



Autoloading & Namespaces



Autoloading and Namespaces

Jonathan Daggerhart

- **Developer at Hook 42**
- **Organizer for Drupal Camp Asheville**



Drupal.org: daggerhart

Twitter: @daggerhart

Blog: <https://www.daggerhart.com>

Drupal Camp Asheville

Site: <https://drupalasheville.com>

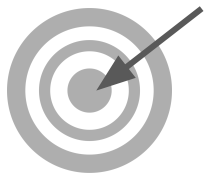
Twitter: @drupalasheville





Autoloading and Namespaces

What we will cover



1. **Autoloading** - configuring PHP to automatically load classes as they are instantiated
2. **Namespaces** - encapsulating a set of values and methods, or a class
3. **PSR-4** - the PHP standard for autoloading that leverages namespaces
4. **Composer Autoloading** - including the composer generated autoloader



Autoloading and Namespaces

What is autoloading & what problem does it solve?

Autoloading is a way to have PHP automatically include the PHP class files of a project.

The Problem:

Consider an OOP PHP project that has more hundreds of PHP classes. How might we make sure that all our classes are loaded before using them?

```
<?php
require_once "includes/BearAbstract.php";
require_once "includes/BlackBear.php";
require_once "includes/BrownBear.php";
require_once "includes/KodakBear.php";
// etc... for another hundred bears

$bear = new BlackBear();
$bear->roar();
```



Autoloading and Namespaces

Autoloading - The Solution

What if we could have PHP automatically load class files when we need it? We can, we only need to two things:

The Solution:

1. Write a function that looks for files based on a given Class name

```
function my_custom_autoloader( $class ){  
    $file = __DIR__ . '/includes/' . $class . '.php';  
    if ( file_exists($file) ){  
        require $file;  
    }  
}
```

2. Register that function with the PHP core `spl_autoload_register()` function

```
spl_autoload_register( 'my_custom_autoloader' );
```



Autoloading and Namespaces

Autoloading - The Results

With a working registered autoloader, PHP will now automatically include a class file as we instantiate the class.

How it works: assume we attempt to instantiate a new Bear object

```
$bear = new BlackBear();
```

1. PHP will run our autoload function: `my_custom_autoloader()`
2. Our autoloader will look for the file: `./includes/BlackBear.php`
3. If found, our autoloader will `require_once()` that file
4. PHP will instantiate the object, and continue



Autoloading and Namespaces

Autoloading - All together

1. Autoloader function

2. Register new function

3. Instantiate object, without explicitly requiring the class file:

`includes/BlackBear.php`

```
function my_custom_autoloader( $class ){
    $file = __DIR__ . '/includes/' . $class . '.php';
    if ( file_exists($file) ){
        require $file;
    }
}

spl_autoload_register( 'my_custom_autoloader' );

$bear = new BlackBear();
$bear->roar();
```



Autoloading and Namespaces

What are Namespaces?

Namespaces are a way to encapsulate items. A very easy (and somewhat practical) way of thinking of this is like an operating system's directory structure, in that folders “encapsulate” the files and folders within them.

quote php.org:

As a concrete example, the file `foo.txt` can exist in both directory `/home/greg` and in `/home/other`, but two copies of `foo.txt` cannot co-exist in the same directory. In addition, to access the `foo.txt` file outside of the `/home/greg` directory, we must prepend the directory name to the file name using the directory separator to get `/home/greg/foo.txt`.



Autoloading and Namespaces

Namespaces - How to

To “namespace” our class, we must use the `namespace` keyword at the top of our PHP file.

“Namespace”d class:

```
namespace Example;  
  
class Object {  
    function say( $text ){  
        echo $text;  
    }  
}
```

Accessing namespace for instantiation:

```
$example = new \Example\Object();  
$example->say('hello');
```



Autoloading and Namespaces


Namespaces - What problem do they solve?

To answer this question, we need to look back in time to a PHP without namespaces. Previous to PHP version 5.3, we could not encapsulate our classes, therefore they were always at risk of conflicting with another class of the same name.

It was (and still is to some degree) not uncommon to prefix class names resulting in something more like this:

```
class Some_Really_Long_ClassName_ToAvoid_Conflicts {}  
$example = new Some_Really_Long_ClassName_ToAvoid_Conflicts();
```

That's a lot.
No one wants
to write that
much code.





Autoloading and Namespaces

Namespaces - Solving a problem...

But given the previous example, we could change all those prefixes into namespaces and end up with a class defined like this:

```
namespace Some\Really\Long\ClassName\ToAvoid;  
class Conflicts {}
```

And we would instantiate it using its namespace:

```
$example = new \Some\Really\Long\ClassName\ToAvoid\Conflicts();
```

Hmm... there must be more to it than that.

But wait, is that actually better? Looks very similar in length to--

```
$example = new Some_Really_Long_ClassName_ToAvoid_Conflicts();
```



Autoloading and Namespaces

Namespaces - the “use” keyword

The `use` keyword in PHP “imports” the given namespace into the scope of the current file.

quote php.org: This is similar to the ability of unix-based file systems to create symbolic links to a file or to a directory.

Import
namespace
into scope

Example:

```
use Some\Really\Long\ClassName\ToAvoid\Conflicts;  
  
$example = new Conflicts();
```

Use the class
without global
name



Autoloading and Namespaces

Namespaces - Syntax

Still working with the “file folders” analogy, namespaces can have both relative and absolute paths. Like a file system, to access a namespaced class from anywhere (global / absolute), we must prefix it with a backslash:

```
$absolute = new \Some\Really\Long\ClassName\ToAvoid\Conflicts;
```

Alternatively when we import a namespace with the `use` keyword, we do not add a prefix slash when instantiating it:

No prefix slash

```
use Some\Really\Long\ClassName\ToAvoid\Conflicts;  
$relative = new Conflicts();
```



Autoloading and Namespaces

Namespaces - Extended example

Import the desired namespaces

```
use MyApp\TemplateEngine\Loader;  
use MyApp\TemplateEngine\Render;
```

Both `Loader()`
and `Render()`
are accessed relative
to the imported
namespaces

```
$template = new Loader('path/to/template/file');  
$render = new Render( $template );  
$render->output();
```

Awesome! Now that we know autoloading and namespaces, next we'll put them together to form ~~Voltron~~ PSR-4!



Autoloading and Namespaces

PHP Standard Recommendation (PSR) 4

PHP Standard Recommendation 4 (PSR-4) is a commonly used pattern for organizing a PHP project so that the namespace for a class matches the relative file path to the file of that class.

For example, if we are working within a project that makes use of PSR-4 and we are dealing with a namespaced class like this:

```
\MyApp\TemplateEngine\Loader();
```

We can be sure that the file for that class can be found in this relative location within the file system:

```
<relative root>/MyApp/TemplateEngine/Loader.php
```



Autoloading and Namespaces

PSR-4 - How does it work?

PSR-4 leverages the two techniques we've mentioned, autoloading and namespaces. In fact, the autoloader implementation can be pretty simple to start.

Autoloader:

Replace
backslashes in
class namespace
with forward slashes
(like a file system)

```
function my_simple_psr4_autoloader( $class ){  
    $class_path = str_replace('\\', '/', $class);  
    $file = __DIR__ . '/src/' . $class_path . '.php';  
    if (file_exists($file)) {  
        require $file;  
    }  
}  
  
spl_autoload_register( 'my_simple_psr4_autoloader' );
```




Autoloading and Namespaces

PSR-4 - Autoloader walkthrough

```
use TemplateEngine\Loader;  
$template = new Loader('path/to/template/file');
```

```
function my_simple_psr4_autoloader( $class ){  
    $class_path = str_replace('\\', '/', $class);  
    $file = __DIR__ . '/src/' . $class_path . '.php';  
    if (file_exists($file)) {  
        require $file;  
    }  
}  
  
spl_autoload_register( 'my_simple_psr4_autoloader' );
```

1. `TemplateEngine\Loader()` instantiated
2. Absolute class name (including namespace) is passed into the autoloader `TemplateEngine\Loader`
3. Autoloader converts backslashes to forward slashes `TemplateEngine/Loader`
4. Look for a file in roughly that location, and include it if found `src/TemplateEngine/Loader.php`



Autoloading and Namespaces

Composer - PHP Package manager

Composer is a command line PHP package manager. You may have seen a project before with a `composer.json` file in its root directory. This file tells Composer about our project, including our project's dependencies.



Simple `composer.json` example:

```
{
  "name": "dcav1/example",
  "description": "This is an example composer.json file",
  "require": {
    "twig/twig": "^1.24"
  }
}
```



Autoloading and Namespaces

Composer - Dependency Management

composer.json

```
{
  "name": "dcavl/example",
  "description": "An example",
  "require": {
    "twig/twig": "^1.24",
    "guzzlehttp/guzzle": "^6.2"
  }
}
```

We can add new dependencies to our project with the following command:

```
$ composer require <vendor name>/<package>
```

Example: Twig

```
$ composer require twig/twig
```

Example: Guzzle

```
$ composer require guzzlehttp/guzzle
```

`composer require` will automatically update our composer.json file.



Autoloading and Namespaces

Composer Autoloading

Yes, composer will generate an autoloader for our project dependencies, and place an “autoload.php” file in the root of the “vendor” folder. Simply include that file, and we’re ready to go:

1. Include generated autoloader
2. Begin using classes immediately, without including any more files

```
require_once "vendor/autoload.php";

$loader = new Twig_Loader_Filesystem('/path/to/templates');
$twig = new Twig_Environment($loader, array(
    'cache' => '/path/to/compilation_cache',
));

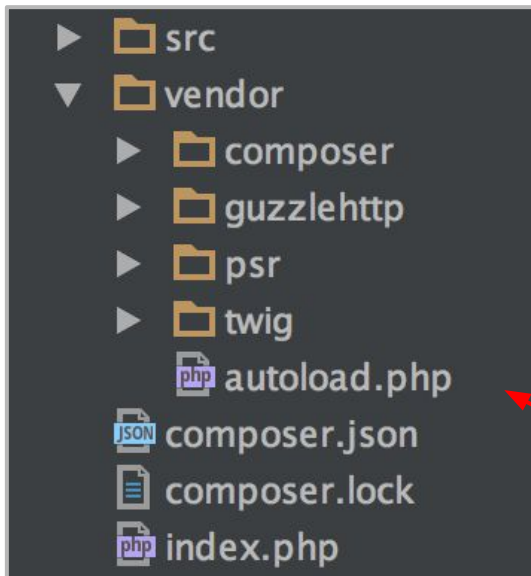
$template = $twig->loadTemplate('index.html');
echo $template->render(array('variables' => 'here'));
```



Autoloading and Namespaces

Composer require - results

Project files:



Note:

- All dependencies will be placed in the “vendor” folder of our project.
- If one of our project dependencies has its own dependencies, composer will bring those in as well.
- Composer will generate a PSR-4 autoloader for our dependencies.

“Generate an autoloader” you say. That seems relevant. Hmm....



Autoloading and Namespaces

What this means: PSR-4 & Drupal 8

Now that we understand what is going on behind the scenes with autoloading, namespaces, and composer, we are ready to find anything we need within Drupal 8. There are just a few things to keep in mind.

- Drupal **core**'s base directory for autoloading is the `/core/lib/Drupal` folder
- Drupal **core** modules use the following base directory pattern:
`/core/modules/<module name>/src/`
- Drupal **core**'s dependencies use the standard composer base directory:
`/vendor/` (outside of the `/core` folder)
- Site modules use the following base directory pattern:
`/modules/<module name>/src/`



Autoloading and Namespaces

Drupal Core Classes & Where They Live

Let's look at a few examples of finding classes within Drupal core based on their namespace:

Drupal Namespaced Class	Class File Location
<code>\Drupal\Core\Block\BlockBase</code>	<code>core/lib/Drupal/Block/BlockBase.php</code>
<code>\Drupal</code>	<code>core/lib/Drupal/Drupal.php</code>
<code>\Drupal\Core\Field\FormatterBase</code>	<code>core/lib/Drupal/Field/FormatterBase.php</code>
<code>\Drupal\rest\Plugin\ResourceBase</code>	<code>core/modules/rest/src/Plugin/ResourceBase.php</code>



Autoloading and Namespaces

Drupal Module Autoloading

Drupal core assists the autoloader in finding classes for modules during the registry build. Module classes are expected to be within a directory named “src” within the module folder.

Behind the scenes

Drupal informs the autoloader where to look for module namespaces.

```
* @return string[]
*   Array where each key is a module namespace like 'Drupal\system', and each
*   value is the PSR-4 base directory associated with the module namespace.
*/
protected function getModuleNamespacesPsr4($module_file_names) {
    $namespaces = [];
    foreach ($module_file_names as $module => $filename) {
        $namespaces["Drupal\\" . $module] = dirname($filename) . '/src';
    }
    return $namespaces;
}
```




Autoloading and Namespaces

Drupal Module Class Namespaces

The autoloader expects classes provided by modules to be placed within a folder named “src” within the module, and their namespaces begin with “`Drupal\<module_name>`”.

<code>\Drupal\<module_name>\(dir)\Class</code>	<code>(core/)modules/<module_name>/src/(dir)\Class.php</code>
<code>\Drupal\block\Entity\Block</code>	<code>core/modules/block/src/Entity/Block.php</code>
<code>\Drupal\node\Form\NodeDeleteForm</code>	<code>core/modules/node/src/Form/NodeDeleteForm.php</code>
<code>\Drupal\flag\Entity\Flag</code>	<code>modules/contrib/flag/src/Entity/Flag.php</code>
<code>\Drupal\key\Form\KeyEditForm</code>	<code>modules/contrib/key/src/Form/KeyEditForm.php</code>