12

Configurando nosso DAO

Transcrição

Como os dados já estão chegando no Bean, precisaremos usar um `DAO` (Data Access Object) para enviar o Livro ao banco de dados. Criamos uma classe chamada `LivroDao` no pacote `br.com.casadocodigo.loja.daos`. Dentro de nossa classe, faremos uso do `EntityManager` que você já deve conhecer. Ele é o objeto responsável por gerenciar nossas entidades, mantendo a ligação delas com o banco de dados. Usamos a annotation `@PersistenceContext` em cima do EntityManager para que o JPA possa injetá-lo no nosso Dao. Nossa classe ficará assim:

```
package br.com.casadocodigo.loja.daos;

import javax.persistente.EntityManager;
import javax.persistence.PersistenceContext;

public class LivroDao {

    @PersistenceContext
    private EntityManager manager;
}
```

Se você estiver com qualquer dúvida sobre essa parte mais básica do JPA, recomendo que faça o curso da plataforma da Alura (https://cursos.alura.com.br/course/jpa?preRequirementFrom=java-ee-webapp) e que é pré-requisito. No de curso de Java EE, estudaremos recursos mais avançados do JPA.

Agora vamos criar o método `salvar()` para que nosso livro vá efetivamente para o banco de dados. Usaremos o `EntityManager` com o método `persist()` para que ele envie nosso livro para o banco. Ficando nosso método assim:

```
    public void salvar(Livro livro) {
        manager.persist(livro);
    }
```

Precisaremos informar ao JPA que nossa entidade Livro está vinculada a uma tabela do banco de dados, e para isso usamos a anotação `@Entity` do pacote `javax.persistence.Entity`. Além disso, precisamos informar quem é o `Id` que será ligado à tabela e sua forma de incremento, para que nosso banco crie automaticamente a chave primaria e sua estratégia de incremento.

Usaremos o MySQL para este curso por ser um banco simples, fácil de usar e bem aceito no mercado.

```
@Entity

public class Livro {

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

    private String titulo;
    @Lob
```

```
    private String descricao;
    private BigDecimal preco;
    private Integer numeroPaginas;

    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
```

Observe que adicionamos o `@GeneratedValue`, em que usamos o `strategy`. A chave-primária será criada automaticamente pelo MySQL.

Cada um dos atributos da classe Livro, se tornará uma coluna em nossa tabela automaticamente. Para finalizar esta classe, vamos anotar o atributo `descricao` com @Lob para informar que ele pode receber um grande valor de texto.

E então, precisamos de mais alguma configuração?

Sempre que usamos JPA, precisaremos do arquivo padrão de configuração, o `persistence.xml`. Esse arquivo deve ser criado na pasta `src/java/resources/META-INF/`, que é a pasta *default* que a especificação procura o `persistence.xml`. Assim, a configuração básica do *persistence* ficará assim:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="2.1" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="casadocodigo-dev" transaction-type="JTA">
        <description>Dev persistence unit</description>
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <!-- java transaction api || JNDI -->
        <jta-data-source>java:jboss/datasources/casadocodigoDS</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect'
        </properties>
    </persistence-unit>
</persistence>
```

Dentro do arquivo, não temos muita novidade. Destacando as principais tags, temos `provider`, que informa ao JPA nosso provedor (ou seja, implementação da especificação) que vamos usar. No caso o Hibernate que já é disponibilizado pelo *Widfly*.

As tags padrão de propriedades, que seguem o formato *chava / valor*, ficando

- `hibernate.hbm2ddl.auto` onde pedimos para o Hibernate criar e manter nosso banco atualizado de acordo com as entidades, através do valor `update`;
- `hibernate.show_sql` pedimos para o Hibernate exibir o SQL que ele gera, através do valor `true`;

- `hibernate.format_sql` pedimos para o SQL exibido na configuração anterior, ser formatdo bonitinho através do valor `true` ;
- `hibernate.dialect` dizemos ao Hiberante o dialeto do banco, para ele criar *Queries* próprias para aquele banco de dados;

Outra tag importante é a `<jta-data-source />` . Um `datasource` é a fonte de dados do sistema. Essa fonte (local onde obtemos os dados) é ligada a JTA, que é mais um carinha novo que temos no curso, porém muito usado no mundo JavaEE. O JTA (Java Transaction API) é a API de transações Java, que cuida de toda a transação da nossa aplicação, onde começa e termina, e quando realizar o rollback das transações.

Toda vez que trabalhamos com banco de dados, o banco nos exige que comecemos uma transação antes de qualquer *operação que altera o estado do banco*. Se você já trabalhou com banco de dados sem cuidar de transações, certamente o seu sistema realizava commit automático. Esse não é o padrão, geralmente temos que trabalhar com a transação, e se ninguém fizer, você terá que fazer na mão. Um código de exemplo, seria:

```
public void salvar(Livro livro){
    manager.getTransaction().begin();
    manager.persist(livro);
    manager.getTransaction().commit();
}
```

Desta forma, abrimos e alteramos o banco, depois, comitamos. Lembrando que transações são úteis apenas em casos que alteram o estado do banco de dados. Mas temos a opção de deixar a cargo do JTA o gerenciamento das transações, ou seja, não precisamos mais inicializar e nem finalizar as transações.

Ainda no arquivo `persistence.xml` na tag `jta-data-source` , vemos que o valor informado começa com `java:` . Isso é uma referência ao framework de nomes do java, o **JNDI** (Java Naming and Directory Interface). Ele nos permite relacionar um nome a um recurso, desta forma podemos concluir que o valor "java:jboss/datasources/casadocodigoDS" tem um recurso relacionado a ele e este recurso é o nosso datasource.

Mas e onde estão os dados do datasource? Como usuário, senha, driver, URL e etc...? Nós informamos essas configurações no datasource que é mantido no próprio servidor. O Wildfly nos provê uma area específica de data source onde podemos colocar tudo isso.

Abra o arquivo `standalone-full.xml` e localize a tag `<datasources>` , dentro desta tag nós podemos configurar nosso datasource, já temos até um exemplo, porém esse exemplo não serve para nosso sistema. Vamos criar o nosso próprio datasource.