15 Dicas, Boas Práticas e Fundamentos do PHP

Conheça 15 Dicas, Boas Práticas e Fundamentos **Indispensáveis** Para Desenvolver Scripts PHP **Profissionais**

ULTIMATE PHP

Sumário

Seja Muito Bem-Vindo(a)!

Antes de qualquer coisa, quero te parabenizar e te agradecer!

Parabéns por tomar essa iniciativa de aprender as Boas Práticas e os Fundamentos sobre o PHP!

Conhecer os Fundamentos é o que realmente destaca e diferencia um Programador Profissional de um mediano.

Também quero agradecer você por me dar esta oportunidade de conversar com você e me apresentar melhor, além de mostrar como aprender PHP não precisa ser um bicho de sete cabeças! Muito pelo contrário: é muito mais simples do que muitos "professores" por aí dizem!

Para Quem É Este Guia?

É uma ótima pergunta!

E vamos à resposta.

Este guia é para todos aqueles que querem programar em PHP de forma **Profissional**.

Não basta aprender a programar em PHP e achar que é suficiente.

Existem Boas Práticas que devem ser seguidas, se você quiser, realmente, se destacar da maioria.

E, se você está aqui, tenho total certeza de que você faz parte dessa minoria dedicada a se aperfeiçoar na Arte da Programação!

Parabéns novamente pela iniciativa! :)

Tem muita gente por aí que se diz *Programador PHP*, mas escreve códigos horríveis!

Você será diferente! Tenho certeza disso!

Vou mostrar aqui as **Boas Práticas de Programação PHP**.

Também vou apresentar os Fundamentos principais para que você se torne um excelente profissional de PHP.

Muitas delas podem (e devem) ser aplicadas para qualquer outra linguagem. Mas, como o objetivo aqui é o PHP, o foco será sempre o PHP.

Guarde bem esta palavra:

FUNDAMENTOS

São os Fundamentos do PHP que vão destacar você como Profissional.

Vou falar disso mais pra frente.

Então Mãos à Obra!

Chega de papo e vamos ao que interessa!

Boas Práticas de Programação PHP

1 Utilize Sempre as Tags Completas

O PHP suporta quatro tipos de tags. São elas:

Listagem 1: Tags suportadas pelo PHP

```
1 <?php ?>
2 <? ?>
3 <script language="php"></script>
4 <% %>
```

A primeira forma é a padrão e preferencial, porém muitos utilizam a segunda, por ser mais curta e prática. Com ela, também é possível usar esta sintaxe para exibir uma informação:

Listagem 2: Exibindo dados usando tags curtas

```
1 <?= "Hello World!" ?>
2 // equivale a
3 <? echo "Hello World!"; ?>
```

Porém, a utilização dessa tag também não é recomendada. A disponibilidade das tags curtas pode estar desabilitada no servidor, conforme o valor da diretiva **short_open_tag**. Se ela estiver em *off*, essas tags serão ignoradas, fazendo com que o script não seja interpretado.

A terceira opção não é muito comum, mas não é afetada pela diretiva short_open_tag, sendo tão segura quanto a primeira. As tags ASP <% %>, que também devem ser evitadas, só serão interpretadas se a diretiva asp_tags estiver em *on*, o que é pouco comum, visto que o seu valor padrão é off.

Mais detalhes podem ser vistos no link a seguir.

Tags do PHP

2 Indentação de Código

Indentação (do inglês *Indent*) significa manter blocos de códigos alinhados. Ou seja, sempre que iniciar um novo bloco, aumente o nível de distância da margem esquerda.

A correta indentação facilita a organização e a leitura do código. Veja os seguintes exemplos:

Listagem 3: Código em uma só linha e sem indentação

```
1 if($a < 4) { echo "Hello World!"; }
```

Listagem 4: Código em um múltiplas linhas mas sem indentação

```
1 if($a < 4) {
2 echo "Hello World!";
3 }
```

Listagem 5: Código com indentação adequada

```
1 if ( $a < 4 )
2 {
3     echo "Hello World!";
4 }
```

A ausência de indentação em um pequeno trecho de código, como o mostrado acima, pode não fazer muita diferença. Porém tente imaginar 300 ou 400 linhas sem indentação. Ou pior: imagine fazer uma alteração em um script escrito nesse formato!

ISSO É **UM TERROR**!

Alguns programadores iniciantes acreditam que, reduzindo o número de linhas de código, facilita-se a navegação e a sua leitura. Ou que um código mais curto é executado mais rapidamente pela CPU. Nem um nem outro é verdade.

O recomendado, e também adotado por muitos, é uma indentação de quatro espaços. Dê atenção especial às estruturas de controle, onde o "aninhamento" delas podem causar confusão de leitura.

Listagem 6: Estruturas aninhadas e sem indentação

```
1 if($a) {
2 if($b == $a) {
3 //código
4 } else {
5 //código
6 }
7 }
```

Agora a versão corretamente indentada.

Listagem 7: Estruturas aninhadas e com indentação adequada

```
1 if($a) {
2     if($b == $a) {
3         //código
4     } else {
5         //código
6     }
7 }
```

Indentar com Espaços ou TABs?

Espaços!

Sim, use sempre espaços em vez de `TAB`s.

Ao utilizar TABs, alguns editores de texto exibem o código mal formatado, pois alguns consideram o TAB equivalente ao espaçamento de 2, 4 ou 8 espaços. Além disso, se houver mistura de espaços e TABs, o código pode parecer bem feio em determinados editores de texto. Quando copiamos código de outro lugar, principalmente de *sites*, eles aparecem com espaços. E, se estivermos usando TABs, pode haver conflitos.

Se um dia você programar na linguagem Python, verá o problema que isso dá! Python não tem delimitador de blocos (chaves). Blocos são delimitados pela indentação. E espaços e TABs são considerados diferentes. Se houver TAB e espaço em seu script, ele não será executado. OK, mas este guia não é sobre Python, então não falarei mais que isso. :P

Você pode configurar seu editor de textos para transformar TABs em espaços. Assim, você pode teclar TAB para indentar, mas será inserida uma sequência de espaços em vez de um TAB literal.

Chaves na Mesma Linha ou na Linha Seguinte?

Isso fica a seu critério.

Vejamos o exemplo anterior.

Listagem 8: Chaves na mesma linha

```
1 if($a) {
2     if($b == $a) {
3         //código
4     } else {
5         //código
6     }
7 }
```

Podemos colocar as chaves sempre na linha seguinte:

Listagem 9: Chaves na linha seguinte

```php
if ( $a )
{
    if ( $b == $a )
    {
        //código
    }
    else
    {
        //código
    }
}
```

Note que, além de colocar as chaves na linha seguinte, eu adicionei espaços nos `if`'s. Ambos aspectos são questão de preferência. Eu prefiro usar espaços nos `if`'s, `while`'s etc, além de sempre colocar as chaves na linha seguinte. Mas você pode preferir da primeira forma.

O PSR-1 e o PSR-2 são padrões sugeridos para programação PHP. Veja mais sobre eles nos links a seguir.

PSR-1: Padrão Básico de Codificação PSR-2: Guia de Estilo de Codificação

3 Funções para Verificar Existência de Variáveis e Constantes

É importante verificar se uma variável, constante ou mesmo chave de array existe e possui valor, antes de resgatar o seu valor e utilizá-lo. Isso evita resultados inesperados e erros em tempo de execução. Veja o exemplo abaixo:

Listagem 10: Usando variáveis não inicializadas

```
1 $a = $_POST['a'];
2 $b = " World!";
3
4 echo $a.$b;
```

Se esse script for processado após o envio de um formulário pelo método POST, com o campo "a" definido, a variável $a receberá seu valor e o script funcionará. Se enviarmos o valor "Hello" pelo formulário, o script exibirá "Hello World".

Porém, se o usuário não digitar valor no campo "a", o script vai gerar um aviso do nível `E_NOTICE`, alertando que não existe o índice "a" no array `$_POST`, gerando a saída " World".

Por isso é importante verificarmos a existência do índice "a" no array `$_POST`. E é muito simples. Basta usarmos a função `isset`, desta forma:

Listagem 11: Verificando existência de variáveis

```
1 if (isset($_POST['a'])) {
2     $a = $_POST['a'];
3 } else {
4     $a = "Hi";
5 }
6 $b = " World!";
7
8 echo $a . $b;
```

Uma alternativa mais curta é usar o Operador Condicional Ternário, desta forma:

Listagem 12: Verificando existência de variáveis com operador ternário

```
1 $a = isset( $_POST['a'] ) ? $_POST['a'] : "Hi";
2 $b = " World!";
3
4 echo $a.$b;
```

Para o caso de constantes, usamos a função `defined`, como a seguir:

Listagem 13: Verificando existência de constantes

```
1 define( "CAMINHO", "www/meudiretorio/" );
2
3 if ( defined( "CAMINHO" ) )
```

```
4  {
5      echo "A constante já foi definida.";
6  }
7  else
8  {
9      echo "A constante não foi definida.";
10 }
```

Veja mais detalhes no link a seguir:

Variáveis no PHP

4 Tipos de Dados e Casting

PHP não é uma linguagem tipada. Ou seja, não precisamos definir os tipos das variáveis quando a definimos. Isso pode ser bom por um lado, mas muito ruim em algumas situações. Algumas vezes, uma variável que deveria conter um número inteiro recebe uma string, ou um booleano. Isso não gera erro no PHP, mas pode trazer problemas para a integridade dos dados da aplicação. Vejamos um exemplo.

Listagem 14: Tipos de dados

```
1 //Retorna 'integer'
2 $var = 2;
3 print gettype($var);
4
5 //Retorna 'string'
6 $var = "6";
7 print gettype($var);
```

Usamos a função gettype() para retornar o tipo de variável declarada. No segundo trecho temos a declaração da mesma variável `$var`, que passa a receber o tipo de variável *string*. Este dado é convertido automaticamente ao fazer uma soma, por exemplo, voltando novamente a ser um valor do tipo inteiro. Para contornar esse tipo de situação e evitar surpresas, recomenda-se moldar o tipo de variável, também chamado de *casting*, assegurando que a variável será do tipo esperado. Para isso, pode-se utilizar a função settype(), desta forma:

Listagem 15: Usando settype

```
1 //Retorna 'integer', mesmo com o valor declarado como 'string'
2 $var = "6";
3 settype($var, 'integer');
4 print gettype($var);
```

A moldagem de tipos também pode ser feita no momento da declaração da variável, antecedendo o seu valor com o tipo desejado, entre parênteses:

Listagem 16: Fazendo casting de variáveis

```
1 //Define o tipo 'integer'
2 $var = (int) "6";
3 print gettype($var);
```

Mais detalhes podem ser vistos no link a seguir.

Tipos de Dados do PHP

5 Adotar Padrões para Nomes de Funções, Classes, Constantes e Variáveis

Adotar padrões de nomenclatura ao declarar funções, variáveis, constantes e classes auxilia o desenvolvimento. Com isso, evitam-se dúvidas e erros, por exemplo, ao chamar funções que não existam, simplesmente por se ter um nome confuso ou uma falta de padrão. Você deve estar ciente da padronização que irá fazer, e respeitá-la em todo o projeto.

Vejamos um exemplo.

Listagem 17: Falta de padronização

```
1 function primeira_funcao()
2 {
3       //...
4 }
5
6 function segundaFuncao()
7 {
8     //...
9 }
```

Programar sem um padrão, como no exemplo acima, compromete o desenvolvimento. Dessa forma será muito fácil chamar por funções inexistentes. É muito importante analisar os padrões já utilizados por outros programadores ou desenvolver seu próprio padrão e seguí-lo à risca, garantindo um projeto claro e consistente.

Essa mesma situação pode ser enfrentada para nomes de constantes, variáveis ou classes. Por isso, é importante seguir um padrão, como o exemplo a seguir.

Listagem 18: Código com padronização

```
1  $var = 7;
2  $minhaVar = "PHP";
3  define( "CONSTANTE", "Valor da constante" );
4  define( "SEGUNDA_CONSTANTE", 100 );
5
6  function minhaFuncao( $argumento )
7  {
8      print "<b>$argumento<b>";
9  }
10
11 classe MinhaClasse
12 {
13     public function nomeMetodo()
14     {
15         // implementação
16     }
17 }
```

Para o nome de variáveis, prefira utilizar nomes sempre em minúsculas. Para variáveis com palavras compostas, separe-as alterando as palavras seguintes com letra inicial maiúscula (técnica chamada de *CamelCase*).

Nomes de constantes preferencialmente devem ser declarados em maiúsculas, utilizando *underline* ("_")

como separador de palavras.

As funções e seus argumentos, assim como as classes e seus métodos, devem ser declaradas utilizando-se o mesmo padrão para as variáveis. Como forma de diferenciar as classes de funções, estas devem ter seus nomes com a inicial também em maiúscula.

Mais detalhes e padrões sugeridos podem ser vistos no link a seguir.

PSR-2: Guia de Estilo de Codificação

6 Habilitar Exibição de Erros em Ambiente de Desenvolvimento

O PHP é uma linguagem muito flexível. Isso, pra variar, tem seus lados positivos e negativos. Uma das coisas que o PHP permite fazer é esconder mensagens de erro. Sabe quando você vai limpar a casa e joga a poeira embaixo do tapete? É feio fazer isso, né? Poie é. E esconder erros é tão feio quanto isso.

Por isso é muito importante manter a exibição de **todas** as mensagens de erro durante o desenvolvimento do seu projeto. As mensagens de erro te ajudam a melhorar seu código, encontrar os problemas e corrigi-los.

Existem duas diretivas do `php.ini` que são responsáveis por exibir os erros: `display_errors` e `error_reporting`. A primeira é simples: recebe valor 0 (erros desativados) ou 1 (erros ativados). A segunda recebe valores que variam conforme os níveis de erros que desejamos exibir. O ideal é exibir **todos** os níveis de erros e alertas. Para isso, podemos fazer o seguinte:

Listagem 19: Habilitando todas as mensagens de erro do PHP

```
1 <?php
2 ini_set( 'display_errors', 1 );
3 error_reporting( E_ALL | E_STRICT );
4
5 // restante do código
```

NOTA: A partir do PHP 6, o nível `E_STRICT` será inserido no nível `E_ALL`. Ou seja, quando o PHP 6 estiver disponível e você o estiver utilizando, poderá definir `error_reporting` apenas com o valor `E_ALL` em vez de `E_ALL | E_STRICT`.

Vale lembrar que, em ambiente de produção, o ideal é não exibir as mensagens de erro. Isso porque elas mostram detalhes internos do seu projeto, como a linguagem utilizada, o *framework* (caso esteja utilizando algum) e a estrutura de diretórios. Logo, em ambiente de produção, prefira manter `display_errors` com o valor 0. Nessa situação, é preferível fazer log dos erros. Falaremos de logs mais à frente.

Veja mais detalhes sobre as funções de manipulação de erros no link a seguir:

Funções para Manipulação de Erros

7 Não ocultar erros utilizando @ ([arroba])

Ainda falando de sujeira embaixo do tapete...

Pior que esconder erros com `display_errors` e `error_reporting` é esconder erros com o @ (arroba)!

Assim como as diretivas acima, o arroba existe e tem sua utilidade. Esse operador somente deve ser usado nos casos em que a ocorrência de erro seja uma exceção, como na função `mail()` e funções de conexão com banco de dados, IMAP e semelhantes.

Veja um exemplo de como usar o arroba de forma correta:

Listagem 20: Situação onde é correto usar o arroba

```
1 if ( ! @mysqli_connect ("servidor", "usuario", "senha" ) )
2 {
3     exit( "Erro ao conectar com o banco de dados MySQL" ).
4 }
```

Considerando-se que os argumentos da função `mysqli_connect()` acima estejam corretos, erros de conexão tornam-se uma exceção, como em caso de servidor fora do ar.

A partir do PHP 5, é possível manusear erros por meio de Exceções, mas esse tema não será abordado neste guia.

Nota importante: O operador de controle de erro "@" sempre desativa mensagens de erro, mesmo para erros críticos, que terminam a execução de scripts. Além de outras coisas, isto significa que se você usar "@" para suprimir erros de certas funções e elas não estiverem disponíveis ou com tipos incorretos, o script vai parar exatamente aí, sem nenhuma indicação do motivo da interrupção.

Veja mais detalhes no link a seguir.

Operadores de controle de erro

8 Desabilite register_globals

A `register_globals` é uma diretiva que passou a estar desabilitada (*Off*), por padrão, a partir do PHP 4.2.0. Ela, quando habilitada no `php.ini`, faz com que as variáveis globais sejam registradas (inicializadas) diretamente no script em execução.

Listagem 21: Funcionamento da register_globals

```
1  // Script acessado com valor via GET
2  // meu_script.php?var=Teste
3  if ( isset( $var ) )
4  {
5      print "A variável foi inicializada pela register\_globals;
6  }
7  else
8  {
9      print "A variável $var não foi inicializada";
10 }
```

O exemplo acima, apesar de inofensivo, pode sugerir a vulnerabilidade que o uso incorreto da diretiva `register_globals` pode acarretar. Imagine uma função que verifica a autenticação do usuário e define uma variável como verdadeira:

Listagem 22: Vulnerabilidade ao usar register_globals

```
1  if ( usuario_logado() )
2  {
3      $usuario_autenticado = true;
4  }
5
6  if ( $usuario_autenticado )
7  {
8      include "pagina_secreta.php";
9  }
10 else
11 {
12     print "Você não tem permissão.";
13 }
```

Neste exemplo fica clara a brecha de segurança. Basta inicializar a variável `$usuario_autenticado` como verdadeira (true) ao acessar o script para que a página secreta esteja disponível, burlando a função de verificação de autenticação do usuário. Se o atacante descobrir o nome dessa variável, basta ele passar o valor `true` pela URL, assim: `site.com/pagina.php?usuario_autenticado=true`. Pronto. Ele será considerado um usuário logado.

Para corrigir este problema, basta inicializar a variável com o valor `false`. Além de uma boa prática, previne que o valor seja "forçado" a ser verdadeiro.

Listagem 23: Evitando vulnerabilidade ao usar register_globals

```
1  // Inicializando a variável $usuario_autenticado
2  // como false, evitando que a register_globals
```

```php
// interfira no código
$usuario_autenticado = false;
if ( usuario_logado() )
{
    $usuario_autenticado = true;
}

if ( $usuario_autenticado )
{
    include "pagina_secreta.php";
}
else
{
     print "Você não tem permissão.";
}
```

Se você usa PHP 5.4 ou superior, não precisa se preocupar com `register_globals`, pois ela nem sequer existe mais!

Isso equivale a dizer que ela está desativada, como é o recomendado.

A maioria já deve usar PHP 5.4 ou superior, mas preferi incluir esta Boa Prática neste guia, principalmente por questões históricas, já que essa diretiva existiu por muito tempo e causou muitas discussões.

Mais detalhes podem ser lidos neste link:

Usando a diretiva Register Globals

9 Utilizar Arquivo de Inicialização (Bootstrapping)

Um arquivo de inicialização (*Bootstrapping*) é um arquivo que é processado em todas as requisições. Ele contém configurações gerais do sistema, que serão úteis em diversas partes da aplicação.

Essa técnica é muito importante pois ela garante integridade ao seu sistema. Seguindo esse modelo, é possível centralizar as configurações do sistema. Assim, sempre que precisar alterar uma configuração, basta fazer isso em um único arquivo, e ela se refletirá em toda a aplicação.

Os pontos mais importantes em um arquivo de inicialização de sistemas são:

- Definir configurações do PHP para toda a aplicação;
- Definir valores de constantes que serão usadas na aplicação (configurações gerais, como credenciais de acesso a bancos de dados).

Normalmente, nomeamos esse arquivo como `init.php`. Você deve incluí-lo no topo de todos os scripts que são acessados diretamente. Caso utilize um sistema em que todas as requisições passem, primeiro, pela `index.php` (que é o recomendado), basta incluir o `init.php` no `index.php`.

Escrevi um post no meu blog específico sobre isso. Leia-o no link a seguir.

Bootstrapping com PHP e Arquivo de Inicialização

10 Gerenciamento de Dependências de Software

Dependências de Software são bibliotecas de terceiros utilizadas por sua aplicação.

Por exemplo, se você usa o PHPMailer para enviar emails, ele é uma **dependência** do seu projeto.

Por serem bibliotecas externas, é importante atualizá-las com frequência, para que nossas aplicações utilizem sempre a versão mais atualizada de suas dependências.

E gerenciar diversas dependências, em um projeto grande, pode ser trabalhoso.

Por isso foi criado o **Composer**.

O Composer é o gerenciador de dependências para PHP mais utilizado hoje em dia em todo o mundo. Ele permite que você defina bibliotecas externas usadas em seu projeto de forma simples.

O próprio Composer se encarrega de baixar a biblioteca e carregá-la automaticamente em sua aplicação.

É ridiculamente fácil usar o Composer! E ele faz todo o trabalho pesado de gerenciar as bibliotecas, baixar as versões corretas e carregá-las em seu projeto.

Neste artigo eu falo sobre a principal funcionalidade do Composer, que é o gerenciamento de dependências.

Porém ele possui diversos outros recursos. Outra excelente funcionalidade dele é o *autoloader*, que carrega todas as suas classes e inclui arquivos automaticamente, sem dores de cabeça e seguindo o padrão PSR-4.

Para conhecer tudo sobre o Composer, veja o meu curso **Ultimate Composer**. Nele falo não apenas sobre o gerenciamento de dependências, mas, também, sobre o *autoloader*, os comandos do Composer e todas as configurações que ele permite fazer.

Aproveite que o curso está em promoção, por tempo limitado, custando **apenas R$ 7,00**!

Conheça Melhor o Curso Ultimate Composer

11 Mantenha-se Sempre Atualizado(a)

A linguagem PHP está em constante atualização. Novas versões, com correções e novos recursos, são frequentemente disponibilizadas. Por isso, acesse periodicamente o site oficial do PHP (http://www.php.net) e verifique se há uma nova versão. Se houver, baixe-a e instale-a o mais breve possível.

É importante ter sempre essas atualizações, pois elas normalmente corrigem falhas e brechas de segurança. Logo, com todos os recursos atualizados (PHP, Sistema Operacional etc), será muito menos provável que seu sistema seja vítima de ataques.

Também é importante prestar atenção às alterações feitas em cada versão, para que um script não gere incompatibilidade entre duas ou mais versões do PHP. Para isso, leia o *ChangeLog* das versões.

Mais detalhes podem ser vistos nos links a seguir:

- Usando códigos antigos com a nova versão do PHP
- Função `phpversion`
- Mantendo-se Atualizado

12 Comentários e Documentação

Você não tem uma memória perfeita e infinita. Não adianta dizer que tem!

Muitas vezes, tomamos alguma decisão sobre nosso script que, após algumas semanas ou alguns meses, não lembramos a razão. Por isso é importante inserir comentários nos scripts, explicando por que determinadas ações foram tomadas.

Mais que isso, você não é o(a) único(a) programador(a) do mundo! UFA! Imagine se fosse! Teria que programar 36 horas por dia! :P

Ou seja, você vai precisar dar suporte em códigos de outras pessoas. E outras pessoas darão suporte aos seus códigos.

Além de não termos memória infinita, não temos bola de cristal! Nunca saberemos o motivo de algo que um outro programador fez em um script sem comentários.

Mais um motivo para comentar seu código!

Além do bom e velho comentário, existe a Documentação. Existem ferramentas, como o **PHPDocumentator**, que geram documentações completas a partir de comentários que seguem o seu padrão. Veja um simples exemplo:

Listagem 24: Exemplo de código documentado para o PHPDocumentator

```
1  /**
2   * Descrição da classe MinhaClasse
3   */
4  class MinhaClasse
5  {
6      /**
7       * Descrição do método meuMetodo
8       * @param  tipo $param1 Descrição do parâmetro
9       * @param  tipo $param2 Descrição do parâmetro
10      * @return tipo_retorno Descrição do retorno
11      */
12     public function meuMetodo( $param1, $param2 )
13     {
14
15     }
16 }
```

Você pode conhecer melhor o PHPDocumentator, e todas as suas tags, no site oficial. Veja o link a seguir.

http://www.phpdoc.org

13 Versionamento de Código

Controlar versão é algo muito útil. A cada modificação que você fizer no código, você cria um *commit*, que é um ponto de mudança. Caso você faça modificações futuras e se arrependa delas, poderá voltar facilmente para pontos antigos.

O Git é uma das ferramentas mais usadas hoje em dia. E, na minha opinião, a melhor de todas elas. Outra bem conhecida e utilizada é o SVN.

Independentemente se você vai escolher Git ou SVN, procure sempre versionar seus projetos. Tudo ficará bem mais organizado.

Quer uma dica? Use Git! :P

No meu blog tenho um post sobre o básico do Git. Veja:

Git: controlando versão de seus programas

14 Faça Logs de Execução

Logs são sequências de mensagens, que criam um histórico de tudo o que ocorreu durante a execução de um programa. Geralmente é colocada a data e a hora do acontecimento. Quem está familiarizado com ambientes Linux (e outros tipos de Unix) deve conhecer o diretório `/var/log`, onde são armazenados os arquivos de logs do sistema de de outros programas.

Um arquivo de log normalmente se parece com o exemplo a seguir.

```
[data hora] Realizando operação 1
[data hora] Realizando operação 2
[data hora] Realizando operação 3
```

Muitas vezes ocorre algum erro no sistema e não sabemos qual a sua causa. É nesse momento em que muitos saem enchendo o código de `echo` e `var_dump` em todo lugar, pra descobrir onde a execução parou e por qual motivo.

Isso não é totalmente errado. Porém, não é a forma mais adequada e prática de se depurar nossos programas.

O ideal seria criar logs de execução. Assim, toda ação (ou apenas as principais ações) é registrada em um arquivo de log. Esse arquivo nada mais é que um simples arquivo de texto, com a data e a hora da ação e sua descrição.

Dessa forma, sempre teremos um arquivo que lista as principais ações do nosso sistema, juntamente com o horário preciso de quando elas ocorreram.

Uma simples função para gerar logs segue esta lógica:

Listagem 25: Exemplo de função para gerar logs

```
1  function logMsg( $msg, $level = 'info', $file = 'main.log' )
2  {
3      switch ( $level )
4      {
5          case 'info':
6              $msg = '[INFO] ' . $msg;
7              break;
8
9          case 'warning':
10             $msg = '[WARNING] ' . $msg;
11             break;
12
13         case 'error':
14             $msg = '[ERROR] ' . $msg;
15             break;
16     }
17
18     // data atual
19     $date = date( 'Y-m-d H:is' );
20
21     $msg = '[' . $date . '] ' . $msg;
22
23     // adiciona quebra de linha
24     $msg .= PHP_EOL;
25
26     file_put_contents( $file, $msg, FILE_APPEND );
```

```
27 }
```

A função `logMsg` aceita três parâmetros, mas só o primeiro é obrigatório. São eles:

1. Mensagem a ser inserida no log
2. (opcional) Nível da mensagem (pode ser "info", "warning" ou "error")
3. (opcional) Arquivo onde o log será escrito

Alguns exemplos de uso:

Listagem 26: Exemplo de uso da função para gerar logs

```
1  logMsg( 'Executando ação X' );
2  // ação x aqui
3
4  if ( ! isset( $y ) )
5  {
6      logMsg( 'Valor y não definido. Usando valor padrão', 'warning' );
7  }
8
9  if ( erro_terrivel_aconteceu() )
10 {
11      logMsg( 'Oh, não! Isso não deveria ter acontecido', 'error' );
12      exit();
13 }
```

15 Use Codificação UTF-8 Sem BOM

Seus arquivos de código-fonte possuem uma codificação. Ou seja, um conjunto de caracteres válidos. Você pode visualizar e alterar a codificação em qualquer editor de textos.

As duas codificações mais utilizadas são ISO-8859-1 (também chamada de **Latin1**, ou apenas *ISO*) e a UTF-8.

A ISO não consegue representar os mesmos caracteres que a UTF-8 consegue. Se estivermos falando apenas do idioma inglês, sem caracteres especiais, a ISO resolve praticamente todos os problemas. Mas, se incluirmos as letras acentuadas do Português, além de caracteres europeus e orientais, é necessário usar UTF-8.

Por isso é importante adotar UTF-8 como padrão. E é preciso configurar o editor de textos pra usar essa codificação. Mas não para por aí. O cabeçalho HTML também deve ser definido para UTF-8, bem como o banco de dados, caso sua aplicação o utilize.

Mais sobre isso pode ser visto no seguinte post que escrevi no meu blog:

Problemas com codificação: acentos não interpretados

O *BOM* (*Byte Order Mark*, ou **Marca de Ordem de Byte**) é uma sequência de caracteres que é inserida no início de um arquivo para definir a ordem dos *bytes*.

No UTF-8, o uso do BOM é desnecessário, inclusive é não recomendado. Isso por que o UTF-8 dispensa o uso dessa sequência. Se utilizado, haverá caracteres inválidos, que poderão causar comportamento inesperado na sua aplicação, principalmente com funções que envolvem cabeçalho HTTP (como cookies, sessões e a função `header`). Isso ocorre pois esse caractere inválido é enviado para o *buffer* de saída e, ao tentar enviar cabeçalhos, é gerado um erro no PHP dizendo que os cabeçalhos já foram enviados.

Em suma, use sempre UTF-8 sem BOM e não terá problemas.

Conclusão

Se você chegou até aqui, é porque realmente quer aprender a programar do jeito certo!

PARABÉNS!

Poucos começam com essa mentalidade.

E é ótimo que você esteja dentro desse grupo seleto de pessoas que desejam fazer as coisas da melhor maneira possível!

15.1 Um Pequeno Favor...

Agora que te pedir um simples favor.

Se este guia foi realmente útil pra você, clique nos botões abaixo e compartilhe este material com seus amigos nas Redes Sociais.

Ajude a melhorar o conhecimento de todos os Programadores PHP do Brasil!

Vamos juntos preparar nossos programadores para serem profissionais excepcionalmente incríveis no que fazem!

Estamos juntos nessa!

Um Pouco Sobre Mim e o PHP

Meu nome é Roberto Beraldo, mais conhecido apenas por **Beraldo**.

Eu conheci o PHP lá pelos idos de 2005 ou 2006. Eu conhecia HTML e precisava fazer um simples formulário de contato para um site que eu tinha feito. Foi aí que trombei com o PHP. E desde então não o larguei mais! :)

Aprendi grande parte do PHP lendo a documentação e analisando dúvidas de outras pessoas, em alguns fóruns, como o **iMasters**, do qual, aliás, sou moderador até hoje.

Tentei ler alguns livros sobre PHP, mas nunca gostei muito da didática deles. Outro problema era que, naquela época, o PHP 5 tinha acabado de ser lançado. Muitas práticas antigas já eram consideradas obsoletas na versão 5. E sem contar que o PHP 5 trouxe muitas novidades. Ou seja, os livros estavam **todos** obsoletos!

Mais tarde, em 2008, iniciei o curso de **Bacharelado em Ciência da Computação**, na **Universidade Federal do Paraná (UFPR)**. Foram anos bem corridos, cheios de trabalhos, noites de pouco sono e muito estudo. Mas valeu a pena! Aprendi muita coisa!

Nesse meio tempo, resolvi criar o meu blog, o **Blog do Beraldo**. Lá escrevi (e ainda escrevo) alguns tutoriais, artigos e dicas, sobre diversas áreas da computação, mas principalmente sobre PHP.

Durante todo esse tempo, nunca deixei de estudar e trabalhar com PHP. Mais do que isso, estive sempre participando do **Fórum iMasters**, ajudando o pessoal principalmente de PHP. Ou seja, sei bem quais são as principais dificuldades dos iniciantes.

E assim o **Blog do Beraldo** foi crescendo. Muitos visitantes me mandavam mensagens sugerindo que eu criasse um curso completo de PHP, pois gostavam muito dos artigos que eu postava no Blog e também no **iMasters**. Eu sempre agradecia o reconhecimento, mas dizia que não tinha tempo para criar um curso completo.

Depois de algum tempo, pensei:

> Se tantos me pedem para criar um curso, por que não criar?

Eu sabia exatamente as principais dúvidas dos iniciantes e intermediários em PHP. Ou seja, eu poderia ensinar tudo isso de forma fácil, enfatizando os pontos mais críticos.

E assim surgiu o **ULTIMATE PHP 2.0**!

15.2 Fundamentos: Por Que Insisto Tanto Nisso

Falei bastante aqui sobre **Fundamentos**.

Mas, afinal, por que isso é tão importante?

Simples...

Tecnologias novas vêm e vão. Sempre vai surgir uma nova ferramenta, uma nova biblioteca, um novo *framework*...

Mas tem uma coisa que essas ferramentas usam e que **NUNCA** muda nem vai embora...

Adivinha o que é...?

Sim, os **Fundamentos**.

Tem muita gente por aí preocupada achando que precisa estudar a Ferramenta X, o Framework Y...

Lógico que é bom conhecer tudo isso. Mas se você não conhecer os Fundamentos do PHP, nunca vai entender direito como essas ferramentas funcionam.

Ou até pior...

Nunca vai saber se virar sem elas!

É disso que falo **neste vídeo**.

Por isso hoje insisto tanto em ensinar os Fundamentos do PHP.

Tem muito curso por aí ensinando diversas ferramentas, mas sem focar no que realmente importa.

E é por isso que o **Curso ULTIMATE PHP 2.0** é diferente de todos os outros.

Um Convite

Gostaria de convidar você a conhecer o curso **ULTIMATE PHP 2.0**.

Eu criei esse curso especialmente para você, que faz parte desse pequeno grupo que deseja começar já do jeito certo, aprendendo todos os Fundamentos do PHP, sem perder tempo com mini-cursos e apostilas obsoletas e defasadas, que ensinam PHP da forma errada.

A versão 2.0 do curso está atualizada com todas as novidades do PHP 7.

Aliás, o **ULTIMATE PHP 2.0** foi o primeiro curso do Brasil a ensinar sobre o PHP 7!

CONHEÇA O ULTIMATE PHP 2.0 E APRENDA OS FUNDAMENTOS DO PHP EM 7 SEMANAS OU MENOS

Obrigado

Gostaria de agradecer você novamente por ter lido este guia.

Espero que você tenha gostado e achado útil.

Mais que isso, espero que você utilize essas dicas no seu dia-a-dia de programação.

Muitas vezes, na correria, esquecemos de alguns "bons modos" da Programação, criando as famosas "gambiarras". Nem sempre é fácil seguir o melhor caminho, principalmente quando você está em uma empresa e o chefe quer que você faça o super hiper mega master sistema sofisticado para gestão de forças ocultas do escritório. Aí você faz da forma rápida, deixando as Boas Práticas de lado...

> "Que Feio!!"

Nesse caso (**e apenas nesse caso!**) eu vou te dar um desconto. Entendo que você não queira perder seu emprego, então precisa cumprir o prazo que o patrão estipulou.

Mas, mesmo assim, não esqueça das Boas Práticas. Tenha-as sempre em mente, siga-as sempre que puder e tente explicar pro seu chefe que as coisas bem feitas precisarão de menos manutenção posteriormente. E isso é bom pra você e pra ele. Você não precisará voltar a mexer nos códigos pra corrigir as gambiarras. E seu chefe não gastará tempo e dinheiro em manutenção de *software*.

Ok... nem é sempre fácil convencer nossos chefes sobre isso.

Mas não custa tentar, né? :)

E O Que Fazer Agora...?

O que eu mostrei pra você aqui é só o começo. Há muitas técnicas e Fundamentos de Programação PHP!

E de nada adianta conhecer as Boas Práticas sem **conhecer bem** a linguagem com a qual estamos trabalhando e seus Fundamentos.

Por isso você **precisa** aprender os **Fundamentos** do PHP, pra colocar em prática as técnicas e dicas que aprendeu neste guia.

E, coincidentemente, eu criei o **curso perfeito** para você, que **quer** aprender PHP **de verdade** e **do jeito certo**, sem enrolação, sem perder tempo!

CONHEÇA O ULTIMATE PHP 2.0 E APRENDA OS FUNDAMENTOS DO PHP EM 7 SEMANAS OU MENOS

Conheça o curso! Tenho certeza de que não irá se arrepender.

Vejo você no curso! Até mais!

Ah, e caso você ainda não tenha compartilhado este guia, você tem mais uma chance agora! :)