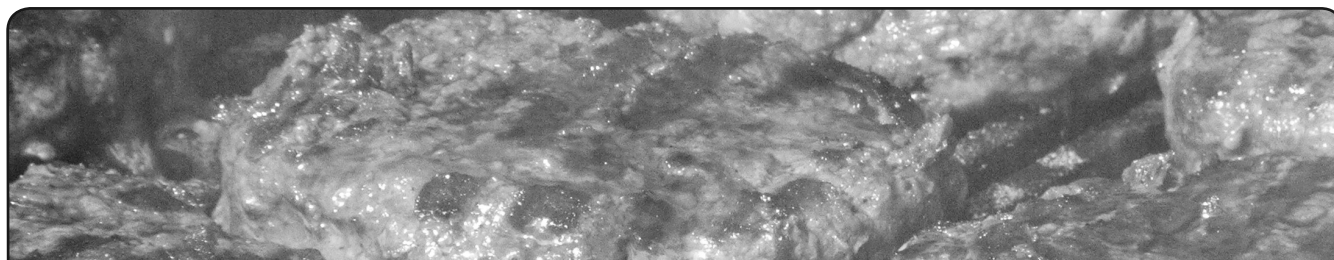


## Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP



### Programando em PHP usando MVC

Aprenda técnicas de Orientação a Objetos juntamente com o conceito MVC para criar aplicações em PHP

Muita gente já ouviu falar alguma vez sobre esta metodologia de desenvolvimento. Mas afinal, o que vem a ser uma aplicação desenvolvida em MVC? Neste artigo, irei explicar de uma forma sucinta sobre este tipo de arquitetura de desenvolvimento e como este método pode ser aplicado utilizando a linguagem PHP.

MVC vem da sigla (Model-View-Controller) e é um padrão de arquitetura de software. Em aplicações complexas, que enviam uma série de dados para o usuário, o desenvolvedor frequentemente necessita separar os dados ("Model") da interface ("View"). Desta forma, alterações feitas na interface não afetarão a manipulação dos dados, e estes poderão ser reorganizados sem alterar a interface do usuário. O *model-view-controller* resolve este problema através da separação das tarefas de acesso aos dados e lógica do negócio da apresentação e interação com o usuário,

introduzindo um componente entre os dois: o *Controller*. MVC é usado em padrões de projeto de software, mas MVC abrange mais da arquitetura de uma aplicação do que é típico para um padrão de projeto.

A **Figura 1** representa um diagrama simples exemplificando a relação entre *Model*, *View* e *Controller* onde linhas sólidas indicam associação direta e as tracejadas indicam associação indireta. É comum dividir a aplicação em camadas separadas: apresentação ("interface"), domínio e acesso a dados. Em MVC a camada de apresentação também é separada da *view* e da *controller*.

MVC é muito visto também em aplicações para Web, onde a *View* é geralmente a página HTML, e o código que gera os dados dinâmicos para dentro do HTML é o *Controller*. E, por fim, o *Model* é representado pelo conteúdo de fato, geralmente armazenado em bancos de dados ou arquivos XML.



**Daniel Ribeiro**

([daniel@danielribeiro.com](mailto:daniel@danielribeiro.com))

analista de Sistemas e especialista em desenvolvimento avançado em PHP há 6 anos. Bacharel em Ciências da computação pela Universidade do Grande ABC. Desenvolvedor e coordenador de projetos do grupo Sonda Procwork.

## Usando Objetos para entender o conceito aplicado de MVC

Muita gente me pergunta sobre a maneira que eu programo usando MVC com PHP. É claro que já existem vários *frameworks* que utilizam essa técnica no desenvolvimento PHP. Porém resolvi apresentar aqui o conceito do que uso que pode servir de material de auxílio para outros programadores também.

Para se ter uma idéia, vamos partir para o modelo de programação orientada a objetos. A maneira mais prática de se programar usando MVC nada mais é do que programar usando orientação a objetos. Vamos pensar então no conceito de objeto. Um exemplo, um veículo, onde teremos a classe de veículo e seus métodos (**Listagem 1**).

Agora digamos que temos vários veículos, tais como: carros, motos, bicicletas e cada um terá eventos diferentes um do outro. Para isso criamos novas classes estendendo a classe *Veiculo* no qual as novas classes irão herdar os métodos e propriedades da classe *Veiculo*, como podemos ver a seguir na **Listagem 2**.

Desta forma quando instanciarmos o objeto *Bicicleta*, podemos utilizar os métodos de *Veiculo*, pois *Bicicleta* é uma extensão de *Veiculo*, como no código seguinte:

```
<?php
$meuVeiculo = new Bicicleta();
$meuVeiculo->trocarMarcha();
$meuVeiculo->andar();
$meuVeiculo->parar();
?>
```

Agora vamos construir outra classe *Automovel* que instancia *Veiculo* (**Listagem 3**). Desta forma quando instanciarmos o objeto *Automovel*, podemos utilizar os métodos de *Veiculo*, pois *Automovel* é uma extensão de *Veiculo*, como podemos na **Listagem 4**.

Percebeu que desta forma, se quisermos mudar o comportamento de um veículo (independente do que ele seja) basta modificar a classe, que todos os tipos de veículos serão alterados (pois eles estendem a classe *Veiculo*). Isso facilita muito na manutenção do código.

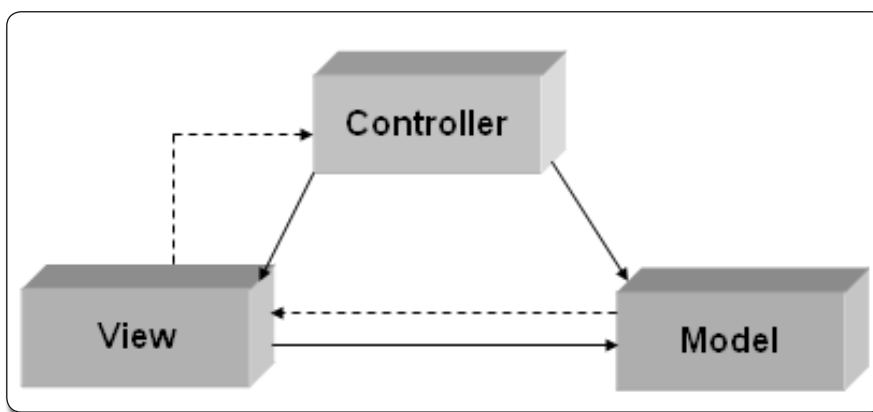
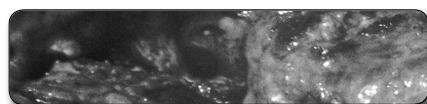


Figura 1. Representação gráfica do conceito MVC

### Listagem 1. Exemplo de classe em PHP

```
<?php
class Veiculo{

    /* Qualquer veículo tem os seus métodos padrões: */
    public function andar(){

    /* Programação da função */
    }

    public function parar(){

    /* Programação da função */
    }

}
?>
```

### Listagem 2. Classe veículo estendida

```
<?php
class Bicicleta extends Veiculo{

    /* A bicicleta por sua vez contem propriedades */
    public $rodas;

    /* E tbm tem um metodo construtor */
    public function __construct(){
        $this->rodas = 2;
    }

    /* E tbm seus métodos */
    public function trocarMarcha(){

    /* Programação da função */
    }

}
?>
```

### Listagem 3. Classe Automóvel que vem de Veículo

```
<?php
class Automovel extends Veiculo{
    public function ligar(){

    /* Programação da função */
    }

    public function desligar(){

    /* Programação da função */
    }

}
?>
```

### Listagem 4. Criação do objeto Automovel

```
<?php
$meuVeiculo = new Automovel();
$meuVeiculo->ligar();
$meuVeiculo->andar();
$meuVeiculo->parar();
$meuVeiculo->desligar();
?>
```

## Criando sua aplicação em MVC

Pois bem, onde queremos chegar com tudo isso? Uma aplicação MVC nada mais é do que classes principais de:

- (*M*)odel - Utilizada como o negócio da sua aplicação, (tudo o que for utilizado para armazenamento de dados ou obtenção dos dados da aplicação);
- (*V*)iew - Utilizada para exibir para o usuário tudo o que a aplicação produz;
- (*C*)ontroller - Utilizada para realizar o controle integrando o *Model* e *View* (é aqui e controlamos o fluxo da aplicação e as ações do usuário);

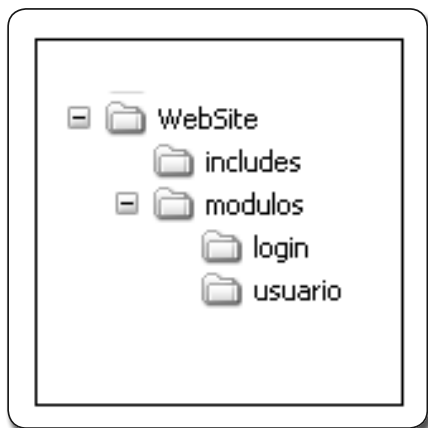


Figura 2. Estrutura de pastas de nosso exemplo

Bem, se sabemos que o *Model* interage com o banco de dados, então criamos uma classe *model* principal com os métodos principais que ele tem, veja na **Listagem 5**. O código dado é somente um mero exemplo, pra você ter uma idéia. É claro que em uma *Model* existem mais métodos para acessar os dados, dependendo da sua aplicação. Crie uma estrutura de pastas para seu site como mostrado na **Figura 2**. Em seguida abra o Bloco de Notas ou outro editor de textos a sua escolha e digite o código da **Listagem 5**. Salve esse arquivo dentro da pasta \WebSite\includes com o nome "Model.php". Agora temos a classe *View* na **Listagem 6**. Fiz questão de comentar todas as listagens para que entendamos melhor cada função, cada declaração e cada implementação.

Se você perceber, o *Smarty* segue um critério semelhante, não sei se você já usou *Smarty*, mas ele funciona como na **Listagem 7**. Já que temos métodos semelhantes, podemos então aplicar em nossa *View* todos os métodos do *Smarty*, para não ter que se preocupar em trabalhar com manipulação de *templates*. Veja como ficaria nossa *View* baseando-se em *Smarty* na **Listagem 8**. Digite o código

da **Listagem 8** em um novo arquivo e salve-o como "View.php".

Talvez você esteja se perguntando: "Se vou usar o *Smarty*, por que então tenho que criar uma *View* e usar os métodos do *Smarty* dentro da *View*? Não poderia usar diretamente o *Smarty*?"

Poderia, mas digamos que você posteriormente não queira mais utilizar o *Smarty*, e sim outro gerenciador de *templates*. Você teria nesse caso que mexer em sua aplicação inteira? Fazendo uma classe dessa forma, você só mexe na *View*, e programa *Ela* pra usar outro gerenciador de *templates*.

Passamos para a classe *Controller*, que contém os métodos básicos que farão a sua aplicação funcionar. Até mesmo métodos genéricos que você utiliza dentro da sua aplicação. Veja o código da **Listagem 9**. Novamente crie um novo arquivo e digite o código da **Listagem 9** que encontra-se comentado.

Resumindo, a classe *Controller* irá utilizar métodos que são genéricos para todos os módulos de sua aplicação. Agora que já temos o *Model*, *View* e *Controller*, vamos criar os módulos de nossa aplicação usando MVC.

### Listagem 5. Criação da classe Model

```
<?php
class Model{
    public function conectar(){
        /* Código pra conectar-se ao banco de dados */
        $conexao = mysql_connect("localhost", "database", "senha");
        mysql_select_db("meu_database");
    }
    public function query($tabela){
        /* Código pra executar uma query */
        $sql = "SELECT * FROM " . $tabela;
        $query = mysql_query($sql);
    }
    public function desconectar(){
        /* Código pra desconectar do banco */
        mysql_close($conexao);
    }
}
?>
```

### Listagem 6. Criação da classe View

```
<?php
class View{
    public function mostrarNaTela($template){
        /*
        Mostra na tela usando o template escolhido,
        aqui vem o código pra ele dar include em
        algum arquivo html modificando os valores
        que já estão atribuídos a View etc...
        */
    }
    public function atribuirValor($var, $valor){
        /* Código pra atribuir valores na view */
    }
}
?>
```



## Nota do DevMan

### O que é Smarty?

O *Smarty* é uma classe de *templates*. Funciona de uma forma que separe interface da lógica de programação e tem por objetivo, facilitar e melhorar o desenvolvimento de qualquer aplicação em PHP.

Por ser muito difundido no mundo inteiro, e estar ligado ao site oficial do PHP, o *Smarty* tem uma comunidade grande de desenvolvedores. Isso ajuda no suporte e discussão de melhorias.

A sua última versão é a 2.6.11, lançada em Dezembro de 2005.

### Principais funcionalidades

- \* Melhor desempenho de execução;
- \* Acaba com overhead de template, pois compila apenas uma vez;
- \* Você pode criar suas próprias funções, é extensível;
- \* Você configura a sintaxe da tag de delimitação do template. Assim pode-se usar {}, {{}}, <!-->, etc;
- \* Uso de funções condicionais de forma simplificada dentro dos templates;
- \* Sem limitação para loops, ifs, etc;
- \* Modo de cache embutido;
- \* Arquitetura de plugins.

## Criando módulos estendendo suas classes MVC

Um dos módulos é o de *login*, então teremos o *loginModel.php*, *loginView.php* e *loginController.php*. Cada um desses serão classes que estenderão suas respectivas “classes mãe”. Agora sim iremos para a classe do controlador, que irá interagir com *loginModel.php* e *loginView.php*. Veja o código de cada classe nas **Listagens 10, 11 e 12**. O que faremos é separar cada Listagem em um arquivo a parte, como pode notar. Portanto **Listagem 10** será *loginModel.php*, **Listagem 11** igual a *loginView.php* e *loginController.php* conterá a **Listagem 12**. Todos esses arquivos deverão ser salvos na pasta `\WebSite\modulos\login`. Note que no início de cada script temos uma chamada à função *include* do PHP. Isso significa que estamos incluindo o código dos arquivos *Model.php*, *Controller.php* e *View.php* que possuem as definições das *classes mãe*.

**Nota:** Para fins didáticos, simplificamos ao máximo a estrutura das classes e também por estarem fora do escopo desse artigo, alguns métodos não foram implementados, tais como *gravaSessao*, *apagaSessao* e *log*.

Agora que já temos o nosso controlador *loginController.php*, temos que fazer com que ele seja instanciado e chamado pelo usuário, para isso teremos um arquivo de *login* onde irá fazer a instância do nosso controlador. Algo como “*login.php*”, este arquivo será chamado pelo navegador, onde fará a requisição da sua classe.

## Requisitando os módulos em sua aplicação

Pensando da maneira como foi descrito anteriormente, você terá que fazer um arquivo para cada módulo do seu sistema, se o seu sistema tiver vários módulos, então terá vários arquivos:

- *login.php*;
- *usuarios.php* (módulo de controle de usuários);
- *produtos.php* (módulo de controle de produtos);

Se formos ver, todos os arquivos que fazem as requisições de suas classes

### Listagem 7. Aplicação do Smarty

```
<?php
$smarty = new Smarty();
/* Em nossa View o método atribuirValor() */
$smarty->assign('var1', $var1);
$smarty->assign('var2', $var1);
/* Em nossa View o método mostrarNaTela() */
$smarty->display('template.tpl');
?>
```

### Listagem 8. View baseada em Smarty

```
<?php
class View{
    /*
        Declaramos a propriedade Smarty aqui dentro da nossa View (veja que é uma propriedade
        private, pois somente a classe View pode acessá-la.
    */
    private $smarty;
    public function __construct(){
        /*
            Quando então for chamado o construtor da classe, ele instancia o objeto Smarty.
        */
        $this->smarty = new Smarty();
    }
    public function mostrarNaTela($template){
        /*
            Chama o método do smarty pra exibir o template na tela.
        */
        $this->smarty->display($template);
    }
    public function atribuirValor($var, $valor){
        /*
            Chama o método do smarty para atribuir o valor a variável.
        */
        $this->smarty->assign($var, $valor);
    }
}
?>
```

### Listagem 9. Código de criação do Controller

```
<?php
class Controller{
    public function redirect($url) {
        header('Location: ' . $url);
    }
    public function error($error){
        /*
            Se algum erro acontecer, chamamos esse método, que vai fazer alguma coisa (mandar
            um email, gravar log, etc
        */
        $this->log($error);
    }

    public function log($mensagem){
        /*
            Grava uma mensagem de log em algum arquivo qualquer definido aqui
        */
    }
}
?>
```

### Listagem 10. Criação da classe loginModel herdada de Model

```
<?php
require_once('includes/Model.php');
class loginModel extends Model{
    /*
        * Método para verificar a senha do usuario
        * @param string $usuario
        * @param string $senha
        * @return bool
    */
    public function verificarSenhaUsuario($usuario,
        $senha){
        $this->conectar();
        $sql = "SELECT * FROM usuarios WHERE usuario =
            $usuario AND senha = $senha";
        $resultado = $this->query($sql);
        $this->desconectar();
        return $resultado;
    }
}
?>
```

#### Listagem 11. View da classe Login

```
<?php
require_once('includes/View.php');
class loginView extends View{
    /* Exibe a tela de login */
    public function exibirTelaLogin(){
        $this->mostrarNaTela('login.tpl');
    }
    /* Exibe a tela de erro */
    public function exibirTelaErro(){
        $this->mostrarNaTela('erro.tpl');
    }
    public function exibirTelaLogado(){
        $this->mostrarNaTela('logado.tpl');
    }
}
?>
```

#### Listagem 12. Controller do Login

```
<?php
class loginController extends Controller{
    /*
     * loginController vai agregar loginModel e loginView, então criamos esses objetos aqui
     * @var loginModel
     */
    private $model;
    /* @var loginView */
    private $view;
    public function __construct(){
        /* Instanciamos os objetos */
        $this->model = new loginModel();
        $this->view = new loginView();
    }
    public function telaLogin(){
        /* O método contrutor já chama a view do login pra exibir a tela de login */
        $this->view->exibirTelaLogin();
    }
    public function fazerLogin(){
        /* Tenta ver se o usuário colocou a senha certa */
        if ($this->model->verificarSenhaUsuario(
            $_POST['usuario'], $_POST['senha'])){
            $this->gravaSessao();
            $this->log('usuário logou');
            $this->view->exibirTelaLogado();
        } else {
            $this->log('Usuário tentou logar mas não conseguiu');
            $this->view->exibirTelaErro();
        }
    }
    /* Esse método é private, significando que só pode ser chamado dentro da classe */
    private function gravaSessao(){
        /* Grava a sessão usando session ou cookie */
    }
    /* Esse método é private, significando que só podem ser chamando dentro da classe */
    private function apagaSessao(){
        /* Limpa a sessão ou cookie */
    }
}
?>
```

#### Listagem 13. Código do arquivo login.php

```
<?php
$objLogin = new loginController();
/*
 * Verificamos que método o browser está chamando.
 * Se não chamar nenhum método, exibimos a tela de login. Desta forma se for chamado a
 * url http://www.seusite.com.br/login.php ele vai chamar o método padrão telaLogin
 */
if (!$_REQUEST['action']){
    $_REQUEST['action'] = 'telaLogin';
}
/* Agora executamos o método passado via url */
eval('$objLogin->' . $_REQUEST['action']);
?>
```

#### Listagem 14. Código do arquivo main.php com chamadas generalizadas aos módulos

```
<?php
/* Instância o módulo */
eval('$instancia = new ' . $_REQUEST['modulo'] . 'Controller()');

/* Define uma acao default */
if (!$_REQUEST['action']) {
    /* Isso implica que todos os controllers terão que ter um método chamado acaoPadrao */
    $_REQUEST['action'] = 'acaoPadrao';
}
/* Agora eu executo o método passado via url */
eval('$instancia->' . $_REQUEST['action']);
?>
```

são semelhantes, então por que não generalizar e fazer em um arquivo? Um *script* que pode ser usado para chamar qualquer módulo do sistema? Mãos à obra. Crie um novo arquivo PHP, dessa vez na pasta raiz de nossa aplicação, e salve-o como “main.php”. Nele digite o código da **Listagem 14**. Agora você tem um arquivo que pode chamar qualquer módulo/método do seu sistema, bastando especificar na url.

```
http://www.seusite.com.br/main.
php?module=login
http://www.seusite.com.br/main.
php?module=usuario
http://www.seusite.com.br/main.
php?module=produtos
```

## Conclusão

O que escrevi aqui tem como principal objetivo mostrar, de forma didática, como construir aplicações usando o modelo MVC. É claro que depois de entender perfeitamente o conceito, tudo fica mais fácil e torna-se perfeitamente possível aperfeiçoar ainda mais sua aplicação personalizando, se necessário, do jeito que melhor se adapte às necessidades de seus projetos.

Existem muitos *frameworks* que auxiliam na codificação de suas aplicações usando a arquitetura MVC. Para mais referências veja alguns deles na seção Links. ●

#### Links

**CakePHP** - Framework MVC para PHP  
<http://www.cakephp.org/>

**CodeIgniter** - Framework MVC para PHP  
<http://www.codeigniter.com/>

**Framework PHP MVC** - Framework em PHP no padrão MVC  
<http://www.phpmvc.net/>

**Prado** - Framework MVC para PHP  
<http://www.xisc.com/>

**Symfony** - Framework MVC para PHP  
<http://www.symfony-project.com/>

**Zend Framework** - Framework em PHP no padrão MVC  
<http://framework.zend.com/>

#### Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

[www.devmedia.com.br/clubedelphi/feedback](http://www.devmedia.com.br/clubedelphi/feedback)

