UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DIO CEZAR GO

LARAVEL

Parte 2

SISTEMA DE AUTENTICAÇÃO

CRIANDO SISTEMA DE AUTH

- o *Artisan* do *Laravel* possui algumas ferramentas de automatização;
- uma delas é a criação completa dos recursos necessários para um sistema de login/cadastro/autenticação automática;
- execute então o comando:

```
php artisan make:auth
```

CRIANDO SISTEMA DE AUTH

- e como fazer para autorizar ou desautorizar uma *controller*?
- em uma *controller* que precisará de autenticação, basta inserir em seu construtor a chamada do *middleware auth*;
- todas a estrutura é criada no diretório *Controllers/Auth;*
- todas as views são armazenadas em *Views/auth;*

CRIANDO SISTEMA DE AUTH

EXEMPLO UTILIZAÇÃO DE UMA CONTROLLER COM AUTORIZAÇÃO

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class AgendaController extends Controller{
    public function __construct(){
        $this->middleware('auth');
    }

    public function index(){
        echo "Agenda Aqui!";
    }
}
```

QUERY BUILDER

QUERY BUILDER

- o *Query Builder* fornece uma fluente e conveniente interface para criar e executar consultas no banco de dados;
- o *Laravel* utiliza a associação de parâmetros do PDO para proteger a aplicação contra *SQL Injection*;

QUERY BUILDER - RECUPERANDO RESULTADOS

- se quisermos utilizar o banco de dados (de forma direta) em alguma classe, precisaremos importar a classe: `use Illuminate\Support\Facades\DB;`
- basta utilizar o método `table()` da classe abstrata `DB` para iniciar uma consulta;
- `table()` retorna os dados em uma coleção formatada, que utiliza objetos do PHP para formatar dados;
- note que nunca é recomendado recuperar dados do banco diretamente na *controller*, utilize o exemplo a seguir apenas como fins didáticos;

QUERY BUILDER - RECUPERANDO RESULTADOS

EXEMPLO UTILIZAÇÃO DO QUERY BUILDER PARA RECUPERAR DADOS

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;

class UserController extends Controller{
      public function index(){
            $users = DB::table('users')->get();
            foreach ($users as $user)
                  echo $user->name;
            return view('user.index', ['users' => $users]);
      }
}
```

QUERY BUILDER - UTILIZANDO WHERE

- em determinado momento pode ser necessário filtrar os resultados;
- por isso, pode-se utilizar o método aninhado *where()* do método *table()*;
- o seguinte exemplo recupera os resultados onde nome é John e retorna apenas a primeira tupla;

```
$user = DB::table('users')->where('name', 'John')->first();
echo $user->name;
```

QUERY BUILDER - UTILIZANDO WHERE

- mas você pode não precisar de todos os dados, apenas de uma coluna, para isso, é possível retornar somente o valor desejado:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

QUERY BUILDER - SELECTS

- nem sempre você irá precisar selecionar toda a tabela, para isso utilize o método *select* para especificar uma seleção personalizada;
- nesse caso, você irá selecionar apenas o `name` e o `email`;
- o email possuirá um alias `user_email`;
- note que é preciso do `->get()` para obter os dados;

```
$users = DB::table('users')->select('name', 'email as user_email')->get();
```

QUERY BUILDER - SELECT DISTINCT

- o método distinct permite que se receba apenas dados únicos, sem repetições;

```
$users = DB::table('users')->distinct()->get();
```

QUERY BUILDER - JOINS E UNIONS

- também é possível a utilização de *joins* e *unions* com o query builder;

QUERY BUILDER - JOINS E UNIONS

EXEMPLO DE UTILIZAÇÃO DE JOIN E UNION

```
// JOIN
$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();

// UNION
$first = DB::table('users')
            ->whereNull('first_name');

$users = DB::table('users')
            ->whereNull('last_name')
            ->union($first)
            ->get();
```

QUERY BUILDER - WHERE

- a utilização do where em sua forma mais simples utiliza 3 parâmetros;
- o primeiro → é o parâmetro referente ao campo;
- o segundo → é o parâmetro referente a operação;
- o terceiro → é o parâmetro referente ao valor;
- ainda é possível passar um *array* com vários parâmetros para a função;

QUERY BUILDER - WHERE

EXEMPLO DE UTILIZAÇÃO DE WHERE

```
$users = DB::table('users')
                ->where('votes', '>=', 100)
                ->get();

$users = DB::table('users')
                ->where('votes', '<>', 100)
                ->get();

$users = DB::table('users')
                ->where('name', 'like', 'T%')
                ->get();
```

QUERY BUILDER - WHERE

EXEMPLO DE UTILIZAÇÃO DE WHERE

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1']
])->get();

// com statments

$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

QUERY BUILDER - INSERTS

- é possível utilizar o método `insert()` para inserir dados em uma tabela;
- ou ainda, é possível passar múltiplos *inserts* a serem realizados;

QUERY BUILDER - INSERT

EXEMPLO DE UTILIZAÇÃO DE INSERT

```
// único insert

DB::table('users')->insert(
    ['email' => 'john@example.com', 'votes' => 0]
);

// múltiplos inserts

DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => 0],
    ['email' => 'dayle@example.com', 'votes' => 0]
]);
```

QUERY BUILDER - UPDATES

- assim como o `insert`, também é possível fazer `update` em suas tabelas;
- deve-se aplicar o método agregado `where` para indicar a condição do *update*;

QUERY BUILDER - UPDATE

EXEMPLO DE UTILIZAÇÃO DE UPDATE

```
DB::table('users')
            ->where('id', 1)
            ->update(['options->enabled' => true]);
```

QUERY BUILDER - DELETES

- e ainda, assim como o `insert` e `update`, também é possível fazer `delete` em suas tabelas;
- deve-se aplicar o método agregado `where` para indicar a condição do `delete`;

QUERY BUILDER - DELETE

EXEMPLO DE UTILIZAÇÃO DE DELETE

```
// apaga tudo da tabela users

DB::table('users')->delete();

// apaga com uma condição especifica

DB::table('users')->where('votes', '>', 100)->delete();
```

QUERY BUILDER - DELETE

EXEMPLO DE UTILIZAÇÃO DE DELETE

```
// apaga tudo da tabela users

DB::table('users')->delete();

// trunca a tabela toda

DB::table('users')->truncate();

// apaga com uma condição especifica

DB::table('users')->where('votes', '>', 100)->delete();
```

QUERY BUILDER - RAW

- pode ser necessário, em algum momento escrever um SQL sem a utilização do *Query Builder*;
- para isso pode-se utilizar as expressões em *Raw*;
- IMPORTANTE → tome cuidado com SQL Injection, pois essas strings serão inseridas diretamente na consulta;

QUERY BUILDER - RAW

EXEMPLO DE UTILIZAÇÃO DE RAW

```
$users = DB::table('users')

$users = DB::table('users')
         ->select(DB::raw('count(*) as user_count, status'))
         ->where('status', '<>', 1)
         ->groupBy('status')
         ->get();
```

ELOQUENT

O ELOQUENT

- é um *Object Relational Model* que mapeia cada tabela do banco de dados em uma model específica no sistema;
- necessita da importação da classe: `Illuminate\Database\Eloquent\Model`

O ELOQUENT

- a forma mais fácil de se criar uma model é a utilização do artisan:

```
php artisan make:model User
```

O ELOQUENT

- vamos utilizar um exemplo para armazenamento de uma tabela de vôos;
- por *default* utiliza-se a notação "*snake case*" que transforma o nome do model em seu respectivo plural, em letras minúsculas;
- por exemplo: o model *Flight* ficaria com o nome de tabela *flights*;
- mas opcionalmente, podemos definir um atributo para especificar o nome da tabela a ser trabalhada com o modelo em questão;

O ELOQUENT - FLIGHT

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
      // opcional
      protected $table = 'my_flights';
}
```

O ELOQUENT - PRIMARY KEYS

- por padrão o *Eloquent* atribui que toda tabela tem uma *pk* com o nome *id*;
- você pode alterar isso pelo atributo: `protected $primaryKey;`

O ELOQUENT - TIMESTAMPS

- por padrão o eloquent espera que todas as suas tabelas tenham dois campos obrigatórios: `created_at` e `updated_at;`
- estes campos guardam automaticamente cada criação e alteração de uma tupla;
- caso você não queira estes campos pode-se definir:

```
protected $timestamps = false;
```

O ELOQUENT - OBTENDO OS MODELS

- seu *model* está conectado a seu banco e por isso é muito simples manipular os seus dados;
- imagine que o *Eloquent* é um poderoso *Query Builder* que permitirá realizar as operações com suas tabelas;
- note que, no exemplo, se importa o *model*, e consequentemente ele vira um objeto com todas as propriedades do *Query Builder*, sem a necessidade de se especificar a tabela;

O ELOQUENT - OBTENDO OS MODELS

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT COM MODEL

```
<?php

      use App\Flight;

      // exemplo 1

      $flights = App\Flight::all();

      foreach ($flights as $flight) {
            echo $flight->name;
      }

      // exemplo 2

      $flights = App\Flight::where('active', 1)
                  ->orderBy('name', 'desc')
                  ->take(10)
                  ->get();
```

O ELOQUENT - INSERT

- para salvar (inserir) um *model*, basta instanciar um objeto, inserir os seus atributos e chamar o método *save()*;

O ELOQUENT - INSERT

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT COM INSERT

```
<?php

      namespace App\Http\Controllers;

      use App\Flight;
      use Illuminate\Http\Request;
      use App\Http\Controllers\Controller;

      class FlightController extends Controller{
      public function store(Request $request){
            // Validate the request...
            $flight = new Flight;
            $flight->name = $request->name;
            $flight->save();
      }
}
```

obviamente aqui você criará um método na model que recebe o request e faz o save

O ELOQUENT - UPDATE

- o mesmo se aplica ao *update*, mas nesse caso, deve-se antes selecionar um objeto específico, alterar seus atributos e depois utilizar o save();

O ELOQUENT - UPDATE

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT COM UPDATE

```
<?php

      namespace App\Http\Controllers;

      use App\Flight;
      use Illuminate\Http\Request;
      use App\Http\Controllers\Controller;

      class FlightController extends Controller{
      public function store(Request $request){
            $flight = App\Flight::find(1);
            $flight->name = 'New Flight Name';
            $flight->save();
      }
}
```

obviamente aqui você criará um método na model que recebe o request e faz o update

O ELOQUENT - DELETE

- o mesmo se aplica ao *delete*, mas nesse caso, deve-se antes selecionar um objeto específico, alterar seus atributos e depois utilizar o delete();
- pode-se ainda utilizar o `Illuminate\Database\Eloquent\SoftDeletes;`
- com isso apenas uma exclusão lógica do registro será efetuada;
    - nesse caso um outro campo chamado `deleted_at` será criado em sua tabela;

O ELOQUENT - DELETE

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT COM DELETE

```
<?php

      namespace App\Http\Controllers;

      use App\Flight;
      use Illuminate\Http\Request;
      use App\Http\Controllers\Controller;

      class FlightController extends Controller{
      public function store(Request $request){
            $flight = App\Flight::find(1);
            $flight->delete();
      }
}
```

obviamente aqui você criará um método na model que recebe o request e faz o delete

RELACIONAMENTOS DO ELOQUENT

O ELOQUENT - RELACIONAMENTOS

- as tabelas, como sabemos são relacionadas umas com as outras;
- o *Eloquent* fornece uma estrutura para a gerenciar automaticamente os relacionamentos dos tipos:
  - One To One;
  - One To Many;
  - Many To Many;
  - Has Many Through;
  - Polymorphic Relations;
  - Many To Many Polymorphic Relations;

O ELOQUENT - RELACIONAMENTOS

- Has Many Through → fornece um atalho conveniente para acessar relacionamentos distantes; por exemplo, *Country* tem muitos *Posts* através de *User*;
- Polymorphic Relations → permite que uma *Model* pertença a mais de uma *Model* em uma associação única; por exemplo, poderíamos ter uma *Model Comentários* que pudesse pertencer a *Posts* e a *Vídeos*;
- Many To Many Polymorphic Relations → Pensando no exemplo: *Post* e *Vídeo*, que possuem *Tags*, permite a obtenção de uma lista única de *Tags* que é compartilhada entre *Posts* e *Vídeos*;

O ELOQUENT - ONE TO ONE

- é um relacionamento simples no qual uma tabela tem apenas um registro da outra;
- por exemplo *User* poderia ter um *Phone*;
- vamos usar este exemplo para construir todo o cenário de relacionamentos que está disponível pelo *Eloquent*;
- a grande sacada é definir dentro da *Model* um método que irá "abstrair" o relacionamento e retornar o objeto em questão;

O ELOQUENT - ONE TO ONE

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT ONE TO ONE

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    public function phone(){
        return $this->hasOne('App\Phone');
    }
}
```

O ELOQUENT - ONE TO ONE

- depois que o relacionamento é definido, pode-se utilizar dentro de sua regra de negócios um telefone de um usuário específico:

```
$phone = User::find(1)->phone;
```

O ELOQUENT - ONE TO ONE

- o *Eloquent* determina a chave estrangeira do relacionamento com base no nome do *Model*;
- nesse caso, *Phone* assume que tem uma chave estrangeira automaticamente formatada como "user_id";
- mas é possível sobrescrever essa definição utilizando:

```
return $this->hasOne('App\Phone', 'foreign_key');
```

O ELOQUENT - ONE TO ONE

- e pode ser útil a definição do relacionamento inverso, no qual em *Phone* definimos que este pertence a um *User*;

O ELOQUENT - ONE TO ONE BELONGS TO

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT ONE TO ONE BELONGS

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model{
    public function user()    {
        return $this->belongsTo('App\User');
    }
}
```

O ELOQUENT - ONE TO MANY

- define um relacionamento no qual uma tabela possui muitos registros de determinado tipo;
- segue os mesmos padrões definidos em *one to one*;
- vamos estudar um exemplo de: *Posts* e *Comentários*;
- obviamente também podemos obter um *Post* a partir de seu *Comentário*, e para isso definimos o `belongsTo` em *Comentários*;

O ELOQUENT - ONE TO MANY

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT ONE TO MANY

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model{
    public function comments(){
        return $this->hasMany('App\Comment');
    }
}

// em outro momento...

$comments = App\Post::find(1)->comments;
foreach ($comments as $comment) {}
```

O ELOQUENT - ONE TO MANY INVERSE

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT ONE TO MANY INVERSE

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model{
    public function post(){
        return $this->belongsTo('App\Post');
    }
}

// em outro momento

$comment = App\Comment::find(1);
echo $comment->post->title;
```

O ELOQUENT - MANY TO MANY

- estes relacionamentos são um pouco mais complicados;
- necessitam de uma tabela intermediária entre as tabelas em questão;
- vamos pensar no seguinte exemplo: *Usuários* e *Papéis*;
- entre essas tabelas precisaremos de uma tabela intermediária, ficando então com: *user*, *roles* e *role_user*;
- o nome *role_user* é gerado automaticamente pela ordem alfabética entre as tabelas, e contém também os campos fks: *user_id* e *role_id;*

O ELOQUENT - MANY TO MANY

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT MANY TO MANY

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    public function roles(){
        return $this->belongsToMany('App\Role');
    }
}

// em outro momento...

user = App\User::find(1);
foreach ($user->roles as $role) {}
```

O ELOQUENT - MANY TO MANY

EXEMPLO DE UTILIZAÇÃO DO ELOQUENT MANY TO MANY

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model{
    public function users(){
        return $this->belongsToMany('App\User');
    }
}
```

O ELOQUENT - MANY TO MANY

- mas ainda é possível definir o nome da tabela intermediária e dos campos de chaves estrangeiras:

```
return $this->belongsToMany('App\Role', 'role_user');
return $this->belongsToMany('App\Role', 'role_user', 'user_id', 'role_id');
```

O ELOQUENT - MANY TO MANY

- ainda é possível definir uma *Model* para a tabela intermediária:

```
return $this->belongsToMany('App\User')->using('App\UserRole');
```

THAT'S ALL FOLKS

MATERIAIS COMPLEMENTARES

- https://laravel.com/docs/5.5/queries
- https://laravel.com/docs/5.5/eloquent
- https://laravel.com/docs/5.5/eloquent-relationships