Artigo

Hibernate Annotations

Mapeamento Objeto-Relacional (MOR) utilizando anotações para descrever os metadados necessários. Nesse artigo serão usadas diversas anotações da especificação JPA que o Hibernate implementa e algumas classes providas pelo Hibernate para que a persistência e recuperação dos dados seja tão fácil quanto o mapeamento das classes

Atenção: esse artigo tem um vídeo complementar. Clique e assista!

De que se trata o artigo:

Mapeamento Objeto-Relacional (MOR) utilizando anotações para descrever os

mapeamento das classes.

Para que serve:

Conhecer um framework de mapeamento de fácil aprendizado e utilização para diminuir o tempo gasto com a persistência e recuperação de objetos em um banco de dados relacional.

Em que situação o tema é útil:

É útil no desenvolvimento de aplicações de qualquer natureza, onde o desenvolvedor não queira perder muito tempo escrevendo códigos para a persistência e recuperação de seus objetos em um banco de dados relacional.

Resumo DevMan:

Neste artigo vimos como utilizar o framework Hibernate Annotations como uma implementação da especificação JPA para realizar o mapeamento objeto-relacional através de anotações. Utilizando o escopo de uma aplicação para uma locadora de DVDs, mapeamos as tabelas do banco de dados em classes através de anotações nestas e em seus atributos. Uma vez que as classes e seus atributos estão anotados, utilizamos apenas o objeto Session para persistir e recuperar os objetos do banco de dados.

Desde que surgiu o Hibernate para ser uma ponte entre a orientação a objetos e a persistência de dados em bancos de dados relacionais, tudo o que o desenvolvedor precisa fazer é descrever suas classes com alguns metadados para

Durante um bom tempo os metadados foram descritos através de XML (e são até hoje). Para cada classe que seria persistida no banco de dados era criado um novo XML explicando como o Hibernate deveria realizar o mapeamento entre atributos da classe e colunas da tabela.

Atualmente este não é mais o único modo de descrever os metadados, existe também o Hibernate Annotations, projeto que provê a descrição dos metadados através de anotações. Deste modo, ao invés de um arquivo XML para cada classe, os metadados podem ser anotados na própria classe e em seus atributos, facilitando ainda mais o processo de mapeamento.

Nesse artigo serão apresentados os conceitos necessários para entender como utilizar o framework Hibernate para persistir as classes Java em bancos de dados relacionais apenas com a descrição de metadados providos por anotações, utilizando, para isso, a IDE NetBeans 6.9.1 e o banco de dados MySQL.

Hibernate e JPA

Algum tempo depois do Hibernate fazer sucesso entre os desenvolvedores, a Sun reconheceu que o mapeamento objeto-relacional (para saber mais veja o **quadro** "O que é Mapeamento Objeto-Relacional?") era algo necessário no desenvolvimento de aplicações e criou a especificação JPA (*Java Persistence API*) baseando-se nas funcionalidades que o Hibernate já havia implementado.

Como a Sun tinha maior alcance e influência sobre os desenvolvedores do que os criadores do Hibernate, foi natural que a JPA ficasse conhecida rapidamente. Após o lançamento da especificação os criadores do Hibernate a implementaram

Esta é uma situação que gera grandes confusões e dúvidas nos desenvolvedores: apesar do Hibernate ter surgido antes da JPA e ter sido utilizado como referência para a sua especificação, ele também a implementa, ou seja, pode ser utilizado como um framework independente ou como uma implementação da JPA.

Para empregar o Hibernate como implementação da JPA deve-se utilizar as anotações e classes que se encontram no pacote javax.persistence.*.

> Caso esteja desenvolvendo uma aplicação que utilize implementações de especificações da Java EE (EJB, JAX-WS, JSF, entre outras) é aconselhado o uso das anotações definidas pela JPA, pois tais implementações possuem grande integração com a JPA (que também é uma especificação da Java EE).

O que é Mapeamento Objeto-Relacional?

> Mapeamento Objeto-Relacional (MOR ou ORM do inglês Object-Relational Mapping) é uma técnica utilizada para diminuir o tempo que um desenvolvedor leva para persistir objetos em bancos de dados relacionais. As tabelas do banco de dados são representadas através de classes e os registros das tabelas são objetos (instâncias das classes).
>
> Utilizando essa técnica o desenvolvedor não precisa se preocupar tanto com os comandos SQL para as operações CRUD (*Create, Read, Update and Delete*), pois a própria ferramenta que implementa essa técnica deverá se preocupar com tais comandos.
>
> Para aplicar a técnica de MOR, os atributos da classe devem representar colunas na tabela do banco de dados, porém as tabelas e classes não precisam ser idênticas, pode existir um atributo na classe que não esteja diretamente relacionado com a tabela do banco de dados.

> classes e definida pelo próprio desenvolvedor. Tal processo é mais conhecido como "mapeamento".
>
> A maneira como o desenvolvedor mapeia a relação entre tabelas e classes varia de acordo com o framework utilizado (cada um tem suas particularidades). O Hibernate, por exemplo, propicia duas formas: descrever os metadados necessários em um XML ou então descrever os metadados na própria classe utilizando anotações.

Anotações

A JPA possui várias anotações para o mapeamento de classes em tabelas. A seguir veremos as anotações que iremos utilizar no projeto descrito no próximo tópico e um pequeno comentário sobre elas:

• @Entity – toda classe que represente uma tabela no banco de dados deve ser anotada com essa anotação;

• @Table(name) – anotação responsável por dizer qual tabela no banco de dados a classe irá representar. O parâmetro name deve ser uma String contendo o nome da tabela;

• @Column(name, nullable, length) – cada atributo da classe que represente uma coluna da tabela no banco de dados deve ser anotado com essa anotação. Segue detalhes sobre os parâmetros:

o name – nome da coluna (String);

o nullable – valor que representa se o atributo poderá ter valor nulo ou não (Boolean);

o enumerador será persistido no banco de dados:

o EnumType.STRING – o valor textual da opção será armazenado no banco (caso a opção escolhida seja Sexo.M será armazenada a String “M”);

o EnumType.ORDINAL – o número que representa a posição da opção será armazenado no banco (caso seja escolhido Sexo.M será armazenado 0, Sexo.F será 1).

• @Temporal(TemporalType) – utilizada para atributos que representam data.

o TemporalType.DATE – armazena somente a data (dia, mês e ano) do atributo no banco;

o TemporalType.TIME – armazena somente o horário (hora, minuto e segundo) do atributo no banco;

o TemporalType.TIMESTAMP – armazena data e horário do atributo no banco.

• @Id – informa qual atributo é a chave primária da tabela. Só um atributo da classe pode estar anotado com essa anotação.

• @SequenceGenerator(name, sequenceName) – pode anotar tanto um atributo quanto a classe. É utilizado para mapear um gerador de sequência do banco. Seus parâmetros são:

o name – nome que as outras anotações deverão utilizar para referenciar o gerador;

o sequenceName – nome do gerador no banco de dados.

ou GenerationType.IDENTITY ou GenerationType.SEQUENCE ou GenerationType.AUTO) que informa qual algoritmo será utilizado para gerar o próximo número da sequência. Como varia muito de banco para banco é uma boa prática utilizar GenerationType.AUTO;

o generator – nome do gerador (anteriormente especificado pela anotação @SequenceGenerator).

• @OneToMany(fetch, mappedBy) – utilizada para relacionamentos 1:N (um para muitos). Anota uma lista de objetos que referenciam a classe atual. Ex: Venda tem uma lista de ItemVenda. Tal lista deve ser anotada com @OneToMany. Seus parâmetros são:

o fetch = informa o modo de carregar os objetos dessa lista. Se a opção selecionada for FetchType.EAGER, todos os objetos da lista são carregados no momento em que o objeto pai é carregado. Se a opção selecionada for FetchType.LAZY, a lista não vai ser carregada junto com o objeto pai. Ela será carregada do banco de dados na sua primeira utilização (ex: na primeira vez que o método getItemVendaList() for chamado);

o mappedBy – nome do atributo na classe filha que representa a classe pai.

• @ManyToOne(fetch) – utilizada para relacionamentos 1:N (um para muitos). Anota o atributo que representa a classe pai. O parâmetro fetch é igual ao descrito acima;

• @JoinColum(name, referencedColumnName, nullable) – utilizado em atributos anotados com @ManyToOne. Descreve quais colunas serão utilizadas para fazer

JOIN. Apenas o nome da coluna deve ser informada. Mesmo que outras tabelas tenham colunas com o mesmo nome o Hibernate irá saber com qual tabela fazer o JOIN, pois o atributo anotado com essa anotação está sendo referenciado através do parâmetro mappedBy da anotação @OneToMany no seu pai;

o nullable – mesmo comportamento deste parâmetro na anotação @Column.

• @ManyToMany – utilizada para mapear relacionamentos N:N. Assim como a anotação @OneToMany anota uma lista, com a diferença de que um relacionamento N:N não tem entidades filhas, cada entidade do relacionamento possui uma lista da outra entidade;

• @JoinTable(name, joinColumns, inverseJoinColumns) – utilizada para anotar uma lista que já esteja anotada com @ManyToMany. Especifica qual tabela será utilizada para recuperar os dados do relacionamento N:N. Seguem detalhes dos parâmetros:

o name – nome da tabela que representa o relacionamento;

o joinColumns – recebe um vetor de @JoinColumn(name) que representa as colunas da classe atual que serão utilizadas no JOIN;

o inverseJoinColumns – recebe um vetor de @JoinColumn(name) que representa as colunas da outra classe do relacionamento que serão utilizadas no JOIN.

Existem muitas outras anotações disponíveis, porém essas são as mais utilizadas em aplicativos de pequeno e médio porte, e serão suficientes para o mapeamento da aplicação descrita a seguir.

Locadora de DVDs. Neste contexto, o modelo de dados utilizado é o representado pela **Figura 1** e o SQL utilizado para criar as tabelas é o descrito na **Listagem 1**.

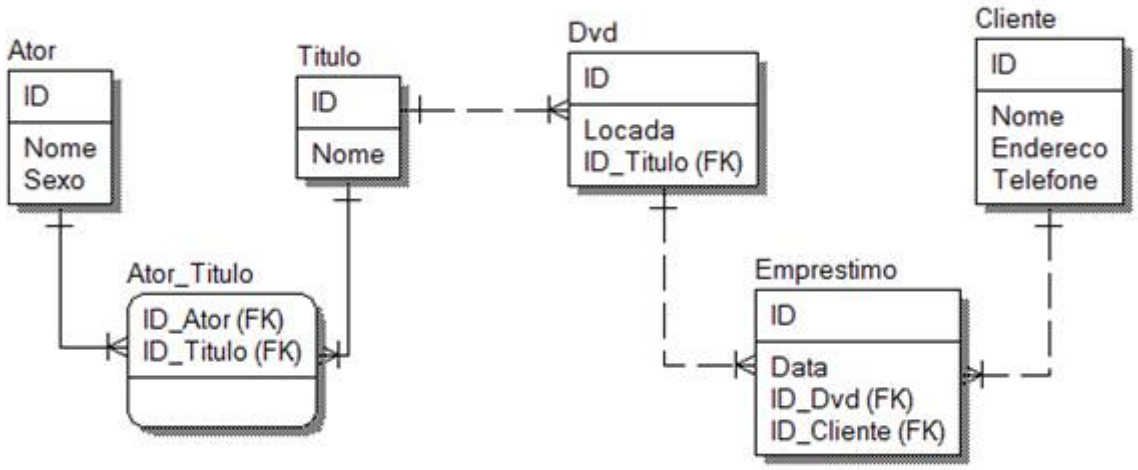

Figura 1. Modelo entidade-relacionamento da locadora.

Listagem 1. Código SQL para criação das tabelas.

```
1  CREATE TABLE ator (
2    ID bigint(20) NOT NULL auto_increment,
3    NOME varchar(200) NOT NULL,
4    SEXO varchar(1) NOT NULL,
5    PRIMARY KEY (ID)
6  );
7
8  CREATE TABLE ator_titulo (
9    ID_TITULO bigint(20) NOT NULL,
10   ID_ATOR bigint(20) NOT NULL,
11   PRIMARY KEY (ID_TITULO, ID_ATOR)
12 );
13
14 CREATE TABLE cliente (
15   ID bigint(20) NOT NULL auto_increment,
16   NOME varchar(50) NOT NULL,
17   ENDERECO varchar(200) NOT NULL,
```

```
  ID bigint(20) NOT NULL auto_increment,
  ID_TITULO bigint(20) NOT NULL,
  PRIMARY KEY (ID)
);

CREATE TABLE emprestimo (
  ID bigint(20) NOT NULL auto_increment,
  `DATA` date NOT NULL,
  ID_CLIENTE bigint(20) NOT NULL,
  ID_DVD bigint(20) NOT NULL,
  PRIMARY KEY (ID)
);

CREATE TABLE titulo (
  ID bigint(20) NOT NULL auto_increment,
  NOME varchar(200) NOT NULL,
  PRIMARY KEY (ID)
);

ALTER TABLE ator_titulo ADD CONSTRAINT ATOR_TITULOFK_TITULO FOREIGN

ALTER TABLE ator_titulo ADD CONSTRAINT ATOR_TITULOFK_ATOR FOREIGN KE

ALTER TABLE dvd ADD CONSTRAINT DVDFK_TITULO FOREIGN KEY (ID_TITULO)

ALTER TABLE emprestimo ADD CONSTRAINT EMPRESTIMOFK_DVD FOREIGN KEY (

ALTER TABLE emprestimo ADD CONSTRAINT EMPRESTIMOFK_CLIENTE FOREIGN K
```

A locadora terá em seu acervo vários DVDs, onde cada registro na tabela Dvd significa uma mídia física para um título. O título representa um filme, show, documentário e outros que podem ser feitos por um ou vários atores. A locação é restrita a um Dvd por Cliente.

referente ao banco de dados que será utilizado (MySQL neste caso), ou seja, deve possuir as bibliotecas necessárias no classpath. Para o desenvolvimento deste projeto adotaremos a IDE NetBeans 6.9.1.

Com um novo projeto criado, basta clicar com o botão direito na pasta *Bibliotecas*, selecionar a opção *Adicionar biblioteca...* e selecionar as opções: Hibernate JPA e MySQL JDBC Driver, de acordo com a **Figura 2**.

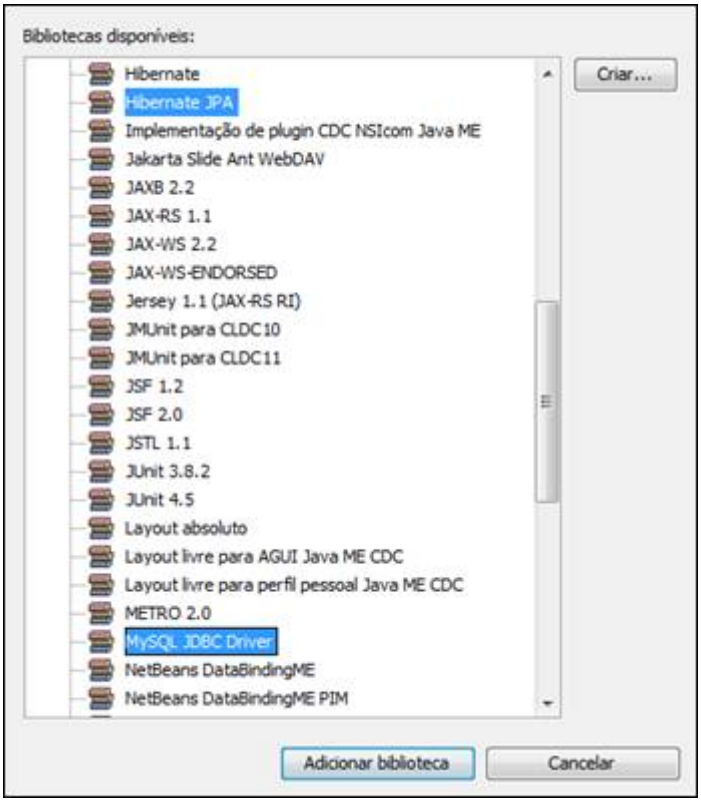

Figura 2. Bibliotecas necessárias para o projeto.

> Esta é uma funcionalidade do NetBeans. Caso você esteja utilizando outra IDE, basta realizar o download de tais bibliotecas em seus respectivos sites e adicioná-las ao classpath.

torna mais fácil a visualização e o entendimento do escopo da nossa aplicação.

Dessa forma, o sexo será representado por um enumerador devido ao fato de ter apenas duas opções possíveis. O relacionamento N:N entre Ator e Titulo não possui uma classe intermediária como no modelo ER (Entidade-Relacionamento), pois o Hibernate faz esta associação internamente. Quanto aos relacionamentos 1:N, a entidade cuja cardinalidade é 1 possui uma lista da entidade cuja cardinalidade é N (Titulo possui uma lista de Dvds) e a situação inversa é que a entidade cuja cardinalidade é N possui uma referência à entidade cuja cardinalidade é 1 (Dvd possui apenas um Titulo). A **Figura 3** representa o diagrama de classes que será utilizado para esta aplicação.

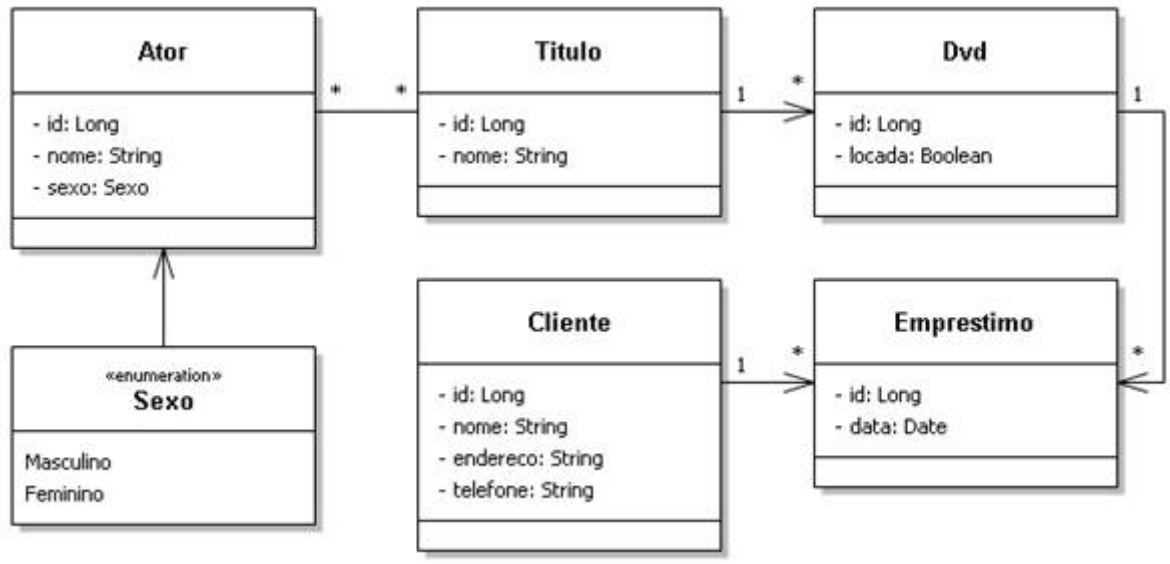

Figura 3. Diagrama de classes para o aplicativo da Locadora de dvds.

Como já informado, todas as classes que representam uma tabela do banco de dados deverão ser anotadas com @Entity. Se a aplicação for Web, recomenda-se que as entidades implementem a interface Serializable para evitar possíveis erros durante a transmissão delas através da Internet.

aconselhado, pois atributos do tipo Integer podem possuir o valor null, já os do tipo int não (quando não possuem valor o padrão é 0). Esta prática é utilizada porque quando o Hibernate recebe uma entidade com a chave primária nula (com o valor null), significa que é um novo registro no banco, já no caso de receber uma chave primária com o valor 0, é assumido que este registro já exista e 0 seja seu identificador.

Uma exigência do Hibernate é que todo atributo que for mapeado possua métodos getters e setters seguindo o padrão de nomenclatura CamelCase, pois é através deles que o Hibernate consegue atribuir e recuperar os valores dos atributos.

Como dito anteriormente, a entidade que representa o Sexo foi projetada (**Figura 3**) para ser um enumerador que possui apenas duas opções (Masculino e Feminino). O enumerador Sexo é descrito na **Listagem 2**.

A chave primária da classe Ator (descrita pela **Listagem 3**) é representada pelo atributo id, que tem seu valor gerado automaticamente pelo gerador de sequência SEQ_ATOR (definido no banco de dados). O atributo sexo da classe Ator é representado pelo enumerador Sexo (descrito na **Listagem 2**) e está anotado com @Enumerated(EnumType.STRING), o que significa que caso o ator seja do sexo masculino a coluna Sexo da tabela Ator irá receber o valor M, e caso seja feminino, o valor F.

O atributo tituloList da classe Ator (**Listagem 3**) representa um relacionamento N:N, pois está anotado com @ManyToMany. Deve-se tomar cuidado com os parâmetros joinColumns e inverseJoinColumns: o primeiro recebe a coluna **ID_ATOR**, que existe na tabela local (Ator) e o segundo recebe a coluna

representado pelo atributo titulo. Todas as vezes que um Dvd for recuperado pela aplicação será necessário mostrar o nome do título que ele possui (lembrando que a entidade Dvd representa uma mídia física e o Titulo representa um filme, seriado, documentário, entre outros), por isso o parâmetro fetch da anotação @ManyToOne está definido com FetchType.EAGER. Assim, quando um Dvd for carregado, o Titulo também será.

A classe Titulo (**Listagem 5**) possui todos os Dvds que a referenciam através do atributo dvdList. Para evitar que todos os Dvds sejam carregados juntamente com o objeto da classe Titulo, o parâmetro fetch da anotação @OneToMany recebe FetchType.LAZY, ou seja, quando a aplicação carregar um Titulo do banco de dados os Dvds não serão carregados. Quando a aplicação fizer o primeiro acesso ao atributo dvdList (através do método getDvdList()), o Hibernate realizará uma consulta no banco para recuperar tais registros.

O parâmetro mappedBy da anotação @OneToMany (que anota o atributo dvdList da classe Titulo) recebe a String "**titulo**", indicando que na classe Dvd existe um atributo do tipo Titulo cujo nome é titulo. Tal parâmetro indica ao Hibernate que para recuperar a lista de Dvds ele deve fazer um JOIN entre as duas tabelas (Titulo e Dvd) utilizando as colunas especificadas na anotação @JoinColumn do atributo titulo da classe Dvd.

A classe Emprestimo (**Listagem 6**) representa a locação de Dvds para Clientes (**Listagem 7**). Como no escopo desta aplicação a locação será de **um** Dvd para **um** Cliente, a classe Emprestimo tem apenas uma referência para Dvd e uma referência para Cliente, ambas anotadas com @ManyToOne, pois o Cliente pode fazer vários empréstimos e um Dvd pode ser emprestado várias vezes. Como a

> O gerador de sequência é definido no banco de dados, mas para facilitar a vida do desenvolvedor, o Hibernate pode criar o gerador automaticamente, caso ele não exista. Para isso basta atribuir o valor **update** para a propriedade **hibernate.hbm2ddl.auto**, explicada no tópico **Configuração do Hibernate** e demonstrada na **Listagem 8**.

Listagem 2. Enumeração que representa o Sexo.

```
1  public enum Sexo {
2    M("Masculino"), F("Feminino");
3
4    private Sexo(String sexo){
5
6    }
7  }
```

Listagem 3. Entidade que representa os Atores.

```
1  @Entity
2  @Table(name = "ATOR")
3  @SequenceGenerator(name = "SEQ_ATOR", sequenceName = "SEQ_ATOR")
4  public class Ator implements Serializable {
5
6    @GeneratedValue(strategy = GenerationType.AUTO, generator = "SEQ_A
7    @Id
8    @Column(name = "ID", nullable = false)
9    private Long id;
10
11   @Column(name = "NOME", length = 200, nullable = false)
12   private String nome;
13
```


```
    joinColumns = @JoinColumn(name = "ID_ATOR"),
    inverseJoinColumns = @JoinColumn(name = "ID_TITULO")
  )
  private List<Titulo> tituloList;

  //getters e setters
}
```

Listagem 4. Entidade que representa um DVD.

```
@Entity
@Table(name = "DVD")
@SequenceGenerator(name = "SEQ_DVD", sequenceName = "SEQ_DVD")
public class Dvd implements Serializable {

  @GeneratedValue(strategy = GenerationType.AUTO, generator = "SEQ_D
  @Id
  @Column(name = "ID", nullable = false)
  private Long id;

  @ManyToOne(fetch=FetchType.EAGER)
  @JoinColumn(name = "ID_TITULO", referencedColumnName = "ID", nulla
  private Titulo titulo;

  @Column(name = "LOCADA", nullable = false)
  private Boolean locada;

  //getters e setters
}
```

Listagem 5. Entidade que representa um Título (filme, seriado, show, documentário, entre outros).

```
6    @GeneratedValue(strategy = GenerationType.AUTO, generator = "SEQ_T
7    @Id
8    @Column(name = "ID", nullable = false)
9    private Long id;
10
11   @Column(name = "NOME", length = 200, nullable = false)
12   private String nome;
13
14   @OneToMany(fetch = FetchType.LAZY, mappedBy = "titulo")
15   private List<Dvd> dvdList;
16
17    @ManyToMany
18   @JoinTable(name = "ATOR_TITULO",
19     joinColumns = @JoinColumn(name = "ID_TITULO"),
20     inverseJoinColumns = @JoinColumn(name = "ID_ATOR"))
21   private List<Ator> atorList;
22
23   //getters e setters
24 }
```

Listagem 6. Entidade que representa a locação de um Dvd para um Cliente

```
1  @Entity
2  @Table(name = "EMPRESTIMO")
3  @SequenceGenerator(name = "SEQ_EMPRESTIMO", sequenceName = "SEQ_EMPR
4  public class Emprestimo implements Serializable {
5
6    @GeneratedValue(strategy = GenerationType.AUTO, generator = "SEQ_E
7    @Id
8    @Column(name = "ID", nullable = false)
9    private Long id;
10
11   @Column(name = "DATA", nullable = false)
12   @Temporal(TemporalType.DATE)
13   private Date data;
```

```
20    @JoinColumn(name = "ID_CLIENTE", referencedColumnName = "ID", nul
21    private Cliente cliente;
22
23    //getters e setters
24  }
```

A classe Cliente, descrita pela **Listagem 7**, não tem nenhuma funcionalidade diferente das mostradas até aqui. Possui um gerador de sequência, um identificador, nome e a lista de empréstimos que já realizou. Como não é toda hora que precisaremos mostrar os empréstimos que o cliente fez, o parâmetro fetch da anotação @OneToMany do atributo emprestimoList recebeu o valor FetchType.LAZY.

Listagem 7. Entidade que representa um Cliente.

```
1   @Entity
2   @Table(name = "CLIENTE")
3   @SequenceGenerator(name = "SEQ_CLIENTE", sequenceName = "SEQ_CLIENTE
4   public class Cliente implements Serializable {
5
6     @GeneratedValue(strategy = GenerationType.AUTO, generator = "SEQ_C
7     @Id
8     @Column(name = "ID", nullable = false)
9     private Long id;
10
11    @Column(name = "NOME", length = 200, nullable = false)
12    private String nome;
13
14    @OneToMany(fetch = FetchType.LAZY, mappedBy = "cliente")
15    private List<Emprestimo> emprestimoList;
16
17    //getters e setters
18  }
```

Configuração do Hibernate

O fato de as classes estarem mapeadas não é o suficiente para que a aplicação funcione corretamente. O Hibernate deve saber quais classes estão mapeadas, qual banco acessar, qual o usuário e a senha do banco, enfim, ele deve ser configurado.

Para tal, deve-se criar um arquivo XML de configuração. Deste modo, clique com o botão direito no projeto, depois em *Novo* e selecione a opção *Outro*. Na próxima janela selecione a pasta *XML*, *Documento XML*, dê o nome "hibernate.cfg" e escolha a pasta *src* para criá-lo.

A **Listagem 8** apresenta a estrutura básica de um arquivo de configuração do Hibernate.

Listagem 8. Estrutura básica do XML de configuração do Hibernate.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Cor
3  <hibernate-configuration>
4    <session-factory>
5
6    </session-factory>
7  </hibernate-configuration>
```

A configuração é feita informando algumas propriedades e seus respectivos valores. Algumas das propriedades mais utilizadas são:

• hibernate.connection.driver_class – nome da classe do driver JDBC;

• hibernate.connection.username – nome do usuário do banco de dados pelo qual o Hibernate irá se conectar;

• hibernate.connection.password – senha do usuário do banco de dados;

• hibernate.dialect – dialeto específico do banco de dados utilizado. Cada banco possui suas particularidades quanto ao SQL. O Hibernate utiliza o dialeto para saber quais comandos pode utilizar no SQL que irá executar;

• hibernate.show_sql – caso possua o valor true, todas as queries que o Hibernate executar no banco de dados serão impressas no console;

• hibernate.format_sql – caso possua o valor true, as queries não serão impressas em apenas uma linha, serão formatadas e identadas em várias linhas para que sua leitura fique mais fácil;

• hibernate.hbm2ddl.auto – permite que o próprio Hibernate gere os scripts DDL (*Data Definition Language*) para a criação e atualização das tabelas no banco de dados. Atribuindo o valor "**create**" para esta propriedade o Hibernate irá gerar e executar os scripts de acordo com o mapeamento, ou seja, as tabelas serão criadas a partir dos metadados informados nas classes. Caso ocorra uma alteração em um atributo que deva ser refletida no banco de dados basta atribuir o valor "**update**" que o Hibernate irá gerar e executar o script de alteração sozinho.

> O comportamento da propriedade **hibernate.hbm2ddl.auto** varia de acordo com o banco de dados. Ex: para criar as tabelas no MySQL deve-se utilizar o valor **create**, já no Oracle deve-se

utilizada a tag <mapping> e para informar as propriedades é utilizada a tag <property>. A **Listagem 9** mostra o arquivo de configuração com todas as propriedades e mapeamentos informados.

Listagem 9. XML de configuração do Hibernate.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Cor
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.jdb
    <property name="hibernate.connection.url">jdbc:mysql://localhost
    <property name="hibernate.connection.username">usuario</property
    <property name="hibernate.connection.password">senha</property>

    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDi

    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.format_sql">true</property>

    <property name="hibernate.hbm2ddl.auto">false</property>

    <mapping class="entity.Ator"/>
    <mapping class="entity.Cliente"/>
    <mapping class="entity.Dvd"/>
    <mapping class="entity.Emprestimo"/>
    <mapping class="entity.Titulo"/>
  </session-factory>
</hibernate-configuration>
```

> Lembre-se de que o nome das classes dentro das tags **<mapping>** deve ser o nome qualificado, ou seja, com o pacote.

Após terminar a configuração, basta instanciar os objetos necessários para a inicialização do Hibernate para poder utilizar os benefícios que ele provê. A **Listagem 10** apresenta uma classe cujo método **main()** possui os objetos necessários para isso.

O objeto do tipo Configuration precisa ser instanciado como um AnnotationConfiguration porque o mapeamento que queremos carregar é todo feito com anotações. Deve ser informada a localização do arquivo de configuração para o objeto cfg, através do método configure(String), para que ele recupere o arquivo e se comporte de acordo com as propriedades que setamos.

Depois da chamada ao método configure(), é criado um objeto SessionFactory, o qual é responsável por carregar todos os metadados em memória para que as regras definidas possam ser seguidas. Somente um objeto SessionFactory deve ser instanciado por aplicação, pois é na sua inicialização que todos os metadados são validados e o Hibernate cria as estruturas de dados necessárias para realizar o mapeamento objeto-relacional.

A principal função do objeto SessionFactory é criar objetos do tipo Session, que são os responsáveis pela persistência e recuperação dos objetos mapeados no banco de dados.

Os métodos referentes às operações CRUD que o objeto session proporciona são:

• save() – cria um novo registro na tabela referente ao objeto;

• update() – atualiza o registro na tabela para ficar com os dados que estão no objeto;

Listagem 10. Executando a aplicação.

```
1  public class Main {
2
3    public static void main(String[] args) {
4      Configuration cfg = new AnnotationConfiguration();
5      cfg.configure("hibernate.cfg.xml");
6
7      SessionFactory sessionFactory = cfg.buildSessionFactory();
8
9      Session session = sessionFactory.openSession();
10
11     //Operações
12
13     session.close();
14   }
15 }
```

Como o arquivo de configuração foi criado na pasta raiz da aplicação, foi enviado apenas o seu nome para o método **configure()** do objeto **cfg**. Caso o arquivo estivesse em subpastas, deveria ser informado o caminho completo a partir da pasta raiz, por exemplo: /conf/hibernate.cgf.xml.

Listagem 11. Exemplo de uso dos métodos básicos do objeto Session.

```
1  public void criaAtor(Session session, String nome, Sexo sexo) {
2    Ator ator = new Ator();
3    ator.setNome(nome);
4    ator.setSexo(sexo);
```


```
    dvd.setTitulo(titulo);
    session.save(dvd); // salva o dvd
    titulo.getDvdList().add(dvd);
  }

  public void vendaDvd(Session session, Dvd dvd) {
    session.delete(dvd); // remove o dvd
  }

  public void locaDvd(Session session, Dvd dvd) {
    dvd.setLocada(true);
    session.update(dvd); // atualiza o dvd
  }

  public Cliente buscaClientePorId(Session session, Long id) {
    Cliente cliente = (Cliente) session.get(Cliente.class, id); // rec
    return cliente;
  }
```

Conclusão

Este artigo demonstrou como realizar o mapeamento objeto-relacional através das anotações providas pela especificação JPA e implementadas pelo framework Hibernate. Após anotar sua classe e seus atributos basta utilizar o objeto Session para realizar a persistência e a recuperação dos dados.

Com toda essa facilidade para persistir e recuperar os objetos do banco de dados, o desenvolvedor passa menos tempo escrevendo códigos para tal finalidade. Por consequência, ele consegue direcionar seus esforços para as regras de negócio da aplicação, o que acarreta em um prazo menor para a conclusão do projeto.

http://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/

Guia de referência do Hibernate Annotations.

http://www.hibernate.org/downloads.html

Download do Hibernate.

http://dev.mysql.com/downloads/connector/j/

Download do driver JDBC do MySQL.

Tecnologias:

Hibernate Java JPA MySQL UML

Por **Felipe**

Em 2011