02

Gerenciando conexões com Pool de conexão

Transcrição

Um EntityManager por requisição?

No capítulo anterior, aprendemos que um novo `EntityManager` é aberto a cada método que chamamos (ou a cada transação) . Mas, em nosso caso essa não era a melhor solução, já que para renderizar o JSP na tela precisávamos que ele permanecesse ativo. Aprendemos a configurar o Spring para que ele crie um `EntityManager` no início da requisição e segurando-o até o fim, conseguimos renderizar o JSP e consultar um relacionamento *lazy*.

Muitas ou poucas conexões?

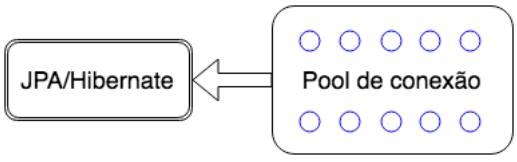
Um dos gargálos de escalabilidade mais comum enfrentado pelas aplicações se relaciona com o modo que elas tratam as conexões com o banco de dados. Abrir o tempo todo conexões com o banco de dados envolve abrir vários *sockets* para trafegar dados entre aplicação e o banco de dados.

No último capítulo configuramos a JPA para deixar um `EntityManager` aberto até o fim da requisição, isso significa que ela vai abrir uma conexão a cada requisição que é feita ao servidor. De fato, precisamos melhorar a forma como essas conexões são obtidas.

Podemos tentar resolver esse problema criando uma única conexão para ser compartilhada entre todos os clientes. De fato, essa opção é muito útil quando trabalhamos em uma aplicação *two-tier*, por exemplo, uma aplicação Desktop conversando com um banco de dados. Quando temos uma aplicação Web com uma única conexão, um segundo cliente deverá aguardar o primeiro terminar de usá-la para prosseguir com seu processamento. Dessa maneira, essa estratégia comprometerá a escalabilidade da aplicação e certamente não é o que queremos.

Gerenciando conexões

O que precisamos é encontrar um meio-termo entre: abrir uma conexão individual para cada cliente e apenas uma conexão para todos os clientes. Assim, o ideal seria um número *fixo* de conexões abertas em algum lugar para que sejam compartilhadas e reaproveitados por todos os clientes. A esse lugar daremos o nome de **pool**.



O *Hibernate* já possui um *pool* nativo de conexões com o banco, que **não deve ser usado em produção**. Existem diversas implementações no mercado além de servidores de Aplicação (JBoss WildFly) e Servlet Containers (Apache Tomcat) que possuem implementações sofisticadas de pool de conexões. No nosso caso, vamos utilizar o **C3P0** que é [indicado pela própria documentação do *Hibernate*][1].

##Abstração do Pool: DataSource

Assim como no Java EE, o Spring possui um recurso que permite isolarmos as informações de acesso ao banco de dados (usuário, senha, local do banco e classe do driver) para que, a partir dele, sejam criadas as conexões. A esse recurso,

chamamos de **DataSource**. Na nossa aplicação já temos configurado um *Datasource* padrão. Vamos dar uma olhada na classe `JpaConfigurador`:

```
@Bean
public DataSource getDataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();

    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost/projeto_jpa");
    dataSource.setUsername("root");
    dataSource.setPassword("");

    return dataSource;
}
```

Em nosso caso, como queremos usar um *pool*, precisaremos de um *DataSource* especial que permita configurarmos informações relevantes ao tamanho do *pool* e a quantidade de conexões. O que iremos fazer é trocar para um *Datasource* do *C3P0* que se chama**ComboPooledDataSource**:

```
@Bean
public DataSource getDataSource() throws PropertyVetoException {
    ComboPooledDataSource dataSource = new ComboPooledDataSource();

}
```