02

Organizando o código

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 pacto 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!!