ULTIMATE GUIDE TO OBJECT ORIENTED PHP FOR WORDPRESS DEVELOPERS

LEVELING UP AS WORDPRESS DEVELOPER WITH OBJECT-ORIENTED PHP

BY JOSH POLLOCK

TORQUE

WP engine

WHAT'S INSIDE

INTRODUCTION

The first step in leveling up your skills as a developer is learning object-oriented programming for PHP or OOP.

There are a lot of reasons why WordPress is so popular, but one is by far the flexibility and ease of use of the two languages it is written in: PHP and JavaScript. Both languages have a fairly low barrier to entry. But, while it is easy to get started, that does not mean it is always easy to learn. More advanced skills are required to create performant, maintainable, reusable and testable code.

The first step in leveling up your skills as a developer is learning object-oriented programming for PHP or OOP.

OOP is about more than using classes in your code. It's about creating code that is less focused on a specific action and more focused on objects — small, reusable containers for data and functionality.

PHP is the most popular programming language in the world — powering 84 percent of all websites. The server-side scripting language is known for its ability to create dynamic websites and for its use as a general-purpose programming language.

PHP is open source, which combined with its ubiquity and capabilities, make it a perfect match for WordPress, the CMS that now powers more than 25 percent of the internet.

Although WordPress users don't need to learn PHP to manage their WordPress-powered websites, if you're a plugin or theme developer, or just want to modify the default behavior of your site, you will need to have a basic understanding of PHP.

There are two types of PHP: OO PHP vs. Procedural.

This ebook will equip you with the knowledge and skills you need to get started with object-oriented PHP as a WordPress developer, including:

- PHP Fundamentals
- Visibility and inheritance
- PHP 7
- WP_Query
- Magic methods

And more

PHP IN A REST API WORLD

In the era of the API-driven JavaScript interface, becoming the norm for WordPress development, PHP is more important than ever for WordPress users. This may sound strange, but all of these cool interfaces require a well built server-side application to power that application. That server-side code, written in PHP, will require a PHP developer that is well-versed in WordPress' inner workings.

Moving forward, WordPress core, plugins and maybe themes will need to do a better job of decoupling logic and CRUD from the display. The same low-level systems will need to serve both the traditional WordPress admin and theme interfaces, as well as REST API endpoints.

New plugins have an advantage in this department, but it requires discipline and more work to make a plugin that meets the requirements I listed above. Adapting an older plugin to meet these demands without breaking backward compatibility is more challenging.

If you're a WordPress developer excited about the future of WordPress driven by API-powered JavaScript interfaces, then you should learn more JavaScript. And yes choosing a JavaScript framework to learn — I recommend AngularJS — will help launch you into JavaScript development. Most importantly, you should challenge yourself to write better PHP and develop a strong respect for the separation of concerns and the single responsibility principle.

PHP 7
Guest Chapter
by Tom Ewer

In this chapter, we'll take a look at the development of PHP 7 to date.

WordPress has played a hugely significant role in helping PHP conquer the web over the last 12 years. While WordPress has whizzed through successive versions at an ever-increasing rate during that time, the language that still powers the majority of the platform has remained incredibly stable behind the scenes.

Big changes are finally in the offing with the arrival of PHP 7, however, and major WordPress hosts such as WP Engine are already kicking the tires of the latest release and getting ready to fully support it for their users.

In this chapter, we'll take a look at the development of PHP 7 to date, what the major changes are, what they mean for WordPress users, and consider whether you should be thinking of making the switch to the new version straight out of the gate.

Let's tee things up with a brief trip down memory lane.

HOW PHP TOOK OVER THE WEB

PHP's current omnipresence is almost taken for granted these days, but there was very little to suggest that it would go on to dominate the web when it was first cobbled together by Rasmus Lerdorf back in 1994.

In many ways, PHP's rise to the top has been a triumph of good, old-fashioned elbow grease over abstract programmatic concerns. In contrast to competing solutions such as Java and Perl, the language was straightforward enough to attract an audience new to the web, and simple enough on the server side to quickly become a standard install option at hosts worldwide. Put simply, PHP enabled a generation of coders to just get it done.

Its early adoption by a host of popular CMS offerings sealed the deal, with WordPress being by far the most significant of them. The PHP 5.x series sprang into life in 2004, and if you're running WordPress today, you're almost certainly running a minor version of this under the hood as we speak.

The 5.x series has served PHP well over time, but 12 years is a long time between major versions. Sooner or later, a change was bound to come.

PHP 7 FINALLY HEAVES INTO VIEW

Before we get into the nitty-gritty of PHP 7, let's get some potential naming confusion out of the way. The last stable release of PHP was PHP 5.6 in 2014, so at this stage, you might well be wondering what happened to PHP 6.

To cut a long story short, there was a previous attempt at a new major version using the name PHP 6 from 2005 to 2010 that never fully got off the ground, and to avoid muddying the waters, the decision was eventually made to go straight from the 5.x series to PHP 7.

PHP 7 has been under active development since 2014, and was officially released in December 2015. Its development arrived at an interesting time in the wider PHP world, as new initiatives such as Facebook's HipHop Virtual Machine were simultaneously expanding what was previously thought possible with the language.

SCALAR TYPE HINTS

2x FASTER

IMPROVED PERFORMANCE

SPACERSHIP OPERATOR

IT IS OUT NOW

PHP 7 ships with a host of developer-friendly features.

Add in the fact that it's been over a decade since the last major release, and there was understandably a lot of excitement and anticipation leading up to PHP 7 actually landing – and land it duly did!

Let's step through the main points that have been setting developers' pulses racing since then:

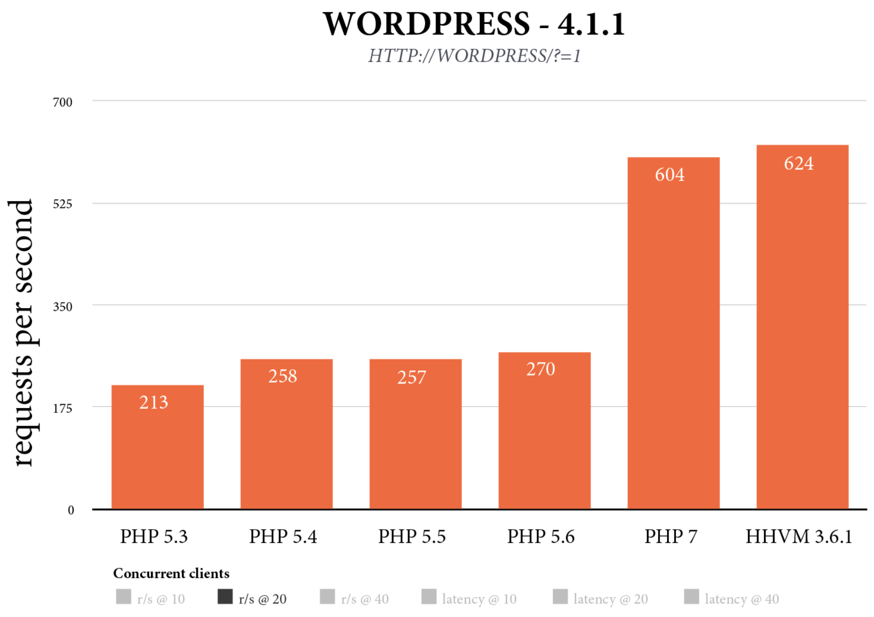
- *It's a true major release.* A major release is effectively a clean slate, and clears the decks for major (potentially breaking) changes. With the amount of cruft that PHP has gathered over the years, this is excellent news for all concerned.
- *There's a brand new engine under the hood.* The Zend Engine II has been doing sterling work on the PHP 5.x series over the years, but PHP 7 will be firing on all cylinders thanks to the spanking new PHPNG (Next Generation) engine that powers it.
- *A host of powerful new language features are available.* The latest version of PHP 7 ships with plenty of new options for developers to explore, including type declarations, space ship operators, and significantly improved error handling. Check out Treehouse's excellent run-down of the main items for an in-depth overview of the main significant points.

The main appeal of the new version (and the thing that will be of most interest to the average WordPress user) can be summed in one word – speed. Let's look at it in a bit more depth.

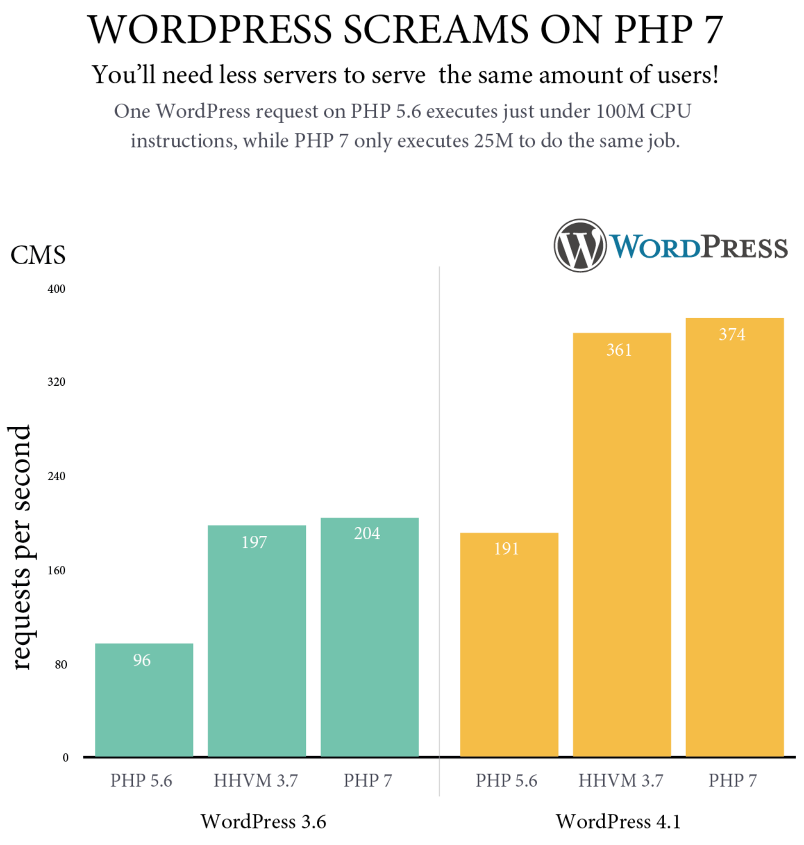
WORDPRESS IS ABOUT TO GET WINGS

Compared to its predecessors, PHP 7 is blazingly fast.

Compared to its predecessors, PHP 7 is blazingly fast. The common consensus seems to be that it's at least twice as fast across the board, and requires substantially fewer resources to actually execute code – two factors that are excellent news for WordPress users as shown in the early test results below:



Tests from Zend and WP Engine (among many others) have confirmed the significant improvements we can expect in both speed and performance, and the WordPress Core team has been beavering away at getting the platform ready for the new version since mid-2015.



WordPress performance is significantly improved.

THERE ARE TWO KEY POINTS TO EMPHASIZE HERE:

1. There are backward incompatible changes that plugin and theme developers will have to take into account going forward.
2. WordPress' own commitment to backward compatibility is as strong as ever – both PHP 7 and previous versions will continue to be supported.

From the average end user point of view, it's fair to say PHP 7 will be a slow burn in terms of when they really see the advantages.

Anecdotal evidence from around the web suggests there is still a lot of work to be done on popular themes and plugins before many are ready for the new hotness. WP Engine has officially rolled out support of PHP 7 to all its customers. According to their CTO, Jason Cohen, "We've made it not only easy to test site readiness for PHP 7 but incredibly easy for anyone on our platform to migrate to PHP 7."

"As of November 2016, only 3.5 percent of the WordPress community had upgraded to PHP 7," said Cohen.

Taking a broad view, we can expect end user switchover to PHP 7 to be slow and steady as the core team, hosting partners, and developers continue to work towards offering bulletproof solutions.

Speaking of which, let's examine whether developers themselves are ready to pull the trigger yet.

WHY DEVELOPERS ARE HOLDING FIRE ON SWITCHING (FOR NOW)

Developers are (rightly) a cautious bunch at the best of times.

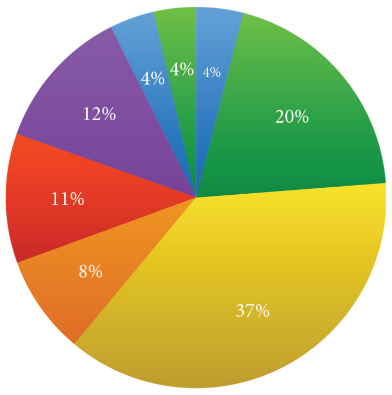
Developers are (rightly) a cautious bunch at the best of times, and it seems likely the majority will ease into PHP 7 slowly, rather than charging in all guns blazing. Organic factors – such as PHP 7 gradually becoming the default PHP package shipped with various Linux distributions – will help, but don't expect a stampede any time soon.

A developer survey by PHP Classes in the run-up to the official release offers a decent snapshot of sentiment across the community. Respondents were asked three straightforward questions:

3. Are you going to use PHP 7 in production?
4. Are you going to use PHP 7 in your development environment?
5. What is the latest PHP version you are using in production?

You can see the full results over on the survey page, but the range of answers broadly shows around half of the respondents actively considering using it in production in the short- to medium-term future, and roughly the same amount are either already or about to start using it in their development environments:

ARE YOU GOING TO USE PHP 7 IN PRODUCTION?



Yes, I am using already since versions before the first stable 7.0.0 release	21	4%
Yes, I want to start using only since the official 7.0.0 version is released	104	19.8%
Yes, I will wait a few weeks or months after the 7.0.0 version is released	196	37.3%
It depends on the customers that I work for	44	8.4%
No, not anytime soon, I need to migrate a lot of my code and that will take me a long time	58	11%
No, not now, I only plan to use it for future new projects	64	12.2%
No, only if my hosting company forces me to use it and does not provide an older version	20	3.8%
Other	19	3.6%

That's great news for both developers and users as the REST API.

When you consider the sheer range and variety of things that can go wrong in any development setup, this softly-softly approach makes a lot of sense. That said, a number of larger outfits with the resources to really dive in and explore have already successfully made the switch, with Badoo being the main one to spill the beans so far.

WHAT'S NEXT?

In contrast to previous PHP releases (both major and minor), this one looks to be proceeding smoothly straight out of the gate, and there's been a refreshing lack of drama associated with it so far.

The WordPress benchmarks that have been run against the latest releases show that genuinely transformative speed and performance increases are very much there to be had. That's great news for both developers and users as the REST API simultaneously starts to make its presence felt.

All that said, the road to full adoption will be a slow and cautious one, and there's plenty of work remaining for hosting companies and developers to make sure there are no nasty bumps along the way.

PHP Fundamentals

Like many WordPress developers, the first time I wrote PHP, I didn't know anything about it or software development in general. As I started to tackle more advanced concepts, I struggled because I had never learned the fundamentals of PHP that were assumed in the materials I encountered.

Regardless of your stage in your process of learning PHP as a WordPress developer, it's important to make sure you know the basics.

Regardless of your stage in your process of learning PHP as a WordPress developer it's important to make sure you know the basics. In this chapter, I will discuss PHP fundamentals: variables, constants, data types, functions, and scope.

This knowledge will equip you to learn PHP and other languages, as most of these concepts are fundamental to software development in general.

Like variables, constants are another item in which content can be stored.

VARIABLES AND CONSTANTS

The most simple "hello world" PHP program looks like this:

```
echo 'Hi Roy';
```

This prints the words, "Hi Roy." But as our program grows, we might want to print these words more than once, or use conditional logic to decide when or where to print them.

While in principle we could just cut and paste those lines of code into multiple locations, it directly violates the "Don't Repeat Yourself" (DRY) principle, one of the most sacred principles in software development. To avoid this, store the words "Hi Roy" into a variable before expanding the program, like this:

```
$hi = "Hi Roy";
echo $hi;
```

Now we can reuse this variable to change its value:

```
$hi = 'Hi Roy';
echo $hi;
$hi = 'Hi Shawn';
echo $hi;
```

In software development, a variable is a value that changes based on the conditions or information passed to the program. When you create a variable and put data in it, that data now exists somewhere on the server as a unique value in RAM.

Like variables, constants are another item in which content can be stored. As the name suggests, as opposed to variables — whose values can vary during a request — the content of constants can not.

For example, here is a constant we define in our wp-config.php:

```
define( 'WP_DEBUG', true );
```

If I tried to define WP_DEBUG anywhere else on my site, I would get an error because constants never change. That's why we tend to use them less frequently than variables and primarily in program configuration.

DATA TYPES

It's important to note that PHP is a "dynamically typed language".

In the previous section, I used two different types of data to set the value of constants and strings. I used strings for the variable and a boolean for the constant. Strings contain combinations of letters, numbers, and other characters. Booleans can be either true or false. These are two of PHP's eight different data types.

It's important to note that PHP is a "dynamically typed language," which means that a variable can change its type. This is not true in many other languages. An upside to this is that we have more flexibility, but we also have to be make sure the variable is the type we expect it to be before using it.

PHP is also very flexible. For example, PHP is not concerned about the difference between a string that holds a number and a variable of the integer — i.e., number — data type. For example, this will work totally fine:

```
$one = 1;
$three = '3';
$four = $one + $three;
```

We have four simple data types: strings, integers, booleans, and floats. Floats differ from integers in that they can have decimal values, where integers must be whole numbers. These simple data types only contain one value. There are, however, two "compound" data types — arrays and objects — which can contain more than one value and those values can be of any data type.

We create arrays using the function array() or bracket notation. The latter option was added in PHP 5.4. I prefer the bracket notation, but have to be mindful of the backward compatibility issues because most WordPress sites run an obsolete version of PHP.

Arrays are structured representations of multiple pieces of data, which can be written like this:

```
$post_titles = [
     '10 Things You Always Wanted To Know About Coffee',
     '8 Worthless Facts About Coffee',
     '5 Things That Would Matter Except You Need More Coffee'
];
```

Arrays can gain depth by nesting other arrays within them.

This is a simple, one-dimensional array, which means it has only one level of depth. Arrays can gain depth by nesting other arrays within them, like this:

```
$posts = [
     'drinks' => [
            'coffee' => [
                   '10 Things You Always Wanted To Know About
Coffee',
                   '8 Worthless Facts About Coffee',
                   '5 Things That Would Matter Except You Need
More Coffee'
            ],
            'tea' => [
                   'Tea and Other Alternative Caffeine
Delivery Systems?',
                   'Decaf Tea: As Opposed To Coffee, This Is A
Good Thing',
            ]
     ],
     'foods' => [
            '9 Tasty Vegan BBQ Solutions',
            '5 Sandwiches of Epic Victory'
     ]
];
```

We would call this a multidimensional array because of its nested structure.

Some arrays also have “keys.” We could access the food “key” like this:

```
$food = $post_titles[ 'food' ];
```

Nested keys must be accessed using that hierarchy. You can’t access the coffee key without accessing the drink key. For example, this would generate a warning:

```
$coffee = $post_titles[ 'coffee' ];
```

However, this would work:

```
$coffee = $post_titles[ 'drinks' ][ 'coffee' ];
```

Note that if we do not specify keys, then PHP indexes the array using numbers, starting with zero. So to get the second entry in the coffee array we would use the number one like this:

```
$title = $post_titles[ 'drinks' ][ 'coffee' ][1];
```

Arrays are mutable. That means that we can change their contents at any time. For example, we could add another item to the array like this:

```
$post_titles[] = '8 Fun Facts About Juice';
```

The entry in the array would get pushed onto the end and numerically indexed, which doesn’t make much sense with our structure. It would be better to define a key for juice and place it there:

```
$post_titles[ 'juice' ][] = '8 Fun Facts About Juice';
```

FUNCTIONS AND SCOPE

Variables defined outside of that function are not accessible.

As I mentioned before, part of the reason to have variables is to avoid writing repetitive code that violates the DRY principle. While variables are containers for information, functions are containers for functionality. If we need to do something, we almost always want to encapsulate that functionality in a function.

I'm going to discuss functions and scope together now. This is necessary as functions separate code from the rest of the program.

Doing so not only keeps things DRY, but also makes that functionality available throughout the program and lets us run it as many times as we need.

Functions in PHP are created using the "function" keyword, followed by a name for the function and parenthesis, which might contain function arguments. Function arguments are how we pass information into a function.

Inside of a function, there are only variables passed in as arguments. Variables defined outside of that function are not accessible. They are considered outside the "scope."

Let's consider this function, called slug_get_product:

```
function slug_get_product( $slug ){
     return get_post( [ 'post_type' => 'my-product', 'post_
name' => $slug ] );
}
```

This function is basically a wrapper for WordPress' function get_post(). It gets a post with a specific slug in the post type "my-product." Having one utility function for this saves having to write the same argument set for get_posts() multiple times. It also means that if you change the name of that post type, there is only one line of code to change.

Note that we passed in the variable $slug as an argument. That's why we could use $slug inside of the function. On the other hand, this wouldn't work:

```
$slug = 'red-shoes';
function slug_get_product( ){
     return get_post( [ 'post_type' => 'my-product', 'post_
name' => $slug ] );
}
```

Variables can be placed in global scope or accessed from global keyword.

The reason this doesn't work is that inside the function we are trying to use a variable called slug but in that scope no such variable exists. Even though one of that same name does exist, it is in a different scope and therefore can not be accessed.

In fact, we could write code like this with two variables named $slug, but because one is inside the function and one is not they are two totally different variables with totally different values:

```
$slug = 'red-shoes';
function slug_get_product( ){
     $slug = 'blue-shoes';
     return get_post( [ 'post_type' => 'my-product', 'post_
name' => $slug ] );
}
```

This may seem complicated as it looks like they are the same variable because they both have the same name. That's a coincidence. They are in different scopes and are therefore totally different. Despite having the same name, they represent a different configuration of transistors in your server's RAM.

By the way, notice how I added the prefix "slug" to the function names? That's because there can only ever be one function in a program with the same name. Adding a second function of the same name generates a fatal error.

This is why you should always prefix your functions with a unique prefix that is consistent throughout your plugin or theme. When you see articles like this use a prefix like "slug," which means you should change "slug" to your own prefix.

So far scope has been limited to inside a function and outside of it. We also have what is called global scope. Like its name suggests, global scope is available everywhere. Constants are in global scope and can be accessed anywhere.

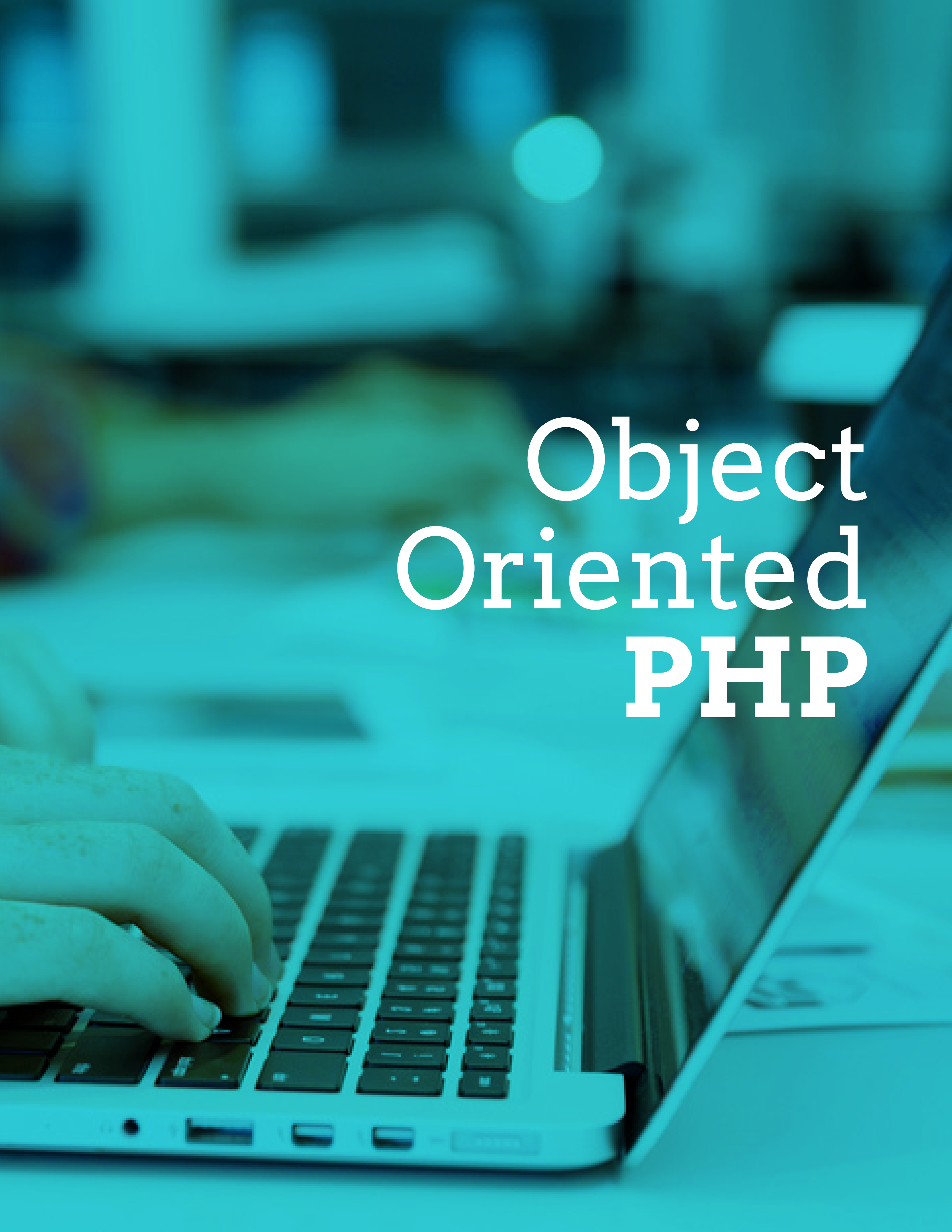
Variables can be placed in global scope or accessed from global keyword. Global scope should be avoided as much as possible. WordPress uses a lot of global variables. This is mainly because it was originally written in PHP 4 and there was no other way to solve the problems global variables solved.

WordPress' use of global variables, while not wonderful, is ultimately not that bad as it is the core application that plugins and themes work with. Plugins and themes, for the most part, should not add variables to the global scope.

Still, as a WordPress developer, you need to understand global scope. Inside of the WordPress loop, the current post is stored in a global variable called post. So if you had a function that ran inside of the loop, to access the current post, you would do this:

```
global $post;
```

In general it is better to use functions like get_post() or get_the_ID() that do that for you. The more you avoid accessing global variables or seeing them as ways to share data between functions, instead of passing that data in as an argument, the better.



Object Oriented **PHP**

Object-Oriented PHP helps you create more flexible code by allowing it to be only defined once but used in many places. In this chapter, you will learn the concepts behind Object-Oriented PHP in WordPress and walk through some practical examples.

BEFORE WE BEGIN

It's important that you try out the various examples provided. While you could do this in a theme template file on a test site, it's quicker and easier to use the debug console that's made available by an add-on to **Debug Bar**, a development plugin. It can be installed separately or as part of the **Developer** plugin bundle in the repo.

One of my favorites (and the one you should install) is* Debug Bar Console*, which gives you a place to quickly test PHP and MySQL in the context of your current site configuration. If you haven't already installed it, go ahead and do so now.

METHODS VS. FUNCTIONS

In OOP, the functions inside a class are called methods.

In functional programming, we're used to working with functions — which, once declared, are always available. In OOP, the functions inside a class are called methods, which are only accessible in the context of that class.

When using non-object oriented PHP in your themes or plugins, call a function directly and prefix the function names with a unique slug to avoid conflicts with other plugins or themes.

Keep in mind that you should still prefix class names to avoid conflicts with other themes or plugins. In OOP, methods, or functions inside classes, do not need to be prefixed since they are unique to that class. However, remember that you will need to access them through the class.

For this example, create two classes. Both will have one method, the class name, which will echo the name of the class. Here are the two classes: hat and shoe.

```
class hat {
    function class_name() {
        return "hat";
    }
}
```

These two methods have the same name, which is not an issue since they are in two different classes.

```
class shoe {
    function class_name() {
        return "shoe";
    }
}
```

As you can see, in each class there's a method called class name, which returns a string. These two methods have the same name, which is not an issue since they are in two different classes.

To use them, we must first instantiate each class and place them in a variable by setting a variable equal to a new instance of the class, like this:

```
$hat = new hat();
$shoe = new shoe();
```

To test this, echo the class_name method from each class:

```
echo $hat->class_name();
echo $shoe->class_name();
```

Play around in the console with these two classes, and try adding different methods to the classes. Keep in mind that methods always need to be accessed in the context of a class. To access a method of a class inside another method in that class, use the variable "$this," as-in "this class."

Here is a simple class that has two methods.

One gets a post object and tests that it is valid, and the other checks if that object is from a post in one of several post types.

```
class clothing_post_types {
     function get_post ( $id ) {
            $post = get_post( $id );
            //check if $post is an object of the stdClass
            if ( is_object( $post) && is_a( 'stdClass' ) ) {
                   return $post;
            }
     }

    function check_post_type( $id ) {
            //get post object
            $post = $this->get_post( $id );
            //check that $this->get_post didn't return false
(ie invalid post ID was used)
            if ( $post ) {
             //get post type
             $post_type = $post->post_type();
            if ( in_array( $post_type ), array( 'hat',
'shoe', 'shirt' ) ) {
                return true;
            }
        }
    }
}
```

PROPERTIES VS. VARIABLES

In a class, you can share a variable inside the class. We call this the class variable properties.

One of the biggest limitations of procedural programming (the opposite of OOP — not using classes) is how difficult it is to share variables between functions. This leads to using global variables, which is an overkill.

In a class, you can share a variable inside the class. We call this the class variable properties.

In the previous example, where we looked at the global post object, we were working with class properties. In the console, take a look at a post object, again, like this:

```
$post = get_post( 1 );
print_r( $post );
```

This will show you the post object for the "Hello World" post that introduces every new WordPress site to the web. You can access the properties for the post content, post type, and more like this:

```
$post->ID;
```

USING HOOKS IN CLASSES

Outside of a class, we follow this pattern to hook a function to a WordPress action or filter:

```
add_action( 'hook', 'callback' );
function callback() {
    //do something
}
```

When hooking a class method to a WordPress filter or array, you must provide the context for the callback function. Do this by using an array containing an instance of the class and the method name, usually inside of a magic method in your class.

OOP has a set of magic methods, and the magic method __construct() is run whenever your class is instantiated. This makes it a perfect place to use add_action and add_filter to hook class methods to WordPress hooks. Since it's in the class, you can use the variable $this as the class object.

Let's take a look at a typical use of this in a class:

```
class add_elements {
    function __construct() {
        add_action( 'wp_footer', array( $this, 'inline_
script' ) );
```

```
        add_filter( 'the_content', array( $this, 'end_of_
post_message' ) );
    }

    function inline_script() {
        echo "
        //add javascript here
        ";
    }

    function end_of_post_method( $content ) {
        $message = _('Text to output at end of every post');
        $content = $content.$message;
        return $content;
    }
}
```

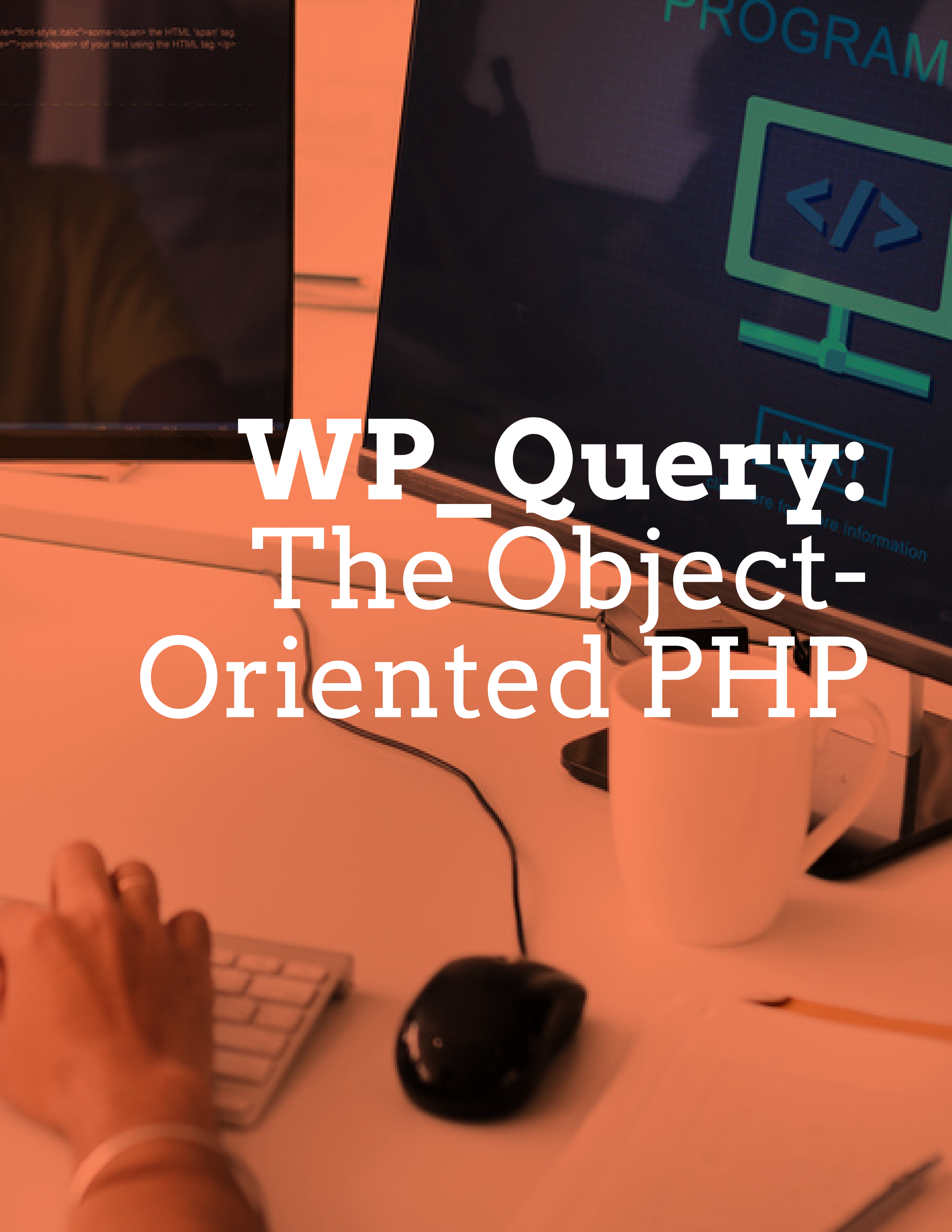
MORE FUN WITH __CONSTRUCT()

As I said in the previous section, __construct() runs when the class is instantiated.

As I said in the previous section, __construct() runs when the class is instantiated. This means that if you want a class to run its methods in a specific order and pass data between them based on the results of those methods, you can set that up inside __construct(). This is very useful if you're using a class to run a series of functions that use each other's output.

Of course, you may want to pass some data into the class when you instantiate it so that the object you build will be based on specific data. In fact, this is exactly what's going on when you create a new WP_Query object.

If you take a look at **the __construct() method in WP_Query**, you will see that the first thing the class does is check that that array of arguments isn't empty, and then it passes it to another class method. That starts a string of events that sets the properties of the class and allows you to use the class methods to access posts based on the arguments you set.



WP_Query: The Object-Oriented PHP

When I started out as a WordPress developer, I was working with object-oriented PHP before I even really knew what it was, thanks to WordPress post API class, WP_Query.

In PHP, objects and arrays are the only two data types that are actually considered compound. When I started out as a WordPress developer, I was working with object-oriented PHP before I even really knew what it was, thanks to WordPress post API class, WP_Query.

In PHP, objects and arrays are the only two data types that are actually considered compound. Objects are similar to arrays in that they provide structure for storing data, however, unlike arrays, objects are created with blueprints called classes, which can contain functions.

In this chapter, we'll look at WP_Query as a way of understanding how classes and objects work.

A LITTLE BACKGROUND

Classes are not actually objects but rather they are rules for creating an object.

Before we dive in, let's talk about syntax and terminology. It's important to make sure you have a good understanding of the terms "class," "object," and "instance." In any program, you can only have a class of a given name once. If I create a class in a WordPress plugin called "WP_Query" I would get a fatal error, since that class is already defined.

Classes are not actually objects but rather they are rules for creating an object. We can create as many objects of the WP_Query class as we want, however, each time we do, we are instantiating a new instance of the WP_Query class.

As I noted at the beginning of this chapter, arrays and objects are similar in that they can both contain other data types. With arrays, we use bracket notation to define and access index or keys of an array. With objects, we don't have keys or indexes, instead we have properties. Properties act just like variables, except they are inside of an object.
For example look at this class:

```
class say_hi {
     public $hello = 'Hi Roy';
}
```

In this class, we have a property called "hello," which looks and acts like a variable but can only be accessed in the context of the class. If we instantiate an object of this class by adding the keyword "new," then we can access the property like this:

```
$hi = new say_hi();
echo $hi->hello;
```

Classes are more than just containers for data stored in properties. They also encapsulate functionality. Inside of a class, a function is called a method. Methods are like functions, but they can only be used in the context of the object they are a part of. This allows us to make use of property or method visibility, as well as processing the data stored in the properties of the object.

Let's add a method to this class that will work with the property we defined:

```
class say_hi {
     public $hello = 'Hi Roy';
     public function hi_roy(){
          echo $this->hello;
     }
```

Inside of a class, the current instance is held in a special variable called $this. That's why we had to use $this to access the property. Just like functions outside of classes, methods define their own scope. That is why we can have a variable with the same name as a class property:

```
class say_hi {
     public $hello = 'Hi Roy';

     public function hi_roy(){
          return $this->hello;
     }

     public function hi_shawn(){
          $hello = 'Hi Shawn';
          return $hello;
     }
}
```

Methods are also accessed using the special property $this. One thing to keep in mind is that you should never have a method with the same name as a class. In PHP4 and PHP5 this acted as a static constructor.

WP_QUERY

Most of the standard "template tags" we use in WordPress are written this way.

Now that we have the right terminology in place, we can move on to WP_Query.

WP_Query is WordPress' post API. We use it to get collections of posts, which we store in the $posts property of WP_Query. This determines information about the post collection and loop through those collections.

If you've ever done any WordPress development, you've seen a standard WordPress posts loop.

If you look inside of all of those functions — seriously you should, reading the source is important for mastering WordPress — you will see they are all using an object stored in the global variable $wp_query. Most of the standard "template tags" we use in WordPress are written this way.

In short, the frontend of WordPress creates a WP_Query object based on the current URL and places it in the global variable $wp_query. As a result, we could rewrite that loop like this:

```
global $wp_query;
if ( $wp_query->have_posts() ) {
     while ( $wp_query->have_posts() ) {
          $wp_query->the_post();
          echo apply_filters( 'the_content',
$wp_query->post->post_content );
     }
}
```

This makes it clear that we are iterating through the posts — to be more specific, the objects of the WP_Post class is stored in the posts property of the WP_Query instance, which is stored in the global variable $wp_query.

This "main instance" of WP_Query doesn't have to be the only one. A recent posts or popular posts widget can exist on the same page and will use a different instance of the WP_Query class to get and display those collections of posts.

MAKING YOUR OWN

We can pass our array of arguments into the object when we instantiate.

I said earlier that instances of classes are created using the new keyword. I also said that class instance can contain different data in their properties. That means that even though they have the same methods, those methods will behave differently since they are acting on different data.

Property values get set in one of two ways: internally by the object or by modifying the property externally. Most of the properties of WP_Query are public, so we could technically change them from outside of the object, but that's not a great idea. Normally with WP_Query, we pass it an array of arguments that define what posts it should query for and that sets its properties internally. If you're interested, you can check out every parameter of WP_Query.

We can pass our array of arguments into the object when we instantiate. For example, to tell WP_Query to get 5 posts of the "my-product" post type, we could do this:

```
$args = [
     'post_type' => 'my-product',
     'posts_per_page' => 5
];
$query = new WP_Query( $args );
```

When we create an instance this way — passing arguments inside of two parenthesis — we are passing those arguments to the class' __construct() method. Constructors are a type of magic method that is called when the class instantiated. These methods can accept any number of arguments, WP_Query's constructor method takes one argument, but other classes take more.

WORDPRESS AS A GATEWAY TO OBJECT-ORIENTED

PROGRAMMING

WordPress may not be a totally object-oriented application, but we have classes for all of our content types that we use for finding and iterating through collections of posts, using WP_Query; users, via WP_User_query; taxonomy terms, using WP_Tax_Query, and comments thanks to WP_Comment_Query.

I encourage you to read through each of these classes to understand how they work, what properties they have, and what methods you can use from them. Understanding these basic query APIs will make your life as a WordPress developer easier. It will also help you learn how classes are designed.

Visibility In Object-Oriented PHP

The rules on how we access properties, variables, and methods.

OOP promotes the encapsulation of code — separated functions, classes, and methods — each with their own scopes and purposes. The rules on how we access properties, variables, and methods from different encapsulated scopes are determined by visibility.

In this chapter, you will learn how and why visibility works, which will enable you to write better code with clearer intent. You will also be able to take better advantage of PHP's ability to extend classes.

Keep in mind that this chapter refers to how these principles work in PHP 5.4 or later. PHP 5.3 and below are missing key features for doing OOP properly and are dramatically slower than 5.4.

ENCAPSULATION AND SCOPE

In software design, think of encapsulation as the principle by which code and data are bundled together in a way that restricts their access to the rest of the programming. The simplest example of encapsulation is a function. Consider this PHP code:

```
$post = get_post( 1 );

   function slug_get_post_five(){
      $post = get_post( 5 );
      var_dump( $post );
   }

   var_dump( $post );
   slug_get_post_five( 5 );
```

In this example, which isn't using any Object-Oriented PHP, we have two different variables called "post" that are being printed with "var_dump();". They will print two totally different things because the second "$post" is encapsulated inside of a function, while the first is not.

In this case, the second $post was encapsulated in the function "slug_get_post_five()" and place in a different scope than the $post above it, which makes it a completely different variable. They look similar because they share the same name, but they have no relationship. In fact, they are stored as two separate entities in the server's RAM when being executed.

In non-OOP PHP, we have no good way of declaring a controlling access outside of a function to a variable declared in the function.

We can return one value from a function and we can also use global scope, which is messy for a lot of reasons beside the lack of control over how variables are accessed from outside the function.

OOP, on the other hand, gives us the ability to control access to variables of a class, which we refer to as properties. This is one of the major advantages of OOP.

CLASSES VS. OBJECTS

Objects are created by instantiating a class, which determines the function of the object.

Before we dive into visibility, let's clearly define the differences between classes and objects. Classes define a set of rules for objects they create. Objects are created by instantiating a class, which determines the function of the object.

We see this all the time in WordPress with the WP_Query class. In the global variable, "$wp_query" is an object of the WP_Query class that WordPress creates based on the current HTTP request. But we can create as many additional WP_Query objects as we need during a session.

Each WP_Query object has the same methods and properties, but the values of those properties, or the results of those methods, may be different.

For example, let's say that during a session created by requesting the URL for a category term archive, we create a new WP_Query object to list posts of the custom post type "product" and store it in a variable called $products. In this case, "$products" and the global "$wp_query" are both objects of the WP_Query class. Both have a property called "$posts" that holds the queried posts, but both hold completely different posts. In fact, each post is represented by an object of the WP_Post class — same class but with totally different objects.

Classes can also be extended by adding a second class, called a subclass, which adds to and modifies the rules of the parent class. Keep in mind that a subclass can override a property or method.

THE THREE LEVELS OF VISIBILITY

In OOP PHP, we have three visibility levels for properties and methods of a class: public, protected, and private, all of which can be declared using a keyword.

The three levels define whether a property or method can be accessed outside of the class, and in classes that extend the class.

PUBLIC

The first level is "public." This level has no restrictions, which means it can be called in any scope. This means that a public property of an object can be both retrieved and modified from anywhere in a program — in the class, a subclass, or from outside of the class, for example.

This level is the default behavior when visibility is not declared because of backward compatibility concerns with PHP 4, which did not have visibility.

Technically a method declaration does not need to be proceeded by a visibility keyword, which makes it public. Also, a property can be defined using the "var" keyword to make it public. But for future compatibility reasons, and so your code is explicit in its intent, you should always use a visibility keyword and not use the var keyword.

PROTECTED

The second level is "protected." Protected properties and methods can be accessed from inside the class they are declared, or in any class that extends them. They can't be accessed from outside the class or subclass.

PRIVATE

While protected properties and methods are accessible anywhere in the object, the third level "private" is more restrictive.

A private property or method can't be accessed by a subclass of the class it is defined in.

If you have a class with a protected property and a private property and then extend that class in the subclass, you can access the protected property, but not the private property.

RULES OF PROPERTY VISIBILITY

The following code shows three classes. The second and third classes extend the first, and the third will also create an error as it violates the rules of visibility within a class:

```
class force {
     /**
      * This protected property can be accessed in a subclass
      */
     protected $jedi;

     /**
      * This private property can not be accessed in a
subclass
      */
     private $sith;

     public function set_force_users( $jedi, $sith ){
            //Both of these are legal because we are in the
same class, private vs protected is not a very meaningful
distinction
            $this->jedi = $jedi;
            $this->sith = $sith;
     }
}

class jedi extends force {
     /**
      * This is a good example.
      */
     public function set_jedi( $jedi ){
            //totally legal because jedi is protected and 
therefore accessible in this subclass
            $this->jedi = $jedi;
     }
}

class sith extends force {
     /**
      * Example of something bad, you know like the Sith, 
don't do this.
      */
     public function set_sith( $sith ){
            //Not allowed, we can't set a private property of 
the parent class in a sub class
            $this->sith = $sith;
     }
}
```

Let's walk through this. Our base class declares two properties: the first property, "$jedi," is protected, while the second property, "$sith," is private. The class uses one method to set both properties. In this case, the distinction between private and protected is meaningless because we couldn't set their values from outside of the class, but inside it's fine.

The second class extends the first class and adds a new way to set the protected property, "jedi." Protected properties are accessible in subclasses, and available throughout the object. The parent class and subclass are part of that object.

The third class, on the other hand, violates the rules of visibility. The private property $sith can't be accessed in a subclass. It can only be accessed in the class that it is defined in. You can make this legal by defining a private property called sith in the subclass, however, that is a bad workaround.

Keep in mind that if we were to instantiate the "jedi" class, and put it in the variable "$luke," we could use the "set_jedi()" method to set the protected property "$jedi," but we could not set it directly.

```
//allowed since the set_jedi method is public
$luke = new jedi;
$luke->set_jedi( 'Luke Skywalker' );

//not allowed since the jedi property is protected
$kylo = new jedi();
$kylo->jedi = 'Kylo Ren';
```

In the first two lines, the public method "set_jedi()" is used properly to set the protected property of the class. This is legal because the setting happens inside of the class.

The second two lines in this example would create a PHP error because we are trying to set a protected property from outside of the class. That is forbidden by the PHP interpreter. The same thing would happen if we tried to echo the property.

These kinds of changes are allowed but are not considered good practice.

It is also important to know that in a subclass a property can be made more visible, but not less visible. These kinds of changes are allowed but are not considered good practice.
A subclass can override a protected property of the parent class and make it public, but not make it private.

```
class ship{
     protected $model;
}

/**
 * Making model MORE visible is allowed but is confusing and
convoluted.
 */
class crusier extends ship{
     public $model;
}

/**
 * Making model MORE visible not allowed.
 */
class star_destroyer extends ship {
     private $model;
}
```

RULES OF METHOD VISIBILITY

A protected method can't be called outside of a class, but can be called in a subclass.

I have focused on properties, rather than methods, so far because almost everything I have said about properties also applies to methods, with the exception of a few additional rules. The basic principles still apply:

- A public method of a class can be called outside of the class or in a subclass.
- A protected method can't be called outside of a class, but can be called in a subclass.
- A private method of a class can only be called inside of the class it is declared in.

```
class force_user{
     private $is_force_user;

     protected $name;

     public function __construct( $name, $is_force_user ){
          $this->name = $name;
          $this->set_is_force_user( $is_force_user );
     }
     private function set_is_force_user( $is_force_user ){
          $this->is_force_user = boolval( $is_force_user );
     }
```

```
	protected function is_force_user(){
		return $this->is_force_user;
	}
}

class jedi extends force_user{

	public function has_the_force(){
		//Legal because is_force_user() is protected and therefore accessible in subclass
		return $this->is_force_user();
	}
}

class sith extends force_user  {

	public function remove_force(){
		//not legal, method is private, not accessible in subclass
		$this->set_is_force_user( false );

		//also not legal beacuse property is private
		$this->is_force_user = false;
	}
}
```

In this example, the base class uses a private function to set a private property. Since it is exposed by a protected function, it can be accessed by subclasses. This is what happens in the "jedi" subclass, however since the property and the setter method are private, there is no way to change it in a subclass. The "sith" subclass attempts to use a private property and a private method declared in the parent class, which is not legal.

Just like with properties, a method can become more visible in a subclass. We could have solved one of the two problems in the "sith" class by overriding the private method with a protected method. To make that method useful, we would have had to make the property it sets protected as well. That would have worked, but it would call into question why we were using a subclass instead of just writing a whole new class.

Just like with properties, a method can become more visible in a subclass.

Here is a rewritten "sith" class that implements these changes to prevent errors:

```
class sith extends force_user  {
	//now protected so we can access it
	protected $is_force_user;

	public function remove_force(){
		//not legal, method is private, not accessible in subclass
		$this->set_is_force_user( false );
```

```
                //also not legal beacause property is private
                $this->is_force_user = false;
        }

        //changed to protected, which required copying its
functionality manually
        protected function set_is_force_user( $is_force_user ){
                $this->is_force_user = $is_force_user;
        }
}
```

As you can see, there is really no point in extending the base class here. Now, if we change how the method set_is_force_user() works in the parent class, we will have to change it manually in the subclass because we have lost the utility of extending a class. These types of workarounds are sometimes necessary when working with other people's code. This sort of thing is considered to have a bad "code smell" because it isn't technically wrong, but that doesn't mean it is a code idea.

With properties, there is no way to prevent changing visibility or to prevent them from being overridden. With methods there is — we can use the "final" keyword in a method declaration. Once we do, it is illegal to override them which precludes changing visibility.

The final keyword should be used lightly. While it can be used to prevent visibility changes that you might now want, it is not a great idea to use it solely for that reason. You should give other developers flexibility and freedom.

I was recently working on a project where I need to extend a class of a third-party library just so I could change one method. In that method, I needed to access a private property so I changed its visibility to protected in my subclass. Without being able to do that, I would have had to write a much more convoluted workaround, probably by overriding the constructor and making a copy of that private property there.

WHY VISIBILITY MATTERS

So far I've discussed the rules of visibility, but I haven't really addressed why we care about it and should use it with intent.

There are a few reasons to use visibility in OOP PHP code:

- It helps us show the intent of our code
- It reduces our need to validate properties when used internally
- And it helps dictate how classes should be used.

For example, consider this class:

```
class lightsaber{
     public $name;

     protected $color;

     public function set_name( $name ){
            if( is_string( $name ) ){
                   $this->name = $name;
            }
     }
}
```

In this case, we can set the property name to anything. This is a problem if we want to write code later that uses "$name" and assumes it is a string. We would have to make sure it was a string every time we used that property.

Conversely, the property color is protected. It has a public set function that validates if it is a string before setting. Now we can assume when calling that property internally that it is a string. Assuming the type of variable is a dangerous assumption in a dynamically typed language, but this is one way to reduce that risk.

Also, by having public functions for getting protected or private properties, we make it clear what the utility of a class was. For example, if I have a class that has a constructor that takes in a bunch of data, creates some markup and puts it in a protected variable called $html, and then has a public method to get that variable, it is pretty clear the point of the class was to make HTML markup.

A FEW LAST WORDS

The other thing to keep in mind is not to overuse visibility.

I want to make two last points about Object-Oriented PHP. The first is that the complexity of these rules is one of the many reasons why you should be using an IDE with a PHP interpreter included. I use PHPstorm and when I was writing the example code for the "bad examples," PHPStorm added red underlines to my illegal code and gave me a little pop-up explaining why I was wrong when I moused over the bad code. It also did not auto-complete the properties and methods that I couldn't use.

The other thing to keep in mind is not to overuse visibility. As I said in the last section, visibility helps us clarify the intent of our code, but overusing can make life more difficult for you or another developer working with your code.

Class Inheritance In Object-Oriented PHP

In Object-Oriented PHP, inheritance is the ability to create a class that extends another class and takes on some or all of its features. This is one of the most useful functions of object-oriented programming in PHP. This chapter deep dives into class inheritance. You'll learn what it is, why it's important, and how to use it.

CLASS INHERITANCE

When a plugin or library gives you a class that is almost perfect.

The first time I tried to write a really big WordPress plugin, I had four main classes that dealt with four different post types. These four classes had some differences but had several similarities created by writing it in one class and then copying and pasting it in the other two.

Although it seemed to be working at the time, I later realized that I made some mistakes that were literally copied and pasted into four different places. To address the issue, I had to make the change four times. Similarly, if any future change was necessary it would require a change in four places. This is quantifiably bad practice.

Luckily for me, I had just read Carl Alexander's articles on inheritance and on abstract classes, which showed how to do this right. This article provided a basic introduction to class inheritance in Object-Oriented PHP, which equipped me to write better, more reusable code and be able to customize other people's code more efficiently.

When a plugin or library gives you a class that is almost perfect, instead of copying it and changing one method, you can extend that class and override the method. When you have two classes that are almost identical, you can write a base class and then extend it twice.

EXTENDING CLASSES AND OVERRIDING

A class extends another by using the "extends" keyword in its declaration. If you want to extend WP_Query, start your class with "product_query extends WP_Query." Any class can be extended unless it is declared with the final keyword.

When a class is extended, you can consider it to be the parent class and the class that is extending to be the subclass. Using class inheritance properly requires a strong understanding of property and method visibility, which I discussed in Chapter 6.

A subclass starts the same as a parent class. For example, if you use the following query for our product_query class, it would be the same as WP_Query:

```
product_query extends WP_Query{ }
```

Of course, that doesn't accomplish anything. On the other hand, if you were working on a project that needed a lot of queries for posts in the product post type, you could simplify things with this class:

```
class product_query extends WP_Query {
      public function __construct( $query ){
            $query[ 'post_type' ] = 'product';
            parent::__construct( $query );
      }
}
```

This class follows all of the same rules of WP_Query, but doesn't require that you keep telling it that you just want product posts. Also, it is clear that the objects created with it are for product posts.

Note that in my extended class, I made a function called __construct() and that WP_Query also declares a constructor. Any method — magic or not — can be overridden in a subclass, but you can still access the method of the parent class using the "parent" keyword.

Keep in mind that in PHP 5 overriding a method and changing its signature — changing its parameters or their types — will trigger a strict standards notice and in PHP 7 will trigger a warning. You should not do this because it makes the code more difficult to read.

If you are using return type declarations in PHP 7, don't change the return type of a method when overriding it in a subclass or a fatal error will occur. PHP 5 does not support return type declaration.

A good example of class inheritance is WP_HTTP_Response, which is in the wp-includes/rest-api directory, but could be used for any type of HTTP response. It is extended by the WP_REST_Response class, which is specifically used when responding to REST API requests.

ABSTRACT CLASSES

Unless a class is declared as final it can still be extended. PHP provides an abstract convention for classes that can not be instantiated directly and acts only to provide base classes for other classes that extend them.

The other special rule of an abstract class is that they can have abstract methods, which must be overridden by the class that extends them or an error will occur. This is a really useful system because it lets you define how the subclasses will function.

Abstract methods cannot have a body, and the methods that override them must not alter their signature. If you try to change the parameters or types of parameters in PHP 5 or in PHP 7, a fatal error will occur.

This is slightly different than overriding non-abstract methods, though in practice it's the same since you should never change the signature of a method even if you technically can in PHP 5 with strict standards disabled.

Methods overriding abstract methods must also use the same signature.

Interfaces provide a similar role, but the methods of an interface must be public whereas abstract methods can be public, protected, or private. Methods overriding abstract methods must also use the same signature.

The WordPress REST API's WP_REST_Controller is a great example of an abstract class. It is extended by all of the endpoints to provide a standard system of how endpoint classes should work.

Note that it does not use abstract methods because abstract methods are not supported in older versions of PHP. Instead, the base class has methods that call __doing_it_wrong(). This is a workaround to force those methods to be extended.

For another example, let's consider a plugin that connects to social networks via oAuth and then stores public and secret keys for later use. Since the actual fetching of keys will be different, you need different code for that, but that code can be reused to store and get the keys. So you could write an abstract method called "connect" that handles everything but getting the keys, which would happen in an abstract method.

```
abstract class social {
    protected $public;

    protected $secret;

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

    abstract public function connect();

    public function get_public(){
        return $this->public;
    }

    public function get_secret(){
        return $this->secret;
    }
}
```

This reduces code redundancy and enforces a particular pattern in the code, where all of the classes that extend this class have the same method for getting the keys, and could safely call that method of every subclass.

For example, you could make a class called "twitter," and one called "facebook" that extended these classes. Both would just need a method called connect.

With the connect method in place, you can assume its existence and use two classes or more to do the same thing, like this:

```
class facebook extends social{
    public function connect(){
            //do some things to get public secret keys and
set in the right properties
    }
}

class twitter extends social{
    public function connect(){
    //do some things to get public secret keys and set in
the right properties
    }
}

foreach( ['twitter', 'facebook' ] as $social_network ){
    /** @var social $obj */
    $obj = new $social_network;
    $obj->connect();
    update_option( 'public_' . $social_network, $obj->get_
public() );
    update_option( 'secret_' . $social_network, $obj->get_
secret() );
}
```

LESS CODE, BETTER CODE

This brief introduction to class inheritance in PHP should help you write smaller classes with more reusable code. Reusable code isn't just more efficient, it is easier to read, easier to maintain, and requires fewer unit tests.

There are several uses for class inheritance, which you should really try to explore. Instead of reusing WP_Query over and over again, you could write a base class that calls WP_Query and then extends it a few times to fit your needs.

If you are building a plugin that needs a custom REST API endpoint, for example, you can extend the WP_REST_Controller class, which will save you a lot of work.

If you are working on a site that interacts with multiple transactional email services, write a base class that handles the common functionality that all interactions with the APIs of those transactional email services need and extend it once per service.

I could go on with examples, but you should look at your existing code and at plans for projects and think about how class inheritance could help you skip the copy/paste method and improve your code.

How To Use Asynchronous PHP In WordPress

PHP executes code in series, which means one thing is done after another. This can be problematic when you need to do a lot of processing in one session or if you are relying on calls to external APIs. In this chapter, you will learn how to use asynchronous PHP in WordPress to address this issue.

It isn't uncommon for a server to be configured with a 30-second timeout limit for each request, which puts a hard cap on the amount of time each session has to do its work. Of course, if that processing is required in order to complete a front-end request, the end user is unlikely to stick around for 30 seconds.

I recently had a situation where the requirements I had to address could not be handled in one request. For this, my client used a lead form to trigger three external API requests, one of which was to a service that was slow and unreliable.

In addition, they wanted to get the site visitor to the thank you page as quickly as possible. Because I knew I had to wait for the API request, there was going to either be a thank you page that takes 20 seconds to load or a timeout.

Since the unreliability of the remote API made it necessary to save the data in WordPress so it could be re-sent if needed and to log the status of the API requests, the amount of work needed couldn't be done within the 30-second session. There was also no way to save the data, make the requests, update the status of those requests, and then redirect the user to the "thank you" page before they became frustrated by the slow page-load time.

The answer was to save the data, schedule tasks with an asynchronous task manager, and then redirect the user. Then, each of the three API requests would run in their own individual session, have their own 30 seconds to complete, and not impact user experience. I probably could have moved saving the data into an asynchronous task as well — but it wasn't necessary.

There are several asynchronous task managers specifically designed for WordPress, but one of the best and easiest to use is wp-async-task, a task manager open sourced by TechCrunch and developed by 10up.

HOW IT WORKS

The task manager makes a new POST request.

To use the wp-async-task, you need to use an action hook. This could be with a hook in core, another plugin, or via do_action(). Normally, you would hook directly to that hook in order to do some process, in series, when that hook happens.

With wp-async, the processing on that hook is deferred to a later session. It's actually fairly simple to do. You need the hook, a class with a property, and two methods to instantiate that class. And then you need to hook into a hook generated by that hook. The task manager makes a new POST request to WordPress and then passes data from the original hook to your new hook in the second session — which is not something you can normally do.

THE PROBLEM: ALL AT ONCE OR NOTHING

In the next section, I will walk you through the process. But first, let's look at the kind of code you would modify to work with this:

```
add_action( 'save_post', 'josh_send_to_api' );
function josh_send_to_api( $id ) {
    $thing = get_post_meta( $id, 'something', true );
    $r = wp_safe_remote_post( add_query_arg( 'id', $thing,
'http://apiexample.com/' ) );
    if ( ! is_wp_error( $r ) ) {
        $body = json_decode( wp_remote_retirve_body($r) ) );
        if ( isset( $body->key ) ) {
            update_post_meta( $id, 'api_response',
$body->key );
        } else {
            update_post_meta( $id, 'api_response', 'none' );
        }
    }
}
```

The problem with this code is that the process of saving a post will not complete until the request to the remote API is complete. If the request to the remote API takes longer than the server's timeout, then it will not complete.

Saving the post, sending data to the remote API, and recording the result as post meta does not have to be one discrete process. Instead, you can let the post save as normal, and then, in a second PHP session, retrieve that data and record the response as post meta.

SETTING IT UP

Using a wp-async-task is pretty straight forward, as all of the heavy lifting is handled by the library itself. Implementing the library requires a class that extends the class WP_Async_Task and does three simple things.

This class should have a protected property action, which is the name of the action this asynchronous task uses. In this case, it will be “save_post.” I should note that, in my experience, I’ve found that hooking multiple tasks to one hook is unreliable. Instead, I’ve created three hooks that run one after another and use one for each of my three API requests.

To keep things simple let’s stick to “save_post.” Let’s start the class with that:

```
class Josh_Task extends WP_Async_Task {
        /**
         * Action to use to trigger this task
         *
         * @var string
         */
        protected $action = 'save_post';
}
```

This method will be passed an array of data containing all of the parameters of the hook.

The next thing you need is a protected method called “prepare_data,” which you will use to prepare the data. The important thing to know about this method is that it runs during the session that triggers the asynchronous task, not the one that processes it. That is important to keep in mind as all of the globals and superglobals of the current session are available for you as data that will be passed to the next session.

This method will be passed an array of data containing all of the parameters of the hook. If you use a hook that exposes three parameters, the first one will be in key zero, the second in key one, and so on. In this example, you just need to take the first parameter from save_post and the post ID and send it to the next session.

The prepare_data method forms the POST data for the session that executes the asynchronous task. Anything you want in that session must be returned at prepare_data or stored in the database.

Here is the updated class to send the post ID:

```
class Josh_Task extends WP_Async_Task {
        /**
         * Action to use to trigger this task
         *
         * @var string
         */
       protected $action = 'save_post';

        /**
         * Prepare POST data to send to session that processes
the task
         *
         * @param array $data Params from hook
         *
         * @return array
         */
       protected function prepare_data($data){
           return array(
               'post_id' => $data[0]
           );
       }
}
```

This method is executed in a different session than prepare_data.

The third and final method of this class, “run_action” is a protected function that is used to actually run the task. This method is executed in a different session than prepare_data. As a result, inside of it, you must use the POST superglobal to retrieve the data you need to run the task. Luckily, the POST data is what you set up in prepare_data.

In this method, validate that the post_id in POST is set and is an integer, and if so use it to fire another action. The wp-async-task’s authors suggest using wp_async_ as a prefix.

```
class Josh_Task extends WP_Async_Task {
     /**
      * Action to use to trigger this task
      *
      * @var string
      */
    protected $action = 'save_post';

     /**
      * Prepare POST data to send to session that processes
the task
      *
      * @param array $data Params from hook
      *
      * @return array
      */
    protected function prepare_data($data){
       return array(
             'post_id' => $data[0]
       );
    }

     /**
      * Run the asynchronous task
      *
      * Calls send_to_api()
      */
    protected function run_action() {
       if( isset( $_POST[ 'post_id' ] ) && 0 < absint(
$_POST[ 'post_id' ] ) ){
             do_action( "wp_async_$this->action",
$_POST[ 'post_id' ], get_post( $_POST[ 'post_id' ] ) );
       }
    }
}
```

WIRING IT UP

Now that the class is in place, you can pull it all together in two simple steps.

The first step is at plugins_loaded or later to instantiate the class.

Obviously, the wp-async-task library must be present and included before this point. I recommend installing it via composer. You can also install it as a plugin, which I think is a very bad idea as plugins can be deactivated.

The second thing you need to do is hook your new action, which is fired inside the class to the original callback.

Here is how that looks:

```
add_action( 'wp_async_save_post', 'josh_send_to_api' );
function josh_send_to_api( $id ) {
    $thing = get_post_meta( $id, 'something', true );
    $r = wp_safe_remote_post( add_query_arg( 'id', $thing,
'http://apiexample.com/' ) );
    if ( ! is_wp_error( $r ) ) {
      $body = json_decode( wp_remote_retirve_body( $r ) ) );
      if ( isset( $body->key ) ) {
            update_post_meta( $id, 'api_response',
$body->key );
      } else {
            update_post_meta($id, 'api_response', 'none');
      }
    }
}
```

MORE ASYNCHRONOUS

I hope this chapter has helped you understand why you need to use PHP in an asynchronous manner. With the practical example, you should now understand how to do it using a regular WordPress site.

REST APIs and PHP

In this chapter, I will discuss Object-Oriented PHP and how it relates to the WordPress REST API. I will bring them together to show you how to build out a collection of custom REST API routes while applying the principles of object-oriented PHP.

One of the great things about inheritance in Object-Oriented PHP is that it lets us share code between classes. There are a lot of ways to avoid repeating ourselves in our code, and inheritance is one of them, but should only be used when two or more classes have similar purposes.

If you have a class that creates an admin page and one that manages searches, they might need to share some code but it doesn't make sense to have them extend the same base class because they have different functionality. On the other hand, if you have a class to make REST API endpoints for products, and one for generating REST API endpoints for documentation on those products, it would make sense that both classes would share a common base class.

In this chapter, you'll learn how to design a system for building WordPress REST API endpoints, taking advantage of class inheritance and making use of visibility and other principles I've discussed previously. If you're unfamiliar with custom REST API endpoints, then you should review The Ultimate Guide To The REST API before moving forward.

I also recommend looking at how post type routes are built in the REST API plugin. That plugin makes great use of class inheritance to build out routes for each post type.

DESIGNING THE SYSTEM

STRUCTURE

For a complete example of a REST API system, we have three general asks:

- Generating routes
- Creating responses
- Booting the system

When creating routes, by way of example, I'll create an abstract class for CRUD routes.

When creating routes, by way of example, I'll create an abstract class for CRUD routes. I'll also create a more generic interface, which those routes will implement. If you want to add routes that are not great matches for that pattern, you shouldn't force them into it, which is why you have an interface.

At the same time, the class that boots the system needs to be able to expect a certain class structure of the route objects it is working with. So the method in the class that boots the system will be type hinted to accept only classes that implement that interface. It would have worked to make it accept classes that extend the CRUD base class, but then we would have lost a lot of flexibility, and would have had to rewrite the system if non-CRUD routes were added.

The third task is creating responses. For the sake of this example, I will just be creating a generic successful response class and an error class. They will only extend WP_REST_Response and WP_Error respectively. Because all of the routes will return these objects, if you need common functionality in your responses later on, there is no need to refactor, just add the methods to those classes.

In this example, I'm going to use the PSR-4 directory structure and namespacing. PSR-4 is great because it uses your directory structure to help define what your code does. I often start a project by just laying out the directories to help me visualize what I will need.

START WITH THE CONTRACT

Our CRUD base route class will define one pattern.

Interfaces create a contract which must be followed by the classes that implement them. In this example, we are only going to use one interface, and it makes it so every class that implements it has to have a method called "add_routes" that accepts one argument called namespace.

As a result, we can reliably know that all objects that implement this have that method and can be used in a predictable manner. How they handle adding routes doesn't matter. Our CRUD base route class will define one pattern. The classes that extend it should follow that pattern or they can override that method. Other classes can implement this interface as long as they can fit the terms of agreement defined by the interface.

Here is our very simple interface:

(According to Marie Dodson, Josh has to resubmit code for this post)

CREATING AN ABSTRACT CLASS

Now let's make an abstract class called crud which implements this interface. Of course, it has to have a method called "add_routes," so let's start with that. The naming convention for the set of routes this is going to create is largely lifted from the core REST API plugin. It's a good pattern for CRUD results and sticking to that core standard helps other WordPress developers make sense of our code.

Here is what that method looks like:

```
interface route {
     public function add_routes( $namespace );
}
```

Outside of that method this class will have three groups of methods: endpoint callbacks, permissions callbacks, and utility methods. I'll walk through each group one by one.

Let's start with the utility methods. These are three simple methods that we need to make sense of the rest of the code in this class. The first is a method called "not_yet_response." This method returns a 501 "Not Yet Implemented" HTTP status code. All of our routes are going to return this by default. That way the subclasses will be able to respond to all of the possible CRUD endpoints just by virtue of being there, and we can slowly add the functionality as needed. That method looks like this:

This method returns a 501 "Not Yet Implemented" HTTP status code.

```
public function add_routes( $namespace ) {
          $base = $this->route_base();
          register_rest_route( $namespace, '/' . $base, [
                      [
                            'methods'           => \WP_REST_
Server::READABLE,
                            'callback'          => [ $this,
'get_items' ],
                            'permission_callback' => [
$this, 'get_items_permissions_check' ],
                            'args'              => [
                                  'page' => [
                                        'default' => 1,
                                        'sanitize_
callback'  => 'absint',
                                  ],
                                  'limit' => [
                                        'default' => 10,
                                        'sanitize_
callback'  => 'absint',
                                  ]
                            ],
                      ],
                      [
                            'methods'           => \WP_REST_
Server::CREATABLE,
                            'callback'          => [ $this,
'create_item' ],
                            'permission_callback' => [
$this, 'create_item_permissions_check' ],
                            'args'              => $this-
>request_args()
                      ],
                ]
          );
          register_rest_route( $namespace, '/' . $base . '/
(?P<id>[\d]+]', '
                      [
                            'methods'               => \WP_
REST_Server::READABLE,
                            'callback'              => [
$this, 'get_item' ],
                            'permission_callback' => [
$this, 'get_item_permissions_check' ],
                            'args'                  => [
                                  'context' => [
                                        'default' =>
'view',
                                      ]
                               ],
                        ],
                        [
                               'methods'             => \WP_
REST_Server::EDITABLE,
                               'callback'            => [
$this, 'update_item' ],
                               'permission_callback' => [
$this, 'update_item_permissions_check' ],
                               'args'                =>
$this->request_args(  )
                        ],
                        [
                               'methods'             => \WP_
REST_Server::DELETABLE,
                               'callback'            => [
$this, 'delete_item' ],
                               'permission_callback' => [
$this, 'delete_item_permissions_check' ],
                               'args'                => [
                                      'force' => [
                                             'default'  =>
false,
                                             'required' =>
false,
                                      ],
                                      'all'   => [
                                             'default'  =>
false,
                                             'required' =>
false,
                                      ],
                                      'id'    => [
                                             'default'
=> 0,
                                             'sanatization_
callback' => 'absint'
                                      ]
                               ],
                        ],
                  ]
           );
```

You can see it makes use of the two response classes I mentioned before. I'll show you how they work shortly.

The second method, "route_base" is a method to get the base for the route. If a class called "shoes" extended this class, this method would return "shoes." You might have noticed this is used in the add_routes method to form the endpoint URLs.

```
protected function not_yet_response() {
            $error =  new error( 'not-implemented-yet',
__( 'Route Not Yet Implemented :(', 'your-domain' )  );
            return new response( $error, 501, [] );
}
```

The third method in this group is called "request_args," which will define the endpoint arguments for this class. This method is declared abstract, so the subclasses will have to define a set of arguments via that method.

You might have noticed that these three methods are protected. This visibility level makes sense as these are methods that are only to be used internally. I did not use private visibility because I want flexibility to override them in subclasses.

If I was writing this class for PHP 7 only, I would make sure each of these three methods had a return type declared to ensure they always returned the correct data type. That way if they were overridden in a subclass, they would definitely return the right type.

Subclasses must de ne and create item permission, but that will be the same for deleting and updating items unless set otherwise.

The next group is the endpoint response methods. Each of these methods returns the not_ yet_response method. One advantage of this is in the subclasses of this class the inline doc can use @inheritdoc instead of duplicating the same inline docs everywhere else.

One other thing to keep in mind about these methods is that they are public. They have to be, because they are called by the WP_REST_Server class. Any other visibility setting would generate an error. But we don't want these methods used in any context besides responding to a REST API request. That is why I type hinted the argument they accept as an object of the WP_Rest_Request class. This means they can be used by the WP_REST_Server class, and I can pass mock requests in as part of my unit tests.

That last paragraph also applies to the third group of methods in this class -- the permissions checks methods. These methods are used to determine permissions for each of the routes. I made the decision to declare the get_item and create_item methods abstract and to have the other methods return one of those two methods.

The logic behind that decision is that it forces each subclass to define at least the general read and write permission. But, since the permission to read one item or many items is generally the same, there is no need to define that separately. The same goes for write routes. Subclasses must define and create item permission, but that will be the same for deleting and updating items unless set otherwise.

This could lead to subclasses adding the same permission callbacks, and, for that reason, you might wish to create abstract subclasses of this class to handle permissions.

For example, here is a simple class called "public_route," which makes it so the read routes are public, but only admins can write data:

```
namespace josh\api\routes;


abstract class public_route extends crud {
     /**
      * @inheritdoc
      */
     public function get_items_permissions_check( \WP_REST_
Request $request ){
            return true;
     }

     /**
      * @inheritdoc
      */
     public function create_item_permissions_check(
\WP_REST_Request $request ){
            return current_user_can( 'manage_options' );
     }
}
```

The error response class doesn't even have a body.

RESPONSES

As I said before, this system doesn't do much with the response classes. The error response class doesn't even have a body. By starting with these classes in place and using them instead of the core classes they extend, it's easy to add utility methods to these classes or override methods in the core classes as the system grows, which can then be applied to all responses from this API.

Here is my main response class:

```
namespace josh\api\error;

class response  extends \WP_REST_Response{

     public function __construct( $data, $status,
array $headers ) {
            parent::__construct( $data, $status, $headers );
            if( empty( $data ) ){
                   $this->set_status( 404 );
            }
     }
}
```

As you can see this is a very simple starting point. I made the decision to check if the data being returned was empty and if so, reset the status code to 404. That's an opinionated decision, but in my experience it simplifies things.

As I said before, my error response just extends WP_Error and doesn't do anything else, but it's the best option.

```
namespace josh\api\error;

class error extends \WP_Error {
}
```

PUTTING IT TOGETHER

The core REST API plugin does this for the post type routes.

Now, to show how this can work, let's make a class that extends "public_route" so we have all of the endpoints we need and the permission checks in place. I'll show how to use endpoints to get one item or many items.

In my example, I'll show how to flesh out getting items of a post type called "my-product." In this route, the database query, via WP_Query will be in the route. I would almost always abstract this into separate CRUD operations not tied to the REST API for use in a plugin or app. That is true even if those CRUD operations were all wrappers for WP_Query as it would make it easier for me to move my data away from posts and custom fields later. I'm a firm believer that creating a database abstraction from the start is essential, as Pippin Williamson demonstrates here.

Because I don't want my whole REST API to be tied to WP_Query, I did not put the prepare_item_for_response and prepare_items_for_response methods in the base class. The core REST API plugin does this for the post type routes. While this makes sense in that context, your custom API should be flexible in what database abstraction it uses — for example, you might use posts, custom tables, or another database system entirely. Those are decisions I like to keep out of the base classes.

Here are those methods, which as you can see are type hinted to expect WP_Query and WP_Post objects:

```
protected function prepare_item_for_response( \WP_Post $item ){
     return [
            'name' => $item->post_title,
            'description' => $item->post_excerpt,
            'price' => get_post_meta( $item->ID, 'price', true )
     ];
}
protected function prepare_items_for_response( \WP_Query $query ){
     $items = [];
     if( ! empty( $query->posts ) ){
            foreach ( $query->posts as $post ){
                   $items[ $post->ID ] = $this->
prepare_item_for_response( $post );
            }
     }
     return $items;
}
```

In this class, I added a query class that just wraps WP_Query. The reason for this is that in the future I might subclass WP_Query or remove WP_Query and I need the flexibility to easily do that. Flexibility and the single responsibility principle are the two reasons why I made a separate method for creating arguments for WP_Query.

```
protected function query_args( $type = '', $page = 1 ){
    $args = [
        'post_type' => $this->post_type,
        'paged' => $page,
    ];

    if( ! empty( $page ) ){
        $args ['tax_query' ] =[
            [
                'taxonomy' => 'product_type',
                'field'    => 'slug',
                'terms'     => $type,
            ],
        ];
    }
    return $args;
}
```

This class has a lot of small methods, which is great as it is designed both to create a collection of endpoints for products and also to serve as the base class for other, more specific endpoints. Here is a subclass that just changes the query_args and request_args class to make a set of endpoints called "shoes" that only returns products with the "shoe" terms of the "product-type" taxonomy."

```
namespace josh\api\routes;

final class shoes extends product {
    /**
     * @inheritdoc
     */
    public function request_args(){
        $args = parent::request_args();
        unset( $args[ 'type' ] );
        return $args;
    }

    /**
     * @inheritdoc
     */
    protected function query_args( $type = '', $page = 1 ) {
        return parent::query_args( 'shoes', $page );
    }
}
```

In both methods, I used the parent keyword to call the method being overridden and customize its behavior. This simplifies things and compared to cutting and pasting from the parent method, makes it easier to sync future changes.

STARTING IT UP

THE BOOT CLASS

This class is going to have a constructor that takes the API namespace that is applied to all routes.

The one thing I have not shown you is how to start up the system. This is fairly simple. There is one class that takes all of the objects of the route classes and calls the add_routes method -- a method we will safely assume exists because of our interface.

This class is going to have a constructor that takes the API namespace that is applied to all routes. Having this as an argument means the same class can be used to load multiple sets of endpoints that might have different namespaces or different version numbers appended to the namespace.

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

The constructor does not actually take the routes. I designed it this way as we need to construct an array of objects that implement the interface. I could have made it so this class took an array in its constructor, and then checked the contents of that array, however, this is poor practice, so instead I added a method to add routes to that array and type hinted its one argument.

```
public function add_route( route $route ){
     $this->routes[] = $route;
}
```

Because adding route objects to this class happens after the class is instantiated, I needed a method to loop through that array instead of doing it in the constructor. The method just loops that array and calls the add_routes method on each of the objects.

```
public function add_routes(){
     if( ! $this->booted && ! empty( $this->routes ) ){
          /** @var route $route */
          foreach ( $this->routes as $route ){
               $route->add_routes( $this->namespace );
          }
          $this->booted = true;
     }
}
```

Note that I am using a property to prevent this method from adding to the routes twice.

BASE COMPOSITION

So far I have not hooked any of this code to any actions. As a result, we have a totally isolated system that can be called in a variety of ways. It's safe to assume that we will want this code loaded on the rest_api_init hook.

Here is the base file for the plugin I built for this example, which registers an autoloader and instantiates the route and boot classes at rest_api_init. If you are applying this system to a plugin or theme, this probably doesn't make sense. For a specific site or app, however, it makes perfect sense.

```
use \josh\api\boot;
add_action( 'rest_api_init', function(){
     //http://www.php-fig.org/psr/psr-4/examples/
     spl_autoload_register(function ($class) {

            // project-specific namespace prefix
            $prefix   = 'josh\\api\\';
            $base_dir = __DIR__ . '/api/';
            $len      = strlen( $prefix );
            if ( strncmp( $prefix, $class, $len ) !== 0 ) {
                   return;
            }

            $relative_class = substr( $class, $len );

            $file = $base_dir . str_replace( '\\', '/',
$relative_class ) . '.php';

            if ( file_exists( $file ) ) {
                   require $file;
            }
     });

     //make product route
     $product = new \josh\api\routes\product();

     //OMG(s) shoes! Shoes!
     $shoes = new \josh\api\routes\shoes();

     //make API go
     $api = new boot( 'store\v1' );
     $api->add_route( $product );
     $api->add_route( $shoes );
     $api->add_routes();
});
```

It is worth noting that you may want to use it in different contexts. For example, if we want to generate an API response without an HTTP request. One potential use for that is if we had a theme or other front-end interface that worked from a JSON object. On initial page load, we could call the route directly and print the JSON response or pass it to wp_localize_script.

STARTING WITH A SYSTEM

I put all of this code up in a git repo, which can be added your own project as is. That said, I'd encourage you to fork it and make it work with your own needs.

More importantly, I encourage you to use this as an example of how to take different parts of your plugins, themes, sites, and apps and create systems around common functionality. This article uses the REST API as an example because it's useful and increasingly important to WordPress. In addition, I learned a lot about how following the principles of object-oriented PHP makes for an easy to understand, extensible, and testable system by reading the code for the REST API version 2.

I encourage you to design different groups of functionality in your code this way. Too often I find myself diving right into the code to accomplish a task, and then find myself working backwards to pull a lot of what I had done into abstract classes and interfaces and then start building systems around them.

Defining the system first forces you to think about flow. It forces you to think about how something would ideally work. Most of the time, thinking in terms of repeatable systems, even if you only need it once, will lead to building better, more testable, and more flexible systems that needs less refactoring as the application grows.

```
ublic Optimization process
{
    String[] sArray = words
    for (String str : sArra
    {
        System.o
    }
return null;
}

public List<List<Integer>>
return null;
}
i.processData("{3,39,20,#,

    }
    }
```

PHP Magic Methods

If you're not using these tools, you should be.

Chances are, if you've looked at PHP code, you've encountered magic methods. You can tell it is a magic method because the method is preceded by two underscores. If you're not using these tools, you should be.

Magic methods are functions that allow you to show an object how to react when something happens to it. You get to define how your object should react in an event. This can be a huge tool in preventing errors and reducing redundant code.

In this chapter, I am going to introduce you to some of the magic methods that PHP provides. I have chosen these methods because they are most useful to you and help you see the patterns of how they work.

CONSTRUCT

The most used magic method is __construct(). This function is called as soon as a class is instantiated. It has several uses.

The first use is to inject data into the class. For example, when creating an instance of the WP_Query class, you would normally pass an array of arguments directly to it:

```
$args = [ 'post_type' => 'hiroy' ];
$query = new WP_Query( $args );
```

When you pass arguments into the class at instantiation, those arguments are passed to its constructor. This allows you to have one class that produce different objects.

The second use is to set off the sequence of events that must happen before the object is usable. For example, let's say you have a class that is designed to create specific HTML markup, by doing some of the work based on a collection of posts.

You could use the constructor to set off that process that creates the HTML so that it is ready when you need it, or you can trigger an error later. This is useful when using "getters" and "setters" as I will discuss later on in this chapter.

In PHP 4 and PHP 5, it is ok to have a method with the same name as the class functions, however, for the sake of future compatibility, this is never a good idea. Also, this "pseudo-constructor" can not be declared static or an error will be thrown.

SETTING HOOKS IN THE CONSTRUCTOR

It is a common practice to add hooks in the class constructor. This may be a bad idea because hooks called using the $this reference to current object are hard to unhook. Also, it makes the class less reusable.

One solution is to move hooks outside of object context. The other is to keep all hooks inside of a specific "set up" class or classes that have no functional reason to be instantiated twice, or implement the singleton pattern to ensure they are only instantiated once.

Another option is to use a "setup action" and the did_action() function to prevent running again, which will look like this:

```
<?php
class stuff {
     public function __construct(){
          if( ! did_action( 'init_stuff' )  ){
               $this->add_hooks();
          }
     }
     protected function add_hooks(){
          //add your hooks
          do_action( 'init_stuff' );
     }
}
```

This pattern works but violates the single-responsibility principle.

MAGIC SETTERS AND GETTERS

We refer to a method of a class that is used to set or get the value of a class property.

In general, when we talk about a "setter" or "getter" method, we refer to a method of a class that is used to set or get the value of a class property. By convention, we use the prefix "set_" and "get_" for these types of method. This pattern is useful for properties of a class that we want to be accessible outside of the class, but do not want to be modifiable outside of the class.

Here is an example class that takes two dates and queries for the most commented on posts during that period:

```
<?php
class popular_posts {
     /**
      * WP_Query object
      *
      * @var WP_Query
      */
     protected $query;
     /**
      * Create object with WP_Query of most commented on
posts in a given period
      *
      * @param string $before
      * @param string $after
      */
     public function __construct( $before, $after ){
          $this->set_query( $before, $after );
     }
     /**
      * Get WP_Query
      *
      * @return \WP_Query
      */
```

```
        public function get_query(){
                return $this->query;
        }
        /**
         * Set query property
         *
         * @param string $before
         * @param string $after
         */
        protected function set_query( $before, $after ){
                $args = array(
                        'date_query'            => array(
                                'after' => $before,
                                'before' => $after,
                                'inclusive'         => true,
                        ),
                        'orderby'           => 'comment_count',
                        'order'                     => 'DESC',
                        'posts_per_page'        => '5',
                        'paged'                     => '1',
                );
                $this->query = new WP_Query( $args );
        }
}
```

This class has a protected property called query. It is set by the method set_query(). That method is protected since it only makes sense to call it once. It is called by the constructor and then it is done.

It's protected to prevent it being changed after the an object of this class is instantiated. That's good, but the object would be worthless if you couldn't do something with that object. That's why we have the "getter" method get_query(), which just returns the property $query.

Using get_ and set_ for getters and setters is a convention, but it is not required. PHP does provide a magic getter that if present is called when a property is accessed from outside of the scope of the object. A __get() magic method can be used to make some or all protected and private methods of a class public.

The magic __get() is passed the name of the property. This enables you to allow specific properties to be returned, or return something else. Also, you may wish to use __get() to call a setter.

It can be used to create an arbitrary setter for all properties of a class.

The __set() magic method is passed the name of the property being set and its value. It can be used to create an arbitrary setter for all properties of a class, and making some or all protected and private properties writeable outside from outside of the class.

Magic getters and setters have some usages, for example, validating types before allowing properties to be changed. For example, a class where it makes sense for one property to be mutable outside of the object, but you want to ensure the type stays the same.

Here is a class for using any WP_Query through a provided partial. It gets around the issue that a public property can change its type. This class would throw errors if the property $query was changed to an array, or an object of any class other than WP_Query.

```
<?php
class query_loop {
	/**
	 * @var WP_Query
	 */
	protected $query;
	protected $partial_partial;
	public function __construct( $partial_partial ){
		$this->partial_partial = $partial_partial;
	}
	public function __set( $property, $value ){
		if( 'query' == $property && ( is_a( $value, 'WP_
Query' ) ) ) {
			$this->query = $value;
		}
	}
	public function __get( $property ){
		if( 'query' == $property ){
			return $this->query;
		}
	}
	public function output_loop(){
		$query = $this->query;
		ob_start();
		foreach( $query->post as $post ){
			$post = setup_postdata( $post );
			include $this->partial_partial;
		}
		return ob_get_clean();
	}
}
```

RAW

Note that this class' use of the __get() and set() doesn't really make it any more compact than using explicit getters and setters. Also, it makes the inline docs less useful. This shows the weakness of using magic get and set methods.

You could theoretically make a class that allows a property to be set or retrieved. Here is a class that acts almost the same as the stdClass, but checks if a property is set before returning it in the __get() magic method:

```
<?php
class anything {
	/**
	 * Get any property that is set
	 *
	 * @param string $property Name of property
	 *
	 * @return mixed
	 */
	public function __get( $property ){
		if( isset( $this->$property ) ){
			return $this->$property;
		}
	}
	/**
	 * Add anything to this class
	 *
	 * @param string $property Name of property
	 * @param mixed $value Value to set
	 */
	public function __set( $property, $value ){
		$this->$property = $value;
	}
}
```

RAW

Magic getters and setters are useful when a class has a lot of properties, not all of which may be used each time. With lots of properties that need to be made "semi-public" and potentially call a private explicitly defined "getter" this can be useful with a __get() like this:

```
abstract class check_set {
	/**
	 * Get a property of this class -- if it exists, will
call this_{$property_name} if not set an such a method
exists
	 *
	 * @param string $property Property name
	 *
	 * @return mixed Property value
	 */
	public function __get( $property ){
		if( property_exists( $this, $property ) ) {
			if( ! isset( $this->$property ) && method_
exists( $this, 'set_' . $property )  ){
```

Magic get and set are useful, but are often more trouble than they are worth.

```
                                        call_user_func( [ $this, 'set_' .
    $property ] );
                               }
                               return $this->$property;
                    }
          }
    }
```

Magic get and set are useful, but are often more trouble than they are worth. They can save a lot of redundant code, but they make the code less explicit. They do solve many problems. Just don't use them because it's clever or you can. Use them because they solve a real problem.

For example, the magic __get() method in the WP_Post class is used to allow post objects to access meta values of that post, of which there is an arbitrary number.

CONVERTING OBJECTS TO STRINGS

There are many situations where we may need a string representation of an object. PHP provides a __toString() magic method that allows you to use an object as a string. This allows you to echo an object and have a meaningful string returned instead of throwing an error.

Another, more useful, example is when we wish to store an object in a MySQL database. This requires serializing the object into a string. When an object is passed to the function serialize() the magic method __sleep is called first. When an object is passed to unserialize() the magic method __wakeup() is called.

You should be very careful with these methods as they can be used to perform remote data execution attacks. In fact, it is very common to use them to prevent serialization of a class.

A good use for these methods is to reduce the size of objects. For example, if a class has properties that are calculated, they can be removed before serialization and re-calculated after serialization. This saves space in the database and might speed up queries.

For example, consider a class that creates objects for conversion rate of a page. Conversion rate is the number of conversions divided by the number of page views. Here is an example class that takes conversions and page views and calculates the conversion rate, which makes use of both sleep and wakeup and provides for another useful example of the __get() magic method:

```
<?php
class conversion_rate {
     /**
      * @var int
      */
     protected $conversions;
     /**
      * @var int
      */
     protected $total;
     /**
      * @var float|int
      */
     protected $conversion_rate;
     /**
      * @param int $conversions
      * @param int $total
      */
     public function __construct( $conversions, $total ){
            $this->conversions = $conversions;
            $this->total = $total;
     }
     /**
      * @return float|int
      */
     public function get_conversion_rate(){
            return $this->conversion_rate;
     }
     public function __sleep() {
            return [ 'total', 'conversions' ];
     }
     public function __wakeup() {
            $this->set_conversion_rate();
     }
     /**
      * @param string $property
      * @param int $value
      */
     public function __set( $property, $value ){
            if( 0 <= $property && in_array( $property, [
'conversions', 'total' ] ) ){
                   $this->$property = (int) $value;
                   $this->set_conversion_rate();
            }
     }
```

```
    protected function set_conversion_rate(){
        if ( 0 != $this->total ) {
            $this->conversion_rate = $this->conversions 
/ $this->total;
        }else{
            $this->conversion_rate = 0;
        }
    }
}
```

This class uses the __sleep() magic method to tell PHP to only serialize the total and conversions properties. That's not going to shorten the serialized string by much, but if there was a lot of these objects being saved, or you wanted additional math, it could save a lot of space in the database.

While it doesn't use a magic method, there is a way to get similar functionality as __sleep() provides for serializing when converting an object to JSON. If a class implements the JsonSerializable interface, it can have a method called JsonSerializable() which is used to define how the object is converted to JSON when passed to the function json_encode().

Here is a new class, which extends the earlier class, that can be used to create a JSON string that has all three properties in it.

```
<?php
class json_conversion_rate extends conversion_rate 
implements \JsonSerializable {
    public function jsonSerialize() {
        $this->set_conversion_rate();
        return [
            'total' => $this->total,
            'conversions' => $this->conversions,
            'conversion_rate' => $this->conversion_rate
        ];
    }
}
```

USE MAGIC, LEARN MORE MAGIC

Take a look at all of the magic methods listed in the PHP manual and see which other ones may be useful to you.

But also remember that magic methods let you do clever stuff, but that doesn't mean that you should use them.

Any advanced wizard will tell you, too much magic, no matter how well-intentioned, is dangerous and is bound to have unintended consequences.

Namespaces

This chapter assumes that you're already familiar with the basics of Object-Oriented PHP.

Namespacing and class autoloaders are two important tools provided by PHP that WordPress developers should take advantage of more frequently. This chapter offers an in-depth introduction to namespaces in PHP.

Namespaces are the better, more flexible version of using unique preferences in your class names. In addition they help structure your directories and allow you to take full advantage of autoloaders that follow the latest standards, including Composer's autoloader.

This chapter assumes that you're already familiar with the basics of Object-Oriented PHP. If you're not, be sure to read my introduction to the topic. I also have more resources on learning Object-Oriented PHP on my blog.

I'm also assuming you are using at least PHP 5.3. If you're still using PHP 5.2, you really should stop, as that version of PHP, which reached its end of life in January 2011, is slow and insecure. Keep in mind that if you are looking to apply this in a plugin or theme for public release, you will run into users who are using PHP 5.2 and using namespaces will create a fatal error for those users, who almost certainly will not know why.

NAMESPACING YOUR CLASS

I'm going to walk you through using namespacing. In these examples, let's assume that the plugin is called Fun Machine.

Here is what a typical class looks like using normal WordPress class naming standards:

```
<?php
class Fun_Machine_Shortcode {
     /**
      * Process the shortcode
      *
      * @param $atts
      *
      * @return string
      */
     public static function the_shortcode( $atts ) {
            $atts = shortcode_atts( array(
                   'foo'   => 'fighter',
                   'dave'  => 'grohl'
            ), $atts, 'fun-machine' );

     return self::shortcode_callback( $atts['foo'],
$atts['dave'] );
     }
     /**
      * Callback for shortcode
      *
      * @param string $foo
      * @param string $dave
      *
      * @return string
      */
```

This system gets cumbersome, especially if you like having small, single-purpose classes.

```
    public static function shortcode_callback($foo, $dave) {
        return $foo.$dave;
    }
}

add_shortcode( 'fun-machine', array( 'Fun_Machine_
Shortcode', 'the_shortcode' ) );
```

The class gets named first for the plugin's name and then for the purpose of the class. This makes sense because naming it just for its purpose, IE "shortcode," is a major problem since two classes can't have the same name or a fatal error will occur. What would happen if another plugin added a class called shortcode? There would be a fatal error and both plugin authors would be guilty of _doing_it_wrong().

This system gets cumbersome, especially if you like having small, single-purpose classes. Instead, we can put this class in the "fun_machine" namespace. Here's the class, just the outline, skipping the content with a namespace:

```
<?php
namespace fun_machine;

class shortcode {
    //do the same stuff
}
```

Now that we have a namespace, the way we used add_shortcode() to register the shortcode from the first example isn't going to work. We have to use the namespace in it. Namespaces are separated by a forward slash. The shortcode registration would become:

```
add_shortcode( 'fun-machine', array( '\fun_machine\
shortcode', 'the_shortcode' ) );
```

If we wanted to call the shortcode_callback method in another class, we would use:

```
\fun_machine\shortcode::callback(  $foo, $dave );
```

While I have been starting with a forward slash, that is not always necessary. The starting forward slash puts us into the global namespace, so it is not needed when in the same namespace.

To illustrate the difference, let's look at three small classes:

```
<?php
namespace fun_machine;

class add_shortcode {
     /**
      * Process the "fun" shortcode
      *
      * @param $atts
      *
      * @return string
      */
     public static function fun( $atts ) {
            $atts = shortcode_atts( array(
                   'foo'   => 'fighter',
                   'dave'  => 'grohl'
            ), $atts, 'fun' );

            return shortcode_callbacks::fun( $atts['foo'],
$atts['dave'] );
     }
}
?>

<?php
namespace fun_machine;

class shortcode_callbacks {
     /**
      * Callback for "fun" shortcode
      *
      * @param string $foo
      * @param string $dave
      *
      * @return string
      */
     public static function fun( $foo, $dave ) {
            return \less_fun\teeth::do_something_to_foo( $foo
            );
     }
}
?>

<?php
namespace less_fun;

class teeth {
     /**
      * Does two fairly opposite things
      *
      * @param string $foo
      *
      * @return string
      */
```

```
        public static function do_something_to_foo( $foo ) {
                return strtolower( strtoupper( $foo ) );
        }
}
```

In the first class a method, add_shortcode(), from the second class is called, and it works the same way as it would without a namespace, as they are in the same namespace. In the second class, a method from the third class is called. Since they are in different namespaces, we had to use the fully qualified namespace: \less_fun\teeth::do_something_to_foo

NESTING NAMESPACES

So far we have only used single namespaces, but namespaces can be inside of each other.

So far we have only used single namespaces, but namespaces can be inside of each other. For example, we could have a "shortcodes" namespace under the main namespace "fun_machine." Here is what the classes would like:

```
<?php
namespace fun_machine\shortcode;

class add_shortcode {
        /**
         * Process the "fun" shortcode
         *
         * @param $atts
         *
         * @return string
         */
        public static function fun( $atts ) {
                $atts = shortcode_atts( array(
                        'foo'   => 'fighter',
                        'dave'  => 'grohl'
                ), $atts, 'fun' );

                return shortcode_callbacks::fun( $atts['foo'],
$atts['dave'] );
        }
}
?>

<?php
namespace fun_machine\shortcode;

class shortcode_callbacks {
        /**
         * Callback for "fun" shortcode
         *
         * @param string $foo
         * @param string $dave
         *
         * @return string
         */
        public static function fun( $foo, $dave ) {
              return \less_fun\teeth::do_something_to_foo( $foo
              );
        }
  }
  ?>
```

Now to register the shortcode we would need to do this:

```
add_shortcode( 'fun-machine', array( '\fun_machine\shortcode\
add_shortcode', 'fun' ) );
```

Nesting namespaces helps make it easier to show what a class does, based on its namespaces. It also helps group classes by purpose and dictates file structure in a logical and consistent manner.

ALIASING NAMESPACES

Fortunately, aliasing allows us to use methods of a class in a different namespace using only the class name.

Without the ability to alias namespaces in classes, however, we would have to write those long namespaces over and over again. Fortunately, aliasing allows us to use methods of a class in a different namespace using only the class name.

Aliasing is accomplished with a simple "use" statement between the namespace declaration and the beginning of the class. Here is the shortcode_callbacks class from the previous examples, modified to alias the \less_fun\teeth class.

```
  <?php
  namespace fun_machine;

  use\less_fun\teeth;

  class shortcode_callbacks {
        /**
         * Callback for "fun" shortcode
         *
         * @param string $foo
         * @param string $dave
         *
         * @return string
         */
        public static function fun( $foo, $dave ) {
                return teeth::do_something_to_foo( $foo );
        }
}
?>
```

As you can see, I added a use statement, and as result I was able to change the line "return \less_fun\teeth::do_something_to_foo($foo);" to "return teeth::do_something_to_foo($foo);." As you can imagine, if that class gets used more than once, this can simplify things greatly.

NAMESPACE ALL THE THINGS!!!

That's all you need to know in order to start using namespaces in your WordPress development projects.

After applying just a few new concepts — all of which are made easier by using a good IDE like phpStorm — your code will be easier to follow and more organized.

Also, now that you understand namespaces, you are better prepared for using class autoloaders (which I will cover next time) and for creating and using Composer libraries.

Improving Development Workflow with composer

There are lots of dependency managers out there, but the standard for PHP development is Composer.

A dependency manager is one of those magical tools, like a smartphone, that most people (myself included) didn't know they needed until they tried it, but once they did, they were hooked.

A dependency manager wrangles together all of the different pieces that go into a plugin or theme — such as frameworks, JavaScript libraries, jQuery plugins, or even the plugins, themes, and libraries that make up a WordPress site.

There are lots of dependency managers out there, but the standard for PHP development is Composer. Composer is user friendly and works great with WordPress.

Keep in mind that one of the things that WordPress does is dependency management, and that may be enough for you. A WordPress site is a combination of WordPress core, which is managed via the core updater, and a collection of themes, plugins, mu-plugins, and drop-ins that are managed via the theme/plugin manager and updater.

In some cases that's enough, but for theme and plugin development, or for sites with lots of moving pieces, Composer can be a real frustration saver.

Many people use Git or some other version control system for what a dependency manager, such as composer, should be used for. They have one Git repository for all of their code — both code they are writing and code other people are writing. It also leads to mixing multiple parts of the code you are writing in one repository. Composer actually makes working with multiple repositories easier.

Composer allows you to have multiple repositories with the theme and plugins you're developing for a site, along with the code you're using managed from one central location. You update all of your dependencies with one command:

```
composer update
```

WHEN AND WHERE TO USE COMPOSER

There are three separate situations where you may want to use composer to manage dependencies in WordPress:

1. To manage dependencies for a theme or a plugin you're developing
2. To manage your themes and plugins used on a site
3. For total site dependency management

Each of these situations has its own needs for dependency management, as well as pros and cons for using Composer. For example if you're developing a site or a WordPress-powered app, you have more options for choosing dependent libraries than when making a plugin or theme for release. So let's address each of these potential uses individually.

The first case is when you're developing a theme or plugin for public release. Themes and plugins are often reliant on third-party JavaScript or SASS/LESS libraries. Composer really does a great job keeping these up to date, as opposed to managing them via Git Submodules, or via cut and paste.

This does, however, get tricky as your dependencies for a theme or plugin may include libraries bundled with WordPress or other plugins, which in this situation, Composer is not great for managing.

On the other hand, Composer is great for keeping a framework like Bootstrap or Foundation updated in your theme, or managing jQuery plugins used in your WordPress plugin.

If you are developing a full site or WordPress powered app, this means you control all of the code on your site, and you have more flexibility than when writing code that might be used by others. Using Composer allows you to break your code into multiple version control repositories, which maximizes code reusability, and manages and updates them from a single location.

Whether Composer manages just your content directory or the entire site, it's an important decision to make. If done right, you can use Composer to manage the entire site, and have the site's Git repo just contain wp-config.php — a index.php and the composer.json file.

A simple example of this can be found in Simeon Wheatley's VVV demo configuration. The actual Git repository just creates the structure. Then Composer is used to fill in WordPress and the plugins and themes. You can use a setup to move between local development and a live server, with relative ease. On the live server, you would just clone the repo and then run composer update.

For a more complex setup that adds automated deployment, I recommend checking out Bedrock. Bedrock uses Capistrano to deploy from local to production, and in turn automates the process of running Composer on the live site.

Composer is great for keeping a framework like Bootstrap or Foundation updated in your theme.

A simpler, though less automated setup, would be to add a composer.json file in the root of your content directory and use it to install plugins and themes only.

One great thing about using Composer for managing dependencies for a full site is that it reduces redundancy. If you want to use one JavaScript library in two custom plugins, you can do so with confidence that managing and updating that library will be easy and installed in a consistent location.

In order to use Composer, you must have it installed on your server and/or development machine. The need to have it on your server might not be valid if you're using Composer for theme/plugin development. In addition, you need a valid composer.json file.

INSTALLING COMPOSER

Composer needs to be installed via the command line, but this is very easy to do.

Composer needs to be installed via the command line, but this is very easy to do on your computer or server. To install Composer globally, which enables using the composer command in any directory with a valid composer.json, you only need two simple commands:

```
curl -sS https://getcomposer.org/installer | php

mv composer.phar /usr/local/bin/composer
```

This assumes your computer has curl installed, which it should. Also keep in mind that you must have command line access to your server, which is not something that all WordPress hosts support. If your host does not give you this access, you will have to run composer locally and push your changes via SFTP or some other method.

WRITING A COMPOSER FILE

The composer.json file defines what files Composer will get for you. These may be plugins, themes, libraries frameworks, or even WordPress itself. In addition, it lists some information about the project.

The dependencies must be from either a Packagist server, in a version controlled repository with a valid composer.json of their own, or point to a zip file.

A Packagist server is a repository for Composer packages.

The main Packagist server is Packagist.org, but as a WordPress developer, you may be more intrested in WPackagist.org, which is a mirror of the WordPress.org theme and plugin repositories as composer packages, which is awesome. I'll show you how to use both shortly.

Here is an example composer.json file for a plugin:

```
{
    "name" : "vendor/plugin-name",
    "description" : "Automatic front-end output of Pods Tem-
plates.",
    "type" : "wordpress-plugin",
    "keywords" : [ "wordpress" ],
    "license" : "GPL-2.0+",
    "authors" : [
        {
            "name" : "Developer Name",
            "email" : "email@somewhere.com",
            "role" : "Lead Developer"
        },
        {
            "name" : "Developer Name",
            "email" :"email@somewhere.com",
            "role" : "Developer"
        },
    ],
    "require" : {
        "composer/installers" : "~1.0.0",
        "php" : ">=5.2.4",
        "twbs/bootstrap" : "3.2"
    },
    "extra" : {
        "installer-name" : "plugin-name"
    }"
    "homepage" : "http://plugin-name.com",
    "support": {
        "issues": "https://github.com/vendor/plugin-name/is-
sues",
        "source": "https://github.com/vendor/plugin-name/
pods-frontier-auto-template"
    }
}
```

You'll notice that most of this is information about the plugin — such as who made it, what it does, where to report issues, etc.

The two most important things to notice are the "require" section and the "type" section. While it's not formalized, there is an effective standard that WordPress plugins use the type "wordpress-plugin" and themes use the type "wordpress-theme." The type is important, as when we look at the Composer file for site management we'll see we can specify a custom path for installing plugins and themes.

In the "requires" section, we list the requirements for this project. You'll notice the line:

```
"twbs/bootstrap" : "3.2"
```

That's an example of how to add a third-party dependency from Packagist.org, in this case Bootstrap. I've listed a specific version of Bootstrap, but you can get the latest version by setting the version number to "*." This will install Bootstrap in a folder called "vendor" in your plugin.

You can change the name or path of that default vendor directory by
adding a "vendor-dir" declaration to the config section of your Composer file, like this:

```
"config"      : {
     "vendor-dir":  "libraries"
},
```

Alternatively, if you're using Composer to manage your entire site, you will want to list all of your plugins and themes, and define a path for where to install them. As I said before, there is a Packagist mirror of the WordPress.org theme and plugin repository at WPackagist.org. This means that we can add any plugin or theme using wpackagist-plugin/plugin-slug or wpackagist-theme/theme-slug. For the plugin or theme slug you use the page slug of the theme or plugin's page on WordPress.org.

Here is an example composer.json for a full WordPress site, that manages plugins, themes, and WordPress itself:

```
{
     "name"         : "vendor/project",
     "description" : "Site build stack.",
     "keywords" : [ "wordpress" ],
     "license" : "GPL-2.0+",
     "authors" : [
             {
                 "name" : "Developer Name",
                 "email” : "email@somewhere.com",
                 "role” : "Lead Developer"
             },
             {
                 "name" : "Developer Name",
                 "email" : "email@somewhere.com",
                 "role" : "Developer"
             },
     ],
     "type"          : "project",
     "minimum-stability": "dev",
     "repositories": [
            {
                  "type": "composer",
                  "url" : "http://wpackagist.org"
            },
             {
                 "type": "git",
                 "url": "https://github.com/pods-framework/
pods-json-api"
             }
     ],
     "config"      : {
            "vendor-dir":  “vendor"
     },
     "require"       : {
             "johnpbloch/wordpress"
: "4.0",
            "wpackagist-plugin/pods"
: "*",
             "wpackagist-plugin/json-rest-api"
: "*",
             "json-rest-api"
: "*",
             "wpackagist-plugin/log-viewer"
: "*",
             "wpackagist-plugin/caldera-forms"
: "*"
     },
     "require-dev" : {
            "wpackagist-plugin/log-deprecated-notices"
: "*",
             "wpackagist-plugin/debug-bar"
: "*",
             "wpackagist-plugin/debug-bar-console"
: "*",
             "wpackagist-plugin/user-switching"
: "*",
             "wpackagist-plugin/simply-show-ids"
: "*"
     },
     "extra"         : {
            "wordpress-install-dir": "public_html/wp",
            "installer-paths": {
                  "public_html/content/plugins/{$name}/" :
["type:wordpress-plugin"],
                  "public_html/conent/{$name}/"   :
```

The two most important things to notice are the “require” section and the “type” section.

```
["type:wordpress-theme"]
            },
        }
}
```

Some of this is similar to the last example, but some is very different. Let's walk through the important parts.

First, notice that in the repositories section is a link to a Git repository. This is necessary, as this particular plugin is not available via Packagist or WPackagist. As a result, we need to tell Composer to treat it as a Git repository and provide it with the path to GitHub repository that the plugin is in.

This is an incredibly powerful feature of Composer, as the GitHub repository could be private, but as long as the computer that you run has access to the repository (i.e., you've added its public SSH key to your GitHub or other Git hosting account), Composer can install it.

The next thing to notice is that we have two separate require sections, the second of which will only be used in development. That's full of useful development plugins that you wouldn't want running on your live server.

You probably noticed that the first line of the require section is installing WordPress itself. While there is no official WordPress package on Packagist, John Bloch's package is a reliable source for all versions of WordPress.

Make sure to pay extra attention to the "extra" section at the very end, which defines custom paths for WordPress itself and WordPress plugins and themes.

Without this, all repositories would go in the vendor file, which is not useful.

The first line of this section defines a path for installing WordPress, which in this case is public_html/wp. Make sure to customize this and the value of ABSPATH to fit your needs. The next two lines tell composer where to install plugins and themes. Again, make sure you match these to your install directory and the value of WP_CONTENT_DIR and WP_CONTENT_URL. You probably saw that those paths include a variable {$name}. That matches the value of "installer-name" that we saw in the example plugin composer.json earlier. When installing from WPackagist, that will be the plugin's or theme's slug.

WAVING THE BATON

There are lots of great resources.

I hope this introduction to Composer has helped highlight some of the amazing capabilities of Composer and how it can improve your development workflow.

There are lots of great resources out there for learning about using Composer with WordPress. Andrey "Rarst" Savchenko curates a collection of resources on using Composer with WordPress. I also recommend checking out David Smith's article on using Composer as the cornerstone of team-based WordPress development.

Using A Class Autoloader To Improve WordPress Development

Using features of PHP that are not backward compatible with PHP 5.2 is an issue with projects that get released publicly.

In the last few chapters, I looked at the benefits of using class autoloaders and namespaces in PHP. Despite the clear benefits of using these features, WordPress developers tend to shy away from them. One of the big reasons for this is because of WordPress' continued support of PHP 5.2, which reached its end of life years ago.

Class autoloaders have been available in various forms since before PHP 5.3. However, the more up-to-date class autoloader — as is generally used in PHP development — require namespaces, a feature added in PHP 5.3.

In this chapter, I will show you how to create and use your own class autoloader, following established PHP community standards. I will also discuss using the Composer autoloader.

I don't think WordPress core should adopt an autoloader at this point. I also don't think all of the standards for core development— specifically class naming, and not using an autoloader or namespaces — should extend to plugin or site development.

Using features of PHP that are not backward compatible with PHP 5.2 is an issue with projects that get released publicly.

In my experience, most users who are affected by this have no idea that they are running such an outdated version. Once confronted with that knowledge, and made aware of the security and performance benefits of upgrading, most users will insist that their hosting provider upgrade their PHP version. That update is usually possible with a few clicks in cPanel.

WHAT IS A CLASS AUTOLOADER?

A class autoloader is a system for automatically loading uninstantiated classes, including the files that contain them. When following an established system for file naming and a standard autoloader, you can reliably use any class, without manually including the file.

Not having to include the file manually sounds like a small thing but it has a big impact: It makes it easier to create small, more manageable and reusable classes.

It also allows you to more easily refactor your classes using an IDE that makes doing so easy, such as phpStorm. In addition, it eliminates one more place for human error.

CHOOSING A STANDARD

The PHP Framework Interop Group is a group of PHP users with voting representatives from the most major PHP frameworks and content management systems — with the notable exception of WordPress. This group sets the standards for common PHP development practices. They have defined two standards for autoloaders: PSR-0 and its successor, PSR-4.

For the most part, these two standards are very similar, however there are two major differences. First, is that PSR-4 requires the use of namespaces. Second, the root directory for a project following the PSR-4 standard does not have to match its root namespace.

For the most part, it is better to follow the PSR-4 standard than the PSR-0 standard it replaced. That is, unless PHP 5.2 backward compatibility is an issue — in which case the PSR-4 standard can not be used, since it requires namespaces.

NOT CHOOSING A STANDARD

You do not have to follow a standard to use class autoloading. Nor does it require the use of a special autoloading class or the use of Composer autoloader. You can make your own.

Following a standard makes it easier to use Composer, and it allows you to share one autoloader for multiple libraries. The latter case is especially useful in full-site development, when you develop multiple plugins, each for a specific function.

The Composer autoloader does not require following the PSR-0 or PSR-4 standard. But following one of those standards does make it much easier to use. If you do not follow one of those standards, you can always use Composer's classmap to autoload your classes using the Composer autoloader.

USING THE COMPOSER AUTOLOADER

One of the easiest ways to utilize a class autoloader is rely on Composer's autoloader by pushing all of your code into composer libraries. This makes it easier to reuse and share your code.

Using the Composer Autoloader is as simple as adding to your code:

```
require_once( 'vendor/autoload.php' );
```

For full-site development, it may be possible to have one vendor directory for the entire site. For plugin development and other cases you may end up with many vendor directories. That is OK, though it will make deployments a little trickier.

USING A PSR-4 AUTOLOADER

Each class must have the same name as the file and if you are nesting namespaces.

Using a PSR-4 autoloader is very easy, as long as you namespace your classes correctly. Each library has a root directory and namespace. Each class must have the same name as the file and if you are nesting namespaces, then the file directory structure must match the namespace hierarchy.

For example, if your root namespace is "fun" and you have a class called "process" in the "shortcodes" sub-namespace then the following must be true about this class:

1. It should be in a file called "process.php"
2. It should be in a directory called "shortcodes"
3. That directory should be a subdirectory of the main namespace

Here is what that class would look like:

```
<?php
namespace fun\shortcode;

class process {
}
```

Of course, for this to work, your library must be registered with the autoloader in use. This can be done in one of two ways. If it is a composer library, then you simply define that in your composer.json file.

```
"autoload": {
      "psr-4": {"fun\\": "src/"}
}
```

The above example assumes the library's root directory is called "src." It doesn't have to be, but by convention it should be.

The root level of a Composer library generally contains only a readme, composer file, and two directories ("src" and "test"). The latter will contain the library's unit tests.

If you need to use your own autoloader, simply copy the example autoloader from the PSR-4 standard. Be sure to prefix the class name properly, and manually include the file it is in.

Once you have that, you need to register each root namespace. Sub-namespaces will work automatically.

Once you have that, you need to register each root namespace.

Here is how we would register our "fun" namespace:

```
<?php
// instantiate the loader
$loader = new \slug_autoloader

// register the "fun" namespace
//we are assuming that the root namespace is in "src", a
subdir of the current directory.
$loader->addNamespace('fun', dirname( __FILE__ ) . '/src' );

// register the autoloader
$loader->register();
```

Keep in mind that there is no need to use more than one autoloader on a site. If you are doing full-site development, with careful planning you can share them between multiple, purpose-specific plugins.

THAT'S ALL IT TAKES

In this chapter, I showed you how to use a class autoloader, following established PHP community standards in your WordPress plugin, theme, or even for a full site.

Whether you add your own, or use Composer's, it's an easy thing to do. It really does make life a lot easier when you use one.

Also, understanding how the standards work makes it easier to work with third-party PHP libraries.

PHP Design Patterns For WordPress Developers

That is where PHP design patterns come into play.

Software development is about repeating yourself intelligently by using functions to encapsulate repetitive code, saving you the hassle of writing it out each time. This doesn't just mean finding a repeatable pattern and going with it, it's important to find the right pattern. That is where PHP design patterns come into play.

While we often think of this in terms of choosing to write a function or class, or to import a library, this approach also extends to application architecture. The architecture of a framework, CMS, plugin, theme, class, or system is often described as conforming to a pattern.

Being aware of the classic software PHP design patterns and architectures as well as common patterns employed in WordPress can be very instrumental in helping us write better code.

EVENT DRIVEN VS. MODEL VIEW CONTROLLER

WordPress uses an event-driven architecture, in which there are hooks in the core software and plugins and themes that act as events. When WordPress encounters a hook, it executes all code "hooked" to that event.

This loosely conforms to the publisher/subscriber pattern where WordPress or a plugin or theme "publishes" an event with apply_filters() or do_action() that can be "subscribed to:" with add_filter() or add_action().

JavaScript runs in the browser using a similar event-driven approach. In JavaScript, we add event listeners to happen at specific events that are either triggered by page loading, such as window.onload() or based on user interaction with the browser such as a click event.

The event-driven architecture that WordPress and in-browser JavaScript use is fairly linear which makes it easy to understand. It can be summarized as "when this happens, Wdo these things."

The Model View Controller (MVC) pattern is not as easy to understand because it describes a real-time, circular relationship between the user and the application. There are many variations on the MVC pattern but in general, there are three parts: the view, controller, and model.

- The view or template defines the visual representation of the data, based on the current state of the model, and can change based on user input.
- The controller is the intermediary between the view and the data source or remote API. It also updates the model based on your interactions with the view and remote API.
- The model is the current set of data, defined by the controller and displayed by the view.

In general, an application, framework, or language using MVC architecture is more difficult to understand than one using event-driven architecture. Neither is "better" or "more powerful," but they are different and suited to different uses.

There are a few frameworks that have been created on top of WordPress that implement the MVC pattern. That's great when they fit a specific need, however, it is just important to keep in mind that they are MVC on top of an event-driven architecture.

USEFUL PHP DESIGN PATTERNS

WordPress's event-driven architecture by itself is not a software pattern.

WordPress's event-driven architecture by itself is not a software pattern. But it is implemented using the publisher/subscriber variant of the observer pattern. It's not always a perfect match for the textbook definition of those patterns, but it doesn't actually matter.

Learning about or using software PHP design patterns isn't about scoring points for your impressive knowledge of the subject or application of that knowledge. Instead, understanding these patterns is about helping you use them when appropriate and knowing whether or not they are the right solution to a problem.

It also helps in reading other people's code, which is important for improving your skills and helping you integrate it or debug it. Identifying a common PHP design pattern helps you make sense of the code you are looking at.

I'm not going to cover all of the software design patterns. Instead, I want to look at two formal patterns that are important for WordPress developers. I also want to talk about some patterns that are not formal, but are used a lot in WordPress.

THE SINGLETON

The singleton pattern is very common in software development and in the development of WordPress plugins — however, it can be overused in the development of WordPress plugins.

The point of the singleton pattern is to ensure that there is only one instance of the class that implements it. In PHP, by default, any class can be instantiated any number of times. For the most part, this is great as it allows for using classes to create multiple objects with the same structure that represents different data.

But what about a class that is designed to load a plugin or application? Why would you want that to run twice? You probably only want one instance of that class. Or what about a class that builds an object that holds a plugin's or application's configuration? Again, only one instance of that class makes sense.

You'll only need a single instance of the class, thus the name.

The singleton works by declaring the class constructor as private or protected. In this pattern, a private or protected static variable is used to hold the class instance. Then a public static method is used to retrieve that variable. Before doing so, it checks if that variable is null, and if so, it instantiates the class and stores the instance in that variable.

Here is an example:

```
<?php
class singleton_example {
	/**
	 * Holds class instance
	 *
	 * @access private
	 *
	 * @var singleton_example
	 */
	private static $instance;
	/**
	 * Private constructor to prevent new instances.
	 */
	private function __construct(){
		//feel free to do stuff that should only happen
once here.
	}
	/**
	 * Get class instance
	 *
	 * @return singleton_example
	 */
	public static function get_instance(){
		if( null === self::$instance ){
			self::$instance = new self();
		}
		return self::$instance;
	}
}
```

Because of the private constructor, you can not access this class using the new keyword like this:

```
$object = new singleton_example();
```

This would cause PHP to call a private method, which is illegal. Instead, you would get the single instance using the get_instance() method:

```
$object = singleton_example::get_instance();
```

Now the variable $object contains the instance of the class and you can use it to call non-static methods of that class.

Now the variable $object contains the instance of the class and you can use it to call non-static methods of that class.

The above example is fairly typical for WordPress since it supports PHP 5.2. In some cases, you may wish to subclass a class that implements the singleton pattern. There are not a lot of cases where you would want to do this, but you can using late static bindings:

A singleton often makes sense for the class of a plugin that is responsible for loading a plugin.

```php
<?php
class singleton_example_two {
	/**
	 * Holds class instance
	 *
	 * @access protected
	 *
	 * @var singleton_example_two
	 */
	protected static $instance;
	/**
	 * Protected constructor to prevent new instances.
	 */
	protected function __construct(){
		//feel free to do stuff that should only happen
once here.
	}
	/**
	 * Get class instance
	 *
	 * @return singleton_example
	 */
	public static function get_instance(){
		if( null === static::$instance ){
			static::$instance = new static();
		}
		return static::$instance;
	}
}
```

This example is similar but uses protected instance variable. It also replaces the self keyword, with the static keyword to ensure that PHP is referring to the current instance, not the parent instance when subclassing.

Keep in mind that singletons have a lot downsides and are overused in WordPress. That doesn't make them bad, but be careful. A singleton often makes sense for the class of a plugin that is responsible for loading a plugin.

Your plugin may need a single instance of a class, but others integrating with your plugin or add-ons for your plugin may need their own instances.

Often times, WordPress plugins have one main class with a singleton that adds an instance of many other classes as properties of it. This allows them to avoid multiple singletons, while having the benefit of easy access to a specific instance of specific classes.

Another case where singletons get misused are classes that have no functional reason to be instantiated multiple times, but there is no harm in doing so. Often times, these are classes that don't really represent a data object, but are a collection of related functions. Declaring all methods of these classes static can simplify things dramatically.

THE FACTORY PATTERN

The factory pattern is not used a lot in WordPress, but it is worth understanding as it makes complex systems that involve many separate classes explicit. A class implementing the factory pattern is designed to construct an object or an object of other classes. That's why we call it a factory, it creates other objects.

Here is a simple example of a factory pattern that shows how it works. This basic factory is used to construct instances of the WP_Post class via a variety of means:

```
class post_factory{
     /**
      * @var WP_Post
      */
     protected $post;

     public function __construct( $post = null ) {
            if( is_a( $post, 'WP_Post' ) ){
                   $this->post = $post;
            }elseif ( is_numeric( $post ) || is_a( $post,
'stdClass' ) ){
                   $_post = get_post( $post );
                   if( is_object( $_post ) ) {
                          $this->post = $_post;
                   }
            }else{
                   $this->post = get_post();
            }
     }

     public function get_post(){
            return $this->post;
     }
}
```

This is a very simple example, and might be an overly complicated way of ensuring a variable is really a WP_Post. That said, it limits having to repeat validation code all over the place while safely ensuring a WP_Post object, a post ID, or WP_Query args really representa WP_Post object.

Again, this is way too simple to be really useful, and shows a downside of the factory pattern. The extra overhead might not be worth it. But, as in this example, it can be used to introduce validation of inputs, which may be worth it. Carl Alexander has an excellent article on designing a class to manage WordPress posts that is worth reading when thinking about a factory like this.

Plugins often have similar add-ons that use the same hooks and employ a lot of repetitive code.

There are many cases where a factory might be worth using and is a great way to build a system that creates multiple parts of the application from a single configuration.

For example, when we add a REST API endpoint or set of routes, we normally define the fields for those endpoints in the class that handles the route those endpoints are a part of. But what if we needed that configuration elsewhere? What if we wanted to inject that configuration into an endpoint generator as well as other generators?

That's when we would use a factory or other factories. I recently published a WordPress REST API endpoint generator that implements a factory pattern. While it is useful on its own, I am also working on a library to consume the same configuration but will create a backbone-generated interface that uses those endpoints. Since the configuration is loosely tied to the code, I hope to make it adaptable to generating the UI in Angular, and also adopting the Fields API spec once it is finalized.

Another place where factories could be useful is plugin add-ons. Plugins often have similar add-ons that use the same hooks and employ a lot of repetitive code. One of the uses for a factory pattern would be to automatically setup all of those hooks.

WORDPRESS CORE GLOBALS

One reason to use a singleton PHP design pattern is that it avoids having to use a global variable to store an instance of a class. A singleton introduces a state that is like a global into an application, without using a global variable.

PHP global variables are completely mutable. They can be unset, or redefined at any time. We all agree not to ever change the global $post variable in WordPress to an array, but we can. There is nothing illegal about changing the global $post to a random array or unsetting its post_title property. The next time it is accessed, that's when the problems will manifest, possibly as a fatal error.

WordPress core doesn't use singletons. I can't say for sure, but it is likely an artifact of WordPress and its predecessor b2/cafelog originating before static properties were introduced in PHP.

Instead, WordPress puts many class instances into a global variable. These classes tend to have a public constructor. In the case of WP_Rewrite, which is strongly tied to the saved rewrite rules, there is probably no chance you would ever want an instance of WP_Rewrite other than the one in the global $wp_rewrite.

If WordPress were written today, WP_Rewrite would probably implement the singleton pattern. Instead of:

```
global $wp_rewrite;
$wp_rewrite->flush_rules();
```

We would probably use:

```
WP_Rewrite::get_instance()->flush_rules();
```

But, that might be an over use of the singleton PHP design pattern, maybe there would be one main WordPress class, that had a singleton and held an instance of WP_Rewrite, WPDB, WP_REST_Server, etc.

The global $wp_rewrite is pure technical debt that occurs when a project is over 10 years old and is committed to not break backward-compatibility. And let me be clear, being more "technically correct" is not, in my mind, worth breaking almost every WordPress site.

The use of globals in WordPress for classes that have a "main instance" but can be reused makes a lot of sense. For example, there is a global $wp_query object that holds the main query based on the current request.

This allows us to call the instance of WP_Query for the current request, and also make our own instances of WP_Query for other queries. If WP_Query used a singleton, we would have to write our own SQL queries whenever we needed to do multiple queries for posts in the same request, which would be terrible.

Yes, there are better ways to store that main query if your framework isn't over 10 years old. My point isn't to put down WordPress. My point is to identify a pattern we should look at because it helps us understand WordPress. Also, it helps us see how we could do better in our own work.

WHY?

Again the point of this isn't to impress yourself or your friends with the ability to identify and implement common PHP design patterns.

No end user is going to go to a WordPress site and be impressed by the sparing, yet sensible, use of the singleton pattern, or the implementation of an MVC framework somewhere in the stack.

But, if learning to identify these patterns can help you better understand what's going on in other people's code you have to work with, or helps you write better code or better conceptualize the system you are working in, then you should be able to advance as a developer.

It's Time To Level Up Your PHP Skills

A few years ago I was working on a WordPress-powered web application that used several custom post types, each of which required a different class to query it. The classes included a lot of repetitive code because the queries and a lot of other things I was doing were different.

Somewhere during this project, I noticed that the pattern I established and cut and pasted across four different classes was very wrong. I recently came across my first Carl Alexander article on polymorphism in PHP, which introduced me the concept of abstract classes in Object-Oriented PHP.

I re-read the article a few times and put it to use. I fixed the bad pattern, and, instead of repeating it four times, I made an abstract class and extended it four times. The next time I needed to make a change to that code, I made the change once — and the other four classes inherited it. We call this leveling up.

It was not the first time I leveled up thanks to Carl. Since then I have recommended a lot of Carl's article to other developers and have leveled up several more times.

There has been a lot of hype recently about getting a deeper understanding of JavaScript. If you ignore it, however, you will remember that WordPress is primarily a PHP application. Constantly leveling up our PHP skills is essential to pushing forward the state of WordPress development and improving our skills as WordPress developers.

I had the pleasure of meeting Carl in person at WordCamp US, where we talked about PHP and WordPress development. I recently spoke to him again about ways that we as WordPress developers can improve our PHP chops. This chapter highlights some of the takeaways from our conversation.

READ MORE SOURCE

To learn how to better utilize the WordPress REST API without any documentation.

I've long been an advocate of learning by reading source code from other developers. To learn how to better utilize the WordPress REST API without any documentation, I read the source.

I didn't just do it to learn how to do something. I did it because the WordPress REST API is really well written and I wanted to learn by studying how they made it work.

Carl agreed. He told me that reading other's source code is *"the secret sauce for improving your coding skills."* He also gave me some really helpful questions that we should ask ourselves when reading other developer's code to make sure we get the most out of it:

- What problem are they trying to solve?
- What are they doing to solve it?
- What do I need to do to make their solution work for me?

This type of thinking will improve your skills as a developer because you're not just copying and pasting code. You're actually emulating the developer who wrote it by putting yourself in their shoes and thinking like them.

MISUSED PATTERNS

One of the reasons we read code is to learn patterns and to see how they get used in real life.

Software design is about patterns. We constantly re-use common patterns to accomplish specific goals. One of the reasons we read code is to learn patterns and to see how they get used in real life. That said, there are two patterns commonly used in WordPress that Carl feels are misused.

These two patterns are the singleton pattern and the pattern of putting hooks in the class constructor. These two issues are related.

As I discussed in chapter 13, the singleton is a design pattern that is used to ensure that there is only one instance of a class. It is helpful when a class is only used in an application where there is one single object of that class that all other classes and functions share.

The singleton pattern is generally accomplished using a private constructor and a static method to get a class instance. For example:

```
<?php
class something {

	private static $instance;

	private function __construct(){
		//do stuff
	}

	public function instance(){
		if( is_null( self::$instance ) ) {
			self::$instance = new self();
		}
		return self::$instance;
	}
}
```

Since the construct method of this class is private, instantiating the class directly will cause an error. We instead want to use the public static method to call any other methods, therefore always working on the same single instance of this class.

This is great when used properly. But according to Carl, and I agree, it's overused.

"The main reason [the singleton is] popular with WordPress developers is because of the plugin API," Carl Said. "It's hard to use object-oriented programming with it. To solve that problem, they tend to put all their hooks inside the class constructor. But this creates a new problem.

Now, all those hooks get registered whenever there's a new instance of that class. At that point, they feel compelled to use the singleton pattern to lock down the constructor.

But the real solution isn't the singleton pattern. It's to not register those hooks inside the constructor."

I would add that this is often a symptom of classes that have too many responsibilities. Carl has a great article on the single responsibility principle. Similarly, Tom McFarlin's article on the same subject had a huge impact on me and led to me writing smaller classes with a lot less going on in each.

COMPOSER AND CROSS-POLLINATION

Speaking of small reusable classes: Composer and relying on an autoloader, which I use Composer for as well, has enabled me to write more reusable code. Carl sees Composer as not just a software tool, but an enabler for collaboration across projects, languages, and more.

"The PHP community has embraced that idea a lot over the last few years," Carl said. "I can't give enough credit to Composer for helping share code between PHP projects. That said, the package manager, the idea behind Composer, isn't new. It's just a cross-pollinated idea from other languages like Python and Ruby.

You learn a lot as a developer by looking at how others solve problems. This is something I wish the WordPress community promoted more. We don't spend enough time outside our own borders. That's the biggest difference I see between WordPress developers and other developers."

If you reading PHP from non-WordPress developers, you will quickly notice that they are less likely to be concerned about supporting out of date versions of PHP.

One major advance in PHP since 2009 is anonymous functions.

"On the surface, anonymous functions might not look like much to you," Carl said. "But there are a lot of cool things that you can do with them. In some situations, they can even make you rethink how you write your code.

That's because they act as a disposable functions. You can use them once or even pass them around in a variable. But, like a variable, they don't exists beyond the scope where you defined them.

This lets you do things that were a lot harder or impossible to do before. It also lets you cleanup your code by removing functions that you used only once. Instead, you pass anonymous functions as your callbacks."

If there is a writer who has helped you level up your PHP skills please let us know.

YOUR TURN

You can learn a lot from reading tutorials and articles from developers like Carl, Tom McFarlin, Micah Wood, and others. If you're not already reading Carl's posts I encourage you to sign up for his mailing list and read his back catalog.

If there is a writer who has helped you level up your PHP skills please let us know. While I'm going to keep reading everything Carl writes in 2016, I'm also going to make it a point to heed his advice to read more non-WordPress PHP code and seek out articles written about PHP by non-WordPress developers. I hope you will too.

CONCLUSION

PHP is here to stay. Not only is it the most popular programming language, currently powering 84 percent of all websites, but it also the language used by the world's most popular CMS — WordPress. And, as WordPress continues to grow in popularity (it now dominates more than 25 percent of the internet), PHP becomes increasingly more important to add to your dev toolkit.

Moreover, recent strides in the evolution of PHP, as seen in PHP 7, provide revolutionary updates to the programming language, which dramatically improve overall site performance and page-load speed. PHP7 is not just faster, it introduces new syntaxes and new tools that make it a better tool for the job.

Object-Oriented PHP utilizes classes to organize the data and structure of an application, which makes code more flexible and less redundant. The organization of data and structure created with object-oriented PHP, enables better collaboration between WordPress developers, which is essential in open source. This enables WordPress to be the flexible and extendible CMS it is today.

The examples and insight provided in this ebook should equip you with the skills to get started with object-oriented PHP in WordPress. I encourage you to use this ebook as a springboard into more in-depth development projects in the future.

Happy developing!

ABOUT THE AUTHOR: JOSH POLLOCK

Josh is a WordPress developer and educator. He is Founder/ Lead Developer/ Space Astronaut Grade 3 for Caldera Labs, makers of awesome WordPress tools including Caldera Forms — a drag and drop, responsive WordPress form builder.

Also, he is a WordPress core contributor, the author of The Ultimate Guide To The WordPress REST API and a member of The WPCrowd.

About WP Engine

WP Engine powers amazing digital experiences for websites and applications built on WordPress. The company's premium managed hosting platform provides the performance, reliability and security required by the biggest brands in the world, while remaining affordable and intuitive enough for smaller businesses and individuals. Companies of all sizes rely on WP Engine's award-winning customer service team to quickly solve technical problems and create a world-class customer experience. Founded in 2010, WP Engine is headquartered in Austin, Texas and has offices in Limerick, Ireland; San Francisco, California, San Antonio, Texas, and London, England.

About Torque

Torque is a news site featuring all things WordPress. We are dedicated to informing new and advanced WordPress professionals, users, and enthusiasts about the industry. Torque focuses primarily on WordPress News, Business, and Development, but also covers topics relating to open source and breakthrough technology. Torque made its debut in July 2013, at WordCamp San Francisco, and has since produced valuable content that reflects the evolution of WordPress, both as a platform and a community. Torque is a WP Engine publication, though maintains complete editorial independence.