***A Complete Guide to PHP Namespaces***

[OCTOBER 9, 2018](https://www.thoughtfulcode.com/a-complete-guide-to-php-namespaces/)[DAVID](https://www.thoughtfulcode.com/author/d_admin/)[12 COMMENTS](https://www.thoughtfulcode.com/a-complete-guide-to-php-namespaces/#comments)

Namespaces are a great feature of PHP. They’ve taken it from being a complex-to-scale and rather small-scale-only language into something where most projects have a healthy-level of outside code (pulled in with the [Composer dependency manager](https://getcomposer.org/)) and a whole lot more clarity about what is where and why.

While namespacing code is widely used, some places in the PHP ecosystem, most obviously my beloved WordPress, are slow to adopt it. So our goal here is to welcome all comers. If you mostly understand namespaces, feel free to skip around, I bet you’ll learn something. If you’re mostly in the dark about them, expect to read linearly, I put a lot of thought into the sequence of things here.

**Why PHP Namespaces**



In short, PHP namespaces address the problem of isolation. Before they were introduced to PHP, if you had a function, class, or constant called, say, Validation\_Error in one part of your code, you simply couldn’t have another one like that anywhere else. This isn’t the end of the world — just name yours something like DBH\_Validation\_Error. More importantly though, you also wouldn’t know who a given Validation\_Error class was coming from. The name is generic, and the runtime doesn’t really have extra insight about it for you.

**What Namespaces Give Your Code**

In short, namespaces give you unique prefixes for your code. You pull all your code inside of a DBH\Access namespaces that’s about your company’s access systems. You put other code for an open source project in a PHP Namespace of CFFC\Access. And if you want, you can even use both of those bits of your code in some third file without problems.

Analogically, you’ll often see people refer to and think of namespaces as “folders” or “directories.” This analogy is not enforced by PHP, but most people do follow a variation on this convention for the sake of programmer sanity. It’s a sad coder who discovers that her coworker has made a \DBH\AccessControl\CredentialNotFound exception, but located it at a path like inc/random/whatever/file.php.

**Poor Alternatives to Namespaces I’ve Seen (People Use)**

In WordPress, for reasons of compatibility with version of PHP without namespaces (aka PHP 5.2), you will see a namespace-like convention which is common. I alluded to it above. I often call it the “poor person’s namespace.” It’s just a convention of prefixing each and every function or class name with your custom preable. So rather than my class being at DBH\Access, I just make a DBH\_Access class.

I also sometimes see people use PHP static classes (which [I recently explored at some length over on WPShout](https://wpshout.com/php-static-methods-in-depth/)) for a similar role for functions they write. So again rather than DBH\Access\login(), I’ll locate a function at DBHAccess::login(). There are many arguments against static classes that I won’t go into here. Mostly I will simply say that *PHP has namespaces, and they’re a better way to do that*.

**PHP Has Had Namespaces Since 5.3**



In general, unless you’re being like WordPress (with much love in my heart, I’ll say: don’t be like WordPress), you can take it for granted that any “modern” PHP system you’ll find your way to will work fine and support namespaces. Why? Because the basic features of namespaces were added to PHP with 5.3, which came out all the way back in 2009.

There are a few things that changed later. In 5.6 we got the ability to say use about functions (more on that later), and with 7.0 we got the ability to more flexibly and concisely include multiple different files in a namespace.

**How PHP Namespaces Work**

In short, a namespaces in PHP is placed at the top of a file, and designates that all the code in that file will be placed in a namespace. As we’ll go into in a bit more detail, this namespaces affects classes, interfaces, functions, and constants. It does no affect variables.

It is quite important to realize that the declaration of a namespace happens at the start of the file, and before (almost) anything else. What does that declaration look like? Generally,

<?php

namespace DBH\Access;

Copy

It is not required that you go multiple levels deep in the declaration. This is just as valid:

<?php

namespace MyName;

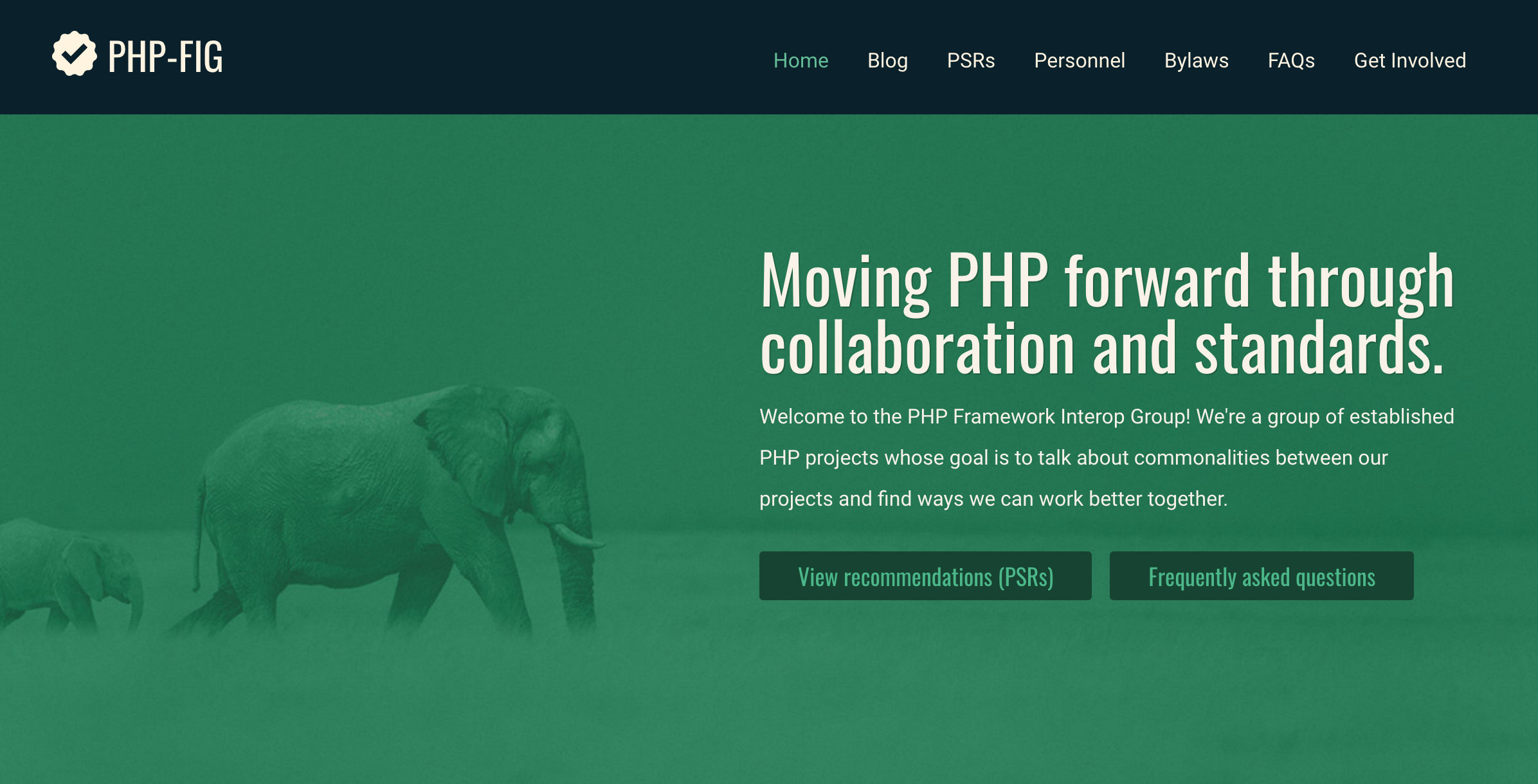
Copy

Then you just declare your functions, classes, and consts as normal. They will just implicitly be within that namespace.

**They’re Sort of Like Folders, But Not**

Because of how cleanly these look like directories (simply having a backslash (\), not a forward slash / which we have as directory separators on most computers), you might think that a namespaces implies something about the location of the file. But as I said above, that’s not strictly required, just a good and common practice. Most often that’s done in-line with PSR-4 or PSR-0, but that’s what you’ll read in the next section.

**What PSR-4 (and PSR-0) Say About PHP Namespaces and Folders**



While it’s not a requirement, the closest thing that PHP has to a “code standards” body is group called [the PHP-FIG](https://www.php-fig.org/) (the PHP Framework Interoperability Group). They create standards, which go by the name of PSR (PHP Standards Recommendations). As you might guess, there’s some controversy about any effort to standardize code. But in general, it’s accepted and common in PHP that people align their code to these standards when it makes sense.

The first PHP-FIG PSR, now retired, was called PSR-0 and recommended a folder structure to accompany your namespaces. That was retired a few years ago, and a standard called PSR-4 replaced it. You’ll sometimes in a search or conversation hear references to either of these, and they are similar-enough that you can treat them the same.

Essentially, [the PSR-4 standard](https://www.php-fig.org/psr/psr-4/) specifies that you’ll match the capitalization of namespaces and subnamespaces, and that they’ll map to directories. When I examined it I was actually a little surprised at how un-prescriptive the standard is. As long as your path is consistent, and you define a “base directory” for your namespace, it just requires that your file name ends in .php and has a letter case match to the file inside.

**Writing a Class in a Namespace**

So to comply with PSR-4, you’ll generally see a code whose “fully qualified class name” is something like \Symfony\Component\HttpFoundation\Request at a location like ./vendor/Symfony/Component/HttpFoundation/Request.php. And you’ll expect (and find) at the top of that file (approximately the following):

<?php

namespace Symfony\Component\HttpFoundation;

// skip use statements, more on those later

class Request

{

// body of class

}

Copy

And with that set-up, you’ll be writing the class. Writing classes inside namespaces is pretty much like normal. The only big difference you’ll want to be aware of is that referring to code from outside of your namespace generally takes one more character, a backslash. (More on that at “How to Work with Code from Other Namespaces.”)

**Functions and Constants Can Also Be in Namespaces**

Similar to classes, the logic of writing constants and function inside namespaces in PHP is the same. Code would look like this:

<?php

namespace DBH\Helpers;

function convert\_to\_kelvin() {}

const MYCONST = 1;

Copy

The one extra wrinkle about functions and constants is that on PHP below 5.6, you can’t use the use statements we’ll describe below. This isn’t true for PHP 7, or even 5.6, but again if you’re in a WordPress-like world of wide compatibility it matters. [More details on the wrinkle can be found on PHP.net.](http://php.net/manual/en/language.namespaces.faq.php#language.namespaces.faq.nofuncconstantuse)

**How to Work with Code From Other Namespaces**

I mentioned this already, but from within a namespace, you must always refer to other classes, constants, functions, etc with their “fully qualified names.” In PHP, these “fully qualified” names will *always* start with a leading backslash.

So if code from your personal namespace wants to refer to the Symfony Request class from above (say inside of a class method), you’ll refer to it as \Symfony\Component\HttpFoundation\Request. If you weren’t inside of any namespace, then Symfony\Component\HttpFoundation\Request would have worked.

The need to lead with a backslash is also true for built-in PHP classes. So to make a new PHP DateTime object, you’ll need to say new \DateTime. This is literally an extra key-stroke, so it’s not a huge problem. But it is a common stumbling block.

**How to Use use to Import Code in PHP**

Fully qualified names can sometimes be pretty long winded, like in the four-layer Symfony one above. You have another option in PHP to deal with that: the use keyword. That’ll generally look like this:

<?php

namespace DBH\Helpers;

use \Symfony\Component\HttpFoundation\Request;

class WebFramework extends Request {}

Copy

What happens by default is that PHP will pull in the fully qualified class name so that you can use it within the current file/namespace as simply the class name. (Request in this case.) You can also alias these imports, by adding an as at the end. When you do that, the phrase you give there will be how you access it in the file. To be very specific, that means you could rewrite the above as:

<?php

namespace DBH\Helpers;

use \Symfony\Component\HttpFoundation\Request as SymfonyRequest;

class WebFramework extends SymfonyRequest {}

Copy

It’s a little more to write. But it’s especially useful to know if you find yourself wanting to, say, make your own Request class. The final possible way to rewrite the above, just to be completely clear is to use the fully qualified name without the use at all. That makes it:

<?php

namespace DBH\Helpers;

class WebFramework extends \Symfony\Component\HttpFoundation\Request {}

Copy

**How to Import Multiple Different Classes from a Package in PHP**

As you start to use many classes deep down a dependency chain, you might be using multiples from the same vendor. In that situation, it can be annoying to have one line for each “import” statement, like:

<?php

namespace DBH\Helpers;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

Copy

To deal with this, PHP 7+ supports the following:

<?php

namespace DBH\Helpers;

use Symfony\Component\HttpFoundation\{Request, Response};

Copy

This syntax isn’t as concise as a use Symfony\Component\HttpFoundation\\* which on might expect from other languages, but it’s definitely more concise than the multi-line version. And it makes it harder to really import more than you need.

**Now You Know PHP Namespaces**

Hopefully you feel a little more informed about PHP namespaces now. They’re a super powerful system, and are at the heart of the usefulness of Composer and the public directory of Composer Packages called [Packagist](https://packagist.org/). They’re both super good to understand in depth, but different enough from namespaces that I’ll explore them another time.

If anything is unclear, please leave a comment below. Happy hacking