01

Apresentação

Transcrição

Olá, pessoal! Boas-vindas ao curso **Persistência com JPA: Introdução ao Hibernate**. Meu nome é **Rodrigo Ferreira** e serei o instrutor que acompanhará vocês durante este treinamento.

Então, neste treinamento aprenderemos sobre a JPA, "Java Persistence API", poque ela veio para substituir - digamos assim - a JDBC, e aprender a utilizá-la na prática.

Nosso treinamento será dividido em duas partes, e nesta parte discutiremos alguns tópicos. O primeiro é **Motivação para utilizar JPA**: o que é a JPA, por que foi criada e para resolver quais problemas. Enfim, a intenção é saber a utilização e motivação principal da JPA. O segundo é **Download e Configurações**: como fazer para baixar a JPA e utilizá-la no projeto (usando o *hibernate* como implementação).

O terceiro é **Arquivo persistence.xml**: se refere às configurações. Aprenderemos para que serve o aquivo persistence.xml e como configurar os elementos da JPA nesse arquivo. O quarto é **Mapeamento de entidades**: classes que mapearão a tabela no banco de dados, ou seja, aprenderemos o que são essas entidades e como mapear de acordo com o banco de dados.

O quinto é **Mapeamento de relacionamentos**: uma entidade que precisa se relacionar com outra, num relacionamento um para um, um para muitos, muitos para muitos, enfim, aprenderemos como fazer o mapeamento correto utilizando a JPA.

O sexto é **Ciclo de vida de uma entidade**: quando mandamos salvar no banco de dados ou quando fazemos uma consulta, o que acontece com a entidade se mexemos no atributo, se será propagado para o banco de dados. O sétimo e último tópico é **Consultas com JPQL**: a JPQL é uma linguagem parecida com a SQL, mas que conta com a orientação a objetos.

Então, estes são os principais tópicos que trataremos nesta primeira parte do treinamento de JPA. No primeiro vídeo, conheceremos um pouco da motivação e ideia da JPA. Vejo vocês lá!!

Transcrição

Para começar o nosso treinamento de JPA, vamos discutir um pouco da motivação, isto é, para que a JPA foi criada - para resolver quais problemas. Além disso, discutiremos sobre o *hibernate* e outras implementações.

Neste vídeo, também trataremos do JDBC, que é a tecnologia padrão do Java para acessar o banco de dados relacionais. Quem aprendeu a programar em Java, provavelmente, quando for desenvolver sistemas, precisará fazer o acesso ao banco de dados.

Para fazer o acesso do Java com um banco de dados, por exemplo, uma SQL, Oracle, ou qualquer outro banco de dados, a tecnologia padrão utilizada é o JDBC. O Java nasceu em 1995, em 1997 veio o JDBC.

Antes do JDBC, se quiséssemos acessar um banco de dados em Java, era necessário aprender tecnologias complexas de *socket* e fazer tudo manualmente: abrir uma conexão com banco de dados e fazer toda comunicação utilizando o protocolo específico daquele banco de dados. Era muito trabalhoso e complicado.

O JDBC veio para facilitar esse processo. Ele nada mais é do que uma especificação para fazer acesso a bancos de dados relacionais no Java. Portanto, se trata de uma camada de abstração para acessar, do código Java, o banco de dados, independente de qual seja o protocolo. Em outras palavras, o JDBC veio como uma camada para simplificar o acesso e facilitar fazer trocas de bancos de dados.

A partir disso, não é mais necessário conhecer o protocolo MySQL, do Oracle, saber os detalhes técnicos, nem ficar abrindo o *socket* e fazendo uma comunicação manual com o banco de dados, basta utilizar o JDBC.

Quem já estudou o JDBC (na Alura temos treinamento de JDBC), sabe que precisamos ter um driver. Esse driver é um JAR, um arquivo com as classes do banco de dados. Ou seja, ele é a implementação do banco de dados em cima da JDBC.

Então, para acessar o MySQL, é necessário baixar o driver do MySQL. Para trocar o banco de dados - usar o PostgreSQL, por exemplo - trocamos o JAR baixando o driver do PostgreSQL, que é outro JAR. Ambos estão implementando o JDBC, de maneira que, no código, o impacto é mínimo.

Ao mudar de um banco de dados para outro, temos poucas mudanças no código. Trocamos basicamente as configurações, mas, a grande parte do código que está fazendo a comunicação com o banco de dados continua igual. Isso facilita muito por não nos prendermos a um só fornecedor, a um banco de dados.

Para não ficar com o código do JDBC espalhado em vários pontos da aplicação, um padrão de projeto bastante utilizado é o "*DAO*", Data Access Object. Com ele, é possível isolar toda a API do JDBC dentro de uma única classe - de uma única camada - para não ficar com os códigos de *Connection*, *ResultSet*, que são classes complicadas do JDBC, espalhadas pela aplicação.

Basicamente, temos alguma classe na aplicação, um **Controller**, uma **Service** ou algo do gênero. Nesta classe está contida a **lógica de negócios**, e, nessa lógica, precisamos acessar o banco de dados. Ou seja, não instanciamos, não chamamos as classes do JDBC, e, sim, uma classe **DAO**. É na classe DAO que está isolado - abstraído, encapsulado - o código do JDBC, é ela também que faz a ponte com o **banco de dados**. Então, existia uma divisão de responsabilidades na aplicação.

Olhando de fora da classe DAO, existiria algo aproximadamente assim:

```
public class CadastroProdutoService {

    private ProdutoDao dao;

        public CadastroProdutoService(ProdutoDao dao) {
            this.dao = dao;
        }
```

```
public void cadastrarProduto(Produto produto) {
    // regras de negocio...

        this.dao.cadastrar(produto);
}
```

COPIAR CÓDIGO

Imagine que temos um `CadastroProdutoService`. Precisaríamos de uma classe DAO e de um método, como `CadastrarProduto()`. Depois receberíamos o objeto `produto`. Teríamos também as regras de negócio, validação, cálculos, e então chamaríamos `dao.cadastrar`.

Olhando de fora, não dá para saber como a classe DAO está funcionando, se ela está usando JDBC, se a persistência é em banco de dados, se é em arquivo, em memória, em um serviço externo, não sabemos se é MySQL, Oracle.

O código está bem fácil de usar: chamamos a DAO, depois o método `cadastrar()`, passamos o objeto `produto`, e pronto, não ficamos presos a como foi implementado o método `cadastrar()`. Assim, não temos acesso à API do JDBC, ela fica bem isolada. Por fora, o código fica bem bonito, mas, por dentro da classe DAO, temos um problema.

Em classes DAO, usando JDBC, acabamos tendo um código bem complicado, porque precisamos usar a API do JDBC que é uma API bastante antiga do Java (foi criada em 1997), com bastante verbosidade, e fez com que as pessoas desenvolvessem certa aversão ao Java, pela impressão de que nele é tudo burocrático, complexo.

Então, é necessário lidar com classes, como `PreparedStatement`, `connection`, `ResultSet`. Também é necessário fazer `tryCatch()`, porque elas lançam *checked exception*. Além disso, precisamos montar uma SQL manualmente, usar o *PreparedStatement* para não ter o problema do *SQL Injection*. Tudo isso deixa o código um pouco complicado.

Esse tipo de código JDBC, embora funcione, apresenta algumas desvantagens que fizeram com que as pessoas pensassem em outras alternativas mais simples. O JDBC tem dois grandes problemas que motivaram o surgimento de tecnologias como

o *Hibernate* e a JPA. O primeiro problema é o **Código muito verboso**. Por exemplo, para salvar um produto no banco de dados, precisamos de cerca de 30 linhas de código.

Em um código tão grande, é difícil e demorado fazer manutenção. Às vezes, é necessário montar uma *Query* nativa do banco de dados, e o código vai ficando cada vez mais difícil de entender e de fazer manutenção. Esse é um grande problema, e nem é o pior.

O segundo problema é o **Alto acoplamento com o banco de dados**. Quando trabalhamos com o JDBC, temos um acoplamento muito forte com o banco de dados. Significa que, qualquer mudança no banco de dados gera um impacto muito forte na aplicação. Por exemplo, se trocamos o nome da tabela, de alguma coluna ou qualquer outro detalhe desses, acabamos impactando a aplicação e teremos que mexer na classe DAO.

O problema é que, pode acontecer de, por exemplo, ao renomearmos a tabela de produto, pode ser que - além da classe `ProdutoDao` - existam outras classes DAO que façam JOIN com a tabela produto. Ou seja, precisaremos procurar todos os lugares que estão referenciando tabela produto e fazer esse *rename*. Isso nos leva a um alto acomplamento com banco de dados. Qualquer mudança de um lado, gera impacto grande do outro.

Esses são os dois principais problemas que as pessoas começaram a perceber no JDBC e, a por conta deles, aprimoraram ideias de como reduzir o impacto que esses dois problemas geram. Foi desse processo que surgiu a JPA. No próximo vídeo, discutiremos com calma como foi a criação da JPA e como ela resolveu os problemas do JDBC. Vejo vocês lá!! Abraços!!

Transcrição

Já discutimos um pouco sobre o JDBC e os principais problemas que Devs passaram a identificar conforme desenvolviam as aplicações. A existência desses problemas foi, justamente, o que motivou as pessoas a buscarem alternativas, tecnologias que fossem mais simples para fazer a ponte, a ligação com o banco de dados.

Dentre essas tecnologias, surgiu uma biblioteca que ficou bem famosa, chamada **Hibernate**. O Hibernate foi lançado em 2001 e criado por Gavin King, que tinha como ideia, justamente, tentar simplificar o JDBC. Supondo que temos uma aplicação web, desktop, é precisamos fazer o acesso ao banco de dados, mas não gostamos muito do modelo do JDBC, porque o código fica muito verboso, complexo, muito acoplado ao banco de dados.

Gavin King começou a pensar em uma maneira de simplificar o código e criou essa biblioteca (Hibernate) e em 2001 fez o lançamento. Mas, se percebermos, ela não tem nada a ver com o Java, ela é uma biblioteca que surgiu no mercado e que ele distribuiu gratuitamente. Assim nasceu o Hibernate, uma biblioteca famosa por fazer persistência no banco de dados como **alternativa ao JDBC e ao EJB 2**.

Na época existia a versão 2.0 do EJB 2, e era uma tecnologia bem complicada de se trabalhar. A ideia do EJB era simplificar o acesso remoto, para ter uma aplicação distribuída (cliente - servidor) e também simplificar alguns detalhes de infraestrutura, como controle transacional, segurança e outros.

Também havia a parte de persistência junto da EJB, mas ela utilizava a JDBC, era um modelo um pouco mais complexo e que favorecia muito a remotabilidade, então, qualquer chamada que se fazia era uma chamada remota, isso tinha um custo de

performance. Enfim, vários problemas surgiram e vários padrões também foram criados para resolver esse problema da EJB 2 na parte de persistência. Isso também motivou o Gavin King a criar o Hibernate.

Posteriormente, **Gavin King foi contratado pela Red Hat**, para continuar os trabalhos no Hibernate. Portanto, o Hibernate é uma teconologia que pertence à JBoss (Red Hat). O Gavin King trabalhou na JBoss e lá deu continuidade ao Hibernate e a outros projetos.

Conforme o tempo passou, foram surgindo novas versões do Hibernate, ele ficou famoso no mundo inteiro, todas as pessoas que trabalhavam com Java queriam utilizar Hibernate nos projetos para não ter que usar a EJB 2 e JDBC. Enfim, virou um projeto mega popular, e foi evoluindo, foram surgindo versões posteriores com novos recursos.

Como se tratava de uma biblioteca do mercado, surgiram também concorrentes. Portanto, estamos falando de uma biblioteca que está famosa, popular, logo, outras bibliotecas passaram a copiar, mas fazendo de outra forma.

Isso gerou um velho problema: imagine que, como Devs, estamos usando uma biblioteca, queremos trocar para outra (uma biblioteca de mercado). Para fazer isso, não é só trocar os JARs, as dependências do projeto, isso causará um impacto considerável no código.

Para todo o código em que estávamos usando o Hibernate, será necessário trocar para essa nova biblioteca, já que são outras classes, outros *imports*. Se o nosso projeto fosse grande, complexo, pensaríamos duas vezes antes de trocar. A Sun, a Oracle, o Java, não gostam disso, porque significa estar preso a um fornecedor.

Posteriormente, uma padronização da biblioteca foi criada, do modelo de persistência, que ficou conhecida como **ORM** (Object Relational Mapping) em Java, com a intenção de fazer o mapeamento, a ponte entre o mundo da "orientação a Objetos" com o "relacional" do banco de dados.

Existem esses dois mundos distintos, e a classe DAO ficava um tanto complexa, porque estávamos fazendo essa ponte entre o mundo "orientação a objetos" e mundo orientado ao "modelo relacional" do banco de dados. O Java criou uma especificação

chamada de **JPA**, Java Persistence API, que é a especificação para padronizar o mapeamento a objeto relacional no mundo Java.

Com a JPA, criou-se um padrão para que não ficássemos reféns de uma biblioteca. Os *frameworks*, as bibliotecas, começaram a implementar a JPA. No código, ao invés de fazer os *imports* das classes e interfaces do Hibernate, passou-se a fazer a da JPA, que é a especificação.

Portanto, a biblioteca se tornava uma implementação, e, para trocar de implementação, só precisávamos trocar os JARs, uma ou outra configuração, mas o código, em si, continuava intacto, inalterado, por não depender de uma implementação.

A JPA só foi **lançada em 2006**, então, de 2001 a 2006, precisávamos utilizar o Hibernate ou os seus concorrentes sem a abstração da especificação. O Hibernate foi evoluindo, foram surgindo novos recursos e depois isso foi incorporado na **versão 2.0 da JPA, lançada em 2009**.

O **Hibernate, em 2010**, um ano depois, lançou a **versão 3.5.0**, que era **compatível com a JPA 2.0**. Portanto, se quiséssemos usar a JPA 2.0, podíamos utilizar o Hibernate como implementação. Se posteriormente quiséssemos trocar por outras implementações, seria fácil, não causaria impacto no projeto inteiro. Essa é a grande vantagem de se utilizar uma especificação ao invés de usar diretamente uma implementação e ficar preso a ela.

No mercado, ficamos com algo parecido com esse diagrama, em que uma seta sai de cada um dos três retângulos (1. Hibernate, 2. EclipseLink, 3. OpenJPA) e aponta para o retângulo que está acima deles: JPA. Ou seja, temos em cima a JPA, que é a especificação, e temos também várias implementações, como o Hibernate, o EclipseLink, OpenJPA, dentre outras implementações. Essas são as três principais da JPA.

Para trabalharmos com a JPA, temos que escolher uma dessas implementações. Isto é, não se usa a JPA "pura", porque ela é só a "casca", a abstração. Nós precisamos de alguém que implemente os detalhes, quem faz esse trabalho são bibliotecas como o Hibernate.

Como o Hibernate foi a primeira biblioteca, ele foi quem começou esse movimento, por isso acabou se tornando a biblioteca padrão, a mais utilizada, mais popular como implementação da JPA. Mas, nada nos impede de usar outras implementações. Se todas estão seguindo a JPA, todas precisam atender ao que está descrito na especificação da JPA.

Além disso, podemos trocar uma implementação pela outra, e tudo que estiver na especificação vai funcionar. Só teremos problemas se estivermos utilizando algo que é específico da implementação. Às vezes, o Hibernate costuma fazer isso, segue a JPA, mas tem alguns recursos que são específicos dele.

Se utilizarmos, está tudo certo, estamos ganhando esse novo recurso. Mas, o lado negativo, já que esse recurso é específico do Habernate. Então, por exemplo, se quisermos trocar o Hibernate pelo EclipseLink, perderemos essa funcionalidade. Assim, temos que tomar esse cuidado e avaliar se vale mesmo a pena essa dependência.

O EclipseLink é a implementação de referência da JPA. Sempre que surge uma nova versão da JPA, o EclipseLink já está implementando, pois, é nele que são feitos os testes e ele sai com a nova versão da JPA. Ele é a implementação de referência, porém, o Hibernate é a principal, a mais popular no mercado. Por isso, neste curso trabalharemos com o Hibernate como implementação da JPA.

Fechamos essa parte de motivação. Nosso objetivo era apenas discutir um pouco e entender o que é a JPA, porque ela foi criada, de que nasceu, o que é o Hibernate, qual a diferença de Hibernate e JPA. Agora que alinhamos esses conhecimentos, no próximo vídeo, partiremos para a prática. Começaremos a aprender como utilizar a JPA e Hibernate no projeto.

Vejo vocês lá!! Abraços!!

Transcrição

Agora que já conhecemos um pouco da história da JPA, chegou a hora de colocarmos a "mão na massa", de começarmos a trabalhar, na prática, com a JPA, utilizando o Hibernate como implementação.

Utilizaremos o Eclipse como IDE (mas você pode usar a IDE de sua preferência). A ideia é criarmos um projeto com Maven para facilitar as dependências do Hibernate e as outras que decidirmos utilizar durante o treinamento.

Vamos começar selecionando a opção "Create a Maven Project" que está na aba lateral esquerda. Na próxima tela, marcaremos "Create a simple project (skip archetype selection) para que ele “pule” o *archetype*, e selecionaremos "Next". Em seguida, preencheremos: "Group Id" com "br.com.alura"; "Artifact Id" com "loja", isto é, nós criaremos uma aplicação como uma loja com cadastro de produto e outras coisas. O resto, podemos deixar com a opção padrão, e agora basta apertar "Finish".

Ele criará um projeto utilizando Maven, com uma estrutura de Diretórios e o arquivo `pom.xml`, onde configuraremos as dependências. O foco do nosso treinamento não é aprender sobre Maven (na Alura existe um treinamento de Maven). Basicamente, lidaremos com o `pom.xml`, encontraremos as configurações do projeto que preenchemos na tela de criação.

```
<modelVersion>4.0.0</modelVersion>
<groupId>br.com.alura</groupId>
<artifactId>loja</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

COPIAR CÓDIGO

Precisaremos fazer duas configurações. Por padrão, no Eclipse, se não indicarmos ao Maven qual a versão do Java, ele considerará que é o Java 5. Então, colaremos a Tag `build` e um `plugin` do Maven para dizer: quero utilizar o Java na versão 11. Nós disponibilizaremos esse trecho de código para que não seja necessário digitar tudo manualmente.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.0</version>
            <configuration>
                <release>11</release>
            </configuration>
        </plugin>
    </plugins>
</build>
```

COPIAR CÓDIGO

Então, se trata de um `plugin` para dizer ao Maven que queremos utilizar o Java 11. O que nos interessa é, logo embaixo da tag `build`, a parte de dependências do Maven. Portanto, abriremos a tag `<dependencies>` e adicionaremos uma dependência, `<dependency>`. No nosso caso, queremos utilizar a JPA, que é a especificação, com o Hibernate como implementação.

Basta adicionar apenas uma dependência do Hibernate e, automaticamente, ele adicionará outras dependências da JPA e de todas as outras que o Hibernate precisa. Continuando, precisamos colocar qual é o `groupId`, no caso `org.hibernate`. E o `artifactId` que adicionaremos será o `hibernate-entitymanager`.

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
    </dependency>
</dependencies>
```

COPIAR CÓDIGO

Precisamos passar também qual é a versão da dependência - qual a versão do Hibernate que vamos utilizar. No momento de gravação deste curso, início de 2021, a última versão que estava disponível era a `5.4.27.Final` e é a que nós utilizaremos.

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.4.27.Final</version>
    </dependency>
</dependencies>
```

COPIAR CÓDIGO

Se olharmos o projeto, no "Maven Dependencies", perceberemos que ele já baixou uma série de dependências e adicionou ao nosso projeto. Dentre elas, temos o "hibernate-entitymanager", que foi o que acabamos de adicionar, e que, por sua vez, depende de todas as outras dependências, como o "hibernate-core", "hibernate-commons-anotations" e o "javax.persistence-api-2.2.jar", que é a JPA em si. Portanto, aí está a especificação.

De forma bem simples, adicionamos o Hibernate e a JPA como dependência de uma aplicação que está utilizando o Maven. Além do Hibernate, precisaremos de mais uma dependência, que é a dependência do driver do banco de dados que utilizaremos.

No nosso caso, vamos usar o H2, que é um banco de dados em memória, só para não perdermos tempo com instalação e configuração de banco de dados. Mas, é possível usar outros bancos de dados, por exemplo, o MySQL ou o Postgres. Então, vamos seguir colocando mais uma dependência `<groupId>com.h2database</groupId>`. E o `<artifactId>` é o `h2`. A versão do `h2` será a 1.4.200.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.200</version>
</dependency>
```

COPIAR CÓDIGO

Então, são duas dependências que precisamos adicionar ao nosso projeto quando trabalharmos com Hibernate. A dependência do Hibernate em si, que baixará todas aquelas dependências que ele tem, além da própria JPA, que é a especificação, e a dependência do banco de dados que estivermos utilizando. É possível também modificar para usar o MySQL, Postgres.

Com isso, já temos a nossa aplicação Java com Maven, utilizando as dependências do Hibernate e da JPA. Ele só está marcando um erro. Vamos corrigi-lo apertando o botão direito "Maven > Update Project", na próxima tela apertaremos "Ok". Resolvido, foi algum problema ao baixar as dependências.

Este é o nosso projeto Maven com Hibernate já baixado e adicionado como dependência do Maven, para não precisarmos baixar os JARs manualmente. O próximo passo seria configurar a JPA, criar o arquivo `persistence.xml`, criar as entidades, fazer o mapeamento e começar a trabalhar com a JPA. Como essa parte é um pouco mais complicada, deixaremos para a próxima aula.

Vejo vocês lá!! Um abraço!!

Transcrição

Já temos o projeto criado e as dependências do Hibernate e do nosso banco de dados H2 e podemos continuar trabalhando com a JPA. Uma das coisas importantes que temos que fazer em uma aplicação que utiliza JPA é a parte de configuração. Na JPA, as configurações ficam no arquivo `.xml`. É possível configurar também via código Java, mas, geralmente, ficam no arquivo `.xml` chamado `persistence.xml`.

Nesse vídeo, entenderemos como é esse arquivo, quais são as configurações e como elas funcionam. Então, vamos criar esse arquivo no nosso projeto. Como se trata de um arquivo de configuração, colocaremos no "src/main/resources". Mas, ele precisa ficar dentro de uma pasta, então, primeiro criaremos a pasta.

Para isso, apertaremos com o botão direito no "src/main/resources", depois selecionaremos "New > Folder". Na próxima tela, preencheremos "Folder name" com o nome da pasta, que tem que ser "META-INF" (tudo em letra maiúscula). Esse é o nome do Diretório.

Criada a pasta, vamos até ela e selecionaremos "New > Other". Na próxima tela, selecionaremos "XML File" e depois chamaremos o arquivo, em "File name", de "persistence.xml". Agora basta apertar "Finish". Ele criará um arquivo `.xml` em branco, só com o cabeçalho do `.xml` que é: `<?xml version="1.0" encoding="UTF-8"?>`.

E esse é o arquivo `.xml`. Ele precisa ter o nome `persistence.xml` e tem que estar na pasta "META-INF", e, como se trata de uma aplicação Maven, fica no "rsc/main/resources". Dentro do `.xml` vêm as tags de configuração da JPA. Vamos colar a tag raiz (principal) e o cabeçalho de configuração do `.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/per

</persistence>
```

COPIAR CÓDIGO

Pronto, aqui está a tag raiz `persistence` e as configurações de *namespace* do `.xml`. Não é necessário decorar, podemos pegar da internet ou de algum outro projeto. Dentro do arquivo, existe a tag raiz chamada `persistence`, e todas as configurações ficam dentro dela. Inclusive, ele já está reclamando, como se dissesse: é obrigatório ter uma tag `persistence-unit` e ela não está aqui.

Então, vamos adicionar a tag `persistence-unit` e dividí-la em dois, da seguinte maneira:

```
<persistence-unit name="">

</persistence-unit>
```

COPIAR CÓDIGO

A tag `persistence-unit` tem duas propriedades principais. Temos a `name`, e nela podemos colocar o nome que quisermos. No caso, escolheremos o nome `"loja"`, o mesmo do projeto. Há outra tag importante chamada `transaction-type="JTA"` com dois valores possíveis: JTA ou RESOURCE LOCAL. Nós utilizaremos a `"RESOURCE_LOCAL"`.

```
<persistence-unit name="loja" transaction-type="RESOURCE_LOCAL">
```

```
</persistence-unit>
```

COPIAR CÓDIGO

A opção pela `JTA` seria mais adequada para quando estamos utilizando um servidor de aplicação, quando vai trabalhar com EJB, JMS ou outras tecnologias do Java EE, e o servidor se encarrega de cuidar da transação. Como, no nosso caso, se trata de uma aplicação *stand-alone*, sem servidor de aplicação, então será `"RESOURCE_LOCAL"` . Nós que gerenciaremos a transação. Depois veremos com calma essa parte de transação.

Então, a tag `persistence-unit` , tem que ficar dentro da tag raiz `persistence` , e dentro da tag `persistence-unit` que vão todas as configurações da nossa aplicação. Vamos pensar no `persistence-unit` como se ele fosse um banco de dados.

Se a nossa aplicação fosse trabalhar com dois, três bancos de dados, deveríamos ter, dentro da tag `persistence` , duas ou três tags `persistence-unit` . Ou seja, é um `persistence-unit` para cada banco de dados.

Dentro da tag `persistene-unit` nós adicionamos algumas propriedades para ensinar a JPA detalhes referentes ao nosso projeto (banco de dados e coisas do gênero). Então, existe uma tag chamada `properties` , e dentro dela, cada propriedade que configurarmos, fica em uma tag chamada `property` que tem um nome e um valor.

```
    <persistence-unit name="loja" transaction-type="RESOURCE_LOCAL">
        <properties>
                <property name="" value=""/>
            </properties>
    </persistence-unit>
</persistence>
```

COPIAR CÓDIGO

Existem algumas propriedades que são obrigatórias, que precisamos informar para a JPA. Essas propriedades são específicas da JPA, então, elas têm um nome específico. Colocamos `"javax.persistence.` e a configuração queremos fazer.

Se quisermos configurar, por exemplo, o driver do banco de dados, então, se estamos utilizando o H2, precisamos dizer para a JPA qual a classe do driver do H2.

Então, a propriedade se chama `javax.persistence.jdbc.driver`. Esse é o nome da propriedade. Não é necessário decorar, essas linhas de `.xml` ficarão disponíveis. É possível também pegar alguma como exemplo na internet ou de outro projeto. Nosso objetivo é apenas entender o que significa cada uma dessas propriedades.

No `value`, passaremos qual a classe do driver do JDBC. No caso do H2, é `"org.h2.Driver"`. Esse driver mudará de acordo com o banco de dados que quisermos utilizar. Se fosse o MySQL, seria `com.mysql.`. Se fosse PostgreSQL, `org.postgresql.`. Ou seja, varia de acordo com o banco de dados.

```
<property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
```

COPIAR CÓDIGO

Essa linha serve para dizer à JPA qual é a classe e onde está o driver do JDBC. Estamos usando a JPA, mas ela nada mais é do que uma camada de abstração em cima do JDBC. Por "baixo dos panos", de forma oculta, a JPA trabalha com o JDBC. Por isso utilizamos as propriedades do JDBC.

Além do driver, precisamos configurar a JPA indicando qual é a URL do banco de dados, isto é, onde está o endereço de conexão com o banco de dados. Esse endereço também varia de acordo com o banco de dados. No caso do H2, será `"h2:mem:loja"`. Isto é, queremos que o *database* no H2 se chame loja.

```
<property name="javax.persistence.jdbc.url" value="h2:mem:loja"/>
```

COPIAR CÓDIGO

Todo banco de dados tem um usuário e uma senha. Portanto, teremos mais duas propriedades, a `jdbc.username"`, e a `jdbc.password"`. Então, essas são as duas propriedades para configurar o usuário e a senha. No caso do H2, será `"sa"`

(geralmente usamos esse usuário no H2), e a senha ficará em branco `value""` .

```
<property name="javax.persistence.jdbc.username" value="sa"/>
<property name="javax.persistence.jdbc.password" value=""/>
```

COPIAR CÓDIGO

Da JPA, são só essas quatro propriedades: qual é o driver do banco de dados; onde está a URL de conexão com o banco de dados; o usuário e a senha. Feito isso, a JPA consegue gerar as conexões para acessar o nosso banco de dados.

Existem propriedades específicas da implementação da JPA que estamos utilizando. Por exemplo, existem propriedades específicas do Hibernate, do TopLink, do OpenJPA, do EclipseLink, enfim, de cada uma das implementações da JPA. Como estamos utilizando o Hibernate, podemos colocar algumas propriedades específicas dele.

Então, vamos adicionar mais uma propriedade que terá o nome `"hibernate.dialect"` . O Hibernate precisa saber qual é a classe que tem o dialeto do banco de dados. Ou seja, saber das particularidades do banco de dados. Por exemplo, no H2 não existe booleano (booleano é inteiro, 0 e 1), no MySQL existe.

Como cada banco de dados pode ter as suas particularidades, o dialeto é o que fará a comunicação correta com o banco de dados. O Hibernate precisa que essa propriedade será fornecida. O valor será a classe do Hibernate que representa o dialeto. No caso do H2, será `"org.hibernate.dialect.H2Dialect"` .

```
<property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
```

COPIAR CÓDIGO

Essas são as principais propriedades de configuração do banco de dados (existe outra propriedade que é interessante configurar, mas adicionaremos posteriormente). O arquivo `persistence.xml` pode até parecer muito complicado, mas se trata, basicamente, de configuração, e precisamos configurar apenas uma vez.

A ideia é ensinar para a JPA detalhes do nosso banco de dados para que ela consiga se conectar e acessar o banco de dados corretamente. A JPA depende disso para se unir ao JDBC e acessar o banco de dados.

Apenas recapitulando, temos o `xml` com a tag raiz chamada `persistence`. Tudo estará dentro desta tag principal `persistence`. Depois temos os *namespaces* (padrão da JPA). Em seguida, vem a tag `pesustence-unit`, que representa uma unidade de persistência (podemos pensar nela como um banco de dados), por isso ela precisa ter um nome (que, no nosso caso, é `"loja"`).

Se tivéssemos vários bancos de dados, teríamos várias tags `persistence-unit` e conseguiríamos diferenciá-las pelo `name`, que precisa ser único para cada uma delas. No caso, colocamos `"loja"`, que é o mesmo nome do projeto, mas não é obrigatório, poderia ser qualquer outro.

Adicionamos também o tipo de transação, `transaction-type`, que é `"RESOURCE_LOCAL"`, no caso de gerenciarmos a transação, ou JTA, se tivermos usando algum Java EE, e o servidor se encarregará de cuidar do controle transacional, que não é o nosso caso.

Dentro da tag `persistence-unit`, temos `properties` e as propriedades da JPA: driver do banco de dados; URL; usuário; e senha do banco de dados. Como implementação do Hibernate, temos o dialeto. Depois veremos outras propriedades.

O objetivo do vídeo era conhecer o arquivo de configuração da JPA, as tags, as propriedades e para que servem cada uma delas. Espero que vocês tenham aprendido um pouco. Na próxima aula continuamos trabalhando no projeto, vendo a questão das entidades e como fazer para acessar de fato o banco de dados. Vejo vocês lá!! Abraços!!

Transcrição

Já fizemos as configurações e podemos continuar. Mas antes, vamos corrigir dois detalhes que acabei errando no vídeo anterior. Na propriedade da URL do JDBC, antes do `"h2:mem:loja"`, faltou o prefixo `jdbc:`. Portanto, o correto é `"jdbc:h2:mem:loja"`. E na propriedade do usuário do banco de dados não é `username"` e, sim, `user"`. É complicado querer decorar, o melhor é pegar de exemplos na internet ou de outro projeto.

Enfim, temos as informações do banco de dados configuradas para a JPA. A JPA já sabe se comunicar com o JDBC e passar essas configurações na hora em que precisar acessar o banco de dados. Agora configuraremos a parte de persistência. Nosso projeto é uma loja, como faremos para integrar? Como ensinaremos para a JPA a nossa tabela no banco de dados?

No nosso caso, teremos, inicialmente, a seguinte **Tabela de produtos**:

Produtos	
id	bigint
nome	varchar
descricao	varchar
preco	decimal

Vamos imaginar que criaremos essa tabela de produtos no banco de dados e que ela tem essas quatro colunas: uma chamada "id", que é a chave primária do banco de dados (autogerada pelo banco de dados); o "nome", que é um "varchar" (um texto, uma descrição do produto); a "descricao" do produto, que também é um texto, um "varchar"; e o "preco", que é um "decimal".

Então, tendo essa tabela no banco de dados, como ensinaremos e a configuraremos para que seja representada de alguma maneira no código Java? Na JPA, isso será feito por uma classe Java, que na JPA é chamda de entidade. Nós mapeamos todas as tabelas no banco de dados por uma entidade, que nada mais é do que uma classe Java.

Vamos criar a classe que representará um produto no banco de dados. Já que é uma classe Java, ficará no "src/main/java". Apertaremos "Ctrl + N", em "Wizards", digitaremos "class" para filtrar e selecionaremos "Next". Na próxima tela, trocaremos o pacote de "loja" para "br.com.alura.loja", e o nome da classe será "Produto".

Essa será a nossa classe chamada `Produto.java`, que representará um produto no banco de dados. Vamos apenas adicionar ".modelo" no pacote, isto é, `br.com.alura.loja.modelo` e podemos prosseguir.

Na JPA, precisamos lembrar que a ideia não é que ela seja uma especificação para um ORM. Com a ORM nós fazemos o mapeamento objeto-relacional. Nós precisamos desse mapeamento, dessa ligação entre o lado da orientação objetos com o lado do mundo relacional do banco de dados. Isso é feito na classe `public class Produto {`, e é ela que está representando a tabela de produtos, portanto, é assim que a indicaremos para a JPA.

A partir da versão 2.0 da JPA, podemos fazer tudo via anotações. Então, em cima da classe, podemos colocar uma anotação da JPA que é o `@Entity`. Assim, é como se disséssemos: JPA, está vendo essa classe `Produto`? Ela é uma entidade, ou seja, existe uma tabela no banco de dados que está mapeando, e que é o espelho dessa classe. Então, é para isso que serve essa anotação `@Entity`.

Agora, apertamos "Ctrl + Shift + O" para importar. Ele sugeriu duas opções para importar e precisamos escolher com cuidado. Existe a opção `javax.persistence.Entity` e a `org.hibernate.annotations.Entity`. A primeira é a anotação da especificação da JPA. A segunda, é a do Hibernate. Então, importaremos a do `javax.persistence.Entity`, que é a da especificação.

Se não queremos ficar presos ao Hibernate - a uma implementação - e desejamos usar o máximo possível a especificação, porque se um dia quisermos trocá-la, não teremos que mexer em todas as classes. Portanto, precisamos ter cuidado com

tudo que importarmos.

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

import javax.persistence.Entity;

@Entity
public class Produto {
```

COPIAR CÓDIGO

Com isso, o Hibernate JPA já sabe que a classe `Produto` está mapeando uma tabela no banco de dados. Só que, no banco de dados, o nome da tabela é "produtos" no plural. Como é possível dizer isso para a JPA, já que não queremos nomear a classe como "Produtos" no plural e com “p” minúsculo? Algo que contrariaria as convenções do Java.

Eventualmente, se o nome da tabela não for o mesmo da entidade, teremos que ensinar isso para a JPA, porque, por padrão, ela considera que o nome da tabela é o mesmo nome da entidade (no nosso caso, não é). Para fazer essa configuração, adicionaremos mais uma anotação em cima da classe que é o `@Table` . Apertaremos "Ctrl + Shift + O" para importar e, de novo, selecionaremos `javax.persistence.Table` .

Na anotação `@Table` , abriremos parênteses, selecionaremos o atributo `name:String - Table` com a qual passaremos o nome da tabela que é `name = "produtos"` .

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

import javax.persistence.Entity;

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

COPIAR CÓDIGO

Com isso, ensinamos a JPA que, embora o nome da entidade seja `Produto`, o nome da tabela é `produtos`. Agora ela já sabe que, ao fazer a ligação, ela precisará fazer também a conversão. Dentro da classe, nós temos os atributos, que nada mais são do que o espelho das colunas no banco de dados. A nossa tabela tem quatro colunas (id, nome, descricao e preco) e nós as transformaremos em atributos. Vamos adicioná-los:

```
private Long id;
private String nome;
private String descricao;
private BigDecimal preco;
```

COPIAR CÓDIGO

Uma curiosidade é que o nome dos atributos é exatamente igual ao nome das colunas no banco de dados. Logo, isso é algo que não precisaremos ensinar para a JPA, ela já assume que o nome da coluna é o mesmo do atributo dentro da entidade. Se fosse diferente, isto é, se o nome da coluna "descricao" fosse "desc", por exemplo, como ensinaríamos para a JPA caso não quiséssemos chamar o atributo de desc e, sim, de "descricao"?

Neste caso, nós colocaríamos, em cima do atributo, uma anotação chamada `@Column` (e apertaríamos "Ctrl + Shift + O" para importar). Da mesma maneira, existe uma atributo chamado `name`, seguido dele, passaríamos o nome da coluna no banco de dados `"desc"`. Ou seja, `"Column(name = "desc")`. É como se disséssemos para a JPA: o nome do atributo é `descricao`, mas o nome da coluna, `@Column`, é `desc`.

Desta maneira, é possível ensinar a JPA quando o nome da coluna for diferente do nome do atributo. No nosso caso, vamos apagar essa anotação, porque o nome da coluna é exatamente igual ao nome do atributo.

Este processo que fizemos tem o nome de **mapeamento**, isto é, fizemos o mapeamento de uma entidade, ensinamos ao Java e JPA que a classe `Produto` representa uma tabela, que o nome da tabela é diferente, no banco de dados, do nome da classe. Ensinamos também quais são os atributos que serão mapeados como colunas.

Só temos mais um detalhe importante para a JPA. No banco de dados, a coluna "id" é a chave primária. Nós precisamos informar qual é a "*primary key*", a chave primária da tabela no mundo relacional. Também precisamos informar para a JPA que, dos quatro atributos, o primeiro, que se chama `id`, é a chave primária, já que ele não associa automaticamente.

Em cima do atributo `id`, colocaremos uma notação chamada `@Id` e apertamos "Ctrl + Shift + O" para importar. No nosso caso, ele importou diretamente do `javax.persistence.Id`. Como, geralmente, quem cuida do `id`, da chave primária é o banco de dados e não a aplicação, também precisamos ensinar para a JPA que quem gerará o identificador não é a aplicação e, sim, o banco de dados.

Quando formos salvar um produto, o `id` estará nulo. Não tem problema, porque é o banco de dados que vai gerar o próximo `id`. Podemos configurar isso com outra notação, que colocamos em cima do atributo `id`, que é o `@GeneratedValue`, isto é, para dizer como o valor da chave primária é gerado.

```
@Id
@GeneratedValue()
private Long id;
private String nome;
private String descricao;
private BigDecimal preco;
```

COPIAR CÓDIGO

Existe um parâmetro que precisamos passar que é a estratégia, `strategy`, isto é, qual é a estratégia de geração da chave primária. Isso dependerá do banco de dados, alguns usam SEQUENCE, outros não. Então, nas estratégias, temos três opções:

IDENTITY; SEQUENCE; e TABLE. Geralmente, utilizamos a IDENTITY, quando não tem SEQUENCE no banco de dados, ou SEQUENCE, quando tem. No nosso caso, será IDENTITY, já que não temos SEQUENCE no H2.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
private String nome;
private String descricao;
private BigDecimal preco;
```

COPIAR CÓDIGO

Feito isso, falta apenas gerar os *Getters e Setters*. Abriremos um atalho com o botão direito e selecionaremos "Source > Generate Getters and Setters". Na próxima tela, marcaremos todos os atributos com "Sellect All" e apertaremos "Generate". Retornando ao `Produto.java`, selecionaremos o comando "Ctrl + Shift + F" para formatar e está pronto o mapeamento da entidade.

Então, é assim que mapeamos quais classes vão representar tabelas no banco de dados. Depois conheceremos outras anotações. Quando tivermos relacionamentos de tabelas, aprenderemos a mapear também esses relacionamentos. Enfim, veremos tudo isso com calma durante o curso.

Um último detalhe para fecharmos esse vídeo. Pela JPA, em relação a toda entidade, além de precisarmos ir até a classe e adicionar anotações da JPA para fazer o mapeamento, também deveríamos adicionar a classe no `persistence.xml`. Fora das `properties` e dentro do `persistence-unit`, exite outra tag chamada `class`.

Pela JPA, deveríamos passar todas as classes/entidades do nosso projeto, ou seja, passaríamos o caminho completo da classe, `br.com.alura.loja.modelo.Produto`. Pela JPA, para cada entidade que mapearmos, além de mapear na classe, temos que adicioná-la no `persistence.xml` com a tag `class`.

```
<class>br.com.alura.loja.modelo.Produto</class>
```

COPIAR CÓDIGO

Porém, se tivermos utilizando o Hibernate, não precisamos adicionar a tag `class`, porque ele consegue encontrar automaticamente as classes/entidades do nosso projeto. Essa é uma particularidade do Hibernate, pode ser que as outras implementações não façam isso e, portanto, teremos que, manualmente, adicionar o `class`.

Como estamos utilizando o Hibernate, e esse processo é meio trabalhoso: ao criar uma nova tabela no banco, temos que criar a classe, fazer o mapeamento e adicionar no `persistence.xml`. Para não esquecermos de nada, não vamos adicionar no nosso código, pois o Hibernate encontrará automaticamente.

Outro detalhe importante, se adicionarmos uma entidade no `class`, temos que adicionar todas. Se esquecermos alguma, o Hibernate só olhará para as que estiverem declaradas. Ou mapeamos todas, ou nenhuma.

Esse era o objetivo do vídeo de hoje, mostrar como fazemos o mapeamento de uma entidade. Agora, já temos o `persistence.xml` com as configurações do banco, já temos uma entidade mapeada e, no próximo vídeo, veremos como fazer para cadastrar um produto no banco de dados (dentro da JPA e das classes Java).

Entidades da JPA

Qual a melhor definição de uma entidade JPA? () () ()

A É uma classe que faz o mapeamento de um banco de dados

Uma entidade JPA faz o mapeamento de uma tabela e não do banco de dados em si

B É uma classe que substitui o pattern DAO

Entidades JPA não substituem o padrão de projeto DAO

C É uma classe onde configuramos o acesso ao banco de dados

Esse não é o objetivo de uma entidade JPA

✓ É uma classe que faz o mapeamento de uma tabela do banco de dados

Alternativa correta! Uma entidade JPA funciona como um *espelho* de uma tabela no banco de dados

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!!

Transcrição

Na última aula, aprendemos como mapear uma entidade, como fazer as configurações da JPA e o *insert* no banco de dados utilizando a classe `EntityManager`. Na aula de hoje, continuaremos fazendo o mapeamento com as entidades e também estudaremos a parte de relacionamento entre entidades. Mas, antes disso, nos dedicaremos um pouco à organização do código.

Nós tínhamos criado a classe `CadastroDeProduto` de método `main` apenas para simular que um usuário preencheu as informações "Nome", "Descricao" e "Preco" em um formulário. E precisamos usar algumas classes da JPA para fazer a persistência.

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

        EntityManager em = factory.createEntityManager();

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

COPIAR CÓDIGO

Mas, assim como havíamos comentado sobre o JDBC, que era comum utilizarmos o padrão DAO, com a JPA, isso também é possível e até recomendável organizar o código e isolar toda a API da JPA para não ficar espalhada por um monte de classes do projeto. Então, podemos criar uma classe DAO e começar a organizar o código. Vamos fazer isso.

Nós temos a entidade `Produto`. Toda a parte de persistência do produto ficará na classe `ProdutoDao`. Vamos criar uma nova classe apertando "Ctrl + N", selecionando "Class" e "Next". Na próxima tela, alteraremos o pacote para "br.com.alura.loja.dao", e o nome da classe será "ProdutoDao". Agora basta apertar "Finish" para criar a `Produto.java`.

Será bem parecido com o exemplo da classe DAO no projeto JDBC, mas, ao invés de trabalhar com uma *connection*, trabalharemos com `EntityManager`. Então, esta classe terá um atributo do tipo `EntityManager`, já que precisaremos utilizá-lo em todos os métodos e transformá-lo em um atributo.

```
private EntityManager em;
```

COPIAR CÓDIGO

Vamos usar a ideia de injeção de dependências para não deixar a classe DAO ser responsável por criar e gerenciar o `EntityManager`, então, criaremos um construtor apertando o botão direito e selelcionando "Source > Generate Cronstuctor using Fields" e, na próxima tela, marcaremos o "em" (EntityManager) e apertaremos "Generate".

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

import javax.persistence.EntityManager;

public class ProdutoDao {

    private EntityManager em;

    public ProdutoDao(EntityManager em) {
```

```
        this.em = em;
    }

}
```

COPIAR CÓDIGO

Quem for instanciar a classe `ProdutoDao` terá que passar o `EntityManager`. Portanto, a classe DAO não é responsável por criar e nem gerenciar o `EntityManager`, ela simplesmente o recebe pronto para ser utilizado. Agora criaremos o método `public void cadastrar()` que receberá, como parâmetro, um produto, isto é, `public void cadastrar(Produto produto)`.

Este é o método onde cadastraremos um produto no banco de dados utilizando a JPA. Teremos, basicamente, uma única linha `this.em.persist(produto);` (this, ponto, entity manager, e passamos o produto).

```
public class ProdutoDao {

    private EntityManager em;

    public ProdutoDao(EntityManager em) {
        this.em = em;
    }

    public void cadastrar(Produto produto) {
        this.em.persist(produto);
    }

}
```

COPIAR CÓDIGO

A transação,nós deixaremos de fora da classe, justamente para deixar a classe DAO bem limpa, simples e enxuta. O único objetivo dela é fazer a ligação com o banco de dados. Desta maneira, estamos apenas usando o `EntityManager`, significa que, não criamos e nem fechamos o `EntityManager` ou gerenciamos transações. Simplesmente, recebemos um `EntityManager` no construtor que já vem com tudo configurado. Com isso, nossa classe fica bem coesa.

Se recordarmos da classe DAO utilizando JDBC, onde tínhamos umas dez, vinte ou trinta linhas de código, perceberemos que, diferente disso, com a JPA ficamos com uma única linha de código, `this.em.persist(produto);`. Portanto, com a JPA, nós resolvemos dois problemas do JDBC: código verboso e alto acomplamento com o banco de dados.

Nós ainda temos acomplamento com o banco de dados. Se mexermos em algo no banco de dados, isso gerará impacto na aplicação, mas será um impacto mínimo. Por exemplo, vamos imaginar que tivemos que renomear a tabela, então, nós precisar alterar apenas na entidade, em `Produto.java`. Vamos ao `@Table(name = "produtos")` e passamos o novo nome.

Não precisamos alterar a nossa classe DAO e nenhuma outra, porque não temos nenhuma outra referência para o nome da tabela ou das colunas. É muito mais fácil fazer uma mudança no banco de dados, e os impactos são mínimos na aplicação. Então,com a JPA, resolvemos os dois problemas da JDBC.

Continuando, nós criamos a classe `ProdutoDao`, agora o código está bem simples. Na classe `CadastroDeProduto` que tem o método `main` (pensando numa aplicação, essa parte se referiria a um Controller, uma Service). Como não temos aplicação web, é apenas uma aplicação de Java pura, *standalone*, e, portanto, não teremos determinados recursos ("Nome", "Descricao", "Preco"), porque estamos simulando um usuário.

É em `EntityManagerFactory factory = Persistence` (que está em `CadastroDeProduto.java`) que cuidaremos de toda a a parte de `EntityManager`, mas, em relação a persistência, nós extrairemos `em.persist(celular);` para a classe DAO. Portanto, precisaremos de um `ProdutoDao dao = new ProdutoDao(em)`. Quando instanciamos um `ProdutoDao`, nós temos que passar um `em` (EntityManager), e ele foi criado em:

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

COPIAR CÓDIGO

E toda a parte de transação é feita na classe, em vez de ficar na classe DAO. Seguindo, `em.persist(celular);` virará `dao.cadastrar(celular);` (estamos passando, portanto, o produto celular).

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

        EntityManager em = factory.createEntityManager();

        ProdutoDao dao = new ProdutoDao(em);

        em.getTransaction().begin();
        dao.cadastrar(celular);
        em.getTransaction().commit();
        em.close();
    }

}
```

COPIAR CÓDIGO

Perceberemos que o código da classe DAO, `ProdutoDao.java`, ficou simples. O código de `CadastroDeProduto.java` ficou um pouco grande e podemos simplificar. Por exemplo, em todos os testes que fizermos, sempre precisaremos criar um `EntityManager`, e para criá-lo, precisamos, antes, criar o `factory`.

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

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

COPIAR CÓDIGO

Então, para não ficar com o código de criação do `EntityManager` e da `factory`, podemos extraí-lo para uma classe utilitária. Vamos criar uma nova classe apertando "Ctrl + N", depois, selecionando "Class" e "Next". Na próxima tela, trocaremos o pacote de "testes" para "br.com.alura.loja.util" e chamaremos a classe de "JPAUtil". Agora basta apertar "Finish".

Na `JPAUtil.java`, criaremos um método que será responsável por criar o `EntityManager` e que fará a utilização da `factory`. Mas, não desejamos precisar, toda vez que criarmos o `EntityManager`, criar uma nova `factory`. Para garantir que a `factory` está sendo criada uma única vez na aplicação, vamos transformá-la em um atributo estático da classe.

Então, faremos `private static final EntityManagerFactory FACTORY =` (podemos colocar como "final", porque se trata de uma constante. E "FACTORY" está em maiúsculo, porque é um nome de constante). Agora vamos trazer o código `Pesistence.createEntityManagerFactory("loja")` para a `JPAUtil.java`.

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

import javax.persistence.EntityManagerFactory;

public class JPAUtil {

    private static final EntityManagerFactory FACTORY = Persistence
            .createEntityManagerFactory("loja");
```

COPIAR CÓDIGO

Quando o Java carregar a classe `JPAUtil`, ele já criará o `EntityManagerFactory`. Agora, vamos criar um método que devolve um `EntityManager`, nós podemos chamar de `getEntityManager()`. Esse é o método que vai criar um `EntityManager`. Quando precisarmos, em qualquer lugar no projeto, nós chamamos este método. Continuaremos fazendo `return FACTORY.createEntityManager();`

```
public static EntityManager getEntityManager() {
    return FACTORY.createEntityManager();

}
```

COPIAR CÓDIGO

O objetivo da classe `JPAUtil` é isolar a criação do `EntityManager` e esconder também o `EntityManagerFactory()`. Agora, na classe `CadastroDeProduto`, podemos tirar os seguintes trechos de código:

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

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

COPIAR CÓDIGO

A única coisa que precisamos passar é o `EntityManager`, por isso, precisamos criar um. Faremos, `EntityManager em = JPAUtil.getEntityManager();`.

```
EntityManager em = JPAUtil.getEntityManager();
ProdutoDao dao = new ProdutoDao(em);

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

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

    }
```

COPIAR CÓDIGO

A criação e a transação do `EntityManager`, em uma aplicação real, um projeto ou aplicação web, não teria esses elementos. Provavelmente, usaríamos algum *framework*, como o Spring, que tem injeção de dependências. Logo, receberíamos injetada a classe DAO, que também teria a injeção do `EntityManager` automaticamente. Portanto, não teríamos nenhuma das linhas anteriores, com exceção da `dao.cadastrar(celular);`.

Teríamos apenas um atributo da classe DAO que seria injetado. Os *frameworks* facilitariam o processo. Mas, como não estamos usando nenhum *framework*, e aprendendo JPA puro, precisaremos das linhas apresentadas anteriormente para criar. Porém, é possível simplificá-las um pouco, e o objetivo do vídeo de hoje era esse.

Assim, o objetivo desse vídeo era simplificar um pouco o código e seguir com o padrão DAO para isolar o acesso, a parte da API da JPA na camada de persistência. Espero que tenham gostado, vejo vocês no próximo vídeo!!

Transcrição

Agora que já organizamos o código, podemos continuar com a parte de mapeamentos. Precisaremos fazer uma mudança na entidade `Produto`. Na tabela de "produtos" - no cadastro de produtos - pediram que adicionássemos mais informações.

Então, além do **nome**, do **preço** e da **descrição** do produto, precisamos cadastrar também: a **data**, isto é, quando esse produto foi cadastrado no sistema; e a **categoria**, pois temos algumas categorias de produtos que são vendidos na loja e eles precisam estar registrados.

Como são informações referentes ao produto, na própria entidade `Produto`, vamos adicionar essas novas informações, que são atributos que a JPA vai mapear para colunas no banco de dados. Para a data, podemos utilizar a API de datas do Java 8, então, pode ser um `private LocalDate`, se desejarmos salvar apenas a data, ou o `LocalDateTime` para salvar a data e a hora. No nosso caso, será apenas a data de cadastro, logo, `private LocalDate dataCadastro`.

```
@Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String descricao;
    private BigDecimal preco;
    private LocalDate dataCadastro
```

COPIAR CÓDIGO

Nós podemos instanciar com `LocalDate.now();` (para pegar a data atual). Sempre que um objeto `Produto` for instanciado, automaticamente preencherá o atributo com a data atual, por exemplo. O `LocalDate` é mapeado automaticamente no banco de dados, então, o Hibernate já sabe que ele virará uma coluna do tipo `Date` ou `DateTime` no banco de dados, sem que seja necessário colocar anotação nenhuma.

O outro campo é a categoria, `private Categoria categoria;` . A princípio, nos disseram que, por enquanto, a loja só vende produtos de três categorias: celular, informática e livros. Portanto, é fixo a uma dessas três categorias.

```
@Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String descricao;
    private BigDecimal preco;
    private LocalDate dataCadastro = LocalDate.now();
    private Categoria categoria;
```

COPIAR CÓDIGO

Onde temos `Categoria` , poderia ser uma *String*, mas não é tão interessante, porque podemos passar valores que não são os desejados. Então, podemos criar um *enum* do Java. Vamos apertar "Ctrl + 1", selecionar "Create enum 'Categoria'". Na próxima tela, deixaremos no próprio pacote de modelo, "br.com.alura.loja.modelo" e apertaremos "Finish". Agora, `Categoria` será um *enum*.

Dentro de `Categoria.java` , nós teremos as três constantes - os três valores possíveis - que são: `CELULARES` , `INFORMATICA` , ou `LIVROS` .

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

```
public enum Categoria {

    CELULARES,
        INFORMATICA,
        LIVROS;

}
```

COPIAR CÓDIGO

Na entidade `Produto`, categoria é um *enum*. Porém, temos um detalhe importante. Quando formos mapear um *enum*, temos que tomar cuidado em como o Hibernate e a JPA mapeiam a coluna `Categoria` para o banco de dados. Até então, estávamos usando os tipos primitivos - padrões - do Java, *Long*, *Int*, *String*, *BigDecimal*, *Double*, que são implícitos.

Por exemplo, se for *Long*, ele colocará um número. Se for *String*, ele colocará um *varchar* no banco de dados. A data virará um *Date*. O *BigDecimal* virará um *decimal*. E o *enum*? Como ele fará o relacionamento dessa coluna no banco de dados? Por padrão, se não indicarmos nada, o que ele vai inserir? Vamos fazer um teste e ver como funcionará.

Vamos gerar os métodos "Getters e Setters" da data de cadastro e da categoria. Só para facilitar, vamos criar um construtor. Então, vamos apertar o botão direito, depois selecionar "Source > Generate Constructor using Fields". Na próxima tela, desmarcaremos o "id", a "dataCadastro", e apertaremos "Generate".

Queremos gerar um construtor para facilitar na hora de instanciar um produto. Para ir direto no construtor ao passar o nome, descrição, preço e categoria, ao invés de *setar* tudo isso via método *setter*.

Na nossa classe `CadastroDeProduto` vai dar erro, porque temos agora que passar as informações, não mais via *setter*, mas no construtor: o nome; a descrição; o preço; o id não, porque é gerado automaticamente; a data de cadastro também não, porque já está sendo instanciada no atributo; e a categoria, que, como é um *enum*, nós passamos `categoria.CELULARES`.

```
public static void main(String[] args) {
        Produto celular = new Produto("Xiaomi Redmi", "Muito legal", new BigDecimal("800"), Categor

    }
```

COPIAR CÓDIGO

Mas, o que ele salvará no banco de dados, já que temos um *enum*? Por padrão, se não indicarmos, a JPA não colocará a coluna `Categoria.CELULARES` como um *varchar*, com o texto "CELULARES". Ela vai colocar uma coluna do tipo *Int* (Inteiro), e o valor que ela preenche lá é o valor da posição da constante. Então, `CELULAR` será 1, `INFORMATICA`, 2, e `LIVROS`, 3. Ela faz isso automaticamente.

```
1 CELULARES
2 INFORMATICA
3 LIVROS
```

COPIAR CÓDIGO

Se adicionarmos um produto com a `categoria.CELULARES`, ela mandará para o banco de dados - para a coluna de categoria - o número 1. Se adicionarmos `INFORMATICA`, será o número 2. Se adicionarmos `LIVROS`, será o número 3. Então, se trata da ordem da constante no *enum*. Isso um tanto estranho, porque existe um risco de alguém alterar essa ordem, teremos um desordenamento.

Além disso, se surgir uma nova constante, e alguém, ao invés de adicionar ao final, inserir em cima ou no meio, também embaralhará as ordens, e ele não atualizará sozinho, no banco de dados, as colunas dos registros que já existem. Sendo assim, mapear *enum* pela ordem das constantes é algo arriscado. O ideal é mapear pelo nome da constante.

Significa que não queremos que a coluna seja do tipo inteiro e sim um texto, um *varchar*, e que ele insira a constante `CELULARES`, `INFORMATICA` e `LIVROS`, de maneira independente da ordem declarada na constante. Sendo assim, se alguém

altera a ordem, nada muda no banco de dados.

Para ensinar isso à JPA, que não é mais o padrão, em cima do atributo `Categoria`, na classe `Produto.java`, colocaremos a anotação `@Enumerated()`. E, nessa anotação, temos como opções os parâmetros "ORDINAL" (que é o padrão, a ordem) ou "STRING". Logo, escolheremos `STRING`, para que ele cadastre o nome da constante no banco de dados, não a ordem.

```
@Enumerated(EnumType.STRING)
private Categoria categoria;

    public Produto(String nome, String descricao, BigDecimal preco, Categoria categoria) {
        this.nome = nome;
        this.descricao = descricao;
        this.preco = preco;
        this.categoria = categoria;
    }
```

COPIAR CÓDIGO

Vamos rodar o `CadastroDeProduto`, (Apertando "Run As > 1 Java Aplication"), e olhar o Console. Verificaremos que ele criou a tabela, "Hibernate: create table produtos". Ao analisar as colunas que ele criou, veremos "categoria varchar(255)". Se tivéssemos deixado como "ORDINAL", teríamos um *Int*, e ele mandaria a ordem da constante. Ele também fez o *insert* corretamente. Está pronto mais um mapeamento de um atributo do tipo *enum*.

Nos tipos do próprio Java, *Int, String, Long, Float, Double*, ou nas classes do Java, como a *LocalDate* e a *BigDecimal*, a JPA faz o mapeamento correto automaticamente, ou seja, não precisamos configurar nada. Apenas no caso de *enum* que, se não configurarmos, ele colocará o "ORDINAL" (pela ordem), mas, o ideal é sempre salvar o nome da constante, aí entra o `@Enumerated`.

A aula de hoje foi para discutirmos isso e, na próxima, faremos uma mudança em relação à `Categoria` para transformá-la em uma entidade e deixar o cadastro mais flexível e teremos que discutir um pouco sobre mapeamento de relacionamentos.
Vejo vocês lá!! Abraços!!

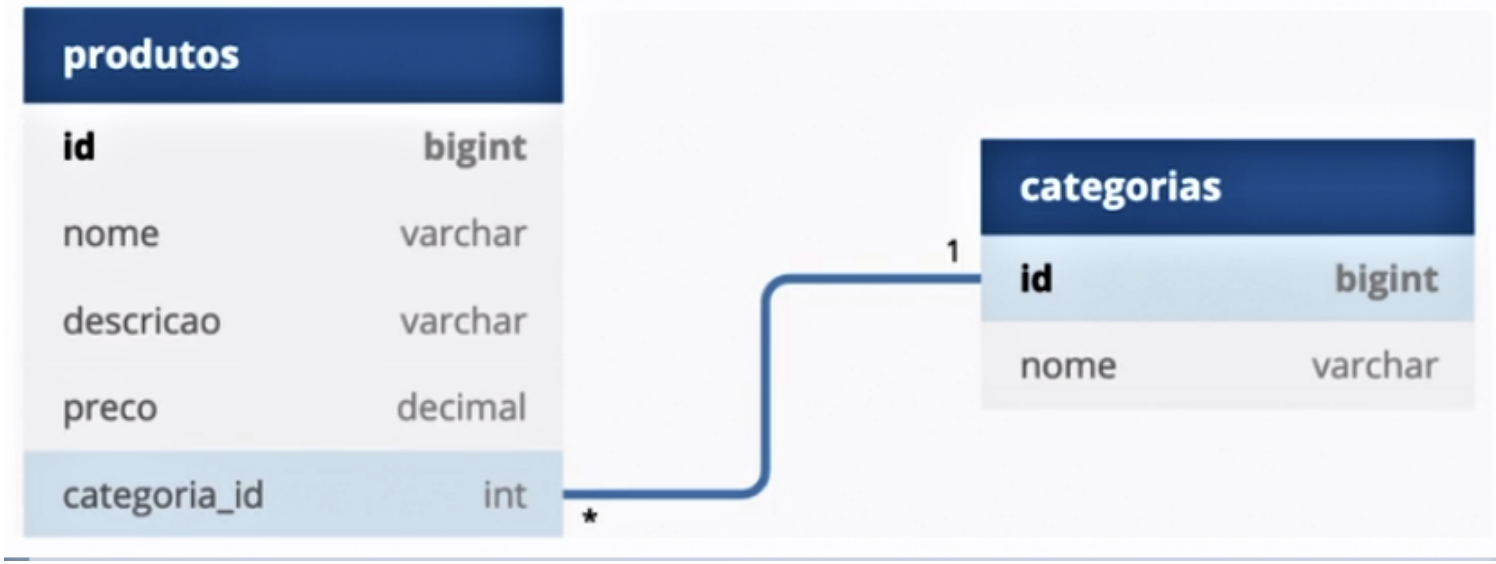
Transcrição

Na última aula, nós fizemos o mapeamento do *enum*. Nesta aula, nós precisaremos fazer uma alteração nele. Quando utilizamos *enum* no Java, passamos as constantes, que no nosso caso são, `CELULARES`, `INFORMATICA` e `LIVROS`, mas elas são fixas conforme o que está no código.

O problema é esse, para o sistema, não existe uma flexibilidade, portanto, se o usuário precisar cadastrar uma nova categoria, teremos que mexer no código fonte da aplicação, fazer um novo *build*, um novo *deploy* e subir outra vez a aplicação no ar.

Seria mais interessante se os próprios usuários pudessem cadastrar essas categorias para que fosse mais flexível. No sistema mesmo teria uma tela, um cadastro de categorias. Então, não queremos mais que fique fixo, mas sim deixar via sistema, com uma tabela para armazenar essas categorias. Ao fazer essa alteração, `Categoria` não será mais um *enum*, mas, sim, uma entidade, pois teremos uma tabela de Categoria.

A ideia é, justamente, a que está no seguinte diagrama.



Antes tínhamos apenas a tabela de "produtos". Agora, temos também a de "categorias" com duas colunas: o "id", que é a chave primária; e o "nome", referente ao nome da categoria. Existe um relacionamento entre o produto e a categoria. Todo produto pertence a uma categoria.

Então, na tabela de produtos, precisaremos adicionar a coluna "categoria_id", que, na verdade, é uma chave estrangeira, uma FK (Foreign Key) que aponta para o "id" na tabela de "categorias". Agora, precisaremos fazer um mapeamento desse relacionamento, que é algo que não tínhamos visto ainda. A JPA suporta normalmente mapear um relacionamento entre tabelas. Vamos ver como isso ficará.

O primeiro passo, na `Categoria.java`, é que `Categoria` não será mais um *enum*, portanto, trocaremos para `class`. Apagaremos as constantes, e teremos dois atributos, o `id` e o `nome`.

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

public class Categoria {

    private Long id;
```

```
        private String nome;

}
```

COPIAR CÓDIGO

A classe `Categoria` é uma entidade, então precisamos colocar as anotações da JPA, que podemos pegar da classe `Produto.java`. Então, pegaremos o `@Entity` e o `@Table(name = "produtos")`. Vamos copiar e colar, trocando o nome da tabela por `"categorias"`.

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

import javax.persistence.Entity;

@Entity
@Table(name = "categorias")
public class Categoria {

    private Long id;
        private String nome;

}
```

COPIAR CÓDIGO

Copiaremos também o `@Id` e `@GeneratedValue(strategy = GenerationType.IDENTITY)`.

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

import javax.persistence.Entity;
```

```
@Entity
@Table(name = "categorias")
public class Categoria {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
        private String nome;

}
```

COPIAR CÓDIGO

Tudo certo! O `nome` é `String`, ele mapeará como *varchar*, não muda nada. Só precisamos agora dos *Getters* e *Setters*. Criaremos também um construtor, assim como fizemos na entidade de produto. Então, no atalho, selecionaremos "Source > Generate Constructor". Na próxima tela, desmarcaremos o "id" (porque ele é gerado pelo banco de dados) e deixaremos só o "nome". Agora basta apertar "Generate".

```
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
        private String nome;

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

COPIAR CÓDIGO

Quando alguém der `new` na classe `Categoria`, já tem que passar o nome. Vamos gerar o método `getNome()` e colocar também um `setNome`. Em teoria, nós não precisaríamos do *Setter*, porque, ele já vem na hora de dar `new`, mas, vamos

deixar aqui, caso precise. Já temos a nossa entidade de `Categoria` mapeada.

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

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

COPIAR CÓDIGO

É simples mapear uma tabela do banco de dados, basta criar a classe e colocar as anotações da JPA, declarar os atributos e fazer os mapeamentos usando as anotações da JPA conforme a necessidade. Porém, na classe `Produto.java` , `Categoria` agora não é mais um *enum*, então, apagaremos o `@Enumerated(EnumType.STRING)` .

Agora, teremos um problema, porque a JPA detectará que, na entidade `Produto` , existe um atributo `Categoria` , e que seu tipo não é mais primitivo do Java, não é mais um *enum* e nem uma classe do Java. Ela detectará que esse tipo é uma outra classe do nosso projeto e que essa classe é uma entidade (`@Entity`). Então, a JPA automaticamente saberá que isso é um relacionamento entre `Produto` e `Categoria` , isto é, um relacionamento de duas entidades.

Desta maneira, temos a obrigação de dizer à JPA qual é a cardinalidade desse relacionamento. Se um produto tem uma única categoria ou várias categorias, é um para um, um para muitos, muitos para um, ou seja, qual é a cardinalidade. Se observarmos o desenho do diagrama, veremos que, de produtos para categorias, temos: muitos para um. Isto é, um produto tem uma única categoria, mas uma categoria pode estar vinculada a vários produtos.

Então, de produtos para categorias: asterisco, 1. Que quer dizer: muitos para um. Na JPA, para informarmos que a cardinalidade desse relacionamento é "muitos para um", temos uma anotação, `@ManyToOne`. Ou seja, muitos produtos estão vinculados com uma `Categoria`. Uma categoria pode ter vários produtos, mas o produto tem uma única categoria.

```
@ManyToOne
private Categoria categoria;

public Produto(String nome, String descricao, BigDecimal preco, Categoria categoria) {
    this.nome = nome;
    this.descricao = descricao;
    this.preco = preco;
    this.categoria = categoria;
}
```

COPIAR CÓDIGO

Existem algumas anotações da JPA para relacionamento, o `@ManyToOne`, `@OneToMany` (que é o contrário), `@OneToOne` e `@MayToMany`. A escolha dependerá da cardinalidade, do tipo de relacionamento entre as tabelas. Agora, no `CadastroDeProduto.java` está dando um erro em `Categoria.CELULARES`, porque `Categoria` não é mais um *enum*. Precisamos de uma categoria cadastrada no banco de dados para associá-la com esse produto.

Sendo assim, vamos colocar uma categoria `celulares`, mas dará um erro de compilação. Vamos selecionar o comando "Ctrl + 1"e criar uma variável local. Nós precisamos da `Categoria celulares`, então, `new Categoria()`, e passaremos o nome `"CELULARES"`.

```
public static void main(String[] args) {
    Categoria celulares = new Categoria("CELULARES");
```

COPIAR CÓDIGO

Mas, nós precisamos salvar essa categoria no banco de dados antes de salvar o produto, ou teremos problemas. Vamos rodar e ver o problema selecionando "Run As > 1 Java Application". No Console, veremos que deu uma *exception* "Transient Property Value Exception". Significa que, existe um valor na propriedade que é "transient", não está salvo, não está persistido.

Na próxima aula, nós discutiremos sobre estados da entidade para saber o que é "transient" e como funciona essa transição de estados de uma entidade na JPA. Mas, a questão é: nós temos uma categoria e acabamos de instanciá-la, mas ela não está salva no banco de dados. Na hora em que criamos um produto, nos vinculamos a essa categoria que não está persistida no banco de dados.

E, na hora que chamamos o `dao.cadastrar(celular);`, a JPA detectou. Ela foi fazer um *insert* do produto, mas viu que esse produto estava relacionado com uma categoria, mas essa categoria ainda não tinha sido persistida no banco de dados. Assim, ela considera que isso é um erro e lança uma *exception*.

O correto é salvar a categoria e, depois, salvar o produto. Sendo assim, a categoria já estará gerenciada pela JPA e estará na tabela, terá um id, e será possível fazer o relacionamento. Então, precisamos cadastar no banco de dados a categoria. Nós havíamos criado uma classe `ProdutoDAO`, podemos criar, também, uma categoria DAO seguindo o mesmo modelo.

Portanto, vamos selecionar com o comando "Ctrl + C" o `ProdutoDao.java` e colar e renomear para `CategoriaDao` em "Enter a new name for 'ProdutoDao'". Agora basta apertar "Ok". Vamos abrir o `CategoriaDao.java`, ele recebe o `EntityManager` da mesma maneira. Enfim, é a mesma estrutura, as classes DAO serão parecidas. Mas, em `CategoriaDao.java`, vez de ser um `Produto`, será uma `Categoria`. Depois, apertaremos "Ctrl + Shift + O" para importar.

```
public CategoriaDao(EntityManager em) {
    this.em = em;
    }

    public void cadastrar(Categoria categoria) {
```

```
            this.em.persist(categoria);
        }

    }
```

COPIAR CÓDIGO

Está pronta a nossa classe DAO. Agora, em `CadastroDeProduto.java`, nós criamos a `Categoria celulares`, o `Produto celular`, o `EntityManager`, o `ProdutoDao`, vamos apenas renomear a variável `dao` para `produtoDao`, porque agora precisaremos ter duas classes DAO - dois objetos DAO.

```
ProdutoDao produtoDao = new ProdutoDao(em);
```

COPIAR CÓDIGO

Nós precisaremos instanciar também o `CategoriaDao`. Vamos copiar a linha anterior e, ao invés de `ProdutoDao`, será `CategoriaDao`. O nome da variável será `categoriaDao`, e apertaremos "Ctrl + Shit + O" para importá-la. Algo interessante é que podemos compartilhar o mesmo `EntityManager` entre as várias classes DAO.

```
CategoriaDao categoriaDao = new CategoriaDao(em);
```

COPIAR CÓDIGO

Então, nós iniciamos a transação. Logo abaixo, fizemos o *commit*. Antes de salvar o `produtoDao` no banco de dados, chamaremos `categoriaDao.cadastrar()`, passando a categoria `celulares`.

```
EntityManager em = JPAUtil.getEntityManager();
        ProdutoDao produtoDao = new ProdutoDao(em);
        CategoriaDao categoriaDao = new CategoriaDao(em);
```

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

        categoriaDao.cadastrar(celulares);
        produtoDao.cadastrar(celular);

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

}
```

COPIAR CÓDIGO

Agora, sim, salvamos a categoria, `CategoriaDao.cadastrar(celulares)`, no banco de dados, e depois salvamos o produto, `produto.Dao.cadastrar(celular)`, no banco de dados, já que ele está vinculado com essa categoria. Por isso, é provável que não aconteça mais a *exception* do "transient property value", pois o produto está associado com uma categoria que, sim, está persistida.

Vamos rodar ("Run As > 1 Java Application") e verificar se ele faz o *insert* corretamente no banco de dados. Ele fez o *insert* da categoria, "Hibernate: insert into categorias", e o *insert* do produto, "Hibernate: insert into produtos". Também criou a tabela de categorias, "Hibernate: create table categorias", e a tabela de produtos, "Hibernate: create table produtos".

Ele também fez um "alter table" para adicionar a chave estrangeira na tabela produtos, que agora precisa da coluna do id da categoria. Portanto, ele alterou a tabela e criou a "constraint" *foreign key*.

É assim que fazemos mapeamento de relacionamento na JPA. Sempre que temos uma entidade, onde o atributo é uma outra entidade, a JPA automaticamente identifica que é um relacionamento, e então, precisamos indicar qual é a cardinalidade. Ela não tem um valor padrão para a cardinalidade, nós que precisamos configurar, e isso dependerá de como escolheremos modelar o banco de dados.

Precisamos saber como está modelado no banco de dados para escolher a anotação de relacionamento. O último detalhe é que, na hora de persistir, temos que ter cuidado, porque, se estamos persistindo com uma entidade e vinculando com outra entidade, essa outra precisa estar persistida antes, ou receberemos uma *exception* "transiente property value excepetion".

O objetivo da aula de hoje foi discutir um pouco sobre relacionamentos, aprendemos como mapeá-lo e também como funciona a parte de persistência. Espero que tenham gostado. Na próxima aula discutiremos sobre os estados das entidades, como funciona o "transient" para a JPA. Vejo vocês lá!! Abraços!!

Transcrição

Na aula de hoje discutiremos um pouco sobre o ciclo de vida das entidades da JPA. Sempre que trabalhamos com as entidades, quando instanciamos um objeto e chamamos os métodos do `@EntityManager`, elas ficam trafegando entre alguns estados no ciclo de vida e é interessante conhecer esses estados para entender melhor como a JPA trabalha "debaixo dos panos" (de forma oculta) e, principalmente, entender os erros, algumas *exceptions* que ela lança, a depender da situação e do que estivermos fazendo na aplicação.

Para deixar mais compreensível, vamos fazer uma alteração na classe `CadastroDeProduto`, temporariamente, para entendermos a questão do ciclo de vida. Vamos apagar a parte de `Produto celular`, as classes DAO, `ProdutoDao` e `CategoriaDao` e também o `categoria.Dao`, deixando apenas a nossa categoria.

```
public class CadastroDeProduto {

    public static void main(String[] args) {
        Categoria celulares = new Categoria("CELULARES");

        EntityManager em = JPAUtil.getEntityManager();

        em.getTransaction().begin();

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

```
}
```

COPIAR CÓDIGO

Então, estamos instanciando uma entidade, `celulares`, passando um parâmetro, `"CELULARES"`, que vai ser setado no atributo. Também criamos o `EntityManager`, demos um `begin()` na transação um `commit()` e um `close()`. Agora faremos manualmente a classe DAO para entendermos melhor e mais simples.

Então, tínhamos um `em.persist(celulares)` (EntityManager persist na categoria celulares). Como funciona a questão das entidades, do ciclo de vida. Toda vez que instanciamos uma entidade, por exemplo: `Categoria celulares = new Categoria("CELULARES")`, neste momento, ela ainda não está salva no banco de dados, o `EntityManager` não conhece essa categoria, então, como isso funciona do ponto de vista da JPA?

Dicutindo, portanto, sobre o **ciclo de vida**, quando damos "new", isto é, quando instanciamos uma entidade, para a JPA, a entidade está em um estado chamado de **TRANSIENT**. O estado transient é de uma entidade que nunca foi persistida, não está gravada no banco de dados, não tem um *id* e não está sendo gerenciada pela JPA.

A JPA desconhece essa entidade. É como se fosse um objeto Java puro, que não tem nada a ver com persistência. Esse é o primeiro estado de uma entidade. Nesse estado, se alteramos o atributo da entidade ou qualquer outra coisa que façamos com a entidade, o `EntityManager` não está gerenciando, nem verificando, e não vai sincronizar com o banco de dados se nós *commitarmos* e fizermos o *close* do `EntityManager`.

Por exemplo, se comentarmos a seguinte linha:

```
//em.persist(celulares);
```

COPIAR CÓDIGO

Nós instanciamos uma entidade, abrimos um `EntityManager`, demos um `begin()`, um `commit()` e um `close()`. Agora, se rodarmos o código, ele imprimirá no Console que só rodou os comandos para criar a tabela. Não deu *insert*, não fez nada. O motivo é que a entidade é transient, isto é, não está sendo gerenciada pela JPA. Assim, ela simplesmente ignora a entidade.

Porém, quando chamamos o método `persist()`, ela move do estado transient para o estado **MANAGED** ou gerenciado. Managed é o principal estado que uma entidade pode estar, portanto, tudo que acontece com uma entidade nesse estado, a JPA observará e poderá sincronizar com o banco de dados, a depender do que fizermos.

É o caso desta parte em `CadastroDeProduto`, quando chamamos o método `em.persist(celulares)` e, agora, a JPA está olhando para essa entidade. Desta maneira, se decidirmos alterá-la, a JPA observará. Ainda no `CadastroDeProduto`, havíamos criado o objeto de nome `"CELULARES"`.

Se na linha abaixo do `persist()` setarmos um nome para outro valor, por exemplo, `celulares.setNome("XPTO");` estamos mexendo na entidade, alterando um atributo dela. Como ela está no estado managed, a JPA está olhando isso.

```
        Categoria celulares = new Categoria("CELULARES");

        EntityManager em = JPAUtil.getEntityManager();
        em.getTransaction().begin();

        em.persist(celulares);
        celulares.setNome("XPTO");

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

}
```

COPIAR CÓDIGO

Quando commitarmos a transação, fizermos um `commit()`, ou, se não finalizarmos a transação, mas se tentarmos sincronizar o estado dessa entidade com o banco de dados, existe um método `flush()` no `EntityManager` que serve para este caso, em que não queremos *commitar* a transação, ainda faremos algumas operações, mas já desejamos sincronizar essa entidade com o banco de dados para ela gerar o id ou vinculá-la a outra.

No momento em que fazemos o `commit()` ou o `flush`, a JPA pega todas as entidades que estiverem no estado managed e sincroniza com o banco de dados. Então, tínhamos uma entidade que era transient, está gerenciada e, depois que commitamos, ele perceberá que a entidade não tem id, que ela era transient, ou seja, é necessário fazer um *insert* dela no banco de dados.

Vamos rodar ("Run As > 1 Java Application) e, no Console, ele deu um *insert* na categoria, "Hibernate: insert into categorias (id, nome) values (null, ?)", e, na sequência, deu um *update*, "Hibernate: update categorias set nome=? where id=?", significa que ele atualizou a categoria.

E o motivo de ter atulizado a catagoria é que fizemos o `persist()`, mas, na sequência, mudamos o nome dela. Como a categoria está gerenciada, se alteramos o atributo, ele saberá que é necessário fazer o *update*, pois atualizamos alguma informação da entidade. Em outras palavras, se a categoria está gerenciada, "managed", a JPA observará, e se alterarmos ela, a JPA sincronizará com o `commit()` ou com `flush()` ao banco de dados.

A partir do momento em que fechamos o `EntityManager`, isto é, `em.close()` ou `clear()` (para limpar as entidades gerenciadas do `EntityManager`), a categoria muda de estado. Se ela estava salva antes, passa para um estado chamado de **DETACHED**, que é um estado destacado.

O detached é um estado em que a entidade não é mais transient, porque tem id, já foi salva no banco de dados, porém, não está mais sendo gerenciada. Portanto, se mexermos nos atributos, a JPA não disparará *update* e nem fará mais nada. Vamos simular esse estado.

Em `CadastrDeProduto.java`, temos a entidade, o `persist()`, o atributo e o `commit()`, ele sincronizou com o banco de dados (fez o *insert* e o *update*) e fizemos o `em.close()` do `EntityManager`. Se, na sequência, pegarmos a entidade `celulares.setNome()` e mudarmos o nome de novo, por exemplo, para `"1234"` e rodarmos o código, ele não deveria fazer um segundo *update*.

```
      Categoria celulares = new Categoria("CELULARES");

        EntityManager em = JPAUtil.getEntityManager();
          em.getTransaction().begin();

          em.persist(celulares);
          celulares.setNome("XPTO");

          em.getTransaction().commit();
          em.close();

          celulares.setNome("1234");
      }

  }
```

COPIAR CÓDIGO

Nós mexemos no nome depois de fechar o `EntityManager`, então, ele não está mais "managed", está no estado "detached". Conferindo no Console, está correto, ele fez um *insert* e um único *update*, de quando ainda estava aberto o `EntityManager`, isto é, quando a entidade ainda estava gerenciada. Se alteramos qualquer coisa abaixo do `em.close()`, ele vai ignorar.

Esse é o ciclo de vida para quando estamos falando em persistência, em *insert*, criação de objetos. Estudamos o que acontece quando criamos uma entidade, instanciamos, chamamos o `persist()`, fechamos o `EntityManager` ou mexemos em um atributo da entidade. Nesta aula, nós simulamos todas essas etapas no código.

Agora, já entendemos como funciona a parte de criação de uma entidade. No próximo vídeo, discutiremos sobre outros cenários, outras possíveis transições de estados e como podem acontecer. Vejo vocês lá!!

Transcrição

Ainda na parte de transição de estados, quando instanciamos, persistimos a entidade e ela fica no estado de managed, nós encontramos uma situação no último vídeo que é a atualização. Nós não tínhamos visto ainda como atualizar uma entidade. Só havíamos colocado, nas classes DAO, um método para cadastrar uma categoria ou um produto, `public void cadastrar(Categoria categoria) {` , mas não vimos como faz para atualizar.

Quando fizemos a simulação, já era possível ver uma atualização. Vamos rodar novamente ("Run As > 1 Java Application). Analisando o Console, perceberemos que ele faz o *insert* e, na sequência, um *update*, isto é, já está atualizando uma entidade.

Toda vez que a entidade está no estado managed, está gerenciada, qualquer mudança que fizermos em algum atributo,a JPA detectará e, no `commit()` ou no `flush()` do `EntityManager` , ela vai, automaticamente, sincronizar essas mudanças no banco de dados, porque sabe que é necessário fazer o *update* no banco de dados.

Portanto, é assim que funciona o *update* no banco de dados, basta pegar uma entidade que esteja no estado managed e alterar os atributos dessa entidade. Quando fizermos o `commit()` da transação ou um `flush()` manualmente, esse estado será sincronizado automaticamente com o banco de dados.

Mas, o problema é que não sabemos se a entidade está no estado managed, talvez ela já esteja no estado detached se chamarmos o `clear()` ou se fecharmos o `EntityManager` com o `close()` . Nesta situação, o *uptade* não acontecerá.

Em `CadastroDeProduto.java`, nós havíamos alterado o nome, commitamos, fizemos o `close()` do `EntityManager`, e alteramos o nome da entidade - da categoria - para `"1234"`, e um segundo *update* não foi disparado.

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

        em.persist(celulares);
        celulares.setNome("XPTO");

        em.getTransaction().commit();
        em.close();

        celulares.setNome("1234")
    }

}
```

COPIAR CÓDIGO

Depois de fechado o `EntityManager`, ele está no estado detached, e, neste estado, nada que alterarmos na entidade será sincronizado automaticamente com o banco de dados. Então, surge a questão de como voltar a entidade para o estado managed.

Se ao invés de fecharmos o `EntityManager`, escrevermos `clear()`, o `EntityManager` ainda estará aberto, quer dizer que ainda podemos trabalhar com ele, porém, com o `clear()` nós tiramos todas as entidades, todas estão detached. Como fazemos para voltar a entidade para o estado managed? Pois, se quisermos atualizar uma informação em `celulares.setNome("1234");` ela não será atualizada.

Ainda considerando o nosso exemplo anterior, fizemos um `clear()`, alteramos o nome e, agora, vamos dar um `flush()`, isto é, `em.flush();`. Vamos também tirar o `commit()` e trocar por `flush()`, porque ainda não queremos commitar a transação, mas, queremos sincronizar com o banco de dados. Será que agora ele fará dois *updates* ou apenas um? Vamos rodar ("Run As > 1 Java Application").

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

        em.persist(celulares);
        celulares.setNome("XPTO");

        em.flush();
        em.clear();

        celulares.setNome("1234")
        em.flush();
    }

}
```

COPIAR CÓDIGO

No Console, notaremos que ele fez apenas um *update*, que foi `celulares.setNome("XPTO")`, quando mudamos o nome para `"XPTO"`. No `flush()`, ele disparou um *insert* e um *update* (que havíamos persistido e mudado o nome), mas demos um `clear()` e, agora, a entidade não está mais gerenciada. Então, por mais que tenhamos alterado um atributo, quando chamarmos o `flush()`, ele não vai sincronizar.

Então, o que precisamos fazer se quisermos voltar para o estado managed? Existe outro método que não havíamos estudado ainda, o `merge()`, que tem como objetivo pegar uma entidade que está no estado detached e retorná-la ao estado managed

(gerenciado).

A partir dali, qualquer mudança que fizermos na entidade será analisada e sincronizada ao banco de dados quando realizarmos o `commit()` da transação ou `flush()` .Vamos simular essa situação. Nós fizemos o `clear()` e a entidade está no estado detached.

Agora, vamos chamar `em.merge()` , passando a entidade `celulares` , que, então, volta para o estado managed. Continuando, alteraremos o nome, e faremos um `flush()` . Portanto, ela deveria fazer dois *updates*. Vamos rodar e verificar se isso de fato acontecerá.

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

        em.persist(celulares);
        celulares.setNome("XPTO");

        em.flush();
        em.clear();

        em.merge(celulares);
        celulares.setNome("1234")
        em.flush();
    }

}
```

COPIAR CÓDIGO

Ao observar o Console, perceberemos que ele nos mandou uma *exception*, "javax.persistence.PersistenceException" e indicou algo importante: "No default constructor for entity: : br.com.alura.loja.modelo.Categoria". Significa que a entidade `Categoria` não tem um construtor *default*.

Retornando à entidade `Categoria.java`, tínhamos criado o seguinte construtor:

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

COPIAR CÓDIGO

Com a intenção de, na hora de dar `new` na categoria, passar também o nome. Mas a JPA precisa que as entidades tenham um construtor padrão. Até então, ela não havia reclamado disso, porque estávamos fazendo apenas *insert*, mas quando chamamos um `merge()`, ele faz um *select* no banco de dados. Ao carregar a entidade do banco de dados e criar o objeto, a JPA precisa do construtor *default*.

Logo, precisamos inserir o construtor *default* nas entidades, tanto na `Categoria.java` quanto na `Produto.java`. Assim, na `Categoria.java` teremos:

```
public Categoria() {
    // TODO Auto-generated constructor stub
}
```

COPIAR CÓDIGO

E no `Produto.java`:

```
@ManyToOne
private Categoria categoria;

public Produto() {
    // TODO Auto-generated constructor stub
}
```

COPIAR CÓDIGO

Vamos rodar mais uma vez nossa classe `CadastroDeproduto.java` e analisar se ele fará dois *updates* agora. No Console, reopararemos que ele fez um *insert*, o primeiro *update* (do nome "XPTO") e fez o *select*, por causa do `merge()`, porém, não fez o *update* referente à mudança de nome para `"1234"`.

Isso aconteceu, porque quando chamamos o método `merge()` e passamos uma entidade, ele não muda o estado dessa entidade para managed, ele devolve uma nova referência, e esta sim, estará no estado managed. Mas, a que passamos como parâmetro, no nosso caso, `"celulares"`, continua detached.

```
em.merge(celulares);
celulares.setNome("1234")
em.flush();
```

COPIAR CÓDIGO

Por isso, quando mudamos o atributo, não adiantou nada, já que fizemos essa mudança na entidade que ainda está detached. Sendo assim, se desejarmos mudar o atributo, é necessário criar uma nova categoria e atribuir, ou, para mudar de fato o objeto, precisamos fazer `celulares = em.merge(celulares);` ("celulares", que é o nosso objeto, agora aponta para o retorno do método merge()). Ou seja, o método `merge()` devolve a entidade no estado managed.

Vamos rodar e analisar o Console. Agora ele fez o *insert*, o *update*, o *select* do `mege()` e o *update* da atualização, já que, agora, sim, estamos trabalhando em cima da entidade que está managed. É assim que o método `merge()` funciona.

Comumente, nos projetos, temos um método para atualizar. Funciona assim: temos um método para cadastrar e estamos na categoria DAO e teremos um método para atualizar uma entidade.

```
    public void cadastrar(Categoria categoria) {
        this.em.persist(categoria);
    }

    public void atualizar(Categoria categoria) {
        }

}
```

COPIAR CÓDIGO

O que esse método faz? Em teoria, ele não precisa fazer nada, pois já recebe a entidade com as informações alteradas, mas, como não sabemos se essa entidade está managed, nós, de certa forma, a forçamos a ficar managed. Então, colocamos `this.em.merge(categoria)`, só para garantir que essa categoria estará no estado managed.

```
public void atualizar(Categoria categoria) {
    this.em.merge(categoria);
}
```

COPIAR CÓDIGO

Não há necessidade de alterar os atributos, porque eles já chegam atualizados, isto é, já chegou uma categoria detached com todos os atributos atualizados, então, quando chamarmos o `merge()`, ele apenas a coloca no estado managed e, depois, quando fizermos o `flush()` da transação, ele disparará o *update* automaticamente.

Aparentemente, não precisamos do método `merge()`, porque sua função não é atualizar, mas, sim, para o caso de, se por um acaso a entidade estiver detached, o método `merge()` a voltará para o estado managed. Para atualizar no banco de

dados, vamos: carregar a entidade do banco, mudar o atributo, commitar a transação. E, pronto, já está managed.

Quando carregamos do banco de dados, ela já está managed. Então, se alterarmos qualquer atributo e fizermos o `flush()` ou o `commit()`, ele fará a sincronização com o banco de dados (fará o *updtae* automaticamente).

No próximo vídeo, falaremos de outros estados das entidades da JPA. Vejo vocês lá!!

Transcrição

Nós estávamos discutindo sobre os estados da entidade da JPA e ainda não estudamos a parte de consultas, que será tema da próxima aula. Mas, apenas para complementar, existe mais uma situação no ciclo de vida, que é quando uma entidade está no banco de dados.

Isto é, já fechamos o `EntityManager` e abrimos um novo, não temos mais uma referência para a entidade no estado detached, então, como faremos para trazê-la para o estado managed. Basicamente, queremos trazê-la do banco para o estado managed, para isso, precisaremos dos métodos `find()` / `createQuery()`. Nós veremos estes métodos de consulta na próxima aula.

Portanto, existe mais essa transição de estados. Além do managed para o banco de dados, quando estamos commitando ou fazendo um `flush()` no `EntityManager`, existe a transição do banco de dados para o managed, quando fazemos uma consulta, uma *query*. Agora, voltando para o código, em `CadastroDeProduto.java`, faltou apenas um último estado, que é quando excluímos uma entidade.

Para excluir, temos a seguinte situação: do estado managed, podemos chamar o método `remove()` do `EntityManager` e ela passa para o estado **REMOVED**. Quando o `commit()` ou o `flush()` for chamado, ele vai sincronizar o `remove()` com o banco de dados disparando um *delete*. Vamos simular essa situação.

Em `CadastroDeProduto.java`, temos a entidade, persistimos, atualizamos o nome, fizemos um `flush()`, ele disparou o *update*, demos um `clear()`, voltamos a entidade para o estado managed chamando o `merge()`, atualizamos o nome e fizemos um `flush()`.

Depois disso, ela ainda está managed, porque não fizemos um `clear()` nem `close()`, então podemos excluí-la do banco de dados com `em.remove()`, passando a entidade `celulares`, e quando fizermos um `flush()`, ela deve disparar um *delete* no banco de dados.

```
celulares = em.merge(celulares);
celulares.setNome("1234");
em.flush();
em.remove(celulares);
em.flush();
```

COPIAR CÓDIGO

Vamos rodar o código e ver a saída no Console. Ele fez o *insert*, o *update*, o *select* devido ao `merge()`, mais um *update* e um *delete*. Então, a JPA deleta baseada no *id*, no atributo da chave primária. Com isso, fechamos todos os cenários possíveis, todas as transições de estados de uma entidade JPA.

Quando uma entidade nasce, ela está no estado transient, para salvá-la no banco de dados, temos que movê-la para o estado managed e, então, entra o método `persist()`. Commitamos a transação ou fizemos um `flush()` no `EntintyManager`, ele sincroniza com o banco de dados. Se fecharmos esse `EntityManager` (ou se dermos um `clear()`), a entidade vai para o estado detached.

Se temos a entidade detached, podemos chamar o método `merge()` para trazê-la novamente ao método managed, ou, se ela já está no banco de dados, é possível fazer um `find()` ou uma `createQuery()` e, por fim, se quisermos excluí-la do banco de dados, do managed, chamamos o `remove()` e ela passa para o estado de removed.

Esses são os estados possíveis e as transições que acontecem na JPA. Conforme vamos utilizando o `EntityManager` e esses métodos, ocorrem transições de estados. Se quisermos complementar a nossa classe DAO, `CategoriaDao.java`, nós já temos o método `cadastrar()` e o `atualizar()`, para fazer um método para excluir, será parecido.

```
public void remover(Categoria categoria) {
    this.em.remove(categoria);
```

COPIAR CÓDIGO

A única ressalva é que essa categoria precisa estar no estado managed. Pode acontecer de não termos essa garantia, de não sabermos se ela está managed. Se ela estiver detached e chamarmos para o `remove()`, será que teremos problemas? Vamos simular o nosso código do método `main()`. Antes de fazer o `remove()`, vamos fazer o `clear()`. Agora ela está detached, e, na sequência, chamaremos o `remove()` para testar se ele deletará do banco de dados.

```
celulares = em.merge(celulares);
celulares.setNome("1234");
em.flush();
em.clear();
em.remove(celulares);
em.flush();
```

COPIAR CÓDIGO

Vamos rodar e ao analisar o Console, receberemos uma *exception*, "IlegalArgumentException: Removing a detached instance", ou seja, não é permitido remover uma entidade que está detached, ela precisa estar managed. Em `CategoriaDao.java`, nós não sabíamos se, no método `atualizar()`, a classe estava managed e chamamos o `merge()` para garantir.

Podemos fazer o mesmo no método `remover()`, chamar o método `merge()` para forçá-la a ficar managed, e na sequência, fazer o `remove()`. Então, pegaremos o `merge()` reatribuindo o objeto, isto é, `categoria = em.merge(categoria)`, e depois fazemos o `remove(categoria)` em cima da categoria.

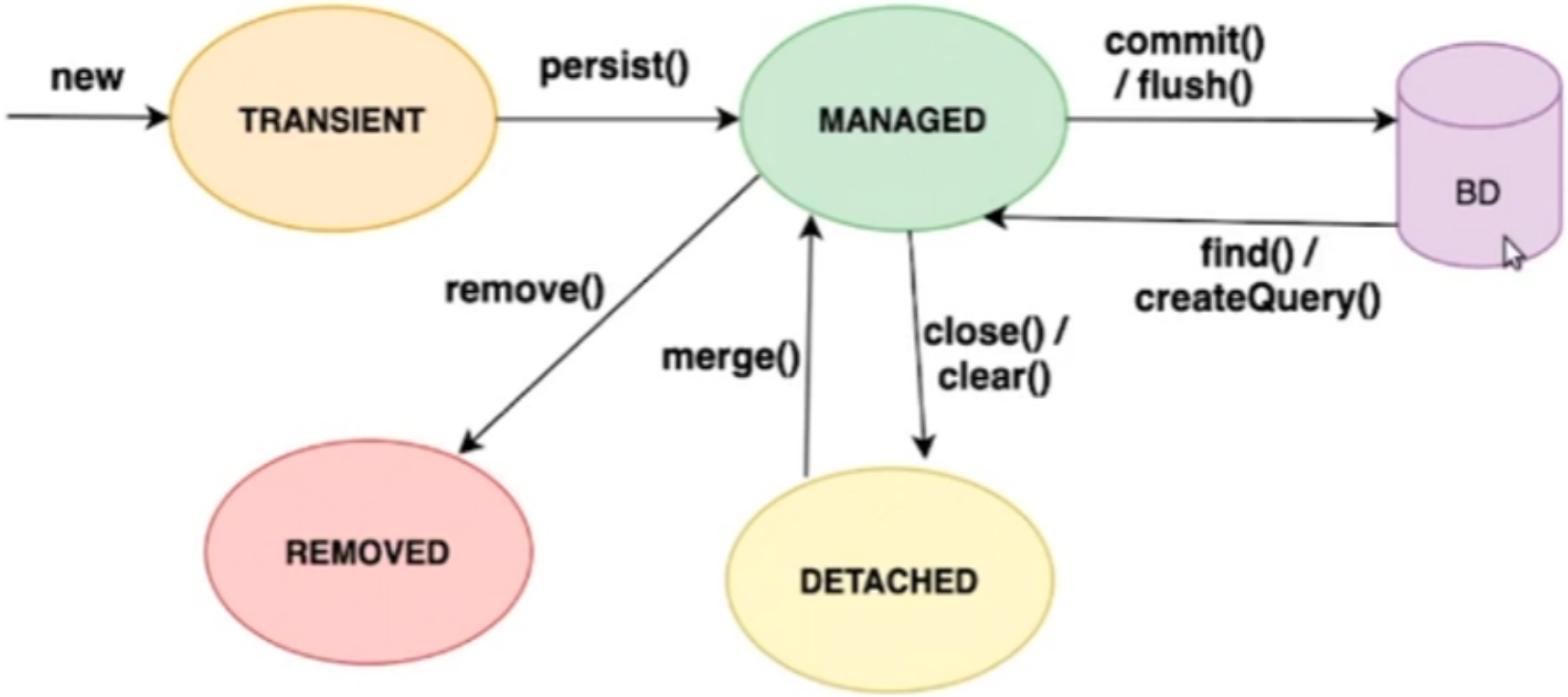
```
public void remover(Categoria categoria) {
    em.merge(categoria);
        this.em.remove(categoria);
    }
```

COPIAR CÓDIGO

É importantíssimo lembrar de reatribuir. Estamos fazendo `merge()`, mas não guardamos a entidade *mergeada*, a entidade que está no estado managed, então, estamos mexendo na categoria que ainda está detached, por isso, precisamos reatribuir. Podemos fazer desta maneira, `categoria = em.merge(categoria);`, só para garantir que a entidade está managed.

Assim fica uma classe DAO com a JPA, temos o `cadastrar()`, o `atualizar()` e o `remover()`, faltam apenas os métodos de consulta, que discutiremos na próxima aula. Espero que tenham aprendido um pouco sobre transições de estados de uma entidade, como elas funcionam, as possíveis transições e as excessões. Vejo vocês na próxima aula!! Abraços!!

Ciclo de vida



Transcrição

Até então, havíamos estudado os métodos de persistência (salvar, atualizar e excluir) e o ciclo de vida. Nesta aula, focaremos na parte de consultas. Vamos continuar na classe `CadastroDeProduto.java` e nela extrairemos todo o código do método `main()` a seguir para um método separado.

```
public static void main(String[] args) {
        Categoria celulares = new Categoria("CELULARES");
        Produto celular = new Produto("Xiaomi Redmi", "Muito legal", new BigDecimal("800"), celular

        EntityManager em = JPAUtil.getEntityManager();
        ProdutoDao produtoDao = new ProdutoDao(em);
        CategoriaDao categoriaDao = new CategoriaDao(em);

        em.getTransaction().begin();

        categoriaDao.cadastrar(celulares);
        produtoDao.cadastrar(celular);

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

}
```

COPIAR CÓDIGO

Para isso, apertaremos o botão direito e selecionaremos "Refactor > Extract Method". Na próxima tela, nomearemos o método como "cadastrarProduto". Assim, deixaremos isolada essa parte de cadastro da categoria e do produto. Ao ser chamado, o método `main()` chama o `CadastrarProduto()` e, em seguida, fazemos todo o código que já existia: criamos uma `Categoria`, um `Produto`, um `EntityManager` as DAOS, salvamos e fechamos tudo.

Já fechamos o `EntityManager`, está tudo no banco de dados e no estado detached, nada está gerenciado. Agora, queremos trabalhar diretamente com o banco de dados fazendo consultas, e a JPA possui alguns métodos para isso. Basicamente, podemos consultar uma entidade pelo id ou fazer *select*, uma *query*, uma consulta para carregar várias entidades, a depender do nosso objetivo.

Portanto, para fazer uma consulta por id, considerando `Long id = 1l;`, isto é, que queremos buscar uma entidade de id 1, nós precisaremos, primeiro, de um `EntityManager` e também criar uma classe DAO.

```
public static void main(String[] args) {
    cadastrarProduto();
        Long id = 1l;
        EntityManager em = JPAUtil.getEntityManager();
        ProdutoDao produtoDao = new ProdutoDao(em);
```

COPIAR CÓDIGO

Agora, podemos criar um método na classe DAO,o retorno deste método é um objeto do tipo `Produto`, ele será chamado de `buscarPorId()`, e receberá um parâmetro `Long id`.

```
public Produto buscarPorId(Long id) {

}
```

COPIAR CÓDIGO

Na JPA, para consultar por id, o `EntityManager` possui um método chamado `em.find(null, id)`, com o qual podemos chamar uma única entidade pelo id. O método `find()` recebe dois parâmetros, o primeiro é: quem é a entidade? Isto é, considerando as muitas tabelas que existem no banco de dados, diremos ao `EntityManager` qual delas desejamos buscar. Sendo assim, passamos a classe, `em.find(Produto.class, id)`, e, a partir da entidade `Produto`, será fornecido o mapeamento.

O segundo parâmetro é o id, ou seja, qual é chave primária, o identificador, da entidade que queremos buscar? Logo, passaremos o parâmetro `id`. Também incluiremos `return`, ou seja, `return em.find(Produto.class, id);`, para dizer que o `find` retorna o objeto do tipo da classe que estamos passando, que, no nosso caso, é `Produto`.

Agora, vamos ao `CadastroDeProduto.java`, nós instanciamos e não precisaremos mais da variável `Long id = 1l;`, podemos apagá-la. Queremos carregar `Produto p = produtoDao.buscarPorId(1l)`. Também podemos inserir um `System.out.println(p.getPreco());`.

```
Produto p = produtoDao.buscarPorId(1l);
System.out.println(p.getPreco());
```

COPIAR CÓDIGO

O produto que cadastramos foi o celular `"Xiaomi Redmi"`, e o preço dele é `"800"`, então, se buscarmos pelo id, precisa vir 800. Vamos rodar (apertando o botão direito e, em seguida, "Run As > 1 Java Application") e observar no Console o que acontecerá. Ele carregou e gerou um *select* com alguns *alias* nas colunas, fez um *join* onde era necessário e buscou pelo id corretamente.

Portanto, ele fez a consulta, devolveu o objeto e imprimiu 800. Essa é uma maneira de fazer uma busca pelo id na qual usamos um método `em.find()` do `EntityManager` que serve para isso. Porém, pode acontecer de desejarmos carregar várias entidades, neste caso, o `find()` não poderá ajudar. Sendo assim, vamos criar mais um método na nossa classe `ProdutoDao.java`.

Agora, teremos um método `List<Produto>` do `java.util` e, em seguida, indicaremos `buscarTodos()`. Nós não queremos mais buscar pelo id e, sim, carregar todos os produtos que estão no banco de dados.

```
public List<Produto> buscarTodos() {

}
```

COPIAR CÓDIGO

A principal maneira de fazer uma busca é utilizar uma linguagem chamada JPQL (Java Persistence Query Language) que é parecida, mas não é SQL. Funciona como se fosse uma SQL orientada a objetos. Assim, com a JPQL, é possível montar a *query* do jeito que quisermos.

Para isso, faremos o método `return em.createQuery()`, passaremos uma String para ele (a JPQL nada mais é do que uma String) e criaremos outra separada: `String jpql = ""`. Nela, montaremos o *select* parecido com o SQL: `String jpql = "SELECT"`.

No SQL, teríamos: `"SELECT * FROM"` e o nome da tabela que é `produtos`, então, `"SELECT * FROM produtos"`. Diferentemente disso, no JPQL, não passaremos a tabela e, sim, o nome da entidade, isto é, `Produto`. Recordando o problema do JDBC referente ao acoplamento com o banco de dados, perceberemos que, tendo o nome da tabela e precisando alterá-la, teríamos que mudar na entidade e em todas as classes DAO do projeto.

Na JPQL no lugar do asterisco, nós podemos utilizar um *alias*: `"SELECT p FROM Produto AS p"` (sendo que o `AS` é opcional), assim dizemos para que o próprio objeto `p` seja carregado, a entidade com todos os atributos. No `em.createQuery(jpql)`,

passamos o `jpql` e, na sequência, um `.getResultList()`. O `em.createQuery(jpql)` não dispara a *query* no banco de dados, apenas monta a *query*, para disparar de fato, chamamos o `getResultList()`.

```
    public List<Produto> buscarTodos() {
        String jpql = "SELECT p FROM Produto p";
            return em.createQuery(jpql).getResultList();
    }

}
```

COPIAR CÓDIGO

Está dando *warning* por causa da tipagem. Ele não sabe que a `em.createQuery(jpql)` retorna um objeto - uma lista - de `Produto`. Mas, para acabar com esse erro, podemos fazer o `createQuery()` em outra versão, na qual passamos o `jpql` e o tipo da classe - da entidade - que será devolvida nessa *query*. Basta passar, `jpql, Produto.class`.

```
    public List<Produto> buscarTodos() {
        String jpql = "SELECT p FROM Produto p";
            return em.createQuery(jpql, Produto.class).getResultList();
    }

}
```

COPIAR CÓDIGO

Agora, ele infere que o `getResultList()` devolverá um `List` de `Produto` e para de dar *warning*. Retornando ao `CadastroDeProduto.java`, vamos chamar o método `produtoDao.buscarTodos()`, selecionar o comando "Ctrl + 1", pedir para que `buscarTodos` seja jogado em uma variável local, chamar a variável de `todos`, fazer um `forEach()` e, dado o produto, faremos um `System.out.println()` no `p.getNome();`.

```
Produto p = produtoDao.buscarPorId(1l);
System.out.println(p.getPreco());

List<Produto> todos = produtoDao.buscarTodos();
todos.forEach(p2 -> System.out.println(p2.getNome()))
```

COPIAR CÓDIGO

Vamos rodar ("Run As > 1 Java Application") e, analisando o Console, perceberemos que ele fez o *select* do id, imprimiu "800", na sequência fez o nosso *select* e imprimiu o nome "Xiaomi Redmi". Perceberemos também que ele converte o JPQL em um SQL. A consulta que fizemos foi simplificada, mas é possível utilizar *join* e simplificar ainda mais, com SQL puro seria mais complicado.

Então, nesta aula aprendemos a fazer consultas com a JPA. Existe a busca pelo id, `buscarPorId()`, onde usamos `em.find()` e indicamos qual é a entidade e qual é o id.

E existe a busca por JPQL, com a linguagem `SELECT p FROM Produto p"`, onde usamos `em.createQuery()` e indicamos onde está a *quey*, qual a classe da entidade que essa *query* vai devolver e utilizamos `getResultList()` para ele de fato carregar essa *query*, convertê-la para um SQL e disparar carregando as entidades e montando os objetos (sem a necessidade de fazer isso manualmente) com *ResultSet*, conforme era no JDBC.

Espero que tenham gostado. Na próxima aula continuaremos discutindo um pouco mais sobre consultas. Vejo vocês lá!!

Transcrição

Agora que já aprendemos a fazer consultas com a JPA, vamos aprofundar um pouco os conhecimentos das aulas anteriores estudando outros recursos de consultas. Nós havíamos criado o método `buscarTodos()` para carregar todos os produtos, que é um *select* sem filtro, isto é, carrega todos os registros do banco de dados. Mas, eventualmente, podemos querer limitar, criar um filtro com algum parâmetro.

Então, vamos criar um novo método chamado `buscarPorNome()`. Supondo que queremos buscar determinados produtos que tenham um determinado nome, nós receberemos, como parâmetro, qual é esse nome.

```
public List<Produto> buscarPorNome(String nome) {
        String jpql = "SELECT p FROM Produto p";
        return em.createQuery(jpql, Produto.class).getResultList();

    }
```

COPIAR CÓDIGO

Agora, nossa *query* será um pouco diferente, não faremos mais `"SELECT p FROM Produto p"`, porque queremos filtrar. Para isso - semelhante ao SQL - adicionaremos, depois do `FROM Produto p`, `WHERE p.` e o nome do atributo, não da coluna na tabela, que, no caso, é `nome` (e, por coincidência, o mesmo nome da coluna).

Portanto, temos `"SELECT p FROM Produto p WHERE p.nome ="`, e precisamos passar o parâmetro que está chegando no método para essa parte. Uma maneira de fazer isso é adicionando dois pontos, para dizer à JPQL que passaremos um parâmetro

dinâmico na *query*, e, depois, dando um apelido para esse parâmetro. No nosso caso, chamaremos de `nome`, mas poderia ser qualquer outro nome.

```
public List<Produto> buscarPorNome(String nome) {
        String jpql = "SELECT p FROM Produto p WHERE p.nome = :nome";
        return em.createQuery(jpql, Produto.class).getResultList();
```

COPIAR CÓDIGO

Antes de disparar essa *query*, temos que substituir o parâmetro que está em: `"SELECT p FROM Produto p WHERE p.nome = :nome"`, pelo que veio em: `buscarPorNome(String nome)`. Para isso, antes de chamar o `.getResultList()`, podemos chamar o método `.setParameter()`. Nele, informaremos qual o nome do parâmetro que, no caso, é `"nome"`, e qual é o valor que queremos substituir, que é o `nome` que está chegando em `buscarPorNome(String nome)`.

Portanto, nós substituiremos o parâmetro do método no parâmetro chamado `nome` que está na nossa *query*.

```
public List<Produto> buscarPorNome(String nome) {
        String jpql = "SELECT p FROM Produto p WHERE p.nome = :nome";
        return em.createQuery(jpql, Produto.class)
                .setParameter("nome", nome)
                        .getResultList();
```

COPIAR CÓDIGO

Um detalhe importante é que em `setParameter("nome", nome)` nós não usamos dois pontos, como fizemos anteriormente no JPQL. Poderíamos ter quantos parâmetros quiséssemos. Por exemplo, para filtrar por outro parâmetro, poderíamos fazer `AND p.categoria =` e passaríamos o valor. Enfim, poderíamos ter vários parâmetros e usar `AND` ou `OR`, semelhante ao SQL.

Terminado, poderemos fazer uma *query* filtrando por algum atributo. Em `cadastroDeProduto.java`, no método `main()`, ao invés de `buscarTodos()`, vamos usar o `buscarPorNome()`, passar o nome que escolhemos, `"XIAOMI Redmi"`, e conferir se ele

fará a busca. Vamos rodar e olhar o Console. Analisando o *select* no Console, encontraremos "Where produto0_.nome=?", significa que ele filtrou corretamente pelo nome.

Essa maneira de passar o parâmetro é chamada de *Named parameter*, onde passamos o parâmetro pelo nome. Mas há outra forma de fazer que é passando `?1`, isto é, `"SELECT p FROM Produto p WHERE p.nome = ?1"`. E em `.setParameter("nome", nome)`, ao invés de passar um apelido, `"nome"`, passamos o `1`, isto é, `.setParameter(1, nome)`.

Ou seja, é possível ter um parâmetro posicional, com interrogação 1, 2, 3 e assim por diante, já que podemos passar vários parâmetros, cada qual com um número distinto. As duas abordagens funcionam.

Agora, vamos retornar à nossa entidade `Produto.java` e recordar que o `Produto` tem um atributo `Categoria`. Neste caso, `Categoria` é um relacionamento, outra entidade. É possível filtrar um produto pela categoria? Pelo nome, não do produto, mas da categoria? A resposta é sim.

Na classe `ProdutoDao.java` criaremos outro método e chamá-lo de `buscarPorNomeDaCategoria()`, significa que agora filtraremos pelo nome da categoria, não mais do produto.

```
public List<Produto> buscarPorNomeDaCategoria(String nome) {
    String jpql = "SELECT p FROM Produto p WHERE p.nome = :nome";
    return em.createQuery(jpql, Produto.class)
            .setParameter("nome", nome)
            .getResultList();
}
```

COPIAR CÓDIGO

A nossa *query* mudará um pouco. Como se trata de um relacionamento, precisaremos fazer um *join*, isto é, fazer uma consulta por um *join* com a tabela de Categoria. Então, no JDBC, no SQL puro, seria por *join*. Na JPA, conseguimos encontrar uma maneira simplificada, se recordarmos que se trata de uma consulta orientada a objetos.

Portanto, basta fazer `"SELECT p FROM Produto p WHERE p.categoria.nome = :nome"`, sendo que `categoria` se refere ao relacionamento.

```
String jpql = "SELECT p FROM Produto p WHERE p.categoria.nome = :nome";
```

COPIAR CÓDIGO

A JPA entenderá que a `categoria` é um atributo da classe produto e, neste caso, um relacionamento. Então, ele quer filtrar por um atributo dentro do relacionamento, desta maneira, a JPA automaticamente gerará um *join*, isto é, ela já sabe que deve filtrar pelo relacionamento e faz o *join* automaticamente, evitando que seja necessário fazer manualmente, como seria no SQL.

O JPQL é um SQL simplificado, orientado a objetos, não ao modelo relacional. Vamos verificar se funciona indo na classe `CadastroDeProduto.java`. Nela, em lugar de `buscarPorNome()`, faremos `buscarPorNomeDaCategoria()`. Também precisamos trocar o nome do produto pelo nome da categoria, que é `"CELULARES"`.

```
List<Produto> todos = produtoDao.buscarPorNomeDaCategoria("CELULARES");
```

COPIAR CÓDIGO

Agora vamos rodar e conferir se ele encontrará os produtos. Analisando o Console, notaremos que ele fez o *select* e encontrou o "Xiaomi Redmi". Ele também gerou o *join* automaticamente: "cross join categorias", "where produto0_.categoria_id=categoria1", "and categoria1_.nome=?". Significa que ele está filtrando pelo nome da categoria, que é o relacionamento.

Anteriormente no `persistence.xml`, para o Hibernate imprimir o SQL, tivemos que colocar `property name="hibernate.show_sql" value="true"/>`. Existe outra propriedade parecida com essa, a `"hibernate.format_sql"`, que serve ele *identar* o código SQL quando for imprimir, principalmente em consultas. Portanto, podemos usar essa propriedade para que o Hibernate formate o SQL.

Em `CadastroDeProduto.java`, vamos rodar outra vez a nossa classe `main`. Observando o Console, perceberemos que agora o SQL está "quebrado". Ele faz o *select*, coloca as colunas que serão selecionadas, o *from*, o *join*, o *where*, enfim, com tudo *identado* fica mais fácil de visualizar. Portanto, podemos usar essa propriedade para facilitar a leitura dos comandos SQL quando o Hibernate for acessar o banco de dados.

No *insert* ele também quebrou a linha, dividindo em colunas, *values*, então, fica um pouco mais fácil de visualizar. Espero que tenham gostado da aula e aprendido a fazer consultas com JPQL e a filtrar os resultados com o atributo de um relacionamento.

Para isso, basta navegar, e se a categoria tivesse outro relacionamento, poderíamos continuar navegando: `p.categoria.xpto.nome`, isto é, ponto + nome do atributo, e o Hibernate gerará dois, três, cinquenta ou mais *joins* segundo o necessário para efetuar a consulta.

No próximo vídeo continuaremos discutindo mais detalhes relacionados a consulta. Vejo vocês lá!!

Transcrição

Agora nós faremos mais uma consulta, mas ela será um pouco diferente. Primeiro, vamos copiar o seguinte trecho de `ProdutoDao.java` que está filtrando pelo nome do produto.

```
public List<Produto> buscarPorNomeDaCategoria(String nome) {
        String jpql = "SELECT p FROM Produto p WHERE p.categoria.nome = :nome";
        return em.createQuery(jpql, Produto.class)
                .setParameter("nome", nome)
                .getResultList();
    }
```

COPIAR CÓDIGO

Até então, nas consultas que estávamos fazendo, nós carregávamos a entidade inteira com `SELECT p`, portanto, ele traz a entidade com todos os atributos. Mas, se quisermos apenas o preço do produto, isto é, buscar um produto por um determinado nome, não todas as informações dele. Então, qual o sentido de carregar uma entidade inteira, se queremos só um atributo? É possível filtrar um atributo devolvido ao invés da entidade inteira? Sim, e aprenderemos nesta aula.

O método deste retorno não será um `List<Produto>` e nem um `Produto`, porque queremos trazer apenas um atributo que, no caso, é o preço. Se entrarmos na classe `Produto.java`, o preço é do tipo `BigDecimal`, então na nossa classe DAO, o retorno do método é `BigDecimal`.

```
public BigDecimal buscarPorNome(String nome) {
    String jpql = "SELECT p FROM Produto p WHERE p.categoria.nome = :nome";
        return em.createQuery(jpql, Produto.class)
                    .setParameter("nome", nome)
                    .getResultList();
    }
```

COPIAR CÓDIGO

Queremos devolver apenas o `BigDecimal`, o preço do produto. Vamos também renomear o nome do método, `buscarPrecoDoProdutoComNome()` e então passamos o nome como parâmetro para buscarmos apenas o preço. Na `Query`, para filtrar, precisamos pensar também como se fosse orientação a objetos. No lugar de `SELECT p`, faremos, `SELECT p.preco`, isto é, o nome do atributo.

Com isso a JPA sabe que o nosso `SELECT` não é para trazer a entidade inteira, e sim um único atributo dessa entidade.

```
public BigDecimal buscarPorNome(String nome) {
    String jpql = "SELECT p.preco FROM Produto p WHERE p.categoria.nome = :nome";
        return em.createQuery(jpql, Produto.class)
                    .setParameter("nome", nome)
                    .getResultList();
    }
```

COPIAR CÓDIGO

O resto continua igual, `FROM Produto p WHERE p.nome = :nome`, depois fazemos `em.createQuery(jpql, Produto.class)`, nela, apenas substituiremos o `Produto.class` por `BigDecimal.class`, porque agora o retorno será esse. Em seguida, temos um pequeno problema: nós setamos tudo, mas o método que estamos chamando é o `getResultList()`, que não devolve um `BigDecimal`, e sim um `List`.

Se queremos apenas um único registro em uma *query*, não podemos chamar o `getResultList()`, nós chamaremos o método `getSingleResult()`, que serve para carregar uma única entidade ou um único registro que virá na consulta, ou seja, não é uma lista, mas, sim, um único resultado. Feito isso, a princípio está pronto o nosso método e podemos testar.

```
public BigDecimal buscarPorNome(String nome) {
    String jpql = "SELECT p.preco FROM Produto p WHERE p.categoria.nome = :nome";
        return em.createQuery(jpql, BigDecimal.class)
                    .setParameter("nome", nome)
                    .getSingleResult();
    }
```

COPIAR CÓDIGO

Vamos abrir a classe `CadastroDeProduto.java` e criar a seguinte linha.

```
BigDecimal precoDoProduto = produtoDao.buscarPrecoDoProdutoComNome()
```

COPIAR CÓDIGO

E passaremos o nome `"Xiaomi Redmi"`. Na sequência, daremos um `System.out.println()` para conferir se ele está carregando corretamente o `precoDoProduto`, que no caso deveria ser 800. Vamos rodar ("Run As > 1 Java Application")

```
BigDecimal precoDoProduto = produtoDao.buscarPrecoDoProdutoComNome("Xiaomi Redmi")
System.out.println(precoDoProduto)
```

COPIAR CÓDIGO

Analisando o Console, notaremos que ele carregou "800.00". Agora, só para garantir que esse foi o `System.out` correto, vamos concatenar `"Preco do Produto: "` com `+precoDoProduto`.

```
System.out.println("Preco do Produto: " +precoDoProduto)
```

COPIAR CÓDIGO

Vamos rodar e observar o Console, e bem ao final, encontraremos "Preco do Produto: 800.00". Ao fazer uma consulta, não precisamos, necessariamente, carregar a entidade inteira. Podemos limitar qual é o atributo para trazer a informação que queremos, evitando um *select* desnecessário com uma grande quantidade de informações.

Inclusive, o SQL que ele gerou é bem enxuto: "Select", "produto0_.preco as col_0_0_". Estes nomes o Hibernate gera na hora de montar a *query*. Provavelmente existe um algoritmo que, para cada atributo, aumenta esses números só para simplificar. Enfim, ele montou a *query* corretamente e trouxe apenas um atributo.

Então, eventualmente, se precisarmos carregar um único atributo de uma entidade, podemos montar a *query* dessa maneira e funcionará corretamente. Espero que tenham gostado do vídeo!! Abraços!!