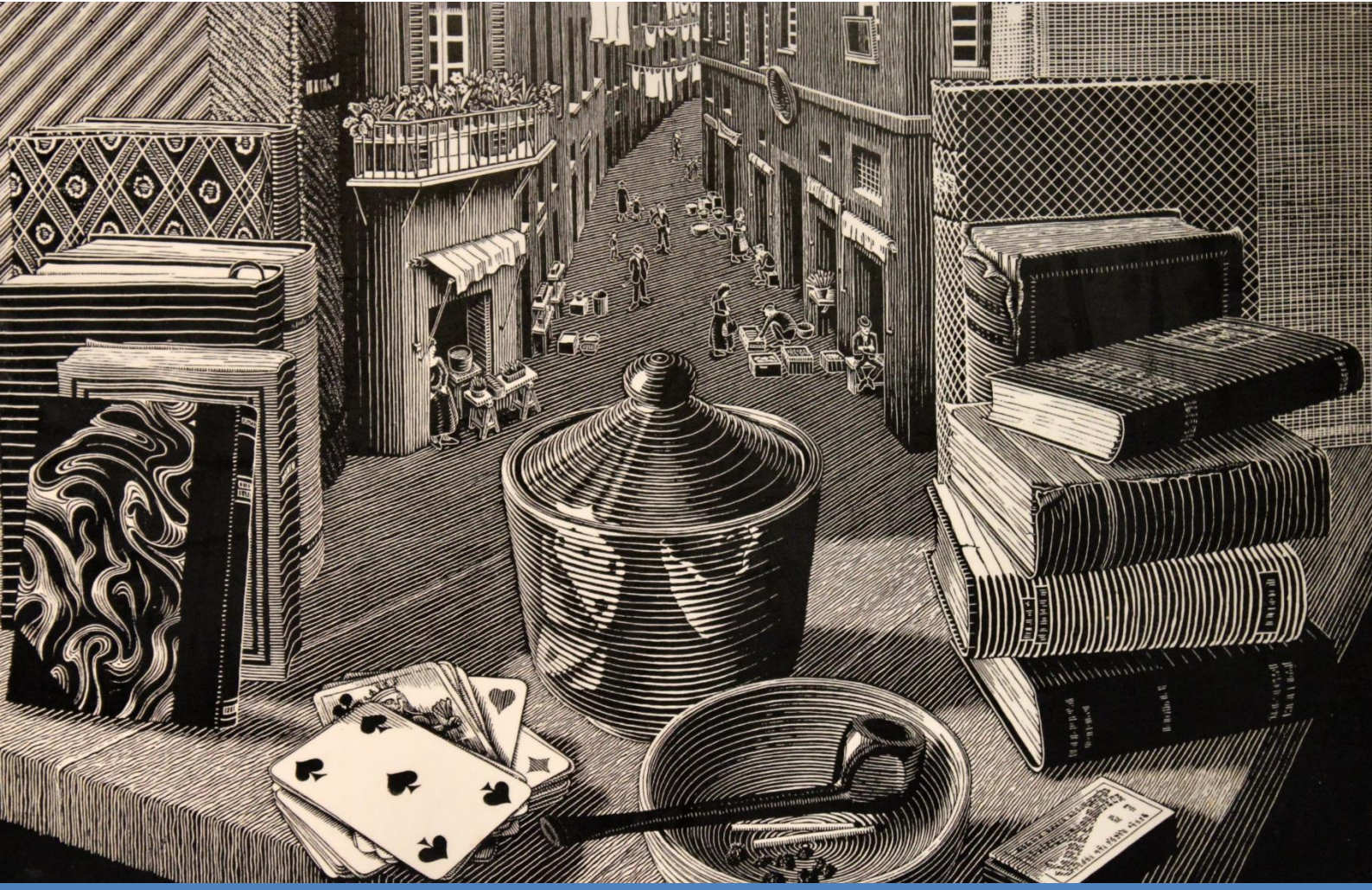

PHPVSF

PHP Very Simple Framework

Manual

Licença/License

Este software é distribuído sob a licença *BSD 3-Clause*.

Considerações iniciais

Surgido nos anos 70, o padrão de arquitetura de software MVC tem sido um dos mais utilizados e bem sucedidos na construção de aplicações com alto grau de reuso de código e separação de conceitos. Sua lógica se baseia na separação da lógica de aplicação em 3 componentes: o Modelo, a Visão e o Controlador. Cada uma desses componentes interage com o outro durante a execução da aplicação, desempenhando a sua função. Na implementação clássica do MVC, os componentes desempenham as seguintes funções:

O Modelo é uma representação dos dados da lógica de aplicação. Ele é manipulação pelos controladores e visualizado pelas visões. Quando seu estado é alterado por algum controlador, o modelo notifica a visão para que esta atualize sua representação do modelo;

A Visão é o que o usuário da aplicação vê. Ela é responsável por obter informação dos modelos e gerar uma saída para os usuários;

O Controlador é o ponto central do MVC. A utilização de uma aplicação MVC ocorre por meio da interação do usuário com os controladores, que ficam encarregados de atualizar o estado dos modelos.

Esse formato do MVC, entretanto, foi desenvolvido antes do surgimento de aplicações web; seu foco originalmente era para o desenvolvimento de interfaces gráficas no ambiente Smalltalk. Com o passar dos anos o MVC deixou de ser uma metodologia da plataforma Smalltalk e passou a ser um padrão de desenvolvimento de software genérico. Para a sua adequação não só à diferentes linguagens mas também à diferentes plataformas e ambientes, o MVC sofreu variações (e derivou novos padrões); por isso, aplicações MVC costumam não seguir o padrão tradicional a risca, devido a limitações e incompatibilidades do ambiente alvo comparado ao ambiente Smalltalk original.

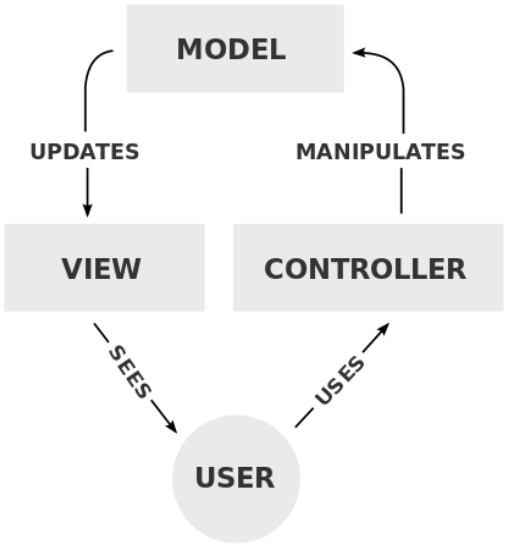

Figura 1. Diagrama representando uma interpretação mais en vogue do modelo MVC.

Na sua implementação para o ambiente web por exemplo, o padrão deve se adequar à natureza *stateless* da web, comparada à *stateful* de aplicações de interface gráfica. Os modelos nesse ambiente são geralmente acoplados ou acompanhados a *layers* de persistência, que na maioria das vezes realizam mapeamento objeto-relacional de instâncias de modelo para entradas numa SGDB e vice-versa.

Na nossa aplicação, o usuário interage com os controladores por meio de URLs. Objetos de controladores são selecionados, assim como os seus métodos, passando para a aplicação a URL correspondente a o nome do controlador e a ação desejada. Por exemplo, o controlador *UsuárioController* seria representado na aplicação por meio do objeto *UsuárioController* e as suas ações seriam métodos desse objeto. Para executar a ação de inserir um usuário, seria apenas necessário acessar a URL:

```
http://NOME_DO_SITE/UsuárioController/add
```

sendo add um método *add*() que existe dentro da classe *UsuárioController*. Caso haja necessidade de passar dados para o controlador como parâmetros da ação, eles podem ser passados pro HTTP GET; pela reescrita de URL embutida, será apenas necessário adicionar o nome do parâmetro ao fim da URL seguido pelo seu valor, como por exemplo:

```
http://NOME_DO_SITE/CONTROLADOR/AÇÃO/PARÂMETRO/VALOR
```

Caso seja necessário passar múltiplos valores para a ação do controlador como parâmetro, pode-se fazer a seguinte chamada de URL:

```
http://NOME_DO_SITE/CONTROLADOR/AÇÃO/PARÂMETRO1/VALOR1/PARÂMETRO2/VALOR2/
PARÂMETRO3/VALOR3/...
```

Esses valores serão transformados pela aplicação em array associativo PHP nos seguintes moldes:

```
['PARÂMETRO1' => 'VALOR1', 'PARÂMETRO2' => 'VALOR2', 'PARÂMETRO3'=>
'VALOR3', ...]
```

Todo parâmetro deve possuir um valor, mesmo que nulo, ou então a chamada de URL será rejeitada pela aplicação por não atender a formatação necessária. Uma vez criada a ação do controlador, o desenvolvedor pode manipular este array associativo da maneira que achar melhor dentro do método especificado do controlador.

Devido a maneira de como as páginas são servidas pelos servidores HTTP e o fato deste protocolo não guardar estado, as ações ocorrem quando páginas são requisitadas e servidas; os *requests* são processados e quando o fim do script é atingido, o "programa" é terminado com uma resposta HTTP para o cliente, portanto, os modelos no PHPVSF não atualizam as views. Num ambiente web PHP o conceito de view se difere um pouco da tradicional view utilizada na criação de interfaces gráficas. Quando um controlador é executado, este busca uma view; essa view mostrará o estado dos modelos no instante em que a ação foi requisitada e fará chamadas a controladores predefinidos. Para que a visão acesse dados atualizados dos modelos é necessário que o controlador seja executado novamente e a view seja renderizada novamente.

Requerimentos

Para o funcionamento do PHPVSF é necessário que os seguintes requisitos sejam preenchidos.

- Servidor HTTP Apache® com as seguintes extensões habilitadas
    - mod_rewrite necessário para que a reescrita e roteamento de requests funcione corretamente;
    - mod_ssl recomendado para acessar páginas de login por HTTPS;
- PHP versão 5.4 ou superior com as extensões
    - php_pdo correspondente ao SGBD que estiver usando;
- Cliente SGBD de sua preferência

Entendendo o PHPVSF

Na raiz do framework existem 2 arquivos que serão utilizados por toda a aplicação:

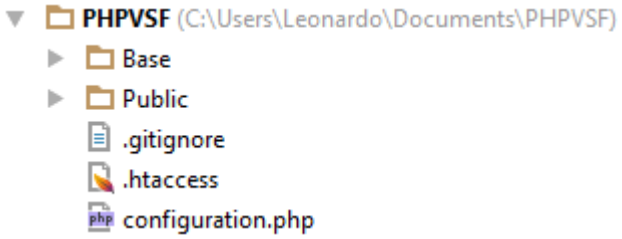

Figura 2. Diretório raiz do PHPVSF

- .htaccess – Este arquivo é utilizado pelo framework para informar vários parâmetros ao servidor HTTP Apache®. Ele realiza a reescrita de URLs *user-friendly* e redireciona todo o tráfego do site para */Public/index.php*, como forma de limitar os pontos de entrada do usuário a apenas um: o arquivo *index.php*. É necessário que você possua a extensão *mod_rewrite* instalada e habilitada no seu servidor HTTP;
- configuration.php – Este é o arquivo principal de configuração do PHPVSF. Nele estão definidas várias constantes que são utilizadas pelo framework e podem ser customizadas pelo desenvolvedor.

O *framework* se divide em 2 pastas, Base e Public. Base contém os arquivos do framework e Public deve conter os arquivos criados pelo usuário, podendo estender as interfaces e classes implementadas pelo framework em Base.

A pasta Base

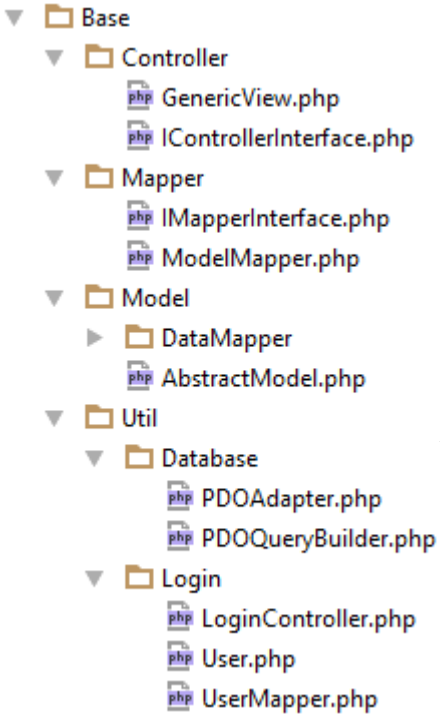

Figura 3. Estrutura de arquivos da pasta Base

Base/Model/ contém os arquivos do framework responsáveis pelo modelo. *Base/Model/AbstractModel.php* é a classe abstrata que deve ser estendida para criar o seu próprio modelo. *Base/Model/DataMapper/IDBAdapterInterface.php* é uma interface que esboça funcionalidades que devem ser implementadas por um adaptador de banco que faça o mapeamento objeto-relacional.

Base/Mapper/ contém os arquivos do framework responsáveis pelos *Mappers*. *Mappers* são classes que transformam operações de consulta em SQL em verbos simples como *find*, *delete*, etc. *Base/Mapper/IMapperInterface.php* é a interface que deve ser implementada para que o mapper seja utilizado na aplicação com os outros componentes. *Base/Mapper/ModelMapper.php* é um *mapper* que faz essa simplificação de uma forma genérica. Ele recebe o nome da tabela do banco de dados onde determinado modelo é salvo e uma instância de adaptador de banco de dados.

```
class ModelMapper implements IMapperInterface
{
    protected $databaseTable;
    protected $databaseAdapter;

    function __construct($table, IDBAdapterInterface $databaseAdapter)
```

Base/Controller é a pasta onde vão os arquivos do framework responsáveis pelos controllers. O arquivo *Base/Controller/IControllerInterface.php* é uma interface que serve de guia para criação de controladores no framework. Esta interface mapeia funções frequentemente utilizadas em aplicações CRUD (Create, Record , Update, Delete).

Base/Util é a pasta do framework onde estão contidos módulos que não são essênciais no funcionamento do framework mas pode ser utilizados para facilitar e acelerar o desenvolvimento para o desenvolvedor. *Base/Util/Database* possui classes que podem ser usadas para operações de banco de dados.

Base/Util/Database/PDOAdapter.php é um adaptador de banco que implementa a interface *Base/Model/DataMapper/IDBAdapterInferface.php* que utiliza a biblioteca PDO do PHP para criar conexões de banco de dados genéricas que manipulam diferentes SGBD de forma transparente para o programador. A esta classe pode ser usada diretamente, utilizando as operações SQL mapeadas para métodos, ou pode ser passada para um *ModelMapper*, onde este último ficará encarregado das operações SQL. A criação de um *PDOAdapter* se dá da seguinte forma:

```
public function __construct(ReflectionClass $mappedModelClass, $dsn,
$username = null, $password = null, array $driverOptions = array())
```

onde *$mappedModelClass* é uma classe de reflexão criada a partir da class de um modelo, que será usada internamente dentro do *PDOAdapter* para ler tuplas do banco de dados em objetos do modelo.

Veja pro esse exemplo de criação de controlador exemplificando estes passos descritos acima:

```
class CarroController implements IControllerInterface
{
    public function __construct()
    {
        $pdoAdapter = new PDOAdapter(new ReflectionClass("Carro"),
DB_FQDN, DB_USER, DB_PASSWORD);
        $this->mapper = new ModelMapper("Carro", $pdoAdapter);
    }
```

É criado um adaptador para o objeto Carro, logo em seguida este adaptador é passado para o ModelMapper, junto com o nome da tabela do banco onde estão as tuplas de Carro.

A pasta *Base/Util/Login* possui arquivos relacionados ao login de usuários. A classe *Base/Util/Login/User.php* é o modelo de Usuário padrão da aplicação. *Base/Util/Login/UserMapper.php* é o *ModelMapper* de usuário, que estende *ModelMapper* mas se difere desse mesmo por implementar uma função *verifyUser(array $parameters = array())* que recebe um array com os parâmetros de usuário que deve buscar e retorna *null* se não achar nenhum resultado, ou retorna uma instância de *User* caso as credenciais fornecidas estejam corretas. Para inserir um objeto de autenticação em determinada ação de controlador, basta criar em um controlador uma instância de *LoginController*. Para proteger uma ação de controlador requerendo o login do usuário, basta chamar a função *login()* da instância de *LoginController* da seguinte forma:

```
class CarroController implements IControllerInterface
{
public function __construct()
{
    $this->loginManager = new LoginController();
}
```

```php
public function find($id)
{
    $this->loginManager->login();
```

Caso o usuário não seja autenticado, ele sempre será redirecionado para a tela de login, correspondente a URL *http://NOME_DO_SITE/LoginController/userAuth* caso o contrário, sua credencial será colocada na superglobal *$_SESSION['login']* podendo ser acessada por toda a aplicação, e o usuário poderá continuar a sua navegação em áreas que requerem o seu login.

Public/

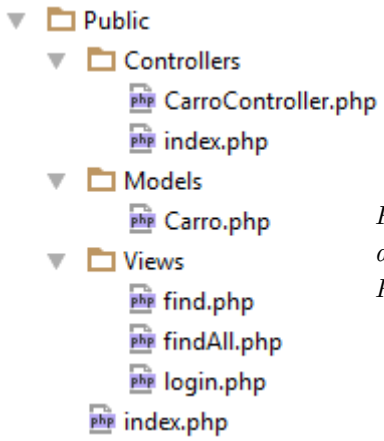

Figura 4. Exemplo de como os arquivos são dispostos na pasta Public

Os controladores do usuário vão na pasta *Public/Controllers*; todos os controladores presentes nessa pasta serão automaticamente importados no arquivo *Public/index.php*. Os modelos do usuário devem ser colocados na pasta *Public/Models* e suas views na pasta *Public/Views*.

Os controladores também implementam uma função *home()*, que é um *fallback* caso nenhuma ação seja informada na chamada de URL ao controlador. Essa função home será mostrada por padrão, ficando a cargo do programador implementar o que desejar em home.

É importante que os modelos criados pelo usuário possuam o mesmo nome para os atributos e as colunas da tabela do banco de dados, caso contrário não será possível utilizar as classes de persistência.

Como a visão é selecionada pelo controlador, o programador poderia usar funções *echo* do PHP dentro dos métodos do controlador gerando tags HTML, entretanto isso criaria um código muito confuso no controlador e seria ruim para a separação de conceitos e reuso. Uma forma interessante de mostrar as views aos usuários seria realizando o processamento necessário dentro do controlador, colocando se necessário os dados no em uma variável, fazendo o include num arquivo de view e acessando as variáveis de mesmo nome criadas no controlador dentro do arquivo da view, como no exemplo abaixo:

```
class CarroController implements IControllerInterface
{
public function findAll(array $parameters = array())
{
    $objects = $this->mapper->findAll($parameters);
    include(PATH . "/Public/Views/findAll.php");
```

Esse processamento ocorrerá dentro de *findAll()* e a variável *$objects* será criada. Como a visão é incluída dentro da função do controlador, todas as variáveis no escopo de *findAll()* também estarão no escopo da view *findAll.php*. Assim, o controlador completa o ciclo de ações MVC: buscar ou atualizar os modelos e gerar a view de acordo com o

estado dos modelos. Para deixar mais claro, vamos mostrar como manipular *$objects* dentro da view *findAll:*

```
[...]
<body>
<?php
$objectClass = $objects[0]->getClassName();
echo "<h2>$objectClass</h2><br />";
echo "<table>";
foreach ($objects as $object)
{
    echo "<tr>";
    $objectId = $object->getId();
    foreach ($object->toArray() as $variableName => $variableValue)
    {
        echo "<td>$objectId</td>";
        echo "<td>$variableName</td>";
        echo "<td>$variableValue</td>";
    }
    echo "</tr>";
}
echo "</table>";
?>
</body>
[...]
```

O arquivo /Public/index.php será fornecido pela aplicação, já que é o principal arquivo do PHPVSF responsável pelo roteamento de URLs, criação e delegação de controles, e gerenciador de sessão PHP.

Outra funcionalidade do PHPVSF é que mesmo com as reescritas de URL e o carregamento dinâmico de controladores, o comportamento de POST do PHP não é alterado; é possível usar as variáveis *$_POST* da mesma forma que elas seriam utilizadas normalmente da maneira tradicional que seria um script por URL. Tudo que o programador precisa para utilizar as variáveis *$_POST* é selecioná-las dentro das ações dos controladores ou da visão, analogamente ao método de um script por URL.