PHP Composer Autoload

Summary: in this tutorial, you'll learn how to use Composer to autoload PHP classes from files using PSR-4 standard.

Loading classes using the require_once construct

First, create the following directory structure with files:

```
.
├── app
│   ├── bootstrap.php
│   └── models
│       └── User.php
└── index.php
```

The `User.php` file in the `models` folder holds the `User` class (https://www.phptutorial.net/php-oop/php-objects/):

```php
<?php

class User
{
    private $username;

    private $password;

    public function __construct($username, $password)
    {
        $this->username = $username;
        $this->password = password_hash($password);
```

```
    }

    public function getUsername(): string
    {
        return $this->username;
    }
}
```

The `User` is a simple class. It has two properties `$username` and `$password`. The constructor (https://www.phptutorial.net/php-oop/php-constructors/) initializes the properties from its arguments. Also, it uses the `password_hash()` function to hash the `$password`.

The `bootstrap.php` file uses the `require_once` (https://www.phptutorial.net/php-tutorial/php-require/) construct to load the `User` class from the `User.php` file in the `models` folder:

```
<?php

require_once 'models/User.php';
```

When you have more classes in the `models` folder, you can add more `require_once` statement to the `bootstrap.php` file to load those classes.

The `index.php` file loads the `bootstrap.php` file and uses the `User` class:

```
<?php

require './app/bootstrap.php';

$user = new User('admin', '$ecurePa$$w0rd1');
```

This application structure works well if you have a small number of classes. However, when the application has a large number of classes, the `require_once` doesn't scale well. In this case, you can use the `spl_autoload_register()` (https://www.phptutorial.net/php-oop/php-autoloading-class-files/) function to automatically loads the classes from their files (https://www.phptutorial.net/php-oop/php-autoloading-class-files/).

The problem with the `spl_autoload_register()` function is that you have to implement the autoloader functions by yourself. And your autoloaders may not like autoloaders developed by other developers.

Therefore, when you work with a different codebase, you need to study the autoloaders in that particular codebase to understand how they work.

This is why Composer comes into play.

Introduction to the Composer

Composer (https://getcomposer.org/) is a dependency manager for PHP. Composer allows you to manage dependencies in your PHP project. In this tutorial, we'll focus on how to use the Composer for autoloading classes.

Before using Composer, you need to download and install it. The official documentation provides you with the detailed steps of how to download and install Composer on your computer (https://getcomposer.org/download/) .

To check whether the Composer installed successfully, you run the following command from the Command Prompt on Windows or Terminal on macOS and Linux:

```
composer -v
```

It'll return the current version and a lot of options that you can use with the `composer` command.

Autoloading classes with Composer

Back the the previous example, to use the Composer, you first create a new file called `composer.json` under the project's root folder. The project directory will look like this:

```
.
├── app
│   ├── bootstrap.php
│   └── models
│       └── User.php
```

```
├── composer.json
└── index.php
```

In the `composer.json`, you add the following code:

```
{
    "autoload": {
        "classmap": ["app/models"]
    }
}
```

This code means that Composer will autoload all class files defined the `app/models` folder.

If you have classes from other folders that you want to load, you can specify them in `classmap` array:

```
    {
        "autoload": {
            "classmap": ["app/models", "app/services"]
        }
    }
```

In this example, Composer will load classes from both `models` and `services` folders under the `app` folder.

Next, launch the Command Prompt on Windows or Terminal on macOS and Linux, and navigate to the project directory.

Then, type the following command from the project directory:

```
composer dump-autoload
```

Composer will generate a directory called `vendor` that contains a number of files like this:

```
.
├── app
```

```
│   ├── bootstrap.php
│   └── models
│       └── User.php
├── composer.json
├── index.php
└── vendor
    ├── autoload.php
    └── composer
        ├── autoload_classmap.php
        ├── autoload_namespaces.php
        ├── autoload_psr4.php
        ├── autoload_real.php
        ├── autoload_static.php
        ├── ClassLoader.php
        └── LICENSE
```

The most important file to you for now is `autoload.php` file.

After that, load the `autoload.php` file in the `bootstrap.php` file using the `require_once` construct:

```php
<?php

require_once __DIR__ . '/../vendor/autoload.php';
```

Finally, you can use the `User` class in the `index.php` :

```php
<?php

require './app/bootstrap.php';

$user = new User('admin', '$ecurePa$$w0rd1');
```

From now, whenever you have a new class in the `models` directory, you need to run the command `composer dump-autoload` again to regenerate the `autoload.php` file.

For example, the following defines a new class called `Comment` in the `Comment.php` file under the `models` folder:

```php
<?php

class Comment
{
    private $comment;

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

    public function getComment(): string
    {
        return strip_tags($this->comment);
    }
}
```

If you don't run the `composer dump-autoload` command and use the `Comment` class in the `index.php` file, you'll get an error:

```php
<?php

require './app/bootstrap.php';

$user = new User('admin', '$ecurePa$$w0rd1');

$comment = new Comment('<h1>Hello</h1>');
echo $comment->getComment();
```

Error:

```
Fatal error: Uncaught Error: Class 'Comment' not found in...
```

However, if you run the `composer dump-autoload` command again, the `index.php` file will work properly.

Composer autoload with PSR-4

PSR stands for PHP Standard Recommendation. PSR is a PHP specification published by the PHP Framework Interop Group or PHP-FIG.

The goals of PSR are to enable interoperability of PHP components and to provide a common technical basis for the implementation of best practices in PHP programming.

PHP-FIG has published a lot of PSR starting from PSR-0. For a complete list of PSR, check it out the PSR page (https://www.php-fig.org/psr/) .

PSR-4 is auto-loading standard (https://www.php-fig.org/psr/psr-4/) that describes the specification for autoloading classes from file paths. https://www.php-fig.org/psr/psr-4/

According to the PSR-4, a fully qualified class name has the following structure:

```
\<NamespaceName>(\<SubNamespaceNames>)*\<ClassName>
```

The structure starts with a namespace, followed by one or more sub namespaces, and the class name.

To comply with PSR-4, you need to structure the previous application like this:

```
.
├── app
│   ├── Acme
│   │   ├── Auth
│   │   │   └── User.php
│   │   └── Blog
│   │       └── Comment.php
│   └── bootstrap.php
```

```
├── composer.json
└── index.php
```

The new structure has the following changes:

First, the `models` directory is deleted.

Second, the `User.php` is under the `Acme/Auth` folder. And the `User` class is namespaced with `Acme/Auth`. Notice how namespaces map to the directory structure. This also helps you find a class file more quickly by looking at its namespace.

```php
<?php

namespace Acme\Auth;

class User
{
    // implementation
    // ...
}
```

Third, the `Comment.php` is under the `Acme/Blog` folder. The `Comment` class has the namespace `Acme\Blog`:

```php
<?php

namespace Acme\Blog;

class Comment
{
    // implementation
    // ...
}
```

Fourth, the `composer.json` file looks like the following:

```
{
    "autoload": {
        "psr-4": {
            "Acme\\":"app/Acme"
        }
    }
}
```

Instead using the `classmap`, the `composer.json` file now uses `psr-4`. The `psr-4` maps the namespace `"Acme\\"` to the `"app/Acme"` folder.

Note that the second backslash (`\`) in the `Acme\` namespace is used to escape the first backslash (`\`).

Fifth, to use the `User` and `Comment` classes in the `index.php` file, you need to run the `composer dump-autoload` command to generate the autoload.php file:

```
composer dump-autoload
```

Since the `User` and `Comment` classes have namespaces, you need to have the `use` statements in `index.php` as follows:

```
<?php

require './app/bootstrap.php';

use Acme\Auth\User as User;
use Acme\Blog\Comment as Comment;

$user = new User('admin', '$ecurePa$$w0rd1');

$comment = new Comment('<h1>Hello</h1>');
echo $comment->getComment();
```

Summary

- Composer is a dependency management tool in PHP.
- Use PSR-4 for organizing directory and class files.
- Use the `composer dump-autoload` command to generate the `autoload.php` file.