CADERNO ESPECIALISTA JPA

**SUMÁRIO**

[1 INTRODUÇÃO 3](#_Toc127106069)

[1.1 O QUE É MAPEAMENTO OBJETO-RELACIONAL (ORM) 3](#_Toc127106070)

[1.2 Entendendo a diferença entre os métodos persist e merge 3](#_Toc127106071)

[2 MAPEAMENTO 4](#_Toc127106072)

[2.1 Mapeamento de Relacionamentos 4](#_Toc127106073)

[2.1.1 Necessidade de mapear a volta em relacionamentos @ManyToOne, @OneToMany 5](#_Toc127106074)

[2.1.2 Remoção e entidade já referenciadas 5](#_Toc127106075)

[2.1.3 Entendendo o funcionamento de Eager e Lazy Loading 5](#_Toc127106076)

[2.1.4 Atributo Optional 6](#_Toc127106077)

[2.2 Conhecendo o EntityManager 6](#_Toc127106078)

[2.2.1 Estados e ciclo de vida dos objetos 6](#_Toc127106079)

[2.2.2 Funcionamento do método flush 6](#_Toc127106080)

[2.2.3 Callbacks para eventos do ciclo de vida 7](#_Toc127106081)

[2.3 Mapeamento Avançado 8](#_Toc127106082)

[2.4 Mapeando entidades para geração do DDL 8](#_Toc127106083)

# INTRODUÇÃO

## O QUE É MAPEAMENTO OBJETO-RELACIONAL (ORM)

Mapeamento objeto-relacional é você representar uma tabela do banco de dados relacional usando uma classe de forma que possa automatizar a geração do SQL.

Exemplificando a comparação de modelagem de dados em um banco de dados de uma linguagem OO:

|  |  |
| --- | --- |
| Banco de Dados | Linguagem OO |
| Tabela | Classe |
| Coluna | Atributo |
| Linha | Objeto |

## Entendendo a diferença entre os métodos persist e merge

Uma das diferenças é que o persist serve apenas para inserir dados e o merge serve para inserir e fazer updates dos dados. O persist guarda a instância do objeto persistido o que pode ser útil para realizar alterações na instância do objeto. Já com o merge ele faz uma cópia do objeto e guarda a antiga na memória do Entity Manager, não guarda a atual na memória desse modo se você precisar realizar alterações no objeto que acabou de ser inserido ou atualizado você precisa capturar a instância do objeto no tempo do merge.

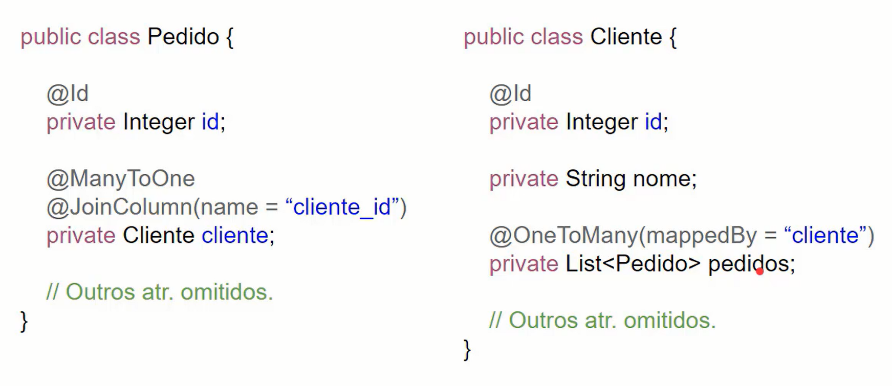
# MAPEAMENTO

## Mapeamento de Relacionamentos

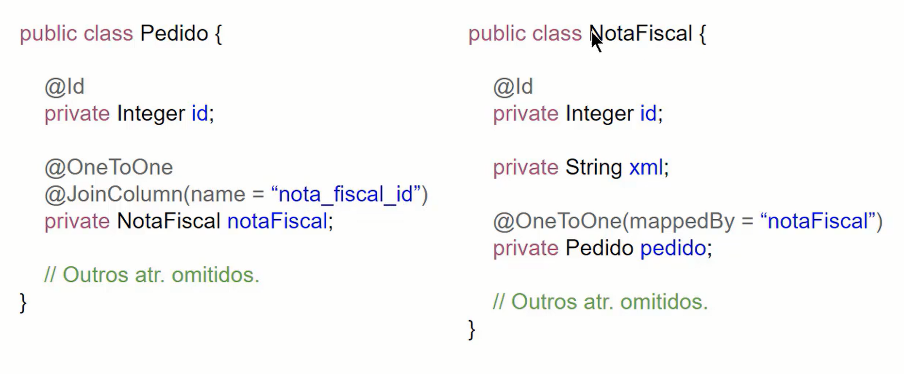
**@ManyToOne:** É quando temos a possibilidade de ter vários registros, de uma tabela qualquer, fazendo referência para um registro especifico que pode ser da mesma tabela ou, como na maioria das vezes, uma tabela diferente.



**@OneToMany:** É quando temos a possibilidade de um registro especifico ser referenciado por vários outros. Basicamente, o contrário do que foi descrito para @ManyToOne. Só que agora estamos falando de coleções.



**@OneToOne:** É quando uma tabela restringe seus registros a não fazerem referências repetidas. Uma mesma referência não pode ser feita duas vezes.



**@ManyToMany:** É quando temos a possibilidade de ter um registo referenciando vários outros, ao mesmo tempo que pode ser referenciado várias vezes. Como estamos falando de múltiplas referências de e para um mesmo registro, lidamos também com coleções.



### Necessidade de mapear a volta em relacionamentos @ManyToOne, @OneToMany

Você não precisa mapear a volta quando criar relacionamentos @ManyToOne, apenas analise se você precisa da volta, e caso você queira utilizar a volta você vai na outra classe e mapeia o atributo com o @OneToMany(mappedBy = “own\_class”) e o atributo do mapeamento sempre vai ter que ser uma collection, porque é ”UM” para “MUITOS”.

### Remoção e entidade já referenciadas

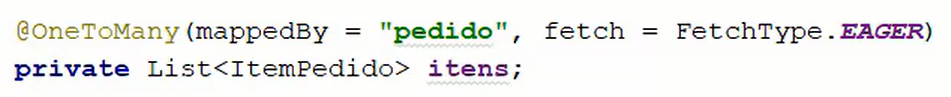
Quando uma entidade possui relacionamentos com outras entidades, mapeamentos, e você vai tentar remover essa entidade vai ocorrer exceções de foreign key, para resolver esse problema é necessário utilizar operações em cascata. Ou outra forma é remover cada dependência do relacionamento antes de remover a entidade.

### Entendendo o funcionamento de Eager e Lazy Loading

Eager (Ansioso) e Lazy (Lerdo) são utilizados pelo Hibernate para manipular a forma com que os atributos são retornados pela consulta no banco de dados. Por padrão atributos únicos são Eager, ou seja, eles são retornados assim que a classe é instanciada, de forma automática. Pelo contrário atributos de collections, como listas tem como padrão Lazy que significa que o hibernate não retorna eles automaticamente na instância da classe, eles só serão carregados caso você utilize os objetos da collection, mas não basta só chamar o atributo, você tem que utilizar um método para o JPA incluir ele na consulta. O JPA não retorna os dados de consulta de atributos Lazy.

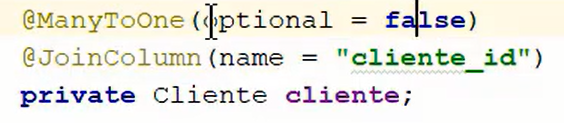
Você pode modificar esse comportamento caso necessite que um atributo de lista seja incluído na consulta imediatamente assim que a classe é chamada, e também pode omitir valores únicos de serem carregados imediatamente com Lazy.

Para mudar você utiliza a parâmetro fetch na annotation de relacionamento:



### Atributo Optional

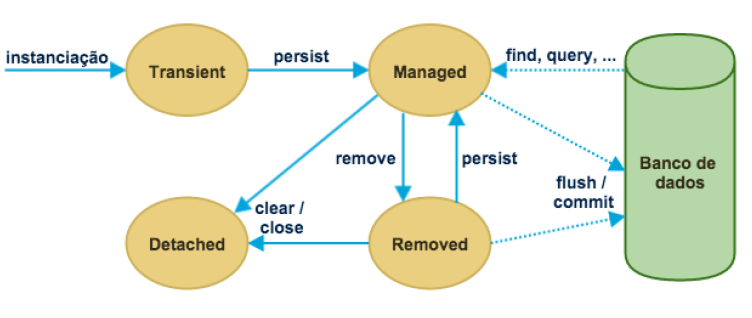
Sempre que precisarmos definir que um atributo seja obrigatório nós utilizamos o atributo optional na @annotation de relacionamento. Caso necessitemos que seja obrigatório utilizamos:



Por padrão todos os atributos são opcionais, para mudar precisa setar manualmente para false.

## Conhecendo o EntityManager

### Estados e ciclo de vida dos objetos



Os objetos possuem ciclo de vida no JPA, a cada novo objeto instanciado temos o estado transient, e quando este objeto inicia uma ação de persistência com o JPA, como por exemplo o persist, ele entra no estado de Managed, ação de busca com find também é o estado de managed, onde o JPA gerencia o objeto.

Quando removemos o objeto ele passa ao estado de removed, e quando usamos o clear / close ele muda para o estado de detached, o JPA gerencia todos os estados de cada objeto a partir das transações.

### Funcionamento do método flush

O método flush() serve para forçar o JPA enviar updates para o banco de dados, ele obriga o JPA a pegar tudo o que está na memoria e ainda não está sincronizado com o banco e sincronizar com o banco. Pode ser utilizado para forçar um update mesmo que abaixo das execuções haja um rollback() e a transação seja cancelada, mesmo assim ao utilizar o flush() irá ocorrer um update.

Dificilmente o flush() será necessário, porque isso você pode tratar com regras de negócios em outras classes, e validar os dados antes de enviar.

### Callbacks para eventos do ciclo de vida

Eventos de call-back servem para executar métodos antes ou depois de determinados eventos, no caso do JPA nos eventos de Persist, Update, Remove e Load. Veja o código de exemplo abaixo:

// @PrePersist  
// @PreUpdate  
 public void calculateTotal() {  
 if (orderedItems != null) {  
 total = orderedItems.stream()  
 .map(OrderedItem::getProductPrice)  
 .reduce(BigDecimal.**ZERO**, BigDecimal::add);  
 }  
 }  
  
 /\* \\\\\\ Other pre-processors \\\\\\  
 Each class can only have one callback annotation per class, if a class have  
 the same callback annotation more than once, will throw an exception. Because of that  
 use only one per class, this is explicit in this class for example, in calculateTotal()  
 method I would use the same callback annotations again, I commented both, because they  
 were causing errors. Use only one per class.  
 \*/  
  
 @PrePersist  
 public void whenPersist() {  
 creationDate = LocalDateTime.now();  
 calculateTotal();   
 }  
  
 @PreUpdate  
 public void whenUpdate() {  
 lastUpdateDate = LocalDateTime.now();  
 calculateTotal();   
 }  
  
 @PostPersist  
 public void afterPersist() {  
 System.**out**.println("After persist Order.");  
 }  
  
 @PostUpdate  
 public void afterUpdate() {  
 System.**out**.println("After update Order");  
 }  
  
 @PreRemove  
 public void beforeRemove() {  
 System.**out**.println("Before remove Order");  
 }  
  
 @PostRemove  
 public void afterRemove() {  
 System.**out**.println("After remove Order");  
 }  
  
 @PostLoad  
 public void afterLoadEntity() {  
 System.**out**.println("After load entity Order");  
 }

Com os callbacks você pode executar determinadas tarefas em determinado tempo da sua entidade, dependendo de quando ela for instanciada. Só pode haver uma determinada annotation de call-back por classe, não pode haver repetição, caso tenha duas ou mais irá causar erros.

## Mapeamento Avançado

### Conhecendo detalhes da anotação @Column

Podemos limitar o update de um atributo adicionando um parâmetro na annotation @Column, veja o código abaixo:

@Column(name = "creation\_date", updatable = false)  
private LocalDateTime creationDate;  
  
@Column(name = "last\_update\_date", insertable = false)  
private LocalDateTime lastUpdateDate;

O paramentro updatable serve para impedir que um atributo seja alterado após ter sido inserido uma vez, ou seja o valor nunca será atualizado se manterá como foi inserido a primeira vez. Já o parâmetro insertable serve para impedir que o valor seja inserido, o campo tem que ser nulo, não pode haver inserção.

### Mapeando chave composta com @IdClass

Podemos utilizar chaves compostas em entidades, chaves compostas servem para quando você necessita mapear mais de um ID em uma classe, essas situações ocorrem quando você quer amarrar mais de um atributo a outras entidades, e transformar todos eles na sua chave primaria composta, para que essa entidade contenha todos esses atributos únicos, muito útil em relacionamentos @ManyToOne ou @OneToMany. Veja esse exemplo abaixo:

@Getter  
@Setter  
@Entity  
@Table(name = "ordered\_item")  
@IdClass(OrderedItemId.class)  
@EqualsAndHashCode(onlyExplicitlyIncluded = true)  
public class OrderedItem {  
 @Id  
 @EqualsAndHashCode.Include  
 @Column(name = "order\_id")  
 private Integer purchaseOrderId; // Composite Keys  
  
 @Id  
 @EqualsAndHashCode.Include  
 @Column(name = "product\_id")  
 private Integer productId; // Composite Keys  
  
 @ManyToOne(optional = false)  
 @JoinColumn(name = "order\_id", insertable = false, updatable = false)  
 private PurchaseOrder purchaseOrder;  
  
 @ManyToOne(optional = false)  
 @JoinColumn(name = "product\_id", insertable = false, updatable = false)  
 private Product product;  
  
 @Column(name = "product\_price")  
 private BigDecimal productPrice;  
  
 private Integer quantity;  
}

Repare que todos os atributos de chave primária estão marcados com @Id, e não utilizamos o parâmetro de geração automática: strategy = GenerationType.IDENTITY. Também precisamos criar uma classe auxiliar para marcar essas chaves compostas, e para marcar utilizamos a annotation @IdClass para marcar que esta classe está utilizando a outra classe como ID. Veja o exemplo:

@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@EqualsAndHashCode(onlyExplicitlyIncluded = true)  
public class OrderedItemId implements Serializable {  
 /\*  
 This class is used for map composite key used in the class OrderedItem class.  
 \*/  
 @EqualsAndHashCode.Include  
 private Integer purchaseOrderId;  
  
 @EqualsAndHashCode.Include  
 private Integer productId;  
}

Utlizamos algumas annotations do lombok, não utilizamos annotations do JPA aqui.

### Mapeando chave composta com @EmbeddedId

Lorem

## Mapeando chave primária e estrangeira na mesma coluna com @MapsId

A annotation @MapsId serve para setar um atributo para que ele receba um valor de outro atributo em uma entidade externa ou interna na mesma classe, e esse valor seja o ID da entidade, você pode passar ele sem parâmetros ou com parâmetros para endereçar a entidade ao atributo desejado.

Veja esse exemplo:

public class Invoice {  
 @Id  
 @EqualsAndHashCode.Include  
 @Column(name = "order\_id")  
 private Integer id;  
  
 @MapsId  
 @OneToOne(optional = false)  
 @JoinColumn(name = "order\_id")  
 private PurchaseOrder purchaseOrder;

private String xml;

@Column(name = "issue\_date")  
 private Date issueDate;  
}

No código acima estamos atribuindo o valor do atributo PurchaseOrder para que seja o ID da entidade, e marcamos esse atributo com MapsId para que o JPA faça todo o trabalho de gerar o ID de forma automática.

Agora veja este outro exemplo, que utiliza uma classe externa com @EmbeddedId:

public class OrderedItem {  
  
 @EmbeddedId  
 private OrderedItemId id;  
  
 @MapsId("purchaseOrderId")  
 @ManyToOne(optional = false)  
 @JoinColumn(name = "order\_id")  
 private PurchaseOrder purchaseOrder;  
  
 @MapsId("productId")  
 @ManyToOne(optional = false)  
 @JoinColumn(name = "product\_id")  
 private Product product;  
  
 @Column(name = "product\_price")  
 private BigDecimal productPrice;  
  
 private Integer quantity;  
}

Ela utiliza esta classe para os Ids:

@Embeddable  
@EqualsAndHashCode(onlyExplicitlyIncluded = true)  
public class OrderedItemId implements Serializable {  
  
 @EqualsAndHashCode.Include  
 @Column(name = "order\_id")  
 private Integer purchaseOrderId; // Composite Keys  
  
 @EqualsAndHashCode.Include  
 @Column(name = "product\_id")  
 private Integer productId; // Composite Keys  
}

Essa abordagem é chamada de Composite Keys, repare que agora utilizamos um parâmetro na annotation @MapsId, inserimos o nome do atributo a ser utilizado, preste atenção para não errar e inserir o parâmetro da coluna do banco de dados, precisa inserir o nome do atributo da classe, não a coluna do BD.

Isso traz uma grande comodidade pois não precisamos mais ter que instanciar uma entidade dependente depois que outra já estiver sido criada e não precisamos utilizar o flush para forçar o update no BD antes, veja o resultado:

public void saveItem() {  
 entityManager.getTransaction().begin();  
  
 Client client = entityManager.find(Client.class, 1);  
 Product product = entityManager.find(Product.class, 1);  
  
 PurchaseOrder purchaseOrder = new PurchaseOrder();  
 purchaseOrder.setClient(client);  
 purchaseOrder.setCreationDate(LocalDateTime.now());  
 purchaseOrder.setStatus(StatusOrder.**WAITING**);  
 purchaseOrder.setTotal(product.getPrice());  
  
 OrderedItem orderedItem = new OrderedItem();  
 orderedItem.setId(new OrderedItemId());  
 orderedItem.setPurchaseOrder(purchaseOrder);  
 orderedItem.setProduct(product);  
 orderedItem.setProductPrice(product.getPrice());  
 orderedItem.setQuantity(1);  
  
 entityManager.persist(orderedItem);  
 entityManager.persist(purchaseOrder);  
 entityManager.getTransaction().commit();  
 entityManager.clear();  
  
 PurchaseOrder verifyPurchaseOrder = entityManager.find(PurchaseOrder.class, purchaseOrder.getId());  
 assertNotNull(verifyPurchaseOrder);  
 assertFalse(verifyPurchaseOrder.getOrderedItems().isEmpty());  
}

Agora o JPA faz o update automaticamente, e só precisamos instanciar a classe embedded sem os atributos getId das duas propriedades, repare que só instanciamos de forma simples:

orderedItem.setId(new OrderedItemId());

## Declarando propriedades transientes com @Transient

Propriedades transientes são propriedades ignoradas pelo JPA, ignoradas completamente, a propriedade anotada com @Transient não é persistida no BD, não da para fazer buscas, você só vai poder trabalhar com a propriedade por meio de call-backs. Veja o exemplo:

public class Client {  
 @Id  
 @EqualsAndHashCode.Include  
 @GeneratedValue(strategy = GenerationType.**IDENTITY**)  
 private Integer id;

private String name;

@Transient  
 private String firstName;

@Enumerated(EnumType.**STRING**)  
 private ClientGender gender;

@OneToMany(mappedBy = "client")  
 private List<PurchaseOrder> purchaseOrders;

@PostLoad  
 public void configFirstName() {  
 if (name != null && !name.isBlank()) {  
 int index = name.indexOf(" ");  
 if (index > -1) {  
 firstName = name.substring(0, index);  
 }  
 }  
 }  
}

## Mapeando coleções de tipos básicos com @ElementCollection

A annotation @ElementCollection pode ser utilizada para criar coleções de tipos básicos do Java, onde o JPA irá gerenciar a lista criando uma tabela automaticamente para guardar os valores, você pode customizar o nome da tabela criada e das colunas, veja o código abaixo:

@ElementCollection  
@CollectionTable(name = "product\_tag",  
 joinColumns = @JoinColumn(name = "product\_id"))  
@Column(name = "tag")  
private List<String> tags;

Caso você não configure um nome para a tabela e colunas desejadas o JPA fará isso automaticamente com nomes gerados a partir do nome do atributo.

## Mapeando coleções de objetos embutidos com @ElementCollection

Lorem