
POR ERIK FIGUEIREDO 25 DE JUNHO DE 2015 10 comentários

Entendendo o padrão MVC na prática

Exemplo prático utilizando PHP e Composer para separar o código entre as diversas camadas do MVC

Práticas modernas do PHP exigem estudo e preparação, e o padrão de projeto que merece muita atenção é o MVC. Muita gente conhece este padrão através dos *frameworks* (isso não é um problema, eu mesmo estou neste grupo), mas ir a fundo é essencial para evitar erros e falar coisas como:

> *Seu MVC está errado, o controller está maior que o model.*

Este erro de definição acontece pois em nenhum lugar está escrito que a quantidade de linhas define o padrão MVC, mas vamos entender isto melhor?

AS CAMADAS DO MVC

O QUE É *MODEL*?

Model é onde fica a lógica da aplicação. Só isso.

Vai disparar um e-mail? Validar um formulário? Enviar ou receber dados do

banco? Não importa. A *model* deve saber como executar as tarefas mais diversa, mas não precisa saber quando deve ser feito, nem como mostrar estes dados.

O QUE É *VIEW*?

View exibe os dados. Só isso.

View não é só o HTML, mas qualquer tipo de retorno de dados, como *PDF*, *Json*, *XML*, o retorno dos dados do servidor *RESTFull*, os *tokens* de autenticação *OAuth2*, entre outro. Qualquer retorno de dados para uma interface qualquer (o navegador, por exemplo) é responsabilidade da *view*. A *view* deve saber renderizar os dados corretamente, mas não precisa saber como obtê-los ou quando renderizá-los.

O QUE É *CONTROLLER*?

O *controller* diz quando as coisas devem acontecer. Só isso.

É usado para intermediar a *model* e a *view* de uma camada. Por exemplo, para pegar dados da *model* (guardados em um banco) e exibir na *view* (em uma página HTML), ou pegar os dados de um formulário (*view*) e enviar para alguém (*model*). Também é responsabilidade do *controller* cuidar das requisições (*request* e *response*) e isso também inclui os famosos *middlewares* (Laravel, Slim Framework, Express, Ruby on Rails, etc.). O *controller* não precisa saber como obter os dados nem como exibi-los, só quando fazer isso.

NA PRÁTICA

Uma sugestão aos desenvolvedores é criar seu próprio *framework* de estudo (e publicar no GitHub) mas nunca os usar em produção. Esta prática te faz compreender o quanto você conhece da linguagem, e daqui a algum tempo, ver o quanto melhorou.

Neste estudo, vamos criar uma aplicação MVC simples com PHP, usando

práticas modernas.

práticas modernas.

Para começar, vamos utilizar a ideia de que não devemos criar nada que já existe: este é o princípio da interoperabilidade buscada pelo PHP-FIG (grupo formado pelas principais empresas e grupos PHP para definir boas práticas e padrões). Utilizaremos PSR-4 e Composer para gerenciar o carregamento das classes.

Para instalar o Composer, cito uma parte do artigo Composer para iniciantes de Andre Cardoso aqui no Tableless:

- Primeiramente você precisa realizar o download do *phar* do composer.
O phar é um empacotamento de uma aplicação e é utilizado para fornecer bibliotecas e ferramentas nas quais o desenvolvedor não tem de se preocupar com sua estrutura. Em outras palavras, é pegar e usar.
- Para que você obtenha o composer há duas maneiras distintas. Através da biblioteca cURL e através do próprio PHP. Basta selecionar uma das opções abaixo e executar em seu terminal.
- Instalando via cURL:
`curl -sS https://getcomposer.org/installer | php`
- Instalando via PHP:
`php -r "readfile('https://getcomposer.org/installer');" | php`

Para saber mais sobre PSR-4 veja o guia oficial aqui.

Na raiz do diretório do seu projeto crie estes 5 arquivos (e diretórios):

- src/App/Mvc/Controller.php
- src/App/Mvc/Model.php
- src/App/Mvc/View.php
- composer.json
- index.php

Ao baixar o composer.phar (explicado acima) você também o terá no diretório raiz, junto ao composer.json e ao index.php

O seu arquivo composer.json deverá ter o conteúdo a seguir:

```
1. {
```

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

Rode o comando `php composer.phar install.`

A ideia é que o nosso *controller* carregue as informações da *model* e as envie para a *view*. Pensando nisso, faremos com que o *controller* carregue ambas as classes: *Model* e *View*. A sequência para criá-las é:

Conteúdo do arquivo src/App/Mvc/Controller.php:

```
<?php
    namespace App\Mvc;
    class Controller
    {
        ...
    }
```

Conteúdo do arquivo src/App/Mvc/Model.php:

```
<?php
    namespace App\Mvc;
    class Model
    {
        ...
    }
```

Conteúdo do arquivo src/App/Mvc/View.php:

```
<?php

    namespace App\Mvc;
```

```
3.    class View
4.    {
5.        ...
6.    }
```

Seguimos algumas regras da PSR-4: primeiro registramos um *namespace* no composer.json que vai até o diretório src/App. Toda classe tem um *namespace* e o App do começo indica o diretório que registramos (src/App). O Mvc é o diretório seguinte (ficando src/App/Mvc) e a classe tem o mesmo nome do arquivo (src/App/Mvc/Controller.php). Com isso podemos carregar as classes dinamicamente:

Conteúdo do arquivo index.php:

```
1. <?php
2.    require 'vendor/autoload.php';
3.    $controller = new App\Mvc\Controller();
```

Nossa classe ainda não faz nada, então vamos testar com algo mais concreto: no Controller.php adicione um novo método chamado index() – os métodos públicos de um *Controller* são chamados de *actions*.

```
1. <?php
2.    namespace App\Mvc;
3.    class Controller
4.    {
5.        public function index()
6.        {
7.            echo 'Olá mundo!';
8.        }
9.    }
```

E no index.php adicione no final a linha:

```
1. $controller->index();
```

Ao rodar o index.php você verá um *"Olá mundo!"* na tela. Agora vamos separar este código nas camadas do MVC.

No *model*, vamos criar o método que serve o texto em questão. Ele poderia carregar um componente que facilitaria as tarefas com o banco de dados, como o Doctrine, por exemplo, mas aqui só retorna um texto.

```
1. <?php
2.     namespace App\Mvc;
3.     class Model
4.     {
5.         public function getText($str = 'Olá mundo!')
6.         {
7.             return $str;
8.         }
9.     }
```

Na *view* vamos imprimir este texto na tela. Poderíamos carregar um *template engine* (Blade, Twig, etc.), ou até criar o nosso próprio, mas ele só fará um *echo* mesmo.

```
1. <?php
2.     namespace App\Mvc;
3.     class View
4.     {
5.         public function render($str)
6.         {
7.             echo $str;
8.         }
9.     }
```

E o *controller* intermediando tudo isso:

```
1. <?php
2.     namespace App\Mvc;
3.     class Controller
```

```
4.    {
5.        public function index()
6.        {
7.            $model = new Model;
8.            $view = new View;
9.            $view->render($model->getText());
10.        }
11.    }
```

Rode o index.php novamente e você vai obter o mesmo resultado anterior, mas agora com uma estrutura MVC.

CONCLUSÃO

Note que neste exemplo a maior classe é o *controller* (com 14 linhas) e mesmo assim não estamos "quebrando o MVC". Também não há nada de absurdo, como carregar a classe *Model* no *Controller* e passar todas as configurações gigantescas ali dentro. Mesmo que não seja uma quebra de MVC, o *Model* ainda vai cuidar de tudo. O ideal é mover o máximo de lógica para dentro da *Model*.

Apenas para reforçar, o exemplo abaixo deveria estar dentro de um arquivo de *Model*, e nunca no *Controller*:

```
1. $users = User::whereRaw('age > ? and votes = 100', [25])->get();
```

Este último exemplo foi retirado de http://laravel.com/docs/5.0/eloquent

Quanto mais organizada e centralizada a lógica, melhor. Pense nisso e comece a pesquisar Dependency Injection. Isso organiza seu código ainda mais.