Comecei tudo de novo. Dessa vez, farei o site através do codeanywhere. Iniciei um container do Wordpress lá.

# Enviando repositório do computador para o Bitbucket

Para iniciar um repositório novo no Bitbucket, a forma mais fácil parece ser criar o repositório no computador e depois enviar o repositório para o bitbucket. Para fazer isso, seguir os seguintes passos:

* 1. Criar pasta do projeto
  2. Dar um gitbash na pasta
  3. Dar um “git init”
  4. Entrar no SourceTree e dar o primeiro commit no projeto (ainda local).
  5. Voltar a pasta e dar os seguintes comandos para enviar para o Bitbucket:
     1. git remote add origin https://fabiomolinar@bitbucket.org/fabiomolinar/photo2me-android.git
     2. git push -u origin –all
     3. git push origin --tags # pushes up any tags

# Abreviações

* -BD: botão direito

# Criando o git

Esse foi fácil. Apenas -BD na pasta onde ficará o projeto a abri o Git Bash.

Inserir o comando “git init”.

Depois, só adicionei a pasta ao Source Tree.

# Criando o laravel

Na pasta do projeto, através do composer, usar o comando “composer create-project --prefer-dist laravel/laravel photo2me”.

# Configurando o servidor

Como eu já tinha o projeto da Evolme rodando nessa máquina e não queria ficar modificando o arquivo httpd.conf toda hora que tivesse que mudar a pasta do projeto apontada para o Apache, modifiquei o arquivo de virtual hosts do Apache para que um url pré-definido me levasse apontasse ao apache uma pasta pré-definida.

No arquivo de hosts do Windows (“c:\windows\system32\drivers\etc\hosts”), adicionei “127.0.0.1 photo2me.com”, para que esse endereço aponte para o loopback.

No arquivo “httpd-vhosts.conf” que se encontra em “C:\xampp\apache\conf\extra”, adicionei as seguintes linhas:

|  |
| --- |
| <VirtualHost \*:80>  ServerName localhost  DocumentRoot "E:\Profissional\EvolMe\Repos v2\public"  </VirtualHost>  <VirtualHost \*:80>  ServerName photo2me.com  DocumentRoot "E:\Profissional\photo2me\Repos\photo2me\public"  <Directory "E:\Profissional\photo2me\Repos\photo2me\public">  Options Indexes FollowSymLinks Includes ExecCGI  AllowOverride All  Require all granted  </Directory>  </VirtualHost> |

Dessa forma, o localhost continua apontando para a pasto do projeto Evolme enquanto o novo url “photo2me.com” aponta para a pasta desse projeto.

# Rotas

Irei criar a seguinte estrutura para as rotas:

* Todas
  + Web
    - Publicas
      * “/”
      * “Cadastro”
      * “Login”
      * “Contato” (não sei se será usado)
    - Privadas
  + App
    - Publicas
    - Privadas

# Layouts padrões

* 1. Publico

O layout foi feito de tal forma que cada página que usa esse layout terá adicionado à ela um conteúdo padrão, estabelecido pelos “@include” e outro conteúdo específico de cada página que pode ser definido, no código da página, através de “@section”.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>photo2me</title>  {{-- Incluir os itens que estarão presentes em todos os heads --}}  @include('includes.head')  {{-- Espaço para heads específicos de cada página --}}  @yield('head')  </head>  <body>  <div class='master' id='header'>  {{-- Incluir os itens que estarão presentes em todos os headers --}}  @include('includes.header')  {{-- Espaço para headers específicos de cada página --}}  @yield('header')  </div>  <div class='master' id='conteudo'>  {{-- Espaço para o conteúdo de cada página --}}  @yield('conteudo')  </div>  <div class='master' id='footer'>  {{-- Espaço para o footer que estará presente em todas as páginas --}}  @include('includes.footer')  </div>  </body>  </html> |

* 1. Input design

Para o design dos inputs, irei usar um código em less que cria um css para os inputs bem legais. Para usar o less, basta:

* Abrir a command line na pasta onde está o arquivo .less e digitar “lessc less.less inputs.css”
* Depois, para usar o design nos inputs, basta usar o seguinte padrão para os inputs:

|  |
| --- |
| <div class="form-group">  <input type="text" name="email" required=''>  <label alt="{{ trans('mensagens.qual-seu-email') }}" placeholder="{{ trans('mensagens.email') }}"></label>  </div> |

* 1. Social buttons design

Para os botões para log in e sign up por redes sociais estou usando uma biblioteca (.css) chamada bootstrap-social.css. E um exemplo de botão do facebook se parece assim:

|  |
| --- |
| <a class="btn btn-block btn-social btn-facebook">  <span class="fa fa-facebook"></span> Facebook  </a> |

* 1. É possível, em um layout, herdar de outro layout. É o que estou fazendo com as páginas do dashboard.
  2. Escondendo/mostrando itens de acordo com viewport no bootstrap

Use a single or combination of the available classes for toggling content across viewport breakpoints.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | xs (<768px) | sm (≥768px) | md (≥992px) | lg (≥1200px) |
| .visible-xs-\* | Visible | Hidden | Hidden | Hidden |
| .visible-sm-\* | Hidden | Visible | Hidden | Hidden |
| .visible-md-\* | Hidden | Hidden | Visible | Hidden |
| .visible-lg-\* | Hidden | Hidden | Hidden | Visible |
| .hidden-xs | Hidden | Visible | Visible | Visible |
| .hidden-sm | Visible | Hidden | Visible | Visible |
| .hidden-md | Visible | Visible | Hidden | Visible |
| .hidden-lg | Visible | Visible | Visible | Hidden |

# Conteúdo padrões do head

# Conteúdo padrões do header

# Conteúdo padrões do footer

# Autenticação

Foi complicado configurar os esquemas de registro, login e logout. Importante de lembrar:

* Adicionar os middlewares responsáveis por cookies e seções às rotas com autenticação.

|  |
| --- |
| Route::group(['middleware' => ['web'], 'as' => 'autenticacao'], function(){  Route::get('auth/login','Auth\AuthController@getLogin');  Route::post('auth/login','Auth\AuthController@postLogin');  Route::get('auth/logout','Auth\AuthController@getLogout');  Route::get('auth/register','Auth\AuthController@getRegister');  Route::post('auth/register','Auth\AuthController@postRegister');  }); |

* E, dentro do AuthController, como minha função de logout se chama getLogout, tive que adicionar essa função a lista de exceções do middleware que faz redirecionamentos automáticos.

|  |
| --- |
| public function \_\_construct()  {  $this->middleware('guest', ['except' => ['logout','getLogout']]);  } |

* E modificar no AuthController quais são os campos que são necessários. Como eu quero que apenas o email e senha seja necessário, preciso retirar o nome da lista do validator.

|  |
| --- |
| protected function validator(array $data)  {  return Validator::make($data, [  //retirei o ‘name’ dessa linha  'email' => 'required|email|max:255|unique:users',  'password' => 'required|confirmed|min:6',  ]);  } |

* O método de registro padrão do Laravel utiliza pelo menos três campos, email, senha e confirmação de senha. Então, não esquecer de colocar um <input> com o “verificar minha senha”.

# JWT-Auth

**Depois de algumas tentativas, decidi que na parte de web usarei a parte de autenticação que vem do próprio laravel. Depois terei que criar uma interface entre o back-end e o aplicativo.** As instruções abaixo são para adicionar o JWT-Auth ao projeto.

Primeiro, segui as seguintes instruções:

|  |
| --- |
| composer require tymon/jwt-auth  php artisan jwt:generate  php artisan vendor:publish --provider="Tymon\JWTAuth\Providers\JWTAuthServiceProvider" |

Agora tive que partir para a criação do banco de dados.

Para usar o JWT-Auth, TENHO QUE ADICIONÁ-LO AOS CONTROLLERS!

# Banco de dados

* 1. Criação

Primeiro, vamos criar o banco de dados no <http://localhost/phpmyadmin>.

Modificado as seguintes linhas do .env:

|  |
| --- |
| DB\_HOST=localhost  DB\_DATABASE=photo2me  DB\_USERNAME=root  DB\_PASSWORD= |

Para criar novas tabelas, usaremos o commando do artisan “make:migration”.

* 1. Atualização

Para fazer a atualização de alguma tabela do banco, criamos uma nova migration relacionada à tabela em questão. Podemos fazer isso através do comando:

|  |
| --- |
| php artisan make:migration update\_nome\_table --table= nome |

E, no arquivo da migration, adicionar as colunas que queremos. Um exemplo de código abaixo.

|  |
| --- |
| public function up()  {  Schema::table('users', function (Blueprint $table) {  $table->enum('tipo\_cliente',array('unico','rede'));  $table->enum('tipo\_usuario',array('usuario','admin'));  $table->date('data\_nascimento')->nullable();  $table->string('sobrenome')->nullable();  $table->boolean('marketing');  });  } |

# Models

* 1. User
  2. Festa

Para fazer validação de cada tentativa de criar uma festa, irei usar do que o laravel chama de “validation”. Ver capitulo sobre validation.

* 1. Foto
  2. Ordem

# Validation

Para fazer uma validação especial, tive que criar um extensão da classe validator dentro do appserviceprovider->boot().

|  |
| --- |
| //Validador extra para evitar eventos com mais de "x" dia(s).  Validator::extend('mais\_x\_dias', function($attribute,$value,$parameters,$validator){  $inicio = Carbon::createFromTimestamp(strtotime($parameters[0]));  $fim = Carbon::createFromTimestamp(strtotime($value));  $dias = $fim->diffInDays($inicio);  if ((strtotime($value) - strtotime($parameters[0])) < 0){  //Data de fim é menor que a data de início. Outra validação é que tem que ser usada nesse caso.  return true;  } else {  if ($dias < $parameters[1]){  return true;  } else {  return false;  }  }  }); |

Com essa nova função de validação, pude usar o validator::make no controller das festas para fazer a validação do formulário de criação da festa.

|  |
| --- |
| $validator = Validator::make($request->all(),[  'numConvidados' => 'required|numeric|min:1',  'inicioFesta' => 'required|date|before:' . $request->fimFesta,  'fimFesta' => 'required|date|after:' . $request->inicioFesta . '|mais\_x\_dias:' . $request->inicioFesta . ',1'  ]); |