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

Construir um sistema de rotas para a aplicativos é sem dúvida muito interessante. E quero que você fique fera nisso!
É uma técnica moderna e que deixará seus aplicativos muito mais elegantes!
Existem muitas formas de se construir um, mas tenho certeza que o modo como construiremos aqui, vai deixar você muito satisfeito!
Estou escrevendo estes artigos da maneira que eu sempre quis encontrar quanto estava tentando aprender a criar o primeiro sistema de rotas. Espero que, se não for exatamente o que você espera, mesmo assim lhe sirva de boa base para o que você procura!

O que são url amigáveis?

Para criar um sistema de rotas bem feito, você precisa compreender o que são url amigáveis.

Em resumo podemos dizer que são URLs bem mais fáceis de entender do que o método convencional.

Quais você acha mais fácil para ler?

http://www.sitedevendas.com.br/index.php?prod=451
http://www.paginadofulano.com.br/artigos.php?cat=23&art=12

ou

http://www.sitedevendas.com.br/produto/451

http://wwww.paginadofulano.com.br/artigos/programacao/variaveis

Eu sei que a pergunta é retórica e a resposta óbiva!

Quando você publica uma url, na verdade esta representa um recurso hospedado em um servidor. O primeiro, exemplo convencional indica exatamente um diretório e script publicado para receber a solicitação. Se aquele caminho existir e aquele script existir (por exemplo: index.php ou artigos.php) Estes serão enviados como resposta a solicitação/requisição do usuário.
Para que um mesmo script possa receber entradas de dados adicionais, ou parâmetros, utilizamos uma sintax que associa pares, sendo um nome a um valor, chamado de query string.

Como diz a wikipidia: (https://pt.wikipedia.org/wiki/URL)

A query string é um conjunto de um ou mais pares "pergunta-resposta" ou "parâmetro-argumento" (como por exemplo nome=fulano, em que nome pode ser, por exemplo, uma variável, e fulano é o valor (argumento) atribuído a nome).É uma string enviada ao servidor para que seja possível filtrar ou mesmo criar o recurso. (opcional)

Legal, isso funciona, mas não é tão bonito quanto ao sistema moderno de url amigáveis.

O sistema de rotas exige url amigáveis? **Não necessáriamente**, mas com url amigáveis ficará realmente *elegante*.

A rota é uma definição de destino para uma solicitação. E como funciona junto a url amigáveis?

Ao invés de vários scripts agirem como porta de entrada para sua aplicação, apenas um script será a entrada. Este script normalmente será o index.php.

Este script não precisa ter códigos HTML, mas precisa ter o meio de redirecionar a solicitação. O script em si não fará isso, mas um objeto roteador irá receber a requisição (request).

Este roteador precisa então verificar em sua coleção de rotas, se aquela rota existe. Se não existir, não atende o pedido, e você poderá tratar essa resposta com um erro do tipo 404 "Não encontrado".

Se a rota for encontrada, o roteador chama um metodo despachante, ou um objeto (Dispacher) despachante para invocar o recurso mapeado para aquela rota.

O recurso em projetos mvc, em geral é uma action de um controller, nome carinhoso dado aos métodos de uma classe controller. Então, suponhamos que você tenha uma rota para que deva abrir a página inicial do seu aplicativo web. Talvez, poderia direcionar para uma action "index" do controller responsável.

Mas se você esperava começar por esta configuração, não vai ser por aí que vamos começar! Isso foi foi apenas para dar uma aquecida e para você ter noção do que vêm por aí! Então vamos primeiro criar algumas classes, estas serão muito importantes.

Conhecendo nossos protagonistas

Vamos criar cinco classes:

- Router – A classe roteadora
- RouteCollection – A classe que mantém a coleção de rotas
- Dispacher – O despachante
- Request – Um classe para manipular a requisição
- Controller – Um controller para prover algumas actions

Além disso, vamos precisar de mais alguns arquivos:

- index.php – A entrada das solicitações
- route.php – O arquivo onde declaramos as routas
- .htaccess – Arquivo no diretório raiz onde criaremo algumas regras
- .htaccess – Arquivo no diretório public onde criaremo algumas regras

A estrutura de diretórios será assim:

- App
    - Controller.php
    - src
    -
- public
    - index.php

- routes
    - routes.php
- src
    - Router.php
    - RouteCollection.php
    - Request.php
    - Dispacher.php
- .htaccess
- bootstrap.php

Essa estrutura vai atender bem nosso propósito, e você pode até usar como lição de casa para aplicar ao artigo de mvc daqui do blog. Vai ficar muito legal!

Começando

Primeiramente, sugiro criar todas os diretórios (ou pastas como preferir).

Você pode criar também os arquivos "ocos" (Sem nada) apenas como texto. Por exemplo, se estiver usando Linux ou o gitBash no windows, poderá usar o touch.

No diretório raiz do projeto:

```
touch bootstrap.php
touch .htaccess
```

No diretório app:

```
touch app/Controller.php
```

No diretório src:

```
touch src/Router.php
touch src/RouteCollection.php
touch src/Dispatcher.php
touch src/Request.php
```

No diretório public:

```
touch public/index.php
touch public/.htaccess
```

No diretório routes:

```
touch routes/routes.php
```

Até aqui você terá basicamente a estrutura física de diretórios e pastas concluída!

Você poderia até estar se perguntando se tudo isso é necessário e eu diria que "depende"!

Esta estrutura acima começa a se parecer um pouco com estruturas usadas em frameworks atuais, então como você pode perceber não estaremos fazendo um trabalho amador aqui.

Começando a dar forma

Vamos começar a criar classes "ocas" primeiro. Isso parece dar mais trabalho, mas será melhor para você acompanhar. Note que vamos definir inclusive os namespaces para auxiliar no carregamento das classes pelo autoloader. Imagino que você deve saber o que é um namespace. Caso tenha dúvidas, segue o artigo do site do php: https://www.php.net/manual/pt_BR/language.namespaces.rationale.php

Primeiro vá até o diretório app e use um editor de textos ou ide de sua preferencia, para editar o arquivo Controller.php

Este arquivo deve receber este conteúdo:

```
1  <?php
2
3  namespace App;
4
5  class Controller
6  {
7
8  }
9
10
```

Salve a classe vá para outro diretório src. Então começe a editar o RouterCollection.php.

Este arquivo deve receber este conteúdo:

```php
<?php

namespace Src;

class RouteCollection
{

}

```

Salve a classe edite o arquivo Request.php.

```php
<?php

namespace Src;

class Request
{

}

```

Salve a classe edite o arquivo Dispacher.php.

```php
<?php

namespace Src;

class Dispacher
{

}

```

Salve a classe edite o arquivo Router.php, preste atenção porque já estaremos importando a classe Request, RouteCollection e Dispacher aqui.

Note as declarações “use” e não esqueça de adicioná-las:

```php
<?php

namespace Src;

use Src\Dispacher;
use Src\RouteCollection;

class Router
{

}


```

Agora, para criar o autoloader, preciamos do composer. Se você não têm o composer instalado, dê uma olhada no site do composer como fazer para instalar (https://getcomposer.org/download/)

Depois de conferir que o composer está instalado e funcionando, vamos começar por criar o arquivo composer.json. Para isso, no terminal, digite:

```
composer init
```

O composer vai iniciar um pequeno assistente fazendo algumas perguntas para ter base para montar o arquivo. Algo assim será mostrado:

Welcome to the Composer config generator

This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [alxbb/rotas]:

Note que para montar as informações sobre o pacote, o composer usa o seu nome de usuário e o nome do diretório raiz do projeto (author/project). Se você não estiver satisfeito, fique a vontade para mudar.

Eu mudei para abbarbosa/artigo_rotas e teclei enter. Depois disso, dei uma descrição, confirmei o e-amil e fui teclando enter até ver o resultado:

This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [alxbb/rotas]: abbarbosa/artigo_rotas
Description []: Artigo sobre a criação de sistema de rotas para mvc
Author [Alexandre Barbosa <alxbbarbosa@hotmail.com>, n to skip]:
Minimum Stability []:
Package Type (e.g. library, project, metapackage, composer-plugin) []:
License []:

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]?
Search for a package:
Would you like to define your dev dependencies (require-dev) interactively [yes]?
Search for a package:

```
{
    "name": "abbarbosa/artigo_rotas",
    "description": "Artigo sobre a criação de sistema de rotas para mvc",
    "authors": [
    {
        "name": "Alexandre Barbosa",
        "email": "alxbbarbosa@hotmail.com"
    }
    ],
    "require": {}
}
```

```
Do you confirm generation [yes]?
```

Para mim está tudo certo, então teclei enter para concluir e gerar o arquivo. Então você vai ver o arquivo composer.json no diretório raiz do projeto.

Vamos então editar o arquivo composer.json para adicionar as declarações que criará o autoloader. abra o arquivo e adicione uma vírgula após as chaves de require, estas linhas, antes do fechamento da última chave:

```
    "autoload" : {
    "psr-4": {
    "App\\" : "app/",
    "Src\\" : "src/"
    }
}
```

O conteúdo ficará parecido 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/"
        }
    }
}
```

Atenção: Note que a declaração do namespace encerra com duas contra-barras "\" ao passo que a declaração do diretório com apenas uma barra "/".

Se por algum motivo, você precisar criar subdiretórios e se referir a eles, siga sempre esta idéia, a separação entre elementos do namespace com duas contra-barras, mas a separação de elementos do diretório com apenas uma barra "/".

Isto posto, salve o arquivo, volte ao terminal e execute o comando:

```
composer dump-autoload
```

Se tudo der certo, a única mensagem que você vai ver será “Generating autoload files”, enquanto o composer está criando o autoloader, depois voltará para o prompt do terminal.

Se tudo der errado, volte para o seu editor e confira cada linha, verificando se elas estão de acordo com o modelo apresentado aqui no artigo. Quando encontrar e corrigir a falha, volte ao terminal e execute o mesmo comando: composer dump-autoload

Após dar tudo certo, note na estrutura de diretórios que um novo diretório vendor foi criado contendo o autoloader e os elementos necessários para seu funcionamento. Agora a cada alteração de diretórios, será necessário executar esse mesmo comando:

```
composer dump-autoload
```

Atenção:Não deixe nenhum espaço antes da tag de abertura do script php: <?php

Se você deixar alí algum espaço, vai ter um erro de interpretação de namespace.

Até aqui, o autoloader já é plenamente capaz de localizar e instanciar as classes, mas para isso acontecer, precisamos já definir que começa a a derrubar os dominos! Faremos isso no arquivo chamado bootstrap.php. Este arquivo não tem nada a ver com o Twitter Bootstrap para css! 🙂

A princípio, precisamos adicionar estas instruções:

```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();
}
```

Neste arquivo, estamos capturando erros para tratá-los, mas também habilitamos explicitamente o log e apresentação de erros do php. Em alguns casos, você não precisa desta informação porque em seu ambiente de desenvolvimento já deve estar configurado. Mas em muitos sistemas não estará, por isso acredito ser importante deixar essas declarações error_reporting() e ini_set() explícitas neste arquivo.

Além disso, também estamos iniciando a sessão. Pode ser que nestes artigos não seja feito uso deste recurso, mas apenas para que você possa entender onde deve ficar esta instrução. Também, quando crio frameworks, costumo definir aqui as configurações de banco de dados para as classes de ORM.

Este arquivo é o ponto de partida fria da nossa aplicação! Isto significa que, iniciamos todos recursos necessários para então dar vida a aplicação aqui. Em muitos frameworks, será instanciado um container App, se estivéssemos construindo uma aplicação completa, seria nesse script que estariamos instanciando um App. Normalmente esse App que iria instanciar um objeto da classe roteadora. Como não estaremos nesses detalhes aqui, vamos encurtar o caminho!

Por isso, instanciamos aqui um objeto roteador e logo em seguida, importamos o arquivo, onde serão declarados as rotas que serão adicionadas a coleção de rotas. Voltaremos a esse arquivo mais tarde, por enquanto não vamos mais fazer nada nele.

Então, abra o arquivo index.php dentro do diretório public e aqui vamos adicionar uma linha para importar esse arquivo bootstrap.php:

```php
<?php

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

Só isso aqui por enquanto, pode fechar o arquivo também!

Vamos criar algumas coisas em paralelo para que você possa compreender como determinada classe tem ligação com outra. Portanto, se você ficar perdido, re-leia a ideia do projeto e comente aí abaixo o que não compreendeu. Só não vale perguntar aqui o que ainda não foi abrangido, tenha calma! 😉

Então vamos começar a construir nosso sistema de roteamento. Primeiro, sabemos que a classe RouteCollection é exatamente o que o nome dela diz sobre si! Isso parece ter ficado redundante! 🙂

Então, vamos definir essa classe inicialmente assim:

```php
<?php

namespace Src;

class RouteCollection
{
    protected $routes_post = [];
    protected $routes_get = [];
    protected $routes_put = [];
    protected $routes_delete = [];


    public function add($request_type, $pattern, $callback)
    {
        switch($request_type)
        {
            case 'post':
                return $this->addPost($pattern, $callback);
                break;
            case 'get':
                return $this->addGet($pattern, $callback);
                break;
            case 'put':
                return $this->addPut($pattern, $callback);
                break;
```

```php
            case 'delete':
                return $this->addDelete($pattern, $callback);
            break;
            defautl:
                throw new \Exception('Tipo de requisição não implementado');
        }
    }

    public function where($request_type, $pattern){

    }


    protected function definePattern($pattern) {

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

    }

    protected function addPost($pattern, $callback){

    }

    protected function addGet($pattern, $callback){

    }

    protected function addPut($pattern, $callback){

    }

    protected function addDelete($pattern, $callback){

    }

}


```

Perceba que adicionei quatro atributos como array, porque nesse caso, vou preferir tratar cada tipo de requisição em um conjunto separado. Claro que poderiamos fazer todos em uma propriedade do tipo array que fosse multidimencional, mas o caso aqui será abstrair mesmo para compreender melhor o entendimento. Depois você quando compreender a idéia, podera refatorar como julgar melhor.

Note que também prefer separa um método publico que faz a triagem do tipo de rota e em seguida, preferi também separar a alimentação das propriedades em métodos protegidos separados. Porquê? Acredito que assim ficará melhor para fazermos depurarmos e darmos manutenção ao código.

Você vai perceber que há também outro método públic "where". Este método será usado pela classe roteadora para obter a rota.

Antes de avançar falando da where, você deve estar imaginado que eu deixaria de falar sobre o método definePattern! Bem este método está armazendo o padrão como um padrão regex. Você logo vai entender porque isso daí vai nos trazer uma super funcionalidade! Veja que antes de armazenar a rota, com este método preparamos a chave para armazenar de uma forma que será melhor para procurá-la depois! Você já deve ter matado a charada, mas se não, continue acompanhando a idéia!

Ok mas voltando a falar dos métodos que adicionam a rota, talvez então, você já tenha uma idéia da implementação destes métodos e de fato é simples, veja o primeiro:

```
1  protected function addPost($pattern, $callback){
2
3      $this->routes_post[$this->definePattern($pattern)] = $callback;
4      return $this;
5
6  }
7
8
```

Fácil? Sim, talvez o que você não esteja acostumado seja com este tipo de retorno. Estamos retornado a própria instância da RouteCollection para que seja possível uma certa fluência de invocação de métodos. Se você não sabe o que é isso, não se preocupe.

Então, podemos adicionar os métodos restantes, com exceção do where:

```php
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;

}
```

Por enquanto, vamos dar uma pausa nesta classe, entender o que precisamos implementar:

Nosso roteador precisa ter um meio de adicionar em sua coleção, novas rotas que forem declaradas no arquivo routes.php. Teremos de usar uma sintaxe onde informamos o padrão $pattern e a $callback. Dessa forma mapearemos nossa rota para um tipo de requisição. Além disso, seria interessante que cada método tivesse o nome do tipo de requisição. Não é dessa maneira que você conhece os sistemas de rotas dos frameworks modernos?

Isto posto, vamos começar assim:

```php
<?php

namespace Src;

use Src\Request;
use Src\Dispacher;
use Src\RouteCollection;

class Router
{

    protected $route_collection;
```

```php

    public function __construct()
    {

        $this->route_collection = new RouteCollection;
        $this->dispacher = new Dispacher;
    }

    public function get($pattern, $callback)
    {

        $this->route_collection->add('get', $pattern, $callback);
        return $this;
    }

    public function post($pattern, $callback)
    {

        $this->route_collection->add('post', $pattern, $callback);
        return $this;
    }

    public function put($pattern, $callback)
    {

        $this->route_collection->add('put', $pattern, $callback);
        return $this;
    }

    public function delete($pattern, $callback)
    {

        $this->route_collection->add('delete', $pattern, $callback);
        return $this;
    }

    public function find($request_type, $pattern)
    {
        return $this->route_collection->where($request_type, $pattern);
    }
}

```

Olhando para nossa classe roteadora, você pode imaginar que a maioria dos métodos são praticamente auto explicativos, não é mesmo? Estamos usando um método com nome do tipo de requisição para adicionar rotas a coleção. Existe um unico método na classe RouteCollection para isso, que recebe três argumentos:o tipo de requisição, o padrão e o callback (que poderá ser uma closure ou uma string com nome de um controller e uma action).

Depois começamos a implementar o retorno, veja que temos um método find que é responsável por procurar na coleção onde está uma rota que seja condizente com aquele padrão. Veja que a lógica está cada vez fazendo mais sentido! Mas ainda não totalmente porque você ainda não sabe como ser implementado na RouteCollection. Mas talvez já tenha uma idéia e sem imaginar, ela já pode funcionar! Então salve esta classe Router e vamos voltar para edição de RouterCollection. Vamos implementar o método where, porque ele está vivo em sua mente!

```
public function where($request_type, $pattern){

    switch($request_type){
            case 'post':
                return $this->findPost($pattern);
                break;
            case 'get':
                return $this->findGet($pattern);
                break;
            case 'put':
                return $this->findPut($pattern);
                break;
            case 'delete':
                return $this->findDelete($pattern);
            break;
            defautl:
                throw new \Exception('Tipo de requisição não implementado');
    }

}


protected function parseUri($uri)
{

    return implode('/', array_filter(explode('/', $uri)));
}


```

Você pensou que tinha acertado! Sim, a sua maneira pode dar certo também! Mas prefiro dividir as reponsabilidades para que os métodos fiquem menores e tenha apenas um papel. No caso do where, estará assim com add, redirecionado a responsabilidade para outros métodos mais específicos.

Note também que há mais um método que dará suporte a interpretação dos padrões. Então veja a implemetação de um destes métodos:

```
protected function findPost($pattern_sent)
{

    $pattern_sent = $this->parseUri($pattern_sent);

    foreach($this->routes_post as $pattern => $callback) {

        if(preg_match($pattern, $pattern_sent, $pieces))
        {
            return (object) ['callback' => $callback, 'uri' => $pieces];
        }
    }
    return false;

}


protected function findGet($pattern_sent)
{

    $pattern_sent = $this->parseUri($pattern_sent);

    foreach($this->routes_get as $pattern => $callback) {

        if(preg_match($pattern, $pattern_sent, $pieces))
        {
            return (object) ['callback' => $callback, 'uri' => $pieces];
        }
    }
    return false;

}


protected function findPut($pattern_sent)
{
```

```
37
38      $pattern_sent = $this->parseUri($pattern_sent);
39
40      foreach($this->routes_put as $pattern => $callback) {
41
42          if(preg_match($pattern, $pattern_sent, $pieces))
43          {
44              return (object) ['callback' => $callback, 'uri' => $pieces];
45          }
46      }
47      return false;
48
49  }
50
51
52  protected function findDelete($pattern_sent)
53  {
54
55      $pattern_sent = $this->parseUri($pattern_sent);
56
57      foreach($this->routes_delete as $pattern => $callback) {
58
59          if(preg_match($pattern, $pattern_sent, $pieces))
60          {
61              return (object) ['callback' => $callback, 'uri' => $pieces];
62          }
63      }
64      return false;
65
66  }
67
68
```

Certo, isso de certa forma já funciona! Se pedirmos para RouteCollection uma rota com um padrão existente, ela vai devolver o callback mapeado para ela, ou false.

Nossa classe de Coleção está praticamente concluída. Veja que o método parseUri, faz apenas um preparo na URI que é tido como um padrão recebido a ser comparado. A comparação ocorre com preg_match. Se o padrão casar com uma das chaves! ele devolverá o callback!

A idéia é boa Alexandre, mas alí ele está devolvendo um objeto contendo o callback e a uri. Porque?

Ótima pergunta! Esse retorno será tratado por outro protagonista: Dispacher!

Eu tenho certeza que se você leu até aqui é porque está gostando bastante e ansioso para ver essa lógica toda pronta! Ainda mais se eu lhe disser que ela funcionará de maneira semelhante a qualquer sistema de rotas usado pelos frameworks moderno!

O problema é que este artigo já está muito logo, e isso é cansativo! Não deixe de conferir parte dois em breve!

Só para aguçar ainda mais seu apetite, veja como podemos configurar algumas rotas no arquivo routes.php

```php
<?php

$router->get('/', function(){
    echo "Página inicial";
});

$router->get('/contatos', function(){
    echo "Página de contatos";
});

$router->post('/contatos/store', "Controller@store");
```