03

JDBC e seus problemas

Transcrição

Para começar o nosso treinamento de JPA, vamos discutir um pouco da motivação, isto é, para que a JPA foi criada - para resolver quais problemas. Além disso, discutiremos sobre o *hibernate* e outras implementações.

Neste vídeo, também trataremos do JDBC, que é a tecnologia padrão do Java para acessar o banco de dados relacionais. Quem aprendeu a programar em Java, provavelmente, quando for desenvolver sistemas, precisará fazer o acesso ao banco de dados.

Para fazer o acesso do Java com um banco de dados, por exemplo, uma SQL, Oracle, ou qualquer outro banco de dados, a tecnologia padrão utilizada é o JDBC. O Java nasceu em 1995, em 1997 veio o JDBC.

Antes do JDBC, se quiséssemos acessar um banco de dados em Java, era necessário aprender tecnologias complexas de *socket* e fazer tudo manualmente: abrir uma conexão com banco de dados e fazer toda comunicação utilizando o protocolo específico daquele banco de dados. Era muito trabalhoso e complicado.

O JDBC veio para facilitar esse processo. Ele nada mais é do que uma especificação para fazer acesso a bancos de dados relacionais no Java. Portanto, se trata de uma camada de abstração para acessar, do código Java, o banco de dados, independente de qual seja o protocolo. Em outras palavras, o JDBC veio como uma camada para simplificar o acesso e facilitar fazer trocas de bancos de dados.

A partir disso, não é mais necessário conhecer o protocolo MySQL, do Oracle, saber os detalhes técnicos, nem ficar abrindo o *socket* e fazendo uma comunicação manual com o banco de dados, basta utilizar o JDBC.

Quem já estudou o JDBC (na Alura temos treinamento de JDBC), sabe que precisamos ter um driver. Esse driver é um JAR, um arquivo com as classes do banco de dados. Ou seja, ele é a implementação do banco de dados em cima da JDBC.

Então, para acessar o MySQL, é necessário baixar o driver do MySQL. Para trocar o banco de dados - usar o PostgreSQL, por exemplo - trocamos o JAR baixando o driver do PostgreSQL, que é outro JAR. Ambos estão implementando o JDBC, de maneira que, no código, o impacto é mínimo.

Ao mudar de um banco de dados para outro, temos poucas mudanças no código. Trocamos basicamente as configurações, mas, a grande parte do código que está fazendo a comunicação com o banco de dados continua igual. Isso facilita muito por não nos prendermos a um só fornecedor, a um banco de dados.

Para não ficar com o código do JDBC espalhado em vários pontos da aplicação, um padrão de projeto bastante utilizado é o "*DAO*", Data Access Object. Com ele, é possível isolar toda a API do JDBC dentro de uma única classe - de uma única camada - para não ficar com os códigos de *Connection*, *ResultSet*, que são classes complicadas do JDBC, espalhadas pela aplicação.

Basicamente, temos alguma classe na aplicação, um **Controller**, uma **Service** ou algo do gênero. Nesta classe está contida a **lógica de negócios**, e, nessa lógica, precisamos acessar o banco de dados. Ou seja, não instanciamos, não chamamos as classes do JDBC, e, sim, uma classe **DAO**. É na classe DAO que está isolado - abstraído, encapsulado - o código do JDBC, é ela também que faz a ponte com o **banco de dados**. Então, existia uma divisão de responsabilidades na aplicação.

Olhando de fora da classe DAO, existiria algo aproximadamente assim:

```
public class CadastroProdutoService {

    private ProdutoDao dao;

        public CadastroProdutoService(ProdutoDao dao) {
            this.dao = dao;
        }

        public void cadastrarProduto(Produto produto) {
            // regras de negocio...

                this.dao.cadastrar(produto);
        }
```

COPIAR CÓDIGO

Imagine que temos um `CadastroProdutoService`. Precisaríamos de uma classe DAO e de um método, como `CadastrarProduto()`. Depois receberíamos o objeto `produto`. Teríamos também as regras de negócio, validação, cálculos, e então chamaríamos `dao.cadastrar`.

Olhando de fora, não dá para saber como a classe DAO está funcionando, se ela está usando JDBC, se a persistência é em banco de dados, se é em arquivo, em memória, em um serviço externo, não sabemos se é MySQL, Oracle.

O código está bem fácil de usar: chamamos a DAO, depois o método `cadastrar()`, passamos o objeto `produto`, e pronto, não ficamos presos a como foi implementado o método `cadastrar()`. Assim, não temos acesso à API do JDBC, ela fica bem isolada. Por fora, o código fica bem bonito, mas, por dentro da classe DAO, temos um problema.

Em classes DAO, usando JDBC, acabamos tendo um código bem complicado, porque precisamos usar a API do JDBC que é uma API bastante antiga do Java (foi criada em 1997), com bastante verbosidade, e fez com que as pessoas

desenvolvessem certa aversão ao Java, pela impressão de que nele é tudo burocrático, complexo.

Então, é necessário lidar com classes, como `PreparedStatement`, `connection`, `ResultSet`. Também é necessário fazer `tryCatch()`, porque elas lançam *checked exception*. Além disso, precisamos montar uma SQL manualmente, usar o *PreparedStatement* para não ter o problema do *SQL Injection*. Tudo isso deixa o código um pouco complicado.

Esse tipo de código JDBC, embora funcione, apresenta algumas desvantagens que fizeram com que as pessoas pensassem em outras alternativas mais simples. O JDBC tem dois grandes problemas que motivaram o surgimento de tecnologias como o *Hibernate* e a JPA. O primeiro problema é o **Código muito verboso**. Por exemplo, para salvar um produto no banco de dados, precisamos de cerca de 30 linhas de código.

Em um código tão grande, é difícil e demorado fazer manutenção. Às vezes, é necessário montar uma *Query* nativa do banco de dados, e o código vai ficando cada vez mais difícil de entender e de fazer manutenção. Esse é um grande problema, e nem é o pior.

O segundo problema é o **Alto acoplamento com o banco de dados**. Quando trabalhamos com o JDBC, temos um acoplamento muito forte com o banco de dados. Significa que, qualquer mudança no banco de dados gera um impacto muito forte na aplicação. Por exemplo, se trocamos o nome da tabela, de alguma coluna ou qualquer outro detalhe desses, acabamos impactando a aplicação e teremos que mexer na classe DAO.

O problema é que, pode acontecer de, por exemplo, ao renomearmos a tabela de produto, pode ser que - além da classe `ProdutoDao` - existam outras classes DAO que façam JOIN com a tabela produto. Ou seja, precisaremos procurar todos os lugares que estão referenciando tabela produto e fazer esse *rename*. Isso nos leva a um alto acomplamento com banco de dados. Qualquer mudar de um lado, gera impacto grande do outro.

Esses são os dois principais problemas que as pessoas começaram a perceber no JDBC e, a por conta deles, aprimoraram ideias de como reduzir o impacto que esses dois problemas geram. Foi desse processo que surgiu a JPA. No próximo vídeo, discutiremos com calma como foi a criação da JPA e como ela resolveu os problemas do JDBC. Vejo vocês lá!! Abraços!!