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

Este é o artigo final sobre sitema de rotas para aplicativos web MVC. A primeira parte desta série de artigos, inicia a contruição da estrutura de diretórios e algumas classes responsáveis pelo processo de roteamento. O sistema ainda não funcionava porque apenas era o começo depois, foi apresentado as lógica a ponto de deixá-lo com uma certa funcionalidade já no segundo artigo desta série

O projeto também está disponível no GitHUB.

Aproveite esta grande final! Mas se você ainda não leu os artigos anteriores, recomendo que o faça antes de avançar neste, pois aqui é a conclusão, e não costumo repetir a amostragem do código a mesmo que seja necessário.

Estes artigos, ficaram longos, mas nem sempre ainda permitem muitos detalhes. Desejei adicionar muitos outros detalhes, mas precisei resumir os artigos, para torná-los menos cansativos. Adianto que não é possível detalhar muito sobre cada método ou o que cada linha em cada método está fazendo. Se fizesse isso, seria necessário mais centenas de linhas.

Mas há um esforço sincero em explicar o máximo sobre cada classe e o que seus métodos estão fazendo.

Começando por revisar a classe RouteCollection

A classe RouteCollection cria um objeto de coleções de rotas que serão adicionadas quando são declaradas no arquivo route.php. Acontece que RouteCollection está atendendo bem o propósito, mas é necessário se extrair informações, por exemplo: quais são os espaços mapeados para variáveis? Neste caso, seria interessante contar com um objeto containner, com propriedades que representam os espaços na rota. Então, quando o objeto roteador encontrar a rota na coleção que casar com a URI, deverá entregar tudo para Dispacher se virar e mandar para o controller correto ou injetar dentro da callback/clojure.

Os primeiros métodos a serem adicionados, servem para processar o padrão definido na rota, coletando supostas informações de variáveis nos espaços coringas ou com meta-caracteres. Dessa forma, é possível construir rotas que mapeiam de forma perfeita as variáveis na declaração de uma URI para um controller.

O primeiro método a ser construído strposarray(), apenas dará suporte ao segundo método toMap(). É uma espécie de strpos() que se baseia em um array para encontrar a posição na string. Na internt você encontra algumas outras implementações parecidas com essa, mas aqui segue mais uma, e que atende muito bem o objetivo.

```php
protected function strposarray(string $haystack, array $needles, int $offset = 0)
{
    $result = false;
    if(strlen($haystack) > 0 && count($needles) > 0)
    {
        foreach($needles as $element){
            $result = strpos($haystack, $element, $offset);
            if($result !== false)
            {
                break;
            }
        }
    }
    return $result;
}


protected function toMap($pattern)
{

    $result = [];

    $needles = ['{', '[', '(', "\\"];

    $pattern = array_filter(explode('/', $pattern));

    foreach($pattern as $key => $element)
    {
        $found = $this->strposarray($element, $needles);

        if($found !== false)
        {
            if(substr($element, 0, 1) === '{')
            {
                $result[preg_filter('/([\{\}])/', '', $element)] = $key - 1;
            } else {
                $index = 'value_' . !empty($result) ? count($result) + 1 : 1;
```

```
                    array_merge($result, [$index => $key - 1]);
                }
            }
        }
        return count($result) > 0 ? $result : false;
    }
```

O segundo método toMap(), cria devolve um array com propriedades que mapeiam as posições dos coringas e meta-caracteres encontrados na rota. Por exemplo:

```
$router->get('teste/{id}', 'Controller@method');
```

Neste caso, coleta um valor que de {id} para uma propriedade id, mas suponhamos que você tenha criado uma rota assim:

```
$router->get('teste/([0-9]+)', 'Controller@method');
```

Então, seria coleta um valor para propriedade value_1.

```
$router->get('teste/([0-9]+)/edit/([A-Za-z]+)', 'Controller@method');

$router->get('teste2/{value_1}/edit/{value-2}', 'Controller@method');
```

Mais tarde, o Dispacher encaminha os valores coletados nas posições para o Controller.

Agora, é preciso preparar a classe RouteCollection para receber mais informações sobre a rota: um nome, namespace, etc. Caso queira que seu sistema de rotas ainda aceite mais parâmetros, deve-se adicionar mais itens por aqui. Todavia, será preciso implementá-los no Dispacher. Por hora, utilizando estes, adione o método parsePattern, responsável por entender o padrão quando chegar em forma de array:

```php
protected function parsePattern(array $pattern)
{
    // Define the pattern
    $result['set'] = $pattern['set'] ?? null;
    // Allows route name settings
    $result['as'] = $pattern['as'] ?? null;
    // Allows new namespace definition for Controllers
    $result['namespace'] = $pattern['namespace'] ?? null;
    return $result;
}
```

Esse método parsePattern() vai adicionar uma funcionalidade incrível, semelhante ao dos grandes frameworks. Você poderá dar nome para rotas e estes poderão ser referenciados para tradução automática em views, controllers, etc. Como dito, poderá inclusive redefinir o namespace do controller alvo! A sujestão é incluir esse método logo abaixo do método definePattern.

Agora, faça uma alteração em cada um dos métodos “add”. Entendo, que muitas vezes será chato, corrigir um método ou uma classe, porém aqui é preciso para atender perfeitamente o projeto. E porque já não fora feito antes? Porque não faria sentido naquele momento, seria confuso, e talvez aqui ainda não esteja bem claro. Então vamos lá, primeiro olhe os métodos atuais:

```php
protected function addPost($pattern, $callback){

    $this->routes_post[$this->definePattern($pattern)] = $callback;
    return $this;

}

protected function addGet($pattern, $callback){

    $this->routes_get[$this->definePattern($pattern)] = $callback;
    return $this;

}

protected function addPut($pattern, $callback){

    $this->routes_put[$this->definePattern($pattern)] = $callback;
    return $this;

}

protected function addDelete($pattern, $callback){

    $this->routes_delete[$this->definePattern($pattern)] = $callback;
    return $this;

}
```

Terão de ser atualizados para ficarem assim (Note que estes métodos cresceram muito! Atenção aos detalhes):

```php
protected function addPost($pattern, $callback){

    if(is_array($pattern)) {

        $settings = $this->parsePattern($pattern);

        $pattern = $settings['set'];
    } else {
```

```
9
10          $settings = [];
11      }
12
13      $values = $this->toMap($pattern);
14
15      $this->routes_post[$this->definePattern($pattern)] = ['callback' => $callback,
16                                                            'values' => $values,
17                                                            'namespace' => $settings['namespace'] ?? null];
18      if(isset($settings['as']))
19      {
20          $this->route_names[$settings['as']] = $pattern;
21      }
22      return $this;
23
24  }
25
26  protected function addGet($pattern, $callback){
27
28      if(is_array($pattern)) {
29
30          $settings = $this->parsePattern($pattern);
31
32          $pattern = $settings['set'];
33      } else {
34
35          $settings = [];
36      }
37
38      $values = $this->toMap($pattern);
39
40      $this->routes_get[$this->definePattern($pattern)] = ['callback' => $callback,
41                                                           'values' => $values,
42                                                           'namespace' => $settings['namespace'] ?? null];
43
44      if(isset($settings['as']))
45      {
46          $this->route_names[$settings['as']] = $pattern;
47      }
48      return $this;
49
50  }
```

```php

protected function addPut($pattern, $callback){

    if(is_array($pattern)) {

        $settings = $this->parsePattern($pattern);

        $pattern = $settings['set'];
    } else {

        $settings = [];
    }

    $values = $this->toMap($pattern);

    $this->routes_put[$this->definePattern($pattern)] = ['callback' => $callback,
                                                         'values' => $values,
                                                         'namespace' => $settings['namespace'] ?? null];
    if(isset($settings['as']))
    {
        $this->route_names[$settings['as']] = $pattern;
    }
    return $this;

}

protected function addDelete($pattern, $callback){

    if(is_array($pattern)) {

        $settings = $this->parsePattern($pattern);

        $pattern = $settings['set'];
    } else {

        $settings = [];
    }

    $values = $this->toMap($pattern);

    $this->routes_delete[$this->definePattern($pattern)] = ['callback' => $callback,
                                                            'values' => $values,
```

```
                                                    'namespace' => $settings['namespace'] ?? null];
        if(isset($settings['as']))
        {
            $this->route_names[$settings['as']] = $pattern;
        }
        return $this;

    }
```

Reconheça: Não foi tão difícil assim, foi?! Mas há mais um detalhe: é preciso voltar até as primeiras linhas da classe RouteCollection e adicionar mais uma propriedade array. Olhe abaixo o código (O restante das linhas foram propositalmente omitidas). Note que há mais uma propriedade protegida: route_names

```
<?php

namespace Src;

class RouteCollection
{
    protected $routes_post = [];
    protected $routes_get = [];
    protected $routes_put = [];
    protected $routes_delete = [];
    protected $route_names = [];  // Esta propriedade deve ser adicionada agora


...

}
```

Mas não faça testes, porque neste momento um objeto Dispacher não entende mais o callback da rota. Por isso, é preciso também alterar a classe Dispacher para aceitar o novo padrão.

Também precisamos há outra uma pequena alteração no método definePattern(). Ele está assim até este momento:

```php
protected function definePattern($pattern) {

    $pattern = implode('/', array_filter(explode('/', $pattern)));
    return '/^' . str_replace('/', '\/', $pattern) . '$/';

}
```

Altere para ficar dessa forma:

```php
protected function definePattern($pattern) {

    $pattern = implode('/', array_filter(explode('/', $pattern)));
    $pattern = '/^' . str_replace('/', '\/', $pattern) . '$/';

    if (preg_match("/\{[A-Za-z0-9\_\-]{1,}\}/", $pattern)) {
        $pattern = preg_replace("/\{[A-Za-z0-9\_\-]{1,}\}/", "[A-Za-z0-9]{1,}", $pattern);
    }

    return $pattern;

}
```

Falta adicionar outro método para devolver o padrão mapeado para um nome. Este método verifica os nomes mapeados para padrões e adicionados na propriedade $route_names. Se a referencia for encontrada, devolve o padrão. Este é o código:

```php
public function isThereAnyHow($name)
{
    return $this->route_names[$name] ?? false;
}
```

Além disso, o Router precisa de um meio que faça a converção do padrão para URI, este método também será inserido em RouteCollection, porque será provido como suporte. Este é o método convert. O método Convert entende o padrão recebido, recebe os parâmetros e converte tudo para um formato de URI contendo os valores no lugar dos espaços de meta-caracteres ou coringas.

```
public function convert($pattern, $params)
{
    if(!is_array($params))
    {
        $params = array($params);
    }

    $positions = $this->toMap($pattern);
    if($positions === false)
    {
        $positions = [];
    }
    $pattern = array_filter(explode('/', $pattern));

    if(count($positions) < count($pattern))
    {
        $uri = [];
        foreach($pattern as $key => $element)
        {
            if(in_array($key - 1, $positions))
            {
                $uri[] = array_shift($params);
            } else {
                $uri[] = $element;
            }
        }
        return implode('/', array_filter($uri));

    }
    return false;

}
```

Agora RouteCollection está concluída e pronta para atender Router, armazenando e servindo todas informações necessária no tempo certo. Logo mais, estes métodos novos farão sentido.

A classe Dispacher, como você já sabe, gera o objeto despachante, reponsável por invocar o controller alvo, injetando os parâmetros recebido na requisição. Essa é a idéia, mas não estava funcionando até agora. Note que um dos argumentos que o método dispach de Dispacher recebe, são os parâmetros em forma de array. A classe RouteCollection já está guardando as posições no padrão, onde se referem a valores. Estes foram mapeados como coringas ou meta-caracteres. Mas, um objeto Router precisa informar corretamente estes parâmetros, e isso não tem acontecido também.

Primeiro, é preciso fazer o Dispacher funcionar de novo. Veja o código como está atualmente, o método dispach:

```php
public function dispach($callback, $params = [], $namespace = "App\\")
{
    if(is_callable($callback))
    {
        return call_user_func_array($callback, array_values($params));

    } elseif (is_string($callback)) {

        if(!!strpos($callback, '@') !== false) {

            $callback = explode('@', $callback);
            $controller = $namespace.$callback[0];
            $method = $callback[1];

            $rc = new \ReflectionClass($controller);

            if($rc->isInstantiable() && $rc->hasMethod($method))
            {
                return call_user_func_array(array(new $controller, $method), array_values($params));

            } else {

                throw new \Exception("Erro ao despachar: controller não pode ser instanciado, ou método não ex
            }
        }
    }
    throw new \Exception("Erro ao despachar: método não implementado");
}
```

A correção para ele é esta (Preste bem atenção! Aqui é fácil de errar):

```php
public function dispach($callback, $params = [], $namespace = "App\\")
{
    if(is_callable($callback['callback']))
    {
        return call_user_func_array($callback['callback'], array_values($params));

    } elseif (is_string($callback['callback'])) {

        if(!!strpos($callback['callback'], '@') !== false) {


            if(!empty($callback['namespace']))
            {
                $namespace = $callback['namespace'];
            }

            $callback['callback'] = explode('@', $callback['callback']);
            $controller = $namespace.$callback['callback'][0];
            $method = $callback['callback'][1];

            $rc = new \ReflectionClass($controller);

            if($rc->isInstantiable() && $rc->hasMethod($method))
            {
                return call_user_func_array(array(new $controller, $method), array_values($params));

            } else {

                throw new \Exception("Erro ao despachar: controller não pode ser instanciado, ou método não ex
            }
        }
    }
    throw new \Exception("Erro ao despachar: método não implementado");
}
```

Note que alguns lugares alteramos de $callback para $callback[‘callback’]. O método cresceu um pouco mas nada muito além do que se espera.

Se você fizer testes, estando tudo certo, maravilha! Vencemos essa! Caso não, volte e dê uma olhada no seu código se está condizente com o do artigo. Não tem problemas em usar a área de comentários para retirar dúvidas, ou mesmo enviar um e-mail.

Depois disso, é preciso editar a classe Router e adicionar um método responsável por ajudar o Router a informar os parâmetros corretamente para Dispacher. Este método getValues recebe um padrão, converte-o para array e usando as posições salvas junto a rota definida, faz a substituição dos espaços pelos valores recebidos na requisição (request):

```
1  protected function getValues($pattern, $positions)
2  {
3      $result = [];
4
5      $pattern = array_filter(explode('/', $pattern));
6
7      foreach($pattern as $key => $value)
8      {
9          if(in_array($key, $positions)) {
10             $result[array_search($key, $positions)] = $value;
11         }
12     }
13
14     return $result;
15
16 }
17
18
```

Perfeito! Agora Router tem o método getValues() que dará suporte ao método resolve(). O método resolve(), recebe um objeto Request que têm os métodos para se extrair as informações (method() e uri()), que já são lançados como argumentos ao método find() de Router. O método find() sabe como obter uma rota por informar o método de requisição (post, get, put, delete) e a URI. Se encontrar uma rota definida, manda para o Dispacher. Senão, invoca o método que responde com erro 404.

Então, olhe bem para o método abaixo em seu estado atual:

```php
public function resolve($request){

    $route = $this->find($request->method(), $request->uri());

    if($route)
    {
        return $this->dispach($route);
    }
    return $this->notFound();

}

```

Agora altere para ficar dessa maneira:

```php
public function resolve($request){

    $route = $this->find($request->method(), $request->uri());

    if($route)
    {

        $params = $route->callback['values'] ? $this->getValues($request->uri(), $route->callback['values']) :

        return $this->dispach($route, $params);
    }
    return $this->notFound();

}

```

Agora o método resolve(0) está informando corretamente os parâmetros como argumento para o Dispacher, pois ele têm o meio de obtê-los usando o método getValues(), adicionado ainda pouco.

Ainda há outra correção no método dispach() de Router. Este método ainda não está recebendo os parametros de resolve() e falhará se tentar executar o aplicativo desta maneira. Para corrigir isso, veja como está o método no momento:

```php
protected function dispach($route, $namespace = "App\\"){

    return $this->dispacher->dispach($route->callback, $route->uri, $namespace);
}

```

Atenção: tanto Router quanto Dispacher possuem um método dispach()

Agora, atenção também a mudança porque é bem sutil. Você deve alterar para ficar dessa forma (Eis aqui mais outro ponto de falha!):

```php
protected function dispach($route, $params, $namespace = "App\\"){

    return $this->dispacher->dispach($route->callback, $params, $namespace);
}

```

Feito isso, é possível testar a passagem de parâmetros por adicionar uma rota assim (arquivo routes.php no diretório routes):

```php
$router->get('/teste/{teste}', function($teste){

    echo "Agora foi recebido da URI o parâmetro: " . $teste;

});

```

Para testar, é precisa digitar uma URL que termine por exemplo em teste e seu nome, talves assim (no meu caso): teste/alexandre.

É possível fazer testes mais exigentes para entender se os parâmetros e padrões estão se ajustando corretamente: **URI** com **rotas** casando-se e argumentos definido injetados na closure:

```php
$router->get('/produto/{produto}/categoria/{categoria}/editar', function($produto, $categoria){

    echo "Recebeu => produto: " . $produto . "<br />";
    echo "Recebeu => categoria: " . $categoria . "<br />";

});

```

Modéstia à parte, se reparar bem, as classes estão bem elegantes! Até aqui, você poderá se divertir um pouco, criando rotas até ficar enjoado! 🙂

Até agora não fou feito nenhum teste com rotas do tipo PUT e DELETE. Estas são especiais e normalmente são usadas para API ou aplicações Resful. Alguns frameworks como Laravel, permitem criar um formulário que envia uma requisição post com um input oculto informando que o método a ser redirecionado é PUT ou DELETE. Isso ocorre porque Browsers não enviam estes tipos de requisições. Aqui não será feito, mas o sistema de rotas está aceitando normalmente estes métodos PUT e DELETE. Para testar, você pode criar rotas e gerar solicitações com aplicativos tais como Postman.

Mas não há nenhum teste com nomes de rotas! Não está funcionando ainda porque não foi implementado. Agora é o momento de trabalhar nesta parte:

```
1  $router->get(['set' => '/cliente/{cliente_id}', 'as' => 'clientes.edit'], function($cliente_id){
2
3      echo "Cliente => " . $cliente_id;
4
5  });
6
```

Se testar esta rota, por digitar no navegador uma URI que case com aquele padrão, deverá funcionar normalmente. Mas o caso não é este, mas antes a tradução da rota é o que queremos. Para isso acontencer, é preciso editar Router e adicionar o método translate() com esse objetivo: Quando um nome for informado, (em um View ou controller, etc), automaticamente, Router deve responder com a URI correta renderizada com os argumentos fixando valores nas suas corretas posições. Este é o código do método translate():

```php
public function translate($name, $params)
{
    $pattern = $this->route_collection->isThereAnyHow($name);

    if($pattern)
    {
        $protocol = isset($_SERVER['HTTPS']) ? 'https://' : 'http://';
        $server = $_SERVER['SERVER_NAME'] . '/';
        $uri = [];

        foreach(array_filter(explode('/', $_SERVER['REQUEST_URI'])) as $key => $value)
        {
            if($value == 'public') {
                $uri[] = $value;
                break;
            }
            $uri[] = $value;
        }
        $uri = implode('/', array_filter($uri)) . '/';

        return $protocol . $server . $uri . $this->route_collection->convert($pattern, $params);
    }
    return false;
}
```

O método translate() precisa contruir uma URI completa de forma confiável, com o protocolo (HTTP ou HTTPS), nome do servidor e caminho onde está o index.php. Esta foi a maneira mais simples de desenvolvê-lo. Em ambiente de testes por exemplo, uma URI seria assim:

```
http://localhost/rotas/public/cliente
```

Pronto! O método de tradução **translate()** está concluído. Mas é claro que depois de compreender bem a idéia, você estará pronto para refatorar e melhorar oquê e como julgar melhor!

Como ver se está tudo funcionando? Volte ao arquivo routes.php no diretório routes, e declare esta rota para teste:

```
$router->get('teste', function() use($router){

    echo '<a href="' . $router->translate('clientes.edit', 1) . '">Clique aqui para testar a rota clientes.edit

});
```

Entendeu a sacada?! Você tem praticamente um helper para criar as rotas nomeada, assim como nos grandes Frameworks. Eu poderei até estar enagando, mas o momento em que publiquei esta ultima parte da série, não havia encontrado nenhuma série de artigos, tutorial ou tema que ensinasse de qualquer forma a criar um sistema de rotas, tão completo quanto este! Você é livre para se expressar e, se não concordar, fique a vontade para me enviar um e-mail com o link do artigo ou tutorial, ensinando de forma semelhante ou superior.

Enfim, o sistema de rotas está concluído! se você já estiver satisfeito e achar que já é suficiente, pois já entendeu a completamente a idéia, entederei se desejar parar por aqui! E Se essa for a sua escolha, obrigado por acompanhar a série de artigos, espero que tenha aproveitado o máximo! Se ficou com alguma dúvida, sugiro reler a sessão onde há dúvidas. abraço!

Mas para aqueles que querem mais

Direcionando dados para um controller

Se preferiu continuar e ver como acionar uma action de um controller, seja bem-vindo! Agora também não será um processo bem complicado. Isto porque o sistema já é inteliente suficiente para invocar métodos das classes, informando parâmetros. O que você precisa compreender é, que estes artigos não são focados na construção de um aplicativo completo conforme a arquitetura MVC. Se por um lado isso agradaria alguns, entendo que um tema mais extenso afastaria outros. Por isso, o controller também será apenas uma classe com dois métodos básicos, para apresentar a idéia e lhe dar a noção de como tudo se encaixa. Se você querer uma aplicação completa, sugiro o desafio: Unir os conhecimentos obtidos nesta série de artigos com o artigo PHP::CRUD COM MVC! Daí se você querer ainda mais, junte com o conhecimento do artigo PHP:: PADRÃO DE PROJETO ACTIVE RECORD – ORM onde mostro uma forma de elaborar um ORM para operar com classes Models, totalmente inspirado na idéia do Eloquent (No entanto, nem neste artigo ou qualquer outro tenho objetivo de criar algum tipo de código concorrente, a idéia é mostrar a você, que está estudando ou quer passar um tempo entendo funcionalidades,

a construir alguma coisa, desenvolver projetos realmente funcionais, satisfatórios e que funcionam semelhante a outros projetos conceituados no mundo). Inclusive, aceitando esse desafio, por unir todos estes elementos, você acabará tendo praticamente um framework, construído do zero! Talvez em artigos futuros poderei fazer esta junção.

Voltando ao assunto, antes do Controller, é necessário adicionar funções helpers para melhor atender o projeto, e continuando com o conceito de alto nível: tornar as coisas mais simples com código limpo! Criar helpers é outra técnica empregada na construção de muitos frameworks.

Para isso, siga até o diretório src e crie um novo subdiretório chamado helpers. Neste diretório, crie um arquivo chamado **helper_routes.php**. Você está livre para escolher a forma que preferir para criar o arquivo. Mas se não pensou em nenhuma forma, dentro do novo subdiretório (se vocÊ estiver usando Linux ou gitbash), digite:

```
touch helper_routes.php
```

Ainda não faça nada neste arquivo, deixe-o oco, salvo sem linhas de código. E porque você tem acompanhando até esta etapa, quero lhe informar que vai valer muito a pena! Entenda já um dos motivos: Aqui o funcionamento do sistema de rotas será aprimorado.

Existe um padrão de projetos conhecido com Singleton. Este padrão de projeto é um método usado para preservar uma única instancia de objetos durante toda execução do nosso aplicativo. Você consegue imaginar onde poderia ser utilizado no projeto atual? Na instância de Router! Então, saia do subdiretório helpers e no volte para diretório “src”, onde estão as classes de funcionamento de todo sistema, adicione uma classe chamada Route, salvando o arquivo como Route.php.

Esta classe, além de ser uma singleton, também atuará como uma Facade para sistema de rotas. O código dela fica assim:

```php
<?php

namespace Src;

use Src\Router;

final class Route
{
    protected static $router;

```

```
11      private function __construct()
12      {}
13
14      protected static function getRouter()
15      {
16          if(empty(self::$router)) {
17              self::$router = new Router;
18          }
19          return self::$router;
20      }
21
22      public static function post($pattern, $callback){
23          return self::getRouter()->post($pattern, $callback);
24      }
25
26      public static function get($pattern, $callback){
27          return self::getRouter()->get($pattern, $callback);
28      }
29
30      public static function put($pattern, $callback){
31          return self::getRouter()->put($pattern, $callback);
32      }
33
34      public static function delete($pattern, $callback){
35          return self::getRouter()->delete($pattern, $callback);
36      }
37
38      public static function resolve($pattern){
39          return self::getRouter()->resolve($pattern);
40      }
41
42      public static function translate($pattern, $params){
43          return self::getRouter()->translate($pattern, $params);
44      }
45
46  }
47
48
```

Não vou detalhar os métodos porque você já os conhece! Contudo, agora eles poderão ser invocados estaticamente. Além disso, preservamos a nível de classe o Router, não precisamos mais instanciar!

Uma característica de classes conforme o design pattern Singleton, é não permitir ser instanciada pelo código cliente, apenas de dentro para fora. Só que neste caso, route não precisa ser instanciada, mas vai guardar a instância de Router a nível de classe. Feito isso, abra o arquivo bootstrap.php e fazer algumas alterações. Ele está assim no momento:

```php
<?php

error_reporting(E_ALL);
ini_set('display_errors', true);

require __DIR__ . '/vendor/autoloader.php';

use Src\Router;

session_start();


try {

    $router = new Router;

    require __DIR__ . '/routes/routes.php';

} catch(\Exception $e){

    echo $e->getMessage();
}


```

Altere para ficar assim:

```php
<?php

error_reporting(E_ALL);
ini_set('display_errors', true);

require __DIR__ . '/vendor/autoloader.php';

session_start();

try {

    require __DIR__ . '/routes/routes.php';

} catch(\Exception $e){

    echo $e->getMessage();
}


```

Sim! a instanciação de Router é excluída (Você já deve compreender o motivo, incusive entendendo o porquê ser desnecessário preservar tudo na maneira anterior), bem como a importação da classe (Também não é mais necessário aqui). A importação da classe deve ir para o arquivo routes.php, bem no início dele, desta foram:.

```php
use Src\Route as Route;


```

Neste momento, o código quebrou! Por isso, abra o arquivo routes.php no diretório routes e, tomando como base o exemplo abaixo, altere todas declarações de rotas para ficar desta maneira:

```php
Route::get('teste', function() {

    echo '<a href="' . Route::translate('clientes.edit', 1) . '">Clique aqui para testar a rota clientes.edit</

});


```

Agora o arquivo index.php em public também deve refletir a mudança. Ele estava assim:

```php
<?php

require __DIR__ . '/../bootstrap.php';

$request = new Src\Request;

$router->resolve($request);


```

Ele deve ficar assim:

```php
<?php

require __DIR__ . '/../bootstrap.php';

use Src\Route as Route;

$request = new Src\Request;

Route::resolve($request);


```

Então feito isso, é possivel editar aquele arquivo chamado helper_routes.php. Abra para edição e inclua o seguinte conteúdo:

```php
<?php

use Src\Route;
use Src\Request;

function request()
{
    return new Request;
}


function resolve($request = null)
{
    if(is_null($request)) {
        $request = request();
    }
    return Route::resolve($request);
}


function route($name, $params = null)
{
    return Route::translate($name, $params);
}

function redirect($pattern)
{
    return resolve($pattern);
}

function back()
{
    return header('Location: ' . $_SERVER['HTTP_REFERER']);
}


```

Se você gostar da idéia pode adicionar mais funções relacionadas ou mais arquivos com funções. Certo, mas e como fazer para carregar essas funções no sistema? composer é a resposta!

Estas funções helpers poderão ser acionadas em qualquer parte do sistema, tornando mais simples a criação dos controllers e views. Além disso, se neste projeto encarregam de devolver os objetos como se fossem fábricas.

Abra o arquivo composer.json no diretório raiz e adicione instruções para autoload de funções. Então, edite o arquivo composer.json e ele deve ter uma semelhança com este:

```
{
    "name": "abbarbosa/artigo_rotas",
    "description": "Artigo sobre a criação de sistema de rotas para mvc",
    "authors": [
    {
        "name": "Alexandre Barbosa",
        "email": "alxbbarbosa@hotmail.com"
    }
    ],
        "require": {},
        "autoload" : {
            "psr-4": {
            "App\\" : "app/",
            "Src\\" : "src/"
        }
    }
}
```

Altere para ficar assim:

```
{
    "name": "abbarbosa/artigo_rotas",
    "description": "Artigo sobre a criação de sistema de rotas para mvc",
    "authors": [
    {
        "name": "Alexandre Barbosa",
        "email": "alxbbarbosa@hotmail.com"
    }
    ],
        "require": {},
        "autoload" : {
            "psr-4": {
            "App\\" : "app/",
            "Src\\" : "src/"
        },
        "files" : [
            "src/helpers/helper_routes.php"
        ]
    }
}
```

Feito isso, novamente, execute o comando:

```
composer dump-autoload
```

Pronto! As funções de helpers já estão disponíveis e funcionam muito bem!

Opcionalmente, se você quiser, pode até voltar de novo lá no index.php no diretório public, e "limpar" para ficar dessa forma:

```php
<?php

require __DIR__ . '/../bootstrap.php';

resolve();


```

Só isso mesmo, e mais nada! Fale a verdade: Estes códigos são muitos elegante e limpos! não é mesmo? Junte eles aos conceitos de Testes Unitários para investir na qualidade e logo até seus modos pessoais serão mais exigentes e inevitavelmente sistemáticos! Mas até certo ponto isso é bom!

Você se lembra do Controller.php no diretório/pasta app? Sim, edite este arquivo e adicione dois métodos, que estarão logo a seguir.

Observação: Se algum dado usado para teste realmente existir, é mera coincidência! Aqui foi definido um array para simular dados, como se viessem de um banco de dados.

```php
<?php

namespace App;

class Controller
{

    protected $clientes = [
        ['id' => 1, 'nome' => 'Antônio Silva', 'telefone' => '119990000'],
        ['id' => 2, 'nome' => 'João Silva', 'telefone' => '158999999'],
        ['id' => 3, 'nome' => 'Maria Silva', 'telefone' => '119999001'],
        ['id' => 4, 'nome' => 'Marta Santos', 'telefone' => '189990001'],
        ['id' => 5, 'nome' => 'Paulo Moura', 'telefone' => '1799990002'],
```

```
14      ];
15
16      public function index()
17      {
18
19          $data = array_map(function($row){
20
21              return '<tr><td>' . $row['nome'] . '</td><td>' . $row['telefone'] . '</td><a href="' . route('clie
22
23          }, $this->clientes);
24
25          echo "<h1>Listagem</h1><br /><hr>";
26          $table = '<table width="100%"><thead><tr><td>Nome</td><td>Telefone</td>Ações<td></tr></thead>';
27          $table .= '<tbody>'. implode('', $data) .'</tbody></table>';
28          echo $data;
29
30      }
31
32      public function show($id)
33      {
34
35          foreach($this->clientes as $row)
36          {
37              if($row['id'] == $id)
38              {
39                  $cliente = $row;
40              }
41          }
42
43          echo "<h1>Detalhes:</h1><br /><hr>";
44          $data = 'nome: ' . $cliente['nome'] . '<br>telefone: ' . $cliente['telefone'];
45          $data .= '<br /><a href="' . route('clientes.index') . '">Clique aqui para voltar para lista</a>';
46          echo $data;
47
48      }
49
50
51
```

Controller está concluído! Agora se você já treinou , usando outras rotas e quiser excluí-las do arquivo routes.php, fique a vontade. Será melhor porque pode haver confilitos. Mas adicione estas (arquivo routes.php no diretório routes):

```
1  Route::get(['set' => '/cliente', 'as' => 'clientes.index'], 'Controller@index');
2
3  Route::get(['set' => '/cliente/{id}/show', 'as' => 'clientes.show'], 'Controller@show');
4
5
6
7
8
```

Note que foi dado um nome para as rotas e além disso, o valor capturado na variável deve ser encaminhado normalmente para o método do controller. Aqui são dois métodos, sendo que em ambos temos o helper route() traduzindo o nome de rota para URI.

Agora! Faça o teste para ver o que você acha do resultado! Tenho certeza de que ficou muito satisfatório! É claro que o controller usado aqui é bem minimalista, porque o objetivo, como já posto, não era explorar todo o MVC, mas apenas rotas. Por isso, não criamos Models e muito menos Views.

Agora, sim! Concluímos esta série de artigos, sendo este a terceira e última parte. Você pode rever, ou mesmo se ainda não leu, seguir para eles nestes links: primeira parte ou segunda parte e espero mesmo que tenha alcançado meu objetivo de apresentar para você uma série incrível de 3 arigos sobre a construção de um sistema de rotas completo. Se você quiser mais desafios, faça um clone deste diretório que você criou, contendo todos os arquivos, toda estrutura. Então, como dito antes, faça a fusão de conhecimentos, com o obtido no artigo PHP::CRUD COM MVC! Daí se você querer ainda mais, junte com o conhecimento do artigo PHP:: PADRÃO DE PROJETO ACTIVE RECORD – ORM. Tenho certeza que você vai gostar muito mais do resultado!

Agora, também é importante deixar claro que mesmo neste projeto, há muitas coisas a melhorar. Por exemplo: tratativa de falhas, validações: Controller existe? Controller têm tal método, o método têm tais argumentos? etc. Isso enriquecerá ainda mais o projeto.

Nunca deixe de comentar o que achou, porque esse feedback é muito importante para dar um direcionamento a este blog. Já houveram temas que realmente não valeram a pena investir em preparo de várias horas como é o caso deste. Portanto, por seu feedback tenho como saber se estou na direção certa!