

PHP::Construir um sistema de Rotas para MVC – Segunda parte

A [primeira parte desta série](#) de artigos, começamos a contruir a estrutura de diretórios e algumas classes responsáveis pelo processo de roteamento. O sistema ainda não funcionava porque apenas estávamos começamos a desenvolver a estrutura e a lógica.

Se você já conhece meus artigos, sabe que constumo detalhar o funcionamento da estrutura e por isso, não é possível resumir este tema em apenas um artigo sem omitir informações importantes. Portanto, aproveite bem essa série de artigos e desenvolva os seus programas. Se tiver dúvidas, leia os artigos mais de uma vez e tente compreender a idéia. Aqui nenhum padrão está sendo imposto, mas uma forma bem satisfatória que poderá ser utilizada em seus projetos e até melhorada!

URL amigáveis

Se você não se lembra o que são URL amigáveis, dê uma olhada no [artigo anterior](#), pois fiz uma breve comparação com o método convencional e apresentei uma breve explicação que não demora nada em lhe convencer sobre o motivo de escolher esta forma.

Se você já tem certeza de que o servidor do seu ambiente de desenvolvimento está configurado, pode pular estas etapas seguintes.

Configurando servidor Web (Se você usar Linux Debian/Ubuntu)

Para conseguirmos configurar as url amigáveis, precisamos fazer algumas configurações no servidor web. Se você estiver usando Apache no Linux, e esta foi uma configuração feita no Ubuntu, poderia primeiro verificar se o módulo rewrite está ativo. Siga até o terminal e vá até o diretório do apache, neste caminho:

```
cd /etc/apache2/mods-enabled
```

Veja se há um arquivo “rewrite.load” e se existir, não precisa fazer nada nesta etapa, mas se não existir, digite este comando:

```
sudo a2enmod rewrite
```

O módulo estando ativado, será necessário alterar um arquivo de configuração do apache. Você poderá editá-lo com o nano, dessa maneira:

```
sudo nano /etc/apache2/apache2.conf
```

Após digitar sua senha para acesso como sudo, encontre um código como esse:

```
<Directory /var/www/>  
    Options Indexes FollowSymLinks  
    AllowOverride None  
    Require all granted  
</Directory>
```

Modifique “AllowOverride” de “None” para “All” e deverá ter aspecto semelhante a este:

```
<Directory /var/www/>  
    Options Indexes FollowSymLinks  
    AllowOverride All  
    Require all granted  
</Directory>
```

Então reinicie o servidor Apache utilizando este comando:

```
sudo /etc/init.d/apache2 restart
```

Configurando servidor Web (Se você usar Windows WAMP)

Você pode simplesmente:

- Clicar em wampmanager
- Seguir até Apache e http.conf
- Se não estiver ticado/marcado, marque

Mas se você prefere fazer manualmente:

Você precisa encontrar o arquivo httpd.conf que fica armazenado diretório de instalação do Apache:

C:\wamp\bin\Apache\conf\httpd.conf

Edite este arquivo e procure pela linha que está com essa configuração:

```
#LoadModule rewrite_module modules/mod_rewrite.so
```

Retire o cerquilha “#” para ficar dessa maneira

```
LoadModule rewrite_module modules/mod_rewrite.so
```

Reinicie o apache e estará pronto!

Criando nossos arquivos .htaccess

Você deve se lembrar que deixamos arquivos “ocos”, prontos para receberem dados e agora chegou a vez dos arquivos .htaccess.

Vamos primeiro alterar o que está dentro do diretório public. Antes, vamos relembrar o que precisamos:

Em um método convencional, vários scripts php são portas de entrada para seu aplicativo web, mas não é isso que esperamos. Queremos que apenas o script index.php seja a porta de entrada para o nosso aplicativo.

O que você precisa saber é que quando digita um endereço no browser por exemplo:

`http://www.meusite.com.br/`

`http://www.meusite.com.br/contatos/1/editar`

Temos aí o que chamados de URI (Uniform Resource Identifier) ou identificador uniforme de recurso. Esta URI é uma cadeia de caracteres para identificar ou denominar um recurso na Internet. Você pode obter mais detalhes neste Link: <https://pt.wikipedia.org/wiki/URI>

Certo mas, e daí?

Você sempre aprendeu, ou já ouviu falar que o nome disse é URL, certo? E você tem razão! Uma URI pode ser classificado como uma URL. Esta URI, identifica o local exato de um recurso, mas em nosso servidor Web, pode não existir fisicamente um local:

`http://www.meusite.com.br/contatos/1/editar`

Seria assim:

- “/” – Raiz
 - contatos/
 - 1/
 - editar

Neste caso o raiz seria exatamente em:

<http://www.meusite.com.br/> <----- Aqui

Talvez as configurações apontem o script index.php para:

<http://www.meusite.com.br/index.php>

Mas porque index.php não aparece ? e o restante? /contatos/1/editar

Você deve saber que quando temos um arquivo do tipo index.php, representa um script que indexa nosso site a partir daquele diretório que é encontrado. Como o servidor quase sempre espera encontrar um arquivo index.php, ou index.html, etc, não vamos precisar declarar seu nome. Se ele existir, o servidor imediatamente vai abri-lo.

Certo Alexandre, suponha que seja assim:

<http://www.meusite.com.br/index.php/contatos/1/editar>

e o restante? /contatos/1/editar

Nosso servidor receberá uma configuração para entender que deve ignorar qualquer diretório ou script diferente. Para ele só valerá index.php. Então, significa que estas strings adicionais não serão mais vistas como um local físico, e sim uma espécie de informação extra e que pode ser armazenada em uma variável e coletada pelo script index.php para ser tratada.

Daí podemos inventar alguma coisa com estes dados! 😊

Este local:

`http://www.meusite.com.br/`

no exemplo que estamos tratando nesses artigos, nossa configuração passa a ser mapeado para o diretório raiz, na verdade no diretório public, onde está o script `index.php`.

A partir daí, se quisermos utilizar subpastas, nós mesmos é que teremos de tratar isso e não mais o servidor. Por isso podemos mapear combinações de dados que chegarem ali para recursos em nossas aplicações, direcionar para classes, controllers, e daí: surgem as rotas!

Elas vão mapear estas combinações de scripts para recursos. Dessa forma, criamos um meio inteligente o suficiente para casar estes dados vindo na URI com padrões tais como `/^contatos/[0-9]{0,}/editar$/`. Se casou, é porque temos um recurso mapeado, se não encontrou nada, poderemos responder com um 404.

Agora acredito que arranquei de você aquele: ah!

Então vamos começar a configurar o `.htaccess`

Nesse momento vou apresentando linha a linha e explicando logo abaixo. Se você já conhece estas configurações, então pode passar adiante, mas se não conhece, sugiro procurar entender bem a idéia.

```
<IfModule mod_rewrite.c>
```

Queremos que seja verificado pelo servidor se o `mod_rewrite` está ativo, e se estive, execute os comandos seguintes. (Lembra do `mod_rewrite.so` lá em cima? é esse mesmo) E o comando é todo junto assim mesmo, ali não errei na digitação! rs

RewriteEngine On

Então ative o mod_rewrite

RewriteBase /rotas/public

Esta linha diz onde é o diretório base, onde fica o script index.php. Talvez em seu ambiente de desenvolvimento, você vai utilizar um endereço no browser assim <http://localhost/rotas/public>. Note que neste caso você criou um subdiretório para este projeto, e deve indicar o diretório base dele desde o raiz até o public. Então, se for <http://localhost/projetos/joao/maria/familia/rotas/public> o RewriteBase ali deve ser /projetos/joao/maria/familia/rotas/public, certo?

RewriteCond %{REQUEST_FILENAME} !-f

Aqui não deixa entrar em outro script, se receber um outro nome tipo tentativa.php, vai ignorar e não vai abri-lo

RewriteCond %{REQUEST_FILENAME} !-d

Semelhante a idéia anterior, não deixa entrar em outro diretório, se receber uma instrução outro nome tipo contato/1/edit, também não vai sair daqui, vai ignora e continua no mesmo diretório public

RewriteRule ^(.+)\$ index.php?uri=\$1 [QSA,L]

Aqui será o palco do acontecimento! A reestruturação da URI, onde definimos como queremos coletar as informações, será aqui! Veja que temos uma expressão regular. Se você não sabe bem como funciona, eis um básico para entender aqueles metacaracteres:

- o ^ – Circunflexo indica o início de tudo que ele coletar na string, ou seja, tudo DEPOIS de <http://www.meusite.com.br/>
- o () – Os parênteses determinam um grupo, tudo que estiver aqui dentro será testado para ver se casa com a informação
- o .* – Este ponto e asterisco significa “qualquer coisa”
- o \$ – O cifrão/dólar é o fim da linha ou do que coletar, então desde ^ até \$ teríamos **/contatos/1/editar** para uma URI assim <http://www.meusite.com.br/contatos/1/editar>

Alexandre ficou muito inseguro, como podemos melhorar isso? Embora não seja o nosso foco aqui, mas segue para aguçar o apetite!

RewriteRule ^([A-Za-z0-9_\\W-]+)\$ index.php?uri=\$1 [QSA,L]

Agora estamos usando uma lista representado pelos caracteres []. Somente os caracteres que estão nesta lista serão válidos. Note que a barra e a contra barra, bem como o sinal de subtração foram escapados com uma barra “\”. Aquele “+”, significa um ou mais caracteres daquela lista. Se você realmente gostou de expressões regulares e não tem pouco ou nenhum conhecimento, ou mesmo quer melhorar, recomendo o livro “Expressões Regulares – Uma abordagem divertida” de Aurélio Marinho Jargas publicado pela Novatec. Este sem dúvida é um excelente livro para todos que querem de fato compreender bem expressões regulares.

QSA – Qualquer cadeia de caracteres passado com a URI original, deve ser reescrita para \$1 de **index.php?=\$1**

Significa que tudo que casar com aquele padrão, será reescrito para `index.php?uri=$i`, onde a cadeia de strings será lançada naquela variável uri. Sendo assim para <http://www.meusite.com.br/contatos/1/editar> teremos <http://www.meusite.com.br/index.php?uri=contatos/1/editar>

L – Se a regra casar com a cadeia de caracteres, não processar mais qualquer RewriteRules abaixo desta.

Ufa! Espero ter conseguido ser meio termo entre objetivo mas também esclarecedor o suficiente! O arquivo .htaccess ficará assim:

```
<IfModule mod_rewrite.c>
RewriteEngine On
RewriteBase /rotas/public
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^([A-Za-z0-9\\-]+)$ index.php?url=$1 [QSA,L]
```

Tudo bem, mas e o outro .htaccess que está no diretório raiz do projeto? Adicione apenas essa linha:

```
Options -Indexes
```

Isso vai desabilitar a indexação de diretórios do site inteiro e vai impedir acesso indevido ao diretório antes acima de public.

Vamos fazer um teste se está funcionando?

Vamos abrir o arquivo index.php em public e adicionar uma linha para teste:

```
echo $_GET['uri'] ?? '/' ;
```

Feito isso, abra o navegador e tente ir entrando com valores diferentes na URI. Se não funcionar, leia o seu código e confira com o do artigo para ver o que está faltando.

Percebeu que agora estamos com a faca e o queijo na mão?

Classe Request

Vamos criar a nossa classe para administrar solicitações, ela será bem básica para nosso propósito. Siga para diretório src e edite a classe Request.php. Talvez será um arquivo vazio, então adicione o seguinte código:

```
1  <?php
2
3  namespace Src;
4
5
6  class Request
7  {
8
9      protected $files;
10     protected $base;
11     protected $uri;
12     protected $method;
13     protected $protocol;
14     protected $data = [];
15
16     public function __construct()
17     {
18         $this->base = $_SERVER['REQUEST_URI'];
19         $this->uri = $_REQUEST['uri'] ?? '/';
20         $this->method = strtolower($_SERVER['REQUEST_METHOD']);
21         $this->protocol = isset($_SERVER["HTTPS"]) ? 'https' : 'http';
22         $this->setData();
23
24         if(count($_FILES) > 0) {
25             $this->setFiles();
26         }
27     }
28
29 }
```

```
30 protected function setData()
31 {
32     switch($this->method)
33     {
34         case 'post':
35             $this->data = $_POST;
36             break;
37         case 'get':
38             $this->data = $_GET;
39             break;
40         case 'head':
41         case 'put':
42         case 'delete':
43         case 'options':
44             parse_str(file_get_contents('php://input'), $this->data);
45     }
46 }
47
48 protected function setFiles() {
49     foreach ($_FILES as $key => $value) {
50         $this->files[$key] = $value;
51     }
52 }
53
54 public function base()
55 {
56     return $this->base;
57 }
58
59 public function uri(){
60     return $this->uri;
61 }
62
63 public function method(){
64
65     return $this->method;
66 }
67
68 public function all()
69 {
70     return $this->data();
71 }
```

```

72
73     public function __isset($key)
74     {
75         return isset($this->data[$key]);
76     }
77
78     public function __get($key)
79     {
80         if(isset($this->data[$key]))
81         {
82             return $this->data[$key];
83         }
84     }
85
86     public function hasFile($key) {
87
88         return isset($this->files[$key]);
89     }
90
91     public function file($key){
92
93         if(isset($this->files[$key]))
94         {
95             return $this->files[$key];
96         }
97     }
98 }
99
100

```

Gostou da criança?

Esta classe prove um meio orientado a objetos de manipularmos a requisição. Isso torna nosso código limpo e profissional, nos dando métodos necessários para tratar qualquer solicitação recebida.

Esta implementação é simples, mas ela atende nosso propósito aqui. Os métodos são quase auto explicativos, mas vamos entender o que estamos coletado:

- BASE – é a URI original
- URI – é a forma que precisamos para trabalhar

- METHOD – captura se é 'post', 'get', etc (Lembrando que métodos além de post e get não podem ser testados via solicitação padrão do browser, precisamos de um aplicativo que se testa API, exemplo Postman)
- PROTOCOL – é um plus! Talvez não utilizemos aqui, mas apenas identifica se o protocolo que estamos usando é HTTPS/HTTP (Se você não gostar dele, pode até ignorar essa implementação)
- DATA – coleta os dados enviados
- FILES – coleta arquivos enviados

Depois disso, temos uma série de métodos para obter os dados, sejam eles dados enviados na requisição, como procedente de tags via post, get, arquivos de tags do tipo file, etc.

Não esqueça de a cada classe criada, rodar o comando no terminal

```
composer dump-autoload
```

Se tudo deu certo, você pode fazer um breve teste da classe Request, mudando o index.php para esse:

```
1 <?php
2
3 require __DIR__ . '/../bootstrap.php';
4
5 $request = new Src\Request;
6
7 echo $request->uri();
8
9
```

Se você digitou tudo certo, o resultado deverá ser o mesmo, mas agora pela classe Request. Precisa tudo isso Alexandre para criar um sistema de rotas? Não, não precisa. Mas o fato é que aqui eu quero te deixar na cara do gol para você implementar naquele framework de brinquedo que você tem criado para entender o MVC. Por isso, esse projeto destes artigos começam a se parecer com um framework.

Para então fazer nosso sistema começar a funcionar, vamos criar a classe Dispatcher. Então abra para edição o arquivo Dispatcher.php. Temos que entender o que o Dispatcher faz: Apenas despachar uma solicitação, ele é quem sabe como invocar o controller. Poderíamos fazer isso direto em Router? Sim, mas não é papel dele! Ele precisa rotear, os dados para aplicação, e passar para quem sabe despachar. Por isso, o Router consulta sua tabela de rotas e

ao encontrar uma que casar com a solicitação repassa o pedido para o despachante que invocará o método correto.

```
1  <?php
2
3  namespace Src;
4
5
6  class Dispatcher
7  {
8
9      public function dispatch($callback, $params = [], $namespace = "App\\")
10     {
11         if(is_callable($callback))
12         {
13             return call_user_func_array($callback, array_values($params));
14         } elseif (is_string($callback)) {
15             if(!!strpos($callback, '@') !== false) {
16
17                 $callback = explode('@', $callback);
18                 $controller = $namespace.$callback[0];
19                 $method = $callback[1];
20
21                 $rc = new \ReflectionClass($controller);
22
23                 if($rc->isInstantiable() && $rc->hasMethod($method))
24                 {
25                     return call_user_func_array(array(new $controller, $method), array_values($params));
26                 } else {
27                     throw new \Exception("Erro ao despachar: controller não pode ser instanciado, ou método nã
28                 }
29             }
30         }
31         throw new \Exception("Erro ao despachar: método não implementado");
32     }
33 }
34
35
36
37
38
```

Note que o Dispatcher só tem um método, `dispatch`. O método é inteligente o suficiente para distinguir se o callback é de fato apenas um callback, ou será um controller que precisa ser instanciado. Se for, ele fará testes para além de saber se é um objeto válido, também saber se têm o método. Para fazer todo esse trabalho, será necessário apoio da classe `ReflectionClass`, que extrai informações de uma classe fornecida como argumento.

Muito bem, Dispatcher está preparada! Agora vamos voltar a Router e adicionar um método para que ela saiba lidar com Dispatcher. Então abra o arquivo Router para edição e adicione este método no final:

```
1  protected function dispatch($route, $namespace = "App\\"){
2
3      return $this->dispatcher->dispatch($route->callback, $route->uri, $namespace);
4  }
5
6
7
```

Router agora tem um método para invocar o Dispatcher, bem como tem um método para procurar pela rota. Porém não existe ainda um método que dá o comando para que essas coisas funcionem. Então, vamos adicionar estes dois métodos:

```
1  protected function notFound()
2  {
3      return header("HTTP/1.0 404 Not Found", true, 404);
4  }
5
6
7  public function resolve($request){
8
9      $route = $this->find($request->method(), $request->uri());
10
11     if($route)
12     {
13         return $this->dispatch($route);
14     }
15     return $this->notFound();
16 }
17
18
19
20
```

Router agora tem um método que pode ser invocado para tratar as requisições! vamos testar então!

Salve a classe Router e outros que você ainda não tenha salvo, e abra novamente o arquivo index.php para edição. Vamos alterar agora para este formato:

```
1  <?php
2
3  require __DIR__ . '/../bootstrap.php';
4
5  $request = new Src\Request;
6
7  $router->resolve($request);
8
9
10
```

Se você testou e apareceu página inicial, aposto que veio o sorriso!

Sim! É isso mesmo! Nosso sistema de rotas já está funcionando! Mas por enquanto apenas com closures. No final do [artigo anterior](#), este era o código de teste:

```
1  <?php
2
3  $router->get('/', function(){
4      echo "Página inicial";
5  });
6
7  $router->get('/contatos', function(){
8      echo "Página de contatos";
9  });
10
11 $router->post('/contatos/store', "Controller@store");
12
13
```

Ignore por enquanto o último tipo, mas crie novas rotas get e faça o teste! Por enquanto vamos ficando por aqui! Mas no último artigo dessa série, iremos construir o controller e fazer outros tipos de testes mais avançados com nosso sistema de roteamento. Não deixe de acompanhar, pois a cereja do bolo ainda está por vir!