What's new in PHP 5.5 & 5.6

By Luis Atencio

ABOUT

PHP is a server-side scripting language designed for the development of web applications. With its concise syntax, rich function library, and powerful language constructs, PHP continues to be the platform of choice amongst web developers at large. It's no surprise that PHP powers around 240+ million sites. Each new release of PHP brings new innovations and enhancements. While the road to PHP6 remains elusive delusion, the 5.5 and 5.6 releases have added numerous enhancements to the core and new language features, but also introduced a few backward incompatibilities that are worth considering.

NEW IN 5.5

New features added in 5.5 include support for generators, the finally keyword, password hashing APIs, a new list construct, array and string literal dereferencing, and more. For a list of these features, visit http://php.net/manual/en/migration55.new-features.php.

NEW IN 5.6

PHP 5.6 adds support for constant scalar expressions, variadic functions, argument unpacking, and changes to the use keyword, among many others. For a list of these features, visit http://php.net/manual/en/migration56.new-features.php.

INCOMPATIBILITIES

Some backward incompatibilities were also introduced to the platform, which means developers should take caution when upgrading PHP to either version.

NEW IN 5.5

The 5.5 release comes bundled with a number of new features.

GENERATORS

The general foreach statement in PHP looks like the following:

```
foreach($collection as $item) {
   // process item
}
```

This works for arrays and iterators containing efficiently small data sets since all data is stored in memory. For larger data sets, we need another solution.

A generator is a special function used to augment the iteration behavior of a loop statement. In basic terms, generators are similar to functions that return an array or sequence of values; instead of returning all values at once, generators return values one at a time, thereby utilizing less memory and processing time. This is a very powerful language feature that has been present in other languages, such as Python, for quite some time. And now it's being added to PHP.

Generators are very closely linked to iterators. Iterators are a language construct available in PHP via the SPL library and the `Iterator` interface.

ITERATORS

Added in 5.0, the Iterator interface can be used to create custom external iterators. A class is considered iterable if it extends the following interface:

Embedding an instance of this class in a foreach() will treat it as if it were a collection, so long as the valid() and next() calls continue producing valid results.

The latest versions of PHP5+ take advantage of this and provide a variety of canned iterators to choose from via the SPL library. For more information on SPL iterators, please visit http://php.net/manual/en/spl.iterators.php.

For more information on iterators in general, visit http://php.net/manual/en/class.iterator.php.

GENERATORS

Generators provide yet another alternative when creating simple iterators without having to implement any interfaces.

Similar to iterators, generators allow developers to write code that leverages foreach()to traverse a set of data in both a memory and processing-efficient manner. Instead of traversing elements all at once, generator functions take advantage of the yield keyword to produce results as many times as needed to provide the value set being iterated over. The presence of yield turns any function into a generator. Behind the scenes, PHP will save the state of the generator when it yields a value so that it can be "woken up" next time a value is required.

When a generator function is called, an instance of the Generator class is returned. Behind the scenes, this class implements the Iterator interface previously discussed and adds 3 methods:

```
Generator implements Iterator {
   public mixed send ( mixed $value )
   public mixed throw ( Exception $exception )
   public void __wakeup ( void )
}
```

These methods allow a generator internally to save its state and resume where it left off next time it's called.

The PHP `range($start, $end, [, number $step = 1])` function is used to generate an array of values with every element in it from start to end parameters inclusively, separated by the step value.

```
foreach (range(0, 100000) as $number) {
    echo $number;
}
```

This statement will create an array of 100001 elements (including 0), loop over all of them and print each one. Understandably, the performance of the range function is degraded as the end value becomes bigger. Generators can be used to implement range() much more effectively. Let's call it `xrange($start, $end, [, number $step = 1])`.

```
function xrange($start, $limit, $step = 1) {
    if ($start < $limit) {
        for ($i = $start; $i <= $limit; $i += $step) {
            yield $i;
        }
    } else {
        for ($i = $start; $i >= $limit; $i += $step) {
            yield $i;
        }
    }
}
```

Generator functions return a Generator instance, which we can place inside of a foreach statement and iterate over a range of elements without ever needing additional memory.

```
foreach (xrange(0, 100000) as $number) {
    echo $number;
}
```

Let's look at simplifying another task done on a daily basis where generators can play a significant role, file I/O:

```
Let's look at simplifying another task done on a daily
basis where generators can play a significant role,
file I/O:
function readLinesFrom($filename) {
    $fh = fopen($filename, 'r');

    if (!$fh)
       throw new Exception('Error opening file '.
$filename);

    while (($line = fgets($fh)) !== false)
        yield $line;
    }
    fclose($fh);
}
```

This function can be used to read lines from a text file. The important statement in this code is the loop. As lines are read, the generator function yields each line one at a time, starting where it left off without creating any additional memory allocations. The calling code looks like the following:

```
foreach (readLinesFrom('myFile.txt') as $line) {
   // do something with $line
}
```

Another feature of generators is the ability to inject or send a value to the generator, which could be useful for stopping the generation process. Consider a simplified version of the xrange() function above:

```
function xrange($start, $limit, $step = 1) {
   if ($start < $limit) {
      for ($i = $start; $i <= $limit; $i += $step) {
         $res = (yield $i);
         if($res == 'stop') {
           return;//exit the function
         }
      }
   }
}

$gen = xrange(1, 10);

foreach($gen as $v)
{
    if($v == 5) {
       // stop generating more numbers
       $gen->send('stop');
    }
    echo "{$v}\n";
}
```

The result of the current yield expression can be captured and evaluated. By calling the send function in the client code, a value can be injected into the generator and halt the process.

More information on generators can be found at http://php.net/manual/en/language.generators.overview.php.

EXCEPTION HANDLING AND THE FINALLY KEYWORD

Exception handling had been added to PHP in version 5. Code can be surrounded in a try-catch block to facilitate the catching of potential errors. Classic uses of exception blocks are file I/O and network I/O functions. Each try is accompanied by at least one `catch` block and multiple catch blocks can be used to handle different types of exceptions.

PHP 5.5 adds support for the finally keyword. Similar to Java, the finally block is appended to try-catch blocks and it is guaranteed to run regardless of whether an exception occurs within the try block or before normal execution resumes. Here is the general syntax:

```
$handle= fopen('myFile.txt', 'w+');
try {
   // Open a directory and run some code
   fopen($handle, 'Some Text');
}
catch (Exception $e) {
   echo 'Caught exception: ',  $e->getMessage(), "\n";
}
finally {
   fclose($handle);
   echo "Close directory";
}
```

The finally block is used typically to free up resources that were taken up before or during the execution of the try statement. Adding support for finally is another step at enhancing PHP's error handling mechanism to compete with other programming languages, and also to improve the quality of error handling code. Unlike Java, however, PHP does not establish a distinction between checked and unchecked exceptions – essentially all exceptions are runtime exceptions.

SECURITY

PHP 5.5 has a new API that facilitates the generation of password hashes using the same underlying library as crypt(), which is a one-way password hashing function available in PHP since version 4. Password hashing is very useful when needing to store and validate passwords in a database for authentication purposes.

The new password_hash() function acts as a wrapper to crypt() making it very convenient to create and manage passwords in a strong and secure manner. Streamlining this API will make it easier for developers to start adopting it over the much older and weaker md5() and sha1() functions, which are still heavily used today.

Since it is built-in to the PHP core, this extension has no dependencies on external libraries, and does not require special php.ini directives to use.

The API consists of the following functions:

FUNCTION NAME	DESCRIPTION
password_get_info	Returns information about the given hash as an array: salt, cost and algorithm used
password_hash	Creates a password hash
password_needs_rehash	Checks if the given hash matches the options provided
password_verify	Verifies password matches a given hash

Here is some sample code:

```
// generate new hash
$password = 'MyP@ssword!';
$new_hash = password_hash($password, PASSWORD_DEFAULT);
print "Generated New Password Hash: " + $new_hash + "\n";

// get info about this hash
$info = password_get_info($new_hash);
var_dump($info);

// verify hash
$isVerified = password_verify($password, $new_hash);
if($isVerified) {
        print "Password is valid!\n";
}

// check for rehash
$needs_rehash = password_needs_rehash($new_hash,
PASSWORD_DEFAULT);
if(!$needs_rehash) {
        print "Password does not need rehash as it is
valid";
}
```

The new hashing API supports a few options when generating password hashes: salt, the algorithm, and a cost.

OPTION NAME	DESCRIPTION
salt	This is an optional string to base the hashing algorithm on. The new hashing API does not require or recommend the use of a custom salt.
algo	The default algorithm built in to PHP 5.5 is the Bcrypt algorithm (also called Blowfish), but you are free to use others supported by the crypt() function such as Blowfish or Standard DES.
cost	Associated with an encryption algorithm is a cost value that represents the number of iterations used when generating a hash. In the password hash API, a default baseline value of 10 (2^10 = 1024 iterations) is used, but you can increment this on better hardware. Cost values range from 04 – 31.

More information on password hashing, visit http://php.net/manual/en/book.password.php.

PHP 5.5 PLATFORM ENHANCEMENTS

The PHP 5.5 release also contains minor enhancements to the language constructs.

ENHANCEMENT TO *FOREACH( )*

Support for list( )

The foreach() statement now supports the unpacking of nested arrays (an array of arrays) to perform quick variable assignments via the list() function. list() had been present in PHP since version 4, but could not be used in this capacity until now.

Let's see an example:

```
$vector = [
    [0, 1, 2],
    [3, 4, 5],
    [2, 1, 2],
];

foreach ($vector as list($x, $y, $z)) {
    printf("Vector components [%d, %d, %d] \n", $x, $y,
$z);
}
```

At each iteration of the loop, $x, $y, and $z will be set to each corresponding array index. If there aren't enough values to fill the array, a PHP notice will be thrown. On the other hand, you could use fewer values to initialize variables:

```
$array = [
    [1, 2],
    [3, 4],
];

foreach ($array as list($x)) {
    // Note that there is no $y here.
    echo "$x\n";
}
```

Support for non-scalar keys

PHP 5.5 lifts the restriction on foreach() valid keys to allow arbitrary types, in particular arrays and objects. This change applies only to iterator keys (iterator::key()) present in the loop statement; non-scalar keys still cannot occur in native PHP arrays.

This change is very useful when using the SPL SplObjectStorage iterator, which provides a mapping of objects to data. This dual purpose can be useful when having the need to uniquely identify data with a complex user defined structure.

Suppose you want to uniquely identify cities by its coordinates. Using simple scalar keys will not be enough; for instance, we can use a custom Coordinate class, as such:

```
$coords = new SplObjectStorage();

$ny = new Coordinate(40.7127, 74.0059);
$mia = new Coordinate(25.7877, 80.2241);


$coords [$ny] = “New York”;
$coords [$mia] = “Miami”;

foreach($coords as $c -> $city) {
   echo “City: $city | Lat: $c->lat | Long: $c->long”;
}
```

Before 5.5, the code above would not have been allowed, as the declared key $c is a non-scalar.

Note: as of this writing, the PHP manual for the Iterator::key() function has not been updated to account for the non-scalar return type.

Find more information at http://php.net/manual/en/iterator.key.php.

ARBITRARY EXPRESSIONS AND *EMPTY()*

Sometimes you want to perform operations before AngularJS starts. These operations can be configurations, achieving relevant data or anything else you might think about. AngularJS enables you to manually bootstrap the application. You will need to remove the ng-app directive from the HTML and use the angular.bootstrap function instead. Make sure that you declare your modules before using them in the angular.bootstrap function. The following code shows a snippet of using manual bootstrap:

```
!isset($var) || $var == false
```

Let’s take a look at some examples:

```
function is_false() {
    return false;
}

if (empty(is_false())) {
    echo “Print this line”;
}

if (empty(true)) {
    echo “This will NOT be printed. “;
}
if (empty(0) && empty(“”) && empty(array())) {
    echo “This will be printed. “;
}
```

Overall, empty() is a safe way to check for null or empty conditions. For more information on the use of empty, visit http://php.net/manual/en/function.empty.php.

ARRAY AND STRING LITERAL DEREFERENCING

Similar to Perl and Python, PHP 5.5 has added language support for direct dereferencing via the index operator. This provides a quick way to obtain the character or index value of an array or string literal, staring at zero.
Examples:

```
echo ‘Direct Array dereferencing: ‘;
echo [1, 2, 3, 4, 5, 6, 7, 8, 9][2];  // will print 3

echo ‘String character dereferencing: ‘;
echo ‘DZONE’[0];                      // will print D
```

If you’re dereferencing an index value outside of the length of an array or a string, an Undefined offset or Uninitialized string offset error will be thrown, respectively.

CLASS NAME RESOLUTION VIA ::CLASS

PHP 5.5 has made the addition of the basic keyword class which, preceded by the scope resolution operator ::, is treated as a static or constant property of a class. This keyword will be used to obtain a string containing the fully qualified name of a class. This is especially useful with namespaced classes. For more information on namespaces, visit http://php.net/manual/en/language.namespaces.php.

For example:

```
namespace MyAppNs\Sub {

    class ClassName {

    }

    echo ClassName::class;

    // this will print MyAppNs\Sub\ClassName
}
```

For more information on class name resolution, visit http://php.net/manual/en/language.oop5.basic.php - language.oop5.basic.class.class.

CLASS NAME RESOLUTION VIA ::CLASS

PHP is an interpreted language; every time a script runs it gets compiled to bytecode form, which is then executed. Since scripts don’t change on every request, it makes sense to include a level of opcode caching (code compilation caching) that will improve the performance of script execution.

The Zend Optimiser + OPcache has been added to PHP as the new OPCache extension, replacing APC bytecode caching. Opcode caching is a platform optimization feature applied to the execution lifecycle of a PHP script. This extension drastically improves the performance of PHP by storing precompiled script bytecode into shared memory and removing the need to reload and parse entire scripts on each request. There have been cases where a performance factor of 3x has been seen by just enabling an OPcode cache.

This extension is bundled into PHP 5.5 but is also available as a PECL package, which can be used with previous versions of PHP. PECL is a repository for the distribution and sharing of PHP extensions.

http://pecl.php.net/package/ZendOpcache

To check whether you are using the Zend OPcode cache, execute the following function from a PHP terminal:

```
!isset($var) || $var == false
```

You should see the following included as part of the output:

```
‘version’ =>
         array(2) {
         ‘version’ => string(9) “7.0.4-dev”
         ‘opcache_product_name’ => string(12) “Zend
OPcache”
```

Recommended php.ini settings for effective OPcache functionality:

KEY	VALUE
opcache.memory_consumption	128
opcache.interned_strings_buffer	8
opcache.revalidate_freq	60
opcache.fast_shutdown	1
opcache.enable_cli	1
opcache.save_comments	enable or disable depending on application
opcache.max_accelerated_files	4000

For more information about the meaning of these directives please visit the OPcache configuration page:
http://php.net/manual/en/opcache.configuration.php

NEW FUNCTIONS ADDED

Many functions were added in PHP 5.5 — too many to be listed here. Specifically, the internationalization or localization module was completely revamped. To get the complete list of all functions added, please visit http://php.net/manual/en/migration55.new-functions.php.

In addition, here are some other functions added that are useful to keep at hand:

KEY	USAGE
array_column($array, $column_key)	When dealing with nested array elements, you can use this function to obtain the values of a specific column
boolval($var)	Obtain the boolean value of any type of object
json_last_error_msg()	Obtain the error string of the last json_encode() or json_decode() function call
mysqli::begin_transaction($flags)	Marks the beginning of a transaction
mysqli::release_savepoint($name)	Rolls back a transaction to the named
	savepoint
mysqli::savepoint()	Sets a named transaction savepoint

PHP 5.5 INCOMPATIBILITIES

PHP 5.5 introduces a few backward incompatible changes as well as some deprecated APIs worth noting. All of these issues should be checked thoroughly when planning to do a migration to 5.5

DEPRECATIONS

- Ext/mysql will generate an E_DEPRECATED warning when attempting to connect to a database. Use MySQLi or PDO_MySQL extensions instead. The biggest change here is that mysql_connect() has been deprecated.
- The /e escaping modifier in preg_replace() has been deprecated. This modifier was used to ensure that no syntax errors occur when using backreferences in strings with either single or double quotes. Also, you could pass arbitrary PHP code into the replacement parameter. Instead, use preg_replace_callback(), a faster, cleaner, and easier-to-use function.

For instance:

```
$line = 'Hello World';
echo preg_replace_callback( '/(\\w+)/ ',
        function ($matches) {
            return strtolower($matches[0]);
        },
        $line
    );  // will print hello world
```

- Encryption functions from the mcryt extension have been deprecated:

```
o mcrypt_cbc()
o mcrypt_cfb()
o mcrypt_ecb()
o mcrypt_ofb()
```

BACKWARD INCOMPATIBLE CHANGES

- Windows XP and 2003 support were dropped, which means the upgrade path and security fixes for these versions will stop. The Windows build for PHP will now require Windows Vista or newer.
- Up until now, PHP language constructs are case-sensitive. Meaning, you could write if-else and for loop statements in any case (including mixed case).
- All case-insensitive function matching for function, class, and constant names is now locale-independent and following ASCII rules. This improves support for languages with unusual collating rules, such as Turkish. This may cause issues with code bases that use case-insensitive matches for non-ASCII characters in UTF-8. This was done because some locales (like Turkish) have every unexpected rule for lowercasing and uppercasing. For example, lowercasing the letter "I" gives different results in English and Turkish. Therefore, system functions were moved to ASCII lowercasing.
- As a result of the previous point, `self`, `parent`, and `static` keywords are now case-insensitive. Prior to 5.5, these keywords were treated in a case-sensitive manner. This has been resolved; hence, `self::CONSTANT` and `SELF::CONSTANT` will now be treated identically.

BACKWARD INCOMPATIBLE CHANGES

In PHP 5.6, it is now possible to use scalar expressions involving numeric and string literals in static contexts, such as constant and property declarations as well as default function arguments. Before 5.6, you could only do this with static variables. Let's take a look:

```
const ONE = 1;
const TWO = ONE + ONE;

class MyClass {
    const THREE = TWO + 1;
    const ONE_THIRD = ONE / self::THREE;
    const SENTENCE = 'The value of THREE is '.self::THREE;

    public function f($a = ONE + self::THREE) {
        return $a;
    }
}

echo (new MyClass)->f();  // prints 4
echo MyClass::SENTENCE;   // prints The value of THREE is 3
```

In previous releases, this would have generated parser or syntax errors. PHP 5.6 augments the parsing order allowing PHP expressions as constant values, which can contain variables as well as other constants. Similar to static variable references, you can access a class's constant values via the `::` scope resolution operator.

VARIADIC FUNCTIONS

This is an enhancement to the variable-length argument list (or varargs) parameters. As of PHP 5.5, you need to use the combination of the functions `func_num_args()`, `func_get_arg()`, and `func_get_args()` in order work with variable-length arguments.

Dynamic languages such as Python and Ruby use the star operator "*" to tell the interpreter to pack as many parameters provided in a function invocation into an array. Starting with PHP 5.6, arguments may include the ellipsis token to establish a function as variadic, analogous to Java's ellipsis parameters. The arguments passed in will be treated as an array.

```
function sum(...$numbers) {
    $s = 0;
    foreach($numbers as $n) {
        $s += $n;
    }
    return $s;
}
echo sum(1, 2, 3, 4);  // prints 10
```

As with any other parameter, you can pass varargs by reference by prefixing the ellipsis operator with an ampersand (&).

ARGUMENT UNPACKING

In somewhat similar fashion to the use of list()introduced in PHP 5.5, you can use the ellipsis token to unpack a Traversable structure such as an array into the argument list.

```
function sum($a, $b, $c) {
    return $a + $b + $c;
}

echo sum(...[1, 2,2]); // prints 6
$arr = [1, 2, 3];
echo add(...$arr);     // prints 6
```

Furthermore, you can mix normal (positional) arguments with varargs, with the condition that the latter be placed at the end of the arguments list. In this case, trailing arguments that don't match the positional argument will be added to the vararg.

```
function logMsg($message, ...$args) {
    echo "${message} : on line ${args[0]} Class:
${args[1]}";
}
logMsg("Error", 32, "MyClass");
// prints Error: on line 32 Class MyClass
```

EXPONENTIATION

PHP 5.6 has added the operator **, a right-associative operator used for exponentiation, along with a **= for shorthand assignment.

```
printf("num= %d ", 2**3**2); // prints num= 512

$a = 2;
$a **= 3;
printf("a= %d", $a);         // prints a=8
```

NAMESPACING AND ALIASING

Introduced in PHP 5.3, namespaces provided a way to encapsulate or group a set of items, in very much the same way to how directories group files. Two files with the same name can live in different directories without conflict. The same goes for PHP. This is analogous to Java's package structure of directories.

Before namespaces existed, PHP developers would include the directory structure as part of their class names, yielding incredibly long class names. In other words, you would see class names like:

```
PHPUnit_Framework_TestCase
```

This was the best practice at the moment. This would refer to the class TestCase that lives inside the PHPUnit/Framework directory. Namespaces are designed to solve 2 problems:

1. Avoid name collisions among application code, PHP core code, and third party libraries
2. Improve readability by aliasing (shortening) long class names

The use keyword provides developers the ability to refer to an external, fully-qualified name with an alias. This is similar to Python's use of the import keyword. All versions of PHP that support namespaces and up to PHP 5.5 support aliasing, or importing a class name, interface name, or namespace. PHP 5.6 has extended this behavior to allow aliasing of functions as well as constant names.

Here is an example of the different uses of namespaces and the use keyword. The following statements build on each other:

```
namespace foo;

// Alias (shorten) full qualified class
use com\dzone\Classname as Another;

// Similar to using com\dzone\NSname as NSname
use com\dzone\NSname;

// Importing a global class
use ArrayObject;  // ArrayObject is a global PHP class

// Importing a function (PHP 5.6+)
use function com\dzone\functionName;

// Aliasing a function (PHP 5.6+)
use function com\dzone\functionName as func;

// Importing a constant (PHP 5.6+)
use const com\dzone\CONSTANT;

// Instantiates object of class foo\Another
$anotherObj = new namespace\Another;

// Similar to instantiating object of class com\dzone\
Classname
$anotherObj = new Another;

// Calls function com\dzone\NSname\subns\func
dzone\subns\func();

// Instantiates object of class core class ArrayObject
$a = new ArrayObject(array(1));

// Prints the value of com\dzone\CONSTANT
echo CONSTANT;
```

Here is an example of the different uses of namespaces and the use keyword. The following statements build on each other:

```
namespace com\dzone {
    const FOO = 42;
    function myFunc() { echo __FUNCTION__; }
}

namespace {
    use const com\dzone\FOO;
    use function com\dzone\ myFunc;

    echo FOO;   // prints 42
    myFunc ();  // prints com\dzone\myFunc
}
```

DEBUGGING

Until now, the most common way to debug PHP scripts was by using the Xdebug extension. Xdebug would instrument a running PHP script by setting up a special connection to the server. Using specific protocols, it could expose the internals of a running script to developers.

PHP 5.6 now includes a built-in interactive debugger called *phpdbg* implemented as a first-class SAPI (Server API) module. As a result, the debugger can exert complete control over the environment without any additional connections and overhead. This lightweight, powerful debugger will run without impacting the functionality or performance of your code. For more information about the debugger and to install it, visit the homepage:

http://phpdbg.com/

phpdbg has several features including:

- Step-through Debugging
- Flexible Breakpoints
- Easy Access to PHP with built-in eval()
- Easy Access to Currently Executing Code
- Userland API
- SAPI Agnostic - Easily Integrated
- PHP Configuration File Support
- JIT Super Globals
- Optional readline Support - Comfortable Terminal Operation
- Remote Debugging Support - Bundled Java GUI

STRING COMPARISON

PHP 5.6 added a function called hash_equals()to compare strings in constant time. This is not meant to be used for all string comparison, but rather when there is a specific need to protect or hide the amount of time taken to compare 2 strings. This is very useful to safeguard against timing attacks.

Timing attacks attempt to discover username and/or password lengths by doing a relative comparison of response time differences, as a result of processing a login form with an invalid username against a known valid one.

PHP 5.5 with its simplified password hashing APIs (as discussed before) already makes use of this function.

```
bool hash_equals ( string $known_string , string
$user_string )
```

OBJECT INFO

The magic method ___debugInfo has been added in order to control the printed output of an object as a result of calling var_dump(). This is very similar to a class overriding the toString() method in Java.

```
class MySquareClass {
    private $val;

    public function __construct($val) {
        $this->val = $val;
    }

    public function __debugInfo() {
        return [
            'valSquared' => $this->val ** 2,
        ];
    }
}

var_dump(new C(3));
```

This will print:

```
var_dump(new C(3));

This will print:

object(MyClass)#1 (1) {

  ["propSquared"]=>

  int(9)
}
```

ENCODING

In PHP 5.6 and onwards, "UTF-8" will be used as the default character encoding for `forhtmlentities()`, `html_entity_decode()` and `htmlspecialchars()` if the encoding parameter is omitted. The value of the `default_charset` php.ini will be used to set the default encoding for `iconv` functions as well as multi-byte `mbstring` functions.

PHP 5.6 INCOMPATIBILITIES

PHP 5.6 introduces a few backward incompatible changes as well as some deprecated APIs that are worth mentioning. With each new release of PHP, the core team has committed to reducing incompatibility changes going forward.

DEPRECATIONS

- Methods from an incompatible context are deprecated and will generate an E_DEPRECATED warning. This occurs when invoking a non-static function in a static context. For instance:

```
class A {
    function f() { echo get_class($this); }
}

class B {
    function f() { A::f(); }  // incompatible context
}
```

  Support for these calls will later be removed altogether.
- The mbstring configuration options in regards to character encoding have been deprecated in favor of default_charset.

BACKWARD INCOMPATIBLE CHANGES

- Array keys won't be overridden when defining an array as a property of a class via an array literal. Before 5.6, arrays declared as a class property with mixed keys could have array elements silently overridden when an explicit key had the same value as a sequential implicit key. For example:

```
class MyClass {
    const ONE = 1;
    public $array = [
         1 => 'foo',
        'bar,
        'joe',
    ];
}

var_dump((new C)->array);
```

In PHP 5.5, this will output:

```
array(3) {
  [1]=>
  string(3) "foo"
  [2]=>
  string(3) "bar"
  [3]=>
  string(4) "quux"
}
```

- The function json_decode will reject non-lowercase variations of the JSON literals for true, false, and null, following the JSON specification.
- In line with the new password hashing APIs introduced in PHP 5.5, the Mcrypt functions now require valid keys and initialization vectors (IV) to be provided.

CONCLUSION

In sum, PHP 5.5 and 5.6 have introduced many important language enhancements that improve the overall quality and stability of the platform. Among the most popular enhancements, we find the introduction of function generators and the yield keyword; the addition of the finally block to try-catch statements; a streamlined password hashing API; improved OPcaching; constant scalar expressions, variadic functions, a lightweight debugger, and many additional functions and changes to the core platform.

While these releases did not send tremors through the PHP community, some backward incompatible changes were introduced that can make migration to this platform a bit unnerving. Thus, a degree of caution and overall regression-testing measures must be taken when upgrading production systems to 5.5 or 5.6. The PHP team is strongly committed to making future releases much more backward compatible.

Before migration, please take a look at the following pages, for both 5.5 and 5.6, respectively:

http://php.net/manual/en/migration55.php

http://php.net/manual/en/migration56.php

ABOUT THE AUTHOR

Luis Atencio is a Staff Software Engineer for Citrix Systems in Ft. Lauderdale, FL. He has earned both B.S. and M.S. degrees in Computer Science. Luis works full time with both Java and PHP platforms. In addition, he writes a developer blog at http://www.luisatencio.net focusing on software engineering. When Luis is not coding or writing, he likes to practice soccer and Muay Thai kickboxing, and play guitar.

RECOMMENDED BOOK

If PHP is the duct tape of the web, then the line between tape and metal ducting has gotten pretty blurred in recent years. This brand-new book assumes that you know PHP basics but may not be sure how companies like Facebook have built massive and super-performant systems with 'duct tape' -- but do you want to learn how to do awesome stuff at HipHop-level sophistication. Includes detailed treatment of new features, up-to-date best practices, and tutorials on how to make the most of modern PHP.

BUY NOW

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0 $7.95