

DATABASE: MIGRATIONS



Migrations

As migrations ou migrações são ferramentas que nos auxiliam no versionamento da estrutura de nossas bases de dados. Funciona basicamente como uma documentação da linha histórica do crescimento da estrutura do banco, criação das tabelas, ligações e etc.

Vale ressaltar que o conceito migrações não é algo do Laravel mas que ele se utiliza para trazer mais essa opção e facilidade para quem está desenvolvendo. Criar tabelas e seus aparatos se torna mais fácil quando realizamos isso do ponto de vista de código que após um comando é traduzido para o banco em questão.

As migrations dentro do nosso projeto podem ser encontradas dentro da pasta `database/migrations`. Nesta pasta você já vai encontrar alguns arquivos de migração iniciais, como a migração para a tabela de usuários e a para a tabela de reset de senhas, e também temos uma migration a mais disponível, a da tabela de jobs falhos do sistema de filas do framework.

Migrations

Vamos dar uma olhada na migration da tabela de usuários e entendermos como é formado um arquivo de migração.

Veja abaixo o arquivo 2014_10_12_000000_create_users_table.php:

```
1  <?php
2
3  use Illuminate\Database\Migrations\Migration;
4  use Illuminate\Database\Schema\Blueprint;
5  use Illuminate\Support\Facades\Schema;
6
7  class CreateUsersTable extends Migration
8  {
9      /**
10       * Run the migrations.
11       *
12       * @return void
13       */
```

Migrations

```
14     public function up()
15     {
16         Schema::create('users', function (Blueprint $table) {
17             $table->bigIncrements('id');
18             $table->string('name');
19             $table->string('email')->unique();
20             $table->timestamp('email_verified_at')->nullable();
21             $table->string('password');
22             $table->rememberToken();
23             $table->timestamps();
24         });
25     }
26
27     /**
28     * Reverse the migrations.
29     *
30     * @return void
31     */
32     public function down()
33     {
34         Schema::dropIfExists('users');
35     }
36 }
```

Migrations

Perceba acima que a classe de migração para a tabela users, inicialmente estende de uma classe base chamada `Migration` e traz a definição de dois métodos, o método `up` e o método `down`.

O método `up` será executado quando pegarmos essa migração e executarmos em nosso banco de dados.

E o método `down` contém a definição do inverso do método `up`, no método `down` nós definimos a remoção do que foi aplicado no método `up`, isso nos permite voltarmos para estados anteriores pré-execução do último lote de migrações.

Migrations

Vamos dar uma atenção ao método up:

```
1  public function up()
2  {
3      Schema::create('users', function (Blueprint $table) {
4          $table->bigIncrements('id');
5          $table->string('name');
6          $table->string('email')->unique();
7          $table->timestamp('email_verified_at')->nullable();
8          $table->string('password');
9          $table->rememberToken();
10         $table->timestamps();
11     });
12 }
```

Migrations

Temos uma classe base responsável pela definição dos schemas (`Schema`) da base de dados e é com ela que criamos nossa tabela por meio do método `create` informando como primeiro parâmetro o nome da tabela e o segundo parâmetro será um `callback` (ou função anônima) onde definiremos os campos e estrutura da tabela em questão, aqui a tabela `users`.

Para definir os campos de nossa tabela precisamos do objeto `Blueprint` que nos permite criarmos os campos e tipos por meio de seus métodos, por isso tipamos o parâmetro `$table`, do `callback`, como `Blueprint`.

O `Blueprint` contém métodos para tudo o que é necessário de manipulação de nosso banco, geração e remoção de colunas, os mais variados tipos de dados para as colunas em questão, definição de chaves estrangeiras, criação de índices e muito mais.

Migrations

Podemos criar nossas chaves primárias e auto-incrementáveis utilizando o método `bigIncrements`, como temos na linha abaixo:

```
1 $table->bigIncrements('id');
```

Podemos definir campos do tipo string (Varchar), como ocorre abaixo:

```
1 $table->string('name');
```

```
2 $table->string('email')->unique();
```

Acima temos a definição de um campo Varchar e por default recebe 255 caracteres, caso queira especificar um tamanho para o campo, basta preencher o segundo parâmetro com o valor inteiro, correspondente ao tamanho do campo desejado.

Migrations

Perceba também que para o e-mail atribuímos uma definição na coluna, assinalando este campo como `unique` ou seja evitando a duplicação de linhas com o mesmo email, tornando assim, a partir da base, o usuário único por email.

Podemos definir campos de data e hora por meio do método `timestamp`, veja abaixo:

```
1 $table->timestamp('email_verified_at')->nullable();
```

O campo `email_verified_at` que também será nulo, definido pelo método `nullable`. Temos também a definição de mais um campo tipo `VARCHAR` para a coluna `password`:

```
1 $table->string('password');
```

Migrations

E por fim temos a chamada do método `rememberToken()` e também do `timestamps()`, o que são esses métodos ou que eles fazem?

- 1 `$table->rememberToken();`
- 2 `$table->timestamps();`

O método `rememberToken` irá criar uma coluna chamada de `remember_token`, `varchar` com tamanho 100 e aceitando o valor nulo.

Já o método `timestamps` irá criar dois campos do tipo `timestamp`, um chamado de `created_at` e outro chamada de `updated_at` ambos representando a data de criação e atualização do dado em questão, o mais interessante é que o Laravel controla os valores destes dois campos automaticamente via Models.

Migrations

Se o método `up` define a criação da tabela de `users`(usuários) o `down` defini a remoção desta tabela. Veja sua definição:

```
1 public function down()  
2 {  
3     Schema::dropIfExists('users');  
4 }
```

A exclusão ocorre por meio do método `dropIfExists` do objeto `Schema` onde informamos a tabela que queremos remover e se ela existir na base, será removida. Isso simplifica bastante pois poderemos voltar um passo anterior se tivermos executado esta migração em algum momento.

Agora como pegamos esse código Orientado a Objetos e jogamos para uma base relacional? É o que vamos ver a seguir.

Executando primeira migração

Se já temos estas migrações disponíveis vamos executá-las em nossa base, epa, espera aí, não temos base ou banco de dados!??

É claro que para executarmos as migrações precisamos está conectados com nossa base de dados em questão.

Para isso, na gerenciador de sua escolha crie um banco de dados chamado blog e adicione as configurações de acesso em seu arquivo `.env` na raiz do projeto.

Basta modificar os parâmetros com os valores de sua conexão:

Executando primeira migração

```
1 ;Parâmetros dentro do arquivo .env
2
3 DB_CONNECTION=mysql
4 DB_HOST=127.0.0.1
5 DB_PORT=3306
6 DB_DATABASE=blog
7 DB_USERNAME=root
8 DB_PASSWORD=
```

Caso você esteja utilizando outro banco de dados que não mysql será necessário alterar o drive na variável DB_CONNECTION.

Executando primeira migração

Para testarmos se nossa conexão ocorreu com sucesso, no cenário em que estamos no momento, vamos ao nosso terminal e na raiz do projeto vamos executar o comando abaixo:

```
1  php artisan migrate:install
```

Se tudo ocorrer bem como mostra o resultado abaixo, sua conexão está correta e setada com sucesso:

```
blog: php artisan migrate:install  
Migration table created successfully.  
blog: █
```

Executando primeira migração

Agora o que este comando que executamos acabou de fazer? Bem simples, ele apenas criou a tabela de controle de migrações executadas na base. Se você acessar sua base verá que existe lá uma tabela chamada de migrations que registra o nome da migração executada e o lote em que esta migração foi executada.

Ainda não executamos a execução das migrações existentes no projeto até o momento, então, como realizamos esta execução?

Para rodarmos e executarmos os arquivos de migração existentes é necessário executar o comando abaixo:

```
1  php artisan migrate
```

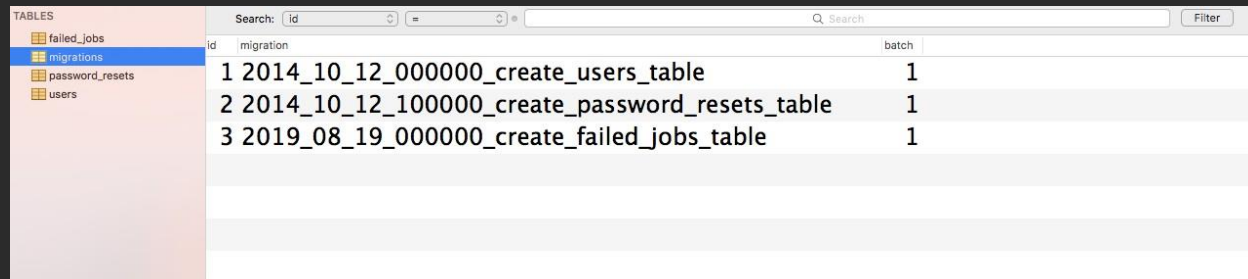
Executando primeira migração

Resultado:

```
blog: php artisan migrate
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (0.14 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (0.08 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (0.07 seconds)
blog: █
```


Executando primeira migração

Veja que agora cada arquivo de migração existente foi executado em nossa base e já estão registrados na tabela migrations com o lote (coluna batch) como 1, primeiro lote de execução:



id	migration	batch
1	2014_10_12_000000_create_users_table	1
2	2014_10_12_100000_create_password_resets_table	1
3	2019_08_19_000000_create_failed_jobs_table	1

E é claro as tabelas também foram criadas e estão em nosso banco agora.

Mas como posso criar minhas migrações para tabelas do meu projeto também? Certo! Vamos fazer isso agora!

Criando Nossas Migrações

Primeiro passo é irmos ao nosso terminal e executarmos o comando para geração de nosso arquivo de migração:

```
1  php artisan make:migration create_table_posts --create=posts
```

```
blog: php artisan make:migration create_table_post --create=posts
Created Migration: 2019_09_23_095103_create_table_post
blog: █
```

O comando acima criará nosso primeiro arquivo de migração dentro da pasta de migrações, chamado de 2019_09_23_095103_create_table_post, o nome do arquivo de migração respeita a data de criação mais o timestamp e o nome escolhido, em nosso caso: create_table_posts. Essa definição da data e timestamp permite o Laravel organizar a ordem das migrações.

Criando Nossas Migrações

O parâmetro `--create=posts` adicionará para nós o código da classe `Schema` e o método `create` como podemos ver no conteúdo do arquivo gerado abaixo:

```
1  <?php
2
3  use Illuminate\Support\Facades\Schema;
4  use Illuminate\Database\Schema\Blueprint;
5  use Illuminate\Database\Migrations\Migration;
6
7  class CreateTablePost extends Migration
8  {
9      /**
10       * Run the migrations.
11       *
12       * @return void
13       */
14     public function up()
```

Criando Nossas Migrações

```
15     {
16         Schema::create('posts', function (Blueprint $table) {
17             $table->id();
18             $table->timestamps();
19         });
20     }
21
22     /**
23      * Reverse the migrations.
24      *
25      * @return void
26      */
27     public function down()
28     {
29         Schema::dropIfExists('posts');
30     }
31 }
```

Criando Nossas Migrações

Por termos utilizado o parâmetro `--create` além das definições dos métodos `up` e `down` foram adicionados seus conteúdos com alguns detalhes de campos iniciais no `up`, a definição do campo de auto incremento e a definição dos campos de criação e atualização dos registros.

O `down` já trouxe o movimento contrário, neste caso a remoção da tabela `posts`.

Criando Nossas Migrações

Agora vamos as nossas adições, a adição dos nossos campos para nossa tabela de posts.

Criaremos os seguintes campos:

```
title: string 255;
```

```
description: string 255;
```

```
content: text;
```

```
slug: string 255;
```

```
is_active: boolean;
```

Criando Nossas Migrações

Agora como podemos representar estes campos acima dentro do nosso arquivo? Vamos lá, após a definição do `bigIncrements` defina o código abaixo:

```
1      $table->string('title');  
2      $table->string('description');  
3      $table->text('content');  
4      $table->string('slug');  
5      $table->boolean('is_active');
```

Criando Nossas Migrações

Simple, acima realizamos as definições dos nossos campos. Agora estamos aptos a executar esta migração em nosso banco de dados, para isso vamos ao nosso terminal executar o comando que já conhecemos.

Veja abaixo:

```
1  php artisan migrate
```

Ao executarmos o comando acima novamente, o Laravel só executará as migrações que ainda não foram executadas. Em nosso caso, e no momento, a única que não foi executada foi a que geramos acima. Por isso teremos o resultado abaixo:

```
blog: php artisan migrate
Migrating: 2019_09_23_095103_create_table_post
Migrated: 2019_09_23_095103_create_table_post (0.06 seconds)
blog: █
```


Relacionamentos via Migrations

Agora que temos nossa tabela posts criada, vamos mapear nosso primeiro relacionamento entre posts e usuários caracterizando assim a relação de posts e autor.

O relacionamento aqui que irei mapear será de 1:N (Um para Muitos) onde 1 autor(usuário) poderá ter N(vários) posts e 1 post poderá ter ou pertencer a apenas um 1 autor/usuário.

Relacionamentos via Migrations

Como estamos definindo nossa base via migrations vamos aprender aqui a definir este relacionamento e de quebra saber como alterar uma tabela já existente por meio de migrations, neste caso alterar posts para adicionarmos a referência para user e ainda criar nossa chave estrangeira.

Para isso execute o comando abaixo:

```
1  php artisan make:migration alter_table_posts_add_column_user_id --table=posts
```

Perceba que o comando continua o mesmo, eu apenas criei outro arquivo e aqui temos a chamada de um novo parâmetro, quando queremos criar uma tabela e suas definições nós utilizamos o método, do objeto Schema, chamado de create como vimos no trecho passado.

Relacionamentos via Migrations

Agora que eu quero alterar uma tabela já existente eu preciso utilizar o método table por isso chamei o parâmetro --table e o nome da tabela posts, este parâmetro vai gerar o conteúdo do arquivo conforme podemos ver abaixo:

```
....
7  class AlterTablePostsAddColumnUserId extends Migration
8  {
9      /**
10       * Run the migrations.
11       *
12       * @return void
13       */
14     public function up()
15     {
16         Schema::table('posts', function (Blueprint $table) {
17             //
18         });
19     }
```

Relacionamentos via Migrations

```
20
21  /**
22   * Reverse the migrations.
23   *
24   * @return void
25   */
26  public function down()
27  {
28      Schema::table('posts', function (Blueprint $table) {
29          //
30      });
31  }
32 }
```

Relacionamentos via Migrations

Ele já traz os conteúdos do método `up` e do método `down` chamando o método `table`, o que nos resta é só adicionarmos nossas modificações para a tabela desejada.

O nome da tabela vai no primeiro parâmetro do método `table` e o segundo, mesmo pensamento do `create`, vai as definições para a tabela na função anônima ou callback.

Como vamos adicionar a referência para `user`, irei criar um campo `user_id` respeitando as mesmas configurações do campo `id` na tabela de `users` para que nossa ligação na chave estrangeira possa ser satisfeita.

Relacionamentos via Migrations

Precisei criar um campo do tipo `bigInteger` e `unsigned` para isso adicione no método `up` a definição abaixo:

```
1 $table->unsignedBigInteger('user_id')->after('id');
```

A definição já satisfaz o tipo e configurações esperadas para esta referência, e aqui temos mais um conhecimento.

Como estou alterando a tabela `posts` pós criação, posso informar em que posição quero que o campo novo seja adicionado, neste caso, depois do campo `id` de `posts` e isso é realizado pelo método `after` (em pt depois).

Relacionamentos via Migrations

Agora, ainda no método `up`, precisamos criar nossa chave estrangeira para a coluna `user_id` referenciando a coluna `id` na tabela `users`. Adicione o trecho abaixo, após a definição anterior.

```
1 $table->foreign('user_id')->references('id')->on('users');
```

Acima assinala a criação da chave estrangeira com o método `foreign` informando o nome da coluna, no caso `user_id` e informei o campo e a tabela remota para esta referência, neste caso o campo `id` por meio do método `references` e a tabela `users` por meio do método `on`.

Ou seja se fossemos traduzir: crie uma chave estrangeira para o campo `user_id` de `posts` que faz referência para o `id` lá na tabela `users`.

Relacionamentos via Migrations

O nosso método `down`, por ser o reverso do `up`, conterá a remoção da chave estrangeira e também da coluna `user_id`. Então, adicione as duas definições abaixo:

```
1 $table->dropForeign('posts_user_id_foreign');
```

```
2 $table->dropColumn('user_id');
```

Quando o Laravel, por meio das migrations, cria a chave estrangeira ela recebe o nome respeitando a estrutura abaixo:

```
1 tabela_coluna_foreign
```

Por isso no `down` estou apagando a chave estrangeira, por meio do método `dropForeign` e informando a string `posts_user_id_foreign` e logo após, por meio do método `dropColumn` removo a coluna `user_id` completando assim o reverso do que é executado no método `up`.

Relacionamentos via Migrations

Agora para que isso seja executado em nossa base, basta irmos ao nosso terminal e executarmos o comando abaixo:

```
1 php artisan migrate
```

Comando que já conhecemos e que vai executar a última migração criada, porque ainda não temos registro dela, na tabela de migrations. Veja o resultado:

```
blog: php artisan migrate
Migrating: 2019_09_23_135618_alter_table_posts_add_column_user_id
Migrated: 2019_09_23_135618_alter_table_posts_add_column_user_id (0.13 seconds)
blog: █
```

Agora já temos nossa tabela para salvar as postagens e também nossa referência para criação do autor da postagem em questão. Sobre migrations, por enquanto, ficaremos por aqui.

É claro que ainda veremos bastante elas aqui no livro mas por hora já tivemos um excelente conhecimento a respeito de sua utilização. Agora vamos conhecer mais dois caras que nos auxiliam no momento do nosso desenvolvimento e banco de dados.

ELOQUENT

Eloquent, trabalhando com Models

Continuando nosso trabalho com a camada de dados e persistência, vamos subir o nível conhecendo a camada dos models e como podemos trabalhar buscas, inserções, atualizações, remoções e até mesmo os relacionamentos da base relacional no nível dos objetos, que são nossos models.

Vamos começar primeiro pelas queries e ir crescendo nosso conhecimento no decorrer das aulas.

Para isto vamos usar nosso controller PostsController que se encontra dentro da pasta Admin em controllers.

Os Models!

No Laravel os **models** são a representação, do ponto de vista de objetos, das tabelas do nosso banco de dados. Representação essa, pensando em uma entidade que represente todos os dados da tabela em questão.

Por exemplo, por convenção do framework, se eu tenho uma tabela chamada **posts** na base, a representação em model desta tabela será uma classe chamada de **Post**. Se eu tenho uma tabela **users** sua representação via model será uma classe chamada de **User**.

Quando nós temos entidades/models no singular o Laravel automaticamente tentará, por convenção, resolver sua tabela no plural por ter o pensamento, na base, de uma coleção de dados.

Os Models!

Por exemplo:

```
1 return \App\Post::all();
```

O resultado do método acima, se fossemos pensar em uma query na base seria uma sql como esta:

```
1 select * from posts posts
```

Onde o Laravel pegará automaticamente o nome do seu model e tentará resolver ele no plural na execução da query, por exemplo model Post tabela posts.

Os Models!

Agora vamos conhecer o conteúdo do model Post que geramos no final do capítulo passado. Veja abaixo:

```
1  <?php
2
3  namespace App;
4
5  use Illuminate\Database\Eloquent\Model;
6
7  class Post extends Model
8  {
9      //
10 }
```

Veja nosso model acima, somente com sua definição, bem seca inclusive, já podemos realizar diversos trabalhos e operações em cima de nossa tabela posts associada ao model Post.

Os Models!

Se, por ventura, você quiser utilizar um nome de tabela de sua escolha e não quiser que o Laravel resolva o nome dela, você pode subscrever o atributo dentro do seu model como mostrado no conteúdo abaixo:

```
1  <?php
2
3  namespace App;
4
5  use Illuminate\Database\Eloquent\Model;
6
7  class Post extends Model
8  {
9      protected $table = 'nome_da_sua_tabela';
10 }
```

Eloquent?

O Eloquent é o ORM padrão do Laravel, é a camada via objetos para manipulação dos dados de seu banco. O ORM é a camada que traduz sua estrutura de objetos, Models, para a camada relacional da sua base.

Por exemplo, veja um exemplo de inserção de uma postagem utilizando o Eloquent:

```
1 //O código poderia estar em um método do controller
2
3 $post = new Post();
4 $post->title = 'Post salvo com eloquent e active record';
5 $post->description = 'Descrição post';
6 $post->content = 'Conteúdo do post';
7 $post->slug = 'post-salvo-com-eloquent-e-active-record';
8 $post->is_active = true;
9 $post->user_id = 1;
10
11 $post->save(); //Aqui a inserção do post com o conteúdo acima é inserida na tabela.
```


Eloquent na prática

Lembra que já temos um controller para utilização e criação de um CRUD para posts?

Por meio deste CRUD vamos focar nos pontos mais cruciais para você conhecer o trabalho do Eloquent em nossas aplicações. Nosso controller encontra-se na pasta dos controllers, dentro da pasta Admin e o controller PostsController.

Primeiramente vamos definir nosso método index. Trabalhando do ponto de vista dos models eu consigo realizar operações de busca dos dados em minhas tabelas, como mostrado anteriormente posso buscar todos os registros do banco de dados por meio do método all.

Trecho abaixo:

```
1 return \App\Post::all();
```

Eloquent na prática

Ou posso retornar os dados paginados, para exibição em uma tabela html na view. Então vamos começar a implementar o método index no controller PostController que está na pasta Admin.

Veja o conteúdo dele abaixo:

```
1 public function index()
2 {
3     $posts = Post::paginate(15);
4
5     dd($posts); //depois vamos mandar para a view
6 }
```

Veja o código acima, por enquanto ainda não vamos utilizar a view, retornaremos a ela e os pontos do Blade no próximo capítulo. Voltando ao código acima, perceba que chamei o Post::paginate informando que quero 15 postagens, neste caso, os dados vão vir do banco paginados e teremos todas as postagens por cada tela da paginação.

Não esqueça de importar a classe post em seu controller:

```
1 use App\Post;
```

Eloquent na prática

Agora, vamos lá no arquivo de rotas e realizar uma pequena alteração no que já havíamos feito, para o conjunto de rotas de post, então, o que está assim:

```
1  Route::prefix('admin')->namespace('Admin')->group(function(){
2
3      Route::prefix('posts')->name('posts.')->group(function(){
4          Route::get('/create', 'PostController@create')->name('create');
5          Route::post('/store', 'PostController@store')->name('store');
6      });
7
8  });
```

Eloquent na prática

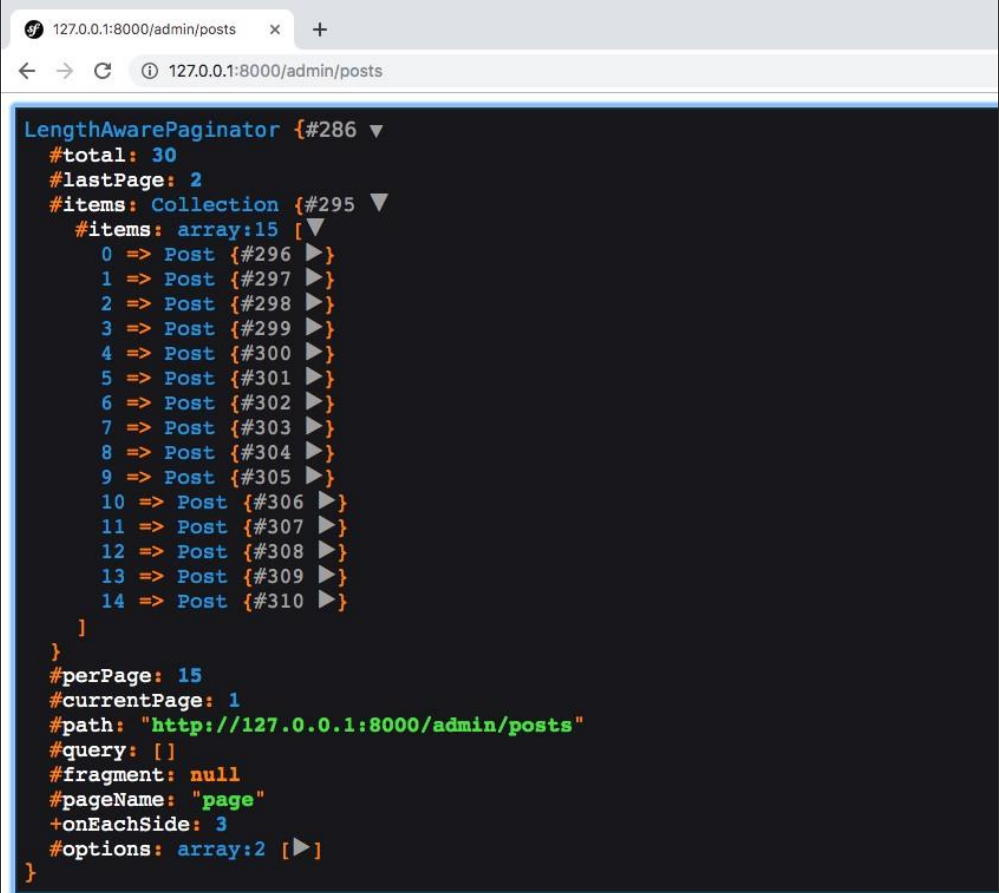
Vamos atualizar para a chamada do controller como recurso, ficando como vemos abaixo:

```
1  Route::prefix('admin')->namespace('Admin')->group(function(){  
2  
3      Route::resource('posts', 'PostController');  
4  
5  });
```

Eloquent na prática

Perceba que agora simplificamos mais ainda nossas rotas dentro do grupo para o admin. Quando trabalhamos com request e os métodos create e store que já existem no controller tomei o cuidado de já deixá-los dentro do pensamento para o roteamento com o método resource e controllers como recurso.

Se você for ao seu browser agora e acessar <http://aula.test/admin/posts>



```
LengthAwarePaginator {#286 ▾
  #total: 30
  #lastPage: 2
  #items: Collection {#295 ▾
    #items: array:15 [▾
      0 => Post {#296 ▹}
      1 => Post {#297 ▹}
      2 => Post {#298 ▹}
      3 => Post {#299 ▹}
      4 => Post {#300 ▹}
      5 => Post {#301 ▹}
      6 => Post {#302 ▹}
      7 => Post {#303 ▹}
      8 => Post {#304 ▹}
      9 => Post {#305 ▹}
      10 => Post {#306 ▹}
      11 => Post {#307 ▹}
      12 => Post {#308 ▹}
      13 => Post {#309 ▹}
      14 => Post {#310 ▹}
    ]
  }
  #perPage: 15
  #currentPage: 1
  #path: "http://127.0.0.1:8000/admin/posts"
  #query: []
  #fragment: null
  #pageName: "page"
  +onEachSide: 3
  #options: array:2 [▹]
}
```

Eloquent na prática

Veja que expandi o atributo items e seu nível a dentro, onde temos a coleção de dados retornada.

Veja que tivemos todos os itens retornados.

Para navegar entre as páginas é bem simples, basta nós informarmos na url o parâmetro page=2 (como querystring) por exemplo.

Quando formos trabalhar com as views e o blade veremos uma forma simples de criar a paginação no frontend, simplificando esta navegação.

Buscando apenas um post

Para buscarmos dados podemos trabalhar com diversos métodos. Por exemplo, se você quiser buscar uma postagem pelo id dela, você pode usar o método find. Veja:

```
1 Post::find(1);
```

Ou ainda, buscando pelo id, você pode usar o método findOrFail que caso não encontre o dado em questão irá lançar uma Exception. Podemos usar desta maneira:

```
1 Post::findOrFail(1);
```

Buscando apenas um post

Vamos criar agora o método em nosso controller PostController para recuperação de uma postagem, para nossa tela de edição. Veja o conteúdo do método show e já adicione ele em seu controler:

```
1 public function show($id)
2 {
3     $post = Post::findOrFail($id);
4
5     dd($post); //em breve mandaremos pra view
6 }
```

Usarei o findOrFail, para mais a frente tratarmos melhor estas exceptions com blocos try e catch. Se você acessar a postagem de id 1 em seu browser pelo link <http://aula.test/admin/posts/1> você terá o resultado com a postagem de id 1 retornada.

Buscando apenas um post

Abaixo eu destaco só o atributo original que traz o dados retornados do método findOrFail, a postagem em questão:

```
#original: array:9 [▼
  "id" => 1
  "user_id" => 7
  "title" => "quis perspiciatis tenetur quis"
  "description" => "Voluptatem modi iusto et officiis aperiam."
  "content" => ""
    Inventore expedita incidunt consequatur ex omnis sequi. Quis maiores numquam provident voluptatem qui voluptas unde
    it ipsum ut quas impedit.\n
    \n
    Assumenda esse ut quia vero necessitatibus. Voluptates deleniti labore natus accusantium non. Ullam distinctio ut
    obis ea qui. Est et sit et m ▶
    \n
    Sit cupiditate ab dolor corporis. Nobis impedit mollitia et omnis consequuntur. Et et sed eum distinctio tempora n
    \n
    Velit dolorem voluptatum cum vitae harum ad in. Voluptatem aut voluptate voluptates. Eaque voluptatem sed magnam p
    \n
    Quia veritatis non et non. Quaerat laborum impedit aut possimus non aut rerum. Labore vel ullam eos fuga.\n
    \n
    Nobis numquam numquam consequatur suscipit sit quia sint. Magnam ipsa et harum perspiciatis blanditiis sit consect
    illum et culpa eum sequi vel ▶
    \n
    Est adipisci ducimus iusto dicta nobis magni deleniti. Est facilis quia quidem. Nemo cum iure aut omnis debitis bl
    \n
    Quo ut vero tempora cum recusandae voluptatum. Ut accusantium expedita corrupti animi blanditiis minima necessitat
    si assumenda consequatur eos ▶
    \n
    Consequatur rem similique et repudiandae vel. Dolorum beatae praesentium rerum non corrupti repellat atque. Sunt q
    uatur tenetur dolore aut vol ▶
    ""
  "slug" => "quo-accusamus-fuga-placeat-saepe-non-esse"
  "is_active" => 0
  "created_at" => "2019-09-23 20:34:18"
  "updated_at" => "2019-09-23 20:34:18"
```

Buscando apenas um post

Veja na íntegra:

```
1  <?php
2
3  namespace App\Http\Controllers\Admin;
4
5  use Illuminate\Http\Request;
6  use App\Http\Controllers\Controller;
7  use App\Post;
8
9  class PostController extends Controller
10 {
11     public function index()
12     {
13         $posts = Post::paginate(15);
14
15         dd($posts); //no próximo capítulo vamos mandar para view...
16     }
17
18     public function show($id)
19     {
```

Buscando apenas um post

```
20         $post = Post::findOrFail($id);
21
22         dd($post);
23     }
24
25     public function create()
26     {
27         return view('posts.create');
28     }
29
30     public function store(Request $request)
31     {
32         if($request->hasAny(['title', 'content', 'slug'])) {
33             var_dump($request->except(['title']));
34         }
35
36         return back()->withInput();
37     }
38 }
```

Agora vamos ao método store pois nele vamos trabalhar a inserção de dados propriamente dita!

Inserindo dados com Eloquent

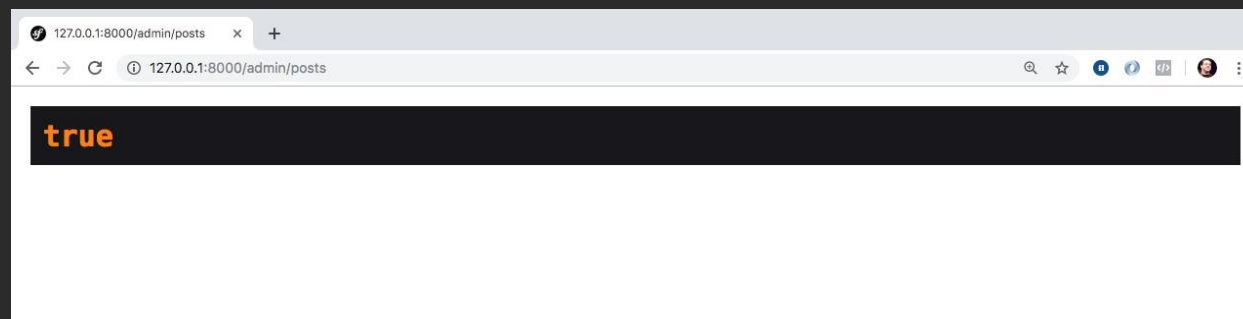
Como estamos tratando aqui de criação, vou mostrar a título de conhecimento o salvar dos dados usando [Active Record](#) dentro de nosso método, veja como ficaria. Altere o conteúdo do método store, já existente, para o conteúdo abaixo:

```
1  public function store(Request $request)
2  {
3      $data = $request->all();
4
5      $post = new Post();
6
7      $post->title      = $data['title'];
8      $post->description = $data['description'];
9      $post->content     = $data['content'];
10     $post->slug        = $data['slug'];
11     $post->is_active   = true;
12     $post->user_id     = 1;
13
14     dd($post->save()); //veja o resultado no browser
15 }
```

Inserindo dados com Eloquent

Veja que agora pego os dados de campo da request, vindas do formulário e repasso para cada atributo e na chamada do método save nós criamos este registro na base.

Para testarmos vamos ao nosso formulário no link <http://aula.test/admin/posts/create> e enviar uma informação de lá. Veja o resultado na imagem abaixo:



Inserindo dados com Eloquent

Perceba que o resultado do método `save` foi o valor booleano `true`, confirmando assim a criação do registro em nossa base. Se você quiser atualizar um registro usando Active Record, basta, ao invés de instanciar um `model post`, passar o resultado de um `find` por exemplo:

```
1  $post = Post::find(1);
2
3  $post->title      = $data['title'];
4  $post->description = $data['description'];
5  $post->content     = $data['content'];
6  $post->slug        = $data['slug'];
7  $post->is_active   = true;
8  $post->user_id     = 1;
9
10 dd($post->save());
```

Inserindo dados com Eloquent

Como temos a referência na variável `$post`, agora de um dado vindo da base, ao chamarmos o método `save` o Eloquent irá atualizar este registro ao invés de criar um novo.

Este trecho foi um rápido demonstrativo do `active record` no Eloquent, quero te mostrar uma técnica mais direta e que é mais utilizada hoje em dia dentro do Laravel, via Eloquent.

Esta técnica é o que chamamos de `Mass Assignment` ou `Atribuição em Massa`.

Vamos conhecer esta técnica.

Mass Assignment

Mass Assignment ou Atribuição em Massa é uma forma de inserir ou atualizar os dados por meio de uma única chamada e de uma vez só.

Por exemplo eu poderia passar todo o array vindo da request e já salvar isso direto no banco por meio de um método do Eloquent, o método create.

Então vamos a alteração, mais uma vez do nosso método store, que está assim:

Mass Assignment

```
1  public function store(Request $request)
2  {
3      $data = $request->all();
4
5      $post = new Post();
6
7      $post->title      = $data['title'];
8      $post->description = $data['description'];
9      $post->content     = $data['content'];
10     $post->slug        = $data['slug'];
11     $post->is_active   = true;
12     $post->user_id     = 1;
13
14     dd($post->save()); //veja o resultado no browser
15 }
```

Mass Assignment

Agora, passará a ficar assim:

```
1 public function store(Request $request)
2 {
3     $data = $request->all();
4     $data['user_id'] = 1;
5     $data['is_active'] = true;
6
7     dd(Post::create($data));
8 }
```

Mass Assignment

Perceba a redução acima, ao invés de chamarmos os atributos chamamos apenas o método create passando para ele nosso array recuperado da request. Atente só a um detalhe, o array passado pro método create deve respeitar, em suas chaves, os nomes das colunas da tabela em questão.

Obs.: Perceba que adicionei na mão a chave e valor para o user_id e a do is_active. Vamos trabalhar o user_id diretamente na parte de relação entre o Autor e a Postagem, como já mapeamos no banco. O is_active pode ir pro formulário com um select com as opções ativo ou inativo esta alteração faremos quando formos para o blade nas próximas aulas.

Mass Assignment

Se você for ao browser e testar isso enviando os dados do formulário, perceberá que teremos uma exception sobre a adição de um campo na propriedade \$fillable do model, aqui entra um ponto importante.

Antes de comentar o erro, você pode está se perguntando: Esta atribuição em massa não pode ser problemática, já que ela pelo visto aceita tudo?!

Veja a exception lançada:

The screenshot shows a web browser window with the URL `127.0.0.1:8000/admin/posts`. The page displays an exception message: `Illuminate\Database\Eloquent\Model::fill(): Add [_token] to fillable property to allow mass assignment on [App\Post].` Below the message is the URL `http://127.0.0.1:8000/admin/posts`.

Below the browser window, a stack trace is shown. The stack trace includes the following frames:

- `vendor/laravel/framework/src/Illuminate/Database/Eloquent/Model.php` (line 331)
- `6 vendor frames...`
- `app/Http/Controllers/Admin/PostController.php` (line 42)
- `App\Http\Controllers\Admin\PostController` (line 56)
- `1 unknown frame`

The stack trace also shows the source code for `Illuminate\Database\Eloquent\Model::fill` in `vendor/laravel/framework/src/Illuminate/Database/Eloquent/Model.php`, starting at line 316. The code includes a comment: `* @throws \Illuminate\Database\Eloquent\MassAssignmentException` and a public function `fill(array $attributes)` that iterates over the attributes and assigns them to the model.

Mass Assignment

Para resolver a exception lançada acima, sobre o atributo `$fillable` e o seu questionamento ao mesmo tempo, nós precisamos de fato definir este atributo `$fillable` lá no model Post.

Agora para que serve este atributo, tecnicamente ele é bem simples. Como estamos passando esta atribuição em massa, precisamos indicar para o Model/Eloquent que ao salvarmos os dados ou atualizarmos usando a atribuição em massa, que ele preencha somente os valores para os campos definidos no array desta propriedade, ou seja, ele só vai permitir valores para as colunas que estiverem registradas no atributo `$fillable`.

Então vamos adicionar ele em nosso model Post e logo após comentarmos mais um pouco sobre este detalhe.

Mass Assignment

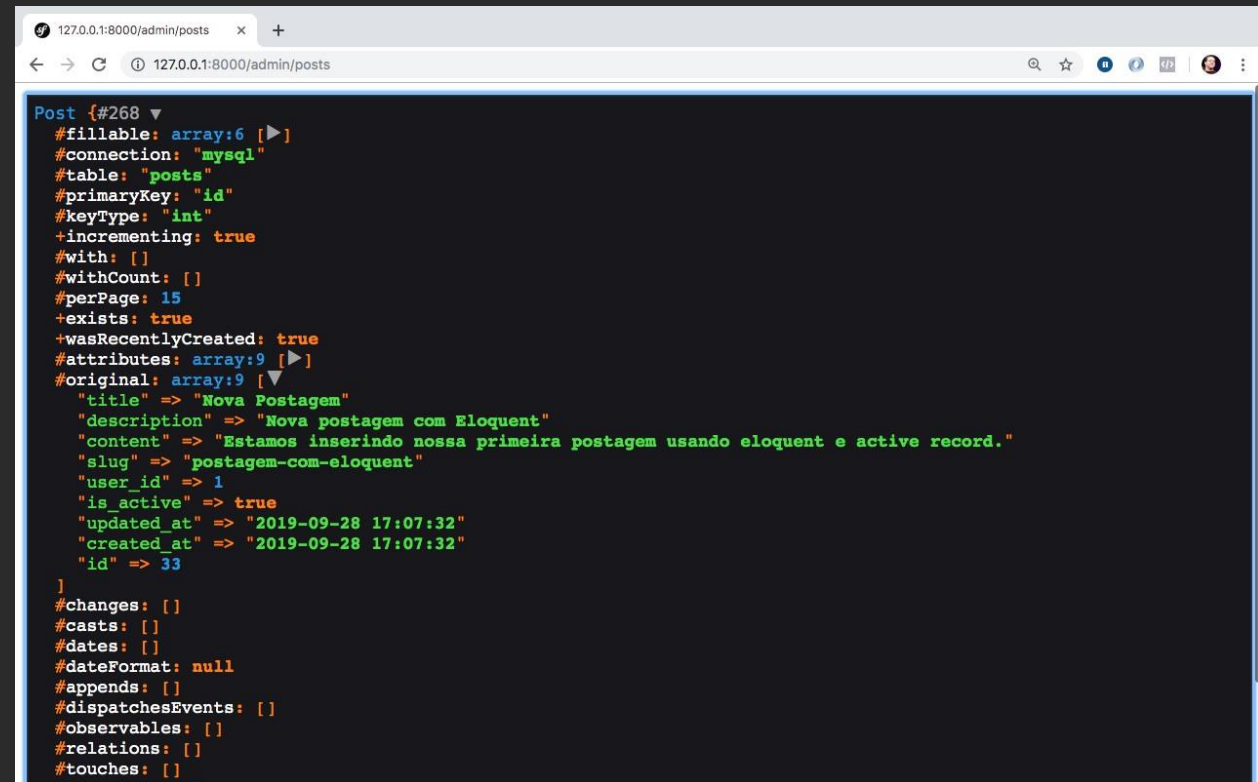
Veja a alteração em Post.php:

```
1  <?php
2
3  namespace App;
4
5  use Illuminate\Database\Eloquent\Model;
6
7  class Post extends Model
8  {
9      protected $fillable = [
10          'title',
11          'description',
12          'content',
13          'slug',
14          'is_active',
15          'user_id'
16      ];
17  }
```

Mass Assignment

Agora com os campos adicionados no atributo \$fillable vamos enviar os dados novamente do nosso formulário.

Veja o resultado, no dd, vindo do método create:



```
Post {#268}
  #fillable: array:6 [▶]
  #connection: "mysql"
  #table: "posts"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: true
  #attributes: array:9 [▶]
  #original: array:9 [▼]
    "title" => "Nova Postagem"
    "description" => "Nova postagem com Eloquent"
    "content" => "Estamos inserindo nossa primeira postagem usando eloquent e active record."
    "slug" => "postagem-com-eloquent"
    "user_id" => 1
    "is_active" => true
    "updated_at" => "2019-09-28 17:07:32"
    "created_at" => "2019-09-28 17:07:32"
    "id" => 33
  ]
  #changes: []
  #casts: []
  #dates: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
```

Mass Assignment

O método create ao criar um dado, retorna este dado criado junto com seu id na base como resultado.

Veja o conteúdo da informação abrindo a propriedade original.

A segurança do método create, usando a atribuição em massa, se dá pela propriedade \$fillable no model que uma vez definida e tendo as colunas permitidas só teremos o preenchimento das informações para a coluna mapeada nesta propriedade.

Agora como fazemos a atualização do dados massa? Vamos lá.

Atualizando Dados em Massa

Para atualizarmos os dados vamos trabalhar aqui com nossa view de edição e conhecer mais alguns detalhes do Laravel, crie lá dentro da pasta resources/views/posts o arquivo edit.blade.php e adicione o conteúdo abaixo:

Atualizando Dados em Massa

```
1  <form action="{{route('posts.update', ['postId' => $post->id])}}" method="post">
2
3      @csrf
4      @method("PUT")
5
6      <div class="form-group">
7          <label>Titulo</label>
8          <input type="text" name="title" class="form-control" value="{{ $post->title }}"
9      ">
10
11      </div>
12
13      <div class="form-group">
14          <label>Descrição</label>
```

Atualizando Dados em Massa

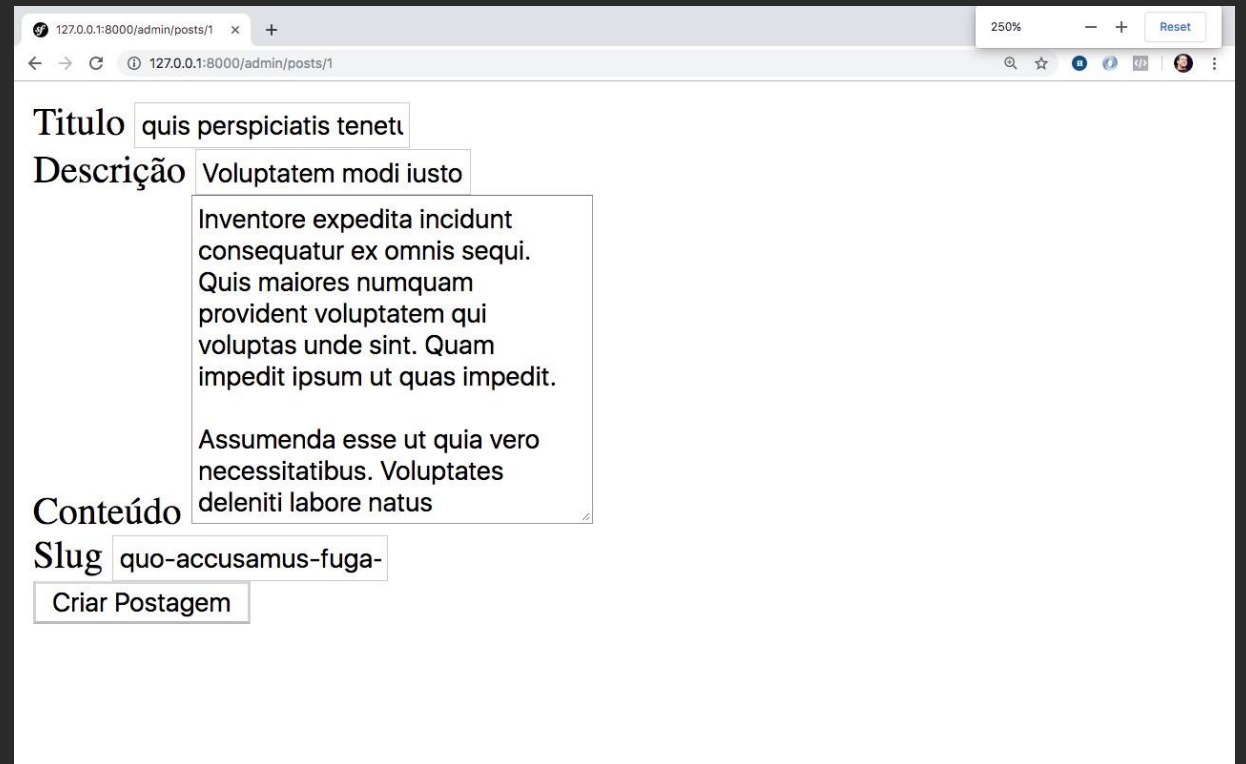
```
14         <input type="text" name="description" class="form-control" value="{{ $post->d\
15     escription }}">
16     </div>
17
18     <div class="form-group">
19         <label>Conteúdo</label>
20         <textarea name="content" id="" cols="30" rows="10" class="form-control">{{ $p\
21     ost->content }}</textarea>
22     </div>
23
24     <div class="form-group">
25         <label>Slug</label>
26         <input type="text" name="slug" class="form-control" value="{{ $post->slug }}">
27     </div>
28
29     <button class="btn btn-lg btn-success">Atualizar Postagem</button>
30 </form>
```

Atualizando Dados em Massa

Agora, lá no método show do PostController substitua a linha do dd pelo trecho abaixo:

```
1 return view('posts.edit', compact('post'));
```

Se você acessar o link <http://aula.test/admin/posts/1> você obterá o resultado seguinte:



127.0.0.1:8000/admin/posts/1 x + 250% - + Reset

← → ↻ ⓘ 127.0.0.1:8000/admin/posts/1 🔍 ☆ 🌐 📄 👤 ⋮

Título

Descrição

Conteúdo

Slug

Atualizando Dados em Massa

Agora vamos entender os códigos do formulário de edição anterior. Vamos lá. Primeiramente atente a chamada da rota na action do formulário:

```
1 <form action="{{route('posts.update', ['post' => $post->id])}}" method="post">
```

Enviaremos nossos dados para a rota de apelido `posts.update` atribuída pelo método `resource` do `Route`, informamos o nome do parâmetro dinâmico da rota no array do segundo parâmetro da função helper `route` que será o id da postagem, estes dados serão enviados para o método `update` lá do `PostController`, que vamos criar.

Temos agora mais uma alteração/novidade, a chamada da diretiva [`@method`](#).

Atualizando Dados em Massa

Vamos entender ela: Sabemos que os formulários html só suportam os verbos http: post e get. Por meio da diretiva `@method` fazemos o Laravel interpretar o formulário em questão com o verbo definido na diretiva, ou seja, como usei o valor PUT este formulário será interpretado pelo Laravel como sendo enviado via verbo PUT e cairá para a execução do método `update` do controller, que é o método que recebe as solicitações quando usamos os verbos PUT ou PATCH em nossa requisição.

Continuando, agora em nossos inputs recebemos os valores vindos lá do controller. Quando fazemos uma busca pela postagem desejada, ele retorna um objeto populado com os dados desta postagem, o que nos resta é acessarmos eles respeitando os nomes das colunas mas chamando como atributos do objeto e isto pode ser visto em cada atributo `value` dos inputs do nosso formulário de edição.

Atualizando Dados em Massa

Agora precisamos definir o método para manipulação do dados enviados do formulário de edição, para isso crie um método chamado de update em seu controller com o conteúdo abaixo:

```
1 public function update($id, Request $request)
2 {
3     //Atualizando com mass assignment
4     $data = $request->all();
5
6     $post = Post::findOrFail($id);
7
8     dd($post->update($data));
9 }
```

Atualizando Dados em Massa

Perceba que busquei a postagem usando o método `findOrFail` pelo id vindo da url, o parâmetro dinâmico. Neste caso como quero atualizar, o método da atribuição para atualização em massa, é o método do eloquent `update` que me retorna um booleano para o sucesso ou falha desta execução.

Acessando o formulário de edição no link <http://aula.test/admin/posts/1> vamos alterar algum campo e clicar em Atualizar Postagem. Veja o resultado:

Vamos ao passo final do CRUD, a remoção de um dado com Eloquent.

Deletando Registros

Bom para completarmos nosso ciclo, vamos deletar postagens de nossa base. Isso será realizado pelo método delete do Eloquent. Veja o método abaixo, a ser adicionado no PostController:

```
1 public function destroy($id)
2 {
3     $post = Post::findOrFail($id);
4
5     dd($post->delete());
6 }
```

O método do controller como recurso que responde a chamada para remoção de um dado é o destroy, entretanto, o método do Eloquent que irá remover um dado é o delete que retorna um booleano para o caso de sucesso ou falha da operação de remoção do dado da base.

Deletando Registros

Perceba que como queremos remover uma postagem, precisamos buscar por ela via Eloquent para então removermos.

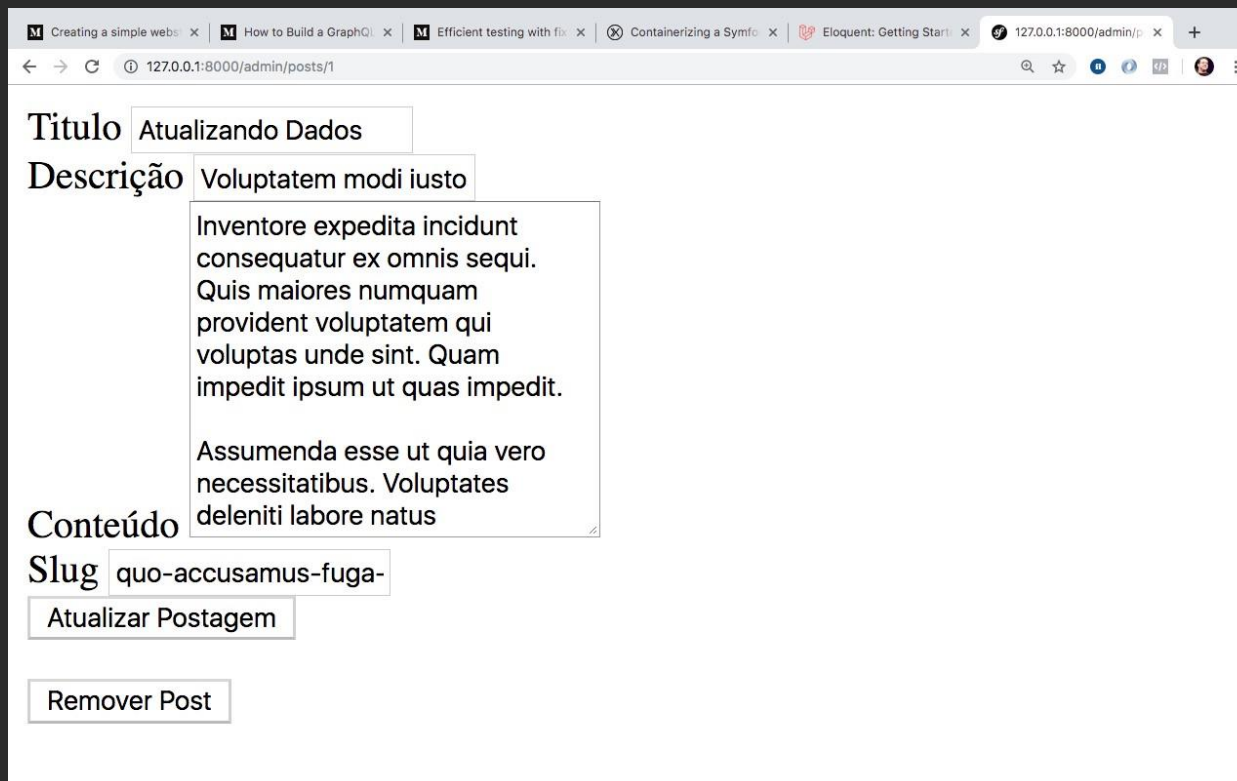
Agora precisamos adicionar o botão de remoção lá na tela de edição, neste caso como precisamos simular o envio do verbo http DELETE, vou precisar usar um form para o botão de remoção. Adicione o form abaixo, após o formulário de edição:

```
1 <form action="{{route('posts.destroy', ['post' => $post->id])}}" method="post">
2     @csrf
3     @method('DELETE')
4     <button type="submit">Remover Post</button>
5 </form>
```

Deletando Registros

Perceba também que temos adicionado o controle csrf para esta operação além da definição da diretiva [@method com o verbo http DELETE](#).

A tela de edição fica assim:

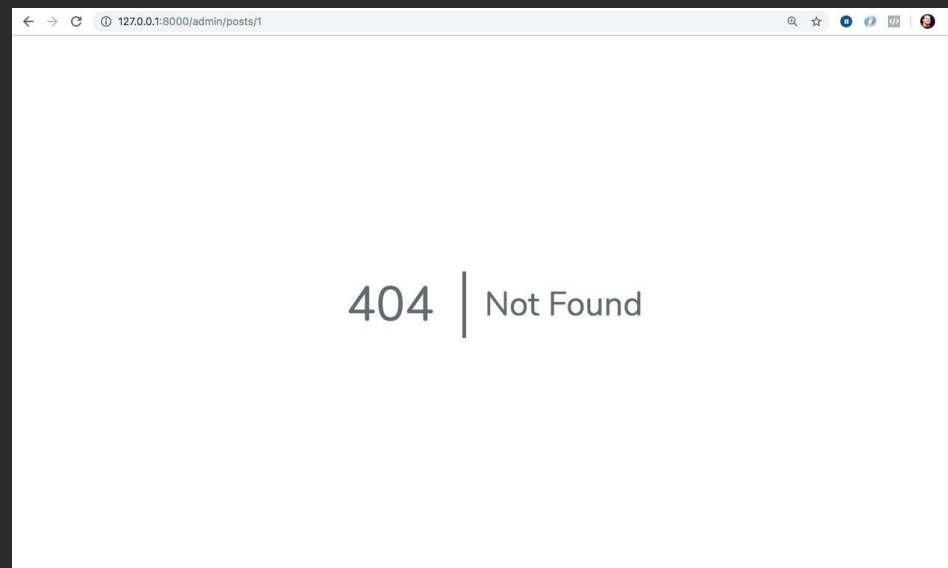


The screenshot shows a web browser window with multiple tabs. The active tab is titled '127.0.0.1:8000/admin/posts/1'. The browser address bar shows the URL '127.0.0.1:8000/admin/posts/1'. The page content is a form for editing a post. The form has the following fields and buttons:

- Título**: Atualizando Dados
- Descrição**: Voluptatem modi iusto
- Conteúdo**:
Inventore expedita incidunt consequatur ex omnis sequi.
Quis maiores numquam provident voluptatem qui voluptas unde sint. Quam impedit ipsum ut quas impedit.
Assumenda esse ut quia vero necessitatibus. Voluptates deleniti labore natus
- Slug**: quo-accusamus-fuga-
- Buttons**:
 - Atualizar Postagem
 - Remover Post

Deletando Registros

Ao clicar no botão remover você verá o resultado na tela, true para sucesso na remoção ou false para o caso de falha. Após removido se tentarmos acessar a mesma postagem teremos uma tela de 404 :



Agora completamos as 4 etapas de um CRUD completo utilizando o Eloquent que nos permite trabalhar com o banco pela visão de objetos.