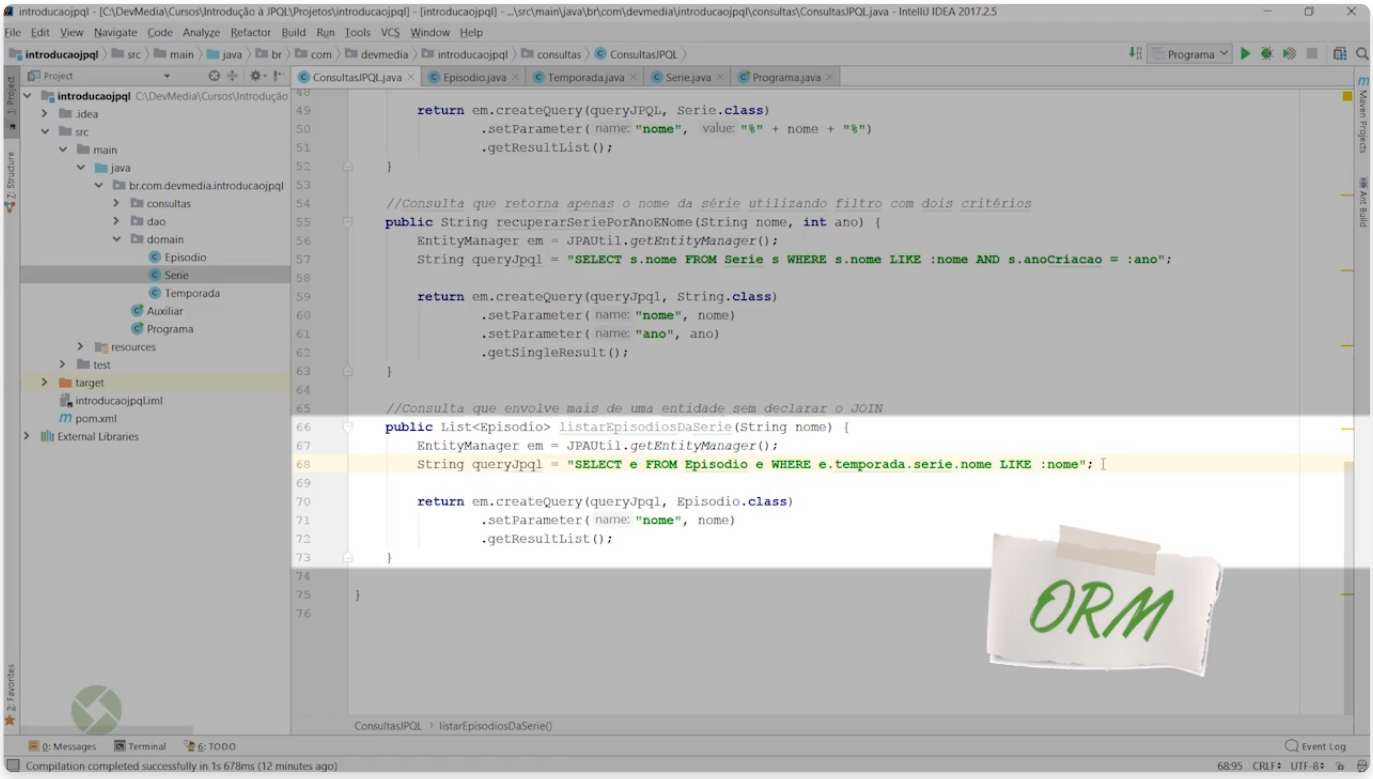
Você está em

DevMedia

6. Consultas que acessam mais de uma tabela

Tire sua dúvida | Marcar como concluído



Outra consulta bastante comum é a que precisa acessar dados em mais de uma tabela para obter o resultado. Em geral, com a linguagem SQL sempre declaramos JOINs. No caso da JPQL, no entanto, temos uma facilidade quanto a isso. Vamos conhecê-la neste vídeo.

05:18 min

Ir para o código

os dados de uma tabela se o filtro que está restringindo a busca utiliza os dados de outra tabela?

Para facilitar, imagine o seguinte cenário: O que fazer para selecionar todos os episódios da série cujo nome seja o informado por parâmetro? Note que é preciso selecionar os dados de uma tabela e utilizar, no filtro, os dados de outra.

Quando lidamos com a linguagem SQL, para atender a esse cenário, logo criaríamos uma consulta utilizando JOINs. Com a JPQL, no entanto, isso nem sempre é necessário.

Vejamos como atender ao nosso problema. Para recuperar os episódios por nome de seriado, programe o seguinte método:

```
1  public List<Episodio>
2    listarEpisodiosDaSerie(String nome) {
3
4    EntityManager em = JPAUtil.getEntityManager();
5
6    String queryJpql = "SELECT e FROM Episodio e
7      WHERE e.temporada.serie.nome LIKE :nome";
8
9    return em.createQuery(queryJpql, Episodio.class)
10     .setParameter("nome", nome)
11     .getResultList();
12 }
```

Repare, na assinatura desse método, que agora vamos retornar uma lista de episódios, e como parâmetro, passamos o nome do seriado a ser buscado.

Já na chamada ao método `createQuery()`, no segundo parâmetro, note que agora informamos a classe *Episodio.Class*, e não mais *Serie.class*. Em `setParameter()`, por sua vez, note que como esperamos que o usuário informe o nome do seriado com precisão, não especificamos o caractere coringa.

banco. Até aqui tudo bem, correto?

Agora, lembrando que desejamos selecionar todos os episódios de apenas um seriado, precisamos indicar isso de alguma forma em nosso filtro. A dificuldade, neste caso, é que não estamos manipulando a entidade `Serie`. Caso esse fosse o caso, bastaria escrever algo como: `WHERE s.nome LIKE :nome`. Neste exemplo estamos manipulando a entidade `Episodio`.

Sabendo disso, como podemos acessar o atributo `nome` da entidade `serie` a partir de um episódio? Simples! Do mesmo modo que estamos acostumados a fazer na orientação a objetos.

Assim, voltando a falar do caminho `e.temporada.serie.nome`, observe que iniciamos com a entidade `Episodio`, através do apelido `e`. No código da classe `Episodio`, note que há um atributo simples, de nome `temporada`, que é do tipo `Temporada`. Agora, na classe `Temporada`, note que temos um atributo simples, de nome `serie`, do tipo `Serie`. Por fim, acessando a classe `Serie`, temos um atributo de nome `nome`.

Entendeu agora o porquê desse caminho? Estamos apenas navegando entre os atributos que ligam as classes `Episodio`, `Temporada` e `Serie` para acessar aquilo que desejamos. Com isso, a partir de um episódio podemos acessar o nome da série da qual ele faz parte.

Quando temos esse tipo de cenário, a partir do qual podemos acessar outra entidade através de um objeto simples, não é necessário declarar JOINs na consulta. Os JOINs serão inseridos por debaixo dos panos pelo framework ORM, deixando nosso código mais simples. Então, ao escrever `WHERE e.temporada.serie.nome LIKE :nome`, apenas os episódios que pertencem à série de nome especificado por parâmetro são retornados.