Considere usar uma ferramenta de mapeamento objeto relacional

Entre os frameworks mais utilizados pelo mercado, o Struts 1.x era o mais requerido pelas vagas de emprego até o início do ano de 2009 [2], provavelmente por causa da sua adoção em larga escala desde o seu início, quando era praticamente a única alternativa ao uso de Servlets e JSP puros. Muitos reclamam das limitações do Struts mas, mesmo assim, ele continua forte dada a quantidade de sistema legados existentes e mesmo novos projetos que utilizam as "arquiteturas de referência" já homologadas.

No segundo lugar da procura pelo mercado, a disputa era intensa entre o Hibernate e o Spring e, hoje em dia, ambos são mais requisitados que o Struts. Não é a toa a popularidade do Hibernate: consultas SQL, gerenciamento de transações, caches e a decisão de quando persistir e buscar dados do banco são preocupações frequentes em grande parte dos projetos e demandam muito tempo e trabalho. Soluções que visam diminuir a necessidade do uso do JDBC apareceram a um longo tempo e ganham cada vez mais força.

O que há de errado com o JDBC?

Não há nada de errado com o JDBC. É uma das APIs mais antigas do Java, que entrou no JDK 1.1 e foi sempre sendo atualizada. A versão atual, a 4.0, traz melhorias significativas e oferece muito poder até mesmo a bancos de dados específicos [47].

O código para executar uma simples transação com JDBC puro pode ser escrito de maneira bem sucinta:

```
public void insere(String email) throws SQLException {
    String sql = "INSERT INTO Destinatario (email) VALUES ('" + email + "')";
    Connection con = abreConexao();
    con.setAutoCommit(false);
    con.createStatement().execute(sql);
    con.commit();
    con.close();
}
```

Pode não estar óbvio, mas este tipo de código ingênuo é a causa dos principais problemas ao acessar bancos de dados com Java: vazamento de memória, transações não finalizadas corretamente, excesso de conexões abertas com o servidor pela ausência de pool, baixa escalabilidade, vazamento de conexões e até mesmo SQL injection.

```
public class DestinatarioDAO {
    private DataSource dataSource = new ComboPooledDataSource();

    public void insere (String email) throws DAOException {
        String sql = "INSERT INTO Destinatario (email) VALUES (?)";

        Connection con = null;
        PreparedStatement stmt = null;
```

```
        try {
            con = this.dataSource.getConnection();
            con.setAutoCommit(false);
            stmt = con.prepareStatement(sql);
            stmt.setString(1, email);
            stmt.execute();
            con.commit();
        } catch (SQLException e) {
            try {
                if (con != null)
                    con.rollback();
                throw new DAOException("Problemas ao inserir", e);
            } catch (SQLException e) {
                throw new DAOException("Problemas ao inserir e no rollback", e);
            }
        } finally {
            try {
                if (stmt != null)
                    stmt.close();
            } catch (SQLException e) {
                throw new DAOException("Problemas ao fechar Statement", e);
            } finally {
                try {
                    if (con != null)
                        con.close();
                } catch (SQLException e) {
                    throw new DAOException("Problemas ao fechar Connection", e);
                }
            }
        }
    }
}
```

O código mais robusto utiliza `PreparedStatement` para evitar SQL injection e tirar proveito de alguns bancos que pré compilam as queries; toma cuidado com o `try`, `catch`, `finally` para não deixar transações pendentes e conexões abertas; usa um pool de conexões através de um `DataSource` para evitar o vazamento destas e não utilizar conexões que possivelmente possam ter sido fechadas pelo servidor [52]. E o código acima ainda pode ter melhorias: o esforço em lançar exceptions com boas mensagens e não deixar vazar SQLException poderia ser maior com mais blocos *try/catch* para tratar os erros mais separadamente (como o caso de falhar a abertura da conexão). É válido reparar aqui que um container de injeção de dependências ajudaria bastante o trabalho de evitar esse código repetitivo (o *boilerplate code*).

Mesmo tomando todo esse cuidado há aqui um trabalho ainda de design: onde deve ficar esse código SQL? Dentro de um DAO? Dentro de um *properties*? Um XML? Nas novas anotações do JDBC 4.0?

Antes mesmo de usar uma ferramenta de mapeamento objeto relacional (ORM, *object relational mapping*), há a opção de abstrair o acesso direto ao JDBC para fazer esse trabalho tomando as devidas precauções e usando as melhores práticas. São muitos

os casos em que não conseguimos usar uma ferramenta ORM: problemas políticos com DBAs, uso excessivo de *stored procedures* ou migração de uma aplicação com queries SQLs já existentes e até espalhadas pelo código. Um *Data Mapper* bastante utilizado é o **iBatis**, um framework que já encapsula as necessidades quando utilizamos o JDBC e até organiza as queries SQLs [5]. Utilizar o Spring JDBC template é uma outra alternativa.

Mapeamento objeto relacional

O passo adiante é adotar uma ferramenta ORM. Qual das diversas opções devemos escolher? Sua ferramenta deve ser capaz de fazer a ponte entre o mundo entidade relacional e o orientado a objetos de forma a minimizar o abismo entre os conceitos desses dois mundos, conhecido como *object relational impedance mismatch* [4].

Um exemplo são as tabelas compridas que aparecem com certa frequência no mundo relacional e, diferentemente de classes com muitos atributos na orientação a objetos, não são vistas com tanta repulsa. Ferramentas de mapeamento devem possibilitar que mais de uma classe seja utilizada para representar uma única tabela. O contrário também pode acontecer: uma classe ser mapeada em mais de uma tabela.

Outro caso são as tabelas associativas simples. Elas são a forma de fazer o relacionamento *muitos para muitos* entre entidades no modelo relacional. Ao mapear isso, sua ferramenta deve arrumar uma forma elegante de esconder esses aglomerado de pares de chaves, provavelmente nem criando uma classe para tal, usando de coleções e arrays de objetos.

Já a herança é um conceito que não bate com o mundo entidade-relacional: como então representá-la? Em alguns frameworks, o uso da herança pode formar um sério gargalo de acordo com a forma de representação adotada no banco de dados (as conhecidas estratégias da JPA e do Hibernate), dado que diversas tabelas precisam ser consultadas em algumas queries polimórficas, em especial quando é impossível usar alguns recursos do SQL, como *unions* [53]. Muitos vão desaconselhar a herança, preferindo usar composição, conforme visto no tópico *Evite herança, favoreça composição* [21, **?**].

Com a força que as especificações Java EE têm, é comum que a escolha por uma ferramenta ORM acabe caindo sobre uma das implementações da *Java Persistence API* - JPA. Dentre elas, as principais concorrentes são o Oracle TopLink, a OpenJPA, a BEA Kodo, o EclipseLink e o Hibernate. Seja qual for sua escolha, há alguns cuidados que devem ser tomados.

O primeiro é entender e usar corretamente os modos *eager* e *lazy* de se obter objetos relacionados. Sempre que temos relacionamentos com `@xxToOne`, o padrão é trazer o objeto relacionado (*eager*); e, quando temos `@xxToMany`, a coleção de objetos relacionados só é recuperada quando for acessada (*lazy*). Para boa parte dos casos, o comportamento padrão é suficiente, mas é preciso identificar casos onde precisamos configurar um `FetchType` diferente (por exemplo quando um objeto relacionado é muito pesado e queremos comportamento *lazy* ao invés de *eager*).

Há também a possibilidade de se ajustar o fetch type diretamente nas consultas, por exemplo fazendo uma busca específica como *eager* onde o relacionamento está configurado como *lazy*. Algumas implementações de JPA, como a OpenJPA e o Hibernate,

ainda permitem que campos básicos sejam *lazy* sem a necessidade de uma pré compilação, possibilitando que uma entidade seja parcialmente carregada e que campos chaves sejam inicializados somente quando e se necessários.

Outro cuidado a ser tomado é na integração com o restante da aplicação. Na Web, acessamos entidades do nosso modelo nos JSPs e, se essas entidades possuem relacionamentos lazy, precisamos que o `EntityManager` que as carregou esteja aberto até o fim da renderização do JSP. Caso ele esteja fechado, a maioria das implementações de JPA não vai reconectar ao banco de dados e lançará uma exception (`LazyInitializaionException`, no caso do Hibernate).

Para que isso não ocorra, devemos manter o `EntityManager` aberto, seja através de um filtro, de um interceptador ou algum outro mecanismo. Esse é o conhecido pattern *Open EntityManager in View* (derivado do *Open Session in View*, termo cunhado pelo Hibernate).

Outra forma de escapar do problemas de proxies *lazy* é utilizar um container de injeção de dependências. Frameworks como o Spring e VRaptor já possuem filtros e interceptadores para esses casos, e até mesmo usando anotações padrão do Java EE 5 para injeção de dependências podemos obter o mesmo efeito (no caso, a `@PersistenceContext`) [56].

Com o advento do Java EE 5 e algumas formas de injeção de dependências, um grande número de patterns que antes existiam para o J2EE sumiram, porém algumas preocupações ainda persistem. A principal delas é como acessar um `EntityManager`? Dentro de um objeto que age como um DAO? Um Domain Model? Um Model Façade? Ou devemos acessá-lo diretamente da nossa lógica de negócios? Apesar desse ainda ser um debate onde muitas opiniões divergem, a grande maioria dos desenvolvedores vai optar por encapsular o acesso ao `EntityManager`, não permitindo que qualquer código o acesse e também ajudando a reaproveitar métodos que encapsulam rotinas de acesso a dados que são mais que apenas a execução de uma query [45, 57].

Por uma questão de ajuste de granularidade, para que não seja exposto mais dados que o necessário, ou para agrupar diferentes pedaços de entidades em um único objeto, é possível que haja a necessidade do *Data Transfer Object* [46]. Outros patterns também foram catalogados mas muitos deles não foram largamente adotados pela comunidade ou possuem nomenclatura e implementações divergentes [42, 44].

Considerações importantes sobre o Hibernate

A quase onipresença do Hibernate, tanto através do uso da JPA ou diretamente, faz com que o conhecimento profundo das funcionalidades oferecidas por este framework mereça especial atenção.

Apesar do esforço da JPA 1.0 em unificar uma grande parte de funcionalidades comuns a todas as ferramentas ORM em uma especificação, muito ficou de fora. A JPA 2.0 tenta até diminuir essas grandes diferenças entre os diversos fabricantes existentes, criando novas anotações e estendendo a interface `EntityManager` com novos métodos e recursos.

Dentre alguns recursos importantes que existem no Hibernate porém não na JPA, podemos citar alguns que são vitais para a escalabilidade e performance da aplicação. O desconhecimento dessas funcionalidades pode fadar até mesmo um projeto simples

ao fracasso, seja pelo consumo excessivo de memória, ou pelo disparo exagerado de queries [54].

Usar um *pool de conexões*, por exemplo, é praticamente obrigatório em qualquer aplicação que use banco de dados. Um problema comum em aplicações que abrem conexões indiscriminadamente é sobrecarregar o banco com um número muito grande de conexões. O uso do pool permite controlar o máximo de conexões que podem ser abertas. Além disso, abrir e fechar conexões são operações custosas. Não vale a pena estabelecer toda a comunicação com o banco, abrir a conexão, para usá-la rapidamente e depois logo fechá-la. Usar o pool nos economiza bastante tempo ao manter as conexões abertas e compartilhá-las.

Há diversos pools de conexão no mercado já implementados e prontos para uso. Desde os próprios *data sources* dos servidores até bibliotecas separadas como o C3P0 e o DBCP. E usar um pool de conexões com o Hibernate é extremamente fácil, bastando umas poucas linhas de configuração XML [27].

Um problema que devemos tratar com o uso de um pool é ficar com conexões quebradas dentro do pool, o que exibe a famosa `java.net.SocketException: Broken pipe` quando tentamos usá-las. Temos que tomar o cuidado de checar de alguma forma se a conexão ainda está aberta, seja testando a cada vez que pegamos uma conexão do pool ou fazendo checagens periódicas [52].

Outro ponto importante para uma boa performance e escalabilidade é evitar consultas repetidas em intervalos pequenos de tempo: o sistema responde rápido e o banco de dados não é acionado. Dentro de uma mesma `Session`, as entidades gerenciadas por ela ficam em um cache, o cache de primeiro nível (*first level cache*), até que essa sessão seja fechada ou a entidade seja explicitamente removida (através do `clear` ou `evict`). Esse primeiro nível de cache é bastante útil, já que dentro de uma mesma funcionalidade é possível que a mesma informação seja puxada de mais de uma forma, mais de uma vez.

Pensando em um contexto Web, o cache de primeiro nível costuma existir durante o ciclo de vida de uma requisição. Mas podemos ir além, já que muitas entidades sofrem poucas alterações e podem ter seus dados cacheados por uma região maior que a delimitada por uma única requisição. O **cache de segundo nível** (*second level cache*) no Hibernate tem esse papel. E através de simples anotações podemos tirar proveito de robustos mecanismos de cache como o *Ehcache* ou o *JBoss Cache*. É possível ainda configurar detalhes como a política do ciclo de vida das entidades no nosso cache (LRU, LFU, FIFO, ...), seu respectivo tamanho, os tempos de expiração, entre outros [12].

Quando uma determinada entidade for buscada através de sua chave primária e ainda não estiver no cache de primeiro nível, o Hibernate vai passar pelo cache de segundo nível para verificar se essa entidade se encontra lá. Caso o seu tempo de expiração não tenha passado, uma cópia desse objeto será devolvida sem necessidade de acesso ao banco de dados. E se as tabelas em questão são apenas acessadas por sua aplicação, é possível configurar até que esse cache nunca expire se não houver uma atualização na entidade.

Fazer um cache eficiente evitando o vazamento de memória e problemas de concorrência sem perder escalabilidade é uma tarefa difícil e aqui toda essa infraestrutura já está disponível. Mas toda estratégia de cache é periogosa porque, para evitar ficar com

dados obsoletos, é preciso uma boa política de invalidação em caso de atualizações. O Hibernate faz tudo isso de forma transparente se as atualizações são feitas através dele. Mas se nossos objetos tiverem muitas atualizações em comparação com o número de consultas, o efeito pode ser o inverso do desejado: o Hibernate gastará muito tempo nas invalidações de cache e a performance provavelmente será prejudicada. **Use cache apenas em entidades que não mudam nunca ou nas que mudam muito pouco em relação às consultas**.

O que vimos até aqui é o chamado cache de entidades. Mas frequentemente executamos pesquisas que retornam o mesmo resultado até que as tabelas envolvidas sofram alterações. O Hibernate pode cachear até mesmo o resultado de queries, através do *query cache*. Guardar todo o resultado de uma query consumiria muita memória e dados repetidos e é por isso que a ferramenta vai apenas armazenar as chaves primárias das entidades devolvidas pela query. Quando a query for executada mais uma vez, o Hibernate vai pegar esse conjunto de chaves primárias e buscá-las, sem a necessidade de executar a query de novo. Por isso é importante também colocar essas entidades, não apenas a query, no cache de segundo nível. Esse resultado de query cacheado será invalidado ao menor sinal de modificação nas tabelas envolvidas, pois isso pode alterar o que essa query iria retornar. Queries que envolvam tabelas modificadas constantemente não são boas candidatas ao *query cache*.

É possível que você não queira invalidar todo o cache de uma query a cada modificação nas respectivas tabela, pois diversas vezes é aceitável que a resposta para uma query não seja consistente com os dados mais atuais. Esse comportamento de invalidação do cache de queries pode ser modificado para um ajuste mais fino, através da substituição da `StandardQueryCache`.

Além das estratégias de cache, o Hibernate ainda disponibiliza outros mecanismos de otimização. Quando executamos uma query, é comum iterarmos sobre todas as entidades retornadas. Se o resultado de uma query é um `List<Livro>` e `Livro` possui como atributo uma `List<Autor>` que será carregado de maneira *lazy*, temos então um potencial problema: ao fazermos o laço para mostrar todos os autores desses livros, para cada invocação `livro.getAutores()` uma nova query será executada, por exemplo:

```
select * from Autor where livro_id = 1
select * from Autor where livro_id = 3
select * from Autor where livro_id = 7
select * from Autor where livro_id = 12
select * from Autor where livro_id = 15
select * from Autor where livro_id = 16
```

Esse problema é conhecido como `n+1`, pois ao executarmos uma única query, temos como efeito colateral a excecução de mais `n` queries, onde `n` é o número de entidades resultante da query original e que pode ser bem grande.

Podemos configurar o Hibernate para que ele inicialize a coleção *lazy* de autores dos livros não de uma em uma mas sim de `5` em `5`, valor que pode ser definido através da anotação `@BatchSize(size=5)` e diminuiria bastante as nossas queries de inicialização:

```
select * from Autor where livro_id in {1, 3, 7, 12, 15}
```

```
select * from Autor where livro_id in {16}
```

Assim, minimizamos ainda mais o número de queries executadas, unindo diversas queries em uma única, diminuindo o número de *roundtrips* até o banco de dados.

Mas como detectar que o problema do `n+1` está ocorrendo? Não é viável ficar olhando o log de queries em produção, pois podemos deixar escapar facilmente diversos gargalos. O Hibernate oferece a classe `Statistics` que, se habilitada, vai armazenar dados sobre entidades, relacionamentos, queries e cache. É possível ver com detalhes quantas vezes cada query está sendo executada, os tempos máximos, mínimos e médios; ver ainda quantas vezes o cache de segundo nível é invalidado em comparação a quantas vezes ele é usado; ou até ver quantas transações foram abertas e não comitadas, entre outras coisas. Através desta classe, é muito mais fácil identificar gargalos e problemas para sabermos onde colocar um novo cache, onde usar o `@BatchSize`, onde é melhor ser eager ou lazy, etc.

Ainda falando em otimizações, sabemos que, frequentemente, critica-se o Hibernate em situações que envolvem o processamento de muitos dados de uma vez. Mas o Hibernate possui mecanismos específicos para trabalhar nesses casos [26]. Imagine que precisamos fazer o processamento de um imenso arquivo TXT e inserir milhões de linhas no banco de dados de uma vez. Se chamarmos simplesmente `session.save()` (ou `entityManager.persist()`) várias vezes provavelmente vamos receber um `OutOfMemoryError`.

A noção de contexto persistente atrelado à sessão faz com que todos os objetos inseridos sejam armazenados no cache de primeiro nível da sessão até que ela seja fechada. E inserir muitos objetos nesse cache não é uma boa ideia. Soluções possíveis vão desde chamar `session.clear()` de vez em quando para limpar esse cache até usar diretamente a `StatelessSession`, um outro tipo de sessão que não guarda estado (e por isso perde vários benefícios do Hibernate, mas resolve problemas de manipulação de muitos objetos).

Já quando o problema for trazer muitos registros do banco para a memória, melhor que executar uma simples *query* e pegar uma imensa lista de objetos que provavelmente estourará a memória, é estudar estratégias de como pegar poucos objetos por vez. Nesse cenário, os `ScrollableResults` permitem que usemos o cursor diretamente no banco de dados evitando trazer todos os dados de uma vez para a memória.

Mas temos até que considerar se vale a pena trazer os elementos para a memória. Muitos processamentos podem ser feitos diretamente no banco de dados através de *batch updates*. Por exemplo, reajustar em 10% os preços de todos os produtos de uma loja: ao invés de trazer todos os produtos para a memória e atualizar chamando o `setPreco`, podemos executar um único *UPDATE* diretamente no banco de dados. O Hibernate suporta esse tipo de processamento através de *UPDATE*, *DELETE* e *INSERT...SELECT* diretamente pela HQL.

Devemos sempre lembrar também que é possível executar queries nativas pelo Hibernate. Embora uma das grandes facilidades do framework seja a geração das queries, existem situações onde queremos escrever algumas delas diretamente. Um bom motivo é aproveitar aquelas queries super complicadas que o DBA escreveu com todos os detalhes de otimização do banco; ao invés de tentar traduzir tudo para o HQL, porque não usar a query que já está pronta? Com queries nativas conseguimos chamar recursos

bem específicos do banco de dados que estamos usando, inclusive *stored procedures*, e obter um grau a mais de otimização nas situações em que performance for mais importante que portabilidade de banco de dados.

Citamos aqui diversas funcionalidades que podem ajudar muito o desenvolvimento com o Hibernate e a JPA, e não usar algumas dessas pode criar um enorme gargalo de performance. Há ainda diversas outras funcionalidades, como a existência do modo de *fetch extra lazy*, fazendo com que uma coleção não seja inteiramente inicializada: queries diferentes serão disparadas para chamadas como `lista.size()`, `lista.get(15)` e até mesmo `lista.add(entidade)`, evitando carregamento de dados desnecessários.

É importante conhecer a fundo as funcionalidades que afetam a performance do Hibernate [28]. Novamente aqui, **usar um framework sem um conhecimento significativo sobre suas funcionalidades pode causar gargalos que seriam facilmente evitados**.