www.phparch.com

September 2016
VOLUME 15 - Issue 9

php[architect]

legacy code of the ancients

Illuminating Legacy Applications

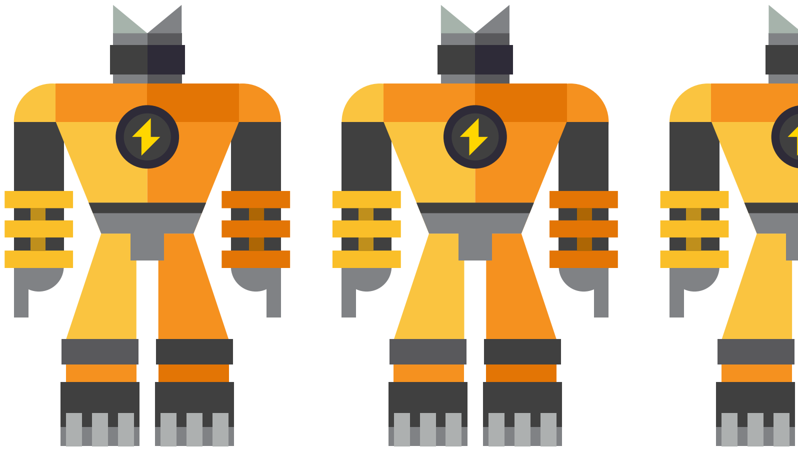
The Modernization of Multiple Legacy Websites

Legacy Code Needs Love Too

FREE Article!

ALSO INSIDE

The Modernization of Multiple Legacy Websites



Jack D. Polifka

Most developers will need to maintain legacy code at some point in their career. That code will hopefully be contained in a single codebase, but may instead be contained across several, and working with multiple legacy systems can be daunting. In this article, I share my experiences in maintaining multiple legacy websites which were primarily written with procedural PHP. I describe the steps I used to migrate the existing code to a single codebase utilizing a Composer-based Model-View-Controller framework. After describing the process, I will share points of success, as well as possible improvements.

Introduction

The process I am about to describe should be helpful for any developer who needs to work with any legacy code in PHP, not just code which contains multiple websites or systems. Additional elements will be present to address the feature of having multiple websites. I hope to give a high-level overview of the steps I used for the process with only finer details as needed. I will not be naming specific libraries or components. Most developers already have personal favorite libraries or components when it comes to specific functionality. Instead, I will just name a few which can fulfill the desired functionality when needed.

Planning

The first step in working with any legacy code is planning. The process used for interacting with legacy code will depend on a project's resources and goals. For this specific project, I was the only developer, making resources limited. Regarding goals, there were three main goals. The first was to **keep each website operational** during the migration. The second goal was to be able to **continue to respond to clients' needs** and other maintenance issues. Last, to **combine all of the websites into a single codebase** to share configuration settings and common dependencies. Since I was the only developer maintaining these websites, I believed this would reduce the amount of development time required for future tasks by removing the need to interact with separate codebases.

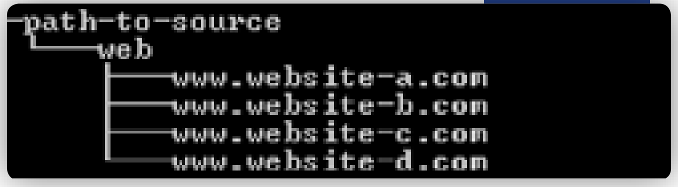
Based on those resources and goals, an iterative process was selected to handle changes in sections until the groundwork for a Model-View-Controller (MVC) architecture was set up. An iterative process was selected for two primary reasons. First, it allowed me to divide the migration tasks into smaller pieces, which allowed me to be more readily available to client requests. Second, it reduced the level of associated risk with each set of changes. By reducing the number of changes required for each task, the possibility for program errors was also reduced. Once MVC was properly implemented, all future tasks were completed using that architecture. This was for code which added new functionality and any existing code that needed to be updated.

Setting up the Basics

When I started the migration, only one of the websites I inherited had a development website. Since programming in a production environment inevitably leads to errors on a live website, I set up development environments for each website. For the short term, production tables and data were copied from the production databases to the development databases. Later, this process would be automated. Finally, a Git repository was set up and all code for each website was version controlled.

With a proper development environment, I could begin without the risk of affecting production. The first development task was to put the code of each website into subfolders of another folder called `web`. Each subfolder was named according to the website code it contained. This can be seen in Figure 1. This `web` folder would become the entry point for each website. Each website was set up as a symlink to its corresponding web folder. This setup is done to share configuration settings and common dependencies which are discussed shortly.

Initial Directory Layout **FIGURE 1**



```
path-to-source
└───web
    ├───www.website-a.com
    ├───www.website-b.com
    ├───www.website-c.com
    └───www.website-d.com
```

Routing and the Front Controller

The next step in the migration was the addition of routing and a front controller for each website. PHP has a great selection of routers such as FastRoute[1], Symfony Routing[2], and Aura Router[3]. In the event the router selected requires configuration files for mapping responses to controller/action combinations, a settings directory can be set up with a subfolder for each website. With a router selected, an appropriate `composer.json` was set up so it and other dependencies could be downloaded with Composer. Remember, if the router selected uses non-PHP files, then an appropriate method for parsing them will need to be selected and added to the `composer.json`.

With the router downloaded, a front controller was added to the root of each website folder in the web directory. The front controller would try to match the address of any request sent to the website to a controller and action. If a request did not map to any controller/action combination or if the controller/action combination did not exist, a `404 Not Found` response would be sent. Each front controller would resemble Listing 1.

LISTING 1

```php
<?php

require_once(__DIR__ . '/../autoload.php');

// The router of your choice would be setup here and
// would try to match requests based on your mapping of
// requests to controllers and actions

if ($routerFoundMatch === true) {
    $controller->action();
} else {
    $this->redirect('/404/');
}
```

To use front controllers, all requests need to be forwarded through them. Since Linux, along with Apache, were being used for my websites, mod_rewrite was used. An `.htaccess` file was created for each website folder in the web directory. These `.htaccess` files in their simplest form looked like the following:

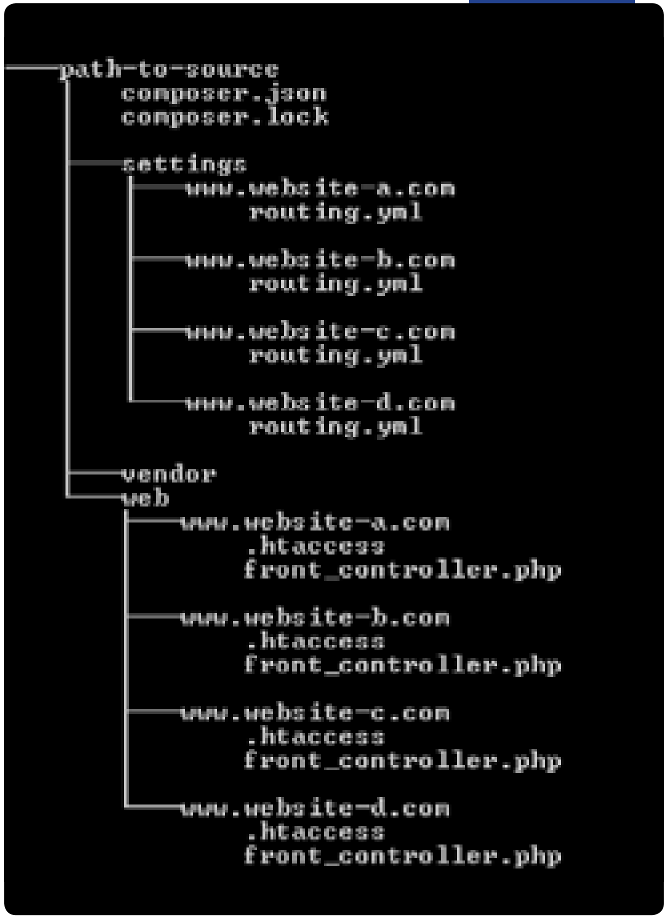
```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^front_controller.php [QSA,L]
```

This redirected any requests for non-existent files and directories to the front controller. It should be noted there were special cases where the `.htaccess` files did not redirect requests to legacy files even if they existed. This was for the case where index files were absent from web requests such as `www.website-a.com/request/address/`. To account for these, a simple if statement was added to each front controller that looked for `.php` in strings. If it was not found, it was assumed to be an index file. If a mix of PHP and HTML files were used, it would have been updated to include `.html` as well.

Lastly, since the request for files would come from the front controller at times, code such as `require`, `require_once`, and `include` were modified to account for the new directory layout,

1 FastRoute: https://github.com/nikic/FastRoute
2 Symfony Routing: http://symfony.com/doc/current/routing.html
3 Aura Router: https://github.com/auraphp/Aura.Router

Directory Layout with Front Controllers **FIGURE 2**



```
──path-to-source
     composer.json
     composer.lock
   │
   ├──settings
   │   ├──www.website-a.com
   │   │      routing.yml
   │   │
   │   ├──www.website-b.com
   │   │      routing.yml
   │   │
   │   ├──www.website-c.com
   │   │      routing.yml
   │   │
   │   └──www.website-d.com
   │          routing.yml
   │
   ├──vendor
   └──web
       ├──www.website-a.com
       │      .htaccess
       │      front_controller.php
       │
       ├──www.website-b.com
       │      .htaccess
       │      front_controller.php
       │
       ├──www.website-c.com
       │      .htaccess
       │      front_controller.php
       │
       └──www.website-d.com
              .htaccess
              front_controller.php
```

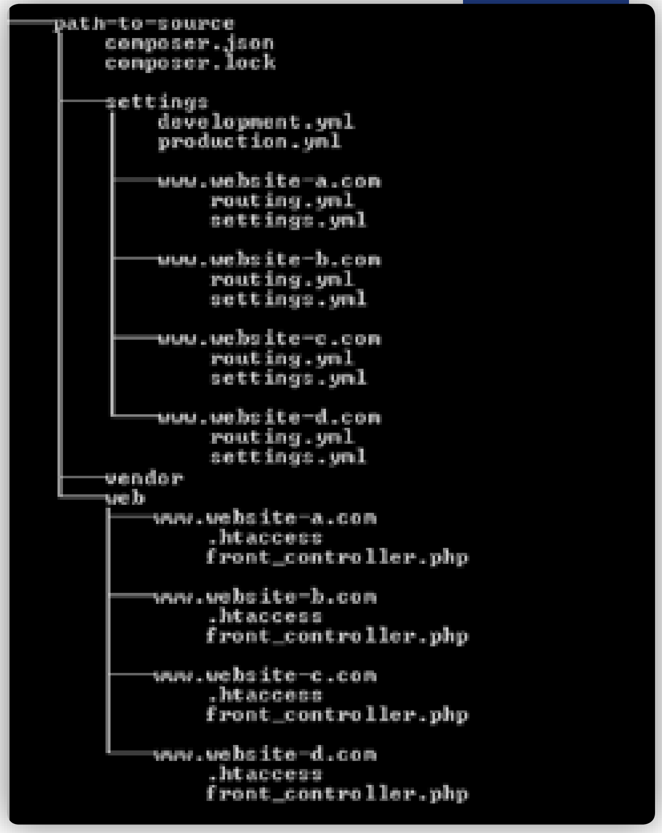
which now resembles Figure 2. This is done to all legacy files to be consistent.

Configuration

With each website having a front controller and common point for requests, the configuration was set up for each website. Overall, three sets of configurations were created: a set of production settings to be shared among all of the websites, a development set whenever a website was accessed from a development website, and a set of website specific settings. Configuration was set up in a cascade style so production settings would always be read first, overridden by development settings, and finally website specific settings. Like the router, I am not going to tell how you should store your application settings, but make sure to add the necessary updates to `composer.json`. After configuration setting files were added, the directory layout looked like Figure 3. Listing 2 shows the updated source for each front controller.

Directory Layout with Configuration Files

FIGURE 3



```
path-to-source
    composer.json
    composer.lock
    settings
        development.yml
        production.yml
        www.website-a.com
            routing.yml
            settings.yml
        www.website-b.com
            routing.yml
            settings.yml
        www.website-c.com
            routing.yml
            settings.yml
        www.website-d.com
            routing.yml
            settings.yml
    vendor
    web
        www.website-a.com
            .htaccess
            front_controller.php
        www.website-b.com
            .htaccess
            front_controller.php
        www.website-c.com
            .htaccess
            front_controller.php
        www.website-d.com
            .htaccess
            front_controller.php
```

LISTING 2

```php
<?php

require_once(__DIR__ . '/../autoload.php');

$yourConfig->getSettings('../settings/production.yml');
if ($developmentEnvironment === true)  {
    $yourConfig->getSettings('../settings/development.yml');
}

// Get website specific settings
$yourConfig->getSettings('../settings/' . $nameOfSite
                         . '/settings.yml');

// The router of your choice would be setup here and
// would try to match requests based on your mapping of
// requests to controllers and actions with settings

if ($routerFoundMatch === true) {
    $controller->action();
} else if ($phpFileTypeFound !== true) {
    $this->include($uri . '.php');
} else {
    $this->redirect('/404/');
}
```

Template System and Future Updates

The last major component missing from the MVC setup was a template system. Examples of template systems include Twig[4] and Blade[5]. When the template system was set up, the first MVC migration occurred. From there, all future tasks would be completed in MVC whether they were for new functionality or to update and refactor existing code.

Testing

Each major update in the migration process was accompanied by user acceptance testing and/or unit tests. User acceptance testing was used for most updates unless the changes were not notable, such as a simple text update. Because users of the system had the most experience with interacting with the application, it was best for them to confirm areas of functionality remained the same after refactoring. People who had less experience with the system might have missed edge-cases which only someone experienced with the system would catch. After using the system on the development site and then approving that the updates were working as intended, changes would be moved to the production site.

Unit tests were written using PHPUnit. After a section of code was updated to be object oriented, unit tests were written for that section. Tests were primarily written for functionality which was commonly used, business critical, or had multiple edge-cases. Because I did not have access to systems such as Jenkins or Bamboo, unit tests were manually run after committing code changes to version control or before moving changes to production.

Areas of Success

One area of success was the low amount of scope creep. It is often tempting during migrations to want to add new functionality while refactoring different areas of code. With proper planning, outlining of goals, and taking note of resources, the objectives of a rewrite can be focused on just refactoring. Having a clear timeline and high level goals from the start can help steer those involved in a project from losing sight of the end product.

Another point of success was the use of the iterative process. During the migration, none of the production websites experienced any major interrupts. In addition, non-migration development tasks were still completed at a steady pace. By using an iterative process to make updates, change sets of

4 *Twig: http://twig.sensiolabs.org*
5 *Blade: https://laravel.com/docs/5.2/blade*

manageable size were pushed to production instead of large ones. This decreased the chance of poor or malfunctioning code making it through testing. At the same time, an iterative process allowed for the migration to be broken down into parts where other development tasks could be completed in between phases of the migration when time was available.

Room for Improvement

One improvement could be the use of a standard framework like Laravel or Symfony instead of a Composer-based MVC framework. Whatever framework was selected, a catch-all routing system could have been set up where any request that did not match any of the predefined routes would automatically set forward to the correct PHP file. This would have allowed the whole system to use an already established set of practices that have base communities. Events such as troubleshooting would be improved by the number of users in those communities. One possible drawback with this, though, would be the time investment in the beginning. First, all the details may not be known about the selected framework requiring learning. (Not to say that learning is bad, but if development speed is critical, this should be considered). Second, using a framework from the beginning would have increased the amount of programming required at the start. Several migration tasks would need to be done at one time versus over several instances, which could increase the chances of a malfunction in production, due to testing oversight.

Another improvement would be the implementation of more unit tests. As stated before, the unit tests that were written were for functionality that was commonly used, business critical, or had multiple edge-cases. Having more tests would have provided more confidence in what was developed and what was tested. Also, they would act as a litmus test for any future enhancements or refactors making sure functionality covered by tests continue to work correctly.

Conclusion

While the process used here was successful for me in addressing a migration with multiple websites, I believe what should be noted are the non-programming aspects of the process. Specifically, the planning stage before any coding happens, having a development environment including development websites and version control, and the use of an iterative process. The planning stage helps developers take inventory of their goals, determine resources they have available to complete the project, and to create a plan. A development environment with version control allows a developer to focus on programming without affecting production websites. An iterative process allows work for a migration to be broken down into smaller steps, thereby reducing the amount of associated risk and also increasing the turnaround time. Any migration would benefit from these advantages.

Additional Reading

Modernizing Legacy Applications in PHP[6], by Paul M. Jones. The book describes a step by step process which can be used to update a legacy codebase made of procedural code to an MVC framework, very similar to the scenario I encountered. Before describing the migration process, Jones emphasizes one of the most important ideas for any migration which is the use of an iterative process. By using an iterative process, no major rewrites need to take place where the level of resources required is high. In addition, using an iterative process allows work to be completed in smaller steps, which can go hand-in-hand with a steady flow of everyday development tasks.

So You Just Inherited a $Legacy Application… slides[7], by Joe Ferguson, a presentation from php[tek] 2016. The presentation slides can found online. This presentation emphasizes the planning stage before a migration occurs. As stated earlier, proper planning before touching code can prevent a lot of headaches during the migration process. Ferguson points out six common issues via questions which can answered in the planning stage. For each question, several resources are presented which can be used to address the issue. For example, one of the questions ask, "Is there a development environment?" Following are slides with references to using Vagrant, Docker, or physical hardware. The other questions are:

1. Is there a framework?
2. Is there a coding standard?
3. Is there any autoloading?
4. How are dependencies behind being handled?
5. Is there an Object-relational mapper or how is the database being utilized?

Jack is a software engineer for the Graduate College at Iowa State University. His past is a mix of academics and industry with a Master of Science in Human Computer Interaction, two and a half years at Opticsplanet.com, and a Bachelor of Science in Web and Digital Media Development. @jack_polifka

6 *Modernizing Legacy Applications in PHP:* https://leanpub.com/mlaphp

7 *So You Just Inherited a $Legacy Application… slides:* http://phpa.me/inherited-legacy-app-slides