RELACIONAMENTOS COM ELOQUENT

Relacionamentos com Eloquent

Vamos começar agora a falar sobre relacionamentos de uma base relacional (nosso banco de dados) e a representação destes relacionamentos do ponto de vista de Objetos, representados por nossos Models junto com o Eloquent.

Vamos mostrar os relacionamentos e suas nuances aplicados em nosso blog onde teremos as seguintes representações:

Relacionamento 1:N(Um para Muitos): Autor x Postagens;

Relacionamento N:N(Muitos para Muitos: Postagens & Categorias;

Já temos definido na base o primeiro relacionamento da listagem acima e vamos começar por ela. Para o relacionamento entre postagem e categoria, vou criar os insumos(Models, Controllers e etc) no momento que forem necessários, até para darmos uma relembrada.

Relacionamento 1:N (Um para Muitos e Inverso)

Primeiramente vamos definir nossa relação entre Autor(User) e suas Postagens(Post). Para isto precisamos definir métodos dentro de cada model que representem esta ligação.

Primeiramente vamos criar o método do ponto de vista de Post em relação ao nosso User. Veja o método abaixo adicionado ao model Post:

```
1   public function user()
2   {
3           return  $this->belongsTo(User::class);
4   }
```

Definimos o método acima para representar a ligação entre nossos models, neste caso, quando precisarmos acessar dados desta relação ou criar um dados vamos chamar o método user do ponto de vista de Post.

Relacionamento 1:N (Um para Muitos e Inverso)

O método belongsTo vêm do Eloquent e indica que o model Post pertence a(belongsTo) User, por isso passei o nome qualificado do model User no primeiro parâmetro do método belongsTo.

Outro ponto a ressaltar aqui é que, o Laravel vai tentar resolver o nome da coluna referência na tabela posts por meio do nome do método, se coloquei user ele vai buscar dentro da tabela post, no banco de dados, pela referência user_id.

Relacionamento 1:N (Um para Muitos e Inverso)

Se por ventura você usou um nome de coluna, que representa a referência na sua tabela diferente, por exemplo, não usou o user_id mas sim author_id. Neste caso você precisa informar para o Eloquent o nome da coluna, veja um trecho exemplo abaixo:

```
1    return $this->belongsTo(User::class,    'author_id');
```

Desta forma quando o Eloquent for acessar suas tabelas e gerar as queries sql dos acessos irá buscar pela coluna author_id. Lembrando, isso vale apenas para o nome da coluna, que recebe a chave estrangeira, que não seja user_id em nosso caso.

Relacionamento 1:N (Um para Muitos e Inverso)

Agora como digo que o model User têm muitas postagens? Vamos lá no model User e vamos definir o método abaixo:

```
1    public function posts()
2    {
3            return  $this->hasMany(Post::class);
4    }
```

Se indiquei que o Post pertence a User por meio do método belongsTo, dentro de User eu indico que ele têm muitos(hasMany) posts por meio do método hasMany, informando também o nome do model no primeiro parâmetro, neste caso Post. Agora toda vez que eu precisar acessar as postagens de um usuário, eu irei acessar o método posts para tal processo.

Relacionamento 1:N (Um para Muitos e Inverso)

As definições do ponto de vista do Model são estas, agora, para entendermos melhor vamos realizar algumas queries para testarmos esta relação.

Vamos exibir lá na listagem de posts o autor da postagem. Para isto adicione mais uma coluna no thead, depois da coluna do id(#):

```
1    <th>Autor</th>
```

No tbody vamos adicionar o conteúdo para esta coluna, veja abaixo:

```
1    <td>{{$post->user->name}}</td>
```

Relacionamento 1:N (Um para Muitos e Inverso)

Antes de vermos o resultado no browser, vamos ver na íntegra a view de posts agora, com esta alteração:

```
1  @extends('layouts.app')
2
3  @section('content')
4          <div class="row">
5              <div class="col-sm-12">
6                  <a href="{{route('posts.create')}}" class="btn btn-success float-right">\
7  Criar Postagem</a>
8                  <h2>Postagens Blog</h2>
9                  <div class="clearfix"></div>
10             </div>
11         </div>
12         <table class="table table-striped">
13             <thead>
14                 <tr>
```

Relacionamento 1:N (Um para Muitos e Inverso)

```
15                  <th>#</th>
16                  <th>Autor</th>
17                  <th>Titulo</th>
18                  <th>Status</th>
19                  <th>Criado em</th>
20                  <th>Ações</th>
21              </tr>
22          </thead>
23          <tbody>
24          @forelse($posts as $post)
25              <tr>
26                  <td>{{$post->id}}</td>
27                  <td>{{$post->user->name}}</td>
28                  <td>{{$post->title}}</td>
29                  <td>
30                      @if($post->is_active)
31                          <span class="badge badge-success">Ativo</span>
```

Relacionamento 1:N (Um para Muitos e Inverso)

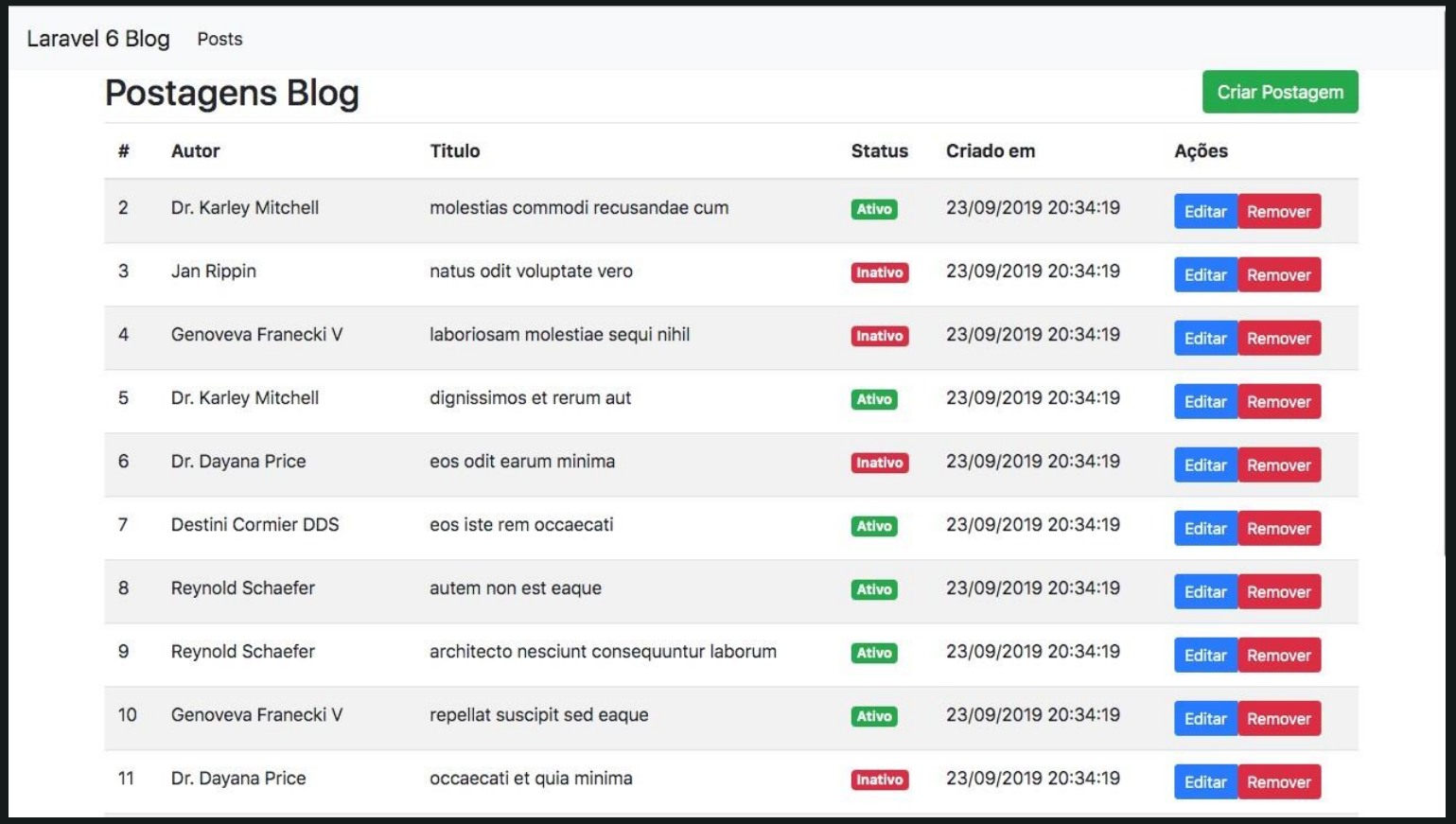
```
32                          @else
33                              <span class="badge badge-danger">Inativo</span>
34                          @endif
35                      </td>
36                      <td>{{date('d/m/Y H:i:s', strtotime($post->created_at))}}</td>
37                      <td>
38                          <div class="btn-group">
39                              <a href="{{route('posts.show', ['post' => $post->id])}}" cla\
40  ss="btn btn-sm btn-primary">
41                                  Editar
42                              </a>
43                              <form action="{{route('posts.destroy', ['post' => $post->id]\


44  )}}" method="post">
45                                  @csrf
46                                  @method('DELETE')
```

Relacionamento 1:N (Um para Muitos e Inverso)

```
47                                  <button type="submit" class="btn btn-sm btn-danger">Remo\
48  ver</button>
49                                  </form>
50                              </div>
51                          </td>
52                      </tr>
53                  @empty
54                      <tr>
55                          <td colspan="4">Nada encontrado!</td>
56                      </tr>
57                  @endforelse
58                  </tbody>
59          </table>
60
61          {{$posts->links()}}
62  @endsection
```

Relacionamento 1:N (Um para Muitos e Inverso)

Veja o resultado:



Relacionamento 1:N (Um para Muitos e Inverso)

Por meio da definição do método user em Post e posts em User criamos as relações entre os objetos dos dois pontos de vistas. Como precisei exibir o autor da postagem precisei chamar o método user e assim consigo acessar os atributos deste user, associado ao post em questão, onde por exemplo peguei o nome dele:

```
1   {{$post->user->name}}
```

Agora você pode está se perguntando, como eu acesso user como se fosse um atributo mas defini um método lá no model?!!

Aqui tecnicamente é bem simples, quando você acessar a ligação como se fosse o atributo da classe uma coleção é retornada, especialmente quando usamos o hasMany e o belongsToMany(veremos este método na próxima sessão) ou o objeto representado o model será retornado, pro caso especial do método belongsTo ou hasOne(veremos este método no próximo exemplo) da ligação.

Relacionamento 1:N (Um para Muitos e Inverso)

Para simplificar:

Como user tem muitos posts o retorno de $user->posts seria a coleção de posts ligadas ao usuário em questão;

Já como uma postagem está ligada ou pertence a apenas um usuário, a chamada $post->user retornará o objeto User com as informações do usuário ligado a aquela postagem.

Relacionamento 1:N (Um para Muitos e Inverso)

Se em algum momento você quiser recuperar as postagens de um usuário, por exemplo do usuário 2, você pode seguir o pensamento do trecho de código abaixo:

```
1  ...
2  #No controller fazer a  busca   pelo usuário
3
4  $user =  User::find(2);
5
6  #Acessar a  ligação de  usuário e  postagens
7
8  $user->posts;  //retornará  as  postagens do  user
```

Inserindo Autor da Postagem

A minha intenção nesta inserção é termos o usuário logado e quando este criar a postagem nós pegamos a referência dele na sessão para adicionarmos no post mas como ainda não chegamos na parte de autenticação vou colocar direto no código a criação desta relação diretamente.

Vamos ao nosso método store lá no PostController. Faça a seguinte alteração no código, o que está assim:

```
1   public function store(Request $request)
2   {
3           //Salvando com mass  assignment
4           $data =  $request->all();
5
6           $data['user_id'] =  1;
7           $data['is_active'] =  true;
8
9           $post =  new Post();
10
11           dd($post->create($data));
12   }
```

Inserindo Autor da Postagem

Ficará assim:

```
1   public function store(Request  $request)
2   {
3               $data  =  $request->all();
4               $data['is_active']  =  true;
5
6               $user  =  User::find(1);
7
8               //Continuamos  a  salvar  com  mass  assignment  mas  por  meio  do  usuário
9               dd($user->posts()->create($data));
10
11   }
```

Não esqueça de importar User:

```
1   use   \App\User;
```

Inserindo Autor da Postagem

Perceba que agora nós inserimos uma nova postagem por meio da ligação que temos com o usuário, como eu quero ter acesso aos métodos da ligação eu preciso chamar de fato o método posts() ao invés de chamar como atributo posts.

O Eloquent ao criar a postagem, irá adicionar a referência do usuário que buscamos por meio do método find automaticamente. Como a postagem pertence ao usuário, defini ele como prioridade quando busquei pelo mesmo e depois adicionei uma postagem ao seu conjunto de posts.

ManyToMany com Eloquent: Categorias e Posts

Agora neste ponto vamos começar a parte da relação de Muitos para Muitos, a relação aqui será entre Posts e Categorias. Primeiramente vamos criar o model Category(Categoria), suas migrations e todo o CRUD deste participante.

Podemos já começar criando nosso model com todo o aparato necessário de uma vez só, criar o model, a migration e o controller como recurso de uma vez só!

Execute na raiz do seu projeto o comando abaixo:

```
1    php  artisan make:model Category  -m  -c -r
```

Considerações:

-m: cria a migration deste model;

-c: cria o controller deste model;

-r: cria o controller como recurso para este model.

ManyToMany com Eloquent: Categorias e Posts

Veja o resultado:

```
blog: php artisan make:model Category -m -c -r
Model created successfully.
Created Migration: 2019_10_05_162546_create_categories_table
Controller created successfully.
blog:
```

Obs.: O comando irá criar o controller na pasta de Controllers normalmente, apenas mova este controller para a pasta Admin e corrija o namespace, de namespace App\Http\Controllers; para

namespace App\Http\Controllers\Admin; e adicione o import do controller base:

use App \Http\Controllers\Controller

ManyToMany com Eloquent: Categorias e Posts

Veja o controller depois das observações acima:

```
1   <?php
2
3   namespace App\Http\Controllers\Admin;
4
5   use   App\Category;
6   use   App\Http\Controllers\Controller;
7   use   Illuminate\Http\Request;
8
9   class  CategoryController extends Controller
10   {
11          /**
12           *  Display a  listing  of the resource.
13           *
14           *  @return \Illuminate\Http\Response
15           */
.....
```

ManyToMany com Eloquent: Categorias e Posts

Por enquanto vamos deixá-lo assim, vamos da uma atenção lá para nossa migration, acesse a pasta de migrations e abra o arquivo: *_create_categories_table.php.

Adicione o seguintes campos abaixo:

```
1  $table->string('name');
2  $table->string('description')->nullable();
3  $table->string('slug');
```

Como você pode ver o Laravel pegou o nome do nosso model e já preparou nossa migration para o plural, pela convenção já comentada aqui. Neste caso teremos a tabela categories com os campos: id; name; description; slug; created_at; updated_at;

ManyToMany com Eloquent: Categorias e Posts

Após as alterações que comentei anteriormente vamos criar uma nova migration, a migração para nossa tabela pivot ou intermediária para esta relação. Muitos para Muitos permite que uma postagem tenha várias categorias e uma categoria tenha várias postagens, pensando nisso precisamos de uma tabela intermediária para manter esta relação/ligação.

Execute em seu terminal o comando abaixo para criação da migração para esta tabela intermediária:

1- php artisan make:migration create_table_posts_categories --create=posts_categories

ManyToMany com Eloquent: Categorias e Posts

```
14 public function up()
15 {
16     Schema::create('posts_categories', function (Blueprint $table) {
17         $table->unsignedBigInteger('post_id’);
18         $table->unsignedBigInteger('category_id');
19
20          $table->foreign('post_id')->references('id')->on('posts');
21         $table->foreign('category_id')->references('id')->on('categories');
22     });
23 }
```

ManyToMany com Eloquent: Categorias e Posts

```
24
25 /**
26 * Reverse the migrations.
27 *
28 * @return void
29 */
30 public function down()
31 {
32         Schema::dropIfExists('posts_categories');
33 }
```

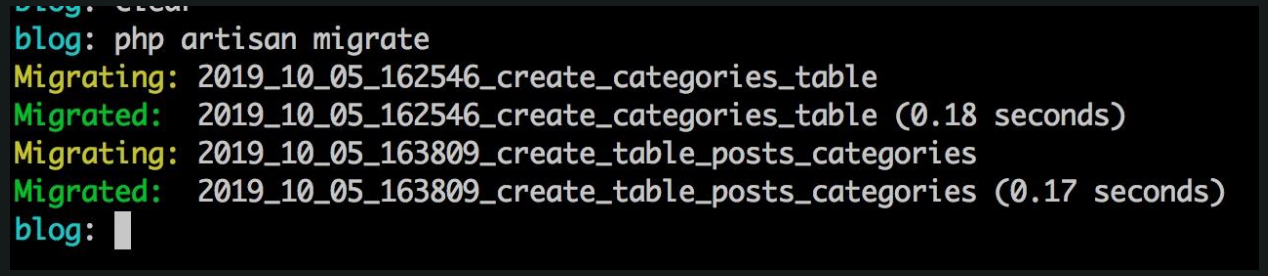
Do método up remova o bigIncrements e o timestamps e adicione o conteúdo como acima. Nesta tabela precisamos apenas das referências, um para o post e o outro para a categoria, além das chaves estrangeiras referenciando cada tabela para o id de cada uma.

ManyToMany com Eloquent: Categorias e Posts

Agora podemos executar estas migrações na base de dados, vamos ao terminal executar o comando abaixo:

```
1  php artisan migrate
```

Veja o resultado:



ManyToMany com Eloquent: Categorias e Posts

Com as migrations definidas, como podemos mapear nossa relação do ponto de vista do model? Existe no Eloquent o método para esta relação, o belongsToMany que estará nos dois models por conta da tabela intermediária.

Então, vamos adicionar o método abaixo dentro do nosso model Post:

```
1    public function categories()
2    {
3                return  $this->belongsToMany(Category::class,  'posts_categories');
4    }
```

ManyToMany com Eloquent: Categorias e Posts

E no model Category adicione o método abaixo:

```
1    public function posts()
2    {
3                return  $this->belongsToMany(Post::class,  'posts_categories');
4    }
```

ManyToMany com Eloquent: Categorias e Posts

O método belongsToMany nos permite o mapeamento na relação de muitos p/ muitos entre os dois models, como ambos apontam para a mesma tabela, o método de ambos os lados também é o mesmo.

Aqui vale um aprendizado e um adendo, o segundo parâmetro do método belongsToMany é o nome da tabela intermediária(posts_categories em nosso caso), mas por que eu preenchi este valor?

Eu preenchi valor deste parâmetro pois escolhi meu próprio nome de tabela intermediária que não é o mesmo que o Laravel iria tentar resolver de forma automática.

ManyToMany com Eloquent: Categorias e Posts

Mas como eu faço se eu quiser que o Laravel resolva automaticamente a tabela sem precisar do segundo parâmetro?!

O Laravel, caso não exista o segundo parâmetro como fiz, vai buscar a tabela no banco respeitando o nome da tabela intermediária no singular e em ordem alfabética, ou seja, se temos posts e categories ele supõe que a tabela intermediária seja category_post, nomes no singular separados pelo _ e em ordem alfabética c antes do p.

Antes de criarmos nosso post com suas categorias vamos criar rapidamente o CRUD de categorias com alguns detalhes a mais a serem adicionados também em nosso CRUD de posts melhorando assim nossa aplicação, Blog.

CRUD de Categorias

Vamos criar nosso CRUD com os controles de exceptions, bloco try...catch, e com redirecionamentos e mensagens de execução para o usuário. Primeiramente vamos instalar um pacote para exibirmos mensagens de execução de cada processo para nosso usuário, execute o comando abaixo na raiz do projeto:

```
1  composer require laracasts/flash
```

Este pacote nos permite de forma simples e rápida criarmos mensagens de execução para nossos usuários, mensagens que digam o que aconteceu no processo.

CRUD de Categorias

Por exemplo, após a inserção de uma postagem podemos jogar a mensagem via larcasts/f lash, pro caso de sucesso da inserção, da seguinte maneira:

```
1    flash('Postagem criada com sucesso!)->success();
```

Isso produzirá na tela um sessão com a mensagem informada e com as classes do Twitter Bootstrap: alert e alert-success, exibindo um alert verde pra nós. Como na imagem:

CRUD de Categorias

Agora vamos ao noso controller, irei colocar método por método do controller de Categorias, o CategoryController, para vermos os pontos importantes sobre cada método.

Vamos ao primeiro método, o método index e o construtor:

```
1  /**
2   * @var Category
3   */
4  private $category;
5
6  public function __construct(Category $category)
7  {
8          $this->category = $category;
9  }
10
11  /**
12   * Display a listing of the resource.
13   *
14   * @return \Illuminate\Http\Response
15   */
16  public function index()
17  {
18      $categories = $this->category->paginate(15);
19
20      return view('categories.index', compact('categories'));
21
22  }
```

CRUD de Categorias

O primeiro ponto importante é a utilização do nosso model como dependência do construtor da classe controller. Com isso teremos por meio do container de serviços do Laravel a instância do nosso model sempre que nosso controller for chamado. Isso simplifica muito as coisas e nos ajuda a mantermos nosso controller mais focado.

Falarei um pouco mais sobre o container de serviços mais a frente mas aqui já temos uma pequena pitada do que é possível fazer com esse cara!

Continuando para o método index, a diferença direta aqui é que ao invés de chamar o paginate assim Category::paginate(15) chamamos pelo atributo do controller que receberá a instância do nosso model.

CRUD de Categorias

O método a seguir é o create, que dispensa comentários até então, veja ele:

```
1  /**
2   * Show the form for creating a new resource.
3   *
4   * @return \Illuminate\Http\Response
5   */
6  public function create()
7  {
8          return view('categories.create');
9  }
```

CRUD de Categorias

Vamos continuando para o método store, onde de fato persistimos nossos dados, neste caso categoria. Veja ele e logo após os comentários:

```
1   /**
2    * Store a newly created resource in storage.
3    *
4    * @param \Illuminate\Http\Request $request
5    * @return \Illuminate\Http\Response
6    */
7   public function store(Request $request)
8   {
9               $data = $request->all();
10
11          try {
12                      $category = $this->category->create($data);
```

CRUD de Categorias

```
13
14                      flash('Categoria criada com sucesso!')->success();
15                      return redirect()->route('categories.index');
16
17          } catch (\Exception $e) {
18                  $message = 'Erro ao criar categoria!';
19
20                      if(env('APP_DEBUG')) {
21                              $message = $e->getMessage();
22                      }
23
24                      flash($message)->warning();
25                      return redirect()->back();
26          }
27  }
```

CRUD de Categorias

Aqui mais uma referência ao nosso atributo que contêm a instância do nosso model, usamos o mass assignment que já conhecemos. Agora temos mais alguns detalhes, primeiramente estamos controlando nossos processos com os blocks try...catch que nos permite um maior controle caso as Exceptions com erros sejam lançadas.

Usamos também, tanto para as mensagens de execução de sucesso e de erro, o pacote que instalamos anteriormente, o laracasts/flash, onde definimos as mensagens e o tipo delas: se sucesso, success(), se error o warning().

Logo após o set das mensagens de execução usamos o redirecionamento do Laravel por meio do helper redirect onde pro sucesso, redirecionamos o usuário para a tela principal de categorias dentro do admin, mas aqui por boa prática, redirecionamos pelo apelido da rota por isto chamo o método route que já usamos nas views!

```
1  flash('Categoria criada com sucesso!')->success();
2  return redirect()->route('categories.index');
```

CRUD de Categorias

Veja o bloco catch:

```
1    catch (\Exception $e) {
2            $message = 'Erro ao criar categoria!';
3
4            if(env('APP_DEBUG')) {
5                    $message = $e->getMessage();
6            }
7
8            flash($message)->warning();
9            return redirect()->back();
10   }
```

Aqui faço um controle para só exibirmos as mensagens reais dos erros, se lá em nosso arquivo .env a variável APP_DEBUG estiver como verdadeira(true), ou seja, só veremos as mensagens de erro reais em ambiente de desenvolvimento, em produção teremos mensagens padrões como: "Erro ao criar categoria".

CRUD de Categorias

O método show têm um pequeno detalhe que preciso comentar, veja:

```
1    /**
2     * Display the specified resource.
3     *
4     * @param  \App\Category  $category
5     * @return \Illuminate\Http\Response
6     */
7   public function show(Category $category)
8   {
9           return view('categories.edit', compact('category'));
10   }
```

CRUD de Categorias

Perceba de cara o parâmetro tipado do método show, tipado para aceitar apenas instâncias do nosso model Category. Um ponto importante aqui é que quando temos um parâmetro tipado assim, o Laravel automáticamente vai converter este parâmetro para um objeto populado para o id informado, por isso que não estou realizando um find ou findOrFail aqui, porque quando chegamos na execução deste método do controller já temos a instância do objeto Category populada com o id informado como comentei.

Então, só precisamos mandar para a view edit de categorias.

Quando nós geramos um controller como recurso, temos a geração também do método edit. Que serviria de fato para exibição do nosso formulário de edição.

CRUD de Categorias

Particularmente não teremos tela somente visualização dos dados de uma categoria em questão, para o qual o método show foi idealizado, por isso já exibo o formulário neste método e não no edit por escolha propria, geralmente o que adiciono no método edit é apenas fazer um redirecionamento, como fiz, veja abaixo:

```
1   /**
2    * Show the form for editing the specified resource.
3    *
4    * @param \App\Category $category
5    * @return \Illuminate\Http\Response
6    */
7   public function edit(Category $category)
8   {
9           return redirect()->route('categories.show', ['category' => $category->id]);
10  }
```

CRUD de Categorias

Agora caímos para o método update, realmente aqui não temos muito a comentar uma vez que os detalhes já são conhecidos e a forma de atualização também:

```
1 /**
2 * Update the specified resource in storage.
3 *
4 * @param \Illuminate\Http\Request $request
5 * @param \App\Category $category
6 * @return \Illuminate\Http\Response
7 */
8 public function update(Request $request, Category $category)
9 {
10          $data = $request->all();
11
12          try {
13                      $category->update($data);
```

CRUD de Categorias

```
14
15                  flash('Categoria atualizada com sucesso!')->success();
16                  return redirect()->route('categories.show', ['category' => $category->id]);
17
18          } catch (\Exception $e) {
19                  $message = 'Erro ao atualizar categoria!';
20
21                  if(env('APP_DEBUG')) {
22                          $message = $e->getMessage();
23                  }
24
25                  flash($message)->warning();
26                  return redirect()->back();
27          }
28 }
```

CRUD de Categorias

A remoção/delete também segue o mesmo pensamento, veja:

```
1 /**
2 * Remove the specified resource from storage.
3 *
4 * @param \App\Category $category
5 * @return \Illuminate\Http\Response
6 */
7 public function destroy(Category $category)
8 {
9         try {
10                 $category->delete();
11
```

CRUD de Categorias

```
12                  flash('Categoria removida com sucesso!')->success();
13                  return redirect()->route('categories.index');
14
15          } catch (\Exception $e) {
16                  $message = 'Erro ao remover categoria!';
17
18                  if(env('APP_DEBUG')) {
19                          $message = $e->getMessage();
20                  }
21
22                  flash($message)->warning();
23                  return redirect()->back();
24          }
25  }
```

CRUD de Categorias

Não esqueça dos parâmetros convertidos automaticamente para o model populado com base no id, resolvido dinamicamente pelo Laravel.

Ah, e não esqueça do atributo $fillable lá no model Category, para o mass assignment:

```
1   protected $fillable = [
2               'name',
3               'description',
4               'slug'
5   ];
```

CRUD de Categorias

Agora, vamos as views.

As views também não têm novidades, mas antes, crie a pasta categories dentro da pasta views e dentro os arquivos:

index.blade.php;

create.blade.php;

edit.blade.php.

CRUD de Categorias

index.blade.php:

```
1   @extends('layouts.app')
2
3   @section('content')
4           <div class="row">
5               <div class="col-sm-12">
6                   <a href="{{route('categories.create')}}" class="btn btn-success float-right">Criar
Categoria</a>
7
8                   <h2>Categorias Blog</h2>
9                   <div class="clearfix"></div>
10              </div>
11          </div>
```

CRUD de Categorias

```
12          <table class="table table-striped">
13              <thead>
14                  <tr>
15                      <th>#</th>
16                      <th>Nome</th>
17                      <th>Criado em</th>
18                      <th>Ações</th>
19                  </tr>
20              </thead>
21              <tbody>
22              @forelse($categories as $category)
23                  <tr>
24                      <td>{{$category->id}}</td>
25                      <td>{{$category->name}}</td>
```

CRUD de Categorias

```
26                          <td>{{date('d/m/Y H:i:s', strtotime($category->created_at))}}</td>
27                          <td>
28                              <div class="btn-group">
29                                  <a href="{{route('categories.show', ['category' => $category\
30  ->id])}}" class="btn btn-sm btn-primary">
31                                      Editar
32                                  </a>
33                                  <form action="{{route('categories.destroy', ['category' => $\
34  category->id])}}" method="post">
35                                      @csrf
36                                      @method('DELETE')
37                                      <button type="submit" class="btn btn-sm btn-danger">Remo\
38  ver</button>
```

CRUD de Categorias

```
39                                  </form>
40                              </div>
41                          </td>
42                      </tr>
43                  @empty
44                      <tr>
45                          <td colspan="4">Nada encontrado!</td>
46                      </tr>
47                  @endforelse
48                  </tbody>
49          </table>
50
51          {{$categories->links()}}
52  @endsection
```

CRUD de Categorias

create.blade.php:

```
1  @extends('layouts.app')
2
3  @section('content')
4          <form action="{{route('categories.store')}}" method="post">
5
6                  @csrf
7
8                  <div class="form-group">
9                          <label>Nome</label>
10                          <input type="text" name="name" class="form-control" value="{{old('name')\
11 }}">
12                 </div>
13
```

CRUD de Categorias

```
14              <div class="form-group">
15                  <label>Descrição</label>
16                  <input type="text" name="description" class="form-control" value="{{old(\
17  'description')}}">
18              </div>
19
20              <div class="form-group">
21                  <label>Slug</label>
22                  <input type="text" name="slug" class="form-control" value="{{old('slug')\
23  }}">
24              </div>
25
26              <button class="btn btn-lg btn-success">Criar Categoria</button>
27          </form>
28  @endsection
```

CRUD de Categorias

edit.blade.php:

```
1  @extends('layouts.app')
2
3  @section('content')
4        <form action="{{route('categories.update', ['category' => $category->id])}}" met\
5  hod="post">
6
7              @csrf
8              @method("PUT")
9
10              <div class="form-group">
11                    <label>Nome</label>
12                    <input type="text" name="name" class="form-control" value="{{$category->\
13   name}}">
14              </div>
```

CRUD de Categorias

```
15
16              <div class="form-group">
17                  <label>Descrição</label>
18                  <input type="text" name="description" class="form-control" value="{{$cat\
19  egory->description}}">
20              </div>
21
22              <div class="form-group">
23                  <label>Slug</label>
24                  <input type="text" name="slug" class="form-control" value="{{$category->\
25  slug}}">
26              </div>
27
28              <button class="btn btn-lg btn-success">Atualizar Categoria</button>
29          </form>
30  @endsection
```

CRUD de Categorias

Criada as views de categorias, precisamos adicionar lá no nosso layout (app.blade.php) a possibilidade de exibição de nossas mensagens do pacote laracasts/flash. Dentro do nosso bloco onde se encontra o yield('content'), adicione acima dele o include abaixo:

```
1  @include("flash::message")
```

Veja como ficou o trecho do yield:

```
1  ...
2  <div  class="container">
3          @include("flash::message")
4          @yield('content')
5  </div>
6  ...
```

CRUD de Categorias

Ah e não esqueça de link no menu do layout.blade.php o menu de categorias, veja o trecho adicionado logo após o link de posts:

```
1     <li  class="nav-item active">
2           <a  class="nav-link"  href="{{ route('categories.index') }}">Categorias</a>
3    </li>
```

Esta inclusão exibirá os alertas com as classes do twitter bootstrap conforme o método escolhido, na cor verde para a chamada do ->success() na cor amarela para a chamada ->warning().

Agora que nosso CRUD de categorias está pronto, faça alguns testes e crie algumas categorias para termos insumo no momento da inclusão das categorias para uma postagem.

Inserindo Muito para Muitos (Post x Category)

Para inserção de muitos para muitos, nós temos 3 métodos principais, o método attach, o detach e o sync que particularmente gosto de utilizar. Utilizarei aqui o sync mas darei uma rápida amostra do que os 2 outros fazem.

O método attach recebe um array com os ids a serem incluídos na referência, se estivermos salvando do ponto de vista de Post os ids serão de categorias e vice-versa. Veja o exemplo:

```
1    $post->categories()->attach([1,2]);
```

Inserindo Muito para Muitos (Post x Category)

Com o attach acima estou adicionando para uma postagem salva recentemente duas categorias, de id 1 e 2 respectivamente.

Se em uma tela de edição eu quiser remover a categoria 1 deste post, eu posso utilizar o detach, como abaixo:

```
1    $post->categories()->detach([1]);
```

Inserindo Muito para Muitos (Post x Category)

Agora como o sync funciona, ele é bem simples. Basicamente ele irá sincronizar as referências da ligação, por exemplo:

```
1    $post->categories()->sync([1,2]);
```

Inserindo Muito para Muitos (Post x Category)

Acima o sync irá inserir as duas referências caso elas não existam na ligação, dentro da tabela intermediária. Uma vez salva, se eu utilizar o sync novamente da maneira abaixo:

```
1    $post->categories()->sync([1]);
```

Ele irá remover os ids que não estiverem no array informado a ele, neste caso, a categoria de id 2 será removida da ligação.

Alterando Posts para Inserção N:N

Vamos adicionar a possibilidade de adição de categorias em nossas postagens. O primeiro passo é mandarmos para as views de criação e edição nossas categorias.

Veja como ficou os métodos agora create e update lá no PostController:

```
1    public function create()
2   {
3               $categories = \App\Category::all(['id', 'name']);
4
5               return view('posts.create', compact('categories'));
6   }
```

Alterando Posts para Inserção N:N

```
1   public function show(Post  $post)
2   {
3               $categories  =  \App\Category::all(['id',  'name']);
4
5               return  view('posts.edit',  compact('post',  'categories'));
6   }
```

Agora mandamos para cada view as categorias existentes em nossa base, neste caso passei um array para o método all que me retornará apenas o id e o nome de cada categoria, que é o que necessitamos pra este momento.

Alterando Posts para Inserção N:N

Agora precisamos exibir essas categorias em nossos formulários. Adicione o trecho abaixo em ambos os formulários:

```
1   <div class="form-group">
2       <label>Categorias</label>
3       <div class="row">
4           @foreach($categories as $c)
5               <div class="col-2 checkbox">
6                   <label>
7                       <input type="checkbox" name="categories[]" value="{{$c->id}}"> {\
8   {$c->name}}
9                   </label>
10              </div>
11          @endforeach
12      </div>
13  </div>
```

Alterando Posts para Inserção N:N

Este trecho exibirá diversas checkboxes com as categorias e seus ids como valores para cada checkbox.

Perceba um detalhe importante, o checkbox nos permite marcarmos quantos itens quisermos, neste caso como preciso mandar um array de ids(das categorias) para sincronizar, usei a notação no nome do campo com colchetes([]) desta maneira: categories[].

Assim enviaremos na requisição um array com os ids selecionados no campo categories, vindo dos checkboxes.

Alterando Posts para Inserção N:N

Uma vez feita esta alteração, vamos adicionar a possibilidade de save da relação, adicionando assim as categorias do post. Para isto, basta adicionarmos logo após a linha de criação e atualização dentro dos seus métodos (strore, update), a chamada do sync como vemos abaixo:

```
1    $post->categories()->sync($data['categories']);
```

Alterando Posts para Inserção N:N

Veja o método create e o update na íntegra, pós alteração:

```
1   public function store(Request  $request)
2   {
3               $data = $request->all();
4           try{
5               $data['is_active'] = true;
6
7               $user = User::find(1);
8
9               $post = $user->posts()->create($data); //Retornará o Post inserido, atribuímos \
10   ele a variável post para usarmos abaixo no sync
11
12              $post->categories()->sync($data['categories']); //aqui
13
14
```

Alterando Posts para Inserção N:N

```
15                  flash('Postagem inserida com sucesso!')->success();
16                   return redirect()->route('posts.index');
17
18          } catch (\Exception $e) {
19                  $message = 'Erro ao remover categoria!';
20
21                  if(env('APP_DEBUG')) {
22                          $message = $e->getMessage();
23                  }
24
25                  flash($message)->warning();
26                  return redirect()->back();
27          }
28 }
```

Alterando Posts para Inserção N:N

```
1   public function  update(Post $post,  Request $request)
2   {
3         $data =  $request->all();
4
5               try{
6                         $post->update($data);
7                         $post->categories()->sync($data['categories']); //aqui
8
9                     flash('Postagem atualizada  com sucesso!')->success();
10                     return redirect()->route('posts.show',  ['post' => $post->id]);
11
12          } catch  (\Exception  $e) {
13                     $message =  'Erro ao  remover categoria!';
14
15                     if(env('APP_DEBUG'))  {
16                               $message =  $e->getMessage();
17                     }
18
19                     flash($message)->warning();
20                     return redirect()->back();
21          }
22  }
```

Alterando Posts para Inserção N:N

Desta forma já adicionamos a possibilidade de inserção e edição das categorias para as postagens. Agora precisamos selecionar as categorias da postagem a ser edita, na tela de edição e sabermos quais as categorias do post quando entrarmos nesta tela.

Como temos uma relação de muitos para muitos, se acessarmos por exemplo:

```
1   $post->categories;
```

Teremos uma Collection, como vimos anteriormente, com as categorias existentes para a postagem em questão. Essa Collection nos permite verificarmos se determinado Objeto existe dentro da coleção, por meio do método contains. O que precisamos é marcar com checked cada input que bater com a categoria existente na relação, dentro do loop de categorias teremos cada categorias em questão, o que precisamos é passar para o método contains cada linha para que o Laravel verifique se aquela categoria em questão está na relação!

Alterando Posts para Inserção N:N

Vamos a alteração para entedermos melhor, na view de edição (edit.blade.php) de posts o que está asssim, dentro do loop de categorias:

```
1  <input type="checkbox" name="categories[]"  value="{{$c->id}}"> {{$c->name}}
```

Ficará assim:

```
1  <input  type="checkbox"  name="categories[]"  value="{{$c->id}}"
2
3  @if($post->categories->contains($c))  checked  @endif
4
5  > {{$c->name}}
```

Alterando Posts para Inserção N:N

Como estamos fazendo o laço nas categorias, em cada linha teremos o objeto Category populado com a categoria em questão. Com isso podemos verificar se na relação, do model, entre categorias e posts, existe aquele objeto disponivel usando o método contains.

Existindo, o contains retornará verdadeiro, e aí nós adicionamos a propriedade checked no input. Selecione uma categoria e atualize a postagem de sua escolha e verifique que agora a(s) categoria(s) pertecente a postagem estará ou estarão checkada(s).

Alterando Posts para Inserção N:N

Na tela de inserção só precisamos selecionar as categorias que desejarmos que elas já serão salvas pelo sync e isso já fizemos.

Faça seus testes!