ELOQUENT

Fernando Carrasco
Desenvolvedor

Relembrando:

https://github.com/fcarrasco0/dojo-2019 **(link do projeto do treinamento).**

Criando Models:

php artisan make:model Aluno

Criando Models com migration:

php artisan make:model Aluno -m

Executando o Tinker:

php artisan tinker

O que é Eloquent?

É uma ferramenta do laravel que facilita a interação com o banco de dados utilizando Models. Cada tabela tem uma Model correspondente e através da Model podemos acessar, modificar e consultar a tabela do banco de dados.

Eloquent Model Conventions:

- Nome da Model = Nome da tabela no singular.
- Chave primária = id.
- Chave primária incrementa automaticamente(autoincrement).
- Timestamps(created_at, updated_at).

ejcm

Eloquent Model Conventions:

Chave Primária - Pode ser definida utilizando a propriedade

protected $primarykey = “sua_chave_primária”;

Caso deseje utilizar uma chave que não seja um inteiro autoincrement, como email, deve desativar o autoincremente pela propriedade:
public $incrementing = false;

Timestamps - Para desativar o timestamps utilize a propriedade

public $timestamps = false;

OBS: Não é aconselhável desativar o timestamps, pois é muito útil saber quando sua tabela foi alterada.

ejcm

Recuperando Dados:

```
<?php

// Adicionamos o endereço do arquivo da model
use App\Samurai;
//salvamos toda a tabela que a model referencia com o método all() em $samurais
$samurais = App\Samurai::all();

//Listamos o nome de todos os registros da tabela salvos em $samurais.
foreach($samurais as $samurai){
        echo $samurai->nome.”\n”;
}
```

Recuperando Dados - Adicionando Restrições(queries):

Também podemos restringir a coleta de dados que queremos e obter um resultado mais preciso utilizando restrições(querys) como no exemplo abaixo:

```
$samurais = App\Samurai::where('missoes', '>', 1)
        ->orderBy('nome', 'desc')
        ->take(10)
        ->get();
```

Recuperando Dados - Adicionando Restrições(queries):

- **where(arg1, arg2)** - no arg1 selecionamos o campo da tabela e no arg2 o valor que queremos.
  Também pode ser usado em comparações, neste caso terá 3 argumentos e o argumento do meio será o comparador, exemplo: ->where('idade', '>' , 1 7);

- **orderBy(arg1, arg2)** - no arg1 escolhemos o campo pelo qual queremos ordenar(no exemplo **'nome'**) e no arg2 se é ordem crescente('asc') ou decrescente('desc').

ejcm

Recuperando Dados - Adicionando Restrições(queries):

- **take(10)** - seleciona 10 registros no banco de dados.

- **get()** - atribui todos os registros encontrados que satisfazem as restrições à variável $samurais.

OBS: Estas não são as únicas cláusulas de restrições existentes.

ejcm

Recuperando Dados - Selecionando UMA Model:

Muitas vezes é necessário recuperar apenas um registro específico.Podemos fazer isso pela sua chave primária com o método **find()**:

```
// recupera os dados do samurai com chave primária = 1
$samurai = App\Samurai::find(1);
```

ou utilizar restrições com o método **first()** neste caso, é útil quando não sabemos a id, mas sabemos o nome por exemplo:

```
// recupera os dados do primeiro samurai com nome ‘Joao’ encontrado na tabela.
$samurai = App\Samurai::where(‘nome’, ‘Joao’)->first();
```

ejcm

Recuperando Dados - Selecionando UMA Model:

EXCEPTIONS:

findOrFail() - Quando queremos saber se o registro está no banco de dados podemos utilizar este método, caso não esteja ele retornará uma exception:

```
$samurai = App\Samurai::findOrFail(1);
```

firstOrFail() - O objetivo é o mesmo do **findOrFail()**, mas utiliza o método **first()** no lugar do **find()** :

```
$samurai = App\Samurai::where('nome', 'Joao')->firstOrFail();
```

ejcm

Recuperando Dados - Valores Agregados:

O que são?

Valores agregados são métodos que retornam geralmente um total de valores de uma tabela, como por exemplo a soma de todas as faltas dos alunos da escola.

- **count()** - conta o número de registros que satisfazem as restrições impostas.

```
$katanas = App\Samurai::where('arma', 'katana')->count();
```

- **max()** - Seleciona o registro(s) com o valor máximo(mais alto) da tabela.

```
$experiente = App\Samurai::max('idade');
```

- **min()** - Seleciona o registro(s) com o valor mínimo(menor valor) da tabela.

```
$novato = App\Samurai::min('missoes');
```

ejcm

Recuperando Dados - Valores Agregados(cont):

- **avg()** - Retorna a média de todos valores da coluna selecionada da tabela.

```
$media_alunos = App\Dojo::avg('alunos_atuais');
```

- **sum()** - Retorna a soma de todos os valores da coluna selecionada da tabela.

```
$total_formados = App\Dojo::sum('alunos_formados');
```

Exercícios - Parte 1(consultas):

Utilize o tinker para os exercícios abaixo.

Na tabela Samurais e Dojos:

Exercício 1 - Listar todos os samurais com mais de 10 missoes;

Exercício 2 - Listar todos os samurais em ordem alfabética.

Exercício 3 - Listar o primeiro samurai com idade > 30 e menos de 20 missões

Exercício 4 - Calcular o aproveitamento de um dojo de acordo com alunos formados e reprovados.

Exercício 5 - Listar o dojo com aproveitamento > 70% e formados < 100

Exercício 6 - Verificar a média dos aproveitamentos dos dojos.

Inserindo Models na Tabela:

Para inserir um novo registro através da model, precisamos primeiro criar uma nova instância da model, lembrando que para acessar as propriedades de uma Model devemos incluir use App\Samurai no início do arquivo:

```
$novo_samurai = new Samurai;
```

Agora podemos acessar seus campos da seguinte forma:

```
$novo_samurai->idade = 27;
```

Por fim utilizamos o método save() para inserir os novos dados na tabela:

```
$novo_aluno->save();
```

Inserindo Models na Tabela - Exemplo:

```
//Exemplo de um código usado em uma model.
public function insereSamurai($request){

        $this->nome = $request->nome; //atribui o nome
        $this->idade = $request->idade; // atribui idade
        $this->posto = $request->arma; // atribui arma

        $this->save(); //salva o registro

}
```

Inserindo Models na Tabela - Exemplo:

```
//Exemplo de chamada do método da model na controller:

public function createSamurai(Request $request){

        $novo_samurai = new Samurai; //instancia a model
        $novo_samurai->insereSamurai($request);

         return response()->success($novo_samurai);

}
```

Atualizando valores da Model:

Para atualizar um registro primeiro precisamos recuperá-lo, ou seja, recuperar seus dados utilizando **find()**:

```
$samuraiX = App\Samurai::find(10);
```

Agora que temos o registro que queremos podemos alterar da mesma forma como inserimos:

```
$samuraiX->nome = “Kenshin”;

$samuraiX->posto = “Ronin”;

$samuraiX->save();
```

ejcm

Atualizando valores da Model:

Ou também podemos atualizar após recuperar os dados da seguinte forma:

```
$samuraiX->update([     “nome” => “Kenshin”,
                        “idade” => 50,
                        “posto” => “Ronin”]);
```

Mas é aconselhável fazer da outra forma por questões de segurança dos dados recebidos(para entender melhor, rever mass assignment).

Atualizando valores da Model:

Mass Updates(atualizações em massa):

Também podemos fazer atualizações em massa através de queries, por exemplo:

```
App\Samurai::where(“missoes”, “>”, 50)
    ->where(“idade”, “<” , 40)
    ->update([‘posto’ => ‘Senpai Nemae’]);
```

OBS: Caso use este método deve-se ter **EXTREMA ATENÇÃO** em relação a query que está sendo utilizada para ter **CERTEZA** que está alterando os registros corretos. Após fazer a atualização, a mesma **não poderá ser desfeita!**

Exercícios Parte 2(insert e update):

Utilizar a controller e model Aluno:

Exercício 1 - Fazer a função para inserir alunos na model e usá-lo na controller.

Exercício 2 - Fazer a função para atualizar samurais na model e usá-lo na controller.

Exercício 3 - Atualizar o posto de TODOS os samurais pela quantidade de missões:

- samurai com missões > 100, “Fudai”
- missoes < 25, “Kohai”
- o resto “Senpai”.

Desafio - criar o método alteraSamurai($request) na model que substitui em uma função exercício 1 e 2 e utilizá-lo na controller.

Bônus - atualizar o renome de todos os dojos de acordo com aproveitamento. exemplo: aprov=10.00 => renome = 1, aprov = 57 => renome=5.

ejcm

Deletando Models:

Para deletar um registro primeiro precisamos recuperá-lo e depois aplicar o método delete():

```
$harakiri = App\Samurai::find(1);
$harakiri->delete();
```

Também é possível deletar registros pela sua chave primária com o método destroy(), neste caso o registro é deletado sem ser instanciado:

```
App\Samurai::destroy(1);
```

ejcm

Soft Delete:

O que é Soft Delete?

Soft Delete é quando apagamos um registro no banco de dados **LOGICAMENTE**, ou seja, o registro não é apagado de verdade no banco de dados, ele é marcado como deletado e as futuras consultas ignoram os registros marcados como se não existisse mais no banco de dados.

O método que deleta o registro “fisicamente” no banco de dados chamamos de hard delete.

ejcm

Soft Delete(continuação):

Como usar:

Para poder utilizar o soft delete devemos incluir no inicio do arquivo da model a seguinte linha

```
Use Illuminate\Database\Eloquent\SoftDeletes;
```

Além disso devemos incluir na classe da model o seguinte código:

```
Class Aluno extends Model{

    use SoftDeletes;
    protected $dates = [‘deleted_at’];
}
```

Soft Delete(continuação):

Como usar(continuação):

Por fim precisamos incluir uma coluna “deleted_at” na tabela(migration) que a model referencia, podemos fazer isto desta forma:

```
Schema::table(‘alunos’, function($table){
    $table->softDeletes();
});
```

Agora quando utilizarmos algum dos métodos para deletar um registro, ele será deletado logicamente e a hora em que foi deletado será registrado na coluna “deleted_at”.

Soft Delete(continuação):

Recuperando registros deletados:

Para incluir as models deletadas usando softDeletes em uma query normal, utilizamos o seguinte código:

```
Samurai::withTrashed()->get();
```

O método “withTrashed()” recupera todas as model incluindo as que tem a coluna ‘deleted_at’ diferente de null, que são as deletadas logicamente.

Soft Delete(continuação):

Recuperando apenas registros deletados:

Para incluir apenas as models deletadas pelo softDeletes, utilizamos o seguinte código:

```
Samurai::onlyTrashed()->get();
```

O método “onlyTrashed()” recupera apenas as models que tem a coluna ‘deleted_at’ diferente de null, que são as deletadas logicamente.

Soft Delete(continuação):

Restaurando registros deletados:

Se por algum motivo precisar restaurar um registro deletado com softDeletes, utilizamos o seguinte código:

$samurai->restore() - neste caso restauramos uma instância

Samurai::onlyTrashed()->restore(); - neste restauramos todos registros deletados.

O método “restore()” restaura as models que tem a coluna ‘deleted_at’ diferente de null, que são as deletadas logicamente.

Exercícios Parte 3(Deletando):

Usar o tinker, exceto no exercício 1:

Exercício 1 - Deletar um registro com delete(). (crie a função na model)

Exercício 2 - Deletar um registro com destroy(). (pode ser feito na controller).

Exercício 3 - Descomente os códigos nos arquivos SamuraiController.php e samurais.blade.php linha 43.

Exercício 4 - atualize a página e verifique como ficou a tabela após o Exercício 3.

Exercício 5 - Delete todos os samurais com mais de 45 anos.

Exercício 6 - Agora modifique o método “index” na AlunoController.php para exibir apenas os registros deletados.

ejcm

Dúvidas?