03

Consultas com Join Fetch

Transcrição

[00:00] Olá, pessoal! Agora que já colocamos todos os relacionamentos como *lazy*, os relacionamentos *to one*, nós já percebemos que, nas consultas, acabou ficando mais rápido, ficaram menos informações sendo carregadas, porque a JPA vai evitar fazer os *joins*. Sempre que você carregar, por exemplo, o "Pedido", ele não vai carregar o "Cliente".

[00:19] Isso resolve esse problema, deixa mais performática a aplicação e evita gargalos, porém isso pode gerar um efeito colateral. No vídeo de hoje vamos discutir justamente sobre isso. Para mostrar esse problema, vamos pegar aquele nosso *main* da "PerformanceConsultas" e eu vou alterar aqui. Ao invés de imprimir os itens do pedido, eu vou imprimir o nome do cliente, `System.out.println(pedido.getCliente(). getNome());` .

[00:44] Se eu rodar desse jeito, vamos rodar aqui, "Run As > Java Application". Vamos ver o resultado. Perceba, ele fez o "select" do pedido e não carregou o cliente

[00:56] Mas, na linha seguinte, como eu mandei ele imprime o nome do cliente, ele precisou disparar o "select" para buscar esse cliente. Então funcionou corretamente, trouxe aqui o nome do cliente certo.

[01:06] Porém, nem sempre vai ser esse resultado. Quando eu acessar a informação, a JPA vai e dispara o "select" para carregar as informações que estavam faltando, que estavam como *lazy*, porque pode acontecer de, nesse momento, quando eu for dar esse *system out*, o *entity manager* já estar fechado.

[01:28] Se simularmos dessa maneira, `em.close();` . Teremos um problema, porque quando chegar nessa linha do `System.out` , a JPA vai detectar `pedido.getCliente()` - opa, é um relacionamento *lazy*, eu não carreguei ainda, vou disparar o *select* para carregar do banco de dados.

[01:46] Mas o *entity manager* - lembra que o *entity manager* é a nossa ponte com o banco de dados - ele já foi fechado. Então a JPA não vai conseguir disparar esse *select* para carregar essa informação. Vamos rodar aqui e ver o que vai acontecer. Rodei. Vou maximizar aqui o console. Olha lá.

[02:02] Ele fez o "select" do pedido, fechou o *entity manager*. Quando eu tentei acessar o nome do cliente, *exception*.

[02:09] É uma *exception* bem famosa do *hibernate*, que é a tal da "LazyInitializationException". Ele fala aqui: “olha, eu consegui inicializar um *proxy*, eu não conseguir carregar a entidade cliente de ID 1, porque não havia sessão, o *entity manager* já estava fechado”. Esse é o efeito colateral que eu havia comentado.

[02:31] Talvez vocês estejam pensando, “Rodrigo, é só não fechar o *entity manager*, já que você vai fazer um novo *select* aqui embaixo, não feche o *entity manager*”. Porém, nem sempre isso será possível. Aqui nós conseguimos fazer isso porque, de novo, estamos testando com o método *main*, temos total controle do *entity manager*.

[02:50] Se eu sei que vou precisar carregar uma informação, eu não fecho o *entity manager*. Porém, em uma aplicação real, uma aplicação *web* em uma API, pode acontecer de, no momento de você acessar uma determinada informação, o *entity manager* já estar fechado, porque é muito comum nas aplicações *web*, o *entity manager*, o escopo dele, o tempo de vida dele, durar apenas pouco tempo, durar apenas a chamada de um método.

[03:16] Por exemplo, a classe Dao. Eu iniciei a chamada de um método na classe Dao, o *entity manager* é criado. Saí do método, fiz o *return*, o *entity*

manager é fechado.

[03:27] Se dali para frente você, em qualquer ponto da aplicação, no seu *controller*, no seu *service*, em qualquer outro lugar, na sua página, na exibição da *view* da sua aplicação, se você tentar acessar uma informação que é *lazy*, que não havia sido carregada antes, pode ser que o *entity manager* já esteja fechado e você tome essa *exception*, que é o "LazyInitializationException".

[03:46] Em projetos reais, é muito comum isso acontecer, você tomar essa *exception*, porque o *entity manager* já foi fechado. Às vezes você nem tem o controle, é o próprio servidor de aplicação, é o próprio *frameworK* que está gerenciando o *entity manager*, não foi você que instanciou e que fez o *close*, então você nem sabe onde foi fechado.

[04:06] Como você resolve essa situação? Em um primeiro momento, o que você iria fazer? Iria tirar o `(fetch = FetchType.LAZY)` de `Cliente` de "Pedido", ia voltar para *eager*. Porém, isso é ruim, porque sempre que você carregar o pedido, você vai trazer o cliente junto, e não necessariamente sempre que você carregar o pedido você quer trazer as informações do cliente.

[04:24] A solução é você usar uma ideia que é chamada de *query* planejada. A ideia é: se em um determinado ponto da aplicação você não precisa imprimir nada, não precisa trazer nenhuma informação do cliente ou do relacionamento que é *lazy*, por exemplo, se eu comentar essa linha do `System.out.println` - na verdade, invés de eu comentar, se eu der um `(pedido.getData());` , eu carreguei um pedido, estou acessando a data de um pedido.

[04:51] É uma informação do próprio pedido, não estou acessando informação de nenhum relacionamento. Perceba, vai funcionar normalmente e ele não vai fazer o *join*. Agora, se no seu código você precisa acessar uma informação que é *lazy*, então a ideia é: na sua consulta, já traga essa informação junto. Ou seja, já carregue o pedido com o cliente junto, porque aqui eu preciso dos dados do cliente.

[05:20] Então a solução aqui acaba sendo um pouco mais chata, não podemos mais usar o método `.find`, porque no *find* eu não consigo dizer para a JPA: JPA, nessa consulta, nesse *find*, carregue o pedido e carregue junto o cliente. Para resolver esse problema, teremos que montar uma *query*.

[05:34] Por exemplo, vamos abrir o "PedidoDao", agora eu quero trabalhar com a classe "PedidoDao" mesmo. Na classe "PedidoDao", eu tenho o meu método de buscar por ID, eu teria que criar um outro método, que faz uma consulta no pedido carregando junto o cliente. Isso é a tal da *query* planejada.

[05:54] É uma *query* onde eu planejo essa *query*, eu já penso bem no que eu preciso trazer de informação e já carrego tudo o que eu precisar, para evitar fazer outros *selects*, para evitar tomar um *lazy initialization exception* se o *entity manager* estiver fechado. Como fazemos isso? Vamos criar um novo método.

[06:12] `public`, o retorno dele é um objeto `public Pedido`. Vou chamar esse método de `public Pedido buscarPedidoComCliente()`, algo assim. Ele recebe como parâmetro o ID desse cliente, `(Long id)`. Parecido com o método de buscar por ID. Vai depender do que é a sua consulta, se terá filtro ou não.

[06:31] Imagine que eu estou buscando por ID, só que quero carregar o *cliente* junto. A ideia é fazermos o seguinte, `return em.createQuery()`. Agora não é mais um *find*, porque o *find*, ele só carrega a entidade em si, não tem como controlar os relacionamentos. `em.createQuery(" ", )` e vou passar a minha *query* aqui.

[06:54] Vou já passar a *query* direto aqui invés de criar uma *string*, e vou colocar que o retorno é um objeto do tipo `(" ", Pedido.class).getSingleResult();`, já que eu estou filtrando pelo ID, só virá um único resultado. E qual será a minha *query* aqui? Seria uma *query* tradicional, de buscar pelo ID.

[07:13] Seria algo assim, `em.createQuery("SELECT p FROM Pedido p WHERE p.id = :id", Pedido.class)`. Preciso criar um parâmetro aqui e antes do `.getSingleResult()` eu preciso *setar* esse parâmetro, `.setParameter()`. Parâmetro 1, qual é o valor - parâmetro 1 não, parâmetro `("id", )`, porque eu coloquei `:id`, então aqui é `("id", )`.

[07:40] E é o ID que foi passado como parâmetro. Essa seria a consulta. Mas essa consulta, ela é a mesma coisa que o *find*, ela não carrega os relacionamentos. Então, na consulta, para dizermos para a JPA, olha, JPA, nessa consulta, já carregue determinado relacionamento junto, porque eu vou precisar dessa informação e eu não quero tomar um *lazy initialization exception*.

[08:01] Você tem que usar uma palavra aqui. Aqui, antes do *where*, é `SELECT p FROM Pedido p JOIN FETCH`. Só que é um *join* diferente, é um *join* com a palavra *fetch*. Não é um *join* porque você quer filtrar alguma coisa do relacionamento, é um *join fetch*. Eu já estou falando: JPA, já faça o *fetch*, já carregue, já busque junto esse relacionamento. Qual?

[08:23] `SELECT p FROM Pedido p JOIN FETCH p.cliente WHERE` e o nome do atributo, que é `.cliente`. Essa palavra mágica, o *join fetch* é que faz essa mágica de carregar um relacionamento, que é *lazy*, nessa consulta, apenas nessa consulta. É como se ele virasse *eager*. Então, nessa consulta, ele virá com a entidade principal.

[08:43] Então vamos voltar para o nosso código. Em "PerformanceConsultas" agora eu preciso criar um "PedidoDao", `PedidoDao pedidoDao = new PedidoDao()`, passo o *entity manager* como parâmetro, `(em);`. E agora eu vou fazer aquela consulta. `pedidoDao.buscarPedidoComCliente(1l);`, eu passo o ID, quero trazer o pedido de ID 1.

[09:08] Fechei o *entity manager*, `em.close();`. Vou atribuir o `buscarPedidoComCliente` em uma variável, que eu esqueci, `Pedido pedido =`

`pedidoDao.buscarPedidoComCliente(1l);` . Na linha debaixo, após fechar o *entity manager*, estou acessando uma informação do cliente. Vamos ver se agora vai funcionar sem dar a exceção. Maximizei o console, olhe.

[09:27] Funcionou. Fez os "inserts" e, no mesmo "select", ele já trouxe os dados do pedido, mas como eu coloquei o *join fetch*, ele também já trouxe os dados do cliente.

[09:39] Fez o "from" e fez um "inner join" aqui. Carregou os dados corretamente. Então, mesmo que o *entity manager* esteja fechado, na minha consulta eu já trouxe essa informação, ele não vai disparar outro *select*, já veio carregado. Essa que é a ideia, para isso que serve esse tal de *join fetch*.

[09:57] Será bem comum você encontrar esse tal de *join fetch* em projetos que você for trabalhar. E ele será útil quando você tiver um relacionamento que é *lazy* e na mesma consulta, naquele mesmo momento, você já quer carregá-lo, você quer que ele seja *eager*, mas não na aplicação inteira, apenas nessa consulta em específico.

[10:14] Com isso você deixa mais organizada a sua aplicação, onde você precisa carregar um determinado relacionamento, você carrega, onde você não precisa, você não carrega. Você só traz as informações necessárias em cada tela da aplicação e evita sobrecargas, evita esses problemas de gargalos de consultas, que são muito, mas muito comuns nas aplicações, porque costumamos só fazer a funcionalidade, implementamos a funcionalidade, funcionou, está valendo.

[10:43] Já integra, já vai para a próxima demanda, e nem olhamos o que está acontecendo. Então lembre-se, sempre que usamos um *frameworK*, nós precisamos conhecer o que ele está fazendo por debaixo dos panos, senão viramos refém daquele *framework*, caímos em armadilhas, como essa. Nós geramos a *query*, mandamos, testamos na aplicação, funcionou, nem olhamos o que foi o SQL gerado pelo *hibernate*.

[11:04] Então habilite, durante o desenvolvimento, esse *log* do SQL, para ele imprimir no console. Dê uma olhada no SQL final que ele está gerando, veja se ele não está gerando um milhão de *selects*, porque isso é mega comum nas aplicações. E, se estiver gerando, siga a boa prática: todos os relacionamentos *to one*, coloque como *lazy*, e as consultas, você planeje.

[11:26] Só cuidado: se você já tem um sistema em andamento e você troca, coloca tudo como *lazy*, é provável que dê erro em um monte de lugar. Um monte de lugar começará a dar *lazy initialization*, porque tem lugares que você estará acessando os relacionamentos que estão *lazy* com o *entity manager* já fechado.

[11:42] Então cuidado: quando trocar para *lazy*, certifique-se de que os lugares que estão fazendo essa consulta e, se for o caso, você terá que ir na sua classe Dao, na sua classe *repository* da vida e criar novos métodos para carregar as informações necessárias. É mais chato, porque tem que criar mais um método, mas você evita problemas de gargalos, então isso é uma boa prática. Eu vejo vocês no próximo vídeo, um abraço.