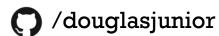
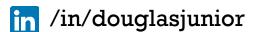
BANCO DE DADOS COM LARAVEL (PARTE 2)

Douglas Nassif Roma Junior







nassifrroma@gmail.com



Slides: https://git.io/vAd6S



AGENDA

- Introdução ao Eloquent ORM
- Definindo modelos
- Convenções dos modelos
- Recuperando modelos
- Coleções
- Recuperando registros únicos
- Inserindo e atualizando modelos
- Excluindo modelos
- Eventos
- API Resources



INTRODUÇÃO

 O Laravel inclui um poderoso ORM chamado Eloquent, que fornece uma implementação do ActiveRecord simples e elegante para trabalhar com seu banco de dados.

- Toda tabela do banco de dados tem um "Modelo" (Model) correspondente que é usado para interagir com a tabela.
- Modelos permitem que você consulte dados das tabelas, bem como inserir novos dados, editar e excluir dados existente.



DEFININDO MODELOS

• Modelos normalmente ficam dentro do diretório app, mas você é livre para coloca-los em qualquer lugar que possa ser carregado pelo auto-loader configurado no composer. json.

■ Todo modelo deve herdar da classe
Illuminate\Database\Eloquent\Model



DEFININDO MODELOS

A maneira mais fácil de criar um modelo é utilizando o Artisan:

php artisan make: model Tarefa

 Você também pode gerar uma migração junto ao seu modelo, para isso basta utilizar o parâmetro --migration

php artisan make:model Tarefa --migration



DEFININDO MODELOS

• Estrutura básica de um modelo:

```
c?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tarefa extends Model
{
    //
}
```



CONVENÇÕES DOS MODELOS

- Perceba que no exemplo anterior, nós não dissemos ao Eloquent qual tabela o modelo Tarefa representa.
- Por convenção, "snake case" e plural serão usados nos nomes das tabelas.

• Sendo assim, caso você queira definir um nome diferente para a tabela de tarefas, você deve utilizar a propriedade \$table.

```
protected $table = 'minhas_tarefas';
```



CONVENÇÕES DOS MODELOS

• O Eloquent também assume que toda tabela tem uma coluna id como sua chave primária. Para alterar você deve definir a propriedade \$primaryKey.

```
protected $primaryKey = 'codigo';
```

 Adicionalmente, também assume-se que toda chave primária é um número inteiro incremental. Para alterar este comportamento você pode substituir as propriedades \$keyType e \$incrementing.

```
protected $keyType = 'string';
protected $incrementing = false;
```



CONVENÇÕES DOS MODELOS

• Por padrão o Eloquent também espera que sua tabela contenha as colunas created_at e updated_at, se você não possui essas colunas você deve configurar a propriedade \$timestamps.

```
public $timestamps = false;
```

• Se você deseja alterar a formatação de seus timestamps você pode utilizar a propriedade \$dateFormat, assim como para renomea-los, você pode utilizar as constantes CREATED AT e UPDATED AT.

```
const CREATED_AT = 'data_criacao';
const UPDATED_AT = 'data_ultima_atualizacao';
protected $dateFormat = 'U';
```



RECUPERANDO MODELOS

Uma vez que o modelo foi definido, você já está pronto para utilizá-lo.
 Por exemplo, para consultar todos os registros, faça:

```
use App\Tarefa;
...
$tarefas = Tarefa::all();
```



RECUPERANDO MODELOS

 Todo modelo do Eloquent também fornece um Query Builder, sendo assim, você pode utilizar o que aprendemos no tópico anterior:

```
$tarefa = Tarefa::where('ativa', 1)->get();
```



COLEÇÕES

 Para métodos do Eloquent como get e all que retornam múltiplos registros, será retornado uma instância de Illuminate\Database\Eloquent\Collection.

- A classe **Collection** fornece uma <u>variedade de métodos úteis</u> para trabalhar com os resultados do Eloquent.
- Por exemplo, o reject, utilizado para filtrar itens da coleção:

```
$tarefasAtivas = $tarefas->reject(function ($tarefa) {
   return $tarefa->ativa === 1;
});
```



COLEÇÕES

 Assim como no Query Builder, o Eloquent possui o método chunk para percorrer muitos registros do banco de dados.

```
Tarefa::where('ativa', 1)->chunk(100, function ($tarefas) {
   foreach ($tarefas as $tarefa) {
      //
   }
});
```

 Já o método cursor permite iterar em uma grande quantidade de registros utilizando uma única consulta:

```
foreach (Tarefa::where('ativa', 1)->cursor() as $tarefa) {
   //
}
```



RECUPERANDO REGISTROS ÚNICOS

 Assim como recuperar todos os registros, você também pode recuperar registros únicos com find ou first.

```
// Retorna um registro por sua chave primária
$tarefa = Tarefa::find(1);

// Retorna o primeiro registro compatível com a consulta
$tarefa = Tarefa::where('ativa', 1)->first();

$tarefas = Tarefa::find([1, 2, 3]);
```



RECUPERANDO REGISTROS ÚNICOS

• Em alguns casos você pode querer disparar uma exceção se um modelo não for encontrado. Para isso você pode utilizar as funções findOrFail e firstOrFail.

```
$tarefa = Tarefa::where('ativa', 1)->findOrFail(1);
$tarefa = Tarefa::where('ativa', 1)->firstOrFail();
```



RECUPERANDO REGISTROS ÚNICOS

 Você também pode usar count, sum, max e <u>outros</u> com os modelos do Eloquent.

```
$tarefa = Tarefa::where('ativa', 1)->count();

$tarefa = Tarefa::where('ativa', 1)->max('id');
```



 Para criar um novo registro no banco de dados, você deve criar uma instância do modelo desejado, setar os atributos no modelo e então chamar o método save.

```
$tarefa = new Tarefa;
$tarefa->titulo = $request->titulo;
$tarefa->descricao = $request->descricao;
$tarefa->data = $request->data;
$tarefa->save();
```



• O método save também pode ser utilizado para atualizar os dados de um registro. Para atualizar um modelo, você deve recuperá-lo do banco de dados, setar os atributos desejados e então chamar o método save.

```
$tarefa = Tarefa::findOrFail($id);
$tarefa->titulo = $request->titulo;
$tarefa->descricao = $request->descricao;
$tarefa->ativa = $request->ativa;
$tarefa->save();
```



 Para atualizar diversos objetos, você pode utilizar o método update diretamente.

```
Tarefa::where('ativa', 1)
  ->where('titulo', 'Minha tarefa')
  ->update(['ativa' => 0]);
```



Outro recurso útil existentes nos módulos no Eloquent, é a
 possibilidade de buscar por um registro, e se ele não existir, cria-lo.
 Para isso existem os métodos firstOrCreate e firstOrNew.

```
// Recupera o usuário pelo e-mail, ou cria e insere no banco de dados se não existir
$usuario = Usuario::firstOrCreate(['email' => 'administrador@mail.com']);

// Recupera o usuário pelo e-mail, ou cria e insere no banco de dados se não existir

// com as propriedades email e nome
$usuario = Usuario::firstOrCreate(
    ['email' => 'administrador@mail.com'], ['nome' => 'Administrador']
);

// Recupera o usuário pelo e-mail, ou instancia o objeto se não existir
$usuario = Usuario::firstOrNew(['email' => 'administrador@mail.com']);

// Recupera o usuário pelo e-mail, ou instancia o objeto se não existir
// com as propriedades email e nome
$usuario = Usuario::firstOrNew(
    ['email' => 'administrador@mail.com'], ['nome' => 'Administrador']
);
```



 Assim como o firstOrCreate, você pode atualizar registros ou cria-los com updateOrCreate.

```
// Se existir um usuário com e-mail e senha especificados, então muda o nome para Admin
// Se não, cria-o.
$usuario = Usuario::updateOrCreate(
   ['email' => 'administrador@mail.com', 'senha' => '123456'],
   ['nome' => 'Admin']
);
```



EXCLUINDO MODELOS

• Você pode excluir um modelo que foi recuperado do banco de dados, simplesmente chamando o método delete.

```
$tarefa = Tarefa::findOrFail($id);
$tarefa->delete();
```

 Você também pode deletar um modelo pela chave primária chamando a função destroy.

```
$deletadas = Tarefa::destroy($id);

$deletadas = Tarefa::destroy(1, 2, 3);
$deletadas = Tarefa::destroy([1, 2, 3]);
```



EXCLUINDO MODELOS

• Por fim, você pode deletar registros com base em uma consulta. Para isso basta realizar o filtro desejado com a função where e em seguida invocar a função delete.

```
$deletadas = Tarefa::where('ativa', 0)->delete();
```



- Os modelos do Eloquent disparam uma variedade de eventos, permitindo que você possa interceptar estes eventos em diversos pontos do ciclo de vida dos modelos.
- São eles: retrieved, creating, created, updating, updated, saving, saved, deleting, deleted, restoring e restored.
- Eventos permitem que você execute facilmente códigos toda vez que um modelo específico seja recuperado, salvo ou atualizado na base de dados.



• Para registrar um evento, você deve sobrescrever a propriedade dispatchesEvents na classe do modelo desejado. Por exemplo:

```
use App\Events\TarefaDeleted;
use App\Events\TarefaSaved;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Notifications\Notifiable;
class Tarefa extends Model
  use Notifiable;
  protected $dispatchesEvents = [
    'saved' => TarefaSaved::class,
    'deleted' => TarefaDeleted::class,
```



• Ou ainda, é possível registrar um Observer para agrupar vários eventos de um mesmo modelo.

• Primeiro, você precisa registrar o Observer desejado no método boot do arquivo AppServiceProvider.

```
use App\Tarefa;
use App\Observers\TarefaObserver;
...
public function boot() {
   Tarefa::observe(TarefaObserver::class);
}
```



• E a definição da classe Observer segue a seguinte estrutura.

```
namespace App\Observers;
use App\Tarefa;
class TarefaObserver {
  public function created(Tarefa $tarefa) {
  public function deleting(Tarefa $tarefa) {
```



- Quando estamos criando uma API, você pode precisar de uma camada de transformação entre o modelo do Eloquent e o JSON devolvido na resposta.
- As classes API Resources permitem que você transforme seus modelos ou coleções em JSON de maneira fácil e expressiva.



 Para gerar um Resource, a maneira mais simples é através do comando make:resource.

php artisan make:resource TarefaResource

 Porém, em alguns casos você pode querer transformar coleções inteiras de objetos, e não apenas objetos individuais.

Para isso você pode utilizar seguinte comando:

php artisan make:resource Tarefas --collection





 Note que os arquivos foram criados no diretório app/Http/Resources.

• Exemplo de **Resource** individual:

```
class TarefaResource extends JsonResource {
  public function toArray($request) {
    return [
        'id' => $this->id,
        'titulo' => 'T: ' . $this->titulo,
        'descricao' => 'D: ' . $this->descricao,
        'data' => $this->data,
        'ativa' => $this->ativa === 1 ? true : false,
    ];
  }
}
```



• Exemplo de Resource de coleções.

```
use Carbon\Carbon;

class TarefaCollection extends ResourceCollection {
  public function toArray($request) {
    return [
        'resultado' => $this->collection,
        'metatados' => [
            'data_local' => new Carbon(),
        ],
     ];
  }
}
```



 Para utilizar estes Resources, basta instanciá-los e retorná-los nos métodos do controller.

```
$tarefa = Tarefa::where('ativa', 1)->findOrFail($id);
return new TarefaResource($tarefa);

$tarefas = Tarefa::where('ativa', 1)->get();
return new TarefaCollection($tarefas);
```



 Provavelmente você já percebeu, que a coleção de Resources contem uma propriedade data envolvendo todo o seu conteúdo.

• Este recurso pode facilmente ser desabilitado através da função withoutWrapping do facade Illuminate\Http\Resources\Json\Resource.

```
public function boot()
{
   Resource::withoutWrapping();
}
```



PAGINAÇÃO

 Os Resources de coleções também podem aceitar um paginator como conteúdo.

```
$tarefas = Tarefa::where('ativa', 1)->paginate(5);
return new TarefaCollection($tarefas);
```

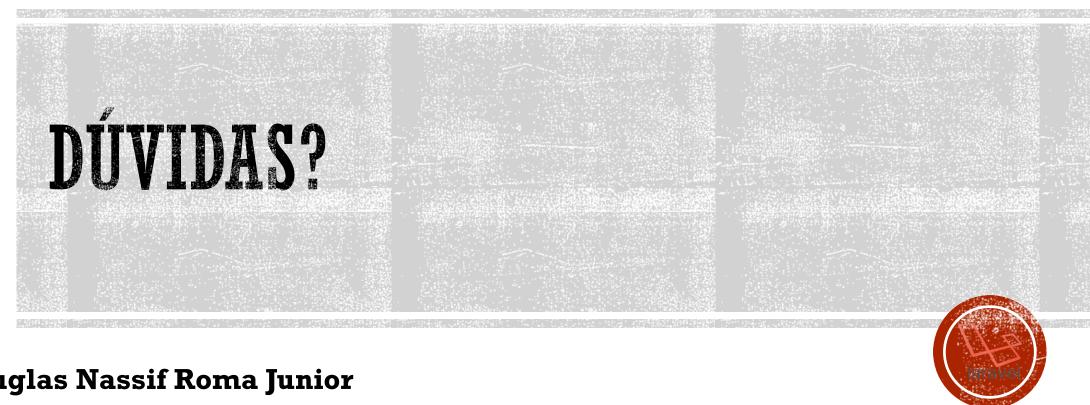
- Exemplo de consultas:
 - http://projeto-laravel.local/api/tarefas?page=1
 - http://projeto-laravel.local/api/tarefas?page=2



PAGINAÇÃO

 Para trabalhar com paginação, basta passar o parâmetro page na URL e você verá um resultado semelhante a:

```
"data":
"links": {
  "first": "http://projeto-laravel.local/api/tarefas?page=1",
  "last": "http://projeto-laravel.local/api/tarefas?page=3",
  "prev": null,
  "next": "http://projeto-laravel.local/api/tarefas?page=2"
"meta": {
  "current_page": 1,
  "from": 1,
  "last_page": 3,
  "path": "http://projeto-laravel.local/api/tarefas",
  "per_page": 2,
  "to": 2,
  "total": 5
```



Douglas Nassif Roma Junior

- /douglasjunior
- /in/douglasjunior
- douglasjunior.me
- massifrroma@gmail.com

Slides: https://git.io/vAd6S