Elaborando relacionamentos entre modelos no Laravel

Nesse artigo explicarei de forma detalhada como funciona a criação de relacionamentos entre modelos no Laravel.

O que são modelos (models)? Cada tabela de banco de dados tem um “Modelo” correspondente em nossa aplicação localizada na raiz do diretório **app/** que é usado para interagir com essa tabela. Os modelos permitem que você consulte dados em suas tabelas, bem como inserir novos registros e também apagar os dados caso necessário.

Neste post irei partir do princípio de que você já sabia o básico sobre banco de dados, ta bem? Então tá bem, vamos lá!

Tópicos do artigo

Criando ambiente

Em cada exemplo desse artigo iremos trabalhar com exemplos singulares.

Relacionamento 1:1

```
--- user
-- id
-- name
-- email

--- addresses
-- id
-- user_id
-- address
-- city
-- state
```

O relacionamento 1:1 é o relacionamento mais básico que existe. Ele consiste em dizer que um registro em uma tabela pertence a outro registro em outra tabela. Nesse relacionamento iremos trabalhar tanto o caminho de ida quanto o caminho de volta do relacionamento 1:1. Vamos começar estruturando o nosso model de **User**.

User.php

No model de nossos usuários iremos criar a seguinte função:

```
public function address()
    {
        return $this->hasOne(Address::class, 'user_id', 'id');
    }
```

Leitura da função: o usuário (**user**) possui um (**hasOne**) endereço (**Address::class**) onde na tabela **address** o meu **id** é uma foreign key no campo **user_id**.

Com o nosso relacionamento pronto, para sabermos qual é a cidade de um determinado usuário basta usarmos o seguinte código:

```
$user->address->city;
```

O atributo **address** é o nome da função que criamos para o nosso relacionamento hasOne.

Address.php

Agora em nosso model Address nós iremos realizarmos o caminho inverso do nosso relacionamento hasOne.

```
    public function user()
    {
        return $this->belongsTo(User::class, 'user_id', 'id');
    }
```

Leitura da função: o endereço pertence a um (**belongsTo**) usuário (**user**), onde o campo **user_id** na tabela **address** é uma foreing key que se relaciona com o id da tabela **users**

Com o nosso relacionamento inverso definido, podemos identificar qual é o nome de usuário através de um determinado endereço, por exemplo:

```
$address->user->name;
```

E mais uma vez, o atributo **user** é o nome da função que criamos para o nosso relacionamento belongsTo.

Relacionamento 1:N

```
--- user
-- id
-- name
-- email

--- posts
-- id
-- user_id
```

```
-- slug
-- content
```

O relacionamento 1:N diz que um model poderá se relacionar com mais de um registro em outra tabela. Você pode estar se queixando de que essa tabela de posts é bem parecida estruturalmente com a tabela do exemplo anterior, porém a lógica do negócio muda de acordo com as necessidades do projeto que está sendo desenvolvido. Por isso, antes de iniciar qualquer projeto é muito importante modelar o banco seguindo as normalizações de dados.

User.php

```
public function posts()
{
    return $this->hasMany(Post::class, 'user_id', 'id');
}
```

Leitura da função: o usuário (**user**) possui muitos (**hasMany**) posts (**Post::class**) onde na tabela **posts** o meu **id** é uma foreign key no campo **user_id**.

Com o nosso relacionamento **hasMany** estruturado para listarmos os posts de um determinado usuário basta chamar da seguinte forma:

```
@foreach ($user->posts as $post)
    <h3>{{ $post->title }}</h3>
    <p>{{ $post->description }}</p>
@endforeach
```

E mais uma vez, o atributo **posts** é o nome da função que criamos para o nosso relacionamento **hasMany**. Ele está dentro de um foreach pois como é um relacionamento 1:n, o Laravel já entende que o retorno será uma coleção de dados, sendo assim o uso do foreach é necessário para a execução da listagem dos dados sem erros.

Posts.php

Agora em nosso model Post nós iremos realizarmos o caminho inverso do nosso relacionamento **hasMany**. O modelo dessa função de retorno tanto do relacionamento **hasOne** e **hasMany** é exatamente igual, ambos irão ter usar a função de retorno **belonsTo**.

```
public function author()
{
    return $this->belongsTo(User::class, 'user_id', 'id');
}
```

Leitura da função: o post pertence a um (**belongsTo**) autor (**User::class**), onde o campo **user_id** na tabela **posts** é uma foreing key que se relaciona com o id da tabela **users**

Com o nosso relacionamento **belongsTo** definido, podemos identificar qual é o autor de um post da seguinte forma:

```
$post->author->name;
```

Dessa vez mudamos o nome do atributo para **author**, essa mudança ocorre da declaração da função de retorno que criamos para o nosso relacionamento **belongsTo**.

Relacionamento N:N

O relacionamento N:N a princípio pode parecer assustador, porém é tão simples quantos os anteriores. O que consiste em um relacionamento é definir que ambos os dados de duas tabelas podem se relacionar entre si através de uma tabela intermediaria infinita vezes. Iremos utilizar nesse exemplo as tabelas de **posts** e **categories** e para relacionar os dados delas duas, iremos utilizar a tabela intermediaria **post_categorie**.

Estrutura do banco:

```
--- posts
-- id
-- title
-- content

-- categories
-- id
-- title

--- post_categorie
-- id
-- post_id
-- category_id
```

Post.php

```
public function categories()
    {
        return $this->belongsToMany(Category::class, 'post_category',
'post_id', 'category_id');
    }
```

Leitura da função: o post se relaciona com as categorias (**Category::class**) através da tabela **post_category**, sendo o **id** da tabela do meu **model** o **post_id** e o campo da categoria que estou me relacionando sendo o **category_id**.

Com o nosso relacionamento **belongsToMany** estruturado, para listarmos as categorias de um determinado post basta chamar a função da seguinte forma:

```
@foreach ($post->categories as $category)
     {{ $category->name }}
@endforeach
```

E mais uma vez, o atributo **categories** é o nome da função que criamos para o nosso relacionamento **belongsToMany**. Ele está dentro de um foreach pois como é um relacionamento n:n, o Laravel já entende que o retorno desse chamado será uma coleção de dados, sendo assim o uso do foreach é necessário para a execução da listagem correta dos dados.

Category.php

A relação contrária do nosso model de categorias não muda muito a estrutura da relação anterior, na verdade o retorno também é **belongsToMany**.

```
public function posts()
    {
        return $this->belongsToMany(Post::class, 'post_category',
'category_id', 'post_id');
    }
```

A leitura da função é a mesma que a anterior, mudando apenas a ordem dos últimos dois parâmetros.

E com o nosso relacionamento **belongsToMany** estruturado, para listarmos as postagens de uma determinada categoria basta chamar a função da seguinte forma:

```
@foreach ($category->posts as $post)
    <h3>{{ $post->title }}</h3>
    <p>{{ $post->description }}</p>
{{ $post->name }}
@endforeach
```

Uma recomendação a seguir é: toda função que for retornar uma coleção de dados, é sugerido que o nome dessa função seja no plural. E toda função que for retornar apenas um dado, seja usada no singular.

Inserindo relação na tabela pivo

Em nosso exemplo acima temos a tabela **post_category** e se você percebeu, ela não possui um controlador e nem um model de gestão dos dados. Essa tabela é chamada de **tabela pivo**, essa é apenas uma tabela de relação, portanto não é necessário a criação de um model e nem de um controlador para ela.

Beleza, mas como eu insiro e retiro os dados dessa tabela?

```
$post = Post::find(1);
$post->categories()->attach([3]);
```

Para isso iremos utilizar o método acima. **$post** é o nosso post atual, **categories()** é a nossa relação no nosso model e o **attach()** é o método que faz a inserção na tabela pivo. Esse método pode receber ou um número inteiro ou um array. No exemplo acima foi colocado um array para ilustrar melhor a situação.

Para remover um ou mais, podemos utilizar o método de **detach()**

```
$post->categories()->detach([3]);
```

Porém, o melhor método para utilizarmos nesse modelo de relação da tabela pivo, é o método **sync()**, pois ele já faz a assinação e a desassociarão dos dados na tabela pivo de forma automática. Exemplo:

Vamos dizer que o post 5 esta associado a 3 categorias: 9, 4 e 10, mas vamos dizer que eu quero mudar esse post e colocar 10 e 5, ou seja: teremos que remover a categoria 9 e o 4 e adicionar a categoria 5.

```
$post->categories()->sync([5, 10]);
// 5, 10
```

Agora caso você precise adicionar, mas sem remove o que já existia, podemos utilizar o método **syncWithoutDetaching()**

```
$post->categories()->syncWithoutDetaching([5, 6, 7]);
// 5, 10, 6, 7
```

Relacionamento polimórfico

O relacionamento polimórfico é utilizado quando precisamos usar a estrutura de uma única tabela para armazenar dados de modelos diferentes. Por exemplo: em uma aplicação onde tantos os usuários quanto as empresas precisam ter o registro de seus endereços.

Estrutura da tabela da tabela para esse exmplo:

```
--- user
-- id
-- name
-- CPF

--- companies
-- id
-- name

-- CNPJ

--- addresses
```

```
-- id
-- model_type
-- model_id
-- address
-- city
-- state
```

Migration addresses

Em nosso arquivo de migration de nossa tabela de endereços, não iremos mais utilizar a FK de **user_id** como no exemplo do nosso relacionamento de 1:1. Ela vai ficar assim:

```
$table->increments('id');
$table->morphs('model');
$table->string('address');
$table->string('city');
$table->string('state');
```

A função **morphs('model')** irá criar em nossa na tabela addresses em nossa base de dados dois campos:

model_type = o modelo que esse registro pertence – User ou Company
model_id = o id do modelo que esse registro pertence

Address.php

No nosso model Address, nós iremos sinalizar ao Laravel de que esse modelo é polimórfico.

```
public function address()
{
   return $this->morphTo();
}
```

Leitura da função: o modelo **Address** é um model polimórfico e ele irá se relacionar com seus respectivos models através da função **address()**.

User.php

Agora em nosso model User, nós iremos realizar a criação correta do relacionamento

```
public function address()
{
   return $this->morphOne(Address::class, 'address');
}
```

Leitura da função: o modelo User possui um relacionamento polimórfico de **1:1** (morphOne) com Address através da do parâmetro **address**

Company.php

```
public function address()
{
   return $this->morphMany(Address::class, 'address');
}
```

Leitura da função: o modelo User possui um relacionamento polimórfico de **1:n** (**morphMany**) com Address através da do parâmetro **address**.

Criando endereços

```
// Para empresas

$company->address->create([
'address
' => 'Rua Rio Jatapu',
'city
 => 'Boa Vista',
'state' => 'RR'
]);

// Para usuários
$user->address->create([
'address
' => 'Rua Rio Amazonas',
'city
 => 'Camboriú',
'state' => 'SC'
]);
```

Ao visualizarmos o nosso banco esse será o resultado:

 Na imagem há uma tabela do banco de dados contendo as informações dos registros dos modelos de empresas e usuários

A forma de resgatarmos os é a mesma das anteriores, apenas fique atento para o retorno se é de 1:1 ou 1:n.

Espero que tenham gostado 🙂

23 de outubro de 2020 Matheus Teixeira Laravel hasMany, laravel, models, morph, relacionamentos

2 comentários em “Elaborando relacionamentos entre modelos no Laravel”

Francisco

24 de outubro de 2020 às 18:54

Cara muito bom o artigo mas faltou conteúdo na relação polimórfica e você não abordou o relacionamento Through.
Como fica a questão de N:N polimórfica?
Como é feita a relação inversa neste tipo de relacionamento?
Tem muita gente começando que poderia utilizar o teu artigo como fonte tira-dúvidas.
Se puder complementa essas informações.

Rafael Melo

25 de outubro de 2020 às 12:55

Parabéns pela matéria! ficou muito boa!
