# MVC com PHP puro

# Índice

# Advertência

Esta apostila é fruto de pesquisa por vários sites e autores e com alguma contribuição pessoal. Geralmente o devido crédito é concedido, exceto quando eu não mais encontro o site de origem. Isso indica que esta apostila foi criada, na prática, "a várias mãos" e não somente por mim. Caso identifique algo que seja de sua autoria e não tenha o devido crédito, poderá entrar em contato comigo para dar o crédito ou para remover: ribafs @ gmail.com (remova os dois espaços).

É importante ressaltar que esta apostila é distribuída gratuitamente no repositório:

https://github.com/ribafs/apostilas



**Sugestões**

Sugestões serão bem vindas, assim como aviso de erros no português ou no PHP com MVC. Crie um issue no repositório ou me envie um e-mail  ribafs @ gmail.com.

**Como ser um bom programador**

É fácil seguir uma apostila como esta e criar aplicativos, pois ela foi testada e provavelmente não aparecerá erros. Mas não tem jeito, eles sempre aparecem. As mensagens de erro geralmente ajudam mas nem sempre são precisas. Neste momento precisamos ter um espírito tipo detetive e sair rastreando o problema pelas trilhas mais prováveis até encontrar o erro e corrigir.

É nestes momentos que a programação mostra para alguns que eles estão no lugar errado e que devem fazer outra coisa da vida.

Caso realmente queira ser programador, empenhe-se, organize-se e siga seus instintos de detetive.

Costumo dizer que pegar um exemplo e executar é fácil, mas que nem sempre é fácil corrigir erros. Mas um programador não desiste com dificuldades, elas o estimulam a corrigi-las, tornando-se assim melhores.

# Autor

**Ribamar FS**

ribafs @ gmail.com

https://ribamar.net.br

https://github.com/ribafs

---

Fortaleza, 18 de dezembro de 2021

# Recomendações

O conhecimento teórico é importante para que entendamos como as coisas funcionam e sua origem, mas para consolidar um conhecimento e nos dar segurança precisamos de prática e muita prática.

Não vale muito a penas ser apenas mais um programador. É importante e útil aprender muito, muito mesmo, ao ponto de sentir forte segurança do que sabe e começar a transmitir isso para os demais.

Tenha seu ambiente de programação em seu desktop para testar com praticidade todo o código desejado.

Caso esteja iniciando MVC com PHP recomendo que leia por inteiro e em sequência. Mas se já entende algo dê uma olhada no índice e vá aos assuntos que talvez ainda não domine.

Não esqueça de ver e testar também o conteúdo da pasta Anexos do repositório.

Caso tenha alguma dúvida, me parece que a forma mais prática de receber resposta é através de grupos. Depois temos o Google.

**Dica:** quando tiver uma dúvida não corra para pedir ajuda. Antes analise o problema, empenhe-se em entender o contexto e procure você mesmo resolver. Assim você está passando a responsabilidade para si mesmo, para seu cérebro, que passará a ser mais confiante e poderá te ajudar ainda mais. É muito importante que você confie em si mesmo, de que é capaz de solucionar os problemas. Isso vai te ajudar. Somente depois de tentar bastante e não conseguir, então procure ajuda.

Veja que este material não tem vídeos nem mesmo imagens. Isso em si nos nossos dias é algo que atrai pouca gente, pois vídeos e fotos são mais confortáveis de ler e acompanhar. Ler um texto como este requer mais motivação e empenho. Lembrando que para ser um bom programador precisamos ser daqueles capaz de se empenhar e suportar a leitura e escrita também.

**Autodidata**

Não tive a pretensão de que esta apostila fosse completa, contendo tudo sobre MVC, que seria praticamente impossível. Aquele programador que quando não sabe algo seja capaz de pesquisar, estudar, testar e resolver praticamente sozinho. Este é um profissional importante para as empresas e procurado.

# 1 – Introdução

Criação de um aplicativo em PHP puro usando MVC

**Obs**
Esta apostila descreve como criei o aplicativo simplest-mvc e passa algumas informações relativas. Não estarei explicando com detalhes o código, pois aqui o objetivo é criar/monta um aplicativo em PHP mas usando a arquitetura MVC com alguns detalhes sobre como namespace, router, etc.
https://github.com/ribafs/simplest-mvc

Um aplicativo bem simples, sem nenhuma pretensão de ser usado em produção.
A intenção é somente a de mostrar detalhes do funcionamento de um aplicativo em PHP que usa MVC, como fazem os grandes frameworks.

Quando quiser usar em produção a boa recomendação é escolher um dos bons frameworks PHP.

**Recursos:**
- Bons padões de codificação
- PHPOO com MVC de forma organizada
- Rotas
- PSR-4 com autoload, namespace e composer
- Front controller
- BootStrap 4
- CRUD
- Busca
- Form master/details

Detalharei a seguir a criação de alguns arquivos

Detalhe: este tutorial partiu do aplicativo https://github.com/ribafs/simplest-mvc , parte 12.

**Testado em**

- Windows 10 com PHP 8
- Linux Mint 20.1 com PHP 7.4

Usarei uma única tabela, clientes, com apenas dois campos, nome e idade, além do id, para facilitar o entendimento e a digitação.

**Justificativa**

Inicialmente fiz uma longa pesquisa via google e também perguntei em alguns grandes grupos de PHP internacionais por indicação de tutorial para a criação do meu próprio tutorial. As respostas são divididas, parte defende com paixão que se crie seu próprio framework e parte condena, dizendo que devemos usar os existentes que são muito bons. Alguns até citam estatísticas de que os frameworks atuais atendem a 95% das necessidades (não sei como conseguiram este número). Muitos dos tutoriais e vídeo aulas que encontrei estão desatualizados, uns ainda usando as funções mysql_connect, outros usando o mysqli, outros que não usam o PSR-4 e na maioria que não executa corretamente. Acusa erro. Assim também com os exemplos de pequenos frameworks que encontrei. Resumindo, somente o mini3 realmente executou bem, tem uma estrutura simples que me permitiu

entender, usa o PSR-4, o PDO, tem também rotas e em apenas uma classe e .htaccess e não tem nenhuma dependência de pacote de terceiros. Aí a vantagem do software livre e open source, mesmo que não tenha um tutorial detalhado explicando como foi criado o aplicativo, como ele é aberto, se estivermos preparados podemos elaborar este tutorial e é o que irei fazer. Lembrando disso, voltei novamente ao repositório do mini3 e agradeci o autor, clicando na estrela e também fiz um fork novamente, pois havia excluído, para garantir que terei seu código por perto. Também mantive o rodapé do original, onde o autor faz a propaganda de uma hospedagem, pois acho que ele merece que eu faça isso. O grupo que defende a criação do seu próprio framework é muito enfático e me parece lógico, pois se ninguém mais quisesse criar o seu não haveria mais progresso.

Outra sugestão muito apontada é a criação de um aplicativo/framework usando componentes de outros grandes frameworks. Veja que muitos grandes frameworks como Laravel, CakePHP, etc usam componentes do Symfony. Inclusive encontrei um tutorial muito bom, onde o autor ensina a criar seu próprio framework apenas com componentes do Symfony2. Inclusive foi traduzido para o português e está abaixo nas referências. Não senti muita atração por esta alternativa, preferi partir do mini3 e, como o compreendi, vou adicionar minhas classes e outros recursos.

Tutorial de criação de um pequeno aplicativo em PHP com MVC e outros bons recursos. É um aplicativo muito simples e bem documentado, contendo 3 dependências mas todas opcionais. Portando pode ser modificado para atender outras necessidades.

## 1.1 - Introdução ao MVC

O MVC (Model View Controller) é um dos padrões de arquitetura mais utilizados atualmente. A maioria dos grandes frameworks e CMS o utilizam para separar o código em camadas lógicas. Cada camada  tem uma resposabilidade. Veja abaixo.

**Model**
Representa a letra M do MVC. Nesta camada são realizadas as operações de validação, leitura e escrita de dados no banco de dados. É responsável por salvar e receber dados do banco de dados, como também efetua diversos processamentos com os dados.

Basicamente qualquer coisa para ler, alterar, salvar ou excluir dados é nesta camada. A camada Model é a camada que sofreu a maior transformação na versão 3.

Uma boa prática é trazer para esta camada tudo que diz respeito às regras de negócio, como cálculos ou validações de integridade de dados.

**Controller**
É o responsável pela integração entre as camadas Model e View. Basicamente a View irá realizar uma solicitação para o Controller como por exemplo uma coleção de dados ou a solicitação de remover algum item do banco e o Controller, por sua vez, irá enviar a instrução para a camada Model executar.

Os controllers correspondem ao ‘C’ no padrão MVC. Após o roteamento ter sido aplicado e o controller correto encontrado, a ação do controller é chamada. Seu controller deve lidar com a interpretação dos dados de uma requisição, certificando-se que os models corretos são chamados e a resposta ou view esperada seja exibida. Os controllers podem ser vistos como intermediários entre a camada Model e View. Você vai querer manter seus controllers magros e seus Models gordos. Isso lhe ajudará a reutilizar seu código e testá-los mais facilmente.

Mais comumente, controllers são usados para gerenciar a lógica de um único model. Por exemplo, se você está construindo um site para uma padaria online, você pode ter um RecipesController e um IngredientsController gerenciando suas receitas e seus ingredientes. No CakePHP, controllers são nomeados de acordo com o model que manipulam. É também absolutamente possível ter controllers que usam mais de um model.

Os controllers fornecem uma série de métodos que são chamados de ações. Ações são métodos em um controller que manipulam requisições. Por padrão, todos os métodos públicos em um controller são ações e acessíveis por urls.

Nesta camada (Controller) também podemos realizar verificações que não se referem às regras de negócio, visto que a boa prática é manter as regras de
negócio no Model.

**View**
Representa a letra V do MVC. É a camada responsável por tudo que é visual, páginas, formulários, listagens, menus, o HTML em geral. Tudo aquilo que interage com o usuário deve estar presente nesta camada. Representadas por HTML.

A View não realiza operações diretamente com o banco de dados nem trata diretamente com o Model. Ela as solicita e e exibe através do Controller, que
intermedia suas solicitações com o Model.

**Fluxo das Informações no MVC**
- Geralmente Nascem na View quando um usuário faz uma solicitação, clicando num botão submit ou num link ou entrando um link diretamente no navegador
- Daí são enviadas para o Controller, que a filtra (se for o caso) e a envia para o Model
- O Model analisa de acordo com a solicitação (uma consulta ao banco) e a devolve ao Controller
- O Controler por sua vez devolve o resultado para a View
- E a View renderiza o resultado e o mostra para o usuário

Abordagem sobre as 3 camadas: [C]ontroller, [V]iew e [M]odel

Um exemplo bem organizado de uso do MVC é o Framework CakePKP, que traz as 3 camadas bem definidas e organizadas.

Model - representa os dados. A parte do código que manipula os dados para ler e escrever no banco.
View - representa a visualização dos dados. A parte do código que mostra os dados para o usuário.
Controller - manipula e roteia as requisições dos usuários. A parte do código que recebe as requisições do usuário através de um site (exemplo), processa, roteia e envia para o model, se for o caso. Também é responsável por receber dados do banco e devolver para a View.

De forma mais completa o fluxo das informações entre as 3 camadas acontece assim no CakePHP:
- O usuário clica num link para editar um registro
- O dispatcher (expedidor) verifica a URL requisitada (/cakes/comprar) e redireciona ao controller correto;
- O controller executa a lógica específica da aplicação. Por exemplo, verifica se o Ricardo está logado e tem acesso ao site;

- O controller também usa os models para acessar os dados da sua aplicação. Muitas vezes, os models representam as tabelas do banco de dados, mas podem representar registros LDAP, feeds de RSS ou até mesmo arquivos do sistema.
- Neste exemplo, o controller usa o model para trazer ao usuário as últimas compras do banco de dados;
- Depois que o controller fez sua mágica sobre os dados, ele repassa para a view. A view faz com que os dados fiquem prontos para a representação do usuário;
- Uma vez que a view tenha usado os dados provenientes do controller para construir a página, o conteúdo é retornado ao browser do usuário.

**Benefícios**

Por que usar MVC? Porque é um verdadeiro padrão de projeto (design pattern) e torna fácil a manutenção da sua aplicação, com pacotes modulares de rápido desenvolvimento. Elaborar tarefas divididas entre models, views e controllers faz com que sua aplicação fique leve e independente. Novas funcionalidades são facilmente adicionadas e pode-se dar nova cara nas características antigas num piscar de olhos. O design modular e separado também permite aos desenvolvedores e designers trabalharem simultaneamente, incluindo a habilidade de se construir um rápido protótipo. A separação também permite que os desenvolvedores alterem uma parte da aplicação sem afetar outras.

Se você nunca desenvolveu uma aplicação neste sentido, isso vai lhe agradar muito, mas estamos confiantes que depois de construir sua primeira aplicação em CakePHP, você não vai querer voltar atrás.

**Referências**

https://www.youtube.com/watch?v=VInLNcHm8tA&list=PLtxCFY2ITssBl_nihh4HC5-ZlnIPEpVQD - Curso de PHP + MVC grátis em 21 aulas
https://www.youtube.com/watch?v=vvS7JgEcmic - PHP MVC Fácil
https://www.youtube.com/watch?v=GlMZDMyy-jE&list=PLxNM4ef1Bpxiah1JPIqK1mkwi0h20EoQ1 - PHP7 com MVC
https://www.youtube.com/watch?v=2dqI8o6bvjM&list=PLLfNZbkxufIUsLRzQCCGaek4PxQB4RLGe - Curso de MVC com PHP OO

**Front Controller** - é o único ponto de entrada para o aplicativo. Podemos usar um .htaccess para redirecionar tudo para este ponto. Um exemplo é ter um diretório public e dentro dele um index.php. Este index.php será nosso FrontController.

Nos próximos capítulos mostrarei com detalhes a criação do aplicativo, com seus arquivos, inclusive o banco de dados simples.

# 2 – Namespace

Começaremos agora a criação do aplicativo que terá a seguinte estrutura de arquivos.

## 2.1 - Estrutura de arquivos

```
.htaccess
composer.json
/public
    .htaccess
    index.php
    /assets
            /css
            /image
            /js
/App
    /Controllers
        ClienteController.php
    /Models
        Cliente.php
    /views
        /clientes
            index.php
            search.php
        /error
            index.php
        /templates
            footer.php
            header.php
/Core
    Application.php
    config.php
    ErrorController.php
    Model.php
```

Tentarei seguir uma estrutura lógica, passo a passo, enquanto chamo pelo navegador, que irá indicar o próximo passo, mostrando que está faltando certo arquivo.

Criarei agora o aplicativo na minha pasta web. Como uso Linux Mint minha pasta web fica em:
/var/www/html

Então criarei o aplicativo na pasta
/var/www/html/php_mvc

Agora criarei a estrutura principal dos arquivos, apenas as pastas:

/public
/App
/Core

## 2.2 – Composer

O composer será usado para criar algo bem importante em nosso aplicativo, que é o namespace, usando autoload com PSR-4.

Podemos usar o comando

composer init

Para criar um composer.json inicial mas eu já passarei a estrutura do arquivo que uso e assim facilitará o trabalho.

## 2.3 - composer.json

O primeiro arquivo a ser criado será o

composer.json

Na pasta /var/www/html/php_mvc. Quem usa o Xampp no windows: c:\xampp\htdocs\php_mvc

Contendo:

```
{
   "name": "ribafs/php-mvc",
   "description": "Aplicativo Simples em PHP usando MVC",
   "type": "project",
   "license": "MIT",
   "authors": [
      {
         "name": "Ribamar FS",
         "email": "ribafs@gmail.com"
      }
   ],
   "minimum-stability": "dev",
   "require": {},
   "autoload":
   {
      "psr-4":
      {
         "App\\" : "App/",
         "Core\\" : "Core/"
      }
   }
}
```

Faça as devidas  adaptações para você.

Mais detalhes sobre o namespace no próximo item.

## 2.4 – PSR-4

https://www.php-fig.org/psr/psr-4/

O composer, seguindo as instruções do composer.json, criará uma estrutura de namespace que facilitará a navegação do aplicativo, sem o uso de includes.

A seção do composer.json:

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

Indica que a pasta App será apontada para o namespace App (usei assim para facilitar que eu me lembrasse do namespace).
A pasta Core apontada para o namespace Core e seguindo o padrão da PSR-4.

Detalhes sobre namespace e exemplos simples podem ser vistos em:

https://github.com/ribafs/apostilas

Na apostila PHPOOApostila.pdf e exemplos na respectiva pasta Anexos.

Abra o terminal ou prompt e vamos executar o comando

```
cd php_mvc
composer du
```

Veja que então ele cria a pasta vendor, que contém os arquivos responsáveis pelo autoload.

# 3 - .htaccess

Agora vamos criar os dois arquivos .htaccess do aplicativo.

Estando no raiz (/var/www/html/php_mvc) crie o arquivo

.htaccess

Contendo:

```
RewriteEngine on
RewriteRule ^(.*) public/$1 [L]
```

Veja que ele procura redirecionar todas as requisições que chegam no raiz do aplicativo para a pasta public. Então vamos testar isso

Chame pelo navegador:

http://localhost/php_mvc

Receberá o erro:

Internal Server Error

Vamos agora criar o segundo arquivo

public/.htaccess

Contendo

```
# Necessary to prevent problems when using a controller named "index" and having a root index.php
# more here: http://httpd.apache.org/docs/2.2/content-negotiation.html
Options -MultiViews

# Activates URL rewriting (like myproject.com/controller/action/1/2/3)
RewriteEngine On

# Prevent people from looking directly into folders
Options -Indexes

# If the following conditions are true, then rewrite the URL:
# If the requested filename is not a directory,
RewriteCond %{REQUEST_FILENAME} !-d
# and if the requested filename is not a regular file that exists,
RewriteCond %{REQUEST_FILENAME} !-f
# and if the requested filename is not a symbolic link,
RewriteCond %{REQUEST_FILENAME} !-l
# then rewrite the URL in the following way:
# Take the whole request filename and provide it as the value of a
# "url" query parameter to index.php. Append any query string from
# the original URL as further query parameters (QSA), and stop
# processing this .htaccess file (L).
RewriteRule ^(.+)$ index.php?url=$1 [QSA,L]
```

Este arquivo redireciona tudo que chega na pasta public para o arquivo index.php.

Testando novamente pelo navegador

http://localhost/php_mvc

Agora recebemos o erro

**Forbidden**

Façamos um teste. Criemos um arquivo public/index.php vazio, sem nada mesmo.

Chamar novamente

http://localhost/php_mvc

Agora nenhum erro. Realmente está funcionando:

- .htaccess do raiz envia para public/.htaccess
- public/.htaccess envia para public/index.php

Remover o index.php criado.

# 4 – Criação do Aplicativo Interativamente

Este arquivo será a porta de entrada do aplicativo. Este é o chamado front controller.

- Front Controller
- Constantes/Config
- Rotas/Application
- Controllers
- Models
- Views
– Templates

**Criar nosso Front controller em**

public/index.php

Contendo

```
<?php

define('ROOT', dirname(__DIR__) . DIRECTORY_SEPARATOR);
define('APP', ROOT . 'App' . DIRECTORY_SEPARATOR);
define('CORE', ROOT . 'Core' . DIRECTORY_SEPARATOR);
define('VIEWS', APP . 'views' . DIRECTORY_SEPARATOR);

require_once ROOT . 'vendor/autoload.php';
require_once CORE . 'config.php';

use Core\Application;

$app = new Application();
```

Aqui é incluido o autoload.php e também o config.php.
Ao final chama a classe router/Appication

Vejamos como fica o navegador agora

http://localhost/php_mvc

Agora ele nos pede Core/config.php. Então  vamos criar:

```
<?php

define('URL_PUBLIC_FOLDER', 'public');
define('URL_PROTOCOL', '//');
define('URL_DOMAIN', $_SERVER['HTTP_HOST']);
define('URL_SUB_FOLDER', str_replace(URL_PUBLIC_FOLDER, '', dirname($_SERVER['SCRIPT_NAME'])));
define('URL', URL_PROTOCOL . URL_DOMAIN . URL_SUB_FOLDER);

define('HOST', 'localhost');
define('USER', 'root');
define('PASS', 'root');
define('DB', 'testes');
```

http://localhost/php_mvc

Agora  nos pede Core/Application. Vamos criar e detalhar um pouco:

Esta é a classe responsável pelas rotas. Dependendo do que for pedido através de um link ou indicado na URL, esta classe encaminha para o respectivo recurso. Caso ela não encotnre o recurso ela emitirá uma mensagem de erro através de um controller e view.

```
<?php
namespace Core;

class Application
{
    private $urlController = null;
    private $urlAction = null;
    private $urlParams = array();

    public function __construct()
    {
        $this->splitUrl();

        // Controller default, caso nenhum tenha sido definido
        if (!$this->urlController) {
            $page = new \App\Controllers\ClienteController();
            $page->index();

        } elseif (file_exists(APP . 'Controllers/' . ucfirst($this->urlController) . 'Controller.php')) {
            $controller = "\\App\\Controllers\\" . ucfirst($this->urlController) . 'Controller';
            $this->urlController = new $controller();

            if (method_exists($this->urlController, $this->urlAction) && is_callable(array($this->urlController, $this->urlAction))) {

                if (!empty($this->urlParams)) {
                    call_user_func_array(array($this->urlController, $this->urlAction), $this->urlParams);
                } else {
                    $this->urlController->{$this->urlAction}();
                }

            } else {
                if (strlen($this->urlAction) == 0) {
                    $this->urlController->index();
                } else {
                    $page = new \Core\ErrorController();
                    $page->index();
                }
            }
        } else {
            $page = new \Core\ErrorController();
            $page->index();
        }
    }

    /**
     * Get and split the URL
     */
    private function splitUrl()
    {
        if (isset($_GET['url'])) {

            // split URL
            $url = trim($_GET['url'], '/');
            $url = filter_var($url, FILTER_SANITIZE_URL);
```

```
            $url = explode('/', $url);

            $this->urlController = isset($url[0]) ? $url[0] : null;
            $this->urlController = substr($this->urlController, 0, -1);

            $this->urlAction = isset($url[1]) ? $url[1] : null;

            // Remove controller and action from the split URL
            unset($url[0], $url[1]);

            // Rebase array keys and store the URL params
            $this->urlParams = array_values($url);
        }
    }
}
```

Veja que a classe tem um método que separa os elementos da URL e os oferece para o construtor.

Caso o controller procurado não exista ele envia para o ClienteController. Caso o controller exista ele montará seu nome de acordo com a URL e chamará o devido recurso.

Caso seja solicitado um controller ou método inexistente receberá uma mensagem de erro disparada pelo controller de erro e sua view correspondente.

http://localhost/php_mvc

Agora ele nos pede o App/Controllers/ClienteController. Vamos criá-lo.

App/Controllers/ClienteController.php

```
<?php
namespace App\Controllers;

use App\Models\Cliente;

class ClienteController
{
    public function index(){
        $cliente = new Cliente();
        $clientes = $cliente->index();

        require_once APP . 'views/templates/header.php';
        require_once APP . 'views/clientes/index.php';
        require_once APP . 'views/templates/footer.php';
    }

    public function add(){
        // if we have POST data to create a new cliente entry
        if (isset($_POST["submit_add_cliente"])) {
            $cliente = new Cliente();
            $cliente->add($_POST["nome"], $_POST["idade"]);
        }

        header('location: '.URL.'clientes/index');
    }

    public function delete($id)
    {
        if (isset($id)) {
```

```
            $cliente = new Cliente();
            $cliente->delete($id);
        }
        header('location: ' . URL . 'clientes/index');
    }

    public function edit($id)
    {
        // Verifica se $id tá setado
        if (isset($id)) {
            $cliente = new Cliente();
            $cliente = $cliente->edit($id);

            if ($cliente === false) {
                $page = new \Core\ErrorController();
                $page->index();
            } else {
                require APP . 'views/templates/header.php';
                require APP . 'views/clientes/index.php';
                require APP . 'views/templates/footer.php';
            }
        // Caso $id não esteja setado, redireciona para a index
        }else{
            header('location: ' . URL . 'clientes/index');
        }
    }

    public function update()
    {
        if (isset($_POST["submit_update_cliente"])) {
            $cliente = new Cliente();
            $cliente->update($_POST["nome"], $_POST["idade"], $_POST['id']);
        }

        header('location: ' . URL . 'clientes/index');
    }

    public function search()
    {
        if (isset($_GET["submit_search_cliente"])) {
            $clientes = new Cliente();
            $clientes = $clientes->search($_GET['keyword']);

            require_once APP . 'views/templates/header.php';
            require_once APP . 'views/clientes/search.php';
            require_once APP . 'views/templates/footer.php';
        }

        header('location: ' . URL . 'clientes/index');
    }
}
```

Ele traz os métodos comuns de um CRUD mais o método search().

http://localhost/php_mvc

Agora ele pede

App\Models\Cliente.php

Vamos criar

```
<?php
namespace App\Models;

use Core\Model;

class Cliente extends Model
{
    public function index(){
        $stmte = $this->pdo->query("SELECT * FROM clientes order by id desc");
        $executa = $stmte->execute();
        $clientes = $stmte->fetchall();

        return $clientes;
    }

    public function add($nome, $idade)
    {
        $sql = "INSERT INTO clientes (nome, idade) VALUES (:nome, :idade)";
        $query = $this->pdo->prepare($sql);
        $parameters = array(':nome' => $nome, ':idade' => $idade);
        $query->execute($parameters);
    }

    public function delete($id)
    {
        $sql = "DELETE FROM clientes WHERE id = :id";
        $query = $this->pdo->prepare($sql);
        $parameters = array(':id' => $id);
        $query->execute($parameters);
    }

    public function edit($id)
    {
        $sql = "SELECT id, nome, idade FROM clientes WHERE id = :id LIMIT 1";
        $query = $this->pdo->prepare($sql);
        $parameters = array(':id' => $id);
        $query->execute($parameters);
        return ($query->rowCount() ? $query->fetch() : false);
    }

    public function update($nome, $idade, $id)
    {
        $sql = "UPDATE clientes SET nome = :nome, idade = :idade WHERE id = :id";
        $query = $this->pdo->prepare($sql);
        $parameters = array(':nome' => $nome, ':idade' => $idade, ':id' => $id);

        $query->execute($parameters);
    }

    public function search($keyword)
    {
        $sql = "select * from clientes WHERE nome LIKE :keyword order by id";
        $sth = $this->pdo->prepare($sql);
        $sth->bindValue(":keyword", "%".$keyword."%");
        $sth->execute();
        $rows =$sth->fetchAll();

        return $rows;
    }
}
```

Também os métodos do CRUD.

http://localhost/php_mvc

Agora nos pede Core/Model. Vamos criar

Core/Model.php

Esta é'a classe de conexão com o banco de dados

```
<?php
namespace Core;

class Model
{
    public $pdo;
    private $host;
    private $user;
    private $pass;
    private $db;

    public function __construct()
    {
        $this->host = HOST;
        $this->user = USER;
        $this->pass = PASS;
        $this->db = DB;
        $dsn = 'mysql:host='.$this->host.';dbname='.$this->db;

        try {
            // Para que usemos as variáveis como objeto ($client->id) e mostre mais detalhes nas mensagens de erro
            // A barra antes do PDO (\PDO) é para indicar que estamos usando ele em seu próprio namespace e não é uma classe do
nosso namespace
            $options = array(\PDO::ATTR_DEFAULT_FETCH_MODE => \PDO::FETCH_OBJ, \PDO::ATTR_ERRMODE => \
PDO::ERRMODE_WARNING);
            $this->pdo = new \PDO($dsn, $this->user, $this->pass, $options );

            return $this->pdo;

        } catch (\PDOException $e) {
            print "Error!: " . $e->getMessage() . "<br/>";
            die();
        }
    }
}
```

Agora nos pede:

App/views/templates/header.php

Vamos criar:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="utf-8">
    <title>Aplicativo Simples em PHP com MVC</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <!-- CSS -->
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.5.3/dist/css/bootstrap.min.css" integrity="sha384-
TX8t27EcRE3e/ihU7zmQxVncDAy5uIKz4rEkgIXeMed4M0jlfIDPvg6uqKI2xXr2" crossorigin="anonymous">
</head>
<body>
    <div class="container">
        <br>
        <h1 class="text-center">Aplicativo Simples em PHP com MVC</h1>
        <h4 class="text-center">(Com Autoload, PSR-4 e Rotas)</h4>
        <br>
```

```
    <div class="text-center">
        <a href="<?php echo URL; ?>clientes" class="btn btn-warning">Clientes</a>   
    </div>
    <br>
```

Veja que estamos usando um CDN do bootstrap.

Ao chamar agora ele já abre o template de clientes mas pedindo App/views/clientes/index.php

Vamos criar
App/views/clientes/index.php

```
<!-- Busca form -->
<br>
<div class="row">
    <div class="col-md-4"></div>
    <div class="col-md-8">
        <form action="<?=URL?>clientes/search" method="get" >
        <div class="pull-right topo" style="padding-left: 0;">
            <span class="pull-right">
            <label class="control-label" for="palavra" style="padding-right: 5px;">
                <input type="text" value="" placeholder="nome ou parte" class="form-control" name="keyword">
            </label>
            </span>
            <input type="submit" name="submit_search_cliente" value="Buscar" class="btn btn-primary"/>
        </div>
        </form>
    </div>
</div>
<br>

<?php
// Testar se vem de ClienteController/add
if(isset($clientes)){
?>

<!-- Add form -->
<div class="row">
    <div class="col-md-3"></div>
    <div class="col-md-8">
    <form action="<?=URL?>clientes/add?>" method="POST">
            <input type="text" name="nome" value="" required placeholder="Nome"/>
            <input type="text" name="idade" value="" required placeholder="Idade"/>
            <input type="submit" name="submit_add_cliente" value="Adicionar" class="btn btn-primary"/>
        </form>
    </div>
</div>

<?php
// Testar se vem de ClienteController/edit
}elseif(isset($cliente)){
    $action = 'edit';
    $id = isset($cliente->id) ? $cliente->id : null;
    $nome = isset($cliente->nome) ? $cliente->nome : null;
    $idade = isset($cliente->idade) ? $cliente->idade : null;
?>

<!-- Edit form -->
<br><br>
<div class="row">
    <div class="col-md-3"></div>
    <div class="col-md-7">
        <form action="<?=URL?>clientes/update?>" method="POST">
            <input type="text" name="nome" value="<?=$nome?>" required placeholder="Nome"/>
            <input type="text" name="idade" value="<?=$idade?>" required placeholder="Idade"/>
            <input type="hidden" name="id" value="<?=$id?>" />
            <input type="submit" name="submit_update_cliente" value="Atualizar" class="btn btn-primary"/>
        </form>
```

```
    </div>
  </div>

  <?php
  }

  if(isset($clientes)){?>

  <!-- Listagem index -->
  <div class="text-center">
    <div class="row">
      <!-- Form de busca-->
          </div>

    <br><br>
    <h3>Lista de clientes</h3>
    <table class="table table-striped table-sm table-bordered table-hover">
      <thead class="bg-dark text-white">
      <tr>
        <td><b>Id</td>
        <td><b>Nome</td>
        <td><b>Idade</td>
        <td colspan="2"><b>Ação</td>
      </tr>
      </thead>
      <tbody>
      <?php
        foreach ($clientes as $cliente) { ?>
        <tr class="table-success">
          <td><?php if (isset($cliente->id)) echo $cliente->id; ?></td>
          <td><?php if (isset($cliente->nome)) echo $cliente->nome; ?></td>
          <td><?php if (isset($cliente->idade)) echo $cliente->idade; ?></td>
          <td><a href="<?=URL . 'clientes/edit/' . $cliente->id?>">Editar</a></td>
          <td><a href="<?=URL . 'clientes/delete/' . $cliente->id?>">Excluir</a></td>
        </tr>
      <?php
        }
      } ?>
      </tbody>
    </table>
  </div>
</div>
```

É uma estrutura que permite ter a busca integrada e algo como master/details.

Assim já abre a view index de clientes mas reclama de

/App/views/templates/footer.php

Vamos criar:

```
  <br><br>
  <hr>
  <footer class="text-center">
    Encontre em <a href="https://github.com/ribafs/simplest-mvc">Aplicativo Simples PHP com MVC</a>.
  </footer>
</body>
</html>
```

Agora abre sem qualquer erro. Mas nosso aplicativo não está completo.
Falta a view de erros e o controller respectivo. Então criemos App/views/error/index.php e o conroller Core/ErrorController.php

Caso tente chamar pela URL um controller ou método que não exista receberá um erro do aplicativo e não no PHP.

# 5 - PHP Moderno

**Front Controller**
O Front Controller é o ponto de entrada do aplicativo, é o arquivo PHP que lida com todas as solicitações de um aplicativo. É o primeiro arquivo PHP que uma solicitação acessa no seu aplicativo e (essencialmente) o último arquivo PHP que uma resposta percorre ao sair do aplicativo.

public/index.php

Para auxiliar o index.php acima tenho no raiz um .htaccess redirecionando tudo para o /public e dentro do /public tenho outro .htaccess redirecionando tudo que chega ao /public para o /public/index.php

Algo assim

```
<?php
declare(strict_types = 1);

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

**strict_types**
Para o topo de todos os arquivos .php adicione no início

```
<?php
declare(strict_types=1);
```

**Namespace**
Namespace com composer e PSR-4
O uso deste padrão do FIG possibilita que tenhamos facilidade ao trabalhar com classes e não precisemos ficar digitando includes ou requires.
Pra isso usamos o

composer.json

**Routing**

A URI /products/purple-dress/medium
Deve ser manipulada pelo controller ProductDetails com o action purple-dress e o argumento medium passado

# 6 - Boas práticas

Boas Praticas

Obrigado ao grupo do PHP-FIG (https://www.php-fig.org/) pelos autoloads.

código bem organizado, em diretórios claros, com nomes seguindo os bons padrões

Cada classe lida somente com um único assunto. Similar ao que acontece com tabelas de bancos de dados normalizadas. As classes também devem ser criadas apenas uma por arquivo

Objetos, que são instâncias de classes, existem somente durante certo período de tempo, iniciando na sua criação (instanciação) e terminando com sua destruição, que acontece explicitamente com o __destruct() ou quando o script é encerrado.

Para usar propriedades ou métodos em strings com aspas duplas usar delimitados com chaves:
echo "Este é o nome {$this->nome}";

**Classes - CamelCase**
Arquivos de classes tem nomes como as classes, CamelCase. Ex: ClientesController.php e classe ClientesController()

Arquivos e pastas comuns, que não contém classes: Tudo em minúsculas, separados por sublinhado para palavras compostas. Exs: app, core, public, config.php, bootstrap.php, meu_arquivo.php.
Propriedades - sempre declaradas no início da classe e camelCase
Métodos e propriedades – camelCase.
Nomes de constantes – tudo em maiúsculas. Ex: MINHA_CONST
Nomes de arquivos de classes devem ter o mesmo nome da classe mais .php

Nomes de propriedades, métodos e classes bem claros e simples de entender e ler.

Usar metodologia DRY
Evitar arquivo muito grande. Se for o caso criar dois ou mais arquivos e usar o require_once.
Comentar em cada trecho em que necessite. Evitar comentar em exagero, apenas comentar detalhes que podem gerar dúvida.
Configurar o editor para que quebre a linha ao invés de precisar rolar a tela para ler toda a linha.

Usar um bom sistema de rotas, que redirecione adequadamente uma solicitação de URL para o respectivo recurso (controller/action).

Usar em todos os arquivos para para forçar a declaração de tipos:
declare(strict_types=1);
Nomes simples de métodos/actions dos controllers: index, add, view, edit, delete e nos models também.

Codificar no desktop usando exibição de erros E_ALL e display_error = On

Quando não tiver acesso aos arquivos do servidor, usar nos arquivos PHP:

ini_set()
ini_set('error_reporting', E_ALL);
ini_set('display_errors', 1);

Nunca camufle erros com @ e evite usar include e include_once,
Prefira require_once

Preferir delimitar strings com apóstrofos/aspas simples, que são mais seguras e mais rápidas, visto que não recebem atenção do processador.

$nome = 'Ribamar';

Para métodos de formulários evitar o uso de GET. Prefira o POST

Utilize sempre as tags completas:
Em arquivos com somente código PHP, não usar a tag de fechamento

```
<?php
...
?>
```

E também a sintaxe reduzida nas views:
```
<?=$teste?>
```

**Indentação**
Usar 4 espaços para a indentação e não tab

Não começar o código já na linha da tag de abertura, mas uma linha abaixo:

```
<?phpif($x==0){

<?php

if($x==0){
```

Usar sempre na primeira linha de código, para forçar a declaração explícita de tipos
declare(strict_types = 1);

Usar parêntesis ao final do nome da classe, ao instanciar
```
$obj = new NomeClasse();
```

**Uso de chaves:**

```
class Nome
{

}
```

**Métodos**

```
public function nome()
{

}
```

**Laços**

```
foreach()
{

}
```

**if**

```
if($x == 0) {

}
```

- Uso de .htaccess
- Front controller. Somente esta pasta é acessível ao público
- Constantes facilitam a comunicação entre arquivos do aplicativo
- Nomes de classes: CamelCase e Nomes de métodos e de propriedades: camelCase
- Gosto destes nomes: ClientesController nos nomes de classes e dos arquivos
- Uso do composer com PSR-4 (facilita a criação do aplicativo) e informações sobre o aplicativo e sobre o autor
- Acompanhar um detalhado README.md
- Não publique seu projeto antes que esteja pronto. Isso pode queimar seu filme. Além de estar pronto capriche no mesmo, faça o seu melhor.
- Acompanhar um script sql no raiz para teste do aplicativo com facilidade
- Usar código que avisa que composer ainda não foi executado (abaixo) evitando a mensagem de erro
- Fornecener login e senha logo no início do README.md, caso precise de login e senha para testar aplicativo
- Que o usuário seja capaz de entender e colocar pra funcionar sem mesmo ler o README.md
- Preferir o uso do PDO, pois é o mais usado e com mais recursos  (genérica)
- Criar classes primárias em core/ e extender nas classes em app/
- Ao programar ter sempre em mente de facilitar a vida do usuário (genérica)
- Use CSS, o BootStrap ou outro framework CSS para deixar seu aplicativo mais elegante
- Procurar usar as melhores práticas e as recomendações de segurança (genérica)
- Nos diretórios app/ (no caso da estrutura que sugeri) inserir arquivos README.md com instruções de como o usuário proceder. Exemplo: Criar um controller, um model e uma pasta view para cada tabela extra adicionada. E sempre que achar por bem crie um README.md com orientações
- Tamves seja mais prático de lembrar que o namespace seja similar ao diretório. Exemplo:

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

Evitar comunicação direta entre view e model. Sempre intermediar essa comunicação com o controller
Usar e abusar de expressões condicionais. Exemplo abaixo (genérica)
Crie o aplicativo que se destina ao público de forma que funcione em uma pasta do docRoot, pois se exigir um virtualhost dificulta o teste será mais trabalhoso.

Minha estrutura preferida atualmente

Dividir arquivos entre app (aplicativo, negócio) e core (arquivos fixos)

Raiz limpo, com apenas .htaccess, README.md e composer.json
public
  assets
App
  Controllers (constrollers do aplicativo)
  Models (Models do aplicativo)
  views (Views do aplicativo)
Core (arquivos do core, não alterados pelo programador)
  config.php (configurações do banco e constantes)
  bootstrap.php (início do aplicativo e algumas constantes)
  Controller.php
  ErrorController.php
  Model.php
  Application.php (router)
Meu exemplo de estrutura está nos aplicativos simplest-mvc.

# 7 - Fluxo das informações

- Veja que não mais existe um index.php. Quando alguém chega no raiz de mvc5 é redirecionado para a apsta public pelo .htaccess

- Quando chega na pasta public será redirecionado para o public/index.php pelo public/.htaccess

**Quando o public/index.php é chamado**

- Ele define algumas constantes para o aplicativo
- Inclui o autoload
- Inclui o Core/config.php, que também define outras constanes importantes
- Então inclui a classe Core/Router através do use
- Instancia a classe Router. Veja que esta classe tem dois métodos, o construct e o Url(), mas ela chama o Url() dentro do __construct(), portanto logo que a classe Router é instanciada todo o seu código é executado e está disponível, no caso o tratamento das rotas acessadas pelo usuário.
- A captura da URL digitada é feita pelo método Url(), que é executado inicialmente pelo __construct() e então testado, para que de acordo com a URL acessar o respectivo controller.
- Veja que se alguém chama o "clients" pela URL o Router dispara o ClientController, de forma semelhante, se for chamada "products" o ProductController será disparado. Caso a URL digitada ou chamada via link não seja para clients nem para products será disparado o ErrorController.

**Quando o ClientController é chamado:**

- Na versão anterior ele incluiu o ClientModel.
  - Agora adicionamos o namespace na primeira linha
  - E cada include é substituido por um "use". Veja como ficou agora no ClientController
    namespace App\Controllers;
    use App\Models\ClientModel;
- O restando do código permanece
- Instancia o model
- Chama seu método index(), que retorna o array $clients e armazena na variável $list;
- Então o Controller inclui a view App/views/clients/index.php com o respectivo template, que mostrará o resultado recebido do model

**Quando o ClientModel é chamado**

- Adiciona namespace e troca include por use
- Em seu método index() efetua uma consulta para trazer todos os registros da tabela clients
- E retorna a consulta na variável $clients, que será recebida pelo controller

**Quando a view views/clients/index.php é chamada**

- Mostra algum texto explicativo na tela
- Cria uma tabela HTML
- Joga na tabela o resultado de um for trazendo o array $list do Controller e mostra na tela os registros da tabela clients

De forma semelhante procedi com o ProductController e o ProductModel

No caso a View recebe do Controller e não diretamente do Model, o que é uma boa prática.

# 8 – Alguns Padrões de Projeto

Geralmente é uma boa ideia seguir à padrões comuns, pois isso irá fazer com que seu código seja mais fácil de manter e de ser entendido por outros desenvolvedores. São soluções para problemas comuns que encontramos no desenvolvimento ou manutenção de um software orientado a objetos.

Engenheiros de softwares por décadas desenvolveram padrões de projeto para resolver problemas comuns.

A teoria dos padrões de projeto da Microsoft é: "O documento apresenta padrões e os apresenta em um repositório ou catálogo organizado para ajudá-lo a localizar a combinação certa de padrões que resolve seu problema".

Isaac Newton certa vez dissera:
Se cheguei até aqui foi porque me apoiei no ombro de gigantes.

**Front Controller**
O padrão front controller é quando você tem um único ponto de entrada para sua aplicação web, no nosso caso: public/index.php, que trata de todas as requisições. Esse código é mínimo, contendo apenas um require para o src/bootstrap.php, que é responsável por carregar todas as dependências, processar a requisição e enviar a resposta para o navegador. O padrão Front Controller pode ser benéfico pois ele encoraja o desenvolvimento de um código modular e provê um ponto central no código para inserir funcionalidades que deverão ser executadas em todas as requisições (como para higienização de entradas). Assim como também protege nosso código fonte, que fica na pasta src, no caso fica fora do alcance do servidor web.

### Model-View-Controller
O padrão de arquitetura model-view-controller (MVC) e os demais padrões relacionados como HMVC and MVVM permitem que você separe o código em diferentes objetos lógicos que servem para tarefas bastante específicas. Models servem como uma camada de acesso aos dados e esses dados são requisitados e retornados em formatos nos quais possam ser usados no decorrer de sua aplicação. Controllers (Controladores) tratam as requisições, processam os dados retornados dos Models e carregam as Views para enviar a resposta. E as Views são templates de saída (marcação, xml, etc) que são enviadas como resposta ao navegador.

O MVC é o padrão arquitetônico mais comumente utilizado nos populares Frameworks PHP.

http://br.phptherightway.com/pages/Design-Patterns.html

**Quando usar o padrão de arquitetura MVC**

O padrão de arquitetura MVC ajuda-nos a implementar a separação de interesses entre as classes de modelo, visualização e controlador nas aplicações.

A separação de interesses torna fácil testar nossa aplicação, já que a relação entre os diferentes componentes da aplicação é mais clara e coerente. O MVC nos ajuda a implementar uma abordagem de desenvolvimento orientada a testes, na qual implementamos casos de teste automatizados antes de escrevermos o código. Esses casos de teste de unidade nos ajudam a pré-definir e verificar os requisitos de um novo código antes de escrevê-lo.

Se estamos fazendo um aplicativo com bastante estímulo sério no lado do cliente para se recusar a ir junto com o JavaScript sozinho. Se estivermos desenvolvendo um aplicativo que tem um alto desempenho no lado do servidor e um pouco de comunicação no lado do cliente, não devemos usar a arquitetura de padrão MVC;

em vez disso, devemos usar uma configuração simples como o modelo de formulário baseado na web. A seguir estão algumas características que nos ajudarão a usar a arquitetura MVC em nosso aplicativo ou não:

- i. Nosso aplicativo precisa de comunicação assíncrona no back-end.
- ii. Nosso aplicativo tem uma funcionalidade que resulta em não recarregar uma página inteira, por exemplo, comentando em um post usando o Facebook ou rolando infinitamente etc.
- iii. Manipulação de dados é principalmente no lado do cliente (navegador), em vez de lado do servidor.
- iv. O mesmo tipo de dados está sendo entregue de maneiras diferentes em uma única página (navegação).
- v. Quando nosso aplicativo possui muitas conexões insignificantes que são usadas para modificar dados (botão, switches).

**Vantagens da arquitetura MVC**

- a. A arquitetura MVC nos ajuda a controlar a complexidade do aplicativo dividindo-o em três componentes, ou seja, model, view e controller.
- b. O MVC não usa formulários baseados em servidor, por isso é ideal para desenvolvedores que desejam ter controle total sobre o comportamento de seus aplicativos.
- c. A abordagem de desenvolvimento orientada a teste é suportada pela arquitetura MVC.
- d. O MVC usa o padrão do controlador frontal. O padrão do controlador frontal manipula as diversas solicitações recebidas usando uma única interface (controlador). O controlador frontal fornece controle centralizado. Precisamos configurar apenas um controlador no servidor da web em vez de muitos.
- e. O front controller fornece suporte a comunicações de roteamento avançadas para projetar nosso aplicativo da web.

**Recursos de um framework MVC**
À medida que dividimos a lógica de nossa aplicação em três tarefas (lógica de entrada, lógica de negócios, lógica de interface), o teste desses componentes se tornaria mais fácil. A testabilidade é muito rápida e flexível, já que podemos usar qualquer estrutura de teste de unidade compatível com o framework MVC. É uma estrutura extensível e conectável. Podemos projetar os componentes de nossa aplicação de maneira que sejam facilmente substituíveis ou possam ser facilmente modificados. Podemos conectar nosso próprio mecanismo de visualização, estratégia de roteamento de URL, serialização de restrição de método de ação. Em vez de depender da classe para criar objetos, usamos uma técnica de injeção de dependência (DI) que nos permite injetar objetos nas classes. Outra técnica de inversão de controle (IOC) é usada para mostrar dependência entre objetos, especifica qual objeto precisa de outro objeto. O MVC fornece um componente de mapeamento de URL que nos ajuda a construir usando URLs compreensíveis e pesquisáveis. Em vez de usar extensões de nome de arquivo, os padrões de nomenclatura de URL de suporte MVC são muito úteis para o endereçamento de otimização de mecanismo de pesquisa (SEO) e de transferência de estado representacional (REST). Algumas estruturas do MVC, como a ASP.NET MVC, fornecem alguns recursos integrados, como autenticação de formulário, gerenciamento de sessão, lógica de negócios transacional, segurança de aplicativos web, mapeamento relacional de objeto, localização, associação e funções e autorização de URL, etc. as estruturas disponíveis hoje são backbone.js, ember.js; angular.js e knockout.js.

- I. Backbone. O framework Js-Backbone.js é útil quando nossa aplicação precisa de flexibilidade, temos requisitos incertos. Além disso, queremos acomodar a alteração durante o desenvolvimento do aplicativo.
- II. Ember.js- Quando queremos que nosso aplicativo interaja com a API JSON, devemos usar o framework ember.js em nosso aplicativo.
- III Angular.js- Se queremos mais confiabilidade e estabilidade em nosso aplicativo, queremos testes extensivos para nossa aplicação, então devemos usar o framework angular.js.
- IV. Knockout.js - se quisermos criar uma interface dinâmica complexa de aplicativo, o framework knockout.js será muito útil para nós.

Cada estrutura tem suas próprias vantagens e desvantagens. Os desenvolvedores podem usar qualquer um dos frameworks de acordo com seus requisitos, que se adequam ao seu aplicativo da web.

**A view não deve ir direto ao model**. O controller é mediador/middleware entre V e M.

MVC ajuda DRY.

Ajuda a organizar o código e a torná-lo manutenível.

**Model**
A camada model é o backbone do aplicativo e lida com a lógica de dados. Na maioria das vezes, considera-se que o model é responsável pelas operações CRUD em um banco de dados, que pode ou não ser verdade. O modelo é responsável pela lógica de dados, o que significa que as operações de validação de dados também podem ser executadas aqui. Em palavras simples, os modelos fornecem uma abstração para os dados. As camadas restantes da aplicação não sabem ou não se importam como e de onde os dados vêm ou como uma operação é realizada em dados. É responsabilidade do modelo cuidar de toda a lógica de dados.

O método seguido é modelos gordos e controllers finos, o que significa manter toda a lógica da aplicação nos modelos e os controladores finos, o quanto possível.

**Controllers**
Os controladores respondem às ações executadas por um usuário nas views e respondem a view. Por exemplo, um usuário preenche um formulário e o submete. Aqui, o controlador vem no meio e começa a agir sobre a apresentação do formulário. Agora o controlador irá primeiro verificar se o usuário tem permissão para fazer este pedido ou não.

Em seguida, o controlador executará a ação apropriada, como se comunicar com o modelo ou qualquer outra operação. Em uma analogia simples, o controlador é o meio homem entre vews e models. Como mencionamos antes na seção de modelos, controladores devem ser pequenos. Então, principalmente, controladores são usados apenas para lidar com solicitações e se comunicar com modelos e viws. Todos os tipos de operações de dados são executados em modelos.

O único trabalho do padrão de design MVC é separar as responsabilidades de diferentes partes em um aplicativo. Portanto, os modelos são usados ppara gerenciar os dados do aplicativo. Controladores são usados para realizar ações nas entradas do usuário, e as views são responsáveis pelo visual, pela representação de dados. Como mencionamos anteriormente, o MVC separa as responsabilidades de cada parte, por isso não importa se ele acessa o modelo de controladores ou views; a única coisa que importa é que as views e os controladores não devem ser usados para
executar operações em dados que são de responsabilidade do modelo, e controladores não devem ser usados para visualizar qualquer tipo de dados pelo usuário final, pois este é a responsabilidade da view.

# 9 – Rotas

Roteamento é o que acontece quando uma aplicação determina qual controller e action será executado baseado em uma URL requisitada.

Uma rota é um caminho para acessar um recurso através da composição de uma URL válida.

**Exemplo:**
O framework recebe a URL http://localhost/users/list.html e executa o controller Users e o action list().

Nós precisamos ser hábeis para processar qualquer rota que não case com as que nós definimos para existentes controllers e actions ou mostrar uma mensagem de erro apropriada.

Como uma requisição/request é atendida por uma resposta/response:

- A requisição é recebida pela aplicação
- A aplicação quebra a requisição em seus componentes: métodos (GET, POST, etc), host, path, etc.
- A aplicação procura por uma rota definida que case com a requisição
- Logo que encontre ela determina o controller e o action para atendê-la/response

Rotas não semânticas estão desatualizadas. Não existe razão para um usuário ver uma longa cadeia de query strings na URL. URLs assim não são fáceis de memorizar e expoem a configuração do servidor.

Uma classe para roteamento deve ser capaz de distinguir o tipo de HTTP request. Um request tipo GET requisita geralmente o retorno de um ou mais recursos.

O tipo POST cria um novo recurso.
PUP ou PATCH atualiza um existente.
DELETE remove um recurso existente.

**Protocolo HTTP**

Ele é responsável por prover uma interface uma interface para a web. Ele permite que ocorra a troca de dados entre um dispositivo cliente e um servidor.

Quando um cliente faz um pedido de uma página (ou qualquer outro recurso) para um servidor que “fala” HTTP, ele está fazendo um Request. O servidor por sua vez tem a habilidade de compreender esse pedido e responder a ele com um Response. Esse ciclo se repete o tempo todo e quando você programa em PHP passa boa parte do tempo fazendo isso acontecer.

A PSR-7 (https://www.php-fig.org/psr/psr-7/) é a especificação de um conjunto de métodos que podem vir a ser usados para gerenciar Requests, Responses, Messages, e etc.

As Messages serão montadas no cliente para compor um Request informando alguns dados, em especial o método (Method) e que as Responses serão criadas no servidor respondendo no mesmo método.

Os métodos HTTP mais comuns são GET e POST.

Quanto mais simples para o desenvolvedor mais popular a ferramenta se torna, mas note que sempre tem um custo simplificar algo.

**Exemplo bem simples:**
https://github.com/azeemhassni/simplest-php-router

Crie uma pasta em seu diretório web com os arquivos:

index.php
router.php

index.php

```
<?php
require_once "router.php";

route('/', function () {
    return "Hello World";
});

route('/about', function () {
    return "Hello form the about route";
});

$action = $_SERVER['REQUEST_URI'];
dispatch($action);
```

router.php

```
<?php
/**
 * Holds the registered routes
 *
 * @var array $routes
 */
$routes = [];

/**
 * Register a new route
 *
 * @param $action string
 * @param \Closure $callback Called when current URL matches provided action
 */
function route($action, Closure $callback)
{
    global $routes;
    $action = trim($action, '/');
    $routes[$action] = $callback;
}

/**
 * Dispatch the router
 *
 * @param $action string
 */
function dispatch($action)
{
    global $routes;
    $action = trim($action, '/');
    $callback = $routes[$action];

    echo call_user_func($callback);
}
```

Executar com:

php -S localhost:8080

http://localhost:8080
http://localhost:8080/about

**Mais detalhes em:**

https://phpzm.rocks/php-like-a-boss-3-construa-seu-router-e024ea32ee8a
https://github.com/phpzm/like-a-boss-3
https://www.taniarascia.com/the-simplest-php-router/

**Referências**

O grande pontapé deste aplicativo foi o

https://github.com/panique/mini3

Com ele consegui criar meus aplicativos em PHP com MVC e já criei vários que podem ser vistos no Github.com/ribafs, o último foi o simplest-mvc

**Download desta apostila e outras**

- PHP
- MySQL
- PDO
- PHPOO
- Laravel

https://github.com/ribafs/apostilas

## 9.1 - Como criar o teu aplicativo em PHP com MVC?

Uma forma simples é usar o aplicativo fornecido com esta apostila e adaptar para o seu caso. Minha sugestão é que crie uma tabela pequena, com dois campos, para facilitar o teu trabalho. Então faça uma cópia do aplicativo atual para o que deseja e saia ajustando cada arquivo.