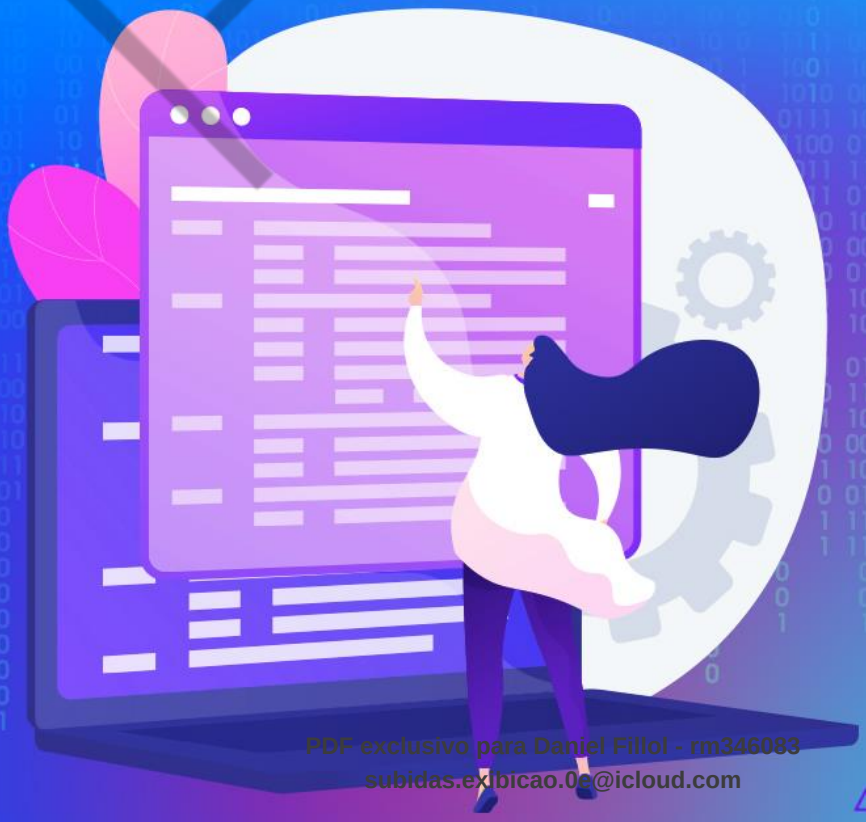
FIAP ON

FUNDAMENTOS JAVA

PERSISTÊNCIA OO *(SETUP)*

JOSÉ YOSHIRIRO



7

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

7 PERSISTÊNCIA OO (SETUP)

7.1 ORM – Mapeamento Objeto-Relacional

7.1.1 Introdução

O **“Mapeamento Objeto-Relacional”** (do original em inglês Object-Relational Mapping) é uma técnica que permite “espelhar” as tabelas de um banco de dados relacional em classes de uma linguagem de programação que seja orientada a objetos, como é o caso de Java. Esse processo pode ser feito a partir de tabelas novas, criadas junto com o projeto Java, ou de tabelas legadas, isto é, existentes e usadas até por sistemas antigos.

Essa técnica facilita muito o trabalho em projetos com muitas tabelas, pois torna mais fáceis as manutenções evolutivas e corretivas. Isso se deve ao fato de ficarem explícitos a quantidade das tabelas, seus campos, suas Chaves Primárias e os relacionamentos. Por fim, essa abordagem facilita a criação dos testes unitários automatizados.

7.1.2 Nomes de tabelas e campos x Nomes de classes e atributos

Durante o processo de ORM, é importante respeitar as diferenças das convenções dos nomes entre o mundo dos bancos de dados relacionais e o mundo das linguagens de programação orientadas a objeto, como é o caso do Java.

Bancos de dados: snake_case

Nos bancos de dados relacionais, é muito comum os nomes das tabelas e de seus campos seguirem o padrão ***snake_case***, ou seja, todas as letras minúsculas e palavras separadas por *underline* (“_”) em vez de espaço em branco. Exemplos: tabela “**tipo_estabelecimento**”. Campo “**id_tipo_estabelecimento**”.

Java: camelCase

Na linguagem Java, o padrão é o **camelCase**. Aqui, a primeira letra pode ou não ser maiúscula, dependendo do que estamos nomeando. Não há nenhum

caractere separador entre as palavras, mas cada palavra nova inicia com letra maiúscula. No caso das **classes**, **interfaces** e **enums,** a primeira letra é maiúscula e, para todos os demais, é minúscula. Exemplos: classe **TipoEstabelecimento**. Atributo **idTipoEstabelecimento**.

Outro ponto é que o Mapeamento Objeto-Relacional não deve ser uma mera tradução **snake_case** para **camelCase**. É comum algumas tabelas terem algum prefixo no nome (“tb” ou “tab” ou “tbl” etc.) para deixar claro que são tabelas. Algo parecido ocorre com campos: muitas vezes, têm um prefixo que indica o tipo de dado (“fl” para flag, “dt” para data, “ds” para descrição etc.), além de possuírem um sufixo, que normalmente é o próprio nome da tabela. Assim, um campo de “data de criação de um tipo de tarifa” numa tabela “tipo_tarifa” poderia ser algo como **“dt_criacao_estabelecimento”**.

Esses prefixos e sufixos ocorrem principalmente quando os nomes das tabelas e campos foram definidos por profissionais chamados **DA** (*Data Administrador*) ou **DBA** (*Database Administrador*). Eles preferem usar esses prefixos e sufixos porque facilitam suas tarefas de administração e segurança dos bancos de dados. **Importante:** costumamos ignorar esses prefixos e sufixos nas classes Java e seus atributos quando fazemos o ORM, pois os fatores que motivam seu uso nos bancos de dados não existem em projetos Java.

Existe ainda a possibilidade das tabelas e campos terem nomes totalmente diferentes de suas finalidades, aparentemente até aleatórios. Isso costuma ser feito em sistemas cujos dados possuem alto grau de confidencialidade, como dados de investidores milionários em um sistema de custódia, por exemplo.

No quadro, temos um exemplo de como os nomes de uma tabela e seus campos poderiam ficar mapeados numa classe Java.

	Tabela	Classe
Nome da tabela	tipo_estabelecimento	TipoEstabelecimento
Campus	id_tipo_estabelecimento	id
	nome_estabalecimento	nome
	dt_inauguracao	dataInauguracao

Quadro 7.1 – Nome da tabela e seus campos numa classe Java
Fonte: Elaborado pelo autor (2017)

7.1.3 Tipos de campos x Tipos de atributos

Outro ponto muito importante no processo de ORM é saber que há uma relação entre os tipos de campos usados pelos bancos de dados relacionais e os tipos em Java. Às vezes, um mesmo tipo de campo no banco pode ser representado por diferentes classes ou tipos primitivos. O quadro mostra a relação dos principais tipos de campos com seus possíveis tipos em Java.

Tipo de campo	Classes ou tipos primitivos Java
VARCHAR	String
INT	Integer, int, Long ou lng
NUMBER	Integer, int, Long, long, Double, double ou BigDecimal
DATE	Date ou Calendar
TIMESTAMP	Date ou Calendar
BYTEA	Byte[ ] ou byte[ ]
BLOB	Byte[ ] ou byte[ ]
CLOB	String

Quadro 7.2 – Tipos de campos x Classes Java
Fonte: Elaborado pelo autor (2017)

7.2 ORM com JPA e Hibernate

7.2.1 Introdução

Em Java, as tecnologias mais usadas para esse mapeamento são: o **JPA** (Java Persistence API), que é apenas uma especificação, e o **Hibernate**, framework que implementa essa especificação. A seguir, veremos os primeiros detalhes do funcionamento deles.

7.2.2 Configurando JPA e Hibernate num projeto Java

Para usar JPA e Hibernate num projeto Java, basta adicionar uma dependência no projeto, ou seja, no arquivo **pom.xml**. Exemplo no código-fonte:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>RELEASE</version>
</dependency>
<!-- Dependência do driver do SGBD -->
```

Código-fonte 7.1 – Dependência necessária para uso de JPA e Hibernate
Fonte: Elaborado pelo autor (2017)

A dependência do driver de algum banco de dados também deve ser incluída. O próximo código-fonte tem alguns exemplos de dependências de drivers de alguns dos principais bancos do mercado.

```
<!-- MySQL e MariaDB -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>RELEASE</version>
</dependency>

<!-- PostgreSQL -->
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>RELEASE</version>
</dependency>
```

```
      <!-- SQL Server -->
      <dependency>
          <groupId>com.microsoft.sqlserver</groupId>
          <artifactId>mssql-jdbc</artifactId>
          <version>RELEASE</version>
      </dependency>
```

Código-fonte 7.2 – Dependências dos drivers de alguns dos principais bancos de dados
Fonte: Elaborada pelo autor (2017)

Os servidores de banco de dados Oracle e DB2 não possuem drivers atuais em repositórios públicos *Maven*, sendo necessário fazer o download e a configuração manual deles no projeto.

Para configurar o driver para Oracle, você deve fazer o download do arquivo do driver (**ojdbc8.jar**) a partir do site da Oracle, em <**http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html>**. O arquivo pode ficar em qualquer diretório do seu computador, mas não é recomendado que fique no diretório do seu projeto. Isso porque é um arquivo grande e não precisa estar junto do código-fonte de seu projeto. O código-fonte *Como configurar dependência do arquivo local no pom.xml* possui um exemplo de como configurar a dependência do arquivo local no pom.xml.

```
<dependency>
      <groupId>oracle</groupId>
      <artifactId>jdbc-driver</artifactId>
      <version>12</version>
      <scope>system</scope>
      <systemPath>/home/seu_usuario/Downloads/ojdbc8.jar</sys
temPath>
</dependency>
```

Código-fonte 7.3 – Como configurar dependência do arquivo local no pom.xml
Fonte: Elaborada pelo autor (2017)

As tags **groupId**, **artifactId** e **version** poderiam ter, na verdade, qualquer valor (contanto que não entrem em conflito com nenhum nome de repositório remoto Maven). Todavia, os valores usados no exemplo correspondem aos valores ideais. A tag **scope** deve ser, necessariamente, **system**. A tag **systemPath** deve conter o endereço completo do arquivo **ojdbc8.jar**.

Se a dependência foi configurada corretamente e sua IDE conseguir fazer o download das dependências, poderá importar em suas classes, por exemplo: **org.hibernate.annotations.Formula** (classe exclusiva do framework Hibernate) e a **javax.persistence.EntityManager** (do JPA).

7.2.3 Anotações JPA e Hibernate

Para implementar o ORM com JPA e Hibernate, a técnica mais usada atualmente é incluir **Anotações** (***Annotations***) nas classes que vão "espelhar" as tabelas do banco de dados. Algumas anotações ficam sobre a classe e outras sobre os atributos. É possível fazer o mapeamento com arquivos XML, porém, é uma técnica muito pouco utilizada atualmente e não será abordada.

Quando uma classe Java é mapeada para uma tabela, seja com o uso de anotações, seja com XML, podemos chamar essa classe de **Entidade**. O próximo código-fonte contém um exemplo de uma entidade na qual foram usadas anotações JPA e Hibernate.

```
      @Entity
      @Table(name = "tbl_estabelecimento")
      public class Estabelecimento {

          @Id
          @GeneratedValue(strategy = GenerationType.IDENTITY)
          @Column(name = "id_estabelecimento")
          private Integer id;

          @Column(name = "nome_estabelecimento", length = 50)
          private String nome;

          @CreationTimestamp
          @Temporal(TemporalType.TIMESTAMP)
          @Column(name = "dh_criacao")
          private Calendar dataCriacao;

          @Formula("(select  avg(a.nota)+1  from  avaliacao  a
where a.id_estabelecimento = id_estabelecimento)")
          private Double mediaAvaliacoes;

          // construtores, getters e setters

      }
```

Código-fonte 7.4 – Dependência necessária para uso de JPA e Hibernate
Fonte: Elaborado pelo autor (2017)

A seguir, as principais anotações para ORM do JPA e Hibernate serão explicadas. Algumas delas terão mais explicações e exemplos no próximo capítulo.

7.2.3.1 @Entity (JPA)

Nome completo: *javax.persistence.Entity*

Objetivo: indicar que a classe será usada para mapear uma tabela do banco de dados.

Onde deve ser incluída: sobre a classe somente. Obrigatória.

Atributos: não possui.

Como atuou no exemplo: indicou para o JPA que a classe é uma Entidade ORM, ou seja, que serve para mapear uma determinada tabela do banco de dados.

7.2.3.2 @Table (JPA)

Nome completo: *javax.persistence.Table*

Objetivo: configura as informações da tabela que está sendo “espelhada” na classe. Se essas anotações forem omitidas, o JPA vai procurar uma tabela com o EXATO nome da classe no banco. O interessante desse atributo é que a tabela pode ter um nome diferente de sua classe mapeada. Isso ajuda a resolver a questão de convenções dos nomes abordados neste capítulo.

Onde deve ser incluída: sobre a classe somente. Opcional.

Atributos:

- **name** (obrigatório): nome da tabela no banco de dados.
- **catalog** (opcional): catálogo da tabela no banco de dados.
- **schema** (opcional): esquema da tabela no banco de dados.

- **indexes** (opcional): vetor de objetos que mapeiam índices no banco de dados.
- **uniqueConstraints** (opcional): vetor de objetos que mapeiam as restrições de valor único no banco de dados.

Como atuou no exemplo: indicou o nome da tabela no banco de dados como sendo “tbl_estabelecimento”.

7.2.3.3 @Id (JPA)

Nome completo: *javax.persistence.Id*

Objetivo: indica qual atributo da classe será mapeado para a **Chave Primária** da tabela.

Onde deve ser incluída: sobre um atributo*. Obrigatório em pelo menos 1 (um) atributo.

Atributos: não possui.

Como atuou no exemplo: indicou que o atributo **id** está mapeado para o campo de **Chave Primária** na tabela.

7.2.3.4 @GeneratedValue (JPA)

Nome completo: *javax.persistence.GeneratedValue*

Objetivo: configura a forma de preenchimento automático do valor do campo da Chave Primária. Se não for usada, o programa deve configurar “manualmente” o valor do atributo da Chave Primária.

Onde deve ser incluída: Sobre um atributo*. Opcional.

Atributos:

- **strategy** (opcional): mostra a estratégia para a geração do valor do atributo. Os tipos de estratégias são os valores da *enum* **javax.persistence.GenerationType**:

- **AUTO**: aponta que a estratégia-padrão de preenchimento automático do banco de dados configurado será utilizada. Em alguns bancos, a Chave Primária "cresce sozinha", ou seja, possui um valor com ***autoincremento***. Para outros, o JPA pegará o maior valor atualmente na tabela e usará mais 1. Se o atributo **strategy** for omitido, esta estratégia é a que será usada.
- **IDENTITY**: em alguns bancos, a Chave Primária "cresce sozinha", ou seja, possui um valor com ***autoincremento***. O IDENTITY indica que o JPA irá gerar uma instrução de *insert* apropriada para o banco de dados configurado para que use esse recurso no momento da criação de um novo registro.
- **SEQUENCE**: alguns bancos de dados não possuem o recurso de **autoincremento** de valor. Assim, uma forma de fazer o valor "crescer" de forma consistente é consultar o novo valor de uma ***sequence*** no banco de dados. Essa opção indica e configura o uso desse recurso para a obtenção do valor que será usado na Chave Primária.
- **TABLE**: opção muito parecida com a **SEQUENCE**. A diferença é que com ela se indica uma ***tabela*** e não uma ***sequence*** de onde se pega o novo valor que será usado na Chave Primária.

- **generator** (opcional, porém, obrigatório para **SEQUENCE**): caso tenha usado o **SEQUENCE** no **strategy**, nesse atributo deve indicar o mesmo valor que usou no **name** da anotação **@SequenceGenerator** (descrita a seguir). Caso tenha usado o **TABLE** no **strategy**, nesse atributo deve apontar o mesmo valor que usou no **name** da anotação **@TableGenerator** (descrita a seguir).

Como atuou no exemplo: indicou que o campo **id** terá seu valor preenchido automaticamente com uso da estratégia **IDENTITY**.

7.2.3.5 @SequenceGenerator (JPA)

Nome completo: *javax.persistence.SequenceGenerator*

Objetivo: configura o acesso a uma ***sequence*** do banco para ser usada na Chave Primária.

Onde deve ser incluída: sobre um atributo*. Opcional. Obrigatória apenas se for usado **SEQUENCE** no atributo **strategyType** na anotação **@GeneratedValue**.

Atributos:

- **name** (obrigatório): indica o nome da ***sequence*** na classe mapeada. O valor desse atributo é que deve ser usado no atributo **generator** da anotação **@GeneratedValue**.
- **sequenceName** (obrigatório): aponta o nome da ***sequence*** no banco de dados.
- **schema** (opcional): nome do ***esquema*** do banco onde está a ***sequence***. Se omitido, o JPA irá considerar que está no mesmo que a tabela.
- **catalog** (opcional): nome do ***catálogo*** do banco onde está a ***sequence***. Se omitido, o JPA irá considerar que está no mesmo que a tabela.

7.2.3.6 @TableGenerator (JPA)

Nome completo: javax.persistence.TableGenerator

Objetivo: configura o acesso a uma **tabela** do banco para ser usada na Chave Primária.

Onde deve ser incluída: sobre um atributo*. Opcional. Obrigatória apenas se for usado **TABLE** no atributo **strategyType** na anotação **@GeneratedValue**.

Atributos:

- **name** (obrigatório): indica o nome da ***tabela*** na classe mapeada. O valor desse atributo é que deve ser usado no atributo **generator** da anotação **@GeneratedValue**.
- **table** (obrigatório): aponta o nome da ***tabela*** no banco de dados.
- **valueColumnName** (obrigatório): indica o nome do ***campo*** da tabela no banco de dados.

- **schema** (opcional): nome do ***esquema*** do banco onde está a ***tabela***. Se omitido, o JPA irá considerar que está no mesmo que a tabela.
- **catalog** (opcional): nome do ***catálogo*** do banco onde está a ***tabela***. Se omitido, o JPA irá considerar que está no mesmo que a tabela.

7.2.3.7 @Column (JPA)

Nome completo: *javax.persistence.Column*

Objetivo: mapeia uma coluna da tabela junto a um atributo na classe.

Onde deve ser incluída: sobre um atributo*. Opcional.

Atributos:

- **name** (opcional): indica qual o nome do campo na tabela. Se omitido, o JPA entenderá que o campo possui exatamente o mesmo nome do atributo. O interessante desse atributo é que um campo pode ter um nome na tabela diferente de seu atributo mapeado na classe. Isso ajuda a resolver a questão das convenções dos nomes abordados neste capítulo.
- **length** (opcional): aponta o tamanho do campo na tabela. Por exemplo, para um campo ***varchar,*** esse campo mostraria a quantidade dos caracteres que comporta.
- **precision** (opcional): indica a precisão do campo. Esse atributo se aplica a campos numéricos.
- **scale** (opcional): mostra a escala do campo. Esse atributo se aplica a campos numéricos.
- **nullabe** (opcional): aponta se o campo é obrigatório (**false**) ou se é possível criar/atualizar um valor de um registro, deixando-o em vazio (**true**) .
- **unique** (opcional): indica se o campo deve possuir valor único na tabela, ou seja, se é um campo com a restrição ***unique*** na tabela.

- **insertable** (opcional): mostra se o campo pode ter valor no momento da criação de um registro. Se esse atributo for **false,** um eventual valor do atributo da classe será ignorado no momento da criação de um registro.
- **updatable** (opcional): aponta se o campo pode ter valor no momento da atualização de um registro. Se esse atributo for **false,** um eventual valor do atributo da classe será ignorado no momento da atualização de um registro.

Como atuou no exemplo: indicou os vários atributos que estão mapeados para campos da tabela. Sobre o atributo **id,** mostrou que seu respectivo campo na tabela era **id_estabelecimento**; sobre **nome,** apontou que seu respectivo campo na tabela era **nome_estabelecimento**; sobre **dataCriacao,** indicou que seu respectivo campo na tabela era **dh_criacao**;

7.2.3.8 @Temporal (JPA)

Nome completo: *javax.persistence.Temporal*

Objetivo: usado para indicar o tipo de dado **temporal** que será guardado no campo do atributo mapeado.

Onde deve ser incluída: sobre um atributo*. Opcional. Pode ser usada em atributos dos tipos **Calendar** ou **Date**.

Atributos:

- **value** (obrigatório): indica o tipo de dado temporal do campo. Os tipos de estratégias são os valores da *enum* **javax.persistence.TemporalType**:
  - **TIMESTAMP**: mostra que o campo receberá a data e a hora. Muito usado em campos dos tipos **datetime** e **timestamp**.
  - **DATE**: aponta que o campo receberá somente a data. Muito utilizado em campos do tipo **date**. Quando recuperado do banco, o atributo do tipo **Calendar** ou **Date** estará sempre com a hora “zerada” (ex: “00:00:00”).
  - **TIME**: indica que o campo receberá somente a hora. Muito usado em campos do tipo **time**. Quando recuperado do banco, o atributo do tipo

Calendar ou **Date** estará com o dia em 1 de janeiro de 1970, sendo relevante apenas a hora, minuto, segundo e milissegundo. Ocorre que esses dois tipos em Java não conseguem representar somente uma "hora".

Como atuou no exemplo: apontou que o atributo **dataCriacao** receberia valores com **data e hora**.

7.2.3.9 @Enumerated (JPA)

Nome completo: javax.persistence.Enumerated

Objetivo: usado em campos mapeados em atributos de tipos de **enums**. Indica se no banco será armazenado o valor literal do **enum** (valor alfanumérico) ou sua ordem na **classe enum** (número inteiro, a partir do 0).

Onde deve ser incluída: sobre um atributo*. Opcional. Pode ser usada em atributos de tipos de **enums**.

Atributos:

- **value** (obrigatório): aponta a estratégia para o preenchimento e recuperação do valor do atributo. Os tipos de estratégias são os valores da *enum* **javax.persistence.EnumType**:
  - **STRING**: mostra que o valor do **enum** será literalmente convertido em uma String para ser armazenado no campo mapeado. Por exemplo, um **tipo enum** com os valores **AZUL** e **VERMELHO** teria os valores **"AZUL"** e **"VERMELHO"**, respectivamente, como possibilidades para o campo.
  - **ORDINAL**: indica que a ordem do **enum** em sua classe será usada para determinar o valor que será armazenado no campo mapeado. Por exemplo, um **tipo enum** com os valores **AZUL, VERDE** e **VERMELHO** teria os valores **0, 1** e **2**, respectivamente, como possibilidades para o campo.

7.2.3.10 @CreationTimestamp (Hibernate)

Nome completo: *org.hibernate.annotations.CreationTimestamp*

Objetivo: mostra que o atributo receberá automaticamente a data e a hora do sistema no momento da **criação** de um registro.

Onde deve ser incluída: sobre um atributo*. Opcional. Pode ser usada em atributos do tipo **Calendar** ou **Date**.

Atributos: não possui.

Como atuou no exemplo: apontou que o **dataCriacao** teria seu valor preenchido automaticamente com a data e a hora atuais do sistema quando um novo registro fosse criado.

7.2.3.11 @UpdateTimestamp (Hibernate)

Nome completo: *org.hibernate.annotations.UpdateTimestamp*

Objetivo: indica que o atributo receberá automaticamente a data e a hora do sistema no momento da **atualização** de um registro.

Onde deve ser incluída: sobre um atributo*. Opcional. Pode ser usada em atributos do tipo **Calendar** ou **Date**.

Atributos: não possui.

7.2.3.12 @Formula (Hibernate)

Nome completo: *org.hibernate.annotations.Formula*

Objetivo: usada para mostrar que um determinado atributo não está mapeado para um campo da tabela, mas que seu valor, sempre que solicitado, será uma ***sub select*** ou uma **função de agregação**. Muito útil para os chamados **campos calculados**.

Onde deve ser incluída: sobre um atributo*. Opcional.

Atributos:

- **value** (obrigatório): atributo no qual apontamos a **instrução SQL** que será usada para determinar o valor do atributo da classe anotado com **@Formula**. Para evitar efeitos colaterais, é recomendado que o ***select*** dentro da instrução esteja entre parênteses. Se, em vez de um ***sub select,*** for apenas usada uma função de agregação (***avg***, ***sum***, ***count***, ***min***, ***max*** etc.), os parênteses não serão necessários.

Como atuou no exemplo: indicou que o campo mediaAvaliacoes não corresponde a nenhuma tabela no banco, mas que, ao ser solicitado, seu valor corresponderia à *sub select* **select avg(a.nota)+1 from avaliacao a where a.id_estabelecimento = id_estabelecimento**.

CONCLUSÃO

Neste capítulo, vimos o conceito de ORM e alguns detalhes sobre sua realização. Ficamos a par das informações iniciais sobre o JPA e Hibernate, que são as ferramentas mais usadas em Java para o Mapeamento Objeto-Relacional.

Recorra sempre a este capítulo quando precisar de mais detalhes, principalmente sobre as anotações de ORM do JPA e Hibernate. No próximo capítulo, veremos um pequeno estudo de caso com algumas tabelas mapeadas num projeto Java.

REFERÊNCIAS

JBOSS.ORG. **Hibernate ORM 5.2.12. Final User Guide**. Disponível em: <https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html>. Acesso em: 20 out. 2017.

JENDROCK, Eric. **Persistence – The Java EE5 Tutorial**. Disponível em: <https://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html>. Acesso em: 20 out. 2017.

PANDA, Debu; RAHMAN, Reza; CUPRAK, Ryan; REMIJAN, Michael. **EJB 3 in Action.** 2. ed. Shelter Island, NY, EUA: Manning Publications, 2014.