JPA: Persistindo dados com um framework ORM para Java

Luis Fernando de Barros[1], Rafael Fernando Rutsatz[1], Rodrigo Lucke[1], Geisa Giovana Kochenborger[1], Diego Rezek[1], Daniela Inês Fengler[1]

[1]Universidade de Santa Cruz do Sul (UNISC)
Santa Cruz do Sul – RS – Brazil

{luibs91,rafa.rutsatz,digolucke}@gmail.com, geisa.giovana@hotmail.com

{dih.rezek,danielafengler}@gmail.com

Abstract. *This paper aims to demonstrate how to apply the Object-Relational Mapping (ORM) concepts using the Hibernate framework available for Java. We introduce the concepts involved and how to apply them in an actual application. We demonstrate how to perform this mapping through annotations.*

Resumo. *Este artigo visa demonstrar como aplicar os conceitos de ORM (Mapeamento Objeto-Relacional) utilizando o framework Hibernate disponível para Java. Apresentamos os conceitos envolvidos e como aplicá-los num aplicativo real. Demonstramos como realizar esse mapeamento através das annotations.*

1. Introdução

Quem já trabalhou com Java, já deve ter usado JDBC para interagir com o banco de dados. Com ele, nos deparamos com rotinas chatas e entediantes ao escrever comandos SQL. Um exemplo disso é quando fazemos um while para popular uma lista com objetos recuperados da base de dados. Podemos também cometer erros de SQL, como um erro de sintaxe ao escrever um insert ou um erro no relacionamento ao construirmos um Join. Segundo [Coelho 2014], para evitar toda essa verbosidade do JDBC, surgiu a JPA, que a cada dia que passa, ganha mais espaço no mercado. Ela pode aumentar consideravelmente o tempo de desenvolvimento da equipe, além de facilitar muito a implementação do código que manipula o banco de dados.

2. Linguagem Java

Java é uma linguagem de programação interpretada orientada a objetos desenvolvida na década de 90 por uma equipe de programadores chefiada por James Gosling, na empresa Sun Microsystems.

O Java foi projetado para permitir o desenvolvimento de aplicações portáteis de alto desempenho para a mais ampla variedade possível de plataformas de computação. [https://www.java.com/ 2017]

Diferente das linguagens de programação convencionais, que são compiladas para código nativo, a linguagem Java é compilada para um bytecode que é interpretado por uma máquina virtual (Java Virtual Machine, mais conhecida pela sua abreviação JVM) [pt.wikipedia.org 2017]. Isso faz com que o Java seja livre de plataforma.

Entre outras características estão o uso eficiente de memória, suporte nativo a threads, permitindo que o programa rode vários processos simultâneos, tratamento de erros por meio de exceções, controle de recursos, arquivos e rede, e uma extensa gama de classes nativas [http://codigofonte.uol.com.br 2008].

A necessidade de persistir e recuperar dados estão presentes em grande parte das aplicações, e para as aplicações escritas em Java não é diferente.

3. Persistência de dados

Em ciência da computação, persistência é um atributo de dados que garante que um objeto esteja disponível mesmo além do tempo de vida de uma aplicação. Sem essa capacidade, ele só existiria na memória RAM e, sendo assim, seria perdido quando a RAM parasse, por exemplo quando o computador fosse desligado.

Para uma linguagem orientada a objetos como o Java, a persistência garante que o estado de um objeto seja acessível mesmo após o aplicativo que o criou parar de executar.

Existem diferentes formas para se que possa alcançar a persistência. Uma maneira seria armazenar os dados em arquivos planos. Porém é difícil administrar maiores quantidades de dados dessa forma, pois a manutenção e consistência dos dados se torna um problema, uma vez que há dificuldade para o acesso simultâneo e a busca pelos dados é demorada.

A abordagem mais adequada e comum atualmente é o uso de um banco de dados para repositório dos dados, sendo os do tipo relacional os mais utilizados.

4. JDBC

Tradicionalmente as aplicações Java utilizam a API JDBC (Java Database Connectivity) para persistência de dados em bancos de dados relacionais. Essa API utiliza instruções SQL, formuladas na aplicação e transmitidas ao banco de dados, para executar operações de criação, leitura, atualização e exclusão (CRUD).

O código JDBC está incorporado em classes Java, sendo fortemente acoplado à lógica de negócios. Também é bastante dependente do SQL, que possui bancos com certas particularidades. Dessa forma a migração de um banco de dados para outro é um pouco mais difícil.

Ao trabalhar com grandes aplicações, o JDBC também pode ser tornar um empecilho. Isso se deve ao fato de estarmos integrando dois paradigmas diferentes. Por um lado temos o Java, uma linguagem orientada a objetos, com diferente enfoque da tecnologia de banco de dados relacional, voltada à tabelas. Logo quando é necessário que se utilize as duas tecnologias em conjunto há uma certa incompatibilidade.

5. ORM

O mapeamento objeto-relacional (ORM) aparece como uma melhor solução para a aproximação dos paradigmas objeto-relacional. Sendo uma técnica que persiste de maneira transparente nos objetos de aplicação nas tabelas em um banco de dados relacional.

O ORM se comporta como um banco de dados virtual, escondendo a arquitetura de banco de dados subjacentes do usuário, onde as tabelas do banco de dados são repre-

sentadas através de classes e os registros de cada tabela são representados como instâncias das classes correspondentes.

Existem diversos frameworks ORM disponíveis para o Java. A implementação mais famosa e utilizada no mercado é o Hibernate, mas existem outras alternativas no mercado como iBATIS, OpenJPA, EclipseLink, Batoo e outras. Cada implementação tem suas vantagens e desvantagens na hora da utilização.

5.1. Annotations

Para realizar o mapeamento ORM em Java, utilizamos as annotations fornecidas pelos frameworks. Elas são as responsáveis por identificar as classes e atributos e converter para o modelo relacional.

As annotations são iniciadas pelo caractere @ seguidos de uma palavra reservada. As principais são listadas a seguir:

- @Entity - Indica que a classe representa uma tabela.
- @Id - Indica a PK da tabela.
- @OneToOne - Mapeia um relacionamento.
- @OneToMany - Mapeia um relacionamento 1 para n.
- @ManyToMany - Mapeia um relacionamento n para n.
- @JoinColumn - Indica a coluna que será usada no relacionamento.
- @EmbeddedId - Indica que a entidade possui uma chave composta, que é mapeada pelo atributo anotado.
- @Embeddable - Indica que a classe mapeia uma chave composta.

6. Hibernate

O Hibernate é uma solução de mapeamento objeto-relacional leve e de código aberto. Sua principal característica é o suporte para modelos baseados em objetos, o que permite que ele forneça um mecanismo transparente para a persistência. Ele utiliza XML para mapear um banco de dados para um aplicativo e suporta objetos finos.

Temos incluso no Hibernate uma linguagem de consulta chamada Hibernate Query Language (HQL). O HQL é muito semelhante ao SQL, porém é completamente orientado a objetos, possibilitando que sejam aproveitadas as principais características da programação orientada a objetos, com herança, polimorfismo e associação. Ele, por exemplo, retorna resultados de consulta como objetos que podem ser acessados e manipulados diretamente.

Outra característica é o suporte a muitos bancos de dados, incluindo MySQL, Oracle, Sybase, Derby e PostgreSQL, e também modelos baseados em objetos Java simples (POJO). O Hibernate gera código JDBC com base no banco de dados escolhido removendo a necessidaade de escrever o código JDBC.

Internamente, o Hibernate usa JDBC, que fornece uma camada de abstração para o banco de dados, enquanto emprega Java Transaction API (JTA) e JNDI para se integrar com outras aplicações.

O Hibernate contém diversas funcionalidades como Engenharia Reversa, Exportação de Schema facilitado e anotações próprias para simplificar a utilização da API [http://hibernate.org/orm 2017].

7. iBATIS

O iBATIS é uma estrutura de persistência que fornece os benefícios do SQL, mas evita a complexidade do JDBC. Ao contrário da maioria dos outros frameworks de persistência, o iBATIS incentiva o uso direto do SQL e garante que todos os benefícios do SQL não sejam substituídos pela própria estrutura.

Simplicidade é a maior vantagem do iBATIS, pois fornece um mapeamento simples e uma camada API que podem ser usadas para criar código de acesso a dados. Nessa estrutura, o modelo de dados e o modelo de objeto não precisam se mapear com precisão. Isso ocorre porque o iBATIS usa um mapeador de dados, que mapeia objetos para procedimentos armazenados, instruções SQL ou ResultSets através de um descritor XML, em vez de um mapeador de metadados, que mapeia objetos no domínio para tabelas no banco de dados. Assim, o iBATIS permite que o modelo de dados e o modelo de objeto sejam independentes uns dos outros.

8. OpenJPA

A JPA usa anotações de metadados e/ou arquivos de descritor XML para configurar o mapeamento entre objetos Java no domínio do aplicativo e tabelas no banco de dados relacional. É uma solução ORM completa e suporta herança e polimorfismo. Ele também define uma linguagem de consulta semelhante ao SQL, JPQL (Java Persistence Query Language).

Com o JPA, você pode conectar qualquer provedor de persistência que implemente a especificação JPA em vez de usar qualquer provedor de persistência padrão com seu container Java EE. É feita a utilização de classes de entidades, gerentes de entidade e unidades de persistência para mapear e persistir os objetos de domínio e as tabelas no banco de dados.

9. Comparando tecnologias de persistência

Observando os três mecanismos citados anteriormente podemos perceber que cada um possui seus prós e contras. Se tratando de simplicidade podemos dizer que a melhor opção seria o iBATIS, pois este requer somente o conhecimento de SQL, dispensando treinamento adicional. Com o iBATIS temos um maior controle sobre as consultas usando diretamente o SQL, porém se o objetivo é independência do SQL, o Hibernate se torna a melhor opção, pois gera consultas eficientes em tempo de execução. O Hibernate juntamente com a JPA usam suas próprias linguagens de consulta.

Em relação à portabilidade para diferentes bancos de dados, temos maior facilidade no Hibernate, que usa uma propriedade de dialeto de banco de dados no arquivo de configuração, que pode ser alterada facilmente. No que diz respeito ao desempenho, no Hibernate é fornecido instalação de armazenamento em cache que ajuda com uma recuperação mais rápida dos dados do banco de dados. O iBATIS utiliza consultas SQL que podem ser ajustadas de acordo com a necessidade de melhor desempenho. O desempenho da JPA depende da implementação do fornecedor. A escolha é particular para cada aplicação.

10. Persistindo dados com Hibernate

Veremos um caso de uso do Hibernate para persistir dados.

10.1. Modelo ER

Veja o modelo da figura 1:

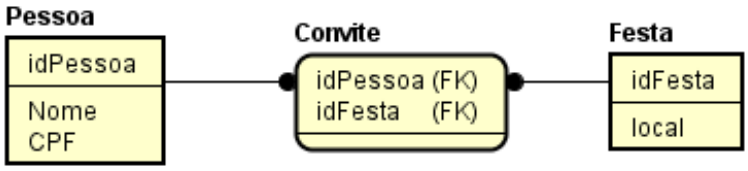

Figura 1. Modelo ER

Neste exemplo, temos um cadastro de pessoas e festas, e podemos convidar pessoas para as festas.

10.2. Mapeamento ORM

A figura 2 demonstra o mapeamento da classe Pessoa. Nela, podemos notar o uso de duas annotations novas: @SequenceGenerator e @GeneratedValue. Elas são relacionadas a estratégia de geração da chave primária. Como estamos usando o banco Postgres, adotamos a estratégia de usar Sequence.

```
@Entity
@SequenceGenerator(name = "idPessoaSeq", sequenceName = "id_Pessoa_Seq", allocationSize = 1)
public class Pessoa {

    @Id
    @Column(name = "id_Pessoa")
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "idPessoaSeq")
    private Integer idPessoa;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "id_Pessoa", foreignKey = @ForeignKey(name = "id_Pessoa_fk"))
    private List<Convite> convites = new ArrayList<>();

    private String nome;

    private String CPF;
```

Figura 2. Mapeamento da classe Pessoa

Na figura 3 temos o mapeamento da classe Festa.

A figura 4 apresenta o mapeamento da classe Convite. Na abordagem utilizada nesse exemplo, ela possui uma chave composta. Nessas situações, utilizamos uma classe que ficará responsável por mapear essa chave, que nesse caso, é utilizada a classe ConvitePK para essa função.

A figura 5 mostra a classe ConvitePK mapeando a chave composta da classe Convite.

10.3. Implementação em Java

Para persistir os dados e realizar consultas em Java, utilizamos uma classe auxiliar que é usada em todo o sistema. A figura 6 apresenta a implementação dessa classe.

A classe apresenta três métodos:

```java
@Entity
@SequenceGenerator(name = "idFestaSeq", sequenceName = "id_Festa_Seq", allocationSize = 1)
public class Festa {

    @Id
    @Column(name = "id_Festa")
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "idFestaSeq")
    private Integer idFesta;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "id_Festa", foreignKey = @ForeignKey(name = "id_Festa_fk"))
    private List<Convite> convites = new ArrayList<>();

    private String local;
```

Figura 3. Mapeamento da classe Festa

```java
@Entity
public class Convite implements Serializable {

    @EmbeddedId
    private ConvitePK convitePK;
```

Figura 4. Mapeamento da classe Convite

- getEntityManager();
- closeEntityManager();
- closeEntityManagerFactory().

Quando iniciamos a aplicação, chamamos o método getEntityManager(), o qual é responsável por chamar o método Persistence.createEntityManagerFactory("PU_FESTA"). Repare que esse método é chamado apenas uma vez, pois ele é o responsável por fazer a leitura do arquivo persistence.xml (esse arquivo contém as informações de conexão do banco e algumas configurações), e fará a leitura das classes mapeadas, validar a conexão com o banco de dados e, caso configurado, criar tabelas, criar as sequences etc. Como essa operação é muito custosa, não vale a pena repeti-lá diversas vezes. Por isso, a instância dessa classe está marcada como static, ou seja, vale para toda a aplicação. Após instanciado, ele retorna um EntityManager, que gerecia as transações da aplicação. O método closeEntityManager() comita as transações, caso houver alguma ativa e fecha o EntityManager. E o método closeEntityManagerFactory() fecha o EntityManagerFactory. Após a chamada desse método, ele encerra a comunicação com a base de dados, logo, não é mais possível persistir ou recuperar dados, sem realizar uma nova chamada do método Persistence.createEntityManagerFactory("PU_FESTA").

10.4. Recuperando Dados

Para buscar dados, utilizamos a JPQL. A figura 7 apresenta a busca de todas as festas cadastradas. Utilizamos a classe Query disponibilizado pelo framework para criar a consulta, através do método createQuery(), e passamos uma string com a consulta que queremos realizar. Após, utilizamos o método query.getResultList() para buscar esses dados. Também podemos montar consultas mais elaboradas, aplicandos filtros e outros recursos

```java
@Embeddable
public class ConvitePK implements Serializable {

    private static final long serialVersionUID = 1L;

    @Column(name = "id_Pessoa")
    @JoinColumn(name = "id_Pessoa")
    private Integer idPessoa;

    @Column(name = "id_Festa")
    @JoinColumn(name = "id_Festa")
    private Integer idFesta;
```

Figura 5. Mapeamento da classe ConvitePK

```java
public class JpaUtil {

    private static ThreadLocal<EntityManager> threadEntityManager
            = new ThreadLocal<EntityManager>();

    private static EntityManagerFactory entityManagerFactory;

    public static EntityManager getEntityManager() {

        if (entityManagerFactory == null) {
            entityManagerFactory = Persistence.createEntityManagerFactory("PU_FESTA");
        }
        EntityManager entityManager = threadEntityManager.get();

        if (entityManager == null || !entityManager.isOpen()) {
            entityManager = entityManagerFactory.createEntityManager();
            JpaUtil.threadEntityManager.set(entityManager);
        }
        return entityManager;

    }
```

Figura 6. Classe auxiliar JpaUtil

disponíveis nos bancos relacionais. A figura 8 demonstra como podemos aplicar um filtro numa consulta e como podemos relacionar duas tabelas com JPQL.

10.5. Persistindo Dados

Na figura 9 temos a implementação do método responsável por persistir uma entidade Pessoa. Nele temos o controle manual das transações, com a chamada dos métodos entityManager.getTransaction().begin() e entityManager.getTransaction().commit(). O método entityManager.persist(pessoa) recebe o objeto pessoa por parâmetro e ele que será responsável por gravar os dados no banco de dados.

11. Conclusão

Quando corretamente aplicado, a JPA se torna uma ferramenta que ajuda em todas as funções de um CRUD (Create, Read, Update and Delete), além de contribuir com diversas otimizações de performance e consistência de dados. É nesse exato momento que podemos entender por que as siglas ORM são tão faladas. Object Relational Mapping

```
EntityManager entityManager = JpaUtil.getEntityManager();

String consulta;
Query query;

consulta = "select f from Festa f";
query = entityManager.createQuery(consulta);
```

Figura 7. Buscando Dados

```
EntityManager entityManager = JpaUtil.getEntityManager();

String consulta;
Query query;

consulta = "select p "
         + "   from Pessoa p"
         + "   join p.convites c"
         + "  where c.convitePK.idFesta = :idFesta";
query = entityManager.createQuery(consulta, Pessoa.class);
query.setParameter("idFesta", festa.getIdFesta());
List p = query.getResultList();
```

Figura 8. Buscar pessoas filtrando por festa

(ORM) quer dizer basicamente: “transformar as informações de um banco de dados, que estão no modelo relacional para classes Java, no paradigma Orientado a Objetos de um modo fácil”.

```
public void salvarPessoa() {
    EntityManager entityManager = JpaUtil.getEntityManager();
    entityManager.getTransaction().begin();

    Pessoa pessoa = new Pessoa();
    pessoa.setCPF(txCPF.getText());
    pessoa.setNome(txNome.getText());

    entityManager.persist(pessoa);

    entityManager.getTransaction().commit();

    JpaUtil.closeEntityManager();
}
```

Figura 9. Implementação do método salvarPessoa

Referências

Coelho, H. (2014). *JPA Eficaz: As melhores práticas de persistência de dados em Java*. Casa do Código Editora, São Paulo.

http://codigofonte.uol.com.br (2008). http://codigofonte.uol.com.br/artigos/artigo-sobre-a-linguagem-java. Acesso em: 18 de junho de 2017.

http://hibernate.org/orm (2017). http://hibernate.org/orm/. Acesso em: 18 de junho de 2017.

https://www.java.com/ (2017). Acesso em: 18 de junho de 2017.

pt.wikipedia.org (2017). https://pt.wikipedia.org/wiki/Java_(linguagem_de_programação). Acesso em: 18 de junho de 2017.