

INSTITUTO FEDERAL  
SÃO PAULO



# Persistência em Banco de dados com Spring Data JPA

Desenvolvimento Web Backend (BRADWBK)

Prof. Luiz Gustavo Diniz de Oliveira Vêras

E-mail: [gustavo\\_veras@ifsp.edu.br](mailto:gustavo_veras@ifsp.edu.br)



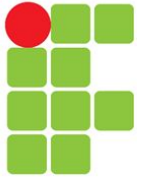
# Objetivos

- ✓ Object Relational Mapper (ORM)
- ✓ Java Persistence API (JPA)
  - ✓ POJO
  - ✓ Anotações JPA em um POJO
- ✓ Spring Data JPA
  - ✓ Criando um Repository
- ✓ Relacionamentos e cardinalidade em banco de dados
- ✓ *Annotations* de Relacionamentos do JPA



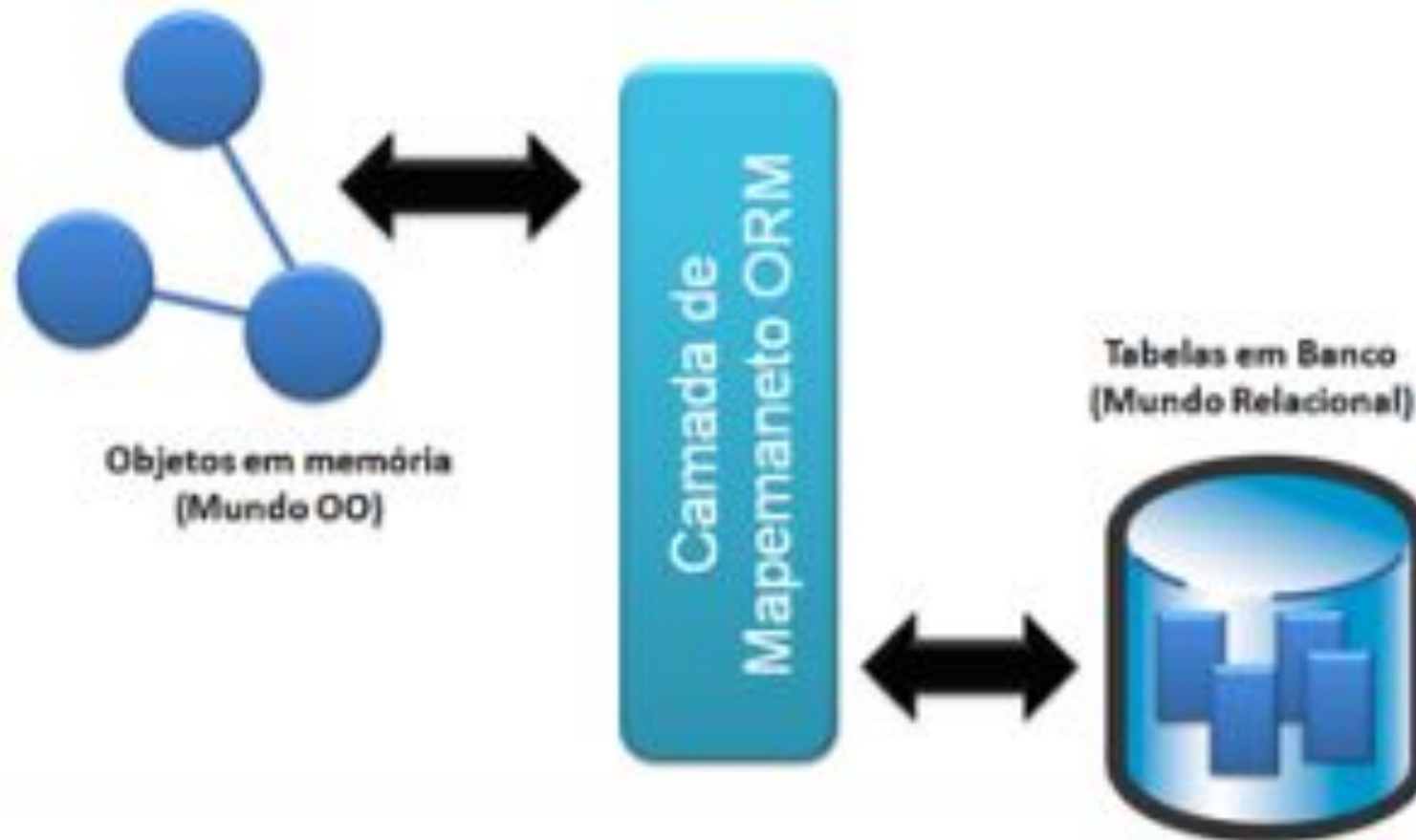
# Object Relational Mapping (ORM)

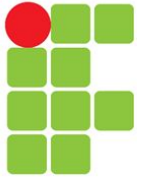
- ORM (Object Relational Mapping) é uma técnica de mapeamento objeto relacional que permite fazer uma relação dos objetos com os dados que os mesmos representam. Ultimamente tem sido muito utilizada e vem crescendo bastante nos últimos anos, devido:
  - Agilidade que traz na criação de aplicações com persistência.
  - Abstrai a necessidade de ter que aprender um código SQL específico de um SGDB, pois o ORM irá gerar o SQL para você, já no formato do banco usado.
  - Se for necessário trocar de SGDB (de MySQL para PostgreSQL), essa tarefa se torna mais ágil com o ORM.



# Object Relational Mapper (ORM)

- Diagrama de um ORM

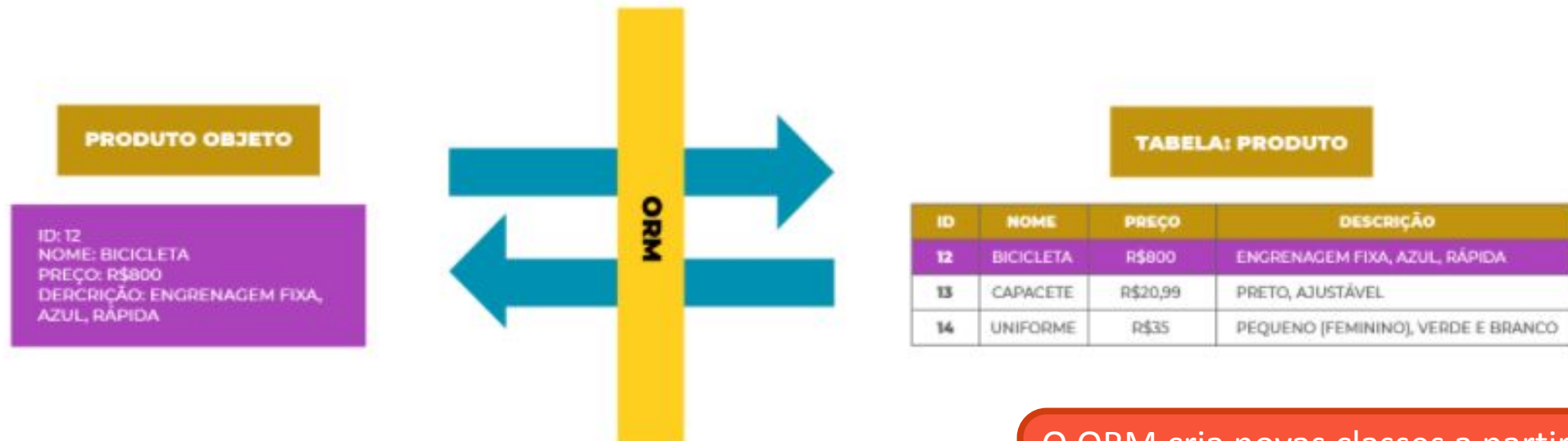




# Object Relational Mapper (ORM)

Como o ORM trabalha

O ORM pode criar, a partir de uma classe tabelas e seus relacionamentos no SGBD.



O ORM cria novas classes a partir das tabelas e de seus relacionamentos que estão no banco.



# ORM e JPA (Java Persistence API)

ORM são implementados em bibliotecas e para diversar linguagens.  
Alguns exemplos são:

## Java

- Hibernate
- EclipseLink
- ActiveJPA

## PHP

- Doctrine
- Eloquent

## Kotlin

- Exposed

## C#

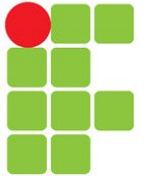
- Entity Framework
- Nhibernate
- Dapper

## Node

- Sequelize

## Python

- DjangoORM
- SQLAlchemy



# ORM e JPA (Java Persistence API)

ORM são implementados em bibliotecas e para diversar linguagens.  
Alguns exemplos são:

## Java

- Hibernate
- EclipseLink
- ActiveJPA

## PHP

- Doctrine
- Eloquent

## Kotlin

- Exposed

## C#

- Entity Framework
- Nhibernate
- Dapper

## Node

- Sequelize

## Python

- DjangoORM
- SQLAlchemy

Usaremos estes, mas nem vamos perceber pois o Spring irá gerenciar tudo para nós.



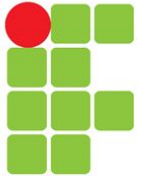
# HIBERNATE



# Java Persistence API (JPA)

- **JPA (Java Persistence API)** é uma especificação leve, baseado em *Plain Old Java Object POJO* (ou “*Os Singelos Clássicos Objetos Java*”) para persistir objetos Java, além de oferecer outras funcionalidades.
- É por meio da JPA que iremos nos comunicar com o ORM e indicar para ele como as classes devem ser mapeadas para uma tabela no banco de dados.
- Essencialmente, o JPA é um conjunto de *annotations* que nos permitem indicar nas nossas classes como elas devem ser mapeadas para tabelas seus atributos no banco de dados.
- As anotações da JPA estão no pacote **`javax.persistence`**.





# Java Persistence API (JPA)

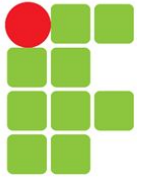
Primeiramente, devemos entender o que é um **POJO**.

- **Conceitualmente:** Classes simples que não implementam nenhuma interface nem herda de classes de bibliotecas externas (que não sejam padrão no Java). Assim ela pode ser utilizada por qualquer aplicativo Java.
- **Na prática:** É o formato de classe que já trabalhamos na programação das nossas classes Model, onde deve ser seguida a seguinte estrutura.
  - Sempre definir construtor *default* sem argumentos;
  - Pode ser Serializavel (implementar a interface *java.io.Serialization*)
  - Métodos que seguem o padrão de *getters* e *setters* para seus atributos.

Essa classe é um POJO pois:

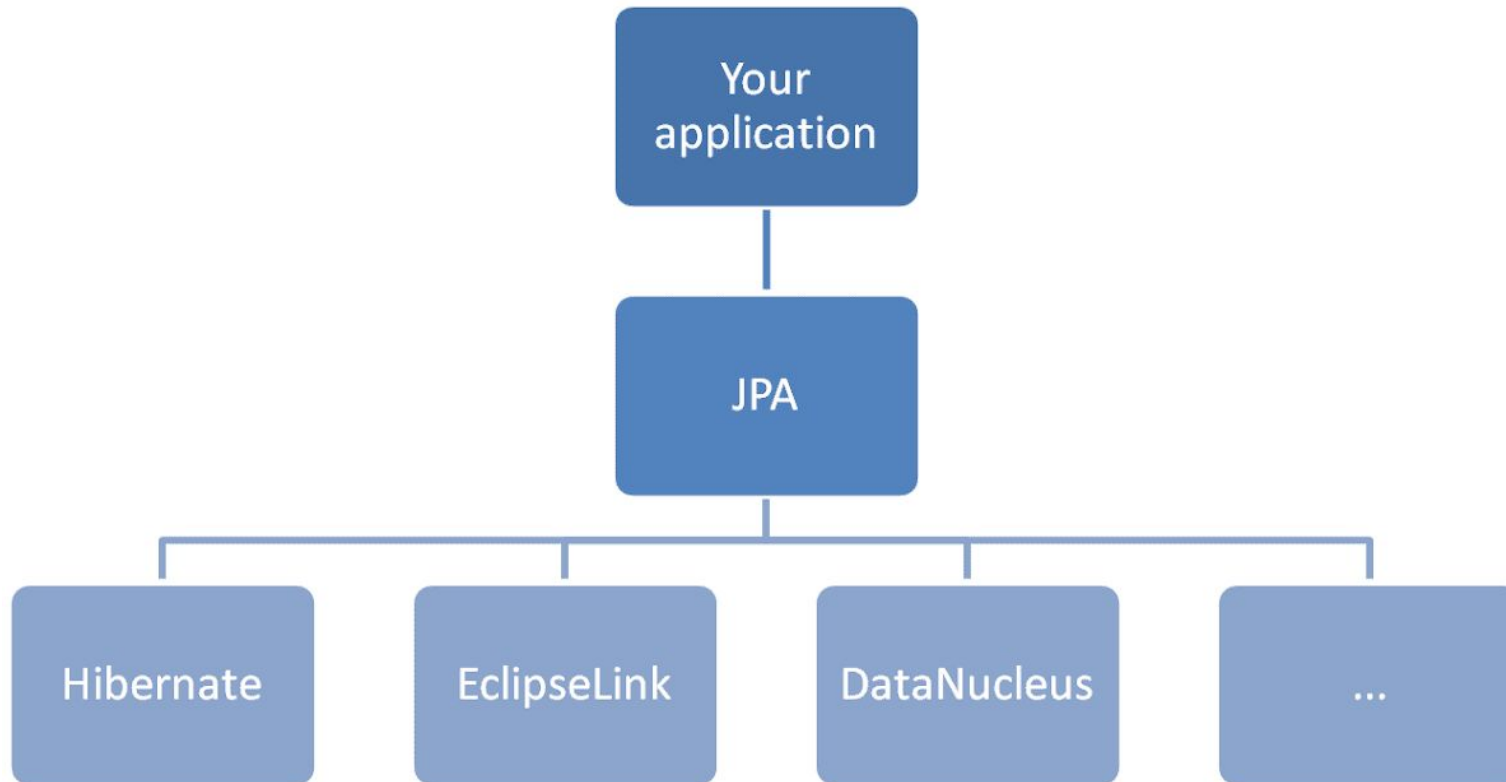
- construtor default sem argumentos;
- métodos que seguem o padrão de getters e setters para seus atributos.

```
public class Carro {  
    private String nome;  
    private String cor;  
    public Carro() {  
    }  
    public Carro(String nome, String cor) {  
        this.nome = nome;  
        this.cor = cor;  
    }  
    public String getCor() {  
        return cor;  
    }  
    public void setCor(String cor) {  
        this.cor = cor;  
    }  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```



# Java Persistence API (JPA)

## JPA x Hibernate



**JPA** é a interface, não possui implementação (algoritmos), apenas nomes de annotations.

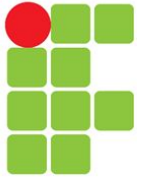
**Hibernate** é uma implementação do JPA.



# Java Persistence API (JPA)

## Annotations JPA que iremos utilizar

- **@Entity**: Define uma classe como sendo uma entidade que será transformada em tabela no banco. O ORM que “enxergar” uma classe marcada com a *annotation* **@Entity** também saberá criar suas SQLs automaticamente.
- **@Column**: Aplicado em campos ou métodos. Indica o campo ou método correspondente à uma coluna na tabela no banco de dados. Não é obrigatório, pois o ORM saberá inferi-las automaticamente pelo nome dos campos da classe que representa uma Entity.

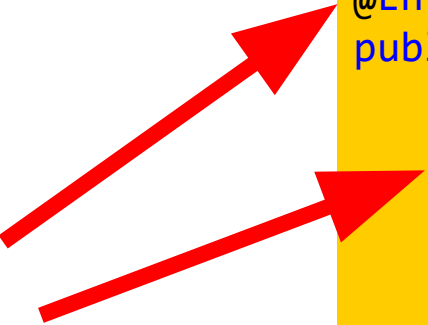


# Java Persistence API (JPA)

## Annotations JPA que iremos utilizar

- **@Id**: Indica o campo da classe como sendo a chave primária
- **@GeneratedValue**: Define que a chave primária será gerada automaticamente.
  - **Atributo “Strategy”**: Forma como o ID será gerado automaticamente pelo banco de dados.
    - **GenerationType.AUTO**: Irá definir uma estratégia de geração de ID a depender do banco de dados.
    - **GenerationType.IDENTITY**: Indica que o banco deve usar uma coluna de identidade (auto incremento do ID)
    - **GenerationType.SEQUENCE**: Id será gerado a partir de uma sequence (uma função no banco que gera a ID)
    - **GenerationType.TABLE**: Usa uma tabela para gerenciar as chaves primárias (menos recomendado)

## Anotações com JPA no POJO



```
@Entity
public class Carro {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nome;
    private String cor;
    public Carro() {
    }
    public Carro(String nome, String cor) {
        this.nome = nome;
        this.cor = cor;
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getCor() {
        return cor;
    }
    public void setCor(String cor) {
        this.cor = cor;
    }
    ... //código omitido
}
```

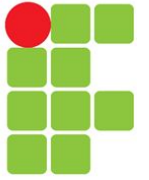


# Spring Data JPA

- Tudo o que precisamos para trabalhar com o JPA e ORM, o Spring nos fornece através de um módulo chamado **Spring Data JPA**.
- Para utilizar o **Hibernate**, geralmente precisamos fazer várias configurações de arquivos. O Spring Data JPA reduz, em muito, a necessidade de realizar tais configurações.
- Ele também nos fornece outras partes muito importantes:
  - Interfaces para a criação de repositórios JPA.
  - Suporte para paginação (busca gerenciada de partes da tabela do banco para evitar sobrecarga);
  - Entre outras facilidades.



**Spring Data JPA**



# Spring Data JPA

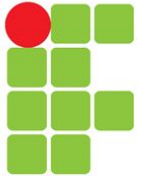
- Tudo o que precisamos para trabalhar com o JPA e ORM, o Spring nos fornece através de um módulo chamado **Spring Data JPA**.
- Para utilizar o **Hibernate**, geralmente precisamos fazer várias configurações de arquivos. O Spring Data JPA reduz, em muito, a necessidade de realizar tais configurações.
- Ele também nos fornece outras partes muito importantes:
  - **Interfaces para a criação de repositórios JPA.**
  - Suporte para paginação (busca gerenciado de partes da tabela do banco para evitar sobrecarga);
  - Entre outras facilidades.

**Essa é a parte que nos interessa!**



**Spring Data JPA**

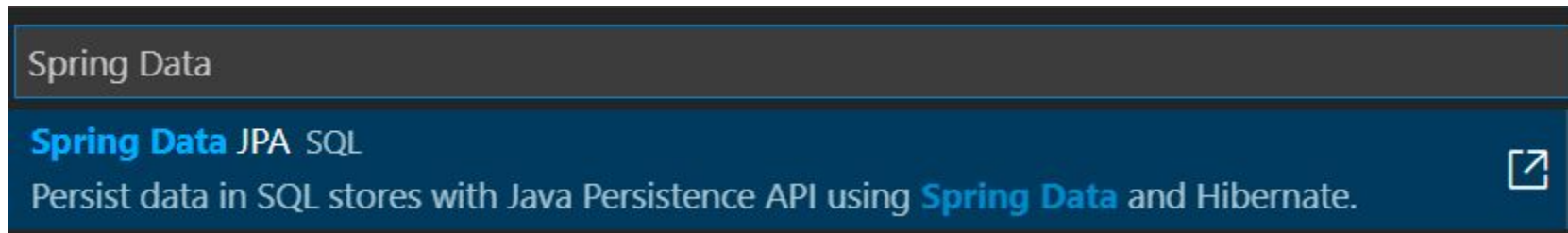




# Spring Data JPA

## Configurando o Spring Data JPA em seu projeto.

- Se estiver criando um projeto novo, poderá adicionar o Spring Data JPA pelo Inicializador de projeto do VSCode quando estiver adicionando dependências.





# Spring Data JPA

## Configurando o Spring Data JPA em seu projeto.

- Se já tiver um projeto pronto, abra o arquivo **pow.xml**, e dentro da tag **<dependencies>** adicione o spring data jpa

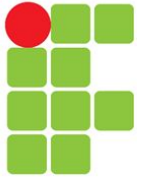
```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>
```



# Spring Data JPA

## O que é um repositório JPA?

- É a forma como um ORM que implementa o JPA gerencia o mapeamento entre uma classe e a sua tabela correspondente no banco de dados.
- Um repositório é criado para uma classe através de ***interfaces*** específicas. Através dessa interface, o ORM saber criar consultas de CRUD automaticamente para você e tratar todas as verificações necessárias para realizar operações bem sucedidas com o banco de dados.
- O **Spring Data JPA** nos fornece várias interfaces, para diferentes situações. Para nós a interface **CrudRepository<Entity, TypeId>** é suficiente.



# Spring Data JPA

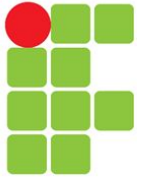
Só isso já é suficiente para criar um CRUD para a nossa classe POJO.

Assim como na herança entre classes, quando fazemos herança entre interface usamos extends.

```
public interface CarroRepository extends CrudRepository<Carro, Long>{  
}
```

O Repositório será de Carro, então a utilizamos aqui!

Tipo da chave primária (id) na classe mapeada.



# Spring Data JPA

Criar uma nova interface estendendo **CrudRepository<Entity, TypeId>** vai nos fornecer métodos de CRUD para interagir com o banco de dados:

**count()**

Retorna o número de entidades disponíveis.

**delete(T entity)**

Exclui uma determinada entidade.

**deleteAll()**

Exclui todas as entidades gerenciadas pelo repositório.

**deleteAll(Iterable<? extends T> entities)**

Exclui as entidades fornecidas.

**deleteAllById(Iterable<? extends ID> ids)**

Exclui todas as instâncias do tipo T com os IDs fornecidos.

**deleteById(ID id)**

Exclui a entidade com o ID fornecido.

**existsById(ID id)**

Retorna se existe uma entidade com o ID fornecido.

**findAll()**

Retorna todas as instâncias do tipo.

**findAllById(Iterable<ID> ids)**

Retorna todas as instâncias do tipo T com os IDs fornecidos.

**findById(ID id)**

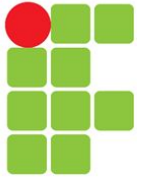
Recupera uma entidade por seu id.

**save(S entity)**

Salva uma determinada entidade.

**saveAll(Iterable<S> entities)**

Salva todas as entidades dadas.



# Spring Data JPA

Para termos acesso a uma instância do repositório, vamos utilizar a *annotation* **@Autowired**. (<https://www.baeldung.com/spring-autowire>)

Ela indica ao Spring que um componente/objeto gerenciado pelo Spring seja vinculado/conectado (daí vêm o *wired* do nome da anotação) para um campo automaticamente quando a aplicação for iniciada.

Repositórios são gerenciados pelo Spring. Para utilizá-lo faça como segue.

```
@Autowired  
MeuRepository meuRepository;
```



# Spring Data JPA

## Utilizando o repositório

```
/*Basta definir um campo do tipo do repositório com a
   annotation @Autowired que o Spring irá criar um objeto
   pra gente (isso se chama Injeção de Dependência).*/
@Autowired
CarroRepository carroRepository;

//Utilizando alguns métodos do repositório
Carro carro = new Carro();
carroRepository.findAll();
carroRepository.findById(10L);
carroRepository.save(carro);
carroRepository.delete(carro);
```



# Spring Data JPA

Para executar a aplicação, vamos precisar de um banco de dados. Para as aulas, vamos utilizar o MySQL.

**#1** - Abra o arquivo **pow.xml**, e dentro da tag **<dependencies>** o SQLite

```
<dependency>  
  <groupId>mysql</groupId>  
  <artifactId>mysql-connector-java</artifactId>  
  <version>8.0.30</version>  
</dependency>
```



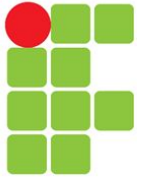


# Spring Data JPA

Para executar a aplicação, vamos precisar de um banco de dados. Para as aulas, vamos utilizar o MySQL.

**#2** - Abra o arquivo **application.properties**, e adicione os dados de conexão com o MySQL. Certifique-se de ter criado o banco **carros\_db** no MySQL.

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/carro_db
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#spring.jpa.show-sql: true
```



# Spring Data JPA

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/carro_db
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#spring.jpa.show-sql: true
```

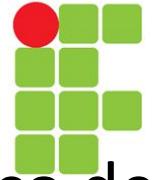
Aqui, **spring.jpa.hibernate.ddl-auto** pode ser **none**, **update**, **create** ou **create-drop**.

- **none**: O padrão para MySQL. Nenhuma alteração é feita na estrutura do banco de dados.
- **update**: O Hibernate altera o banco de dados de acordo com as estruturas de entidade fornecidas.
- **create**: Cria o banco de dados todas as vezes, mas não o descarta ao fechar.
- **create-drop**: Cria o banco de dados e o descarta quando SessionFactory fecha.

# Spring Data JPA

Política de criação do banco de dados.

URL de acesso ao banco de dados.



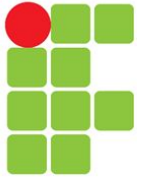
```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/carro_db
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#spring.jpa.show-sql: true
```

Usuário do banco de dados.

Se quiser ver as SQLs geradas basta remover # do início da linha.

Senha do banco de dados.

Classe de comunicação do Java com o MySQL. Fornecido pela dependência que adicionamos.

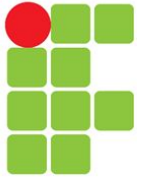


# Spring Data JPA

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/carro_db
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#spring.jpa.show-sql: true
```

Aqui, **spring.jpa.hibernate.ddl-auto** pode ser **none**, **update**, **create** ou **create-drop**.

- **none**: O padrão para MySQL. Nenhuma alteração é feita na estrutura do banco de dados.
- **update**: O Hibernate altera o banco de dados de acordo com as estruturas de entidade fornecidas.
- **create**: Cria o banco de dados todas as vezes, mas não o descarta ao fechar.
- **create-drop**: Cria o banco de dados e o descarta quando SessionFactory fecha.



# Spring Data JPA

```
spring.jpa.hibernate.ddl-auto=update
```

```
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/carro_db
```

```
spring.
```

```
spring.
```

```
spring.
```

```
spring.
```

```
#spring
```

Você deve começar com **create** ou **update**, porque você ainda não tem a estrutura do banco de dados.

Após a primeira execução, você pode alternar para **update** ou **none**, de acordo com os requisitos do programa

Aqui, **spring.jpa.hibernate.ddl-auto** pode ser **none**, **update**, **create** ou **create-drop**.

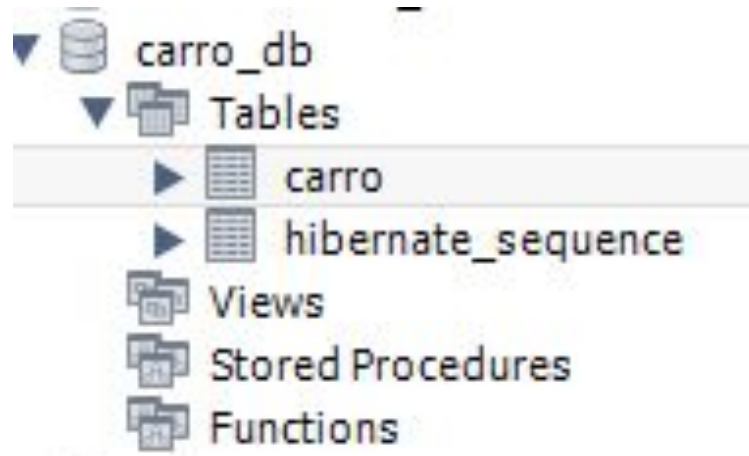
- **none**: O padrão para MySQL. Nenhuma alteração é feita na estrutura do banco de dados.
- **update**: O Hibernate altera o banco de dados de acordo com as estruturas de entidade fornecidas.
- **create**: Cria o banco de dados todas as vezes, mas não o descarta ao fechar.
- **create-drop**: Cria o banco de dados e o descarta quando SessionFactory fecha.



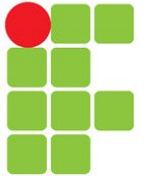
# Spring Data JPA

Para testar, crie um controller para o **Carro** e faça endpoints com Post e Get usando o repositório.

- Teste usando o Thunder Client;
- Veja os dados inseridos no MySQL Workbench.



	id	cor	nome
▶	1	Cinza	Monza
	2	Verde	Kwid
•	NULL	NULL	NULL

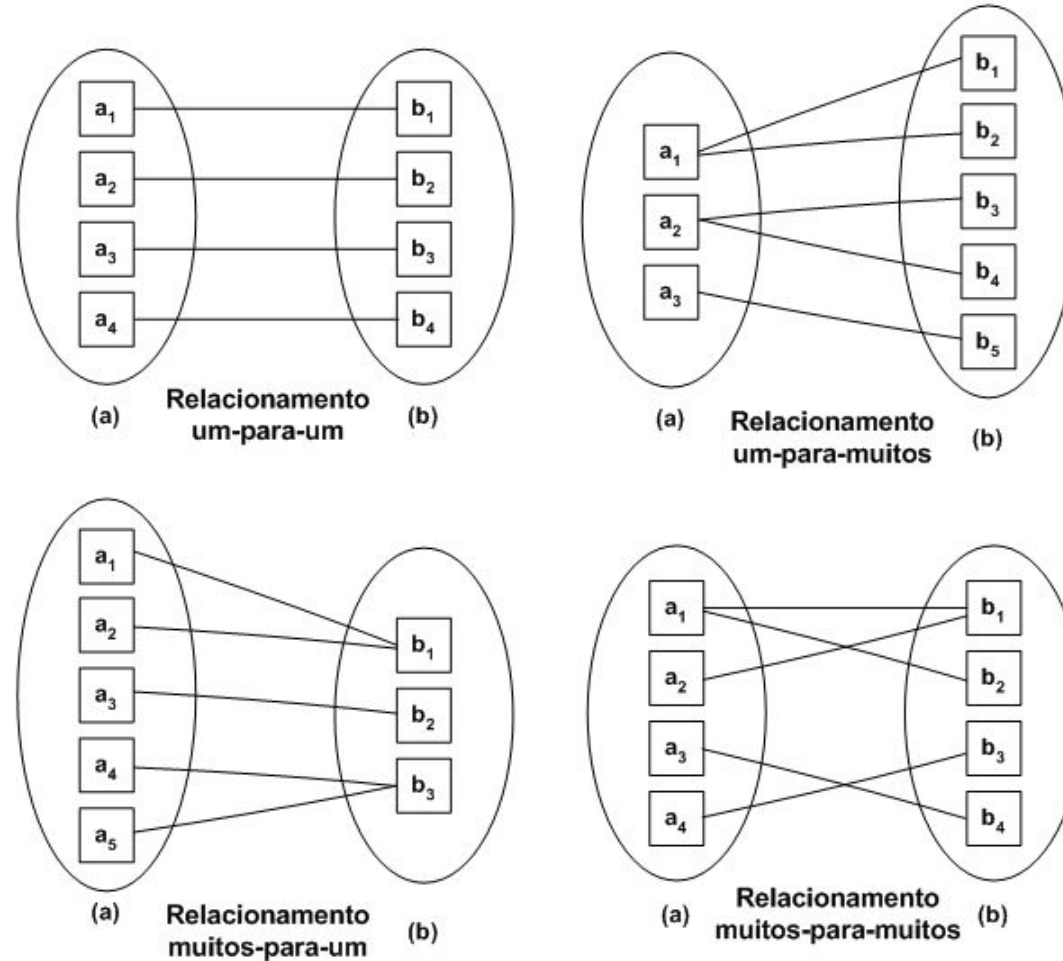
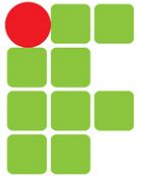


# Cardinalidade de relacionamento entre tabelas

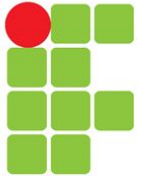
Em banco de dados, as tabelas podem apresentar diversos tipos de cardinalidade, que são as formas de relacionamento entre as tabelas do banco. Temos os seguintes tipos de cardinalidade:

- **Um-para-um (1:1):** uma linha da tabela A está associada no máximo a uma linha da tabela B e uma linha da tabela B está associada no máximo a uma linha da tabela A;
- **Um-para-muitos (1:M):** uma linha da tabela A está associada a qualquer número de linhas de B. Uma linha da tabela B, entretanto, pode estar associada no máximo a uma linha de A;
- **Muitos-para-um (N:1):** uma linha da tabela A está associada no máximo a uma linha de B. Uma linha da tabela de B, entretanto, pode estar associada a qualquer número de linhas de A;
- **Muitos-para-Muitos (N:M):** uma linha da tabela A está associada a qualquer número de linhas da tabela B e uma linha de B está associada a qualquer número de linhas de A.

# Cardinalidade de relacionamento entre tabelas







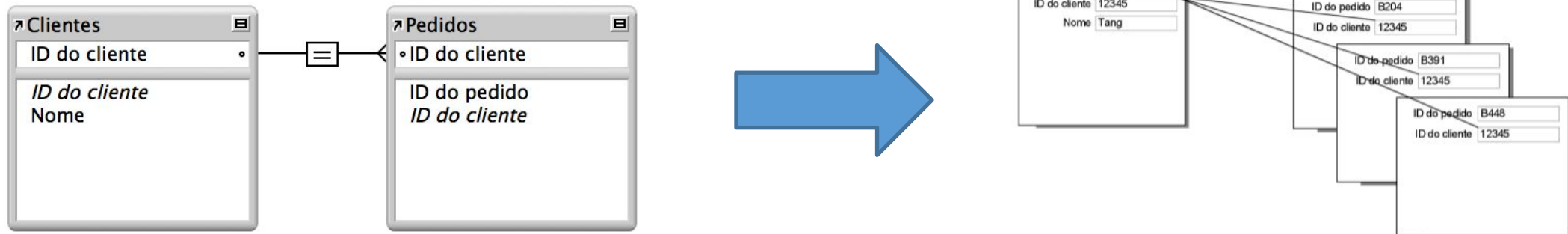
# Cardinalidade de relacionamento entre tabelas

Em banco de dados, geralmente essas associações acontecem através da **chave primária (primary key)**.

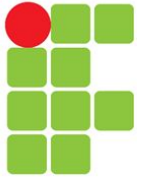
Abaixo temos um exemplo de um relacionamento **Um-para-Muitos**.

Perceba que o id do **Cliente** (**\*tabela dominante**) vai como um campo na tabela **Pedidos** (**\*tabela subordinada**).

A tabela subordinada é aquela na qual uma linha não pode existir, caso não haja uma linha correspondente na tabela dominante.



**\*Na teoria de banco de dados, seria um Conjunto de Entidades ao invés de Tabela, mas estamos chamando-a assim por simplicidade e clareza.**

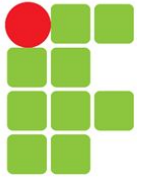


# Anotações de cardinalidade em JPA

Nosso interesse aqui é saber como representar esses relacionamentos em Java por meio do JPA. Usaremos anotações correspondentes aos relacionamentos.

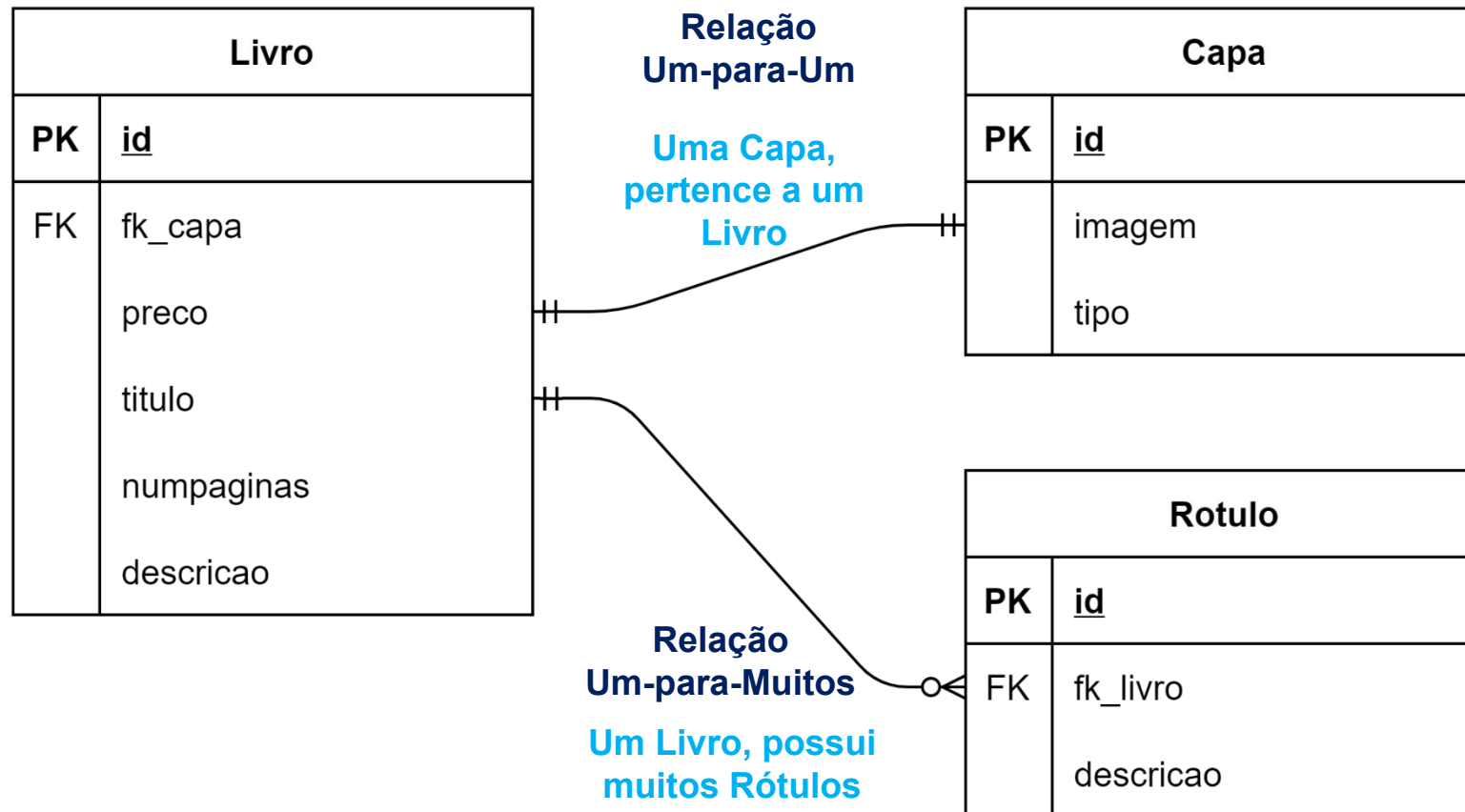
Faremos um paralelo entre **Tabela dominante/Classe dominante** e **Tabela Subordinada/Classe subordinada** para dizer em quais classes vão as anotações.

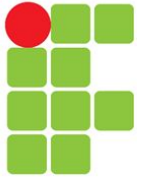
Relacionamento	Classe Dominante	Classe subordinada	Direcionamento
Um-para-um	@OneToOne	@OneToOne	Unidirecional/Bidirecional
Um-para-Muitos/Muitos-para-Um	@OneToMany	@ManyToOne	Bidirecional
Muitos-para-Um	@OneToMany	-	Unidirecional
Muitos-para-Muitos	@ManyToMany	@ManyToMany	Unidirecional/Bidirecional



# Anotações de cardinalidade em JPA

Vamos utilizar este modelo para demonstrar o uso das anotações!

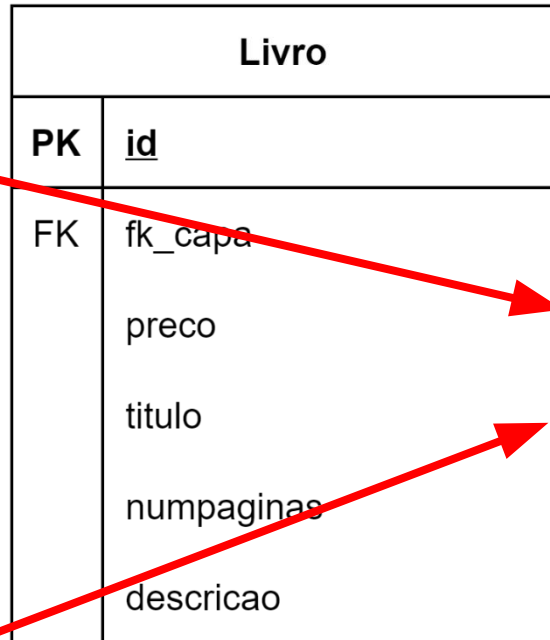




# Anotações de cardinalidade em JPA

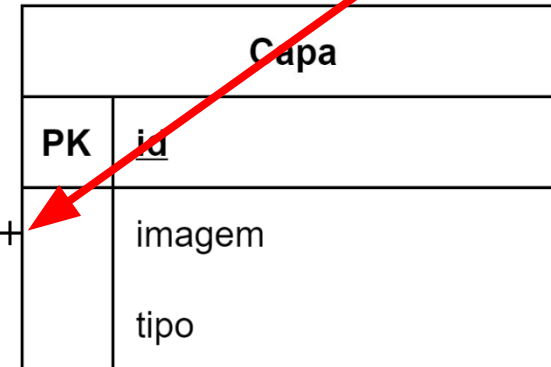
Vamos utilizar este modelo para demonstrar o uso das anotações!

Número **MIN** de Livros  
associados à Capas: 1  
Número **MAX** de Livros  
associados à Capas: 1



Relação  
Um-para-Um

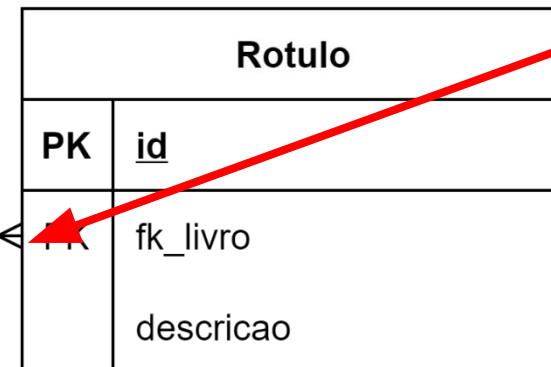
Uma Capa,  
pertence a um  
Livro



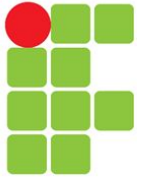
Número **MIN** de Capas  
associadas a Livros: 1  
Número **MAX** de Capas  
associadas a Livros : 1

Número **MIN** de Livros  
associados à Rótulos: 1  
Número **MAX** Livros  
associados à Rótulos: 1

Relação  
Um-para-Muitos  
Um Livro, possui  
muitos Rótulos



Número **MIN** de  
Rótulos associados à  
Livros: 0  
Número **MAX** de  
Rótulos associados à  
Livros: Muitos



# Anotações de cardinalidade em JPA

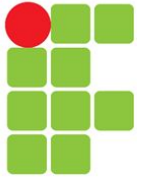
- Associações (Um-para-Um)

```
15 public class Livro {  
16  
17     @OneToOne  
18     @JoinColumn(name="fk_capa")  
19     private Capa capa;  
20  
21     //...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes

```
7 public class Capa {  
8  
9     @OneToOne(mappedBy="capa")  
10     private Livro livro;  
11  
12     //...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
IMAGEM	BLOB	Yes
TIPO	CHAR(1)	Yes



# Anotações de cardinalidade em JPA

- Associações (Um-para-Um)

## @JoinColumn

Elemento opcional que permite definir informações sobre a chave estrangeira (e.g., nome da coluna da chave estrangeira). É o que formará o vínculo entre as tabelas Livro e Capa.

```
15 public class Livro {
16
17     @OneToOne
18     @JoinColumn(name="fk_capa")
19     private Capa capa;
20
21     //...
```

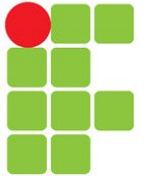
```
7 public class Capa {
8
9     @OneToOne(mappedBy="capa")
10     private Livro livro;
11
12     //...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
IMAGEM	BLOB	Yes
TIPO	CHAR(1)	Yes

## Atributo mappedBy

Recebe o nome do do campo na classe dominante que irá receber uma instância da classe subordinada (no caso é Capa) referente a uma linha na tabela subordinada.



# Anotações de cardinalidade em JPA

- Associações (Um-para-Muitos/Muitos-para-Um)

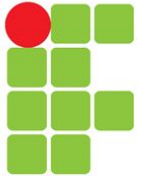
```
13 @Entity
14 public class Livro {
15     @OneToMany(mappedBy = "livro")
16     private List<Rotulo> rotulos;
17     //...
```

```
11 @Entity
12 public class Rotulo {
13     @ManyToOne
14     private Livro livro;
15     // ...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
DESCRICAO	VARCHAR2(255)	Yes
ID_LIVRO	NUMBER(19,0)	Yes





# Anotações de cardinalidade em JPA

- Associações (Um-para-Muitos/Muitos-para-Um)

Perceba que iremos receber uma lista **List** nesta relação. Assim poderemos receber várias linhas da tabela Rotulo.

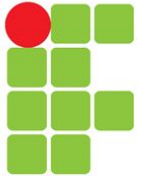
```
13 @Entity
14 public class Livro {
15     @OneToOne(mappedBy = "livro")
16     private List<Rotulo> rotulos;
17     //...
```

```
11 @Entity
12 public class Rotulo {
13     @ManyToOne
14     private Livro livro;
15     // ...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
DESCRICAO	VARCHAR2(255)	Yes
ID_LIVRO	NUMBER(19,0)	Yes

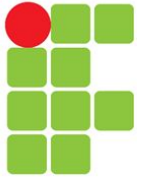




# Anotações de cardinalidade em JPA

## Mais detalhes sobre a relação Um-para-Muitos

- Carregamento Eager vs Lazy:
  - Se tivéssemos 1 milhão de Rótulos para serem retornados na List dentro da classe Livro, o sistema ficaria muito pesado e demorado para responder.
  - O JPA fornece duas formas de carregamento neste caso dentro de **@OneToMany**:
    - **Eager loading** (Carregamento guloso): Carrega todas as linhas dentro da List que permite a relação.
    - **Lazy loading** (Carregamento preguiçoso): Os elementos da relação não serão carregados imediatamente, apenas quando realmente forem necessários.
  - Configuramos isso no atributo **fetch = FetchType.EAGER | FetchType.LAZY**

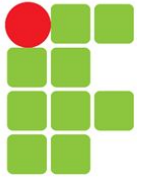


# Anotações de cardinalidade em JPA

- Associações (Um-para-Muitos/Muitos-para-Um)

```
14  @Entity
15  public class Livro {
16      @OneToMany(mappedBy = "livro", fetch = FetchType.LAZY)
17      private List<Rotulo> rotulos;
18      //...
```

Os Rótulos só serão carregados quando necessário, trazendo melhor performance para o seu programa.



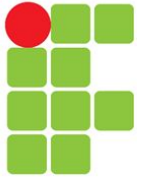
# Anotações de cardinalidade em JPA

## Mais detalhes sobre a relação Um-para-Muitos

- Operações com o banco em Cascata (Cascading)
  - Na relação Um-para-Muitos a existência de uma linha numa tabela subordinada as vezes pode depender da existência de uma linha na tabela dominante. Isso implica em situações que devemos lidar:
    - Sem um Livro, não faz sentido um Rótulo existir. Daí, ao criar um Livro, o JPA pode criar todos os seus Rótulos de sua **List** automaticamente para nós.
    - Quando deletarmos um Livro, os Rótulos associados podem ser removidos também automaticamente.
    - Se um Livro for atualizado, todos os Rótulos dele também podem ser atualizados automaticamente.
  - Automaticamente = Sem código Java escrito.
  - Configurando o atribute **cascade** da anotação **@OneToMany** na classe dominante, indicando ao JPA o que ele deve fazer em diferentes operações com o banco de dados. Configuramos utilizando **CascadeType**.

Estratégias para cada tipo de operação com o banco:

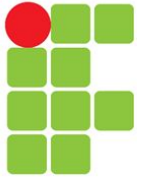
**PERSIST, MERGE, REMOVE, REFRESH, DETACH, and ALL** (Que combina todas as anteriores)



# Anotações de cardinalidade em JPA

## Mais detalhes sobre a relação Um-para-Muitos

Configuração do <i>Cascade</i>	Descrição
<b>CascadeType.PERSIST</b>	Quando a linha na tabela dominante <b>for salva</b> , as linhas na tabela subordinada também serão.
<b>CascadeType.MERGE</b>	Quando a linha na tabela dominante <b>for atualizada (elas já existem no banco e serão modificadas)</b> , as linhas na tabela subordinada também serão.
<b>CascadeType.REMOVE</b>	Se a linha na tabela dominante <b>for removida</b> , as linhas na tabela subordinada também serão
<b>CascadeType.REFRESH</b>	Se a linha na tabela dominante <b>for relida (atualiza o objeto Java caso haja alterações no banco)</b> , as linhas na tabela subordinada também serão.
<b>CascadeType.DETACH</b>	Se a linha na tabela dominante forem <b>desvinculadas do objeto Java que a representa</b> , as linhas na tabela subordinada também serão.
<b>CascadeType.ALL</b>	Que combina todas as anteriores



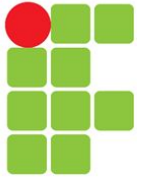
# Anotações de cardinalidade em JPA

- Associações (Um-para-Muitos/Muitos-para-Um)

```
15  @Entity
16  public class Livro {
17      @OneToMany(mappedBy = "livro", fetch = FetchType.LAZY, cascade = CascadeType.ALL)
18      private List<Rotulo> rotulos;
19      //...
```

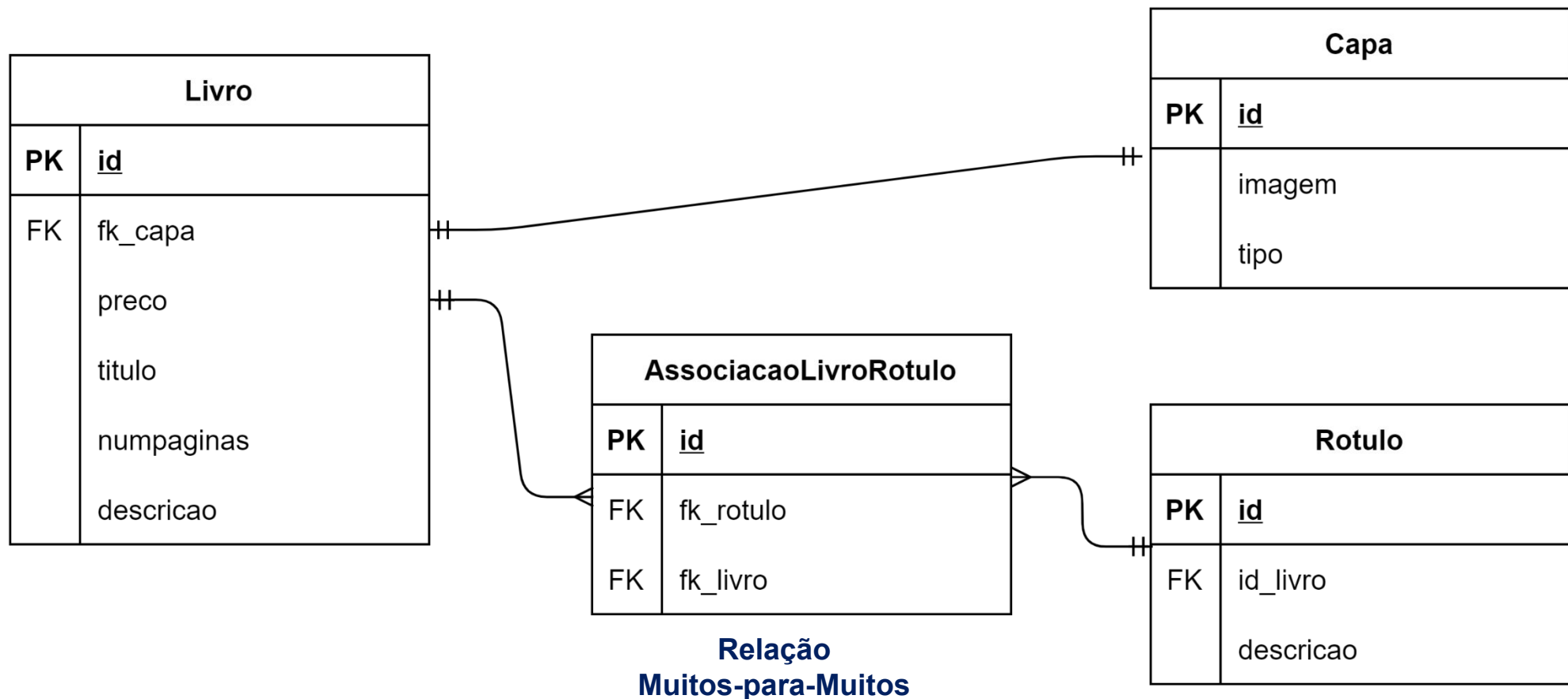
Se um objeto Livro tiver elementos na sua List de Rótulos, as operações de banco de dados executadas também serão refletidas na tabela Rotulo.

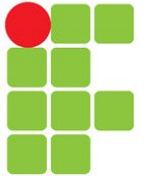
Se não fizermos isso, teríamos que realizar operações de banco primeiro com o objeto que representam a tabela subordinada (Rotulo) para depois executar operações com a tabela dominante (Livro)



# Anotações de cardinalidade em JPA

Alterando um pouco o modelo, podemos representar uma relação Muito-para-Muitos.





# Anotações de cardinalidade em JPA

- Associações (Muitos-para-Muitos)

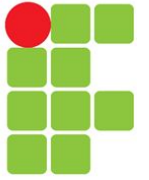
```
17 public class Livro {
18
19     @ManyToMany
20     @JoinTable(name = "associacao_livro_rotulo",
21               joinColumns = @JoinColumn(name = "fk_livro"),
22               inverseJoinColumns = @JoinColumn(name = "fk_rotulo"))
23     private List<Rotulo> rotulos;
24
25     //...
```

```
9 @Entity
10 public class Rotulo {
11
12     @ManyToMany(mappedBy = "rotulos")
13     private List<Livro> livrosEmQueAparece;
14
15     //...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes

Column Name	Data Type	Nullable
FK_LIVRO	NUMBER(19,0)	No
FK_ROTULO	NUMBER(19,0)	No

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
DESCRICAO	VARCHAR2(255)	Yes



# Anotações de cardinalidade em JPA

- Associações (Muitos-para-Muitos)

```
17 public class Livro {
18
19     @ManyToMany
20     @JoinTable(name = "associacao_livro_rotulo",
21               joinColumns = @JoinColumn(name = "fk_livro"),
22               inverseJoinColumns = @JoinColumn(name = "fk_rotulo"))
23     private List<Rotulo> rotulos;
24
25     //...
```

```
9 @Entity
10 public class Rotulo {
11
12     @ManyToMany(mappedBy = "rotulos")
13     private List<Livro> livrosEmQueAparece;
14
15     //...
```

**@JoinTable**  
Configura a tabela intermediária que permitirá o relacionamento **Muitos-para-Muitos**.

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes

Column Name	Data Type	Nullable
FK_LIVRO	NUMBER(19,0)	No
FK_ROTULO	NUMBER(19,0)	No

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
DESCRICAO	VARCHAR2(255)	Yes





# Fontes e Links

- <https://www.devmedia.com.br/orm-object-relational-mapper/19056>
- <https://medium.com/danielpadua/java-spring-boot-vscode-9ef9b8a263cd>
- <https://www.treinaweb.com.br/blog/o-que-e-orm>
- [Getting Started | Building a RESTful Web Service \(spring.io\)](https://spring.io/guides/gs/accessing-data-jpa/)
- <https://www.devmedia.com.br/jpa-como-usar-a-anotacao-generatedvalue/38592>
- <https://www.oreilly.com/library/view/spring-data/9781449331863/ch04.html>
- <https://www.baeldung.com/spring-annotation>
- [https://www.tutorialspoint.com/pg/jpa/jpa\\_entity\\_relationships.htm](https://www.tutorialspoint.com/pg/jpa/jpa_entity_relationships.htm)
- <https://spring.io/guides/gs/accessing-data-jpa/>
- <https://blog.algaworks.com/spring-data-jpa/>
- <https://medium.com/codestorm/spring-boot-jpa-entity-relationships-526d46ae228e>