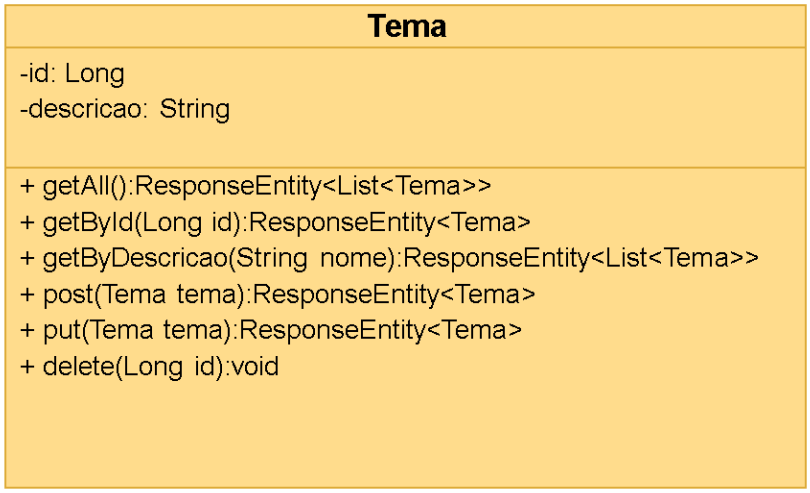
Projeto 02 - Blog Pessoal - Relacionamento entre Classes 01

O que veremos por aqui:

1. Apresentação do Recurso Tema
2. Criar a Classe Model Tema
3. Criar o Relacionamento entre as Classes Postagem e Tema

1. O Recurso Tema

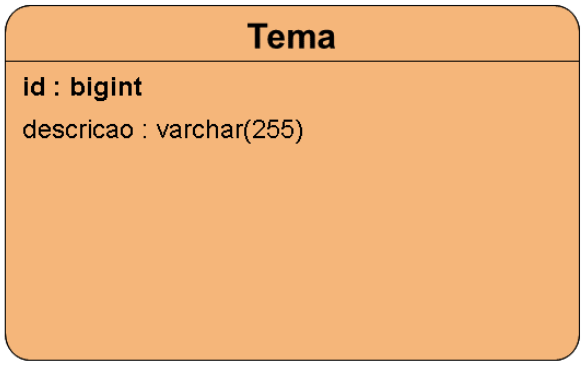
Nesta etapa vamos começar a construir o Recurso Tema. Veja o Diagrama de Classes abaixo:



A **Classe Tema** servirá de modelo para construir a tabela **tb_temas** (Entidade) dentro do nosso Banco de dados **db_blogpessoal**. Os campos (Atributos) da tabela serão os mesmos que estão definidos no Diagrama de Classes acima. Além de construirmos a Classe Tema, também faremos o Relacionamento com a Classe Postagem, construída anteriormente.

Na próxima etapa vamos construir a **Interface TemaRepository**, que irá nos auxiliar na interação com o Banco de dados e a **Classe TemaController**, onde serão implementados os 6 métodos descritos no Diagrama de Classes acima.

Depois de criar a Classe Model Tema, vamos executar o projeto Blog Pessoal no STS. Após a execução veremos que será criado no Banco de dados **db_blogpessoal** a tabela **tb_temas**. Veja abaixo como ficará a estrutura da nossa tabela através do **Diagrama de Entidade e Relacionamentos (DER)**:



O Dicionário de dados da nossa tabela tb_tema será o seguinte:

Atributo	Tipo de dado	Descrição	Chave
id	bigint	Identificador único	PK
descricao	varchar(255)	Título da postagem	

ALERTA DE BSM: ***Mantenha a Atenção aos Detalhes ao criar o Recurso Tema. Todas as Camadas (Pacotes: Model, Repository e Controller), já foram criadas no Recurso Postagem.***

DICA: ***Caso você tenha alguma dúvida sobre como criar a Classe, executar o projeto, entre outras, consulte a Documentação do Recurso Postagem.***

Passo 01 - Criar a Classe Tema na Camada Model

Agora vamos criar a segunda Classe Model que chamaremos de **Tema**.

1. Clique com o botão direito do mouse sobre o **Pacote Model** (**com.generation.blogpessoal.model**), na Source Folder Principal (**src/main/java**), e clique na opção **New → Class**
2. Na janela **New Java Class**, no item **Name**, digite o nome da Classe (**Tema**), e na sequência clique no botão **Finish** para concluir.

Agora vamos criar o código da **Classe Model Tema**:

```
@Entity
@Table(name = "tb_temas")
public class Tema {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull(message = "O atributo Descrição é obrigatório")
    private String descricao;

    /*Insira os Getters and Setters*/

    public Long getId() {
        return this.id;
    }

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

    public String getDescricao() {
        return this.descricao;
    }
```

```
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

}
```

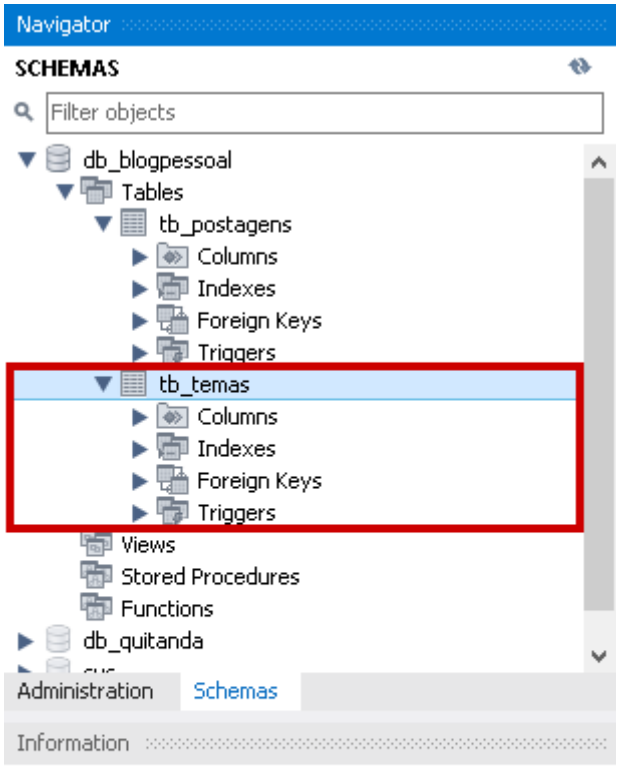
Observe que no atributo **descricao** utilizamos a anotação **@NotNull**, que **não permite que o atributo seja Nulo, mas permite que ele contenha apenas Espaços em branco**. Você pode configurar uma mensagem para o usuário através do atributo **message**.


Documentação: *@NotNull*

Para concluir, não esqueça de Salvar o código (**File → Save All**).

🦶 Passo 02 - Executar o projeto e Checar o Banco de dados

1. Execute o projeto e verifique no **MySQL Workbench** se a tabela **tb_temas** foi criada no Banco de dados **db_blogpessoal**, como mostra a figura abaixo:

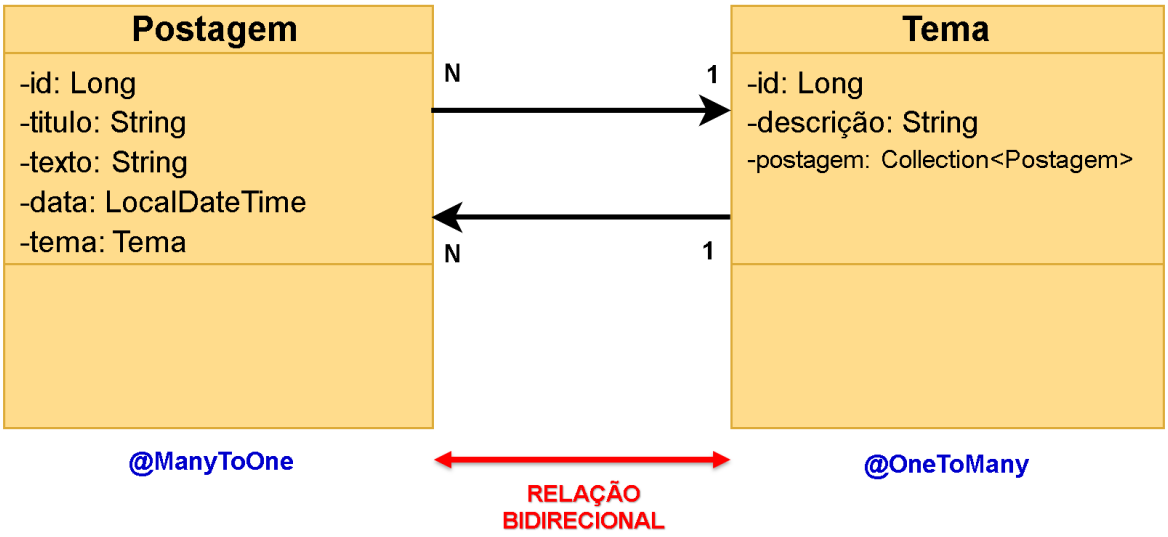


2. Relacionamento de Classes

Mapeamento Objeto-Relacional (ORM) é o processo de conversão de Objetos Java em Tabelas (Entidades) de Banco de dados. Em outras palavras, isso nos permite interagir com um Banco de dados Relacional sem nenhum código SQL. A **Java Persistence API (JPA)** é uma especificação que define como persistir dados em aplicativos Java. O foco principal do JPA é a camada ORM.

O **JPA** simplifica o tratamento do modelo de Banco de dados Relacional nos aplicativos Java quando mapeamos cada Tabela para uma única Classe de entidade (Model). Assim como no SQL, precisamos criar Relacionamentos entre as tabelas, no JPA também precisamos criar **Relacionamentos entre as Classes** e desta forma construirmos os Relacionamentos entre as Tabelas no Banco de dados como fizemos no MySQL.

Nesta etapa vamos construir o Relacionamento do Recurso Tema com o Recurso Postagem. Veja o Diagrama de Classes abaixo:



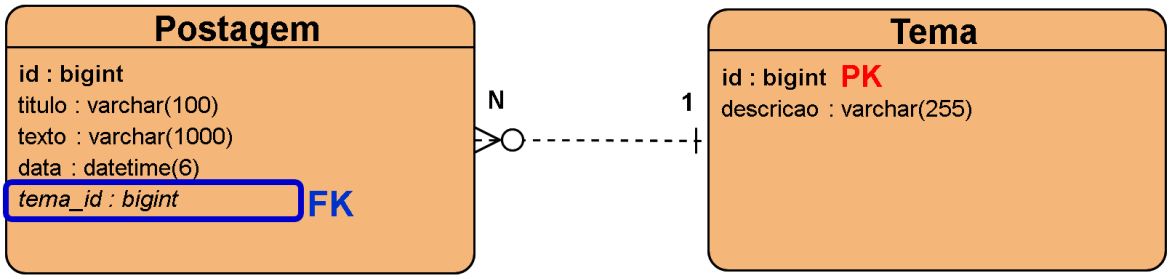
Para construirmos o Relacionamento entre Classes, assim como no SQL, precisamos definir a cardinalidade do Relacionamento. Para definir a Cardinalidade, o JPA utiliza as anotações abaixo:

Cardinalidade	Anotação
1:1	@OneToOne
1:N	@OneToMany
N:1	@ManyToOne
N:M	@ManyToMany

No modelo Relacional, 1:N e N:1 são a mesma coisa, entretanto no Relacionamento de Classes, além da Cardinalidade é necessário definir a Direção do Relacionamento, ou seja, Qual Classe "enxerga" Qual Classe e de que forma enxerga?

No Modelo Relacional todo Relacionamento é **Unidirecional**, ou seja, **apenas a Tabela que possui a Chave Estrangeira acessa a outra Tabela**. No Relacionamento de Classes, existe a possibilidade do Relacionamento ser **Bidirecional**, ou seja, uma **Classe acessa a outra e vice-versa, independente de possuir ou não a Chave Estrangeira**.

Depois de criar o Relacionamento entre as Classes e executar o projeto Blog Pessoal no STS, veremos que será criado no MySQL a Relação entre as tabelas **tb_postagens** e **tb_temas** Unidirecional. Veja abaixo como ficará a estrutura da nossa tabela através do **Diagrama de Entidade e Relacionamentos (DER)**:



Como o JPA faz o mapeamento das Tabelas em Objetos, caso o Relacionamento Bidirecional esteja habilitado, a Relação funcionará independente do Banco de Dados ser Unidirecional.

Vamos construir o Relacionamento Bidirecional (1:N) entre as nossas Classes Tema e Postagem como veremos a seguir.

🦶 Passo 01 - Criar a Relação ManytoOne na Classe Postagem

A Classe Postagem será o lado N:1, ou seja, **Muitas Postagens podem ter apenas Um Tema**. Para criar a Relação vamos inserir depois do último atributo da Classe Postagem (data), as 3 linhas destacadas em vermelho na figura abaixo:

```
33
34      @UpdateTimestamp
35      private LocalDateTime data;
36
37      @ManyToOne
38      @JsonIgnoreProperties("postagem")
39      private Tema tema;
40
41      /*Insira os Getters and Setters*/
42
43      public Long getId() {
44          return this.id;
45      }
46
```

Linha 37: A anotação **@ManyToOne** indica que a Classe Postagem será o lado N:1 e terá um **Objeto da Classe Tema**, que no modelo Relacional será a **Chave Estrangeira na Tabela tb_postagens (tema_id)**.

Linha 38: A anotação **@JsonIgnoreProperties** indica que uma parte do JSON será ignorado, ou seja, como a Relação entre as Classes será do tipo Bidirecional, ao listar o Objeto Postagem numa consulta, por exemplo, o Objeto Tema, que será criado na linha 39, será exibido como um **"Sub Objeto"** do Objeto Postagem, como mostra a figura abaixo, devido ao Relacionamento que foi criado.

```
{
     "id": 1,
     "titulo": "Minha primeira postagem",
     "texto": "Blog Pessoal funcionando!",
     "data": "2021-12-30T16:03:50.629639",
     "tema": {
          "id": 1,
          "descricao": "Spring",
     }
}
```

Ao exibir o Objeto Tema, o Objeto Postagem será exibido novamente e na sequência Tema será exibido novamente, criando um looping infinito dentro do JSON, devido a relação Bidirecional. Para impedir o looping infinito e o travamento da nossa aplicação, utilizamos anotação **@JsonIgnoreProperties** para ignorar o Objeto da Classe Postagem, interrompendo a repetição.

Linha 39: Será criado um Objeto da Classe Tema, que receberá os dados do Tema associado ao Objeto da Classe Postagem. Este Objeto representa a Chave Estrangeira da Tabela **tb_postagens (tema_id)**.

Depois do último Método Set, vamos acrescentar os Métodos Get e Set para o novo atributo que foi adicionado na Classe Postagem. Veja o código completo abaixo:

```
@Entity
@Table(name = "tb_postagens")
public class Postagem {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "O atributo título é Obrigatório!")
    @Size(min = 5, max = 100,
    message = "O atributo título deve conter no mínimo 05
    e no máximo 100 caracteres")
    private String titulo;

    @NotBlank(message = "O atributo texto é Obrigatório!")
    @Size(min = 10, max = 1000,
    message = "O atributo texto deve conter no mínimo 10
    e no máximo 1000 caracteres")
    private String texto;

    @UpdateTimestamp
    private LocalDateTime data;

    @ManyToOne
    @JsonIgnoreProperties("postagem")
    private Tema tema;

    public Long getId() {
        return this.id;
    }

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

    public String getTitulo() {
        return this.titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getTexto() {
        return this.texto;
    }

    public void setTexto(String texto) {
        this.texto = texto;
    }

    public LocalDateTime getData() {
        return this.data;
    }

    public void setData(LocalDateTime data) {
        this.data = data;
    }

    public Tema getTema() {
        return tema;
    }
```

```
    public void setTema(Tema tema) {
        this.tema = tema;
    }

}
```

DICA: ***Toda vez que você adicionar um novo Atributo na sua Classe, não esqueça de criar os Métodos GET e SET do Atributo. Caso contrário, você não conseguirá visualizar ou atualizar os dados do Atributo.***

[Documentação: *@ManyToOne*]

[Documentação: *@JsonIgnoreProperties*]

🦶 Passo 02 - Criar a Relação OneToMany na Classe Tema

A Classe Tema será o lado 1:N, ou seja, **Um Tema pode ter Muitas Postagens**. Para criar a Relação vamos inserir depois do último atributo da Classe Tema (descricao), destacadas em vermelho na figura abaixo:

```
23
24      @NotNull(message = "O atributo Descrição é obrigatório")
25      private String descricao;
26
27      @OneToMany(mappedBy = "tema", cascade = CascadeType.ALL)
28      @JsonIgnoreProperties("tema")
29      private List<Postagem> postagem;
30
31      /*Insira os Getters and Setters*/
32
33      public Long getId() {
34          return this.id;
35      }
36
```

Linha 27: A anotação **@OneToMany** indica que a Classe Tema será o lado 1:N e terá **uma Collection List contendo Objetos da Classe Postagem**. Como a Relação entre as Classes será Bidirecional, a Collection List trará a lista com todos os Objetos da Classe Postagem relacionados com cada Objeto da Classe Tema. Como a Classe Tema será a **Classe Proprietária** da Relação, precisamos adicionar dois parâmetros:

- **mappedBy:** Uma vez que definimos o lado Proprietário do Relacionamento (Classe Tema), o Hibernate já possui todas as informações necessárias para mapear o relacionamento em nosso Banco de dados, criar a Chave Estrangeira e o Relacionamento propriamente dito. Para tornar essa associação Bidirecional no modelo Orientado a Objetos, informamos no

parâmetro **mappedBy** o nome do Atributo da Classe Proprietária, que foi criado na **Classe Postagem (Tema tema)**, que será o Objeto de referência na Relação.

- **cascade:** Os relacionamentos de entidade geralmente dependem da existência de outra entidade, por exemplo, o relacionamento **Tema → Postagem**. Sem o Tema, a entidade Postagem não tem nenhum significado próprio. Quando excluímos a entidade Tema, nossa entidade Postagem também deve ser excluída. **"Cascatear" (cascade)**, é a maneira de conseguir isso. Quando executamos alguma ação na entidade de destino (Tema), a mesma ação será aplicada à entidade associada (Postagem).
- **CascadeType.ALL:** Quando um Objeto da Classe Tema for modificado, consultado ou apagado, todos os Objetos da Classe Postagem também serão modificados, consultados ou apagados. O Inverso não é verdadeiro.

Linha 28: A anotação **@JsonIgnoreProperties** indica que uma parte do JSON será ignorado, ou seja, assim como fizemos na Classe Postagem (Linha 39), também faremos na Classe Tema para impedir o looping infinito.

Linha 29: Será criado uma Collection List contendo Objetos da Classe Postagem, que receberá todos os Objetos da Classe Postagem associadas a cada Objeto da Classe Tema.

Depois do último Método Set, vamos acrescentar os Métodos Get e Set para o novo atributo que foi adicionado na Classe Postagem. Veja o código completo abaixo:

```
@Entity
@Table(name = "tb_temas")
public class Tema {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull(message = "O atributo Descrição é obrigatório")
    private String descricao;

    @OneToMany(mappedBy = "tema", cascade = CascadeType.ALL)
    @JsonIgnoreProperties("tema")
    private List<Postagem> postagem;

    /*Insira os Getters and Setters*/

    public Long getId() {
        return this.id;
    }

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


    public String getDescricao() {
        return this.descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public List<Postagem> getPostagem() {
```

```
        return this.postagem;
    }

    public void setPostagem(List<Postagem> postagem) {
        this.postagem = postagem;
    }

}
```


Documentação: *@OneToMany*


Documentação: *mappedBy*

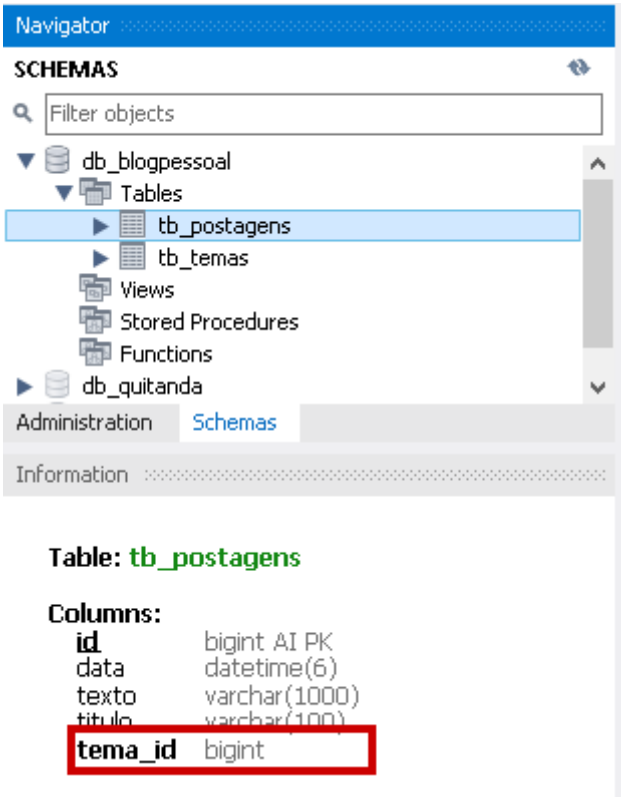

Documentação: *cascade*

Para concluir, não esqueça de Salvar o código (**File → Save All**) e verificar se o Projeto está em execução


Código fonte do Projeto

 Passo 03 - Executar o projeto e Checar o Banco de dados

1. Execute o projeto e verifique no **MySQL Workbench** se a **Chave Estrangeira (tema_id)** foi criada na Tabela **tb_Postagens**, no Banco de dados **db_blogpessoal**, como mostra a figura abaixo:



Anexo I - Tipos de Cascade

Tipo	Descrição
PERSIST	Ele propaga a operação de persistir um objeto **Pai** para um objeto **Filho**, assim quando salvar a Entidade Cliente, também será salvo todas as Entidades Telefone associadas.
MERGE	Ele propaga a operação de atualização de um objeto **Pai** para um objeto **Filho**, assim quando atualizadas as informações da Entidade Cliente, também será atualizado no banco de dados todas as informações das Entidades Telefone associadas.
REMOVE	Ele propaga a operação de remoção de um objeto **Pai** para um objeto **Filho**, assim quando remover a Entidade Cliente, também será removida todas as entidades Telefone associadas.
REFRESH	Ele propaga a operação de recarregar de um objeto **Pai** para um objeto **Filho**, assim, quando houver atualização no banco de dados na Entidade Cliente, todas as entidades Telefone associadas serão recarregadas do banco de dados.
ALL	Corresponde a todas as operações acima (MERGE, PERSIST, REFRESH e REMOVE).
DETACH	A operação de desanexação remove a entidade do contexto persistente. Quando usamos CascaseType.DETACH, a entidade filha também é removida do contexto persistente

 **Documentação: *CascadeType***