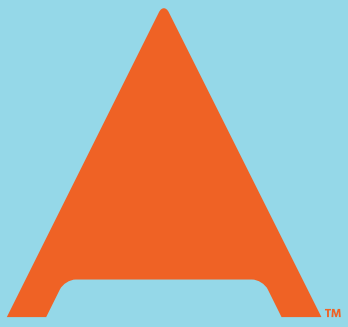
www.phparch.com

December 2016
VOLUME 15 - Issue 12



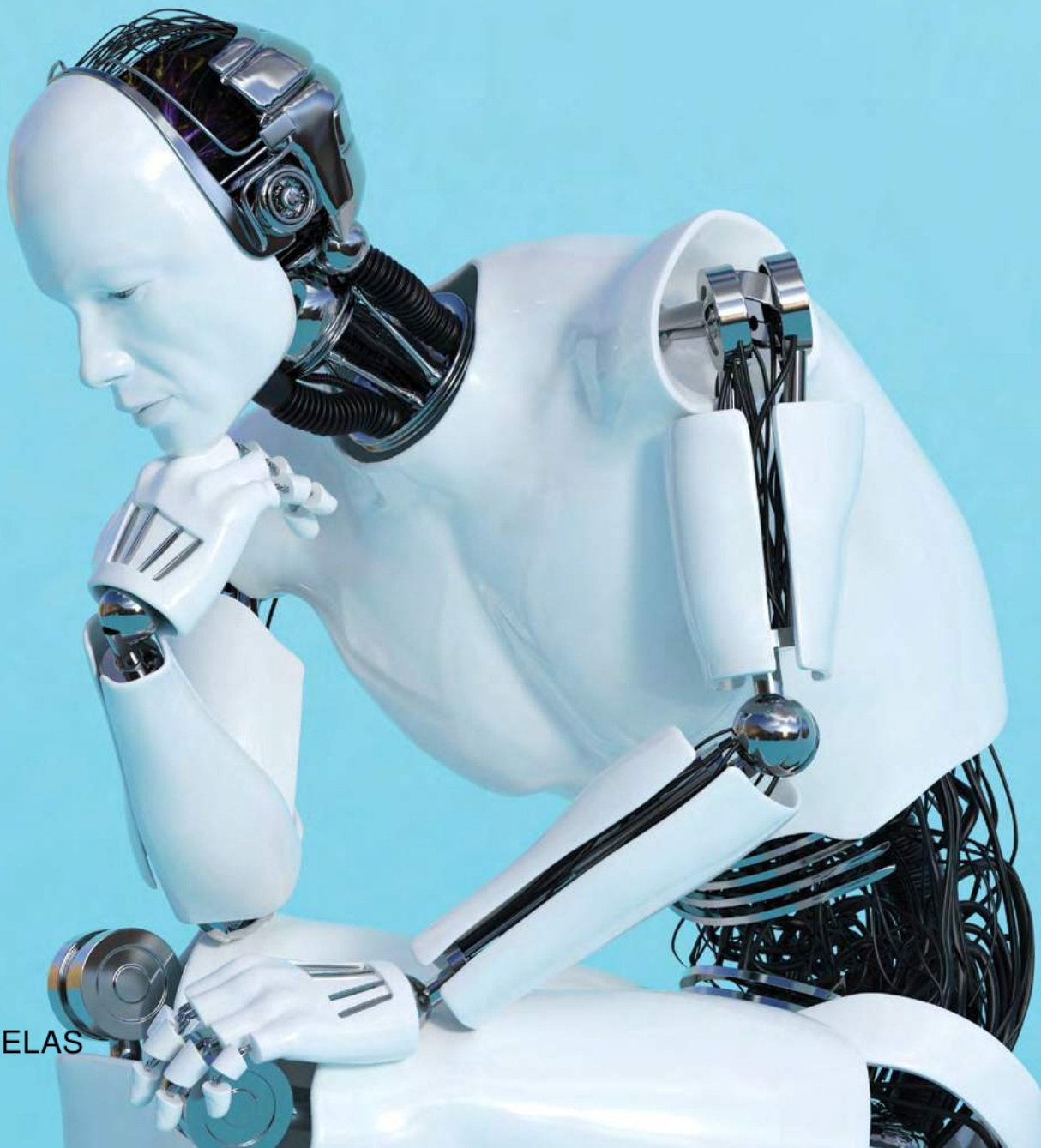
php[architect]

Scrutinizing Your Tests

Behat: Beyond Browser Automation

Strangler Pattern, Part Three: the Rhythm of Test-Driven Development

ALSO INSIDE



Call for Speakers Ends December 30th

photo by tableatny:
https://www.flickr.com/photos/53370644@N06/

php[tek] 2017

Subscribe to our mailing list, or follow us on
Twitter & Facebook for announcements

tek.phparch.com

php[architect]

DECEMBER 2016
Volume 15 - Issue 12

CONTENTS

Scrutinizing Your Tests

Columns

Editor-in-Chief: Oscar Merida

Art Director: Kevin Bruce

Editor: Kara Ferguson

Technical Editors:
Oscar Merida, Sandy Smith

Issue Authors:
Edward Barnard, David Buchmann, Chris Cornutt, Matt Davis, Cal Evans, Konstantin Kudryashov, Matthew Setter, David Stockton, Eli White

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

Scrutinizing Your Tests

Once you dive into testing and start seeing the benefits of having an automated test suite, it's tempting to go all in and start writing tests as fast as possible. Before you know it, you could end up with a test suite that takes forever to run or requires many updates as features are added or changed. Before you get to that point, take a step back to ensure that you're getting the most out of your efforts.

This year, I've been adding Behat tests to a Drupal site I help maintain. Having a good test suite has been a worthy investment. Just the other day, my tests caught that I had missed implementing one of the requirements for a new feature. Still, as the test suite has grown, I've been struggling where and how to separate a feature's desired result from the underlying implementation. It's a tricky balancing act.

In this issue, we have two features that look at effective testing. In *Behat: Beyond Browser Automation*, the creator of Behat and Mink—Konstantin Kudryashov—shares how his use of tests has evolved since his first forays. He explains how Gherkin driven tests should be written to go beyond browser automated tests to ensure you're practicing Behavior Driven Development to test the underlying implementation and not just the user interface. Ed Barnard continues his Strangler Pattern series and in part 3 he writes about *the Rhythm of Test-Driven Development*. As he shows how he writes tests and code to pass the tests, you'll see the real value in Test Driven Development beyond preventing future regressions.

Also in this issue, David Buchmann writes about *Abstracting HTTP Clients in PHP*. See how to keep your code from being tightly coupled to a single HTTP client implementation by using the HTTPlug component in your project. Matt Davis shares the work he did to implement *Decoupled Blocks with Drupal 8 and JavaScript Frameworks*. This is an interesting case study in integrating the front-end JavaScript UI code with Drupal's own output layer to find a solution to enable front end developers to work with the right tool for the job.

In our regular columns, join Matthew Setter in *Education Station: Let's Build a Chatbot in PHP* for a fun project this month. And while it sounds somewhat trivial, you might be inspired to write your own chatbot to make your day-to-day work easier. If you've started thinking of resolutions for the upcoming year, *Building Better Bug Reports* in *Leveling Up* by David Stockton should inspire you. Having a good bug report saves time and frustration when you're asked to fix something that "went wrong." and he'll show you what makes a one. In *Community Corner*, Cal Evans urges us to *Focus on What We Have in Common*. It's a good reminder that the PHP community is not a uniform, monolith. In *Security Corner: Keeping A Secret*, Chris Cornutt looks at the do's and don'ts on storing sensitive information. You already know not to store passwords in plain text, but there are other pieces of data that need to be guarded carefully. *Finally{}*, it's December so Eli White takes a look at the year about to end in *The Year of ???*. What did this year bring and what does 2017 have in store?

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

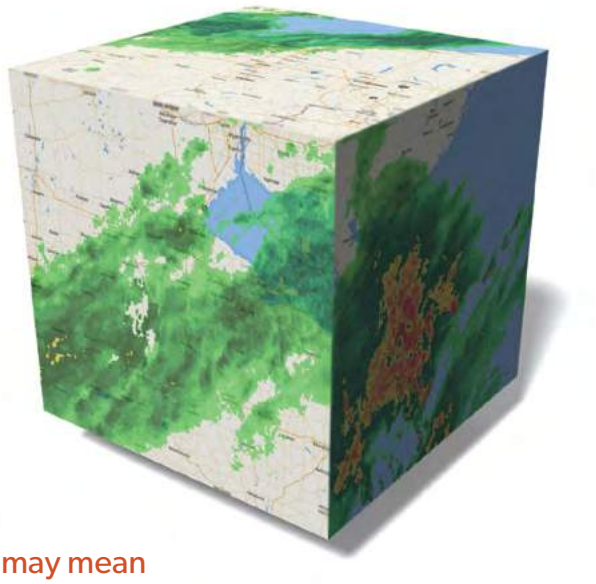
Don't miss out on conference, book, and special announcments. Make sure you're connected with us via email, twitter, and facebook.

- 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/December2016_code

Decoupled Blocks With Drupal 8 and JavaScript Frameworks



Matt Davis

The ongoing JavaScript renaissance is changing the web, and it is affecting the PHP development ecosystem in big ways. The days of sprucing up your UX with a few lines of jQuery are largely behind us, and the gaining popularity of fully decoupled architectures may mean PHP teams are only getting half of the build projects. And, half of those projects may be utilizing API-first CMS resources like Contentful.

Of course, that's just part of the story, and the reality is PHP isn't going anywhere anytime soon. With the release of PHP 7 the language is celebrating a, perhaps more limited, renaissance of its own. Nonetheless, a talented PHP developer looking to build the best possible experience for their users will inevitably end up looking at ways to integrate these two technologies sooner or later.

The Decoupled Blocks project, born out of a "progressive decoupling" model championed by The Weather Company[1], an IBM Business, offers one powerful way to integrate functionality from the latest and greatest JavaScript frameworks into Drupal 8 projects. It also suggests a viable model for handling decoupling pieces of your PHP-based site for rendering by JavaScript components, regardless of the specific PHP system you are using.

weather.com on Drupal

The weather.com team launched its Drupal 7 site in December 2014, making it one of the highest traffic Drupal sites in the world and proving Drupal could scale to meet the demands of such a uniquely demanding website. They chose Drupal from a long list of both proprietary and open source technologies, based on the size of the developer community and contributed module feature breadth, as well as the ability of a small team to proof-of-concept an architectural vision in a matter of weeks.

There were several architectural challenges the weather.com project posed, both of a technical and logistical nature. The technical features combined to create a singular performance and scalability challenge. These characteristics are high variability in traffic, hyper-localized data, and extremely current data being shown to end users.

By high variability we mean that, on an average day, the site sees about 25 million page views. During a major weather event that number spikes to over 100 million. There are 1.3 billion possible coordinate combinations for which a user could request weather data, and the ever important severe weather alerts and current conditions data cannot be delivered in the base page and cached there, since users expect up-to-the-minute accuracy.

There were also internal considerations which impacted the architecture. With large JavaScript and editorial teams, the Drupal system needed to carefully manage how these teams would interact with it. The editorial teams wanted the ability to create and modify not only content, but pages and sections of the site without direct developer involvement. Meanwhile, the JavaScript developer team wanted, simply put, to write JavaScript, and to be able to maintain a swift feature velocity without having to learn anything about Drupal.

Approaching Our Challenges

When considering how to approach this system, the first thing the team did was take a long look at a forecast page since they are the highest trafficked and exemplify each of our challenges. What we saw was different sections of the page had very different needs regarding cacheability and numerous parts of the page were essentially unchanged across locations. We began breaking the page into pieces and grouping them based on their commonality across pages and cacheability.

1 *The Weather Company:* http://www.theweathercompany.com

But wait, as we were beginning to categorize the pieces of pages into buckets, we needed some clarity around just what buckets we had available to us. There were only two main buckets, the same ones available to most modern web applications: a server-side bucket and a client-side bucket. The server side was the logical place to do heavy lifting and any kind of processing which could be cached and reused across multiple pages or multiple users. The client side was the logical place to handle user-specific calculations and rendering.

There was a third piece to this puzzle, though, as we also knew we would be heavily relying on our CDN partner Akamai to offload caches from our origin servers and reduce latency by answering requests with servers which were as physically close to the end user as possible. We needed to avoid repeated calculations wherever possible, and by combining some clever Akamai rules with Edge Side Includes (ESI), we could stitch together different pieces of the page server-side at the edge before delivery to the client.

Our ultimate solution, then, had three places where work was done: the page wrapper, including the theme, header and footer, as well as the most static and cacheable pieces of content were generated at origin, identically for all locations and users. At the edge, we request the wrapper from origin and a complete location context using ESI. That wrapper and context are passed to the client, where Angular does the personalized work of retrieving current conditions and presenting the forecast. Other content data generated by Drupal, like nodes and views, is stored in a services layer using a key/value store, which can then be queried directly by the client. Figure 1 shows how these pieces fit together.

To learn more about the caching and ESI side of this architecture, watch my presentation at DrupalCon Dublin, called Origin, Edge, Client: Where to do the Work?[2]

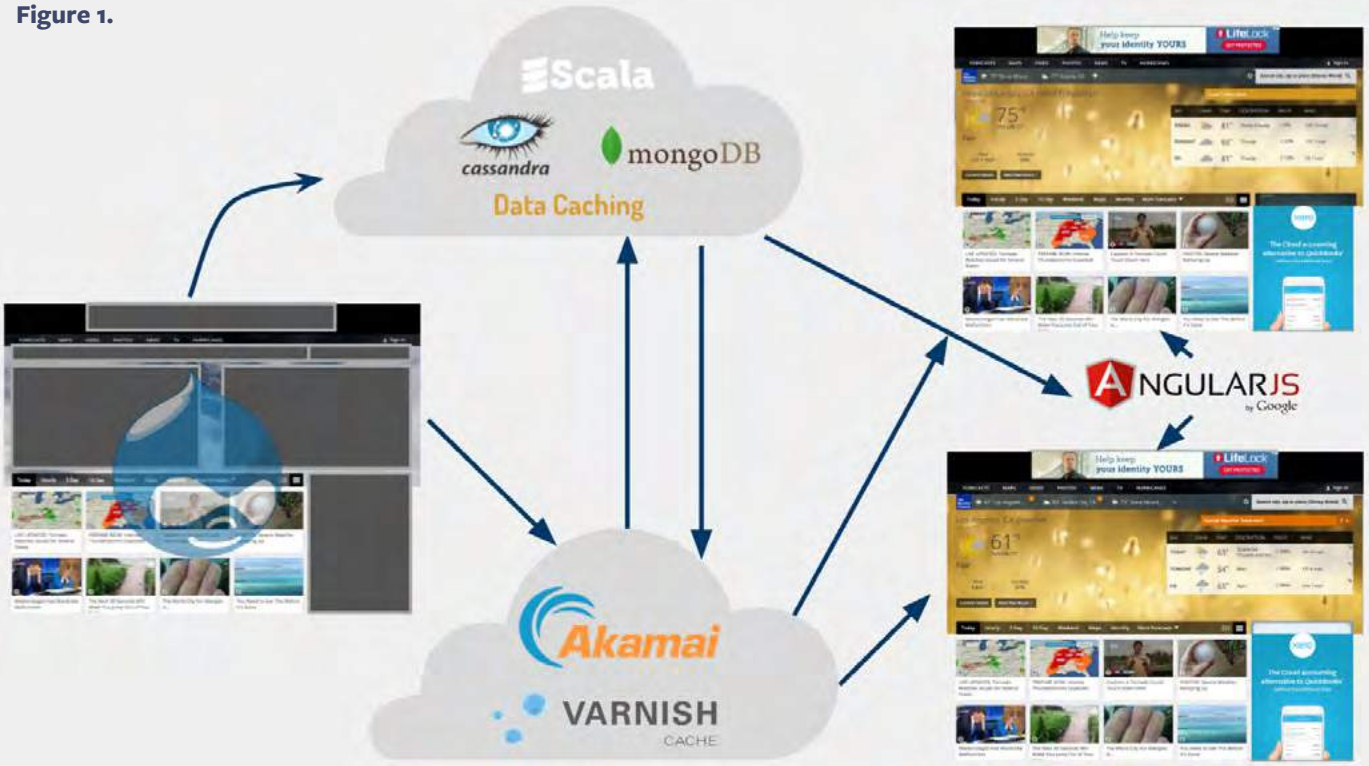
The Presentation Framework

We called our solution *The Presentation Framework*, and it would ultimately become the progressive decoupling model which paved the way for Decoupled Blocks. Let's take a brief look at how this all came together under the hood in Drupal, and how this solution met the needs of all stakeholders while being performant and scalable.

We decided early on our pages were going to be built using the Panels module, for several reasons. First, by training editorial teams in using the Panels interface we would allow them to move around, add, and remove panes from pages, and even create brand new pages. Panels variants would allow us to maintain URL parity while serving different versions of pages based on device, for example, and CTools contexts were a powerful way to pull a small piece of data from a request URL and expand it into a larger context object. Meanwhile, page configurations would be fully exportable as Features. Individual panes would be reusable across pages as well, facilitating our development workflow.

2 *Origin, Edge, Client: Where to do the Work? :* https://youtu.be/Lt4XOqa_yMI

Figure 1.

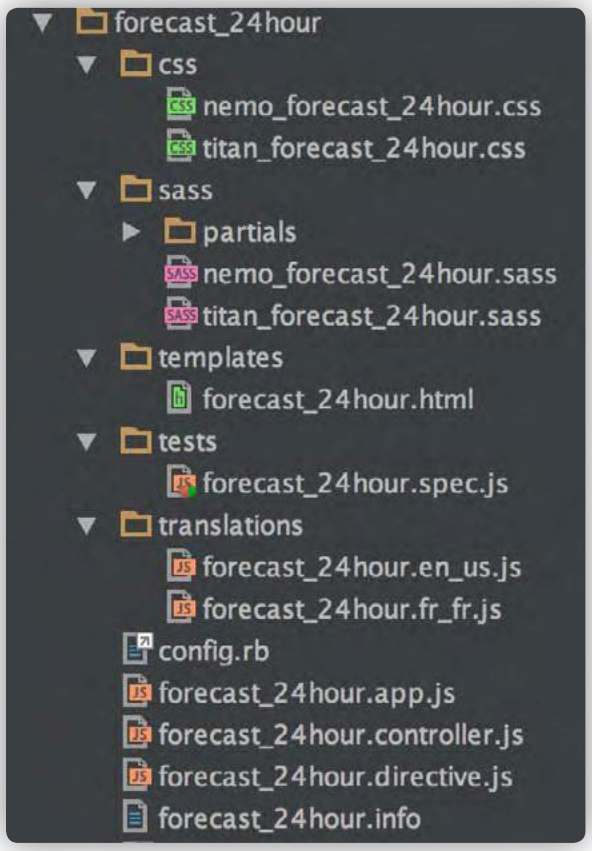


Panels[3] is a Drupal module which allows site builders to create pages through a drag-and-drop UI where they can select different items to show on the page. CTools context can be used to customize these items based on conditions like the URL, the logged in user, and more.

There was one big problem with this solution. We knew pieces of our pages were going to be written by JavaScript developers in Angular, and we wanted those pieces to be manageable through the Panels interface. But, what we certainly didn't want was to have to teach all these JavaScript folks how to write CTools plugins in PHP so their Angular components would show up as Panels panes.

The Presentation Framework module solves this problem by allowing JavaScript developers to create their components independently, without having to learn anything about Drupal's APIs. Even though Angular 1 is not inherently component based, we enforced a rough component structure onto their work—each piece of functionality must be compartmentalized in its own directory with all the assets it needs, including its template, CSS, JavaScript, and any static assets like images (Figure 2). Then, by writing a small JSON-formatted file with metadata about that component, the developer could make their piece of functionality discoverable to Drupal, see Figure 3.

Figure 2.



3 *Panels:* https://www.drupal.org/project/panels

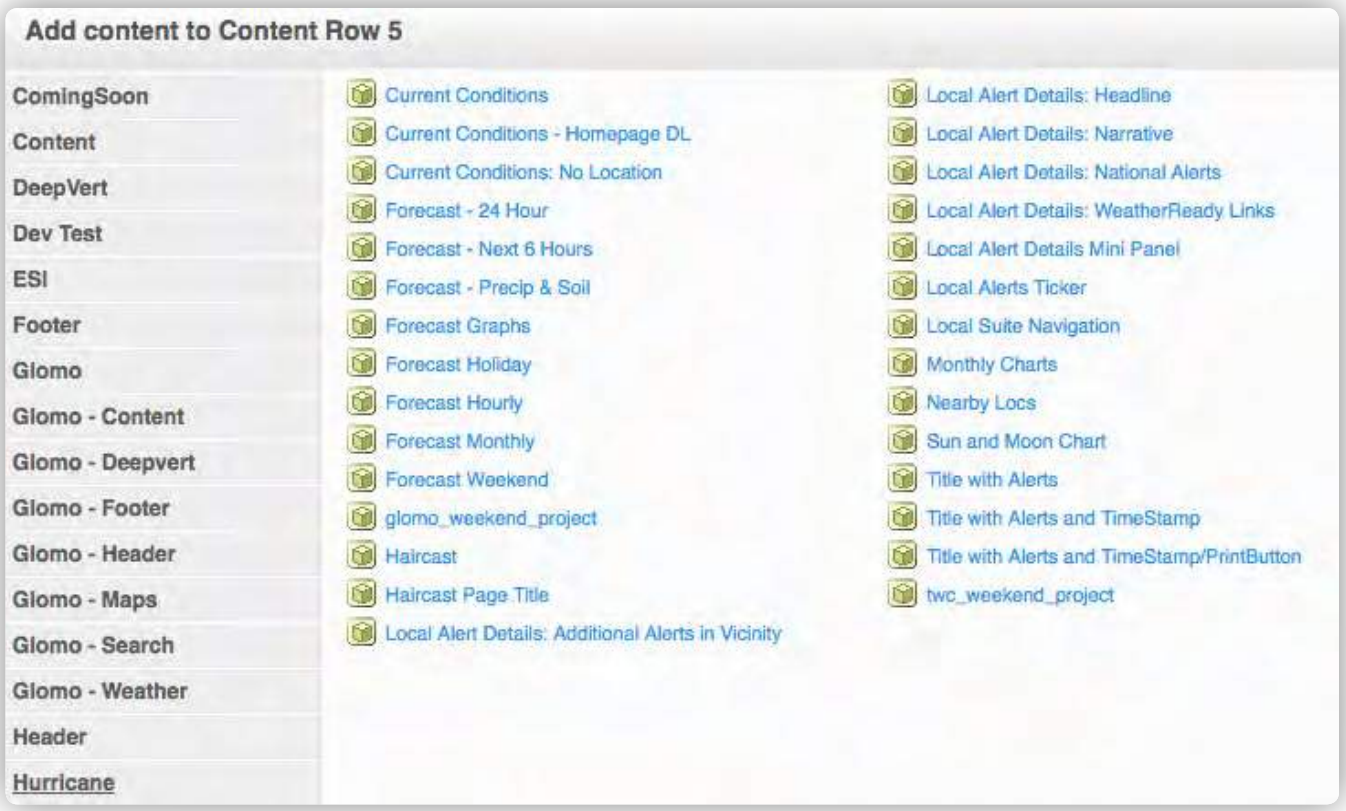
Figure 3.

```
{
    "readable_module_name": "Forecast - 24 Hour",
    "description": "Module description goes here",
    "category": "Weather",
    "type": "module",
    "module_status": "active",
    "version": "1.0.26",
    "presentation": "angular",
    "template": [
        {
            "file": "templates/forecast_24hour.html",
            "machine_name": "default",
            "label": "Template 1",
            "type": "static"
        }
    ],
    "context_required": [
        {
            "type": "twclocation"
        }
    ],
    "add_js": {
        "footer": [
            {
                "src": "forecast_24hour.app.js",
                "group": "angular.general",
                "weight": 10
            },
            {
                "src": "forecast_24hour.controller.js",
                "group": "angular.general",
                "weight": 11
            },
            {
                "src": "forecast_24hour.directive.js",
                "group": "angular.general",
                "weight": 12
            }
        ]
    },
    "add_css": {
        "header": [
            {
                "src": "css/titan_forecast_24hour.css",
                "group": "angular.general",
                "theme": "reboot",
                "weight": 11
            }
        ]
    },
}
```

Our module's first job is to create a CTools plugin which walks the directories known to contain these components, finds the JSON info files containing the metadata about them, and generates a new Panels pane type for each one it discovers. The info files contain all the information Drupal needs to build these panes, including cache time to live, dependencies, versioning, required contexts, and whether it should be rendered at origin or using ESI.

One unique feature is worth mentioning. Our JSON info files also allow configuration fields to be declared, using what amounts to a JSON-ified version of Drupal's Forms API. While this is a slight bending of our rule concerning JavaScript developers with Drupalisms, the basics of declaring fields using the Forms API is straightforward, and allows them to pass instance configuration choices to the editorial team.

Figure 4.



In the end, the flow looks like this: JavaScript developers create new components independently of any back-end developer, placing each one in its own subdirectory in one of the configurable places our module looks for them. Our module walks these directories, finds the components and turns them into Panels panes. Then, editorial teams place those panes onto pages and configure them for display as shown in Figure 4. As of today, weather.com developers have written several hundred of these components in Angular 1, moving far more quickly than would have been possible if front and back-end developers had needed to collaborate on each new component.

Weather Underground Weighs In

The weather.com[4] site is owned by The Weather Company, who also owns an independent weather site, wunderground.com[5] (Weather Underground, or WU for short). At the end of 2015, WU began migrating its site onto the Drupal platform built for weather.com. Early on in planning out the migration, a big question was raised: should we be using Angular 1 for a brand new build, when Angular 2 is largely complete?

My job as the Drupal architect on the project was to evaluate what it would take to make our Presentation Framework, which was tightly coupled to the existing Angular 1.3 implementation, work with a completely different JavaScript framework. Of course, once the weather.com team heard talk of other frameworks, many of them were interested in the idea as well, and soon the task became refactoring the Presentation Framework to make it JavaScript agnostic.

Because this platform had to always be fully backward compatible and not break weather.com, the complexity of refactoring the Presentation Framework was substantial, and a solid proof of concept took a couple of months of effort (it is worth noting this is also the reason Drupal 8 was not on the table for WU, because of the need to maintain backward compatibility). Once complete, though, the potential was clear: a JavaScript-framework-agnostic, progressive decoupling tool for Drupal which would allow front end functionality to be written without Drupal developer support.

A contrib[6] version of this functionality had been discussed several times since the Presentation Framework's inception, but the large amounts of business logic deeply coupled into the module complicated this. Although The Weather Company has been very receptive to supporting open source and giving code back, there was little appetite for another large refactoring. Further, in terms of its use to the Drupal community, a Drupal 7 contrib module of this type was questionable, since it would mainly be used on new builds, and more new builds are being done in Drupal 8.

Decoupled Blocks is Born

Because my full-time job was working on these two Drupal 7 projects, I was not getting any billable time to explore Drupal 8. I had spent some time contributing to Drupal 8 core and working on upgrading some of the Drupal 7 examples modules, but most of that effort was over a year before

4 *weather.com:* http://weather.com

5 *wunderground.com:* http://wunderground.com

6 contrib *is Drupal shorthand for a community contributed module*

Drupal 8's official release. One weekend I was sitting in my apartment in San Francisco and realized I didn't have a whole lot planned. I decided it would be fun to finally play with writing my first Drupal 8 module. It helped that I already had a good idea for a module in mind.

Dries Buytaert, the creator of Drupal, had been blogging about progressive decoupling, where Drupal serves the page wrapper but pieces of the page are decoupled. I had been living in that problem space for quite some time, and had a proven model of how to do it. So I started digging into what it would take to build the Presentation Framework in Drupal 8.

As soon as I began investigating, a few things became apparent about how the Drupal 8 version of this code would diverge from the Presentation Framework. First and foremost, I began to look at the CTools, Page Manager, and Panels ecosystem in Drupal 8, and found something which excited and surprised me: CTools Content Types, also known as Panels panes in Drupal 7, were no more. The new Panels was built leveraging Drupal 8's Blocks API directly, and Blocks in Drupal 8 are vastly improved.

This meant if I made the Drupal 8 version of the Presentation Framework use the Blocks API, it would work *either* in Panels, or in the core Blocks UI, or in any other block placement UI which may come along later. We could have all the same great features for our decoupled components, but with more flexibility around how they would be used by site builders, and no more Panels dependency!

The next step was to figure out how the component discovery mechanism would work in Drupal 8. The Presentation Framework uses a third party JSON schema validator and a bunch of custom code to iterate over component directories and extract the metadata we need. But I knew Drupal 8 had an internal discovery mechanism for finding YAML files used to declare modules, themes, and profiles, so I began digging into core to see how that might be extended to meet our needs.

As it turns out, our extension scan and discovery was exceedingly simple, with core's version allowing us to easily add a new *pdb* (Progressively Decoupled Blocks) type of `info.yml` file to scan. We implement a `ComponentDiscovery` class extending `ExtensionDiscovery`, and define a `getComponents` method which executes the actual scan. This will scan every directory core looks for stuff in by default. Listing 1 has this method's code.

To tie it all together, we need the discovered components to be turned into blocks automatically so site builders can place them onto pages. Figuring out how to dynamically generate block types involved some more digging around in core, which led me to the notion of *deriver* classes. Derivers allow you to generate many related plugins at once, which is exactly how we want our block types to behave. The `PdbBlockDeriver` class triggers the `getComponents` scan, then turns each component found into a derivative block.

LISTING 1

```
01. public function getComponents() {
02.   // Find components.
03.   $components = $this->scan('pdb');
04.
05.   // Set defaults for module info.
06.   $defaults = array(
07.     'dependencies' => array(),
08.     'description' => '',
09.     'package' => 'Other',
10.     'version' => NULL,
11.   );
12.
13.   // Read info files for each module.
14.   foreach ($components as $key => $component) {
15.     // Look for the info file.
16.     $component->info = $this->infoParser->parse(
17.       $component->getPathname()
18.     );
19.     // Merge in defaults and save.
20.     $components[$key]->info = $component->info + $defaults;
21.   }
22.
23.   $this->moduleHandler->alter('component_info',
24.                               $components);
25.   return $components;
26. }
```

Agnostic Out of the Gate

One of the big advantages of doing a complete rewrite of the Presentation Framework for Drupal 8 is instead of a refactor to wedge in the ability for multiple JavaScript frameworks to be usable, Decoupled Blocks has been written from the ground up to be agnostic. The main `pdb` module handles the overall tasks of finding components and turning them into blocks, and provides some other helpful functionality which can be used across any framework.

Different JavaScript frameworks define their unique implementations in their own sub-module, for example, the `pdb_ng2` module defines the behavior of our Angular 2 components. Then, in the component `info.yml` file by setting `type` to `pdb` and `presentation` to `ng2`, Decoupled Blocks recognizes this is an Angular 2 component which will need to bootstrap an Angular 2 application on any page containing such a block.

Components Need Drupal Data

One thing which becomes evident pretty quickly when building decoupled JavaScript components in a Drupal site is very often these components will need some data from the Drupal site they'd like to display. Drupal is great at marshaling relational data, and our components will be at their best when they take advantage of that. There are a few ways this is possible with Decoupled Blocks.

The first and most obvious way to get data from Drupal is the same way a fully decoupled site would, through a request to a REST-ful API endpoint. For example, if you want to display related content in a component, you could create a view in Drupal which outputs the needed data as JSON, and then in your component make an HTTP request to the view's path and iterate over the data as you build out your component. The disadvantage of this approach is it adds a request

and that request is being made client side. This could be a performance concern, especially if multiple components on the same page are making additional requests.

Decoupled Blocks offer another way to make data available, using the global `drupalSettings` JavaScript object. When you place a block, you create a new component instance, and all instance configuration and component metadata is passed forward through `drupalSettings` automatically. Further, if you specify in your component's `info.yml` file that it requires a node entity context, for example, the component will only render when such a context is present, and the entire entity object will be passed forward. Relying on `drupalSettings` allows us to avoid additional requests for the data we need, but as applications grow we will want to be mindful not to cause too much bloat as numerous components add their own settings.

There are other possibilities for retrieving Drupal data which Decoupled Blocks supports. Using the GraphQL library and module[7] , you can set up a single dynamic API endpoint where components can retrieve whatever data they need. Similarly, the Waterwheel SDK[8] can be used to improve client side communication with the Drupal back end. We encourage developers looking to build more complex Decoupled Blocks applications to consider the entire range of possibilities, and also using a combination of them, to achieve the best results.

Current State

As of this writing, the module remains in an alpha state, but I am working towards being ready for the first beta release by php[world] in December. The Angular 2 sub-module is by far the most fleshed out because it's the framework I've been engaging most closely with at my day job, but several other contributors have begun building on the React implementation. There has been some basic proof-of-concepting around Ember and Vue modules which have yet to be committed. Meanwhile, the Angular 2 module has full npm integration, a gulpfile for handling common development tasks like watching changes to component files and compiling typescript to JavaScript, and more.

> *To download Decoupled Blocks, you can visit the Drupal.org page[9] or the Github repository[10]*

What's Next?

Decoupled Blocks aims to make it easier to integrate JavaScript components seamlessly into your Drupal site. Whether it is a large-scale app with numerous components all hand written by your developers or a single open source component you found online, the goal is to make adding those to your site as simple as adding some `info.yml` files describing your component(s). This means the module could be useful both to enterprise level projects, like the one it was born out of, or to small projects where Drupal meets the needs of the entire site, except for one piece where a richer UX is desired which only a JavaScript component can realize.

There are still a few hard problems left to solve to realize this vision, and I could use help from both Drupal 8 developers and especially experts in JavaScript frameworks to get us there. Questions remaining include:

- How do we best facilitate dependency management when components will have their own `package.json` files which need to complement those of the framework module itself?
- How do we put good guard rails in place and offer usable off-the-shelf build tools without being overly opinionated?

I would love to see developers with an interest in other frameworks like React, Ember, or Vue (or your favorite) help build out those modules and example components, and I would love more and better example components for Angular 2.

Beyond actually helping with development tasks, I think there are a lot of projects which can benefit from the infrastructure this module provides, and I want to hear about your use cases so I can be sure to cover as many uses as possible. With the goal of providing the richest and most intuitive experience for our users, Decoupled Blocks is a tool worth considering for your next JavaScript heavy Drupal project.

Matt Davis is a Lead Drupal Architect for Mediacurrent living in the mountains of Asheville, North Carolina. He has been working with Drupal full time since 2007. This year he was the Coding & Development track lead for DrupalCon New Orleans, and also helped lead the migration of wunderground.com onto the Drupal / Angular platform originally built for weather.com. He is the creator of the Decoupled Blocks module for Drupal 8, a progressive decoupling tool for multiple javascript frameworks. @im_mr_jmd

7 *GraphQL library and module: https://www.drupal.org/project/graphql*

8 *Waterwheel SDK: https://www.drupal.org/project/waterwheel*

9 *Drupal.org page: https://www.drupal.org/project/pdb*

10 *Github repository: https://github.com/mrjmd/pdb*

Behat: Beyond Browser Automation

Konstantin Kudryashov

Behat is a tool written in PHP to support teams in practicing Behavior-driven Development. In its simplest form, it is a test automation tool which focuses on comprehensibility more than it does on validity. Behat provides you with a simple developer-agnostic language—Gherkin, and then gives you automation capabilities on top of it. Gherkin's semantic flexibility is both its biggest asset and its biggest flaw.

The First Years

In the Gherkin[1], and by extension Behat[2] world, this:

```
Scenario: Product costing less than 10 results in \
delivery cost of 3
  Given there is a product with SKU "RS1" and a cost of \
5 in the catalog
  And I am on "/catalog"
  When I press "Add RS1 to the basket"
  And I follow "My Basket"
  Then I should see "Total price: 9"
```

is as good of a feature as this:

```
Scenario: Product costing less than 10 results in \
delivery cost of 3
  Given there is a product with SKU "RS1" and a cost \
of 5 in the catalog
  When I add the product with SKU "RS1" from the \
catalog to my basket
  Then the total price of my basket should be 9
```

However, when you consider the larger implications of the evolution of these features, the second one is leaps and bounds better than the first. We will come back to the exact reasoning later in this article. As with any "evolution," the difference is very hard to spot on the first sight, before you get to more complicated cases.

When I first started developing Behat, its ability to test applications end-to-end through the browser was the most attractive feature. It was about time I tried to get into Test-driven Development, so the simplicity of interface-focused tests became the answer to that drive and a little obsession of mine. This drive was so entrenched in my thinking in the first Behat years you couldn't hear me talking about Behat without talking about a user interface. Perhaps the greatest example of this overwhelming focus on UI automation was how quick after initial introduction of Behat I created and integrated it with Mink[3]—the browser automation tool. Back then, browser automation and Behat were almost indistinguishable concepts for me. Deep Mink integration, default Mink step definitions, and the rest were the product of that time. But things have changed drastically since then; I learned how wrong I was about that approach.

I remember when I first heard from my Ruby friend about this cool new testing tool—Cucumber[4]. Cucumber and its integration with Capybara[5] seemed like an incredible story of taking something as complex as TDD and making it as simple as writing down interactions your users should have with the website. I started looking for a possibility to test PHP applications with Cucumber. There were ways, but they all shared a similar drawback—even if your tests are end-to-end, you do need to have the ability to quickly and easily access the application persistence to clean the environment, or do a test setup. Obviously, this was a fairly tricky proposition for a testing tool written in Ruby. I was essentially forced to start writing a PHP implementation of Cucumber—something destined to become Behat.

I love to tell this story because it highlights how our methods and tools shape our thinking. As I stated previously, my first years with Behat went under the sign of browser automation. As the time went on, though, things started to change rapidly. I like to think the first year in Behat life was all about me developing Behat and the rest of our years together were all about Behat developing me. Through development of the tool I learned the way I was using and understanding the tool were far from optimal. That's when I stepped on the transitional path from being a Behat user to being a Behavior-driven Development practitioner. And with every new step on this path, browser automation was losing its hold on me.

1 *Gherkin:* https://cucumber.io/docs/reference

2 *Behat:* http://behat.org

3 *Mink:* http://mink.behat.org

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

5 *Capybara:* https://github.com/jnicklas/capybara

Behavior-Driven Development

BDD[6] is a methodology coined by Dan North and Chris Matts. It is a collaborative process created as an outcome of collaboration between a developer (Dan North) and a business analyst (Chris Matts). BDD was developed as a way to bring practice as technical as TDD to the context as non-technical as traditional business analysis. Dan and Chris recognized there's not much difference between how our code and our business processes go. From their perspective, every computer program and every business process follow the same flow:

1. When the action is performed
2. Then, the output is produced

Every method call, every function has the same reason to its existence— when called (action is performed), it produces some effect (output is produced) like saving an entity to the database, checking form validity, or switching DNS records. Interestingly, every business interaction or a process has the same effect—every tax form submission, every support phone call has the same purpose—producing outputs (or outcomes). The only addition Chris made based on his experience is that he introduced the context into the mix:

1. Given the context
2. When the action is performed
3. Then, the output is produced

6 *BDD:* http://behaviourdriven.org

Context was the third and the last important part of this equation. From this point onwards BDD became a way to have conversations about systems at different levels of said systems, both business and technical ones. When given a task of implementing a feature (e.g. "product delivery cost") you would ask, "can you give me an example of this?" and from this point you would channel the discussion into the Given-When-Then structure. The structure here is the interesting part, not the keywords themselves (Given-When-Then). Sometimes, your examples might lack any of above keywords yet would still follow the Context-Action-Output structure. Structure is an interesting bit, not the syntax.

Another great thing about BDD is how devoid it is from implementation or technical details, which is intentional. By disconnecting our problem discussion from our solution finding processes Dan and Chris effectively created an explicit space for teams to explore the problem domain enough without being overwhelmed by one single solution. Database layer, architecture, programming language used, or even the UI being employed, all these are left outside of said discussion. That is the lost essence of Behavior-driven Development.

Gherkin and Browser Automation

When you create something as seminal as BDD, people notice and start following. Technical people tend to follow with processes and tools. Eventually, Given-When-Then spawned the language-agnostic specification format called Gherkin, and a little later the tool employing the format for test automation—Cucumber. Aslak, the creator of Cucumber, famously said Cucumber is the world's most misunderstood automation tool (because of the lack of interest in underlying principles). As you probably guessed from the beginning of this article, I can proudly say I was one of the first people who completely misunderstood the underlying principles of Cucumber.

Gherkin made it very simple to provide an automation layer for your application without much knowledge of programming languages, design, testing, or even development practices. This fact alone attracted thousands of people to both Cucumber and Behat as test automation tools devoid of a need to worry much about good practices of design or automation. Easiness of testing brought by these tools suddenly meant thousands of people who otherwise didn't know where to start with automation now had a very clear path—installing a tool and writing a "BDD test" in Gherkin. Sadly, as with anything "very simple" and "trivial," crucial drawbacks are hidden in using Behat and Cucumber this way.

I worked on tens of projects employing both BDD as a practice and Behat as an automation tool and many of them shared the same problem— something which seemed like a good idea at the beginning of a project ended up being a huge block towards the middle of it. One such great idea which turned bad was using your Gherkin features primarily for

browser automation. It is easy to see why using Gherkin for browser automation might seem like a great idea at the beginning; no architecture implications, no test setup, or technical skills required. But why does this turn out to be a bad idea in the long run? In my experience there are two reasons why browser automation kills your test suite and BDD practice in general:

1. User interface tests are notoriously slow and brittle.
2. Scenarios focused on user interface are tightly coupled to a particular implementation.

In the following sections, I'll address each of these points individually and try to explain why are they such a big problem for your usage of Behat.

User Interface Tests Are Notoriously Slow and Brittle

Your test is as stable as the least stable layer of the stack it is going through. Your test is as fast as the slowest layer in the stack it is going through. Let's say we are describing the delivery cost calculation for a particular a basket. Let's also assume we are using extensively rich UI driven by AJAX and the discount logic is a combination of core objects and services. In this case, each of our end-to-end Behat tests will execute in the following manner:

1. Behat executes a single step in the scenario.
2. The step sends a command to Selenium server.
3. Selenium server sends a command to the JavaScript block injected into the browser.
4. JavaScript block injected into the browser forces an operation inside the browser window.
5. Browser performs and operation forcing it to send an AJAX request to the server.
6. Server receives a request and parses it into a controller call.
7. Controller delegates the call to the combination of core objects and services.
8. Core objects and services do calculation and return the result back to the controller.
9. Controller packs the result into an AJAX response and sends it back to the browser.
10. Browser receives the response and updates the UI.
11. UI update forces JavaScript block injected into the browser to notify Selenium server.
12. Selenium server notifies back the step in your Scenario.
13. Behat goes to the second step in your Scenario.

This is what happens every time you call `followLink()` in your step definition, sometimes tens of times per step definition. In real world terms, this is somewhere close to a 1s of your test execution time. The interesting part begins when you start asking the question: out of this entire stack, what is the single most important thing for the "delivery cost calculation" test you wrote? It's step eight: "Core objects and services do calculation and return the result back to the controller." Everything else is an indirection necessary to render the interface and capture the output from the user securely. User input/output must be tested, but should it be tested together with the "bundled discount price for all the products in a basket" logic? That's quite a question.

Effectiveness of tests as with anything else in the world is defined by the costs and values they add to the process. What is the cost of a test? It's the time it takes to write and maintain a test! What is the value of a test? It is two-fold; part of it is about helping you design the right solution correctly and another part is about protecting the right solution from degradation or breakage. Let's look closely at how our 13-layers delivery cost test performs in terms of its cost and value.

First, UI tests are notoriously cheap to implement the first time around, but what I am interested in is how cheap are they to maintain. How often do you think your business will ask you to change the delivery cost calculations? My guess would be every time business rethinks its business model, operations and marketing strategy, in other words—not very often. How often do you think you will need to update the UI? Based on my experience, every time a new major JS framework is released. Likely, every couple of minutes. Jokes aside, the rate of UI change is obviously faster than the rate of change in the business logic, by a huge margin! What do you think happens when each element of your business logic is tested through UI? You are artificially linking rates of change for both, which results in the smallest common denominator— every time your UX team moves the basket controls around the UI you are forced to fix all your tests, even the ones testing the calculations which didn't change a single bit. That results in a test suite which is extremely costly to maintain, because you are effectively forced to update the majority of your test suite every time UI is updated. And, UI is frequently updated. Otherwise, your business will end up being far behind its competition.

OK, entangling UI and core business logic inside your tests drastically increases the cost of maintaining your test suite, so what? Surely the value these end-to-end tests provide us outweigh the cost. Except they don't! Remember, the value your tests are supposed to deliver is all about design support and reduction of breakages. So how do your UI-focused tests help with the core object design? They don't, because all the business logic is now hidden behind levels of indirection such as UI and persistence. Even more, all the information your Gherkin scenarios are obsessed with is how a particular button is called or where is it located on the page, not how the discount calculation is supposed to work.

How about a reduction in breakages? Ask yourself, when was the last time your UI test highlighted that you broke the logic rather than the fact you broke the test by changing the CSS class of a button. Every single time your test fails without the underlying logic (the one being actually tested) being

broken, you pay a high price; you and your team loose trust in your test suite, one bit at a time. Your test suite becomes less and less reliable every minute. Maintaining the test suite stops being a useful activity to support the team's progress and starts being a chore.

I've observed many teams struggle when their test suite "suddenly" started taking 30+ minutes to execute and is now constantly residing in a red state. A test suite which takes longer than 30 minutes to run is a test suite no one wants to run or spend time fixing. These teams are often asking me for a single "get out of a jail card"—Page Objects or any other obscure, non-obvious testing pattern which would immediately relieve them from all their pain. The truth is much harder to swallow. In the same way it took them months to get into the situation they're in, it will take them weeks or even months to get out. And, the reason for that is the second biggest drawback of UI-obsessed Gherkin features—they are very inflexible towards their implementation.

UI Focused Scenarios are Inflexible to Implementation

The biggest benefit of using Gherkin is it allows you to separate the problem definition from the implementation, or even to a certain extent, testing. Properly written examples give you an incredible amount of flexibility around how you're going to test or implement the feature. In contrast, poorly written, obsessed with user interface examples are extremely tricky to implement or test in any other way than through the user interface.

Let's look at the first example from the beginning of this article:

```
Scenario: Product costing less than ·10 results in \
delivery cost of ·3
  Given there is a product with SKU "RS1" and a cost of \
5 in the catalog
  When I add the product with SKU "RS1" from the \
catalog to my basket
  Then the total price of my basket should be ·9
```

Do you think we can test this feature through the UI? Absolutely, we can. Do you think we are forced to? Can we test this exact scenario directly against the core domain objects, exercising them closer to the logic we care about? Yes, we can! The magic of this style of Gherkin scenarios is they allow us flexibility in their implementation. You can, for example, start by testing and implementing this particular scenario through UI with Mink. Then, when your test suite starts reaching towards a "couple of minutes per suite execution" barrier, you can go back and refactor your step definitions for this scenario to go through the business logic, separately from the infrastructure or the framework. And all without a single change to the scenario itself!

Now how about this example instead:

```
Scenario: Product costing less than ·10 results in \
delivery cost of ·3 Given there is a product with SKU \
"RS1" and a cost of 5 in the catalog
 And I am on "/catalog"
 When I press "Add RS1 to the basket"
 And I follow "My Basket"
 Then I should see "Total price: ·9"
```

Can we test this scenario against anything but UI? No, we can't. Even more worrisome is it is hard to understand what this scenario is supposed to actually test—all we see here is buttons and forms. It is famously tricky to extract the business essence from your UI scenarios. By writing your scenarios in this way you are effectively locking yourself out of other options for testing this specific feature later.

UI focused scenarios are notoriously inflexible to implementation because they are obsessed with it—pages, nodes, buttons, labels, and forms are spilled all over them. The more UI details your scenarios have, the harder it is to spot the essence of what is it you're actually trying to demonstrate here with this example—a product discount calculation.

Getting out of Jail

Imagine you're one of hundreds of teams across the world who use Behat. Let's also assume you were working on your project for quite some time and now you have hundreds of UI-obsessed scenarios and a test suite which takes 30 minutes or more to execute and is constantly broken because the Selenium is not as stable as you expect it to be. How do you get out of this?

You, and many before you (including me), skipped the most crucial part of the tool usage—the set of principles it is built upon. Remember that definition of Behavior-driven Development I gave way back? Remember the conversation in the form of examples, devoid of any particular implementation as the central piece of the BDD puzzle? Well, guess what, Behat is a BDD tool and you skipped (as did I on multiple occasions) the most important part of BDD: exploration of the problem. It is time we pay our due!

The first step in getting out of mess is, unsurprisingly, the same for any kind of software design problem—refactoring, except this time we're not talking about refactoring of our code, we're talking about refactoring of our understanding. The first thing you do is you find the most important feature file you have at this particular moment and you finally start having the conversation with your stakeholders. You come to your business experts and ask them for help; say you feel you overcomplicated some of the crucial parts of the application and it is now required for you to understand what this feature actually means:

```
Scenario: Product costing less than ·10 results in \
delivery cost of ·3
  Given there is a product with SKU "RS1" and a cost of \
·5 in the catalog
  And I am on "/catalog"
  When I press "Add RS1 to the basket"
  And I follow "My Basket"
  Then I should see "Total price: ·9"
```

Liz Keogh, a fellow BDD practitioner, uses pixies as a way to remove the implementation detail from the scenarios. What if there were no backend, frontend, PHP, JavaScript or else behind the scene? What if all your feature did was to send a written request to a bunch of pixies and they tried to fulfil it as hard as they could without any technical skills whatsoever. If you look at every single feature in your application as a black box with a bunch of magic inside (including UI), you will find very quickly it isn't hard to focus on the essence of what business wants from your application. You will quickly find that what the business you serve cares about is not this:

```
Scenario: Product costing less than 10 results in \
delivery cost of 3
  Given there is a product with SKU "RS1" and a cost of \
5 in the catalog
  And I am on "/catalog"
  When I press "Add RS1 to the basket"
  And I follow "My Basket"
  Then I should see "Total price: 9"
```

It is actually closer to this:

```
Scenario: Product costing less than 10 results in \
delivery cost of 3
  Given there is a product with SKU "RS1" and a cost of \
5 in the catalog
  When I add the product with SKU "RS1" from the \
catalog to my basket
  Then the total price of my basket should be 9
```

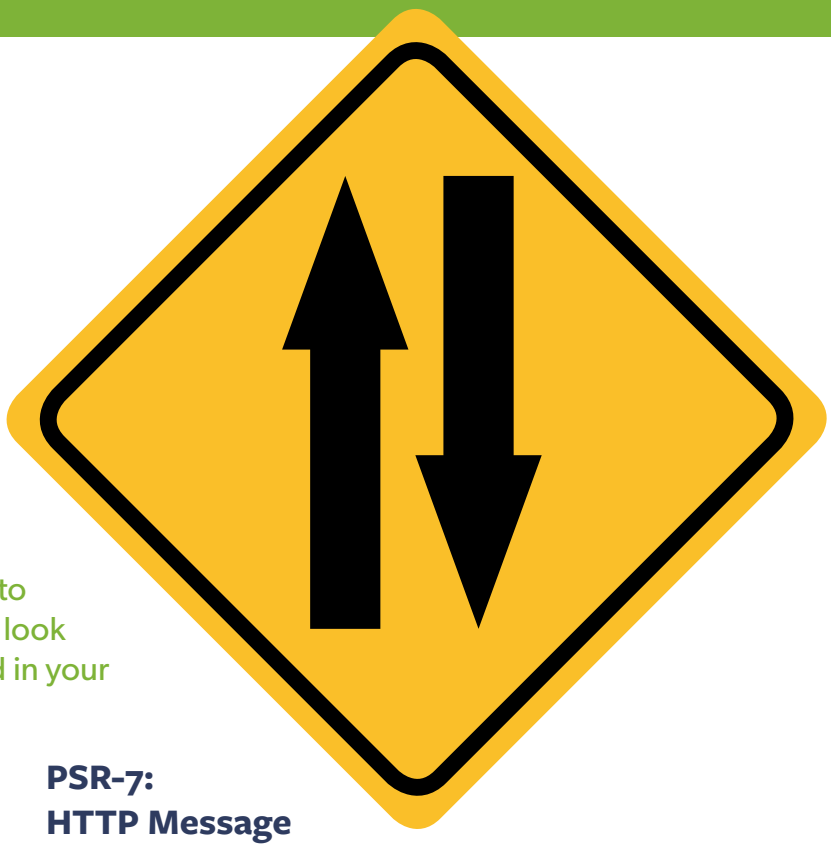
When going through this exercise together with your stakeholders, you quickly find the outcome is not a particularly better way to test your application or design your software. You will find the outcome is an approach which enables you to chose from more than one option. Do I hate browser automation in general or Mink in particular (even though I wrote the damn thing myself)? Of course I don't. What I do hate is the situation when the only option for your team going forward is the browser automation. Browser automation must be a deliberate and well-considered choice for every single scenario you develop. What it shouldn't be is the default for every single one of your tests.

When not blogging Konstantin Kudryashov is a prominent public speaker, organiser of BDD London meetups, the creator of Behat, Mink, co-creator of PhpSpec and leads the Behaviour-Driven Development (BDD) practice at Inviqa, a leading digital consultancy in London. As a communication coach, he has helped teams in many organisations bridge the gap between business and IT using Agile processes and development practices like Scrum, Kanban, BDD, TDD, Collaborative Product Ownership and Deliberate Discovery. @everzet

Abstracting HTTP Clients in PHP

David Buchmann

Many PHP applications are designed as server applications which receive a request and return a response to the client. However, PHP is also used to write API clients. This can be CLI tools written in PHP or server applications which need to talk to another system over HTTP. In this article, we'll look at a library to abstract out the HTTP client used in your code to keep it flexible and future-proof.



The PSR-7 standard[1] published by the Framework Interoperability Group (FIG) defines domain model for HTTP request and response. A standard for request and response is enough for server applications and middleware[2]. However, when we write code which acts as an HTTP *client*, we need to send a request. PSR-7 requests are value objects and can't send themselves—which is good, as they are also used to represent the request in a server application. There is no PSR for HTTP clients (yet).

HTTPlug[3] defines interfaces and behaviors for interoperable clients. Until we have a PSR defining HTTP clients, it's a good first step. Instead of writing code hard coupled to a specific client library like Guzzle[4], it can be written against HTTPlug and used with any supported client. You can use one client today with the knowledge if a better client comes along or your current one is no longer supported, you'll be able to easily switch it. This is valuable for all applications, and particularly useful for reusable libraries like general API clients which are meant to be integrated into applications. The HTTP client implementation should not be forced by a reusable library, but a choice of the integrator. HTTPlug is provided by the PHP HTTP group[5], along with additional tools. The organization consists of individuals interested in improving the state of HTTP support in PHP. At the time of writing, there are HTTPlug adapters for Guzzle 5 and 6, Buzz and React, as well as native cURL and socket clients.

1 *PSR-7 standard: http://www.php-fig.org/psr/psr-7/*

2 *Middleware is code which alters a request or response object and then passes it to the next layer.*

3 *HTTPlug: http://httplug.io*

4 *Guzzle: http://docs.guzzlephp.org*

5 *the PHP HTTP group: https://github.com/php-http*

PSR-7: HTTP Message Interfaces

PSR-7 defines interfaces for HTTP messages. Both requests and responses can have headers and a body. Requests additionally specify the requested URL and an HTTP method, like `GET`, `POST`, `PUT`, and so on. Responses have a status code. In addition to the messages themselves, PSR-7 also defines the `StreamInterface` to represent the (potentially huge) body, and the `UriInterface` to handle request URI information.

An important design decision is request and response objects are immutable. Methods to alter headers or add a body return an altered copy of the message, leaving the original message unchanged.

The full specification is available at http://www.php-fig.org/psr/psr-7/. For example, in the code below the `with·` methods return a copy of the original request.

```
$request = new GuzzleHttp\Psr7\Request();
$request = $request
   ->withMethod('GET')
   ->withUri($uri)
;

echo $request->getMethod(); // prints "GET"
```

Why Abstract Further?

PSR-7 by itself is a big help in interoperability. However, applications acting as HTTP clients need to create requests and send them with a client. There should be no hard coupling to specific PSR-7 and HTTP client implementations. This is even more important for reusable libraries. PHP does not support different parts of an application using different versions of the same library. Reusable libraries require a

specific major version of an HTTP client implementation or risk running into conflicts. Ideally, an application should only need to use one HTTP client implementation.

Most libraries should not need to know what client they are using. They need to send HTTP requests and receive responses. With the HTTPlug client interface, the library can state it needs an HTTP client without tying directly to a specific implementation. The only time specifics about the client configuration—like timeouts or special headers—have to be present on each request (e.g. API token) should be when creating the HTTP client. This should be handled in the bootstrapping part of the application and when the client is injected into the library. Bootstrapping is discussed in detail later in this article.

The HTTPlug Client

When we write PHP server applications, we see HTTP as:

1. Receive a request,
2. return a response.

Client applications, on the other hand, perceive HTTP as:

1. Send a request,
2. get a response back.

Clients could be CLI tools written in PHP or part of a server application that needs to send requests to other systems.

To send a request, we need a client. Instead of hard coupling our code to a specific client implementation or even raw cURL functions, we can use the HTTPlug interface. The HTTPlug interface consists of a single method:

```
sendRequest(RequestInterface \$request)
```

This method accepts any PSR-7 request and returns a PSR-7 response. It does not get much simpler!

Support for Asynchronous Requests

There is also an interface for asynchronous clients. Its method `sendAsyncRequest` returns a `Promise` which will eventually contain the response or an exception. This behavior is defined specifically for HTTPlug as PHP has no built-in support for promises and the PSR has not yet released their standard.

The HTTPlug promise has a `then()` method which accepts two callbacks. One of them will be executed at some point in the future. This allows your application to continue while a request is being sent, for example to send several asynchronous requests or perform other operations.

The first callback is called with the response, once it arrives. The second callback is called with an exception if and when the client throws an exception instead of returning a response.

It is important to note if the request has not yet been sent when the PHP process terminates, the requests will never be sent and the callbacks will not be called. To avoid issues, you should call the `Promise::wait()` method that blocks until the promise resolves with either success or an error. When the wait method returns, you can be sure one of the two callbacks has been executed. If it is important that the requests are handled, the strategy is to collect all promises and wait for each promise before exiting the application. On the console, that would be at the very end of the script; in web applications it is ideally *after* the response has been sent to the client. In Symfony, it would be an event listener which triggers on kernel.terminate. Listing 1 shows sending a request asynchronously and specifying the callbacks.

LISTING 1

```
$promise = $httpAsyncClient->sendAsyncRequest($request);
$promise->then(function (ResponseInterface $response) {
    // onFulfilled callback
    echo 'The response is available';

    return $response;
}, function (Exception $e) {
    // onRejected callback
    echo 'An error happens';

    throw $e;
});
// ...
$promise->wait();
```

Factories for PSR-7

If we do not want to bind the client application to a specific message implementation, we need a message factory which can create requests. The PHP-HTTP organization defines factories for messages, streams, and URIs. The package PHP-HTTP/message[6] provides factories for the Guzzle and Zend Diactoros PSR-7 implementations. Bootstrapping aside, the implementation is no longer coupled to a specific implementation. Listing 2 is an example usage of the PSR-7 factories.

LISTING 2

```
<?php
use Http\Message\MessageFactory\DiactorosMessageFactory;
use Http\Message\StreamFactory\GuzzleStreamFactory;

$messageFactory = new DiactorosMessageFactory();
$streamFactory = new GuzzleStreamFactory();
// ...
$request = $messageFactory->createRequest(
    'GET', 'http://example.com'
);
$stream = $streamFactory->createStream('stream content');
$request = $request->withBody($stream);
```

Middleware

With PSR-7 setting a standard for requests and responses, middleware becomes easy to implement. From an HTTP client perspective, Middleware can change a request before it is sent out, or alter the response before it goes back to the application. It is usually implemented as a chain of logic which passes the request on and returns the response. From

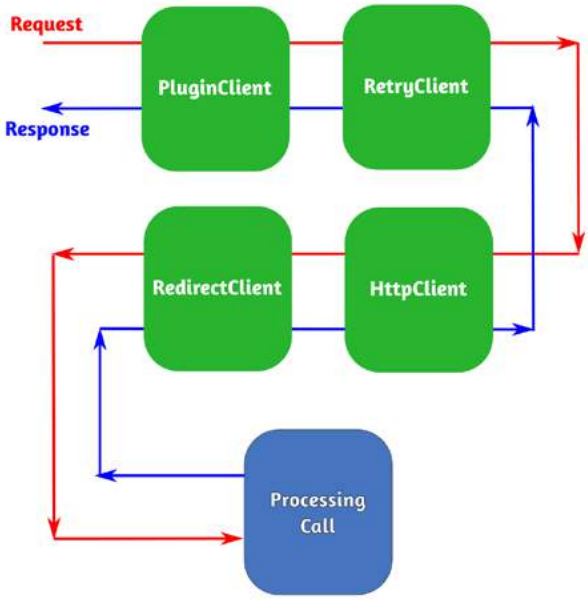
6 *PHP-HTTP/message:* http://docs.php-http.org/en/latest/message.html

the consumer point of view, middleware and the actual client do the same; they transform a request into a response. The difference is in the implementation—a middleware acts in the chain and forwards the request, while a client actually knows how to send the request to the server.

> *"From the consumer point of view, middleware and HTTP client transform a request into a response."*

A middleware could, for example, implement HTTP caching and check if we have a valid cached response for a request, and return the cached response instead of continuing the middleware chain. If no cache is found the middleware continues the chain and looks at the response to decide if the response can be added to the cache for the future.

Figure 1



As this example illustrates, the order of middlewares matters: those coming before the cache will already have altered the request. The changes they do to the response will be applied to cache hits as well. Middleware which comes after the cache is not executed on a cache hit. On the other hand, changes to the response done by middleware between the cache and the actual client get stored in the cache. Imagine a middleware which transforms HTTP status codes into domain exceptions—if this middleware is put at the start of the chain, the error could be stored in the cache and cache hits on the error response would still be transformed into exceptions. If the exception is thrown after the caching plugin, caching will not be attempted as the exception aborts the chain and will go straight to the library.

In HTTPlug, middleware is used with the PluginClient class that decorates a client and applies the plugins before forwarding to the client. As the order of plugins is relevant, plugins can only be set in the constructor of the PluginClient.

Using HTTPlug in Your Application

Let us build a simple application which uses HTTPlug to load the homepage of www.phparch.com.

Installation

HTTPlug is an abstraction from client implementation, but in the end your application will need a concrete client. In our example, we're using Guzzle 6. In an empty folder, run the following commands and choose to not select dependencies interactively:

```
composer init
composer require php-http/guzzle6-adapter php-http/message
```

Using HTTPlug

Once Composer has installed our dependencies, the smallest possible application looks as follows:

LISTING 3

```
01. #!/usr/bin/env php
02. <?php
03.
04. use Http\Adapter\Guzzle6\Client;
05. use Http\Message\MessageFactory\GuzzleMessageFactory;
06. require 'vendor/autoload.php';
07.
08. // HTTP implementation specific bootstrap code
09. $httpClient = new Client();
10. $messageFactory = new GuzzleMessageFactory();
11. // HTTP client agnostic application
12. $request = $messageFactory->createRequest(
13.     'GET', 'https://www.phparch.com/'
14. );
15. $response = $httpClient->sendRequest($request);
16. echo substr($response->getBody(), 0, 600) . '...';
```

If you invoke the script at the command line, you'll see output similar to:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">

<head profile="http://gmpg.org/xfn/11">
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />
    <meta name="viewport" content="initial-scale=1,
          maximum-scale=1">
    <title>php[architect] &#8211; Magazine, Training,
           Books, Conferences</title>

    <link rel="shortcut icon" ...
```

Only the bootstrapping code needs to know which implementation is used. The rest of the application doesn't change when we need to adjust the client configuration or switch to a different implementation.

Let's add a plugin to the client to send an API key. First, we need to install the plugin decorator:

```
composer require php-http/client-common
```

With the code installed, we can adjust our bootstrap from the above code example to use the plugin:

```
$httpClient = new Http\Adapter\Guzzle6\Client();
$httpClient = new Http\Client\Common\PluginClient(
   $httpClient, [
       new Http\Client\Common\Plugin\HeaderSetPlugin(
          ['API-Key' => 'my-api-key']
       ),
]);
```

The application itself does not need to change; it will continue to talk to an object which implements `HttpClient`.

As a last and slightly more complicated example, we can cache responses. The cache plugin is in a separate package because it has an additional dependency on PSR-6[7]—the caching standard—and we also need a cache implementation:

```
composer require php-http/cache-plugin cache/filesys-
tem-adapter
```

The bootstrap for caching consists in setting up Flysystem and telling it the root folder of the cache, see Listing 4.

LISTING 4

```
01. use Cache\Adapter\Filesystem\FilesystemCachePool;
02. use Http\Client\Common\PluginClient;
03. use Http\Client\Common\Plugin\HeaderSetPlugin;
04. use Http\Client\Common\Plugin\CachePlugin;
05.
06. $filesystem = new FilesystemCachePool(
07.     new League\Flysystem\Filesystem(
08.         new League\Flysystem\Adapter\Local(
09.           sys_get_temp_dir() . DIRECTORY_SEPARATOR . 'phparch'
10.         )
11.     )
12. );
13. $cacheOpts = [
14.     'respect_cache_headers' => false,
15.     'default_ttl' => 60,
16. ];
17.
18. $httpClient = new PluginClient($httpClient, [
19.     new HeaderSetPlugin(['API-Key' => 'my-api-key']),
20.     new CachePlugin($filesystem, $streamFactory, $cacheOpts),
21. ]);
```

By default, the cache plugin respects the Cache-Control headers sent by the server. But the homepage of phparch.com has headers that forbid to cache. For the sake of the example, we configured the plugin to ignore the caching instructions from the server, and cache everything for one minute (60 seconds).

Writing an HTTP Client Agnostic Library

When writing a reusable library which does HTTP requests, the bootstrapping should not be in the library, to be completely implementation agnostic. This means that you also don't require a specific client implementation in your composer file, but only HTTPlug and the virtual package php-http/client-implementation:

Listing 5 is a `composer.json` for a HTTP client agnostic library. It is then up to the application consuming your library to chose which HTTPlug client implementation to use.

LISTING 5

```
01. {
02.     "require": {
03.         "php-http/client-implementation": "^1.0",
04.         "php-http/httplug": "^1.0",
05.         "php-http/message-factory": "^1.0"
06.     },
07.     "require-dev": {
08.         "php-http/mock-client": "^0.3",
09.         "guzzlehttp/psr7": "^1.0"
10.     }
11. }
```

Bootstrapping a Shared Library

In recent years, Inversion of Control (e.g. with Symfony 2 using the Dependency Injection pattern) has become popular in the PHP world. HTTPlug and the PSR-7 factories work very well with code built for Dependency Injection (DI). To use inversion of control, have your classes require the HTTP client and factories in their constructor. Then your code does not need to know which factories to instantiate or how to build the client. See Listing 6 for a constructor of a class which needs to send HTTP messages.

LISTING 6

```
01. <?php
02.
03. use Http\Client\HttpClient;
04. use Http\Message\MessageFactory;
05.
06. class ApiClient
07. {
08.     public function __construct(
09.         HttpClient $httpClient,
10.         MessageFactory $messageFactory
11.     ) {
12.         $this->httpClient = $httpClient;
13.         $this->messageFactory = $messageFactory;
14.     }
15.
16.     // ...
17. }
```

7 *PSR-6: http://www.php-fig.org/psr/psr-6/*

Discovery

In addition to DI, the PHP-HTTP/discovery[8] package provides static discovery functions to find a currently available implementation of the client or, of factories. This is mainly useful for a simple start with minimal configuration. It is recommended to only use discovery as a fallback, and always allow the user of the library to explicitly inject the required client and factories. Listing 7 shows a constructor with optional zero-config thanks to discovery.

LISTING 7

```
01. <?php
02. use Http\Client\HttpClient;
03. use Http\Discovery\HttpClientDiscovery;
04. use Http\Discovery\MessageFactoryDiscovery;
05. use Http\Message\MessageFactory;
06.
07. class ApiClient
08. {
09.     public function __construct(
10.         HttpClient $httpClient = null,
11.         MessageFactory $messageFactory = null
12.     ) {
13.         $this->httpClient = $httpClient ?:
14.                             HttpClientDiscovery::find();
15.         $this->messageFactory = $messageFactory ?:
16.                                 MessageFactoryDiscovery::find();
17.     }
18. }
```

Provide an HTTP Client Factory

If a library needs a specifically set up client, e.g. with plugins, the best way is to provide a factory. Together with discovery, you can keep the base client an optional argument allowing your users to fine tune the client to their needs if needed. Refer to Listing 8 for an example client factory which sets the host and accepts additional plugins.

When using an HTTP client factory, make the `$httpClient` constructor argument of the API client class required. This allows your users to wire factory and API client together.

Outlook

This article discussed decoupling code from HTTP client implementations. Ideally, these interfaces would not be provided by a particular group, but exist as PSR standards. There is hope; PSR-15[9], currently in draft status, attempts to standardize middleware on the server side. Once a PSR for promises is implemented, a PSR for client side middleware can be implemented to replace the PHP-HTTP Plugin interface. For message factories, there is PSR-17, also currently a draft. Message implementations will likely provide factories that implement the PSR-17 interfaces and the message factory part of PHP-HTTP can be retired.

8 *PHP-HTTP/discovery:* http://docs.php-http.org/en/latest/discovery.html

9 *PSR-15:* http://www.php-fig.org/psr/

LISTING 8

```
01. <?php
02.
03. class HttpClientFactory
04. {
05.     public static function createClient(
06.         string $host,
07.         array $plugins = [],
08.         HttpClient $httpClient = null
09.     ) {
10.         $host = UriFactoryDiscovery::find()->createUri($host);
11.         if (!$host->getHost()) {
12.             throw new \InvalidArgumentException(sprintf(
13.                 'server uri must specify the host: "%s"',
14.                 $host
15.             ));
16.         }
17.
18.         $plugins[] = new AddHostPlugin($host);
19.
20.         if (!$httpClient) {
21.             $httpClient = HttpClientDiscovery::find();
22.         }
23.
24.         return new PluginClient($httpClient, $plugins);
25.     }
26. }
```

For the client interface, the PHP-HTTP group is preparing to propose a PSR. This would enable clients to implement the interface directly instead of the adapters PHP-HTTP now provides.

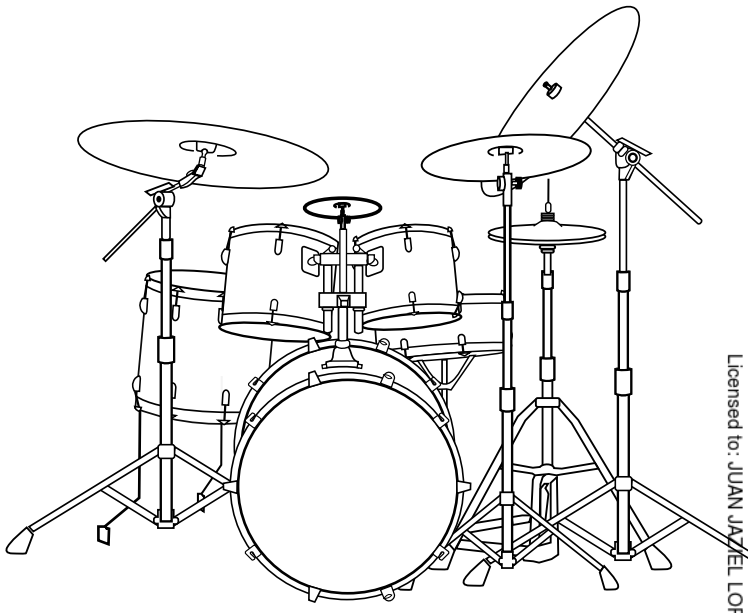
Even without these PSRs, there are early adopters using PHP-HTTP to decouple their libraries from the HTTP client: Payum, Geocoder, Mailgun, the KNPLabs GitHub client, the Happyr LinkedIn client and others.

If you take away one thing from this article, it should be that API clients should not be coupled to a specific HTTP client implementation. HTTPlug is the best we currently have to decouple the client. Once the whole topic is covered by PSRs, migrating to the PSR standards should be easier if you start with HTTPlug instead of coupling to a specific client implementation.

David Buchmann works at Liip AG as Symfony expert. He is maintaining the Symfony Content Management Framework, co-author of the FOSHttpCacheBundle and active with the PHP-HTTP HTTPlug client abstraction. When he is not coding, he enjoys travelling with his girlfriend, music and boardgames. @dbu

Strangler Pattern, Part Three: the Rhythm of Test-Driven Development

Edward Barnard



Test-driven Development (TDD) would seem to be all about the tests. But if you focus on writing the tests, you miss most of the value. In my experience, TDD can take two to three times longer in initial development time. The most-often-stated value comes from the dramatically-reduced debugging time needed with production deployment.

However, TDD's greatest value comes in the future. Software must remain agile, open to change. The Rhythm of TDD, often called "red-green-refactor," provides that value. Let's introduce Test-driven Development with a focus on building future value.

The Strangler Pattern, Part One[1] laid out our design approach. Automated tests play well with this approach:

1. Identify those parts of *new* feature work which can be offloaded from our web servers.
2. Implement the offloaded portions using the latest PHP and CakePHP available.
3. Integrate these offloaded portions in such a way we can evolve with newer PHP and CakePHP versions.

The automated tests (which do not yet exist) play a crucial role in fulfilling the third objective. As we update libraries, packages, frameworks, extensions, and so on, our code can break. When we have extensive coverage of *our* code base via automated tests we carry far higher confidence in rapid upgrades. We can *know* stuff continues to work.

Greenfield Development

> *In many disciplines a* **greenfield project**[2] *is one which lacks constraints imposed by prior work. The analogy is to that of construction on greenfield land where there is no need to work within the constraints of existing buildings or infrastructure.*

When we created our *Strangler Pattern* approach, we carefully created the ideal conditions for Test-driven Development (TDD). We have a *greenfield* project. We're creating new code from scratch. We need not worry about trying to retrofit unit tests to our existing code base. PhpStorm, CakePHP, and PHP all support unit tests via PHPUnit. We have everything we need.

Part Two: Beginning to Design for Scale With RabbitMQ[3] describes our intended structure and data flow. We are creating our *BATS (Batch System)* infrastructure. We have our requirements; we can start coding.

The question is, what do we start coding first? This is a relatively large project. We need *all* the pieces before we can run any actual work through production. In particular, we want to foster trust by having things go smoothly when they hit production.

You've surely heard the advice to tackle the hardest parts first, and the rest will follow. With TDD we take the opposite approach. Take on the *easy* pieces first. Your automated tests covering these easy cases become the foundation for gradually working your way to the more difficult or complex pieces. You'll be able to tackle that complexity with confidence knowing the underlying code, already under test, is solid.

1 *The Strangler Pattern, Part One:* https://www.phparch.com/magazine/2016-2/october/

2 **greenfield project**: https://en.wikipedia.org/wiki/Greenfield_project

3 *Part Two: Beginning to Design for Scale With RabbitMQ:* https://www.phparch.com/magazine/2016-2/november

Generating Tokens

The *Correlation Identifier*[4] or Correlation ID[5] is central to our design and indeed to many messaging systems. While there are many packages around which generate unique identifiers, I'm already creating cryptographically secure tokens from a previous project.

Given that BATS is running on PHP 5.6, we can use Paragon Initiative's `random_compat` library[6]. We are creating a separate package which can then be included in our various projects via Composer[7]. Finally, The League of Extraordinary Packages[8] maintains a skeleton project to use as a starting point for their packages. We'll use it as our starting point as well.

1. Clone the skeleton package:
   `git clone git@github.com:thephpleague/ \ skeleton.git`.
2. Rename the folder:
   `mv skeleton DemoGenerateToken`.
3. Tell PhpStorm to "create a new project from existing files."
4. Run the `prefill.php` script to customize your packages with author name, etc.: `php prefill.php`.
5. Remove the `prefill.php` script as instructed.
6. Run `composer update` to bring in PHPUnit and other dependencies.
7. Get a clean run from PHPUnit. In my case, to run it on my Mac, I invoke it via PhpStorm with a special `php.ini` file to load the `xdebug` extension.
   `vendor/bin/phpunit`

Your initial run should report `OK (1 test, 1 assertion).`

You should have a source file `src/SkeletonClass.php` which starts with the following code.

```
<?php
namespace ewbarnard\demo_generate_token;
class SkeletonClass { /* ...stuff... /* }
```

Your namespace will be different based on your `prefill.php` customization.

You should have a test file `tests/ExampleTest.php` which contains:

```
<?php
Namespace League\Skeleton;
Class ExampleTest extends \PHPUnit_Framework_TestCase
{ /* ... */ }
```

You can delete both of those files, but take note of the namespace and the class extended.

4 *Correlation Identifier:* https://www.amazon.com/dp/0321200683/

5 *Hohpe p. 163*

6 `random_compat` *library:* https://github.com/paragonie/random_compat

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

8 *The League of Extraordinary Packages:* http://thephpleague.com

The TDD Microcycle

James Grenning explains in *Test-Driven Development for Embedded C*[9]

> *At the core of TDD is a repeating cycle of small steps known as the TDD microcycle. Each pass through the cycle provides feedback answering the question, does the new and old code behave as expected? The feedback feels good. Progress is concrete. Progress is measurable. Mistakes are obvious.*[10]:

Grenning explains the microcycle:

1. Add a small test.
2. Run all the tests and see the new one fail, maybe not even compile.
3. Make the small changes needed to pass the test.
4. Run all the tests and see the new one pass.
5. Refactor to remove duplication and improve expressiveness.

Learning Tests

We plan to use the `random_compat` library. Let's start by writing one or more tests which ensure the library works as expected. Grenning[11] calls these *learning tests*. We write tests which exercise the library the way we intend to use it.

Learning tests provide long-term memory. Picture yourself coming back to the project months later. You don't need to learn the library again from scratch. Instead, follow your thought process by reviewing (and running) the learning tests.

These tests will have an important role in accepting new releases of library code. When your tests cover how you use the library, then any change in interface or behavior will show itself. Your tests will immediately inform you of the incompatibility.

The `random_compat` online documentation shows our use case:

```
$string = random_bytes(32);
var_dump(bin2hex($string));
```

We need 128 bits of entropy, which is 16 bytes. `bin2hex()` of the result should give us 32 characters (2 characters per byte). That can be our test: verify we get 32 hex characters.

Install `random_compat` as instructed:

```
composer require paragonie/random_compat
```

9 *Test-Driven Development for Embedded C:* https://www.amazon.com/dp/193435662X

10 *Grenning p. 7*

11 *Grenning p. 291*

Create a new test file `tests/LearningTest.php`. Use our primary namespace and extend `PHPUnit_Framework_Testcase`:

```
<?php
namespace ewbarnard\demo_generate_token;

class LearningTest extends \PHPUnit_Framework_TestCase
{

}
```

Since this is a new setup, we first need to ensure we can write a test which runs. Create a test method in `LearningTest` that fails:

```
public function testHere() {
   static::fail('Line ' . __LINE__);
}
```

Running the test produces:

```
There was 1 failure:

1) ewbarnard\demo_generate_token\LearningTest::testHere
Line 15
FAILURES!
Tests: 2, Assertions: 1, Failures: 1.
```

Now, let's make sure we can write a passing test. Change `static::fail(...);` to `static::assertTrue(true);`. Re-run our tests:

```
OK (2 tests, 2 assertions)
```

Success! Let's now write the test to ensure random_bytes(16) can be converted to exactly 32 hex characters. We don't have any way for this to fail, so we'll place a `static::fail()` call right after to ensure the code was executed.

Always watch your test fail first. I won't tell you how many times I've chased a bug assuming a test was passing, only to later figure out the test never ran. If the test *had* run, it would have told me the problem immediately. *Always watch your test fail.*

```
public function test16bytes() {
   $string = random_bytes(16);
   $hex = bin2hex($string);
   $actualLength = strlen($hex);
   static::assertSame(32, $actualLength);
   static::fail('fixme');
 }
```

As expected:

```
There was 1 failure:

1) ewbarnard\demo_generate_token\LearningTest::test16bytes
fixme
```

Fix the test by deleting the last line. Run the tests:

```
OK (3 tests, 3 assertions)
```

Notice, we went from two tests to three tests. Delete the ExampleTest, and we'll be down to our own "2 tests, 2 assertions."

Real Tests

What tests do we need to write? I generally work from the outside in:

1. Exercise the constructor and any constructor edge cases.
2. Exercise the API and return values. Exercise API boundary values, types, return types, etc.
3. Dive deeper into the internals of the item being tested.

Our token generator is simple. Here is my list of tests:

- `GenerateToken::token()` returns a string (we have hereby declared what the API should look like).
- The return value has a string length of 22.
- The return value does not contain +, /, or =.
- Two consecutive calls do not return the same value.

Create the new class `GenerateTokenTest` with its first test:

```
<?php
namespace ewbarnard\demo_generate_token;
class GenerateTokenTest extends \PHPUnit_Framework_TestCase
{
   public function testResultString() {
      $result = GenerateToken::token();
      $string = (string)$result;
      static::assertSame($string, $result);
   }
}
```

As expected, the tests blow up with:

```
PHP Fatal error:  Class 'ewbarnard\demo_generate_token\
GenerateToken' not found
```

Create the class with static function `token()`:

```
<?php
namespace ewbarnard\demo_generate_token;
class GenerateToken {
   public static function token() {
   }
}
```

Now we have a new failure:

```
There was 1 failure:

1) ewbarnard\demo_generate_token\GenerateTo-
kenTest::testResultString
Failed asserting that null is identical to ''.
```

This makes sense. We're not yet returning a string. Our mission, now, is to make the *smallest* possible change which will cause the test to pass. Here is our new `token()` function:

```
public static function token() {
    return '';
}
```

Success: `OK (3 tests, 3 assertions)`. I'm sure this code bothers you. It's not right! That's okay. Our later tests will force us to fill in the correct code. What's important to realize here is *the test is correct.*

- Our specification states the token generator is to return a string.
- We demonstrated the test fails when the token

generator returns null.

- We observed the test passing when we return a string.

To be sure, it's an empty string, but we've nailed down that portion of the specification. If future development were to cause the function to return anything other than a string, this test will start failing again. As long as we run our tests and they are passing, we can trust this function returns a string.

With TDD, a large number of very small wins like this add up to an enormous future value.

Do It Again

That was easy! And that, really, was the point. Let's complete our package development with the remaining tests.

```
public function testResultLength22() {
   $result = GenerateToken::token();
   static::assertSame(22, strlen($result));
}
```

Run our tests. We have a failure: `Failed asserting that 0 is identical to 22.` Let's make the smallest change which could cause this test to pass:

```
public static function token() {
   return str_repeat('x', 22);
}
```

Yes, we "cheated." But the tests pass: `OK (4 tests, 4 assertions)`. We have additional tests on our list which will force us to write real code.

Our next test checks for specific characters. We'll use PHPUnit's data provider to test each of those characters one at a time. We don't want one failure to cover up another failure. First, while our current tests are passing, let's add the failing characters to our token generator. Our current tests should continue to pass. We can write the new test which detects the problem and fails.

Our new code:

```
public static function token() {
   return '=+/' . str_repeat('x', 19);
}
```

All tests pass. Let's write a failing test:

```
/**
 * @dataProvider dataBadCharacters
 */
public function testResultBadCharacters($bad) {
   $result = GenerateToken::token();
   static::assertNotContains($bad, $result);
}

public function dataBadCharacters() {
   $data = [];
   $data[] = ['+'];
   $data[] = ['/'];
   $data[] = ['='];
   return $data;
}
```

Running the tests gives us three failures, one for each of the "bad characters." With `@dataProvider`, PHPUnit runs each input set as a separate test. That is, the test *code* is the same, but it has different inputs each run, and thus it is exercising a different *test case* each time.

It's time to write the production code:

```
public static function token() {
   $raw = random_bytes(16);
   $string = base64_encode($raw);
   $clean = str_replace(['+', '/'], ['-', '_'], $string);
   return substr($clean, 0, 22);
}
```

All tests pass: `OK (7 tests, 7 assertions)`. We got it! Now, with all of the tests passing, we can refactor the code to clean it up and improve expressiveness. Let's remove the three intermediate variables:

```
public static function token() {
   return substr(
      str_replace(['+', '/'], ['-', '_'],
                  base64_encode(random_bytes(16))), 0, 22
   );
}
```

All tests continue to pass. We have one test still to write. We need to ensure two calls don't generate the same token. We know this should never happen. But how do we know the library won't break and start sending continuous zeroes, for example? It will only take a moment; let's make sure two tokens are not the same.

```
public function testGenerateDifferent() {
   $result1 = GenerateToken::token();
   $result2 = GenerateToken::token();
   static::assertSame($result1, $result2);
}
```

The above test fails as expected. If you change `assertSame` to the correct `assertNotSame` and all tests pass.

Future Value

I've seen the opinions that "TDD is dead," and "TDD only works on trivial cases." For sure, the example we just wrote is a trivial case. It *is*, however, production BATS code. I don't write all code using TDD. I have criteria which I'll share in both this article and the next in this series.

TDD does have tremendous value in helping you get your code right. It's great at helping keep you from introducing bugs in the first place. But we're here to look at the *future* value of your current Test-driven Development.

Let's take another look at our *Strangler Pattern* design principles with an eye on where TDD can bring us a significant future value.

Single Responsibility Principle

"Uncle Bob" Martin restates his *Single Responsibility Principle*[12]:

> *Gather together the things that change for the same reasons. Separate those things that change for different reasons.*
>
> *However, as you think about this principle, remember that the reasons for change are people. It is people who request changes. And you don't want to confuse those people, or yourself, by mixing together the code that*

12 *Single Responsibility Principle:* http://phpa.me/8th-light-srp

many different people care about for different reasons.

Legacy Software

Martin Fowler[13] notes:

> *Let's face it, all we are doing is writing tomorrow's legacy software today.*

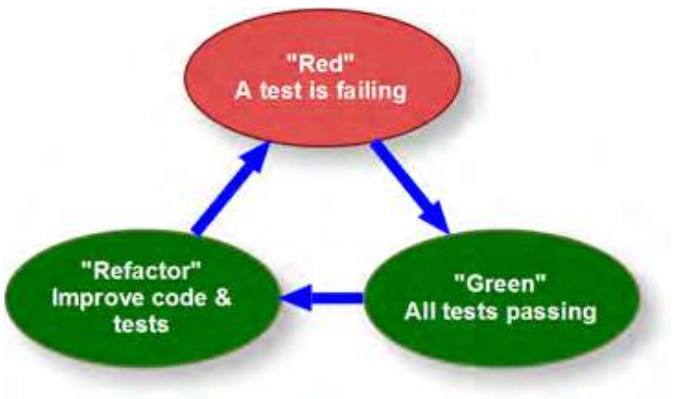
Michael Feathers in *Working Effectively with Legacy Code*[14] explains[15]:

> **Software Vise**
>
> **vise** *[n.]. A clamping device, usually consisting of two jaws closed or opened by a screw or lever, used in carpentry or metalworking to hold a piece in position. The American Heritage Dictionary of the English Language, Fourth Edition*
>
> *When we have tests that detect change, it is like having a vise around our code. The behavior of the code is fixed in place. When we make changes, we can know that we are changing only one piece of behavior at a time. In short, we're in control of our work.*

Red-Green-Refactor

You will hear TDD practitioners call the rhythm embodied by the microcycle *Red-Green-Refactor*. With Figure 1, Grenning explains [16]:

Figure 1. The Rhythm of TDD



> **Red-Green-Refactor and Pavlov's Programmer**
>
> *The rhythm of TDD is referred to as Red-Green-Refactor. Red-Green-Refactor comes from the Java world, where TDD practitioners use a unit test harness called JUnit that provides a graphical test result representation as a progress bar. A failing unit test turns the test progress bar red. The green bar is JUnit's way of saying all tests passing.*
>
> *Initially, new tests fail, resulting in an expected red bar and a feeling of being in control. Getting the new one to pass, without breaking any other test, results in a green bar. When expected, the green bar leaves you feeling good. When green happens and you expected red, something is wrong—maybe your test case or maybe just your expectation.*
>
> *With all tests passing, it is safe to refactor. An unexpected red bar during refactoring means behavior was not preserved, a mistake was detected, or a bug was prevented. Green is only a few undo operations away and a safe place to try to refactor from again.*

The Internet of Things in PHP

When I read *Building for the Internet of Things in PHP*[17] by Adam Englander, I found him twiddling GPIO pins (General Purpose I/O) on the Raspberry Pi and other microcontrollers. I had no idea it could be done in PHP!

That's the normal realm of "embedded" software development. With the "Internet of Things," embedded software development is coming back into fashion.

"Uncle Bob" Martin notes in his Foreword[18] that Grenning's book should be titled *Crafting Embedded Systems in C*. "This book provides a very complete and highly professional approach to engineering high-quality embedded software in C, quickly and reliably."

If you're looking into the Internet of Things, Grenning's book is your shortcut to solid embedded software engineering. It's also the best teaching of Test-driven Development I've ever encountered. He teaches TDD in the context of real-world software engineering. Most authors, naturally enough, focus on Test-driven Development as the topic in hand, whereas Grenning distils his 30+ years of experience with embedded software engineering into this book, thoroughly teaching and Test-driven Development along the way.

13 *Martin Fowler:* http://www.martinfowler.com/bliki/StranglerApplication.html

14 *Working Effectively with Legacy Code:* https://www.amazon.com/dp/0131177052

15 *Feathers, p. 10*

16 *Grenning, p. 9*

17 *Building for the Internet of Things in PHP:* https://www.phparch.com/magazine/2016-2/september/

18 *Grenning, p. xvi*

The Rhythm of Test-Driven Development

In my experience, ongoing web development usually consists of getting the next idea (a "feature") into production as quickly as possible. Also in my experience, TDD can take two to three times longer in initial development time. When both you and your client are focused on getting the feature into production as quickly as possible, nobody cares about the "future value" TDD allegedly brings you.

Furthermore, if your organization is not consistently running automated tests, you will *never* see the future value. I *always* saw automated tests running throughout the late 20th Century. I have *rarely* seen automated tests consistently run this century. Go figure.

In *Building Microservices*[19] Sam Newman writes[20]

> *Deploying a monolithic application is a fairly straightforward process. Microservices, with their interdependence, are a different kettle of fish altogether. If you don't approach deployment right, it's one of those areas where the complexity can make your life a misery.*

One foundation of Sam's microservices deployment strategy is automated testing. *Continuous Integration (CI)* includes running your automated tests. Test-driven Development provides both immediate and future value here. You are building your regression test suite as you build your feature.

That's great, but you're still under pressure to get that new feature into production *now*. Kent Beck promotes this motto[21]:

- Make it work.
- Make it right.
- Make it fast.

Under "as quickly as possible" pressure, we tend to stop with "make it work" and put the idea into production. Agile sprints and daily stand-up meetings tend to reinforce the problem. Product managers want to hear about "make it work." It makes little sense to explain you're still "working on it" when it already works. You're laying in the technical debt and moving on.

Do you see how TDD simplifies "make it right" and "make it fast?" When it works (all tests green), you can fearlessly "make it right." You don't need to re-test everything after you changed things. Just re-run your existing tests after each edit. In the same way, do whatever you need to do to make it fast. You have your "software vise" in place. As long as your tests continue to run green, you know your functionality remains intact.

Furthermore, if someone (probably your future self) accidentally breaks your feature, the existing tests will tell you. Your tests are preventing that future bug from ever happening.

One of our *Strangler Pattern* design objectives is to make the system as easy as possible for other developers to add new offline functionality. As I developed the BATS infrastructure, I found more and more code which should become library (composer package) code usable by all new BATS projects.

Since I was indulging in TDD, the continuous major restructuring was easy. The tests ensured I didn't break anything while moving things around. I did break things, many times, but a quick "undo" got me back to green.

Perspective

If you're not continuously running automated tests, it's difficult to see the full value of Test-driven Development. If you're not doing Test-driven Development, you likely don't have much in the way of automated tests to run.

You can build extensive unit test suites without doing TDD. That's great for *future* value but you've missed out on all of the *current* value Test-driven Development provides. Unfortunately, in my experience with such an environment, a third to a half of total development time is spent fixing now-broken tests.

Fragile Test

> *Past efforts at automated testing have often run afoul of the "four sensitivities" of automated tests. These sensitivities are what cause fully automated tests that previously passed to suddenly start failing… Fragile tests increase the cost of test maintenance by forcing us to visit many more tests each time we modify the functionality of the system or the fixture. They are particularly deadly when projects rely on incremental delivery, as in agile development.*

With Test-driven Development I no longer encounter the fragile test[22] problem[23]. This is anecdotal evidence to be sure, and it could be due to greater experience writing tests. I suspect there is a real reason, though. The test-first methodology provides a "laser focus" on precisely what is to be accomplished (and therefore tested).

If your organization is wallowing in the "fragile test" problem, you're unlikely to get excited about taking a deeper dive into "test first" rather than "test later." And, sure enough, as you begin learning Test-driven Development, you'll find initial development takes a *lot* longer. That is, it takes longer to "make it work" and neither "make it right," nor "make it fast" are part of the picture. Some TDD practitioners do report, with experience, TDD does reduce overall software development time. That's without even counting the future value of continuing to run those automated tests.

19 *Building Microservices:* https://www.amazon.com/dp/1491950358

20 *Newman, p. 103*

21 *Grenning, p. 267*

22 *fragile test:* https://www.amazon.com/dp/0131495054

23 *Meszaros, p. 239*

The BATS infrastructure itself is a good example of TDD. The project manager put me down for two weeks' development time. When I asked, in the daily Scrum meeting, I was told two weeks. I said it would take a month. Our CTO blinked and wrote down four weeks. This was a win, so I didn't press the difference between a month and four weeks. Nobody could understand *why* the project would take all of four weeks.

To be sure, I'd spent fewer than five seconds coming up with this detailed project time estimate. Having attended Jared Faris' *How long will it take? Estimation methods to answer the impossible question*[24]. I knew I was on solid ground.

Test-driven Development does help you be more consistently correct with time estimates. Given a clear picture of the software to be built, I visualize the flow of TDD microcycles and imagine how long that flow will take to reach the destination. Like a mountain stream, with experience the rate of flow becomes remarkably predictable. I need to allow for substantial integration time because this is our first time using either PHP 5.6 or RabbitMQ in production. But, I didn't bother to build in debugging time. I didn't need it, other than getting some MySQL table indexes correct (which was a feature using BATS, rather than BATS infrastructure).

BATS has proven to be solid and efficient in its first weeks of production. I know that's due to TDD, but I imagine our organization assumes it's simply due to Ed being a slow and careful writer of code. *Shhh! Don't tell! TDD is my secret weapon!*

24 *How long will it take? Estimation methods to answer the impossible question:* http://phpa.me/faris-midwest-estimation

Looking Ahead

Test-driven Development would be ultimately useless if it only applied to the most trivial of code. In the real world, of course, our code must work through other classes, functions, APIs, and databases. These dependencies become a formidable challenge to writing unit tests.

In practice, any time we write "real world" code it's easy to spend an hour getting structures set up for a three-line test. Things don't just get out of hand to the point where it's far wiser to just forget about the tests and write the code, hoping for the best.

In *Part Four: Unit Test Design With Mockery* we'll see BATS components efficiently developed and tested using Spies and Mock Objects. We'll demonstrate strategies to use in keeping your unit test development sane.

Part Five: Producer-Consumer Programming in CakePHP/RabbitMQ brings our case study full circle. We'll see a bit more code, and look at the radically different way of thinking which gets us there.

Ed Barnard began his career with CRAY-1 serial number 20, working with the operating system in assembly language. He's found that at some point code is code. The language and details don't matter. It's the craftsmanship that matters, and that craftsmanship comes from learning and teaching. He does PHP and MySQL for InboxDollars.com. @ewbarnard

Let's Build a Chatbot in PHP

Matthew Setter

In this, the final edition of Education Station for 2016, I'm not going to present a new service, library, or package. Instead, I'm going to celebrate the time Christmas should be; a time of relaxing, when we all have some extra time on our hands. And what, as developers, do we do when we have some time on our hands? We write code.

Also, as it's Christmas, I'm not going to be too serious. It's the festive, silly, season after all. I'm going to step you through a fun project I built a little while ago. A chatbot. That's right, a chatbot. What could be more fun than something you can, sort of, talk to; something you can ask a question of and receive an answer?

I'll be straight up; I'm no chatbot expert. I was put up to this by php[architect]'s Editor-in-chief, Oscar Merida. I was stuck for ideas and asked him what he thought might be something fun. He suggested a chatbot. And there I was, thinking I'd never actually considered building one before.

If you're in a similar position, of not having ever made one before, then it's going to be a fun experience for the both of us, I hope.

What Is a Chatbot?

As a bit of background, before we dive into the code, chatbots, according to an excellent article in Chatbots Magazine, *The Complete Beginner's Guide To Chatbots*[1] —yes, there's a magazine for them—are:

> *A service, powered by rules and sometimes artificial intelligence, that you interact with via a chat interface. The service could be any number of things, ranging from functional to fun, and it could live in any major chat product (Facebook Messenger, Slack, Telegram, Text Messages, etc.).*

So there's the core of chatbots. They're a service we interact with, to find out something we need to know or to help us do something we need to achieve—but there's no person at the other end. The article goes on to provide a series of chatbot suggestions, including:

- **A weather bot**, which gives the weather forecast.
- **A grocery bot**, which helps you pick out and order groceries for the week.
- **A news bot**, which you can ask about interesting, recent, news.

You can find the code for the chatbot in this article on GitHub: settermjd/chatbot[2],

What Chatbot Would I Write?

As I sat there, late one night, reading through the article, and feeling very inspired, I wondered what kind of chatbot I could create. As it would be my first, I didn't want to go overboard and attempt to do everything all at once. Instead, I wanted to take some baby steps and get my feet wet.

I looked at the service suggestions (*Messenger*, *Slack*, etc.) and pondered what might be a useful service I could develop, and whether I'd go down the route of question and answer or code in some AI.

After a little bit of thought, I decided to build a question and answer style chatbot, specifically for use with Slack, where I (or anyone else) could give it the name of a PHP function, and it would reply with some information about it.

I know, original, huh? But it's something I know about, and something I could, practically, use on a daily basis. With the basic concept in place, how would it work? Where would I get the information from?

I'd never integrated anything with Slack before. I figured they'd have detailed documentation about how to integrate with their service. It turns out they do, but you don't initially need it.

All you need to do, after you have an account[3], is to create what Slack calls a Slash Command[4].

1 *The Complete Beginner's Guide To Chatbots:* http://phpa.me/beginners-guide-chatbots

2 *settermjd/chatbot:* https://github.com/settermjd/chatbot

3 *after you have an account:* https://slack.com/create#email

4 *a Slash Command:* https://slack.com/apps/A0F82E8CA-slash-commands

Quoting the documentation, a slash command:

> *…allow you to listen for custom triggers in chat messages across all Slack channels. When a Slash Command is triggered, relevant data will be sent to an external URL in real-time.*
> *For example, typing* `/weather 94070` *could send a message to an external URL that would look up the current weather forecast for San Francisco by Zip Code and post it back to Slack.*

Sounds like what I need. To work with a slash command, what I specifically needed to do was:

1. Register a new slash command.
2. Write a chatbot (a service) for the slash command to send an event payload to.
3. Have my chatbot retrieve and format the necessary information and returns the result to Slack.

That's something I can do; so I got underway. From the slash commands page, I clicked "**Add Configuration**" and was shown a page to enter all the relevant details.

While there are many options, not all of them are required. You can see the mandatory ones in Figure 1, which are:

- **Command:** the name of the command a user will need to type, to use the service
- **Url:** the URL of the chatbot

Figure 1

Slack defaults to sending `POST` requests to the service, which is fine, and generates a token, which provides the ability to limit access to the service to only valid users.

The other things I did were:

- A custom icon
- Autocomplete help text
- A descriptive (command) label

The custom icon is used when providing the answer to the question. Naturally, I chose the PHP Elephant! I set the autocomplete help text to "**Get a PHP function's description (`mysql_query`)**", and the descriptive label to: "**Find the description of a PHP command from the official PHP manual**".

If someone were using the service for the first time, then they'd better understand what it was about. With that done, it was time to code the chatbot.

Coding the Chatbot

The first question was, where would I get the information. The simplest and best, place I could think of was the PHP manual itself. It's the source of truth for all things PHP, after all.

I'm not aware of any API, so I decided to create a simple web scraper to get the information required. It turns out, given the site's construction it is quite straightforward to locate details about the command, including:

- The version of PHP which introduced the command
- The command synopsis
- The command description

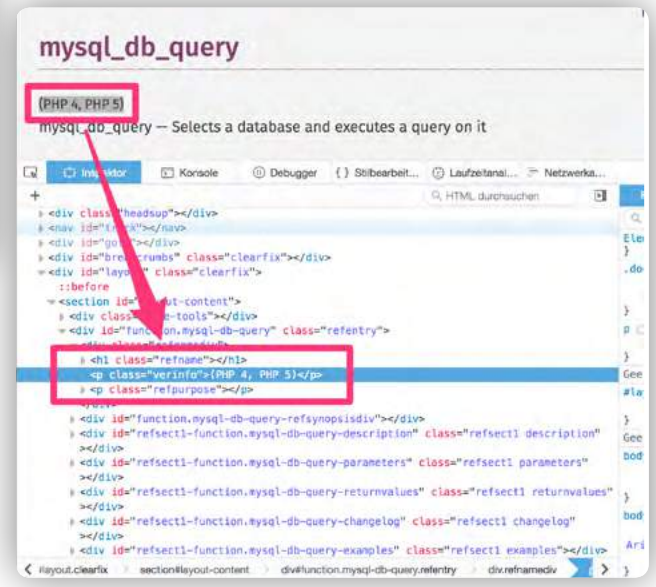
But what information should the service display? There's not a lot of space available in a Slack input field. Well, let's start with the command's description.

Next question: "**How can that information be programmatically extracted**?"

If you're familiar with using XPath expressions[5], which are rather like SQL for HTML documents, it can be done quite intuitively.

In Figure 2, you can see I have PHP's `mysql-db-query` command loaded in Firefox, and in the developer tools, I've highlighted the version information element.

Figure 2



The XPath query `//p[@class="verinfo"]` would extract this information from the page for us. This query would search the document looking for a `p` tag which has a class attribute set to `verinfo`.

5 *XPath expressions:* http://phpa.me/xpath-examples

Sadly, there's not enough space to go into any real depth on XPath expressions Sadly, there's not enough space to go into any real depth on XPath expressions here. But it's a fascinating topic because XPath expressions save so much work; especially if you consider how much code you'd have to write using string parsing functions.

Given I'm a big fan of Composer, I did a quick search on Packagist for a package which provides support for executing XPath expressions on HTML content and found the DomCrawler component[6], from the Symfony team.

With the ability to extract content from an HTML page taken care of by DomCrawler how was I going to get the page contents in the first place?

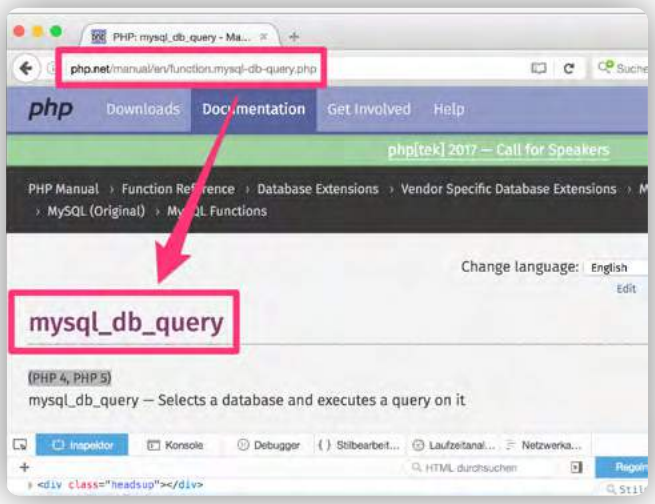
One package sprung straight to mind—Guzzle[7]. It's fair to say it's the go-to package for remote requests in PHP, a lot like Composer is for package management.

The User Interface

I have a slash command setup and have packages for retrieving and filtering a page in the PHP manual. What next? Well, what would the user need to enter as command text and how would that translate to a page in the manual?

I guess you may have, but the URL in the PHP manual for a command always converts underscores in the command name to hyphens, as you can see in Figure 3.

Figure 3



As an example, say you wanted to retrieve details on `mysql_drop_db`, well the URI would be:

http://php.net/manual/en/function.mysql-drop-db.php.

If you wanted to retrieve the details of `mysql_fetch_field`, then the URI would be:

http://php.net/manual/en/function.mysql-fetch-field.php.

Given that consistency, finding the URI for the command should only require doing some quick string replacement and string interpolation. We now have the minimum requirements for building the chatbot. So let's see some code.

The Chatbot Code

In Listing 1, we're setting up the core class, `PhpManualChatBot`, giving it a `Guzzle` client and a `DomCrawler` instance as constructor dependencies.

LISTING 1

```
01. <?php
02.
03. namespace ChatBot;
04.
05. use Guzzle\Http\Exception\ClientErrorResponseException;
06. use Guzzle\Service\ClientInterface;
07. use Symfony\Component\DomCrawler\Crawler;
08.
09. class PhpManualChatBot
10. {
11.     private $client;
12.     private $crawler;
13.
14.     public function __construct(
15.         ClientInterface $client,
16.         Crawler $crawler
17.     ) {
18.         $this->client = $client;
19.         $this->crawler = $crawler;
20.     }
```

In Listing 2, we define `lookupFunction()` which, as the name implies, looks up a function in the PHP manual. It does this by using Guzzle to make a request to the relevant function's page in the manual, interpolating the URI with the function name supplied.

LISTING 2

```
01.     public function lookupFunction($functionName)
02.     {
03.         try {
04.             $response = (
05.                 $this->client->get(
06.                     sprintf(
07.                         'http://php.net/manual/en/function.%s.php',
08.                         $functionName
09.                     )
10.                 )
11.             )->send();
12.         } catch (ClientErrorResponseException $e) {
13.             throw new FunctionNotFoundException(
14.                 sprintf(
15.                     "The function '%s' was not found",
16.                     $functionName
17.                 )
18.             );
19.         }
20.         $this->crawler->add($response->getBody(true));
21.
22.         return new PhpManualChatBotResponse(
23.             $this->crawler->filterXPath(
24.                 '//p[@class="para rdfs-comment"]'
25.             )->text()
26.         );
27.     }
28. }
```

6 *the DomCrawler component: http://phpa.me/sy2-dom-crawler*

7 *Guzzle: http://docs.guzzlephp.org/en/latest/*

If the page isn't available, it then catches the Guzzle `ClientErrorResponseException` which would be thrown and throws a more specific exception: `ClientErrorResponseException`. This isn't strictly necessary—and could be better implemented. But, for a simple example, it works.

Assuming the page does exist, then the body of the response is supplied to the DomCrawler's `add()` method. This gives the crawler the HTML content to search the call to its `filterXPath()` method. This method is passed the XPath query to use to locate the DOM element which contains the information which we want to extract.

I've not done any error handling should that section not exist. But if this code were to be improved, this would be a good place to start.

Assuming the XPath finds a match, then we can call the `text()` method on it, which will return the text located inside that match. Otherwise, an XPath object would be returned. All we need is the text. That text is then passed to the constructor of a new `PhpManualChatBotResponse` object, which you can see below.

The PhpManualChatBotResponse Object

`PhpManualChatBotResponse` is a value object which can store some key details about a retrieved PHP function. This isn't strictly necessary. But as I was developing the chatbot, using a value object seemed like a handy and efficient way of manipulating the information in the page.

Perhaps, at the start, all I wanted to do was to retrieve a description. Fine. I could just retrieve and return that. But what if, later on, I wanted to retrieve more information, such as the PHP version, function synopsis, and so on?

To me at least, it makes sense to factor some flexibility in at the start, even if it's not implemented. I say that in cautious tones. I don't mean to infer I'd code up a complex solution on the off chance it was used.

What I'm inferring here is that I'd structure what I did in such a way it was easy to expand later, yet without over-committing at this early, uncertain point. Enough with the qualifications; let's step through the value object.

First, we initialize the object, setting its `$methodDescription` member variable to the value of `$methodDescription` supplied, which is passed through the `filterParameter()` method. See Listing 3.

The `filterParameter()` method , shown in Listing 4, uses `Zend\InputFilter`'s `Input` and `InputFilter` classes to sanitize the information supplied, stripping new lines, trimming the string, and finally removing duplicate white spaces.

This is some elementary sanitization, added based on the information which was supplied from the calls to the manual during testing.

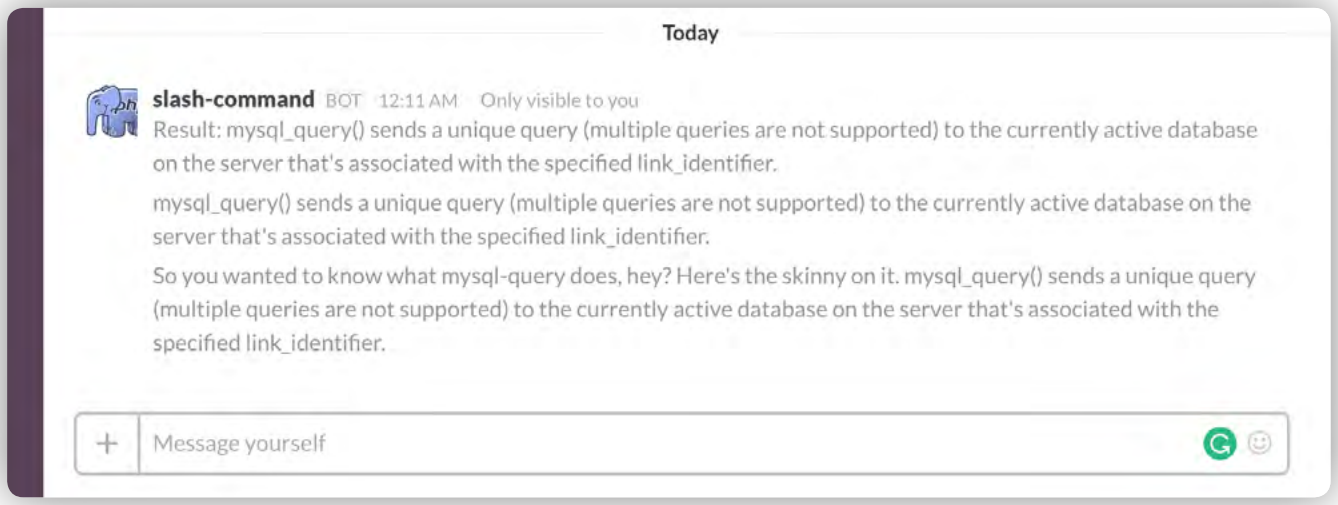
LISTING 3

```
01. <?php
02.
03. namespace ChatBot;
04.
05. use Zend\Filter\PregReplace;
06. use Zend\Filter\StringTrim;
07. use Zend\Filter\StripNewlines;
08. use Zend\InputFilter\Input;
09. use Zend\InputFilter\InputFilter;
10.
11. class PhpManualChatBotResponse
12. {
13.     private $methodDescription;
14.
15.     public function __construct(
16.         $methodDescription
17.     ) {
18.         $this->methodDescription = $this->filterParameter($methodDescription);
19.     }
```

LISTING 4

```
01.     private function filterParameter($parameter) {
02.         $input = new Input('parameter');
03.         $input->getFilterChain()
04.             ->attach(new StringTrim())
05.             ->attach(new StripNewlines())
06.             ->attach(new PregReplace([
07.                 'pattern' => '!\s+!',
08.                 'replacement' => ' '
09.             ]));
10.           $inputFilter = new InputFilter();
11.         $inputFilter->add($input);
12.         $inputFilter->setData(['parameter' => $parameter]);
13.             return $inputFilter->getValue('parameter');
14.     }
15.
16.     /**
17.      * @return string
18.      */
19.     public function getVersionInfo() {
20.         return $this->versionInfo;
21.     }
22.
23.     /**
24.      * @return string
25.      */
26.     public function getReferenceName() {
27.         return $this->referenceName;
28.     }
29.
30.     /**
31.      * @return string
32.      */
33.     public function getMethodSynopsis() {
34.         return $this->methodSynopsis;
35.     }
36.
37.     /**
38.      * @return string
39.      */
40.     public function getMethodDescription() {
41.         return $this->methodDescription;
42.     }
```

Figure 4



Running the Chatbot

Now that the core classes of the chatbot have been written, how do we run it? Well, as I was developing it, I thought about how to use it. The way I'll step through in a moment is specific to Slack. But that doesn't make it easily adapted to other circumstances, such as *Facebook Messenger*, *IRC*, *the command line*, and so on.

What I've been considering is using an Adapter pattern[8] implementation, using different adapter instances for various approaches. I've not done that as yet, but it's something I'm planning to. For now, though, let's step through the existing implementation.

In the code in Listing 5, we first retrieve two essential variables from the supplied `POST` data; these are `$text` and `$token`. `$text` is the name of the function which the user supplied when using the slash command in Slack. `$token` is the auto-generated token which Slack provides in the request payload to the chatbot.

LISTING 5

```
01. <?php
02. error_reporting(E_ALL | E_STRICT);
03. require __DIR__ . '/vendor/autoload.php';
04.
05. use Guzzle\Service\Client;
06.
07. $text = $_POST['text'];
08. $token = $_POST['token'];
09.
10. // TODO move this to dotenv file
11. $secret = 'enter_your_token';
12.
13. if ($token !== $secret) {
14.     $msg = "The token for the slash command doesn't match."
15.         . " Check your script.";
16.     header('HTTP/1.0 401 Unauthorized');
17.     die($msg);
18. }
19.
20. echo (new \ChatBot\PhpManualChatBot(new Client()))
21.     ->lookupFunction($text)
22.     ->getMethodDescription();
```

8 *an Adapter pattern:* https://sourcemaking.com/design_patterns/adapter

You can test the ChatBot locally without connecting it to a chat service by running it with PHP's built-in web server and using a local client to send POST requests.

By using `$token`, we can ensure only valid users can make requests to the service; displaying a message letting them know if they're not able to use it. If they are a valid user, then we instantiate a new `PhpManualChatBot` instance, supplying it with a Guzzle client and DomCrawler instance.

We then call its `lookupFunction` method, supplying it the text which the user supplied. Finally, we call the `getMethodDescription method`, and echo the response.

All being well, this then returns the description of the PHP function which the user requested to Slack, where it's then rendered in the chat window, rather like in Figure 4.

You can see I've called it a few times, each time giving a bit more love and thought to the way in which the response is returned. This made sense, given the overall feel with which Slack works.

In Conclusion

And those are the essentials of writing a chatbot in PHP. I won't claim it's the most masterful of code. But it does show how to create chatbot in PHP. I hope you've gotten a feel for how the process works, and are even inspired to do it yourself.

If you'd like to know more about creating a chatbot, check out https://chatbotsmagazine.com. They have loads of information on the topic. Otherwise, get out there and experiment. It truly does make coding fun!

Matthew Setter is a software developer specializing in PHP, Zend Framework, and JavaScript. He's also the host of http://FreeTheGeek.fm, the podcast about the business of freelancing as a software developer and technical writer, and editor of Master Zend Framework, dedicated to helping you become a Zend Framework master? Find out more http://www.masterzendframework.com.

Focus on What We Have in Common

Cal Evans

For my U.S. friends, I did not vote the way you did. That statement is true for half of you. If that statement makes you angry, please read this. If you are nodding in agreement with me because you think you know me, please read this.

This article isn't about politics, but they drove it. In the PHP community, we have people from all walks of life, including most religions and belief systems. In many ways, our community has achieved a level of diversity—diversity in gender, age, and most importantly, diversity in thought. Yes, we still have work to do to increase diversity in all areas; this is not a call to slack off, but we also need to preserve these hard-won gains.

Remember, most of us don't hang out with each other for our political views, our religious beliefs, or because we agree on whiskey versus beer. (HINT: It's **always** whiskey!) We gather together as a community because we have one thing in common; we use PHP to get things done. It's our willingness to set aside our differences, technical or otherwise, allowing such gains as Drupal building on top of Symfony. Our gains with PSRs over the past few years are evidence that when we work together, it benefits everyone. PHP is our common ground, and it is a very large common ground for us to build on.

In the past, I have had people chastise me for bringing up my religion in talks. As much as I can, I try to minimize this because I do not want to drive anyone away from the community. I want everyone to feel like they can come and talk PHP with me regardless of their beliefs.

I'm not asking anyone to give up their strongly held beliefs. I'm not even asking anyone to keep them to themselves. I am asking when we gather together as a community—online or in real life—we make a concentrated effort to ignore the things separating us and to focus on the things we have in common.

Let's make our community spaces—conferences, forums, etc.—welcoming spaces for developers of all kinds. There are a lot of great people in the PHP community. Some of these wonderful people do not think the way you do and do not believe the things you do. If they are in the PHP community, though, it is a safe bet you both like and use PHP, whether it's via WordPress, Drupal, Laravel, Symfony, Zend Framework, your own homegrown scripts, or another popular framework.

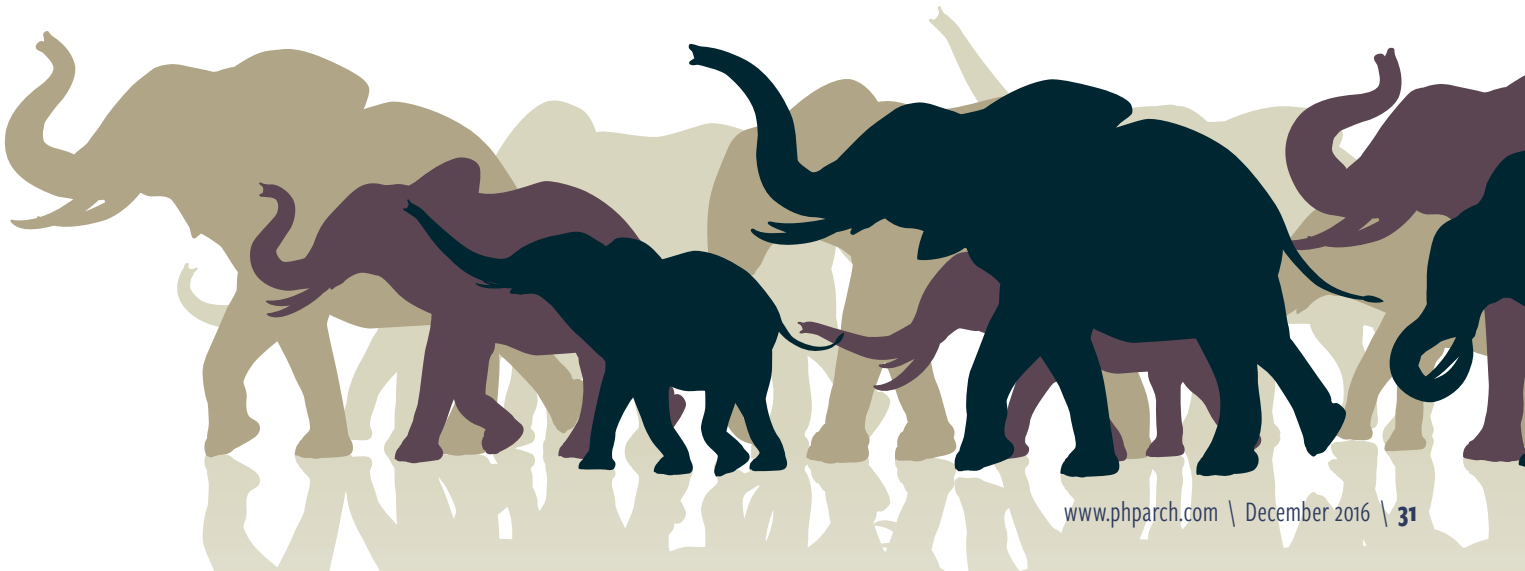
2016 has seen enough hate; not even all of it was political. Let's take some time and effort in 2017 to come together. When we get together, let us as a community focus on what we have in common, instead of our differences. It is the only way our community can grow and become even more diverse.

Past Events

November

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

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



Upcoming Events

December

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

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

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

January 2017

PHPBenelux Conference 2017
January 27–28, Antwerp, Belgium
https://conference.phpbenelux.eu/2017/

February

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

PHP UK Conference 2017
February 16–17, London, U.K.
http://phpconference.co.uk

March

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

Midwest PHP 2017
Bloomington, Minnesota,
https://2017.midwestphp.org

April

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

DrupalCon 2016
April 24–28, Baltimore, MD
https://events.drupal.org/baltimore2017

May

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

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

International PHP Conference 2017
Berlin, Germany
https://phpconference.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.

Building Better Bug Reports

David Stockton

One of my favorite quotes about programming comes from Edsger W. Dijkstra. It says, "If debugging is the process of removing bugs, then programming must be the process of putting them in." It indicates the maxim that as long as there is code, there will be bugs. Of course, we want to minimize the number of bugs in our code. Understanding what the desired behavior of our software is, and how that differs from the actual behavior is the key to understanding what the bug is and how to fix it.

What Are Bugs?

In this magazine over the past two months, Vesna Vuynovich Kovach wrote about the history of women in computing, so this story should be familiar. On September 9, 1947, Admiral Grace Hopper, recorded the first instance of a computer bug in her log book. In her case, the room-sized computer, the Harvard Mark II, had a program which was not behaving properly. After careful inspection of the literal inner workings of the computer, she found and removed the bug in her program. It was an actual moth caught in relay number 70 in panel F of the computer. Figure 1 shows the log book. See if you can spot the bug.

Since then, undesirable or errant code behaviors have received the label "bug." In some bug trackers, the term is labeled "defect," but the bottom line is whatever you call it, chances are you want to be able to find and remove them from your code. They are problems in the logic and working which prevent correct and proper execution of a program.

According to Admiral Hopper's log book, she started a multi-adder test. I imagine somewhere along the line, probably pretty early in that test run, she noticed something which supposed to add up properly, wasn't. At this point, programming was much closer, literally, to coding on the bare metal. I imagine she looked at the tests that worked and looked at the ones that didn't and started tracing through the code, into the actual hardware. This process likely involved following physical wires and cables until the bug was found in the relay. The bug was then removed, the tests re-run, and everything passed and she stopped and went home at 5 PM.

Imagine how different this scenario could have been today when we are able to (mostly) take for granted the hardware is doing what it should. If this multi-adder code was in a library, it could have hidden or internally corrected for the bug. The problem with removing these bugs then, is if the code is relying on them being there, then removing it can cause breakage. It's also important to note when tracking down and removing bugs, it's best to start with our own code and exhaust the possibility the bug is in our code before moving to blame the framework or the language. For the vast majority of people, the bug will be in their code, not in the framework or the language. I've seen loads of bug reports where the developer thinks they've found a problem in a framework or the language when what they actually have is a bug in the code they wrote.

Bug Reporting

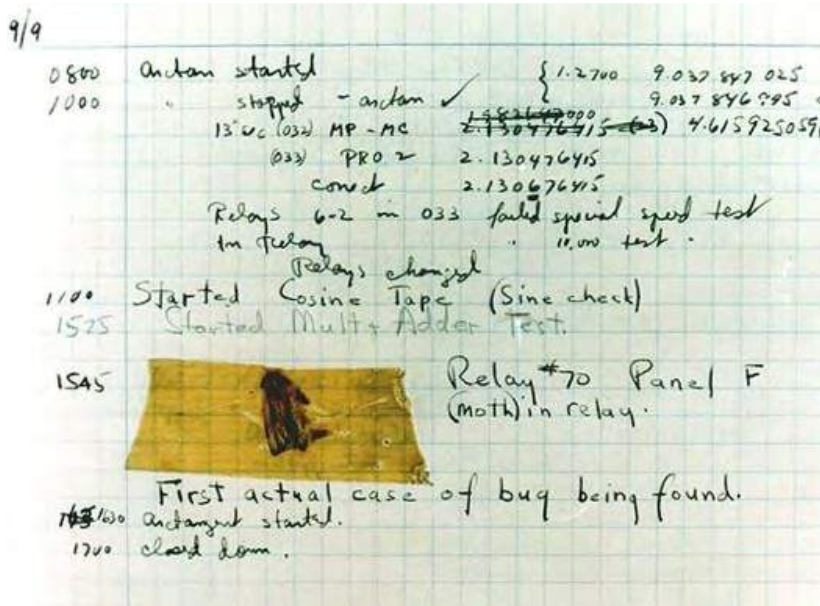
In my article last month, I mentioned changing bug tracking software if it was not helping build software or was interfering with your processes. It was not intended to be a promotion or endorsement of any particular bug tracker, but rather to find and use a bug tracker which works for you and your processes. Don't let bad tools get in the way of improving your process.

We've all likely seen horrible bug reports:

> *It doesn't work.*

In this case, we have no idea what doesn't work. Is it because the entire site is gone, a sign-up process is broken, custom reporting functionality is not right, the dashboard isn't rendering, or the user's Wi-Fi is down? We literally have no idea what they are telling us is broken and little hope of

Figure 1 - First recorded computer bug



being able to help.

> *Your site is broken.*

This report is not much better than the one above. It could be anything still, up to and including problems on the user's computer or network. Without more context, helping the user out will be very hard. Giving this report to a developer will be an exercise in frustration for them and whoever reported it.

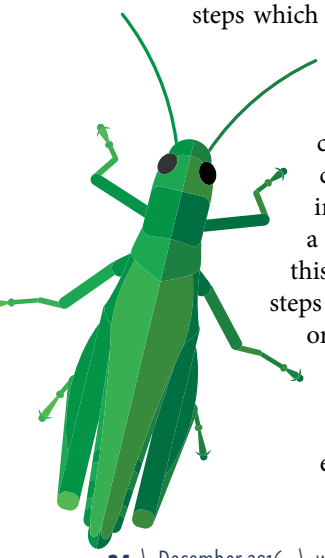
> *The Internet is down.*

First of all, no, probably not, unless the Mirai botnet has taken over a lot more cameras and toasters. In all likelihood, this report indicates some sort of an issue which exists entirely on the user's side. A good indication of this is if they entered a bug like this through email or through your bug tracker directly. If the internet were down, those wouldn't have worked. Again, in all likelihood, the user is having a problem with one aspect or utility, but doesn't know enough about how things work to properly report the issue.

Building Good Bug Reports

There are three critical properties of a good bug report. A good bug report is one which will help the person fixing the issue track down the problem quickly and easily, identify what the issue is, and fix it.

The first aspect of a good bug report is a clear and ideally minimal set of steps which can reproduce the issue. This is not always possible to provide. If it can be provided, it can be invaluable in tracking down a problem. Ideally, this would be a set of steps from some known or common starting point which will result in the issue occurring. For example:



1. Clear browser cookies
2. Go to the homepage
3. Click on "Login" menu
4. Provide valid credentials
5. Click login button

In this case, as a developer digging into this, we now have a sequence of reproducible steps which should cause the problem to happen. Not all of these steps might be strictly necessary to cause the problem, but since there are only a few steps, it's not that big of a deal. We don't necessarily need the person reporting the bug to understand everything about what is happening. In some cases, the number of steps to reproduce may be large. Many of the steps may be unnecessary, but if there are steps which can cause the problem, then we're already starting off in a better place than most bug reports.

In some cases, a bug report with a huge number of steps may be what the developer needs. I remember one such case where the bug filed by the QA Engineer appeared to involve clicking on nearly every navigation link in the application. I mentioned it in passing to one of the developers. My recollection of the exact click path of the bug was sketchy at best, even at the time and involved saying something like "and then you click on this, and this, and then do that about 19 more times on other parts and then the app stops responding." It turns out my inexact relaying of the bug report to the developer was all he needed to theorize what the problem was. Clicking through two specific navigation items caused the problem. They both loaded the same component into the DOM and that component had the same HTML ID. Once that was in place, code that assumes that IDs are unique in the DOM stopped working and the application stopped responding to clicks.

The second aspect of a good bug report is to list what the expected result is. Often this will be short and to the point—"The user is logged in" or "The site doesn't crash and show a stack trace to the user" or "The user is greeted by name." If the expected behavior is short and to the point, many developers will be able to make the logical leap to guessing what the actual behavior was—that is, whatever the expected behavior was, negated.

The third critical piece to a bug report is the actual result. While it may be possible, in many cases, to guess the actual behavior is the expected behavior, but negated, this is not always the case. Providing the actual behavior explicitly will help the developer sort out what is happening. Without it, the developer may encounter a different bug or what they think is a bug, and then fix that behavior, only to have the bug report re-opened. This leads to frustration on the part of the bug reporter and the developer since work is being done, but the desired result is not happening.

Missing Features Are NOT Bugs

I want to step back for just a moment to reiterate some things I've said in previous articles. It is important we do not categorize as bugs features which have not been implemented. Suppose we look at the bug report from above, specifically the one indicating the user should be greeted by name once they are logged into the application. Under different scenarios, this could be seen as a bug report, while in others, it's most definitely a feature request.

If the site has never greeted the user by name, chances are you're looking at a new feature request, not a bug. That's not always the case, but in general, if the application never did the behavior being referenced in the bug report, it's probably not a bug report but a feature request. Properly categorizing features versus bug reports is important because it can and will affect team morale.

That being said, I can understand the argument if there were requirements that the site greet the user by name and the site is delivered without that functionality, it could be seen as a bug. I'd argue greeting the user by name should have been a separate feature request all on its own, rather than bundled in with a bunch of other functionality descriptions and requests. This would indicate if there was a ticket, story or feature

request for displaying the logged in user's name. If the site doesn't do it, then the greet-by-name functionality was never delivered. If it was delivered and worked at some point, but no longer does, then it would almost certainly qualify as a bug.

Bug Priorities and Severity

Bugs in software are a fact of life. We will never be able to remove all of them from any given (non-trivial) piece of software. We should work toward removing every bug we can from our software. However, not all bugs are created equally. Some bugs have a relatively minor impact on the site or its users. A bug involving a spelling error or a link being the wrong color shade or a navigation item being in the wrong order is certainly a different level of impact than users not being able to log in, users seeing someone else's information after they log in, or the site crashing and displaying a stack trace if they hit the back button on the browser.

Most bug tracking software will have a concept of *priority* as well as the concept of *severity*. It's important to understand what these mean.

The severity of a bug indicates what its impact is. This could include data loss or corruption, monetary loss, crashing, causing areas of functionality to be unavailable, etc. Often the severity of a bug will be rated with terms like critical, major, moderate, minor or cosmetic.

For priority, we're talking about the order we should fix the bugs. Typically these are going to use words like low, medium, or high. The reason to have these separated is there is not necessarily a direct correlation between the severity and the priority of a bug. You could, for instance, have a bug where there was a typo in the motto on your company's logo image. The bug may be categorized as cosmetic severity, but with a high priority. This indicates that while there's no data loss or crashing, it is important to resolve the issue right away. Conversely, you may have a critical bug which breaks data, but it only happens for one person in your QA team because they are running IE 6 with a weird plugin and admin privileges. It's still important to fix it, but you may have the option of telling your tester to use a different browser until you can fix the issue.

Let's take a look at what these levels of severity and priority mean.

Severity Levels

- **Critical** - This severity level indicates the bug causes a crash in one or more areas of the system. The bug can be data loss or corruption. There is no acceptable workaround or alternative path to allow a different way to accomplish the same task.
- **Major** - This level can be seen as essentially the same as critical: the application crashes or area of the system is not usable. But there is a usable workaround allowing users to continue and complete their work, but not in an optimal way.
- **Moderate** - At this level, we are not crashing, but the system may be producing incorrect or inconsistent results.
- **Minor** - A minor defect would not result in a crash, but a bug at this level could affect the usability of the system. There should be an easy way to work around the defect to obtain the desired result.
- **Cosmetic** - If the bug involves the look and feel, but doesn't involve any crashing or data corruption. At this level, we're really talking about things like typos, incorrect images, or colors.

Priority Levels

- **Low** - Fixes for low priority items can be delayed or deferred until later. They are annoying, but more serious issues should be fixed first.
- **Medium** - Medium priority bugs should be fixed during the normal course of development. It should be fixed soon, but it isn't something which needs to interrupt or change the normal flow or schedule of deployment.
- **High** - High priority issues need to be resolved as soon as possible. These are the bugs which should or can be deployed outside the normal deployment schedule.

Some examples of bugs with varying severity and priority levels:

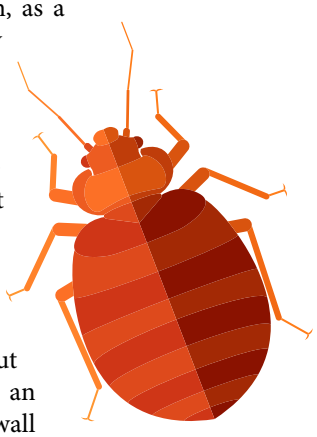
High priority and critical severity: This is a bug which happens in the normal and common use of the application. It will prevent the user from using the application.

High priority and low severity: This could be a spelling error or typo on the front page or start of the application. It's something everyone will see, but it's not affecting the functionality of the site. It is important these be fixed right away to maintain the appearance of professionalism, but the bug is not really causing problems with using the application.

Low priority and low severity: These would be cosmetic issues which are not seen or commonly encountered. It could be a typo in the middle of a paragraph of text in the help section of the site. Perhaps it's something people don't often encounter, and if they do it's not going to affect their usage of the application.

Managing Defects

In general, I try to treat any defect with a higher priority than what it was assigned when it was created. What I mean is: if there are any defects reported against my application, I like to move them up in priority so all known or reported defects are resolved before working on new functionality. I like to treat any bug which causes a crash or stack trace, no matter how bizarre, obscure or unlikely it is the user can cause it to happen, as a critical issue. By doing this, I keep defects to a minimum. There will always be more bugs found, but by trying to fix all bugs before building new features, I don't have to worry about running into an insurmountable wall

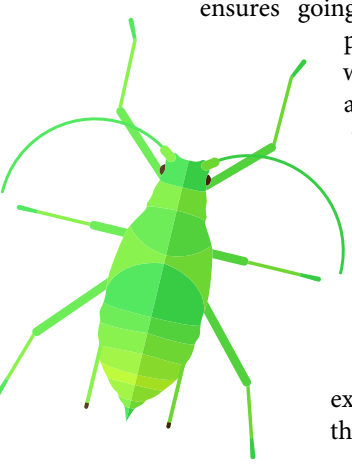


of bugs eventually bringing the application to a grinding halt.

I try to be unmerciful with bugs. The applications I'm currently building are typically API driven. This means if we have new functionality to add to an application, we'll build an API which allows that functionality to work. I'll push that code into the application and give the QA team information and documentation about how the API should work. Keep in mind at this point, there's absolutely nothing that's using or relying on this new API. However, if and when bugs are reported, I jump on them, eager to fix them and provide the most solid API I can before I turn it over to my UI team to connect with and use. That gives both them and myself a lot of confidence we have built an application which will be solid and is not likely to change much. It also gives them confidence that if they run into an issue when integrating with the API, we'll be quick and responsive to fix anything they found.

Making Sure It Never Happens Again

If you've been reading my column for any amount of time, you may have noticed I'm a big fan of using code to test code. This includes automated tests of all kinds: unit tests, behavioral test, functional tests, etc. This means whenever possible if there's a bug reported, I'll try to write a unit test or Behat test that expects the correct behavior. This test will fail, and then I'll create the fix for the bug. The test and the fix will be submitted together which then ensures going forward, that particular bug will not happen again. If it does, the automated builds will catch it and prevent the regression from being deployed.



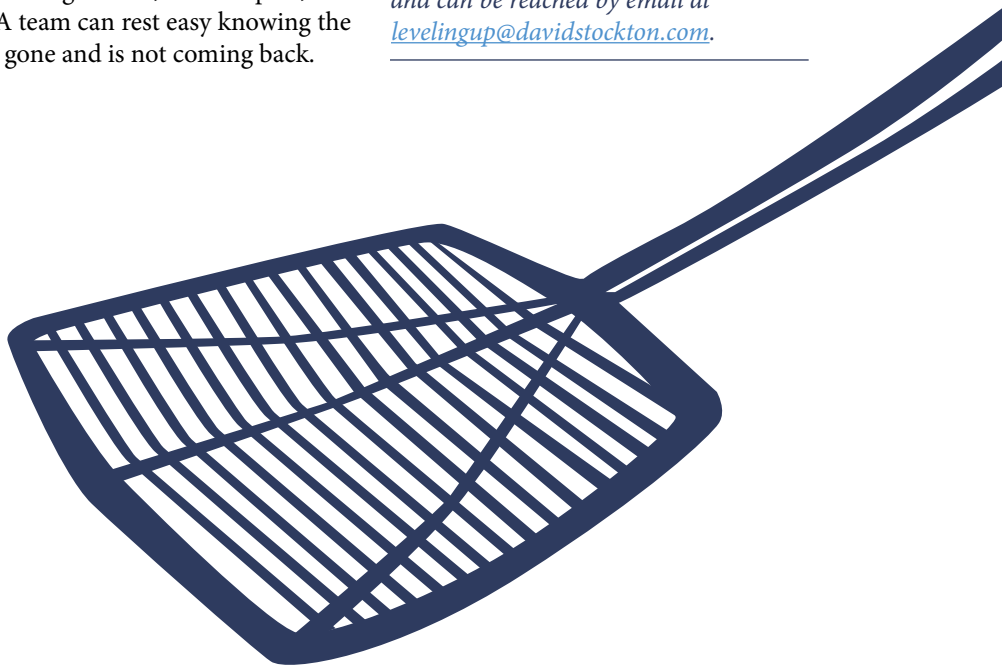
Here's an example of how this might work. In an API you may be passing the ID of a resource via the URL. This ID will be involved in an SQL query to fetch the data for the resource. Suppose the ID field in the database is defined as an integer or similar. With Postgres, and possibly other databases, trying to add a `where` clause on an integer field which compares to a string will give back an SQL error along the lines of "banana is not an acceptable representation of an integer." This means I need to be sure I'm only requesting integers from the field. (Please note, this happens even with prepared statements, I'm not just blindly passing everything to the database.) The Behat test I would write would make a request to the API and pass in a non-numeric string ID. I'd then add expectations that a specific error would happen —perhaps a 404 indicating the URL doesn't represent a resource, or a more specific type of error that shows the ID must be numeric. The test will fail when the SQL error happens, but should pass once the validation of the ID is in place. With the combination of the test and the bug fix, we can be assured this bug is gone.

I also encourage my QA team to build Behat tests when they find bugs. This allows them to enter bug reports with actionable, runnable tests I can use to help find the problem. Additionally, the validation step they'd normally need to perform manually and ensure the bug was properly fixed is no longer needed. Once the bug is fixed, the tests pass, and the QA team can rest easy knowing the bug is gone and is not coming back.

Conclusion

Bug reports that are valuable help developers find and fix the bugs quickly. These reports need to have reproduction steps as well as actual and expected outcomes. By clearly communicating these expectations in the form of a well-written bug report, or making sure the people who are reporting bugs know what is needed, we can help reduce the time it takes to find and fix bugs. Properly categorizing the priority and severity of bugs let developers know what to spend their time on next to improve the application. By fixing code via TDD or by including automated tests along with the bug fix, developers can help ensure regressions are minimized and prevented. At the same time, QA can be assured the bugs they reported have been properly fixed without needing to resort to manual verification of bug fixes. Finally, by treating bugs as something to eradicate quickly after they are found, you help ensure your application works as well as it can and will be maintainable for a long time to come.

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.



Keeping a Secret

Chris Cornutt

With any sufficiently advanced application comes a common piece of functionality—the need for sensitive credential handling. Sometimes, this comes in the form of user-related information (like a user's password), and other times, it's more related to sensitive values required by the code itself (like database connection information). Regardless of where this sensitive information comes from, the same absolute rule should be followed: protect it at all costs.

Securely Protecting Secrets & Credentials

While it might seem obvious that secrets in your application should be diligently protected, there are situations in which this is easy to forget and have them exposed.

First, we'll start with a list of "don'ts" so you'll know what *not* to do. Then, we'll follow it up with advice on good practices you can follow to replace current methods or enhance future development.

Don't Commit

Credential information should never be committed to your repository or anywhere in your code. Having any kind of authentication information hard-coded in the actual code is a major issue, especially if they're stored in plain-text. Committing secrets right alongside your code leaves them vulnerable if any kind of vulnerability allows the attacker to view the source of your application (like a local code execution vulnerability or exposing your `.git` folder via your web server).

Additionally, by committing those secrets, you've also caused an even bigger issue. Even if you realize the mistake, it's too late to pull them back out—the version control history is there, and those values could be retrieved at any time. Sure, most version control systems have the ability to rewrite history and remove some commits, but what happens if you don't discover they've been committed until months down the line? Think of the pain and suffering that one could take to fix.

Do yourself a favor, and keep those credentials and sensitive information out of your committed code. I'll cover better options for the storage of these secrets later in the article.

Don't Store PII Unless You Have To

There's a common term that you might have heard before when it comes to protecting your user information: *PII*. This stands for *personally identifiable information* and consists of data which could potentially be used to directly identify a user of your system. The most common and easy-to-identify examples include (but are not limited to) data such as social security numbers or credit card information. Single pieces of information could be considered PII by themselves, but there's also the possibility of a combination of data becoming PII relevant. PII is a pretty tricky road to navigate, and plenty of court cases have been won and lost based on what could be considered PII in certain situations.

As a rule of thumb, I usually offer this advice: if you're worried that a piece of data might potentially be PII, treat it is as such. That being said, there's also another, simpler way to not have to worry about PII issues at all: store as much of this information on external services as possible. There are two ways to approach this:

1. You can use something internal to store the services that specializes in protecting them at rest, such as Vault from Hashicorp[1], or
2. You can store them on a third-party service that returns an identifier back to you that you can use in future transactions, representing the data as it is stored on their side.

1 *Vault from Hashicorp: https://www.vaultproject.io*

The first option does help protect the secrets and get them out of your code, but it also introduces an internal dependency that you'll need to support yourself. While you may have requirements where that's the best solution (like corporate restrictions), I definitely suggest the use of external, managed services. For example, you can store credit card information with Stripe. With their simplest flow, you use JavaScript from their site that provides the user with a pop-up asking for credit card information. This information is then sent directly to their services and the data is stored in the Stripe system. No requests containing the credit card information are sent back to your application, instead Stripe sends a request to a callback script waiting on your side with a unique identifier or an error message if the charge does not succeed. Your code can reuse this identifier in future transactions for the user's card information, instead of you having to store and send along the card information every time.

LISTING 1

```
<form action="/your-server-side-code" method="POST">
<script
  src="https://checkout.stripe.com/checkout.js" class="stripe-button"
  data-key="pk_test_your-key-here"
  data-amount="2000"
  data-name="Stripe.com"
  data-description="2 widgets"
  data-image="https://stripe.com/img/documentation/checkout/marketplace.png"
  data-locale="auto"
  data-zip-code="true">
</script>
</form>
```

Listing 1 is an example of all that's required to use Stripe to accept payments in your application (assuming you already have an account, naturally):

Obviously, some of the information needs to change to match your account, but that's basically all there is to it. Stripe's JavaScript sends the data to the Stripe server via a HTTPS connection and responds to your endpoint (defined in the form action) with the token and a few other pieces of information.

There are several payment vendors out there that will store information like this; it's not just limited to Stripe. This takes a huge burden off of you and your system having to protect this information and having to deal with the fallout in case of a breach. With a third-party service storage solution like this, even if an attacker was able to gain access to your database, they'd only be able to get that unique identifier. This identifier is worthless without the credentials to connect to the remote service. Hopefully, you're storing those encrypted and preferably in a different place than these identifiers.

Don't Default

Some PHP code is designed to be used as a library or package in a larger project. However, there are projects that are designed to be installed on a server as a stand-alone tool. Control panels are a common example of this. With these kinds of installations, there's usually some kind of login or authentication that comes along with it. Oftentimes, these are in the form of default credentials, and they are usually related to an administrator-level account in the system. Sometimes, the setup process will include a backend script allowing you to specify the default account credentials. More often than not, it just relies on either something simple like "admin/admin" or a single hard-coded password used across every installation of the product.

While using default credentials like this isn't inherently bad (it's not great, but at least it's some kind of protection), leaving these credentials in place opens a large security hole—one that's been exploited by many an attacker in the past. If all instances of your software or service ship with the same credential information and you don't change them, any attacker can easily locate that information and have open access to your system.

I'll detail down in the "force credential changes" section below some things you can to do help remediate this problem. I mostly wanted to bring it up to get you started thinking about your own code and systems to think about where you might be using either default or easily guessable credentials.

Don't Log

In a previous article I mentioned "logging on purpose" when thinking about what to record in your application during the request/response and processing flow. As a quick recap, this basically means only logging things that are known and limiting the logged information as much as possible. This limitation applies to credentials, as well. Good logging practices suggest that you only log internal identifiers and non-PII information when it comes to tracking down errors or creating an audit of user activity in your application.

Apply this same idea to any place where credential information is used in your application. For example, PHP has a nasty habit of dumping a lot of content about the current request into exception messages—potentially including connection information. I've seen error output from a site where the database password and username were exposed directly in the exception output as the PDO connection was created during the execution flow. If you're blindly taking exception messages and logging them in your application, you run the risk of this information making it into your logs. Many companies will spend a lot of time and money defending and hardening their web servers and services but not as much on where the logs are stored. Attackers can take advantage of a weak logging server, and if they are able to locate exception messages with credential information, they could easily use that information in an exploit…an even worse one if the credentials are shared.

Don't Deploy Source Control

This tip goes along with the first one in this list (don't commit) and has to do with the contents of your version control's "special files" or directories. For example, in git repositories, there's the `.git` folder created when you clone or initialize a repository. By its nature, a git checkout is a copy of the entire repository all bundled up and ready for use. Because of this, all of the commits ever made on the repository also come with it. If you make this directory web accessible, you make it so that anyone can harvest the contents of this directory and have complete access to your files.

Sadly, this happens more often than it should. Don't believe me? Try searching for `".git" intitle:"Index of"` on Google, and see what comes up. There are pages and pages of domains that have `.git` directories out there just waiting for an attacker to abuse.

> *This special string is called a "Google dork" and is one of many kinds of special strings that can locate pages and resources matching certain patterns. You can find plenty more by just searching for "google dorks" on Google.*

It's not just `.git` directories that are a problem—sometimes it's directories or files that come with the default installation of a software product that should be—but are never—removed. This could even be something as simple as a bad `robots.txt` that exposes too much or, perish the thought, a "backup" directory of some kind with version control information all its own. Just avoid the hassle. In your deployment process, either don't deploy the version control files or remove them as a final step in the deploy.

Now that we've covered some of the major "don't" items, lets go through good practices to follow.

Do Use Environment Files/Settings

One of the best and easiest things you can do to protect the credentials and secrets in your application is to define them in environment files or settings in your web server. This keeps them out of the code itself. It also has the benefit of being easier to switch out between environments. As an example, let me describe a deployment process I have for one of my projects:

1. I use the Deployer[2] tool to connect to the remote server and grab the latest from the project's repository.
2. The next step then uploads the matching environment settings file based on what environment I'm deploying to (ex: `.env.staging`).
3. Finally, it moves the file into place as just `.env` so the application can find and use it.

The files containing the credentials aren't checked in anywhere; they only exist on my local machine and are deployed directly from there. As a result, the configuration files for the other environments never even make it to the other servers. That's another key—be sure when you're doing your deployment that *only* the credentials for that environment are deployed. Deploying credential and secret information to environments they're not intended for is an additional risk that you just shouldn't have to worry about.

Do Protect with Encryption

Encrypting secret information or any kind of PII in your system is one of the best things you can do to protect it. There's a whole other world that comes with answering the question of "what's the best way to encrypt" and just as many articles/tutorials out there to help you pick the right methods. One method that's been highly touted by the Paragon Initiative group is the use of libsodium[3] to help simplify and protect your data. The only unfortunate thing about libsodium is that it requires extra functionality via an extension, pecl-libsodium[4] which doesn't come installed by default with most PHP installations. However, it's relatively simple to add and is pretty much a "drop-in" piece of functionality.

> *Note that the mcrypt extension for PHP will be deprecated and removed in upcoming PhP versions. If you're still using it for your encryption needs, consider updating to the OpenSSL functionality or even something stronger.*

I've pulled an example from one of their blog posts showing how simple it is to use in practice once installed via Composer. Listing 2 is adapted from Solve All Your Cryptography Problems in 3 Easy Steps[5]

LISTING 2

```
01. <?php
02. use \ParagonIE\Halite\{
03.     Symmetric\Crypto as Symmetric,
04.     KeyFactory
05. };
06.
07. // Make a key and save it (done one time)
08. KeyFactory::save($encryptionKey, '/path/to/encryption.key');
09.
10. $encrypted = Symmetric::encrypt('this is your plaintext',$key);
11. echo 'Encrypted version: ' . $encrypt . "\n";
```

That's basically it. Libsodium handles most of the hard part for you, and it uses a robust encryption method to protect the data at rest. As for the key, you'll need to protect that adequately, but key management is, again, a whole other topic.

2 *Deployer:* https://deployer.org

3 *libsodium:* https://download.libsodium.org/doc/

4 *pecl-libsodium:* https://paragonie.com/book/pecl-libsodium

5 *Solve All Your Cryptography Problems in 3 Easy Steps:* http://phpa.me/paragonie-solve-crypto

Do Force Credential Changes

Previously, I mentioned avoiding the use of default (or easily guessable) credentials in your application and said I'd offer a few other solutions to help you avoid this common pitfall. Here's a list of a few methods you can use, instead of these hard-coded credentials. Some are more difficult than others, but they're all better than the alternative:

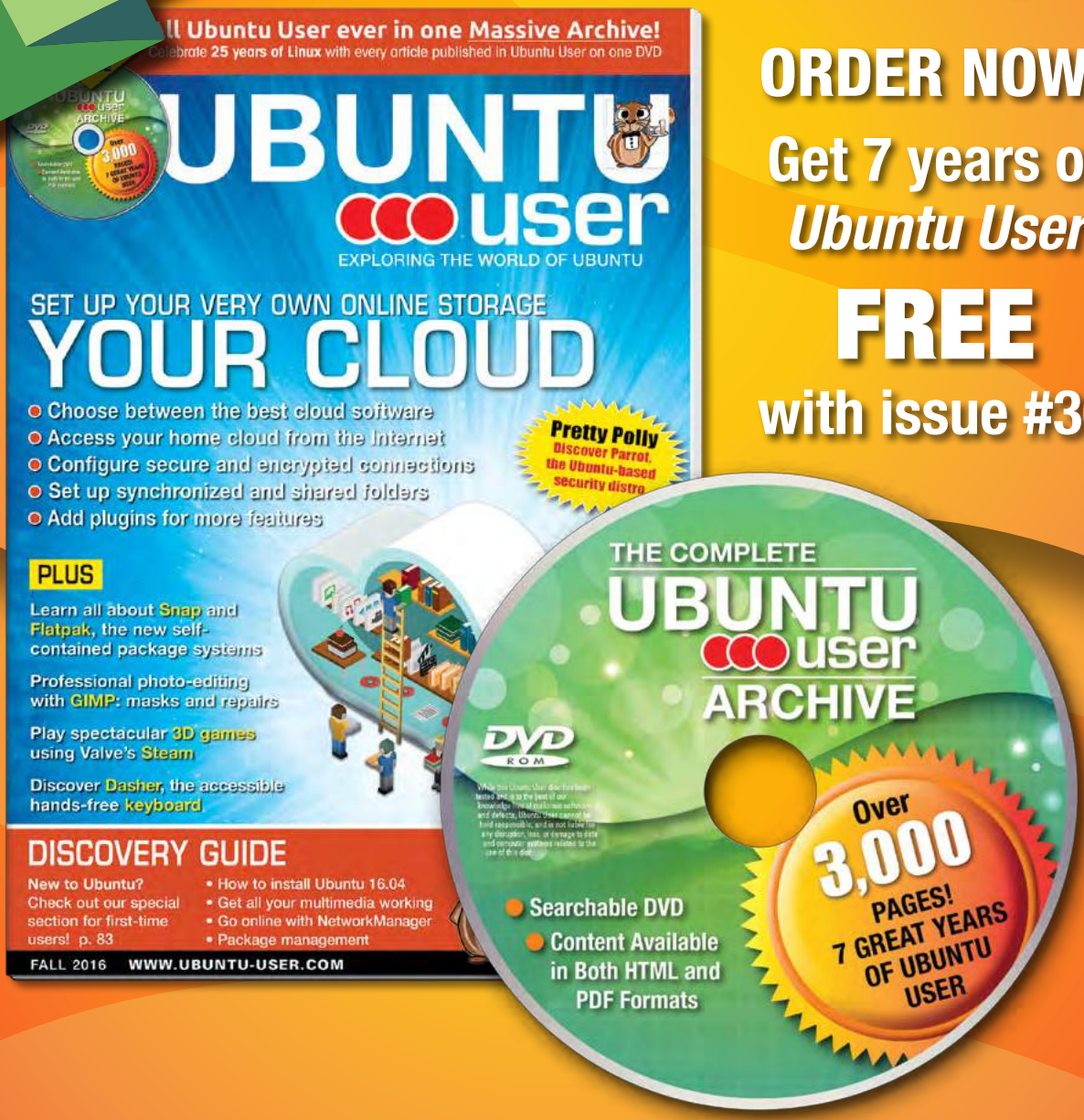
1. Creating user-defined credentials at installation: I've seen several tools that take this approach. As a part of whatever setup steps you have to follow, they ask for the administration credentials before you can continue. WordPress does this when you're doing a new installation via its web interface. The major drawback here is that by allowing the user to define the credentials, they could provide poor or weak values and not be much better than their hard-coded counterparts.
2. Generating credentials during the installation: This is slightly different than #1 in that you're generating the credentials yourself based on some programmatic method to ensure the credentials are robust and not something silly like "admin/admin." This method comes with its own issues, however. Generated passwords are, in general, harder for users to remember, so they tend to write them down and leave them laying around—a whole different kind of security issue.
3. Verifying the credentials from a third-party source: Sometimes tools just don't want to have to verify you based on their own internal credential handling and choose to offload it to some other service or tool. This usually requires additional setup to connect the tool to whatever service it needs to use (like an LDAP server), but in the long run, it can definitely be more secure than other options.

In Summary

I've shared some high-level "do's" and "don'ts" in this article regarding the securing and protecting of your secrets and credentials in your web applications. Hopefully, I've given you some good, actionable advice you can go and immediately apply in your own tools and services to help prevent—or at least minimize—the damage that an attacker could do to your systems if they were able to easily access these secret pieces of information.

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

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

November Happenings

PHP Releases

PHP 7.0.13: http://php.net/archive/2016.php#id2016-11-10-1
PHP 5.6.28: http://php.net/archive/2016.php#id2016-11-10-3
PHP 7.1.0 RC6: http://php.net/archive/2016.php#id2016-11-10-2

News

Kinsta.com Blog: What's New in PHP 7.1.0

On the Kinsta blog there's a post detailing some new features coming in the next release in the PHP 7 series—PHP 7.10. Their list of items includes: nullable types, iterable and void returns, the use of keys in lists, and number operators and malformed numbers. http://phpdeveloper.org/news/24628

Paul Jones: The PHP 7 "Request" Extension

Paul Jones has a new post to his site introducing the "Request" extension he and John Boehr have worked up to make working with HTTP requests in PHP simpler. He gives an example of using the extension to work with both the request and response (`ServerRequest` and `ServerResponse`). This includes cookie values, files handling, content length and much more. http://phpdeveloper.org/news/24623

Alison Gianotto: Demystifying Custom Auth in Laravel 5

Alison Gianotto (a.k.a. Snipe) has a new post on her site talking about custom authentication in Laravel-based applications including built-in functionality and how you can override it to your needs. She starts by mentioning the "fresh" install version of building out the auth pieces (`php artisan make:auth`). She shows you the routes created in the `make:auth` process and how/where you need to modify things to customize it to your system. http://phpdeveloper.org/news/24615

Jordi Boggiano: PHP Versions Stats—2016.2 Edition

In his latest post Jordi Boggiano (of the Composer project) has released his PHP usage statistics for the second half of 2016 based on the information gathered during Composer installations. He compares them to the statistics from May 2016 showing some interesting but not unexpected changes, mostly in the growth of PHP 7+ versions. He shares a few of his own observations of the results and encourages library authors to start focusing on PHP 7 functionality rather than 5.5/5.6 compatibility. http://phpdeveloper.org/news/24610

Tumblr Engineering Blog: PHP 7 at Tumblr

The Tumblr Engineering blog has a new post with details about how they made the switch to PHP 7 in their previously PHP 5 codebase (and the things they learned along the way). They start off with the timeline of events, starting with the original hackday project out through the final PHP 7 deployment in production less than a year later. They cover the testing methods they employed during the transition and the impact of the update on their application on request latency, CPU load and memory usage. http://phpdeveloper.org/news/24587

SitePoint PHP Blog: Beaver in Action: Practical MySQL Optimization

On the SitePoint PHP blog there's a tutorial posted showing how to optimize your MySQL handling with the help of the Beaver query logger package and the details it provides. He shows how to update your MySQL installation to log all queries out to the log location of your choice. This log can then, in turn, be parsed by the Beaver package and provide details about what's happening in the query and where it could be optimized.
http://phpdeveloper.org/news/24584

Andreas Creten: Does Code Need to be Perfect?

On his Medium.com blog Andreas Creten has written up a post which tries to answer the question "Does code need to be perfect?" As developers, we have a drive to take pride in our work and want it to be the best code possible. However, this drive can lead to some bad practices…He offers a few different suggestions for those developers wanting to craft the perfect codebase including coding for "now" not the future and the fact that "perfect code" just doesn't exist.
http://phpdeveloper.org/news/24583

Chris Hartjes: True North PHP Is Done

Chris Hartjes has a new (sad) post on his site today sharing news about the True North PHP conference, the "first PHP-centric event in the Greater Toronto Area since 2006"—the organizers have made the decision to end the conference's run after five years. He gets into the details about why the conference was started, the work he and Peter Meth put into the project and why they made the decision to no longer run the event.
http://phpdeveloper.org/news/24570

Joe Watkins: Expanding Horizons

In his most recent post Joe Watkins talks about a PHP extension he's been working on wrapping the libui library making it easier to use PHP to create cross-platform user interfaces. He points out it is still early on in the development cycle for the extension and libui but there's already documentation for those wanting to investigate.
http://phpdeveloper.org/news/24560

ThisData Blog: Subscribing to Symfony's Security Events

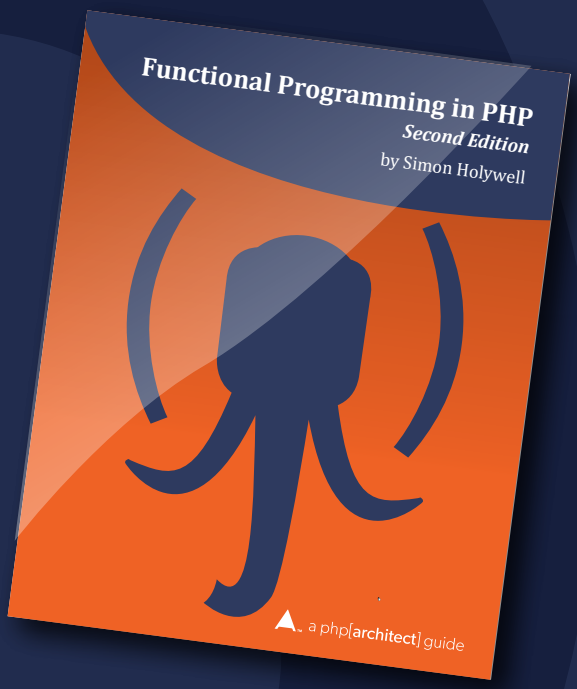
In this recent post to the ThisData blog Nick Malcolm shows you a method for subscribing to the events the Symfony framework throws during its execution with simple listeners. He starts with a Symfony demo application and show the creation of a basic subscriber to specifically listen to the security events.
http://phpdeveloper.org/news/24556

Functional Programming in PHP

Second Edition

by

Simon Holywell



Many languages have embraced Functional Programming paradigms to augment the tools available for programmers to solve problems. It facilitates writing code that is easier to understand, easier to test, and able to take advantage of parallelization making it a good fit for building modern, scalable solutions.

PHP introduced anonymous function and closures in 5.3, providing a more succinct way to tackle common problems. More recent releases have added generators and variadics which can help write more concise, functional code. However, making the mental leap from programming in the more common imperative style requires understanding how and when to best use lambdas, closures, recursion, and more. It also requires learning to think of data in terms of collections that can be mapped, reduced, flattened, and filtered.

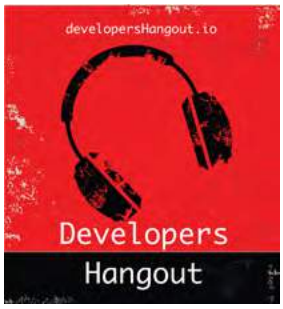
Functional Programming in PHP will show you how to leverage these new language features by understanding functional programming principles. With over twice as much content as its predecessor, this second edition expands upon its predecessor with updated code examples and coverage of advances in PHP 7 and Hack. Plenty of examples are provided in each chapter to illustrate each concept as it's introduced and to show how to implement it with PHP. You'll learn how to use map/reduce, currying, composition, and more. You'll see what external libraries are available and new language features are proposed to extend PHP's functional programming capabilities.

Buy Your Copy Today!

http://phpa.me/functional-programming-in-php-2

Welcome to php[architect]'s new MarketPlace! MarketPlace ads are an affordable way to reach PHP programmers and influencers. Spread the word about a project you're working on, a position that's opening up at your company, or a product which helps developers get stuff done—let us help you get the word out! Get your ad in front of dedicated developers for as low as $33 USD a month.

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



Listen to developers discuss topics about coding and all that comes with it.
www.developershangout.io

Kara Ferguson Editorial

Refactoring your words, while you refactor your code.

@KaraFerguson
karaferguson.net

technology : running : programming
rungeekradio.com

Fred WebTech

The Frederick Web Technology Group
meetup.com/FredWebTech

DC PHP DEVELOPERS

The PHP user group for the DC Metropolitan area
meetup.com/DC-PHP

The Year of ???

Eli White

Once again, as the weather cools in the Northern Hemisphere, we become reflective of the year that has transpired. When looking back at 2016 for the PHP community, however, what do we find?

You see, 2015 was a fantastic, exciting year for PHP. Spirits were high as PHP 7 was finally released bringing a slew of performance enhancements and new, highly anticipated features. Other PHP ecosystems were evolving as well, with the release of Drupal 8, Magento 2, and Symfony 3.

> *The wheel is come full circle.*
> *–William Shakespeare*

Then as should have been expected, we rolled into 2016, and everyone collapsed from exhaustion. I've seen it before in other communities as well. When there is that much excitement building throughout a year, once the big event happens everyone without even thinking about it takes a deep breath and steps back a bit.

So that's it, 2016 in PHP was the Year of the Breath. When we all sat back, saw what had been done in 2015 and relaxed a bit. We saw this overall in reduced community contributions, withdrawal from user groups, and—from chatting with other organizers—lower attendance at conferences throughout the US.

It's nothing that we should be worried about, however; as it's a natural cycle of ups and downs in the technology sphere. Now we've had our year of rest and the release of 7.1 that is imminent. We can expect new, yet undiscovered ideas on the horizon to re-excite the imaginations of PHP developers which in turn will breath new life into the community as we carry forward.

This year may not have been filled with huge, exciting announcements. It's just because we are all still being blown away by the features of PHP 7 which we now get to use.

Eli White is the Conference Chair for php[architect] and Vice President of One for All Events, LLC. He himself has gone full circle and ended up starting another business. Life can be strange sometimes. @EliW

CODERCRUISE

7 days at sea, 3 days of conference

Call for Speakers Open Now

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

July 16-23, 2017 — **Tickets $250**

www.codercruise.com

SWAG

php[architect]

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

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