www.phparch.com

April 2017
VOLUME 16 - Issue 4

SPRING RENEWAL

PSR-7 HTTP Messages In the Wild

Integrating With APIs

Demystifying Multi-Factor Authentication

ALSO INSIDE

Education Station:
Rock Your Deployments With Rocketeer

Community Corner:
Look Out for That Bus!

Security Corner:
Taint Detection in PHP

Leveling Up:
Evaluating Value Objects

Artisanal:
Easy Vagrant Environments as a Service

finally{}:
PHP---By the Numbers

PHP[TEK] 2017

The Premier PHP Conference
12th Annual Edition

May 24-26 — ATLANTA

Keynote Speakers:

Gemma Anible
WonderProxy

Keith Adams
Slack

Alena Holligan
Treehouse

Larry Garfield
Platform.sh

Mitch Trale
PucaTrade

Samantha Quiñones
Etsy

Sponsored By:

php women · ShootProof · Zeamster · Stranger Than Fiction · platform.sh · stickermule

tek.phparch.com

php[architect]

CONTENTS

APRIL 2017
Volume 16 - Issue 4

SPRING RENEWAL

Features

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

Spring Renewal

Oscar Merida

Is the weather a bit wonky where you are? I'm just outside of Washington D.C. and the weather for the past few months has been all over the place. We've gone from snow one day to warm, almost summer-like heat in the span of a few days. We're firmly past the spring equinox here in the northern hemisphere, and before thoughts turn to summer vacation, I want to urge you to re-focus a bit on your skills. Is there a book you want to read? Is there some workflow at your job making your life difficult which you could automate? Have you been meaning to play around with a new framework? Set aside some time to dedicate to it. If it helps, set presenting to your local user group or writing about it on a blog as a goal. Then—let me know what it is at @omerida

This month's issue can help you find one if you can't pick something new to learn. Hannes Van De Vreken writes about *PSR-7 HTTP Messages in the Wild*. Now that PSR-7 has been ratified for some time, there are a number of middleware implementations to consider which make sharing code for working with HTTP requests and responses easier across frameworks and applications. If you need a guide to start working with APIs, Caitlin Bales has an excellent introduction to *Integrating With APIs*. She'll walk you through understanding the low-level PHP functions available for talking to another HTTP server, and show you how to build a client to talk to an API. Want to add another layer of protection to secure your site? In *Demystifying Multi-Factor Authentication*, Brian Retterer shares how adding multiple factors beyond password-challenges helps verify a user's identity. He'll implement sending a token via email to a Laravel application's users to explain the theory.

Matthew Setter has a PHP-based deployment tool for those of you who need to automate getting your code to production without a hassle. Check out *Education Station: Rock Your Deployments With Rocketeer* if you're still using FTP or not happy with your current deployment process. In *Leveling Up*, David Stockton writes on *Evaluating Value Object*. They're a kind of object which helps reduce confusion—and errors—about the kind of data your application consumes. This month's *Community Corner* by Cal Evans warns you to *Look Out for That Bus!* He explains why you—or, more importantly, your boss—should not ignore training. Regular readers know by now to never trust user input. In *Security Corner: Taint Detection in PHP*, Chris Cornutt writes about an extension which triggers an error if you use user-supplied data without validating or filtering it first. He'll show you how to install it and use it in your code. A warm welcome to Joe Ferguson this month as he debuts a new column *Artisanal*. He'll be covering Laravel each month, starting with *Easy Vagrant Environments as a Service*. See how to use Homestead to setup a development environment painlessly. In the *Finally* installment for April, Eli White reviews the results of a Stack Overflow survey of Developers. He looks at what they may say about the state of PHP developers.

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/April2017_code

PSR-7 HTTP Messages In the Wild

Hannes Van De Vreken

Not long ago, when you were building an application to handle HTTP requests, the best practice was to use an object that represents the incoming request and the response that needs to be outputted. One of the best off-the-shelf solutions for this was (and still is) Symfony's HttpFoundation Composer package. It comes with a `Request` object and a `Response` object, which helps you with modeling. It's so good it is used in other frameworks like Slim (version 2), Silex, and Laravel and applications like Drupal 8. Let's look at how we can share request and response related code readily across projects.

The HttpFoundation package has a couple of downsides, though. Granted, some of these might sound puristic. First of all: separation of concerns. The `Response` object has a send method, which deals with low-level PHP functions and output buffers to print the HTTP response on the output stream. An object models the values of an HTTP response (like its status code, headers, and body). It shouldn't have knowledge about the representation on the TCP layer, or the SAPI functions that PHP provides to help with that.

The second downside: coupling to the framework. It does an excellent job; the Symfony HttpFoundation package has very little dependencies and thus is very stable. Nonetheless, it comes with some framework specific code like the `RequestStack` and an `ExpressionRequestMatcher`. If you're only interested in the value objects, then these classes are unnecessary. If Symfony needs these classes to be changed, a new version will be released with no changes to the value objects. Furthermore, the `Request` and `Response` objects are not interfaced. This prevents you from doing flexible things like decoration, injecting mocks, or making your own implementation.

Again, these are just some minor downsides, and still, the Symfony package is an extremely stable and reliable implementation. Using this doesn't mean you're doing it wrong. It just means you're a little bit tied into Symfony as a "vendor." The same is true for zend-http. If your application is based on that, you will only primarily use libraries compatible with those `Request` and `Response` objects, rather than Symfony's. If you find a great library that solves your problem, but it works with other request or response objects than your preferred ones, chances are you're not going to use that library.

Here's where the PHP-FIG comes into the story.

PHP-FIG

The PHP Framework Interoperability Group[1] consists of a bunch of members who represent popular tools (packages, CLI applications, frameworks, and CRMs), including Symfony, Zend, Drupal, Doctrine, and Composer. This group works together to find common ground and minimal interfaces for tools to be used in different applications, without tying into any vendor.

The group decided it was time for a set of interfaces to model HTTP requests and responses. The project took off and was named *PSR-7*. After several rounds of discussions, they finalized the "standard recommendation" in May 2015.

PSR-7

The result is a well-balanced set of interfaces. The interfaces can be found here: php-fig/http-message[2]. The `Request` and `Response` value object interfaces represent requests and responses as described in RFC 7230[3] and RFC 7231[4]. Furthermore, these value objects must be immutable, which means any instance will always have the same internal state for the rest of its lifetime. A new instance with a modified state can be created by calling specific `with` methods, which will create new objects with a modified state, and return those. Since all values in the value objects make up a full request or a response, changing one of those values makes the object represent a different request or response. Thus, immutability is a sane choice.

Value objects should be immutable. Unlike entities with an identifier where you only want one object for each identifier to be instantiated.

```
$newRequest = $oldRequest->with-
Body($body);
var_dump($newRequest === $oldRequest);
> false
```

Both `RequestInterface` and `ResponseInterface` have methods like `withBody`, `hasHeader`, `getHeaderLine`, `withHeaderLine`, and `withoutHeader`. This makes sense since both a request and a response have headers and a (maybe empty) body. The ResponseInterface has three more specific methods: `getStatusCode`, `withStatusCode`, and `getReasonPhrase` since this is something only HTTP responses have. Of course, every HTTP response starts its message over TCP with a line which looks like this:

1 *PHP Framework Interoperability Group:* http://www.php-fig.org

2 *php-fig/http-message:* http://phpa.me/github-http-message

3 *RFC 7230:* http://phpa.me/rfc7230

4 *RFC 7231:* http://phpa.me/rfc7231

```
HTTP/1.1 200 OK
```

The reason phrase ("OK" in the example above) is a simple translation for the status code, mapped out in the spec. For example, 418's reason phrase is, "I am a teapot," which is just an easter egg.

The `RequestInterface` on the other hand has some more specific methods like `getMethod` and `withUri`. These methods are respectively for getting which HTTP request method (GET, POST, OPTIONS, etc.) is used, and to create a new request object with an updated target request URI (remember, these value objects are immutable).

These two interfaces (`RequestInterface` and `ResponseInterface`) follow the RFC specifications to the letter. This makes them extremely useful to not only represent incoming HTTP requests handled by our PHP scripts, but also to represent outgoing HTTP requests sent from our code to any HTTP server (using cURL or any other HTTP client). More on this later.

But, what about our superglobals[5] in PHP (`$_COOKIES`, `$_FILES`, `$_SERVER`, …)? PSR-7 has those covered too! The authors of PSR-7 decided, keeping interface segregation in mind, to help you with that by adding an interface which extends the `RequestInterface`; it's called the `ServerRequestInterface`. Additional methods like `getCookieParams`, `getServerParams`, etc. allow you to access the values given to you by the instantiated PHP process. The `ServerRequestInterface` also allows you to access the files which are uploaded as part of a multipart upload request. You can use `$request->getUploadedFiles()` to get an array of `Psr\Http\Message\UploadedFileInterface` objects. This is a model for PHP's `$_FILES` superglobal and is comparable to Symfony's `Symfony\Component\HttpFoundation\File\UploadedFile`.

The `RequestInterface`, `ResponseInterface`, and `UploadedFileInterface` expose their body through a `StreamInterface`. This helps with memory consumption when dealing with large bodies, or files. For example, if the response is a generated PDF file in a temp folder, you can open a read resource and let the response streamed body refer to that. There is no need to load the entire PDF into memory. Among all PSR-7 interfaces, the `StreamInterface` is the only one not requiring its implementations to be immutable. Of course, when a `Stream` object holds a reference to a PHP resource, it doesn't make sense to make it immutable, because then you would need to have to open new resource handles for each clone.

One last interface in PSR-7 we haven't covered yet is the `UriInterface`. It is only used in the `RequestInterface` through the `getUri` and `withUri` methods to model the requested string URI. In the plaintext HTTP request example below, the `UriInterface` would model the URI `"http://www.example.com/index.html"` and has convenient methods like `getScheme`, `getHost`, etc. to get individual parts or to get `Uri` objects with modified parts. Each `UriInterface` object also can be stringified through the magic `__toString` method.

```
GET /index.html HTTP/1.1
Host: www.example.com
```

We talked about all PSR-7 interfaces which describe the objects. You can find these interfaces in the Composer package `psr/http-message`[6]. There are a couple of implementations available which provide concrete, reference implementations for those interfaces. They all define their packages using `"provides": "psr/http-message-implementation"` in their `composer.json` file. `psr/http-message-implementation` is a virtual package on Packagist. You can get a list of all implementations by browsing to that virtual package. The three most downloaded implementations at the time of writing are `guzzlehttp/psr7`, `zendframework/zend-diactoros`, and `slim/slim` which has an implementation inside. As you can see, both HTTP clients (Guzzle is one of PHP's most used HTTP clients) and HTTP applications (Zend, Slim, etc.) have practical use cases for implementing and using PSR-7.

That's all nice, but, most existing applications are pretty much setup with Zend's HTTP objects or Symfony's HTTP objects, and that just works. There's no reason to rebuild the core of those applications just for the sake of having PSR-7 objects, right? Correct; the cost of refactoring your entire application is too high. But what if we need a PSR request object for an OAuth server package that reads the Authorization header, or want to respond with a `ResponseInterface`'d object generated by a package like league/glide which serves images? We don't want different libraries depending on global state contained in `$_SERVER`, `$_COOKIE`, and other superglobals. Inevitably, each package would begin creating their own request/response dependency to abstract working with them.

Well, that's easy enough to do. Since `Response` and `Request` objects are just value objects, it's relatively easy to convert them from one type to another—from Zend to PSR-7, from PSR-7 to Symfony and vice versa. There are packages to do that. There's `zendframework/zend-psr7bridge` for Zend, and `symfony/psr-http-message-bridge` for Symfony. For example, if you need a PSR-7 ServerRequestInterface'd object in a Symfony based application, you can use this code snippet:

```
$factory = new Symfony\Bridge\PsrHttpMessage\Factory();

$psrServerRequest = $factory->createRequest(
    $symfonyRequest
);
```

There are comparable factory classes with methods available for each possible conversion. This gives you an easy path to start using PSR-7 objects today.

As mentioned before, you can convert Symfony HTTP objects from PSR-7 objects as well. This helps with building new applications which might need to utilize packages that still depend on Symfony objects, for example. But how do we

5 *superglobals:* http://php.net/language.variables.superglobals

6 `psr/http-message`: http://phpa.me/http-message-implementation

Celebrating 25 Years of Linux!

ORDER NOW!

Get 7 years of *Ubuntu User*

FREE

with issue #30

***Ubuntu User* is the only magazine for the Ubuntu Linux Community!**

BEST VALUE: Become a subscriber and save 35% off the cover price!
The archive DVD will be included with the Fall Issue, so you must act now!

Order Now! Shop.linuxnewmedia.com

build an application with PSR-7 objects as its base?

Some frameworks already use PSR-7 objects. For example, Laravel can handle `ResponseInterface` objects when you return them in controller methods. SlimPHP 3 even uses PSR-7 for its main request/response life cycle. If we look at how SlimPHP uses the objects, you will see it sets up a middleware chain, like Symfony's `HttpKernel`, and calls the chain with a `Request` object and captures the resulting response object to print the response. You can add middlewares to the application in general, or on a per route basis.

Middlewares

Most middlewares have a signature that is *sort of* agreed upon and used by tools like middleware builders (`relay/relay`, `mindplay/middleman`, `zendframework/zend-stratigility`) and middleware packages (like `oscarotero/psr7-middlewares`). A middleware would look like Listing 1.

Listing 1.

```
<?php
function (ServerRequestInterface $request,
          ResponseInterface $response, callable $next) {
    // Adjust or check the request
    // ...

    // Call the next middleware
    $response = $next($request, $response);

    // Adjust or check the response
    // ...

    return $response;
}
```

These middlewares are sometimes called double-pass middlewares because they pass both a request object and a response object. With this signature, you can either do something with the incoming request, before calling the next one, or do something with the response object before returning it to the previous middleware, or both. One can also catch exceptions thrown in one of the subsequent middlewares, and translate the exception to an error response.

You may have noticed the second argument of that middleware callable; it's a response object. You might argue it's a bit weird as you can then edit the response, before passing it on to the next middleware. In theory, one should never need to do that. The reason for that argument to be there is middlewares don't have to decide which implementation of `ResponseInterface` to use. An empty response object allows you to create any response you like without having to choose between Zend's Diactoros implementation, Guzzle's implementation, Slim's implementation, etc. This makes sense in one way, but there are tons of different ways to inject any implementation if needed.

For example, on creation of the callable:

```
function (ServerRequestInterface $request,
          callable $next) use ($emptyResponse) {
    /* @var ResponseInterface $emptyResponse */
}
```

Or via constructor injection when the middleware is a full blown class instead of an anonymous function as in Listing 2.

Listing 2.

```
<?php

class Middleware
{
    private $emptyResponse;

    public function __construct(ResponseInterface $emptyResponse) {
        $this->emptyResponse = $emptyResponse;
    }

    public function __invoke(ServerRequestInterface $request,
                             callable $next) {
        try {
            return $next($request);
        } catch (Exception $exception) {
            // Build a new error response
            $response =
                $this->emptyResponse->withStatusCode(400);
            ...

            return $response;
        }
    }
}
```

What if you want to use middleware but don't want to use a framework? You should use a middleware stack builder. I mentioned some before, and they're all recommended. In this article, I will focus on `mindplay/middleman` for a single example. It has a nice way of resolving a middleware, only when the request reaches that layer. But most middleware stack builders have similar feature sets.

Let's talk about what happens in Listing 3. In step one, we decide on a list of middlewares. These can be middleware objects which are `callable`, anonymous functions, or just strings. If they are strings, the dispatcher will intelligently wait until your request/response chain reaches this middleware. When it does, the dispatcher will use a resolver to instantiate a proper middleware object or callable. For this to work, you need to instantiate the dispatcher, in step two, with the previously mentioned list of middleware and a middleware "resolver." My favorite resolver, provided by the package, is the container resolver called `InteropResolver`. This will call an inversion-of-control container using `$container->get($middlewareString)`. But if, for example,

you like factories more you can write a custom resolver to create middlewares from a factory. Last but not least, in step three the dispatcher's dispatch method is called with a `ServerRequestInterface`'d object and an empty response object. The resulting response is what the application logic has rendered in a value object. Note that we can use this as a middleware itself. `[$dispatcher, 'dispatcher']` is a callable you can reuse in a different middleware chain.

A different kind of middleware is being defined at the moment by the PHP-FIG. That standard recommendation will be called PSR-15[7], once completed.

Their approach is lambda style middleware:

```
$psrResponse = $delegate->process($psrServerRequest);
```

The middleware implementations will have a `process` method, which takes two arguments as shown in Listing 4.

This is a different way of defining a middleware, but very similar. The difference is in the fact every middleware needs to implement an interface. And instead of a `callable $next`, whatever needs to be called next, needs to be an object too. So every middleware stack builder needs to create `DelegateInterface` objects, which in their turn know the next middleware and the next delegate in order to call that middleware. A middleware stack builder for PSR-15 might as well implement the `DelegateInterface` itself, so it can be used in different middleware stacks. As such, this can be used to compose complex application trees with different branches of middleware stacks.

Whichever middleware stack we use, we need to call it (in essence the first middleware) with a `ServerRequestInterface` object that represents the request which invoked our PHP script. There are a couple of options to instantiate that. There are several ways, for example Slim's `Slim\Http\Request::createFromEnvironment($environment)` or Guzzle's `GuzzleHttp\Psr7\ServerRequest::fromGlobals()`. We can also use my favorite from the `zendframework/zend-diactoros` package which comes with a `Zend\Diactoros\ServerRequestFactory` class. This factory allows us to create a `ServerRequest` object like this:

```
$factory = new Zend\Diactoros\ServerRequestFactory();

$psrServerRequest = $factory->fromGlobals();
```

At the end of your application stack, you're left with a finalized response object. Now what? Again, in the `zendframework/zend-diactoros` package, there's a tool which allows us to do something with that. There's an interface and implementation called `Emitter` and `SapiEmitter`. To end the PHP script with the output of your finalized response object, you can do this:

```
$emitter = new Zend\Diactoros\Response\SapiEmitter();

$emitter->emit($psrResponse);
```

7 *PSR-15: http://phpa.me/psr17-middleware*

Listing 3.

```
$middlewares = [
    App\Http\Middleware\BasicAuthentication::class,
    // ... other middlewares for caching, sessions,
    // cookies, encryption, ...
    $routing,
];

$dispatcher = new mindplay\middleman\Dispatcher(
    $middlewares,
    new InteropResolver($container)
);

// Let application do the magic.
$psrResponse = $dispatcher->dispatch(
    $psrServerRequest, $emptyPsrResponse
);
```

Listing 4.

```
<?php
use Psr\Http\ServerMiddleware\MiddlewareInterface;

class MyMiddleware implements MiddlewareInterface
{
    public function process(ServerRequestInterface $request,
                            DelegateInterface $delegate) {
        // Do something with the $request
        // ...

        // Call the next middleware through a delegate object.
        $response = $delegate->process($request);

        // Do something with the $response
        //...

        // Return the resulting $response
        return $response;
    }
}
```

Clients

As noted earlier, PSR-7 request and response objects can be used for HTTP applications and HTTP clients.

Let's look at one of the most popular HTTP clients available on Packagist, Guzzle. Guzzle's HTTP client (version 6 and up) uses PSR-7 objects as value objects for outgoing HTTP requests and returned HTTP responses. The dedicated package `guzzlehttp/psr7` provides `psr/http-message-implementation` and thus has objects which implement the PSR-7 interfaces.

```
$client = new GuzzleHttp\Client();

$psrRequest = new GuzzleHttp\Psr7\Request(
    'GET', 'https://google.com'
);

$psrResponse = $client->send($psrRequest);
```

Listing 5.

```
$client = new Http\Adapter\React\Client(
    $responseFactory, $loop, $reactClient
);

$promise = $client->sendAsyncRequest($psrRequest);
$promise->then(ResponseInterface $response) {
    // ...
});
$promise->wait();
```

In the example above, the request object can be swapped out with any `Psr\Http\Message\RequestInterface` object (or a `Psr\Http\Message\ServerRequestInterface` object for that matter). If, for example, you have an older version of Guzzle, or any other HTTP client setup in your application (through an IoC container for example), then you might look into a HTTP client abstraction project like `php-http/httplug`. As long as you type hint `Http\Client\HttpClient`, which is an interface, you can use or write any adapter. For example, if you use Guzzle version 5, you can wrap it with `Http\Adapter\Guzzle5\Client`.

```
$guzzle = new GuzzleHttp\Client();

$client = new Http\Adapter\Guzzle5\Client($guzzle);

$psrResponse = $client->sendRequest($psrRequest);
```

By doing this, you don't have to refactor anything and keep using the same version of your current HTTP client, which might be setup with logging, caching, and authentication plugins for example.

Guzzle's latest client can also handle asynchronous HTTP requests:

```
$client = new GuzzleHttp\Client();

$promise = $client->sendAsync($psrRequest);
$promise->then(ResponseInterface $response) {
    // ...
});
$promise->wait();
```

The same is possible with any implementation of `Http\Client\HttpAsyncClient`, also part of the `php-http/httplug` project. For example, as in Listing 5 for a ReactPHP client:

Note, this does not only work for ReactPHP HTTP client, which is built to be asynchronous from the ground up, but also for Guzzle version 5, 6, and most other HTTP client adapters.

Listing 6.

```
<?php
$promises = [];

// Keep a list of all promises
$promises[] =
$promise = $client->sendAsyncRequest($psrRequest);
$promise->then(function (ResponseInterface $response) {
    // ...
});

GuzzleHttp\Promises\all($promises)->wait();
```

Then, we can send multiple HTTP requests off to fetch data in parallel and wait for them all to come back (see Listing 6). This waits for all promises to be fulfilled before continuing execution.

Wrapping Up

PSR-7 is a set of interfaces which defines immutable value objects for the HTTP spec's request and responses. Several implementations exist. The `ServerRequestInterface` extends the `RequestInterface` to add methods to access and modify data which gives information about the incoming request that invoked the current PHP process.

You can build an entire application by creating a request object. You can then pass it to a chain of middlewares, which are composed to define your application. This chain will respond with a response object, which you can then let an emitter emit to your script's output buffers.

Other than using PSR-7 objects for an HTTP application, you can also pass it to an HTTP client to send it off to any HTTP server. Several HTTP clients exist which can handle PSR-7 objects, either natively or through a wrapper.

I hope you learned something new. You can always watch one of my talks online[8], or tinker with some demos[9].

Hannes van Belgian, Software Engineer at madewithlove, open source user and contributor, blogger, organiser of meetups and also a marathon runner.
@hannesvdvreken

8 *my talks online: https://youtu.be/gOVALgpqHzM*

9 *tinker with some demos: https://github.com/hannesvdvreken/psr7-demo*

Integrating With APIs

Caitlin Bales

REST-based APIs have emerged as a central feature of tech companies' business strategies. Businesses are realizing an API strategy brings them more developer integration, platform stickiness, and development speed. Because these APIs depend on a common set of REST architectural patterns most developers are familiar with, the ramp-up time for the business as well as its customers is very short.

The API Industry

The use cases for APIs are nearly limitless, and the rationale to utilize them is simple: it allows you to build on the work of others quickly and increases the scope of data you can work with. As opposed to full-suite software packages, developers can pick and choose which endpoints bring relevant data into their application and work on their business-differentiated software instead of re-implementing existing features. This lends itself well to developing applications with a microservice architecture, because functionality can be modularized into discrete internal and external APIs.

In this article, we'll walk through the steps required to integrate with an API:

- Communicating with the API using REST
- The request recipe
- How to call an API using cURL or Guzzle
- Authorizing your application
- Debugging your requests

RESTful Web Services

REST is a message-based pattern for developing web services that are centered around resources. Its big selling point is it provides a common transactional model for data transfer. Many APIs that previously communicated with REST's older brother, SOAP, are switching to using REST for its ease of use. SOAP has a high request overhead due to its use of XML to define the data being transferred. It is also highly restrictive on what and how you use its protocol. Using JSON with REST, requests are far more lightweight, and developers have more control over how they define their APIs.

REST can send and receive several types of data, including JSON, XML, and plaintext. JSON is a relatively common choice due to its readability. It is easy to serialize and deserialize JSON data in PHP using the `json_encode` and `json_decode` functions. The `jsonSerialize` method of the JsonSerializable class[1] can also be extended, allowing even greater flexibility with how your data is returned.

1 *JsonSerializable class:* http://php.net/jsonserializable.jsonserialize

The Request Recipe

The REST request recipe looks like this:

```
VERB + HEADERS + URL + BODY
```

Verbs describe what you want to do with the specified endpoint. `GET` and `POST` are the most common, however many sites also use actions such as `PUT`, `PATCH`, and `DELETE`. It is also possible for a service to define its own HTTP verbs, although this is generally discouraged as it provides less standardization across services.

Common usages of verbs are:

- `GET`: get information about an item
- `POST`: call an action
- `PUT`: create a new item
- `PATCH`: update an existing item
- `DELETE`: remove an item

Headers define information about the request, such as the content length and type. The most important header to include in calls to secured APIs is the `Authorization` header. This will provide the service with an access token to unlock the user's data. Services can also define their own custom headers which are unique to the specific application.

The URL defines the endpoint you want to hit. A common URL pattern is `address/version/endpoint`, where `endpoint` is a noun describing a resource in the system.

Finally, some methods require a body to be included in the request (i.e. `POST`, `PUT`, and `PATCH`). This is the information you will send across the wire to update the resource.

The Response Object

The REST response you receive back from an API will look like this:

```
STATUS CODE + HEADERS + BODY
```

The status code indicates the state of the request. Common status codes for API responses include:

- 200: OK
- 201: object created
- 400: bad request

- 401: unauthorized to use this endpoint
- 500: server-side error
- 504: server timeout

Successful PATCH and DELETE requests usually return a 201 or 204(created) status code and do not include a response body or headers.

Headers will include information about the request, such as the content length and type.

If you are performing a GET or POST, you will also receive a response body. A GET request returns the information you requested and a POST returns the object you created or modified.

Calling APIs

There are several ways to call an API from your PHP code:

- **cURL**[2]: a library which is usually included with the default installation of PHP through the libcurl package. You can also use this without PHP in the terminal to send raw requests. It's great for when you're starting out or for creating one-off requests inside a script or program.
- **PECL_HTTP**[3]: part of the PECL package. Depending on your coding style, you may find this more intuitive to code with than cURL.
- **Guzzle**[4]: a Composer package which aims to simplify the process of making requests. It's great if you are heavily relying on API calls to your application. It can also be configured to use a different HTTP handler than cURL if necessary.
- **Additional cURL wrappers**: many developers have created cURL wrappers to handle specific scenarios to make coding simpler and cleaner. If your API integration is strongly tied to your application, try searching for a package that handles the particular difficulties of your use case, such as asynchronous call patterns, security implementations, etc.
- **PHP Streams**: using `file_get_contents` on a URL allows you to make lightweight requests to an API and decode the responses in chunks. This is useful if you are working with large response bodies, such as files.

Calling an API With cURL

At its most basic, a cURL call can be made with the four following commands:

- `curl_init`: creates a new cURL handler
- `curl_setopt`: sets the values for the request. Common options are:
  - `CURLOPT_URL`: the URL to connect to (required)
  - `CURLOPT_POSTFIELDS`: sets the body of the request
- `curl_exec`: executes the call
- `curl_close`: closes the handler

By default, all cURL requests are GET requests. Listing 1 shows an example GET request. cURL options can be set for POST, PUT, or CUSTOMREQUEST.

Listing 2 shows how to configure cURL to POST.

Listing 1.

```php
<?php

$connectUrl = "https://api-example.com/posts/1";

// Initialize the handler
$curlHandler = curl_init();

// Set connection URL
curl_setopt($curlHandler, CURLOPT_URL, $connectUrl);

// Execute the call
$result = curl_exec($curlHandler);

// Close the handler
curl_close($curlHandler);
```

Listing 2.

```php
<?php

$connectUrl = "https://api-example.com/posts/1";
$data = array(
    "id" => 1,
    "userId" => 3,
    "title" => "POSTing with cURL",
    "body" => "Successful post"
);

$curlHandler = curl_init();
curl_setopt($curlHandler, CURLOPT_URL, $connectUrl);

// Set the POST body
curl_setopt($curlHandler, CURLOPT_POSTFIELDS, json_encode($data));

$result = curl_exec($curlHandler);

curl_close($curlHandler);
```

2 *cURL: http://php.net/book.curl*

3 *PECL_HTTP: https://pecl.php.net/package/pecl_http*

4 *Guzzle: http://phpa.me/guzzle-docs*

Calling an API With Guzzle

Guzzle handles boilerplate request and response codes for you to reduce development time. It is also object-oriented, as opposed to the functional style of cURL. It ships with mocks for testing purposes. Setup for Guzzle is similar to cURL (see Listing 3).

- Create new `GuzzleHttp\Client` with `base_uri`
- Call `$client->request(<verb>, <endpoint>)`
- Call `$result->getBody()->getContents()` to retrieve response JSON

Its support for asynchronous calling makes it easy to handle requests efficiently, as you can see in Listing 4. Async calls come in handy when updating large objects or performing server-intensive calculations by allowing your application to process multiple requests at the same time.

Authorizing Your Application

To use most APIs, you need to authorize your application, and potentially authenticate the user interacting with your application. This can often be the hardest part of connecting to a service, but it ensures the data that is transmitted is secured and allows users to trust your applications with their data.

OAuth 2[5] is the most common protocol for authorization. It was created to simplify and standardize the authorization flows for developers and define a vocabulary around these principles.

OAuth will allow you to authorize your application, but if you want your users to authenticate against the API (using login credentials, for example), you will need another layer on top of OAuth. Several companies provide identity services for this purpose. There are several ways to implement this, but many identity providers use OpenID Connect[6]. This is an authentication layer built on top of OAuth and integrates with the OAuth flow to allow you to authorize and authenticate in the same workflow.

To get started with the authorization process, most companies require you to register your application with them. This process will grant you a public/private key pair, sometimes called a client ID and client secret. If you are developing a web app, you will also supply a callback URL, which will send your users back into your application's workflow after authenticating.

There are several authorization flows that OAuth 2 defines, which are good for different types of applications and scenarios:

- **Authorization code flow**: most common way to authorize. Allows a user to log in to a third-party site and grant your application access to its data. It's secure because the client application doesn't get access to any sensitive login data.
- **Implicit grant flow**: useful for client applications that want to authorize on the frontend. Once a user's session has expired, she will have to log in again.
- **Client credential flow**: used when no user interaction is required. The app authorizes only with a client ID

5 *OAuth 2: https://oauth.net/2/*

6 *OpenID Connect: http://openid.net/connect/*

Listing 3.

```php
<?php

use GuzzleHttp\Client;

// Create a new Guzzle client
$guzzleClient = new GuzzleHttp\Client([
    "base_uri" => "https://api-example.com"
]);

// Create a new POST request
$guzzleResult = $guzzleClient->request(
    "POST",
    "/posts",
    ["body" => $requestBody]
);

$guzzleResult->getStatusCode();

// Get result body as a stream
$guzzleResult->getBody();

// Get result body as JSON
$guzzleResult->getBody()->getContents();
```

Listing 4.

```php
<?php

use GuzzleHttp\Client;

$guzzleClient = new GuzzleHttp\Client([
    "base_uri" => "https://api-example.com"
]);

// Create a new async GET request
$promise = $guzzleClient->requestAsync(
    "GET",
    "/posts"
)->then (
    // If the async request succeeded
    function ($result) {
        echo $result->getBody()->getContents();
    },
    // If the async request failed
    function ($reason) {
        error_log("Async call failed: " . $reason->getMessage());
    }
);
```

and secret. Great for scripts and command line applications.

- **Resource owner credential flow**: only suitable for first-party app authorization. It requires you to already have and store your users' credentials.

Both the authorization code and implicit grant flows support authentication. The API provider will give guidance on which authentication layer they use and how to pass users through the flow.

Debugging Tools

It is often important to isolate the raw request and response being sent between servers to understand where REST errors are originating. In addition to debugging information provided by the HTTP client, there are several third-party tools which allow you to inspect data sent across the wire.

Postman[7] started as a simple Chrome plugin for making HTTP requests but has grown into a full desktop application available on Windows, OS X, and Linux. It allows you to test queries quickly without a shell application. It's ideal for testing an API before deciding to integrate it into your application. It also comes with lots of other goodies, including saved queries, query collections, test cases, documentation, and the ability to share requests. You can generate code snippets from Postman into both cURL and PECL_HTTP and paste them directly into your application.

Fiddler[8] and Charles proxy[9] are both HTTP debugging proxy servers. They capture data sent across the wire from internet, intranet, and native application sources. They also allow you to edit requests to see how changes are reflected in the response quickly. You can intercept requests and modify them before responding, or send your own requests similar to the way you would

Figure 1.

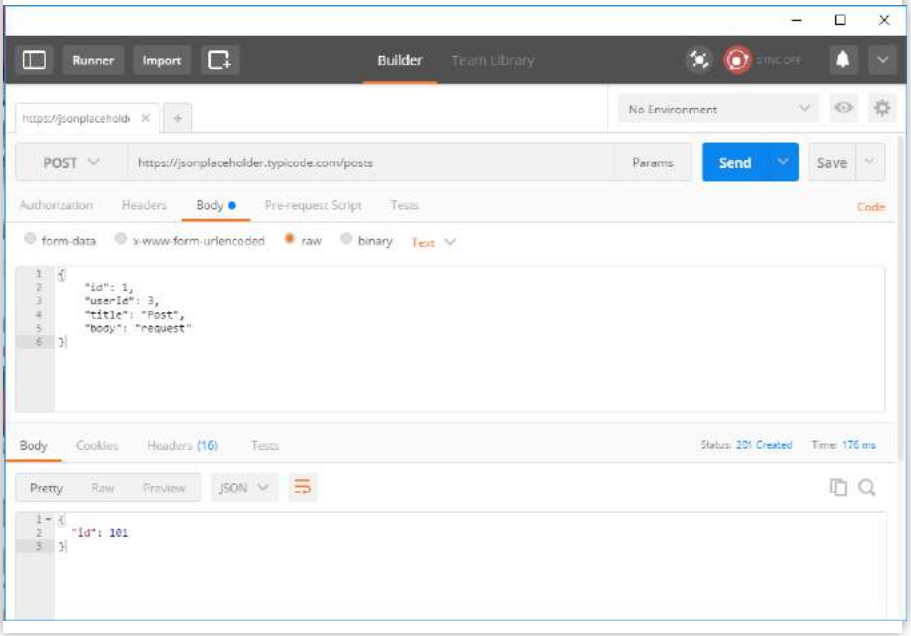
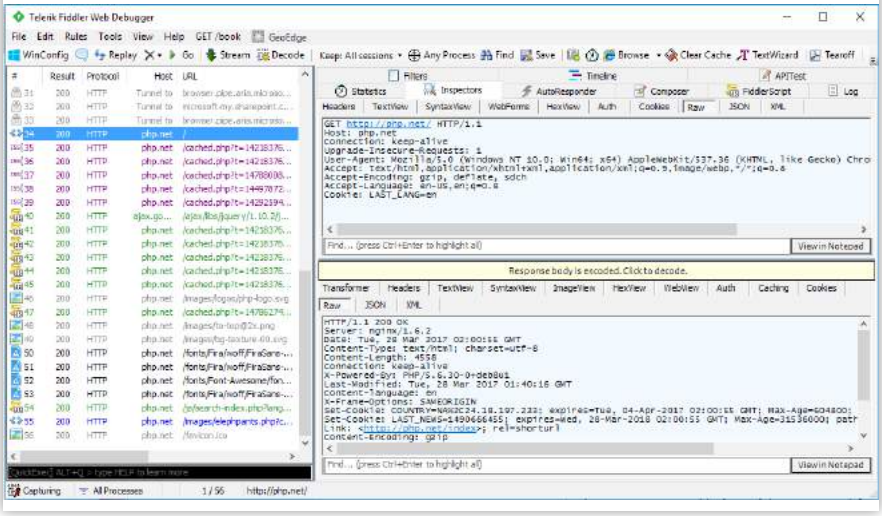


Figure 2.



in Postman.

Modern browser tools, such as DevTools in Chrome and Edge also allow you to see web traffic, but the results are limited to a particular web page. This allows you to see all of the requests and responses you receive, as well as the headers, bodies, and cookies being passed. Network tools also allow you to gather insights that can be used to optimize these calls for faster page load times.

7 *Postman:* https://www.getpostman.com

8 *Fiddler* http://www.telerik.com/fiddler

9 *Charles proxy:* https://www.charlesproxy.com

Listing 5.

```php
<?php
/**
* File: ApiClient.php
*
* Class ApiClient
*
* Client to send requests to the API server
*/
class ApiClient {
    /**
    * A valid access token. In some APIs, this may be a
   * clientId or public key instead
    *
    * @var string
    */
    protected $accessToken;

    /**
    * The base URL to call
    *
    * @var string
    */
    public $connectUrl;

    /**
    * Headers for the request
    *
    * @var array (string => string)
    */
    public $headers;

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

    /**
    * Set the current access token for the client
    *
    * @param string $accessToken A valid access token
    */
    public function setAccessToken(string $accessToken) {
        $this->accessToken = $accessToken;
    }

    /**
    * Call the API using cURL
    *
    * @param string $method    The method to use in the request
    * @param string $endpoint The endpoint of the URL to call
    * @param mixed  $body      The request body
    * @param array  $headers  Additional headers to send w/request
    *
    * @return array $response The response from the API server
    */
    public function callApi(string $method, string $endpoint,
                            $body = null, array $headers = []) {
        // Verify that the access token has been set
        if ($this->accessToken) {
            $this->headers = array_merge(
                $this->_getDefaultHeaders(), $headers
            );
        } else {
            throw new Exception('Call setAccessToken before calling the API');
        }

        $curlHandler = curl_init();

        // Set the body of the request
        if ($body) {
            $body = json_encode($body);
            $this->headers[] = ['Content-Length: ' . strlen($body)];
            curl_setopt($curlHandler, CURLOPT_POSTFIELDS, $body);
        }

        // Set parameters of the request
        curl_setopt_array(
            $curlHandler,
            [
                CURLOPT_RETURNTRANSFER => true,
                CURLOPT_URL => $this->connectUrl . $endpoint,
                CURLOPT_CUSTOMREQUEST => $method,
                CURLOPT_FAILONERROR => false,
                CURLOPT_HTTPHEADER => $this->headers
            ]
        );

        // Send the request
        $response = curl_exec($curlHandler);

        curl_close($curlHandler);

        // Return the decoded response
        return json_decode($response, true);
    }

    /**
    * Get a list of default request headers
    *
    * @return array $headers A list of headers
     */
    private function _getDefaultHeaders() {
        $headers = array(
            'Content-Type' => 'application/json',
            'Authorization' => 'Bearer ' . $this->accessToken
        );
        return $headers;
    }
}
```

php[architect] SWAG

Our CafePress store offers a variety of PHP branded shirts, gear, and gifts. Show your love for PHP today.

www.cafepress.com/phparch

ElePHPants

Laravel and PHPWomen Plush ElePHPants

Visit our ElePHPant Store where you can buy purple or red plush mascots for you or for a friend.

We offer free shipping to anyone in the USA, and the cheapest shipping costs possible to the rest of the world.

www.phparch.com/swag

Putting It All Together

Once you're ready to integrate this code into your application, you can create an `ApiClient` to handle your requests. This will contain the core logic for making your requests, including setting an access token, public key, or client ID; setting request headers; attaching request bodies; sending a request to the service; and decoding the response you receive back (see Listing 5).

You can then use this through your controller, as in Listing 6, by instantiating a new `ApiClient` object. You can use this to make multiple requests against the API. You could also create multiple `ApiClients` to make requests against multiple APIs.

Conclusion

There is far more you can do with APIs than just single sign-on and data pulls. As development becomes more accessible and continues the adoption of open source standards and practices, API integration stories will flourish and diversify, while the underlying principles will strengthen. I encourage you to dig into the docs of your favorite APIs to see what else is possible through their services.

Caitlin Bales is a community engineer who enjoys creating great web experiences for both end-users and developers. She has been coding in PHP both in the startup world and in enterprise. She is currently working on API tools for http://graph.microsoft.io and covertly evangelizing PHP at Microsoft.

Listing 6.

```php
<?php
/**
* File: UserController.php
*
* Class UserController
*/
use ApiClient;

class UserController {
  /**
   * The ApiClient to make requests
   *
   * @var ApiClient
   */
   public $client;

  /**
   * The user's name
   *
   * @var string
   */
   public $name;

  /**
   * The user's email address
   *
   * @var string
   */
   public $email;

   public function __construct() {
       $this->client = new ApiClient("https://connect-url");
       $this->client->setAccessToken("access-token");

       $this->name = $this->getName();
       $this->email = $this->getEmail();
   }

  /**
   * Get the user's name
   *
   * @return string Name
   */
   public function getName() {
       $response = $this->client
                        ->callApi("GET", "/me?select=name");
       return $response["name"];
   }

  /**
   * Get the user's email address
   *
   * @return string Email address
   */
   public function getEmail() {
       $response = $this->client
                        ->callApi("GET", "/me?select=mail");
       return $response["email"];
   }

  **
   * Send an email from the user
   *
   * @param string $recipientEmail The recipient email address
   * @param string $subject The subject of the email
   * @param string $body The contents of the email
   */
   public function sendEmail(string $recipientEmail,
                             string $subject, string $body) {
       $body = [
           "from-email" => $this->email,
           "to-email" => $recipientEmail,
           "subject" => $subject,
           "body" => $body
       ];

       $this->client->callApi("POST", "/me/sendEmail",
                              null, $body);
   }
}
```

Demystifying Multi-Factor Authentication

Brian Retterer

Account security is a hot topic among developers and software users. No dev wants to be responsible for the next big "user accounts breached" headline. With every new headline, users are becoming more concerned about their security, and it's our responsibility to create products our users can trust. But trust between your application and your users can be difficult to obtain. One way we gain that trust is by providing a secure browsing experience via an SSL certificate on your server. We also gain trust by securing user passwords using industry standards such as bcrypt.

As developers, we know that one of the best ways to assure account security and grow user trust is by offering or enforcing Multi-Factor Authentication. A frequent recommendation is to "enable two-factor authentication your account." In this article, we'll look at what this means and how you can implement it in your applications.

What Is Multi-Factor Authentication?

There are many different forms of Multi-Factor Authentication. Even the words used to describe it have a few different versions. You may hear people talk about multi factor, 2-factor, MFA, factored authentication, or more. These are all the same idea; you must have something you know and something you have to authenticate and gain access. The **something you know** part comes from your username and password combination where the **something you have** comes in many different forms. This can be a physical device, such as your cell phone, or even a one-time-use code that is emailed to you.

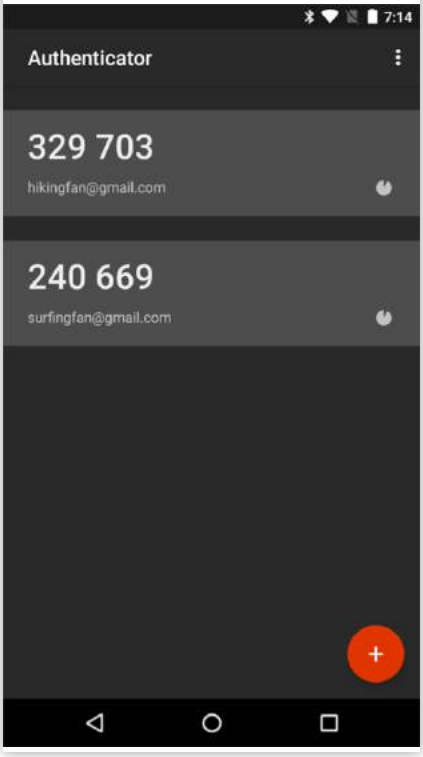
The majority of you have likely experienced Multi-Factor Authentication in some form or another. This is a safe bet for me to make because Multi-Factor Authentication is all around you. If you remember the two main requirements, something you know and something you have, we can apply them to everyday life. A transaction at an ATM fits our criteria. You know your pin number, and you have your debit card.

In the technology world, we have a few options available to us. The aforementioned case of an email being sent with a one-time-use code is one of the most common ways of implementing MFA. Although this is the method I'll be showing you how to set up, I'd be remiss if I didn't cover the other secure options that you have as a developer. A newer standard for handling MFA is to allow the use of a Time-Based one-Time Password, or TOTP. The most common method of implementation would be to use Google Authenticator.

Google Authenticator is a trust between a device and the authentication server. Three items are working together to create a successful login. A shared secret encoded with the Base32 algorithm that your device and the authentication server know, the current time, and a signing function. Once your application server creates the secret, and allows the user to input, or scan a QR code, the user will be given a 6-digit code which is only good for one minute (plus or minus a leeway) that can be used as part of their login.

Another form of MFA you may be more familiar with is a code being sent over SMS. This method has come under scrutiny over the last few years as it is becoming less and less secure, given someone could intercept the code over SMS or pose as your service provider. However, it works essentially the same as all of the other methods. A code

Figure 1.



which is associated with the user trying to log in is sent to the user's registered cell phone number. During the login, they are asked for this code. Once the code has successfully been used, or unsuccessfully used after a set amount of time (usually one to five minutes) the code is removed, and a new one will be generated at the next login.

> **Why is SMS less secure?** *A determined attacker could hijack the messages. They could socially engineer your service provider to send them to a different phone. Government friendly telecom companies could hand them over to the authorities, and fake cell phone towers could intercept them. For more see So Hey You Should Stop Using Texts for Two-Factor Authentication*[1]

How to Implement MFA In Your Project

Now that you know what MFA is and some of the types available, let's look at how you can implement it in your application. The most universal system for MFA is a one-time-use code that is emailed to the user after a matched username and password combination. This article is going to be using a basic scaffolding of the Laravel framework. We will be using the `make:auth` command to provide some basics and will modify from there. If you are unfamiliar with Laravel, the general concept will be the same, but you may need to research how to integrate it with your specific framework. The final code for this project can be cloned from bretterer/email-multi-factor-authentication[2].

To get started, let's create a new Laravel project:

```
composer create-project laravel/laravel \
        email-multi-factor-authentication
```

Once the Laravel project has been created, we need to initialize the authentication system:

```
php artisan make:auth
```

Now, if you were to visit the site in the web browser, you would see our login and register views. We will need to set up some database connections as well as run our migrations.

In your `.env` file:

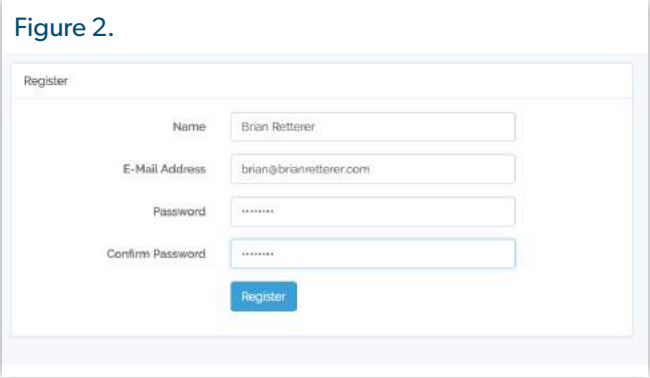
```
DB_CONNECTION=sqlite
MAIL_DRIVER=log
```

Then run:

```
php artisan migrate
```

Once you have all of that setup, you should be able to go to `/register` and register for a new account. You'll see the login form in Figure 2.

Figure 2.



Since Laravel uses the email address as the default for the username field, we have everything we need to continue with our modifications to require a second factor for authentication. Looking back, what we need to have for a factored authentication with email is: (a) something the user knows and (b) something the user has. The user knows their username and password, so we need to provide a code to their email, something they have, so they can fully log in. Since we want the codes to expire after a set amount of time, let's say five minutes, we'll need to create a new table to store all the codes. Let's create and run a new migration as in Listing 1.

Listing 1.

```
1. public function up()
2. {
3.     Schema::create('mfaCodes', function (Blueprint $table) {
4.         $table->increments('id');
5.         $table->integer('user_id');
6.         $table->string('code');
7.         $table->timestamp('expires');
8.         $table->timestamps();
9.     });
10. }
```

First, stub out the migration class file, which will be in the `database/migrations` directory. Then update the `up` method.

```
php artisan make:migration create_mfa_codes_table
```

Then, run the migration:

```
php artisan migrate
```

The remainder of this article assumes you know how to use artisan to scaffold any code we need. If stuck, refer to the project on GitHub.

Next, need to "hijack" the login Laravel does for us since we don't want to set any of the cookies for the user until they successfully input the code that was emailed to them. To do so, we will need to create our own login method which validates the user credentials, and if it is valid, sends an email to the user with a random code which we store in our database for later use (See Listing 2).

1 *So Hey You Should Stop Using Texts for Two-Factor Authentication:* http://phpa.me/wired-sms-2fa

2 *bretterer/email-multi-factor-authentication:* https://github.com/bretterer/email-multi-factor-authentication

Listing 2.

```
1. public function login(Request $request)
2. {
3.     $this->validateLogin($request);
4.
5.     if ($this->hasTooManyLoginAttempts($request)) {
6.         $this->fireLockoutEvent($request);
7.
8.         return $this->sendLockoutResponse($request);
9.     }
10.
11.    $credentials = $this->credentials($request);
12.    $loginCheck = $this->guard()->validate(
13.        $credentials
14.    );
15.
16.    if ($loginCheck) {
17.        $code = $this->createOneTimeUseCode($credentials);
18.        $this->emailOneTimeUseCode($credentials, $code);
19.
20.        return redirect('/login/challenge');
21.    }
22.
23.    $this->incrementLoginAttempts($request);
24.
25.    return $this->sendFailedLoginResponse($request);
26. }
```

The majority of the login method will remain the same, but where we need to take over is using the `validate()` method on the `guard` class instead of the `attempt()` method. You will see these changes on line 11. You will also see we need a new method to save a new code into our table which will be used to email, as well as a method to actually email the user. Now, both of these methods (see Listing 3) are inside of the `LoginController`, but there is no reason they have to live there. You can place these anywhere in your application.

Listing 3.

```
1. private function createOneTimeUseCode($credentials) {
2.     $randomNumber = random_int(100000,999999);
3.
4.     $user = User::where('email', $credentials['email'])
5.                 ->first();
6.
7.     DB::table('mfaCodes')->insert([
8.        'user_id' => $user->id,
9.        'code' => $randomNumber,
10.       'expires' => time() + 300
11.    ]);
12.
13.    return $randomNumber;
14. }
15.
16. private function emailOneTimeUseCode($credentials, $code) {
17.    Mail::to($credentials['email'])
18.        ->send(new LoginCode($code));
19. }
```

Listing 4.

```
1. class LoginCode extends Mailable
2. {
3.     use Queueable, SerializesModels;
4.
5.     /**
6.      * Create a new message instance.
7.      *
8.      * @return void
9.      */
10.    public function __construct($code) {
11.        $this->code = $code;
12.    }
13.
14.    /**
15.     * Build the message.
16.     *
17.     * @return $this
18.     */
19.    public function build() {
20.        return $this->view('mail.logincode')
21.            ->with([
22.               'code' => $this->code
23.            ]);
24.    }
25. }
```

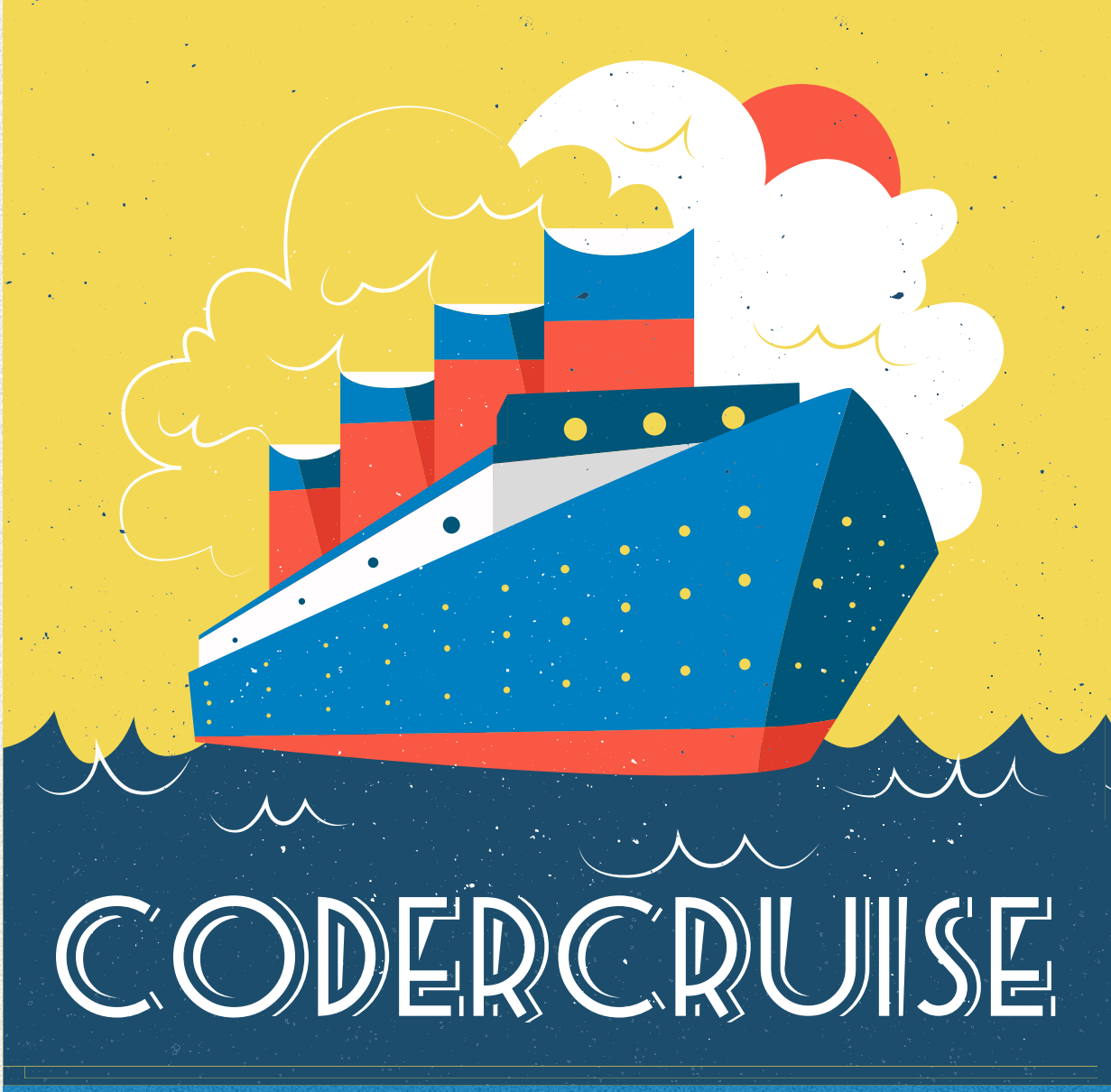
We also need to add a new route, as in Listing 4, for the `/login/challenge` endpoint where our form will live for the user to enter the code sent to their email. We also need a new `Mailable` class that will handle the email sending. If you are unsure what the Laravel `Mailer` class is, or how to use it, you can find more information in the Laravel documentation[3].

We are now in the home stretch of the setup for our emailed code factored authentication. The last step is handling a way for the user to enter the code we just sent them. We'll need a few additions to our login controller, a couple of routes, and a new view. Let's begin with the view, shown in Listing 5. Copy the current login view and modify it slightly since it is close to what we need. Take out the password field, and change the email field over to a code text field. Make sure to also change all the error checks over to a key of `code`. You should also remove any of the extra links on the page that are no longer needed, such as the "forgot password" link.

Next, we add the routes:

```
1. Route::get(
2.     '/login/challenge',
3.     'Auth\LoginController@challenge'
4. );
5. Route::post(
6.     '/login/challenge', [
7.        'uses'=>'Auth\LoginController@validateChallenge',
8.        'as' => 'login.challenge'
9. ]);
```

3 *Laravel documentation:* https://laravel.com/docs/5.2/mail



CODERCRUISE

7 days at sea, 3 days of conference

Leaving from New Orleans and visiting
Montego Bay, Grand Cayman, and Cozumel

July 16-23, 2017 — *Tickets $295*

www.codercruise.com

Presented by One for All Events

And, finally, we add the methods in Listing 6 to our controllers.

The `validateChallenge` method is where all the checks happen. The first thing we do is remove any expired codes from the database to keep things clean. We then find a code in the database that matches the code submitted, also checking to make sure the code is not expired. If a code is found, we get the user ID associated with the code and log the user in using the built in Laravel functionality. If a code is not found, we send the user back to the challenge page with an error.

There are a few flawed items in this example. For instance, we are not asking for the email again from the user. This makes this system a little more vulnerable to brute force attacks, meaning attempts could be made against the challenge page over and over with different numbers until they get one that works. One way to fix this would be asking for the email address again for the code and seeing if it matches the expected user. Another option would be to set a cookie during the first step of the login with a unique code, and then in the email you send, have a link that takes the user to a URL with a parameter that has the same code. There are many other options available for helping solve this problem, and I leave it up to you to decide what is best for your application.

Listing 5.

```
1. @extends('layouts.app')
2.
3. @section('content')
4. <div class="container">
5.   <div class="row">
6.     <div class="col-md-8 col-md-offset-2">
7.       <div class="panel panel-default">
8.         <div class="panel-heading">Multi Factor Code</div>
9.         <div class="panel-body">
10.          <form class="form-horizontal" role="form"
11.                method="POST"
12.                action="{{ route('login.challenge') }}">
13.            {{ csrf_field() }}
```

For the full listing, see this month's code archive

Listing 6.

```
1. public function challenge() {
2.    return view('auth/challenge');
3. }
4.
5. public function validateChallenge(Request $request) {
6.    DB::table('mfaCodes')->where('expires', '<', time())
7.                         ->delete();
8.    $code = $request->get('code');
9.
10.   $codeEntry = DB::table('mfaCodes')
11.                  ->where('code', $code)
12.                  ->where('expires', '>', time())
13.                  ->first();
14.
15.   if ($codeEntry) {
16.       $this->guard()->loginUsingId($codeEntry->user_id);
17.       return $this->sendLoginResponse($request);
18.   }
19.
20.   return redirect()->back()
21.                    ->withErrors([
22.                       'code' => 'That code is not valid'
23.                    ]);
24. }
```

What's Next?

There are many places you can go with this. Although I used Laravel for the sample code, the same logic can be used in virtually any system with authentication.

The sample provided is what I like to call a poor man's version of Multi-Factor Authentication, mainly because it is not as secure as some other options out there. Setting up some of the other options, such as Google Authenticator, take much of the same processes as the sample above. To set those up requires a slightly different logic and some extra setup for the end user.

I'd love it if one day every application required some form of factored authentication to help keep my accounts safe. Now that you know the basics of setting this up, go out and re-vamp your applications to offer this extra step of security to all of your users.

Brian Retterer is a developer advocate at Okta, a Silicon Valley cloud-based identity service. He is focused on serving the PHP community, and always excited to represent and educate developers, especially in the Midwest. From his home base in Ohio, you can often find him advocating for best practices in account security and RESTful API design. You can follow Brian on Twitter at @bretterer

Rock Your Deployments With Rocketeer

Matthew Setter

There are many ways to deploy code, ranging from the very simple to the rather complex. Regardless of how you do it, to help you out, in this month's Education Station, I'm going to walk you through a tool which makes your deployments virtually painless. What's more, it's written in pure PHP; it's easy to get started with, easy to use, and easy to extend.

It's called Rocketeer[1]. But before we dive in too deeply, let's first ask the question: why another deployment tool? This is especially pertinent when there are so many already available. These include *Deployer*, *Capistrano/Webistrano*, *Phing*, *Ant*, and *Make*; and hosted services, including *Codeship*, *simple-ci*, *CircleCI*, and *Travis CI*.

Here's my short answer: *there's more than one way to skin a cat*. Just because one tool works for a large number of people, doesn't mean that it works, nor is suitable, for everyone. I've used Deployer, Phing, Make, and Capistrano. While I like all of them, there are little things, here and there, that don't quite work for me.

Speaking purely for my PHP apps, to me they're not *quite* the right fit. Taking Capistrano as a good example, while it's a very feature-rich and capable tool it's (at least when I last used it) heavily weighted to Ruby and Ruby on Rails apps. Given that, there are some assumptions baked in about how your apps work and what you need to do to deploy them.

These aren't deal breakers. But I want to use something that's as close to what I want as possible, something that's intimately aware of PHP, something which appreciates its quirks and ways of doing things. I also want a tool with a sufficiently detailed commit history, one actively supported, feature rich, and that understands the nomenclature of modern deployment techniques.

More specifically, the reason why I chose to begin using Rocketeer recently, is because one of my side-project applications, the one for my podcast[2], didn't have an automated deployment process—at least up until recently.

I'd heard about Rocketeer before, so I decided to review the repository and documentation to get a gauge of the project's health. In doing so, it was clear the project was alive and well. At the time of writing, the most recent commit was five days ago. Additionally, the documentation was well laid out and of sufficient depth. And the command line interface seemed to have all the options I needed.

I dove in and gave it a go and was singularly impressed by both its simplicity and effectiveness. As a result, I want to walk you through the essentials and hope that you'll be as convinced as I was.

What Is Rocketeer?

My enthusiastic spiel aside, what is Rocketeer? Well, it's not the movie of the same name from 1991[3], starring Billy Campbell and Alan Arkin. No, Rocketeer is a deployment tool, one written in pure PHP. It supports all of the standard options you'd expect from a modern deployment tool, including the ability to:

- Deploy multiple deployments (up to specified amount)
- Rollback to a previous deployment.
- Perform a pretend deployment, to ensure everything works, without actually deploying anything.
- Setup the remote server(s) in preparation for a deployment.
- List the currently deployed version(s).

In addition, the core functionality is extendable via plugins, plugins which can perform a variety of actions, including sending notifications to *Slack channels* and to *HipChat*—those and a whole lot more.

How to Install

With the "sales pitch" out of the way, let's get in and install

Listing 1.

```
# download the latest copy of the Rocketeer Phar file
wget http://rocketeer.autopergamene.eu/versions/rocketeer.phar

# set the executable bit
chmod +x rocketeer.phar

# move it to a place in the system-wide path
sudo mv rocketeer.phar /usr/local/bin/rocketeer
```

1 *Rocketeer:* http://rocketeer.autopergamene.eu

2 *my podcast:* http://freethegeek.fm

3 *the movie of the same name from 1991:* http://phpa.me/rocketeer-film

it. When completed, I'm going to step through the core functionality you'll use on a regular basis. Like most PHP packages, there are several ways to get started.

Usually I'd suggest using Composer to install a package. But this is a tool you'd use with multiple applications, not just one. So, we'll install the phar version instead, using the commands in Listing 1.

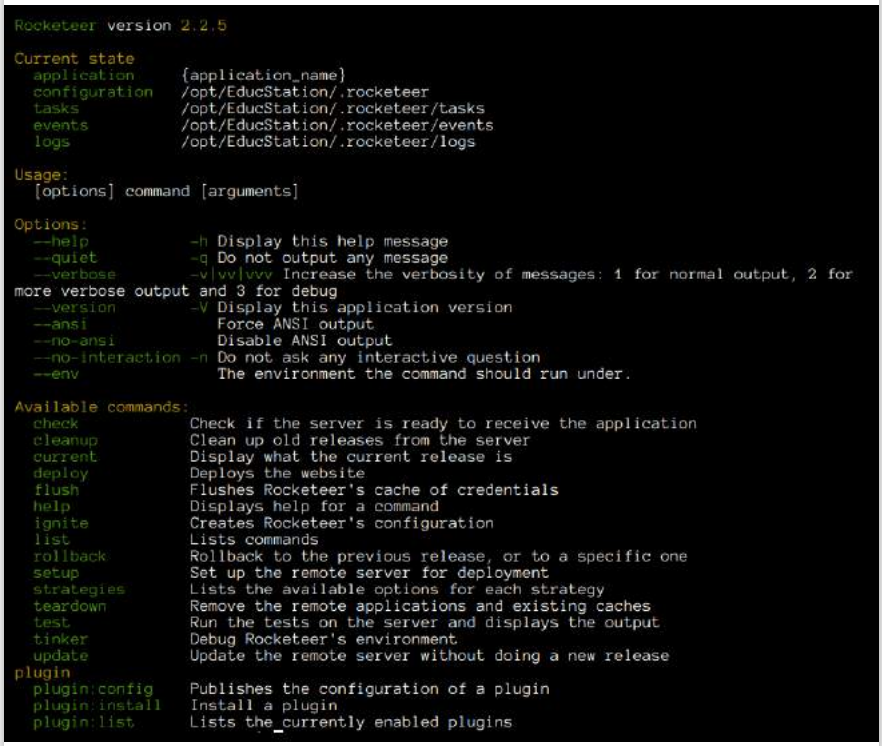
If you're on Windows, use your downloader of choice to download the phar file, and then place it somewhere in your system path or adjust the system path to include the download directory.

If you do want to use Composer, however, I'd suggest installing it globally by running:

```
composer global require anahkiasen/rocketeer
```

Regardless of the installation approach chosen, after you've done all this, test that it's available by running `rocketeer` from the command line. When installed successfully, you should get output similar to Figure 1.

Figure 1.

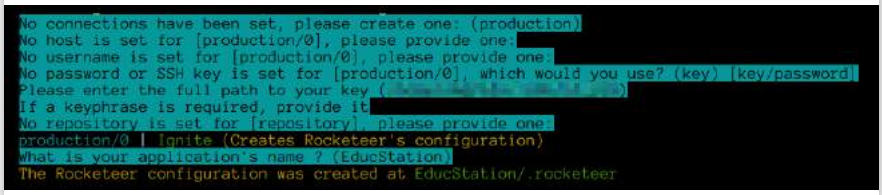


How to Use Rocketeer

Once you have it installed successfully, let's get underway and use it to deploy a project. In the terminal, from the root directory of a project which you want to deploy, run the command `rocketeer ignite`. This will, similar to `git init` and `composer init`, create an initial Rocketeer configuration setup for the current project. Rather an apt name, wouldn't you agree?

During the process, you will be asked a series of questions. To save time, here's what you can expect (see Figure 2).

Figure 2.



To summarize, it gathers enough to create a single deployment configuration. With that information it creates the core configuration files, seeding them with the information you provided, along with default values for everything else. Assuming everything you provided was correct, you'd almost be able to deploy your application at this point.

But let's not be too hasty. Let's have a quick look at the generated configuration, which you can find in a new directory in your project, called `.rocketeer`. In it, you'll see seven files:

1. `config.php`: the core configuration, containing the application's name, what plugins to use, where to store log files, the environments to deploy to and their settings.
2. `hooks.php`: where you specify actions to take both before and after the setup, deploy, and cleanup processes.
3. `paths.php`: where you set the paths to the required binaries.
4. `remote.php`: This is where you set details about the remote servers, such as the root deployment directory, how many releases to keep, the name of the directory to deploy to, whether to use sudo, and the permissions to set on the deployed files.
5. `scm.php`: This is where you store the settings for your source control tool, such as the repository URI, credentials, and the branch to use.
6. `stages.php`: where you manage details about the staging environments (such as deployment, testing, staging, and production).
7. `strategies.php`: where you can configure strategies to use as part of the deployment process.

Validating Your Configuration

With the configuration created, it's time to validate the configuration. We

can do this with Rocketeer, rather like we would lint a PHP file or validate a Composer configuration file, to be sure there are no errors. To do this run `rocketeer check`. All being well, you'll see output similar to Figure 3.

One assumption you should be aware of is Rocketeer expects your project to be using Composer. If you don't have a `composer.json`, the `rocketeer check` command will fail.

There, you can see that the remote server has all of the necessary *binaries*, *extensions*, and *drivers* required for a successful deploy. What's more, you'll also have a log file, located under `.rocketeer/logs`, which also contains this information.

Before we move on, this log file will continue to be updated as you use Rocketeer. So if you're curious to keep track of what it does, or to debug a problem, make sure you periodically check it or tail it.

Deploying a Release

Now, we're ready to run the first deploy. To do so, run `rocketeer deploy`. You should see the output, similar to the abridged version in Figure 4, which shows all the steps Rocketeer executes. You'll want to ensure that your SSH user will have write access to your target directories.

These include:

- Performing a shallow clone of your project's repository.
- Initializing Git submodules.
- Installing the project's dependencies via Composer.
- Generating autoload files.
- Setting ownership and access permissions.
- Setting the release as the current version
- Pruning any old releases.

Now let's analyze a release. Let's assume you configured Rocketeer to deploy to `/var/www/deploy/podcast-site` on your remote host (it also could have been local). Inside `/var/www/deploy/podcast-site`, it will create two directories: *releases* and *shared*. Within the `releases` directory will be a directory named with the release's timestamp. This, in turn, will be symlinked to *current*, alongside releases and shared.

> *Rocketeer's folder architecture is inspired by Capistrano's. If you're not familiar with these aspects, the online documentation does a good job of explaining it in the section on Core folders[4]. Retrofitting an existing site to use Rocketeer may be difficult. Instead, start deploying to a new path on your servers and then switch your web server to use the new document root.*

If this is your first time working with a deployment tool, releases will contain the most recent deployments. Each directory will be a shallow clone[5] of your application's code.

> *When you create a shallow clone with Git, you're not grabbing all the commits in a repo. Instead, you get a defined number of commits for a branch. The limit could be a set number of commits or only commits since a specified date.*

By working this way (and symlinking current to the most recent deployment) if there's a bug in the code that slipped through, or if there's some undiscovered problem running it, it's trivial to rollback your code to a previous release. If that needs to happen, as we'll see later, `current` is updated to symlink to a previous release. A task which takes

Figure 3.

```
| Check (Check if the server is ready to receive the application) [~0.78s]
|-- Check/Php (Checks if the server is ready to receive a PHP application)
|=> Checking presence of /usr/bin/git
$ /usr/bin/git --version
[vagrant@drupal8.dev:2222] (production) git version 1.7.10.4
|=> Checking presence of Composer
|=> Checking PHP version
|=> Checking presence of required extensions
|=> Checking presence of required drivers
|=> Your server is ready to deploy
Execution time: 0.7751s
Saved logs to [redacted]rocketeer/.rocketeer/logs/production--20170328.log
```

Figure 4.

```
| Deploy (Deploys the website)
|-- Primer (Run local checks to ensure deploy can proceed)
|-- CreateRelease (Creates a new release on the server)
|---- Deploy/Clone (Clones a fresh instance of the repository by SCM)
|===> Cloning repository in "/var/www/d8sample/releases/20170328164622"
[vagrant@drupal8.dev:2222] (production) Cloning into '/var/www/d8sample/releases/20170328164622'...
|===> Initializing submodules if any
$ cd /var/www/d8sample/releases/20170328164622
$ /usr/bin/git submodule update --init --recursive
|-- Dependencies (Installs or update the dependencies on server)
|---- Dependencies/Polyglot (Runs all of the above package managers if necessary)
|------ Dependencies/Composer (Installs dependencies with Composer)
$ cd /var/www/d8sample/releases/20170328164622
$ /usr/local/bin/composer install --no-interaction --no-dev --prefer-dist
[vagrant@drupal8.dev:2222] (production) Loading composer repositories with package information
[vagrant@drupal8.dev:2222] (production)
[vagrant@drupal8.dev:2222] (production) Updating dependencies
[vagrant@drupal8.dev:2222] (production)
[vagrant@drupal8.dev:2222] (production) Package operations: 2 installs, 0 updates, 0 removals
[vagrant@drupal8.dev:2222] (production)
[vagrant@drupal8.dev:2222] (production) - Installing paragonie/random_compat (v2.0.10):
[vagrant@drupal8.dev:2222] (production) Loading from cache
[vagrant@drupal8.dev:2222] (production) - Installing ramsey/uuid (3.6.1): Loading from cache
[vagrant@drupal8.dev:2222] (production)
[vagrant@drupal8.dev:2222] (production) Writing lock file
[vagrant@drupal8.dev:2222] (production)
[vagrant@drupal8.dev:2222] (production) Generating autoload files
[vagrant@drupal8.dev:2222] (production)
|=> Sharing file /var/www/d8sample/releases/20170328164622/storage/logs
|=> Sharing file /var/www/d8sample/releases/20170328164622/storage/sessions
$ ln -s /var/www/d8sample/releases/20170328164622 /var/www/d8sample/current-temp
$ mv -Tf /var/www/d8sample/current-temp /var/www/d8sample/current
|=> Successfully deployed release 20170328164622
| Cleanup (Clean up old releases from the server)
|=> No releases to prune from the server
Execution time: 7.1695s
Saved logs to [redacted]rocketeer/.rocketeer/logs/production--20170328.log
```

4 *Core folders:* http://phpa.me/rocketeer-core-folders

5 *a shallow clone:* https://www.perforce.com/node/9471

barely any time.

The shared directory is something that I wasn't familiar with before I worked with Rocketeer. What it does is to contain the files required to persist across releases, such as user data files. The Rocketeer documentation[6] explains the concept quite well:

> *And finally the third folder, shared, is where files that are shared between each release are stored. Take, for example, our Facebook application, it has users that can upload their avatars on it, and they are stored in* `public/users/avatars`*. This is all fine until you decide to deploy again and Rocketeer creates a new release pristine folder from scratch where your uploaded images won't be. To solve this problem, in the config file you have a shared array where you can put paths relative to the root folder of your application, like* `public/users/avatars`*.*

6 *The Rocketeer documentation:* http://phpa.me/whats-rocketeer

Figure 5.

```
|=> The current release is 20170328164622 (1cd5dfad7edd60ab404cf7209a8be539b150bc32
deployed at 2017-03-28 16:46:22)
Here are the available releases :
+---+----------------+---------------------+--------+
| # | Path           | Deployed at         | Status |
+---+----------------+---------------------+--------+
| 0 | 20170328164622 | 2017-03-28 16:46:22 | ✓      |
| 1 | 20170328164518 | 2017-03-28 16:45:18 | ✓      |
| 2 | 20170328164430 | 2017-03-28 16:44:30 | ✓      |
| 3 | 20170328164338 | 2017-03-28 16:43:38 | ✓      |
| 4 | 20170328164039 | 2017-03-28 16:40:39 | ✘      |
+---+----------------+---------------------+--------+
Execution time: 0.5413s
Saved logs to [redacted]rocketeer/.rocketeer/logs/production--20170328.log
```

Validating a Release

At this stage, the next thing to do is to validate the release—especially as it's the first one. There are several ways to do this. Assuming you set up a virtual host on your web server for the code before running the deployment, opening your site in your browser will give you a superficial validation that everything worked.

But, more quantitatively, run `rocketeer current`. This will display output similar to Figure 5. You can see I've deployed three releases, the path to them, and the deployment date and time. Note: `0` is the latest release.

Deploying to Several Environments

Until this point, we've only discussed deploying to a single environment. But this is hardly a typical situation. Often we work with multiple environments, such as testing and staging.

Rocketeer supports deploying to

Listing 2.

```
'connections'      => [
    'testing' => [
        'host'      => 'myhost.com',
        'username'  => 'myuser',
        'password'  => '',
        'key'       => '/home/myuser/.ssh/id_rsa',
        'keyphrase' => '',
        'agent'     => '',
        'db_role'   => true,
    ],
],
```

multiple environments. To show how you can do so, let's setup, manually, a "testing" environment and look at how to deploy to it, instead of production. The first thing to do is to configure a connection to the server, in the `connections` array, in `.rocketeer/config.php` (see Listing 2).

You can see it's just like the production configuration but contains the hostname, credentials, SSH key, etc., for the new host. Then, you can optionally include it in the `default` array. By doing so, if you don't specify a stage when you run `deploy`, it will then be included in the deployment as well.

You also need to define the stages in `stages.php`:

```
'stages'  => ['testing', 'production'],
```

There's one final piece of the puzzle, which you can optionally configure. This is for configuration fine-tuning. In this case, each stage uses a different branch in our Git repository. In `config.php` update the `on` key as in Listing 3.

If you were to add this configuration when you deployed to testing, it would, as before, clone from the `testing` branch. But, when you deployed to production, it would clone from master instead.

To deploy to testing use the `--stage` flag:

```
rocketeer deploy --stage="testing"
```

Rolling Back Deployments

Now, let's say that you've been using Rocketeer for a while, and after the latest release something went wrong, and you want to rollback. There are several ways to do that. I say that because I'm trying to cover a range of functionality in a practical way.

The command you need to use is:

```
rocketeer rollback
```

To this command, you need to specify the release to rollback to, which you can find by calling:

```
rocketeer current
```

Listing 3.

```
    'on'              => [
        // Connections configuration
        'stages' => [
            // Stages configurations
            'testing' => [
                'scm' => ['branch' => 'service'],
            ],
            'production' => [
                'scm' => ['branch' => 'master'],
            ],
        ],
    ],
```

Let's assume that I want to rollback to `20170308112514`, then I'd call:

```
rocketeer rollback 20170308112514
```

In effect it moves the symlink from the latest release directory to `20170308112514`. Now, you may not want to run the rollback, only to test it; in effect a dry run. To do that, pass the `--pretend` switch to the command. Also, you may have multiple stages, instead of just one, as in our example. To do that, pass the name of the stage with the `--stage` switch.

In Conclusion

And that's a rapid run-through of Rocketeer, one of the best deployment tools for PHP. With any library as feature-rich and long-lived as Rocketeer, there's much more functionality than I have space to cover in one column. I'd love to go on at length, but am not able to.

However, if I were to critique it in any way, I'd say it is a little too strongly tied to Laravel. I appreciate that I've not talked about that until now. But, it's something I noticed during my experience with it. It doesn't obstruct you from deploying successfully. But, because of some default Laravel-specific configuration settings, when you first get started, warnings can be thrown.

That said, it's not of any concern. Otherwise, I strongly encourage you to install it, look at the options, read through the official documentation, and test it out on a project of yours. I've genuinely found it a breeze to use and wonder why I didn't use it sooner.

Matthew Setter is an independent software developer, specializing in creating test-driven applications, and a technical writer http://www.matthewsetter.com/about/, focused on security and continuous development. When he's not coding or writing, he loves spending time with his family, cooking, and hiking (oh, and reading a bit too). @settermjd

Look Out for That Bus!

Cal Evans

I'm going to assume you are a programmer. Either that or your dentist has weird taste in magazines for their lobby.

Since you are actively reading a magazine about programming, I'll also assume you understand why training is essential. So if both of those assumptions are true, you probably don't need to read this article. You can, however, rip it out of the magazine (or print it out from the PDF) and slip it onto your boss's desk as they might not yet understand why training is not just important, it is vital to the continued success of your team.

That's right; training isn't important for the individual developer, training is important for the team. Learning as a team helps foster esprit de corps. It builds morale. Learning as a team helps your junior developers because your senior developers can work with them to help understand and apply the concepts you are learning. Here's the big one, though. If you are investing in training, you train your entire team so when someone leaves, others already know what they know. Yep, that's right; training as a team limits your company's "bus factor."

Nobody wants to invest in a developer, train them up, and have them walk out the door. Most managers, however, miss the real cost of losing a developer. The cost is not the training you have invested in for that person. The real cost of losing a developer is the knowledge they have that is walking out the door.

If you have a developer making $50,000 and never gets a raise for ten years, then decides to leave, you have a half a million dollar investment walking out your door. Would you let someone walk out your front door with a half a million dollars worth of hardware? Of course not. But that is what's happening if you only invest in a few of your developers and they decide to leave.

Yes, this entire post is self-serving. Those of you who know me, know all of the developer training endeavors I am involved with, including the one you are reading right now. Helping developers learn is a passion of mine.

As a manager, however, it has to be more than a passion for you. Helping your developers learn has to be your mantra. It has to be a goal you set. It has to be your highest ranking KPI. If you don't invest in your developers, the chances are good their next manager will.

Don't just invest in your senior developers. Chances are good they are learning on their own. Invest in your entire team. Find ways to help them learn together, share knowledge, and grow together. Don't let the bus factor kill your project.

Past Events

March

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

Midwest PHP 2017
March 17–18, Bloomington, Minnesota,
https://2017.midwestphp.org

WordCamp London
March 17–19, London, U.K.
https://2017.london.wordcamp.org

WordCamp Miami
March 24–26, Miami, Florida
https://2017.miami.wordcamp.org

PHP Experience 2017
March 27–28, Sao Paulo, Brazil
https://phpexperience2017en.imasters.com.br

SymfonyLive Paris 2017
March 31–31, Paris, France
http://paris2017.live.symfony.com

Upcoming Events

April

PHP Yorkshire
April 8, York, U.K.
https://www.phpyorkshire.co.uk

Lone Star PHP 2017
April 20–22, Addison, TX
http://lonestarphp.com

DrupalCon Baltimore
April 24–28, Baltimore, MD
https://events.drupal.org/baltimore2017

May

PHP Unicorn Conference (Online)
May 4
http://www.phpunicorn.com

phpDay 2017
May 12–13, Verona, Italy
http://2017.phpday.it

PHPKonf Istanbul
May 20, Istanbul, Turkey
http://phpkonf.org

PHP Tour 2017 Nantes
May 18–19, Nantes, France
http://event.afup.org

php[tek] 2017
May 24–26, Atlanta, Georgia
https://tek.phparch.com

PHPSerbia Conference 2017
May 27–28, Belgrade, Serbia
http://conf2017.phpsrbija.rs

International PHP Conference 2017
May 29–June 2, Berlin, Germany
https://phpconference.com

June

CakeFest 2017
June 9–10, New York, USA
https://cakefest.org

PHP South Coast 2017
June 9–10, Portsmouth, UK
https://2017.phpsouthcoast.co.uk

Dutch PHP Conference
June 29–July 1, Amsterdam, The Netherlands
https://www.phpconference.nl

September

Pacific Northwest PHP Conference
September 7–9, Seattle, Washington
http://pnwphp.com

PHP South Africa
September 27–29, Cape Town, South Africa
http://phpsouthafrica.com

October

ZendCon 2017
October 23–26, Las Vegas, Nevada
http://zendcon.com

November

php[world] 2017
November 15–16, Washington, D.C.
https://world.phparch.com

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.

Evaluating Value Objects

David Stockton

Humans are pretty good at looking at things and telling what they are, in general. If I were to ask you what "red" was, you'd likely tell me it's a color and perhaps list a few things that are red. If I were to ask what 8675309 was, you might say it's a phone number, or perhaps it's the number of animals adopted each year, or if you're more of a math geek, you could tell me it is the 582161st prime number. (It's true, look it up). However, context can change what each of these mean and context is very important.

If you're listening to Tommy Tutone, then 867-5309 is definitely Jenny's number. And red might be a color, but what if I was talking about "That 70's Show?" Now it's just as likely that it refers to "Red," Eric's father. Or perhaps it's a reference to DEFCON levels, and we've got some serious things to think about. As humans, we can gather context from the things that happen around us. If we're standing in front of the WOPR computer, red is more likely to refer DEFCON levels, for instance. Computers don't have that luxury. They rely on the context in-so-far as we give it to them. Sometimes this context is just the variable type. And because PHP is built for the web and the web is built on strings, it happily converts from one type of variable to another. Often we get the right answer, but other times we get surprising and even onerous results.

In PHP, we can add the two strings "2" and "3" and end up with the integer 5. PHP automatically saw that we were trying to add these two strings together and converted them to numbers and added them. Sometimes, though, we don't get what we may expect. Those of us who have been working with PHP for awhile will not be surprised to know that `204 + "1600 Pennsylvania Ave"` is 1804. In fact, if you run this in PHP, you'll get an integer of 1804. Depending on your version, though, you may get just that result, or if you're running PHP 7.1, you'll get a notice that "1600 Pennsylvania Ave" is not a well formed numeric value.

If we look at what the values are, or represent, and try to do the same operation, things start to look wonky. First of all, we have `204` which is a number, specifically an integer. What it represents, we have no idea. We also have `"1600 Pennsylvania Ave"` which presumably represents an address. If we were to add `int` and `int`, we can expect an integer as the result. This seems reasonable and logical. However, what is expected if we added `int` and `Address`? I have no idea what we'd expect the system to do. We could say it should add the int value to the street address part of the address. Or just as reasonable, expect that the int value represents the number of blocks north to move with the expected value being the new address. Or any other of a number of potential "expected" results, none of which makes any more sense than another and is highly dependent on the application context. As a computer or as a programmer, when faced with the task of trying to add an integer to an address, we could immediately indicate there was a problem because we're being asked to do something that makes no sense.

Not All === Values Are Created Equal

PHP has a number of built-in types. There's the commonly known `int`, `float`, `string`, `boolean`, and array. There's also `null`, `resource`, `object`, and the `callable` types. However, those don't tell us the context or meaning behind a value. When you run through your code, there's a decent chance you'll run into various integers or strings right in the code. You may have a loop that counts up to `4` or a section of code the uses "GET" to retrieve something. There may be another `4` somewhere else or another "GET" elsewhere as well. These sneaky bits of data are known as "magic constants." They are hard coded values needed to perform some kind of functionality. But their meaning is what is missing. Once of the "4"s might be looping over a quadrilateral and determining its perimeter. The other may be used to build a table with four columns. Clearly, replacing these with the same variable, while it may work at first, is not the right path.

Some magic constants can be replaced with actual constants. On a project I work on, there's a database table which contains the status of some virtual crops the user works to cultivate. If it has a status of `0` then it's considered active, alive and growing. If it has a status of `1` it has been sold and the record is kept for historical tracking reasons. If the status is `2`, then it has exploded. Rather than have a codebase littered with `0`, `1` and `2`, it helps provide context by using constants like `STATUS_ACTIVE`, `STATUS_SOLD`, or `STATUS_EXPLODED`. It helps the developer understand what the intent was rather than needing to remember what the values are and what the previous developer was trying to do.

But we can go even further in most cases. That's where *value objects* come in, which we'll be discussing for the remainder of this article.

Unlimited Types

PHP may only come with eight built-in types, but we have the ability to create as many types as we want. To me, it makes sense to create these types even if we would normally represent the value by a single primitive type. In that case, we will be wrapping a primitive value in an object. The object carries with it the context—the reason for the primitive value to exist. For instance, suppose we need to deal with phone numbers. It's tempting, fairly easy, and extremely common that we represent a phone number by a string. This gets passed around, stored in databases, represented on the screen, and mangled in various ways. But to avoid bugs, we need to add ever-increasing checks to ensure everything is right and happy in phone number land.

In the U.S., nearly everywhere uses 10-digit dialing. We have a 3-digit area code, followed by a 3-digit exchange and a 4-digit number. These ten digits, in order, make up what many of us see as a phone number. In some cases, the phone number may need to be formatted. It could be done in a number of ways: `(720)555-1234`, `720-555-1234`, `720.555.1234`, or even `+1 720 555 1234`, or any number of other formats. All of them represent a phone number, and if we're expecting users to provide a phone number, we should allow them to enter any of these, and more, and deal with it appropriately. Notice, in the last example, the user provided the country code which means we've got 11 digits provided. Depending on your needs, you may also need to allow a user to input an "alpha" phone number, one which is commonly used in advertising for memorability—"318-4-PHP-411" for instance.

If the methods and functions we build that need a phone number accept and deal with strings, they either need to be able to deal with all the variations along with all the possible invalid variations and validations. This could be needed throughout your application. Even if you build functions for doing all this, there would still likely be repeated blocks of logic to determine if the number is international, if it's correct, etc.

Instead, we could create a `PhoneNumber` class. Whenever we need a phone number, we can pass the wrapped string around in a PhoneNumber object, and we would know we are expecting a phone number. The `PhoneNumber` class, in turn, would be able to ensure we have a valid phone number, and could also give back various primitive representations of the number—formatted, no punctuation, internationalized with country code, original formatting, etc. This would allow code requiring a `PhoneNumber` to type hint on that class, or better yet, type hint on an interface the `PhoneNumber` class implements. (This would allow for different implementations of classes which can give back phone number information, even if that class is not solely representing a phone number.) Then in that code, the developer understands that they're not dealing with a string that looks like a phone number, but an actual phone number. The fact that, internally, it's a string doesn't matter. In fact, internally, it could be represented by multiple fields—country code, area code, exchange, number, etc. The class doesn't need to expose how it's dealing with the data internally.

Wrapping Primitives and More

There are plenty of values we use in code which can be represented by a single primitive value. There are also plenty where there are multiple values. Too often in PHP, we rely on the ever-present array to give us flexibility. Plenty of code ends up expanding to multiple parameters and loads of defaulted values until someone decides it's too many. Then they change the signature to accept an array of key => value parameters. The code may have even taken that into consideration and left some required parameters as standard parameters and then put all the optional values into a `$params` array. Problem solved, right? Not so much.

Value Objects

A Value Object is an immutable type distinguishable only by the values of its properties. Value objects with identical properties are considered equal.

Now we don't know if we have a parameter or not. There's no such thing as a required key in an array.

Inside the code, we either need to decide we're fine with PHP throwing notices for accessing keys which don't exist, or we need to wrap everything in `isset()` before using it, or with PHP 7+, we can use the awesome Null Coalescing Operator. Go back and read that again, but imagine your voice is resonating and echoing and sounds amazingly cool. Ok, here it is again: the Null Coalescing Operator! Fun, right? The Null Coalescing Operator is an excellent way to quickly determine if a value is set in an array (or another variable) and provide defaults, all without PHP complaining that you're accessing a thing that doesn't exist. So back from that aside, we either need to decide we're ok with PHP emitting notices or wrap with `isset()` or use `??`. And we should **not** be okay with notices. Even if they are disabled, PHP still does all the work to figure out you're accessing stuff that doesn't exist, plus making the message and then it throws it all way. Code which emits notices (or warnings) is going to be slower than cleanly coded PHP, even if you don't see the notices.

We also should not be okay with calling `isset()` to determine if something was set in an array used as a parameters bucket or a poor substitute for an object. We shouldn't even be happy with using the Null Coalescing Operator (echo echo) as a substitute for knowing what we're dealing with. Instead, by creating value objects, we can be certain they have specific things we need and that they are valid.

Examples

Suppose we need to deal with email addresses. Instead of using a string to

Listing 1.

```
1.<?php
2.
3. class EmailAddress
4. {
5.    private $address;
6.
7.    public function __construct($email) {
8.       $this->address = $email;
9.    }
10.
11.   public function getEmailAddress() {
12.      return $this->address;
13.   }
14. }
```

represent an email address, we can build a value object as in Listing 1.

It's simple and straightforward, but it gives us a start. When we need to work with an email address, we don't have to rely on a developer seeing we named our parameter `$email`, or have a Docblock specifying the type. Now we can type hint `EmailAddress`.

```
public function passwordReset(EmailAddress $email,
                              string $content) {
   // Send password reset email
}
```

Now there's no question what's expected. We don't even need to be concerned when calling it, what is going to happen with it. If the `EmailAddress` value object provided a way to extract and return the domain, we're still covered. In fact, we could take the example above another step further. Instead of relying on a string for $content, we could make an `EmailContent` class. This could potentially encapsulate both a text and an HTML version of the email we want to send.

As a caller, I don't need to be concerned the password reset function will only send a text-based password reset email. I just know I need to provide an `EmailContent` object with the content of the email. Later, if we decide we want to send a mixed message with HTML and text, I would only need to update the passwordReset method to extract both HMTL and text or even more likely, the passwordReset method would be passing this off to something else that's responsible for actually assembling the email.

Value Objects Should Be Immutable

We talked last month about immutability. That is, the objects internal state should not be allowed to change. This is because the way we should be able to compare value objects isn't by seeing if they are the same object, but rather if the state is the same. Imagine two email address value objects:

```
$email1 = new EmailAddress('phparch@example.com');
$email2 = new EmailAddress('phparch@example.com');

var_dump($email1 == $email2); // true
var_dump($email1 === $email2); // false
```

In short, it's the values in a value object that make the value objects comparable, not the actual object itself. I've given the example of using `DateTime` versus `DateTimeImmutable` in a number of previous columns, so I'll spare you in this one. If you recall, though, we want to compare date objects based on the date and time they contain, not on whether they are literally the same object. Using the immutable version of DateTime means we can avoid a lot of different and hard-to-track errors cause by inadvertently changing the state of an object when we really wanted to create a new object based on an existing one.

How Are Value Objects Different From Entities

You may be wondering how a value object is different from an entity. The identity of an entity object is because of some sort of unique identifier. All the other states of those objects can change, but it would still be considered the same entity. For a value object, if the state changes, we now have something else entirely. Conversely, two value objects with the same state ought to be considered equal. Consider the phone number and email address examples. If the value of the string representing either one were to change, we now have what we'd consider a new value object. The phone number `318-474-7411` is certainly not the same as `720-555-1234` even if we happened to use a mutable object and change the state. However, if you consider an entity object representing a person, even if we change the person's name or phone number or email address, they are still the same person. Their identity is not held within the state of the entity object.

This is another point towards the importance of immutability of the value object. Our entities have a history. There was a path that was taken in order to get them to the current state. We may not store that history in the object or in the database, but there was, somehow, a history nevertheless. For a person, their history may include the assignment of a name shortly after birth. Later, other state changes happen as they grow, age, and make other life choices. All of these changes are part of the history of the object representing that person. However, something like a date, a value object, has no history. It simply *is*. Changing its state means we're dealing with something fundamentally different than what we started with.

Consider how money could be treated. If we modeled physical dollars as objects, including state, we'd potentially include information about the serial number. But we want to represent money as something fungible where any dollar is equivalent to any other. If we changed the state, we're dealing with something else. If the state of a dollar included its value, and we were allowed to change it, we're now dealing

with counterfeiting charges as well as something else entirely.

Considerations and Conclusion

Wrapping primitives in an object seems, at first, like a lot of unnecessary overhead. Indeed, there is some overhead, but not much. For the help the developer receives from their IDE and from the self-documenting of the code we get from type hinting on an object, to the potential errors that can be caught both automatically and through code inspection, I surmise this trade-off is well worth it. The value objects are cheap to create and destroy. It is simple to know what you're doing is correct according to the code you're calling. If you provide a well-formed value object to a method that type hints that it needs that kind of object, you should be well assured it's going to work appropriately. Compared to passing in arrays of data, it is superior. The code you're calling needn't be concerned about whether an optional item was provided, or what to do if it was or was not provided. It simply uses the object. The object can encapsulate logic about what it means if optional values are provided. Additionally, the object can include methods that help with extracting relevant bits of information from the state that would otherwise require global functions or loads of copy-pasted code to deal with throughout a site.

Consider wrapping your primitive types in an object that represents the intent and the context of what that value means. It will help increase understanding of the code while simultaneously reducing errors. Value objects provide an excellent way of shuffling data around along with their context, giving simple types a more coherent and stronger meaning, which allows the programmer to concentrate on getting the logic of the program correct instead of fighting through low-level data manipulations and scattering of business logic and code intent all over the codebase. It may seem weird, but it will be worth it. See you next time.

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.

Taint Detection in PHP

Chris Cornutt

It's no secret the lack of input validation and filtering is a problem in web applications. In fact, the omission of these two controls is usually what leads to the most common vulnerabilities[1] found on the web today.

When it comes to PHP, things get even worse. The language provides direct access to superglobals (like `$_GET` and `$_POST`) allowing them to be used directly. While this may seem attractive to developers, especially those just starting out, it can be detrimental in the long run and introduce numerous security issues. This month, we'll look at how to detect *tainted* data in PHP.

The Danger of Tainted Data

It's a common mantra in just about any security talk, and with any security professional, you should "never trust the user." This especially applies to the data they provide to you. All incoming data should be considered "tainted" and not be trusted without any kind of validation or filtering (preferably both).

When most people think of the use of tainted (untrustworthy) data they think of attacks like Cross-Site Scripting[2] —abbreviated as XSS—where user input is echoed back out directly into the page. While this is one of the more major issues, more globally called "injection," it's not the only kind of problem that could be caused by tainted input. Some other examples include:

- No enforcement of password rules resulting in poor passwords
- Invalid account data used in later operations causing application breaks.
- No checks against other values in the system to ensure the operation is valid.

You can see trusting user input without validation can cause more problems than just someone injecting content or code into your application. There's a whole host of other logic-related issues which come into play as well. If you make the assumption any data coming into your application can be trusted, you're setting yourself up for major issues down the road. Large companies even have this issue with hacks reported every week where someone was able to use some kind of injection to gather information or gain access to systems.

As a side note here, I want to remind you any HTTP request (even HTTPS ones) basically boil down to a piece of software sending a specially formatted text message to a waiting server. There are no formal requirements (unless you implement them) that those requests come from your application. It's trivial for an attacker to look at the `POST` location of your forms and script a few lines of code to try to abuse it.

The "Taint" PHP Extension (PECL)

A few years back a developer named Xinchen Hui, @laruence[3] submitted an RFC[4] proposing taint analysis be included as a part of the core language. This proposal's goal was to make it easier for developers to find and fix the direct use of tainted data in their applications. Here's how the RFC describes it:

> *This is a preliminary implementation of support for tainted variables in PHP. The goal is to help PHP application programmers find and eliminate opportunities for HTML script injection, SQL or shell code injection, or PHP control hijacking, before other people can exploit them. The implementation provides taint support for basic operators and for a selection of built-functions and extensions. A list of what is implemented so far is at the end of this document.*

The RFC defined an extensive set of features to be implemented, but only some of these made it into the current version of the extension. The extension is still getting a few updates here and there, but they are mostly patches for issues other users have found. There's a long list of functions which it performs the "is tainted check" on including:

- `echo`
- `exit`
- `include`
- `eval`
- the use of concatenation

When it detects the use of a potentially tainted value it will give you a warning notifying you of the use. The message will provide a bit of context

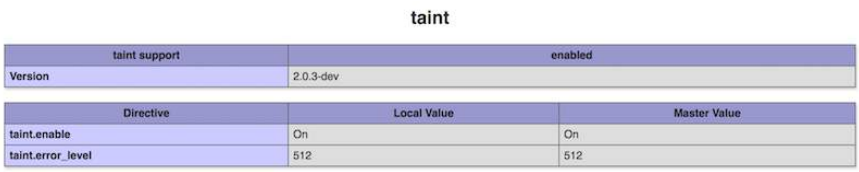
1 *the most common vulnerabilities:* http://phpa.me/owasp-top10v

2 *Cross-Site Scripting:* http://phpa.me/owasp-xss

3 *Xinchen Hui, @laruence:* https://github.com/laruence

4 *an RFC:* https://wiki.php.net/rfc/taint

Figure 1. Taint in phpinfo



taint

taint support	enabled
Version	2.0.3-dev

Directive	Local Value	Master Value
taint.enable	On	On
taint.error_level	512	512

around what operation is being performed but don't expect a full stack trace. Here's an example of using a `$_GET` variable directly on an `echo`:

```
PHP Warning:  main() [echo]: Attempt to
echo a string that might be tainted in /
private/var/www/test/test.php on line 8
```

NOTE: By default, the taint extension uses the `E_WARNING` *error level, but this can be changed through a setting on the extension either in* `php.ini` *or with an* `ini_set`[5].

Installation

The taint extension[6] installs just like any other PHP extension. If you're unfamiliar with extensions in PHP, they're basically modules of functionality that plug into the language. They extend the language with extra features which aren't included in the core of the language. In the past, several of them have worked their way into the core language, but there are still plenty that haven't. The taint extension is one example of functionality that tried to get into core (via the RFC) but, after discussion, was left as a PECL module. Since Xinchen[7] created it as more of a proof of concept than anything it was never pushed much beyond its current state.

You can either install the module with the `pecl` command line tool included in most PHP installations (or you can easily install it via the package manager of your choice), or it can be compiled manually by cloning the GitHub repository[8]. Depending on your distribution, you may need to compile it yourself to use this extension with PHP 7. If compiling the extension, you'll also need the PHP source, which is usually available as a "php-dev" package. An example of the `pecl` version:

```
sudo pecl install taint
```

Simple, right? Once that completes it will tell you where it installed the `taint.so` file. Unfortunately, that doesn't always work, and sometimes you need to compile it yourself. Thankfully, the `phpize` command makes that just a few extra steps:

```
git clone git@github.com:laruence/taint.
git
cd taint
phpize
./configure
make && make install
```

This will also report back where the `taint.so` file was placed. Record this path as you'll need it when updating your `php.ini` file. Obviously, the next step is to edit your `php.ini` file and add in the configuration to pull in the taint module. You can add a section like this:

```
[taint]
extension=/path/to/taint.so
taint.enable=1
```

You'll replace the `/path/to/taint.so` with whatever the command above gave you. If you're using it in a command line context, you're ready to go. If you're using a web server, you'll need to restart it to ensure the latest `php.ini` configuration is loaded correctly. If all goes well you should see something like this in your `phpinfo()` output:

Usage

Once you have the extension installed and showing up in the `phpinfo` output (or `php -m` for the command line) you're ready to try it out. If you already have some code handy you could just run it and see what the extension catches. If you want to try it out on a basic level, though, you can put this code in a file called `test.php` with a query string like `test.php?test=something`:

```
<?php
echo $_GET['test'];
```

Once you save that go to it in your browser, and you should see an error message—if not use `ini_set('display_errors', 1)`—like the one previously mentioned:

```
PHP Warning:  main() [echo]: Attempt to
echo a string that
might be tainted in /private/var/www/
test/test.php on line 1
```

The error message output will give you a bit of information about the failure, but it may require a bit of poking around in the code to find the line actually affected. You'll also notice the type of error it defaults to is `E_WARNING`. I mentioned earlier you could change this. All it takes is a basic ini_set[9] or an update to your `php.ini` configuration to set the `taint.error_level` setting to whatever other PHP error level you might want (`E_ERROR`, `E_NOTICE`, etc.):

```
ini_set('taint.error_level', E_ERROR);
```

If there happens to be one place in a script you don't want the extension to fire off errors, you can also switch it off using the `taint.enable` setting:

```
ini_set('taint.enable', false);
```

What About Frameworks?

Unfortunately, the `taint` extension can only take you so far. Because it was implemented as more of a proof of concept for this RFC[10] (now in "inactive" state), it's not quite developed enough to understand the structure most frameworks use. In the previous

5 `ini_set`: http://php.net/ini_set

6 *taint extension:* http://pecl.php.net/package/taint

7 *Xinchen:* https://github.com/laruence

8 *GitHub repository:* https://github.com/laruence/taint

9 *ini_set:* http://php.net/ini_set

10 *this RFC:* https://wiki.php.net/rfc/taint

examples, you saw how it could detect when a variable was used directly from a superglobal and spit back an error. Frameworks, however, are much more complex than this and usually have a way for you to get to the superglobal information without having to reference `$_GET`, `$_POST`, etc. directly.

So, if this extension won't help in that situation, what can you do? Fortunately most frameworks will only have one or two ways to get at the values from the superglobals, usually though a method in the controller or some other helper method. For example:

Framework	Method(s)
Slim	`$request->getParam();`
Zend Framework v2	`$this->getRequest()` `  ->getQuery($name, $default)`
Symfony 2	`$request->request->get(...),` `$request->cookies->get(...)`

Laravel	`$request->input(...),`
	`$request->cookie(...),`
	`$request->file(...)`

Each of these examples gives you a good starting place to work from. It makes it easier to just search through your code (grep is your friend) and locate the use of these methods. Once you find them, you can then do some manual evaluation of the use of the data they return.

Finally

While the taint extension is a handy tool that can locate some glaring errors in your current codebase, it's not going to be the silver bullet that will make your code secure. Writing secure code and keeping it secure is something that should happen as a part of any normal development cycle, not as an after-thought.

By putting an emphasis on security early in your development process, you not only reduce the number of security issues in the final code but you also don't have major issues to fix when the code is already out in the wild. Tools like this can help catch some of the easy to fix errors, but ultimately it's up to the developers to do things right.

Resources

- Taint in the PHP manual[11]
- php.internals discussion of taint[12]

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

Easy Vagrant Environments as a Service

Joe Ferguson

Vagrant has been a stable and easy way to configure virtual development environments. One common problem is not every developer is a Linux expert, nor do they want to be. Homestead is the official Vagrant box for the Laravel framework ecosystem and is built from the ground up to be easy to use and customize without having to be a server expert. We'll cover how to get started with Vagrant, how to customize Homestead for your project, and how to share your new Homestead environment with other project collaborators easily.

Artisanal What?

Laravel describes itself as "the PHP framework for web artisans." This sometimes casts an odd disposition to a newcomer who may not consider themselves an artisan. By definition, an artisan is a skilled worker. If you're brand new to PHP or if you're a grizzled veteran, you are a skilled worker. We sometimes forget how much skill it takes just to read code and interpret it. Effectively, we're human linters and compilers understanding what the computer will do.

Every PHP developer I've ever met is an artisan at some level. Each one has their specialty, what drives them, and what makes them such an amazing fit for the role of a developer. The hunger to hack that piece of code and bend it to your will or the drive to understand how a technology works are something all developers share and transcends programming languages and culture. When I see the word artisan, I read it as someone who cares deeply about their craft, their code, and their skills. Today I want you to become an artisan of your development environment.

But, Containers!

You're likely reading this and yelling, "But, Containers!" Containers are awesome, and I use them just about every day. However not every application is ready to be containerized. I spent many years in the trenches of legacy PHP projects, and I am comfortable saying Vagrant is in no danger of going away anytime soon due to Docker or any other container system.

Before We Get Started

Ensure you have installed up to date versions of Vagrant and VirtualBox. You need to make sure you have version 5.1 or greater of VirtualBox. Note that you cannot upgrade from 5.0 to 5.1 from within the application. You must visit the website and download the newer version and install.

At the time of writing Vagrant 1.9.2 is freshly released. You should always try to keep up with the newest versions as they often contain new features and bug fixes.

Vagrant Primer

If you've never worked with Vagrant; or if you are completely new to the idea of running a virtual machine on your computer, let's review some of the fundamentals. They have some operating system, typically a Linux distribution, and some other software already installed. A virtual machine is an emulation of another computer system, including hardware and OS. Vagrant is an application which interacts with a provider application to manage virtual machines, making it easier to create, update, and configure them. The provider application we are using in our examples is VirtualBox.

Vagrant uses specially packaged virtual machine snapshots called box files. These box files are a starting point for you to customize a virtual machine further. Box files are published on Atlas[1]. Think of Atlas as Packagist for Vagrant boxes. You can add a box to your computer as easily as you would require a Composer package:

```
vagrant box add laravel/homestead
```

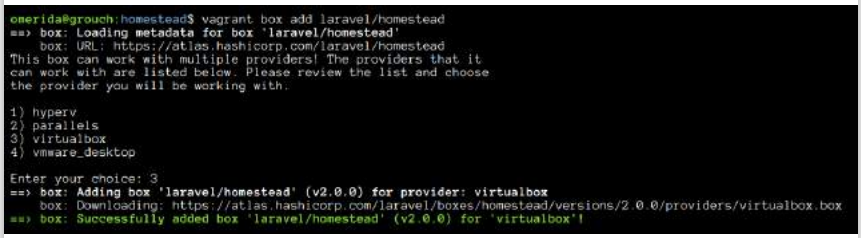
This command tells Vagrant to look up the Homestead box in the Laravel namespace and download the most recent version of the box built for the VirtualBox provider. These box files can take up around a gigabyte or more of space; the download may take a few minutes.

Because we are using VirtualBox, we are virtualizing the entire computer. All of the hardware of your host computer is virtualized by VirtualBox for the Vagrant virtual machine. Don't worry; you don't have to be a computer expert to use Vagrant. Vagrant takes care of nearly everything for us.

The two most important Vagrant commands are `vagrant up` which boots up the virtual machine and `vagrant halt` which shuts down the virtual machine. There are plenty of other commands, however, if you are brand new to Vagrant, but don't worry about those for now.

1 *Atlas:* http://atlas.hashicorp.com

Figure 1.



Homestead Parts

There are two parts to Homestead and since they are both named Homestead it can sometimes be confusing. First, there is a base box named *Homestead* which is the box file you add to your system via the `vagrant box add` command. You can find the Homestead box on Atlas—laravel/homestead[2]. We use a project called Settler[3] to build the Homestead box, so you don't have to wait on things like PHP and NGINX to be installed (or provisioned in Vagrant speak). This allows you to boot a Homestead virtual machine in about a minute, instead of fifteen.

The second part of Homestead is the Homestead application or repository. This is what you download and configure to suit your needs. You can find the Homestead application on Github—laravel/homestead[4]. The Homestead app is controlled by a `Homestead.yaml` configuration file you can use to customize Homestead to your system and your projects.

Getting Homestead

To install Homestead (the base box), open a terminal and run:

```
vagrant box add laravel/homestead
```

The download may take a few minutes depending on your internet connection speed.

To install Homestead (the application), clone the repository from GitHub[5] and check out the most recent release version.

```
git clone https://github.com/laravel/homestead.git
```

You can find the most recent release on the Homestead releases tab[6]. For example, if you want to use version 5.01.10 you would run the command below in the repository on your local machine.

```
git checkout v5.01.10
```

If you want to stay on the absolute bleeding edge, you can remain on the master branch but beware that many times changes in master require a new base box which has not yet been released. If you want to keep Homestead as stable as possible, always check out a release version.

Installing and Configuring Homestead

In a terminal window change directory to the Homestead repo folder. Run the command `bash init.sh` if you are on a Linux or MacOS operating system, run `./init.bat` if you are on Windows. The init scripts copy files into this current folder that you can modify to customize Homestead to your liking. You should see a "Homestead initialized!" message.

The main file you are editing is `Homestead.yaml`. You can see the default in Listing 1.

Lines two through five are provider options where we specify how many CPUs and how much RAM to assign to the virtual machine. Lines seven through ten tell Vagrant what SSH keys to use when connecting to the virtual machine. Line twelve is where we start mapping folders from your host

Listing 1.

```
1.---
2. ip: "192.168.10.10"
3. memory: 2048
4. cpus: 1
5. provider: virtualbox
6.
7. authorize: ~/.ssh/id_rsa.pub
8.
9. keys:
10.     - ~/.ssh/id_rsa
11.
12. folders:
13.     - map: ~/Code
14.       to: /home/vagrant/Code
15.
16. sites:
17.     - map: homestead.app
18.       to: /home/vagrant/Code/Laravel/public
19.
20. databases:
21.     - homestead
22.
23. # blackfire:
24. #     - id: foo
25. #       token: bar
26. #       client-id: foo
27. #       client-token: bar
28.
29. # ports:
30. #     - send: 50000
31. #       to: 5000
32. #     - send: 7777
33. #       to: 777
34. #       protocol: udp
```

2 *laravel/homestead:* http://phpa.me/boxes-homestead
3 *Settler:* https://github.com/laravel/settler
4 *laravel/homestead:* https://github.com/laravel/homestead
5 *GitHub:* https://github.com/laravel/homestead
6 *Homestead releases tab:* http://phpa.me/homestead-releases

operating system to the virtual machine. This is similar to a networked folder but uses VirtualBox's shared folder functionality under the hood. Line sixteen is where we start mapping sites to our mapped folders. This is how Homestead knows what virtual hosts it needs to create and which *mapped* path is the document root. Line twenty is where you can add extra databases to be created when the virtual machine is created.

> *When you're specifying a filesystem path in* `Homestead.yaml`*, make sure to omit the trailing slash.*

Homestead maps folders from your host operating system to the virtual machine because the virtual machine is disposable. Nothing should exist in the virtual machine that isn't created by running `vagrant up`. This allows us to completely destroy our development environment and still be able to bring it back in working order with simple commands without worrying about losing any unsaved work.

Starting, Stopping, and Breaking Homestead

Starting Homestead is as straightforward as running the command `vagrant up`. Vagrant starts by reading the `Vagrantfile` in your Homestead folder and creating a new virtual machine, booting up the machine, and provisioning the machine if needed. When you're done with the virtual machine or need to shut down your host computer, you can run `vagrant halt` to turn off the Vagrant machine. Halt is the equivalent of running `sudo shutdown -h now` from the command line. If you happen to lose track of where your Vagrant machines are you can check the global status of all the Vagrant virtual machines on your system by running `vagrant global-status`; this shows you all of the virtual machines Vagrant is aware of and their status. Sometimes, when dealing with multiple virtual machines, you can find yourself wondering where all of your system resources went; don't forget to halt any virtual machines you are no longer using.

> *I don't use* `vagrant suspend` *because it doesn't always work nicely in my experience. Plus it eats up some extra disk space to save the machine's current state for resuming later.*

Provisioning is a term describing actions taken after the box has booted but before the virtual machine is ready for use. Typically this is when extra software packages are installed or configured. Homestead uses this provisioning time to configure the sites you have mapped in your `Homestead.yaml`. Vagrant will only provision the machine on the first boot. You can tell Vagrant to provision the machine again by running `vagrant --provision`. Provisioning a virtual machine multiple times could cause some issues depending on what you have added onto Homestead. Have no fear, if anything breaks or stops working as you expect, run `vagrant destroy` to destroy this virtual machine. Everything in the machine will be deleted, but we won't lose any data because we have used mapped folders for our application. All of your code is safe on your host operating system.

The first time you bring up your homestead box, you should see output similar to Figure 2.

Using Homestead

Once the `vagrant up` command has completed, you should be able to visit http://localhost:8000 and see the applicatoin you configured earlier. You can SSH to the virtual machine by using the `vagrant ssh` command. Once inside the Vagrant machine, you can run your migrations, database seeders, or any other extra steps you may need to perform to configure your application for use in the development environment.

By default, Homestead forwards a number of ports to localhost so you can easily connect to services. Port 8000 is forwarded to port 80 for HTTP traffic, 33060 is forwarded to 3306 for MySQL/MariaDB. To connect to the database server with a GUI tool, use localhost for the host name, port 33060, MySQL username is homestead, and the password is secret.

> *While homestead grew out of the Laravel project, there's no reason you can't use it with to setup a development environment for other PHP frameworks or applications.*

You can easily add additional sites just by adding to your `Homestead.yaml` configuration file. However, only the first site is available via http://localhost:8000. To view the other sites, you'll need to

Figure 2.

```
==> homestead-7: Running provisioner: shell...
    homestead-7: Running: inline script
==> homestead-7: Running provisioner: shell...
    homestead-7: Running: /tmp/vagrant-shell20170330-28535-1uv28ds.sh
==> homestead-7: Running provisioner: shell...
    homestead-7: Running: script: Creating Site: homestead.app
==> homestead-7: Running provisioner: shell...
    homestead-7: Running: script: Restarting Nginx
==> homestead-7: Running provisioner: shell...
    homestead-7: Running: script: Creating MySQL Database: homestead
==> homestead-7: Running provisioner: shell...
    homestead-7: Running: script: Creating Postgres Database: homestead
==> homestead-7: Running provisioner: shell...
    homestead-7: Running: script: Clear Variables
==> homestead-7: Running provisioner: shell...
    homestead-7: Running: script: Update Composer
==> homestead-7: Updating to version 1.4.1 (stable channel).
==> homestead-7:
==> homestead-7:  Downloading: Connecting...
==> homestead-7:
==> homestead-7:  Downloading: 100%
==> homestead-7:
==> homestead-7:
==> homestead-7: Use composer self-update --rollback to return to version 1.3.2
==> homestead-7: Running provisioner: shell...
    homestead-7: Running: /tmp/vagrant-shell20170330-28535-x1c4nu.sh
```

update your host operating system's hosts file. On Linux and MacOS this file can be found at `/etc/hosts`; Windows users can find the file at `C:\Windows\System32\drivers\etc\hosts`. Note that you need to edit this file with sudo or administrator privileges. If you want to add a new project called "Demo App" with the URL http://demo.app you can add the following to your site's configuration:

```
- map: demo.app
  to: /home/vagrant/Code/DemoApp/public
```

Once complete, run the following command to configure the new site in Homestead.

```
vagrant reload --provision
```

There are Vagrant plugins available that will automatically add these sites to your hosts file; such as Vagrant Hostsupdater[7]

Customizing Homestead

For most users, this may be enough to send you off on your way, diving deep into your project. Other users may want to add some other dependency or software packages to Homestead. Homestead makes customizing the virtual machine easy by way of an additional provisioning script called `after.sh`. You should see `after.sh` in the same folder as your `Homestead.yaml` file. Anything you want to add to Homestead should be done in this file. The last provisioning action Homestead performs is running this `after.sh` file. If you want to install extra packages or configure additional services, this is the place to do those tasks, so they are preserved if you destroy the Vagrant machine. For most users, this may be enough to send you off on your way, diving deep into your project. Other users may want to add some other dependency or software packages to Homestead. Homestead makes customizing the virtual machine easy by way of an additional provisioning script called `after.sh`. You should see `after.sh` in the same folder as your `Homestead.yaml` file. Anything you want to add to Homestead should be done in this file. The last provisioning action Homestead performs is running this `after.sh` file. If you want to install extra packages or configure additional services, this is the place to do those tasks, so they are preserved if you destroy the Vagrant machine.

If you wanted to install Elasticsearch for use in your application you can add the commands to install and configure the service in the `after.sh` file so once the process of creating the Vagrant machine is complete you'll have Elasticsearch installed to your specifications.

An example `after.sh` can be found in Listing 2.

In this example, we import a database from an SQL file in our project folder and install the packages `tmux`, `screen`, and `vim`. If any of these packages are already installed, Apt will skip over them just as it normally would. An important note here is that you do not need to use sudo. Provisioners run as root by default.

If there is a command you use particularly often in your application, you can add it to the `aliases` file. This file is used to create shortcuts for commonly use commands. You can see examples of how to add your own aliases in the file. You will notice there are some common defaults such as using `art` instead of `php artisan`.

Both `after.sh` and `aliases` give you incredibly flexible ways to customize Homestead to your liking. These files are the preferred way to override Homestead defaults without having to modify core files which could cause painful Git conflicts later down the road when you try to upgrade to the newest versions.

Updating Homestead

We try to keep Homestead as stable as we can and follow Semantic Versioning so you can easily tell large, potentially backward-incompatible changes (major version number changes) from small changes (minor or patch version changes). With the two components of Homestead being the box and the application, it can sometimes be difficult to tell what the upgrade process should be. The Homestead application has an internal setting we can use to force specific minimum versions of the base box. Normally, you can keep upgrading to the stable version of the application and upgrade the base box as Vagrant informs you of a new release.

To update the Homestead base box, open a terminal and change directory to your Homestead folder and run:

```
vagrant box update
```

Vagrant will connect to the Atlas service and download the newest version if you do not already have it.

To update the Homestead application check the Homestead releases on GitHub to find the newest version and run the command:

```
git fetch origin && git checkout <tag>
```

Where `<tag>` is the newest version.

Listing 2.

```
1.#!/bin/sh
2.
3. # If you would like to do some extra provisioning you may
4. # add any commands you wish to this file and they will
5. # be run after the Homestead machine is provisioned.
6. echo "Seeding The Homestead Database"
7. mysql -u homestead -psecret homestead < /vagrant/database.sql
8.
9. echo "Install tmux, screen, and vim"
10. apt-get install tmux screen vim
```

It is not recommended you run the Homestead application

7 *Vagrant Hostsupdater:* http://phpa.me/vagrant-hostsupdater

from the master branch. Often, the master branch is the wild west of new things we're trying out and other random acts of experimentation.

Sharing Homestead

A powerful aspect of Vagrant is it can easily be shared between developers on a team with minimal effort. Homestead takes this a step further with its "Per Project Installation" method. This is my favorite Homestead feature (I'm biased; I wrote a lot of it) because it allows the developer to add Vagrant into their project as easily as any other Composer dependency. You obviously can't do:

```
composer install vagrant/vagrant
```

However, you **can**:

```
composer install laravel/homeestead
```

I suggest using:

```
composer require --dev laravel/homstead
```

This ensures it moves into your `"require-dev"` section of `composer.json`.

Once you have Homestead in your project's `/vendor` folder, run the command below to copy the minimal required files to your project's root to use Homestead.

```
php vendor/bin/homestead make
```

If you'd like to use the `aliases` and `after.sh` functionality previously mentioned, run the command with a few extra flags:

```
php vendor/bin/homestead make --after --aliases
```

You should commit all of these files into your version control system of choice, except for `Homestead.yaml`. If you customize `Homestead.yaml` create a file called `Homestead.yaml.example`, version control the example file, and add `Homestead.yaml` to your project's `.gitignore`. The reason is the Homestead `make` command uses full paths to folders, and your coworkers or contributors may use different paths.

When you run the Homestead `make` command and the file `Homestead.yaml.example` exists, we parse as much as we can and apply it to the user's `Homestead.yaml`. However, it should always be double checked to ensure they don't miss anything from the example file.

When a contributor pulls down your project from version control, to create their version of `Homestead.yaml` they just need to run the make command:

```
php /vendor/bin/homestead make
```

Remember: the make command does not overwrite any files that already exist, so you are free to version control `aliases`, `after.sh`, and `Vagrantfile`.

We try very hard only to release box updates once a month at most. Due to the file size being nearly one gigabyte we don't want to force you to keep downloading updates. If you have been using and upgrading Homestead for a while you may want to clean up some of the old versions (assuming you're not using them). To prune all old boxes, you can run the command `vagrant box prune` from your Homestead directory. If you want to go back to an older base box version, Vagrant downloads the older version box file again when requested.

Locking Down Homestead

Homestead development moves relatively fast. Sometimes we drop support for features you may still need, or we may bump versions of software you aren't ready for. The most common example of this is the version of PHP; we always try to support the latest version of PHP as soon as we possibly can. Your projects using Homestead may not want such bleeding edge support, and you may still have older projects that aren't ready to migrate yet.

We have made it easy to lock a project into a specific version of Homestead. I highly recommend you do this via the per project method, so you're locking that *one* particular project to the older version of Homestead.

If you have a project you are running on PHP 7.0, and just aren't ready to upgrade to PHP 7.1 you can easily lock an older version of Homestead. Homestead with PHP 7.0 requires at least version 4.0.0 of the Homestead application and version 1.x of the base box. You can specify in your project's `composer.json` to use:

```
laravel/homestead": "^4"
```

And, in your `Homestead.yaml` add a new top-level setting `box: 1.1.0`. Then, run:

```
composer update
vagrant destroy && vagrant up
```

You should see the 1.1.0 version of the box used which contains PHP 7.0.

You can easily keep up to date with changes in the base box by watching the releases on the Settler GitHub repo[8]. These tagged versions match up with the base box versions on Atlas.

Contributing to Homestead

Do you enjoy hacking on Vagrant machines? We'd love to have you help out with Homestead. You can view our GitHub Issues[9] and help answer questions or verify bugs. We've had a number of people contribute bug fixes and new features. Opening pull requests fixing bugs and adding features is exactly how I got involved and eventually added to the team.

Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They live in the Memphis suburbs with their two cats. @JoePFerguson

8 *Settler GitHub repo: http://phpa.me/settler-releases*

9 *GitHub Issues: http://phpa.me/homestead-issues*

March Happenings

PHP Releases

PHP 7.1.3: http://php.net/archive/2017.php#id2017-03-16-2
PHP 7.0.17: http://php.net/archive/2017.php#id2017-03-16-1

News

SitePoint PHP Blog: Game Development with ReactJS and PHP: How Compatible Are They?

The SitePoint PHP blog has a new tutorial posted from Christopher Pitt (known for his "interesting" uses of PHP) covering the combination of ReactJS and PHP in game development. He wants to answer the question of how compatible they are and provide an example to help illustrate. He hops right in to the code from there, starting with the setup of the backend functionality making use of Aerys for the HTTP/WebSocket handling. http://phpdeveloper.org/news/25046

PhpStorm Blog: Working With PHPUnit and PhpStorm

On the PhpStorm blog (from JetBrains) Gary Hockin reflects on a post from Adam Wathan with tips for combining PHPUnit and PhpStorm for more effective debugging. Gary then goes through the points in Adam's post and shows how they can (mostly) be accomplished directly in PhpStorm. http://phpdeveloper.org/news/25029

Delicious Brains Blog: Dependency Management and WordPress: A Proposal

On the Delicious Blog Ian has written up a post with a proposal for WordPress suggesting that it introduce some functionality to help with dependency management and possible conflicts between the needs of plugins. He points out that, while this does have to do with packages installed through it, Composer itself isn't the issue. He offers a few suggestions and what he sees as an "ideal approach" to the problem based on some of the ideas presented here. http://phpdeveloper.org/news/25027

Colin O'Dell: PHPUnicorn—Visualizing PHPUnit Tests

Colin O'Dell, as mentioned on his blog, has put together the instructions for something he calls "PHPUnicorn" (not to be confused with the PHP Unicorn conference) - a real-time system for visualizing unit test results via a Raspberry Pi, some LEDs and a Unicorn pHAT board. The full instructions are over in this article on the Hackster.io site providing you with a list of the components needed, how you'll need to extend PHPUnit with a custom listener and a simple Python script to interface with the Pi and Unicorn board. http://phpdeveloper.org/news/25005

Esben Petersen: A Modern REST API in Laravel 5 Part 1: Structure

Esben Petersen has kicked off his series on creating a modern REST framework in Laravel 5 with part one in the series. This first tutorial focuses on the setup of the application using a "folders by component" approach. The tutorial covers structure on three different levels (patterns): application flow, project folder structure and resource folder structure. http://phpdeveloper.org/news/24982

Voices of the ElePHPant: What Conference Organizers Wish Speakers Knew

The Voices of the ElePHPant podcast has posted a round table, hosted by Cal Evans, session it recent recorded with several conference organizers sharing what conference organizers wish speakers knew about speaking, the events themselves and how they can improve as a speaker. http://phpdeveloper.org/news/24975

PHPBuilder.com: Using Dependency Injection in PHP

The PHPBuilder.com site has posted a tutorial talking about dependency injection in PHP applications covering not only the use of dependency injection (DI) containers but also constructor, setter, property and reflection based injection methods. The tutorial starts out with a simple example of a set of classes that depend on each other through the creation of internal objects. They then show how the different types of dependency injection can help reduce these dependencies with brief descriptions and sample code for each. http://phpdeveloper.org/news/24963

Zend Framework Blog: Announcing Expressive 2.0

On the Zend Framework blog today Matthew Weier O'Phinney has posted the official announcement of the release of Zend Expressive v2.0, the latest major release with several large changes. The post gets into details on updates in the latest version. It also shows how to install this latest version via Composer (or install the skeleton application to get up and running quickly). http://phpdeveloper.org/news/24974

PHP—By the Numbers

by Eli White

If you read this column regularly, you will know I often quote statistics. Usually, in the form of "82% of all websites use PHP"[1]. I quote stats for multiple reasons. First and foremost I happen to be a computer geek *(and am a retired math geek)* and happen to like numbers. However, more importantly to me, numbers do not lie. Yes, they can be manipulated. Surveys can be tainted, and you can massage numbers to give an appearance to the human mind they are implying something they are not. However, in the end, I prefer to side with the numbers. Instead of emotional appeals to concepts such as "Which programming language is best," numbers get us a unique starting point, and a neutral one, to compare differing opinions.

> *"He uses statistics as a drunken man uses lamp posts—for support rather than for illumination."*
>
> *— Andrew Lang*

To that end, I wanted to write today about a recent set of numbers released about programming languages in general: the results of the Stack Overflow Developer Survey 2017[2]. The full report from them is wide, full of tons of information, and completely polyglot. My goal today is to dive through some of these numbers on your behalf. Pointing out the statistics of interest about or around PHP. Hopefully, I can do so without causing any undue subconscious manipulation of my own.

In future months, Stack Overflow will be releasing the raw anonymized data from this survey, at which point I would like to look at some other details specifically about PHP developers which they did not cover in their extensive report. However, in the meantime, let's look at some concrete takeaways about PHP from this report.

Do note this survey is biased since it's from a self-selected group of people who visit Stack Overflow and who chose to take the questionnaire. This alone will cause some bias to the results, such as:

73% of Developers are Web Developers

While this number may be biased because web developers are more likely to use a website as their data source, I do not think this percentage is far off, as the Internet has become the defacto platform by which we transfer information today. There is, of course, some overlap since people often perform multiple roles, but the fact the far majority of individuals are in web development is a sign we are all still in the correct field. *(The next highest number is desktop application development at 29%)* The percentages add up to more than 100% since people may have been allowed to select more than one choice.

88% of Developers Work on the Back-end

Figure 1 perhaps provides a sigh of relief to those of us coding in PHP for a living. JavaScript has in its own way won the language battle, as no matter what programming language you choose to write web applications in, you still need to write JS code. However, the predicted days where backend code was a thin database API layer (which itself could be programmatically generated) and

1 *Actually 82.6% according to the most recent data from https://w3techs.com*

2 *Stack Overflow Developer Survey 2017: https://stackoverflow.com/insights/survey/2017*

Figure 1. Types of Web Developers

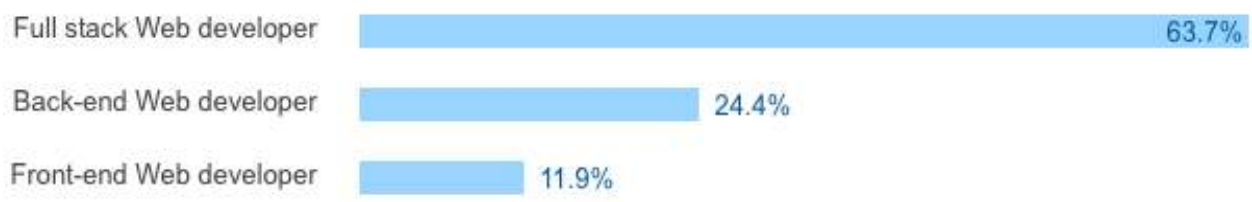
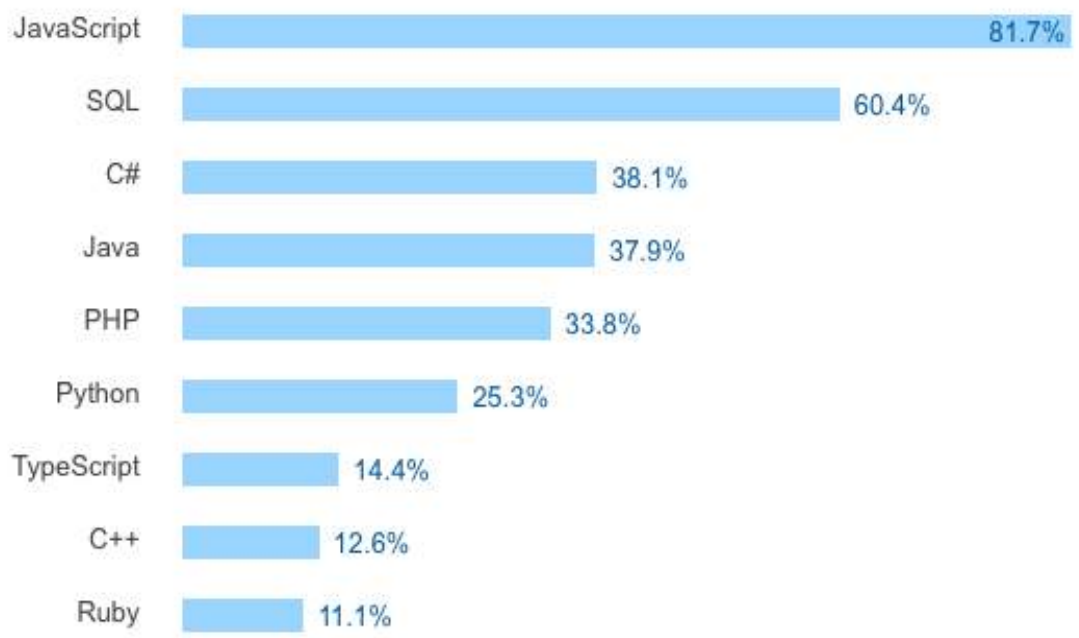


Figure 2. Web Development Language Usage



all the actual code for the application exists in front-end JS has not yet come to pass. The majority of developers do still write back-end code as part of their daily jobs. I will note here however, it does not state how much they code. Someone may have chosen full-stack from the options listed because one day per year they touch back-end code.

82% of the Web Runs on PHP, Only 34% of Web Developers Code in PHP

Perhaps this is shocking to you; perhaps it is not. Now, this number may be influenced slightly again by how often different programming language aficionados happen to use Stack Overflow, and PHP has some fantastic documentation. However again I will posit we take this number as accurate enough within reason.

Why would this be? Frankly, I think this speaks volumes towards the wondrous strides existing PHP frameworks and applications have made. Substantial amounts of the internet run on WordPress, Drupal, Magento, phpBB, and many other pre-built applications. These are all solid tools which either work out of the box, or require only minimal programming to configure to work exactly how you need for your solution.

These PHP-based tools have become ubiquitous, as a result, you do not need as many PHP developers to keep the lights on.

Still, We are in the Top 4 Popular Back-end Programming Languages for Web Development

In fact, we might even be as high as 2nd place. Take a look at Figure 2 again. While this is a graph based upon 'languages people said they used, for individuals who identified as a web developer.' It did not distinguish between whether a person used a language specifically for web development work, or for other tasks they also perform. When looking at the chart again, we can, of course, also discount SQL as not the kind of programming we are discussing. Then JavaScript can mean either front-end or back-end (Node) work. Java, C#, and Python are all technologies used not only for web development but also for desktop application development (Java/C#) or server scripting and data analytics (Python). PHP, on the other hand, is 99% of the time only used to develop websites. While it is impossible at the moment to know where in the top 4 we appear, it is reasonable to think PHP might be in 2nd or 3rd place depending on how those numbers fall.

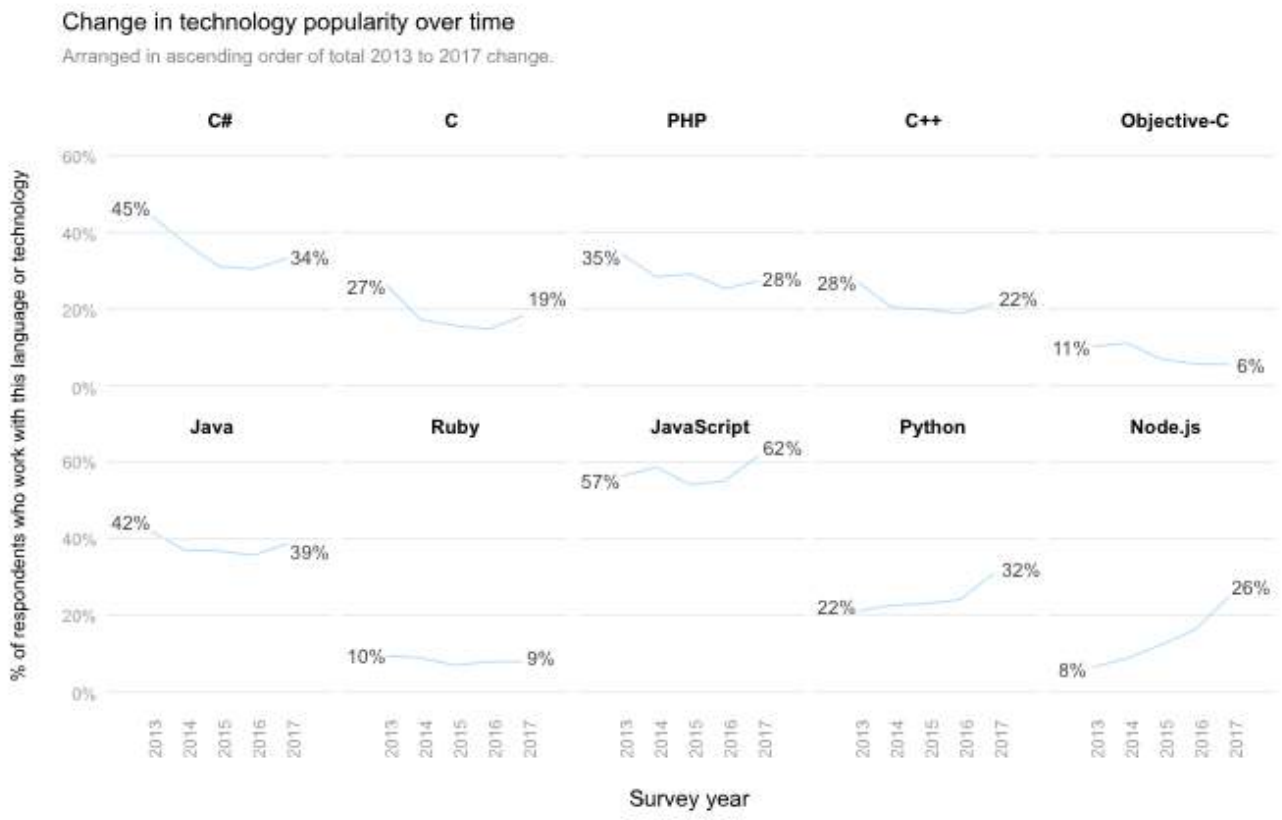
PHP's Popularity is Shrinking

Unfortunately, this is a true statement; however, it is a very slow ride at the moment (with a few bumps along the way). In 2013 their survey had PHP at 35% usage, in 2015 it was 29%, and this year 28%.

However perhaps the most interesting data point in Figure 3, is that every language has shrunk in usage from 2013 to 2017 in this survey, with the exceptions of Javascript/Node.js and Python, which is where PHP lost a little ground. However looking at the chart, we are dropping very slowly compared to other languages which have taken a steep dive.

The takeaway from this? PHP is still

Figure 3. Relative Historical Popularity



around and going strong (in fact we had a slight uptick this year from last), while Python and JavaScript are the hot commodities at the moment.

The Love or Dread of PHP?

Finally, I did want to touch on a set of numbers provided about how many people loved versus dreaded each programming language. Their stats claim 41% of PHP developers love the language, and 59% of developers dread using it.

Though those are strong words to be applied to this, but this question was not exactly what people were asked. Respondents were only asked what languages they currently used which they were interested in continuing to develop with. (Lack of a positive response was then documented as 'dread,' which is certainly an extreme way to interpret those results).

By the same token, the stats say 65% of Hack programmers, 58% of C programmers, 52% of Ruby programmers, and 40% of JavaScript programmers each 'dread' their respective languages.

I take this whole section with rather a large grain of salt. I believe much of this is the continual drive to learn developers in general have, and a desire to seek out and get exposed new technologies.

However, it's important to note that respectively, fewer people were interested in continuing with PHP than some other languages, which is again a bit of confirmation PHP is not the "hot new thing." A trend of which we are all well aware.

Also, it is interesting to note 4% of the respondents said they had never used PHP, but were interested in learning it in the future. There is still interest in the language from those not currently using it, showing some opening for growth.

Conclusion

Breaking down these numbers and what they might mean for PHP has been useful. It's nice to take a realistic look at where we are and where we are going. PHP is not going away anytime soon, and with improvements being made now at a rapid pace with PHP 7 and subsequent releases we have a long future ahead of us.

Hopefully, in a future article, I will be able to get a hold of their raw data and analyze for some more interesting data specifically about PHP web developers.

Eli White is a Conference Chair for php[architect] and Vice President of One for All Events, LLC. He was born as a statistic, ready to consume resources. @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