Symfony versus Flat PHP

symfony.com/doc/current/introduction/from_flat_php_to_symfony.html

Why is Symfony better than just opening up a file and writing flat PHP?

If you've never used a PHP framework, aren't familiar with the Model-View-Controller (MVC) philosophy, or just wonder what all the *hype* is around Symfony, this article is for you. Instead of *telling* you that Symfony allows you to develop faster and better software than with flat PHP, you'll see for yourself.

In this article, you'll write a basic application in flat PHP, and then refactor it to be more organized. You'll travel through time, seeing the decisions behind why web development has evolved over the past several years to where it is now.

By the end, you'll see how Symfony can rescue you from mundane tasks and let you take back control of your code.

A Basic Blog in Flat PHP

In this article, you'll build the token blog application using only flat PHP. To begin, create a single page that displays blog entries that have been persisted to the database. Writing in flat PHP is quick and dirty:

```
<?php
// index.php
$connection = new PDO("mysql:host=localhost;dbname=blog_db", 'myuser',
'mypassword');

$result = $connection->query('SELECT id, title FROM post');
?>

<!DOCTYPE html>
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php while ($row = $result->fetch(PDO::FETCH_ASSOC)): ?>
            <li>
                <a href="/show.php?id=<?= $row['id'] ?>">
                    <?= $row['title'] ?>
                </a>
            </li>
            <?php endwhile ?>
        </ul>
    </body>
</html>

<?php
$connection = null;
?>
```

That's quick to write, fast to deploy and run, and, as your app grows, impossible to maintain. There are several problems that need to be addressed:

- **No error-checking**: What if the connection to the database fails?
- **Poor organization**: If the application grows, this single file will become increasingly unmaintainable. Where should you put code to handle a form submission? How can you validate data? Where should code go for sending emails?
- **Difficult to reuse code**: Since everything is in one file, there's no way to reuse any part of the application for other "pages" of the blog.

Another problem not mentioned here is the fact that the database is tied to MySQL. Though not covered here, Symfony fully integrates Doctrine, a library dedicated to database abstraction and mapping.

Isolating the Presentation

The code can immediately gain from separating the application "logic" from the code that prepares the HTML "presentation":

```
// index.php
$connection = new PDO("mysql:host=localhost;dbname=blog_db", 'myuser',
'mypassword');

$result = $connection->query('SELECT id, title FROM post');

$posts = [];
while ($row = $result->fetch(PDO::FETCH_ASSOC)) {
    $posts[] = $row;
}

$connection = null;

// include the HTML presentation code
require 'templates/list.php';
```

The HTML code is now stored in a separate file `templates/list.php`, which is primarily an HTML file that uses a template-like PHP syntax:

```
<!-- templates/list.php -->
<!DOCTYPE html>
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php foreach ($posts as $post): ?>
            <li>
                <a href="/show.php?id=<?= $post['id'] ?>">
                    <?= $post['title'] ?>
                </a>
            </li>
            <?php endforeach ?>
        </ul>
    </body>
</html>
```

By convention, the file that contains all the application logic - `index.php` - is known as a "controller". The term controller is a word you'll hear a lot, regardless of the language or framework you use. It refers to the area of *your* code that processes user input and prepares the response.

In this case, the controller prepares data from the database and then includes a template to present that data. With the controller isolated, you could change *just* the template file if you needed to render the blog entries in some other format (e.g. `list.json.php` for JSON format).

Isolating the Application (Domain) Logic

So far the application contains only one page. But what if a second page needed to use the same database connection, or even the same array of blog posts? Refactor the code so that the core behavior and data-access functions of the application are isolated in a new file called `model.php`:

```
// model.php
function open_database_connection()
{
    $connection = new PDO("mysql:host=localhost;dbname=blog_db", 'myuser',
'mypassword');

    return $connection;
}

function close_database_connection(&$connection)
{
    $connection = null;
}

function get_all_posts()
{
    $connection = open_database_connection();

    $result = $connection->query('SELECT id, title FROM post');

    $posts = [];
    while ($row = $result->fetch(PDO::FETCH_ASSOC)) {
        $posts[] = $row;
    }
    close_database_connection($connection);

    return $posts;
}
```

The filename `model.php` is used because the logic and data access of an application is traditionally known as the "model" layer. In a well-organized application, the majority of the code representing your "business logic" should live in the model (as opposed to living in a controller). And unlike in this example, only a portion (or none) of the model is actually concerned with accessing a database.

The controller (`index.php`) is now only a few lines of code:

```
// index.php
require_once 'model.php';

$posts = get_all_posts();

require 'templates/list.php';
```

Now, the sole task of the controller is to get data from the model layer of the application (the model) and to call a template to render that data. This is a very concise example of the model-view-controller pattern.

Isolating the Layout

At this point, the application has been refactored into three distinct pieces offering various advantages and the opportunity to reuse almost everything on different pages.

The only part of the code that *can't* be reused is the page layout. Fix that by creating a new `templates/layout.php` file:

```
<!-- templates/layout.php -->
<!DOCTYPE html>
<html>
    <head>
        <title><?= $title ?></title>
    </head>
    <body>
        <?= $content ?>
    </body>
</html>
```

The template `templates/list.php` can now be simplified to "extend" the `templates/layout.php`:

```
<!-- templates/list.php -->
<?php $title = 'List of Posts' ?>

<?php ob_start() ?>
    <h1>List of Posts</h1>
    <ul>
        <?php foreach ($posts as $post): ?>
        <li>
            <a href="/show.php?id=<?= $post['id'] ?>">
                <?= $post['title'] ?>
            </a>
        </li>
        <?php endforeach ?>
    </ul>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

You now have a setup that will allow you to reuse the layout. Unfortunately, to accomplish this, you're forced to use a few ugly PHP functions (`ob_start()` , `ob_get_clean()`) in the template. Symfony solves this using a Templating component. You'll see it in action shortly.

Adding a Blog "show" Page

The blog "list" page has now been refactored so that the code is better-organized and reusable. To prove it, add a blog "show" page, which displays an individual blog post identified by an `id` query parameter.

To begin, create a new function in the `model.php` file that retrieves an individual blog result based on a given id:

```
// model.php
function get_post_by_id($id)
{
    $connection = open_database_connection();

    $query = 'SELECT created_at, title, body FROM post WHERE id=:id';
    $statement = $connection->prepare($query);
    $statement->bindValue(':id', $id, PDO::PARAM_INT);
    $statement->execute();

    $row = $statement->fetch(PDO::FETCH_ASSOC);

    close_database_connection($connection);

    return $row;
}
```

Next, create a new file called `show.php` - the controller for this new page:

```
// show.php
require_once 'model.php';

$post = get_post_by_id($_GET['id']);

require 'templates/show.php';
```

Finally, create the new template file - `templates/show.php` - to render the individual blog post:

```
<!-- templates/show.php -->
<?php $title = $post['title'] ?>

<?php ob_start() ?>
    <h1><?= $post['title'] ?></h1>

    <div class="date"><?= $post['created_at'] ?></div>
    <div class="body">
        <?= $post['body'] ?>
    </div>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Creating the second page now requires very little work and no code is duplicated. Still, this page introduces even more lingering problems that a framework can solve for you. For example, a missing or invalid `id` query parameter will cause the page to crash. It would be better if this caused a 404 page to be rendered, but this can't really be done yet.

Another major problem is that each individual controller file must include the `model.php` file. What if each controller file suddenly needed to include an additional file or perform some other global task (e.g. enforce security)? As it stands now, that code would need to be added to every controller file. If you forget to include something in one file, hopefully it doesn't relate to security...

A "Front Controller" to the Rescue

The solution is to use a front controller: a single PHP file through which *all* requests are processed. With a front controller, the URIs for the application change slightly, but start to become more flexible:

```
Without a front controller
/index.php          => Blog post list page (index.php executed)
/show.php           => Blog post show page (show.php executed)

With index.php as the front controller
/index.php          => Blog post list page (index.php executed)
/index.php/show     => Blog post show page (index.php executed)
```

By using rewrite rules in your web server configuration, the `index.php` won't be needed and you will have beautiful, clean URLs (e.g. `/show`).

When using a front controller, a single PHP file (`index.php` in this case) renders *every* request. For the blog post show page, `/index.php/show` will actually execute the `index.php` file, which is now responsible for routing requests internally based on the full URI. As you'll see, a front controller is a very powerful tool.

Creating the Front Controller

You're about to take a **big** step with the application. With one file handling all requests, you can centralize things such as security handling, configuration loading, and routing. In this application, `index.php` must now be smart enough to render the blog post list page *or* the blog post show page based on the requested URI:

```
// index.php

// load and initialize any global libraries
require_once 'model.php';
require_once 'controllers.php';

// route the request internally
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if ('/index.php' === $uri) {
    list_action();
} elseif ('/index.php/show' === $uri && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('HTTP/1.1 404 Not Found');
    echo '<html><body><h1>Page Not Found</h1></body></html>';
}
```

For organization, both controllers (formerly `/index.php` and `/index.php/show`) are now PHP functions and each has been moved into a separate file named `controllers.php`:

```
// controllers.php
function list_action()
{
    $posts = get_all_posts();
    require 'templates/list.php';
}

function show_action($id)
{
    $post = get_post_by_id($id);
    require 'templates/show.php';
}
```

As a front controller, `index.php` has taken on an entirely new role, one that includes loading the core libraries and routing the application so that one of the two controllers (the `list_action()` and `show_action()` functions) is called. In reality, the front controller is beginning to look and act a lot like how Symfony handles and routes requests.

But be careful not to confuse the terms *front controller* and *controller*. Your app will usually have only *one* front controller, which boots your code. You will have *many* controller functions: one for each page.

Another advantage of a front controller is flexible URLs. Notice that the URL to the blog post show page could be changed from `/show` to `/read` by changing code in only one location. Before, an entire file needed to be renamed. In Symfony, URLs are even more flexible.

By now, the application has evolved from a single PHP file into a structure that is organized and allows for code reuse. You should be happier, but far from being satisfied. For example, the routing system is fickle, and wouldn't recognize that the list page - `/index.php` - should be accessible also via `/` (if Apache rewrite rules were added). Also, instead of developing the blog, a lot of time is being spent working on the "architecture" of the code (e.g. routing, calling controllers, templates, etc.). More time will need to be spent to handle form submissions, input validation, logging and security. Why should you have to reinvent solutions to all these routine problems?

Add a Touch of Symfony

Symfony to the rescue. Before actually using Symfony, you need to download it. This can be done by using Composer, which takes care of downloading the correct version and all its dependencies and provides an autoloader. An autoloader is a tool that makes it possible to start using PHP classes without explicitly including the file containing the class.

In your root directory, create a `composer.json` file with the following content:

```
{
    "require": {
        "symfony/http-foundation": "^4.0"
    },
    "autoload": {
        "files": ["model.php","controllers.php"]
    }
}
```

Next, download Composer and then run the following command, which will download Symfony into a `vendor/` directory:

```
$ composer install
```

Beside downloading your dependencies, Composer generates a `vendor/autoload.php` file, which takes care of autoloading for all the files in the Symfony Framework as well as the files mentioned in the autoload section of your `composer.json`.

Core to Symfony's philosophy is the idea that an application's main job is to interpret each request and return a response. To this end, Symfony provides both a Request and a Response class. These classes are object-oriented representations of the raw HTTP request being processed and the HTTP response being returned. Use them to improve the blog:

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

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();

$uri = $request->getPathInfo();
if ('/' === $uri) {
    $response = list_action();
} elseif ('/show' === $uri && $request->query->has('id')) {
    $response = show_action($request->query->get('id'));
} else {
    $html = '<html><body><h1>Page Not Found</h1></body></html>';
    $response = new Response($html, Response::HTTP_NOT_FOUND);
}

// echo the headers and send the response
$response->send();
```

The controllers are now responsible for returning a `Response` object. To make this easier, you can add a new `render_template()` function, which, incidentally, acts quite a bit like the Symfony templating engine:

```
// controllers.php
use Symfony\Component\HttpFoundation\Response;

function list_action()
{
    $posts = get_all_posts();
    $html = render_template('templates/list.php', ['posts' => $posts]);

    return new Response($html);
}

function show_action($id)
{
    $post = get_post_by_id($id);
    $html = render_template('templates/show.php', ['post' => $post]);

    return new Response($html);
}

// helper function to render templates
function render_template($path, array $args)
{
    extract($args);
    ob_start();
    require $path;
    $html = ob_get_clean();

    return $html;
}
```

By bringing in a small part of Symfony, the application is more flexible and reliable. The `Request` provides a dependable way to access information about the HTTP request. Specifically, the getPathInfo() method returns a cleaned URI (always returning `/show` and never `/index.php/show`). So, even if the user goes to `/index.php/show`, the application is intelligent enough to route the request through `show_action()`.

The `Response` object gives flexibility when constructing the HTTP response, allowing HTTP headers and content to be added via an object-oriented interface. And while the responses in this application are simple, this flexibility will pay dividends as your application grows.

The Sample Application in Symfony

The blog has come a *long* way, but it still contains a lot of code for such a basic application. Along the way, you've made a basic routing system and a function using `ob_start()` and `ob_get_clean()` to render templates. If, for some reason, you needed to continue building this "framework" from scratch, you could at least use Symfony's standalone Routing component and Twig, which already solve these problems.

Instead of re-solving common problems, you can let Symfony take care of them for you. Here's the same sample application, now built in Symfony:

```
// src/Controller/BlogController.php
namespace App\Controller;

use App\Entity\Post;
use Doctrine\Persistence\ManagerRegistry;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class BlogController extends AbstractController
{
    public function list(ManagerRegistry $doctrine)
    {
        $posts = $doctrine->getRepository(Post::class)->findAll();

        return $this->render('blog/list.html.twig', ['posts' => $posts]);
    }

    public function show(ManagerRegistry $doctrine, $id)
    {
        $post = $doctrine->getRepository(Post::class)->find($id);

        if (!$post) {
            // cause the 404 page not found to be displayed
            throw $this->createNotFoundException();
        }

        return $this->render('blog/show.html.twig', ['post' => $post]);
    }
}
```

Notice, both controller functions now live inside a "controller class". This is a nice way to group related pages. The controller functions are also sometimes called *actions*.

The two controllers (or actions) are still lightweight. Each uses the Doctrine ORM library to retrieve objects from the database and the Templating component to render a template and return a `Response` object. The `list.html.twig` template is now quite a bit simpler, and uses Twig:

```
{# templates/blog/list.html.twig #}
{% extends 'base.html.twig' %}

{% block title %}List of Posts{% endblock %}

{% block body %}
<h1>List of Posts</h1>
<ul>
    {% for post in posts %}
    <li>
        <a href="{{ path('blog_show', { id: post.id }) }}">
            {{ post.title }}
        </a>
    </li>
    {% endfor %}
</ul>
{% endblock %}
```

The `layout.php` file is nearly identical:

```
<!-- templates/base.html.twig -->
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

The `show.html.twig` template is left as an exercise: updating it should be really similar to updating the `list.html.twig` template.

When Symfony's engine (called the Kernel) boots up, it needs a map so that it knows which controllers to call based on the request information. A routing configuration map - `config/routes.yaml` - provides this information in a readable format:

```
# config/routes.yaml
blog_list:
    path:     /blog
    controller: App\Controller\BlogController::list

blog_show:
    path:     /blog/show/{id}
    controller: App\Controller\BlogController::show
```

Now that Symfony is handling all the mundane tasks, the front controller `public/index.php` is reduced to bootstrapping. And since it does so little, you'll never have to touch it:

```
// public/index.php
require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../src/Kernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new Kernel('prod', false);
$kernel->handle(Request::createFromGlobals())->send();
```

The front controller's only job is to initialize Symfony's engine (called the Kernel) and pass it a `Request` object to handle. The Symfony core asks the router to inspect the request. The router matches the incoming URL to a specific route and returns information about the route, including the controller that should be called. The correct controller from the matched route is called and your code inside the controller creates and returns the appropriate `Response` object. The HTTP headers and content of the `Response` object are sent back to the client.

It's a beautiful thing.

Where Symfony Delivers

In the rest of the documentation articles, you'll learn more about how each piece of Symfony works and how you can organize your project. For now, celebrate how migrating the blog from flat PHP to Symfony has improved your life:

- Your application now has **clear and consistently organized code** (though Symfony doesn't force you into this). This promotes **reusability** and allows for new developers to be productive in your project more quickly;
- 100% of the code you write is for *your* application. You **don't need to develop or maintain low-level utilities** such as autoloading, routing, or rendering controllers;
- Symfony gives you **access to open source tools** such as Doctrine and the Templating, Security, Form, Validator and Translation components (to name a few);
- The application now enjoys **fully-flexible URLs** thanks to the Routing component;
- Symfony's HTTP-centric architecture gives you access to powerful tools such as **HTTP caching** powered by **Symfony's internal HTTP cache** or more powerful tools such as Varnish. This is covered in another article all about caching.

And perhaps best of all, by using Symfony, you now have access to a whole set of **high-quality open source tools developed by the Symfony community**! A good selection of Symfony community tools can be found on GitHub.