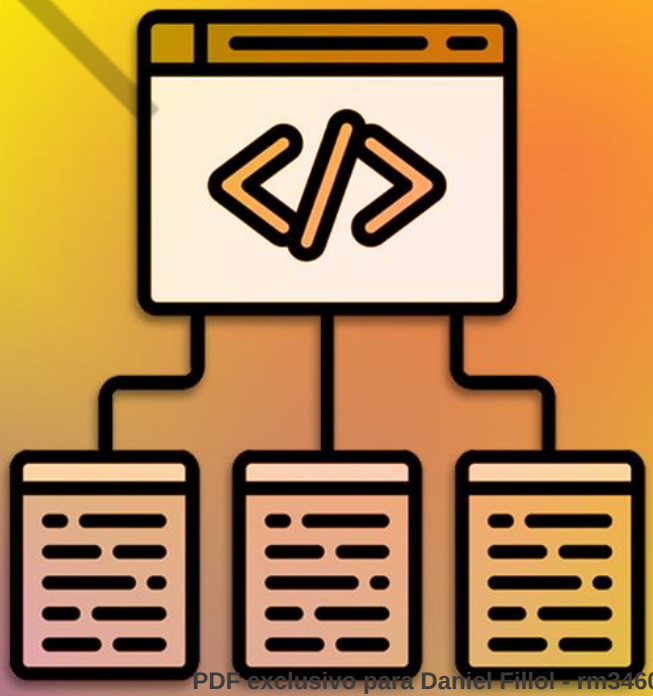
FIAP ON

FUNDAMENTOS JAVA

PERSISTÊNCIA OO (JPA)

JOSÉ YOSHIRIRO



8

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

LISTA DE FIGURAS

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

LISTA DE CÓDIGOS-FONTE

FIAP

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

SUMÁRIO

FIAP

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

8 PERSISTÊNCIA OO (JPA)

8.1 Mapeamento

8.1.1 Introdução

Para que o processo de ORM fique completo, além de anotar uma classe Java com anotações JPA e Hibernate, precisamos mapear o projeto para alguma base de dados, de modo que possamos ser capazes de começar a persistir (salvar), atualizar, excluir e consultar registros.

8.1.2 Alterando a perspectiva do Eclipse para JPA

A primeira coisa a fazer é mudar a perspectiva do Eclipse para **JPA**, caso não seja ela a atual. A Figura – *Botão de troca de perspectiva do Eclipse para a "JPA"* – aponta onde normalmente está o botão que permite essa alteração e o assistente que aparece logo em seguida. Ao mudar de perspectiva, você notará mudanças sutis na IDE, mas as mudanças mais importantes estão nos menus, que passam a ser mais apropriados para projetos JPA.

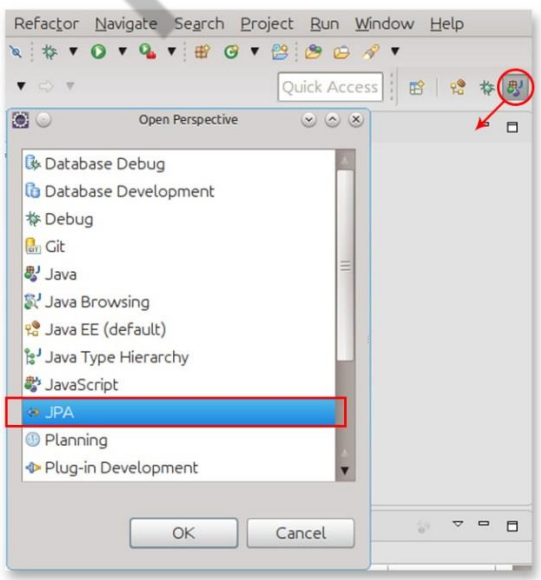


Figura 8.1 – Botão de troca de perspectiva do Eclipse para a "JPA"
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

8.1.3 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...** (Figura *Menu para a criação de novo projeto*);
- Indicar que se trata de um projeto Maven (Figura *Escolha de projeto "Maven" no assistente de novo projeto*);
- Configurar o local do projeto e marcar a opção **Create a simple project (skip archeype selection)** (Figura *Local do projeto e sem "Create a simple project" no assistente de novo projeto*);
- Mostrar as informações Maven do projeto (Figura *Preenchimento das informações "Maven" no assistente de novo projeto*).

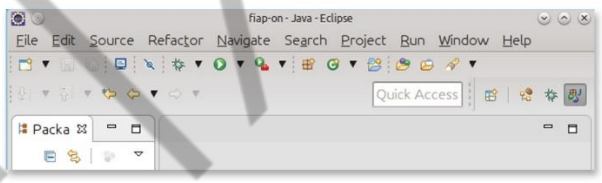


Figura 8.2 – Menu para a criação de novo projeto
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

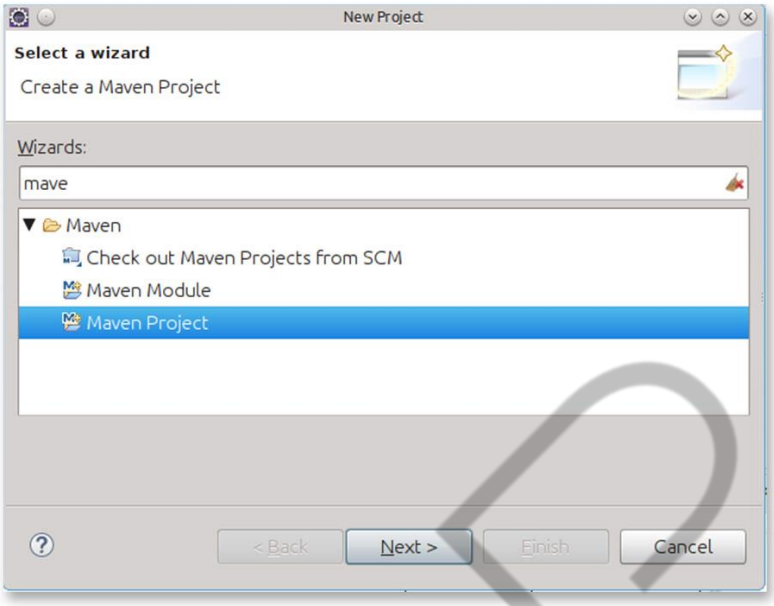


Figura 8.3 – Escolha de projeto “Maven” no assistente de novo projeto
Fonte: Elaborado pelo autor (2017)

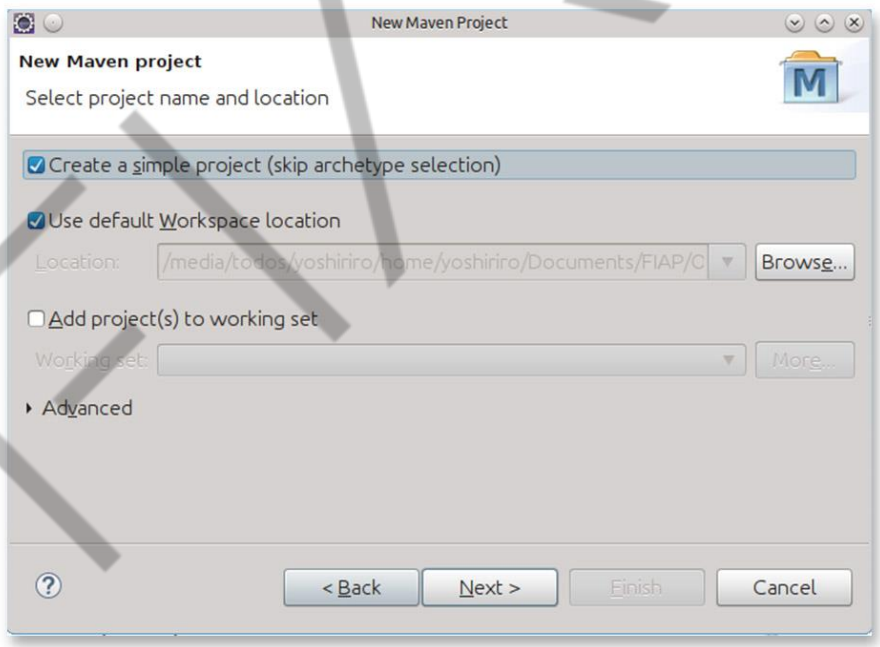


Figura 8.4 – Local do projeto e sem “*Create a simple project*” no assistente de novo projeto
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

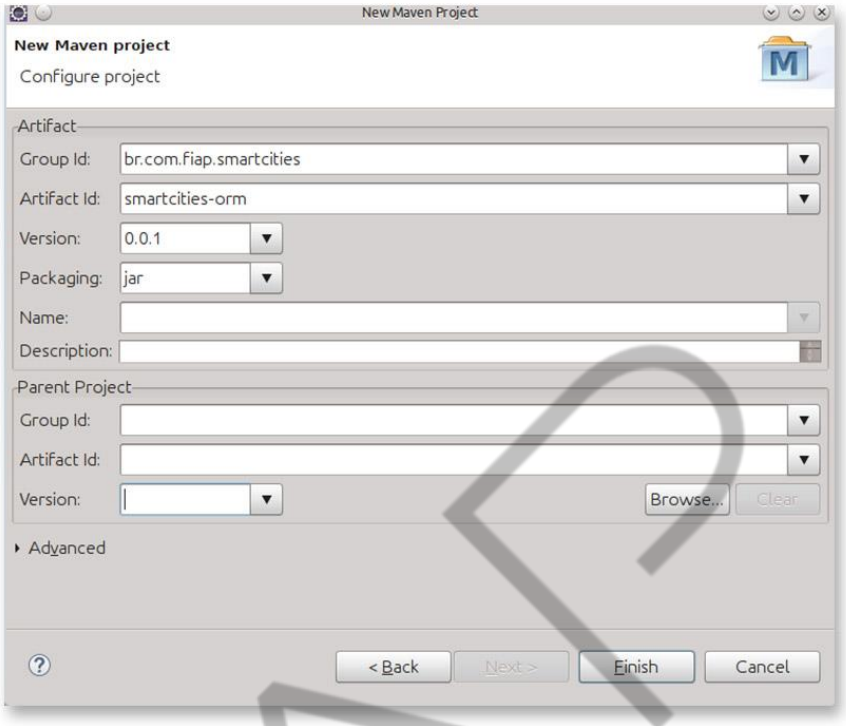


Figura 8.5 – Preenchimento das informações “Maven” no assistente de novo projeto
Fonte: Elaborado pelo autor (2017)

Após criado, o projeto deve ter os diretórios **src/main/java**, **src/main/resources**, **src/test/java**, **src/test/resources** e o arquivo **pom.xml**, conforme a Figura *Projeto “Maven” recém-criado*.

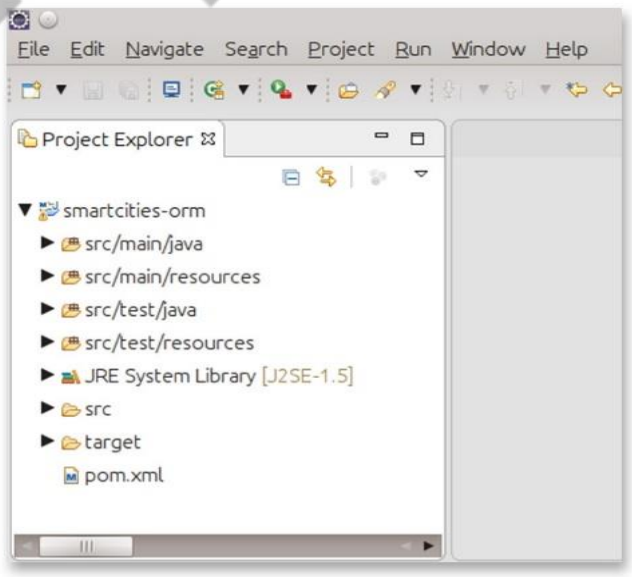


Figura 8.6 – Projeto “Maven” recém-criado
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

Dependendo da configuração de seu Eclipse, ele pode usar o **J2SE-1.5** (ou seja, o **Java 1.5**) para o projeto (como na Figura *Projeto "Maven" recém-criado*). **Apenas se isso ocorrer**, você deve configurar o projeto para usar o Java 8, a fim de usufruir das importantes novidades das novas versões do Java. Comece clicando com o botão direito do mouse sobre o projeto e use o menu **Build Path -> Configure Build Path** (conforme a Figura *Menu de configuração de "Build Path" do projeto*). Caso o Eclipse já tenha deixado o Java 8 configurado para o projeto, pode ir direto para a seção "**Indicando uso de JPA pelo projeto no Eclipse"**.

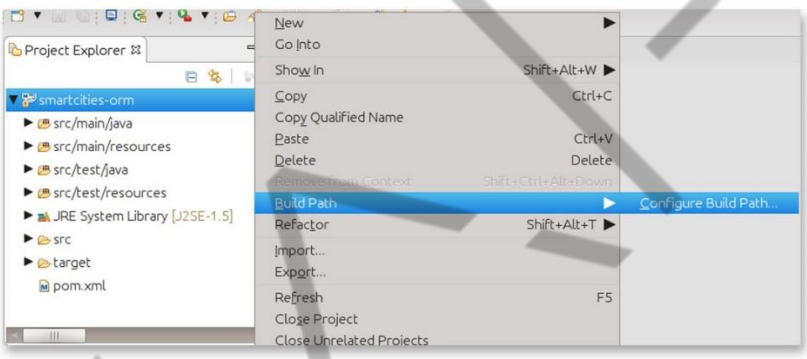


Figura 8.7 – Menu de configuração de "Build Path" do projeto
Fonte: Elaborado pelo autor (2017)

Na janela que aparecer, vá até a aba **Libraries** e marque o item **JRE System Library** (vide Figura *Aba "Libraries" do "Build Path" do projeto*).

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

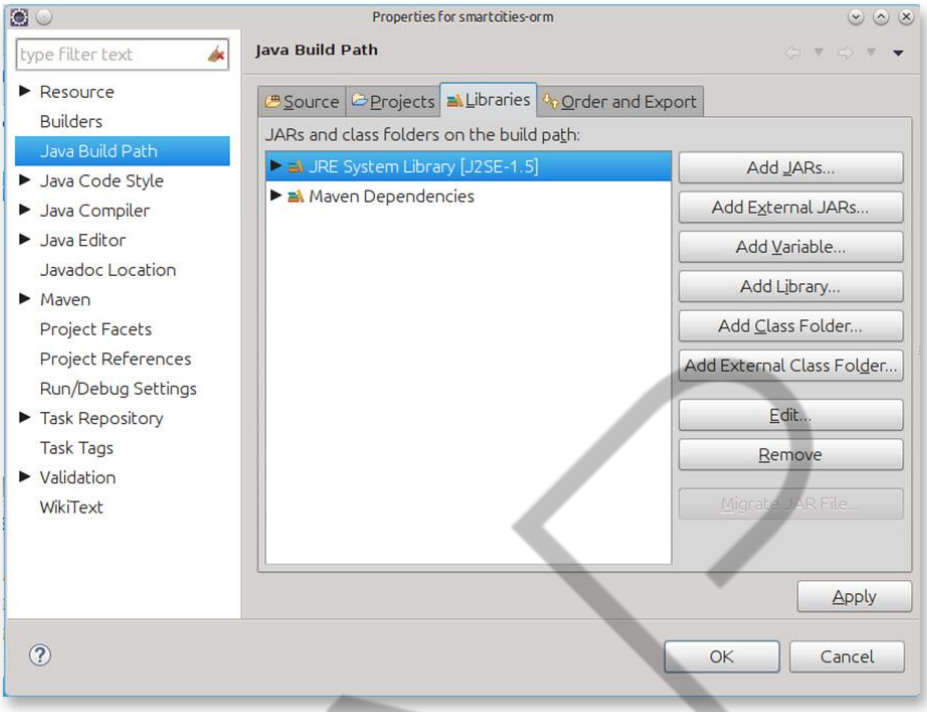


Figura 8.8 – Aba “Libraries” do “Build Path” do projeto
Fonte: Elaborado pelo autor (2017)

A seguir, clique em **Edit**e, entre os itens de ***Execution envinronment***, 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*).

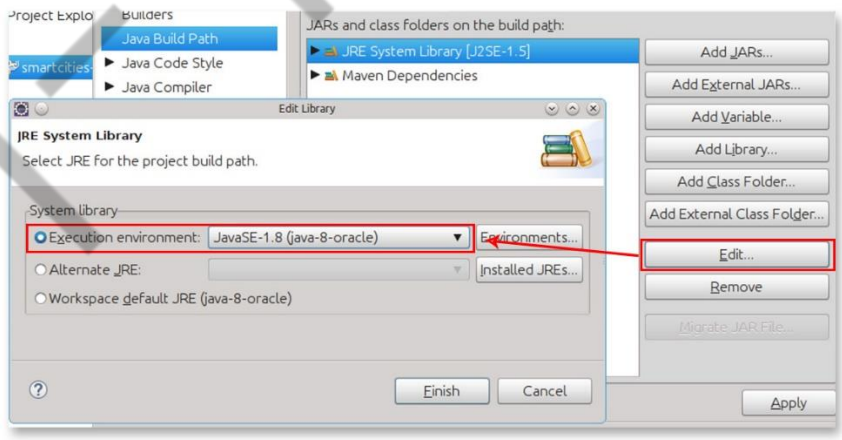


Figura 8.9 – Alteração da versão do Java para Java 8 no “Build Path” do projeto
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

8.1.4 Indicando uso de JPA pelo projeto no Eclipse

Após 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 no menu **Configure -> Convert to JPA Project** (vide Figura *Configurando o projeto para ser um "JPA Project"*).

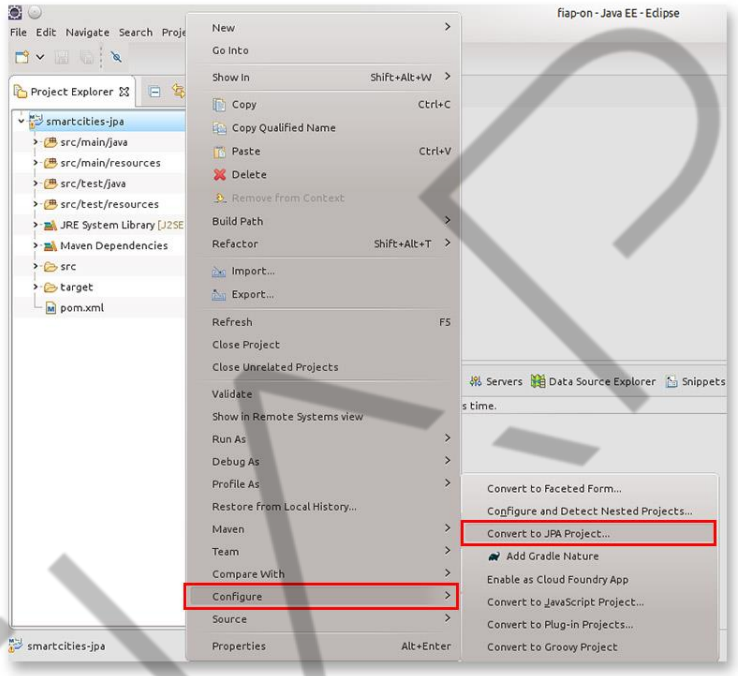


Figura 8.10 – Configurando o projeto para ser um "JPA Project"
Fonte: Elaborado pelo autor (2017)

Após usar o menu indicado, um assistente aparecerá. Confirme se o item **JPA** está marcado e na versão 2.1. Então, clique em **Next** (conforme a Figura *Assistente de configuração de projeto JPA*).

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

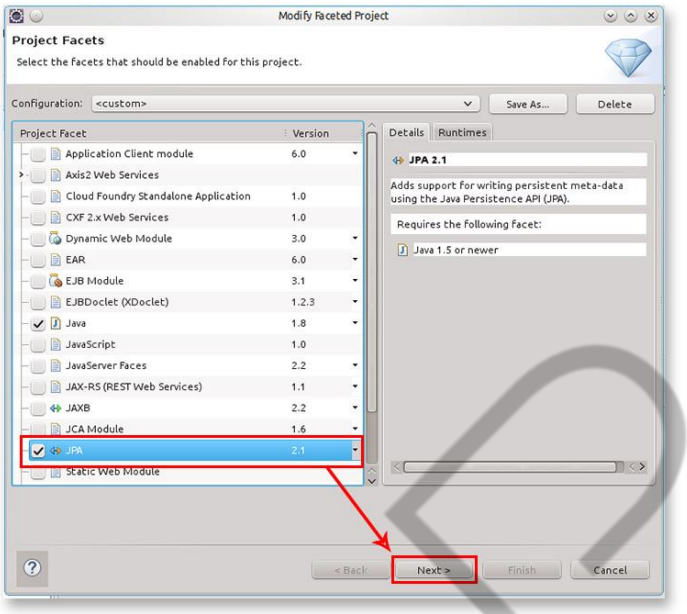


Figura 8.11 – Assistente de configuração de projeto JPA
Fonte: Elaborado pelo autor (2017)

Uma nova tela do assistente solicitará que confirme o diretório de código-fonte do projeto. Apenas clique em **Next**, como indica a Figura *Assistente de configuração de projeto JPA – confirmação de diretório de código-fonte.*

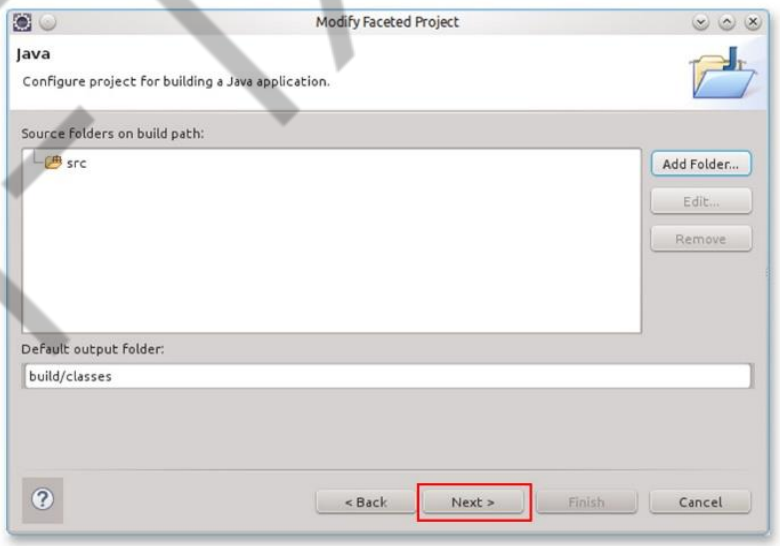


Figura 8.12 – Assistente de configuração de projeto JPA – confirmação de diretório de código-fonte
Fonte: Elaborado pelo autor (2017)

Por fim, será solicitado que indique a plataforma (**Platform**) e a implementação JPA (**JPA Implementation**). Use os valores **Generic 2.1** e **Disable**

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

Library Configuration, respectivamente, como é mostrado na Figura *Plataforma e implementação JPA na configuração de projeto JPA*. Então, clique em **Finish**.

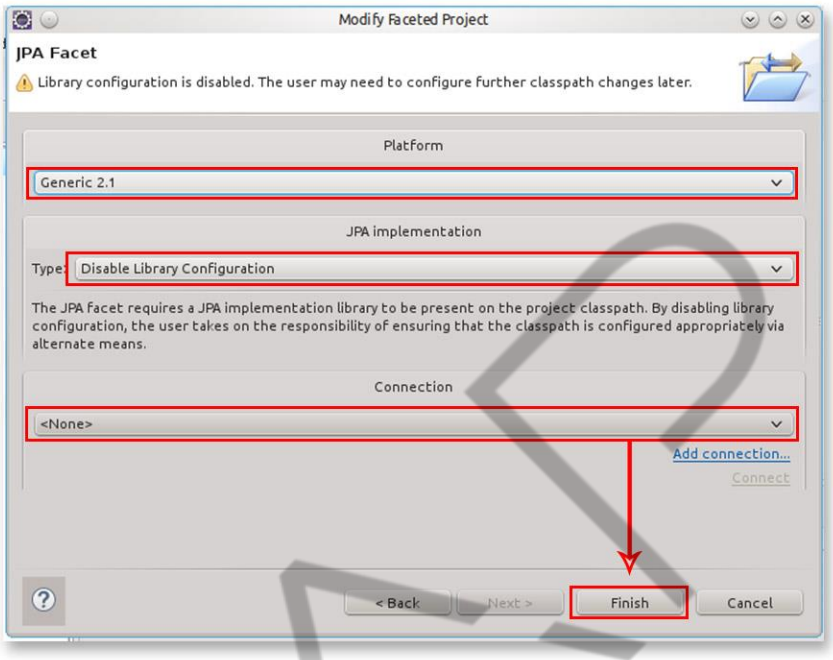


Figura 8.13 – Plataforma e implementação JPA na configuração de projeto JPA
Fonte: Elaborado pelo autor (2017)

Após executar o assistente conforme 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 projeto JPA*.

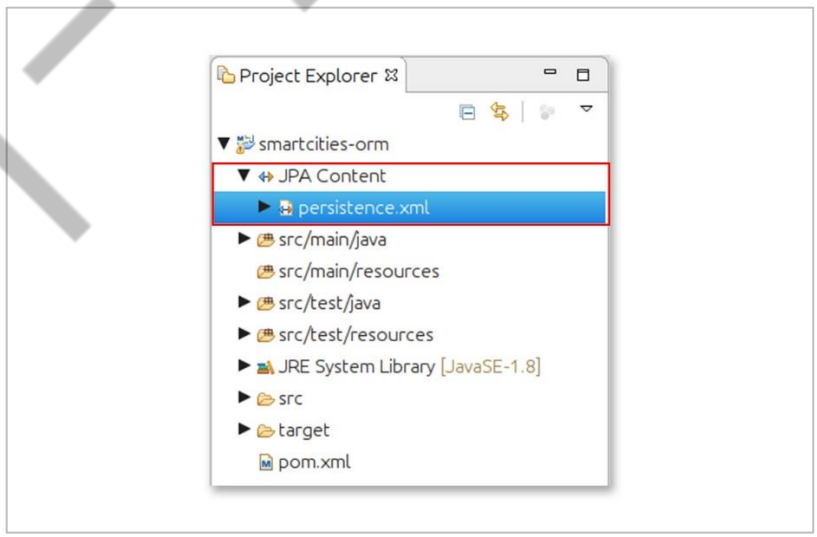


Figura 8.14 – Projeto após a execução do assistente de configuração projeto JPA
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

Abra o arquivo **persistence.xml**, que é o **arquivo central de configuração e mapeamento ORM do projeto JPA**. Um editor diferenciado deve aparecer (como o da Figura *Editor-padrão do arquivo "persistence.xml"*).

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

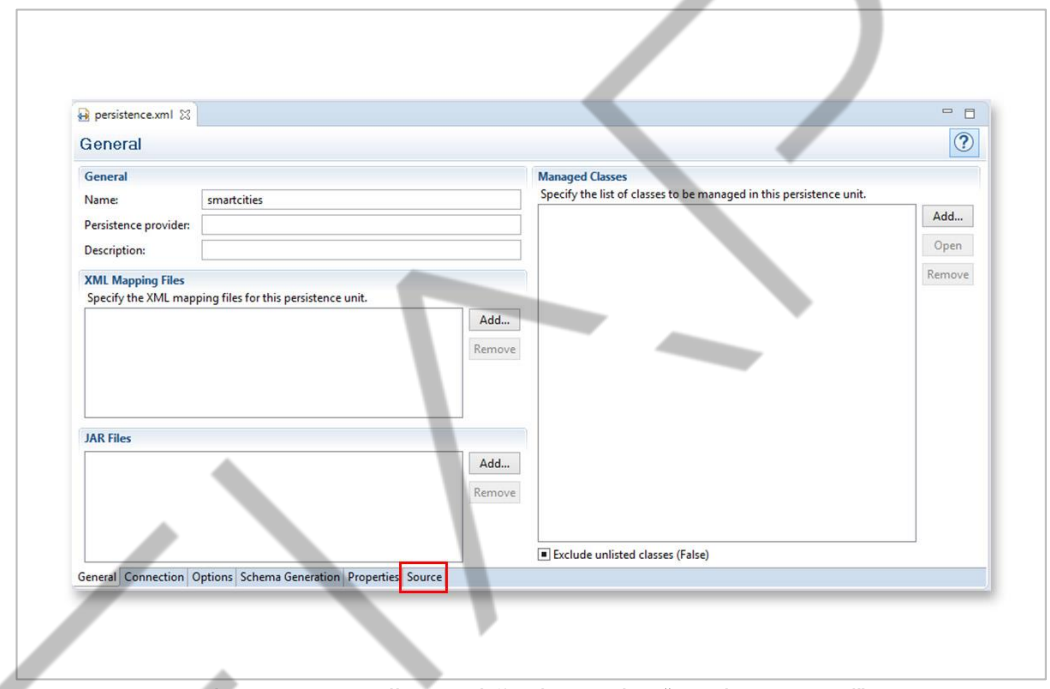


Figura 8.15 – Editor-padrão do arquivo "persistence.xml"
Fonte: Elaborado pelo autor (2017)

Ao pedir para usar a aba **Source**, o código que você verá será um XML como o do Código-fonte *Conteúdo inicial do "persistence.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/persistenc
e
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xs
d">
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

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

    </persistence>
```

Código-fonte 8.1 – Conteúdo inicial do “persistence.xml”
Fonte: Elaborado pelo autor (2017)

8.1.5 Configurando o Persistence.xml

Após criar o projeto e configurá-lo com JPA com o Eclipse, devemos editar o arquivo **persistence.xml**. Esse é o **arquivo central de configuração e mapeamento ORM do projeto JPA**, no qual, indicamos:

- 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.

8.1.5.1 Persistence.xml – Nome/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. Nos nossos exemplos, vamos chamar a **persistence unit** de **smartcities** e usar o tipo de transação **RESOUCE_LOCAL**, como no Código-fonte *Nome e tipo de controle de transação da “persistence unit”*.

```
    <persistence ...>
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

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

    </persistence>
```

Código-fonte 8.2 – Nome e tipo de controle de transação da “persistence unit”
Fonte: Elaborado pelo autor (2017)

8.1.5.2 Persistence.xml – Implementação do JPA

Como foi abordado no capítulo anterior, o JPA é apenas uma especificação e o Hibernate é um framework que o implementa. Porém, como existem outros frameworks que fazem o mesmo que o **Hibernate**, é preciso indicar, explicitamente, que estamos usando **Hibernate**. Para isso, introduzimos a tag **<provider>** no **persistence.xml** com o valor **org.hibernate.jpa.HibernatePersistenceProvider**, como no Código-fonte *Implementação do JPA da “persistence unit”*.

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

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

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

Código-fonte 8.3 – Implementação do JPA da “persistence unit”
Fonte: Elaborado pelo autor (2017)

8.1.5.3 Persistence.xml – Classes de 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, dentro de cada uma delas, colocamos o caminho completo da classe (pacote e nome). Para exemplificar, vamos supor que usaremos a entidade **Estabelecimento** do capítulo anterior. Supondo que ela está no pacote **br.com.fiap.smartcities.domain** (vide o Código-fonte Nome e tipo de controle de transação da “persistence unit”).

```
    <persistence ...>
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

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

<class>br.com.fiap.smartcities.domain.Estabelecimento</c
lass>
<class>br.com.fiap.smartcities.domain.OutraEntidade</cla
ss>
<class>br.com.fiap.smartcities.domain.MaisUmaEntidade</c
lass>

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

Código-fonte 8.4 – Nome e tipo de controle de transação da “persistence unit”
Fonte: Elaborado pelo autor (2017)

8.1.5.4 Persistence.xml – Configurações de acesso ao banco de dados

Como 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:

- **hibernate.dialect**: dialeto que o hibernate usará para montar as instruções SQL que serão enviadas ao SGBD. Não precisa 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 específico 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.

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

- **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 no Código-fonte *Configuração de conexão com Oracle 12 na "persistence unit"* há um exemplo de como seria uma conexão junto a uma base **Oracle 12**. No Código-fonte *Configuração de conexão com Derby na "persistence unit"* há um exemplo de conexão com um **Derby** (também conhecido como **JavaDB**) embarcado, em que você vai notar que usuário e 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" />

<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 8.5 – Configuração de conexão com Oracle 12 na "persistence unit"
Fonte: Elaborado pelo autor (2017)

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

<properties>

<property name="hibernate.dialect"
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
      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 8.6 – Configuração de conexão com Derbu (ou JaaDB) na “persistence unit”
Fonte: Elaborado pelo autor (2017)

8.1.5.5 Persistence.xml – Propriedades adicionais conforme a necessidade

Além das configurações que acabamos de ver, no **persistence.xml**. podemos ainda incluir várias outras configurações, conforme a necessidade. Existe uma infinidade de configurações, como cache, pool de conexões etc. Vamos mostrar aqui apenas as configurações para 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 mostrando 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 no Código-fonte *Habilitando a atualização automática da base na “persistence unit”* como fazer isso.

Importante: Caso você não queira que a base vá sendo atualizada, não inclua essa propriedade ou use-a com o valor validate, que verifica se suas entidades estão compatíveis com as tabelas do banco.

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

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

<properties>

<property.../>

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

</properties>

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

Código-fonte 8.7 – 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 *Habilitando a exibição das instruções SQL enviadas ao banco pelo Hibernate na “persistence unit”*).

```
<persistence ...>
<persistence-unit ...>
<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 8.8 – Habilitando a exibição das instruções SQL
enviadas ao banco pelo Hibernate na “persistence unit”
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

8.1.6 Testando o projeto JPA

8.1.6.1 Criando e testando um EntityManager

Após criar o projeto e configurá-lo com JPA do Eclipse e editar o arquivo **persistence.xml**, podemos testar nosso projeto. Porém, para um teste mais legal, vamos reaproveitar a classe **Estabelecimento** do capítulo anterior e colocá-la no pacote **br.com.fiap.smartcities.model** de nossos códigos-fonte. Vamos manter tudo nela, menos o atributo **mediaAvaliacoes** e seus **get** e **set**.

Vamos criar uma classe que use o JPA e a **persistence unit** que configuramos e ver como são as saídas geradas pelo Hibernate no console do Eclipse. A essa classe vamos chamar **JPATeste** e podemos criá-la no pacote **br.com.fiap.smartcities.testes**. Seu conteúdo seria como o do Código-fonte: *Classe com o código suficiente para testar o projeto JPA*.

```
package br.com.fiap.smartcities.testes;

import javax.persistence.EntityManager;
import javax.persistence.Persistence;
public class JPATeste {

     public static void main(String[] args) {
         EntityManager em = null;
         try {

           // Apenas 1 linha. Quebras para facilitar a
leitura.
           em =  Persistence.

createEntityManagerFactory("smartcities").
                 createEntityManager();

         } catch (Exception e) {

           if (em != null &&
em.getTransaction().isActive()) {
              em.getTransaction().rollback();
           }
           e.printStackTrace();

         }

         if (em != null) {
           em.close();
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
            }
            System.exit(0);
        }
    }
```

Código-fonte 8.9 – Classe com o código suficiente para testar o projeto JPA
Fonte: Elaborado pelo autor (2017)

A classe **EntityManager** é o que representa uma espécie de conexão com o banco de dados. Porém, uma instância desse tipo faz mais coisas que uma simples conexão, o que será abordado com detalhes nos próximos capítulos. A forma de instanciar um **EntityManager** é sempre essa que consta no código de exemplo. O valor entre aspas, o **"smartcities"**, é o nome da **persistence unit**, como explicado para o Código-fonte *Nome e tipo de controle de transação da "persistence unit" da seção* "Persistence.xml – Nome/tipo de controle de transação da persistence unit". Se o **EntityManager** for instanciado com sucesso, indica que as dependências no **pom.xml** estão certas, que o **persistence.xml** está escrito corretamente e que a classe **Estabelecimento** está codificada da forma correta.

No código de exemplo, não fizemos nada com o **em**, que é a instância do **EntityManager**. Porém, é necessário fechá-la com o método **close()** para evitar problemas de conexão com o banco.

A classe **JPATeste** possui um método **main**, o que permite executá-la como uma *Java Application*. Faça a execução e acompanhe a saída na janela **Console** do Eclipse. A saída começará com uma série de informações do Hibernate, como o trecho no Código-fonte *Informações que o Hibernate imprime no Console quando o projeto é executado.* Normalmente, o **Console** colore de vermelho as saídas do Hibernate. **Não se assuste**, não significa erro!

```
    org.hibernate.jpa.internal.util.LogHelper
    logPersistenceUnitInformation
    INFO: HHH000204: Processing PersistenceUnitInfo [
         name: smartcities
         ...]
    org.hibernate.Version logVersion
    INFO: HHH000412: Hibernate Core {5.2.12.Final}
    org.hibernate.cfg.Environment <clinit>
    ... mais informações do Hibernate
```

Código-fonte 8.10 – Informações que o Hibernate imprime no Console quando o projeto é executado
Fonte: Elaborado pelo autor (2017)

Se os atributos de exibição de instruções SQL tiverem sido usados como orientado na **seção "Persistence.xml – Propriedades adicionais conforme a**

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

necessidade”, após imprimir informações sobre sua configuração, o Hibernate começa a imprimir as instruções SQL que ele enviou para o banco, como a do Código-fonte *Exemplo de DDL enviada pelo Hibernate para o banco, solicitando a criação da tabela “estabelecimento”.*

```
Hibernate:

    create table estabelecimento (
       id_estabelecimento integer generated by default
as identity,
        dh_criacao datetime null,
        nome_estabelecimento varchar(50) not null,
        primary key (id_estabelecimento)
    )
```

Código-fonte 8.11 – Exemplo de DDL enviada pelo Hibernate
para o banco, solicitando a criação da tabela “estabelecimento”
Fonte: Elaborado pelo autor (2017)

O Código-fonte acima é uma DDL de criação de tabela. Você verá que o Hibernate gera instruções DDL quando é executado pela primeira vez após a criação de uma entidade mapeada para uma tabela que ainda não existe ou quando alteramos entidades já existentes.

8.1.6.2 Operações básicas de acesso ao banco com EntityManager

Como explicado, um objeto **EntityManager** é uma espécie de conexão. É nele que invocamos os métodos de acesso (consulta, inclusão, alteração e exclusão) ao banco de dados. A seguir, vamos ver como realizar as operações básicas com o banco usando o **EntityManager** e a entidade **Estabelecimento**. Todos os Códigos-fonte Java, do Exemplo de uso do método “persist()” do “EntityManager” ao Provável trecho da saída no console da execução da JPATeste contendo um método “remove()”, devem ser colocados dentro do bloco **try** da classe **JPATeste** (a do Código-fonte *Classe com o código suficiente para testar o projeto JPA*).

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

8.1.6.2.1 Salvando um novo Estabelecimento

Para criar um registro no banco a partir de uma entidade JPA, basta criar um objeto, preencher com os atributos obrigatórios e usar o método **persist()** do **EntityManager** (veja o exemplo no Código-fonte *Exemplo de uso do método "persist()" do "EntityManager"*).

```
import br.com.fiap.smartcities.model;

public class JPATeste {

     public static void main(String[] args) {
        ...
        try {
           // criação do EntityManager (em)

           em.getTransaction().begin();

           Estabelecimento novo = new Estabelecimento();
           novo.setNome("Escolinha Imaginação");

           em.persist(novo);

           em.getTransaction().commit();

        } catch (Expcetion e) {

           // código que já existia no catch
           em.getTransaction().rollback();

        }
     }
}
```

Código-fonte 8.12 – Exemplo de uso do método "persist()" do "EntityManager"
Fonte: Elaborado pelo autor (2017)

Note que, logo após a criação do **EntityManager em**, solicitamos que inicie uma transação por meio do código **em.getTransaction().begin()**. Isso é necessário quando fazemos operações que inserem, alteram ou excluem registros no banco. Como estamos testando a inserção de um registro, foi necessário iniciar a transação. Ao final do bloco **try**, caso nenhuma exceção ocorra, a transação é confirmada (sobre o *commit*) devido ao código **em.getTransaction().commit()**. No bloco **catch**, ao qual o programa chegará caso algum erro ocorra, a transação é desfeita (sobre o *rollback*) devido ao código **em.getTransaction().rollback()**.

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

Ao executar a classe **JPATeste**, a saída no console deverá conter um trecho parecido com o do Código-fonte *Provável trecho da saída no console da execução da JPATeste com o método "persist()" do EntityManager*, com um DML de **insert** para o banco de dados.

```
INFO: HHH10001501: Connection obtained from ...

Hibernate:
    insert
    into
        estabelecimento
        (dh_criacao, nome_estabelecimento)
    values
        (?, ?)
```

Código-fonte 8.13 – Provável trecho da saída no console da execução da JPATeste com o método "persist()" do EntityManager
Fonte: Elaborado pelo autor (2017)

Se nenhuma exceção for lançada, ao consultar o banco com seu cliente favorito o registro deve constar na tabela **estabelecimento**.

Repare que não precisamos indicar um valor para o atributo **id** do **novo** (objeto de **Estabelecimento**). Isso porque, como explicado na seção **@GeneratedValue (JPA)** do capítulo anterior, ele receberá um valor seguro e nunca usado automaticamente.

8.1.6.2.2 Recuperando um Estabelecimento que já existe no banco a partir de sua Chave Primária

Para recuperar um registro do banco a partir de uma entidade JPA e de sua Chave Primária, basta usar o método **find()** do **EntityManager** (vide Código-fonte *Exemplo de uso do método "find()" do "EntityManager"*).

```
import br.com.fiap.smartcities.model;

public class JPATeste {

     public static void main(String[] args) {
        ...
        try {
           // criação do EntityManager (em)
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
                Estabelecimento recuperado =
                   em.find(Estabelecimento.class, 1);


             } catch ...

             }
          }
    }
```

Código-fonte 8.14 – Exemplo de uso do método “find()” do “EntityManager”
Fonte: Elaborado pelo autor (2017)

O método **find()** tenta recuperar um registro da tabela mapeada para a entidade **Estabelecimento** (tabela **estabelecimento**) a partir da chave de valor **1**. Caso exista um registro de **estabelecimento** com o campo **id_estabelecimento** (que é o campo da Chave Primária) igual a **1**, o JPA recupera esse registro, instancia um objeto de **Estabelecimento**, preenche todos os seus atributos conforme os campos da tabela e o devolve pronto para usarmos como quisermos. Caso não exista registro com a Chave Primária **1**, o **find()** devolve **null**.

No Código-fonte *Exemplo de uso do método “find()” do “EntityManager”,* o objeto **recuperado** seria criado com todos os atributos preenchidos conforme o registro encontrado ou receberá apenas **null**, conforme explicado no parágrafo anterior. Se preferir, altere a JPATeste para solicitar o valor da Chave Primária (via *Console* ou *JOptionPane*, como preferir).

Um detalhe importante: como não fizemos alterações em registro, não é necessário iniciar, nem fazer *commit* ou *rollback* em uma transação.

Ao executar a classe **JPATeste**, a saída no console deverá conter um trecho parecido com o do Código-fonte *Provável trecho da saída no console da execução da JPATeste com método “find()” do EntityManager*, com um DML de **select** para o banco de dados.

```
INFO: HHH10001501: Connection obtained from ...

Hibernate:
select
   estabeleci0_.id_estabelecimento as id_estab1_0_0_,
   estabeleci0_.dh_criacao as dh_criac2_0_0_,
   estabeleci0_.nome_estabelecimento as nome_est3_0_0_
from
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
        estabelecimento estabeleci0_
    where
        estabeleci0_.id_estabelecimento=?
```

Código-fonte 8.15 – Provável trecho da saída no console
da execução da JPATeste com método “find()” do EntityManager
Fonte: Elaborado pelo autor (2017)

8.1.6.2.3 Alterando um estabelecimento que já existe no banco após recuperá-lo

Lembra que dissemos que o **EntityManager** é uma espécie de conexão? Ele é mais que uma conexão, literalmente **gerencia** os objetos das entidades. O objeto **recuperado** do Código-fonte *Exemplo de uso do método “find()” do “EntityManager”*, por exemplo, como foi criado a partir de uma consulta JPA, está gerenciado. Isso significa que, caso qualquer atributo seu sofra alterações, quando a transação aberta sofrer *commit*, o próprio JPA enviará a instrução de ***update*** para o banco. Não é necessário invocar nenhum método adicional para isso!

Porém, para que essa “mágica” ocorra, é necessário iniciar uma transação. Afinal, estamos lidando com alterações em registros. Veja o Código-fonte *Exemplo de alteração de objeto gerenciado pelo “EntityManager”,* em que incluímos novamente os códigos de transação antes de recuperar o registro do banco.

```
import br.com.fiap.smartcities.model;

public class JPATeste {

    public static void main(String[] args) {
        ...
        try {
            // criação do EntityManager (em) e inicio da
transação

            Estabelecimento recuperado =
               em.find(Estabelecimento.class, 1);

            recuperado.setNome(“Escola Magic”);

             // commit
        } catch ...
            // rollback
        }
    }
}
```

Código-fonte 8.16 – Exemplo de alteração de objeto gerenciado pelo “EntityManager”

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

Fonte: Elaborado pelo autor (2017)

Usamos um **setNome()** para alterar o nome do recuperado. Como o recuperado está gerenciado pelo **em** (**EntityManager**), o Hibernate enviará um ***update*** como o do Código-fonte *Provável trecho da saída no console da execução da JPATeste com uma entidade gerenciada que sofreu alterações*, pouco antes do *commit*.

```
Hibernate:
   update
      estabelecimento
   set
      dh_criacao=?,
      nome_estabelecimento=?
   where
      id_estabelecimento=?
```

Código-fonte 8.17 – Provável trecho da saída no console da execução da JPATeste com uma entidade gerenciada que sofreu alterações
Fonte: Elaborado pelo autor (2017)

Se a entidade tivesse, por exemplo, 20 atributos e alterássemos o valor de 7 de uma instância gerenciada, o Hibernate enviaria um ***update*** para todos os 7 campos alterados na tabela.

8.1.6.2.4 Excluindo um Estabelecimento que já existe no banco após recuperá-lo

Para excluir um registro a partir de um objeto gerenciado e recuperado via JPA, basta usá-lo como argumento do método **remove()** do **EntityManager**. Também é necessário o uso de transação (iniciar, *commit* ou *rollback*), o Código-fonte *Exemplo de uso do método "remove()" do "EntityManager"* mostra como isso pode ser feito.

```
import br.com.fiap.smartcities.model;

public class JPATeste {

     public static void main(String[] args) {
        ...
        try {
           // criação do EntityManager (em) e inicio da
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
    transação

                Estabelecimento recuperado =
                   em.find(Estabelecimento.class, 1);

                em.remove(recuperado);

                 // commit
             } catch ...

                // rollback
             }
          }
    }
```

Código-fonte 8.18 – Exemplo de uso do método “remove()” do “EntityManager”
Fonte: Elaborado pelo autor (2017)

Após a execução do **remove()**, o Hibernate envia uma instrução ***delete*** para o banco como o do Código-fonte *Provável trecho da saída no console da execução da JPATeste contendo um método “remove()”*.

```
    Hibernate:
       delete
       from
          estabelecimento
       where
          id_estabelecimento=?
```

Código-fonte 8.19 – Provável trecho da saída no console
da execução da JPATeste contendo um método “remove()”
Fonte: Elaborado pelo autor (2017)

Se a entidade tivesse, por exemplo, 20 atributos e alterássemos o valor de 7 de uma instância gerenciada, o Hibernate enviaria um ***update*** para todos os 7 campos alterados na tabela.

É possível realizar muitos outros tipos de operações com o banco e o **EntityManager**. Muitas delas serão explicadas e demonstradas no próximo capítulo.

Nesta seção, vimos como mapear as entidades de ORM no JPA e como usar as operações básicas dele para acessar o banco de dados. Tenha sempre cuidado ao acessar bancos de dados compartilhados em uma empresa. Se possível, use bancos de dados locais em seu computador para realizar os testes.

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

8.2 Relacionamentos

8.2.1 Introdução

É muito difícil imaginar um sistema de informação, por menor que seja, que use apenas uma tabela de um banco de dados relacional. Aliás, os bancos de dados relacionais possuem esse nome pela capacidade explícita de relacionar várias tabelas a fim de garantir, entre outras coisas, a normalização e a consistência de dados. O relacionamento entre tabelas também pode ser mapeado para objetos com o JPA.

Nesta seção, veremos como implementar os principais tipos de relacionamentos.

No capítulo anterior, usamos como exemplo uma classe chamada **Estabelecimento**. Vamos supor que agora precisamos agrupar os estabelecimentos do sistema por tipo para facilitar as pesquisas dos usuários e a geração de relatórios. Assim, entra a tabela **tipo_estabelecimento**, que está ligada à tabela **estabelecimento** por meio de uma **FK** (**Foreign Key** ou **Chave Estrangeira**). O relacionamento entre essas tabelas está representado na Figura *Diagrama Entidade-Relacionamento entre "estabelecimento" e "tipo_estabelecimento".*

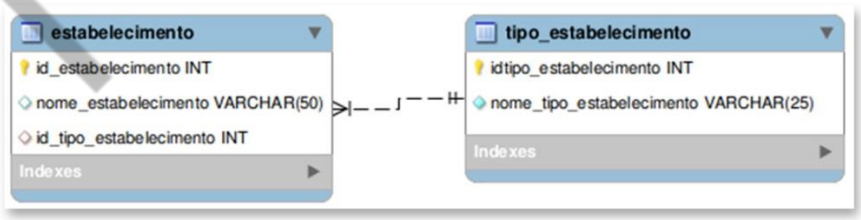


Figura 8.16 – Diagrama Entidade-Relacionamento entre "estabelecimento" e "tipo_estabelecimento"
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

Ou seja, o mapeamento configura que um estabelecimento é de um **tipo_estabelecimento** e que um **tipo_estabelecimento** pode ser de vários **estabelecimentos**.

Vejamos a seguir como mapear esse relacionamento em entidades JPA.

8.2.2 Muitos para Um (ManyToOne)

Primeiro, vamos criar a classe de ORM para a tabela **tipo_estabelecimento**. Um nome apropriado para a classe seria **TipoEstabelecimento**, conforme explicado na seção "**Nomes de tabelas e campos x nomes de classes e atributos**" do capítulo anterior. Um exemplo de como seu código ficaria pode ser visto no Código-fonte *Entidade "TipoEstabelecimento", que mapeia a tabela "tipo_estabelecimento"*.

```
@Entity
@Table(name = "tipo_estabelecimento")
public class TipoEstabelecimento {

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

     @Column(name = "nome_tipo_estabelecimento",
length=25,
nullable=false)
     private String nome;

     // construtores, getters e setters
```

Código-fonte 8.20 – Entidade "TipoEstabelecimento", que mapeia a tabela "tipo_estabelecimento"
Fonte: Elaborado pelo autor (2017)

Vejamos agora a alteração que precisa ser feita da classe **Estabelecimento**, para que o JPA conheça o relacionamento entre ela e **TipoEstabelecimento** no Código-fonte *Entidade "Estabelecimento" configurada para se relacionar com a "TipoEstabelecimento"*.

```
// imports que já existiam

import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
public class Estabelecimento {

      // atributos que já existiam

      @JoinColumn(name = "id_tipo_estabelecimento")
      @ManyToOne
      private TipoEstabelecimento tipo;

      // construtores, getters e setters

}
```

Código-fonte 8.21 – Entidade “Estabelecimento” configurada para se relacionar com a “TipoEstabelecimento”
Fonte: Elaborado pelo autor (2017)

A configuração do relacionamento foi feita por meio das anotações **@ManyToOne** e **@JoinColumn** sobre o atributo **tipo**, do tipo **TipoEstabelecimento.**

Com a anotação **@ManyToOne**, indicamos que podem haver **muitos** objetos do tipo **Estabelecimento** associados a **um** mesmo **TipoEstabelecimento**. Afinal, do estabelecimento de exemplo “Hotel Familiar da Luz” e “Mega Hotel da Sombra” seriam ambos do tipo de estabelecimento “Hotel”.

Com a anotação **@JoinColumn**, mostramos qual o campo na tabela **estabelecimento** deve estar na Chave Estrangeira para a tabela **tipo_estabelecimento**. O nome do campo está no atributo **name** dessa anotação.

Caso as tabelas já existam no banco de dados e o relacionamento também, o JPA faz uma breve análise para ver se a cardinalidade entre as tabelas está compatível com a configuração feita nas entidades. Se não estiver, uma exceção será lançada assim que tentarmos criar o **EntityManager**.

Caso alguma ou nenhuma das tabelas exista ou a Chave Estrangeira não existir, o Hibernate enviará as instruções SQL necessárias para a criação das tabelas e/ou da Chave Estrangeira, se a propriedade **hibernate.hbm2ddl.auto** do projeto estiver com o valor **update**.

FIAP

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

8.2.2.1 Indicando o TipoEstabelecimento de um Estabelecimento

A entidade **Estabelecimento**, de forma muito mais natural, possui um **TipoEstabelecimento**. Logo, ao usarmos o método **setTipo()** e como argumento um objeto do tipo **TipoEstabelecimento** recuperado do banco de dados pelo JPA, o Hibernate cria e envia a instrução necessária para que o registro da tabela **estabelecimento** possua o valor de **id_tipo_estabelecimento** correto. Seriam enviados um **update** ou um **insert**, caso o registro de estabelecimento já existisse ou se estivéssemos criando um novo, respectivamente.

8.2.2.2 Consultando o TipoEstabelecimento de um Estabelecimento

A entidade **Estabelecimento**, de forma muito mais natural, possui um **TipoEstabelecimento**. Logo, ao invocar o método **getTipo()** de um objeto do tipo **Estabelecimento**, obteremos, de forma transparente, o registro da tabela **tipo_estabelecimento** que tem a Chave Estrangeira para o campo **id_tipo_estabelecimento** na tabela **estabelecimento**. É o Hibernate que cria e envia para o banco as instruções ***select*** necessárias para que isso fique fácil no código Java.

8.2.3 Um para Muitos (OneToMany)

Assim como podemos dizer que um **Estabelecimento** é de um **TipoEstabelecimento**, podemos dizer que um **TipoEstabelecimento** pode ser usado por vários **Estabelecimentos**, certo? Como isso é uma associação comum e frequente a necessidade da recuperação dos N itens associados a um determinado registro via Chave Estrangeira, o JPA oferece uma configuração simples para recuperar esse tipo de relacionamento.

Basta utilizar a anotação **@OneToMany**. Um exemplo de como seu código ficaria pode ser observado no Código-fonte *Configuração na "TipoEstabelecimento" para que ela tenha mapeado quais registros "Estabelecimento" estão ligados a cada registro dela,* no qual acrescentamos mais um atributo à classe **TipoEstabelecimento**.

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
import javax.persistence.OneToMany;
import java.util.Collection;

// anotações já existentes
public class TipoEstabelecimento {

   // atributos já existentes

   @OneToMany(mappedBy = "tipo")
   private Collection<Estabelecimento> estabelecimentos;

   // construtores, getters e setters
```

Código-fonte 8.22 – Configuração na "TipoEstabelecimento" para que ela tenha mapeado quais registros "Estabelecimento" estão ligados a cada registro dela
Fonte: Elaborado pelo autor (2017)

Na anotação **@OneToMany**, usamos o atributo **mappedBy**, que indica por qual nome o **TipoEstabelecimento** é conhecido em **Estabalecimento**. No Código-fonte *Configuração na "TipoEstabelecimento" para que ela tenha mapeado quais registros "Estabelecimento" estão ligados a cada registro dela,* podemos ver que o atributo foi chamado de **tipo**, por isso, usamos essa palavra no **mappedBy**.

Note que o atributo **estabelecimentos** é uma **Collection**. Isso serve para representar de forma orientada objetos e a cardinalidade – "um mesmo tipo de estabelecimento pode ser de vários estabelecimentos". Afinal, podem haver vários: "hotel", "escola", "academia", entre outros. E com esse atributo devidamente anotado com **@OneToMany**, sempre que recuperarmos um **TipoEstabelecimento** usando um **EntityManager** e invocarmos o set **getEstabelecimentos()**, o Hibernate enviará automaticamente para o banco um novo ***select*** solicitando todos os registros da tabela **estabelecimento** relacionados com o registro de **tipo_estabelecimento** em questão. Além disso, preencherá a coleção com instâncias de Estabelecimento conforme os registros encontrados ou com um a **coleção vazia**, caso não exista nenhum registro relacionado.

Nesta seção, vimos como é possível mapear as Chaves Estrangeiras em entidades ORM com JPA. Em projetos para tabelas que já existem, verifique quais relacionamentos realmente precisam ser mapeados antes de criar mapeamentos para todas as Chaves Estrangeiras.

FIAP

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

8.3 Chave Primária Composta

8.3.1 Introdução

Algumas tabelas possuem suas Chaves Primárias (*Primary Keys*) compostas, ou seja, não é apenas um campo que compõe a Chave Primária. Uma Chave Composta pode ter dois ou mais campos, conforme necessário.

Quando surge essa necessidade, é preciso criar uma classe de ORM exclusivamente para representar a composição de uma Chave Primária Composta. Feito isso, é necessário indicar na entidade mapeada para a tabela que se está usando uma classe de Chave Composta.

8.3.2 Como mapear uma Chave Composta

Tomemos como exemplo a seguinte situação: temos a tabela **avaliacao**, cuja Chave Primária é composta pelos campos **id_usuario** e **id_estabelecimento**, os quais também são Chaves Estrangeiras para as tabelas **usuario** e **estabelecimento**, respectivamente. Detalhes sobre essa tabela podem ser vistos na Figura *Tabela "avaliacao", sua Chave Primária e seus relacionamentos.*

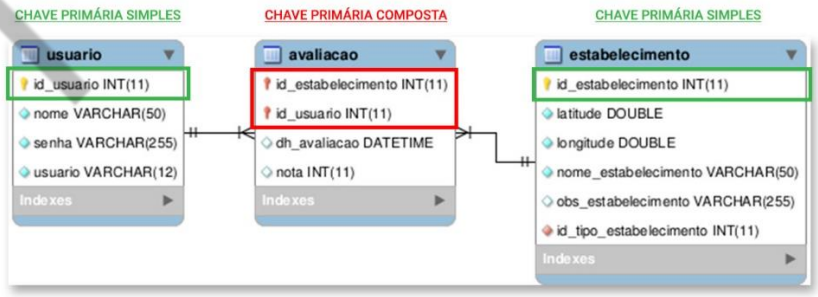


Figura 8.17 – Tabela "avaliacao", sua Chave Primária e seus relacionamentos
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

Para fazer o mapeamento objeto relacional desse exemplo, vamos primeiro criar a classe que mapeará a Chave Primária da tabela avaliação. Para isso, vamos criar a classe **AvaliacaoId** no pacote de entidades. Veja o exemplo no Código-fonte *Classe que mapeia a Chave Primária composta da tabela "avaliacao".*

```
import javax.persistence.Embeddable;

@Embeddable
public class AvaliacaoId implements Serializable {

      @Column(name = "id_usuario")
      private Integer usuarioId;

      @Column(name = "id_estabelecimento")
      private Integer estabelecimentoId;

     // getters e setters

     // equals() e hashCode()
}
```

Código-fonte 8.23 – Classe que mapeia a Chave Primária Composta da tabela "avaliacao"
Fonte: Elaborado pelo autor (2017)

O que torna essa classe uma entidade de mapeamento de Chave Primária Composta é a anotação **@Embeddable** sobre a assinatura da classe. Seu caminho completo está no **import** do início do código.

Um detalhe muito importante é que toda classe de mapeamento de Chave Primária Composta **deve, obrigatoriamente, sobrescrever** os métodos **equals()** e **hashCode()**. O Eclipse possui um assistente que faz isso para nós com alguns cliques. Após criar os atributos e seus *getters* e *setters*, basta clicar com o botão direito do mouse em qualquer parte do código e escolher as opções **Source** -> **Generate hashCode() and equals()** (vide Figura *Geração de "equals()" e "hashCode()" a partir do menu de contexto do código-fonte*). O mesmo assistente também pode ser invocado acessando o menu **Source** do Eclipse (Figura *Geração de "equals()" e "hashCode()" a partir do menu "Source"*).

FIAP

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

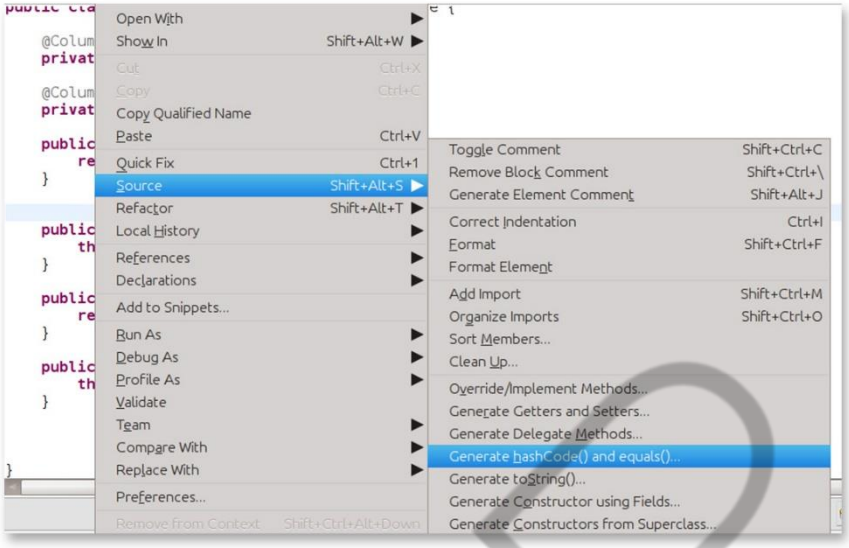


Figura 8.18 – Geração de “equals()” e “hashCode()” a partir do menu de contexto do código-fonte
Fonte: Elaborado pelo autor (2017)

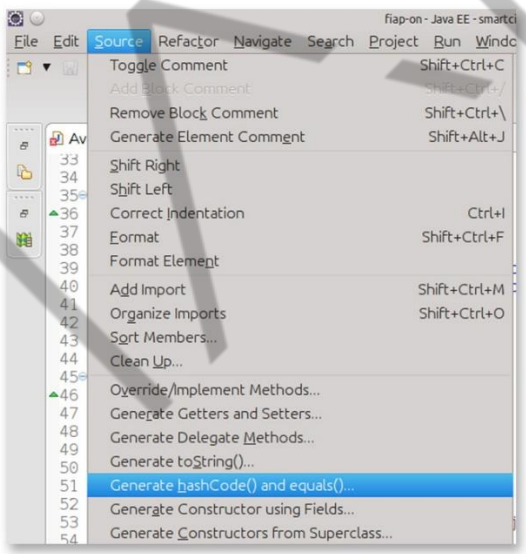


Figura 8.19 – Geração de “equals()” e “hashCode()” a partir do menu “Source”
Fonte: Elaborado pelo autor (2017)

No assistente, marque todos os campos e indique a opção **Last member** no item **Insertion point** (vide Figura *Assistente de Geração de “equals()” e “hashCode()”*). Basta, então, clicar em **OK**.

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

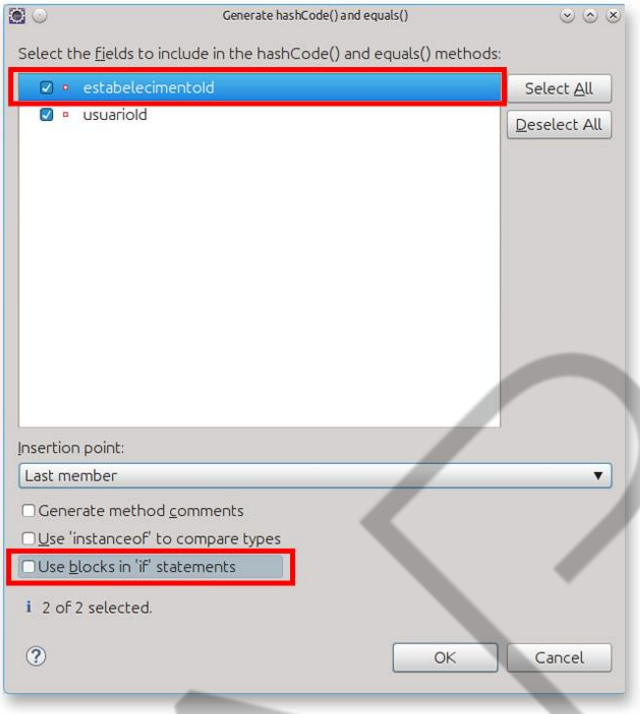


Figura 8.20 – Assistente de Geração de “equals()” e “hashCode()”.
Fonte: Elaborado pelo autor (2017)

O código gerado por esse assistente será como o do Código-fonte *“equals()” e “hashCode()” gerados na classe “AvaliacaoId.*

```
    @Override
    public int hashCode() {
         final int prime = 31;
         int result = 1;
         result = prime * result + ((estabelecimentoId ==
    null) ? 0 : estabelecimentoId.hashCode());
              result = prime * result + ((usuarioId == null)
    ? 0 : usuarioId.hashCode());
              return result;
    }

    @Override
    public boolean equals(Object obj) {
         if (this == obj)
              return true;
         if (obj == null)
              return false;
         if (getClass() != obj.getClass())
              return false;
         AvaliacaoId other = (AvaliacaoId) obj;
         if (estabelecimentoId == null) {
              if (other.estabelecimentoId != null)
                   return false;
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
        } else if
      (!estabelecimentoId.equals(other.estabelecimentoId))
            return false;
        if (usuarioId == null) {
            if (other.usuarioId != null)
                return false;
        } else if (!usuarioId.equals(other.usuarioId))
                return false;
            return true;
        }
```

Código-fonte 8.24 – “equals()” e “hashCode()” gerados na classe “AvaliacaoId
Fonte: Elaborado pelo autor (2017)

Os métodos **equals()** e **hashCode()** são usados pelo Java para saber se duas instâncias de objetos são o “mesmo” objeto. Se não sobrescritos, ele considera se os objetos comparados são o mesmo objeto em memória. Se sobrescritos, como foi feito aqui, ele usa o critério programado neles para saber se dois objetos são “iguais”. Se não forem sobrescritos em todas as classes que mapeiam as Chaves Primárias Compostas, o JPA lança uma exceção logo no início da execução do programa.

Após mapear a Chave Primária Composta, devemos indicar na entidade que mapeia a tabela **avaliacao** que ela está usando nesse mapeamento. Faremos isso na classe **Avaliacao**, cujo código está no Código-fonte *Classe “Avaliacao” e seu uso da “AvaliacaoId” como Chave Primária.*

```
import javax.persistence.EmbeddedId;

@Entity
@Table(name = "avaliacao")
public class Avaliacao implements Serializable {

    @EmbeddedId
    private AvaliacaoId id;

    @JoinColumn(name = "id_usuario", insertable =
false, updatable = false)
    @ManyToOne(optional = false)
    private Usuario usuario;

    @JoinColumn(name = "id_estabelecimento", insertable
= false, updatable = false)
    @ManyToOne(optional = false)
    private Estabelecimento estabelecimento;

    @Enumerated(EnumType.ORDINAL)
    @Column(length = 1)
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
        private NotaAvaliacao nota;

        // @CreationTimestamp
        @Temporal(TemporalType.TIMESTAMP)
        @Column(name = "dh_avaliacao")
        private Calendar dataAvaliacao;

        // getters e setters
    }
```

Código-fonte 8.25 – Classe “Avaliacao” e seu uso da “AvaliacaoId” como Chave Primária
Fonte: Elaborado pelo autor (2017)

O que faz com que essa classe use como Chave Primária a **AvaliacaoId** é a anotação **@EmbeddedId** sobre o atributo **id**. Seu caminho completo está no **import** do início do código.

Um detalhe muito importante é que os relacionamentos com **Usuario** e **Estabelecimento** precisam ser configurados com **insertable=false** e **updatable=false**. Isso é imposição do JPA, para que não haja ambiguidade sobre onde configurar os valores do **id** de **Usuario** e **Estabelecimento**, pois só podem ser informados no atributo **id**, que é uma instância de **AvaliacaoId**. Se esses dois atributos não forem configurados, o JPA lançará uma exceção assim que o programa iniciar.

Para recuperar os valores do **Usuario** e do **Estabelecimento** associados a uma **Avaliacao**, basta usar os métodos **getUsuario()** e **getEstabelecimento()**, respectivamente.

8.3.2.1 Como usar entidades com Chave Composta

Já vimos como mapear uma Chave Composta com JPA. Veremos agora como os atributos de **Avaliacao** e **AvaliacaoId** devem ser preenchidos para que a Chave Composta e os relacionamentos funcionem corretamente.

Quando se deseja **criar** um registro de **Avaliacao**, deve-se primeiro criar uma instância de **AvaliacaoId** e preencher seus atributos com ids de instâncias de **Usuario** e **Estabelecimento** recém-criados ou recuperados do banco de dados. O próximo passo é usar a instância de **AvaliacaoId** recém-criada como o atributo **id** de uma instância de **Avaliacao,** conforme o exemplo do Código-fonte *Criação de instância de “AvaliacaoId” e uso em uma instância de “Avaliacao”.*

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
Usuario usuario = // novo ou recuperado
Estabelecimento estabelecimento = // novo ou recuperado

AvaliacaoId idNovo = new AvaliacaoId();
idNovo.setUsuarioId(usuario.getId());
idNovo.setEstabelecimentoId(estabelecimento.getId());

Avaliacao novaAvaliacao = new Avaliacao();
novaAvaliacao.setId(idNovo);
```

Código-fonte 8.26 – Criação de instância de “AvaliacaoId” e uso em uma instância de “Avaliacao”
Fonte: Elaborado pelo autor (2017)

Assim, o objeto **novaAvaliacao** está pronto para ser persistido no banco de dados pelo JPA. Observe que não é preciso atribuir os valores dos atributos **usuario** e **estabelecimento** para a criação de um novo registro de **Avaliacao**.

Como vimos nesta seção, é possível mapear Chaves Primárias Compostas em entidades ORM com o JPA. Chaves Primárias Compostas estão cada vez mais caindo em desuso, por isso, evite usá-las em projetos novos.

8.4 Herança

8.4.1 Introdução

Um dos recursos mais interessantes nas linguagens orientadas a objetos, como Java, é a possibilidade de herdar características entre classes. Quando queremos que uma classe herde de outra, criamos uma **subclasse** (nomenclatura usada pelo Java) ou uma **classe derivada** (nomenclatura usada pelo C#). A classe da qual herdam é sua **superclasse** (nomenclatura usada pelo Java) ou **classe base** (nomenclatura usada pelo C#).

Podemos usar esse recurso também no ORM quando temos classes parecidas e/ou quando tabelas foram “divididas” em várias por questões de normalização. Mas essas divisões só tornam a programação mais trabalhosa em linguagens orientadas a objetos.

Um exemplo clássico disso é quando se tem o cenário “um sistema precisa cadastrar clientes. Um cliente pode ser pessoa física ou jurídica. Se pessoa física, deve-se cadastrar seu estado civil e escolaridade. Se pessoa jurídica, deve-se cadastrar sua inscrição estadual e nome fantasia”. Nesse contexto, os profissionais

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

de banco de dados costumam criar três tabelas com nomes como estes: **cliente**, **cliente_pf**, **cliente_pj**, conforme o diagrama da Figura *Conceito de cliente "dividido" em três tabelas para fins de normalização.*

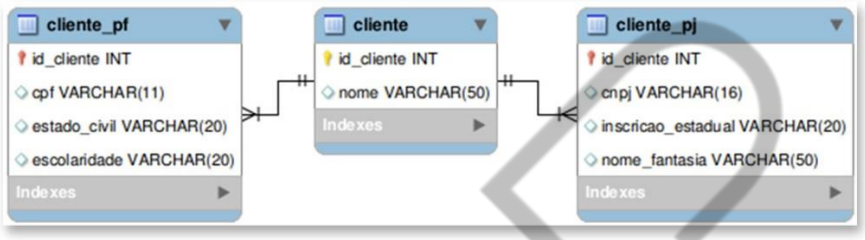


Figura 8.21 – Conceito de cliente "dividido" em três tabelas para fins de normalização
Fonte: Elaborado pelo autor (2017)

Para fazer o mapeamento do objeto relacional desse exemplo, podemos usar três estratégias que o JPA nos oferece. Uma que usaria as tabelas exatamente como estão na Figura *Conceito de cliente "dividido" em três tabelas para fins de normalização* e outras que levariam a diferentes propostas de modelagem.

Independentemente da estratégia adotada, precisaremos de três classes: **Cliente**, **ClientePf** e **ClientePj**, descritas nos Códigos-fonte Entidade "Cliente"; Entidade "ClientePf"; e Entidade "ClientePj", respectivamente.

```
import javax.persistence.Inheritance;

@Entity
@Table(name="cliente")
@Inheritance
public class Cliente {

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

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
        @Column(length = 50)
        private String nome;
```

Código-fonte 8.27 – Entidade “Cliente”
Fonte: Elaborado pelo autor (2017)

```
@Entity
@Table(name="cliente_pf")
public class ClientePf extends Cliente {

     @Column(name="estado_civil", length = 20)
     private String estadoCivil;

     @Column(length = 20)
     private String escolaridade;

     // contrutores, getters e setters
}
```

Código-fonte 8.28 – Entidade “ClientePf”
Fonte: Elaborado pelo autor (2017)

```
@Entity
@Table(name="cliente_pj")
public class ClientePj extends Cliente {

     @Column(name="inscricao_estadual", length = 20)
     private String inscricaoEstadual;

     @Column(length = 50)
     private String nomeFantasia;

     // contrutores, getters e setters
}
```

Código-fonte 8.29 – Entidade “ClientePj”.
Fonte: Elaborado pelo autor (2017)

Quanto à entidade **Cliente**, note que ela está anotada com **@Inheritance**. Essa anotação mostra que tal classe pode ser estendida por outra(s). Ela possui um atributo **strategy**, que é o que define a estratégia de herança usada. Ele será explicado e exemplificado mais à frente.

As entidades **ClientePj** e **ClientePj** estendem de **Cliente**, sendo, portando, subclasses de **Cliente**. É necessário incluir todas três classes no arquivo **persistence.xml**, conforme é orientado na seção **“Persistence.xml – Classes de ORM (entidades)”** deste capítulo.

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

A seguir, vejamos as diferentes formas de programar um relacionamento usando herança com JPA.

8.4.2 Herança com a estratégia JOINED

Para o mapeamento, espere que as tabelas sejam como as da Figura *Conceito de cliente "dividido" em três tabelas para fins de normalização*, e usamos a herança de estratégia **JOINED**. Para ver como indicar essa estratégia na anotação **@Inheritance**, observe o exemplo do Código-fonte *Indicando a estratégia de herança "JOINED"*.

```
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Table(name="cliente")
@Inheritance(strategy=InheritanceType.JOINED)
public class Cliente {

   // atributos, construtores e métodos já existentes

}
```

Código-fonte 8.30 – Indicando a estratégia de herança "JOINED"
Fonte: Elaborado pelo autor (2017)

Ou seja, a estratégia **JOINED** define que cada classe fica mapeada para uma tabela diferente, porém, **os atributos da tabela da superclasse não são repetidos nas tabelas das subclasses**. Nas tabelas das subclasses, a **Chave Primária** acaba sendo também uma **Chave Estrangeira** para a tabela da superclasse.

8.4.3 Herança com a estratégia TABLE_PER_CLASS

A estratégia **TABLE_PER_CLASS** (exemplificada no Código-fonte *Indicando a estratégia de herança "JOINED"*) funciona de forma muito parecida com a **JOINED**. A diferença é que todos os atributos nas diferentes tabelas se repetem, não importa que representem "a mesma" informação.

```
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
```

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

```
@Entity
@Table(name="cliente")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Cliente {

   // atributos, construtores e métodos já existentes

}
```

Código-fonte 8.31 – Indicando a estratégia de herança “JOINED”
Fonte: Elaborado pelo autor (2017)

Com a **TABLE_PER_CLASS**, as tabelas ficariam como na Figura Tabelas “cliente”, “cliente_pf” e “cliente_pj” compatíveis com herança “TABLE_PER_CLASS”, em que é possível notar que **não é criado nenhum relacionamento entre as tabelas**.

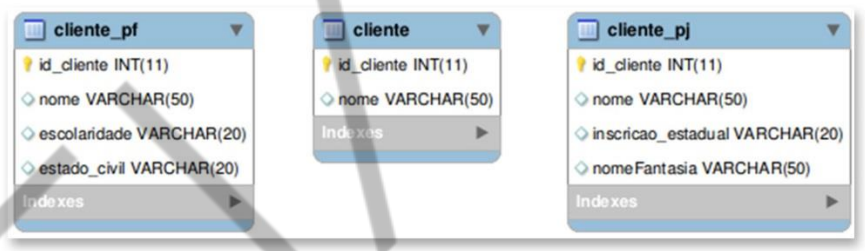


Figura 8.22 – Tabelas “cliente”, “cliente_pf” e “cliente_pj” compatíveis com herança “TABLE_PER_CLASS”
Fonte: Elaborado pelo autor (2017)

8.5 Herança com a estratégia SINGLE_TABLE

A estratégia **SINGLE_TABLE**, como o nome sugere, mostra que apenas **uma** tabela ficará mapeada para a superclasse e suas subclasses, cujo nome será o indicado na **@Table** da superclasse. Veja no Código-fonte *Indicando a estratégia de herança “SINGLE_TABLE”* como indicar essa estratégia.

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

Essa é a estratégia-padrão do JPA, caso o atributo **strategy** seja omitido na anotação **@Inheritance**.

```
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Table(name="cliente")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Cliente {

   // atributos, construtores e métodos já existentes

}
```

Código-fonte 8.32 – Indicando a estratégia de herança “SINGLE_TABLE”
Fonte: Elaborado pelo autor (2017)

Com a **SINGLE_TABLE**, as tabelas ficariam como na Figura *Tabela “cliente” compatível com herança “SINGLE_TABLE”,* em que é possível notar que **apenas uma tabela inclui todos os campos de todas as classes envolvidas no mapeamento com herança**. A anotação **@Table** nas subclasses é ignorada nessa estratégia.

Figura 8.23 – Tabela “cliente” compatível com herança “SINGLE_TABLE”
Fonte: Elaborado pelo autor (2017)

Repare que há um campo chamado **DTYPE** na tabela “cliente” que serve para indicar o “tipo” de cliente, de acordo com a subclasse usada para a criação do registro. Como temos **ClientePf** e **ClientePj**, os valores desse campo nos registros poderiam ser, literalmente, “ClientePf” e “ClientePj”. A Figura *Registros de uma*

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

tabela "cliente" compatível com herança "SINGLE_TABLE" contém o resultado da consulta aos registros de uma tabela **cliente** criada para ser compatível com a estratégia **SINGLE_TABLE**. Observe os valores do campo DTYPE na primeira coluna.

#	DTYPE	id_cliente	nome	inscricao_estadual	nomeFantasia	escolaridade	estado_civil
1	ClientePj	1	NULL	6876787866-3	Loja Bá	NULL	NULL
2	ClientePf	2	NULL	NULL	NULL	Superior Completo	Solteiro

Figura 8.24 – Registros de uma tabela "cliente" compatível com herança "SINGLE_TABLE"
Fonte: Elaborado pelo autor (2017)

Caso você queira que o campo que represente o tipo tenha outro nome, como **tipo_pessoa**, basta usar a anotação **@DiscriminatorColumn** sobre a entidade da superclasse (no caso, a Cliente). Veja o exemplo no Código-fonte *Indicando a estratégia de herança "SINGLE_TABLE" e o nome do campo que indica o tipo de "cliente".*

```
import javax.persistence.DiscriminatorColumn;

// demais anotações
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn( name="tipo_pessoa" )
public class Cliente {

   // atributos, construtores e métodos já existentes

}
```

Código-fonte 8.33 – Indicando a estratégia de herança
"SINGLE_TABLE" e o nome do campo que indica o tipo de "cliente"
Fonte: Elaborado pelo autor (2017)

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com

Nesta seção, vimos como é possível ter diferentes abordagens de distribuição de campos em tabelas mapeadas para classes que herdam atributos entre si.
Em projetos novos, converse com a equipe de banco de dados sobre quais são os níveis de normalização que desejam, para escolher a melhor estratégia.
Em projetos para bancos de dados já existentes, escolha a estratégia conforme as estruturas das tabelas.

FIAP

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.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: 20 out. 2017.

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

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

FIAP

PDF exclusivo para Daniel Fillol - rm346083
subidas.exibicao.0e@icloud.com