SENAI

Serviço Nacional de Aprendizagem Industrial

PELO FUTURO DO TRABALHO

Relacionamentos (BD)

Desenvolvimento de Sistemas

Prof. Me. Reneilson Santos

Abril/2024

SENAI
PELO FUTURO DO TRABALHO

Agenda

- Relacionamentos
  - *One-to-One*
  - *One-to-Many e Many-to-One*
  - *Many-to-Many*
- Bibliografia

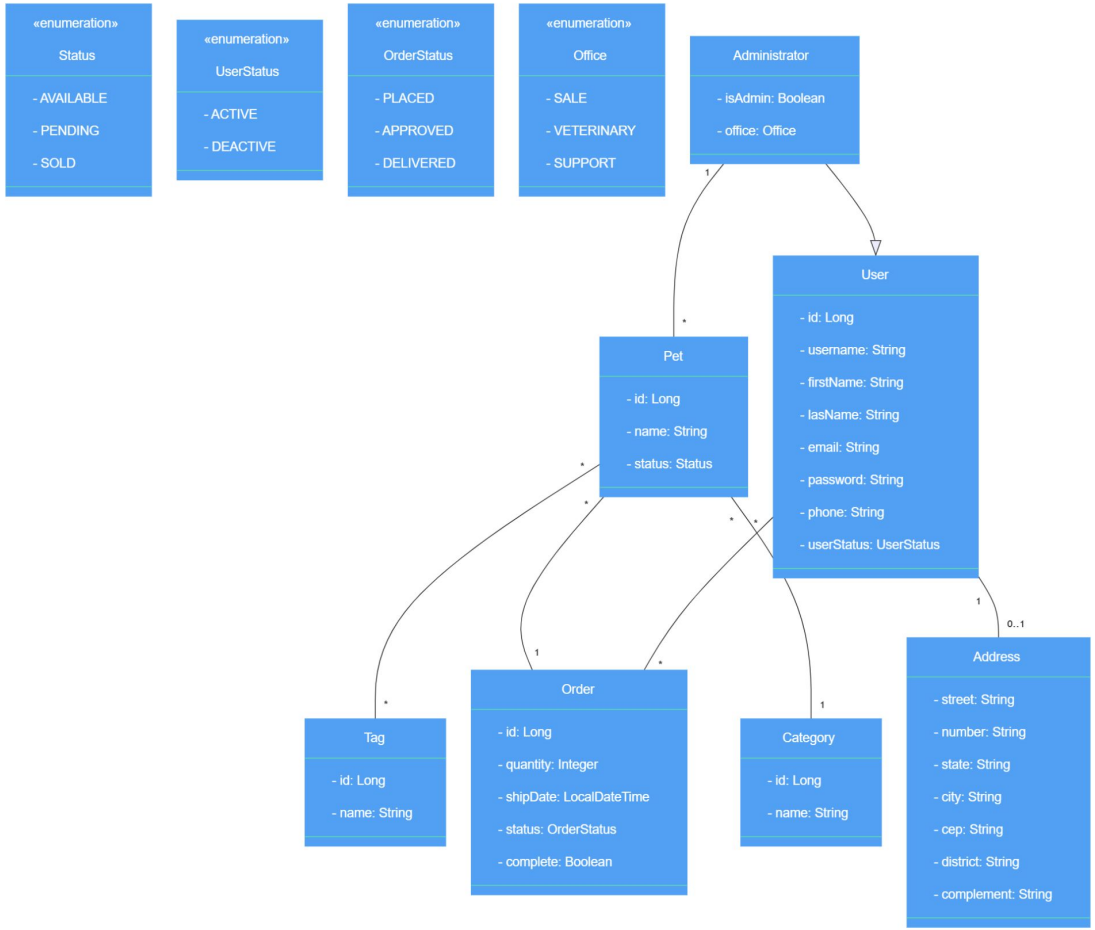
Relacionamentos

Relacionamentos

Os relacionamentos de banco de dados são associações entre diferentes tabelas.

De acordo com a cardinalidade existem 3 tipos básicos de relacionamentos entre as entidades:

- RELACIONAMENTOS **UM PARA UM (1:1)**
- RELACIONAMENTOS **UM PARA MUITOS (1:M)** *[ou muitos para um (M:1)]*
- RELACIONAMENTOS **MUITOS PARA MUITOS (M:N)**


«enumeration»
Status
- AVAILABLE
- PENDING
- SOLD
«enumeration»
UserStatus
- ACTIVE
- DEACTIVE
«enumeration»
OrderStatus
- PLACED
- APPROVED
- DELIVERED
«enumeration»
Office
- SALE
- VETERINARY
- SUPPORT
Administrator
- isAdmin: Boolean
- office: Office
User
- id: Long
- username: String
- firstName: String
- lasName: String
- email: String
- password: String
- phone: String
- userStatus: UserStatus
Pet
- id: Long
- name: String
- status: Status
Tag
- id: Long
- name: String
Order
- id: Long
- quantity: Integer
- shipDate: LocalDateTime
- status: OrderStatus
- complete: Boolean
Category
- id: Long
- name: String
Address
- street: String
- number: String
- state: String
- city: String
- cep: String
- district: String
- complement: String
1
*
1
0..1

Relacionamentos

- Pet pode ter várias tags;
- Pet tem uma categoria;
- Usuário pode fazer vários pedidos;
- Um pet pode estar em apenas um pedido;
- Uma tag pode estar em vários pets;
- Uma categoria pode estar em vários pets;

Relacionamentos

Antes de prosseguir com os relacionamentos e sua implementação, precisamos identificar as cardinalidades dos relacionamentos, ou seja, se são **OneToOne**, **OneToMany**, **ManyToOne** ou **ManyToMany.**

O próprio diagrama de classes representa essas cardinalidades do relacionamento.

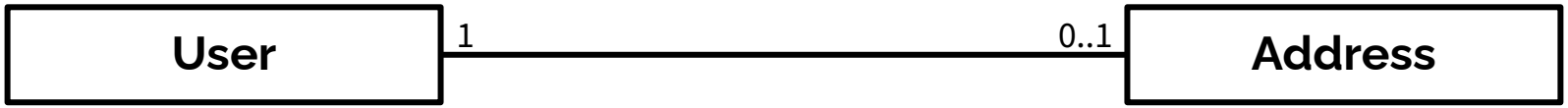
OneToOne

OneToOne

Define uma associação de valor único para outra entidade que possui multiplicidade um para um.

Normalmente **não é necessário especificar a entidade de destino associada** explicitamente, pois pode ser inferida a partir do tipo de objeto (classe) que está sendo referenciado.

Se o relacionamento for bidirecional, o lado não proprietário deve usar o elemento **mappedBy** da anotação @**OneToOne** para especificar o campo de relacionamento ou propriedade do lado proprietário.


User
1
0..1
Address

OneToOne (Forma Simples)

- Quando a associação é direta com o id das classes.

```
@Entity
public class User {
    ...
    @OneToOne(optional = true)
    private Address address;
}
```

OneToOne (Forma Simples)

- A classe mapeada usando a forma simples precisa ser uma entidade (que possui um id específico).
- Por padrão, é o id que será mapeado como a *foreign key*.

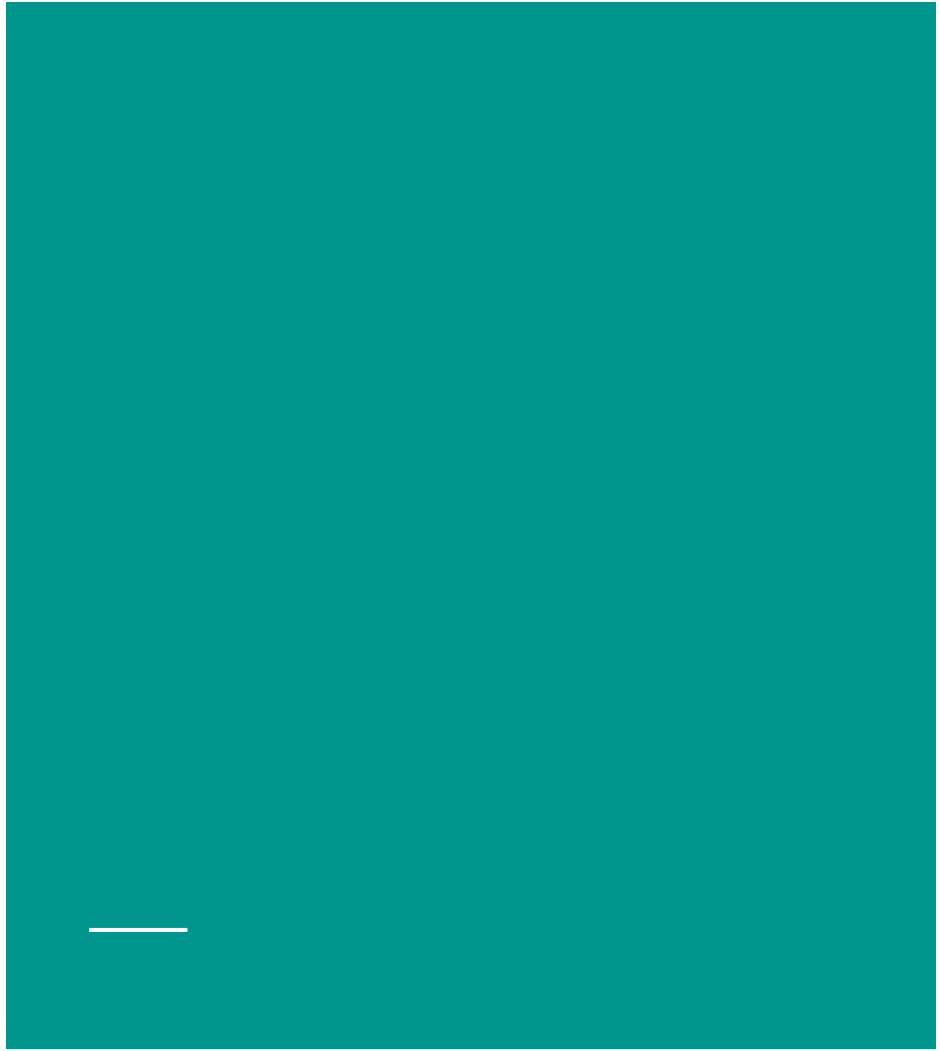
```
@Entity
public class Address {
   @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String street;
   private String number;
   private String district;
   private String city;
   private String state;
   private String cep;
}
```

OneToOne (Bidirecional)

- Quando queremos que seja possível encontrar os valores em ambas as classes, usamos um relacionamento bidirecional, como por exemplo, poderíamos ter um relacionamento bidirecional entre Endereço e Usuário, dessa forma, seria possível recuperar o usuário referente àquele endereço pelo java.
- Para isso, usamos o mappedBy na classe que não inserirá a foreign key na tabela, ou seja, o campo serve apenas para facilitar a implementação caso seja necessário recuperar essa informação dentro do sistema.

```java
@Entity
public class Endereco {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String street;
    private String number;
    private String district;
    private String city;
    private String state;
    private String cep;
    @OneToOne(mappedBy = "address", optional = true)
    private User user;
}
```

Embeddable



OneToOne (Embeddable)

- O embeddable é utilizado quando queremos inserir na mesma tabela todos os campos da tabela de relacionamento para evitar a necessidade de *joins* para recuperar essa informação.
- Para isso usamos na classe que será “embarcada” dentro da outra a anotação *Embeddable* ao invés de *Entity* e não precisaremos de um id dentro da classe.
- Na classe que tem o relacionamento OneToOne, usamos a anotação *Embedded.*

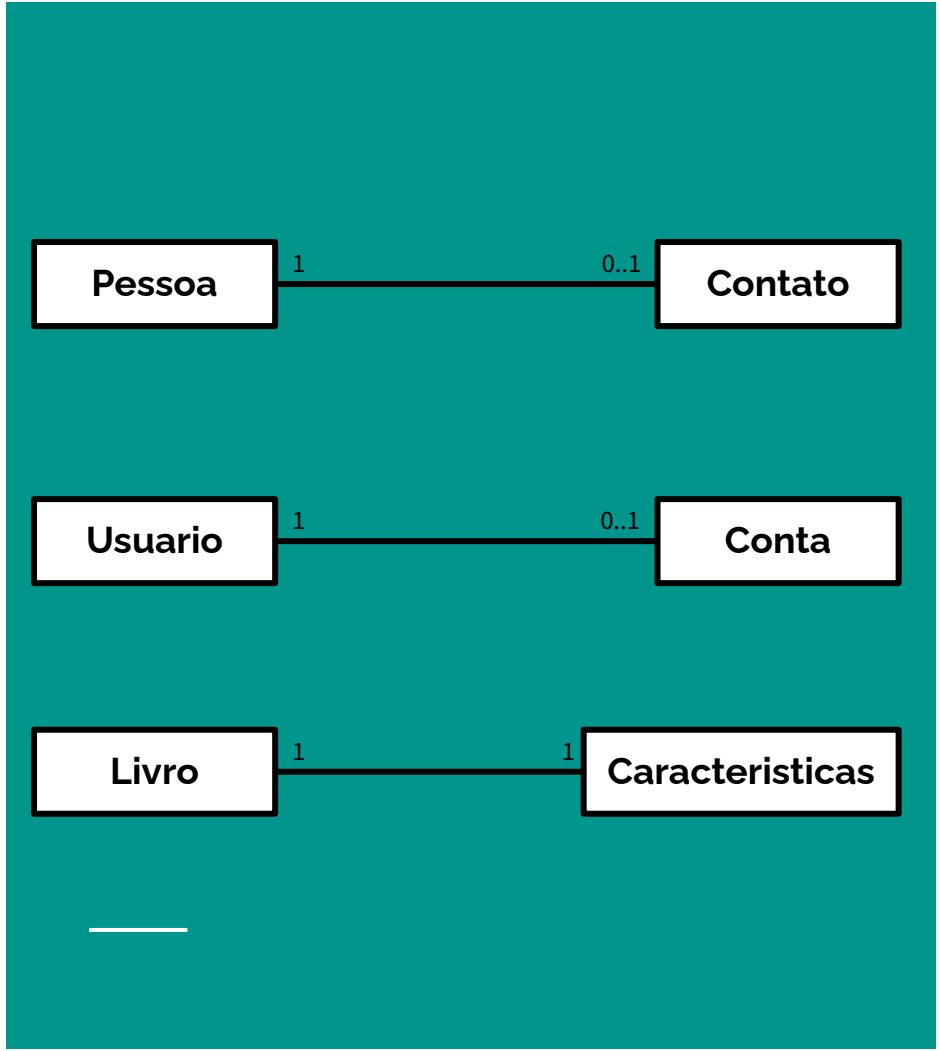
OneToOne (Embeddable)

```java
@Entity
public class User{
    ...
    @Embedded
    private Address address;
}
```

```java
@Embeddable
public class Address {
    private String street;
    private String number;
    private String district;
    private String city;
    private String state;
    private String cep;
}
```

Atividade

Especificar como ficariam as tabelas Pessoa, Contato, Usuario, Conta, Livro e Caracteristicas para cada um dos casos ao lado.



Pessoa
- nome - cpf

Contato
- celular - email

Usuario
- email - senha

Conta
- numero - saldo

Livro
- titulo

Caracteristicas
- editora - autor

ManyToOne ou OneToMany

Muitos para Um ou Um para Muitos

Neste caso, uma das entidades envolvidas **pode** referenciar **várias** unidades da outra, porém, do outro lado **cada uma das várias unidades referenciadas só pode estar ligada uma unidade** da outra entidade.

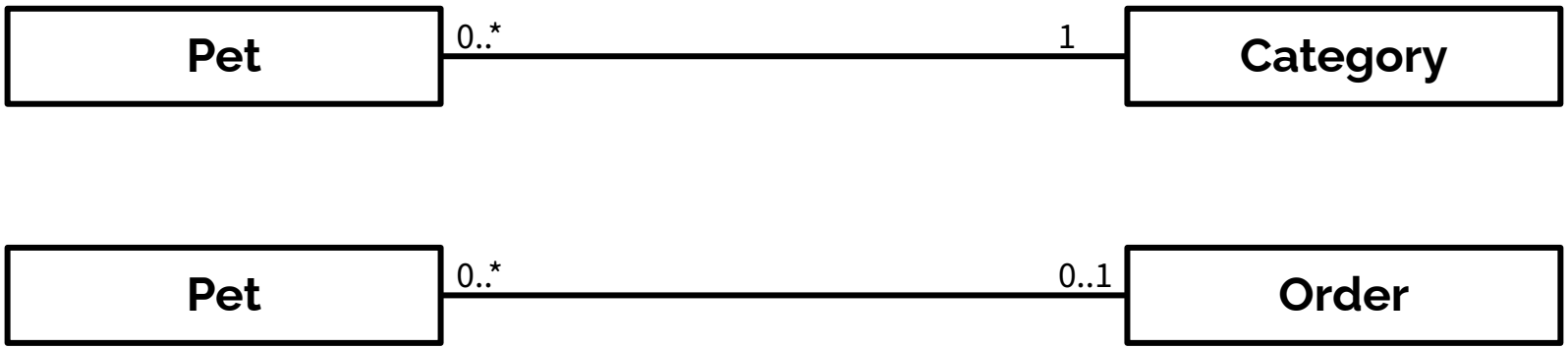
Ou seja, a tabela de chave primária contém somente um registro relacionado a nenhum, a um ou a muitos registros da tabela relacionada.

OneToMany vs ManyToOne

Ao usar a anotação OneToMany, precisamos criar uma lista para armazenar os dados, e isso será transformado em uma tabela de relacionamento dentro do banco de dados.

Ao usar a anotação ManyToOne é criado apenas uma ForeingKey na tabela onde está o ManyToOne.

Ao usar o mappedBy, seja no OneToMany ou no ManyToOne, a anotação não se transforma em tabela, apenas auxilia o código java no desenvolvimento.


Pet
0..*
1
Category
Pet
0..*
0..1
Order

```
public class Pet {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @Enumerated(EnumType.STRING)
    private Status status;
    @ManyToOne(optional = true)
    @JoinColumn(name="order_fk")
    private Order order;
    @ManyToOne
    @JoinColumn(name="category_fk")
    private Category category;
}
```

JoinColumn

A anotação @JoinColumn não é estritamente obrigatória, mas é altamente recomendada ao mapear relacionamentos @ManyToOne ou @OneToOne no Hibernate.

Quando você não especifica explicitamente um @JoinColumn, o Hibernate gera automaticamente um nome padrão para a coluna de chave estrangeira com base no nome do atributo da entidade e o nome da chave primária da entidade referenciada.

No entanto, é uma boa prática definir @JoinColumn para maior clareza e controle sobre o mapeamento.

OneToMany (Bidirecional)

No Spring Boot, se o relacionamento for bidirecional, o elemento **mappedBy** deve ser usado para especificar o campo de relacionamento ou propriedade da entidade que é o proprietário do relacionamento.

O mapeamento bidirecional permite que encontre mais facilmente uma listagem, por exemplo, se quisermos saber todos os eventos criados por um usuário, poderíamos ter esse relacionamento bidirecional dentro da classe usuário, apontando para uma lista de eventos, usando o OneToMany com o parâmetro mappedBy.

OneToMany (Bidirecional)

Todo elemento com o mappedBy não cria nada de novo no banco de dados, nem uma coluna nem uma nova tabela, ele será utilizado por meio de um join no momento que for solicitado.

Sem o mappedBy, uma nova tabela de relacionamento entre usuário e evento seria criada.

```java
@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class Category {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToMany(mappedBy = "category")
    private List<Pet> pets;
}

@Entity
@Table(name="_order")
@Data @NoArgsConstructor @AllArgsConstructor
public class Order {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Integer quantity;
    private LocalDateTime shipDate;
    private OrderStatus status;
    private Boolean complete;
    @OneToMany(mappedBy = "order")
    private List<Pet> pets;
}
```

Cascade

Ao trabalhar com relacionamentos, é interessante usar o Cascade da forma correta para evitar erros.

Temos os seguintes possíveis valores para o Cascade no Hibernate:

- ***ALL***
- ***PERSIST***
- *MERGE*
- *REMOVE*
- *REFRESH*
- *DETACH*

Detalhes OneToMany

Em relacionamentos OneToMany é importante usar o CASCADE para evitar possíveis erros ao tentar salvar as entidades. Nesse caso, podemos usar tanto o ALL (para todas as operações) ou o PERSIST (apenas para operações de persistência).

É interessante também inicializar as listas para evitar verificações na adição de novos dados.

Além disso, é importante salientar a necessidade de métodos para adicionar elementos na lista OneToMany quando este estiver com mappedBy, pois faz-se necessário realizar a operação dupla de adicionar nas duas classes:

```java
public class Order {
    …
    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
    private List<Pet> pets = new ArrayList<Pet>();

    public void addPet(Pet pet){
        this.pets.add(pet);
    }
}
```

```java
public class Category {
    …
    @OneToMany(mappedBy = "category", cascade = CascadeType.ALL)
    private List<Pet> pets = new ArrayList<Pet>();

    public void addPet(Pet pet){
        this.pets.add(pet);
    }
}
```

ManyToMany

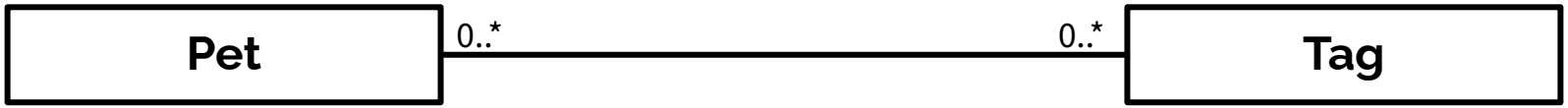
ManyToMany

Cada associação muitos-para-muitos tem dois lados, o lado do *proprietário* e o lado do *não proprietário.*

A tabela de junção (*join table*) é especificada no lado do proprietário.

Se a associação for bidirecional, qualquer lado pode ser designado como o lado proprietário.

Se o relacionamento for bidirecional, o lado não proprietário deve usar o elemento mappedBy da anotação ManyToMany para especificar o campo de relacionamento ou propriedade do lado proprietário.


Pet
0..*
0..*
Tag

```java
public class Pet {
    …
    @ManyToMany(mappedBy="pets")
    private List<Tag> tags = new ArrayList<Tag>();
}
```

```java
public class Tag {
    …
    @ManyToMany(cascade = CascadeType.ALL)
    private List<Pet> pets = new ArrayList<Pet>();
}
```

JoinTable

A anotação **@JoinTable** não é estritamente obrigatória ao configurar um relacionamento many-to-many (muitos-para-muitos) em Hibernate. No entanto, ela é frequentemente usada para personalizar a tabela intermediária (tabela de junção) que representa a associação entre as entidades.

Quando você não usa a anotação @JoinTable, como no caso do nosso exemplo, o Hibernate cria automaticamente uma tabela intermediária com um nome gerado automaticamente e colunas padrão para as chaves estrangeiras. Essa tabela intermediária é criada de acordo com as convenções de nomenclatura padrão.

Usando JoinTable

```
public class Tag {
    ...
    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(
        name = "tag_pet",
        joinColumns = @JoinColumn(name = "tag_id"),
        inverseJoinColumns = @JoinColumn(name = "pet_id")
    )
    private Set<Pet> pets = new HashSet<Pet>();
}
```

```
public class Pet {
    ...
    @ManyToMany(mappedBy="pets")
    private Set<Tag> tags = new HashSet<Tag>();
}
```

Exemplo

Finalizar todos os relacionamentos do projeto de Pets.

Atividade

Criar relacionamentos no projeto da SA.

Bibliografia

Bibliografia

- Annotations Hibernate (https://www.objectdb.com/api/java/jpa/annotations)
- Implementing Hibernate with Spring Boot and PostgreSQL (https://stackabuse.com/implementing-hibernate-with-spring-boot-and-postgresql/)
- Swagger PetStore (https://petstore.swagger.io/)