Banco de Dados com JPA

1 – Motivação

Desde o surgimento da linguagem java foram criadas diversas APIs para fazer a interface do código Java com um SGBD (Sistema Gerenciador de Banco de Dados). Uma das primeiras bibliotecas com esse intuito foi o JDBC(Java Database Connectivity), que surgiu no fim dos anos 90 e ainda é muito utilizado.

No JDBC, o acesso ao banco de dados é feito através de comandos (Statements) escritos na linguagem SQL (Structured Query Language), que é a linguagem padrão dos gerenciadores de banco de dados.

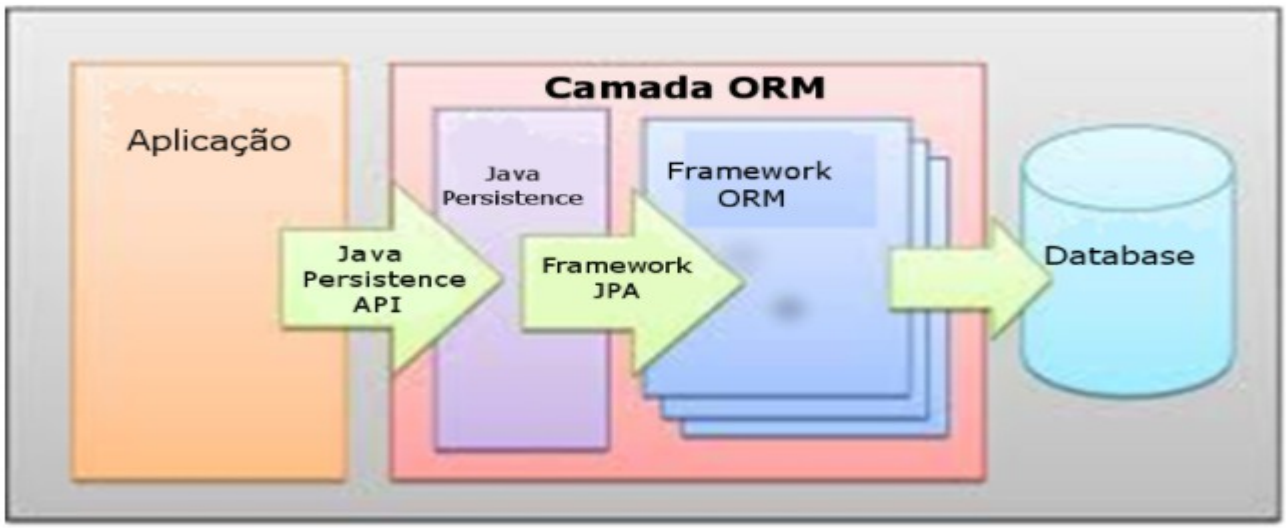
Em seguida surgiram as tecnologias de mapeamento objeto-relational, conhecidas por ORM (Object-Relational Mapping). Tais tecnologias estabelecem uma ponte entre esses dois modelos de dados: o modelo orientado a objetos, utilizado em linguagens de programação como o Java, e o modelo relacional, utilizado pela maioria dos SGBDs.

Deste modo, foram sendo desenvolvidas APIs que realizam esse mapeamento em Java. O hibernate foi uma das primeiras bibliotecas criadas com esse intuito. Com essas ferramentas, é possível fazer tanto a geração de código Java a partir das tabelas da base de dados, como a geração de scripts de criação de banco de dados (em SQL) a partir das classes Java.

2 – JPA

É uma especificação, e como uma especificação, ela preocupa-se com a persistência, o que significa vagamente qualquer mecanismo pelo qual os objetos Java sobrevivam ao processo do aplicativo que os criou. Nem todos os objetos Java precisam ser persistidos, mas a maioria dos aplicativos persiste os principais objetos de negócios.

Abaixo segue um diagrama sobre ORM – Object Relational Mapping (Mapeamento Objeto Relacional). Esse diagrama mostra o fluxo de funcionamento da API JPA, de onde a ela é inicialmente chamada, e até onde ela vai.



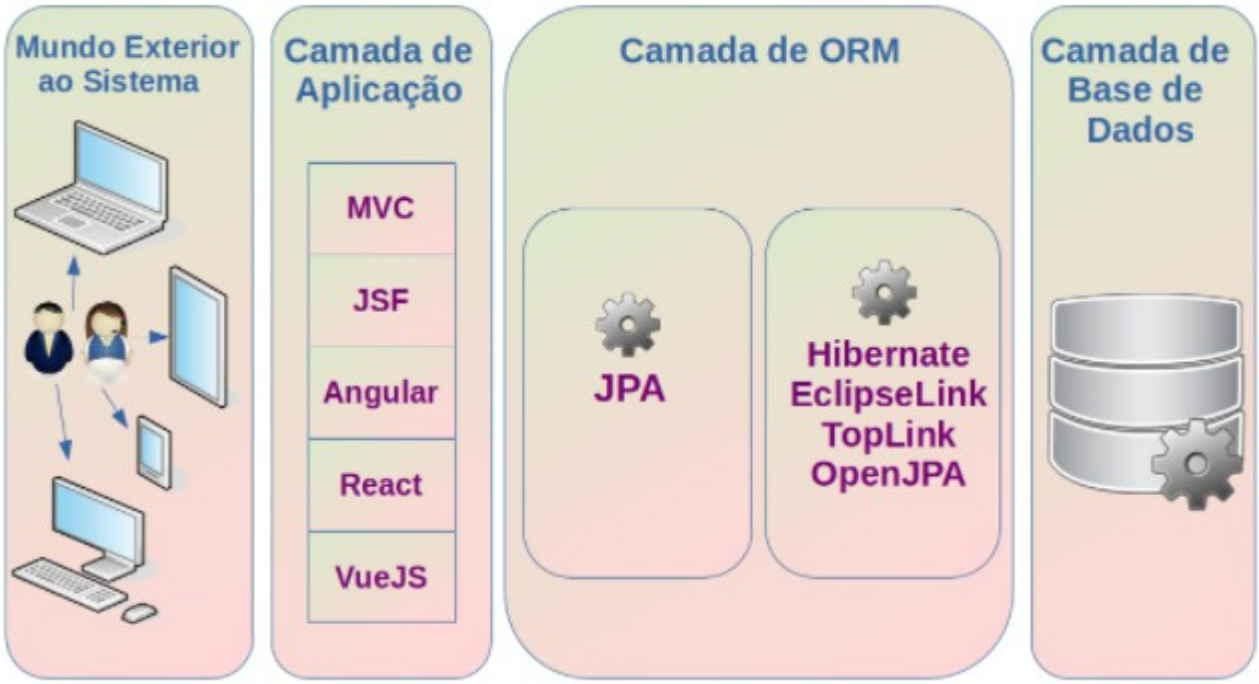
Como o primeiro framework Java para ORM foi o Hibernate, no início eram utilizados arquivos XML, os conhecidos hbm.xml, depois veio o XDoclet, antecessor das Annotations. Era utilizado com ejb's e o Hibernate.

Mas independente do framework ORM utilizado, seja o Hibernate, o EclipseLink, o TopLink, o OpenJPA, etc, isso não importa, a aplicação será portável para qualquer banco de dados que possua driver JDBC. E não será preciso reescrever o código-fonte, pois ele será o mesmo para todos os bancos de dados.

3 – Funcionamento da JPA

O JPA funciona através de qualquer framework ORM (Mapeamento Objeto Relacional) baseado na especificação JPA. Utilizando frameworks como Hibernate, EclipseLink, TopLink e OpenJpa.

Abaixo segue um diagrama bem detalhado sobre ORM – Object Relational Mapping (Mapeamento Objeto Relacional). Nesse diagrama podemos ver o mundo exterior ao sistema onde os usuários podem acessar a aplicação de qualquer dispositivo, seja um computador, um tablet, um smartphone ou um notebook, eles vão através da aplicação chamar a camada ORM, onde e localiza a interface JPA e o framework utilizado seja o Hibernate, o EclipseLink, o TopLink, o OpenJPA, etc. E nesse diagrama podemos ver detalhadamente a divisão entre a interface JPA e o framework ORM utilizado.



Na API JPA estaremos utilizando objetos que permitem a utilização deste processo como:

EntityManagerFactory – Interface usada para interagir com a fábrica do gerenciador de entidades para a unidade de persistência. Quando o aplicativo termina de usar a fábrica do gerenciador de entidades e/ou no encerramento do aplicativo, o aplicativo deve fechar a fábrica do gerenciador de entidades. Depois que um EntityManagerFactory é fechado, todos os seus gerenciadores de entidade são considerado no estado fechado.

EntityManager – Interface usada para interagir com o contexto de persistência. Uma instância de EntityManager está associada a um contexto de persistência. Um contexto de persistência é um conjunto de instâncias de entidade em que, para qualquer identidade de entidade persistente, existe uma instância de entidade única. Dentro do contexto de persistência, as instâncias de entidade e seu ciclo de vida são gerenciados. A API EntityManager é usada para criar e remover instâncias de entidade persistentes, para localizar entidades por sua chave primária e para consultar entidades.

O conjunto de entidades que podem ser gerenciadas por uma determinada instância EntityManager é definido por uma unidade de persistência. Uma unidade de persistência define o conjunto de todas as classes relacionadas ou agrupadas pelo aplicativo e que devem ser colocadas em seu mapeamento em um único banco de dados.

Para definição da JPA em nossos objetos java precisamos utilizar anotações definidas pela API JPA como:

@Entity – Serve para a API JPA saber que a classe anotada com @Entity corresponde a uma tabela da base de dados. Uma entidade corresponde a uma tabela.

@Table – É usada em conjunto com a anotação @Entity, serve para a API JPA saber o nome da tabela, em que a classe anotada com @Table, tem um atributo “name”, que contém o nome da tabela.

4 – JPA na Prática

Para testarmos a utilização de JPA na prática, iremos criar um projeto que permita o cadastro de livros, onde podemos classifica-los por tipo.

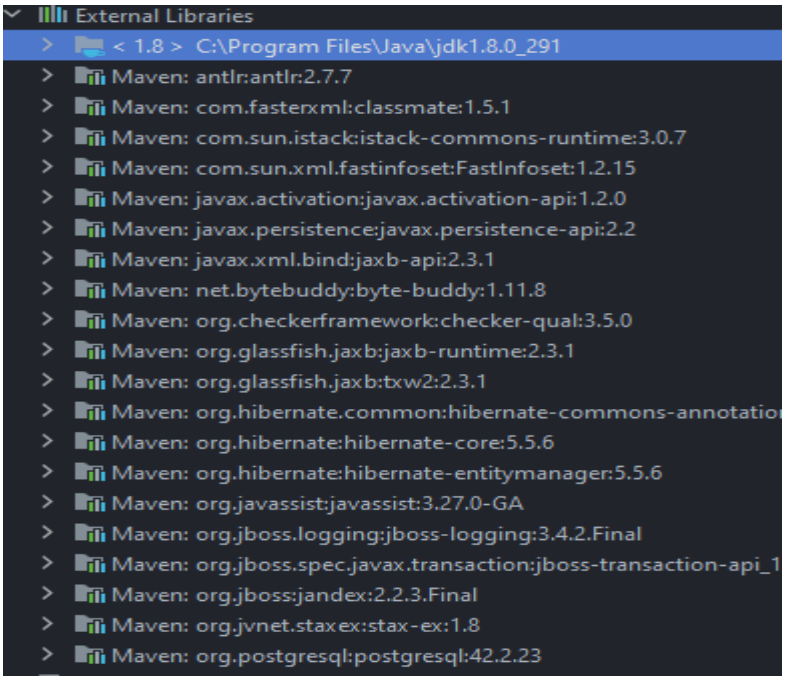
Então devemos a acessar o Postgres e criar o Banco de Dados BethaCodeLivros.

No IntelliJ iremos criar o projeto LivrosApp utilizando o MAVEN.

No arquivo POM. Xml, precisaremos adicionar as dependências relacionadas a JPA, postgres e ao Hibernate que será a implementação da JPA utilizada em nossa aplicação, ficando da seguinte forma:

```xml
<dependencies>
    <dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>javax.persistence-api</artifactId>
        <version>2.2</version>
    </dependency>

    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.5.6</version>
    </dependency>

    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.5.6</version>
    </dependency>

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.2.23</version>
    </dependency>
</dependencies>
```

Com a busca do MAVEN pelas dependências teremos o seguinte resultado:



Nossa próxima tarefa será criar a classe LivrosApp no pacote aplicacao e tentar utilizar os objetos das implementações da JPA. O EntityManagerFactory é responsável por criar a conexão com o banco de dados, enquanto o EntityManager tem a responsabilidade de persistir objetos java no banco de dados, referente a conexão recebida. Neste caso temos a indicação da minhaConexao.

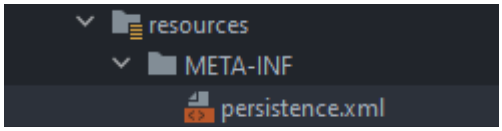
```java
package aplicacao;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class LivrosApp {
    public static void main(String[] args) {
        EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory( persistenceUnitName: "minhaConexao");
        EntityManager entityManager = entityManagerFactory.createEntityManager();
    }
}
```

Ao executar nossa aplicação teremos como resultado a seguinte informação:

```
"C:\Program Files\Java\jdk1.8.0_291\bin\java.exe" ...
set 19, 2021 5:15:50 PM org.hibernate.jpa.boot.internal.PersistenceXmlParser doResolve
INFO: HHH000318: Could not find any META-INF/persistence.xml file in the classpath
Exception in thread "main" javax.persistence.PersistenceException Create breakpoint : No Persistence provider for EntityManager named minhaConexao
    at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:85)
    at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:54)
    at aplicacao.LivrosApp.main(LivrosApp.java:9)
```

Isso ocorre, porque quando utilizamos JPA com o framework Hibernate, precisamos na pasta resource, criar a pasta META-INF e dentro desta pasta o arquivo persistence.xml com os dados de nossa conexão com o banco de dados.

Deve ser criado em resources a pasta META-INF, e na pasta META-INF o arquivo persistence.xml.



No arquivo persistence.xml, teremos as informações da nossa conexão com o banco. Poderá ser verificado que temos o persistence-unit, pois podemos ter várias conexões configuradas e identificar qual será utilizada em nossa aplicação.

Podemos ainda adicionar uma descrição a nossa conexão, pela tag description. Porém o mais importante é o conteúdo de properties, onde estarão os dados de nossa conexão como url, usuário, senha e driver.

Ainda teremos algumas configurações diferentes nessa lista:

-hibernate.show_sql : Esta configuração permite que a cada persistência realizada pelo hibernate, sejam demonstrados os sqls aplicados no banco.

-hibernate.format_sql: Esta configuração está ligada a anterior, para demonstrar estes sqls formatados.

-hibernate.hbm2ddl.auto: Esta configuração permite que a estrutura de banco, seja criada ao executar a aplicação. No caso será criado as tabelas e colunas no banco de dados, conforme o mapeamento das entidades na aplicação.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence 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_2.xsd"
             version="2.2">
    <persistence-unit name="minhaConexao">
        <description>Unidade de Conexão do LivroApp do Curso Betha Code</description>
        <!-- Configuracoes de conexao ao banco de dados -->
        <properties>
            <!-- Configuracoes do banco de dados -->
            <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost/BethaCodeLivros" />
            <property name="javax.persistence.jdbc.user" value="postgres" />
            <property name="javax.persistence.jdbc.password" value="postgres" />
            <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="create" />
        </properties>
    </persistence-unit>
</persistence>
```

Depois de criado o persistence.xml, não será gerado erro ao executar a aplicação. Nossa próxima etapa será a criação de nossas entidades e o seu devido mapeamento.

Iremos criar a classe Tipo no pacote modelo. Esta classe terá os atributos id Integer e descricao String. Para identificar que esta é uma entidade JPA (persistível), precisamos adicionar o @Entity na classe.

Quanto aos atributos id e descricao, precisaremos adicionar as propriedades JPA para estas.

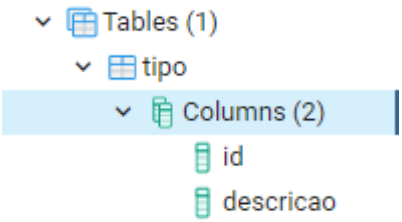
No atributo id, iremos adicionar @Id para indicar ser a chave da tabela tipo e o tipo de geração deste id @GeneratedValue(strategy = GenerationType.AUTO).

No atributo descrição, iremos informar que será uma coluna, com tamanho 100 e não pode ser nula. @Column(nullable = false, length = 100). Também deve ser criados os getter e setter para a classe.

```java
import javax.persistence.*;

@Entity
public class Tipo {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    @Column(nullable = false, length = 100)
    private String descricao;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
```

Desta forma ao executar novamente a aplicação a estrutura será criada.

```
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [
Hibernate:

    drop sequence if exists hibernate_sequence
Hibernate: create sequence hibernate_sequence start 1 increment 1
Hibernate:

    create table Tipo (
       id int4 not null,
        descricao varchar(100) not null,
        primary key (id)
    )
set 19, 2021 5:53:15 PM org.hibernate.engine.transaction.jta.platfo
```



Com nossa entidade Tipo mapeada, iremos criar novos tipos ao nosso banco de dados. Em nosso exemplo estamos criando três tipos de Livros (Romance, História e Biografia).

Então utilizamos o entityManager para iniciar uma nova transação com o banco de dados. Existindo uma transação basta utilizarmos o método persist.

O método persist vai receber um objeto de Tipo, como não temos o ID será realizado um insert no banco de dados, caso contrário seria realizado o update.

```
public class LivrosApp {
    public static void main(String[] args) {
        EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory( persistenceUnitName: "minhaConexao");
        EntityManager entityManager = entityManagerFactory.createEntityManager();

        Tipo tipo01 = new Tipo();
        tipo01.setDescricao("Romance");

        Tipo tipo02 = new Tipo();
        tipo02.setDescricao("História");

        Tipo tipo03 = new Tipo();
        tipo03.setDescricao("Biografia");

        entityManager.getTransaction().begin();
        entityManager.persist(tipo01);
        entityManager.persist(tipo02);
        entityManager.persist(tipo03);

        entityManager.getTransaction().commit();

        entityManager.close();
        entityManagerFactory.close();
    }
}
```

Atualmente nossa aplicação está configurada, para ao iniciar excluir/criar novamente nossa estrutura. Vamos alterar o persistence.xml para apenas atualizar a estrutura quando necessário. Para isso precisamos alterar a propriedade ddl.auto para update.

```
        <property name="javax.persistence.jdbc.password" value="postgres" />
        <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
        <property name="hibernate.hbm2ddl.auto" value="update" />
```

Para buscar um tipo cadastrado temos o método find de EntityManager. Neste método passa a Entidade que desejamos buscar a informação no caso Tipo.class e o código do Id da entidade.

```
public class LivrosApp {
    public static void main(String[] args) {
        EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory( persistenceUnitName: "minhaConexao");
        EntityManager entityManager = entityManagerFactory.createEntityManager();

        Tipo tipoCadastrado = entityManager.find(Tipo.class,  o: 1);
        System.out.println(tipoCadastrado);
    }
}
```

Teremos como retorno:

```
Tipo{id=1, descricao='Romance'}
```

Para realizarmos Update em nossos objetos, basta alterar a informação do tipo encontrado e realizar novamente o persiste. Como teremos a informação do Id o framework Hibernate vai executar o update.

```
public class LivrosApp {
    public static void main(String[] args) {
        EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory( persistenceUnitName: "minhaConexao");
        EntityManager entityManager = entityManagerFactory.createEntityManager();

        Tipo tipoCadastrado = entityManager.find(Tipo.class,  o: 1);
        tipoCadastrado.setDescricao("Romance Saltless");

        entityManager.getTransaction().begin();
        entityManager.persist(tipoCadastrado);
        entityManager.getTransaction().commit();

        entityManager.close();
        entityManagerFactory.close();
    }
}
```

Para realizar a exclusão de um registro, basta trocar o persiste pelo remove, quando temos a informação do id no objeto;

```java
public class LivrosApp {
    public static void main(String[] args) {
        EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory( persistenceUnitName: "minhaConexao");
        EntityManager entityManager = entityManagerFactory.createEntityManager();

        Tipo tipoCadastrado = entityManager.find(Tipo.class,  o: 1);

        entityManager.getTransaction().begin();
        entityManager.remove(tipoCadastrado);
        entityManager.getTransaction().commit();

        entityManager.close();
        entityManagerFactory.close();
    }
}
```

Para concluirmos nosso projetos, vamos criar a classe Livro e realizar o seu mapeamento. A única difrença entre os mapeamentos utilizados em Tipo, será a existência de um relacionamento, pois cada Livro será de um Tipo.

Desta forma teremos a propriedade tipo da classe Tipo. Adicionamos @ManyToOne, pois neste caso vários livros terão um Tipo. A informação de fetch é para busca de informações deste relacionamento, como Lazy os dados do Tipo do Livro serão selecionados no banco apenas quando precisarmos da propriedade. Enquanto o JoinColumn estamos indicando como deve se chamar no banco de dados a coluna que vai ser responsável por este relacionamento.

```java
@Entity
public class Livro {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    @Column(nullable = false, length = 100)
    private String titulo;

    @Column(nullable = false, length = 100)
    private String autor;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "id_tipo")
    private Tipo tipo;
```

Desta forma podemos criar um Livro, com um tipo previamente cadastrado.

```
public class LivrosApp {
    public static void main(String[] args) {
        EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory( persistenceUnitName: "minhaConexao");
        EntityManager entityManager = entityManagerFactory.createEntityManager();

        Tipo tipoCadastrado = entityManager.find(Tipo.class,  o: 2);

        Livro livro01 = new Livro();
        livro01.setTitulo("Senhor dos Anéis");
        livro01.setAutor("J. R. R. Tolkien");
        livro01.setTipo(tipoCadastrado);

        entityManager.getTransaction().begin();
        entityManager.persist(livro01);
        entityManager.getTransaction().commit();

        entityManager.close();
        entityManagerFactory.close();
    }
}
```

Assim, concluimos as operações de CRUD com JPA. Porém podemos selecionar os dados cadastrados apenas pelo Id. Em nosso próximo encontro utilizaremos o JPQL para buscar informações por outras colunas de nossas entidades.

5 – Material Complementar

https://www.youtube.com/watch?v=MGWJbaYdy-Y

https://www.youtube.com/watch?v=vtR3WAbC6IA

https://www.caelum.com.br/apostila-java-web/uma-introducao-pratica-ao-jpa-com-hibernate#configurando-o-jpa-com-as-propriedades-do-banco

6 – Exercício

Desenvolva o seu projeto JPA, de forma que possa ser realizado o Cadastro de Pessoas, informando a sua cidade, com a seguinte estrutura.

Tabela de Cidades

ID	Integer
nome	char(80)
uf	char(2)

Tabela de Pessoas

ID	Integer
nome	char(80)
idade	Integer
rua	char(60)
numero	char(10)
bairro	char(60)
id_cidade	integer