PHP 5.3 parte 1: Namespaces

Muitos dos recursos previstos para a versão 6 do php foram incluídos na versão 5.3. Namespaces é uma delas. Namespaces vem para ajudar a evitar conflitos entre nome de funções, classes e constantes. Até a versão 5.2 muitas aplicações utilizavam prefixos nos nomes de classes e funções para evitar estes conflitos. Na aplicação de blog chamada wordpress, usa-se o prefixo “wp_” em nome dos elementos. Por exemplo o nome de algumas funções: wp_update_post, wp_create_user !
Com Namespaces você agrupa classes, funções e constante de forma que possa haver elementos com nomes iguais em diferentes “grupos” ou namespaces, funcionando sem conflitos. Dessa forma, pode-se diminuir a utilização de prefixos, deixando o código mais limpo.

Definindo Namespaces

Para definir um namespace com o nome de foo:

```
<?php
namespace foo;
```

Vamos montar um ambiente de testes para entender melhor o funcionamento de namespaces. Segue a abaixo a declaração de duas classes com o mesmo nome, chamadas User, cada uma em um namespace diferente:

Arquivo UserBlog.php, namespace Blog:

```
<?php
namespace Blog;

class User {
   private $name;

   public function setName ($username) {
     $this->name = $username;
   }

   public function getName() {
     return "The username in Blog is " . $this->name;
   }

}
```

Arquivo UserCms.php, namespace Cms:

```
<?php
namespace Cms;

class User {
   private $name;

   public function setName ($username) {
     $this->name = $username;
```

```
   }

   public function getName() {
     return "The username in CMS is " . $this->name;
   }

}
```

PHP 5.3 parte 1: Namespaces

Usando Namespaces

É possível referenciar os elementos de uma namespace em até 3 maneiras diferentes:

* Nome não qualificado. Exemplo de uma chamada de nome não qualificado: *$a = new User();* Se a namespace corrente do arquivo for “Blog”, *namespace Blog;* , a declaração anterior será transformada para *Blog\User();*. Caso o código for global, ou seja, código sem namespace, será transformada para simplesmente *User();*

* Nome qualificado. São chamadas utilizando nomes relativos, quando usa-se sub-namespaces. Exemplo: *namespace Company\Blog;*. A seguinte declaração instancia a classe User utilizando “Nome Qualificado”: *$a = new Blog\User();*. Como a namespace raiz do arquivo é “Company”, a declaração anterior será transformada para *Company\Blog\User();*. Vamos ver a utilização de sub-namespaces mais adiante.

* Full Qualified Name. São chamadas utilizando nomes absolutos. Exemplo: *$a = new \Blog\User(); $b = new \Cms\User(); $c = new \Company\Blog\User();*

Vamos criar um novo arquivo para exemplificar a utilização de namespaces:

Arquivo TestNamespace.php:

```
<?php
// incluir as classes com mesmo nome, porém com diferentes namespaces.
require_once("UserBlog.php");
require_once("UserCms.php");

// Instanciar class User do namespace Blog. Usando full qualified name.
$user = new \Blog\User();
$user->setName("Douglas");
print $user->getName();

print PHP_EOL;

// Instanciar class User do namespace Cms. Usando full qualified name.
$user2 = new \Cms\User();
$user2->setName("Douglas");
print $user2->getName();
```

A saída do script acima:

```
The username in Blog is Douglas
The username in CMS is Douglas
```

Nome não-qualificado

No exemplo anterior instanciamos as classes User de diferentes namespaces usando o full-qualified-name. Para ilustrar a chamada de uma classe usando nome não-qualificado vamos definir a namespace do arquivo TestNamespace.php para “Blog”:

```
namespace Blog;
```

Desta maneira temos o arquivo TesteNamespace.php na mesma namespace que a classe User do Arquivo UserBlog.php. Portanto no lugar de usar “\Blog\User();” podemos simplesmente usar:

```
$user = new User();
```

Para instanciar a classe User do namespace Cms, continuamos a ter que usar o full-qualified-name.

Sub-namespaces

Semelhantes à diretórios e arquivos, namespaces em php podem ser declaradas em forma de hierarquia. Exemplo:

```
namespace Project\Blog;
```

Nome Qualificado

Para exemplificar a chamada de uma classe usando nome qualificado vamos definir sub-namespaces em nossos 3 arquivos:

Troque a linha que define a namespace para cada um dos arquivos:

UserBlog.php

```
namespace Project\Blog;
```

UserCms.php

```
namespace Project\Cms;
```

TesteNamespace.php

```
namespace Project;
```

Como todos os arquivos tem definido o namespace raiz para Project, podemos usar “nome-qualificado”, ou nome relativo, para acessar as classes dentro do arquivo TesteNamespace.php:

```
$user = new Blog\User(); // transformado para \Project\Blog\User();
...
$user1 = new Cms\User(); // transformado para \Project\Cms\User();
```

Automaticamente *$user = new Blog\User()* vai ser transformado para *$user = new \Project\Blog\User();* e *$user1 = new Cms\User();* vai ser resolvido para *$user1 = new \Project\Cms\User();* de forma transparente.

PHP 5.3 parte 1: Namespaces

Importing e Aliasing

A opção de referenciar um elemento utilizando um nome mais curto, ou *alias*, é uma importante característica de namespaces.

A palavra chave *use* é utilizada para importing/aliasing.

Vamos alterar nosso arquivo TesteNamespace.php para exemplificar a utilização de importing/aliasing.

TesteNamespace.php:

```
namespace Project;
use Project\Blog\User as BlogUser;
use Project\Cms\User; // É o mesmo que utilizar "use Project\Cms\User as User;"
...
$user = new BlogUser(); // Instancia o objeto da classe Project\Blog\User
...
$user2 = new User(); // Instancia o objeto da classe Project\Cms\User
```

Obs: Nomes full qualified são absolutos e não são afectados por imports.

PHP 5.3 parte 1: Namespaces

Global Space

Sem a definição de namespace, todas as classes e funções pertencem ao espaço global . Funciona da mesma maneira que as

antigas versões do php onde ainda não existiam namespaces.

Vamos criar uma nova classe User, desta vez pertencendo ao espaço global. Perceba que não definimos nenhuma namespace. Isto significa que a classe pertence ao espaço global.

Arquivo User.php, “global space”:

```
<?php
class User {
   private $name;

   public function setName ($username) {
     $this->name = $username;
   }

   public function getName() {
     return "The username in Global is " . $this->name;
   }

}
```

Para chamarmos um nome que esta no espaço global, devemos colocar o prefixo “\” no nome, ou simplesmente chamar o nome diretamente caso não exista nomes iguais no namespace corrente.

Vamos alterar mais uma vez o arquivo TesteNamespace.php onde vamos adicionar a chamada para a classe User que esta no espaço global.

TesteNamespace.php:

```
...
require_once("User.php"); // incluir classe global
...
print PHP_EOL;
$user3 = new \User(); // chamando a classe "User" no espaço global.
$user3->setName("Douglas");
print $user3->getName();
...
```

A nova saída após a execução do arquivo TesteNamespace.php:

```
The username in Blog is Douglas
The username in CMS is Douglas
The username in Global is Douglas
```

PHP 5.3 parte 1: Namespaces

Funções e constantes

Além das classes, namespaces podem ser usadas em funções e constantes.

```
<?php
namespace A\B\C;
const TESTCONST = true;


function fopen() { // função em A\B\C\fopen
  $f = \fopen(...); // espaço global
  return TESTCONST; // constante em A\B\C\TESTECONST
}
```

PHP 5.3 parte 1: Namespaces

A constante __NAMESPACE__

O valor da constante __NAMESPACE__ é uma string com o valor da namespace corrente. Em um código sem namespace, espaço global, o valor da constante é um string vazia.

A palavra chave “namespace” também pode ser usada para requisitar um elemento referente a namespace corrente.

```
<?php
namespace Foo\Bar;

echo __NAMESPACE__; // exibe Foo\Bar
namespace\func(); // chama a função em Foo\Bar\func();
```

PHP 5.3 parte 1: Namespaces

Multiplos namespaces em um mesmo arquivo

PHP permite que você defina mais de uma namespace em um mesmo arquivo. É possível mesclar código sem namespace com código com namespace. Ao utilizar mais de uma namespace em um mesmo arquivo é fortemente recomendável o uso de chaves “{“. Exemplo:

```
namespace Projeto\Secao1 {
// código inserido arqui estará no namespace Projeto\Secao1
    const USER_STATE = 1;
    function get_user_state() { ... }
    class UserState() { ... }
}

namespace Projeto\Secao2 {
// código inserido arqui estará no namespace Projeto\Secao2
    const USER_STATE = 1;
    function get_user_state() { ... }
    class UserState() { ... }
}

namespace {
// código inserido arqui estará no espaço global.
    const USER_STATE = 1;
    function get_user_state() { ... }
    class UserState() { ... }
}
```

PHP 5.3 parte 2: Late Static Bindings

Na primeira parte desta série abordamos a utilização de namespaces. Nessa segunda parte vamos abordar a utilização de “Late Static Bindings” que é um novo recurso incluído à partir da versão 5.3, que por sinal, é muito interessante.
É mais comum o uso de “Late static bindings” em chamadas de métodos staticos em um contexto de herança. Apesar que não ser limitado somente a métodos staticos.

Para melhor entendermos onde se aplica a utilização de “Late static bindings” vamos analisar o código abaixo:

```
<?php
class A {
  public static function who() {
    echo __CLASS__;
  }
  public static function test() {
    self::who();
  }
}

class B extends A {
  public static function who() {
    echo __CLASS__;
  }
}
```

```
B::test();
```

As referências staticas para a classe corrente, como *self::* e *__CLASS__*, são resolvidas para a classe no qual o método esta definido e não para a classe chamada em tempo de execução, que no caso a acima foi a class *B*, filha da classe *A*.
Dessa forma a saída da execução do script php anterior será:

```
A
```

Afim de resolver essa limitação foi escolhido uma palavra-chave para que seja possível referenciar a classe que é chamada em tempo de execução ao invés da classe em que o método pertence. A palavra-chave já reservada é *static::*.

Vamos alterar o script para que possamos acessar o médoto *who()* na classe *B*. Perceba que modificamos *self::* para *static::*.

```
<?php
class A {
  public static function who() {
    echo __CLASS__;
  }
  public static function test() {
    static::who(); // static:: no lugar de ::self
  }
}

class B extends A {
  public static function who() {
    echo __CLASS__;
  }
}

B::test();
```

Ao executar o script php anterior, o resultado agora é diferente:

```
B
```

PHP 5.3 parte 2: Late Static Bindings

A resolução de “Late static bindings” não acontece para chamadas staticas completas, usando o nome da classe. Exemplo

de uma chamada statica usando o nome completo: *A::foo();*. Veja o código abaixo:

```
<?php
class A {
  public static function foo() {
    static::who();
  }
  public static function who() {
    echo __CLASS__."\n";
  }
}

class B extends A {
  public static function test() {
    A::foo(); // Para essa chamada não acontece resolução de Late Static Bindings
    parent::foo();
    self::foo();
  }
  public static function who() {
    echo __CLASS__."\n";
  }
}

class C extends B {
  public static function who() {
    echo __CLASS__."\n";
  }
}
C::test();
```

Segue o resultado do código acima:

```
A
C
C
```

Podemos concluir no exemplo anterior que:

 * *A::foo()* não acontece a resolução de “Late static bindings”, mesmo usando a palavra-chace static:: dentro da função *foo()*. O nome da classe exibida é aquela no qual a função *who()* pertence, ou seja, classe *A*.
 * Para *parent::foo()* e *self::foo()* já acontece a resolução de “Late static bindings”. Portanto o nome da classe exibida é aquela que foi chamada em tempo de execução, ou seja, classe *C*.

PHP 5.3 parte 3: Lambda e Closures

Nesta terceira parte de nossa série vamos abordar a utilização de Lambda e Closures que também foram recursos incluídos na versão 5.3. Funções *Lambdas* são conhecidas como funções anônimas. Closures são uma forma mais avançada de *Lambdas* permitindo trabalhar com funções anônimas de forma mais flexível e com menos limitações. Vamos entender agora o funcionamento desses dois novos rescursos.

Lambdas

Lambdas são conhecidas como funções anônimas. Estas funções podem ser definidas em qualquer lugar, podendo ser atribuídas à uma variável. A função existe somente no escopo de onde a variável foi definida. Caso a variável saia fora do escopo, a função também se torna indisponível.

É mais comum o uso de *Lambdas* em lugares que aceitam funções callback como parâmetro. Um exemplo é a função *array_map*. Segue a abaixo um exemplo usando função anônima com *array_map*:

```
<?php
$r = array_map(function($value) { // criando a função lambda como parâmetro para array_map
               return $value * 4;
             }, array(2, 4, 6, 8, 10));

print_r ($r);
```

A saída do script anterior será:

```
8
16
24
32
40
```

Perceba que usamos apenas 1 parâmetro na função anônima, $value:. O número de parâmetros varia de acordo com o número de arrays passados para a função *array_map*.

Poderíamos ter utilizado a função *create_function*, disponível desde a versão 4 do php, para criar funções anônimas. Porém tem algumas desvantagens. Uma delas é que ela é compilada em tempo de execução (runtime) ao invés de ser em tempo de compilação (compile time). Isto impacta no desempenho do script, além de ser uma syntax desagradável.

PHP 5.3 parte 3: Lambda e Closures

Closures

O recurso de *Lambda* não adiciona nenhuma funcionalidade em especial em relação a versão 4, onde podemo reproduzir o uso de funções anônimas através da função *create_function*. É ai que entra em cena, no php 5.3, o recurso chamado *Closures*. *Closures* funcionam como *Lambdas*, porém de forma mais inteligente. Permitem a interação com variáveis fora do ambiente de onde são definidos.

Veja abaixo um exemplo bem simples da utilização de *Closures*:

```
<?php
$string = "Olá teste!";
$closure = function() use ($string) { echo $string; };

$closure();
```

A saída do código anterior:

```
Olá teste!
```

Variáveis de fora do ambiente do closure podem ser importadas através da palavra-chave *use*. No exemplo anterior a variável *$string* foi importada para dentro do closure. Por padrão as variáveis importadas são passadas por valor e não por referência. Dessa forma se quisermos que a variável importada seja modificada dentro do closure, e que a mudança continue aplicada após a chamada da função, devemos utilizar o operador & para especificar que a variável esta sendo passada por referência e não por valor.

Segue abaixo mais um exemplo da utilização de closures importando variáveis por referência:

```
<?php
$a = 10;
$closure = function() use (&$a) { $a += 50; };

$closure();
var_dump ($a);

$closure();
var_dump ($a);

$closure();
var_dump ($a);
```

A saída do script anterior:

```
int(60)
int(110)
int(160)
```

PHP 5.3 parte 3: Lambda e Closures

Método __invoke

Foi adicionado no php5.3 um método mágico chamado *__invoke*, utilizado em classes, que permite que o objeto seja chamado como se fosse uma função, ou closure. Segue o exemplo:

```
<?php
class DebugAll {
   public function __invoke($var) {
     var_dump($var);
   }
}

$obj1 = new DebugAll;
$obj1(50);
$obj1(100);
```

A saída do script anterior será:

```
int(50)
int(100)
```

PHP 5.3 parte 3: Lambda e Closures

Exemplos úteis de utilização de Closures

Closures são bastante úteis em funções que utilizam callbacks. Outra forma de tirar proveito de closures é refactorar velhos códigos afim de deixá-los mais limpos. Veja o exemplo abaixo, onde usamos o método *Logger::log* para logar uma observação para cada query executada no banco de dados:

```
<?php
$db = mysqli_connect("server","user","pass");
Logger::log('debug','database','Conectando com a base de dados');

$db->query("insert into users (nome, descricao) values ('Douglas','Administrador')");
Logger::log('debug','database','Insert Douglas into to users table');

$db->query("insert into users (nome, descricao) values ('Andréia','Redatora')");
Logger::log('debug','database','Insert Andreia into to users table');
```

Refactorando o código acima para utilizar closures deixamos o código mais agradável para ler e entender.
Segue abaixo o código refactorado:

```
<?php
// definindo o closure
$logdb = function ($string) { Logger::log('debug','database',$string); };

$db = mysqli_connect("server","user","pass");
$logdb('Conectando com a base de dados'); // usando o closure
$db->query("insert into users (nome, descricao) values ('Douglas','Administrador')");
$logdb('Insert Douglas into to users table'); // usando closure
$db->query("insert into users (nome, descricao) values ('Andréia','Redatora')");
$logdb('Insert Andreia into to users table'); // usando closure
```

PHP 5.3 parte 4: Arquivos Phar

Arquivos Phar são semelhantes aos arquivos JAR do Java. Recurso incluído na versão 5.3 do php, esta nova funcionalidade permite que você distribua sua aplicação PHP inteira ou uma biblioteca em um único arquivo, chamado de arquivo Phar. Esta funcionalidade tornou-se nativa do php na versão 5.3. Não existe uma ferramenta externa para criar arquivos phar como acontece no java. Toda manipulação, criação e utilização dos arquivos são realizados através de códigos PHP. Pelo fato de ser nativo, praticamente não há perca de desempenho em sua utilização.

Criando arquivos Phar

Para criar arquivos Phar será necessário alterar a diretiva *phar.readonly* no php.ini para *Off*. Por padrão, esta diretiva vem habilitada por motivos de segurança. Em servidores de produção, arquivos Phar não precisam ser criados, somente executados, por isso a preocupação com a segurança.

O primeiro passo para criar um arquivo Phar é instanciar a classe Phar:

```
<?php
$p = new Phar('/project/app.phar', 0, 'app.phar');
$p->startBuffering();
```

O primeiro parâmetro para o construtor da classe Phar é o caminho no sistema de arquivos onde será criado ou acessado o arquivo Phar.
O segundo parâmetro são flags passadas para a classe pai.
O terceiro parâmetro é um alias que poderá ser utilizado para referenciar o arquivo Phar em chamadas de funções de streams.
O método *startBuffering* é utilizado com a finalidade de desempenho. Todas mudanças que fizermos no arquivo Phar serão efetivadas somente quando chamarmos a função *stopBuffering*. Quando não utilizamos *startBuffering* cada mudança realizada no arquivo Phar será efetivada imediatamente.

PHP 5.3 parte 4: Arquivos Phar

Adicionando arquivos

Existem várias maneiras de adicionar arquivos dentro do pacote. Uma delas é utilizando o método *addFile*:

```
$p->addFile("/project/userBlog.php");
$p->addFile("/project/userCMS.php", "/lib/CMS.php");
```

Na primeira chamada de método addFile estamos adicionando o arquivo */project/userBlog.php* dentro do pacote. O segundo comando é semelhante, porém estamos passando um segundo parâmetro. Através do segundo parâmetro definimos o caminho que o arquivo será armazenado no pacote. Quando precisarmos utilizar esse arquivo no pacote, vamos referênciá-lo através de */lib/CMS.php*.

O objeto Phar utiliza ArrayAccess SPL que permite acessar o conteúdo do pacote Phar através de array. É a maneira mais simples de se adicionar arquivos:

```
<?php
$p['index.php'] = file_get_contents("/project/index.php");
$p['teste.txt'] = "Teste 123 456";
```

Normalmente quando precisamos adicionar muitos arquivos para dentro do pacote torna-se inviável utilizar as opções anteriores, onde temos que adiocionar um a um. Existem métodos que facilitam essa tarefa, como *buildFromDirectory* e *buildFromIterator*.

buildFromDirectory:

```
$p->buildFromDirectory('/project', '/\.php/');
```

A função *buildFromDirectory* adiciona os arquivos de um diretório dentro do pacote Phar. O segundo parâmetro, opcional, é uma expressão regular usada para filtrar os arquivos que serão incluídos. No exemplo anterior estamos incluíndo todos os arquivos com a extensão *.php* do diretório */project* dentro do pacote Phar.

PHP 5.3 parte 4: Arquivos Phar

Arquivo Stub

O arquivo Stub é um pedaço de código que será executado quando o arquivo Phar for carregado. Para definir o pedaço de código que será executado ao carregar o arquivo Phar pode-se usar o método *setStub* que aceita o código em forma de *string*. Exemplo:

```
$p->setStub('<?php Phar::mapPhar();
             include "phar://app.phar/index.php"; __HALT_COMPILER(); ?>');
```

O código deve terminar com a função *__HALT_COMPILER()*. O método statico *Phar::mapPhar()* lê e inicializa o arquivo Phar a ser executado. O streammer *phar://* é usado para referenciar os arquivos que estão dento do pacote Phar. No exemplo anterior ao carregar o arquivo *app.phar* será executado o arquivo *index.php*.

Caso não existam nenhum procedimento especial ao carregar o arquivo Phar, onde você deseja simplesmente exeturar um arquivo php diretamente, usá-se o método *createDefaultStub*. É só passá-lo como parâmetro para o método *setStub*.

```
$p->setStub($p->createDefaultStub('index.php'));
```

Vamos a um exemplo prático onde utilizaremos os conceitos já abordados até aqui. Criar arquivo index.php:

```
<?php
echo "Olá mundo";
```

PHP para criar pacote Phar (cria_phar.php):

```
<?php
$p = new Phar('teste.phar', 0, 'teste.phar');
$p->startBuffering();
$p['index.php'] = file_get_contents("index.php");
// $p->addFile('index.php'); - segunda opção para incluir arquivos.
$p->setStub($p->createDefaultStub('index.php'));
$p->stopBuffering();
```

Criando o arquivo teste.phar através da linha de comando:

```
# php cria_phar.php
```

Após a execução do script acima será criado o arquivo *teste.phar*.
Executando o pacote phar:

```
# php teste.phar
```

A saída da execução do script acima será:

```
Olá Mundo
```

PHP 5.3 parte 4: Arquivos Phar

Trabalhando com arquivos Phar

Integrar arquivos Phar em sua aplicação é simples e fácil. A maneira mais simples de se usar arquivos de dentro do pacote Phar é incluí-los em sua aplicação da mesma maneira que você inclui outros arquivos usando *include*.
Parar referênciar algum arquivo dentro do pacote Phar usá-se o streammer *phar://*. Segue abaixo o exemplo:

```
<?php
include 'teste.phar';
include 'phar://teste.phar/index.php';
```

Na primeira linha estamos incluíndo o pacote Phar. Ao rodar essa linha será chamado o código especificado no stub, que no caso é a execução do arquivo index.php. Na segunda linha estamos incluindo diretamente o arquivo index.php. Qualquer uma das duas chamadas são válidas e uma não depende da outra para rodar. Poderíamos ter chamado ou somente a primeira linha ou somente a segunda linha. De acordo com a segunda linha do exemplo podemos observar que é possível incluir qualquer arquivo existente dentro do pacote Phar.

Agora vamos ver um exemplo de como trabalhar com arquivos Phar via web. Para isso vamos utilizar o método statico *Phar::webPhar*. Ele funciona como um *front controller* redirecionando as requisições web para os arquivos dentro do pacote Phar. No próximo exemplo vamos criar um novo arquivo Phar para ser utilizado em um ambiente web:

```
<?php;
// Criando arquivo Phar:
try {
  $phar = new Phar('exemplo.phar');
  $phar['index.php'] = '<?php echo "Hello World Index"; ?>';
  $phar['admin.php'] = '<?php echo "Hello World Admin"; ?>';
  $phar->setStub('<?php Phar::webPhar(); __HALT_COMPILER(); ?>');
} catch (Exception $e) {
// Tratando erro
}
print "Arquivo Phar criado";
```

O método *webPhar()* deve estar dentro do código stub e será executado ao chamarmos o pacote exemplo.phar. Após criarmos o arquivo exemplo.phar devemos colocá-lo dentro do diretório raiz do servidor web. Devemos configurar o servidor web para que ele saiba o quê fazer com arquivos de extensão .phar. Segue abaixo a configuração que deve ser realizada no apache:

```
AddType application/x-httpd-php .php .phar
```

Apenas adicionamos a extensão .phar ao final da cláusula AddType já existente no httpd.conf.

Agora podemos chamar a aplicação pelo browser:

http://localhost/exemplo.phar

A saída no browser será:

```
Hello World Index
```

Para chamar o arquivo admin.php:

http://localhost/exemplo.phar/admin.php

A saída no browser será:

```
Hello World Admin
```

PHP 5.3 parte 4: Arquivos Phar

Outros formatos

Além do formato padrão, formato Phar, você pode trabalhar com outros formatos. Outros formatos permitidos são ZIP e TAR. Vamos aos exemplos:

Convertendo para ZIP:

```
$p = $p->convertToExecutable(Phar::ZIP);
```

Convertendo para TAR/GZ:

```
$p = $p->convertToExecutable(Phar::TAR, Phar::GZ, '.phar.tgz');
```

Finalizando

Existem diversas outras maneiras de se trabalhar os arquivos Phar. Neste artigo procurei demonstrar os principais métodos da classe Phar. Para se aprofundar acesse http://www.php.net/phar.

PHP 5.3 parte 5: Outras melhorias

Neste último artigo da série vamos abordar algumas melhorias pequenas incluídas na versão 5.3 do PHP. Nos artigos anteriores já abordamos os principais recursos incluídos, como, por exemplo, Namespaces, Static Late Binding, Lambda e Closures, Arquivos Phar. Espero que tenham aproveitado esta série.

Mysqlnd

Mysqlnd, ou Mysql Native Driver para PHP, é um modo alternativo para se conectar no MySQL, disponível a partir do PHP 5.3 ou mais recente. Trabalha com as versão 4.1 ou mais recente do MySQL. É uma opção para substituir a libmysql, MySQL Client Library, oferecendo diversas vantagens sobre ela. Não há planos para remover a libmysql para as extensões do mysql. Foi adicionado somente um novo driver mais eficiente e você pode escolher qual dos dois usar.

Vantagens

As vantagens do mysqlnd são várias. O driver é totalmente integrado com o Zend Engine. É mais rápido para executar. A perfomance de diversas funções foram melhoradas. Não há necessidade de linkar com bibliotecas externas. A compilação se tornou mais fácil e simples. Não há necessidade de se preocupar com a versão do MySQL. Mysqldn utiliza PHP License que resolve alguns problemas relacionados com a licensa do MySQL.

Instalando mysqlnd

O novo driver mysqlnd pode ser utilizado nas três extensões do mysql existentes para PHP: mysql, mysqli e PDO_MYSQL. Ele vem junto com o código fonte do php, obtido em http://www.php.net.
Caso esteja compilando a partir do código fonte, basta passar as seguintes opções para o configure: –with-mysql=mysqlnd, –with-mysqli=mysqlnd e –with-pdo-mysql=mysqlnd.

Na distribuição oficial do PHP para Windows, nas versões >= 5.3, o Mysql Native Driver (mysqlnd) já vem habilitado por padrão. Portanto não há necessidade de nenhuma configuração adicional para usá-lo.

Limitações

O novo driver mysqlnd não trabalha com as versões 4.0 ou mais antigas do MySQL. Também não tem suporte à SSL e à compactação. A opção de compactação já teve o seu desenvolvimento iniciado. O suporte à SSL virá mais para frente.

PHP 5.3 parte 5: Outras melhorias

Melhorias na Linha de comando.

No php 5.3 a interface de linha de comando, conhecida como CLI (Command Line Interface), teve algumas melhorias. A principal melhoria foi a função *getopt* que se tornou independente de plataforma. Ou seja, agora funciona também no Windows.

Ainda na função *getopt*, agora é possível especificar parâmetros no script php, pela linha de comando, usando o caractere de atribuição =. Veja o exemplo abaixo:

Arquivo teste_opt.php:

```
<?php
$options = getopt("i:");
var_dump ($options);
```

```
# php teste_opt.php -i="valor teste123"

array(1) {
  ["i"]=>
  string(14) "valor teste123"
}
```

Veja que chamamos o script php passando o parâmetro *-i* com um determinado valor, usando o caractere de atribuição = .Poderiamos ter usado da seguinte forma também: -i “valor teste123?.

Também é possível que o valor de um parâmetro passado pela linha de comando seja opcional. Para isso devemos utilizar a sequência *::* após o nome do parâmetro. Veja o exemplo abaixo:

Arquivo testes_opt.php:

```
<?php
$options = getopt("i::");
var_dump ($options);
```

Veja que passamos a sequencia *::* após o nome do parâmetro *i*

```
# php teste_opt.php -i

array(1) {
  ["i"]=>
  bool(false)
}
```

```
# php teste_opt.php -i "valor teste123"

array(1) {
  ["i"]=>
  string(14) "valor teste123"
}
```

Criamos os dois exemplos acima demonstrando que o valor para o parâmetro *i* pode ser opcional.

PHP 5.3 parte 5: Outras melhorias

Novos Error Levels

Na versão 5.3 do PHP finalmente E_STRICT faz parte de E_ALL. Agora E_ALL compõe todos errors levels. Nas versões anteriores E_STRICT era um error level separado de E_ALL.

Foram criados dois novos levels: E_DEPREACTED e E_USER_DEPRECATED. E_DEPRECATED alerta sobre funções e recursos que serão descontinuados em versões futuras do PHP, no caso a versão 6 do php. E_USER_DEPRECATED tem a intenção de indicar recursos que serão descontinuados relacionados ao código do usuário.

Algumas fuções que serão descontinuadas em versões futuras: *ereg*, *ereg_replace*, *split*, *session_register*, *session_unregister*, etc. Para uma lista completa acesse: http://www.php.net/manual/pt_BR/migration53.deprecated.php

O exemplo abaixo demonstra a utilzação de uma função que será descontinuada da versão 6 do php. Veja o retorno obtido após a execução do script.

c:\php5\error\teste.php:

```
<?php

echo ereg("[0-9]", "d0uglas");
```

Resultado:

```
Deprecated: Function ereg() is deprecated in C:\php5\error\teste.php on line 3
1
```

PHP 5.3 parte 5: Outras melhorias

__callstatic

__callstatic é um método mágico incluso na versão 5.3. Sua função é tratar chamadas para métodos estáticos inexistentes na classe. Quando você fizer uma chamada para um método estático em uma classe onde o método não existe o php irá procurar pela existência de um método mágico chamado *__callstatic*, passando como parâmetro o nome do método e os argumentos passados caso haja algum. Esse método é semelhante a outros médotos mágicos como *__call*, *__set*, *__get*.

Veja o exemplo abaixo onde demonstramos a utilização do método estático. Estamos fazendo a chamada para dois métodos estáticos. O primeiro, *printText()*, existe na classe *StaticClass* e portanto é executado normalmente. Já a chamada para o segundo método, *createUser()*, não existe na classe *StaticClass*, portanto será chamado o método *__callstatic* passando os devidos parâmetros.

```
<?php
```

```
class StaticClass {
   public static function printText() {
     var_dump("PrintText static function called");
   }

   static function __callstatic($methodname, $args) {
     var_dump($methodname);
     var_dump($args);
   }
}

StaticClass::printText();
StaticClass::createUser("Elizandra");
```

Segue a saída do script anterior:

```
string(32) "PrintText static function called"
string(10) "createUser"
array(1) {
  [0]=>
  string(9) "Elizandra"
}
```

PHP 5.3 parte 5: Outras melhorias

Chamadas variáveis estáticas

Agora é possível fazer chamadas estáticas onde o nome do método esta contido em uma variável. Já podiamos fazer isso com variável variáveis e chamada de método variável. Agora pode utilizar esse conceito em métodos estáticos.

Veja a utilização no exemplo abaixo: (Estamos aproveitando a classe utilizada como exemplo no tópico anterior).

```
<?php
$class_name = "StaticClass";
$static_method_name = "printText";
$class_name::$static_method_name();
```

Segue a saída do script anterior:

```
string(32) "PrintText static function called"
```

Veja como ficou a chamada para nosso método estático usando variável no nome: *$class_name::$static_method_name();*

PHP 5.3 parte 5: Outras melhorias

Novas funções

Duas novas funções inclusas foram *array_replace* e *array_replace_recursive*.
A função *array_replace* irá atualizar os dados do primeiro array de acordo com os dados dos arrays seguintes. Se uma chave do primeiro array existe no segundo, o valor da chave no primeiro array será atualizado. Caso exista a chave somente no segundo array e não no primeiro, será criado um novo elemento no primeiro array.

A função *array_replace_recursive* atualiza os elementos da primeira array procurando nos arrays de forma recursiva.

Novos recursos SPL

SPL significa Standard PHP Library. É uma coleção de interfaces e classes com o objetivo de resolver problemas padrões e implementar de uma forma eficiente o acesso à dados de classes e intefaces.

Segue abaixo as classes SPL inclusas no php 5.3:

* SplFixedArray
* SplStack
* SplQueue
* SplHeap
* SplMinHeap
* SplMaxHeap
* SplPriorityQueue

Segue abaixo links onde vocês poderão encontrar mais informações sobre as classes SPLs incluídas na versão 5.3 do php.

http://www.php.net/manual/en/class.splstack.php
http://www.slideshare.net/tobias382/new-spl-features-in-php-53