Transcrição

Já temos tudo configurado, já criamos o nosso projeto Maven adicionando o Hibernate e o banco de dados H2 como dependência, criamos o `persistence.xml`, configuramos o nosso banco de dados, as propriedades do JDBC, da JPA, do Hibernate e mapeamos a nossa entidade `Produto` com a tabela "produtos" no banco de dados. Enfim, está tudo pronto e podemos começar a persistir, carregar e fazer toda a manipulação desses objetos no banco de dados.

No vídeo de hoje aprenderemos, justamente, como fazemos para cadastrar um produto no banco de dados, isto é, se quisermos inserir um objeto produto no banco de dados - na tabela de produtos - como funcionará? Vamos criar uma nova classe e colocar esse código nela. Então selecionaremos o comando "Ctrl + N", depois, "Class" e, por fim, "Next".

Na próxima tela, trocaremos o pacote. No lugar de "modelo" escreveremos "testes", isto é, "br.com.alura.loja.testes", sendo que "testes" se refere à classe onde faremos os testes de acesso ao banco de dados, e o nome da classe será "CadastroDeProduto". Agora basta apertar "Finish" e termos criado a classe `CadastroDeProduto`. Dentro dela, geraremos um método `main` escrevendo "main" e apertando "Ctrl + Barra de espaço".

```
package br.com.alura.loja.testes;

public class CadastroDeProduto {

   public static void main(String[] args) {

     }
```

```
}
```

COPIAR CÓDIGO

Agora, vamos imaginar que temos um produto. Vamos criá-lo escrevendo `Produto celular = new Produto()` (portanto, o atributo é "celular"). Após termos feito o *import* da classe produto, vamos setar as propriedades desse produto. Então, `celular.setNome();` , vamos imaginar que seja um celular da Xiaomi, logo `celular.setNome("Xiaomi Redmi");` .

Prosseguindo, faremos `celular.setDescricao("Muito legal");` e `celular.setpreco(new BigDeCimal("800");` (sendo que "800" se refere ao preço em reais), agora basta apertarmos "Ctrl + Shift + O" para importar o `BigDecimal` .

```
    public static void main(String[] args) {
          Produto celular = new Produto();
              celular.setNome("Xiaomi Redmi");
              celular.setDescricao("Muito legal");
              celular.setPreco(new BigDecimal("800"));

      }

}
```

COPIAR CÓDIGO

Portanto temos, no Java, o nosso objeto produto. Nós o instanciamos e temos todas as informações preenchidas. Estamos com uma classe com método `main` , mas, em um sistema real, essas informações seriam preenchidas por um usuário. Existiria uma tela com os campos para ele preencher e, no Java, instanciaríamos os objetos e setaríamos as informações conforme o que o usuário digita na tela.

Agora precisamos descobrir como pegar o objeto `celular` e fazer o *insert* na tabela de "produtos". Como isso funcionará na JPA? No JDBC, toda a integração com o banco de dados era feita com uma classe chamada *connection*, nós precisávamos

abrir uma conexão e, a partir dela, fazer todo o trabalho para acessar o banco de dados.

Na JPA, tem algo parecido, que não é bem uma conexão, mas uma interface que faz a ligação do Java com o banco de dados, que é uma interface chamada `EntityManager`. Essa classe funciona como se fosse o gerente, o "*manager*" das entidades, ou ainda, o gestor das entidades.

Toda vez que desejarmos acessar o banco de dados, seja para salvar, excluir, atualizar, carregar, fazer um *select*, ou qualquer outra operação que quisermos fazer no banco de dados com a JPA, nós utilizaremos a interface `EntityManager`.

Vamos criar uma variável, que, no nosso caso, chamaremos de `em`. Para instanciar um `EntityManager`, em teoria, seria `new EntityManager()`. Mas, temos um problema: `EntityManager` não é uma classe, é uma interface e por isso não podemos dar `new`, o certo seria dar `new` numa classe que implementa a interface.

Na JPA, não criamos manualmente o `EntityManager`. NA JPA, o padrão de projeto utilizado é o *factory*. Assim, existe uma *factory* de `EntityManager`. Para criar o `EntityManager`, precisamos do `EntityManagerFactory`, ele tem o método que faz a construção do `EntityManager`.

Então, antes de criar o `EntityManager`, precisamos criar outro objeto, que é o `EntityManagerFactory`. Nos padrões de projeto, "*design patterns*", existe esse padrão de projeto chamado *factory*, e, há uma *factory* para isolar a criação do `EntityManager`.

```
    EntityManagerFactory factory =
    EntityManager em =
    }

}
```

COPIAR CÓDIGO

Então, precisamos criar o `EntityManagerFactory`. Nós temos uma variável `EntityManagerFactory` e a chamamos de `factory`. Em teoria, continuaríamos fazendo `new EntityManagerFactory`, mas não é assim. Outra classe foi criada na JPA e se chama `Persistence`, e ela tem um método estático chamado `CreateEntityManagerFactory`. Então, basta chamar `Persistence.createEntityManagerFactory()`

```
    EntityManagerFactory factory = Persistence.createEntityManagerFactory()
    EntityManager em =
    }

}
```

COPIAR CÓDIGO

O método `CreateEntityManagerFactory` está esperando um parâmetro que é uma `String`. Essa `String` é o nome do `persistence-unit`. Vamos recordar o `persistence.xml`. Nós tínhamos nele a tag `persistence-unit`, onde imaginamos que ela fosse como um banco de dados. Vamos recordar também que, nessa tag, tínhamos o atributo `name="loja"`. Então, é esse nome que passamos para o método `CreateEntityManagerFactory`.

Se tivéssemos vários bancos de dados na aplicação, teríamos várias tags `persistence-unit`, cada uma com um `name` distinto, e, na hora de criar a *factory*, passaríamos qual é o `persistence-unit`. Desta maneira, a JPA fica sabendo com qual banco ela deve se conectar. Portanto, temos que adicionar o nome do `persistence-unit`, que, no nosso caso, é `"loja"`.

```
    EntityManagerFactory factory = Persistence.
        createEntityManagerFactory("loja");
    EntityManager em =
    }

}
```

COPIAR CÓDIGO

Agora vamos importar a classe `EntityManagerFactory` e ela virá do pacote `javax.persistence`. Então, criamos a *factory* e podemos criar um `EntityManager` chamando `factory.createdEntityManager()`, e um objeto do tipo `EntityManager` será devolvido.

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

    EntityManager em = factory.createEntityManager();
    }

}
```

COPIAR CÓDIGO

Já temos o `EntityManager` criado e podemos trabalhar com ele. O que queremos fazer é pegar o objeto `Produto`, que está na variável `celular`, e fazer um *insert* no banco de dados, ou seja, queremos inserir um novo registro no banco de dados. Para isso, no objeto `EntityManager()` existe um método chamado `persist()`.

Existem vários métodos que veremos ao longo do curso, mas o método `persist()` serve para persistir, salvar e inserir um registro no banco de dados. Precisamos também passar quem é o objeto, no caso, `celular`.

```
  EntityManager em = factory.createEntityManager();
  em.persist(celular);
    }

}
```

COPIAR CÓDIGO

Terminado, ele fará o *insert*. Podemos nos perguntar em qual tabela ele fará o *insert*. Ele já sabe que é a tabela de `Produto`, pois, o objeto `celular` é do tipo `Produto`, e `Produto` é uma entidade, então, pela entidade, ele fica sabendo de tudo: qual é a tabela, quais são as colunas, quem é a chave primária, como a chave primária é gerada.

Por isso, não precisamos informar nada, basta dizer: `EntityManager`, persista a entidade `celular`. Vá à entidade, na classe dela, e descubra tudo. Portanto, o `EntityManager` fará a ligação para transformar a entidade `Produto` em uma linha na nossa tabela do banco de dados.

A princípio, está pronto o código. Vamos rodar a classe. Com o botão direito, abriremos um atalho e nele selecionaremos "Run As > 1 Java Application". Agora vamos olhar o Console, e, ao que parece, ele rodou sem nenhum erro. Aparecem alguns *logs*, em vermelho, que se assemelham a erros, mas, por padrão, ele imprime em vermelho. Está tudo correto, são apenas *logs* da JPA.

Como conseguimos saber se ele salvou ou não, já que não imprimiu nada? Precisamos nos lembrar das propriedades do `persistence.xml`, porque existem algumas que são utilitárias e que podemos utilizar aqui. Outra propriedade que podemos utilizar do Hibernate é a `hibernate.show_sql` e, no `value`, passamos `true`.

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

COPIAR CÓDIGO

Então, usaremos essa propriedade para falar: Hibernate, toda vez que você gerar um SQL e for ao banco de dados, imprima no Console para mim, por favor. Se quisermos ver o que ele está rodando no banco de dados, habilitamos essa propriedade e conseguimos ver o *insert*, *select*, *delete* enfim, tudo o que está acontecendo no banco de dados, já que não somos nós que geramos o comando do SQL.

É o Hibernate que faz o *insert* automaticamente baseado nas configurações da entidade. Esta é uma facilidade em relação ao JDBC. No JDBC, precisávamos montar a SQL manualmente, agora não precisamos mais fazer isso. Vamos rodar novamente,

porque, na teoria, é para ele imprimir um *insert*, mas, ele não gerou um *insert* no final. Ou seja, ele não salvou a nossa entidade no banco de dados.

No `persistence.xml`, na tag `persistence-unit`, além do `name`, nós temos o `transaction-type`. Nós até comentamos anteriormente que temos dois valores `"RESOURCE_LOCAL"` ou `"JTA"`. O `"JTA"` é indicado para se estivermos em um servidor de aplicação, que controla a transação.

Mas, esse não é o nosso caso, estamos como `"RESOURCE_LOCAL"`, ou seja, não temos o controle de transação automático, por isso, ele não fez o *insert*, porque não delimitamos uma transação. Portanto, ele não começou uma transação e não dispará um insert no banco de dados. Antes de fazer o `pesist`, temos que chamar `em.getTransaction().begin();`.

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

    em.getTransaction().begin();
    em.persist(celular);
       }

  }
```

COPIAR CÓDIGO

É como se disséssemos ao JPA e ao `EntityManager` que pegassem a transação `begin()` e a iniciasse. Dentro dela, rodaremos quais são as operações. No nosso caso, é apenas uma, o `persist()`. Terminado, temos que commitar essa transação no banco de dados, `em.getTransaction().commit();`.

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

    em.getTransaction().begin();
    em.persist(celular);
```

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

}
```

COPIAR CÓDIGO

Então, fizemos `getTransaction().begin();`, depois `em.persist(celular);` referente ao que queremos fazer de operações, no nosso caso, é apenas uma, e, depois de terminado, fizemos o `commit()`. Um detalhe importante é que, depois de usar `EntityManager`, precisamos finalizar com `em.close();`, para que o recurso não fique aberto.

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

em.getTransaction().begin();
em.persist(celular);
em.getTransaction().commit();
em.close();
    }

}
```

COPIAR CÓDIGO

Agora que temos a transação, vamos rodar novamente ("Run As > 1 Java Application") e, em teoria, ele deveria gerar um *insert* no banco de dados. Porém, tivemos uma *exception*: "ERROR: Table "PRODUTOS" not found;". Significa que a tabela "PRODUTOS" não foi encontrada. Nós não criamos a tabela no nosso banco de dados H2.

Na hora em que o Hibernate foi fazer o *insert*, ele indica que conseguiu se conectar com o banco, mas a tabela de "PRODUTOS" não está lá, por isso, ele não consegue fazer o *insert*. Ele, inclusive, mostrou qual seria o *insert* que teria feito "insert into produtos (id, descricao, nome, preco) values (null, ?, ?, ?)".

Portanto, não existe a tabela. Temos que fazer acessar o banco de dados H2 e criar a tabela manualmente. Rodar um comando `CreateTable`. Existe um jeito mais fácil de fazer isso que é com a propriedade do Hibernate que podemos adicionar. Uma propriedade para o Hibernate olhar para as nossas entidades e gerar os comandos SQL para criar o banco de dados automaticamente.

Sendo assim, adicionaremos mais uma propriedade e o nome dela é `"hibernate.hbm2ddl.auto"`. Atenção! Escrevemos "ddl", não "dll". Quem está acostumado com windows, onde temos as "dlls", costuma cometer esse erro.

```
<property name="hibernate.hbm2ddl.auto" value="true"/>
```

COPIAR CÓDIGO

Sobre o valor que devemos passar, temos alguns possíveis. Um deles é o `"create"` em que, toda vez que criarmos um `EntityManagerFactory`, o Hibernate vai olhar as entidades e gerar o comando para criar o banco de dados. Portanto, ele vai apagar tudo e criar do zero as tabelas. Após usarmos a aplicação, ele não apagará as tabelas, elas continuarão lá.

Outra opção é o `"create-drop"`, que cria as tabelas quando rodarmos a aplicação e, depois que terminamos de executar a aplicação, ele imediatamente *dropa*. Há também a opção `"update"`, com a qual ele não vai, em todas as vezes, apagar e criar tabelas, vai apenas atualizar a tabela se alguma mudança surgrir.

Assim, se não existir a tabela, ele cria e se adicionarmos um novo atributo nessa tabela, precisaremos de uma nova coluna e ele fará a atualização para inserir essa nova coluna, mas não *dropa* a tabela, não apaga os registros, apenas atualiza.

Mas, o `"update"` só adiciona coisas novas, por exemplo, se adicionarmos uma nova coluna ou uma nova tabela, ele cria. Mas, se apagarmos uma entidade ou um atributo dela, ele não apaga a tabela e nem a coluna, porque isso pode gerar um efeito colateral.

Existe ainda outra opção que é `"validate"`. Ele não mexe no banco, apenas valida se está tudo ok no banco e gera um *log*. No nosso caso, colocaremos o `"update"` para que ele “atualize”, ou seja, crie uma tabela se ela não existir, se ela já existir,

apenas veja o que mudou. Então, é para isso que serve essa propriedade, para que o Hibernate gere as tabelas, sem que seja necessário conectar ao banco de dados.

```
<property name="hibernate.hbm2ddl.auto" value="update"/>
```

COPIAR CÓDIGO

Vamos rodar a nossa classe de novo (Apertando o botão direito e, depois, "Run As > 1 Java Application") e agora esperamos que ele tenha inserido corretamente. Ele rodou o comando e viu que não tinha tabela, "Hibernate: create table produtos", e gerou corretamente, conforme está mapeado na entidade. Percebeu que existe um "@Table produtos", "id".

Colocou também que é um "generated by default" pelo banco, "identity", "descricao" é um "varchar", "nome varchar", "preco" é um "decimal(19,2)". Portanto, ele gerou tudo corretamente, conforme está mapeado. Ele olha para a entidade para gerar a tabela. Ao final, rodou o *insert*, então, salvou no banco de dados. Ele só não imprime os valores que passamos, coloca interrogação.

Finalizamos o nosso código para inserir e integrar de fato com o banco de dados, falar para JPA ir lá, pegar o objeto e salvar no banco de dados. A parte de iniciar transação, criar `EntityManager` é um pouco complexa e podemos melhorar, extrair para classes, mas isso será assunto para depois.

Espero que tenham gostado, tenham aprendido a integrar com banco de dados, com `EntityManagerFactory` e o `EntityManager`. Nas próximas aulas continuaremos utilizando essas interfaces e vendo outros recursos da JPA. Vejo vocês lá!! Abraços!!