JPA – Hibernate

Namom Alves Alencar

Bancos de dados e JDBC

- Entender a diferença entre a JPA e o Hibernate;
- Usar a ferramenta de ORM JPA com Hibernate;
- Gerar as tabelas em um banco de dados qualquer a partir de suas classes de modelo;
- Inserir e carregar objetos pelo JPA no banco;
- Buscar vários objetos pelo JPA;

Mapeamento Objeto Relacional

- Com a popularização do Java em ambientes corporativos, logo se percebeu que grande parte do tempo do desenvolvedor era gasto na codificação de queries SQL e no respectivo código JDBC responsável por trabalhar com elas.

- Além de um problema de produtividade, algumas outras preocupações aparecem: SQL que, apesar de ter um padrão ANSI, apresenta diferenças significativas dependendo do fabricante. Não é simples trocar um banco de dados pelo outro.

Mapeamento Objeto Relacional

- Há ainda a mudança do paradigma. A programação orientada a objetos difere muito do esquema entidade relacional e precisamos pensar das duas maneiras para fazer um único sistema. Para representarmos as informações no banco, utilizamos tabelas e colunas. As tabelas geralmente possuem chave primária (PK) e podem ser relacionadas por meio da criação de chaves estrangeiras (FK) em outras tabelas.

Mapeamento Objeto Relacional

- Quando trabalhamos com uma aplicação Java, seguimos o paradigma orientado a objetos, onde representamos nossas informações por meio de classes e atributos. Além disso, podemos utilizar também herança, composição para relacionar atributos, polimorfismo, enumerações, entre outros. Esse buraco entre esses dois paradigmas gera bastante trabalho: a todo momento devemos "transformar" objetos em registros e registros em objetos.

Java Persistence API e Frameworks ORM

- Ferramentas para auxiliar nesta tarefa tornaram-se popular entre os desenvolvedores Java e são conhecidas como ferramentas de mapeamento objeto-relacional (ORM). O Hibernate é uma ferramenta ORM open source e é a líder de mercado, sendo a inspiração para a especificação Java Persistence API (JPA). O Hibernate nasceu sem JPA mas hoje em dia é comum acessar o Hibernate pela especificação JPA. Como toda especificação, ela deve possuir implementações. Entre as implementações mais comuns, podemos citar: Hibernate da JBoss, EclipseLink da Eclipse Foundation e o OpenJPA da Apache. Apesar do Hibernate ter originado a JPA, o EclipseLink é a implementação referencial.

Java Persistence API e Frameworks ORM

- O Hibernate abstrai o seu código SQL, toda a camada JDBC e o SQL será gerado em tempo de execução. Mais que isso, ele vai gerar o SQL que serve para um determinado banco de dados, já que cada banco fala um "dialeto" diferente dessa linguagem. Assim há também a possibilidade de trocar de banco de dados sem ter de alterar código Java, já que isso fica de responsabilidade da ferramenta.

Java Persistence API e Frameworks ORM

- Como usaremos JPA abstraímos mais ainda, podemos desenvolver sem conhecer detalhes sobre o Hibernate e até trocar o Hibernate com uma outra implementação como OpenJPA.

Bibliotecas do Hibernate e JPA

- Vamos usar o JPA com Hibernate, ou seja, precisamos baixar os JARs no site do Hibernate. O site oficial do Hibernate é o www.hibernate.org, onde você baixa a última versão na seção ORM e Download.

- Com o ZIP baixado em mãos, vamos descompactar o arquivo. Dessa pasta vamos usar todos os JARs obrigatórios (required). Não podemos esquecer o JAR da especificação JPA que se encontra na pasta jpa.

Bibliotecas do Hibernate e JPA

- Para usar o Hibernate e JPA no seu projeto é necessário colocar todos esses JARs no classpath.

- O Hibernate vai gerar o código SQL para qualquer banco de dados. Continuaremos utilizando o banco MySQL, portanto também precisamos o arquivo .jar correspondente ao driver JDBC.

Mapeando uma classe Contato para nosso Banco de Dados

- Essa é uma classe como qualquer outra que aprendemos a escrever em Java. Precisamos configurar o Hibernate para que ele saiba da existência dessa classe e, desta forma, saiba que deve inserir uma linha na tabela Contato toda vez que for requisitado que um objeto desse tipo seja salvo. Em vez de usarmos o termo "configurar", falamos em mapear uma classe a tabela.

Mapeando uma classe Contato para nosso Banco de Dados

- Para mapear a classe Contato, basta adicionar algumas poucas anotações em nosso código. Anotação é um recurso do Java que permite inserir meta-dados em relação a nossa classe, atributos e métodos. Essas anotações depois poderão ser lidas por frameworks e bibliotecas, para que eles tomem decisões baseadas nessas pequenas configurações.

Mapeando uma classe Contato para nosso Banco de Dados

```
@Entity
public class Contato {

  @Id
  @GeneratedValue
  private Long id;

  private String nome;
  private String email;

  @Temporal(TemporalType.DATE)
  private Calendar dataDeNascimento;

  // métodos...
}
```

Mapeando uma classe Contato para nosso Banco de Dados

- @Entity indica que objetos dessa classe se tornem "persistível" no banco de dados.
- @Id indica que o atributo id é nossa chave primária (você precisa ter uma chave primária em toda entidade)
- @GeneratedValue diz que queremos que esta chave seja populada pelo banco (isto é, que seja usado um auto increment ou sequence, dependendo do banco de dados).

Mapeando uma classe Contato para nosso Banco de Dados

- @Temporal configuramos como mapear um Calendar para o banco, aqui usamos apenas a data (sem hora), mas poderíamos ter usado apenas a hora (TemporalType.TIME) ou timestamp (TemporalType.TIMESTAMP). Essas anotações precisam dos devidos imports, e pertencem ao pacote javax.persistence.

Mapeando uma classe Contato para nosso Banco de Dados

- Mas em que tabela essa classe será gravada? Em quais colunas? Que tipo de coluna? Na ausência de configurações mais específicas, o Hibernate vai usar convenções: a classe Contato será gravada na tabela de nome também Contato, e o atributo descricao em uma coluna de nome descricao também!

Mapeando uma classe Contato para nosso Banco de Dados

- Mas em que tabela essa classe será gravada? Em quais colunas? Que tipo de coluna? Na ausência de configurações mais específicas, o Hibernate vai usar convenções: a classe Contato será gravada na tabela de nome também Contato, e o atributo descricao em uma coluna de nome descricao também!

Mapeando uma classe Contato para nosso Banco de Dados

- Se quisermos configurações diferentes das convenções, basta usarmos outras anotações, que são completamente opcionais. Por exemplo, para mapear o atributo dataDeNascimento numa coluna chamada data_nascimento faríamos:

```
@Column(name = " data_nascimento", nullable = true)
private Calendar dataDeNascimento;
```

Mapeando uma classe Contato para nosso Banco de Dados

- Para usar uma tabela com o nome Cadastro:

```
@Entity
@Table(name=" Cadastro")
public class Contato {
}
```

- Repare que nas entidades há todas as informações sobre as tabelas. Baseado nelas podemos até pedir gerar as tabelas no banco, mas para isso é preciso configurar o JPA.

Configurando o JPA com as propriedades do banco

- Em qual banco de dados vamos gravar nossos Contatos? Qual é o login? Qual é a senha? O JPA necessita dessas configurações, e para isso criaremos o arquivo persistence.xml.

- Alguns dados que vão nesse arquivo são específicos do Hibernate e podem ser bem avançados, sobre controle de cache, transações, connection pool etc...

Configurando o JPA com as propriedades do banco

- Para nosso sistema, precisamos de quatro linhas com configurações que já conhecemos do JDBC: string de conexão com o banco, o driver, o usuário e senha. Além dessas quatro configurações, precisamos dizer qual dialeto de SQL deverá ser usado no momento que as queries são geradas; no nosso caso, MySQL. Vamos também mostrar o SQL no console para acompanhar o trabalho do Hibernate.

- Vamos também configurar a criação das tabelas baseado nas entidades.

Configurando o JPA com as propriedades do banco

- Segue uma configuração completa que define uma unidade de persistência (persistence-unit) com o nome contatos, seguidos pela definição do provedor, entidades e properties:
- Vamos abrir o arquivo xml

- É importante saber que o arquivo persistence.xml deve ficar na pasta META-INF do seu classpath.

Usando o JPA

- Para usar o JPA no nosso código Java, precisamos utilizar sua API. Ela é bastante simples e direta e rapidamente você estará habituado com suas principais classes.

- Nosso primeiro passo é fazer com que o JPA leia a nossa configuração: tanto o nosso arquivo persistence.xml quanto as anotações que colocamos na nossa entidade Contato. Para tal, usaremos a classe com o mesmo nome do arquivo XML: Persistence. Ela é responsável de carregar o XML e inicializar as configurações. Resultado dessa configuração é uma EntityManagerFactory:

Usando o JPA

```
EntityManagerFactory factory =
Persistence.createEntityManagerFactory(“contatos");
```

- Estamos prontos para usar o JPA. Antes de gravar um Contato, precisamos que exista a tabela correspondente no nosso banco de dados. Em vez de criarmos o script que define o schema (ou DDL de um banco, data definition language) do nosso banco (os famosos CREATE TABLE ....) podemos deixar isto a cargo do próprio Hibernate. Ao inicializar a EntityManagerFactory também já será gerada uma tabela Contatos pois configuramos que o banco deve ser atualizada pela propriedade do Hibernate: hbm2ddl.auto.

Configurando o JPA com Hibernate

- Caso você esteja fazendo esse passo de casa. É preciso baixar o ZIP do Hibernate ORM 4.x (http://hibernate.org), extraí-lo, e copiar todos os JARs das pastas lib/required e lib/jpa.

- A versão 5 já está disponível, porém utilizaremos a versão 4, pois é com esta versão que a grande maioria dos desenvolvedores utilizam.

Configurando o JPA e gerando o schema do banco

- Cole todos os JARs na pasta WebContent/WEB-INF/lib
- Anote a classe Contato como uma entidade de banco de dados. Lembre-se que essa é uma anotação do pacote javax.persistence.
- Na mesma classe anote seu atributo id como chave primária e como campo de geração automática.
- Agora é preciso mapear o atributo dataDeNascimento para gravar apenas a data (sem hora) no banco.

Configurando o JPA e gerando o schema do banco

- Vamos configurar o persistence.xml
- Crie a classe GeraTabelas

```
public class GeraTabelas {

  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence.
        createEntityManagerFactory(“contatos");

    factory.close();
  }
}
```

- Rode o programa e veja no banco de dados o resultado.

Configurando o JPA e gerando o schema do banco

- Caso algum erro tenha ocorrido, é possível que o Hibernate tenha logado uma mensagem, mas você não a viu dado que o Log4J não está configurado. Mesmo que tudo tenha ocorrido de maneira correta, é muito importante ter o Log4J configurado.
- Para isso, adicione no arquivo log4j.properties dentro da pasta src para que todo o log do nível info ou acima seja enviado para o console appender do System.out (default do console):

- log4j.logger.org.hibernate=info

Trabalhando com os objetos: o EntityManager

- Para se comunicar com o JPA, precisamos de uma instância de um objeto do tipo EntityManager. Adquirimos uma EntityManager através da fábrica já conhecida: EntityManagerFactory.

```
EntityManagerFactory factory =
Persistence.createEntityManagerFactory(“contatos");

EntityManager manager = factory.createEntityManager();

manager.close();
factory.close();
```

Persistindo novos objetos

- Através de um objeto do tipo EntityManager, é possível gravar novos objetos no banco. Para isto, basta utilizar o método persist dentro de uma transação:

```
Contato c1 = new Contato();
c1.setNome(“Namom");
c1.setEmail(“namomalencar@gmail.com”);
c1.setDataDeNascimento(Calendar.getInstance());
```

Persistindo novos objetos

```
EntityManagerFactory factory =
Persistence.createEntityManagerFactory(“contatos");
EntityManager manager = factory.createEntityManager();

manager.getTransaction().begin();
manager.persist(c1);
manager.getTransaction().commit();

System.out.println("ID do contato: " + c1.getId());

manager.close();
```

Carregar um objeto

- Para buscar um objeto dada sua chave primária, no caso o seu id, utilizamos o método find, conforme o exemplo a seguir:

```
EntityManagerFactory factory =
Persistence.createEntityManagerFactory(“contatos");
EntityManager manager = factory.createEntityManager();

Contato encontrada = manager.find(Contato.class, 1L);

System.out.println(encontrada.getNome());
```

Exercícios: Gravando e Carregando objetos

- 1 – Vamos criar uma classe chamada AdicionarContato.
  - Verifique o console do projeto e veja os sqls gerados pelo hibernate.
  - Depois de rodar sua aplicação verifique no banco de dados: select * from contato;
- 2 – Vamos criar um classe chamada CarregarContato
  - Utilize o find para carregar um contato pelo id
  - Imprima no console de forma amigável todas as informações utilizando o toString.

Removendo e atualizando objeto

- Remover e atualizar os objetos com JPA também é muito simples. O EntityManager possui métodos para cada operação. Para remover é preciso carregar a entidade antes e depois usar o método remove:

```
EntityManager manager = //abrir um EntityManager
Contato encontrado = manager.find(Contato.class, 1L);

manager.getTransaction().begin();
manager.remove(encontrado);
manager.getTransaction().commit();
```

Removendo e atualizando objeto

- Para atualizar um entidade que já possui um id existe o método merge, por exemplo:

```
Contato c1 = new Contato();
c1.setId(2); //esse id já existe no banco
c1.setNome("Estudar JPA e Hibernate");
c1.setEmail(“email@email.com”);
c1.setDataDeNascimento(null);
EntityManager manager = //abrir um EntityManager
manager.getTransaction().begin();
manager.merge(c1);
manager.getTransaction().commit();
```

Removendo e atualizando objeto

- Para atualizar um entidade que já possui um id existe o método merge, por exemplo:

```
Contato c1 = new Contato();
c1.setId(2); //esse id já existe no banco
c1.setNome("Estudar JPA e Hibernate");
c1.setEmail(“email@email.com”);
c1.setDataDeNascimento(null);
EntityManager manager = //abrir um EntityManager
manager.getTransaction().begin();
manager.merge(c1);
manager.getTransaction().commit();
```

Exercicios

- 1 – Vamos criar uma classe chamada AtualizarContato.
  - Verifique o console do projeto e veja os sqls gerados pelo hibernate.
  - Depois de rodar sua aplicação verifique no banco de dados: select * from contato;
- 2 – Vamos criar um classe chamada RemoverContato
  - Depois de rodar sua aplicação verifique no banco de dados: select * from contato;

Buscando com uma cláusula where

- O JPA possui uma linguagem própria de queries para facilitar a busca de objetos chamada de JPQL. O código a seguir mostra uma pesquisa que retorna todas os contatos com nome = namom:

```
EntityManager manager = //abrir um EntityManager
List<Contato> lista = manager
  .createQuery("select t from Contato as t where t.nome = namom")
  .getResultList();

for (Contato t : lista) {
  System.out.println(t.getEmail());
}
```

Buscando com uma cláusula where

- Também podemos passar um parâmetro para a pesquisa. Para isso, é preciso trabalhar com um objeto que representa a pesquisa (javax.persistence.Query):

```
EntityManager manager = //abrir um EntityManager

Query query = manager
    .createQuery("select t from Contato as t "+
        "where t.nome = :paramNome");
query.setParameter("paramNome", “Namom”);

List<Contato> lista = query.getResultList();
```

Buscando com uma cláusula where

- Também podemos passar um parâmetro para a pesquisa. Para isso, é preciso trabalhar com um objeto que representa a pesquisa (javax.persistence.Query):

```
EntityManager manager = //abrir um EntityManager

Query query = manager
    .createQuery("select t from Contato as t "+
        "where t.nome = :paramNome");
query.setParameter("paramNome", “Namom”);

List<Contato> lista = query.getResultList();
```

Buscando com uma cláusula where

- Uma confusão que pode ser feita a primeira vista é pensar que o JPA com Hibernate é lento, pois, ele precisa gerar as nossas queries, ler objetos e suas anotações e assim por diante. Na verdade, o Hibernate faz uma série de otimizações internamente que fazem com que o impacto dessas tarefas seja próximo a nada. Portanto, o Hibernate é, sim, performático, e hoje em dia pode ser utilizado em qualquer projeto que se trabalha com banco de dados.

Exercícios: Buscando com JPQL

- 1 - Crie uma classe chamada BuscaContatos.
  - Busca uma lista de contatos utilizando jpql
  - Verifique a saída no console
  - Imprima no console de forma amigável todos os contatos

Bons Estudos

Namom Alves Alencar