consequentemente aos seus objetos. Essas anotações, chamadas de metadados, são preenchidas e lidas pelo *framework*, que implementa a JPA. As principais anotações são:

@Entity e @table

Quando uma classe representa uma tabela do banco de dados, utiliza-se a anotação **@Entity**, que estabelecerá uma ligação entre a entidade e a tabela, criando uma tabela com o mesmo nome da classe, caso a tabela não exista. Da mesma forma, os objetos criados a partir dessa classe são persistidos nas “linhas” dessa tabela no banco de dados.

Resumindo, em orientação a objetos, uma classe que é uma entidade representa uma tabela no banco de dados e cada objeto dessa classe representa uma linha dessa tabela. Veja um exemplo de uso:

```
        @Entity  
public class Produto {
```

Com esse código criou-se a classe **Produto**, que será uma entidade, ou seja, terá uma tabela de mesmo nome no banco de dados. Já os objetos instanciados da classe **Produto** terão seus dados armazenados na mesma tabela.

Se, por acaso, você estiver mapeando um banco já existente com nomes de tabelas diferentes dos nomes das classes, você pode acrescentar uma anotação **@table** da seguinte forma:

```
        @Entity  
        @Table(name="TABELA_PRODUTO")  
public class Produto {  
  
    }
```

Você ainda pode dar um “apelido” para sua entidade, para referenciá-la sem usar o nome da classe. Isso pode ser usado em consultas, por exemplo. Esse apelido é dado pelo atributo **name** da anotação **@Entity**.

```
        @Entity(nome="EntidadeProduto")  
        @Table(name="TABELA_PRODUTO")  
public class Produto {  
  
    }
```


@Id e @Column



Como você já sabe, as tabelas no banco de dados devem ter uma chave primária e, quando as tabelas do banco são mapeadas pelas entidades da JPA, é importante definir qual será essa chave, e isso é feito pela anotação **@Id**. Confira o exemplo de uso:

```
@Entity
public class Usuario {

    @Id
    private long id;

}
```

Nesse código, a entidade **Usuário** terá como identificador único o atributo **Id**, nesse caso, do tipo *long*, e, por padrão, esse campo estará vinculado à coluna de mesmo nome na tabela do banco de dados.

Caso a tabela tenha uma coluna com nome diferente do nome do campo na entidade, é possível direcioná-la com a anotação **@Column**, que pode ter vários parâmetros que determinam características da coluna, entre eles **name**, para definir o nome da coluna no banco de dados.

```
@Entity
public class Usuario {

    @Id
    @Column(name="usuario_id")
    private long id;

}
```

@GeneratedValue



Essa notação é usada quando se quer indicar que será gerado pelo provedor de persistência o valor do identificador único da entidade. Ela deve ser adicionada logo após a anotação **@Id**.

Utilizar essa opção significa que a responsabilidade de gerar e gerenciar as chaves primárias será do *software*, ou seja, o usuário deverá informar esse dado ou o programa Java gerá-lo ou calculá-lo.

Veja o exemplo a seguir:

```
@Entity
public class Usuario {

    @Id
    @GeneratedValue
    private long id;

}
```

Com esse exemplo de código, a camada JPA passará a utilizar a estratégia de geração de chave primária **do banco de dados** ao qual a aplicação está conectada.

Caso você deseje outro tipo de estratégia de geração de valor para chave, você pode mudá-la adicionando o parâmetro **strategy** à anotação **@GeneratedValue** com o valor que define a estratégia escolhida.

As estratégias possíveis são as seguintes:

GenerationType.AUTO

É o valor padrão, sendo o mesmo que deixar apenas a anotação **@GeneratedValue**, sem nada, atribuindo ao provedor de persistência a escolha da estratégia mais adequada de acordo com o banco de dados.

GenerationType.IDENTITY



Determinar esse valor ao parâmetro indica que os valores devem ser definidos pelo SGBD. Nesse caso, a definição da tabela no banco deve ser *auto increment*. Assim, o SGBD gerará o valor e a JPA replicará esse valor. Alguns SGBDs podem não suportar essa opção.

GenerationType.SEQUENCE

Com esse valor, o provedor de persistência gerará os valores a partir de uma **sequence**. Caso não seja especificado o nome da *sequence*, será utilizada uma *sequence* padrão, que será global, ou seja, será a mesma para todas as entidades. Alguns bancos de dados podem não suportar essa opção.

GenerationType.TABLE

Com a opção **TABLE**, será necessário criar uma tabela que seja responsável por gerenciar as chaves primárias. Essa opção não é muito recomendada, pois o número grande de consultas pode gerar uma sobrecarga ao gerenciar as chaves primárias.

Confira um exemplo de uso do parâmetro **strategy** e do valor **GenerationType.IDENTITY** para definir que o *auto increment* do MySQL estabelecerá o valor das chaves primárias:

```
@Entity
public class Usuario {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

}
```

Aplicabilidade



Para aprofundar seus conhecimentos em alguns conceitos, será criado um novo projeto, no qual serão vistas as configurações e funções da JPA. Nesse caso, será utilizada a implementação Hibernate, por ser esta uma implementação muito comum no mercado de trabalho. Mas lembre-se de que os conceitos aqui apresentados podem ser replicados com bastante similaridade em outras implementações.

Criando o projeto

Como estudado anteriormente, para funcionar, o Hibernate precisa de algumas configurações e bibliotecas. Por isso, este projeto será do tipo Maven, que permite cadastrar em um arquivo de configuração quais são as bibliotecas de que o projeto depende, baixando automaticamente os arquivos **.jar** dessas bibliotecas pela Internet (a partir do repositório central MVN).

Abra o seu NetBeans e crie um novo projeto “Java with Maven” e “Java Application”.

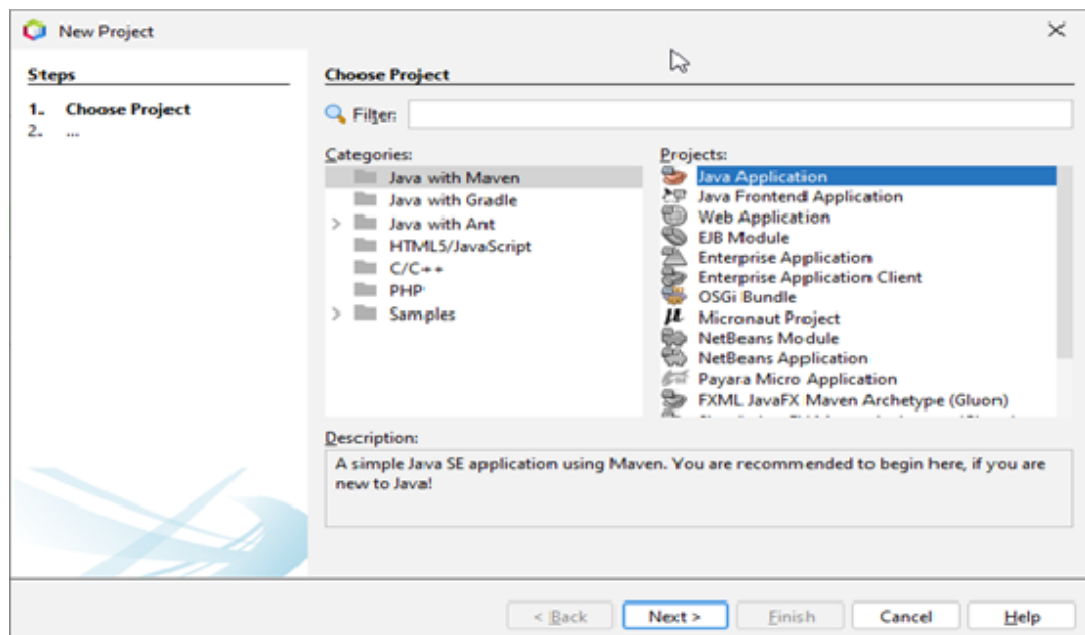


Figura 3 – Novo projeto “Java With Maven” no NetBeans

Fonte: NetBeans 13 (2022)

No passo seguinte, informe o nome do projeto e um **Group Id** que identifica o pacote principal do projeto.

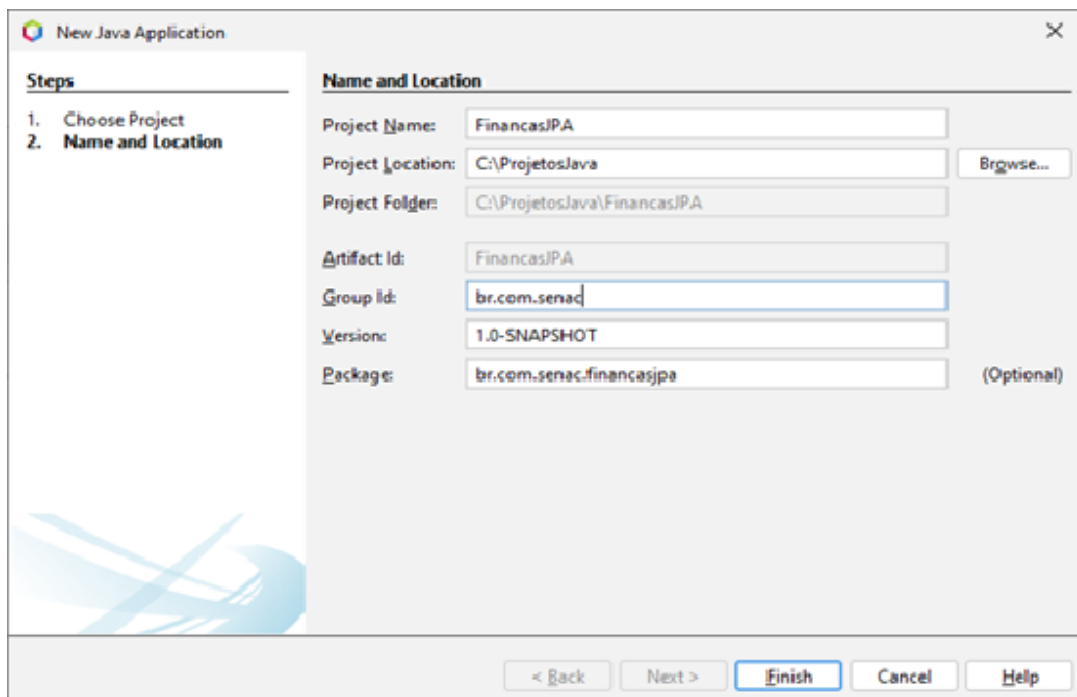


Figura 4 – Criando o projeto “Financas JPA”

Fonte: NetBeans 13 (2022)

Feito isso, o projeto será criado e contará com a seguinte estrutura:

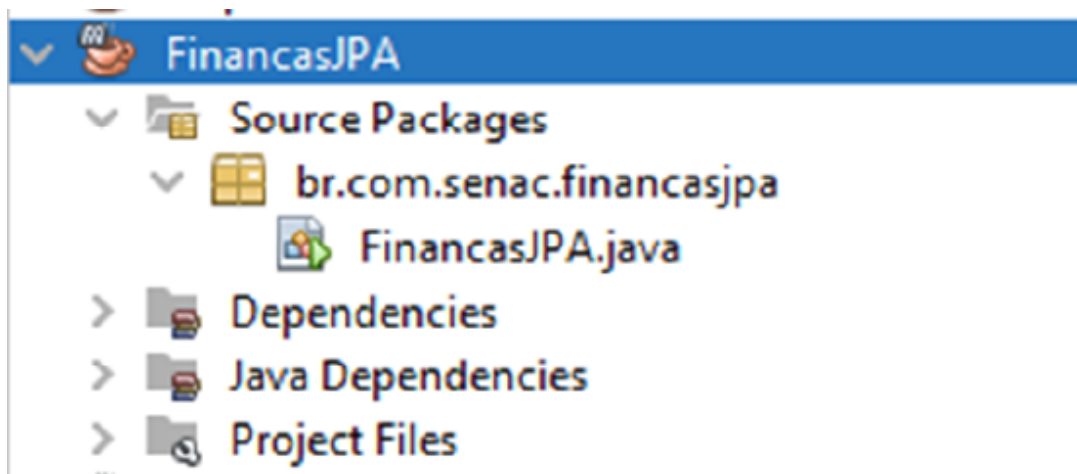


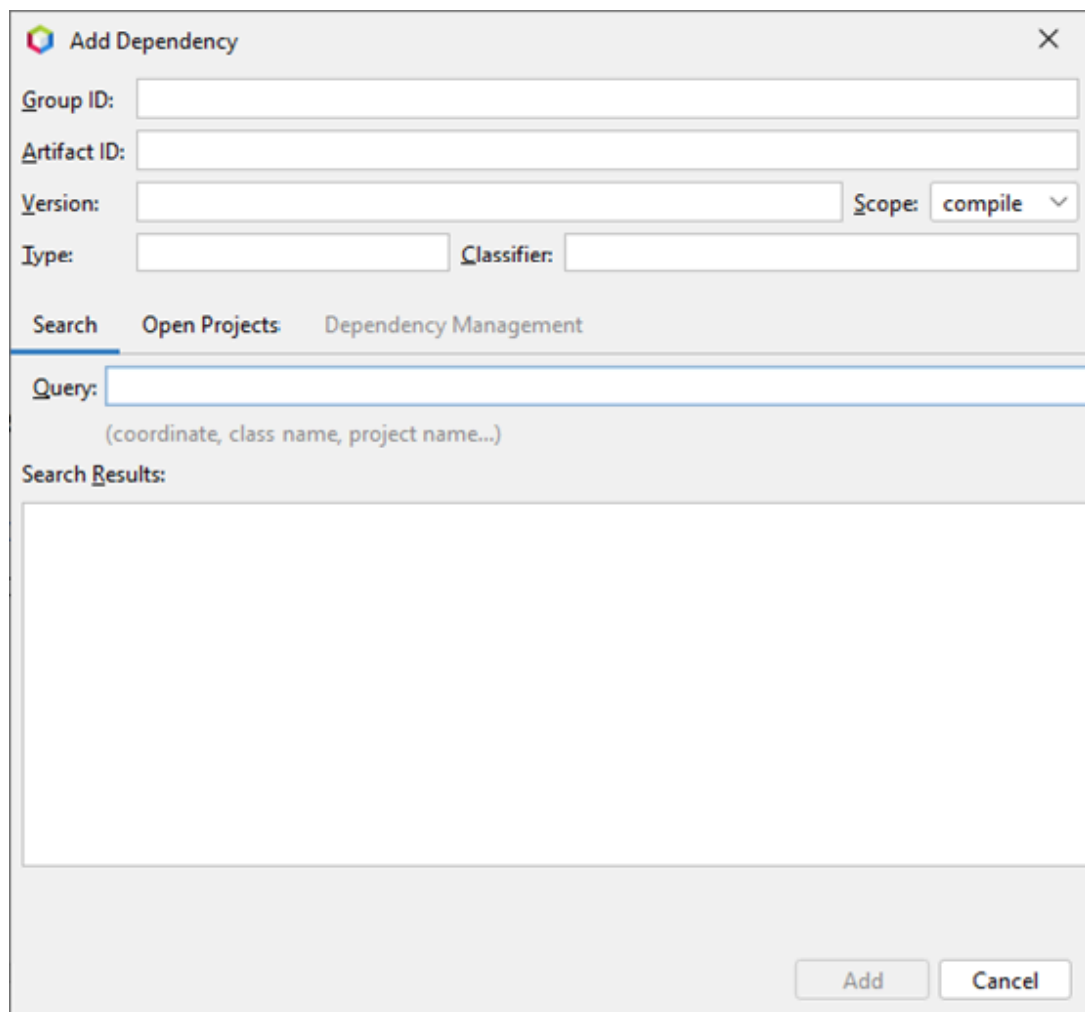
Figura 5 – Projeto “FinancasJPA” no NetBeans

Fonte: NetBeans 13 (2022)

Antes de qualquer coisa, deve-se indicar duas dependências: uma para o Hibernate e outra para o MySQL Connector.

O MySQL Connector é necessário porque o ORM (o Hibernate, no caso) precisa dessa biblioteca para conseguir estabelecer conexão com o banco de dados. Caso seja utilizado outro banco de dados, será preciso o conector correspondente.

Para declarar essas dependências de modo facilitado, você pode clicar com o botão direito na pasta **Dependencies** do projeto e optar por **Add Dependency**. Surgirá uma janela solicitando dados dessa dependência.



The image shows the 'Add Dependency' dialog box in NetBeans. It features several input fields: 'Group ID', 'Artifact ID', 'Version', 'Scope' (a dropdown menu currently set to 'compile'), 'Type', and 'Classifier'. Below these fields are three tabs: 'Search', 'Open Projects', and 'Dependency Management'. The 'Search' tab is selected, displaying a 'Query' input field with a hint '(coordinate, class name, project name...)' and a large 'Search Results' area. At the bottom right, there are 'Add' and 'Cancel' buttons.

Figura 6 – Adicionando dependência ao projeto Maven

Fonte: NetBeans 13 (2022)

É possível descobrir esses dados por meio do *site* do repositório central do Maven. Para isso, pesquise “Maven Central Repository” na Internet. No repositório, localize o campo de busca do *site*, use o termo “org.hibernate.orm hibernate-core”, sem as aspas, e clique no botão de lupa.

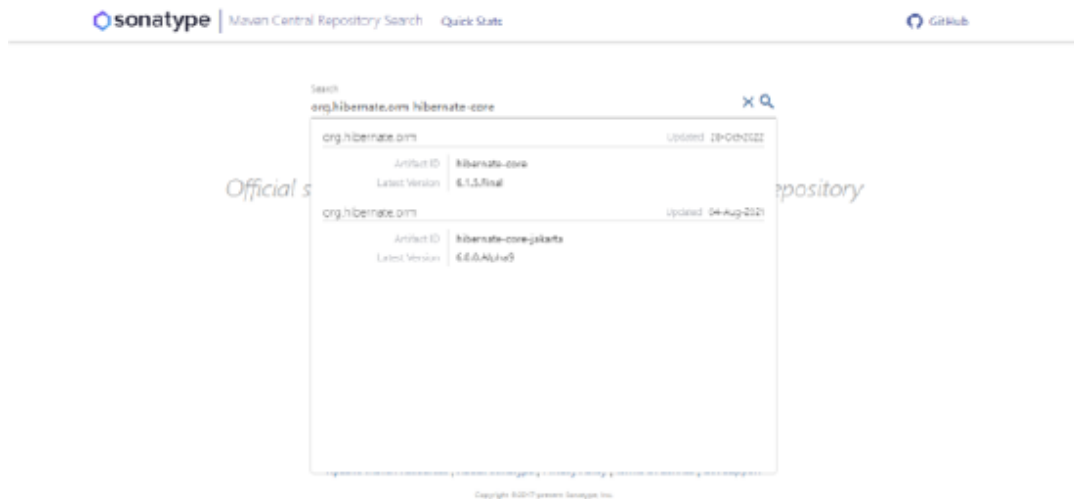


Figura 7 – Buscando a referência da biblioteca Hibernate

Fonte: Sonatype (c2017-2022)

Na tela seguinte, opte pelo item hibernate-core de maior versão (no momento da escrita deste conteúdo, a versão atual é 6.1.5). Clique exatamente no texto com o número da versão.

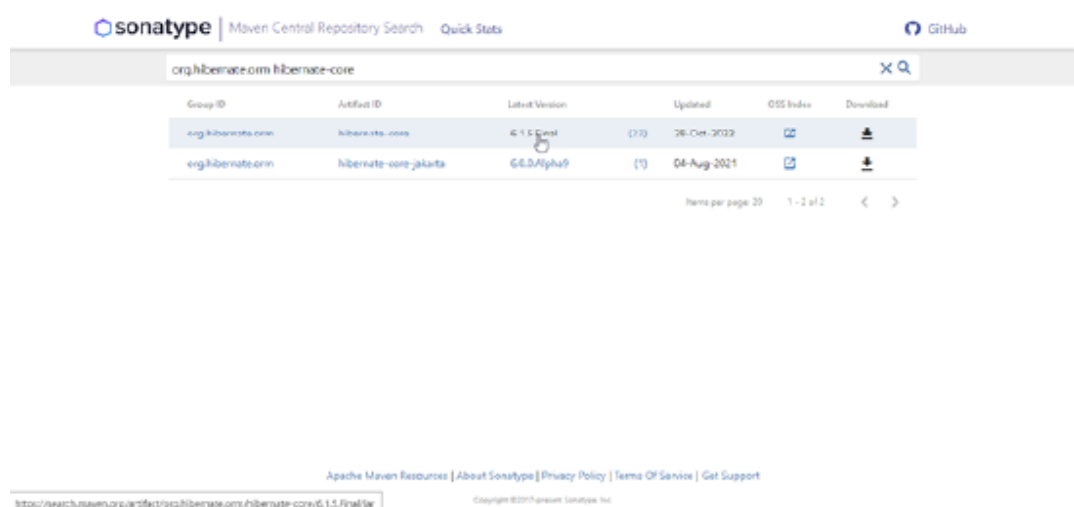


Figura 8 – Selecionando a versão da biblioteca Hibernate

Fonte: Sonatype (c2017-2022)

As informações da dependência surgirão à direita na tela, na área **Apache Maven**.

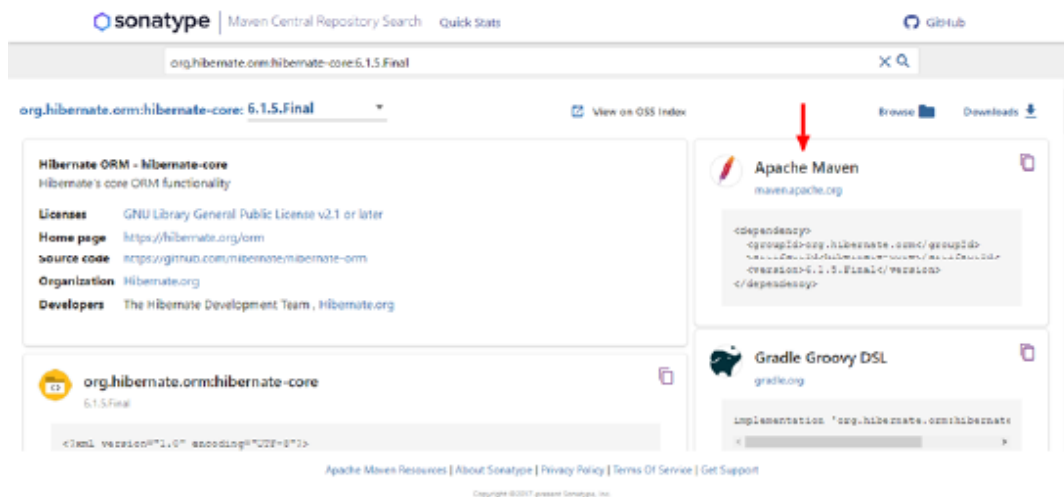
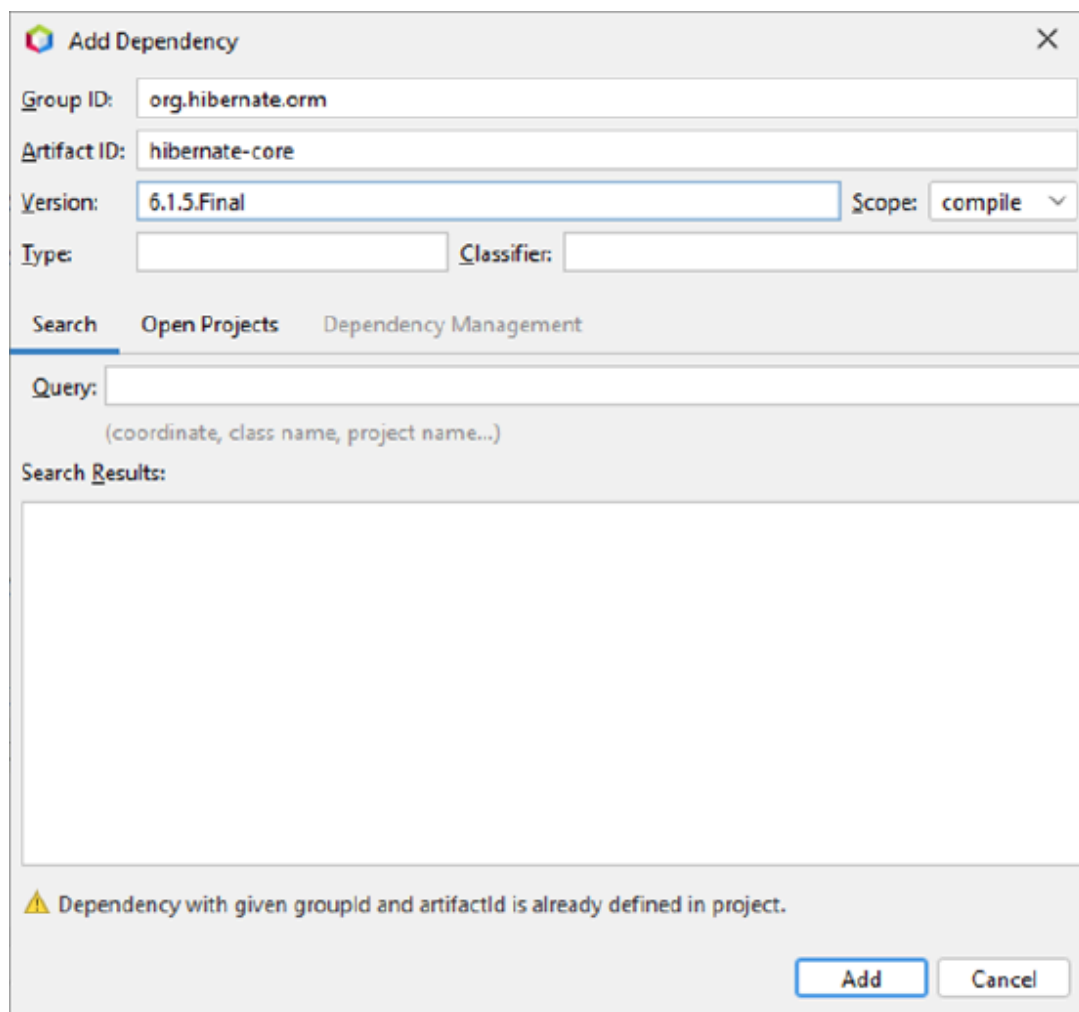


Figura 9 – Dados para inclusão de dependência Maven

Fonte: Sonatype (c2017-2022)

Ainda com essa página aberta, volte ao NetBeans e informe os dados que aparecem ali. Em Group ID, informe “org.hibernate.orm”, que está entre as tags `<groupId>` e `</groupId>`. No campo Artifact ID, informe o valor “hibernate-core”, que está entre `<artifactId>` e `</artifactId>` na página. Em Version, informe o valor que está entre `<version>` e `</version>` (neste exemplo, o valor é “6.1.5.Final”, mas pode ser outro, de acordo com a última versão do Hibernate).



Add Dependency

Group ID:

Artifact ID:

Version: Scope:

Type: Classifier:

Search Open Projects Dependency Management

Query:

(coordinate, class name, project name...)

Search Results:


 Dependency with given groupId and artifactId is already defined in project.

Figura 10 – Configurações da dependência no Projeto Maven

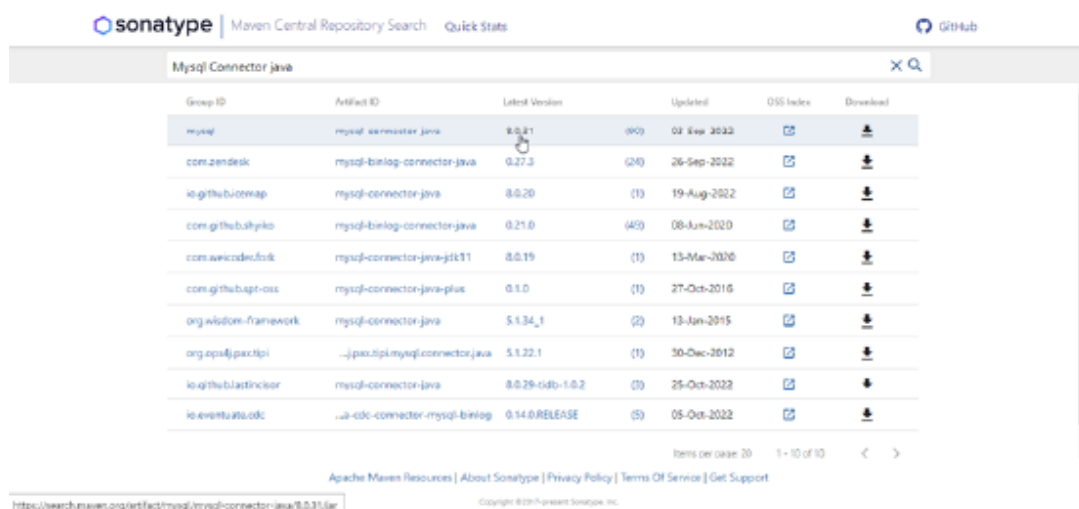
Fonte: NetBeans 13 (2022)

Clicando em **Add**, o NetBeans editará o arquivo **pom.xml**, próprio do Maven, presente na pasta **Project Files** do projeto. Esse arquivo, neste momento, deve estar com um conteúdo semelhante ao seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.senac.financasjpa</groupId>
  <artifactId>FinancasJPA</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.1.5.Final</version>
    </dependency>
  </dependencies>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <exec.mainClass>br.com.senac.financasjpa.FinancasJPA</exec.mainClass>
  </properties>
</project>
```

As linhas destacadas são referentes à dependência que acabou de ser cadastrada. Caso se sinta seguro, você pode, inclusive, editar o arquivo **pom.xml** diretamente, informando as dependências necessárias.

Agora, você fará o mesmo procedimento para a biblioteca MySQL Connector. No *site* Maven Central Repository, busque “MySQL Connector Java” e clique na lupa. O resultado será algo semelhante ao da seguinte imagem:



The screenshot shows the Sonatype Maven Central Repository Search results for 'MySQL Connector Java'. The table lists various artifacts with their Group ID, Artifact ID, Latest Version, Updated date, OSS Index status, and Download link. The first row, representing the 'mysql:mysql-connector-java' artifact, is highlighted, and a mouse cursor is pointing at the version '8.0.31'.

Group ID	Artifact ID	Latest Version	Updated	OSS Index	Download
mysql	mysql-connector-java	8.0.31	02-Sep-2022		
com.zendesk	mysql-binlog-connector-java	0.27.3	26-Sep-2022		
io.github.buonap	mysql-connector-java	8.0.20	19-Aug-2022		
com.github.bahyko	mysql-binlog-connector-java	0.21.0	08-Jun-2020		
com.warcode.fork	mysql-connector-java-jdk11	8.0.19	13-Mar-2020		
com.github.bapt-oss	mysql-connector-java-plus	0.1.0	27-Oct-2016		
org.wildcore.framework	mysql-connector-java	5.1.34_1	13-Jan-2015		
org.op4j.pactipi	~jpac4pi.mysql.connector.java	5.1.22.1	30-Dec-2012		
io.github.bactincisor	mysql-connector-java	8.0.29-0.80-1.0.2	25-Oct-2022		
io.eventualab.cdc	~cdc-connector-mysql-binlog	0.14.0 RELEASE	05-Oct-2022		

Items per page: 20 1 - 10 of 10 < >

Apache Maven Resources | About Sonatype | Privacy Policy | Terms Of Service | Get Support

<https://search.maven.org/artifact/mysql/mysql-connector-java/8.0.31/jar>

Copyright © 2017-present Sonatype, Inc.

Figura 11 – Resultado da busca “MySQL Connector Java” selecionando a versão 8.0.31

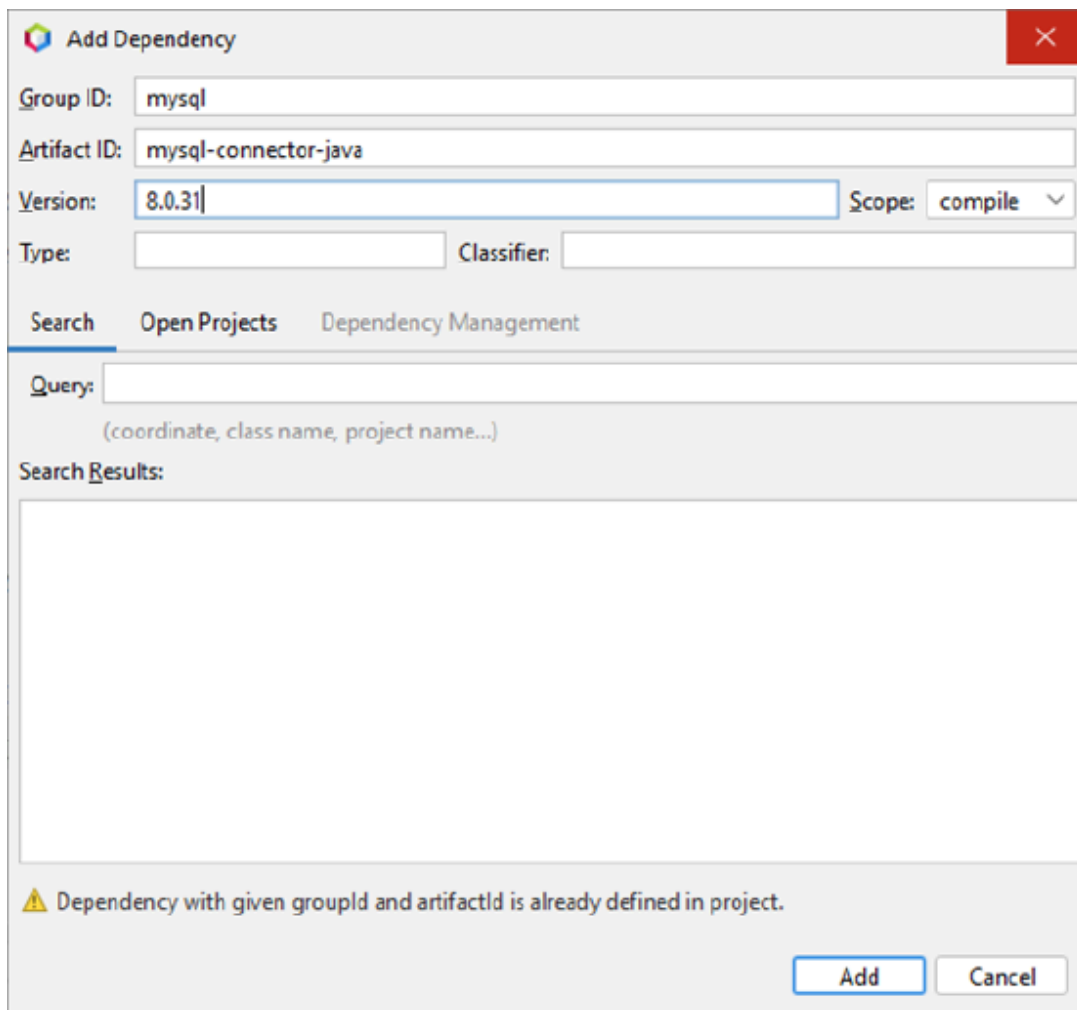
Fonte: Sonatype (c2017-2022)

Opte pelo **Group ID** que seja **mysql** e pela última versão disponível. No momento da escrita deste material, a última versão é a 8.0.31.

Na tela seguinte, a área **Apache Maven** deverá ter um texto XML semelhante ao seguinte:

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.31</version>
</dependency>
```

No NetBeans, você pode adicionar a dependência clicando com o botão direito do *mouse* na pasta **Dependencies** e optando por **Add Dependency**. Informe os valores correspondentes ao que se vê na página do Maven.



The screenshot shows the 'Add Dependency' dialog box in NetBeans. The fields are filled with 'mysql' for Group ID, 'mysql-connector-java' for Artifact ID, '8.0.31' for Version, and 'compile' for Scope. The 'Type' and 'Classifier' fields are empty. Below the fields are tabs for 'Search', 'Open Projects', and 'Dependency Management'. The 'Search' tab is active, showing a 'Query' field and a 'Search Results' area. A warning message at the bottom states: 'Dependency with given groupId and artifactId is already defined in project.' The 'Add' button is highlighted.

Figura 12 – Adicionando a dependência da biblioteca MySQL Connector no projeto Maven

Fonte: NetBeans 13 (2022)

Por fim, clique em **Add** para concluir a inclusão da dependência no arquivo **pom.xml**.

Você pode editar diretamente o arquivo **pom.xml** informando o trecho de XML que a página do Maven mostra. O trecho deve ficar entre **<dependencies>** e **</dependencies>** e pode ser incluído logo após a última *tag* **</dependency>**.

Ao final desses processos, o arquivo **pom.xml** deve estar desta forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.or
g/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.senac</groupId>
  <artifactId>FinancasJPA</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.1.5.Final</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.31</version>
    </dependency>
  </dependencies>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>18</maven.compiler.source>
    <maven.compiler.target>18</maven.compiler.target>
    <exec.mainClass>br.com.senac.financasjpa.FinancasJPA</exec.mainClass>
  </properties>
</project>
```

Depois de concluídos esses passos, você pode clicar no botão **Build** (ícone de martelo) para construir o projeto e o Maven efetivamente baixar os arquivos de dependência. Isso pode levar algum tempo, então aguarde até estar tudo baixado. Certifique-se de estar conectado à Internet.

Configurando a conexão com banco de dados

Além das bibliotecas, é preciso configurar a conexão com o banco de dados que será usado. Diferentemente da JDBC (Java Database Connectivity), em que se pode definir o texto de conexão diretamente no código, o Hibernate mantém essa informação em um arquivo XML de configuração separado. Para isso, é preciso primeiro um caminho de pastas que forme **src/main/resources/META-INF**. Então, no NetBeans, utiliza a aba **Files** no painel da esquerda. Na pasta do projeto, expanda a pasta **src** e clique com o botão direito do *mouse* na pasta **main**. Opte por **New > Folder...**

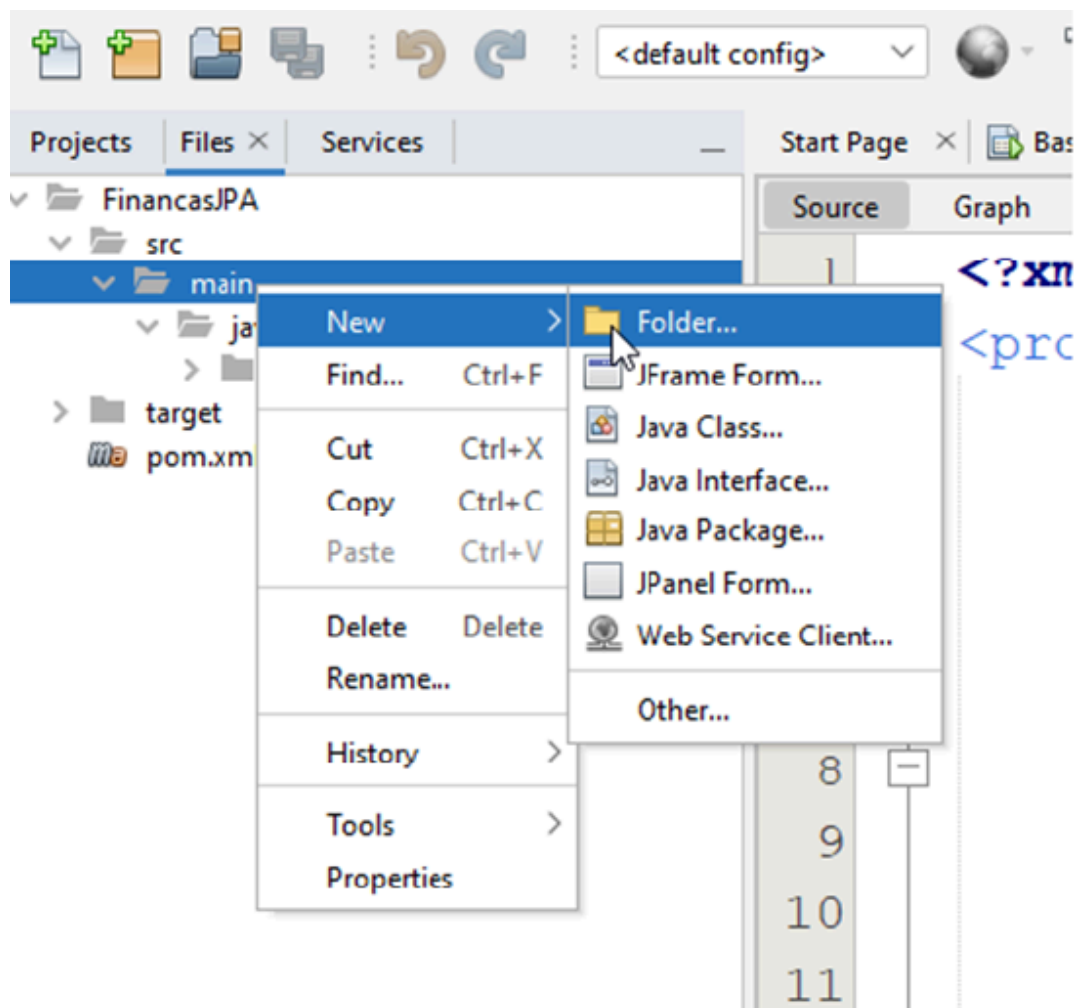


Figura 13 – Criando uma pasta na aba Files do NetBeans

Fonte: NetBeans 13 (2022)

No campo **Folder Name** da nova janela, informe **resources/META-INF**. Em seguida, ainda na guia **Files** do NetBeans, expanda até a pasta **META-INF**. Clique com o botão direito do *mouse* nela e opte por **New > Other...** Na tela **New File**, selecione **XML** e **XML Document**.

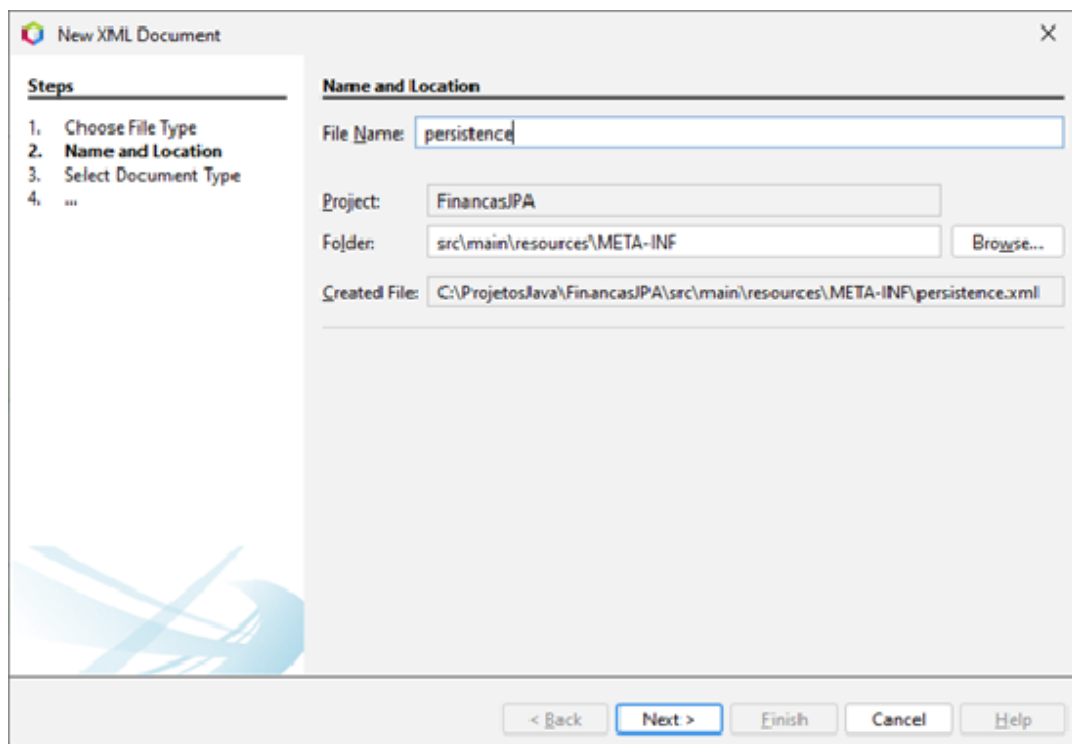


Figura 14 – Tela de criação de arquivo XML

Fonte: NetBeans 13 (2022)

Nomeie como **persistence.xml**. É necessário que o caminho e o nome do arquivo estejam exatamente assim: **src/main/resources/META-INF/persistence.xml**. Caso não esteja, o Hibernate não será capaz de localizá-lo.

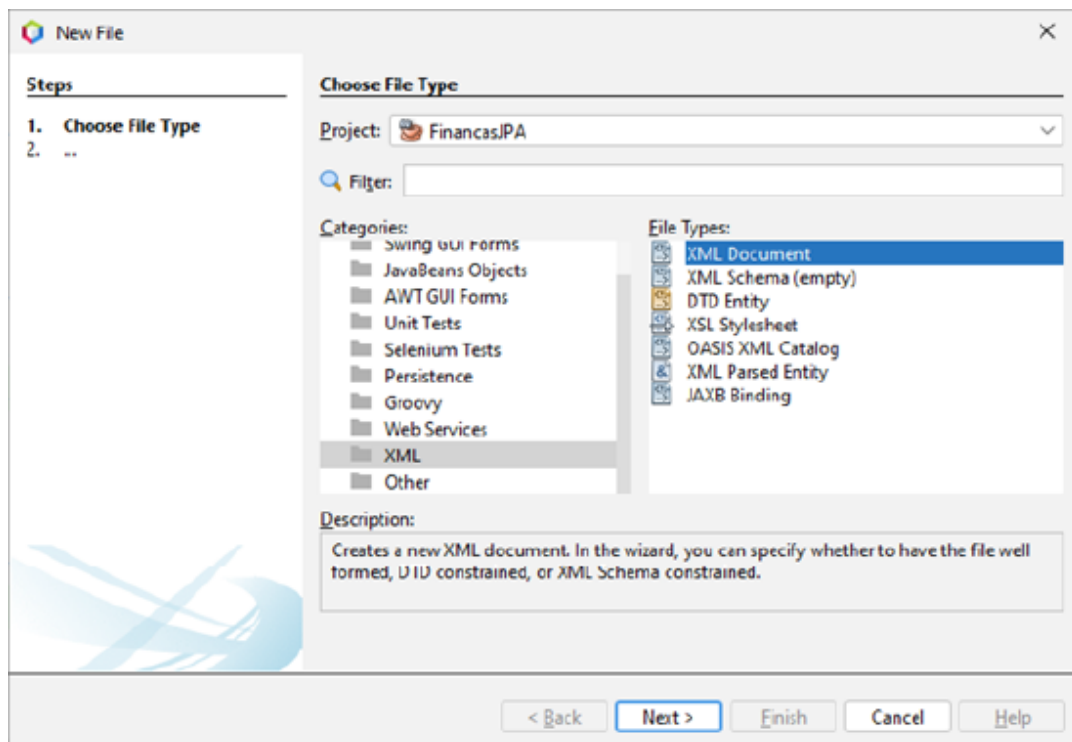


Figura 15 – Configurando nome do novo arquivo XML

Fonte: NetBeans 13 (2022)

Depois de clicar em **Next**, você pode optar por **Well-formed Document** e clicar em **Finish**.

Abra o arquivo e substitua o conteúdo gerado pelo seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="3.0"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">

  </persistence>
```

A tag **<persistence-unit>** define a unidade de persistência do projeto e nela define-se a conexão com o banco de dados. Neste projeto, você nomeará sua unidade de persistência de **Financas-PU**.

Agora, é preciso declarar propriedades que permitirão ao ORM se conectar com o banco de dados. As *tags* a seguir devem ficar entre `<persistence-unit>` e `</persistence-unit>`.


```
<properties>
  <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc
  c.Driver" />
  <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localho
  st/financasjpa" />
  <property name="jakarta.persistence.jdbc.user" value="root" />
  <property name="jakarta.persistence.jdbc.password" value="12345" />

  <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dia
  lect" />
</properties>
```

Entenda agora o que cada *tag* significa:

```
<property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc
c.Driver" />
```

Indica qual classe de *driver* para a conexão será utilizada pelo Hibernate. Como você está usando a biblioteca Connector J do MySQL, o valor será **com.mysql.cj.jdbc.Driver** por padrão.



```
<property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost/financasjpa" />
```

Representa o caminho para o banco de dados que será usado. Neste exemplo, o valor é **`jdbc:mysql://localhost/financasjpa`**, indicando que se utilizou o MySQL, que o banco de dados está na própria máquina em que o programa roda (*localhost*) e que o banco se chama **`financasjpa`**.

Se você conectar em um banco de dados que está em um servidor remoto, você pode substituir o *localhost* pelo endereço de IP do servidor ou pelo nome de domínio, simulando um servidor de banco de dados na mesma rede do servidor de aplicação.

Assim, o resultado seria algo semelhante a isto:

```
<property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://192.168.0.123/financasjpa" />
```

Se o servidor de banco de dados *on-line* for diferente do servidor de aplicação, o endereço do servidor deve vir acompanhado pela porta que o MySQL está usando, o que ficaria semelhante a isto.

```
<property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://meuserv  
idormysql.com.br:3306/financasjpa" />
```

Lembre-se de não copiar simplesmente esses códigos e sim configurar corretamente no seu servidor. O único código que pode ficar idêntico é o do primeiro exemplo, caso o servidor MySQL e a aplicação estejam no mesmo computador.

```
<property name="jakarta.persistence.jdbc.user" value="root" />
```

```
<property name="jakarta.persistence.jdbc.password" value="12345" />
```

Armazenam, respectivamente, os valores para usuário (*root*, neste exemplo) e senha (“12345”, neste exemplo) para conexão com o banco de dados. Troque para o usuário e a senha usados no seu banco de dados. A *tag* de *password* pode ser excluída se o banco não usar senha.

Define o “dialeto” do hibernate com o banco de dados. O valor **org.hibernate.dialect.MySQL8Dialect** define que esse dialeto (usado para o ORM montar internamente as consultas SQL) obedece à definição do MySQL.

```
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect" />
```

O arquivo **persistence.xml** completo deve ficar como este exemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="3.0"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
  <persistence-unit name="teste">
    <properties>
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost/financas" />
      <property name="jakarta.persistence.jdbc.user" value="root" />
      <property name="jakarta.persistence.jdbc.password" value="12345" />

      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect" />
    </properties>
  </persistence-unit>
</persistence>
```

Você pode usar esse modelo de XML para seus próximos projetos alterando apenas o nome do banco de dados na *tag* de propriedade **jakarta.persistence.jdbc.url** e, possivelmente, o usuário e a senha do banco de dados.

A partir desse momento, seu projeto está preparado para a implementação de JPA com Hibernate, conectando a banco de dados MySQL. A seguir, será iniciada a criação das classes que refletirão em tabelas no banco de dados.

Caso você tenha algum problema com a configuração do arquivo, é possível que seu projeto precise que a pasta **META-INF** fique diretamente na pasta **src**. Então, basta mover a pasta **META-INF** e seu conteúdo, e isso deve resolver.



Primeiras operações com JPA

A seguir, será criado um projeto de *software* que permita o cadastramento e a manipulação de despesas e receitas para controle de finanças pessoais.

Uma vez que as configurações foram realizadas no seu primeiro projeto com JPA, você pode testar as operações básicas com banco de dados. Antes de tudo, você deve criar a base de dados e a tabela “despesa”, que será foco dos testes iniciais. Para isso, utilize o seguinte *script* no servidor MySQL:

```
create database Financas;

use Financas;

create table despesa(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  descricao VARCHAR(100),
  valor NUMERIC(5, 2),
  data DATE
);
```

Para despesas, registre a descrição, o valor e a data em que elas aconteceram, além do ID, que é chave primária com autoincremento.

No projeto Java, você precisa de uma classe que seja correspondente a essa tabela. É o que se chama classe de persistência. Crie uma nova classe no pacote principal da aplicação e use a seguinte estrutura.

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import java.time.LocalDate;

@Entity
public class Despesa {

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

    private String descricao;

    private double valor;

    private LocalDate data;

    /**getters e setter
     * @return s*/
    public int getId() {
        return id;
    }

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

    public String getDescricao() {
        return descricao;
    }

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

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public LocalDate getData() {
        return data;
    }

    public void setData(LocalDate data) {
```

```
        this.data = data;
    }
}
```

Se preferir, utilize a ferramenta de sugestão do NetBeans para completar as cláusulas de **import** no código. Veja que, na classe, há termos com um caractere “@” à esquerda. Eles são as *annotations*, mencionadas no início deste conteúdo, e representam alguma função ou classificação específica para uma classe, um atributo ou um método.

Relembre as anotações usadas nessa classe **Despesa**:

@Entity – Representa que no banco de dados haverá uma tabela com nome “despesa”, relacionada diretamente a essa classe.

@Id – Indica que o atributo **int id** representa um identificador, a chave da tabela.

@GeneratedValue – Indica que o atributo tem valor gerado automaticamente pelo banco de dados. É o caso de ID, que, na tabela “despesa”, tem **auto_increment**. Como **auto_increment** foi utilizado como estratégia de geração automática de valor, é preciso especificar um parâmetro para a *anotation* **strategy = GenerationType.IDENTITY**.

Para cada tabela no banco de dados será necessária uma classe mapeada, como a do exemplo anterior.

É importante notar quais tipos de atributos as entidades suportam:

Tipos primitivos

Tipo *string*

Tipos serializáveis (ou seja, que contêm métodos para serem representados por texto):

- Classes correspondentes aos primitivos (como **Integer** e **Double**, por exemplo)

- Classes de tempo (**Date**, **LocalDate**, **Calendar** e outras)

- Classes criadas pelo usuário que implementem serialização

Outras entidades ou coleções de entidades

Para este caso, serão utilizados apenas tipos primitivos (**int**, **double**), *string* e tipo de data.

Uma vez criada a entidade de persistência “Despesa”, realize seu primeiro teste diretamente no método **main()** da classe principal do projeto (em **FinancasJPA.java**). Quando se trabalha com operações no banco com JPA, a primeira coisa que deve ser feita é criar uma fábrica de entidades (você se lembra dos padrões de projeto Factory e Abstract Factory?) e, a partir dela, gerar um objeto da classe **EntityManager**, que será usado para toda a manipulação de dados necessária.

```
EntityManagerFactory fabricaEntidade = Persistence.createEntityManagerFactory("Financas-PU");
EntityManager manager = fabricaEntidade.createEntityManager();
```

Veja que na primeira linha encontra-se **Persistence.createEntityManagerFactory("Financas-PU");**. O parâmetro **Financas-PU** refere-se à unidade de persistência que foi definida no arquivo **persistence.xml** durante a configuração do projeto. Caso o arquivo não

exista, ou esteja em local diferente de **src/main/resources/META-INF/persistence.xml**, ou caso o nome da unidade de persistência esteja diferente do informado no parâmetro, uma falha como a seguinte deverá aparecer:

```
Exception in thread "main" jakarta.persistence.PersistenceException: No Persistence provider for EntityManager named Financas-PU
```

Para verificar se a configuração está correta, recomenda-se, inclusive, que você rode o projeto logo após acrescentar aquelas linhas de código. Se houver erro, reveja com cuidado o arquivo **persistence.xml** (analise local, *tags*, valores etc.).

No trecho anterior ainda foi criado um objeto **EntityManager** a partir da fábrica gerada. Esse objeto será essencial para as próximas operações.

A primeira operação que será realizada é a de **inclusão de dados**. Para isso, após definir o **EntityManager**, você precisa fazer o seguinte:

Crie o objeto com os dados que serão inseridos no banco

Inicie uma transação com o método **getTransaction().begin()** de **EntityManager**

Grave os dados com o método **persist()** de **EntityManager**

Encerre a transação com o método **getTransaction().commit()** de **EntityManager**

Agora no código, primeiro crie o objeto a ser inserido no banco:

```
Despesa gasto = new Despesa();  
gasto.setDescricao("Primeira despesa");  
gasto.setValor(10.50);  
gasto.setData(LocalDate.of(2022, 5, 30));
```

Depois, realize a inserção do objeto no banco de dados:

```
manager.getTransaction().begin();  
manager.persist(gasto);  
manager.getTransaction().commit();
```

O uso de transação é necessário para gravar os dados no banco. Caso você utilizasse apenas o método **persist()**, a entidade seria atualizada em memória e não seria gravada no banco de dados.

Por fim, você precisa fechar a conexão e a fábrica:

```
manager.close();  
fabricaEntidade.close();
```

O código completo do método **main()** nesse momento fica assim:

```
package br.com.senac.financasjpa;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import java.time.LocalDate;

public class FinancasJPA {

    public static void main(String[] args) {
        EntityManagerFactory fabricaEntidade = Persistence.createEntityManagerFactory("Financas");
        EntityManager manager = fabricaEntidade.createEntityManager();

        Despesa gasto = new Despesa();
        gasto.setDescricao("Primeira despesa");
        gasto.setValor(10.50);
        gasto.setData(LocalDate.of(2022, 5, 30));

        manager.getTransaction().begin();
        manager.persist(gasto);
        manager.getTransaction().commit();

        manager.close();
        fabricaEntidade.close();
    }
}
```

Você pode executar o programa e observar o resultado no banco de dados usando o MySQL Workbench. O resultado de um **select** na tabela “despesa” deve ser semelhante ao seguinte:

# id	descricao	valor	data
1	Primeira despesa	10.50	30/05/2022

Para **recuperar dados do banco** via JPA, utilize o método **find()** de **EntityManager**. O código a seguir busca o registro de “Despesa” com o “id 1”.


```
Despesa gasto = manager.find(Despesa.class, 1);
```

O método **find()** não precisa de transação, já que não modifica o banco de dados. Note que o retorno já é um objeto completo da classe **Despesa** – não é necessário percorrer coluna a coluna para preencher um objeto como com a JDBC. Ressalta-se ainda que **find()** é usada apenas quando se precisa de um registro com uma chave específica. Para consultas em geral, utilizam-se as linguagens JPQL ou Criteria, que serão discutidas ainda neste material.

O método **main** para esse teste apresenta-se desta forma:

```
public static void main(String[] args) {  
    EntityManagerFactory fabricaEntidade = Persistence.createEntityManagerFactory("Financas-PU");  
    EntityManager manager = fabricaEntidade.createEntityManager();  
  
    Despesa gasto = manager.find(Despesa.class, 1);  
    System.out.println(gasto.getDescricao());  
  
    manager.close();  
    fabricaEntidade.close();  
}
```

O resultado pode ser um pouco difícil de encontrar no console devido aos textos de *log* do Maven, mas deve estar mais ou menos nesta região:

```
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
Primeira despesa
nov. 01, 2022 8:37:43 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PoolState stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost/financas]
-----
BUILD SUCCESS
-----
```

A **operação de atualização** segue mais ou menos os passos da inclusão. A diferença é que, antes, era preciso encontrar o objeto de entidade a ser atualizado. No código a seguir, a descrição da despesa está sendo atualizada com “id = 1”.

```
Despesa gasto = manager.find(Despesa.class, 1);

manager.getTransaction().begin();
gasto.setDescricao("Despesa atualizada");
manager.getTransaction().commit();
```

O **EntityManager** utiliza objetos gerenciáveis. Isso significa que, de certa maneira, esse objeto recuperado do banco com **find()** está “marcado” como um objeto especial, que tem ligação com o banco. Se fosse utilizado um objeto comum, criado manualmente, a atualização não surtiria efeito.

O código para testes fica desta forma:

```
public static void main(String[] args) {  
    EntityManagerFactory fabricaEntidade = Persistence.createEntityManagerFactory("Financas-PU");  
    EntityManager manager = fabricaEntidade.createEntityManager();  
  
    Despesa gasto = manager.find(Despesa.class, 1);  
    manager.getTransaction().begin();  
    gasto.setDescricao("Despesa atualizada");  
    manager.getTransaction().commit();  
  
    manager.close();  
    fabricaEntidade.close();  
}
```

Em suma, sempre que você quiser atualizar um registro dessa maneira, apenas alterando um ou mais valores do objeto, precisará buscá-lo no banco para se certificar de que ele é um objeto gerenciável.

Para conferir o resultado da execução, é possível realizar uma consulta na tabela “Despesa” a partir do MySQL Workbench.

# id	descricao	valor	data
1	Despesa atualizada	10.50	30/05/2022

Nos casos em que, por algum motivo, não seja possível ou conveniente lidar com um objeto gerenciável do **EntityManager**, você pode usar o método **merge()** de **EntityManager** para produzir uma atualização. Nesse caso, você deve criar um objeto com o ID exato do registro que quer atualizar, preencher os demais atributos e persistir no banco de dados.

```
public static void main(String[] args) {  
    EntityManagerFactory fabricaEntidade = Persistence.createEntityManagerFactory("Financas-PU");  
    EntityManager manager = fabricaEntidade.createEntityManager();  
  
    Despesa gasto = new Despesa();  
    gasto.setId(1);  
    gasto.setDescricao("Atualizado com Merge");  
    gasto.setData(LocalDate.of(2022, 5, 30));  
  
    manager.getTransaction().begin();  
    manager.merge(gasto);  
    manager.getTransaction().commit();  
  
    manager.close();  
    fabricaEntidade.close();  
}
```

Observe nesse exemplo que primeiro criou-se um objeto com “id = 1”. Repare que não se informou dado algum para o atributo “valor”. Em seguida, utiliza-se o método **merge()** informando esse novo objeto. O método buscará no banco de dados o registro com “id = 1” e passará os valores do objeto a esse registro. O resultado, observado pelo MySQL Workbench, é o seguinte:

# id	descricao	valor	data
1	Atualizado com Merge	0.00	30/05/2022

Como não foi informado dado ao atributo “valor”, essa coluna permaneceu com zero, que é o valor padrão para tipo *double*. Por isso, deve-se ter cuidado para não sobrescrever alguma coluna indevidamente.

A **operação de exclusão**, por fim, é realizada a partir do método **remove()** de **EntityManager**. No trecho a seguir pode-se ver a remoção do registro com “id = 1”.

```
Despesa gasto = manager.find(Despesa.class, 1);  
manager.getTransaction().begin();  
manager.remove(gasto);  
manager.getTransaction().commit();
```

Assim como nas operações de atualização, é necessário informar ao método **remove()** um objeto gerenciado. Por isso, utiliza-se o **find()** para recuperar o objeto do banco de dados. O método **main()** completo fica desta forma:

```
public static void main(String[] args) {  
    EntityManagerFactory fabricaEntidade = Persistence.createEntityManagerFactory("Financas-PU");  
    EntityManager manager = fabricaEntidade.createEntityManager();  
  
    Despesa gasto = manager.find(Despesa.class, 1);  
    manager.getTransaction().begin();  
    manager.remove(gasto);  
    manager.getTransaction().commit();  
  
    manager.close();  
    fabricaEntidade.close();  
}
```

O resultado novamente pode ser checado com uma consulta à tabela “despesa” no MySQL Workbench.

Com isso, você experimentou as operações mais básicas com JPA: inclusão, exclusão, atualização e busca, isso tudo sem usar SQL, apenas objetos. Nos próximos passos, essa aplicação será expandida, organizando melhor as classes e as operações e criando telas para interação com o usuário.

Criando aplicação *desktop* com JPA

A partir de agora, será criada uma aplicação usando operações JPA para a persistência de dados. Você partirá do mesmo projeto FinancasJPA já criado, mas precisará realizar algumas modificações. Se preferir criar um novo projeto, fique à vontade; as configurações são as mesmas indicadas anteriormente.

O primeiro passo será a criação de um novo pacote Java. Ele será chamado de **br.com.senac.financasjpa.persistencia**. Para isso, clique no pacote padrão (**br.com.senac.financasjpa**) com o botão direito do *mouse* e opte por **New > Java Package**. Na tela seguinte, informe o nome do pacote.

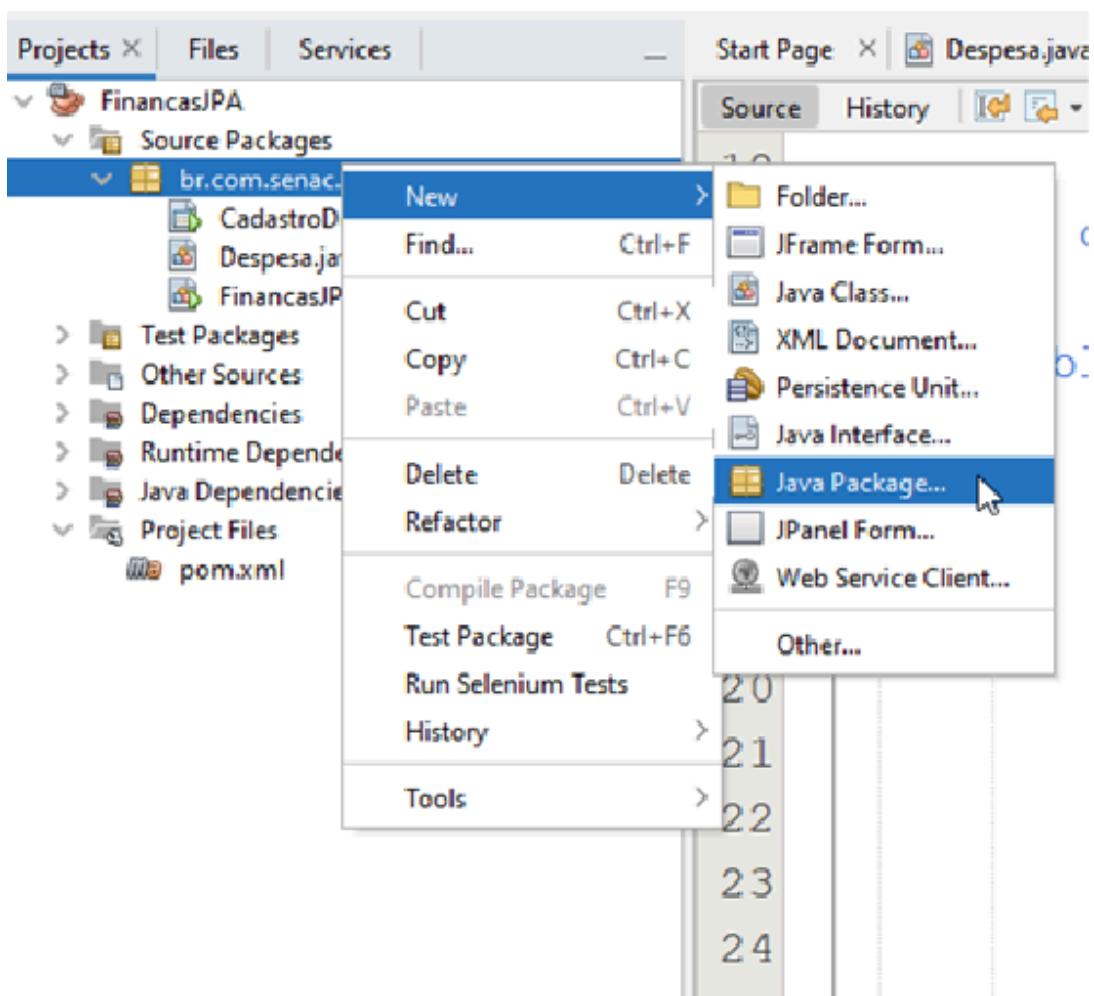


Figura 16 – Criando novo pacote

Fonte: NetBeans 13 (2022)

Como **Despesa** é uma classe de entidade, transfira-a para o novo pacote. Utilize a opção **Refactor** na caixa de opções que surge em seguida.

Crie também uma nova classe chamada **JPAUtil**, na qual você implementará as operações de criação de **EntityManager** e de seu encerramento. Essa classe será implementada no modelo Singleton, ou seja, haverá apenas uma instância dessa classe e seus métodos são estáticos. Use o código a seguir e preste atenção aos comentários incluídos nele.

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

public class JPAUtil {
    //constante para centralizar o nome da unidade de persistência
    // se o nome mudar, precisaremos alterar em um só lugar
    private static final String PERSISTENCE_UNIT = "Financas-PU";

    private static EntityManager em;
    private static EntityManagerFactory fabrica;

    //cria a entidade se estiver nula e a retorna
    public static EntityManager getEntityManager(){
        if(fabrica == null || !fabrica.isOpen())
            fabrica = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);

        if(em == null || !em.isOpen()) //cria se em nulo ou se o entity manager foi
        fechado
            em = fabrica.createEntityManager();

        return em;
    }

    //fecha o EntityManager e o factory
    public static void closeEntityManager(){
        if(em.isOpen() && em != null){
            em.close();
            fabrica.close();
        }
    }
}
```

Note que você está primeiro centralizando a referência à unidade de persistência de **persistence.xml**. Assim, mesmo que vários pontos do código usem essa referência, ficará fácil alterar esse nome se for necessário. Não se esqueça de usar a ferramenta de sugestão de **import** do NetBeans.

No pacote **persistencia**, crie também uma classe DAO (Data Access Object), que será responsável pelas operações diretas com banco de dados (neste caso, usando JPA).

Inicie essa classe com apenas um método para o cadastramento de uma nova despesa. Repare como os métodos de **JPAUtil** são usados no método **cadastrar()** a seguir.

```
package br.com.senac.financasjpa.persistencia;

import jakarta.persistence.EntityManager;

public class DespesaDAO {

    public void cadastrar(Despesa d){
        EntityManager em = JPAUtil.getEntityManager();
        try {
            em.getTransaction().begin();
            em.persist(d);
            em.getTransaction().commit();
        }catch(Exception e){
            em.getTransaction().rollback();
            throw e;
        }
        finally{
            JPAUtil.closeEntityManager();
        }
    }

}
```

Recomenda-se que você teste nesse momento o método criado usando o método **main()** da classe **FinancasJPA**.


```
public static void main(String[] args) {  
    DespesaDAO despesaDao = new DespesaDAO();  
  
    Despesa d = new Despesa();  
    d.setDescricao("Compras de mercado");  
    d.setValor(165.70);  
    d.setData(LocalDate.of(2022, 11, 15));  
  
    despesaDao.cadastrar(d);  
}
```

Se, no banco de dados, esse registro de “compras de mercado” foi incluído, é sinal de que tudo está correto.

Com a segurança de que o método de cadastrar está funcional, implemente agora telas que permitam o cadastro de despesa. Primeiro, crie um novo *package* **br.com.senac.financasjpa.gui** para armazenar as classes GUI (*graphical user interface*) do projeto.

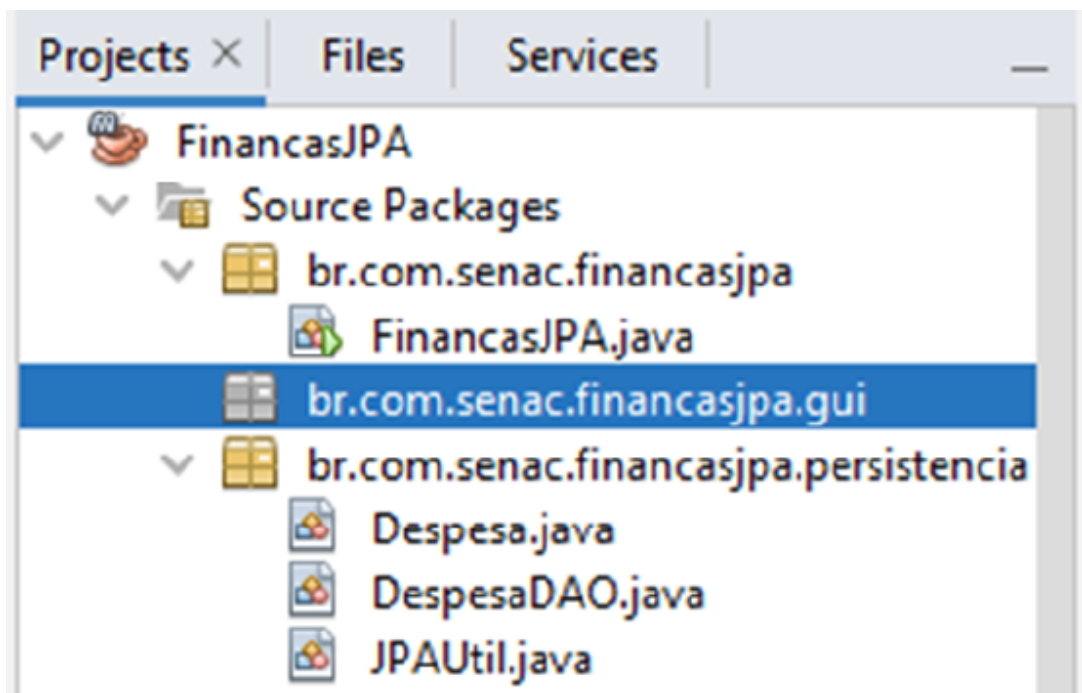


Figura 17 – Novo pacote **br.com.senac.financasjpa.gui** no projeto

Fonte: NetBeans 13 (2022)

Em seguida, crie o *form* (clique com o botão direito do *mouse* sobre o novo pacote: **New > JFrame Form**) com o nome **CadastroDespesa**. Propõe-se a tela com o seguinte arranjo:



Figura 18 – Tela para cadastro de despesa

Fonte: NetBeans 13 (2022)

Os elementos presentes são:

JLabel com nome de variável “lblDescricao” e texto “Descrição:”

TextField com nome de variável “txtDescricao” e texto vazio

JLabel com nome de variável “lblValor” e texto “Valor:”

JFormattedTextField com nome de variável “fmtTxtValor” e texto vazio. É necessário ajustar a propriedade “formatterFactory” configurando **Category** com a opção “number” e **Format** “0.00”.

JLabel com nome de variável “lblData” e texto “Data:”

JFormattedTextField com nome de variável “fmtTxtData” e texto vazio. É necessário ajustar a propriedade “formatterFactory” configurando **Category** com a opção “date” e **Format** com opção “short”.

JFormattedTextField são campos que obrigam o usuário a informar um texto com o formato especificado pela configuração da propriedade “formatterFactory” (que, no NetBeans, abre uma janela como a da figura a seguir). Caso o valor não esteja corretamente formatado, ele será apagado assim que se retira o foco do campo.

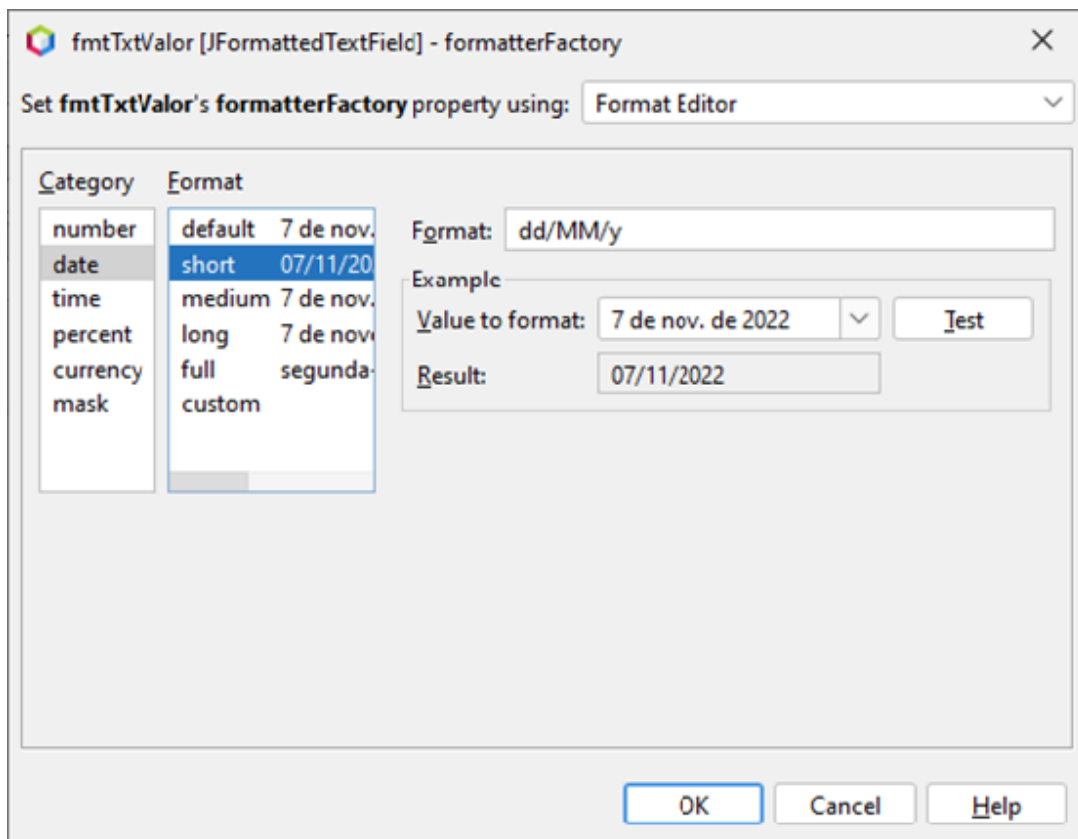


Figura 19 – Configuração da propriedade “formatterFactory” do campo “fmtData” da tela de cadastro de despesa

Fonte: NetBeans 13 (2022)

Há ainda um botão com nome de variável “btnSalvar”. Com um duplo clique nesse botão, você é levado ao código de evento de clique dele, o qual preencherá da seguinte forma:

```
private void btnSalvarActionPerformed(java.awt.event.ActionEvent evt) {  
    Despesa novaDespesa = new Despesa();  
    try{  
        //para a descrição capturamos direto o valor do campo de texto  
        novaDespesa.setDescricao(txtDescricao.getText());  
        //substitui , por . para que fique no formato numérico que o Java entende  
        como double  
        novaDespesa.setValor(Double.parseDouble(fmtTxtValor.getText().replace  
        ("", ".", ".")));  
        //classe DateFormatter necessária para obter o tipo LocalDate a partir de  
        um texto  
        novaDespesa.setData(LocalDate.parse(fmtTxtData.getText(), DateTimeFormatt  
        er.ofPattern("dd/MM/y")));  
  
        DespesaDAO despesaDao = new DespesaDAO();  
        despesaDao.cadastrar(novaDespesa);  
    }catch(Exception e){  
        JOptionPane.showMessageDialog(this, "Ocorreu uma falha:\n" + e.getMessage  
        ());  
    }  
}
```

O código mostra a montagem de um objeto da classe **Despesa**, no qual se destaca a necessidade de ajuste no numeral obtido pelo campo **fmtTxtValor**. A formatação desse campo resulta em um numeral separado das casas decimais por vírgula; daí a necessidade de trocar a vírgula por ponto, pois Java entende numéricos com casas decimais apenas separadas por ponto. Também se destaca a necessidade de formatação de data: **LocalDate.parse()** obtém o valor de data a partir de um texto; porém, é preciso informar ainda um padrão que esse método tem que obedecer ao analisar esse texto. Por isso, utiliza-se **DateTimeFormatter.ofPattern()**, passando por parâmetro o formato de data esperado (dia com dois dígitos/mês com dois dígitos/ano com quatro dígitos).

A persistência em si acontece com a instanciação do objeto **despesaDao** e a invocação do método **cadastrar()**, que foram implementados anteriormente. Teste sua aplicação clicando com o botão direito na classe **CadastroDespesa** e optando por

Run File, ou usando o atalho **Shift + F6**. Informe os dados, clique em **OK** e, após cadastrar, verifique o banco de dados para observar se a operação foi de fato bem sucedida.

Pronto! Sua primeira operação com interface gráfica está concluída. A seguir, você criará novas entidades para completar o sistema.

Implementando receitas

Uma vez que se tem o cadastro de despesas, é o momento de se trabalhar com o cadastro de receitas. Para essa aplicação, cada receita terá sua descrição, seu valor, a data e a conta para a qual foi destinado o valor. Cada conta terá a informação de nome e do banco no qual ela está. O *script* SQL para essa estrutura é o seguinte:

```
CREATE TABLE conta (  
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  nome VARCHAR(100),  
  banco VARCHAR(100)  
);  
  
CREATE TABLE receita(  
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  descricao VARCHAR(100),  
  valor DECIMAL(5,2),  
  conta_id int,  
  data DATE,  
  FOREIGN KEY (conta_id) REFERENCES conta(id)  
);
```

Note que há aqui um relacionamento “1:N”: para cada conta, N receitas e, em cada receita, exatamente uma conta. Execute o código anterior usando o MySQL Workbench e selecionando o banco de dados “financas”, criado anteriormente.

No projeto Java, você pode criar classes de entidade para essas tabelas dentro do pacote **br.com.senac.financasjpa.persistencia**. Primeiramente, criar a classe **Conta** é bastante simples:

```
package br.com.senac.financasjpa.persistencia;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Conta {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String nome;

    private String banco;

    public int getId() {
        return id;
    }

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

    public String getNome() {
        return nome;
    }

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

    public String getBanco() {
        return banco;
    }

    public void setBanco(String banco) {
        this.banco = banco;
    }
}
```

A classe **Conta** não apresenta nenhuma novidade e as mesmas anotações usadas em **Despesa** estão nessa classe (**@Entity**, **@Id**, **@GeneratedValue**).

Agora, na classe **Receita** há uma característica: a tabela correspondente a essa classe contém uma chave estrangeira. Represente esse relacionamento usando uma anotação **@ManyToOne** para que a JPA preencha o objeto de conta associado à receita. Observe a seguir.


```
package br.com.senac.financasjpa.persistencia;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import java.time.LocalDate;

@Entity
public class Conta {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String descricao;

    private String valor;

    @ManyToOne
    @JoinColumn(name="conta_id")
    private Conta conta;

    private LocalDate data;

    public int getId() {
        return id;
    }

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

    public String getDescricao() {
        return descricao;
    }

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

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public Conta getConta() {
```

```
        return conta;
    }

    public void setConta(Conta conta) {
        this.conta = conta;
    }

    public LocalDate getData() {
        return data;
    }

    public void setData(LocalDate data) {
        this.data = data;
    }
}
```

Note ainda que há uma anotação `@JoinColumn`. Ela é usada para identificar qual a coluna faz a referência na chave estrangeira – neste caso, é a coluna “`conta_id`” da tabela “`receita`” que referencia a tabela “`conta`”. Na realidade, essa anotação é até dispensável quando se usa o padrão “`<nome_tabela>_id`” (como em “`conta_id`”) em sua tabela, mas, caso a coluna não siga esse padrão de nomeação, é necessário usar a anotação.

Estando as entidades prontas, crie agora as classes DAO: `ContaDAO` e `ReceitaDAO` no pacote de persistência do projeto.

```
package br.com.senac.financasjpa.persistencia;

public class ContaDAO {

}
```

```
package br.com.senac.financasjpa.persistencia;

public class ReceitaDAO {

}
```

Antes de seguir com o cadastro de **Receita**, realize o desafio a seguir.

Implemente a tela de cadastro de conta. Será necessário criar um método **cadastrar(Conta c)** em **ContaDAO**, de maneira semelhante ao que foi feito em **DespesaDAO**. A janela deve conter campo para cada atributo da classe **Conta**, exceto **id**.

Primeiro implemente um método de persistência em **ReceitaDAO**:

```
package br.com.senac.financasjpa.persistencia;

import jakarta.persistence.EntityManager;

public class ReceitaDAO {

    public void cadastrar(Receita r){
        EntityManager em = JPAUtil.getEntityManager();
        try {
            em.getTransaction().begin();
            em.persist(r);
            em.getTransaction().commit();
        }catch(Exception e){
            em.getTransaction().rollback();
            throw e;
        }
        finally{
            JPAUtil.closeEntityManager();
        }
    }
}
```

Há poucas diferenças do método já implementado para **DespesaDAO**. Siga então para a criação da tela de cadastro. Crie um nome **JFrame** com nome “CadastroReceita” e um *layout* semelhante a este exemplo:

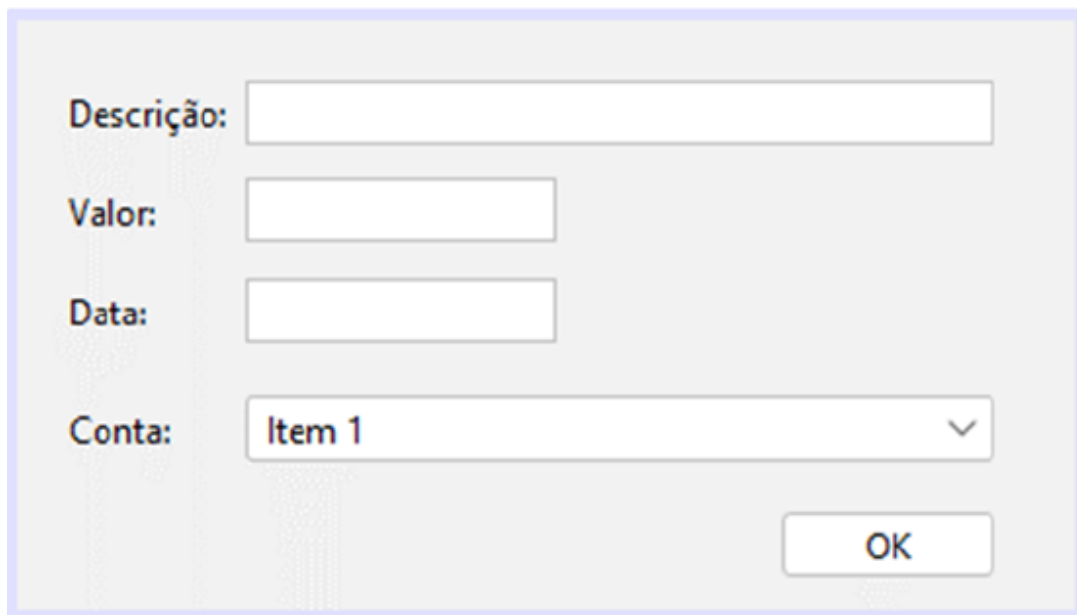
A imagem mostra uma janela de cadastro de receitas com um fundo cinza claro e uma borda azulada. Ela contém quatro campos de entrada: 'Descrição:' com um campo de texto longo; 'Valor:' com um campo de texto curto; 'Data:' com um campo de texto curto; e 'Conta:' com uma caixa de combinação que mostra 'Item 1' e uma seta para baixo. Abaixo dos campos, há um botão 'OK' com o texto em azul.

Figura 20 – Tela de cadastro de receitas

Fonte: NetBeans 13 (2022)

Os componentes presentes são:

JLabel: lblDescricao (texto “Descrição”), lblValor (texto “Valor:”), lblData (texto “Data:”), lblConta (texto “Conta:”)

JFormattedTextField: fmtValor (com propriedade “formatterFactory”, categoria “number” e formato “0.00”) e fmtData (com propriedade “formatterFactory”, categoria “date” e formato “short”)

JComboBox: cbConta

JButton: btnOK (texto “OK”)

A tela de cadastro é bastante semelhante ao cadastro de **Despesa**. Aqui, no entanto, há uma característica importante: é preciso preencher o *combobox* com as contas cadastradas no sistema. Primeiro, devem ser feitas algumas configurações no combo que está na tela: selecione o componente **cbConta**, veja as propriedades na aba **Code**, altere a propriedade “Type Parameters” para “<Conta>” ao invés de “<String>”.

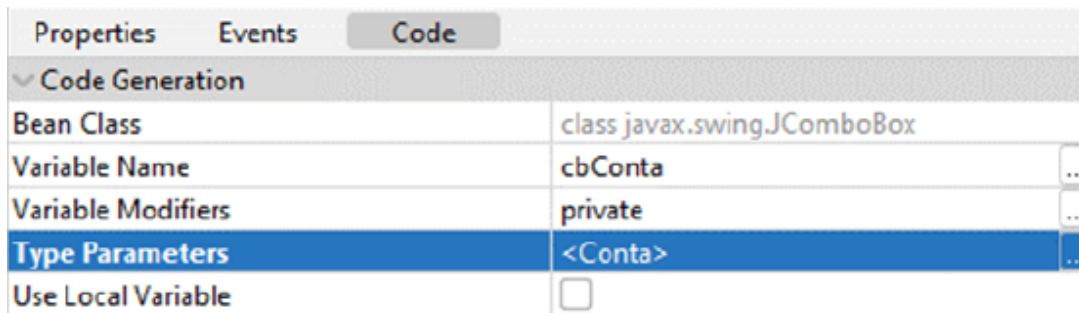


Figura 21 – Propriedade de código “Type Parameters” para o componente de combo

Fonte: NetBeans 13 (2022)

Depois, na aba **Properties**, clique no botão “...” da propriedade “model” e apague todos os itens (ou clique no botão **Restore to Default**).

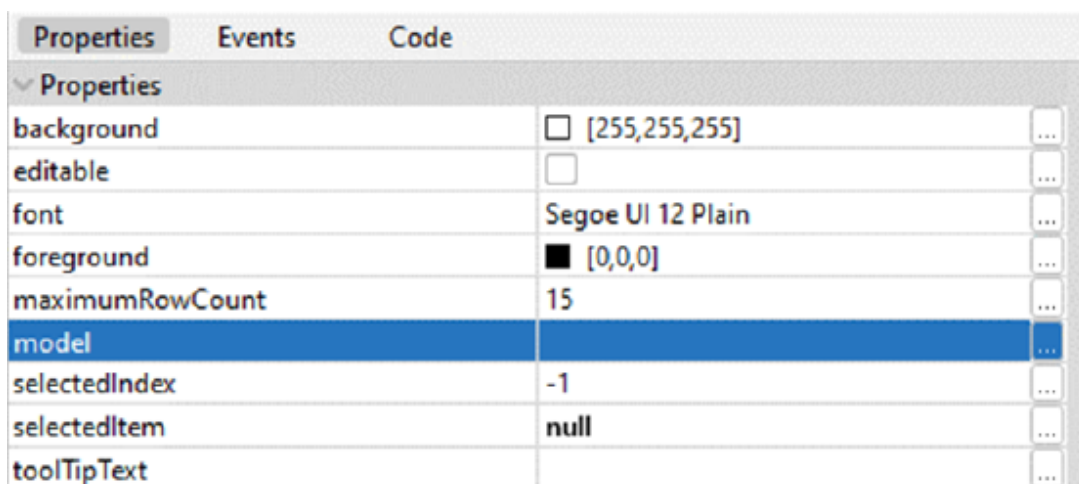


Figura 22 – Propriedade “model” do componente ComboBox

Fonte: NetBeans 13 (2022)

Para isso, em **ContaDAO**, inclua o método **listar()**. Esse será o seu primeiro contato com JPQL, a linguagem de consulta da JPA. Veja com atenção o código a seguir e os comentários contidos nele:

```
public List<Conta> listar(){
    EntityManager em = JPAUtil.getEntityManager();
    //resultado será devolvido nesta lista
    //usa import java.util.ArrayList;
    List<Conta> contas = new ArrayList<Conta>();
    try{
        //atenção: import jakarta.persistence.Query;
        Query consulta = em.createQuery("SELECT c FROM Conta c");
        //atenção: import java.util.List;
        contas = consulta.getResultList();
    }catch(Exception e){
        em.getTransaction().rollback();
        throw e;
    }
    finally{
        JPAUtil.closeEntityManager();
    }
    return contas;
}
```

O primeiro ponto a se observar é o uso de um objeto do tipo *query*, obtido com o método **createQuery()** de um objeto **EntityManager**. Nesse método, informa-se o seguinte comando JPQL:

```
SELECT c FROM Conta c
```

Note que a instrução JPQL é muito semelhante ao SQL. A grande diferença é que aqui se está lidando com entidades e não com tabelas e colunas. Por isso, tenha cuidado ao escrever “Conta” – a palavra deve estar com letra inicial maiúscula, idêntica ao nome da classe “Conta” (as palavras-chave “SELECT” e “FROM” independem de letras maiúsculas ou minúsculas). O **alias** “c” é usado como retorno do **SELECT**, um objeto completo ao invés de coluna a coluna de conta.

O método **getResultList()** da classe **Query** conclui a operação devolvendo uma lista com todos os objetos “Conta” encontrados no banco de dados. Você se lembra de que, com JDBC, era preciso preencher atributo por atributo de um objeto e incluí-lo em uma lista? A JPA já faz todo esse trabalho (enfadonho) por você!

Antes de prosseguir, faça um teste no método **main()** da classe **FinancasJPA** com o seguinte código:

```
ContaDAO cDao = new ContaDAO();
List<Conta> lista = cDao.listar();
for(Conta c : lista){
    System.out.println(c.getNome());
}
```

Concluído o método de listagem, utilize-o na inicialização da tela “**CadastroReceita**”. No código da tela, crie o seguinte método (preferencialmente depois do método construtor):

```
private void preencheComboConta(){
    ContaDAO contaDao = new ContaDAO();
    List<Conta> contas = contaDao.listar();

    for(Conta c : contas){
        cbConta.addItem(c);
    }
}
```

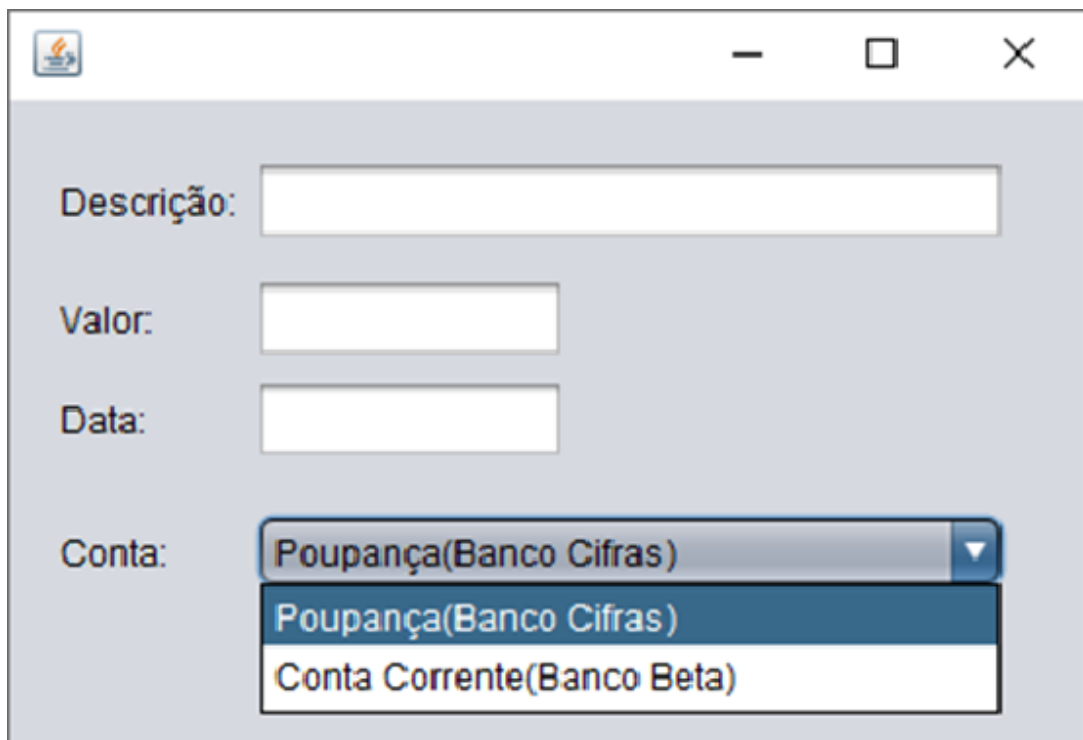
Em seguida, no construtor, faça a chamada desse método que você acabou de criar.

```
public CadastroReceita() {  
    initComponents();  
    preencheComboConta();  
}
```

Bom, um último ajuste deve ser feito na classe **Conta**, para que apareça no combo o nome e o banco da conta:

```
@Entity  
public class Conta {  
    /**  
     restante do código omitido por clareza*/  
  
    @Override  
    public String toString(){  
        return this.nome + "("+ this.banco + ")";  
    }  
}
```

Pronto! Agora, a tela apresentará todas as opções de conta cadastradas em seu banco de dados. Caso não tenha realizado o desafio anterior, insira alguns registros direto no banco para testar.



The screenshot shows a Java Swing window titled "CadastroReceita". Inside the window, there is a form with four labels and corresponding input fields:

- Descrição:** A text input field.
- Valor:** A text input field.
- Data:** A text input field.
- Conta:** A dropdown menu (JComboBox) with a blue border. The dropdown is open, showing three items:
 - Poupança(Banco Cifras)
 - Poupança(Banco Cifras)
 - Conta Corrente(Banco Beta)

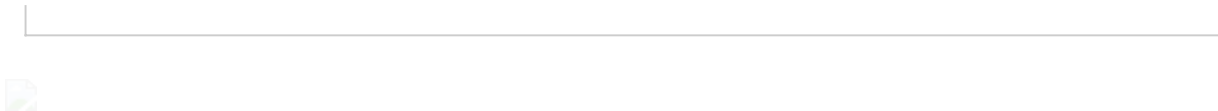
Figura 23 – Cadastro de receitas

Fonte: Senac EAD (2022)

Finalmente, implemente a ação de gravação da tela. Dê um clique duplo no botão **btnOK** para preencher o evento **btnOKActionPerformed()**. Observe o código e os comentários com atenção:

```
private void btnOKActionPerformed(java.awt.event.ActionEvent evt) {  
    Receita novaReceita = new Receita();  
    try{  
        //preenchendo valores, ajustando o que for necessário  
        novaReceita.setDescricao(txtDescricao.getText());  
        novaReceita.setValor(Double.parseDouble(fmtValor.getText().replac  
e(",", " ", "."))));  
        novaReceita.setData(LocalDate.parse(fmtData.getText(), DateTimeFo  
rmatter.ofPattern("dd/MM/y"))));  
  
        //obtendo a conta do combo-box  
        Conta contaSelecionada = (Conta) cbConta.getSelectedItemAt();  
        //preenchendo a referência de conta na receita  
        novaReceita.setConta(contaSelecionada);  
  
        //cadastrando de fato  
        ReceitaDAO receitaDao = new ReceitaDAO();  
        receitaDao.cadastrar(novaReceita);  
        //se foi sucesso, fecha a janela  
        this.dispose();  
    }catch(Exception e){  
        JOptionPane.showMessageDialog(this, "Ocorreu uma falha:\n" + e.ge  
tMessage());  
    }  
}
```

Note que, além dos atributos de dados comuns, foi preenchida a referência à conta de receita com um objeto que deverá ter, no mínimo, seu **id** preenchido. Faça testes preenchendo os campos e corrigindo possíveis erros. Com isso, conclui-se o cadastro de receitas. O próximo passo será o trabalho com listagens.



Definições de JPQL

Com JPA, a opção mais usual para recuperar dados é a linguagem JPQL, que é semelhante à SQL, mas independente de SGBD e tem uma sintaxe mais direcionada à orientação a objetos.

Você viu anteriormente o uso de JPQL na seguinte instrução:

```
Query consulta = em.createQuery("SELECT c FROM Conta c");
```

No exemplo, **Conta** é a entidade a ser buscada – é importante ter em mente que se está consultando entidades e não tabelas do banco de dados. A sintaxe mais básica para uma busca em JPQL é esta:

```
SELECT <alias> FROM <entidade><alias>
```

Palavras reservadas da sintaxe são insensíveis ao caso (letras maiúsculas e minúsculas), mas o nome da entidade não.

Para realizar a consulta, utiliza-se um objeto **Query** ou **TypedQuery<T>** (ambos do pacote **jakarta.persistence**), este último usado para quando se retorna apenas um registro na consulta. Esse objeto é obtido por **createQuery()** de **EntityManager**. Do objeto **Query** invoca-se o método **getResultList()** para obter todas as entidades encontradas pela busca ou **getSingleResult()** quando se deseja apenas um objeto (são esses os métodos que alcançam o banco de dados, transformando implicitamente o JPQL em SQL real). Confira exemplos:

```
EntityManager em = JPAUtil.getEntityManager();

//buscando todo os registros de Despesa
Query consulta1 = em.createQuery("select desp from Despesa desp");
List<Despesa> despesas = consulta1.getResultList();
for(Despesa d : despesas)
    System.out.println(d.getDescricao());

//buscando especificamente a Despesa de id 1
TypedQuery<Despesa> consulta2 = em.createQuery("SELECT d FROM Despesa d
WHERE id = 1", Despesa.class);
Despesa item = consulta2.getSingleResult();
System.out.println(item.getDescricao());
```

Nota: você pode testar os exemplos no método main() da classe principal do projeto.

Como você pôde ver, as consultas JPQL usam cláusulas **WHERE** para filtragem. Veja exemplos:

Despesas anteriores a 20/10/2022:

```
SELECT desp FROM Despesa desp WHERE desp.data
< '2022-10-20'
```

Receitas com descrição iniciada

```
em "sal" e realizadas após 2021 o SELECT r FROM Receita r
WHERE r.descricao LIKE 'sal%' AND r.data>= '2022-01-01'
```

Receitas com "id 1" ou valor superior a 100

```
SELECT r FROM Receita r WHERE id = 1 OR valor >
100
```

Receitas da conta 1

```
SELECT r FROM Receita r WHERE r.conta.id = 1
```

Para testar esses exemplos, use o código anterior proposto substituindo o argumento de **createQuery()** pelo *script* JPQL proposto. Também pode ser necessário alterar o tipo do **List<>** de acordo com o retorno (despesa, receita ou conta).

Note nessa última consulta que, para definir o **id** da conta, você se referirá à propriedade “id” do atributo “conta” da entidade “r”, e não “conta_id”, como seria no banco de dados. Você também poderia referenciar outros campos de conta, como no exemplo a seguir:

```
SELECT r FROM Receita r WHERE r.conta.nome = 'Poupança'
```

Na SQL, a consulta acabaria sendo mais complexa, usando **JOIN** ou subconsulta (o que a JPA acaba fazendo, mas nos bastidores).

A JPQL conta ainda com as cláusulas **ORDER BY** e **GROUP BY**. Confira os exemplos:

Despesas em ordem decrescente de data

```
SELECT d FROM Despesa d ORDER BY d.data DESC
```

Despesas em ordem crescente de nome e depois em valor

```
SELECT d FROM Despesa d ORDER BY d.descricao,  
d.valor
```

Valor médio de receitas agrupadas por conta

```
SELECT AVG(r.valor) FROM Receita r GROUP BY  
r.conta
```

Nota: neste último exemplo, a consulta resultará em uma lista de números reais (**List<Double>**).

O último exemplo apresentou o uso da função de agregação **AVG**, também presente na SQL, para calcular uma média. Além dela, em JPQL também se conta com **MAX** (maior valor), **MIN** (menor valor), **SUM** (somatório), **DISTINCT** (apenas resultados distintos), **COUNT** (contagem).

Outras funções de uso geral estão presentes, tais como:

Funções de texto: **SUBSTRING** (extrai trecho de texto), **CONCAT** (une dois textos), **TRIM** (retira espaço vazio do início ou do fim), **LENGTH** (número de caracteres em um texto), **UPPER** (deixa texto em letras maiúsculas) e **LOWER** (deixa texto em letras minúsculas)

Funções de data: **CURRENT_DATE** (dia atual), **CURRENT_TIME** (hora atual)

Funções aritméticas: **ABS** (módulo de um número), **SQRT** (raiz quadrada), **MOD** (resto de divisão), **SIZE** (tamanho de uma lista)

Parametrização com JPQL

No desenvolvimento de aplicações, obviamente não serão sempre usadas consultas fixas. Pelo contrário, o objetivo é informar dados indicados pelo usuário que influenciem os resultados da consulta. Para isso, pode-se montar a consulta dinamicamente (concatenando valores de variáveis) ou usar parâmetros nomeados, como no trecho a seguir (você pode aplicá-lo em **main()**).

```
Query consulta = em.createQuery("SELECT d FROM Despesa WH  
ERE d.data > :data");  
consulta.setParameter("data", LocalDate.of(2022, 6, 30));  
List<Despesa> despesas = consulta.getResultList();
```

Na primeira linha, especifica-se a consulta JPQL com um parâmetro “:data”, que será preenchido antes de a consulta ser realizada. O uso de “:” seguido de um nome é um método bastante eficaz para a definição de parâmetros dinâmicos para a consulta.

Na segunda linha do bloco de código, há o preenchimento desse parâmetro com o uso do método **setParameter()** de **Query**. Primeiro, informa-se o nome do parâmetro (sem o caractere ‘:’) e depois o valor desejado – claro que ele pode vir de uma variável preenchida por um argumento de método ou por uma entrada de dados por **Scanner**, por exemplo. Por fim, tem-se a concretização da consulta e o retorno dos dados, como usual.

Criando telas de consulta no projeto

Dado esse contexto sobre JPQL, volte agora ao projeto e implemente uma tela com listagem de despesas, contando com filtros para a busca. Para que você compreenda melhor do que precisará, primeiro crie a tela e depois cuide das operações de dados.

Crie um novo **JFrame Form** no pacote **br.com.senac.financasjpa.gui** com o nome “ListagemDespesa”. O *layout* proposto é o seguinte:

O diagrama mostra uma interface gráfica com dois painéis principais. O painel superior, intitulado "Filtros", contém campos de entrada para "Data:" (com dois campos separados por "a") e "Descrição:". Um botão "Pesquisar" está localizado no canto inferior direito deste painel. O painel inferior, intitulado "Dados", contém uma tabela com quatro colunas rotuladas "Title 1", "Title 2", "Title 3" e "Title 4".

Title 1	Title 2	Title 3	Title 4

Figura 24 – Tela de listagem de despesas

Fonte: NetBeans 13 (2022)

A tela conta com o seguinte:

Dois componentes **JPanel** para organização, um para os filtros acima e outro para os dados abaixo, ambos configurados com propriedade “border”, do tipo “Titled Border”. Títulos: “Filtros” e “Dados”.

Dois componentes **JFormattedTextField**: “fmtDataInicial” e “fmtDataFinal”, para receber os filtros de data

JTextField: txtFiltroDescricao

JLabels com valores “Data:” e “Descrição”

JButton: btnPesquisar com o texto “Pesquisar”

JTable: tblDespesa, sem configurações de colunas

Assim, espera-se que a sua pesquisa de dados retorne todos os registros de despesas se nenhum filtro for informado, ou que filtre por um intervalo de datas e/ou um trecho de descrição para as despesas desejadas. Com isso em mente, agora implemente a pesquisa em **DespesaDAO**, em que será criado o método **listar()**.

```
public List<Despesa> listar(){
    EntityManager em = JPAUtil.getEntityManager();
    List<Despesa> despesas = null;
    try{
        Query consulta = em.createQuery("SELECT d FROM Despesa d");
        despesas = consulta.getResultList();
    }finally{
        JPAUtil.closeEntityManager();
    }
    return despesas;
}
```

É importante lembrar que **Query** é uma classe do pacote **jakarta.persistence** (não confunda com **com.mysql.cj.Query** ou com **org.hibernate.query.Query**, que podem aparecer na sugestão do NetBeans) e **List** é do pacote **java.util.List** (não confunda com **java.awt.List**).

O código é simples e o mais relevante está dentro do bloco *try*, no qual se definiu uma consulta JPQL obtendo todas as despesas e o retorno para a lista “despesas”, o qual é feito pelo método.

Neste momento, você pode fazer testes no método **main()** da classe principal com um código semelhante ao seguinte:

```
DespesaDAO despDao = new DespesaDAO();
List<Despesa> despesas = despDao.listar();
for(Despesa d : despesas)
    System.out.println(d.getId() + " - " + d.getDescricao());
```

Agora, inclua parâmetros para o método **listar**, de maneira que se possa usar os filtros que estão presentes na tela. Iniciando com o filtro de descrição, pode-se imaginar a seguinte consulta JPQL:

```
SELECT d FROM Despesa d WHERE d.descricao LIKE ':descricao'
```

Bastaria, portanto, informar um valor para o parâmetro “:descrição”, usando ‘%’ no início e o fim do termo estar por conta do operador **LIKE**. No entanto, reflita: nem sempre o usuário informará um valor de filtro. Assim, você perderá sua consulta que traz todos os registros. Para resolver isso, você pode recorrer a uma construção dinâmica da consulta, como no trecho de código a seguir:

```
public List<Despesa> listar(String filtroDescricao){  
    ...  
    String textoQuery = "SELECT d FROM Despesa d";  
    if(!filtroDescricao.isEmpty()) //informou algum valor no parâmetro "f  
    iltroDescricao"  
        textoQuery = textoQuery + " WHERE d.descricao LIKE :descricao";  
  
    Query consulta = em.createQuery(textoQuery);  
    if(!filtroDescricao.isEmpty())  
        consulta.setParameter("descricao", filtroDescricao);  
    ...  
}
```

A construção fica um tanto complicada, pelas comparações necessárias para saber se o filtro foi informado ou não – apenas no caso de ter sido informado algum valor é que se deve incluir a cláusula **WHERE** e informar parâmetro à consulta JPQL. Essa é uma estratégia que gerará muito código e muita dificuldade de compreensão.

Ao invés disso, você pode usar uma estratégia que garanta que a cláusula **WHERE** esteja presente sem afetar resultados em que não se informe filtro. Observe abaixo:

```
SELECT d FROM Despesa d WHERE (:descricao is null OR d.descricao LIKE :  
descricao )
```

A consulta retornará resultado se o parâmetro informado for nulo ou se a descrição bater com o valor informado por parâmetro. É a condição “:descricao is null” que retira a necessidade de montagem condicional da consulta; se não for informada nenhuma descrição, essa condição será verdadeira e, por estar em uma expressão com “OR”, retornará “verdadeiro” e não afetará a consulta. Caso o parâmetro não seja nulo, a segunda expressão precisará ser verdadeira – no caso “d.descricao LIKE :descrição”.

Reveja o código do trecho anterior, no qual havia duas comparações sobre **filtroDescricao.isEmpty()**. Na nova maneira proposta, essa necessidade já não existe. No entanto, quando informar o parâmetro, você precisará realizar uma pequena verificação, como no exemplo a seguir (usando operador ternário):

```
consulta.setParameter("descricao", filtroDescricao.isEmpty() ? null :  
"%" + filtroDescricao + "%" );
```

Aqui, fica estabelecido que, se o usuário não informou texto de filtro, o valor do parâmetro será nulo; caso contrário, deve-se concatenar o caractere coringa “%” para o operador **LIKE** da consulta.

Antes de ajustar o código de **DespesaDAO**, expanda a consulta JPQL para incluir os outros filtros previstos para sua tela (data inicial e final):

```
SELECT d FROM Despesa d  
WHERE (:descricao is null OR d.descricao LIKE :descricao )  
AND (:dataInicial is null OR d.data >= :dataInicial)  
AND (:dataFinal is null OR d.data <= :dataFinal)
```

Você está usando a mesma estratégia aplicada para “:descricao”, mas agora com outros dois parâmetros, “:dataInicial” e “:dataFinal”. Para esses dois novos parâmetros, o preenchimento do valor do parâmetro é semelhante, com a diferença apenas de que se deve converter o valor recebido por texto para **LocalDate**:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/y");
consulta.setParameter("dataInicial", dataIni.isEmpty() ? null : LocalDate.parse(dataIni, formatter));
consulta.setParameter("dataFinal", dataFim.isEmpty() ? null : LocalDate.parse(dataFim, formatter));
```

O objeto **formatter** permitirá que a informação de data em texto seja compreendida ao convertê-la a **LocalDate**. Nesse caso, espera-se o formato “dia/mês/ano”.

Com tudo isso preparado e explicado, retorne finalmente ao código em **DespesaDAO**. O método **listar()** ganhará parâmetros e alterações significativas em seu código:

```

    public List<Despesa> listar(String filtroDescricao, String dataIni, String dataFim){
        EntityManager em = JPAUtil.getEntityManager();
        List despesas = null;
        try{
            String textoQuery = "SELECT d FROM Despesa d "+
                                " WHERE (:descricao is null OR d.descricao LIKE :descricao ) "+
                                " AND (:dataInicial is null OR d.data >= :dataInicial)"+
                                " AND (:dataFinal is null OR d.data <= :dataFinal)";

            Query consulta = em.createQuery(textoQuery);

            DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");

            consulta.setParameter("descricao", filtroDescricao.isEmpty() ? null : "%" + filtroDescricao + "%" );
            consulta.setParameter("dataInicial", dataIni.isEmpty() ? null : LocalDate.parse(dataIni, formatter));
            consulta.setParameter("dataFinal", dataFim.isEmpty() ? null : LocalDate.parse(dataFim, formatter));

            despesas = consulta.getResultList();
        }finally{
            JPAUtil.closeEntityManager();
        }
        return despesas;
    }

```

A consulta JPQL esquematizada anteriormente está sendo utilizada neste momento, considerando os filtros que a tela usará. Em seguida, informam-se os parâmetros, enviando **null** quando o argumento do método estiver vazio. O código ganhou alguma complexidade, mas imagine se, para cada parâmetro, você tivesse que incluir uma condição separada para montar o texto de consulta; sem dúvida, o código ficaria muito maior e complexo.

Recomenda-se que, antes de prosseguir, você teste esse novo método em **main()** da classe principal do projeto, informando nenhum, um, dois ou todos os parâmetros, como nos exemplos de invocação a seguir:

```
despesaDao.listar("", "", "");
despesaDao.listar("mercado", "", "");
despesaDao.listar("", "15/10/2022", "20/10/2022");
```

Depois de efetuar os testes, implemente a funcionalidade na tela **ListagemDespesa**, criada anteriormente. Dando um duplo clique no botão **btnPesquisar**, você é transportado ao evento de clique do botão, no qual usará o seguinte código: `private void btnPesquisarActionPerformed(java.awt.event.ActionEvent`

```
private void btnPesquisarActionPerformed(java.awt.event.ActionEvent ev
t) {
    try{
        DespesaDAO despesaDao = new DespesaDAO();
        List<Despesa> despesas =
            despesaDao.listar(txtFiltroDescricao.getText(), fmtTxtDataIni
cial.getText(), fmtTxtDataFinal.getText());

        preencheTabela(despesas);
    }catch(Exception e){
        JOptionPane.showMessageDialog(this, "Ocorreu uma falha:\n" + e.ge
tMessage());
    }
}
```

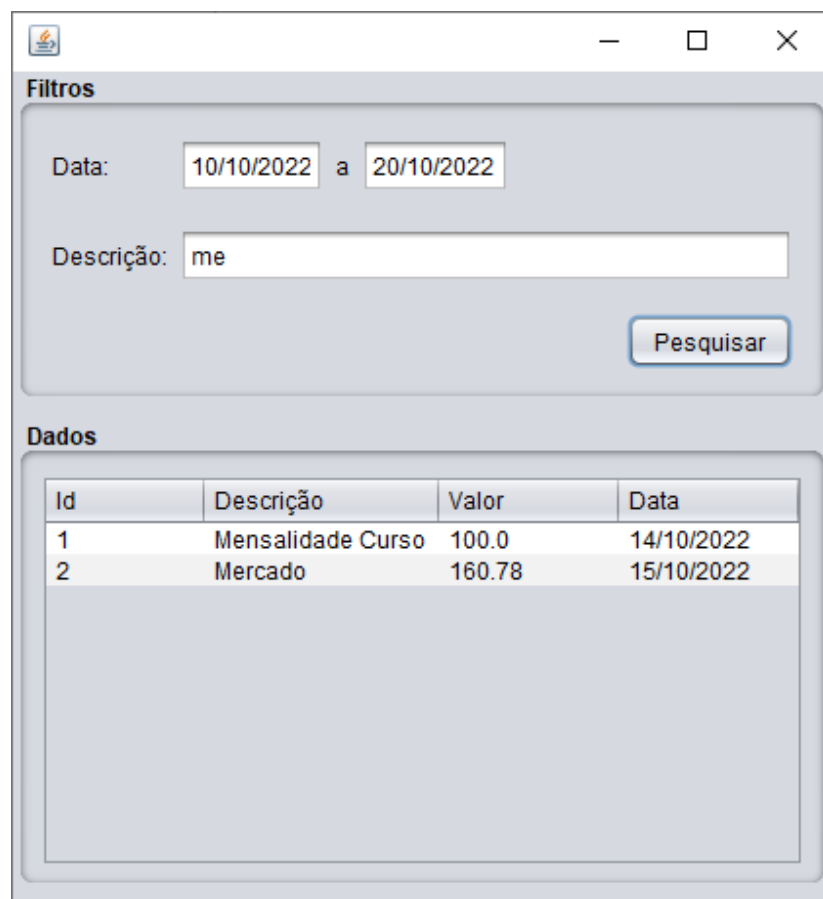
Esse trecho de código está realizando a consulta de **DespesaDAO** e solicitando preenchimento de tabela. O método **preencheTabela()** deve ser incluído na classe **ListagemDespesa**, com o seguinte código:

```
public void preencheTabela(List<Despesa> despesas){
    String columns[] = {"Id", "Descrição", "Valor", "Data"};
    String dados[][] = new String[despesas.size()][columns.length];

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/y");
    int i=0;
    for(Despesa d: despesas){
        dados[i] = new String[]{
            String.valueOf(d.getId()),
            d.getDescricao(),
            String.valueOf(d.getValor()),
            d.getData().format(formatter)};
        i++;
    }

    DefaultTableModel model = new DefaultTableModel(dados, columns);
    tblDespesa.setModel(model);
}
```

O código **preencheTabela** trata apenas dos passos necessários para popular de dados o componente **JTable** da tela. Estando esses dois métodos implementados, você pode testar sua aplicação clicando com o botão direito do *mouse* sobre **ListagemDespesa** e optando por **Run File**, ou usando o atalho **Shift + F6**.



The screenshot shows a web application window with a title bar containing a small icon, a minus sign, a maximize button, and a close button. The window is divided into two main sections: 'Filtros' (Filters) and 'Dados' (Data).

Filtros

Data: a

Descrição:

Dados

Id	Descrição	Valor	Data
1	Mensalidade Curso	100.0	14/10/2022
2	Mercado	160.78	15/10/2022

Figura 25 – Uso da listagem de despesa

Fonte: Senac EAD (2022)

Listagem de receitas e cláusula JOIN

Agora, você implementará a listagem de receitas em seu projeto. Primeiro, crie um método **listar()** na classe **ReceitaDAO**.

```
public List<Receita> listar(){
    EntityManager em = JPAUtil.getEntityManager();
    try{
        Query consulta = em.createQuery("SELECT r FROM Receita r");
        List<Receita> receitas = consulta.getResultList();
        return receitas;
    }finally{
        JPAUtil.closeEntityManager();
    }
}
```

A receita está associada à entidade **Conta**. Em SQL, você precisaria usar **JOIN** para retornar dados das duas tabelas; em JPQL, esse **JOIN** acontece, muitas vezes, implicitamente.

Se você precisasse de todas as contas que estão associadas a receitas, poderia usar esta consulta:

```
SELECT r.conta FROM Receita r
```

Ou também poderia usar esta consulta:

```
SELECT c FROM Receita r JOIN Conta c
SELECT c FROM Receita r JOIN Conta
c
```

Como a entidade **Receita** deixa explícito um relacionamento com **Conta** (veja no código da classe **Receita** a anotação **@ManyToOne**), você pode usar a primeira opção citada, sem referenciar a entidade **Conta** diretamente. Para deixar o código mais simples e compreensível, recomenda-se usar a segunda alternativa, que deixa clara ao programador a junção entre as entidades.

Além de **JOIN**, JPQL conta com **LEFT JOIN**, sendo, nesse caso, necessário uso explícito na consulta.

Voltando ao projeto, implemente a tela **ListagemReceita** criando um **JFrame Form** no pacote **br.com.senac.financasjpa.gui** e monte uma tela semelhante à seguinte.

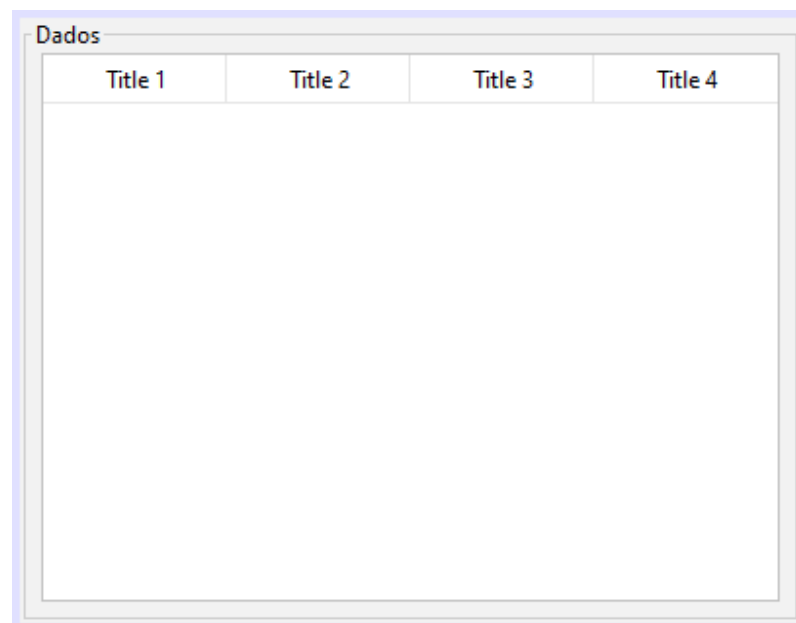


Figura 26 – Tela listagem de receitas

Fonte: NetBeans 13 (2022)

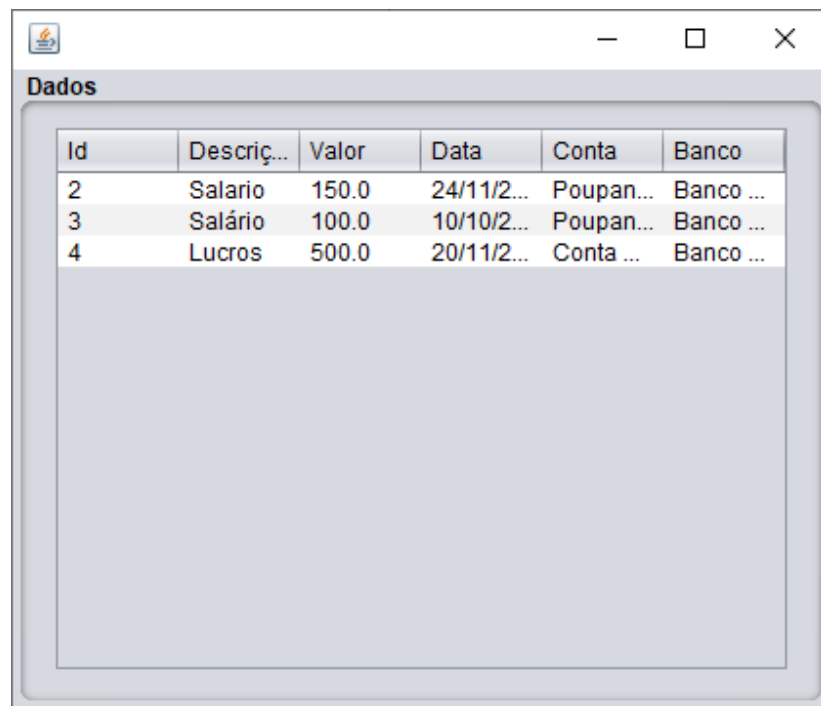
A tela conta apenas com um **JPanel** configurado com borda e título, e um componente **JTable** com nome de variável “tblReceitas”. Alterando o modo da classe para **Source** no NetBeans, implemente o preenchimento dessa tabela, alterando o construtor da classe, como no exemplo:

```
public ListagemReceita() {  
    initComponents();  
  
    ReceitaDAO receitaDao = new ReceitaDAO();  
    List<Receita> receitas = receitaDao.listar();  
    preencheTabela(receitas);  
}
```

E implementando o método **preencheTabela()**.

```
public void preencheTabela(List<Receita> receitas){  
    String colunas[] = {"Id", "Descrição", "Valor", "Data", "Conta", "Banco"};  
    String dados[][] = new String[receitas.size()][colunas.length];  
  
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/y");  
    int i=0;  
    for(Receita r: receitas){  
        dados[i] = new String[]{  
            String.valueOf(r.getId()),  
            r.getDescricao(),  
            String.valueOf(r.getValor()),  
            r.getData().format(formatter),  
            r.getConta().getNome(),  
            r.getConta().getBanco()  
        };  
        i++;  
    }  
  
    DefaultTableModel model = new DefaultTableModel(dados, colunas);  
    tblReceitas.setModel(model);  
}
```

Execute a classe com **Shift + F6** e observe o resultado.



Id	Descriç...	Valor	Data	Conta	Banco
2	Salario	150.0	24/11/2...	Poupan...	Banco ...
3	Salário	100.0	10/10/2...	Poupan...	Banco ...
4	Lucros	500.0	20/11/2...	Conta ...	Banco ...

Figura 27 – Listagem de receitas em ação

Fonte: Senac EAD (2022)

Implemente filtro de intervalo de data, filtro de descrição e filtro de conta para a listagem de receitas.

Consultas com Criteria

Criteria é um conjunto de métodos que formam uma alternativa à JPQL. A vantagem em usar Criteria é o fato de que falhas nas consultas se tornam detectáveis em tempo de compilação, diferentemente do que ocorre com a JPQL. Isso significa que, se você errar o nome de uma entidade em JPQL, só perceberá o problema quando executar a aplicação, já em Criteria, a falha aconteceria na hora da compilação. A desvantagem de Criteria é sua complexidade. Confira abaixo uma estrutura geral para uma consulta simples com Criteria:

```
//preparação do Criteria
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Receita> cQuery = cBuilder.createQuery(Receita.class);
Root<Receita> from = cQuery.from(Receita.class);
cQuery.select(from);
//executando a consulta
TypedQuery<Receita> consulta = em.createQuery(cQuery);
List<Receita> receitas = consulta.getResultList();
```

Os passos são numerosos:

1. Crie uma fábrica para Criteria com **getCriteriaBuilder()** de **EntityManager**.
2. Crie uma consulta Criteria (classe **CriteriaQuery**) para a entidade com o método **createQuery()** da classe **CriteriaBuilder**.
3. Crie uma “raiz” para a consulta (classe **Root<>**) com o método **from()** de **CriteriaQuery**.
4. Defina a seleção (o retorno da consulta) com o método **select()** de **CriteriaQuery** e usando a raiz criada anteriormente.

5. Realize a consulta passando o objeto **CriteriaQuery** ao invés de um texto JPQL.

Geralmente, esse será um padrão para qualquer consulta Criteria. O código anterior é correspondente à seguinte consulta JPQL:

```
SELECT r FROM Receita r
```

A filtragem também acontece por meio de métodos da API Criteria. Na consulta feita anteriormente, se você quisesse filtrar para trazer apenas a receita de “id = 2”, teria o seguinte código:

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Receita> cQuery = cBuilder.createQuery(Receita.class);
Root<Receita> from = cQuery.from(Receita.class);
cQuery.select(from);
cQuery.where(cBuilder.equal(from.get("id"), 2));
```

Observe a linha destacada, na qual se definiu a condição com o método **where()** do objeto da classe **CriteriaQuery**. O método **equal()** de **CriteriaBuilder** define a comparação de igualdade. Você também conta com os métodos **gt()** e **lt()** para comparação de maior e menor, **ge()** e **le()** para maior ou igual e menor ou igual, respectivamente, **isNull()** para verificação de nulo, **like()** para pesquisa em texto, **notEqual()** para comparação de valores diferentes entre outros.

Se você quiser incluir cláusulas **AND** e **OR**, precisará criar vários predicados (correspondentes às comparações) e combiná-los, como no exemplo a seguir (o trecho a seguir substituiria o destacado no trecho de código anterior):

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Receita> cQuery = cBuilder.createQuery(Receita.class);
Root<Receita> from = cQuery.from(Receita.class);
cQuery.select(from);
```

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Receita> cQuery = cBuilder.createQuery(Receita.class);
Root<Receita> from = cQuery.from(Receita.class);
cQuery.select(from);
Query.where(cBuilder.equal(from.get("id"), 2));

Predicate p1 = cBuilder.equal(from.get("id"), 2);
Predicate p2 = cBuilder.gt(from.get("valor"), 100);
cQuery.where(cBuilder.or(p1, p2));
```

Primeiro, defina para o **Predicate p1** uma comparação de igualdade; depois, para o **p2**, uma comparação de “maior que”, verificando o campo de valor de receita e comparando-o com o valor 100. Por fim, combine com o método **or()** de **CriteriaBuilder**. A classe conta também com o método **and()** para o operador “E”.

Pela complexidade e verbosidade na construção das consultas, geralmente é preferível utilizar a JPQL.

Utilizando o método **like()** de **CriteriaBuilder**, crie uma consulta Criteria para obter contas registradas no banco de dados, de acordo com um filtro de texto para a descrição (o filtro você pode preencher com um valor fixo ou com um valor obtido por **Scanner**).

Exclusão de dados

Agora, você implementará a exclusão de dados de **Despesas**. Primeiramente, em **DespesaDAO**, inclua um novo método **excluir()**.

```
public void excluir(int id){
    EntityManager em = JPAUtil.getEntityManager();
    try{
        Despesa d = em.find(Despesa.class, id);
        if(d != null){
            em.getTransaction().begin();
            em.remove(d);
            em.getTransaction().commit();
        }
    }catch(Exception e){
        em.getTransaction().rollback();
        throw e;
    }
    finally{
        JPAUtil.closeEntityManager();
    }
}
```

No método, você primeiro busca o registro que deseja excluir (a partir de seu **id**) e depois o remove no repositório. É importante lembrar que, nessa operação, você deve usar transações: **commit()**, quando tudo estiver OK, ou **rollback()**, caso tenha havido falha.

Na tela **ListagemDespesa**, inclua um novo botão **JButton** “btnExcluir” e, clicando duas vezes, edite seu evento de clique.

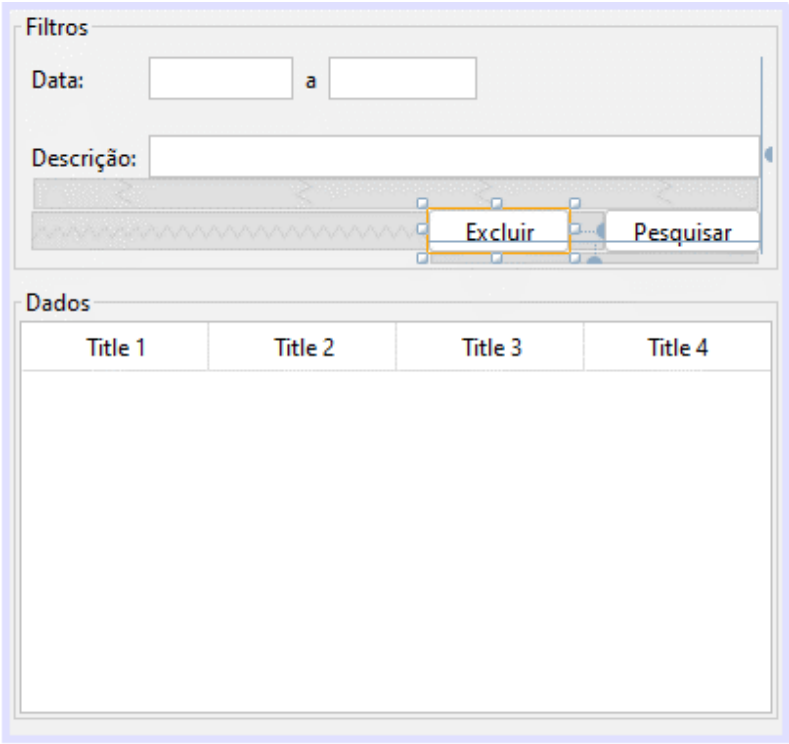


Figura 28 – Incluindo o botão **Excluir**

Fonte: NetBeans 13 (2022)

```
private void btnExcluirActionPerformed(java.awt.event.ActionEvent evt)
{
    try{
        if(tblDespesa.getSelectedRow() >= 0){ //verifica se há algo selecionado na tabela
            //obtem o valor da coluna id da linha selecionada
            String id = (String)tblDespesa.getValueAt(tblDespesa.getSelectedRow(), 0);
            //janela de confirmação
            int resposta = JOptionPane.showConfirmDialog(this, "Deseja mesmo excluir o registro " + id + "?");
            if(resposta == 0)//0- yes, 1- no, 2- cancel
            {
                //realizando a exclusão
                DespesaDAO despesaDao = new DespesaDAO();
                despesaDao.excluir(Integer.parseInt(id));
                JOptionPane.showMessageDialog(this, "Registro excluído com sucesso");
                //refazendo a pesquisa para atualizar a tabela na tela
                btnPesquisarActionPerformed(evt);
            }
        }
    }catch(Exception e){
        JOptionPane.showMessageDialog(this, "Ocorreu uma falha:\n" + e.getMessage());
    }
}
```

O método obtém o valor da coluna “id” da linha selecionada na tabela e invoca o método **excluir()** de **DespesaDAO**, passando por parâmetro esse “id” obtido.

Atualizando dados

A operação de atualização de dados de despesas exigirá um trabalho um pouco maior. Primeiro, em **DespesaDAO**, crie dois novos métodos:

```
public Despesa obter(int id){
    EntityManager em = JPAUtil.getEntityManager();
    try{
        return em.find(Despesa.class, id);
    }finally{
        JPAUtil.closeEntityManager();
    }
}

public void atualizar(Despesa d){
    EntityManager em = JPAUtil.getEntityManager();
    try {
        em.getTransaction().begin();
        em.merge(d);
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
        throw e;
    }
    finally{
        JPAUtil.closeEntityManager();
    }
}
```

O método **obter()** será útil para recuperar o registro selecionado na tela de listagem de despesas. O método **atualizar()**, por sua vez, é responsável por salvar as informações no banco de dados usando o método **merge()** de **EntityManager**. A intenção agora é usar a mesma tela de cadastro para realizar a edição; por isso, ajuste o código da classe **CadastroDespesa** incluindo o seguinte trecho após o construtor:

```
//atributo para controlar se está em edição ou em cadastro
private Despesa despesaEdicao = null;

public void preencheEdicao(Despesa d){
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/
y");
    fmtTxtData.setText(d.getData().format(formatter));
    fmtTxtValor.setText(String.valueOf(d.getValor()));
    txtDescricao.setText(d.getDescricao());

    despesaEdicao = d;
}
```

O atributo “despesaEdicao” será útil na ação do botão **Salvar** para identificar se você está criando novo registro ou atualizando um. O método **preencheEdicao()** será usado após a criação da tela para preencher os campos com os dados da **Despesa** selecionada. Note que “despesaEdicao”, que tem valor padrão **null**, é preenchido aqui com o objeto recebido por parâmetro.

Agora, edite **ListagemDespesa** primeiro incluindo um novo botão **btnEditar** e, em seguida, dando duplo clique para implementar a ação do botão.

Figura 29 – Incluindo botão **Editar** na tela de listagem de despesa

Fonte: NetBeans 13 (2022)

```
private void btnEditarActionPerformed(java.awt.event.ActionEvent evt) {  
    try{  
        //criando nova tela de cadastro  
        CadastroDespesa cadastro = new CadastroDespesa();  
  
        if(tblDespesa.getSelectedRow() >= 0){ //há linha selecionada na t  
abela?  
            String id = (String)tblDespesa.getValueAt(tblDespesa.getSelec  
tedRow(), 0);  
            //obtendo o objeto Despesa do id selecionado...  
            DespesaDAO despesaDao = new DespesaDAO();  
            Despesa despesaSelecionada = despesaDao.obter(Integer.parseIn  
t(id));  
  
            //...e o repassando para a tela de cadastro  
            cadastro.preencheEdicao(despesaSelecionada);  
            cadastro.setVisible(true);  
        }  
    }catch(Exception e){  
        JOptionPane.showMessageDialog(this, "Ocorreu uma falha:\n" + e.ge  
tMessage());  
    }  
}
```

Ao clicar no botão “Editar”, crie um novo **JFrame** de **CadastroDespesa** e informe os dados do registro que você quer editar. Por último, você precisa agora ajustar o evento do botão **btnSalvar** de **CadastroDespesa** da seguinte forma:

```

private void btnSalvarActionPerformed(java.awt.event.ActionEvent evt) {
    Despesa novaDespesa = new Despesa();
    if(despesaEdicao != null)
        novaDespesa = despesaEdicao;

    try{
        //para a descrição capturamos direto o valor do campo de texto
        novaDespesa.setDescricao(txtDescricao.getText());
        //substitui , por . para que fique no formato numérico
        //que o Java entende como double
        novaDespesa.setValor(Double.parseDouble
        (fmtTxtValor.getText().replace(",", ".")));
        //classe DateFormatter necessária para obter o tipo
        //LocalDate a partir de um texto
        novaDespesa.setData(LocalDate.parse(fmtTxtData.getText(),
        DateTimeFormatter.ofPattern("dd/MM/y"))));

        DespesaDAO despesaDao = new DespesaDAO();
        if(despesaEdicao == null)
            despesaDao.cadastrar(novaDespesa);
        else
            despesaDao.atualizar(novaDespesa);

        //fechando a tela após o cadastro
        this.dispose();
    }catch(Exception e){
        JOptionPane.showMessageDialog(this,
        "Ocorreu uma falha:\n" + e.getMessage());
    }
}

```

O trecho atualizado no método está em destaque. Primeiro, se há um objeto de edição nessa tela, então passa-se a referência dele ao **novaDespesa**, que será preenchido com os dados presentes nos campos da tela. Ao fim, no ato da gravação, opte por cadastrar nova despesa se o objeto **despesaEdicao** for nulo, ou por atualizar o registro, caso contrário.

Realize testes rodando a classe **ListagemDespesa**.

Implemente as operações de exclusão e de edição para **Receitas**.

Finalizando a aplicação

Para finalizar seu projeto, crie no pacote **br.com.senac.financasjpa.gui** um novo **JFrame Form** e nele inclua um componente “Menu Bar”, edite os itens padrão para **Despesa** e **Receita** e inclua itens nesses menus (“Cadastro” e “Listagem” para cada menu). Para incluir item de menu, clique com o botão direito do *mouse* sobre o menu e selecione **Add From Pallete > Menu Item**.

Figura 30 – Incluindo item de menu na tela principal

Fonte: NetBeans 13 (2022)

Com um clique duplo em cada item, você implementa as criações das telas de cadastro e da listagem, como no código a seguir:


```

        private void miCadastroDespesaActionPerformed(java.awt.event.ActionEvent
        evt) {
            CadastroDespesa cadastroDespesa = new CadastroDespesa();
            cadastroDespesa.setVisible(true);
        }

        private void miListagemDespesaActionPerformed(java.awt.event.ActionEvent
        evt) {
            ListagemDespesa listagemDespesa = new ListagemDespesa();
            listagemDespesa.setVisible(true);
        }

        private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt)
        {
            CadastroReceita cadastroReceita = new CadastroReceita();
            cadastroReceita.setVisible(true);
        }

        private void jMenuItem2ActionPerformed(java.awt.event.ActionEvent evt)
        {
            ListagemReceita listagemReceita = new ListagemReceita();
            listagemReceita.setVisible(true);
        }

```

Você precisa ainda configurar **CadastroDespesa**, **CadastroReceita**, **ListagemDespesa** e **ListagemReceita** para que não encerrem a aplicação ao clicar no botão fechar. Para isso, no modo **Design**, selecione o objeto **JFrame** e, na aba **Properties**, altere a propriedade “defaultCloseOperation” para “DISPOSE” ao invés de “EXIT_ON_CLOSE” (apenas “Principal” manterá essa propriedade como “EXIT_ON_CLOSE”).

Por fim, substitua o conteúdo de **main()** da classe principal para que ela crie e exiba a janela “Principal”.

```
public class FinancasJPA {  
  
    public static void main(String[] args) {  
        java.awt.EventQueue.invokeLater(new Runnable() {  
            public void run() {  
                new Principal().setVisible(true);  
            }  
        });  
    }  
}
```

Com isso, você conclui sua implementação de um sistema de registro de receitas e despesas.

Encerramento

Você aprendeu neste conteúdo as possibilidades de interagir com diversos bancos de dados sem precisar escrever SQL. Dependendo do banco, é possível omitir até mesmo os comandos de criação do banco e das tabelas.

Não há juízo de valor se desta forma é melhor ou pior, cada programador, ou projeto, definirá a maneira de usar o que mais se encaixa ao seu modelo de trabalho e ao projeto. Você deve testar e verificar qual maneira funciona melhor para você e para seu projeto, e até mesmo ficar de olho nas novas maneiras que surgem todos os dias e identificar as melhores formas de lidar com o Java e os bancos de dados.