2015

Laravel + Angular

Combine um dos melhores frameworks PHP com a biblioteca Javascript mais usada no momento

+ MySql
+ RestFull
+ Composer
+ Bower
+ Bootstrap

Daniel Schmitz

Laravel e AngularJS (PT-BR)

Incluindo também Bootstrap, Bower, Composer e Restfull

Daniel Schmitz e Daniel Pedrinha Georgii

Esse livro está à venda em http://leanpub.com/laravelangular_pt

Essa versão foi publicada em 2016-02-26

Tweet Sobre Esse Livro!

Por favor ajude Daniel Schmitz e Daniel Pedrinha Georgii a divulgar esse livro no Twitter!

O tweet sugerido para esse livro é:

Comprei o livro Laravel+Angular, as duas melhores tecnologias web em um só livro!

A hashtag sugerida para esse livro é #soudev.

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

https://twitter.com/search?q=#soudev

Conteúdo

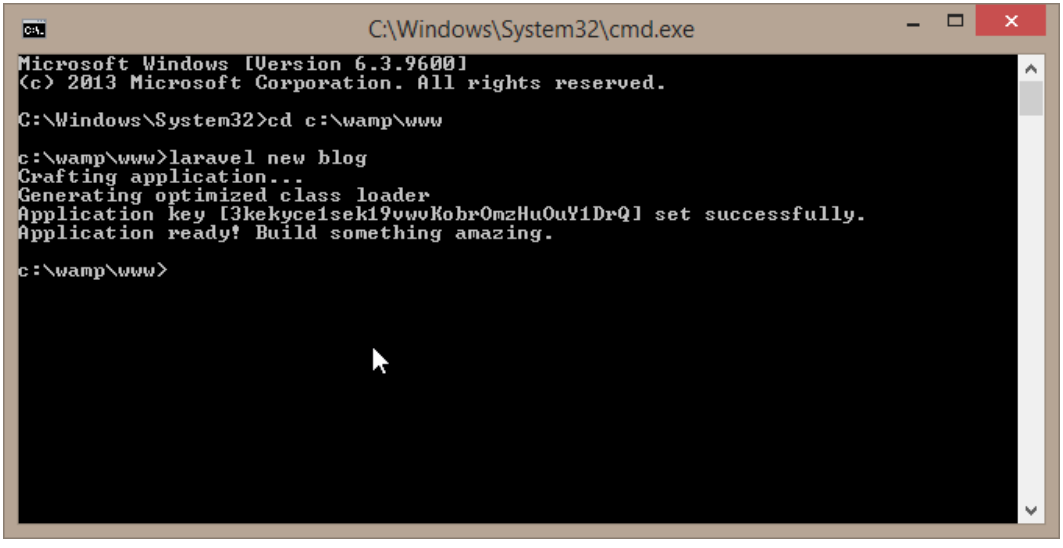
Parte 2 - Laravel

Capítulo 4 - Conhecendo o Laravel

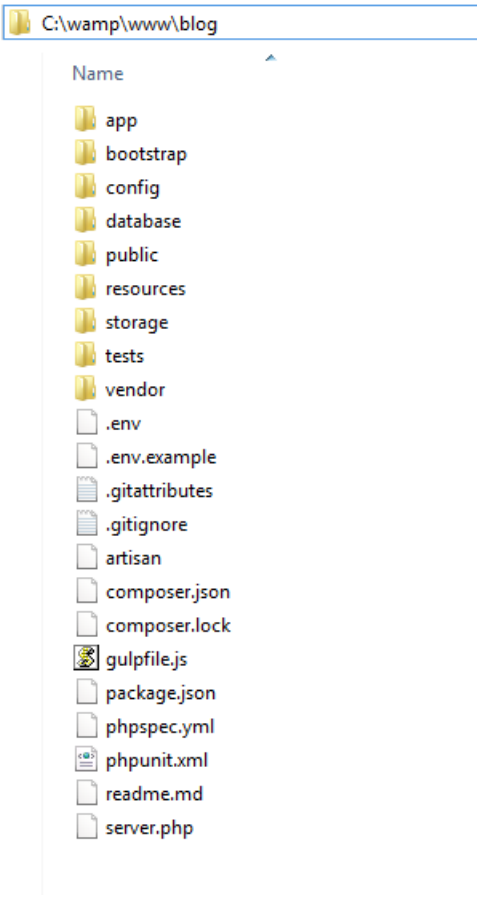
Agora que estamos com todas as bibliotecas devidamente instaladas, podemos dar início ao estudo do Laravel. Uma aplicação em Laravel pode ser criada através do seguinte comando:

```
laravel new blog
```

Neste comando, uma aplicação com o nome `blog` é criada. Vamos executar este comando no diretório web do nosso sistema, que pode ser `/home/user/www` no Linux ou no `c:\wamp\www` no Windows.



A estrutura de arquivos criada no projeto “blog” é semelhante a Figura a seguir:



Configurando o virtual host

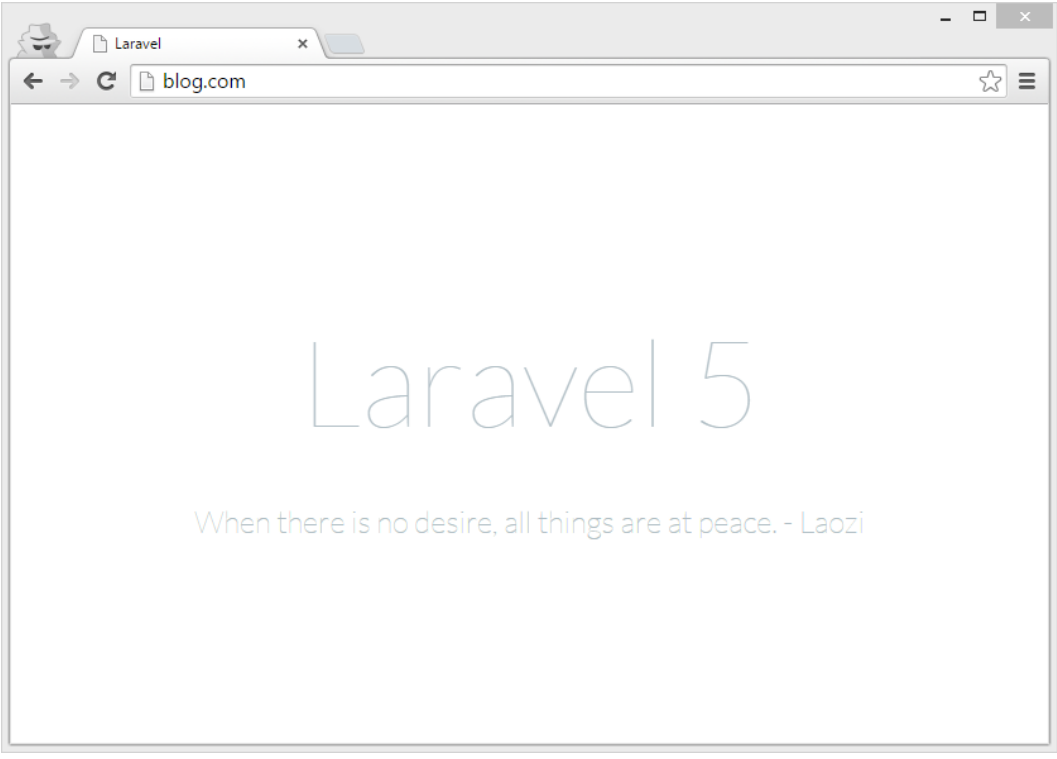
A primeira tarefa após criar a aplicação é configurar o seu *virtual host*. Vamos supor que esta aplicação deva ser acessada através do endereço `blog.com`. Para o ambiente Windows, edite o arquivo `C:\wamp\bin\apache\apache2.4.9\conf\httpd.conf` incluindo no final do mesmo o seguinte texto:

```
<VirtualHost *>
    ServerName blog.com
    DocumentRoot "c:/wamp/www/blog/public"
    <Directory "c:/wamp/www/blog/public">
        Options FollowSymLinks
        AllowOverride All
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

E altere o arquivo `hosts` incluindo o seguinte texto:

```
127.0.0.1         blog.com
```

Após reiniciar o Wamp Server, acesse a url `blog.com`, para obter a seguinte resposta:



Para ambientes no Linux

Siga os passos do capítulo 3 para criar o virtual host, assim como foi feito no domínio mysite.com.

Perceba que o domínio virtual foi criado apontando para a pasta `blog/public`, que deverá ser a única pasta visível ao acesso externo. Por questões de segurança, as outras pastas da aplicação, como "app" e "config", jamais devem ter acesso público. Não crie o domínio virtual apontando para a pasta da aplicação, principalmente em servidores de produção. Crie sempre apontando para a pasta `public`.

Permissão em diretórios

Caso tenha algum problema ao acessar a url `blog.com`, relativo a permissão, como por exemplo `Failed to open stream: Permission denied`, deve-se dar permissão

de escrita ao diretório `storage` da aplicação. No Linux, faça:

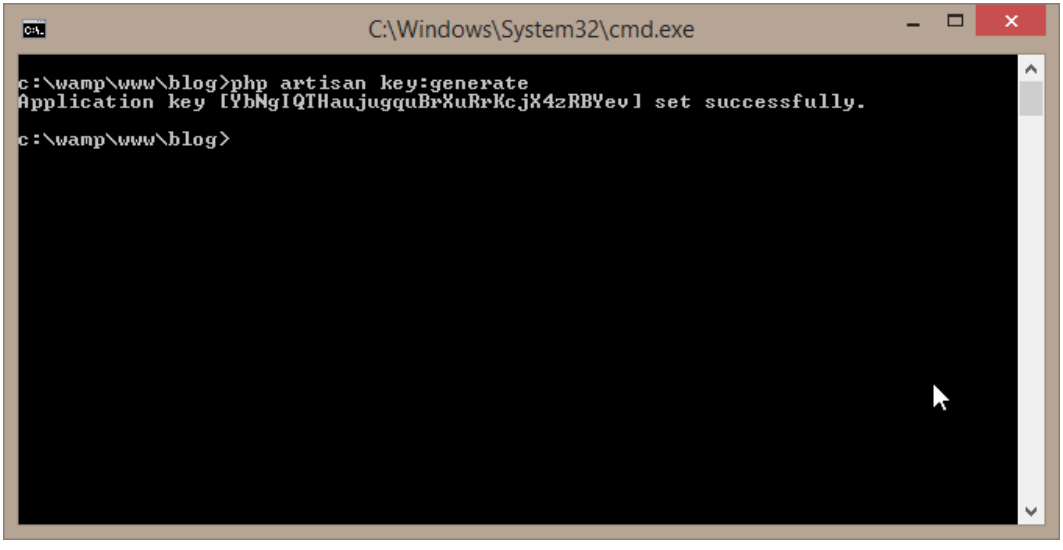
```
$ sudo chmod -R 777 www/blog/storage
```

Gerando uma chave de encriptação

É importante para a segurança da sua aplicação encriptar qualquer tipo de informação que será alocada na sessão ou nos cookies que o sistema cria. Para isso, é necessário executar o seguinte comando:

```
php artisan key:generate
```

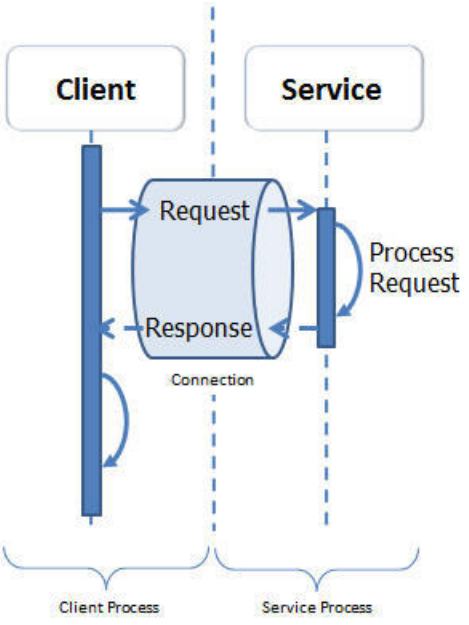
Execute-o no diretório blog, conforme a figura a seguir:



Roteamento (routes)

Na definição mais simples de acesso HTTP, temos sempre duas ações comuns em qualquer tecnologia web: *Requisição* e *Resposta*. Uma *Requisição* é realizada quando

o navegador (que chamamos de cliente) faz um acesso a um servidor, através de uma URL. Esta URL contém, no formato mais básico, o caminho de acesso até o servidor, que comumente chamamos de endereço web, e o tipo de requisição, que pode ser GET, POST, entre outras. Após o servidor web processar esta requisição, ele emite uma *Resposta* ao cliente que a solicitou, geralmente no formato texto. Esta "conversa" entre o cliente e o servidor pode ser ilustrada na figura a seguir:



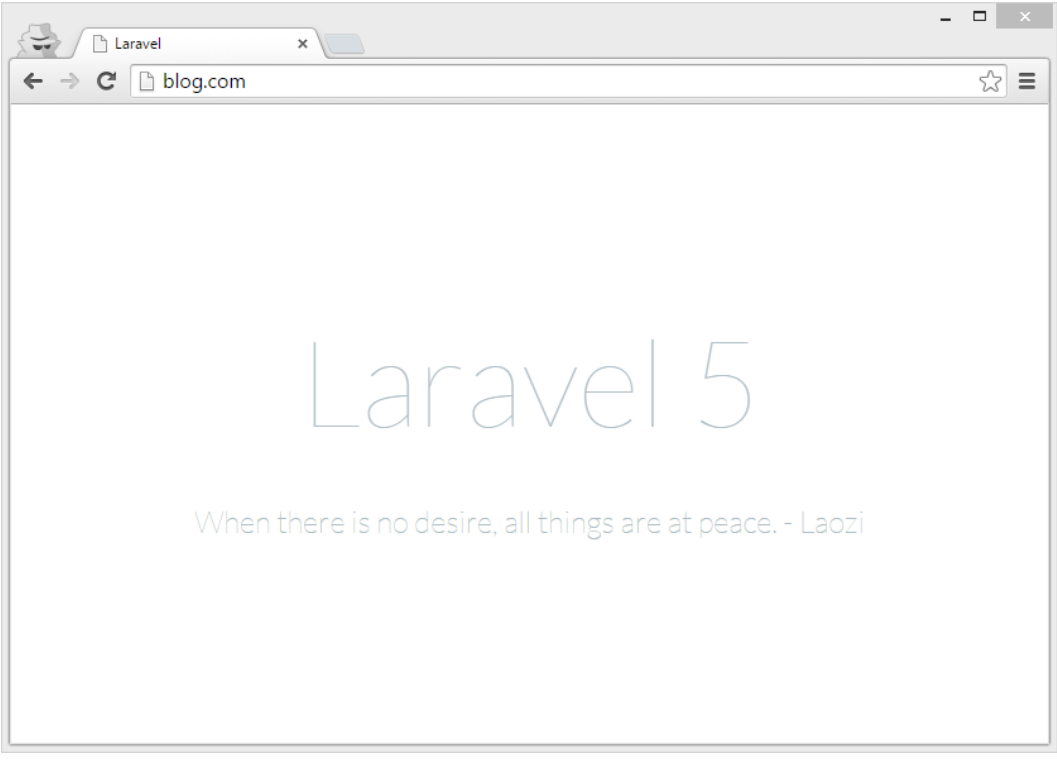
Este entendimento deve ser compreendido para que possamos dar prosseguimento à definição de roteamento. Definir uma rota é possibilitar que possamos configurar uma URL particular para executar algo único dentro do nosso sistema. Ou seja, através do roteamento poderemos criar URLs que irão definir como o AngularJS irá acessar o servidor para obter dados. Esta etapa é fundamental para que possamos entender como o AngularJS irá "conversar" com o Laravel.

Inicialmente, vamos abrir o arquivo de roteamento do Laravel, que está localizado em `app/Http/routes.php`:

blog\appHttp\routes.php

```
<?php
Route::get('/', function () {
    return view('welcome');
});
```

Através do método `Route::get` estamos criando uma configuração personalizada para a requisição GET, que é a requisição realizada pelo navegador quando acessamos alguma URL. O primeiro parâmetro deste método é `'/'` que significa o endereço raiz do site, nesse caso é: `blog.com` ou `blog.com/`. O segundo parâmetro é uma função anônima que será executada para definir como será a resposta ao cliente. Esta função contém o seguinte código `return view('welcome');` que define a criação de uma `view` do laravel, cujo resultado é exibido na imagem a seguir:

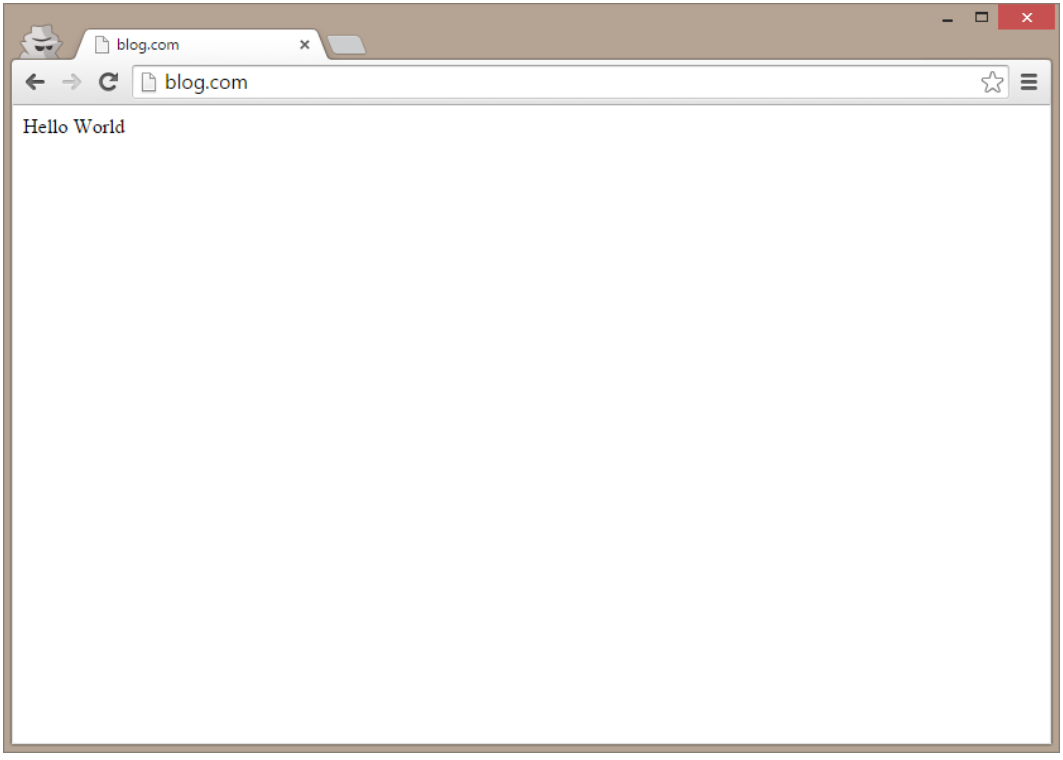


Vamos fazer uma alteração simples neste código, para que ao invés de gerar uma view do Laravel, retorne o texto “Hello World”. Veja:

blog\appHttp\routes.php

```
<?php
Route::get('/', function () {
    return "Hello World";
});
```

Ao recarregamos a página `blog.com`, temos o seguinte resultado:



Com isso, poderemos utilizar o roteamento do Laravel para criar todas as funções necessárias para que o AngularJS possa se comunicar com o servidor. Quando criarmos um projeto real, veremos que o ponto de partida do projeto será criar a

configuração de roteamento, no qual chamaremos de API RESTful, mesmo que o termo API não seja a definição mais correta para este processo.

Para que possamos criar uma API RESTful, é necessário conhecer todas as configurações de roteamento que o Laravel proporciona, e veremos isso a seguir em detalhes.

Tipos de Roteamento (verbs)

Uma requisição web pode ser classificada nos mais diferentes tipos, no qual chamamos pelo termo `VERBS`. Em nossos exemplos, iremos utilizar os seguintes tipos: `GET, POST, PUT, DELETE`. Cada tipo irá definir uma ação comum, que pode ser entendida pela tabela a seguir:

Método	Ação	Exemplo
GET	Responde com informações simples sobre um recurso	GET blog.com/user/1
POST	Usado para adicionar dados e informações	POST blog.com/user
PUT	Usado para editar dados e informações	PUT blog.com/user
DELETE	Usado para remover um registro	DELETE blog.com/user/1

Para utilizar outros tipos de requisição além do GET, pode-se usar o seguinte código:

blog\app\Http\routes.php

```
<?php
Route::get('/', function () {
    return "Hello World";
});

Route::post('user/', function () {
    return "POST USER";
});

Route::put('user/', function () {
```

```
    return "PUT USER";
});

Route::delete('user', function () {
    return "DELETE USER";
});
```

Também pode-se utilizar mais de um tipo de requisição, como no exemplo a seguir:

blog\app\Http\routes.php

```
<?php
Route::match(['get', 'post'], '/', function () {
    return 'Hello World with GET or POST';
});
```

É válido lembrar que, por convenção, sempre quando dados forem alterados deve-se utilizar POST ou PUT, e sempre que um dado for deletado, deve-se utilizar DELETE. Nos navegadores atuais, os tipos PUT e DELETE ainda não são suportados, mas isso não será um problema para o nosso sistema, pois todas as requisições do cliente em AngularJS feitas ao servidor Laravel serão via ajax.

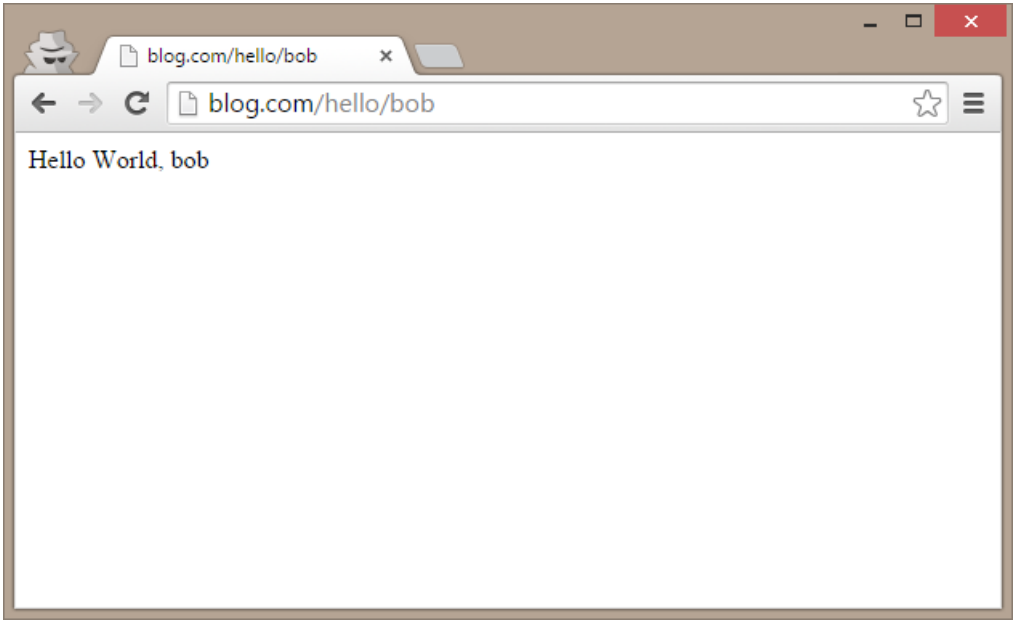
Repassando parâmetros no roteamento

Pode-se configurar um ou mais parâmetros que devem ser repassados pela URL. Suponha, por exemplo, que a URL "/hello/bob" tenha a seguinte resposta: "Hello world, bob!". Para isso, criamos o parâmetro `{name}`, de acordo com o código a sguir:

blog\app\Http\routes.php

```
<?php
Route::get('/hello/{name}', function ($name) {
    return "Hello World, $name";
});
```

Perceba que o parâmetro criado é definido na URL através do uso de chaves, e também definido como um parâmetro na função anônima. O resultado do código acima é visto a seguir:



Pode-se criar quantos parâmetros forem necessários, recomendando-se apenas que use os separadores `/` para cada um deles. Também pode-se repassar um parâmetro opcional, através do uso do `?`. O exemplo a seguir mostra uma forma simples de somar 2 ou 3 números:

blog\app\Http\routes.php

```
<?php
Route::get('/sum/{value1}/{value2}/{value3?}',
                        function ($value1,$value2,$value3=0) {

        $sum =  $value1+$value2+$value3;
        return "Sum: $sum";

});
```

Utilizando expressões regulares

As vezes é necessário estabelecer uma condição para que o reteamento seja bem sucedido. No exemplo da soma de números, o que aconteceria se utilizássemos a url "blog.com/sum/1/dois/4"? Possivelmente teríamos um erro no PHP, já que precisamos somar apenas números. Desta forma, podemos utilizar o atributo `where` como uma condição para que o roteamento seja realizado. O exemplo a seguir assegura que a soma seja realizada apenas com números:

blog\app\Http\routes.php

```
<?php
Route::get('/sum/{value1}/{value2}/{value3?}',
                    function ($value1,$value2,$value3=0) {
    $sum =  $value1+$value2+$value3;
    return "Sum: $sum";
})->where([
                        'value1'=>'[0-9]+',
                        'value2'=>'[0-9]+',
                        'value3'=>'[0-9]+'
                        ]);
```

Para validar uma string, pode-se usar `[A-Za-z]+`. Qualquer validação a nível de expressão regular pode ser utilizada.

Muitos roteamentos utilizam a chave primária do registro em suas URLs, o que torna necessário verificarmos se esta chave é um número inteiro. Por exemplo, para deletar um usuário, poderíamos utilizar a seguinte configuração:

blog\app\Http\routes.php

```
<?php
Route::delete('/user/{id}', function ($id) {
    return "delete from user where idUser=$id";
})->where('id', '[0-9]+');
```

O exemplo citado é perfeitamente válido, mas suponha que quase todas as suas rotas possuam ids, e seja necessário validá-las. Com isso todas as rotas teriam o `->where()`, tornando o código mais repetitivo e quebrando o princípio DRY (Dont Repeat Yourself) que devemos utilizar em nosso código.

Para resolver esse problema podemos configurar que toda variável chamada `id` deve ser obrigatoriamente um número. Para fazer isso no Laravel, devemos adicionar um `pattern` no arquivo `app/Providers/RouteServiceProvider.php`, da seguinte forma:

blog\app\Providers\RouteServiceProvider.php

```
<?php

namespace App\Providers;

use Illuminate\Routing\Router;
use Illuminate\Foundation\Support\Providers\RouteServiceProvider as \
ServiceProvider;

class RouteServiceProvider extends ServiceProvider
{
    protected $namespace = 'App\Http\Controllers';

    public function boot(Router $router)
    {
```

```
        $router->pattern('id', '[0-9]+');

        parent::boot($router);
    }

    /// code continues

}
```

A classe `RouteServiceProvider` provê todas as funcionalidades do roteamento da sua sua aplicação, dentre elas a criação das rotas que estão no arquivo `Http\\routes.php`. Através do método `pattern` é possível definir uma expressão regular padrão para a variável, nesse caso `id`. Desta forma, toda variável chamada `id` no roteamento terá a expressão regular testada.

Nomeando roteamentos

Pode-se adicionar um nome a uma rota, para que a mesma seja usada em outras rotas. Isso evita o uso de se escrever diretamente a URL da rota, melhorando o fluxo de código. No exemplo a seguir, suponha duas rotas distintas, uma irá criar um novo usuário e retornar um texto com um link para o seu perfil. Pode-se escrever o seguinte código:

blog\app\Http\routes.php

```
<?php
Route::get('user/{id}/profile', function ($id) {
    return "Exibindo o profile de $id";
});

Route::get('user/new', ['as' => 'newUser', function () {
    return "Usuário criado com sucesso.
    <a href='blog.com/user/1/profile'>Ver Perfil</a> ";
}]);
```

Perceba que no link `<a href....>` criado, incluímos o nome do domínio `blog.com` e o caminho da rota, o que não é bom pois se houver alguma mudança no nome do domínio ou na URL da rota, este link não irá mais funcionar. Para corrigir isso, devemos inicialmente dar um nome a rota que exibe o Perfil do usuário, da seguinte forma:

blog\app\Http\routes.php

```
<?php
Route::get('user/{id}/profile',
    ['as' => 'profileUser', function ($id) {
    return "Exibindo o profile de $id";
}]);
```

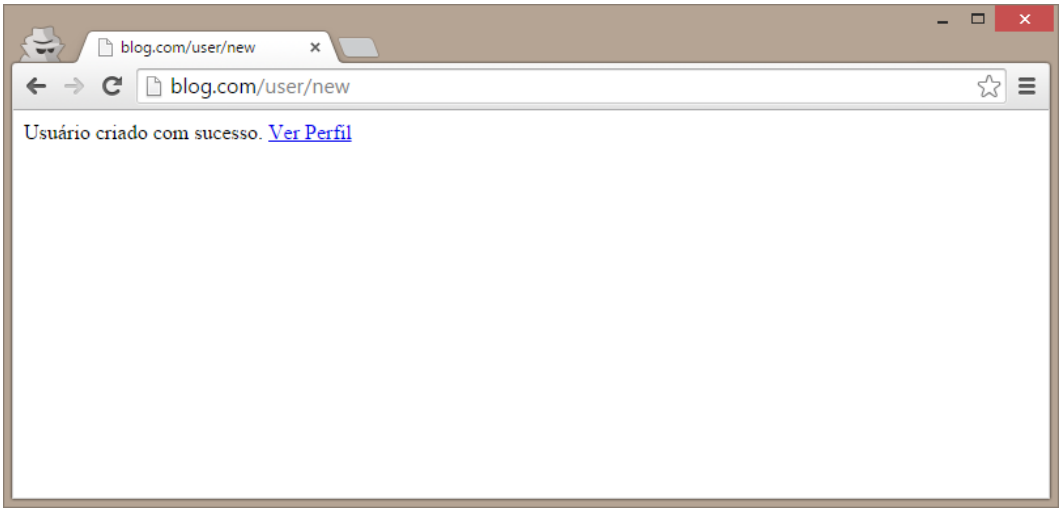
Perceba que o segundo parâmetro da método `Route::get` torna-se um *array*, que contém dois elementos, sendo o primeiro identificado pela chave `as` e que contém o nome do roteamento, e o segundo a função na qual já conhecemos.

Após criar o nome da rota, pode-se obter a URL completa através do método `route`, conforme o código a seguir:

blog\app\Http\routes.php

```
<?php
Route::get('user/{id}/profile',
        ['as' => 'profileUser', function ($id) {
    return "Exibindo o profile de $id";
}]);

Route::get('user/new', ['as' => 'newUser', function () {
    $userProfileLink = route('profileUser',['id'=>1]);
    return "Usuário criado com sucesso.
    <a href='$userProfileLink'>Ver Perfil</a>";
}]);
```

Neste código, usamos o método `route` repassando o nome da rota e o parâmetro `id`. Usamos essa informação para gerar corretamente o link para o perfil do usuário, conforme visto na figura a seguir.



Agrupando rotas

A configuração de rotas do Laravel permite o agrupamento de rotas para manter uma melhor leitura do código fonte. O exemplo a seguir é perfeitamente válido:

blog\app\Http\routes.php

```
<?php
Route::get('/post/new', function () {
    return "/post/new";
});

Route::get('/post/edit/{id}', function ($id) {
    return "/post/edit/$id";
});

Route::get('/post/delete/{id}', function ($id) {
    return "/post/delete/$id";
});
```

Mas o código destes 3 roteamentos pode ser melhorado, através do comando `Route::group` e utilizando um prefixo, da seguinte forma:

blog\app\Http\routes.php

```
<?php
Route::group(['prefix' => 'post'], function () {
    Route::get('/new', function () {
        return "/post/new";
    });

    Route::get('/edit/{id}', function ($id) {
        return "/post/edit/$id";
    });

    Route::get('/delete/{id}', function ($id) {
        return "/post/delete/$id";
    });
});
```

Ou seja, ao criar o prefixo todos os roteamentos dentro do `Route::group` estarão vinculados a ele.

Middleware

Um middleware é uma forma de filtrar as requisições que são processadas pelo Laravel. Existem rotas que, por exemplo, só podem ser executadas se o usuário estiver autenticado, ou qualquer outro tipo de exigência necessária. Um middleware é criado na pasta `app/Http/Middleware`, tendo inclusive alguns prontos para uso.

Para entendermos melhor o conceito, vamos focar na autenticação. O Laravel já possui um sistema pronto para autenticar (login) o usuário, e vamos usá-lo aqui para realizar um teste. O middleware de autenticação está localizado em `app/Http/Middleware/Authenticate.php` e você pode utilizá-lo em sua rota através do exemplo a seguir:

blog\app\Http\routes.php

```
<?php
Route::get('/testLogin', ['middleware' => 'auth', function () {
   return "logged!";
}]);
```

Neste código, ao acessarmos `blog.com/testLogin`, a página será redirecionada para a tela de login, ou então será gerado um erro de acesso não autorizado. Estes detalhes podem ser analisados no arquivo `app/Http/Middleware/Authenticate.php`, no método `handle`. A ideia principal do middleware é fornecer um meio de executar ações antes e depois da requisição, para que se possa realizar alguma tarefa específica.

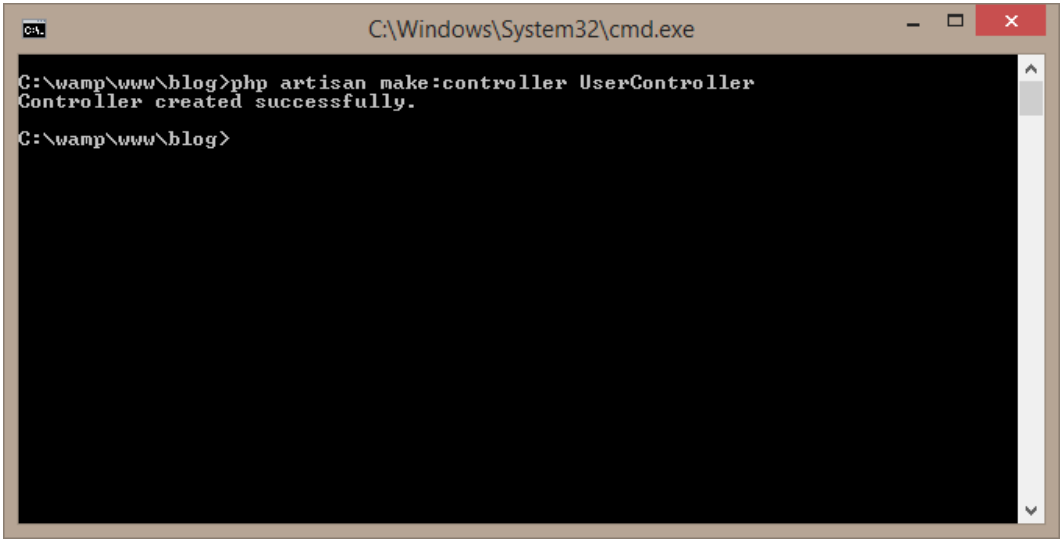
Controllers

Até agora foram abordados diversos conceitos sobre o roteamento no Laravel e sobre como podemos, através da URL de uma aplicação, executar diferentes tarefas. No arquivo `blog\app\Http\routes.php` criamos diversas rotas, mas não será nele que iremos programar todas as funcionalidades do sistema. Seguindo o padrão de desenvolvimento MVC, a maioria das funcionalidades do sistema ficam alocadas no controller e no model da aplicação.

Um controller é um "pedaço" da aplicação que geralmente reflete uma entidade comum. Por exemplo, em um blog temos diversas entidades tais como o usuário, um post e um comentário. Todas estas entidades, que geralmente também são tabelas do sistema, tem os seus próprios controllers. Com isso deixamos o arquivo `routes.php` para configurar rotas menos comuns da aplicação, e usamos os controllers para configurar as rotas que se relacionam com a entidade em questão. Ou seja, as rotas relacionadas com o usuário serão configuradas no controller `User`, já as rotas relacionadas a um comentário serão configuradas no controller `Comment`. Todos os controllers são criados na pasta `app\Http\Controllers`, usam o sufixo "Controller" e estão em inglês.

Criar um controller é relativamente simples, pode-se criar diretamente o arquivo php ou usar a linha de comando, conforme o exemplo a seguir:

```
php artisan make:controller UserController
```



Controllers implícitos (automáticos)

Após executar o comando, o controller `UserController` é criado, com algum código relevante. Mas o que desejamos para o nosso sistema é ligar o roteamento aos métodos do controller. Ou seja, após criarmos o controller `UserController` queremos de alguma forma fazer com que a URL "blog.com/user/new" chame um método qualquer dentro do controller, não havendo mais a necessidade de editar o arquivo `routes.php`. Configurar isso é relativamente fácil através do código `Route::controller`, que deve ser adicionado no arquivo `routes.php`, e deve referenciar a parte da URL que irá conter o redirecionamento e o controller propriamente dito.

No caso do controller `UserController`, temos:

blog\app\Http\routes.php

```
<?php
Route::get('/', function () {
    return "Hello World";
});

Route::controller("user","UserController");
```

Após configurar que a URL que contém `/user` irá chamar os métodos do `UserController`, precisamos apenas criar estes métodos, cujo o padrão é prefixar o tipo de requisição (Get, Post, Put etc) e concatenar com o nome do método. Podemos então alterar a classe `blog\app\Http\Controllers\UserController.php` para:

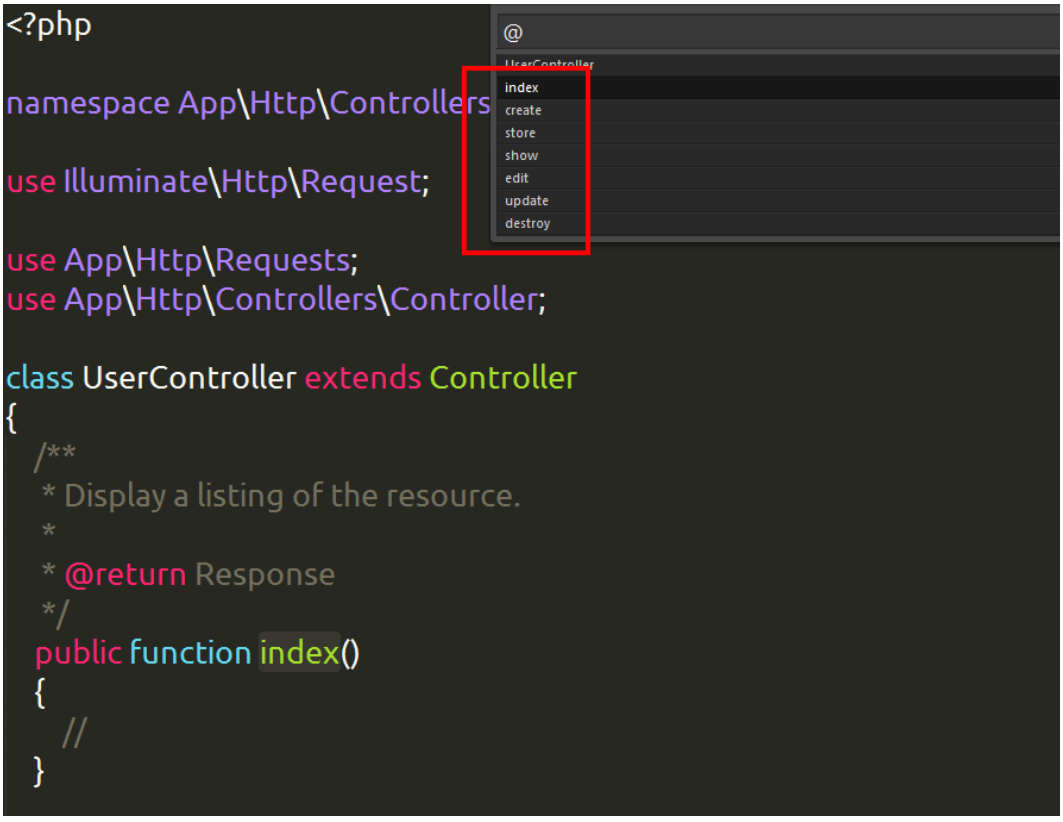
blog\app\Http\Controllers\UserController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class UserController extends Controller
{

    public function getIndex(){
        return "user/";
    }

    // ....

}
```

Isso significa que, ao realizar uma chamada GET com a URL "blog.com/user/show/1", o método `getShow` da classe `UserController` será chamado, atribuindo o parâmetro `$id` do método, e retornando o texto "get user/show/1". Perceba que apenas a configuração do controller está no `routes.php`, e toda a configuração de acesso ao métodos da entidade `User` está na classe `UserController`.

Controllers e Resource

Já vimos como ligar as rotas de uma aplicação a um controller específico, mas podemos avançar um pouco mais na configuração de um controller e criar um resource, que é um conjunto de métodos pré configurados com os seus tipos de requisição pré definidos. Quando criamos o controller através do comando `php artisan make:controller` os seguintes métodos são criados automaticamente:



Crie um resource no arquivo `routes.php`, através do seguinte código:

blog\app\Http\routes.php

```
<?php
Route::get('/', function () {
    return "Hello World";
});

Route::controller("user","UserController");

//Resource:
Route::resource('user', 'UserController');
```

Teremos habilitado a seguinte configuração:

Tipo	Caminho	Ação	Método
GET	/user	index	user.index
GET	/user/create	create	user.create
POST	/user	store	user.store
GET	/user/{id}	show	user.show
GET	/user/{id}/edit	edit	user.edit
PUT	/user/{id}	update	user.update
DELETE	/user/{id}	delete	user.delete

Controller explícitos (manuais)

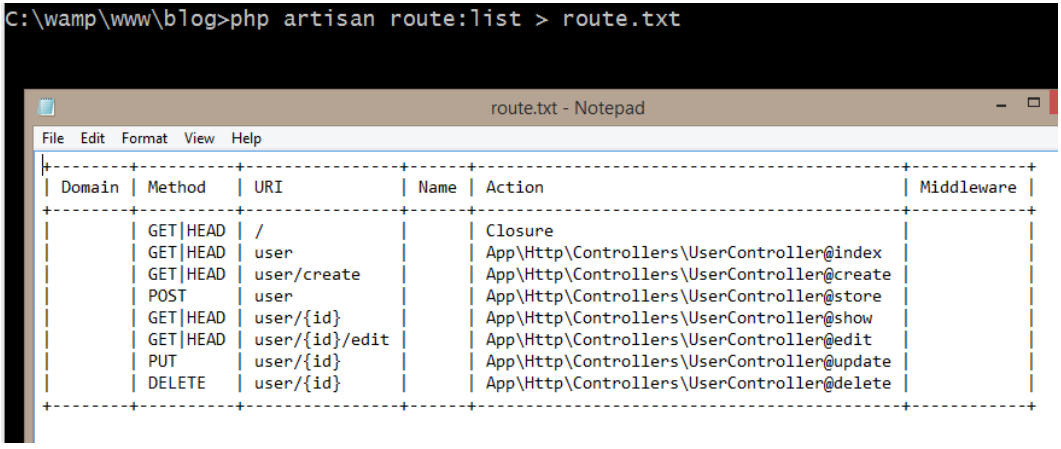
Quando usamos `Route::controller`, definimos um acesso automático da requisição ao controller, bastando apenas prefixar o nome do método do controller. Existe outra forma de configurar o acesso ao controller da aplicação através da criação explícita de cada método, conforme o exemplo a seguir:

blog\app\Http\routes.php

```
<?php
Route::get('/', function () {
    return "Hello World";
});

Route::get('user/', 'UserController@index');
Route::get('user/create', 'UserController@create');
Route::post('user/', 'UserController@store');
Route::get('user/{id}', 'UserController@show');
Route::get('user/{id}/edit', 'UserController@edit');
Route::put('user/{id}', 'UserController@update');
Route::delete('user/{id}', 'UserController@delete');
```

Desta forma, você tem a necessidade de escrever cada rota no arquivo `routes.php`, mas a implementação de cada rota continua no controller. Existem algumas pequenas vantagens em escrever as rotas manualmente. Uma delas é poder usar o comando `php`

`artisan route:list` para obter a lista de rotas de sua aplicação, conforme a figura a seguir.



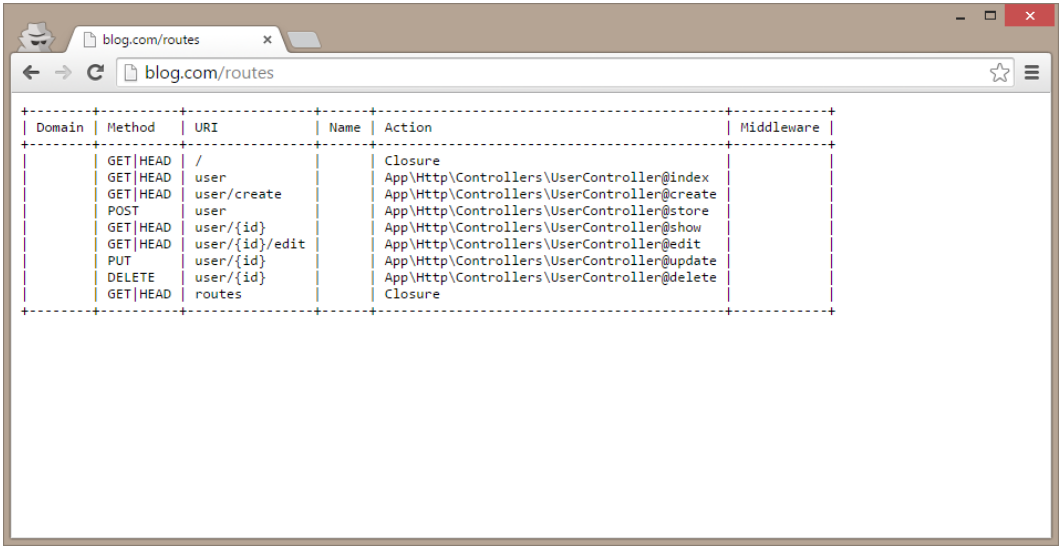
Isso será muito útil para a documentação da API de acesso do servidor. Outra vantagem é ter o controle exato de como a sua aplicação web está sendo exposta através da API, garantindo assim um conforto melhor em momentos de depuração do código.

Pode-se também criar um roteamento para exibir todos os retamentos existentes no próprio navegador, conforme o código a seguir:

blog\app\Http\routes.php

```
<?php

Route::get('routes', function() {
     \Artisan::call('route:list');
     return "<pre>".\Artisan::output();
});
```

Que exibe o seguinte resultado:



Roteamento explícito ou implícito?

Não existe um consenso entre a forma mais correta a ser utilizada. Muitos programadores gostam do modo implícito porque não precisam definir todas as rotas no arquivo `routes.php`, enquanto outros defendem que criar estas rotas manualmente é o mais correto a se fazer. Nesta obra, o foco é criar algo que seja simples e intuitivo, então ao invés de termos diversos métodos no controller que podem ser acessados livremente pelo AngularJS, teremos no arquivo `routes.php` todo o acesso definido, de forma clara e concisa.

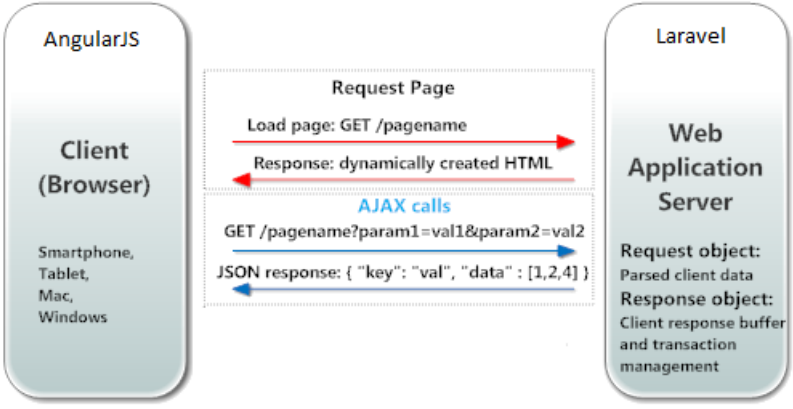
Assim, quando estivermos programando os métodos de acesso ao servidor pelo AngularJS, saberemos que estes métodos estão em um único arquivo (API). Existem outras vantagens ao se manter o acesso explícito, como a formatação dos dados da resposta (JSON ou XML), que serão discutidos em um capítulo posterior.

Comunicação via Ajax

Uma requisição (Request) é o acesso do cliente (navegador) ao servidor. Nesta obra, estaremos abordando um Request como sendo o acesso do AngularJS ao Laravel, via

Ajax. Este processo será amplamente utilizado para se obter dados do servidor, para preencher uma tabela do AngularJS ou simplesmente para persistir dados.

Quando o Laravel responder ao AngularJS, temos a resposta (Response), que obrigatoriamente deverá ser realizada através de uma formatação padronizada, chamada de JSON, que é um formato mais leve que o XML e amplamente usado em requisições Ajax. A imagem a seguir ilustra o processo.



Estamos aqui estabelecendo um padrão de comunicação de dados entre o AngularJS e o Laravel, padrão este utilizado em qualquer comunicação Cliente/Servidor via Ajax. Uma das vantagens desse padrão é que ambos os lados, tanto o servidor quanto o cliente, são independentes no processo, ou seja, se por algum motivo houver uma troca de tecnologia no cliente ou no servidor, o padrão será o mesmo.

Por exemplo, se houver a necessidade de trocar o Laravel por algum framework em Java (perceba que mudamos até a linguagem de programação), basta criar a mesma API com as mesmas respostas em JSON que o cliente em AngularJS continuará o mesmo.

Da mesma forma, se mantermos o Laravel no servidor e usarmos um outro cliente, como o Android em um dispositivo mobile, este pode fazer requisições Ajax ao servidor e obter os mesmos dados do AngularJS.

Respondendo em JSON

Para que o Laravel responda em JSON ao cliente, basta que o método do controller retorne um Array ou um objeto do Eloquent (veremos o Eloquent em um capítulo posterior).

Por exemplo, supondo que o arquivo `routes.php` contenha:

blog\app\Http\routes.php

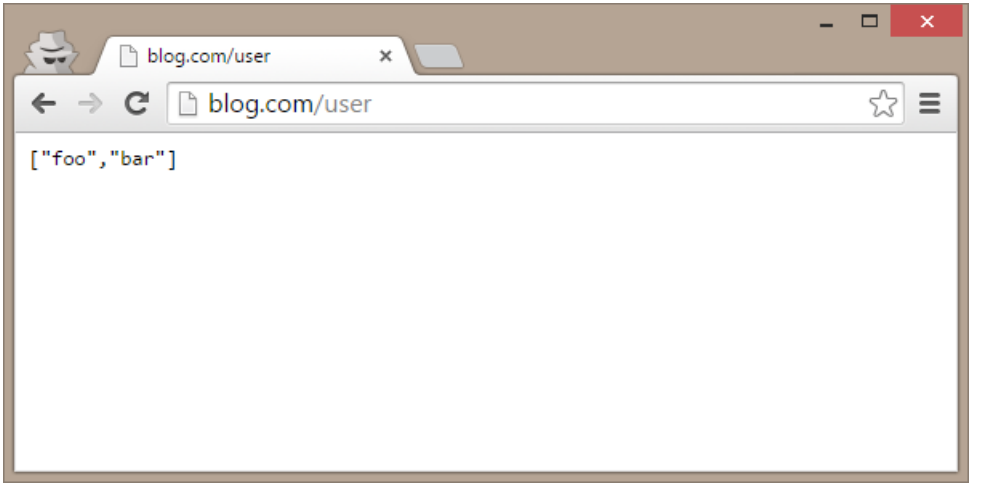
```
<?php

Route::get('user/', 'UserController@index');
```

E o método `index` do `UserController` seja:

blog\app\Http\Controllers\UserController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class UserController extends Controller
{

    public function index(){
        $array = array('foo', 'bar');
        return $array;
    }

    // .......
}
```

Teremos a seguinte reposta no navegador:



Que é o Array no formato JSON. Caso haja a necessidade de retornar um objeto, o ideal é adicionar este objeto a um Array e retorná-lo, como no exemplo a seguir:

blog\app\Http\Controllers\UserController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class UserController extends Controller
{

    public function index(){
        $object = new \stdClass();
        $object->property = 'Here we go';
        return array($object);
```

```
    }

    // .......
}
```

Cuja a resposta será:

Pode-se também utilizar o método `response()->json()`, conforme o exemplo a seguir:

blog\app\Http\Controllers\UserController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class UserController extends Controller
```

```
{

    public function index(){
        $object = new \stdClass();
        $object->property = 'Here we go';
        return response()->json($object);
    }

    // .......
}
```

Cujo resultado é exibido na figura a seguir.

Por convenção, podemos definir que o Controller do Laravel sempre retorne um array ou um objeto do Eloquent, sempre no formato JSON.

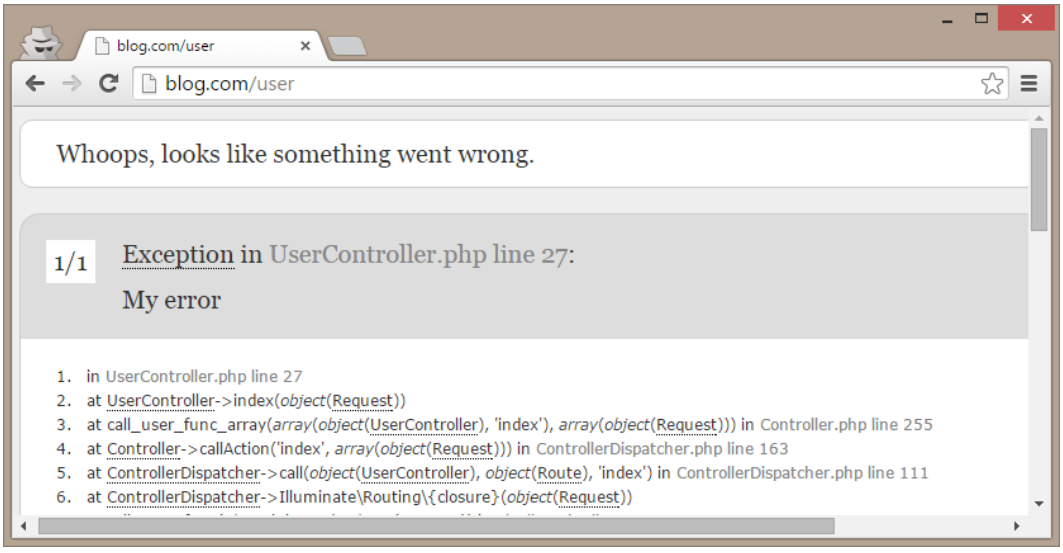
Exceções no formato JSON

É vital que as exceções no Laravel retornem no mesmo formato JSON que estamos adotando como padrão. No exemplo a seguir, ao criar uma exceção genérica:

blog\app\Http\Controllers\UserController.php

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class UserController extends Controller
{

    public function index(){
        throw new \Exception("My error");
        return array("ok");
    }

    // .......
}
```

Temos a seguinte resposta:



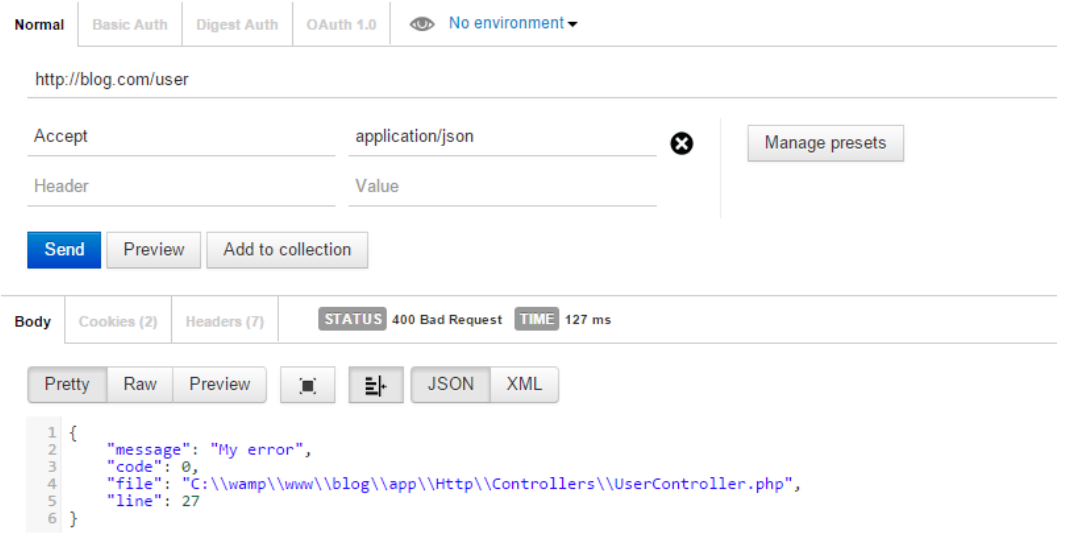
O que, definitivamente, não é válido para o padrão JSON. Temos que, de alguma forma, retornar este erro em JSON. Felizmente isso é perfeitamente possível com o Laravel, bastando apenas alterar o arquivo `app\Exceptions\Handler.php` conforme o código a seguir:

blog\app\Exceptions\Handler.php

```
<?php

namespace App\Exceptions;

use Exception;
use Symfony\Component\HttpKernel\Exception\HttpException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
    protected $dontReport = [
        HttpException::class,
    ];

    public function report(Exception $e)
```

```
    {
        return parent::report($e);
    }

    public function render($request, Exception $e)
    {
        if ($request->wantsJson()){
            $error = new \stdclass();
            $error->message = $e->getMessage();
            $error->code = $e->getCode();
            $error->file = $e->getFile();
            $error->line = $e->getLine();
            return response()->json($error, 400);
        }
        return parent::render($request, $e);
    }
}
```

Neste código, alteramos o método `render` incluindo um erro personalizado no formato JSON. Este erro somente será gerado em requisições Ajax, graças a verificação `if ($request->wantsJson())`. Isso é útil para definir como a mensagem será apresentada ao usuário. Para podermos testar este erro, devemos simular um acesso ajax ao Laravel, e isso pode ser realizado através de uma extensão do *Google Chrome* chamada *Postman*, adicionando a URL `blog.com/user` ao cabeçalho na requisição GET, conforme a figura a seguir:



Resumindo, o erro será exibido conforme a requisição, e visto que o AngularJS sempre realizará uma requisição Ajax ao Laravel, então o erro será exibido no formato JSON.