INSPIRING: ORM & JAVA 2.0

TRADUZINDO OBJETO PARA O MODELO RELACIONAL

4

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

LISTA DE FIGURAS

FIAP

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

LISTA DE QUADROS

FIAP

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

LISTA DE CÓDIGOS-FONTE

FIAP

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

SUMÁRIO

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

1 TRADUZINDO OBJETO PARA O MODELO RELACIONAL

Você sabia que existe um jeito mais fácil de fazer o Java se conectar em um banco de dados? Neste capítulo, aprenderemos o que são *softwares* Object Relacional Mapping – ORM, que prometem mapear o modelo orientado a objetos para o modelo relacional e vice-versa, de uma maneira muito prática!

1.1 Mapeamento Objeto-Relacional – ORM

O **Mapeamento Objeto-Relacional** (do 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 realizado a partir de tabelas novas, criadas com o projeto Java, ou de tabelas legadas, isto é, já 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 ficar explícita a quantidade de tabelas, seus campos, suas chaves primárias e os relacionamentos. Por fim, essa abordagem facilita a criação de testes unitários automatizados.

1.1.1 Nomes de tabelas e campos *versus* nomes de classes e atributos

Durante o processo de ORM, é importante respeitar as diferenças de convenções de 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.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

Exemplos: tabela “**tipo_estabelecimento**”.
Campo “**id_tipo_estabelecimento**”.

- **Java: camelCase**

Já 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 de **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.

Há, ainda, a possibilidade de 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 a seguir temos um exemplo de como os nomes de uma tabela e seus campos poderiam ficar mapeados numa classe Java.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

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

Quadro 1 – Nomes de tabelas e campos e seus campos numa classe Java
Fonte: Elaborado pelo autor (2017)

1.1.2 Tipos de campos *versus* 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 a seguir 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 2 – Tipos de campos *versus* classes Java
Fonte: Elaborado pelo autor (2017)

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 de seu funcionamento.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

1.2 Configurando o JPA e o Hibernate

Para usar JPA e Hibernate em um projeto Java, basta adicionar as dependências no projeto, ou seja, as bibliotecas (jars) do Hibernate e o *driver* do banco de dados que será utilizado. Podemos fazer isso de forma manual, navegando no site do projeto ORM do Hibernate (http://hibernate.org/orm/) para realizar o *download* das bibliotecas. Depois, basta adicionar os arquivos (.jar) no projeto, conforme trabalhamos no passado, com o *driver* do oracle e jstl.

Porém, existe uma forma mais fácil e eficiente, que é muito utilizada no mercado atualmente. Estamos falando do Maven, uma ferramenta que gerencia os projetos, capaz de definir as dependências, controlar a versão, gerar relatórios, garantir a execução dos testes unitários etc.

Nesse momento, vamos utilizar o Maven para gerenciar as dependências do projeto JPA. Assim, criaremos nosso primeiro projeto Maven no Eclipse. Você pode estar se perguntando se é preciso instalar algo para utilizar o Maven? A resposta é: depende, como vamos utilizar o Eclipse, não é preciso instalar nada, pois o Eclipse já vem com um plugin do Maven pronto para uso!

Então vamos ao Eclipse!

1.2.1 Alterando a perspectiva do Eclipse para JPA

A primeira coisa a fazer é mudar a perspectiva do Eclipse para **JPA**, caso ela não seja a atual. Lembra-se de que comentamos que o Eclipse permite o desenvolvimento de vários tipos de projetos? Como Web ou Java Application? Para alguns tipos de projetos o Eclipse possui uma perspectiva, ou seja, um conjunto de janelas e menus que é mais adequado para esse tipo do projeto.

A figura a seguir indica onde normalmente está o botão que permite alterar a perspectiva e o assistente que aparece logo em seguida. Ao mudar de perspectiva, você notará mudanças sutis na IDE, mas as alterações mais importantes estão nos menus, que passam a ser mais apropriados para projetos JPA.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

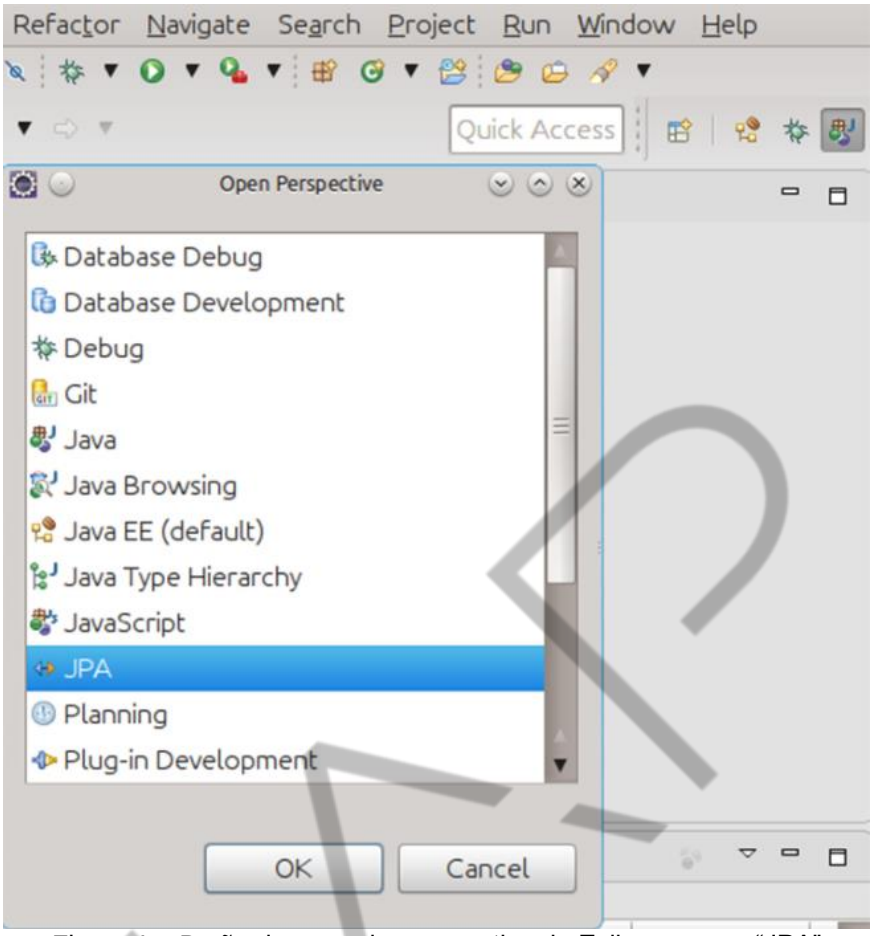

Figura 1 – Botão de troca de perspectiva do Eclipse para a “JPA”
Fonte: Elaborado pelo autor (2017)

1.2.2 Criando um projeto Maven no Eclipse

Para usar o JPA e Hibernate num projeto Java, o ideal é criar um projeto Maven, incluir as dependências necessárias e indicar no Eclipse que o projeto usa JPA. Para criar um projeto assim, basta usar o menu **File -> New -> Project...**, indicar que se trata de um projeto Maven, configurar o local do projeto e marcar a opção **Create a simple project (skip archeype selection)** e indicar as informações Maven do projeto.

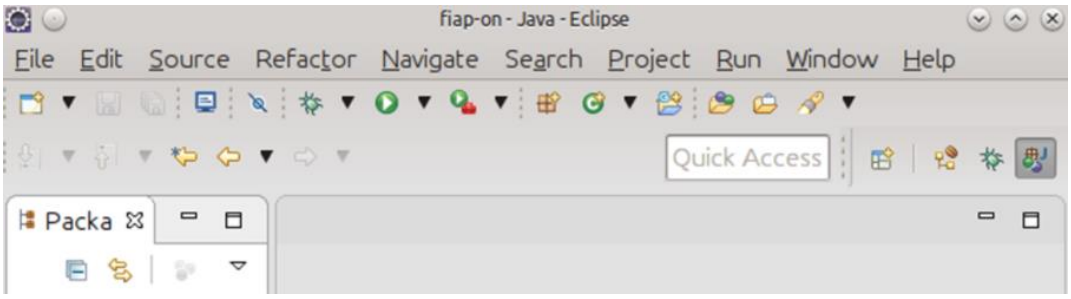

Figura 2 – Menu para a criação de novo projeto
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

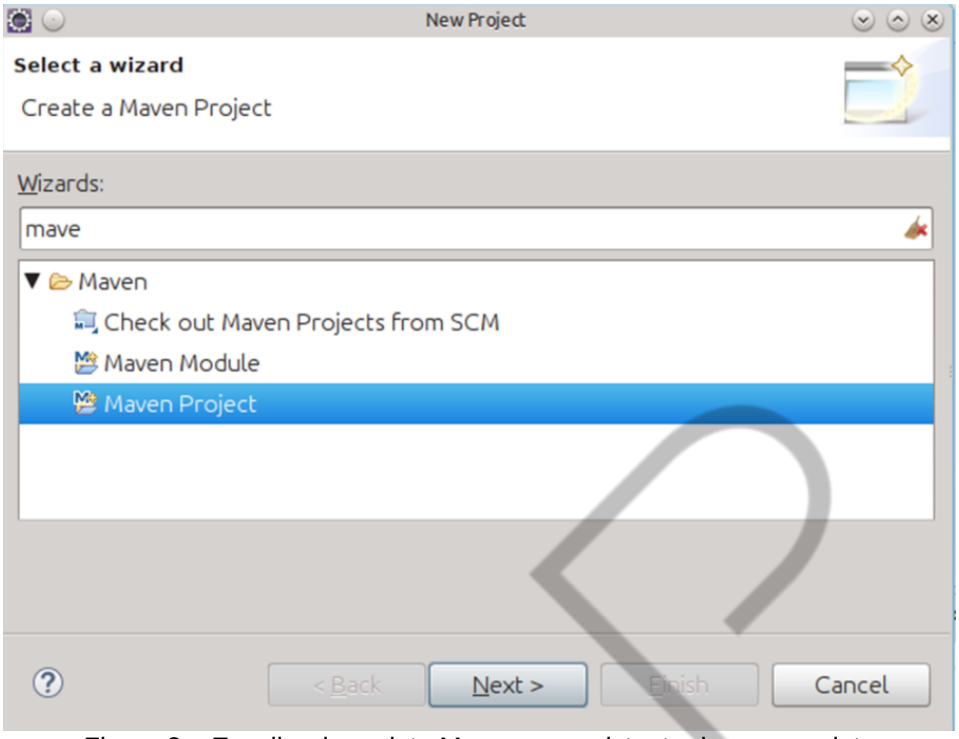

Figura 3 – Escolha de projeto Maven no assistente de novo projeto
Fonte: Elaborado pelo autor (2017)

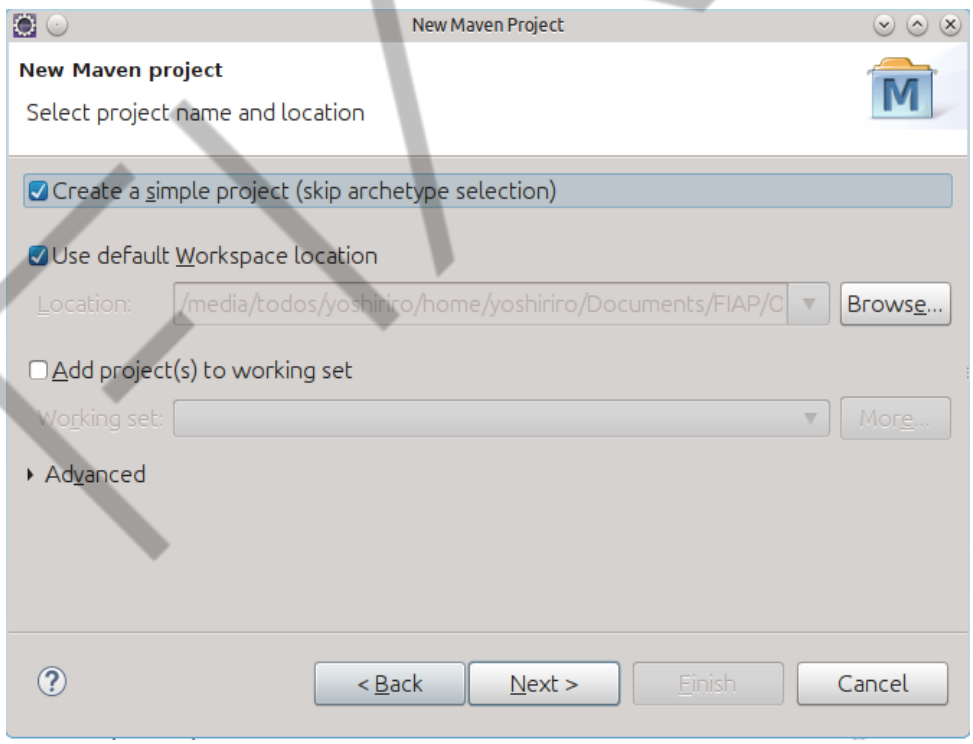

Figura 4 – Local do projeto e sem “Create a simple project” no assistente de novo projeto
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

O próximo passo é configurar o Group Id e o Artifact Id. O grupo é geralmente a URL ao contrário e o Artifact id é o identificar único, geralmente o nome do projeto, que deve ser único dentro do grupo. A versão e o tipo de artefato final não precisam ser alterados.

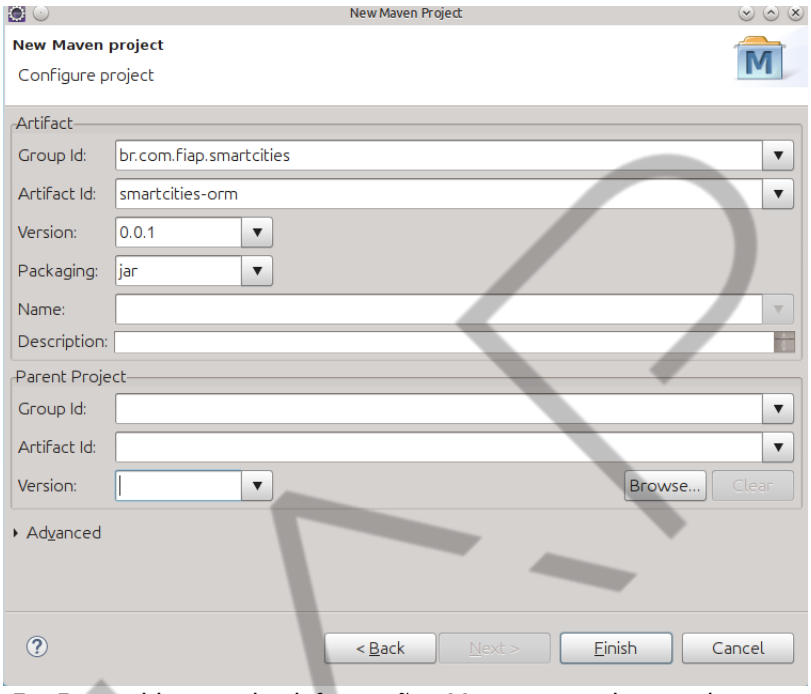

Figura 5 – Preenchimento das informações Maven no assistente de novo projeto
Fonte: Elaborado pelo autor (2017)

Depois de criado, o projeto deve possuir os diretórios **src/main/java**, **src/main/resources**, **src/test/java**, **src/test/resources** e o arquivo **pom.xml**.

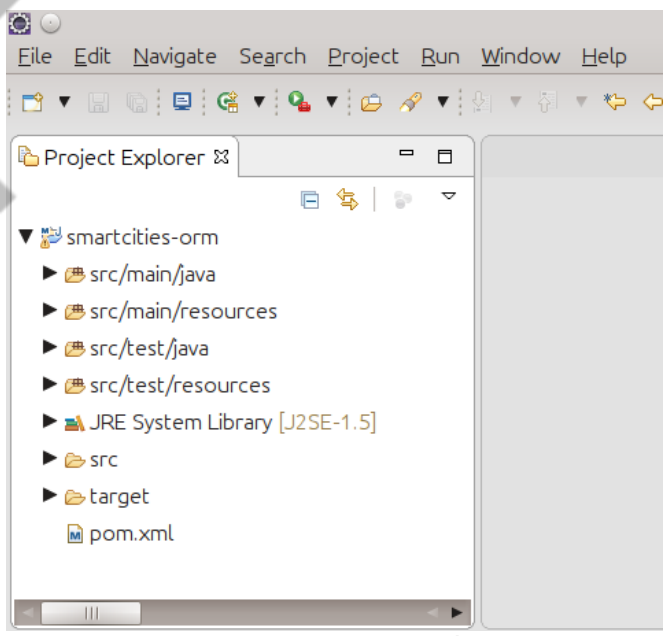

Figura 6 – Projeto Maven recém-criado
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

Dependendo da configuração de seu Eclipse, ele pode usar o **J2SE-1.5** (ou seja, o **Java 1.5**) para o projeto. **Apenas se isso ocorrer**, você deve configurar o projeto para usar o Java 8, a fim de usufruir das importantes *features* dessa versão do Java. Comece clicando com o botão direito do mouse sobre o projeto e use o menu **Build Path -> Configure Build Path**. Caso o Eclipse já tenha deixado o Java 8 configurado para o projeto, pode desconsiderar esse procedimento.

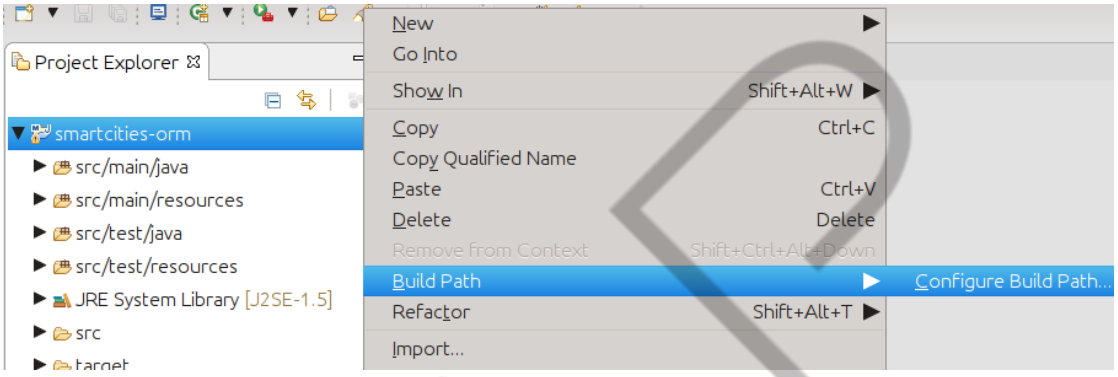

Figura 7 – Menu de configuração de “Build Path” do projeto
Fonte: Elaborado pelo autor (2017)

Na janela que aparecer, vá para a aba **Libraries** e marque o item **JRE System Library**.

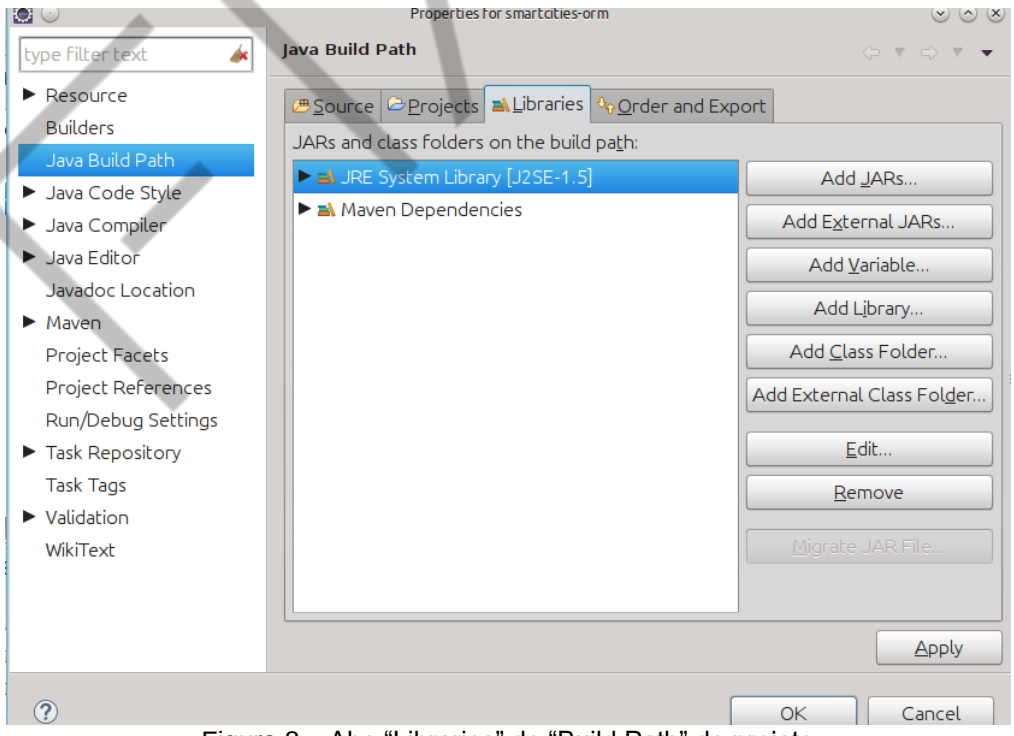

Figura 8 – Aba “Libraries” do “Build Path” do projeto
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

A seguir, clique em **Edit** e, dentre os itens de **Execution environment** escolha a versão JavaSE-1.8 que estiver disponível, conforme exemplifica a Figura “Alteração da versão do Java para Java 8 no ‘Build Path’ do projeto”.

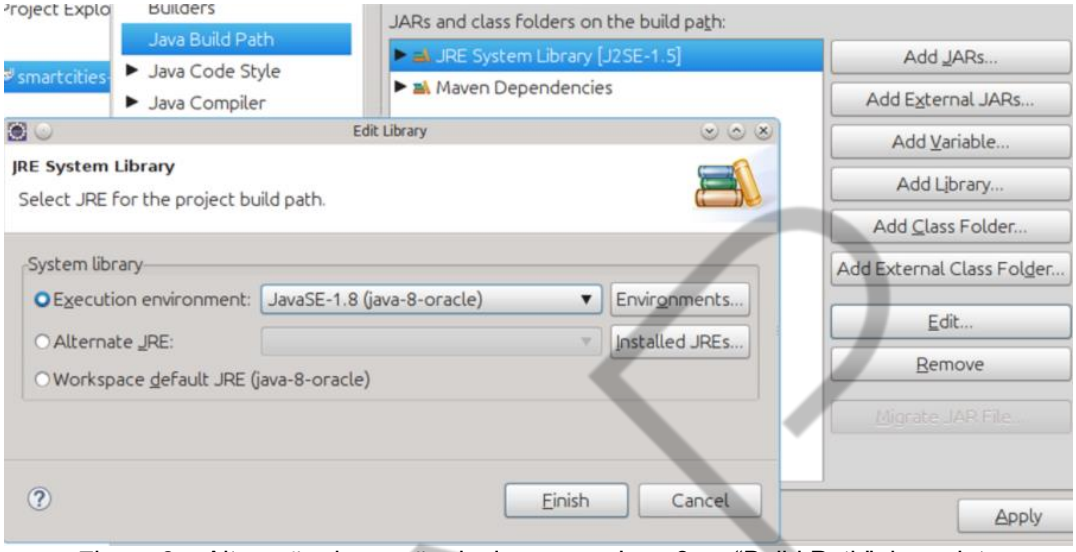

Figura 9 – Alteração da versão do Java para Java 8 no “Build Path” do projeto
Fonte: Elaborado pelo autor (2017)

Depois é só finalizar o procedimento.

Note, no projeto, o arquivo chamado **pom.xml**, esse arquivo é a configuração principal do Maven e define o grupo, artefato id, versão, dependências, *build* etc.

Para utilizar o hibernate, precisamos adicionar a dependência no arquivo **pom.xml**. Veja o exemplo no código-fonte a seguir:

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

Código-fonte 1 – Dependência necessária para uso de JPA e Hibernate
Fonte: Elaborado pelo autor (2017)

As dependências podem ser pesquisadas em um repositório público do maven, como https://mvnrepository.com/. O groupId e ArtifactId determinam a dependência que foi adicionada ao projeto. É possível também especificar uma versão da dependência na tag <version><version>. Se essa dependência (Hibernate, no exemplo) precisar de outras bibliotecas, o Maven gerenciará automaticamente para nós.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

Abra o arquivo pom.xml e adicione a dependência do Hibernate, conforme o código-fonte a seguir. Note que as dependências devem ficar dentro da tag <dependencies></dependencies>.

```
      <project      xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>br.com.fiap.smartcities</groupId>
        <artifactId>smartcities-orm</artifactId>
        <version>0.0.1</version>
        <dependencies>
          <dependency>
              <groupId>org.hibernate</groupId>
              <artifactId>hibernate-core</artifactId>
              <version>RELEASE</version>
          </dependency>
        </dependencies>
      </project>
```

Código-fonte 2 – Pom.xml completo, com a dependência do Hibernate
Fonte: Elaborado pelo autor (2017)

A dependência do driver de algum banco de dados também deve ser incluída. O código-fonte a seguir exibe 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>
```

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

```
        </dependency>
```

Código-fonte 3 – 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.

A fim de 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, ou utilizar o que está disponível aqui na plataforma ou utilizar o que está disponível aqui na plataforma no link: https://drive.google.com/drive/folders/1qrCw2oJT6cmyysdJVHYW5nZbi6U-CG0A?usp=sharing

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 no mesmo lugar que o código-fonte de seu projeto.

O código-fonte abaixo exemplifica 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 4 – Como configurar a dependência do arquivo local no pom.xml
Fonte: Elaborado pelo autor (2017)

As tags **groupId**, **artifactId** e **version** poderiam ter, na verdade, qualquer valor (contanto que não entre 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**.

O código-fonte a seguir apresenta um exemplo de código completo do arquivo pom.xml, com as dependências do Hibernate e *drive* do Oracle.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

```
      <project       xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0
.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
          <modelVersion>4.0.0</modelVersion>
          <groupId>br.com.fiap.smartcities</groupId>
          <artifactId>smartcities-orm</artifactId>
          <version>0.0.1</version>
          <dependencies>
               <dependency>
                    <groupId>org.hibernate</groupId>
                    <artifactId>hibernate-core</artifactId>
                    <version>RELEASE</version>
               </dependency>
               <dependency>
                    <groupId>oracle</groupId>
                    <artifactId>jdbc-driver</artifactId>
                    <version>12</version>
                    <scope>system</scope>

     <systemPath>/Users/Opensource/ojdbc8.jar</systemPath>
               </dependency>
          </dependencies>
       </project>
```

Código-fonte 5 – Exemplo de configuração do pom.xml com as dependências do Hibernate e *drive* do Oracle

Fonte: Elaborado pelo autor (2017)

Não se esqueça de ajustar o *path* com o caminho correto até o *driver* do Oracle.

Se a dependência foi configurada corretamente e sua IDE conseguir fazer o download das dependências, você poderá ver os arquivos jars no **"Maven Dependencies"**.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

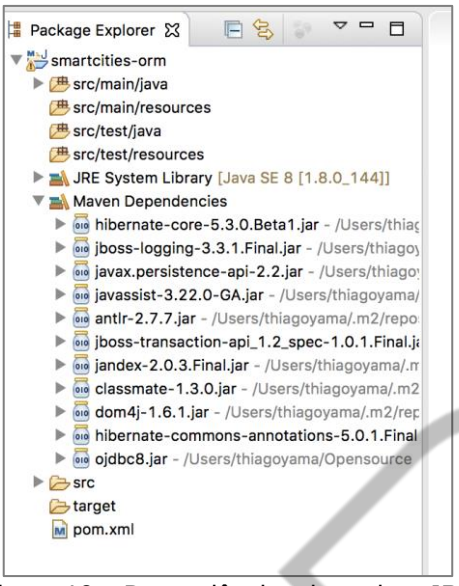

Figura 10 – Dependências do projeto JPA
Fonte: Elaborado pelo autor (2017)

1.2.3 Indicando o uso de JPA pelo projeto no Eclipse

Depois de criar o projeto Maven, devemos indicar para o Eclipse que nosso projeto fará uso do JPA. Basta clicar com o botão direito do mouse sobre o projeto e ir ao menu **Configure -> Convert to JPA Project**.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

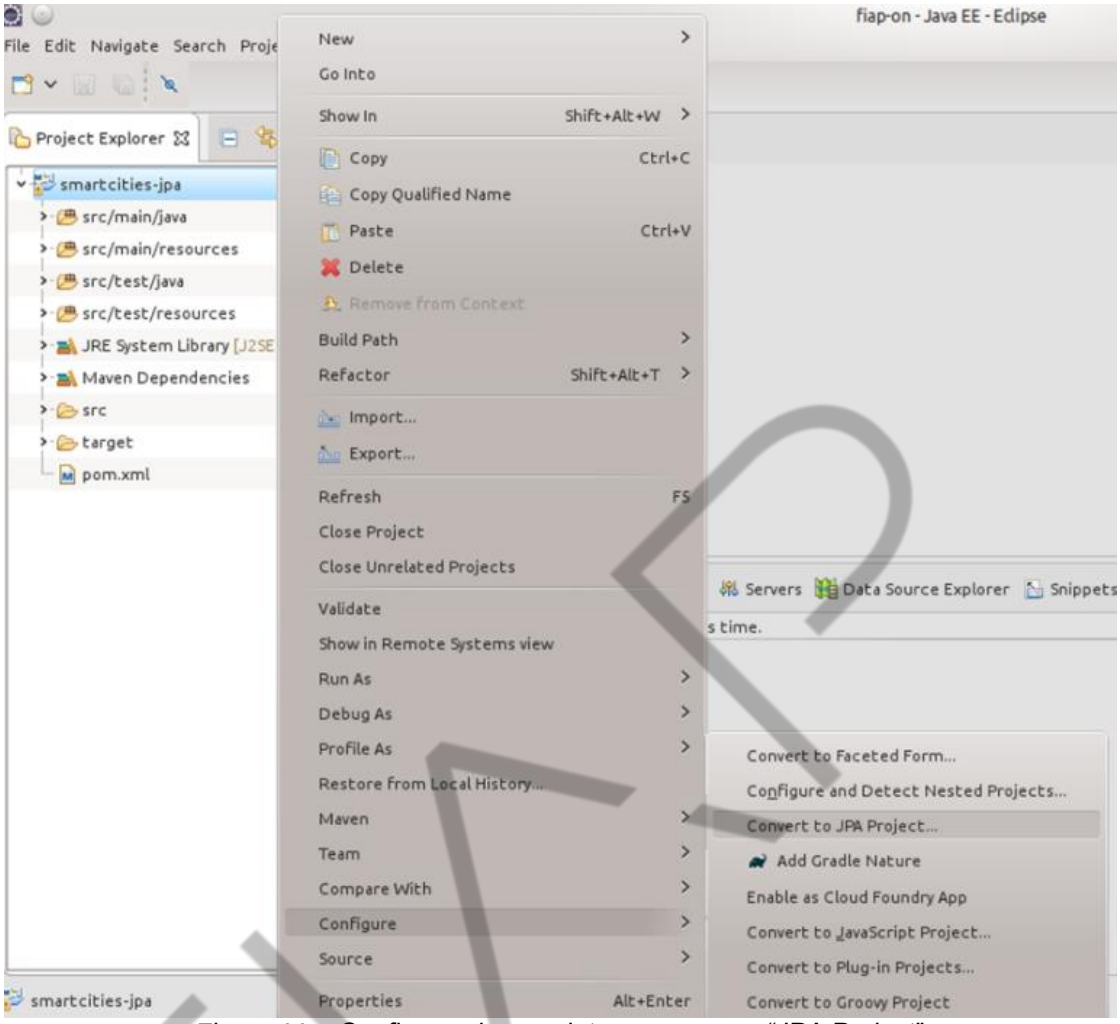

Figura 11 – Configurando o projeto para ser um “JPA Project”
Fonte: Elaborado pelo autor (2017)

Depois de usar o menu indicado, um assistente aparecerá. Confirme se o item **JPA** está marcado e na versão 2.1. Então, clique em **Next**.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

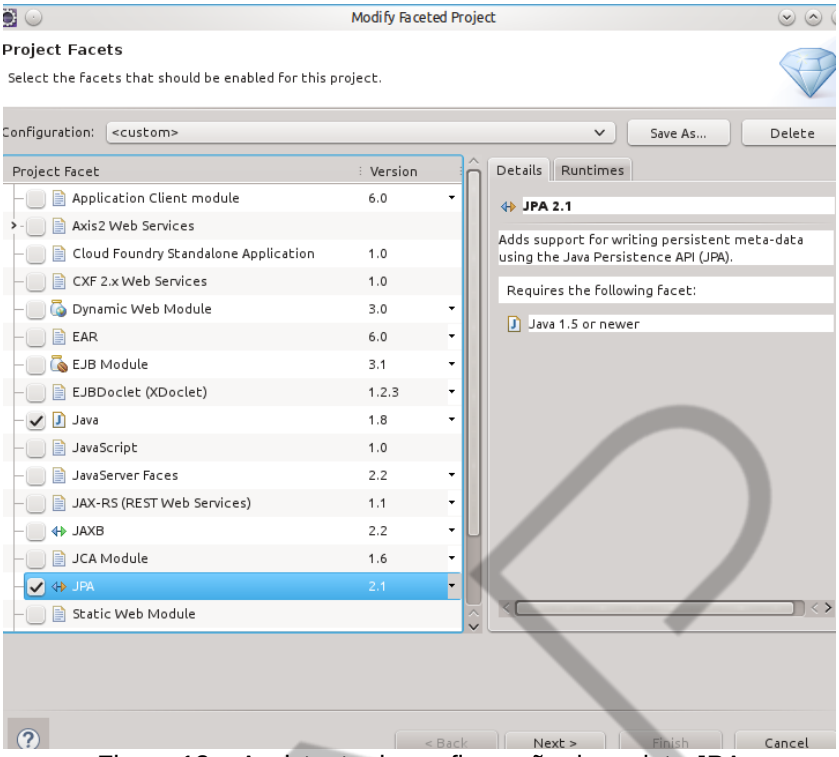

Figura 12 – Assistente de configuração de projeto JPA
Fonte: Elaborado pelo autor (2017)

Uma nova tela do assistente solicitará que confirme o diretório de código-fonte do projeto. Apenas clique em **Next**.

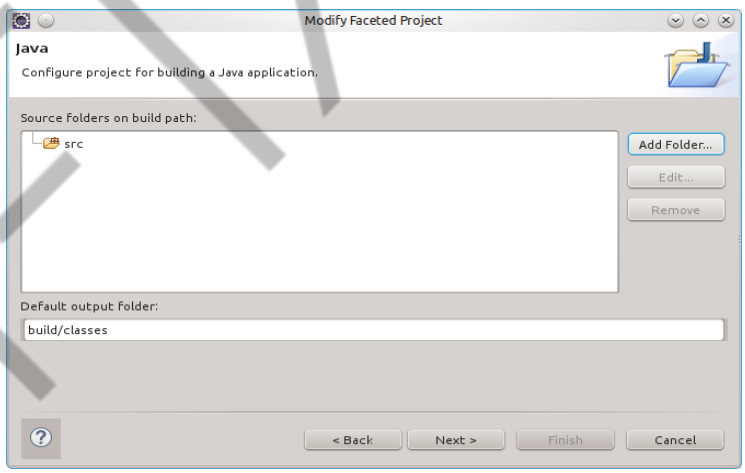

Figura 13 – Assistente de configuração de projeto JPA – confirmação de diretório de código-fonte
Fonte: Elaborado pelo autor (2017)

Por fim, será solicitado que indique a plataforma (**Platform**) e a implementação JPA (**JPA Implementation**). Use os valores **Generic 2.1** e **Disable Library Configuration**, respectivamente, como mostra a Figura “Plataforma e implementação JPA”. Então, pode clicar em **Finish**.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

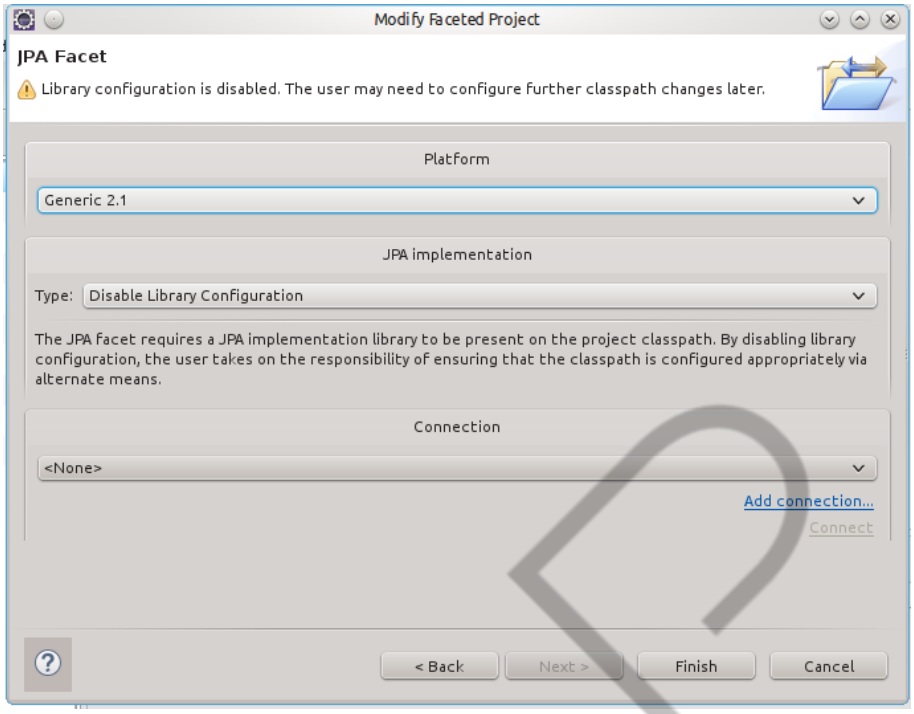

Figura 14 – Plataforma e implementação JPA na configuração de projeto JPA
Fonte: Elaborado pelo autor (2017)

Depois de executar o assistente, como acabamos de ver, para confirmar se tudo ocorreu bem, verifique se o projeto ficou com um aspecto como o da Figura “Projeto após a execução do assistente de configuração do projeto JPA”.

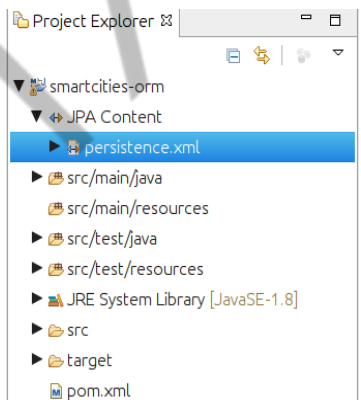

Figura 15 – Projeto após a execução do assistente de configuração do projeto JPA
Fonte: Elaborado pelo autor (2017)

Abra o arquivo **persistence.xml**, que é o **arquivo central de configuração e mapeamento ORM do projeto JPA**. Um editor diferenciado deve aparecer. A recomendação é **nunca** usar esse editor “visual”. O motivo é que, caso passe por alguma situação e recorra a pesquisas na internet, ninguém orientará sobre o que fazer no persistence.xml via editor visual e sim sobre como mexer no arquivo xml de

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

forma textual. Assim, sempre prefira editar esse arquivo por meio da aba **Source**, que fica na parte inferior do arquivo.

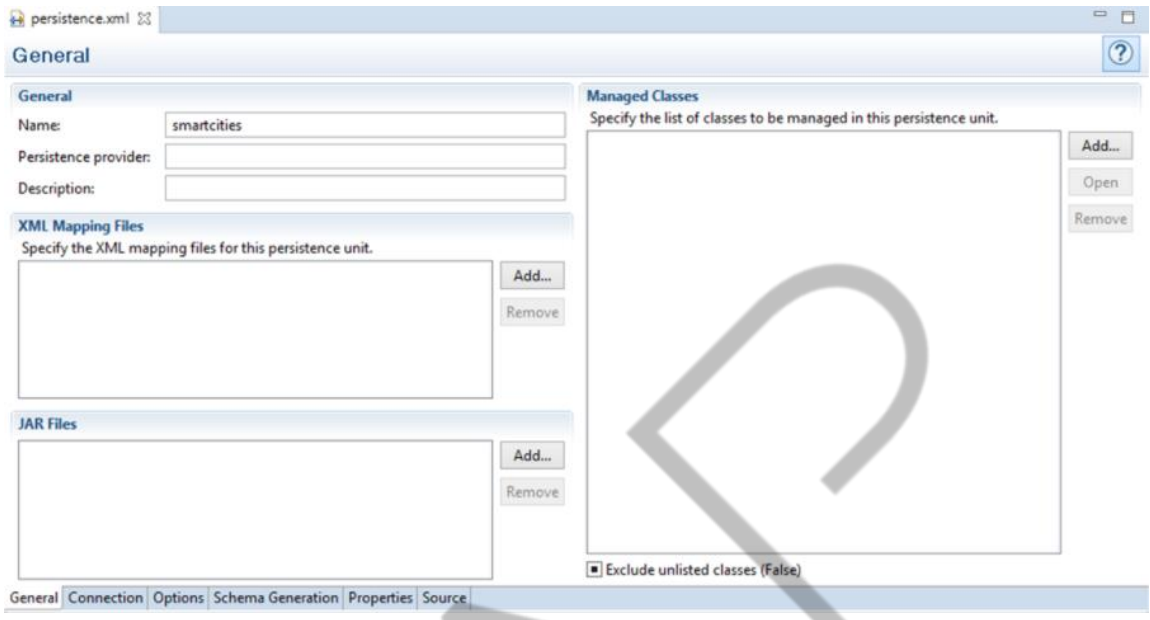

Figura 16 – Editor padrão do arquivo “persistence.xml”
Fonte: Elaborado pelo autor (2017)

Ao pedir para usar a aba **Source**, o código que você verá será um XML. Não se assuste com o conteúdo da tag **<persistence>**. Você não precisa memorizá-lo. O Eclipse (assim como outras IDEs Java) já deixa ela pronta para você.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence                                    version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

          <persistence-unit>
          </persistence-unit>

</persistence>
```

Código-fonte 6 – Conteúdo inicial do “persistence.xml”
Fonte: Elaborado pelo autor (2017)

1.2.4 Configurando o persistence.xml

Depois de criar o projeto e configurá-lo com JPA junto ao Eclipse, devemos editar o arquivo **persistence.xml**, que é o **arquivo central de configuração e mapeamento ORM do projeto JPA**. Nele, indicamos:

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

- Nome e tipo de controle de transação da *persistence unit.*
- Qual implementação do JPA é utilizada.
- Classes de ORM (entidades).
- Configurações de acesso ao banco de dados.
- Propriedades adicionais conforme a necessidade.

A seguir, veremos como fazer cada uma dessas configurações nesse arquivo.

1.2.5 Persistence.xml – nome e tipo de controle de transação da persistence unit

No JPA, uma **persistence unit** é como uma conexão com um banco de dados específico. Embora seja raro, é possível termos mais de uma tag **<persistence-unit>**, ou seja, mais de uma **persistence unit** num mesmo **persistence.xml**. Uma **persistence unit** precisa ter um nome e, opcionalmente, o tipo de transação. Os tipos de transação são **RESOURCE_LOCAL** e **JTA**, sendo que o primeiro é muito mais comum, enquanto o segundo é para situações bem específicas, exigindo uma série de configurações adicionais. Em nossos exemplos, vamos chamar a **persistence unit** de **smartcities** e usar o tipo de transação **RESOUCE_LOCAL**.

```
<persistence ...>

        <persistence-unit name="smartcities" transaction-
type="RESOURCE_LOCAL">
        </persistence-unit>

</persistence>
```

Código-fonte 7 – Nome e tipo de controle de transação da “persistence unit”
Fonte: Elaborado pelo autor (2017)

1.2.6 Persistence.xml – implementação do JPA

Como dito anteriormente, o JPA é apenas uma especificação e o Hibernate é um *framework* que a implementa. Porém, como existem outros *frameworks* que fazem o mesmo que o **Hibernate**, é preciso indicar, explicitamente, que estamos

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

usando **Hibernate**. Para isso, introduzimos a tag **<provider>** no **persistence.xml** com o valor **org.hibernate.jpa.HibernatePersistenceProvider**.

```
<persistence ...>
<persistence-unit ...>

<provider>org.hibernate.jpa.HibernatePersistenceProvider</pro
vider>

</persistence-unit>
</persistence>
```

Código-fonte 8 – Implementação do JPA da “persistence unit”
Fonte: Elaborado pelo autor (2017)

1.2.7 Persistence.xml – classes do ORM (Entidades)

No **persistence.xml** também devemos definir quais classes fazem ORM para tabelas. Para cada classe, usamos uma tag **<class>** (uma após a outra em caso de várias classes) e, em cada uma delas, colocamos o caminho completo da classe (pacote e nome). Para exemplificar, vamos supor que fôssemos usar uma entidade chamada **Estabelecimento**. Supondo que ela está no pacote **br.com.fiap.smartcities.domain**.

```
<persistence ...>
<persistence-unit ...>
<provider ...>

<class>br.com.fiap.smartcities.domain.Estabelecimento</class>
<class>br.com.fiap.smartcities.domain.OutraEntidade</class>
<class>br.com.fiap.smartcities.domain.MaisUmaEntidade</class>

</persistence-unit>
</persistence>
```

Código-fonte 9 – Nome e tipo de controle de transação da “persistence unit”
Fonte: Elaborado pelo autor (2017)

1.2.8 Persistence.xml – configurações de acesso ao banco de dados

Uma vez que o JPA/Hibernate permite acessar um banco de dados e ter as tabelas acessíveis por meio de classes Java, é necessário indicar no **persistence.xml** como se conectar ao banco de dados. Para isso, abrimos uma tag **<properties>** e, dentro dela, várias **<property>**.

As propriedades comuns, independentemente do banco de dados, são:

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

- **hibernate.dialect**: dialeto que o Hibernate usará para montar as instruções SQL que serão enviadas ao SGBD. Não é preciso decorar todos os dialetos, basta saber que estão no pacote org.hibernate.dialect. Os nomes das classes nesse pacote são bem intuitivos, pois contêm o nome do SGBD.
- **javax.persistence.jdbc.driver**: classe do *driver* de conexão com o SGBD. Esse valor é bem especifico por banco. Normalmente descobre-se o valor na documentação oficial do *driver* de conexão fornecido pelo próprio SGBD ou em exemplos espalhados pela internet.
- **javax.persistence.jdbc.url**: URL de conexão com o SGBD. Esse valor é bem específico por banco. Assim como a classe, normalmente descobre-se o valor na documentação oficial do *driver* de conexão fornecido pelo próprio SGBD ou em exemplos espalhados pela internet.
- **javax.persistence.jdbc.user**: usuário de conexão com o SGBD.
- **javax.persistence.jdbc.password**: senha do usuário de conexão com o SGBD.

Os valores para essas propriedades variam um pouco conforme o banco de dados usado, e há um exemplo de como seria uma conexão junto a uma base **Oracle 12**.

No Código-fonte “Configuração de conexão com o Derby (ou JavaDB) na ‘persistence unit’”, há um exemplo de conexão com um **Derby** (também conhecido como **JavaDB**) embarcado, no qual você vai notar que o usuário e a senha são vazios.

```
<persistence ...>
<persistence-unit ...>
<provider ...>
<class>...</class>

<properties>

<property name="hibernate.dialect"
value="org.hibernate.dialect.DerbyDialect" />

<property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.EmbeddedDriver" />
```

FIAP

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

```
<property name="javax.persistence.jdbc.url"
value="jdbc:derby:minhabase;create=true" />

<property name="javax.persistence.jdbc.user"
value="" />

<property name="javax.persistence.jdbc.password"
value="" />

</properties>

</persistence-unit>
</persistence>
```

Código-fonte 10 – Configuração de conexão com o Oracle 12 na “persistence unit”
Fonte: Elaborado pelo autor (2017)

```
<persistence ...>
<persistence-unit ...>
<provider ...>
<class>...</class>

<properties>

<property name="hibernate.dialect"
value="org.hibernate.dialect.Oracle12cDialect" />

<property name="javax.persistence.jdbc.driver"
value="oracle.jdbc.OracleDriver" />

<property name="javax.persistence.jdbc.url"
value="jdbc:oracle:thin:@ip-do-servidor:1521:minhabase" />

<property name="javax.persistence.jdbc.user"
value="meu-usuario" />

<property name="javax.persistence.jdbc.password"
value="minha-senha" />

</properties>

</persistence-unit>
</persistence>
```

Código-fonte 11 – Configuração de conexão com o Derby (ou JavaDB) na “persistence unit”
Fonte: Elaborado pelo autor (2017)

1.2.9 Persistence.xml – propriedades adicionais conforme a necessidade

Além das configurações que acabamos de ver, no **persistence.xml** podemos incluir várias outras configurações, conforme a necessidade. Existe uma infinidade

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

de configurações, como cache, pool de conexões etc. Aqui, vamos mostrar apenas as configurações para a criação e atualização automática das tabelas e exibição das instruções SQL enviadas para o banco, pois são extremamente úteis no desenvolvimento.

O Hibernate possui um recurso bem poderoso, que é o de criar e/ou atualizar as tabelas no banco conforme alteramos as classes de ORM. Por exemplo, se criamos uma nova entidade (classe) indicando uma tabela que não existe, o Hibernate cria essa tabela na base quando a aplicação inicia. Ou, se criamos um atributo novo para um campo que ainda não existe: esse campo será criado na tabela também. Porém, esse recurso não é habilitado por padrão. Para habilitá-lo, usamos a propriedade **hibernate.hbm2ddl.auto** com o valor **update**. Veja como fazer isso no código-fonte a seguir.

IMPORTANTE: caso você não queira que a base seja atualizada, não inclua essa propriedade ou use-a com o valor **validate**, que verifica se suas entidades estão compatíveis com as tabelas do banco.

```
<persistence ...>
<persistence-unit ...>
<provider ...>
<class>...</class>

<properties>

<property.../>

<property name="hibernate.hbm2ddl.auto" value="update" />

</properties>

</persistence-unit>
</persistence>
```

Código-fonte 12 – Habilitando a atualização automática da base na “persistence unit”
Fonte: Elaborado pelo autor (2017)

Para que o Hibernate exiba as instruções SQL que ele envia para o banco (DMLs e DDLs), basta incluir novas tags **<property>** com as propriedades **hibernate.show_sql** e **hibernate.format_sql** e valores **true** em ambas. Vide o código-fonte abaixo.

```
<persistence ...>
<persistence-unit ...>
```

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

```
<provider ...>
<class>...</class>

<properties>

<property.../>

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

</properties>

</persistence-unit>
</persistence>
```

Código-fonte 13 – Habilitando a exibição das instruções SQL enviadas ao banco pelo Hibernate na "persistence unit"
Fonte: Elaborado pelo autor (2017)

O código-fonte a seguir apresenta a configuração final do arquivo persistence.xml, que utiliza o banco de dados Oracle, cria as tabelas assim que o programa executa e exibe as queries enviadas ao banco.

```
      <?xml version="1.0" encoding="UTF-8"?>
      <persistence                                    version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
          <persistence-unit name="smartcities-orm">

    <provider>org.hibernate.jpa.HibernatePersistenceProvider
</provider>
                <class>
br.com.fiap.smartcities.domain.Estabelecimento </class>
                <properties>
                     <property        name="hibernate.show_sql"
value="true"/>
                     <property    name="hibernate.hbm2ddl.auto"
value="create"/>
                     <property         name="hibernate.dialect"
value="org.hibernate.dialect.Oracle12cDialect"/>
                     <property
name="javax.persistence.jdbc.driver"
value="oracle.jdbc.OracleDriver"/>
                     <property
name="javax.persistence.jdbc.user" value=""/>
                     <property
name="javax.persistence.jdbc.password" value=""/>
                     <property
```

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

```
name="javax.persistence.jdbc.url"
value="jdbc:oracle:thin:@oracle.fiap.com.br:1521:orcl"/>
                </properties>
            </persistence-unit>
        </persistence>
```

Código-fonte 14 – Habilitando a exibição das instruções SQL enviadas ao banco pelo Hibernate na "persistence unit"
Fonte: Elaborada pelo autor (2017)

1.3 Criando as classes de ORM

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 neste momento.

1.3.1 Anotações JPA e Hibernate

A configuração das classes de ORM com anotações é mais produtivo e fácil de realizar. A especificação JPA possui um conjunto de anotações que são utilizadas para definir, por exemplo, o nome da tabela, nome das colunas etc. O Hibernate implementa essa especificação e adiciona algumas outras anotações próprias, que não fazem parte da especificação do JPA, porém facilitam alguns processos, como a inserção de data de cadastro ou atualização.

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 código-fonte a seguir 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)
```

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

```
        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 15 – Entidade Estabelecimento
Fonte: Elaborado pelo autor (2017)

Analisando o código-fonte, é possível entender como está configurada a tabela no banco de dados, o seu nome, as suas colunas e a chave primária. Agora, vamos detalhar cada uma das anotações e suas configurações.

1.3.2 @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.

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.

O link: https://drive.google.com/drive/folders/17Pb3UPCI6fNOUZfQGMsJMpbRMwb8Q-Ao?usp=sharing apresenta o projeto desenvolvido até este ponto do curso. Caso você não tenha sucesso na criação e configuração do projeto, faça o download dessa versão e importe para sua IDE (use a opção Import existing Maven projects), altere o nome do usuário e senha do banco de dados e faça novamente a execução da classe de teste.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

1.3.3 @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 de nomes já abordada 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”.

1.3.4 @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 ou método get do atributo. Obrigatório em pelo menos um atributo.

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

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

1.3.5 @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 ou método get do atributo. Opcional.

Atributos:

- **strategy** (opcional): indica 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 o usará mais 1. Se o atributo **strategy** for omitido, essa 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 gerará uma instrução de *insert* apropriada para o banco de dados configurado para que ele 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 modo 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.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

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

1.3.6 @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, no método get do atributo ou classe. 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 no qual está a ***sequence***. Se omitido, o JPA considerará que está no mesmo que a tabela.
- **catalog** (opcional): nome do ***catálogo*** do banco no qual está a ***sequence***. Se omitido, o JPA considerará que está no mesmo que a tabela.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

1.3.7 @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, ou método get do 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 no qual está a ***tabela***. Se omitido, o JPA considerará que está no mesmo que a tabela.
- **catalog** (opcional): nome do ***catálogo*** do banco no qual está a ***tabela***. Se omitido, o JPA considerará que está no mesmo que a tabela.

1.3.8 @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 ou método get do 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

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

na tabela diferente de seu atributo mapeado na classe. Isso ajuda a resolver a questão de convenções de nomes já abordada neste capítulo.

- **length** (opcional): aponta o tamanho do campo na tabela. Por exemplo, para um campo ***varchar***, esse campo mostraria a quantidade de caracteres que ele comporta.
- **precision** (opcional): indica a precisão do campo. Esse atributo só se aplica a campos numéricos.
- **scale** (opcional): mostra a escala do campo. Esse atributo só se aplica a campos numéricos.
- **nullabe** (opcional): aponta se o campo é obrigatório (**false**) ou se é possível criar/atualizar o 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**.

1.3.9 @Temporal (JPA)

Nome completo: *javax.persistence.Temporal*

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

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

Onde deve ser incluída: sobre um atributo ou método get do atributo. Opcional. Só pode ser usada em atributos dos tipos de Data do Java.

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" (exemplo: "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 1º 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**.

1.3.10 @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 ou método get do atributo. Opcional. Só pode ser usada em atributos de tipos de **enums**.

Atributos:

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

- **value** (obrigatório): aponta a estratégia para o preenchimento e a 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 armazenada 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.

1.3.11 @CreationTimestamp (Hibernate)

Nome completo: org.hibernate.annotations.CreationTimestamp

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

Onde deve ser incluída: sobre um atributo ou método get do atributo. Opcional. Só 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 hora atuais do sistema quando um novo registro fosse criado.

1.3.12 @UpdateTimestamp (Hibernate)

Nome completo: org.hibernate.annotations.UpdateTimestamp

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

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

Onde deve ser incluída: sobre um atributo ou método get do atributo. Opcional. Só pode ser usada em atributos do tipo **Calendar** ou **Date**.

Atributos: não possui.

1.3.13 @Formula (Hibernate)

Nome completo: org.hibernate.annotations.Formula

Objetivo: usada para indicar 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 ou método get do 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 usada apenas 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**.

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com

REFERÊNCIAS

JBOSS.ORG. **Hibernate ORM 5.2.12. Final User Guide**. [s.d.]. Disponível em: <https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html>. Acesso em: 15 dez. 2020.

JENDROCK, E. **Persistence – The Java EE5 Tutorial**. [s.d.]. Disponível em: <https://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html>. Acesso em: 15 dez. 2020.

PANDA, D.; RAHMAN, R.; CUPRAK, R.; REMIJAN, M. **EJB 3 in Action.** 2. ed. Shelter Island, NY, EUA: Manning Publications, 2014.

FIAP

PDF exclusivo para Higor dos Reis Vieira - rm94963
higorr776@gmail.com