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

ALSO INSIDE

WordPress, Drupal, Magento, CakePHP, Laravel, Zend Framework, Symfony, JavaScript, and more...

PHP[WORLD] 2016

php[world] is the only conference designed to **bring all the application & framework communities in PHP together** at the same place to learn from each other.

Get your tickets now!
Expo-only tickets start at **$195** and conference tickets at **$895**

- **5 Full-day (and longer) training classes**
- **70 Informative sessions & workshops**
- **5 Amazing keynotes from industry leaders**

php[world] 2016 Conference
Washington, D.C. — November 14-18

world.phparch.com

AUTOMATTIC
SiteGround
contentful
ZEAMSTER
platform.sh
FIG LEAF
FOXY.IO
BLACKMESH

php[architect]

CONTENTS

SEPTEMBER 2016
Volume 15 - Issue 9

Legacy Code of the Ancients

Columns

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith

php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Legacy Code of the Ancients

We don't always have the luxury of working on greenfield projects where we can try out the latest language features, component libraries, or programming techniques. More often, we're asked to take care of and add features to an application that just works and supports a company or organization's objectives—like making money to pay salaries. Unless it's a relatively new project, you are sure to run into corners of the codebase that should be modernized. The trick is to find the time and marshal your team to do so.

> *Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.*
>
> *- John F. Woods* [1]

My first job out of college was at a web design firm. For me, short project engagements, usually a couple of months at best, are one of the most frustrating aspects of agency work. Once a site launched there were few opportunities to revisit the code and improve it. Maybe the next client would benefit from what we learned. Sometimes a client would have a "maintenance" contract, but the focus was really on adding new features to the site. It was almost impossible to pay down any technical debt.

In contrast, at my next job I was in charge of a single website for almost three years. I relished the opportunity to improve the site. One particular hairy aspect were event registration forms. At launch, I customized their behaviors a lot from one to the next, under pressure to have it ready when the site went live. In the months that followed our launch though, I was able to abstract out many of the customizations into reusable settings and functions. In the end, rolling out a custom form was much faster than at launch, which was appreciated by my internal customers.

In this issue, we'll look at tools and approaches to help you refactor and update legacy code without losing your sanity. First, Colin DeCarlo will share how you can integrate Laravel's Components in *Illuminating Legacy Applications*. He'll show you how to use the Service Container, Eloquent, and other components in your application. Did you inherit multiple websites each with their own quirks? Jack D. Polifka found himself in the same situation. In *The Modernization of Multiple Legacy Websites* he writes about how he migrated to a single, composer-based codebase with an MVC framework. Last, in *Legacy Code Needs Love Too*, John Congdon explains why millions of lines of code need your attention and care. He'll share his approach to modernizing your code and making it easier to maintain.

Also in this issue, if you *are* looking for something new to play with, Adam Englander has just the thing. See how to start *Building for the Internet of Things in PHP* via the GPIO pins on a Raspberry Pi.

1 http://phpa.me/2byquZg

Before you start making drastic changes to your application, David Stockton will help you make sure everything works as it always has in *Level Up*. He'll explain how to get started with acceptance tests to specify and validate requirements in *Behavior Driven Development With Behat*.

If you're looking to add a Dependency Injection Container to your code, Matthew Setter takes a look at a relatively new option. Check out *Education Station: Your Dependency Injection Needs a Disco* for the details.

Cal Evans explains how to *Be a Community Builder* in *Community Corner*. He also shares how he's seen leaders step in when needed.

Are you looking to add two-factor authentication to your application? Chris Cornutt shares a library to help you in *Security Corner: Two-Factor All the Things*.

finally{}, this month Eli White writes *On the Value of a Degree…* Know someone thinking of going back to get a Computer Science degree? Share Eli's thoughts with him or her.

Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit https://phpa.me/write or contact our editorial team at write@phparch.com and get started!

Stay in Touch

Don't miss out on conference, book, and special announcments. Make sure you're connected with us via email, twitter, and facebook.

- Subscribe to our list: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download this Issue's Code Package:

http://phpa.me/September2016_code

Illuminating Legacy Applications

Colin DeCarlo

Congratulations! You've finally gotten the stakeholders to realize you can't continue to maintain, *and add new features*, to that 10-year-old PHP application which may be solely responsible for generating the money everyone in your company uses to pay their mortgages. Not only did you get them to realize the fragility of the application (the one keeping the company from bankruptcy) but you also got them to agree to allow you, and your team, to start a concerted effort to refactor it. You had to work hard to get everyone on board with this initiative; on the outside, the app looks great and functions really well. Understanding that looks are deceiving, and the application is in a very precarious position was a hard pill for everyone to swallow and probably didn't make you, or your team, look good. But, you did it by selling them on infinitely more stable application, which is more maintainable and most importantly, easier to extend with new features. It's time for you to deliver, are you ready?

The Work Before the Work Begins

How you start a large scale refactoring project is going to be the most important decision you're going to make in the entire process. This is the point in history where things really start to change. Are future developers going to look back at this as an example of how to successfully resurrect a failing codebase, or is it going to be an example of how to almost kill an entire company with an out of control refactoring effort?

You're going to need a vision beyond "infinitely more stable, more maintainable and easier to extend." Those words are great for selling the project but honestly, they don't mean much when it comes to putting rubber to the road. You need a structured plan to guide the project in the right direction over the long term. A lot of work needs to be done even before the first line of new code is written; it's going to be hard not just to dive in and start ripping out code, but remember, this project is about building up, not tearing down.

Decisions, Decisions

The first decision which needs to be made is a big one. At the end of this project is the application going to be a brand new codebase or a much improved version of the original codebase? It was a refactoring project the stakeholders agreed to, but can the existing application actually be modernized? The urge to start fresh, without the lingering decisions and fallout of the 'original devs' is going to be hard to resist, but you should resist it if you can. While side-by-side rewrites are possible to accomplish[1], even well-planned rewrites are fraught with danger, much more so than any refactoring effort.

Despite the dark corners and even "that code only Kerry is allowed to touch," you've decided to refactor the application. Let's start planning out the journey from where you are now to where you want to be. Time isn't the factor you'll be measuring at this point, you need to plan out milestones. For instance, how much test coverage does the application currently have besides "not as much as we'd like?" What version of PHP does the application currently run on top of? Can you, or do you have to, upgrade? How much do you really know about the low-level parts of the app, how does the autoloading work? Or the logging? How is email sent?

Non-Negotiable Milestones

Some project milestones are going to be dictated by the project itself, for example, if PHP has to be upgraded so you can include the Laravel Illuminate components, you'll need adequate test coverage of at least the "happy path" through your apps' features. This is just being safe; the last thing you need at the start of the of the project is your site going down because of "some upgrade the developers needed." Don't look at this as a setback; this pays huge gains towards your application being more stable than

1 *In a setup like this, the new application and the old application run downstream from one single entry point to the application configured in Nginx. By default, all traffic is directed to the old application. As functionality is added to the new application, new routing rules are added to the Nginx configuration which directs traffic over to it. Both applications need to share user sessions and cookies.*

how it is currently. The most convenient way of accomplishing this is to write automated, browser-based acceptance tests using a tool like Behat[2]. Using Behat with Mink and Selenium WebDriver you can develop a stand-alone acceptance test suite for your application. This suite isn't directly tied to the code of your application in the same way unit or functional tests are, so they won't be thrown away as the refactor progresses. In fact, they'll be instrumental in ensuring your new work is keeping the application in a functionally acceptable state. Moving forward, new acceptance tests should be delivered with any new feature or bug fix in the existing codebase. This way, your test suite will grow naturally with your application. Development doesn't need to grind to a halt so you can backfill your tests.

Additionally, Composer[3], the PHP package management tool is an absolute must have for any modern PHP application. However, adding Composer to a legacy application isn't always as easy as you may think. Aside from managing your project's dependencies, Composer also makes available a very robust and configurable autoloading system. Older PHP applications may make use of the global, user-defined, `__autoload`[4] function to handle loading class definitions. If your application has an `__autoload` function defined, including the Composer autoloader (so you can load classes in vendor packages) will almost certainly break your existing autoloading scheme. Internally, Composer uses PHP's `spl_autoload_register`[5] function to register its own autoloading method. This function, `spl_autoload_register`, configures the PHP runtime to ignore any defined `__autoload` function in favor of the function or functions registered with it. Fortunately, `spl_autoload_register` can be invoked multiple times with different autoloading functions and each function will be placed onto an "autoloading queue." When loading a required class is attempted, the autoloading functions in the queue are invoked in the order in which they were added. If the autoloading function can load the class, it does so and returns `true`, otherwise, it returns false and the next function in the queue is attempted. Shimming an already existing `__autoload` function into one which can be registered with `spl_autoload_register` is a fairly trivial task as seen in the example below.

Note, though, as the refactoring project progresses, your application's reliance on this legacy autoloading function should diminish as new classes, along with existing code, become organized in a manner compatible with one of Composer's supported autoload schemes, such as the *PSR4 Autoloading*[6] standard. Eventually, you'll be able to remove this autoloading shim altogether.

Making Choices

The milestones you can set yourself should be chosen based on the overall value they will deliver to the project. Determining value isn't a subjective process. Each milestone you plan to deliver in this project can be weighed out and compared to each other by examining and evaluating the characteristics of those milestones.

As the value of each milestone is determined, the path your team will take modernizing the application will fall into place. During the project, however, the value of the milestones may (or, are likely to) change as you learn more about the code and how changes to it ripple across the application as a whole. Be prepared to reevaluate regularly and shuffle their order of execution. A dogmatic dedication to "The Plan" is likely going to work against you. Instead, try to adopt an agile approach to delivering this project, one where "The Plan" is limited to the milestone you're delivering at the moment and the next milestone is a suggestion which should be challenged before work on it begins.

LISTING 1

```
<?php
// the existing autoload function
function __autoload($classname) {
    $expected = APP_ROOT . "/classes/{$classname}.php";
    if (is_file($expected)) {
        include $expected;
    }
}

// registering the above function as the first autoloading
// functoon in the autoload queue
$throwErrors = true;
$prependAutoloader = true;
spl_autoload_register(
    function ($classname) {
        $expected = APP_ROOT . "/classes/{$classname}.php";
        if (is_file($expected)) {
            include $expected;
            return true;
        }

        return false;
    },
    $throwErrors,
    $prependAutoloader
);
```

Milestone Characteristics

Effort

How much effort do you think this milestone will take to deliver? Does it address a large, sweeping change with many touch points throughout the entire application or is it isolated to one or just a few areas?

Return on Investment

What does reaching this milestone accomplish for your application and your team? What is its impact? For instance, will reaching this milestone enable you to deliver new features to your users more quickly? Or, perhaps it will aid in improving the testability of the application.

2 *Behat:* http://docs.behat.org/en/v3.0/

3 *Composer:* https://getcomposer.org

4 `__autoload`: http://php.net/function.autoload

5 `spl_autoload_register`: http://php.net/function.spl_autoload_register

6 *PSR4 Autoloading:* http://www.php-fig.org/psr/psr-4/

Risk

How safe do you perceive the implementation of this milestone to be? Does it affect a core element of the application? Are the changes this milestone introduces have an appropriate amount of test coverage?

Opportunity Cost

What are you giving up by taking the time to deliver this milestone? For example, while you are working towards implementing it, will you be able to continue to deliver new or improved features to your users? Does the implementation require a freeze on those things? Will the time sunk into reaching this milestone offset the time it originally took to deliver whatever this milestone addresses enough warrant it?

Dependent Milestones

Does this milestone depend on other milestones to be reached before it can even be attempted? Are there milestones in the project which will make reaching this one easier? Additionally, will having reached this milestone aid in the delivery of others?

Don't Go It Alone

The landscape of PHP development has undoubtedly changed since the first lines of code were written for your aging application. Possibly, the most significant change is the introduction and adoption of web application frameworks. Frameworks promise to eliminate the work of writing the code required to deliver your application over the web. This includes code which interfaces with databases, sessions, HTTP requests, and so on. Being absolved of this work is meant to allow you to focus on delivering what makes your application unique amongst other web applications.

While the majority of the early PHP frameworks came as single, complete packages, the most recent iteration of them have taken a more modular approach. These frameworks are now delivered as a compilation of separate specialized packages, many of which can be used outside of the framework they belong to. One such framework being offered in this manner is Laravel. Since its first release in 2011, Laravel has quickly become the one of the most popular and successful PHP frameworks to date. It has done so through a combination of quality code and focus on developer experience. Laravel's goal[7] of "developer happiness from day 1" is achieved by providing a rich, consistent and intuitive interface to its many robust components. These components all exist under the Illuminate PHP[8] namespace.

Using the Illuminate components in your refactoring effort will add a level of reliability to the entire process. As you unravel the code in your application by separating the business logic from the hand rolled "framework" you'll be able to replace original "framework" functionality with these components. Doing so will likely also unlock additional abilities provided by the components your application was previously unable to perform. It's a win-win!

There's One More Thing

There's been a lot of work completed thus far, and you're so close to being ready to start the coding, but there is one more requirement: bootstrapping. Call it a pseudo Non-Negotiable Milestone, you're going to need a place where on each request to your application, you can bootstrap the Illuminate components you're bringing into the project. These are lower level components of the framework which are required to can use components like Eloquent models and database query builders, Mailers, Event Listeners and Queues.

To start, all your bootstrap code for Illuminate can be contained in a single file. As mentioned previously, including this bootstrap is required on each request to the application and the ease with which you can implement the include is going to depend on the existing structure of the application. If, for example, the application already has a bootstrapping system, the method of bootstrapping the Illuminate components should be no different than how other existing components are bootstrapped as well.

If your application doesn't have a Bootstrapping system, however, it may be a challenge to get Illuminate set up. Essentially, what you'll need to do is ensure the bootstrap is included at each entry point to the app. Applications implemented using a front controller, where there is only a single entry point, are in luck. Since there is only a single entry point, say `index.php`, the bootstrap can be included using a single require statement at the start of the file.

```
<?php
// index.php

require __DIR__ . '/../bootstrap/illuminate.php'
```

Unlucky applications, those with multiple (possibly even an unknown number of) entry points, present a unique challenge. There needs to be a guaranteed method which will ensure the bootstrap is included before the Illuminate components are accessed. A brute force approach, where a require statement is added to the top of each entry point, is an unwelcomed solution and should be avoided at all costs. The reason for this is, especially in situations where the number of entry points are unknown, it is possible an entry point could be missed which sets up the code to fail as soon as one of the Illuminate components is used. A much-preferred approach is to use the PHP initialization directive `auto_prepend_file`. This directive can be set to the bootstrap file which adds Illuminate to the application. On each request, PHP will read this directive and include the bootstrap file automatically before executing any other script.

```
# php.ini
auto_prepend_file = /path/to/bootstrap/illuminate.php
```

Bootstrapping the Illuminate Service Container

The single Illuminate component which will, effectively, be required by any other component is the Illuminate Service Container[9]. The Service Container will act as a Service Locator in most scenarios it will be used in during the refactoring

7 *Laravel's goal: http://phpa.me/laravelphp-dev-happiness*

8 *Illuminate PHP: https://github.com/illuminate*

9 *Illuminate Service Container: https://github.com/illuminate/container*

project.

Add it as a requirement to your `composer.json` file.

```
{
   "require": {
      "illuminate/container": "5.2.*"
   }
}
```

In your bootstrap file, configure a `Container` instance as shown here:

```
<?php

// bootstrap/illuminate.php
require "/path/to/vendor/autoload.php"

Illuminate\Container\Container::setInstance(
    new Illuminate\Container\Container
);
```

Hitting the Road

It's time to start authoring new code; your milestones are defined, and you've bootstrapped the Illuminate Service Container. The container is available to the application as a gateway to the other Illuminate components which you plan to use. What will they be?

Adding the Illuminate Request Object

The Illuminate Request[10] object offers an extremely intuitive and convenient abstraction over PHP's request related superglobals (`$_GET`, `$_POST`, `$_REQUEST`, `$_SERVER`, `$_COOKIES`, and `$_FILES`). It also gives many methods which provide useful information about the request itself, such as header information, which HTTP verb (method) was used and so on.

The `Request` object can be used to declutter code which deals with information coming in from the HTTP request parameters. Instead of many lines of ternary operators checking if indexes in the superglobal arrays are set, you can simply use the requests `input` method to access request parameters regardless of their source (query string/request body).

```
$id = isset($_GET['id']) ? $_GET['id'] : null;

// vs

$id = $request->input('id');
```

The request object can also be configured to resolve the currently authenticated user and it makes this available to the application via the `user` method. The user can be any object and doesn't have to pass any type hints constraints.

In addition, the request can be given access to the session (so long as the session object conforms to the `SessionInterface` of the `Symfony\Component\HttpFoundation\Session` namespace). Once the request has access to the session it can be used to flash data between consecutive requests and retrieve old input from the previous request.

Finally, an added benefit of using a request object over the PHP superglobals is the immediate gain in testability your application receives. In your test suite, mocks or test doubles of the request can be used in place of the actual request.

Adding the Request Object to the Service Container

Add the `illuminate/http` component to your `composer.json` file and update to install them.

```
{
   "require": {
       "illuminate/container": "5.2.*",
       "illuminate/http": "5.2.*"
   }
}
```

A bootstrap file like the one shown in Listing 2 will add the current request to the container.

LISTING 2

```
01. <?php
02. // bootstrap/illuminate.php
03. require "/path/to/vendor/autoload.php";
04.
05. use Illuminate\Http\Request;
06. use Illuminate\Container\Container;
07.
08. Container::setInstance(new Container);
09.
10. // add the current request to the service container
11. Container::getInstance()->instance(
12.     'request', Request::createFromGlobals()
13. );
```

Elsewhere in your application you can retrieve the current request by asking for `request`.

```
$request = Container::getInstance()['request'];
$id = $request->input('id');
```

Adding the Eloquent Database Capsule

The Illuminate Database Component[11] comes with all you need to use the Eloquent ORM and underlying Database Query Builder outside of a Laravel application. The stand-alone component is called Capsule. It is used to configure the various database connections your application may use and to "boot Eloquent" so the Eloquent ORM is in a ready state when your application code uses an Eloquent model.

Adding the Eloquent Database Capsule to the Illuminate Bootstrap Script

Add the `illuminate/database` component to your `composer.json` file and update to install them.

```
{
    "require": {
        "illuminate/container": "5.2.*",
        "illuminate/http": "5.2.*",
        "illuminate/database": "5.2.*"
    }
}
```

Listing 3 shows how to configure and boot the Eloquent capsule for use in your application.

10 Illuminate Request: https://github.com/illuminate/http

11 Illuminate Database Component: https://github.com/illuminate/database

LISTING 3

```
01. <?php
02. // bootstrap/illuminate.php
03.
04. require "/path/to/vendor/autoload.php";
05.
06. use Illuminate\Http\Request;
07. use Illuminate\Container\Container;
08. use Illuminate\Database\Capsule\Manager as Capsule;
09.
10. Container::setInstance(new Container);
11.
12. // add the current request to the service container
13. // ...
14.
15. // add the Illuminate Database Capsule
16. $capsule = new Capsule(Container::getInstance());
17. $capsule->addConnection([
18.     'driver'    => 'mysql',
19.     'host'      => 'localhost',
20.     'database'  => 'database',
21.     'username'  => 'username',
22.     'password'  => 'password',
23.     'charset'   => 'utf8',
24.     'collation' => 'utf8_unicode_ci',
25.     'prefix'    => '',
26. ]);
27. $capsule->setAsGlobal();
28. $capsule->bootEloquent();
29.
30. // clean up the global scope
31. unset($capsule);
```

If your application currently doesn't use an ORM, but instead relies on "raw" SQL queries, you may find migrating to use Eloquent models is difficult to do in a single step as the result sets returned from these queries typically contain data derived from multiple tables. While new features and code could be written to take advantage of the Eloquent ORM, it is advisable to replace the "raw" SQL queries with queries generated by the Illuminate Database Query Builder. Using the query builder will protect you better from SQL injection attacks which your application may be subjected to, see Listing 4.

Finally, if your "raw" SQL statements join many tables together but only select records from a single table in your database, you can take advantage of the Eloquent Query Builder to retrieve a collection of fully hydrated Eloquent models from your query, see Listing 5.

Adding the Illuminate Events Systems

The Illuminate Event Component[12] can be used to break up large procedural style portions of your application code into smaller, more defined, modules which react to events your application raises.

Having your application code separated in this manner, where each module has a clear responsibility, has many benefits. Primarily, you'll benefit from each modules ability to be tested in isolation as a unit.

12 *Illuminate Event Component:* https://github.com/illuminate/events

LISTING 4

```
01. <?php
02. //
03. // you can turn
04. //
05. $link = mysqli_connect("localhost", "username",
06.     "password", "database");
07.
08. $email = isset($_POST['email']) ? $_POST['email'] : null;
09. $query = <<<SQL
10. SELECT users.name, users.password, throttle.last_login,
11.        throttle.last_attempt, throttle.number_of_attempts
12. FROM users
13. JOIN throttle on users.id = throttle.user_id
14. WHERE users.email = '$email'
15. SQL;
16.
17. $results = mysqli_fetch_all(mysqli_query($link, $query));
18.
19. //
20. // into
21. //
22.
23. $request = Container::getInstance()['request'];
24. $email = $request->input('email');
25.
26. $query = Capsule::table('users')
27.     ->join('throttle', 'users.id', '=', 'throttle.user_id')
28.     ->where('users.email', $email);
29.
30. $results = $query->get([
31.     'users.name', 'users.password', 'throttle.last_login',
32.     'throttle.last_attempt', 'throttle.number_of_attempts'
33. ]);
```

LISTING 5

```
01. <?php
02. //
03. // you can turn
04. //
05.
06. $link = mysqli_connect("localhost", "username",
07.                        "password", "database");
08.
09. $query = <<<SQL
10. SELECT users.*
11. FROM users
12. JOIN throttle on users.id = throttle.user_id
13. WHERE NOT throttle.banned
14. SQL;
15.
16. $results = mysqli_fetch_all(mysqli_query($link, $query));
17.
18. //
19. // into
20. //
21.
22. $query = (new User)->newQuery()
23.     ->join('throttle', 'users.id', '=', 'throttle.user_id')
24.     ->whereNot('throttle.banned', true);
25.
26. $users = $query->get(['users.*']);
```

Preparing a test environment for a large, tangled, "does it all" type function can be practically impossible. However, as you separate and extract these elements into smaller, more cohesive classes and functions, testing them becomes possible. Since the modules do less on their own, they are easier to understand and thus easier to prepare a test for.

Additionally, having smaller modules focused on delivering a single outcome gives you an opportunity to reuse and combine them in ways you may not have considered previously.

Adding the Illuminate Event System to the Application

Add the `illuminate/events` component to your `composer.json` file and update to install them.

```
{
    "require": {
        "illuminate/container": "5.2.*",
        "illuminate/http": "5.2.*",
        "illuminate/database": "5.2.*",
        "illuminate/events": "5.2.*"
    }
}
```

In Listing 6, we add the `events` component to our container.

LISTING 6

```
01. <?php
02. // bootstrap/illuminate.php
03.
04. require "/path/to/vendor/autoload.php";
05.
06. use Illuminate\Http\Request;
07. use Illuminate\Container\Container;
08. use Illuminate\Database\Capsule\Manager as Capsule;
09. use Illuminate\Event\Dispatcher;
10.
11. $container = new Container;
12. Container::setInstance($container);
13.
14. // add the current request to the service container
15. // ...
16.
17. // add the Illuminate Database Capsule
18. // ...
19.
20. // add the Illuminate Dispatcher
21. $container->instance('events', new Dispatcher($container));
22.
23. // clean up the global scope
24. unset($capsule, $container);
```

If you look at these large swaths of code you can probably determine what the original intent of it was. The function's name is probably a very good clue. Take, for instance, a function called `register_new_user` which creates new users and then sends them a welcome email. The core responsibility of this method is to create a new user for your application, sending the welcome email is ancillary to the process. You can use the Illuminate Event Component to pull apart these functions, returning them into leaner, more focused actors.

At its simplest implementation, an event listener can be registered with the Event Dispatcher as an anonymous function. Using anonymous functions as listeners allows you to defer developing a larger more complex event system until you actually need it. As your system uses more and more events, the requirements of what is needed by the event system will become more distinguishable and you reduce the chances of building something which doesn't suit your needs.

```
// an example listener reacting to a new user being
// registered

$events = Container::getInstance()['events'];
$events->listen('user.created', function ($user) {
    // compose the welcome email for the user
    //...

    // send the welcome email
    mail($user->email, 'Welcome to MyApp', $message);
});
```

Database Events

The Illuminate Database Component automatically uses the Events Component if one is available. Both the Query Builder and Eloquent raise events to notify the application about the things it is doing or has done.

Registering the Query Log Shutdown Function

As an example, it is quite simple to create a database query logger for your application which you can use for debugging. All that is required is registering a shutdown function for PHP to execute at the end of your script execution.

```
register_shutdown_function(function () {
    $queries = array_column(Capsule::getQueryLog(), 'query');
    $log = LOG_DIR . '/query.log';

    file_put_contents($log, \
                      implode("\n", $queries), FILE_APPEND);
});
```

Logging All Database Queries

If you wish to log every query your application sends to the database then when you set up Capsule, simply call the `enableQueryLog` method on it.

```
// bootstrap/illuminate.php

// add the Illuminate Database Capsule
// ...
$capsule->enableQueryLog();
// ...
```

Logging Specific Queries By Fencing

Otherwise, you can fence the database query with calls to `enableQueryLog` and `disableQueryLog` on the Capsule (see Listing 7).

Adding the Illuminate Hasher

It is all too common for legacy applications to use either plain text passwords or weak MD5 hashes. The Illuminate Hasher Component[13] can be used to beef up the security of your application's user passwords. Migrating the application to use cryptographically secure passwords can even be done transparently to end users.

Adding the Illuminate Hasher to the Service Container

Add the `illuminate/hashing` component to your `composer.json` file and update to install them.

```
{
    "require": {
        "illuminate/container": "5.2.*",
        "illuminate/http": "5.2.*",
        "illuminate/database": "5.2.*",
        "illuminate/events": "5.2.*",
        "illuminate/hashing": "5.2.*"
    }
}
```

In Listing 8, we register the component with our container as `hasher`.

LISTING 8

```
01. <?php
02. // bootstrap/illuminate.php
03.
04. require "/path/to/vendor/autoload.php";
05.
06. use Illuminate\Http\Request;
07. use Illuminate\Container\Container;
08. use Illuminate\Database\Capsule\Manager as Capsule;
09. use Illuminate\Event\Dispatcher;
10. use Illuminate\Hashing\BcryptHasher;
11.
12. $container = new Container;
13. Container::setInstance($container);
14.
15. // add the current request to the service container
16. // ...
17.
18. // add the Illuminate Dispatcher
19. ..
20.
21. // add the Illuminate Database Capsule
22. // ...
23.
24. // add the Illuminate Hasher
25. $container->instance('hasher', new BcryptHasher);
26.
27. // clean up the global scope
28. unset($capsule, $container);
```

13 *Illuminate Hasher Component:* https://github.com/illuminate/hashing

LISTING 7

```
01. Capsule::enableQueryLog();
02.
03. $query = Capsule::table('users')
04.     ->join('throttle', 'users.id', '=', 'throttle.user_id')
05.     ->where('users.email', $email);
06.
07. $results = $query->get([
08.     'users.name', 'users.password', 'throttle.last_login',
09.     'throttle.last_attempt', 'throttle.number_of_attempts'
10. ]);
11.
12. Capsule::disableQueryLog();
```

Migrating From Plain Text to Hashed Passwords

If your application is in the state where passwords are stored in plain text it is almost trivial to migrate your application to one which uses hashed passwords, provided you have the ability to take downtime.

In a situation like this, all that is required is to take the application down and use a one-time script to iterate over your user passwords hashing them and then updating the corresponding user records.

Of course, you will also have to update your new user and credential checking code to make use of the Hasher.

LISTING 9

```
01. // update all plain text user passwords to bcrypt hashed
02. // passwords
03.
04. require "/path/to/bootstrap/illuminate.php";
05.
06. Capsule::transaction(function () {
07.     $hasher = Container::getInstance()['hasher'];
08.     Capsule::table('users')->get()->each(
09.         function ($userRecord) use ($hasher) {
10.             Capsule::table('users')
11.                 ->where('id', $userRecord['id'])
12.                 ->update([
13.                     'password' =>
14.                 $hasher->make($userRecord['password'])
15.                 ]);
16.         }
17.     );
18. });// update all plain text user passwords to bcrypt hashed
19. // passwords
20.
21. require /path/to/bootstrap/illuminate.php
22.
23. Capsule::transaction(function () {
24.     $hasher = Container::getInstance()['hasher'];
25.     Capsule::table('users')->get()->each(
26.         function ($userRecord) use ($hasher) {
27.             Capsule::table('users')
28.                 ->where('id', $userRecord['id'])
29.                 ->update([
30.                     'password' =>
31.                 $hasher->make($userRecord['password'])
32.                 ]);
33.         }
34.     );
35. });
```

Migrating an Application Without Downtime

It is possible, yet decidedly more difficult, to migrate an application without downtime. The entire process must span multiple deployments and extra work needs to be done in the code which manages the user's password verification.

The First Deployment

The first deployment of the code is going to be a database migration which adds a column to the users database table which specifies the password hashing scheme currently in use. These could be values such as 'plain text' or 'md5'. This column is going to be transient; it will only exist while the migration is happening. As such, a plain VARCHAR column is suitable.

```
-- update the users database table with hashing_scheme column

ALTER TABLE users
ADD COLUMN 'hashing_scheme' VARCHAR(10) NOT NULL DEFAULT 'md5';
```

The Second Deployment

The second deployment of the code will include new code as in Listing 10 which, when verifying passwords will respect the `hashing_scheme` used by the user record. Again, this code will only exist as long as the migration is in progress, a simple "if-else" ladder will suffice.

Finally, as well as password validation, this deployment must include code which stores new user passwords as bcrypt'd hashes.

LISTING 10

```
01. function validate_user_password($user, $plainPassword, $scheme) {
02.
03.     $hasher = Container::getInstance()['hasher'];
04.
05.     if ($scheme === 'md5') {
06.         $passes = md5($plainPassword) == $user['password'];
07.
08.         if (!passes) {
09.             return false;
10.         }
11.
12.         // update the user with the hashed password
13.         Capsule::table('users')
14.             ->where('id', $user['id'])
15.             ->update([
16.                 'hashing_scheme' => 'bcrypt',
17.                 'password' => $hasher->make($plainPassword)
18.             ]);
19.     }
20.
21.     return $hasher->check($plainPassword, $user['password']);
```

Adding Facades

The Illuminate Facades come as part of the Illuminate Support Component[14] and are a clever way of providing what seems to be "static" accessors to components which are bound in the Illuminate Service Container. Adding Facades eliminate the requirement to constantly be accessing the container with:

```
Container::getInstance()
```

14 *Illuminate Support Component:* https://github.com/illuminate/support

Each Facade defines its own "accessor" which is a unique name which can be used to resolve its actual concrete implementation from the Service Container. For instance, the `Event` Facade defines its accessor as "events". When the Event Dispatcher Component is bound to the container it is bound to the property "events" which matches the Facade accessor. Any method on the dispatcher can then be invoked using `Event::methodName($param1, $param2)`.

You can register a listener without a Facade:

```
$events = Container::getInstance()['events'];
$events->listen('my.event', function ($foo, $bar) {
    // listener body
});
```

You can also listen for events with a Facade:

```
Event::listen('my.event', function ($foo, $bar) {
    // listener body
});
```

Injecting the Service Container Into the Facade

Since each Facade's concrete implementation is bound by its accessor in the Service Container, the Facade system requires access to the Service Container. This can be accomplished easily in the Illuminate bootstrap.

Facades as Adapters

It's possible to use Facades as Adapters to a legacy object or objects which you wish to replace with Illuminate Components but are unable to for some reason or another. An adapter class can be written which exposes some or all of the same API as the targeted Illuminate Component and adapts those calls to the existing API of the legacy objects.

As an example, in a Laravel application, the Auth Facade provides access to the AuthManager which, in practice, acts as a "go-between" for your application and the active Authentication Guard. The Guard exposes a useful API for many authentication related activities such as password validation, accessing the logged in user, and logging in and out of users.

An adapter could be written which mimics the `attempt`, `user`, `login` and `logout` API of the Illuminate Auth Guard class. The adapter would translate these calls to something the existing authentication system understands.

Using Facades as Adapters can be used to position your code into a place where once it is possible to replace those legacy objects, doing so will not require much change to your existing code.

LISTING 11

```
01. <?php
02. // bootstrap/illuminate.php
03.
04. require "/path/to/vendor/autoload.php";
05.
06. use Illuminate\Http\Request;
07. use Illuminate\Container\Container;
08. use Illuminate\Database\Capsule\Manager as Capsule;
09. use Illuminate\Event\Dispatcher;
10. use Illuminate\Hashing\BcryptHasher;
11.
12. $container = new Container;
13. Container::setInstance($container);
14.
15. // add the current request to the service container
16. // ...
17.
18. // add the Illuminate Dispatcher
19. // ..
20.
21. // add the Illuminate Database Capsule
22. // ...
23.
24. // add the Illuminate Hasher
25. // ...
26.
27. // inject the container into the Facade system
28. Facade::setFacadeApplication($container);
29.
30. // clean up the global scope
31. unset($capsule, $container);
```

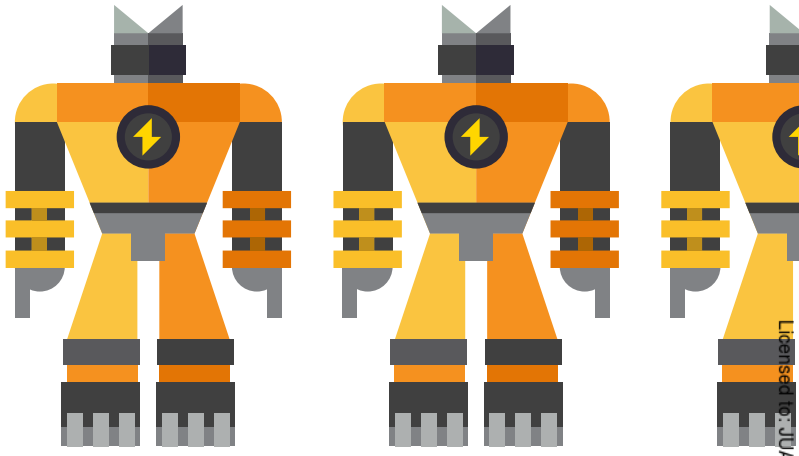
Getting to the End of the Road

Large scale refactoring projects are surely one of the most difficult and therefore most rewarding challenges a developer can face. The road will be bumpy, probably with many detours along the way, but it is important to stay steadfast in your dedication to the project.

The Laravel framework has many attractive features but as you've seen, incorporating the components which make those features available in your project is possible, and certainly easier than starting down a completely different road with unknown obstacles.

Colin is a senior developer with Vehikl, a full services software consultancy, in Waterloo, Ontario, Canada and co-organizer of the GPUG PHP Users Group. With over a decade of professional experience, Colin is never put off by a good challenge. He enjoys refactoring often neglected legacy code bases as much as tackling the wide open space of green field projects. No matter the context, he puts great effort into delivering simple solutions with as little code as possible. @colindecarlo

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

```
01. <?php
02.
03. require_once(__DIR__ . '/../autoload.php');
04.
05. // The router of your choice would be setup here and
06. // would try to match requests based on your mapping of
07. // requests to controllers and actions
08.
09. if ($routerFoundMatch === true) {
10.     $controller->action();
11. } else {
12.     $this->redirect('/404/');
13. }
```

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

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.con
            routing.yml
        www.website-b.con
            routing.yml
        www.website-c.con
            routing.yml
        www.website-d.con
            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
```

include were modified to account for the new directory layout, 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.

LISTING 2

```
01. <?php
02.
03. require_once(__DIR__ . '/../autoload.php');
04.
05. $yourConfig->getSettings('../settings/production.yml');
06. if ($developmentEnvironment === true)  {
07.     $yourConfig->getSettings('../settings/development.yml');
08. }
09.
10. // Get website specific settings
11. $yourConfig->getSettings('../settings/' . $nameOfSite
12.                          . '/settings.yml');
13.
14. // The router of your choice would be setup here and
15. // would try to match requests based on your mapping of
16. // requests to controllers and actions with settings
17.
18. if ($routerFoundMatch === true) {
19.     $controller->action();
20. } else if ($phpFileTypeFound !== true) {
21.     $this->include($uri . '.php');
22. } else {
23.     $this->redirect('/404/');
24. }
```

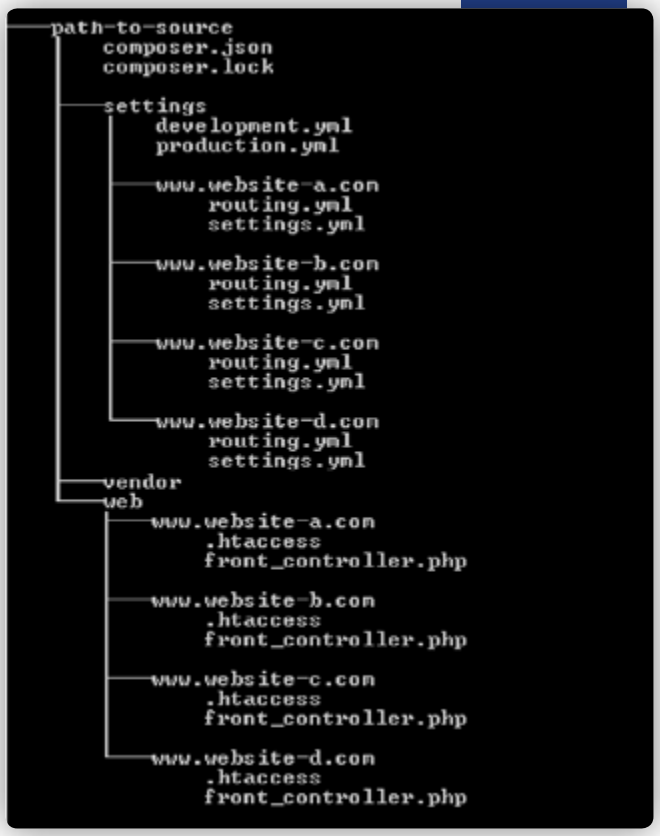
Template System and Future Updates

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.

Directory Layout with Configuration Files **FIGURE 3**



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?

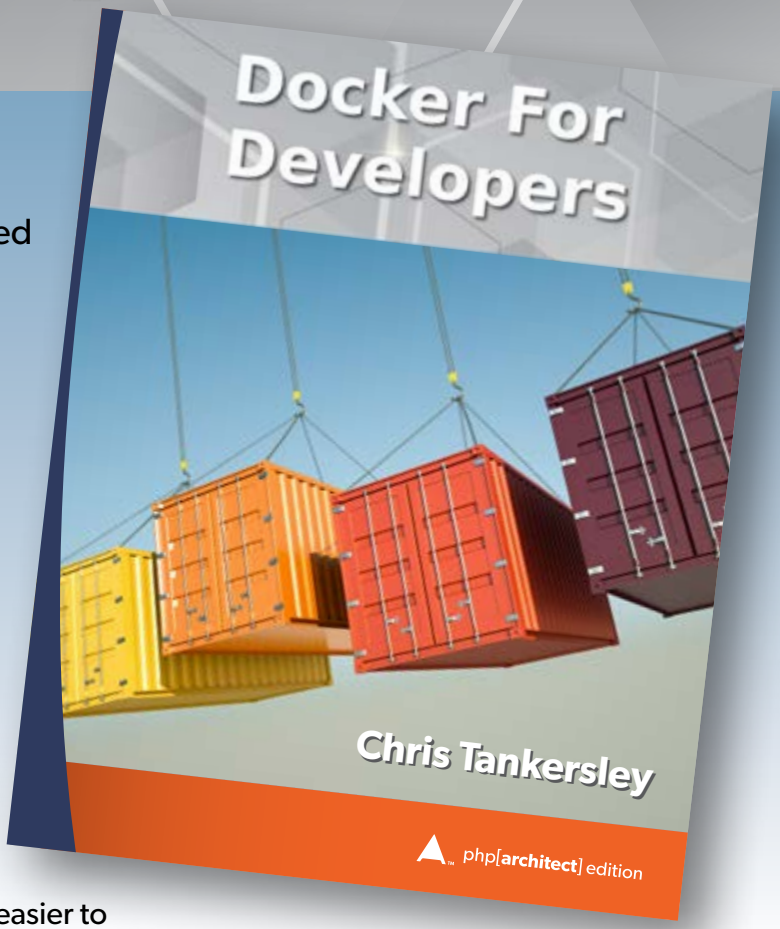
Jack is a software engineer for the Graduate College at Iowa State University. His past is a mix of academics and industry with a Master of Science in Human Computer Interaction, two and a half years at Opticsplanet.com, and a Bachelor of Science in Web and Digital Media Development. @jack_polifka

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

Docker For Developers

by Chris Tankersley



Docker For Developers is designed for developers who are looking at Docker as a replacement for development environments like virtualization, or devops people who want to see how to take an existing application and integrate Docker into that workflow. This book covers not only how to work with Docker, but how to make Docker work with your application.

You will learn how to work with containers, what they are, and how they can help you as a developer.

You will learn how Docker can make it easier to build, test, and deploy distributed applications. By running Docker and separating out the different concerns of your application you will have a more robust, scalable application.

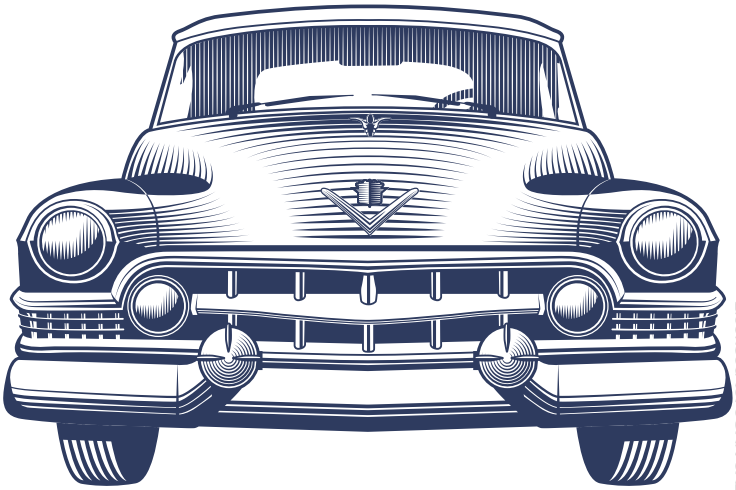
You will learn how to use Docker to deploy your application and make it a part of your deployment strategy, helping not only ensure your environments are the same but also making it easier to package and deliver.

Purchase

http://phpa.me/docker4devs

Legacy Code Needs Love Too

John Congdon



The latest framework of the moment usually gets all the press, but these usually only work when you are starting with a brand new project. There are millions of lines of code out there which are not new and need loving developers like you to help take care of them.

My whole career has involved legacy code. You know… code written without regards to best practices, possibly copy and pasted from who knows where; or, code is written with the intention of going back to clean it up later, but has to be done NOW. The sad part is, this even includes much of the code I have written from scratch. Before I started attending conferences like php[tek] and joining my local user group, I was writing legacy code some poor soul is maintaining today.

Also, as a side note, legacy code is often used in a derogatory manner, but doesn't have to be. Good code written prior to today's "best practices" is still considered legacy. Sometimes we want to bring that good code up to today's standards so we can continue to enhance it and make it more functional. For example, you could have a great application from 15 years ago which does exactly what it is supposed to do, but is storing its passwords either in clear text or just as an MD5 hash. Are you going to leave it that way, just because the code works? I hope not.

What Is Love?

To me, love for a codebase is making it maintainable, and giving it a chance to grow into a mature, preferably current codebase. There are a few ways we are going to get there, and it takes time. A lot of time.

I have a few reasons I prefer to update a legacy codebase before I do anything else.

Numerous times I have fixed a reported bug, tested it, had the client sign off, and then released the fix into production only to later discover some new bug as a result of the fix. My excuse is always sad, "Crap my fix over here broke that, but I didn't even think to test for it. I'm sorry…"

Newer frameworks often do a lot of the heavy lifting, so adding new features to a website becomes much easier. Especially with newer tools like Composer, Bower, and Gulp. Updating the codebase can take awhile, but the time saved afterward can be considerable, even if it's hard to quantify.

Also, well-constructed code is easier to modify as business needs change. How many times are you told how the client wants something only to have it change later? Take the time to architect code up front, and you will be rewarded for it later.

Our goals are going to be:

1. Make sure we have a development environment.
2. Have a repeatable and automated build process.
3. Create a test suite.
4. Refactor.
5. Wrap your code in a warm Laravel-blanket of love.
6. Reap the rewards of immediate wins.

Ensure We Have a Development Environment

I admit, this may sound crazy, and you are probably laughing right now because surely everyone has a development environment. Believe it or not, I have seen and been part of many codebases which were not only maintained but *updated* in production.

Please tell me you are laughing right now.

If not, read on.

Sometimes this takes a very long time to accomplish, but with enough perseverance, you can make it happen. I use Mac OS for my everyday machine, however, my apps in production will run on some flavor of Linux. Vagrant to the rescue. Vagrant is a tool which allows me to spin up virtual machines on my Mac. This sounds way more complicated than it is. Actually, Vagrant is kind of a wrapper around other tools, but the docs do a great job of telling you what you need to get started. If you have not heard of or used Vagrant, please start at *Why Vagrant?*[1].

> *While I chose Vagrant, there are other options available. The new kid on the block is Docker which I would like to learn more about soon. I recently purchased Chris Tankersley's Book, Docker for Developers*[2].

Depending on the age of your project, matching your envgironments can be challenging. The old adage is your development environment should match your production environment exactly. It does

1 *Why Vagrant:* https://www.vagrantup.com/docs/why-vagrant

2 *Docker for Developers:* https://www.phparch.com/books/docker-for-developers/

make it easier, but sometimes finding the exact version of the operating system, or older versions of specific packages can be tricky. The key is to make it match as closely as possible. You will want to match your version of PHP, and ideally, all of the same PHP extensions installed. That alone should limit your "it works for me" scenarios. You will probably want to run the same web server, but versions are a little less important. Ideally, your code will not be aware of the web server being used, but I've seen code which relies on server environment variables which are not provided the same way across different web servers such as Apache and Nginx. There may be other components (such as caching layers, database servers, etc.) and you will have to decide if specific versions matter.

Another potential nightmare which may arise is that many "seasoned" administrators may revel in the fact they compiled everything from scratch. Trying to make sure you compiled everything with the same options can leave you feeling like you belong in an asylum. The good news is you are building a development environment, meaning things can be broken for you, as long as they aren't when it's time to go to production.

I can't stress how important this step is. If you are having issues with this, seek help. There are lots of tools to help you build these types of environments like PuPHPet.com[3], Phansible.com[4], Homestead[5] (if you are using Laravel), and Drupal VM[6]. Don't feel like you are alone; someone will be willing to help. Join the conversation in a Slack channel. Many communities such as ours, have embraced Slack and use tools to allow you to join a team automatically. Try http://slack.sdphp.org as a starting point.

Have a Repeatable and Automated Build Process

Once you have a development environment, getting a build process is also important. I have spent many hours Googling things like "how to create a PHP build script" and "best practices to deploy PHP code." The truth is, there is not a one-size-fits-all answer. A build script is merely the things you learn along the way, which your specific code base needs in order to run properly.

There are almost too many tools out there making this even more confusing. Start simply by writing down anything you do to make code live.

Don't over think this. Start slowly and don't think you have to write tests for the entire system at this point.

If you work in production, then you just code, right? So when you stop working in production, and you start only coding on your new development server, your build process will simply be one step *copy the code from your development server to your production server*. How you do this is up to you (SCP, rsync, git pull, SFTP). This will change over time as your process changes; let it, don't feel tied to a specific formula.

You may find yourself using Composer to include packages or libraries you want to use. The first time you deploy code which relies on this library, you may forget to run `composer install`. Run it quickly to stop the errors, and then immediately add it to your build script. It is now a necessary step you do not want to forget again.

Repeat this cycle. Hopefully, you're learning to add things to your build script before the errors happen in production. Again, seek advice.

Create a Test Suite

We want to ensure any changes we make will not create new issues. This can be any type of test you are comfortable writing which ensures current functionality. If you have gotten the first two parts of this down, then something as simple as Codeception[7] may be the answer. It's a good starting point, especially if you can scrub your database every time it runs. When I say scrub, I don't necessarily mean delete everything, but I do mean getting it back to the same starting state. Every single time.

Don't over think this. Start slowly and don't think you have to write tests for the entire system at this point. The goal of this process is to refactor your old code into a more modern codebase. As you refactor, you are going to begin by writing tests for the code you want to change first. It's amazing how your tests will start to pile up.

Refactor

Refactoring should be a whole topic itself. It's an art form that once you master, will take you a long way into making your code better. You should be comfortable refactoring, because it will allow you to prototype projects fast. Projects in their early stages need to work just enough for a proof of concept. Once they have been proven enough to make money, you then spend the time to refactor and make them maintainable. Putting that kind of time into a project up front can be a waste if the project will be scrapped because it isn't profitable. But, this article isn't about new projects, is it?

So, how do you get management to bite on the fact that you need time (and essentially money) to refactor a project? It all depends on the need. If you are maintaining old code and often find yourself trying to explain to them why fixing problem A caused problem B, then it should be super easy to sell them on the idea.

When I was the only developer in a company, I created a simple presentation I used to convince the CEO a refactor was necessary. I had recently been to php[tek] and learned a ton from the experience; I did a little analysis of potential cost savings. I tried to show that by refactoring, I would be able to add new features in the future without causing other issues. New features could be added x% faster; they would be y% more reliable, etc. I had to show value for something that is hard for the people writing checks to see.

This was easier for me because I was the only developer. If you are on a bigger team, hopefully, everyone agrees with you. This isn't always the case, so how do you convince them? I currently have a project I can't move forward on because other developers don't think we can invest the time to make a huge change. I

3 *PuPHPet.com:* https://puphpet.com

4 *Phansible.com:* http://phansible.com

5 *Homestead:* https://laravel.com/docs/5.2/homestead

6 *Drupal VM:* https://www.drupalvm.com

7 *Codeception:* http://codeception.com

have started slowly, by convincing them we need to start with small improvements, such as adopt coding standards. I've gotten them to switch from SVN to GIT, and we are now using tools such as GitHub to do code reviews. All of these smaller milestones will eventually help us tackle the bigger issues. And, since those milestones have already been tackled, the bigger issue is more manageable.

Wrap Your Code in a Warm Laravel-Blanket of Love

My goal in many of my past projects was to get the code into a framework I could use. This was often easier said than done.

I used Laravel in my title as homage to Eric Van Johnson who titled my presentation for the San Diego Laravel User Group[8]. But this technique should be usable with any framework.

Assuming your code does not have a front controller, in other words, is written in the old style of a script per page, this works great. For example, if your website has an "about us" page at `http://example.com/about-us.php` and you look in your document root and have a file called `about-us.php`, then this technique is perfect.

All of the current frameworks run from a front controller, usually `index.php`. This will cause an issue if you already have a file with the same name. Solve this by creating an empty project somewhere else, and then move the files into your current project. Just rename index.php to something like laravel.php.

Next, the framework will have a snippet for you to copy/paste into an Apache `.htaccess` file, or an Nginx configuration file. Just change `index.php` in these examples to `laravel.php`.

The key here is if the file doesn't exist on the file system, then the framework is fired up to generate the content.

My process would be to choose a page, and start to convert it to Laravel. Let's look at an about-us.php page, since I have already used that example. I would create a new route in Laravel called `/about-us`, without the `.php`. Obviously, this is a contrived example; you would build out the entire page, probably using blade templates.

```
<?php
Route::get('/about-us', [
   'as' => 'profile',
   'uses' => 'PageController@showAboutUs'
]);
```

Once I can go to that URL in my browser, and I am happy with it, I start the process of conversion. There are two options here.

You can delete the `about-us.php`, and make it another route. The benefit is you don't have to change this URL on any other pages. The downside is you will have duplicated routes.

```
<?php
Route::get('/about-us.php', [
   'as' => 'profile',
   'uses' => 'PageController@showAboutUs'
]);
```

8 *San Diego Laravel User Group:* http://sdlug.com

The second option is to create 301 redirects. However, if you are not changing the links in the rest of your code, you are causing people to make two requests each time they visit the page.

Unfortunately, there is a ton of work to do at this point. You have to get your templates set up so you can start bringing features over. You will also have to bring over some of the core functions currently being used. This is a chance to refactor and improve the code. I have been able to use this approach to refactor functions which were 500+ lines of code. Knowing how and what to refactor is a process in itself. Take time to architect your application now. Break up functionality, if possible, into your own libraries. With tools like Composer available now, you should look for 3rd party libraries to do most of the work for you, even if you already have code that works. Many of the libraries are open source and constantly improving, so you can get security updates, new features, and keep up with API changes, etc. much more easily. And, you have the option of contributing back to those communities if you find ways to improve their library.

Reap the Rewards of Immediate Wins

Once you have your code working side by side in a framework, you get a few things you can use immediately. Granted, you don't have to use a framework for them, but they are a side benefit.

Database migrations. If you don't already have a system to maintain your migrations, then start using this feature of your chosen framework right away. Knowing how your DB changes over time, and being able to create new instances of your DB for development and testing is very handy.

Packages built for the specific framework. There are plenty of free standing libraries, but sometimes you find ones built specifically for your framework of choice which makes integrating it super easy.

Conclusion

Legacy code can be frustrating. The goal here is to show you a path I have used to get a codebase updated. It's by no means a fast process, and should not be rushed. Most of us have to walk the fine line between working on the existing stuff that is a passion of ours and the code which will increase revenue and/or profits. We often have little time to improve existing code that is not completely busted. So embrace the process, take a deep breath in, hold it, hold it, now breathe out slowly.

John Congdon is CEO of DiegoDev Group, LLC, a web application development firm, in San Diego, CA. John is a co-organizer of San Diego PHP (SDPHP) and active member of the community. He is passionate about bringing his skills and those of the people around him to a higher level. @johncongdon

Building for the Internet of Things in PHP

Adam Englander

The Internet of Things, or IoT, is the big buzzword nowadays. You can't escape it. It's on the news and everywhere in tech magazine sites. Unfortunately, little of the IoT buzz seems to have made it to the PHP community. We are late to the party. The good news is the party has really just begun, and there's still plenty of room for PHP. If you are like most PHP developers, you probably thought IoT was only for C or JavaScript developers. If so, you'll be surprised to discover you are dead wrong. IoT development is not only possible in PHP; it's fairly easy to do in PHP as well. All it takes is a little education and desire to learn.

What Is the Internet of Things?

The Internet of Things is used to describe a collection of "smart" devices which can communicate with each other and the outside world. You have probably seen (or, at the very least read about) connected cars, smart appliances, or home automation. These are all considered part of the Internet of Things. The number of "smart" devices grows every day and the growth is accelerating year in and year out. It's a dynamic multi-billion-dollar segment of the new tech economy. It's also a niche segment of the geek world where the physical intersects with the virtual in the realm of the "Makers".

What Are Makers?

Makers is a term used to define a group of enthusiasts who like to build things. They build all sorts of things but mostly things driven by electronics and software. They wire together microcontrollers, servos and sensors, and write software to orchestrate the behavior of all of the bits and pieces. Makers have been around for a very long time. Thirty years ago, these were the people building televisions or computers from a Heathkit[1]. Today, it's drones, robots, and home automation. The affordable home computer created the hacker culture of hobbyist software programmers. The number of Makers in today's world is simply exploding. Today, powerful and affordable IoT platforms are fueling the rise of Makers building anything their imagination can dream up.

IoT Platforms

There are a number of IoT platforms Makers can use for building devices. They are separated in two major categories, microcontrollers and what I like to call "tiny computers." Microcontrollers have been around the longest and have a large eco-system built around them. "Tiny computers" have recently become a viable platform for IoT and are often used to provide the central brain to control a collection of integrated circuits and microcontrollers. The multi-purpose nature of "tiny computers" and their all-in-one feature set have been driving their popularity.

Microcontrollers

In the Microcontroller world, Arduino[2] is probably the most recognized brand. Arduino makes a number of microcontrollers and have their own IDE for writing C and publishing the compiled code to their boards. Arduinos, and other boards like them, are usually tied to a single language which is usually JavaScript, Java, or C. The boards are very fast but require a fair bit of learning for PHP programmers and very little applies to day-to-day PHP development.

Tiny Computers

"Tiny computers" are a class of computers which are physically small enough to embed into a smart device. They tend to use Linux kernels for their operating system. Some are full blown kernels and others use micro-kernels. Microsoft has been trying to break into the space and has released Windows 10 IoT[3]. It remains to be seen if they will achieve any real adoption in the IoT space. With respect to the boards themselves, the dominating two are Raspberry Pi and Intel Edison.

The Raspberry Pi[4] was originally developed as an inexpensive training tool for children. Its 35 USD price point has made it a popular choice for hobbyists and IoT developers building prototypes. With the recent introduction of the Zero–if you can find one–at around 5 USD is a true IoT sized inexpensive Raspberry

1 *Heathkit:* https://en.wikipedia.org/wiki/Heathkit

2 *Arduino:* https://www.arduino.cc

3 *Windows 10 IoT:* https://developer.microsoft.com/en-us/windows/iot

4 *Raspberry Pi:* https://www.raspberrypi.org

Pi. At the time of this article, the latest version, Raspberry Pi 3, boasts the following specifications:

- 1.2GHz 64-bit quad-core ARMv8 CPU
- 802.11n Wireless LAN
- Bluetooth 4.1
- Bluetooth Low Energy (BLE)
- 1GB RAM
- Micro SD card slot (now push-pull rather than push-push)
- VideoCore IV 3D graphics core

The Intel Edison[5] was built from the beginning to be an IoT board. It is tiny yet powerful. At the time of this article, the current board boasts the following specifications:

- Intel® Atom™ dual-core processor at 500 MHz
- 1 GB DDR3 RAM
- 4 GB eMMC Flash
- 802.11n Wireless LAN
- Bluetooth 4.0

Intel has two boards in the Edison family. They have a break-out board which is extremely small, but requires soldering of connections. They also have an Arduino board which is fantastic for prototyping. The board for Arduino matches the pin outs for the Arduino Uno so you may use it in conjunction with the Uno backplanes and connectors in the Arduino ecosystem. Additional boards are also available from third party vendors. The Edison is my personal favorite board for Python and Node.js based projects. I find it to be the best of all worlds and the most flexible when dealing with power requirements. I have not used an Edison for a PHP project. At the time of this article, no package for PHP is available in the supported repositories. One day, I will document an install from source for PHP 7.x.

Getting Started

The remainder of this article will be a step-by-step guide to utilizing GPIO on a Raspberry Pi. At first, we will use the Linux shell and move to PHP. At the end, we will build a protocol server with icicle.io. Icicle.io is an emerging PHP based asynchronous event loop. At the time of this article, it is the only such library actively maintained.

> *General-purpose input/output (GPIO) are pins on an integrated circuit which are controllable by the user at run time. The user controls if it's an input or output pin. Effectively, you can think of them as switches you can turn on or off and use them to send signals to another device or receive a signal and act on it.*

Setting Up Your Raspberry Pi

Required Hardware

For hardware, you will need the following items which should cost less than $50:

- Raspberry Pi: any version will work. The examples are for a Model B. The images are for a Raspberry Pi 2.
- A micro USB cord and optional USB wall plug: this cord is for power and usually comes in the package when you buy your Raspberry Pi.
- An SD Card compatible with your Raspberry Pi: this usually comes in the package with your Raspberry Pi.
- SD Card Reader (maybe): this will be for flashing the SD card with your operating system or operating system loader. You can purchase the Raspberry Pi pre-loaded with Raspbian or simply order a Raspbian SD card. You can decide your preference. If you have an SD Card reader, you can ensure you have the latest version of Raspbian without having to upgrade the operating system from the pre-installed version.
- HDMI or Composite Display with Cable: Raspberry Pi's are full computers and need to be connected to a display for the initial setup. I just use my TV. No reason to spend the money on a new display.
- USB Keyboard: A keyboard is required to interact with the terminal.
- USB Mouse (optional): If you really want to use the GUI interface, it'll come in handy. It's not necessary if you know the key combinations for accessing the menus.
- Cat 5 Network Cable or USB Wi-Fi dongle. Wi-Fi should be used as a last resort. If you are awesome at configuring Wi-Fi on Linux, go for it. It has been nothing but trouble for me.
- 3.3 volt LED. You'll want to get a colored LED and be sure it's bright. If it's clear or not bright, you may bang your head on the keyboard because your LED is not lit, only to discover it was lit the whole time. You were just unable to distinguish the difference between off and on. I speak from experience on this little gem.
- Female to female wire leads. The pins on the Raspberry Pi are male and so is your LED. This is how you wire up the LED. (May not be necessary for Raspberry Pi Zero. It has female connections by default.)

If you are an electrician or an electronics wiz, you may be asking "where is the resistor?" If you want to place a resistor inline, feel free. The tutorial does not require one because not having a resistor won't damage the Raspberry Pi, and I have yet to burn out an LED. If you do burn out an LED, you probably have another because you normally get them in packs of twenty-five.

Required Software

Operating System

Your Raspberry Pi will need a Linux operating system. Any flavor should work, but I recommend Debian for two reasons. First, Debian is the only Linux variant officially supported for a Raspberry Pi by the Foundation. Second, all of my examples will be based on Debian. I won't include instructions on how to set up Debian on the Raspberry Pi. It's a fairly simple process and well documented by the Raspberry Pi Foundation on their website: https://www.raspberrypi.org/downloads. It is important to use Debian Precise or later. Squeeze and Wheezy are old and may not have PHP 5.6 or higher available via a package install. We are using the Raspberry Pi because of the availability

5 *Intel Edison:* https://software.intel.com/en-us/iot/hardware/edison

of the PHP packages. If we were going to build PHP from scratch, we'd be using a much cooler device like an Intel Edison which is 1/5th the size and only requires an LED for what we are building. Maybe that will be another article. Only time will tell.

> **Editor's Note:** *See the January 2016 issue[6] to learn how to install Debian, Apache, and PHP on a Raspberry Pi 2 Model B.*

Network Connectivity

Once you have your operating system installed, you will need network connectivity. If you are using the suggested Cat 5 cable, it should already be registered and on the network. If you decided to go with Wi-Fi, configure the wireless network in Debian. We'll wait right here until you finish…

Now that you have your network setup, it's time to install all of the software necessary for the tutorial. As an FYI, this is the only reason you need a network at all.

PHP

The following will be required for the examples:

- PHP5 CLI
- Composer

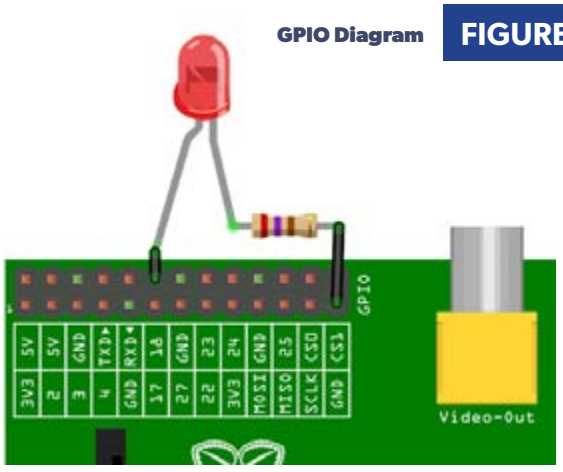
You can install these in a terminal session by typing:

```
sudo apt-get install -y php5-cli
wget -O composer-setup.php
php composer-setup.php
sudo mv composer.phar /usr/local/bin/composer
```

Connect the LED

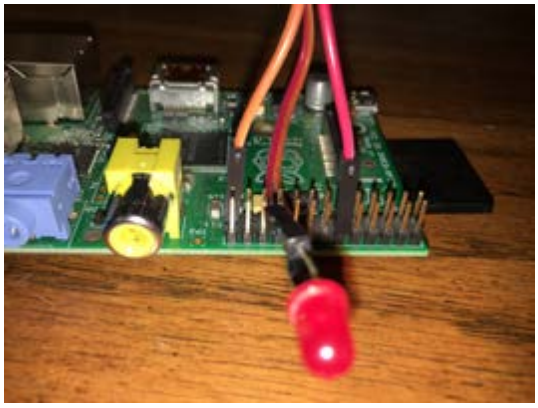
For these examples, we'll use GPIO pin 18. If you are using a Raspberry Pi other than a Zero, you will need to connect the female-to-female wire leads to ground and pin 18 as shown in Figure 1.

GPIO Diagram **FIGURE 1**



The LED will be connected to the other end of the female-to-female wire leads. It should look something similar to the image in Figure 2 when connected.

FIGURE 2



LED Connected

GPIO on Linux Primer

GPIO on Linux is exposed via the file system. The interface is known as GPIO Sysfs Interface for Userspace[7] and is part of the Linux kernel. Nothing we will be doing in the examples that follow are specific to a Raspberry Pi except for the pin numbers and placement.

Export Control a GPIO Pin

Writing the pin number to the file `/sys/class/gpio/export` will ask the kernel to export control of that pin to the user space. If you execute the following, you would request export of control for GPIO pin 18 to the user space:

```
echo 18 > /sys/class/gpio/export
```

Once exported, a directory will now exist under the `/sys/class/gpio` directory as `/sys/class/gpio/gpioN` where *N* is the pin number. For the example, the directory would be `/sys/class/gpio/gpio18`.

Unexport Control a GPIO Pin

Writing the pin number to the file `/sys/class/gpio/unexport` will ask the kernel to remove control of a GPIO pin from the user space. If you execute the following, you would request removal of control for GPIO pin 18 from the user space for pin 18:

```
echo 18 > /sys/class/gpio/unexport
```

Once unexported, the pin directory will no longer exist under the `/sys/class/gpio` directory.

Setting Direction of a GPIO Pin

Writing to the `direction` file in a GPIO pins directory created by an *export* will set the direction of the GPIO pin. Setting the direction to `in` will treat the GPIO pin as an interrupt whose state will be based on the applied voltage to the pin and read by the user space. Setting the direction to `out` will treat the GPIO pin as a switch whose state is set by the user space and will affect the voltage of the GPIO pin. The following example sets GPIO pin 18 in out mode, and it will be treated as a switch:

```
echo out > /sys/class/gpio/gpio18/direction
```

6 *January 2016 issue:* https://www.phparch.com/magazine/2016-2/january/

7 *GPIO Sysfs Interface for Userspace:* https://www.kernel.org/doc/Documentation/gpio/sysfs.txt

Setting and Getting the GPIO Pin's Value

Reading and writing data to and from the `value` file in a GPIO pins directory created by an *export* will read and write the current state of the GPIO pin respectively. The values are `0` *(low)* and `1` *(high)*. Writing any non-zero to a GPIO pin configured for output will be treated as `1` *(high)*. The following example will set the value of GPIO pin 18 to *high*. If you have properly connected the LED on your Raspberry Pi, the following example will turn on the LED:

```
echo 1 > /sys/class/gpio/gpio18/value
```

The following example turns off the LED:

```
echo 0 > /sys/class/gpio/gpio18/value
```

Reading the value can be done as easily from the same file. The following example will read the value from GPIO pin 18:

```
cat /sys/class/gpio/gpio18/value
```

The value returned will be `1` or `0` depending on the state of the GPIO pin.

Determining Interrupt–Setting the Edge

The `Edge` of a GPIO pin determines which change in value it will react to during asynchronous I/O polling. The options are as follows:

- none–Do not interrupt for any change (default).
- rising–Interrupt when the value changes from *low* to *high*.
- falling–Interrupt when the value changes from *high* to *low*.
- both–Interrupt on any change in value.

Inverting the Values

In some circumstances, it may be necessary to invert the value of a GPIO pin. This can be done to standardize the effect of high and low based on the reaction of connected components. Setting the inversion of the values is accomplished by writing `0` *(false)* or `1` *(true)* to the `active_low` file in a GPIO pins directory. The following example would invert the values of *low* and *high*:

```
echo 1 > /sys/class/gpio/gpio18/active_low
```

Examples

There will be three examples of controlling GPIO on the Raspberry Pi. Each will be a little more complex. When completed, you will have the basic knowledge required to start your IoT journey in PHP. The example code is also available on GitHub. Git should already be pre-installed on your Raspberry Pi. The following command will clone the examples into a directory named php-io:

```
git clone https://github.com/aenglander/php-io.git
```

Example 1: Bash

Using the examples in the GPIO on Linux Primer, we will setup your connected LED and turn it on and off.

```
echo 18 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio18/direction
echo 1 > /sys/class/gpio/gpio18/value
echo 0 > /sys/class/gpio/gpio18/value
```

- *If you have previously setup GPIO port 18, you will receive the error "echo: write error: Device or resource is busy." You can either perform an `unexport` for GPIO port 18 and start over, or simply ignore the error and move on.*
- *If the LED connected on GPIO port 18 did not turn on and off, verify your connections.*

Hopefully, this example has helped you see just how easy GPIO can be. If you are like me, you had a light bulb moment and realized controlling GPIO will be super simple in PHP. It becomes obvious in the next example.

Example 2: PHP File Functions

Using the bit of PHP from Listing 1.

LISTING 1

```
<?php
if (!file_exists("/sys/class/gpio/gpio18")) {
   file_put_contents("/sys/class/gpio/export", "18");
   file_put_contents(
      "/sys/class/gpio/gpio18/direction", "out"
   );
}
$current = trim(
    file_get_contents("/sys/class/gpio/gpio18/value")
);
file_put_contents(
    "/sys/class/gpio/gpio18/value",
    ("1" == $current) ? "0" : "1"
);
```

We will be checking to see if GPIO pin 18 has been initialized. If not, we will do so. We will then read the current value of the pin and write the opposite value. This effectively toggles the switch and turns the LED on and off.

If you would like to use the example from GitHub, it is located at `php-file/toggle.php`.

Example 3: Full IoT with Icicle.io

Icicle.io[8] is an asynchronous framework for PHP. It is still in its infancy but shows great promise as an asynchronous framework. ReactPHP was its predecessor, but it has been dead for two years now. The importance of using icicle.io is the ability to react to component input via GPIO interrupts, as well as external input via HTTP requests. Icicle.io also allows for accessing remote endpoints to push data. It is the closest to Node.js or Twisted we have in PHP.

If you would like to use the example from GitHub, it is located in the `icicle` directory.

Including Requirements

Use Composer to include the necessary requirements. You will need the following packages:

- icicleio/http
- icicleio/filesystem

Once the `composer.json` file is added, install them via Composer:

```
composer install
```

8 *Icicle.io:* https://icicle.io

LISTING 2

```
{
    "name": "aenglander/iot-example",
    "require": {
        "icicleio/http": "*",
        "icicleio/filesystem": "^0.1.0"
    },
    "autoload": {
        "psr-4": {
            "aenglander\\IoT\\Example\\": "src/"
        }
    }
}
```

An executable file is next (see Listing 3). It will bootstrap the application, create the listeners and start the event loop:

LISTING 3

```
#!/usr/bin/env php
<?php

require __DIR__ . '/vendor/autoload.php';

use aenglander\IoT\Example\Server;
use Icicle\Http\Server\Server as HttpServer;
use Icicle\Loop;

$server = new HttpServer(new Server());
$server->listen(8080, '0.0.0.0');

Loop\run();
```

The final piece is the request handler shown in Listing 4 [on the next page]. This example `composer.json` requires it to be in the `src` directory as `Server.php`:

The code is somewhat explanatory based on the comments. It may seem a bit confusing as it has the *yield* keyword everywhere. The *yield* keyword is used by Icicle.io to not require callbacks and allow for a more top-down development approach in asynchronous functional programming. What all of this code provides is a true Internet of Things connected device. Its status can be obtained and controlled remotely.

Start the server with the following command in the project root:

```
php server.php
```

You should see the message:

```
HTTP Server listening on 0.0.0.0:8080
```

Once the server is running, you can access the server in your browser. It will include the instructions to access the pin. Your LED's status is accessible at the path `/18`. You can turn the LED on by accessing the path `/18/high` and turn it off by accessing the path `/18/low`.

If you need the IP address of your Raspberry Pi, you can run `ifconfig` in the terminal and it will be the `inet addr` value for `eth0`. In the following example output, the IP address is 192.168.0.45:

```
eth0  Link encap:Ethernet  HWaddr b8:27:eb:d8:3a:67
      inet addr:192.168.0.45  Bcast:192.168.0.255
      Mask:255.255.255.0
      inet6 addr: fe80::abe4:33a6:6747:59f9/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:7737 errors:0 dropped:541 overruns:0 frame:0
      TX packets:899 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:654914 (639.5 KiB)  TX bytes:88090 (86.0 KiB)
```

If you used Wi-Fi, you are on your own. It won't be `eth0` and it won't be `lo`. At worst, try every inet addr you find.

The Next PHP Renaissance Is IoT

I hope you have finished this article and its examples with an understanding of the ease at which you can interact with electrical components via PHP in a Linux system. I implore you to take this bit of knowledge and expand it by building more and more devices. Once you feel confident in your understanding, share your knowledge with others. Blog about it. Give presentations at user groups and conferences. Spread the word that PHP developers can build their own IoT devices.

The PHP community has shown the resiliency to make up significant ground on other programming languages, even when the difference seemed overwhelming. Four years ago, there was a void in open source frameworks and libraries and no decent package manager for PHP. Today we are experiencing a renaissance in PHP thanks to Composer, Packagist, and nearly 100,000 packages which include a number of fantastic frameworks and countless open source libraries that are indispensable. You and I can build the libraries and tools necessary to bridge the gap between PHP and the incumbent, JavaScript. We can begin the next renaissance of PHP with IoT development today.

Adam Englander is a seasoned developer with over thirty years of experience building scalable, manageable, and reliable systems. For the past ten years, he has been doing that with PHP. Adam is vocal advocate for utilizing Behavioral Driven Development as the driver for building great applications. His dedication to the PHP community is well known as the founder of Vegas PHP. In the past few years, Adam has been building IoT apps and has been researching ways to make this accessible in PHP. @adam_englander

LISTING 4

```php
<?php
namespace aenglander\IoT\Example;

use Icicle\Http\Exception\MessageException;
use Icicle\Http\Message\BasicResponse;
use Icicle\Http\Message\Request;
use Icicle\File;
use Icicle\Http\Server\RequestHandler;
use Icicle\Loop;
use Icicle\Socket\Socket;
use Icicle\Stream\MemorySink;

class Server implements RequestHandler
{
    const ACCEPTABLE_TYPES = ['text/json', 'application/json'];

    public function onRequest(Request $request,
                              Socket $socket) {
        $path = $request->getUri()->getPath();

        try {
            if (preg_match('/^\/(\d+)$/', $path, $matches)) {
                // If this has a pin but no action, return
                // the status
                yield $this->gpioPinHandler($matches[1]);
            } elseif (preg_match('/^\/(\d+)\/(high|low)$/',
                                 $path, $matches)) {
                // If this has a pin and a status, set the
                // status and then return the current status
                yield $this->gpioPinHandler($matches[1],
                                            $matches[2]);
            } else {
                // Show instructions if not a pin URL
                yield $this->rootHandler();
            }
        } catch (MessageException $e) {
            $sink = new MemorySink();
            yield $sink->end($e->getMessage());

            $response = new BasicResponse(
                $e->getResponseCode(), [
                    'Content-Type' => 'text/plain',
                    'Content-Length' => $sink->getLength(),
                ],
                $sink);
            yield $response;
        }
    }

    public function onError($code, Socket $socket) {
        return new BasicResponse($code);
    }

    public function rootHandler() {
        // Place the response body in a memory stream
        $sink = new MemorySink();
        yield $sink->end("GPIO with Icicle.io example!
            \n\nAccess the pin number to get the status: /18
            \n\nAccess the pin with high/low to change the
state: /18/high");

        // build the response
        $response = new BasicResponse(200, [
            'Content-Type' => 'text/plain',
            'Content-Length' => $sink->getLength(),
        ], $sink);

        // Yield the response
        yield $response;
    }

    private function gpioPinHandler($pin, $action = null) {
        // If this has an action, we need to change the value
        // of the pin
        if ($action) {
            // If the pin does not yet exist
            if (! (yield File\isDir(
                            "/sys/class/gpio/gpio{$pin}"))) {
                // Export the PIN
                $export = (yield File\open(
                                "/sys/class/gpio/export", "w"));
                yield $export->write($pin);
                yield $export->close();
            }
            // Open the pin value file and write the value
            $value = (yield File\open(
                    "/sys/class/gpio/gpio{$pin}/value", 'w'));
            yield $value->write($action == 'low' ? 0 : 1);
            yield $value->close();
        }

        // Get the pin data
        $pin_data = ['pin' => $pin];
        // if the pin in exported
        if (yield File\isDir("/sys/class/gpio/gpio{$pin}")) {
            // It is initialized
            $pin_data['initialized'] = true;

            // Get the direction
            $direction = (yield File\open(
                "/sys/class/gpio/gpio{$pin}/direction", 'r'));
            $pin_data['direction'] = trim(
                yield $direction->read()
            );
            yield $direction->close();

            // Get the value
            $value = (yield File\open(
                    "/sys/class/gpio/gpio{$pin}/value", 'r'));
            $pin_data['value'] = trim(yield $value->read())
                                    == "1" ? "high" : "low";
            yield $value->close();
        } else {
            // It's not initialized
            $pin_data['initialized'] = false;
        }

        // Place the response body in a memory stream
        $sink = new MemorySink();
        yield $sink->end(json_encode($pin_data));

        // Build the response
        $response = new BasicResponse(200, [
            'Content-Type' => 'application/json',
            'Content-Length' => $sink->getLength(),
        ], $sink);

        // Yield the response
        yield $response;
    }
}
```

Be a Community Builder

Cal Evans

Past U.S. President Ronald Reagan said it best, "The nine most terrifying words in the English language are 'I'm from the government, I'm here to help.'" I feel that way about communities as well. The most vibrant tech communities I've been a part of have not so much had appointed leaders, as they have had leaders rise up for a time to fill a role or solve a problem. Then, these people would quietly fade back into the community. The next time something happened and the community needed a leader, they might rise back up or a new voice might rise up.

To a person, these people were "Community Builders," with an emphasis on the word builder. I've seen these men and women rise up, and raise others up. The best of them recognized what they had was a voice, not necessarily the right answer. They help build others up in the community and overcome the problem.

What I've never seen in a healthy community are "Community Demolition Teams" (CDTs). See how that just doesn't roll off the tongue? It's not natural. Don't get me wrong, I have seen these people rise up in communities, but not in healthy communities. You can tell a CDT easily. Instead of building people up, they begin tearing people and groups down.

I do not believe CDTs operate in bad faith. The ones I've known and talked with have always felt they were doing the right thing. However, when the right thing is labeling, blackballing, and tearing people down, I question their concept of "the right thing."

Thankfully, healthy communities see CDTs as a bigger problem than whatever problem they are trying to solve and move to silence them. Not by using their own tactics against them or tearing them down, but by trying to bring them back into the community. To help them understand their actions, no matter how well intentioned, are hurting the community, not helping.

The great thing about tech communities, especially open source tech communities, is they don't have to be one-size fits all. I was honored to be a member of the Board of Directors for PHPWomen[1] for many years. (Happy 10th Birthday PHPWomen!) That group was born out of a need. They realized there was a need for a more nurturing side of the community. They were welcoming to anyone who wanted to participate as long as they followed the rules. They did not tear down others who did not agree with them or insist the entire community adopt their way of doing things. They quietly went about setting a good example.

When a problem arises in your community, look to those who rise up to solve the problem by building, not by destroying. When it is your turn, be a community builder, we will all be better for it.

As always, GET OFF MY LAWN!

Past Events

August

NorthEast PHP
August 4–5, Charlottetown, Prince Edward Island, Canada
http://2016.northeastphp.org

PHPers Summit 2016
August 19–20, Poznań, Poland
http://www.summit.phpers.pl

PHPConf.Asia 2016
August 22–24, Singapore
http://2016.phpconf.asia

1 *PHPWomen: http://phpwomen.org*

Upcoming Events

September

SymfonyLive London 2016
September 15–16, London, U.K.
http://london2016.live.symfony.com

PNWPHP 2016
September 15–17, Seattle, WA
http://pnwphp.com/2016

DrupalCon Dublin
September 26–30, Dublin, Ireland
https://events.drupal.org/dublin2016

PHPCon Poland 2016
September 30–October 2, Rawa Mazowiecka, Poland
http://www.phpcon.pl

PHP North West 2016
September 30–October 2, Manchester, U.K.
http://conference.phpnw.org.uk/phpnw16

Madison PHP Conference 2016
September 30–October 2, Madison, WI
http://2016.madisonphpconference.com

October

LoopConf
October 5–7, Ft. Lauderdale, FL
https://loopconf.com

Bulgaria PHP 2016
October 7–9, Sofia, Bulgaria
http://bgphp.org

ZendCon 2016
October 18–21, Las Vegas, NV
http://www.zendcon.com

International PHP Conference 2016
October 23–27, Munich, Germany
https://phpconference.com/en/

DrupalSouth
October 27–28, Queensland, Australia
https://goldcoast2016.drupal.org.au

Forum PHP 2016
October 27–28, Beffroi de Montrouge, France
http://event.afup.org

ScotlandPHP 2016
October 29, Edinburgh, Scotland
http://conference.scotlandphp.co.uk

November

TrueNorthPHP
November 3–5, Toronto, Canada
http://truenorthphp.ca

php[world]
November 14–18, Washington D.C.
https://world.phparch.com

December

SymfonyCon Berlin 2016
December 1–3, Berlin, Germany
http://berlincon2016.symfony.com

ConFoo Vancouver 2016
December 5–7, Vancouver, Canada
https://confoo.ca/en/yvr2016

PHP Conference Brazil 2016
December 7–11, Osasco, Brazil
http://www.phpconference.com.br

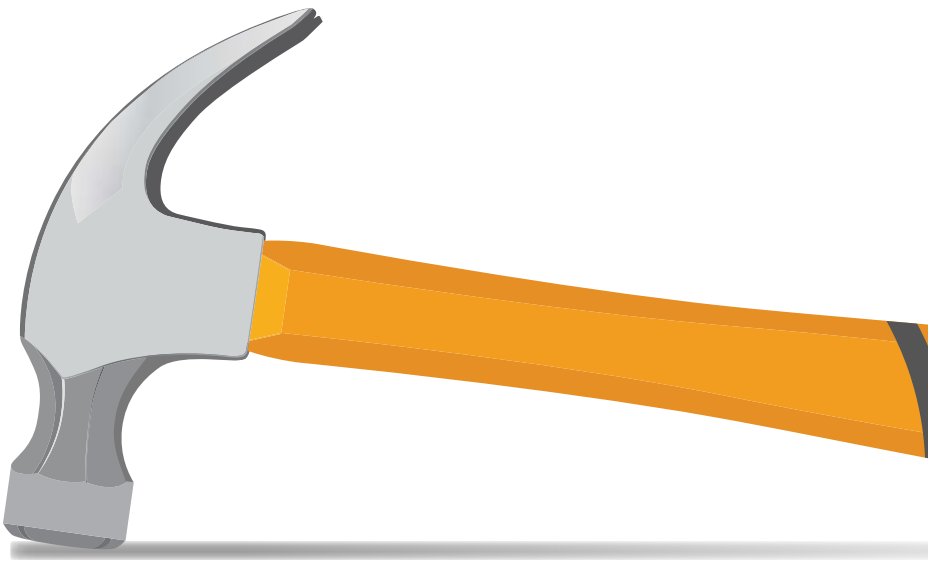
February 2017

SunshinePHP 2017
February 2–4, Miami, Florida
http://sunshinephp.com

March

ConFoo Montreal 2017
March 8–10, Montreal, Canada
https://confoo.ca/en/yul2017/

These days, when not working with PHP, Cal can be found working on a variety of projects like Nomad PHP. He speaks at conferences around the world on topics ranging from technical talks to motivational talks for developers @calevans.



Your Dependency Injection Needs a Disco

Matthew Setter

Dependency injection, what would any modern PHP application be without it? Likely a big ball of hard to manage mud. Right?

While some developers don't like them—even run screaming from them—I'm firmly in the camp that they are worth their weight in gold. When you understand them, and when you use a good one, you're off to the races with an easier to build, configure, and maintain codebase.

Don't believe me? Check out all of the modern implementations in PHP; including Zend ServiceManager[1], Laravel's IoC Container[2], Aura Di[3], and Pimple[4]. And that's just to name a few. When you go looking, there are far, far more.

What Is Dependency Injection (DI)?

If you're not familiar, here's a simplistic explanation. You have a container, which will contain the services (or dependencies) your application will need at some stage during its lifecycle.

You prepare the container by adding to it the services which you'll need. With that done, you then make the container available to your application. Now, your application can make use of the services contained within the container, unconcerned with where they came from, or how they're implemented.

Listing 1 is a small code example to help visualize it using Pimple; arguably the simplest of all of PHP's DI containers.

You can see I've created a small class which has one member variable, called `$name`. I've initialized `$name` with the class' name, and added a method to retrieve its value, `getName()`.

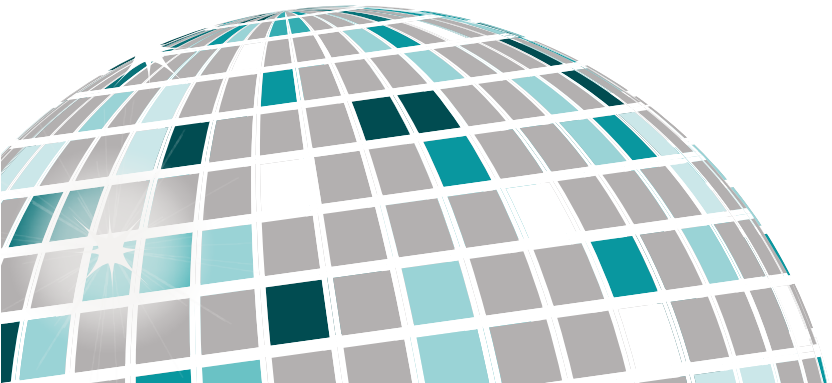
Next, I've initialized a new Pimple container and registered one service with it, using the name (or key) `user`. The service is an instantiation of our class. With that done, I retrieve the service, and call its `getName()` method, which prints out `"User"`.

1 *Zend ServiceManager:* http://phpa.me/zf-service-manager

2 *Laravel's IoC Container:* https://laravel.com/docs/4.2/ioc

3 *Aura Di:* http://auraphp.com/framework/2.x/en/di/

4 *Pimple:* http://pimple.sensiolabs.org



LISTING 1

```
01. <?php
02.
03. use Pimple\Container;
04.
05. class User
06. {
07.     private $name = __CLASS__;
08.
09.     public function getName() {
10.         return $this->name;
11.     }
12. }
13.
14. $container = new Container();
15.
16. $container['user'] = function ($c) {
17.     return new User();
18. };
19.
20. print $container['user']->getName();
```

If we needed to, we could change how the service is implemented, yet our code wouldn't need to change. Now, as I said this is a simplistic example. And in a real-world application, our needs would be a lot more sophisticated. We'd have services for handling connections to database servers, queueing servers, search servers, and a host of other services besides.

If you are familiar with the frameworks mentioned above, you'll know they provide a range of approaches to configuration:

- Laravel supports Closure callbacks and automatic resolution.
- Zend Expressive uses configuration files and, quite often, factories.
- Aura uses builder classes, auto-resolution, and setter methods.

But this month, I want to look at a dependency injection container which takes a different approach, instead using annotations. It's called Disco[5], by bitExpert AG[6].

5 *Disco:* https://github.com/bitExpert/disco

6 *bitExpert AG:* https://www.bitexpert.de

Introducing Disco, the Annotation-Based Container

Disco, to quote the repository documentation, is:

> *A container-interop compatible, annotation-based dependency injection container*

At the very thought of annotations you may, again, scream and run for the hills. But stop! If you feel pangs of fear welling up in your stomach, or sweat starting to bead across your brow, take a deep breath, put the kettle on and give me a few minutes. I'm sure you might just come to like them. I sure did!

That's right, for the longest time I wasn't a fan of annotations either; they were some form of magic, which has no place in code. After a little while of playing with Disco, I changed my mind. Before we can go any further, however, let's make Disco available, so we can use it, not just talk about it.

Installing Disco

Like all modern PHP packages, Disco can be installed using Composer. There are other installation options, but since Composer is so universally recognized and understood, it doesn't make sense to look elsewhere. In the terminal, from the root directory of your project, run the following command, to have Disco added as a dependency of your project:

```
composer require bitexpert/disco
```

Within a few minutes, it will be available inside the `vendor` directory, and Composer's autoloader will be updated and ready to make it accessible to your project. Assuming it's already available, let's continue and have a look at a "hello-world" example.

Configuring Disco

What we're going to do is to create a single service. It will be almost identical to the Pimple example we saw earlier. You can see it in the code in Listing 2. It's composed of two parts, configuration and usage.

LISTING 2

```
01. <?php
02. namespace PhpArch;
03.
04. use bitexpert\disco\annotations\bean;
05. use bitexpert\disco\annotations\configuration;
06.
07. /**
08.  * @configuration
09.  */
10. class AppConfiguration
11. {
12.     /**
13.     * @bean({"singleton"=true, "lazy"=true, "scope"="request"})
14.      * @return user
15.      */
16.     public function user() {
17.         return new user();
18.     }
19. }
```

Begin by importing the classes we'll need: the `bean` and `configuration` annotations. Yes, *bean*! Don't let the name put you off. Despite the nomenclature, we're not going all J2EE here—even if the name suggests otherwise.

This is the core configuration file for our DI container, hence the first annotation `@configuration`. What you name the class is irrelevant, as long as the annotation is present. Next, we move on to define the first, and only, service stored in the container, called `user`. This is an excellent feature of Disco. The name of the service is the name of the method—oh, and it's *case-insensitive* too.

Now, let's look at the `@bean` annotation for the `user()` function. `@bean` indicates it's a service within the container. Maybe Disco should have been called Café instead? That said, there are three options passed to the `@bean` annotation: *lazy*, *scope*, and *singleton*.

Lazy, indicates whether the service should be lazy-loaded or not. This is handy for expensive services, such as database connections, or connections to remote services, which may take quite some setup, before being able to be used.

Scope is quite an interesting one; it indicates whether the scope of the service will be limited to the current request, or whether it will be stored in session. Quite handy, depending on what you're trying to achieve.

If you need information in the service to be persisted across multiple requests, then session comes in handy here. Finally, singleton, as the name implies, controls whether there's only one instance of the service created.

Using Disco

Now let's look at how to use it (see Listing 3). It's one thing to set up a DI container. But if we don't use it, it becomes all rather academic. No?

LISTING 3

```
01. <?php
02.
03. use bitExpert\Disco\AnnotationBeanFactory;
04. use bitExpert\Disco\BeanFactoryRegistry;
05. use PhpArch\AppConfiguration;
06.
07. $container = new AnnotationBeanFactory(
08.     AppConfiguration::class, []
09. );
10. BeanFactoryRegistry::register($container);
11.
12. print $container->get('user')->getName();
```

Here, I've imported the required classes. After that, I've then instantiated a new `AnnotationBeanFactory`, which is the DI container which our application will use.

To it, we pass the name of the class, `AppConfiguration`, which holds the container's configuration. We can optionally pass parameters to the constructor, but in this case, as the configuration is quite trivial, I've not done that. Then, we register the container, by passing `$beanFactory` to `BeanFactoryRegistry`'s `register()` method. This finishes the setup.

Beyond Hello World: Using Disco as Slim's DI Container

With a quick run through finished, what about looking at something more sophisticated? What about taking on something a wee bit more ambitious? Why not?

To that end, with much thanks to Stephan Hochdörfer[7] from bitExpert AG[8] (the company behind Disco), I've set up a Disco configuration which can replace Slim3's default DI container. Through this example, you'll see even more of the functionality and power Disco provides.

If you look through `src/SlimDemo/Config.php`, in the repository[9] accompanying this month's article, you'll see there are some services defined, as required; specifically:

- settings
- environment
- request
- response
- router
- foundHandler
- errorHandler
- notFoundHandler
- notAllowedHandler
- callableResolver

All of them return objects of one type or another. But there's not enough in the examples to warrant much coverage. However, have a look at the definition of `settings` below.

```
/**
 * @Bean
 * @Parameters({
 *      @Parameter({"name" = "slim"}),
 * })
 * @return ArrayObject
 */
public function settings(array $slimConfig = []) {
    return new ArrayObject($slimConfig);
}
```

In this service definition, we see three further annotations: `@Parameters`, `@Parameter`, and `@return`. I'd say it is safe to suggest you can infer what these mean. But let's be pedantic. `@Parameters` contains the list of parameters which the service will receive. `@Parameter` handles a parameter definition. And, `@return` specifies the type of object which the service will return. I'll get into `@Parameter` in just a second.

For now, I want to give you some inside information on @return. Disco 0.5.0 will make substantial use of PHP7. Given that, this annotation is deprecated. Instead, Disco will use PHP7's return types to know what the service will return. In Disco < 0.5.0, services had to be objects. They couldn't be scalar types, such as arrays. In 0.5.0 this is becoming far more flexible.

Now, let's look at `@Parameter`; it accepts three options: *name*, *defaultValue*, and *required*. From what I've seen you only need to define name, as the other two can be implicitly provided by specifying a default value in the function signature, as was done above.

Let's look at where that information comes from, and also at a more sophisticated setup. Listing 4 shows our application configuration, a nested associative array.

You can see it has two core sections: `di` and `slim`. First, `di` contains Disco-specific configuration settings. Next, `slim` contains Slim-specific configuration settings. For `di`, we've specified the class which will hold the configuration for our container, where to cache the generated configuration, and whether we're in development mode or not.

For Slim, we've specified a host of core settings. As these are related to Slim, and we're talking about Disco, I'll leave them for you to dig into, should you feel adventurous.

```
$config = new BeanFactoryConfiguration(
    $APP_CONF['di']['cache'],
    null,
    null,
    !$APP_CONF['di']['devMode']
);
```

Let's return to the more advanced configuration, by introducing a new class: `BeanFactoryConfiguration`. This, as the name implies, provides configuration settings to Disco. Here, we're supplying two: a proxy target directory, and whether to use a proxy autoloader.

These two options are vital, as Disco is annotation-based. Reading and processing the annotations can be quite computationally expensive. This is fine during development, but in production you want all the speed you can muster. A custom autoloader speeds up the processing of the bean configuration, leading to a drastic performance improvement.

So, we've said that, unless we're in development mode, we want it enabled. Then there's the proxy target directory. This is the where the Proxy Manager will store the generated proxy classes if the proxy autoloader is used.

There are two other options, the `GeneratorStrategy` and `Cache`. `GeneratorStrategy`

LISTING 4

```
01. <?php
02. $APP_CONF = [];
03.
04. // Disco configuration
05. $APP_CONF['di'] = [];
06. $APP_CONF['di'] = [
07.     'config' => \SlimDemo\Config::class,
08.     'cache' => sys_get_temp_dir(),
09.     'devMode' => true,
10. ];
11.
12. // Slim configuration
13. $APP_CONF['slim'] = [
14.     'httpVersion' => '1.1',
15.     'responseChunkSize' => 4096,
16.     'outputBuffering' => 'append',
17.     'determineRouteBeforeAppMiddleware' => false,
18.     'displayErrorDetails' => false,
19.     'addContentLengthHeader' => true,
20.     'routerCacheFile' => false,
21. ];
```

7 *Stephan Hochdörfer:* https://twitter.com/@shochdoerfer

8 *bitExpert AG:* https://www.bitexpert.de

9 *the repository:* https://github.com/shochdoerfer/slim-disco-demo

determines how the proxy classes will be generated. `Cache` is a Doctrine cache object, which stores a cached copy of the annotations, which is yet a further way of improving Disco's performance.

```
$container = new AnnotationBeanFactory(
    $APP_CONF['di']['config'],
    $APP_CONF,
    $config
);
```

Now, let's look at a revised version of instantiating the container. This time, we're passing in the configuration class, as before, which we saw earlier in `$APP_CONF`'s definition. Then, instead of passing an empty array for the second parameter, we pass the entire configuration.

Finally, we pass in the `BeanFactoryConfiguration` object, which will guide how Disco works. Now we see how to supply and make use of parameters. You'll remember the configuration definition, contains the key 'slim' and that settings have a parameter called `slim`.

Well, when the container is instantiated, Disco passes that part of the configuration, when settings are retrieved from the container. So, while in this example, we're returning an ArrayObject which contains those settings, which are in turn passed on to Slim when it's instantiated, we could have done more with the information, should we have needed or wanted to.

Instantiating Slim With Disco

As you may have surmised, there's not much left to do, to use Disco as the DI container for Slim. Below, as before, we're registering the container. Then, we use it by passing it as the sole argument to the instantiation of a new `Slim\App` object.

```
BeanFactoryRegistry::register($container);

$app = new Slim\App($beanFactory);
```

Now, we can use Slim just as we otherwise would, such as in the example below, where we define a GET route. Slim can then retrieve all of the services it needs from Disco, and you'd never know you were using anything other than its default DI container.

```
$app->get(
        '/hello/{name}',
        function ($request, $response, $args) {
            $response->write("Hello, " . $args['name']);
            return $response;
        }
);
```

Conclusion

This has been a pretty wild ride, looking at a dependency injection container quite different in nature than most you're likely to encounter in the PHP community.

However, while it's a marked changed from most others, if you give it a chance, I think you'll see it can be as compelling as any other you may be used to. Admittedly, it's not as far along in its development as many of the others, but I believe it holds a lot of promise. Check it out and see how you go.

Matthew Setter is a software developer specializing in PHP and Zend Framework. He's also the editor of Master Zend Framework. Want to become a Zend Framework guru? Start your journey today at http://www.masterzendframework.com.

Behavior Driven Development With Behat

David Stockton

Writing tests for your code is an outstanding way to ensure your application is doing what it should and not what it shouldn't do, without needing to hire a lot of QA staff or relying on your users to beta test and find bugs. Since this column started, I've written about various testing topics and tools—PHPUnit, phpspec, and Humbug, as well as testing philosophies like TDD. It's been six months since the last testing article and I've got another testing tool and another technique to tell you about.

Behavior-Driven Development

We've talked about TDD or Test-Driven Development before. TDD is writing tests which describe what you want the code to do or not do. After that, you build the code which makes those tests pass. Behavior-Driven Development is another type of TDD, but traditionally it is focused at a different level, with a different perspective than we'd normally expect with TDD.

Typically, TDD focuses on unit testing. That is, we're describing what an individual unit of code should do. This "unit" typically means a single public method in an object. If the class has dependencies, we mock out those external classes or systems and fake the behavior so we can ensure our method does what we want when the external dependencies behave in certain ways. This is a great approach, and it's extremely useful in many cases, but there are some downsides.

To test classes with dependencies, we create test doubles to simulate how those dependencies should react; we are ensured the code being tested is working correctly. However, when the system is integrated and it's using the real dependencies, things can go poorly. Imagine integrating with another team's library or another company's API, and they change how it works. Perhaps they fixed a bug your code inadvertently relies on, or they've added or removed fields or functionality. Some aspect of the database has changed, a new field added or a critical field removed; all of your unit tests would still pass, but when the code is all running together—it fails.

This is where we look at a different level of testing besides relying solely on unit tests. There a four widely accepted levels of testing. Let's take a quick look at them before moving into Behat[1] and Behavior-Driven Development (BDD).

Four Levels of Testing

At the smallest level, we have *unit tests*. Unit tests should be written, ideally, as early as possible. They test each bit of code in isolation. This is where we use test doubles like mock objects and stubs to ensure our code does what it should. These tests should run very quickly, and chances are, there will be a lot of them. This is the level of testing we should be at when using a tool like phpspec[2], and can be doing with a tool like PHPUnit[3]. Errors are detected early and quickly at this level. Unit tests should be the most stable type of tests in the system.

The next level is *integration testing*. At this level, we are combining different parts of the system to ensure they are working properly with each other. This could be a test which exercises your method of making an API call or saving values to a database. At the unit test level, we'd provide a fake database or API. At the integration level, we can use the real database or real API. We may just be calling it without needing to go through our full routing, validation, security, and other levels. Integration tests will run more slowly than unit tests, sometimes significantly so. At this level, you can detect errors which happen when libraries or services are changed, but you are also just as likely to run into false failures when the services you're trying to use are unavailable or misbehaving. We can use PHPUnit to perform this sort of test in an automated way.

After integration testing, there's *system testing*. At this level, we're testing the system as a whole. It helps us know everything is configured, provisioned, and working together. These tests are even slower than integration tests. Often, the QA team can be performing this level of testing, but we can also use Behat.

Finally, there's acceptance testing. This is where we determine if the product is ok to ship or deploy, as well as if it's doing what it should, if words are spelled correctly, colors and alignment look good, and no major bugs are found. Tests at this level are even slower and more fragile. Often, this level is not automated, or it's only partially automated. Some acceptance tests can be automated via a tool like Behat.

As we move from unit tests, the fastest and most stable level, up through integration, system, and acceptance testing, the tests become slower and more fragile. This makes sense. At the higher levels, there are more systems, more code, and more requirements in play. Requirements are added or changed which in turn affects the software from the top down. On the other hand, if a change causes a break in a unit test, there's probably only on a single function which needs to be modified.

1 *Behat:* http://docs.behat.org/en/v3.0/

2 *phpspec:* http://phpspec.readthedocs.io/en/stable/

3 *PHPUnit:* https://phpunit.de

How is BDD Different?

With PHPUnit or phpspec (and Humbug), in order to be effective, you need to be able to write and understand PHP code. You probably need to be familiar with the code being tested. As a developer, this is expected and understood, and if you are a developer, you may wonder why I'm even bringing it up at all. However, a QA tester may not be familiar with the code and may not be familiar with writing code. This means, in general, developers need to be involved with building and maintaining any test suites written using PHPUnit or phpspec, if not entirely responsible for them.

My goal is to ensure testing is not only a role of my developers, but it's something my QA team can build on and contribute to. If they are already spending the time writing test cases, I want that effort to be reusable in the form of automated tests. The only viable way to grow software applications is through ensuring we have good tests. If we rely on manual tests, then as the software grows, we must grow the QA team which means more hires, more desks, more computers. It's not sustainable. I'd much rather continue to build on our automation suite and let the QA team find new and weird bugs. As they find bugs, anything which can be automated is added to the test suite which means they don't have to worry about that particular bug ever coming back. If you can automate a test or tests showing any new bugs have been fixed, you can effectively eliminate manual regression testing. This means if your team does not follow a continuous deployment/delivery cycle and uses a concept such as "code freeze" to ensure testers get a chance at testing the application as a whole without new changes coming in, this "code freeze" cycle can be greatly reduced or even eliminated.

Reporting Bugs and Automating Regression Testing

When testers find a bug or defect in the system, usually they'll write up a description of the problem. A good bug report will include reproduction steps, expected outcome, as well as the actual outcome. If there was a way to turn these descriptions and steps into automated tests, then the testers job of verifying the fix is working is simple—they just run that automated test or tests.

If we have this code, we can continue to run it on any change to the system and we are assured those defects haven't snuck back into the system. In other words, our regression cycle isn't happening at the end of the iteration or every once in a while; it's happening on every change, each time a bug is fixed, a feature is added or a pull request is merged.

Describing how new features work can also be thought of as a series of bug reports. Instead of the system doing something incorrect, it isn't doing anything at all. It is harder to describe new features in terms of bug reports, but it may be possible given a good understanding of the requirements and the system.

Introducing Behat

Behat is a BDD testing framework. It is related to, and is now listed as, the official PHP implementation of a system called Cucumber[4], a Ruby BDD framework. The tests in Behat and Cucumber are organized into features (essentially test suites) while each test is referred to as a scenario. There's a standard flow for writing these tests, which are written in a language called Gherkin[5]. Gherkin is a domain specific language with which you can describe the behavior of your software without needing to describe how it is implemented. It can act as both documentation as well as the source for controlling automated tests.

Each "feature" in Behat is described in a `*.feature` file. The first part of a feature file is the "Feature Introduction". Usually, it will look like a standard agile feature description:

```
Feature: Election Management API
  As an election administrator
  I need to be able to manage election details
  So I can properly configure the system for a
  new election
```

This block of text doesn't do anything as far as testing is concerned. It's strictly documentation. You can write whatever you want here.

The next part of the feature file are the scenarios. Each scenario describes some aspect of the functionality which makes up the feature (or bug). For example, a bug report could look like:

```
@BUG-1134
Scenario: A request for an alpha id shouldn't
  match and be a 404 Given I authenticate as an
  administrator When I request
  "GET /api/election/banana"
  Then I should get a "404" response
  And The "detail" field should be "Page not found."
```

There are a couple of bits to note. First of all, this scenario has a tag. That's the `@BUG-1134` part. This is optional, but Behat will allow you to filter which scenarios you run by tag. This means the QA team could verify the bug is fixed by running:

```
vendor/bin/behat --tags @BAL-1134
```

The next line is the scenario description. In a bug report, it can represent the expected behavior from the bug report. Every line after makes up the steps of the scenario. There is a combination of setup and validation. Each step line starts with one of the following words: `Given`, `When`, `Then`, `And`, or `But`. These words don't mean anything in the context of a Behat test, but they do serve to make the scenario flow more naturally. Typically, a scenario will use the word "Given" to indicate setup steps, "When" to indicate the action we are doing, and "Then" to transition to verification and validation. However, each of these words can be used to start any of the scenario phrases in any order. Since these can be used as documentation, I'd recommend choosing the words in a way which makes the most sense when read aloud. This may mean if you're verifying multiple things, the first phrase could start with `Then` while each subsequent verification starts with `And`, or you may decide it makes the most sense to start each verification phrase with `Then`.

When you run the test, by default, Behat will output the scenario description, followed by each of the steps, along with the file and line they are from. If a step fails, it will indicate the failure in the way the step failed. Any steps after a failing step will not be executed, the same way that if an assertion fails in PHPUnit, the rest of the test will stop as well.

4 *Cucumber:* https://cucumber.io

5 *Gherkin:* http://phpa.me/cucumber-gherkin

Scenario Templates

In PHPUnit, if we have a bunch of tests which look the same, we can use the `@dataProvider` doc block annotation and send in a bunch of different arguments to the test. In a similar fashion, Behat provides a way to build a scenario using the `Scenario Outline` or `Scenario Template` designator. Suppose the bug report above also found that if an ID of 0 or an ID larger than the database can handle could cause problems as well. In that case, we may want to ensure the ID fields only route if they are positive integers less than some large value which is smaller than the maximum recognized integer. The only thing changing is the ID, but we still expect the route not to match. We can rewrite the test like so:

```
Scenario Template: Invalid IDs should not route
  Given I authenticate as an administrator
  When I request "GET /api/election/<id>"
  Then I should get a "404" response
  And The "detail" field should be "Page not found."

  Examples:
  | id                  |
  | 0                   |
  | banana              |
  | 1234567890123456789 |
```

With the above example, the scenario will be executed three times, and the `<id>` variable will be substituted with the values from the table, one at a time. Each example is treated as a separate scenario, so if one fails, the others will still run independently.

Background and Common Setup

Suppose you've got a feature and every part of describing the scenarios requires the same setup, perhaps authentication or another step. Instead of repeating those steps on every single scenario, we can do something similar to PHPUnit's `setUp` function. Before each scenario in a feature file, Behat will run any steps in the "Background" section. You'll want to place these steps at the start of your feature file:

```
Background: Given I authenticate as an admin
```

This means that we could eliminate the authentication step from every Behat scenario in the feature file. Depending on what you're testing, this could eliminate a lot of lines of redundant steps.

How Does This All Work?

If you're trying to follow along and write these tests and run them, you're probably not having a lot of luck so far. Behat runs through the scenario file and matches each phrase to a method in a class (by default) called `FeatureContext`. Each scenario we run gets a new `FeatureContext` object which can be used to store state and other information within a scenario execution. Additionally, since each scenario gets a new `FeatureContext`, you don't have to worry about feature state from one scenario leaking into the next.

Inside the `FeatureContext` class, methods will be annotated with a doc block comment containing a regular expression or expressions which match the phrases you see in the feature file.

For example:

```
/**
 * @Then I should get a ".*" response
 */
public function iShouldGetAResponse($statusCode) {
   $responseStatus = $this->response->getStatusCode();
   if ($responseStatus != $statusCode) {
      throw new Exception(
        'Response code was not what was expected'
      );
   }
}
```

Anytime we have a validation or test that fails we can throw an exception and Behat treats it like a failure. This means out of the box, Behat doesn't have any assertions like PHPUnit. However, PHPUnit assertions work based on throwing exceptions when the assertions don't match. This means you can install PHPUnit and use the assertions within your Behat `FeatureContext`. The previous method could be rewritten as:

```
use PHPUnit_Framework_Assert as t;

// ...<snip>...

/**
* @Then I should get a ".*" response
*/
public function iShouldGetAResponse($statusCode) {
   t::assertEquals($statusCode,
                   $this->response->getStatusCode());
}
```

Now, realistically, this is already making a lot of assumptions, and if you build your `FeatureContext`, you'll probably find you want to take care of them in some way. The first assumption is that by the time we make this call, we've already made a request and have a response. If we don't, then chances are `$this->response` will be `null` and running Behat will fail when PHP tries to make a method call on a null object. So adding a check that the response is null and failing with `t::fail('A request must be made before checking the response');` would ensure the test fails gracefully and lets the person who wrote the scenario know they've messed up. In my version of this code, I ensure steps are called in order (that we have a response), as well as outputting the response if the HTTP status code doesn't match. This can help understand what went wrong if an assertion failed. In fact, for nearly all my phrases that fail, I output the response. I find it gives a lot more insight into why a test is failing which helps fix the code.

Because each line of the scenario executes on its own, it's important to build phrases which are stand-alone and to ensure anything they expect to be in place has run first. Just because the Behat scenarios can be built by putting the phrases in any order doesn't mean it will work that way or that it makes sense.

In the examples I've given so far, which are very similar to actual Behat tests I have in my system, some of the phrases are doing very little work, while others do quite a lot. For example, while the "I should get a (.*) response" is essentially a single assertion, the phrase for making a request is a lot more complicated. For me, it's building up a Guzzle request, including a body and headers which may have been configured in previous phrases. It's making that request and then setting the value of the

response into the `FeatureContext` object, as well as parsing it into a different property if it's JSON. It will also automatically fail a test if a request causes a 500 status code or fails to make the request we asked for.

In the Behat `FeatureContext` I have, I've built it for testing APIs. I can make requests against any endpoint with whatever HTTP verb I want, with whatever headers and body I need. I can make assertions about the returned HTTP status code. I can ensure JSON content has fields which are exact values or values matching a regex pattern, or that they contain certain parts. It allows me to specify a path through the JSON object to get to a specific field. I can even extract certain fields and store them in variables I can use when making subsequent requests or assertions.

This means I can quickly and easily write scenarios to test the APIs we have. It's also easy to create more expressive phrases. For instance, I've written many scenarios for various APIs to ensure routing doesn't match for invalid IDs. I noticed every one I'd written ended with the check to ensure there's a 404 as well as checking the detail field was always "Page not found." So, I created a new phrase of "The route should not match" which internally calls the method to assert the status and detail.

What Else

Of course, Behat can be used for more than just API testing. If you integrate it with a library like Mink you can use it to control a browser and ensure your website or application is behaving properly. This would allow for some additional level of system and acceptance testing beyond just ensuring your API is behaving properly. Depending on how you create your `FeatureContext`, you could use Behat to build integration tests as well, or even unit tests, if you were so inclined. I still posit PHPUnit and phpspec are better tools for unit testing, though.

How We Use It

In our project, any time there's an API defect found, we build a Behat test before fixing the defect. Our Behat tests are kept in a different source repository than our application code. Our CI server picks up the pull request and runs it. It will fail, of course, but that's ok. This gives the developer something to work against and it means once the code is fixed, the Behat PR can be merged and will ensure the defect is fixed once and for all. It means for the Behat jobs on our CI server, we have a ton of failures. When following a TDD or BDD workflow, this is expected and ok. By having the tests in a different repo, though, we don't have to interrupt our deployment flow. We can also upgrade dependencies like PHPUnit independently of the main application.

Additionally, to make things a bit simpler since a feature or bug fix based on a Behat test involves two different pull requests in two different repositories, we've also added a feature to the webhooks project I told you about last month. It allows us to leave a comment like "Test this when <url> is merged" where the URL represents the URL of our application pull request that should make the Behat test work.

IDE Integration

One last bit before I wrap up. If you're using PHPStorm, Behat integration is built in. This means you can command-click or control-click from a step in your feature file and it will jump directly to the code that implements the phrase. Additionally, when you need a new phrase, you can type it up in a feature file, click somewhere in it and press alt-enter or option-enter and select "Create step definition." It will prompt you where to put the code, and it will generate the method name as well as the `\@Given` doc block. Then, you can fill it out. If you build a scenario full of phrases that don't exist, it can create the stubs for you all in one shot, as well.

You can also execute all your features, a single feature or a single scenario with a keystroke in PHPStorm. If there are failing scenarios, and you feel you've fixed them, you can also tell it only to re-run the failing scenarios.

Conclusion

Behat provides a nice way to quickly automate system or acceptance tests. Since the tests are controlled using English phrases, it is relatively easy to add new features or ensure bugs have been fixed. It allows for users with little or no development experience to write scenarios that become running and working tests which can inform the developers what they need to do to fix a bug or build a feature. Since it is testing at a higher level, while the tests will be slower than unit tests, they can ensure the system is working as a whole, properly configured, and integrated. I'd highly recommend trying it out. It has become a critical part of my testing strategy, right next to PHPUnit and phpspec. See you next month.

David Stockton is a husband, father and Software Engineer and builds software in Colorado, leading a few teams of software developers. He's a conference speaker and an active proponent of TDD, APIs and elegant PHP. He's on twitter as @dstockto, YouTube at http://youtube.com/dstockto, and can be reached by email at levelingup@davidstockton.com.

Two-Factor All the Things

Chris Cornutt

When it comes to protecting your application, a simple username and password combination isn't enough anymore. In the early days of the web, that combination was defined as a "best practice," mostly as a carry-over from the early days of computing where the same was required to log into large mainframe systems or local terminals. Thanks to this early and widespread adoption, the standby of username and password has stuck with us and just about any large application out there defaults to this for its authentication mechanism.

I've already talked about the problem with passwords in a previous column January 2016 issue[1]) so I'm not going to get into all of that here. Instead, I'm going to talk about something you can do to enhance your application's security without having to replace the username/password combo with something more obtuse: the addition of two-factor authentication.

What Is "Two-Factor Auth?"

By now, most users of the web have probably heard of two-factor authentication and have seen it offered by many services they use daily. You may have even enabled it on some of your accounts. If you have, you can skip this part. If not, you might not be familiar with the overall idea of two-factor authentication and what kind of vulnerability it helps prevent.

First, what kind of problem does the use of two-factor auth solves? As I've mentioned before, people are really bad at two things related to passwords: picking good ones and not reusing them across services. This makes them relatively easy to compromise if an attacker either gains access to the data for another service the person uses or even just starts guessing passwords. All they'd need is to guess the right username and password and—bam!—they're logged in as that user without any other type of proof of identity.

A password in this setup is a "single factor" version of authentication. It's only one piece of data required as proof of identity and an easily discoverable one at that. This is where two-factor authentication comes in: it provides a second piece of data which can be used to verify the user on the other end of the line.

The different "factors" (types) of authentication usually fall into one of a few major categories:

- Something you know
- Something you are
- Something you have
- Some place you are

The username/password setup firmly falls into the "something you know" type. Remember, however, the weakness here is only using a single factor to authenticate the user. Now we're ready to talk implementation. I'll start with some of the most common methods for implementing two-factor in applications. I'll then follow it up with a practical example, using a simple library you can easily drop in to help you get it up and running.

Common Implementations

Two-factor authentication has, thankfully, seen a major rise in use over the past several years with major sites integrating it into their authentication path. There are a few main ways it has been implemented with pros and cons for each. These all fall into the "something you have" category, usually relating to some piece of hardware or software you have direct access to.

SMS One-Time Codes

One of the most common methods now that a large part of the world has some kind of cell device they can receive SMS messages on is the use of one-time SMS codes sent directly to the user. In this flow, the user logs in with the username and password (something you know) and is then presented with a page requesting them to enter the code that's just been sent to them. This code, usually a six-digit number, has been generated, stored, and linked to the user. Then, it is sent via some SMS service (like Twilio[2] or Nexmo[3]) to their previously set up device. They receive the message, type the number into the waiting field and submit. If the number matches, the user is fully logged in. If not, they may be completely logged back out or just told the number was invalid and to "please try again" (or resend a new code).

Recently, though, some security groups have come out against the use of a SMS based system like this, see *US standards lab says SMS is no good for authentication*[4]. With more and more attacks being made on mobile devices and networks, it's getting harder to ensure the message is making it to a device sitting in the hands of the actual user. This isn't to say it's not a viable option, it's just making the general public aware that it's not a "silver bullet" when it comes to two-factor handling.

1 *January 2016 issue:* https://www.phparch.com/magazine/2016-2/january/

2 *Twilio:* https://www.twilio.com

3 *Nexmo:* https://www.nexmo.com

4 *US standards lab says SMS is no good for authentication:* http://phpa.me/register-nist-sms

Push Notifications

If SMS codes aren't recommended as they once were, is there another alternative? Several services have implemented the idea of "push notifications" into their system. Anyone with a smart device is familiar with the notifications popping up from various applications when either an action is requested or some kind of situation comes up which needs their attention. Usually, this comes in the form of some pre-defined event. The idea of a "push notification" is similar but in this case the "event" is a login request coming from the service or site you're trying to use.

Here's the usual flow of a push notification process:

1. The user logs into the service with their credentials.
2. The service sees they have an active device and a matching identifier.
3. The service then sends a push message to the application waiting on the other end (on the user's device).
4. The user hits "Allow" and a message is sent back to the service to complete the login process.

You'll notice it has a similar feel to the SMS code version above but with one major difference: the communication is coming through an application the service controls and requires action on the user's part to finish the login. If the user doesn't hit the "Allow" button on the request, the login fails. While still not impervious to hacking attempts, this method makes it much more difficult for an attacker to intercept and reroute the request, as it's handled more internally and not sent out over public networks.

This also has its own downfalls, however. It requires a bit more infrastructure than the SMS codes, as well as needing either a separate application or special functionality built into your own application to handle the push messaging. There are third party services, like Duo Security[5], which can handle some of this for you but it also means accepting the risk of using an external service as a part of your authentication scheme (and all the issues that come with it).

Despite these problems, however, it is still a better, more secure option than the SMS codes.

Randomized, One-Time Codes

This last type of two-factor I want to mention–and get into more detail about–is the use of randomly generated, one-time codes. These codes come in two flavors:

- HTOP[6]: HMAC-based One-Time Passwords
- TTOP[7]: Time-based One-Time Passwords

These flavors are both based on official RFCs, defining the structure of their algorithm and how they should generate their codes. This results in a cross-implementation setup any application can easily integrate. It also has another major advantage: it can be used just about anywhere, even offline. Unlike the other two methods I've talked about, no external interaction is required to verify the code. No messages are sent to the end user, and no external networks are needed. Instead, the user is required to use their device and an application they've previously installed to retrieve a six-digit code they then enter into the application. As most services usually support the time-based version, these codes rotate out (60 seconds is a popular time frame) and a new one is then generated to replace it.

You may be wondering where the randomness comes from in this equation. After all, what good is a security system without some randomness? Well, this variety comes in the form of the "seed" used when the connection of device-to-user is created. This code should be randomly generated when the device is set up.

Setting up the System

This setup might be a bit more difficult than the others to envision. I'm going to give you an example of using it in a PHP application using the `enygma/gauth` package. This package makes it simple to drop in not only the support for the device setup but also the verification of the resulting code.

First, let's get it installed. I'm going to assume you have Composer already up and running here. If not, visit https://getcomposer.org and follow the steps there to get it working. With Composer up and running, you can load the package with:

```
composer require enygma/gauth
```

This should download the package and make it ready for your use. For the impatient, you can check out the `README.md` file for instructions on its use but I'm going to walk you through the full setup. This is also going to make the assumption you have a database containing a `users` table and a column named `device_hash` which will contain the hash linked to the user.

We need to link the device to the user. You'll need to have some section of your user's administration where they can optionally set up the device and link it to their record in the `users` table. Once the user selects to enable the device your code needs to do two things: generate the random code and display the QR code related to it back to the user. Listing 1 shows how we can do that using this library:

LISTING 1

```
01. <?php
02. $g = new \GAuth\Auth();
03. $pdo = new \PDO('mysql:dbname=appdb;host=127.0.0.1',
04.                 'awesomeuser', 'awesomepassword');
05.
06. // First generate the code and save it to the user's
07. // record (via a PDO connection)
08. $code = $g->generateCode();
09. $stmt = $pdo->prepare(
10.     'UPDATE users SET device_code = :code
11.         WHERE users.id = :userId'
12. );
13. $stmt->execute( ['code' => $code, 'userId' => $userId] );
```

This assumes you already have the user ID stored in the `$userId` variable and makes use of prepared statements to insert the record into the database. Next, we'll need to show the QR code to the user so they can scan it with their device. Fortunately, the library helps you generate this and output it too.

5 *Duo Security:* https://duo.com

6 *HTOP:* http://phpa.me/htop-hmac-passwords

7 *TTOP:* http://phpa.me/ttop-time-passwords

This assumes you're using the same randomly generated code in Listing 1 in `$code`:

```
<?php
// Re-initialize our Auth object with the user's code
$g = new \GAuth\Auth($code);

// Output the image
$qrCodeImageData = $g->generateQrImage(
    $userIdentifier, $appName, 200
);
echo '<img src="'
   . base64_encode($qrCodeImageData) . '"/><br/>';
```

You'll notice two new pieces of data in the above code: the *user identifier* and the *app name*. The `$userIdentifier` is something related to the user like a username or email address. It's usually best to use an already public piece of information here. The `$appName` is whatever you want to show up as your application name on the user's device. All that's left for them to do is set up a new account on their device and scan the code.

> *One thing to note here: because of how the setup works, the user themselves must be the one adding the account to their device by scanning the code. You cannot add it for them, as the device needs the information in the QR code to successfully make the link.*

That's the extent of the setup process. Pretty easy, right? Then comes the verification part. I promise, it's pretty painless too. All you'll need to do is insert a step in your login process which checks to see if the user has a device and, if so, forces them to enter the code before proceeding to any other page in the application. You can then verify the code with a few lines of PHP:

```
<?php
// Get the user's code using the PDO connection
$stmt = $pdo->prepare(
   'SELECT device_code FROM users
      WHERE users.id = :userId'
);
$stmt->execute( ['userId' => $userId] );

$g = new \GAuth\Auth($stmt->fetchColumn(0));
if ($g->validateCode($inputCode) == true) {
    echo 'Valid code! Move along...';
}
```

The above example pulls the code out of the database (you should probably check to be sure it exists, too) and uses it to create the `Auth` object. The code the user inputs—in `$inputCode`—is then verified. Remember, this all happens without any network connection to the outside world. If the code is valid, the `validateCode` result is `true`, and you can continue.

That's all it takes—you now have a fully functional, easy to use and, most importantly, completely internalized two-factor system in your application. This not only helps protect your users, but it also helps to protect your application from potential attackers. I hope this simple setup encourages you to implement this system in your own application. Go forth and two-factor all the things!

For the last 10+ years, Chris has been involved in the PHP community. These days he's the Senior Editor of PHPDeveloper.org and lead author for Websec.io, a site dedicated to teaching developers about security and the Securing PHP ebook series. He's also an organizer of the DallasPHP User Group and the Lone Star PHP Conference and works as an Application Security Engineer for Salesforce. @enygma

August Happenings

PHP Releases

PHP 7.0.10:
http://php.net/archive/2016.php#id2016-08-18-1

PHP 5.6.25:
http://php.net/archive/2016.php#id2016-08-18-2

PHP 7.1.0 Beta 3:
http://php.net/archive/2016.php#id2016-08-18-3

News

Lorna Mitchell: Joind.In Needs Help

Lorna Mitchell has a post to her site sharing a "call for help" related to the open source project she's a lead on: http://joind.in (a popular conference rating/feedback site in wide use across the PHP community). In her post she asks for help with the project and how you can help continue the success of the project/service. She ends the post with an update for those that wonder if this is "abandoning" the project, reinforcing that focuses have shifted more to "keeping the lights on" rather than abandoning the project overall.
http://phpdeveloper.org/news/24303

Laravel News: Laravel 5.3 is Now Released

As is mentioned in this new post on the Laravel News site, the latest version of the Laravel framework (v5.3) has officially been released. The post also lists some of the major updates that come with the v5.3 release including: New Home Page (for the project); New packages like Laravel Passport, Laravel Scout and Laravel Echo; Updated migration handling; Queued job improvements.
http://phpdeveloper.org/news/24327

Alex Bowers: Writing a Hello World PHP 7 Extension

In a recent post to his site Alex Bowers shows you the steps involved in creating a "Hello World" PHP 7 extension with some basic output functionality - basically just echoing out a message. He jumps right into the code and shows you how to: set up the directory and initial files for the extension; write the test case first (a simple PHP file checking if it's loaded and can be used); updating the config.m4 to allow for enabling the extension; the code for src/hello.h to define the function.
http://phpdeveloper.org/news/24340

Dries Vints: Two Tips to Speedup Your Laravel Tests

In this recent post to his site Dries Vints shares two quick tips you can use to help speed up the execution of the tests for your Laravel application. His two tips involve lowering the "cost" factor on the number of "rounds" the user password is hashed and the use of a pre-computed hash in your testing factories. These both help reduce the overhead needed, especially when working with tests that need to create the user every time.
http://phpdeveloper.org/news/24337

Matt Trask: Looking at Ramsey UUID

Matt Trask has put together a new post spotlighting a handy library that's widely used across the PHP ecosystem for generating UUIDs: ramsey/uuid. Matt then goes on to describe each of the different UUID types and provides some code examples as illustration.
http://phpdeveloper.org/news/24332

Zend Developer Zone: Testing Your Project with PHP 7.1

On the Zend Developer Zone author Cal Evans has written up a post showing you how to test your application with PHP 7.1, the upcoming minor release version for the PHP 7.x series. Cal shows how to make use of Docker containers to easily test your application in a more self-contained environment and make it simpler to swap out the PHP versions in your platform. He walks you through the steps you'll need to follow to get the environment set up, pull down required components, install and compile PHP and, finally, install Composer globally. Once set up, he shows how to log in, clone your project and execute its test suite.
http://phpdeveloper.org/news/24329

Matthew Weier O'Phinney: Using Composer to Autoload ZF Modules

Matthew Weier O'Phinney has a new post to his site showing you how to can use Composer to autoload Zend Framework modules right along with the rest of the ZF components. This allows ZF module authors to add details into the "extra" section of their Composer configuration, making it so the plugin understands how to load the module automatically.
http://phpdeveloper.org/news/24312

Symfony Finland: Consider Docker for Your Symfony Projects

On the Symfony Finland site there's a post that makes a recommendation for your Symfony framework based projects: give Docker a try to make setup and maintenance simpler. They then talk about how this "containerized" setup can be used to your advantage, making it simpler to get a Symfony application up and running with a few commands. The post then gets into an example setup of a Symfony Docker environment complete with Nginx, Varnish and PHP-FPM installed and linked.
http://phpdeveloper.org/news/24300

Christoph Rumpel: Build a PHP Chatbot in Ten Minutes

Christoph Rumpel has written up a tutorial showing you how to build a PHP chatbot in 10 minutes by hooking a PHP 7 based script in, via webhooks, to a Facebook Messenger application. He then walks you through the full process if setting up the Facebook Messenger application, a page to host it from and using the Chatbot boilerplate code to connect the application back to the Facebook platform.
http://phpdeveloper.org/news/24298

Jon LeMaitre: A Response To PHP—The Wrong Way

Jon LeMaitre has a new post to his Medium blog sharing his own response to the "PHP The Wrong Way" and some of the points it makes. He starts by pointing out the three main positive points "The Wrong Way" makes. However, as he points out, most of this advice is wrapped in "gross mischaracterizations of the PHP community, the nature of frameworks, standards, and PHP itself." Jon's post gets into a lot more detail on the various sections of "The Wrong Way", breaking them down into a series of quotes and matching responses.
http://phpdeveloper.org/news/24324

Robert Basic: Events in a Zend Expressive Application

Robert Basic has written up a new post sharing a method he came up with for event handling in a Zend Expressive application. He makes use of Zend's own `EventManager` component to integrate it with some of his work from a previous post. He then gets into the code, showing how to install the `EventManager` component and how to create/inject an event manager into a current object (a Command).
http://phpdeveloper.org/news/24270

Welcome to php[architect]'s new MarketPlace! MarketPlace ads are an affordable way to reach PHP programmers and influencers. Spread the word about a project you're working on, a position that's opening up at your company, or a product which helps developers get stuff done—let us help you get the word out! Get your ad in front of dedicated developers for as low as $40 USD.

To learn more and receive the full advertising prospectus, contact us at ads@phparch.com today!

Advertise with us!

Each month, php[**architect**] magazine brings news and information to an audience of thousands of PHP professionals worldwide. Advertising through php[**architect**] is a great opportunity to showcase your products and company to potential customers who have already shown their dedication to PHP and willingness to invest in their careers and knowledge.

Ads are accepted for php[**architect**] magazine in varied formats and are each priced based upon their relative sizing. Advertising is accepted from any company that has products or information that is related to the PHP community and web development. This includes promoting job openings at your company.

We also offer **Marketplace Ads** In addition to our traditional advertising space, we now offer low-cost Marketplace ads for projects, events, podcasts, and job postings. Perfect for small companies, startups, and side projects. Get your ad in front of dedicated developers for **as low as $34 USD**.

To learn more and receive the full advertising prospectus, contact us at ads@phparch.com today!

On the Value of a Degree...

Recently I got into a discussion with a good friend who was considering going back to school after having over a decade of experience in her industry. Many other people were chiming in, saying how going back to school late in their career was the best choice they ever made and how it opened up so many doors for them.

Now not everyone in that conversation was a computer programmer, but it brought this topic back to mind for me about our industry at least.

Initial Value and Types of Degrees

If you knew me 10–15 years ago, when I was working for startups and involved in many hiring decisions, I was vehement that it was important for everyone to have a degree in Computer Science. The grounding in theory that Computer Science gives you helps guide good coding practices. Many issues that I saw in software design could be linked back to simply not understanding some of the core logic, algorithms, and theories of Computer Science in the first place.

At the time, I dismissed people who had degrees in related fields, such as Computer Engineering or Information Systems. When I got involved in hiring at Digg.com, I was often frustrated when the management would engage many younger recent high-school graduates who had no formal training in Computer Science. However, in these cases, it was not just that they did not have *the degree,* it was also that they had little to no experience.

> *The only source of knowledge is experience.*
>
> *- Albert Einstein*

The Value of Experience

You see, one thing that I firmly believe in at this point is that experience is truly what matters. Say, someone has a recent B.S. in C.S. and no experience and someone else does not have a degree but does have ten years of experience. I would take the experienced person in a heartbeat. (This assumes, of course, that the good experience comes with good referrals.)

The value of experience just cannot be underrated. Every year that you spend writing production code, fixing bugs, and dealing with "real world" problems outweighs whether you can calculate the Big O notation of a function.

More importantly to my friend's situation, in the modern workforce every hiring manager worth his or her salt will understand this. If you have ten years of experience, unless you are specifically looking for a government job that (incorrectly) *requires* a certain degree, going back for your degree does not gain you anything.

Essentially, getting a degree is a way to get the "equivalent of experience" when you do not have any expertise in that field yet. Once you have the experience, you do not need the degree. The feeling that many people in that conversation were having amounted to a variation of Imposter Syndrome. They assumed they could not get the job they wanted without having the degree but never applied for the position to find out.

None of this, of course, applies if you are looking to switch fields because you are entering a new field without any experience.

The Current (and Future) Generations

My son has just entered sixth grade and has started thinking a bit about what he wants to do when he grows up. Currently, I am getting to play the role of proud father, especially when he keeps saying that he wants to "help do PHP" with me.

So what advice will I give him in six years when he is graduating from high school? Assuming he still wants to be a programmer, I would still suggest that he go to school and get a solid grounding in Computer Science. I feel that for someone starting fresh out of the gate, it is an excellent way to get knowledge under your belt and is probably worth about three years worth of experience.

Moreover, it is much easier to get the degree and then a job than to attempt to get a job immediately out of high school and find those three years of experience in the early days. However, hey, if you find yourself in a situation where the job can be found, and you take some training courses toward that—once again, experience is really where the deepest value exists.

Eli White is the Conference Chair for php[architect] and Vice President of One for All Events, LLC. He has a BS in CS, but it was so long ago he probably doesn't remember any of it at this point. @EliW

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[**architect**]™

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe