How to create a WordPress plugin using PHP and OOP, with Composer for package management, Symfony components for backend functionality, and Twig for frontend templating.

Mark Caggiano

```
    public function render_shortcode()
    {
        $request = Request::createFromGlobals();
        $data = [
            'title' => 'My Plugin Title',
            'message' => $request->query->get('message')
        ];
        $loader = new \Twig\Loader\FilesystemLoader(__DIR__ . '/../templates
        $twig = new \Twig\Environment($loader);
        echo $twig->render('my-plugin.twig', $data);
    }
}
```

Here’s a step-by-step guide on how to create a WordPress plugin using PHP and OOP, with Composer for package management, Symfony components for backend functionality, and Twig for frontend templating.

Step 1: Setup

First, you'll need to create a new WordPress plugin directory in your `wp-content/plugins` folder. Let's call it `my-plugin`. Inside this directory, create the following files and folders:

- `composer.json`: This file will hold our Composer dependencies and configuration.
- `src`: This folder will hold our plugin's PHP source code.
- `templates`: This folder will hold our plugin's Twig templates.

Step 2: Composer

Next, let's set up our Composer configuration. In `composer.json`, add the following dependencies:

```
{
  "name": "my-plugin",
  "type": "wordpress-plugin",
  "require": {
    "php": ">=7.4",
    "symfony/http-foundation": "^6.0",
    "twig/twig": "^3.0"
  },
  "autoload": {
    "psr-4": {
      "MyPlugin\\": "src/"
    }
  }
}
```

This tells Composer to install PHP 7.4 or greater, as well as the Symfony HTTP Foundation and Twig packages. We've also set up an autoload configuration that maps the `MyPlugin` namespace to the `src` folder.

After saving `composer.json`, run `composer install` from the command line in your `my-plugin` directory to install the dependencies.

Step 3: Plugin Bootstrap

Now let's create a bootstrap file for our plugin. In `src/MyPlugin.php`, add the following code:

```php
<?php

namespace MyPlugin;

use Symfony\Component\HttpFoundation\Request;

class MyPlugin
{
    public function __construct()
    {
        add_action('init', [$this, 'init']);
    }

    public function init()
    {
        add_shortcode('my_plugin_shortcode', [$this, 'render_shortcode']);
    }

    public function render_shortcode()
    {
        $request = Request::createFromGlobals();
        $data = [
            'title' => 'My Plugin Title',
            'message' => $request->query->get('message')
        ];
        $loader = new \Twig\Loader\FilesystemLoader(__DIR__ . '/../templates');
        $twig = new \Twig\Environment($loader);
        echo $twig->render('my-plugin.twig', $data);
    }
}
```

This sets up our `MyPlugin` class, which hooks into the WordPress `init` action and registers a shortcode callback function that will render our Twig template.

We're using the Symfony `Request` class to get any query parameters that might be passed to our shortcode via the URL. We're also creating a `$data` array that will be passed to the Twig template, with a `title` and `message` value.

Finally, we're using the Twig `Environment` class to render our `my-plugin.twig` template, passing in the `$data` array.

Step 4: Twig Templates

Now let's create our Twig template. In `templates/my-plugin.twig`, add the following code:

```
<h1>{{ title }}</h1>
{% if message %}
  <p>{{ message }}</p>
{% endif %}
```

This is a very simple template that outputs the `title` variable as an `<h1>` tag, and if a `message` variable is present, outputs it as a `<p>` tag.

Step 5: Plugin Activation

Now let's add some activation code to our plugin. In `src/MyPlugin.php`, add the following code:

```
<?php

namespace MyPlugin;

use Symfony\Component\HttpFoundation\Request;

class MyPlugin
{
    public function __construct()
    {
        register_activation_hook(__FILE__, [$this, 'activate']);
        add_action('init', [$this, 'init']);
    }

    public function activate()
    {
        // Add any activation code here
    }

    public function init()
    {
        add_shortcode('my_plugin_shortcode', [$this, 'render_shortcode']);
    }

    public function render_shortcode()
    {
        $request = Request::createFromGlobals();
        $data = [
            'title' => 'My Plugin Title',
            'message' => $request->query->get('message')
        ];
        $loader = new \Twig\Loader\FilesystemLoader(__DIR__ . '/../templates');
```

```
        $twig = new \Twig\Environment($loader);
        echo $twig->render('my-plugin.twig', $data);
    }
}
```

We've added a `register_activation_hook()` call to the constructor, which will trigger our `activate()` method when the plugin is activated in the WordPress admin.

Step 6: Plugin Deactivation

Let's also add some deactivation code to our plugin. In `src/MyPlugin.php`, add the following code:

```
<?php

namespace MyPlugin;

use Symfony\Component\HttpFoundation\Request;

class MyPlugin
{
    public function __construct()
    {
        register_activation_hook(__FILE__, [$this, 'activate']);
        register_deactivation_hook(__FILE__, [$this, 'deactivate']);
        add_action('init', [$this, 'init']);
    }

    public function activate()
    {
        // Add any activation code here
    }

    public function deactivate()
    {
        // Add any deactivation code here
    }

    public function init()
    {
        add_shortcode('my_plugin_shortcode', [$this, 'render_shortcode']);
    }

    public function render_shortcode()
    {
        $request = Request::createFromGlobals();
        $data = [
            'title' => 'My Plugin Title',
            'message' => $request->query->get('message')
        ];
        $loader = new \Twig\Loader\FilesystemLoader(__DIR__ . '/../templates');
        $twig = new \Twig\Environment($loader);
```

```
            echo $twig->render('my-plugin.twig', $data);
        }
    }
```

We've added a `register_deactivation_hook()` call to the constructor, which will trigger our `deactivate()` method when the plugin is deactivated in the WordPress admin.

Step 7: Testing

Finally, let's test our plugin. Activate the plugin in the WordPress admin, and then add the `[my_plugin_shortcode]` shortcode to any page or post. You should see the "My Plugin Title" heading and an empty paragraph tag.

If you append `?message=Hello%20World` to the URL, you should see the "My Plugin Title" heading and a "Hello World" paragraph tag.

Congratulations, you've successfully created a WordPress plugin using PHP and OOP, with Composer for package management, Symfony components for backend functionality, and Twig for frontend templating!