02

Entendendo Lazy e Eager

Transcrição

[00:00] Olá, pessoal! Estamos de volta ao treinamento de JPA. Na aula de hoje vamos discutir um pouco sobre consultas, ainda discutir um pouco sobre consultas, mas focando na parte de performance. Como trabalhamos com a JPA, é comum acabarmos esquecendo de observar as consultas que ela está gerando.

[00:17] Nós vamos escrevendo JPQL, vamos testando a aplicação, vemos que está tudo funcionando, só que esquecemos de dar uma analisada nos comandos SQL que foram gerados, se não tem nenhum tipo de problema, nenhum tipo de consulta excessiva, algo que pode gerar algum impacto em performance nas aplicações.

[00:33] Isso acaba acontecendo muito. É muito comum encontrar aplicações que têm problemas de performance por conta da camada de persistência, por conta de *queries* mal planejadas, que geram esses gargalos. Para entender isso, o que vamos fazer? Já para adiantar e não perder muito tempo, eu criei mais uma classe de teste. Eu chamei ela de "PerformanceConsultas", bem parecida com as outras que nós já fizemos.

[00:56] Só que aqui, no meu `popularBancoDeDados` , eu já estou criando tudo. Eu já estou criando as três categorias, os três produtos, um cliente.

[01:03] Criando dois pedidos, dentro de cada pedido adicionando alguns itens. E salvando tudo isso no banco de dados.

[01:10] Então na "PerformanceConsultas", logo aqui embaixo, eu já tenho esse banco de dados todo populado com essas informações. Para começarmos a brincadeira, o que eu vou fazer? Vou fazer uma consulta simples, eu quero buscar um pedido pelo ID. Para agilizar também, eu não vou usar a classe Dao, eu vou fazer a consulta direta com o *entity manager*. `em.find()` .

[01:34] A classe é `em.find(Pedido.class, 1l);` e o ID é `1l` . Quero trazer o pedido de ID 1. Vou jogar esse código em uma variável local, vou chamar a variável de `Pedido` . Fiz uma consulta e embaixo vou dar um `system.out.println()` e vou imprimir a data do peido, `(pedido.getData())` . Vamos rodar e ver o que vai acontecer. No console, aqui são os *inset*. Tudo o que ele fez antes, mas o que interessa está aqui, a partir daqui.

[02:08] Ele fez o *select* conforme o esperado. Fez o *select*, como no caso estou fazendo um *find*, ele carrega a entidade inteira. Fez um *from* aqui e tal. E fez o *where*. Filtrou pelo ID. Porém, perceba essa parte.

[02:23] Esse é o problema que eu não gostei nesse ponto. Ele fez um *join* com a tabela de cliente. Perceba que, nessa consulta, eu estou buscando um pedido e eu quero imprimir só a data desse pedido. Então, por que essa consulta está fazendo um *join* com a tabela de cliente? Eu não estou exibindo os dados do cliente desse pedido, eu estou exibindo apenas uma informação do pedido.

[02:47] Mas, mesmo assim, não importa, a JPA vai sempre fazer um *join* com a tabela de clientes. Isso tem a ver com o mapeamento do relacionamento. Se entrarmos na classe "Pedido" e dermos uma olhada no relacionamento `@ManyToOne` , está mapeamento do relacionamento com o cliente. Perceba que é um relacionamento `@ManyToOne` .

[03:07] Na JPA, existe uma característica em relação a esses mapeamentos de relacionamento, que diz respeito a como ele deve ser tratado em uma consulta. Por padrão, na JPA, todo relacionamento que é *to one*, ou seja, *many to one* ou *one to one*, que é um outro relacionamento, de um para um, automaticamen

a JPA, ela sempre vai fazer um *select*, vai incluir um *join* para carregar esse registro sempre que você carregar a entidade principal.

[03:42] Sempre que eu fizer um *select*, fizer um *find*, na entidade "Pedido", a JPA sempre vai carregar o cliente junto. No caso, ela carregou fazendo um *join*. Então esse que é o problema. Nem sempre eu quero carregar os relacionamentos *to one*. Nessa minha entidade, até que está tranquilo, porque eu só tenho um único relacionamento *to one*.

[04:04] Mas é comum você encontrar entidades que tem 3, 5, 8, 10 relacionamentos *many to one* ou *one to one*. Para cada um desses relacionamentos, a JPA iria fazer um *join* para carregar todas essas informações, sendo que não necessariamente você precisaria de todas essas informações. Isso pode gerar um gargalo no sistema, você carregar coisas demais, coisas desnecessárias.

[04:28] Agora, a nossa entidade "Pedido", ela também tem um relacionamento, que é essa lista `List<ItemPedido>` . Porém, se analisarmos no SQL, ele não fez *join* com a tabela de item pedido.

[04:42] Isso porque o relacionamento `ItemPedido` , a cardinalidade dele, ele é um *to many*. Os relacionamentos *to many*, *one to many* ou *many to many*, ele não tem essa característica de quando você carrega o pedido, a JPA carregar junto essa lista, justamente por conta de ser uma lista. Como é uma lista, pode ser pesado, você poderia sobrecarregar o sistema com inúmeros registros carregados para a memória do computador.

[05:10] Então tudo o que *to one* é carregado automaticamente, tudo o que é *to many* não é carregado automaticamente. Em que momento ele carrega essa lista `List<ItemPedido>` então? Somente se você fizer um acesso a essa lista. Vamos simular esse acesso. Invés de darmos um `System.out.println(pedido.getData());` em "PerformanceConsultas", vou dar um `System.out.println(pedido.getItens().size());` .

[05:31] Vou imprimir o *size*, o tamanho da lista. Deu problema aqui. Na minha classe "Pedido", acho que eu esqueci de gerar os *getters* e *setters* do `itens`. Vou dar um "Ctrl + 1" e vou mandar ele gerar o *getter* e *setter*. Acabou faltando esse detalhe. Então agora eu mudei o `System.out` de "PerformanceConsultas", ao invés de imprimir a data do pedido, eu estou imprimindo quantos itens tem nesse pedido.

[05:54] Agora eu estou acessando a lista de pedidos, que é um relacionamento *to many*. Vamos rodar o código e ver o que vai mudar aqui. Perceba que agora mudou um pouco, ele fez aqui em cima, aqui acabaram os *inserts* do "populaBancoDeDados". Ele fez o "select" do pedido.

[06:15] Fez o "join" com o cliente, é um relacionamento *to one*. E, na sequência, fez um outro "select", que foi quando eu acessei aquela lista de itens.

[06:24] Ele fez um outro "select" para carregar os itens. Só que na entidade "ItemPedido" - vamos dar uma olhada nela. Tem um *many to one* para o `pedido` e um *many to one* para o `produto`.

[06:36] Ele vai fazer um *join* com "Pedido" e vai fazer um *join* com "Produto". Foi isso o que ele fez no console. Ele fez o "select", fez um "join" com produto.

[06:46] E fez também um "join" com a categoria, porque o "Produto", se abrirmos a classe "Produto", tem um *to one* com a Categoria. Percebe o efeito cascata? Tudo o que for *to one*, ele carrega. Carregou uma entidade, acessou uma lista dela, essa entidade tem relacionamentos *to one*? Ele carrega os *to one*.

[07:05] Esses *to ones* têm outros relacionamentos *to ones*? Ele carrega os *to ones*, e vai um milhão de informações sendo carregadas pela memória, sendo que às vezes você nem precisava carregar tudo isso para a memória. Esse é o comportamento padrão da JPA, ela tem uma estratégia de carregamento dos relacionamentos. Essa estratégia, ela tem dois possíveis comportamentos, chamados de *eager* ou *lazy*.

[07:32] Todo relacionamento *to one*, o padrão é ele ser *eager*, ele faz o carregamento antecipado. Então você carregou o pedido antecipadamente, mesmo que você não acesse nada do cliente, que é um relacionamento *to one*, a JPA vai carregar esse relacionamento. Já os relacionamentos *to many*, por padrão, o comportamento é chamado de *lazy*, que é o carregamento preguiçoso, o carregamento tardio.

[07:56] Quando você carrega um pedido, ele não carrega essa lista de pedidos, ele só carrega essa lista se você acessar alguma informação desta lista. Eu acessei o método `.size`, por exemplo. Não tem como, ela terá que saber qual é o tamanho desta lista, terá que disparar um *select* para carregar.

[08:11] Então existem essas duas estratégias de carregamento, o carregamento *eager*, que carrega junto com a entidade, por mais que você não utilize aquele relacionamento, e o carregamento *lazy*, que só carrega se for feito o acesso. Por padrão, todo relacionamento *to one* é *eager*, é carregado junto automaticamente com a entidade. E os relacionamentos *to many* são *lazy*, só são carregados se você fizer o acesso.

[08:36] Isso pode gerar problemas de performance na sua aplicação. Uma mudança, que vamos fazer agora, que é uma boa prática, é: todo relacionamento *to one*, a boa prática é você abrir um parêntese no `@ManyToOne` e existe um parâmetro chamado `(fetch = )`, que é como você controla esse carregamento. Perceba, ele tem dois valores, "EAGER" e "LAZY".

[08:59] A boa prática é: todo relacionamento *to one*, coloque o carregamento para ser *lazy*, `(fetch = FetchType.LAZY)`, porque por padrão, ele é *eager*. Se você carregou o pedido, não importa, ele sempre vai carregar o cliente, se ele for um relacionamento *to one*, então mudamos para *lazy*.

[09:15] No `List<ItemPedido>`, é um *to many*, já é *lazy* o padrão, você não precisa mudar. Então vamos alterar os relacionamentos *to one* para serem *lazy*, só vou carregar se eu fizer o acesso. Na entidade "Cliente" não tem nenhum

relacionamento, beleza. Na entidade "Produto" tem a `Categoria`, vou colocar `@ManyToOne(Fetch = FetchType.Lazy)`.

[09:37] Na "Categoria" não tem nenhum relacionamento. "Cliente" ok. "Produto" ok. "ItemPedido", na "ItemPedido", está relacionado com o `Pedido` e está relacionado com `Produto`, carregamento *lazy*, eu só vou carregar se eu fizer o acesso. Isso é uma boa prática. Vamos rodar aquele código novamente, aquele *main* do "PerformanceConsultas". Se eu rodar aqui, o que será que vai mudar? Vamos dar uma analisada.

[10:06] Aqui acabou os "inserts". Ele fez o "select" porque eu chamei o *find*. Mas perceba, ele não fez mais o *join* com o cliente, ele só carregou as informações do pedido. Um *join* a menos, menos dados que eu trago do banco de dados, mais rápida será essa consulta, menos dados trafegados pela rede.

[10:25] Na sequência, como eu acessei a lista, não teve jeito, ele carregou a lista. Só que perceba, na lista, ele só carregou, só fez o "select" do item pedido, ele não fez mais os *joins* com o produto e nem com o pedido.

[10:38] Porque colocamos o carregamento como *lazy*. Como eu não acessei, do item eu não acessei o produto e do item eu não acessei o pedido, ele não fez mais nenhum *select* aqui para baixo. Então esse é o padrão recomendado, relacionamentos *lazy*, a regra é essa, tudo tem que ser *lazy*.

[10:57] Relacionamentos *to many* já são *lazy*, você não precisa colocar, agora, os *to one*, são *eager* por padrão, então você tem que trocar para *lazy*. Essa é uma boa prática, porém pode gerar um efeito colateral. A partir do momento que nós trocamos o relacionamento para ser *lazy*, podemos ter algum impacto na nossa aplicação. No próximo vídeo nós discutiremos melhor que impacto é esse. Vejo vocês lá.