www.phparch.com

October 2016
VOLUME 15 - Issue 10

php[architect]

Pillars of Development

API Tips From the Front Line

The Strangler Pattern, Part One

Juggle Arrays Using Functional Callbacks

Dev Divas: History's Heroines of Computing, Part One

ALSO INSIDE

Education Station:
Automate Tasks Away With Robo the Faithful Task Runner

Community Corner:
Conference !== Family Reunion

Leveling Up:
Meddling in Middleware

Security Corner:
Two-Factor Authentication Is Just a Text Away

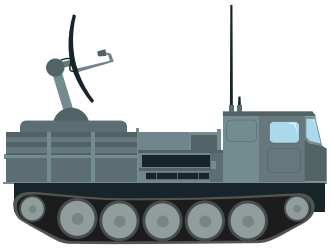
finally{}:
COTS, Open Source, Build it Yourself, or Something in Between?

php[architect]

CONTENTS

OCTOBER 2016
Volume 15 - Issue 10

Pillars of Development



Columns

Editor-in-Chief: Oscar Merida

Art Director: Kevin Bruce

Editor: Kara Ferguson

Technical Editors:
Oscar Merida, Sandy Smith

Issue Authors:
Edward Barnard, Chris Cornutt, Anna Filina, Cal Evans, Andrew Koebbe, Vesna Vuynovich Kovach, David Stockton, Matthew Setter, 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

Pillars of Development

I recently decided to write a non-trivial program in Python to brush up my skills. At first, I felt very out of my depth, but slowly I began to connect how arrays, objects, classes, and functions work in Python with what I knew from PHP. Yes, there are differences, but there are more things in common. If you have a good grasp of the fundamentals, learning another language is easier. You can focus on the differences in syntax and idioms. Remember, you're never done learning if you are a programmer.

It's tempting to always chase what's new and shiny. Whether it's an upcoming RFC for PHP or the next JavaScript framework that will revolutionize front end development, you can spend a lot of time keeping up with it all. There's certainly no shortage of something new to check out. That's why it's important to take a step back at times, revisit what you know, and challenge your assumptions. You may need to re-think how you approach a problem so that your code is easier to read and maintain. A proven way to deepen your understanding of a topic is to teach others. Prepare a presentation for your colleagues or your local PHP user group. And always, be aware of the wider programming community which exists outside your favorite language or framework. You'll gain an appreciation for the pioneering work done by others and its impact on your own career. You might also discover that seemingly new problems were already encountered, and tackled, by someone using another approach.

As in other programming languages, arrays are a fundamental data structures in PHP. If you're still just looping through arrays to manipulate them, then Andrew Koebbe's article is for you. In *Juggle Arrays Using Functional Callbacks*, he'll show you how to replaces `foreach` loops with functional programming paradigms to write simpler code that is easier to understand.

In *Dev Divas: History's Heroines of Computing, Part One* Vesna Vuynovich Kovach digs into the significant contributions women made to developing computing as we know it. We owe a significant debt to them, read her article to learn about the women that shaped our professions.

Before you design your next API, Anna Filina shares her *API Tips From the Front Line*. It's packed with valuable advice on how to structure your URLs, manage result pagination, keep it secure, and more.

In our fourth article, Ed Barnard describes *The Strangler Pattern, Part One* and how he's using it to add functionality to a legacy codebase. See how he and his team keep their main application humming (and making money) while using queues and services to offload jobs and add new features.

Also in this issue, Matthew Setter looks at a PHP native task runner in *Education Station: Automate Tasks Away With Robo the Faithful Task Runner*. David Stockton explains how to implement middleware in *Leveling Up: Meddling in Middleware*. In *Community Corner*, Cal explains why *Conference !== Family Reunion* and shares his advice on making conferences more welcoming for new attendees. Chris Cornutt continues his look at implementing 2FA in *Security Corner: Two-Factor Authentication Is Just a Text Away*. This month, he looks at using SMS codes to authenticate your users. To close the issue, Eli White asks *COTS, Open Source, Build it Yourself, or Something in Between?* He wants to hear how you approach this decision.

What Did You Think?

Do you have a question about an article? Want to share feedback about this issue? Let us know! Leave a comment on the issue page[1], reach us via twitter @phparch, or ask our contributors via twitter or email.

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.

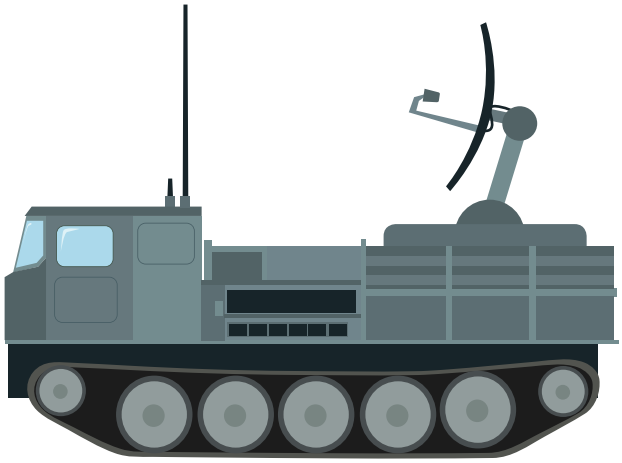
- Subscribe to our list: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download this Issue's Code Package:

http://phpa.me/October2016_code

1 *issue page: https://www.phparch.com/magazine/2015-2/october/*

API Tips From the Front Line



Anna Filina

Starting to write an API is an easy task, but you quickly stumble upon many obstacles and hard decisions. How to manage result pagination? How to handle write operations, file uploads and authentication? In this article, I will share tricks that allowed me to ship high-profile projects in record time, while keeping the code clean and maintainable.

By now, you have probably already heard about RESTful web services. Although they're easy to write at first, you will quickly hit obstacles that will leave you scratching your head and wondering how other developers would do it. Let me share some advice that I acquired while building and consuming countless RESTful APIs. At the end, you will be well-equipped to create elegant and pragmatic APIs.

I will use the term *client* to refer to the application consuming the API, such as a browser or a mobile app, and the term *user* to refer to the end user of the aforementioned application.

Input and Output

The first thing that I recommend you do is make all the decisions concerning the input and output format, as well as organize your endpoints. This will enable the front-end developers to start working on their part without waiting for you to finish implementing the back end. This parallel approach allowed me to ship projects in record time.

There are multiple ways to provide input: in the URL itself, in the HTTP verb, in the body and in the headers of the request. The body should be reserved for submitted data, when you are creating or editing a record. I typically reserve the headers for only a handful of things, such as language, authorization and concurrency control, which I'll get into a bit later.

Endpoints

For a basic list of products, I simply use the plural word as the endpoint (URL):

```
/products
```

If I want to apply some filters to the results, I pass them in the query string.

```
/products?category=games,movies
```

Notice that I can provide multiple values using a comma. The category doesn't necessarily correspond to a database column of the same name. It can have a different name or even be in a different table.

For performance reasons, you want to return a small payload to the client. However, the client might sometimes need additional data on each record. In this case, you can offer such flexibility through the `include` field.

```
/products?include=platforms
```

This adds data about the platforms on which the game can be played, such as the name and icon.

Sorting can be achieved through the `sort` field. The minus sign is shorthand for reverse or descending order.

```
/products?sort=-date
```

Finally, we have pagination. You want the client to specify which page and how many items per page. For performance reasons, make sure to put a limit in the pageSize. This is part of keeping the payload small but prevents someone from exhausting your memory or CPU resources when they request all the data.

```
/products?page=2&pageSize=100
```

All of the provided parameters should have defaults. For example, by default, you might want to fetch the first page, 20 records at a time, and sorted by date.

To fetch an individual record, use a similar endpoint with a slug or id:

```
/products/1
```

I always keep the plural version in the first part, because that avoids confusion. It also makes it much easier to implement for both the backend and front-end developers.

Verbs

Based on the verb, the same endpoint can have different behavior. This is where you leverage HTTP verbs. For example, to create a new record, you would `POST` on `/products`. To update an existing record, you would `PUT` on `/products/1`. To delete a record, you would `DELETE` on `/products/1`.

Headers

I am Canadian, so most of my APIs are bilingual. If you provide your data in multiple languages, then use the `Accept-Language` header in your request. Example:

```
GET /products/1
Accept-Language: fr-CA
```

Output Formatting

You can also return data in different formats, such as JSON, XML, or even HTML, based on the `Accept` header:

```
GET /products/1
Accept: application/xml
```

The good news is that it's not as hard to implement all of these different output formats. You're parsing the same parameters and you're querying the database in the same way. The only thing that changes is the way you decorate your data before output, which can be mostly automated.

Preventing Race Conditions

I mentioned concurrency control. You will have a race condition when multiple clients decide to update the same record after they already fetched it, and the last client will override anything that the previous clients sent. One way to avoid it is a combination of an `ETag` and an `If-Match` header.

The server should send the `ETag` when the client requests the record. I typically create an md5 from the JSON representation of the record.

```
ETag: "2dab91576380ded8b8e6349efe371f73"
```

The client should then use that value whenever it requests a record update:

```
PUT /products/1
If-Match: "2dab91576380ded8b8e6349efe371f73"
```

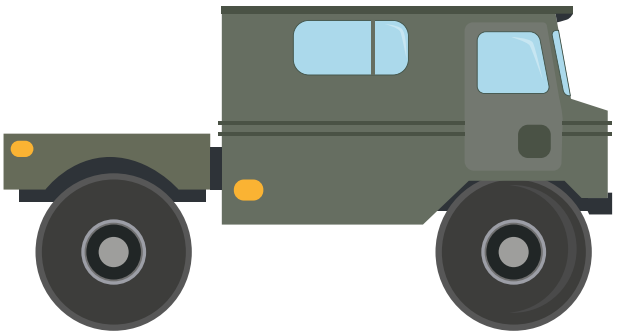
On the server, you'll have to check whether the record matches the md5 checksum. If it does not match, it means that the record has since been modified. You should reject the request with the status code 412 (PreCondition Failed). The client will then need to fetch a newer version of the record, then notify the user that they would be overriding those changes.

Body

When you POST or PUT, send data in the body. I see many people using `Content-Type: application/x-www-form-urlencoded` and sending data inside headers, when they instead should be using `Content-Type: application/json` and sending data inside the body. Yes, both requests and responses use the body. Here is an example of a POST request:

```
POST /products
Content-Type: application/json; charset=UTF-8

{
   "data": { "name": "Overwatch", "price": 49.95 }
}
```



Output

If you're returning a list, wrap the array in a `data` property. This way you can have additional properties where you can put the number of pages and total number of results or anything else that might come up in the future.

```
{
   "data": [
      {"name": "Skyrim"},
      {"name": "Civilization V"}
   ],
   "pagination": { "pages": 50, "count": 100 }
}
```

You can always avoid that by sending the pagination data inside the headers, but that makes it harder for some developers to use. If you open up your API to third parties, this can cut costs on technical support.

For a single record, the response would follow a similar pattern:

```
{
   "data": { "name": "Skyrim", "price": 29.95 }
}
```

Versions

Don't forget to version your API. No matter how much you think that you nailed it, the API will evolve. You have multiple choices for versioning it. Passing headers is more RESTful:

```
Accept: application/vnd.myapi.v1+json
```

But URLs are simpler for some developers:

```
/v1/products
```

Once you decide how to handle your input and output, stick to your decisions. Consistency is key!

Uploads

I have seen many developers struggle with uploads. How to attach an avatar to a profile update? There are two main methods: send it separately or mix it with other data. In any case, I do not recommend using multipart uploads.

You will need to send it separately when you can't obtain the file's bytes prior to the final request. Let's say that the user uploads an image using the provided drag-and-drop widget. The image is immediately uploaded to an endpoint `/images`. The response would include the identifier `63955`. When the user clicks to save the profile, you would send the following to the API:

```
PUT /profiles/afilina
Content-Type: application/json; charset=UTF-8

{
   "data": { "name": "Anna Filina", "avatar": 63955 }
}
```

In cases where you already possess the file's bytes, you can just encode them and include them in place of the identifier:

```
PUT /profiles/afilina
Content-Type: application/json; charset=UTF-8

{
   "data": { "name": "Anna Filina", "avatar":
"/9j/4AAQSkZJ…" }
}
```

In PHP, getting the bytes and encoding them would look like this:

```
$bytes = file_get_contents('avatar.jpg');
$encoded = base64_encode($bytes);
```

The tools that I commonly use for uploading files are Plupload[1] for the front end and Guzzle[2] for the backend.

Status Codes

Don't confuse HTTP status codes with API status codes. This is very important, because many technologies rely on proper HTTP statuses to function as expected. Send the HTTP code appropriate to the situation, and then add your own custom API code in the body.

For example, if you're validating a record, you can have a variety of outcomes. If the input cannot be parsed, such as with an incorrectly formatted JSON, then you would return a `400 (Bad Request)`. If you're able to read the input, but the validation failed, such as when a name is empty or a description is too long, then you would return a code `422 (Unprocessable Entity)`. Along with the status code, you would then send a body containing the error code and message:

```
Status: 412

{
  "errors": [
    { "code": 1001, "message": "Name cannot be empty."},
    { "code": 1002, "message": "Description is too long." }
  ]
}
```

Security

Before I go further, I have to ask you to always use HTTPS. It doesn't matter how secure every other part of your application is—at some point, you'll have to send user credentials over the wire. Second, I highly recommend that you let your framework handle the security part. Implementing security protocols on your own is hard. A respectable framework with hundreds of contributors is a safer bet.

Authentication and Authorization

Letting people into your API using credentials consists of two phases: the authentication and the authorization. Authentication only happens once. It is when the user sends you credentials, such as a username and password, or a secret API key to an endpoint such as `/auth`. Example:

```
{
   "data": { "username": "afilina", "password": "phparch" }
}
```

The API will validate the credentials and issue a temporary token. That token will be sent with every request until it expires. If it expires, you will have to repeat the authentication process and issue a new temporary token. The duration of the token depends on how often you want your users to re-enter their credentials.

1 *Plupload:* http://www.plupload.com

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

The *least* secure way that you can authorize your users is by sending a username and password with each request. First, it will slow down your API, because authentication takes more time than authorization. In addition to hurting your performance, you'll have to store those credentials for easy access on the client side, usually somewhere insecure. Another reason is that hackers can now intercept any request from the client to your API to try and steal credentials, instead of just the initial one. Finally, if there's a third party in the mix, such as an application accessing your API on the user's behalf, then that means trusting the third party with your user's credentials.

There are many ways to handle authorization. You can even use multiple methods, picking the appropriate method based on what the client sends you. I will only cover one in this article.

JSON Web Tokens

This authorization method is one of the simplest to understand and implement, and is suitable for most needs. Once the user credentials have been validated, the server will respond with a JSON Web Token[3]. Example:

```
{
   "data": { "jwt": "eyJhbGciOiJIUzI1NiIsInR5cC·" }
}
```

This completes the authentication step. The client will then store that token and append it to every subsequent API request like this:

```
GET /my-profile
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cC·
```

The API will read that header and decrypt it. No need to look up the user in the database to validate the token or to check permissions. Literally everything the server needs is right there in the token. It's worth noting that the client does not possess the means to decrypt the token, so it cannot be modified to give additional permissions. Only the server can decrypt it.

You can find a more in-depth explanation of JWT online. There are even multiple PHP libraries you can choose from to help you with the task.

Authorize Everything

I strongly recommend that you enforce authorization even for public data. This will enable you to throttle requests and disable abusive users.

Imagine that a normal user makes 30 requests per minute. Suddenly, you notice someone making 300 requests *per second* for the past few hours, in a predictable sequence. This usually means that someone is extracting all of your data. If you wish to prevent that, all you need to do is disable the keys being used. If it fits within your business model, you can throttle such requests and send them an email offering to raise the request limit for a fee. Turn spam into revenue!

Here's another situation. You notice that someone is hitting your API using a multitude of accounts. It's at the point where your servers can't even handle all the requests and legitimate users can't do their business. That's a denial-of-service attack. Using an automated banning technique, you can reject their requests before you even query for the data. At the authorization

3 *JSON Web Token:* https://jwt.io

phase, check whether that key is banned and immediately respond with an error. If it becomes a recurring problem, then you can store a list of banned keys in Redis or an equivalent, so you can drop their requests even faster. Find a way to spend as little time as possible handling bogus requests.

Testing

I usually use two approaches to test APIs. I test individual methods using unit tests and entire endpoints using Guzzle. If you're not familiar with Guzzle, it's an easy way to send HTTP requests.

Guzzle

Guzzle tests are the very first thing that I write for an API, right after I finish deciding on the endpoints and data structure. Those tests establish a contract between the API developers and any applications that will consume it. Say you have a front-end team. If you both agree on the endpoints and JSON, you can both start working simultaneously. They will know what the API will look like, so they can start building around dummy JSON files. You will know exactly what you need to build, because your tests will fail unless it's exactly as expected.

In the test shown in Listing 1, we instantiate the Guzzle Client and we send a request to an endpoint. We assert that the status code is `200 (OK)` and that the output contains that exact JSON string.

LISTING 1

```
01. public function test_GetProduct_ReturnsSuccess()
02. {
03.     $client = new Client();
04.     $response = $client->get(
05.         'http://api.test/products/1',
06.         [
07.             'exceptions' => false,
08.             'headers' => ['Accept' => 'application/json'],
09.         ]
10.     );
11.     $this->assertEquals(200, $response->getStatusCode());
12.
13.     $body = $response->getBody()->getContents();
14.     $this->assertJsonStringEqualsJsonString('{
15.         "data": { "name":"Civilization V", "price": 39.95 }
16.     }', $body);
17. }
```

If you haven't written any code yet, then the status code will be 404 and the test will fail. If your output is different, the test will also fail. Once the test passes, you're done and the front-end developers can switch from dummy JSON files to this endpoint. You'll both work faster and there will be no need to tweak anything to make the two parts fit together.

Of course, this will require creating a test database that is different from the development database. You want to control the data inside it with great precision, in order to avoid constantly fixing your tests. That data, also called fixtures, will have to be reloaded each time you run the tests. Don't forget to write unit tests as well for the classes and methods.

Performance

Performance issues happen to the best of us. Here are the most common causes.

Non-Essential Tasks

Surprisingly, most time is usually lost not while fetching the data, but doing everything before it. Bootstrapping the framework, regenerating the cache, going through many security layers, etc. Check how long each process takes and put it in a diagram.

I recently profiled a legacy application where 868 milliseconds were spent generating URLs, because the routing structure was very complex. That is extremely wasteful, especially when 100 percent of it was taxing the CPU. The solution was to cache that in Redis. Whenever the framework would try to generate a URL based on given parameters, I would first check in Redis whether I have a URL string for that set of parameters. 247 milliseconds were spent on auto-loading classes. The solution was to write a mechanism that would cache the paths to the classes on disk. That cache would not need to be regenerated in production except after deployment.

Too Many Queries

I have, in the past, profiled an application that executed over 3,000 database queries on a single request. That calls for a query logger, which you will then analyze to find similar queries.

Imagine the following scenario. You fetch a list of records. To display these records, you need to fetch some data in a related table. That means that you need to iterate over these records and execute one or more query for each one. Toss a nested loop in the mix and you're doomed. You don't remember writing code like that? Frameworks often do what's called lazy loading. You would fetch a list and call something like this:

```
foreach ($photos as $photo) {
    $photo->Album->User->name;
}
```

The solution is to use database joins in order to fetch the associated data at the same time as the main records. This way, you only execute one query and get everything you need. However, I'm not saying that you should always get as much data as you can in a single query. Sometimes, executing many small queries is faster than a big one if the network latency is low, so you'll have to test different combinations.

Too Much Data

Sometimes, you just forget to paginate. Always limit the number of results. Do it at the query level as opposed to cutting off after N results in PHP. This is because you would be making your database work and return a large result set for nothing.

Other times, you may be fetching too much data. You may be joining on unnecessary tables or fetching columns that you won't use. I strongly recommend that you carefully craft your queries instead of offloading all of that to the framework. You can use a framework's ORM and still write your queries exactly the way you want them.

Hydration

Hydration is the process by which an ORM copies SQL results into a complex object structure, like in the earlier photo example. This operation can surprisingly take a long time and consume a lot of memory. More often than not, it makes sense to ask your ORM to fetch results as an array.

Continuous Performance

If you test for performance just once, then you'll have a few unpleasant surprises along the way. I recently rescued a project—it was in production for years and was well-optimized. One day, the team deployed a tiny change to the code, but forgot to test the performance. The result was a near-disaster, which prompted them to call me for an emergency intervention, saving millions of dollars and their reputation in the process.

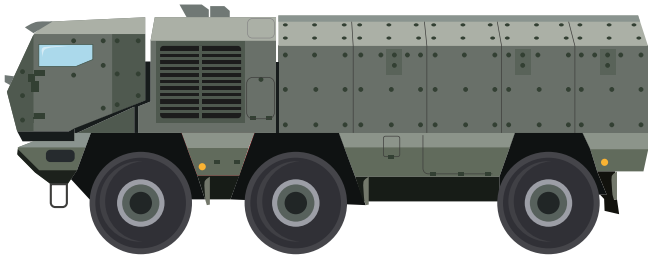
In addition to load-testing large sites before a deployment, you can take simple steps to include performance in your test suite. In development mode, add an additional block to your output:

```
{
   "perf":
      { "time":0.3120, "db_time":0.0036, "memory":19661 }
}
```

You can then have your automated Guzzle tests also check for those metrics and warn you if there's a sudden spike in execution time or memory usage.

I hope that you enjoyed this advice and that you're itching to incorporate it into your projects. You should now be better equipped to write your RESTful APIs.

Anna Filina is a web developer, project rescue expert, Pluralsight author, speaker and conference organizer. She enjoys realizing seemingly impossible things. @afilina



Dev Divas: History's Heroines of Computing, Part One

Vesna Vuynovich Kovach

Picture a software engineer. Did you think of a man? Me too.

When I went looking for stories about women in the history of computing, I found many surprises such as this one. I had no idea the degree to which women shaped what our profession—our world—looks like today. In this article series, I'll share just a few of these stories with you. Here's how I came to start looking for them.

For starters, consider that the software engineer who *invented that very term*[1] is named Margaret Hamilton. As the lead flight software designer for NASA's Apollo and Skylab space missions, she wrote *the code that got the Eagle to the moon and back*[2]. We have her to thank for some of our *fundamental concepts of error checking, end-to-end testing, and asynchronous software*[3]—not to mention a *respectable job title*[4].

Losing Women from Computing

Thanks to the recent recession, I found myself with a lucky break I had never dreamed of. A federal "dislocated worker" program offered me free tuition that gave me a second shot at a goal I'd abandoned years before: computer programming.

The first time around, I'd gotten part way through a second bachelor's degree that I'd hoped might open more doors than my philosophy major had. Then I gave in to the nagging feeling that somehow I didn't belong in the field. No matter. This time I was determined.

Even so, I wondered how I'd fit in, a middle-aged returning student.

Waiting for class to start on the first day, I chatted with the student on my left, a man who looked to be in his sixties. He told me about his years as an enterprise resource planner in South America and his hopes for a new start here in the United States. In another class, I observed the student on my right as he twisted around to prop a lanky elbow on the desk behind him and reminisced with a chum about his high school prom. Suffice it to say, his boutonniere was still fresh—that's how young he was.

But maybe I wouldn't stand out after all, at least not in this community college classroom with its broad spectrum of ages.

But then, I noticed something else about the class demographics, something unnerving. Out of dozens of students in each of those classrooms, I was one of only a few women. The ratio was about the same in the next class, and the next, and in all the semesters to come. And later, too, in the workplaces and professional communities I eventually entered as a software developer.

It turned out I had joined the ranks of the *17 percent*[5]. That's roughly the proportion of women in computer science classes in the United States.

This just didn't seem possible. When I tried to picture my old programming classes, I couldn't remember them being so unbalanced in this specific way. Perhaps because I'd felt like an oddball in so many ways back in those days, I simply hadn't noticed? My COBOL teacher had been a youthful woman, I recalled. Maybe that had somehow skewed my impressions? Surely women's presence in computer tech had only increased over time. Hadn't it?

It had not.

In the mid-1980s, women made up around *one-third of software engineers*[6], *students*[7], and *others in computing*[8]. But over

1 *From Her Code Got Humans on the Moon—And Invented Software Itself,* http://phpa.me/wired-margaret-hamilton

2 *The source code is at* https://github.com/chrislgarry/Apollo-11/

3 *From NASA Honors Apollo Engineer,* http://phpa.me/nasa-honors-apollo-engineer

4 *From Margaret Hamilton, the Engineer Who Took the Apollo to the Moon,* http://phpa.me/margaret-hamilton-interview.

5 *See The Current State of Women in Computer Science,* http://phpa.me/women-in-cs

6 *See The Current State of Women in Computer Science,* http://phpa.me/women-in-cs

7 *NPR: When Women Stopped Coding,* http://phpa.me/npr-women-stopped-coding

8 *Gender Codes: Why Women Are Leaving Computing, 1st Edition* https://www.amazon.com/dp/0470597194

the next few years, the percentage of women crashed. I didn't know it then, but I was right on trend when I walked away from computer programming. Although women were 37 percent of the computer workforce in 1991, today they make up 26 percent overall, with just 11 percent of technical officers in private, venture-backed firms and 7 percent of tech startup founders, according to the *National Center for Women & Information Technology*[9].

Researchers and advocates have spent many years and considerable resources puzzling out this phenomenon and trying to figure out what to do about it. Among many others, the book *Gender Codes: Why Women Are Leaving Computing*[10] (ed. Thomas Misa, IEEE Computer Society Press, 2010) documents some of these efforts in fascinating detail, and the excellent film *Code: Debugging the Gender Gap*[11] (Robin Hauser Reynolds, 2015) explores several hypotheses that have emerged.

When the Personal Computer Came Home

The reasons appear to be myriad. When the proportion of women studying programming and going into the field dropped, it didn't happen in a vacuum. It was a particular moment in history: the moment computers exploded into private life, when interacting with them became specific and intimate. No longer were they just the "giant brains" of popular imagination that insurance companies used to generate billing statements or to which scientists posed their inscrutably technical questions. Computers were suddenly friendly, approachable—even, sometimes, fun. You could play games on them in arcades, where they displaced the mechanical shooting galleries and pinball machines. You could buy one of your own and put it in your own living room or perhaps even put it in your child's bedroom. With his own computer, who knew what he could grow up to be?

And "he" was generally the right pronoun. The personal computers that flooded into households starting in the 1980s were far more likely to go into a boy's room than a girl's. A boy in a given family was more likely to be encouraged to use the household computer than was his sister.

In its early years, the computer game industry created games with little thought to demographics; marketing wasn't specific to any gender or age group. By the late 1980s, it was aimed directly at teenage males.

Fictional computer virtuosos flourished in mainstream movies. Real ones headlined the news. Many people got excited about the possibilities of a future in computing. But there was a catch. Not all types, it seemed, were eligible for such a future. Because, for the first time ever, the public at large learned, from Hollywood and newsrooms, what a "computer person" was supposed to look like. It didn't look like me—or like my COBOL teacher for that matter.

We all learned together, as a society. You could be socially awkward like *The Revenge of the Nerds* crew or a boyish heartthrob like Matthew Broderick in *War Games*. You could be a junk-food crunching misanthrope like Wayne Knight's Dennis Nedry in *Jurassic Park* or a good-looking family man like Miles Dyson in *Terminator 2*. You could be a slick Steve-like Jobs, a teddy-bear Steve like Woz, or a billionaire with a bad haircut like Bill Gates.

But to be brilliant enough to wrangle a computer, there was one thing you could not do without: a Y chromosome.

A first generation of boys grew up with computers in the home, access to them, parental encouragement to use them, and fun things to do with them. They even had their pick of role models to emulate.

By the time the girls of this era got to college, few had the pluck to enroll in computer classes in the face of mountains of social messages that told them they didn't belong there. Those who did found that their male peers were more likely to be ahead of them in experience and confidence. Some report being shamed by professors for their lack of know-how. Even a high-school math whiz, unaccustomed to facing a teacher's scorn, might find herself confronted and belittled. You can hear true stories along these lines in NPR's *Planet Money Podcast #576*[12], also broadcast as a Morning Edition story, *When Women Stopped Coding* from October 2014.

Many young women turned their talents to relatively (emphasis on the "relatively") friendlier fields, like science or medicine—fields where the female-to-male ratio continued to equalize. Or, like me, perhaps they decided that only the liberal arts would suit.

Over the years, computers became something that everybody uses for work and play, for creativity and connectedness, for science and sport, for reference, for archiving, for everything under the sun we humans do. Only a special subgroup of humans get to figure out how to make computers do all these things. Since the late 1980s, nearly all of them have been men. But *for forty years before that*[13], *that wasn't true*[14].

What Happened to the Women Pioneers?

I was struck by a new question. I wondered, with so many women in computing, just what had they accomplished? No household names among them—could that be simply because none of them had done anything important? I needed to find out.

I set out to learn what contributions women had made to our world of computer tech. I thought I would find a handful of marginally relevant anecdotes. I thought it would be hard to find this information. Wrong. I found a lot. And it turned out to be incredibly easy to do so.

Here's why.

9 *Did You Know: Demographics on Technical Women,* http://phpa.me/demographics-technical-women

10 *Gender Codes: Why Women Are Leaving Computing:* https://www.amazon.com/dp/0470597194

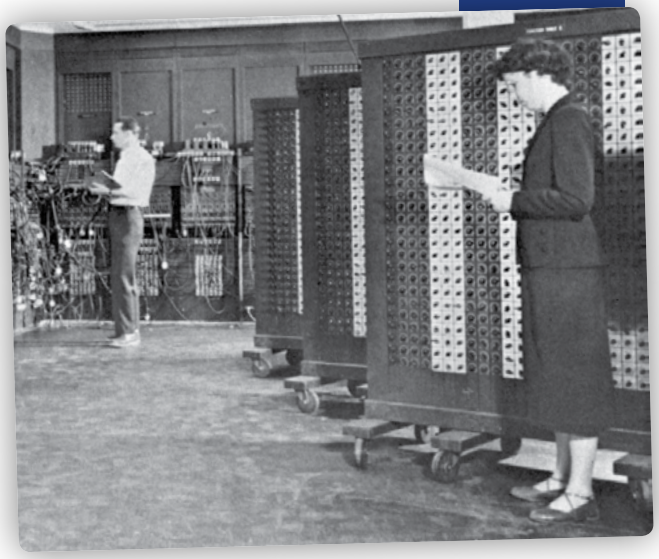
11 *Code: Debugging the Gender Gap:* http://www.codedocumentary.com

12 *Planet Money Podcast #576:* http://phpa.me/npr-women-stopped-coding

13 *Gender and Technology: A Reader,* http://phpa.me/gender-tech

14 *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise (History of Computing),* https://www.amazon.com/dp/0262517965

Betty Holberton programming the ENIAC **FIGURE 1**



U.S. Army Photo, Public Domain, https://commons.wikimedia.org/w/index.php?curid=55124

Grace Hopper and UNIVAC **FIGURE 2**

By Unknown (Smithsonian Institution) - Flickr: Grace Hopper and UNIVAC, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=19763543

Frances "Betty" Snyder Holberton

First, I sat down at my computer to do some research. That was simple enough. All I had to do was tap on a typewriter-like keyboard, thanks to Frances "Betty" Snyder Holberton. Back in the 1940s, Betty was sick of having to spend hours physically rewiring ENIAC, the first electronic computer, for every new program she wrote. So she established the interface convention that made it so easy to get my project going. Betty also wrote the first sorting routine ever and invented things like the flowchart, with the set of arrows and shapes we're all familiar with, but that's another story. The best account I've read of Betty's life, including the foundational technological contributions for which the IEEE Computer Society awarded her its Computer Pioneer Award in 1997, I found in the book *Pioneer Programmer*[15], the autobiography of one of her ENIAC programming colleagues, Betty Jean Jennings.

I didn't need to know a thing about exactly where my computer was putting the stuff I was typing, partly because of the wonderful precedent Betty had set when she *co-wrote the first computer language, BINAC*[16], and partly because of the compiler, which Grace Hopper invented just a few years later.

Grace Murray Hopper

Rear Admiral Grace Murray Hopper started programming even before Betty did, working with the Harvard Mark I, the world's first general-purpose computer. She made it her life's work to promote computers, in part by pushing hard to make them easier to use. In the 1950s, after creating her A-O compiler, Grace went on to write the first natural-language style programming language. By the end of the decade, she had pulled off a Herculean (at the time, most considered it Sisyphean) task: guiding a fractious web of committees of programmers and theorists from the sciences, industry, academia, business, the government, and the fledgling Association of Computing Machinery in a multidisciplinary effort to create a COmmon, Business-Oriented Language. You've heard of it: they gave it the acronymic name COBOL. Grace died in 1992, but billions of lines of her brainchild are live today; it's by far the majority of live code on the planet. Every day, COBOL runs hundreds of times more transactions worldwide than there are Google searches.

Of course, I wasn't using COBOL myself. But I benefited from its development and from everything Grace Hopper did. Early in the game, she changed the prevailing course of computer development by explicitly setting the goal of making computers as easy to use as possible, by as many people as possible, for as many purposes as possible. She succeeded so thoroughly, it's hard now to imagine this could ever have been controversial. The best introduction to her electrifying presence is probably the 1986 *David Letterman show clip*[17] I found on YouTube, and the best summation of her enormous influence on the world you and I inhabit today is the talk her biographer *Kurt Beyer gave at Google*[18], also available on YouTube. If you like watching that, you'll also want to read his book, *Grace Hopper and the Invention of the Information Age*[19]." She was a powerful force advocating for women to get into computer careers. Fittingly, the world's largest gathering of its kind is named for her, with over *15,000 women expected to attend*[20] this year's *Grace Hopper Celebration of Women in Computing*[21].

15 *Pioneer Programmer:* https://www.amazon.com/dp/1612480861

16 *Programming pioneer Betty Holberton is born, March 7, 1917,* http://phpa.me/edn-holberton

17 *David Letterman show clip:* https://youtu.be/1-vcErOPofQ

18 *Kurt Beyer gave at Google:* https://youtu.be/lY6Hk8_eiKs

19 Grace Hopper and the Invention of the Information Age

20 *Attend GHC 16: What you need to know for GHC,* http://ghc.anitaborg.org/2016-attend/

21 *Grace Hopper Celebration of Women in Computing:* http://ghc.anitaborg.org

Radia Perlman **FIGURE 3**

Hedy Lamarr **FIGURE 4**

Radia Perlman

Thanks to Grace and Betty, as easily as I could have typed a sentence onto a piece of paper a century ago, I entered the requests with which I consulted a vast interconnected network of machines.

That network was thanks in large part to Radia Perlman, an MIT alumna who these days works for a Dell subsidiary, EMC. While working at Digital Equipment Corporation, Radia invented the spanning tree, which is used in the data link layer, the lowest in the Internet Protocol Suite (TCP/IP). The spanning tree protocol is what made it possible to put many computers on an Ethernet network, a missing puzzle piece so critical that *sometimes Radia gets called the "mother of the Internet," even though she has said she finds that title "embarrassing."*[22] She's been named one of the "20 Most Influential People in the Industry" twice by Data Communications magazine, she holds 100 patents, and her textbook *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*[23] is considered a classic.

Hedy Lamarr

My own computer wasn't connected by Ethernet though. I was using my laptop, which communicated with my wireless network courtesy of the spread-spectrum technology patented in 1942 by Hedwig Eva Maria Kiesler, under the name "Hedy Kiesler Markey"[24], a young Austrian who had fled to America to escape her controlling husband, the wealthy owner of a munitions factory. For four years, she had played a glamorous dimwit for the engineers and scientists who had visited their mansion to discuss what they were developing for Mussolini and Hitler so she could surreptitiously learn weapons technology.

Hedwig moved to Hollywood, adopted the screen name "Hedy Lamarr," earned wild success as a movie star, and patented a system for undetectable guided torpedoes. She offered the Navy the invention, which it badly needed, along with the offer of a data dump—everything she'd learned about the Axis weapons efforts. The Navy waved her away, only to muddle through the entire war without undetectable guided torpedoes.

Twenty years later, an engineer stumbled across Hedy's by-then-expired *US Patent 2,292,387*[25] (which includes the name of George Antheil, whom she brought on as a collaborator) while researching ways to implement some sort of wireless technology. In it, he found her fully realized description of frequency hopping—now more commonly called spread-spectrum—the technology that underlies Wi-Fi, CDMA, and Bluetooth.

It was yet another 20 years before anyone realized that this was the same invention that had been assumed to be useless, presumably because it sprang from the mind of the actress known as "the most beautiful woman in the world." In 1997, the Electronic Frontier Foundation awarded Hedy Lamarr its Pioneer Award. In 2014 she was inducted into the National Inventors Hall of Fame.

And that was how come my laptop was able to communicate with the Internet. Cordless was a nice bonus for my Bluetooth headphones and RF mouse, too. Thanks, Hedy. You can learn more in *Hedy's Folly, the Life and Breakthrough Inventions of Hedy Lamarr*[26] by Richard Rhodes.

22 *Women in Information Technology (Major Women in Science),* https://www.amazon.com/dp/1422229289

23 *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols:* http://phpa.me/interconnections-book

24 *Confused by all the names associated with her? Hedwig Eva Maria Kiesler is the name she was born with. The patent is in the name Hedy Kiesler Markey. Hedy is a common nickname for Hedwig and Markey is her second husband's last name.*

25 *US Patent 2,292,387:* https://www.google.com/patents/US2292387

26 *Hedy's Folly, the Life and Breakthrough Inventions of Hedy Lamarr:* https://www.amazon.com/dp/0307742954

Adele Goldberg

What really made the mouse handy, though, was using the point-and-click desktop metaphor that came courtesy of Adele Goldberg and Alan Kay's GUI. They invented that revolutionary interface back in the 1970s when they were working at Xerox Palo Alto Research Center, known as PARC. Maybe you've heard the story about how the higher-ups at Xerox forced the PARC team to let Steve Jobs and his Apple gang visit their labs and ask all the questions they wanted. Adele, the manager of the PARC Systems Research Lab, fought the hardest against this, so the story goes, and only let them in under protest.

Thus, thanks to Adele, and her ideas that got incorporated into the Macintosh and eventually all personal computers, I used a mouse to click on links for my research. When I found promising information, I just highlighted it with my mouse and, also thanks to Adele, I copied the text in my web browser and pasted that exact same data right into another application. Both of them were running at the same time! Navigating between the applications was so easy, too, because with Adele's method of overlapping windows, I just clicked on one or the other.

Speaking of the Mac, remember how a third of the computing world used to be female? That held true for the original 12-person Apple Macintosh software programming team, which included Susan Kare, Patti Kenyon, Joanna Hoffman, and Rony Sebok. You'll look in vain, mostly, to see any of these Mac team members represented in the avalanche of biopics that depict Apple's early years. A notable exception is the 2015 *Steve Jobs* movie that includes Joanna Hoffman, played by Kate Winslet. The facts about the physicist/archaeologist who drafted the Mac's User Interface Guidelines and later joined Jobs at NeXT correspond only loosely to the movie, but that's true of most of that film.

Adele Goldbert at PyCon 2007 **FIGURE 5**

By http://www.freesoftwaremagazine.com/articles/thrills_chills_and_pictures_from_pycon_2007, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=39825753

By the way, *Susan Kare*[27] may possibly be the most influential single person behind the visual language we use in our everyday interactions with technology, so she really helped me keep my bearings as I switched between windows—opening this, minimizing that, and drawing around this other thing to take a screenshot.

Transcending the chunky command-line pixel clusters of the day, Susan designed the typefaces that lay at the heart of the Mac's startling new eloquence. The menu font Chicago, for example, and the graceful monospace Geneva were integral to the revolutionary Mac look and feel. With her expressive, yet succinct iconography—the lasso tool, the paint bucket, the happy and sad Macs, and later, the familiar Windows control panel elements and app icons—Susan established the standard for the visual conventions we expect in modern UI.

Conclusion

All this helped make it easy to find out what I wanted to know about the history of women in computing. I was using the very technology developed by the women I was learning about. We all owe so much to the revolutionary work of these and so many more brilliant women—women who made core developments in computing technology.

In the next installment of this two-part series, I'll share some stories about some of the greatest computer collaborations of all time. You'll meet history's first computer team, a trio of intellectuals living in 18th century France, back when a "computer" was a human being who did math. And yes, this team was one-third female. I'll tell you about history's first computer scientist, and as a bonus, science's first "scientist." **Spoiler alert:** both were women. We'll get a better look at Betty Snyder Holberton and the first electronic computer and meet the rest of her astonishing team.

Vesna Vuynovich Kovach is part of the small, passionate, agile development team at OfficeSupply.com. She runs DevDivas.com, a site that celebrates the history of women in technology, and tweets at @dev_divas.

27 *Susan Kare:* http://kare.com/about/

The Strangler Pattern, Part One

Edward Barnard

Picture By: https://commons.wikimedia.org/wiki/User_talk:Siripatwongpin

Our large PHP application at InboxDollars works well and generates revenue. Unfortunately, we're locked into an old version of CakePHP. We've considered the cost of rewriting our ancient beast to use CakePHP 3 and modern PHP. The effort is not worth it to us since the rewrite does not bring any additional revenue.

We created a way to write new feature code using CakePHP 3 (or any other modern framework) while leaving our main application intact. We follow the Strangler Pattern using microservices. Part One explains how the pattern works and begins our implementation.

Revenue

Sometimes it's hard to look ahead. InboxDollars.com[1] has been around 16+ years. We have a relatively large monolithic application which *is* our website. We continuously add and split-test new features. We have grafted-on support for our in-house mobile app.

About InboxDollars

An Inc. 5000 company, we provide millions of members direct cash rewards for their online activities, such as reading emails, taking surveys, playing games, and signing up for offers. Advertisers, market research companies, etc., pay us for our members' activity, and we pass through a percentage of that revenue to our members as cash payouts.

Our purpose in adding features, naturally enough, is to improve company revenue. We evaluate any undertaking on its potential for improving our revenue picture. If we were to consider a non-revenue-producing project, we have the fact it's not producing revenue *and* the opportunity cost of not doing something else that *would* produce additional revenue.

This is the classic setting for loading ourselves down with technical debt. As a small team of eight developers, we can get by with shoving a few more lines of code in where needed. There's little incentive to restructure the code base when just "getting it done" is perfectly sufficient. Which reminds me about this wisdom about *The 10x Engineer*:

> *How to be a 10x engineer: Incur technical debt fast enough to appear 10x as productive as the ten engineers tasked with cleaning it up.*
>
> *– Brian Degenhardt*[2]

Kick the Can

Now, we're stuck. The transition from our current CakePHP 2 code base to using CakePHP 3 is a "breaking" migration. We have little incentive to rewrite our CakePHP 2 code to be CakePHP 3 compatible. There's no revenue increase from doing the rewrite, but it keeps us from doing something that *does* bring more revenue. We're not in business to do rewrites for the fun of doing rewrites!

We have a similar situation with PHP. We've nearly completed migrating from PHP 5.3 to PHP 5.4. However, because the `mysql` driver (as opposed to the current `mysqli` and `PDO` drivers) will be removed, migrating past PHP 5.4 will involve costly rework.

For example, the "escape string" behavior between the `mysql` and `mysqli` drivers is quite different for input which might be null or an empty string. When we feed the "escaped" string into a regular expression, everything breaks because we get apostrophes when none were expected. These issues are in our oldest, most fragile, code which remains central (mission-critical) to our business. We *can* rewrite the code which uses `mysql`, but so far the effort is not worth it to us.

1 *InboxDollars.com:* http://www.inboxdollars.com

2 *Brian Degenhardt:* http://phpa.me/bmdhacks-10x

One option is to continue to "kick the can down the road." We're quite good at continuously grafting additional functionality onto our existing code base. Our code becomes more fragile with ever-greater risk of unexpected side effects. Code development becomes slightly slower. However, given the choice between the slight slowdown and the full-stop of a rewrite, we prefer the slowdown.

Refactoring

Isn't this where refactoring comes in? That is, refactor our code to work with CakePHP 3 and PHP 5.6 (or PHP 7)? No, it isn't, and that's the point of this article! We'll be focusing on *new* feature code and creating a means of dealing with that old code.

Our Solution

At InboxDollars we now make the distinction between the member-facing website and any processing which can be done outside that context.

For example, if a member clicks on his or her "recent earnings" page, we need to produce a view of those recent earnings. We're doing work in PHP and in our database queries which directly impacts the member experience.

On the other hand, things like statistics gathering and logging do not directly affect the page load. One of our member activities is market research (taking surveys). We attempt to provide our member the best survey for that person at that moment. We have various scoring algorithms behind the scenes aimed at providing the best experience, but the scoring processes themselves are not member facing.

Our general approach is this: if the work (the PHP/MySQL code) is not member-facing, if its result is not needed for the current page load, hand it off. Thus, we try to move as much work as possible away from our member-facing servers and our member-facing databases. The databases are not directly member-facing. I'm referring to the member-facing PHP code which uses those databases.

Much of our offline work is plain batch processing. We have reporting, scoring, and crediting jobs which run nightly. On their own they're not very interesting, and I won't mention them further. But you can see in Figure 1 a basic flow for dealing with crediting jobs in nightly batches.

We also have on-demand processing. This is the focus of our solution. For example, we have a "TV Channel" where members get paid to watch short videos with advertisements. We process the member credit on each view. Given there must be at least 20 seconds between credits, we have plenty of time to process the credit offline before the next view. So, rather than do the credit processing and database inserts before the member-facing page load, we "hand off" the credit work to a batch server.

Our solution focuses on the new batch server running PHP 5.6, CakePHP 3, with access to our RabbitMQ servers, MySQL servers, etc. The existing batch servers run PHP 5.3 and CakePHP 2.

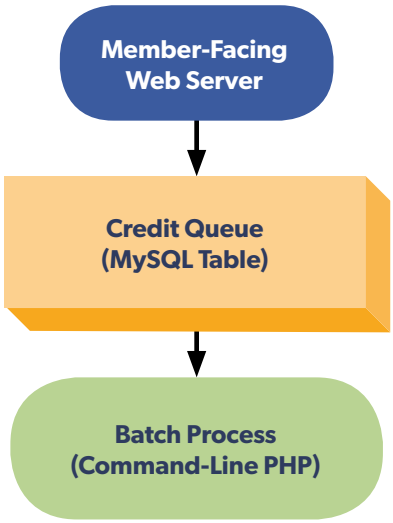
The Wheel

I'd prefer to not re-invent the wheel. I'm quite happy to make use of other's experiences. We're about to take a tour of other peoples' work, which will then bring us to our solution.

Bear in mind the problem we're trying to solve:

- We have a relatively large application which is perfectly happy and generating revenue.
- This application locks us into old software levels.

Table-based work queue **FIGURE 1**



- A full rewrite generates no revenue *and* keeps us from writing other things which *would* generate revenue, thus the full rewrite is not a favorable option.

We are building a solution that:

- Allows us to move to modern software and stay with up-to-date software henceforth.
- Provides low risk.
- Allows us to grow and evolve.

Why Now?

At InboxDollars we are in the process of scaling out our processing capacity. We are scaling out those TV Channel views from 50,000 views per day to allowing for 600,000 views per day. At the same time, we're greatly increasing the amount of information tracked for each of those views.

This expected scale hits the "tipping point" in several areas. We need to consider server load, the number of database connections, internal network connections, and sheer data volume having the possibility of overrunning our SAN storage.

Scaling for an expected 10x increase in traffic is clearly a revenue-producing project. Having done the math, we know we're pushing the tipping point. This presents a win for a certain bluntly opportunistic developer, namely myself.

We have the green light to begin to design for scale. Ready? Let's do this. We begin with Martin and Cindy Fowler's visit to Australia.

Reality vs. Optimism

When we are currently seeing around 50,000 views, and I talk with our DevOps team about the capacity needed for more than ten times the number of views, DevOps quite rightly asks if it will actually occur.

In one sense, *I don't care*. I'm tasked with writing the feature supporting the target 600,000-view-per-day capacity. If I write code which only handles our current traffic level, and we tip over on the way to reaching 10x traffic, I've failed. On the other hand, if I write code to handle 600K views/day and our traffic does not increase, it's not my problem. Or is it?

The answer is we need to consider the continuum. We don't want to waste resources in over-engineering a solution. We want the magic combination of something which handles our current traffic level without wasting resources, but can be scaled out to handle 10x traffic and more—and handle it comfortably.

I'm reminded of the old joke, *What do you call a good PHP programmer? Employed.* In other words, companies rely on us. Our job is to build something which will serve well in a workmanlike manner and continue to evolve and serve well for its lifespan.

The Strangler Application

Martin Fowler reported when he and Cindy went to Australia, he observed an interesting phenomenon in the rain forests of the Queensland coast. The huge strangler vines seed in the upper branches of the fig trees. Over time they cover, strangle, and kill the fig tree.

Martin has spent much of his career doing rewrites of important systems. In 2004, he proposed *The Strangler Application*[3] as a low-risk approach to doing a rewrite of an important system. His article describes gradually creating "a new system around the edges of the old, letting it grow slowly over several years until the old system is strangled."

His article paints such a vivid picture it's easy to miss his final point:

> *"There's another important idea here–when designing a new application, you should design it in such a way as to make it easier for it to be strangled in the future. Let's face it, all we are doing is writing tomorrow's legacy software today. By making it easy to be strangled in the future, you are enabling the graceful fading away of today's work."*

Next, Sam Newman continues that final point, explaining we need to design software capable of evolving and remaining current.

The Evolutionary Architect

Sam Newman, in *Building Microservices*[4] creates *An Evolutionary Vision for the Architect*:

> *Our requirements shift more rapidly than they do for people who design and build buildings—as do the tools and techniques at our disposal. The things we create are not fixed points in time. Once launched into production, our software will continue to evolve as the way it is used changes. For most things we create, we have to accept that once the software gets into the hands of our customers we will have to react and adapt, rather than it being a never-changing artifact.*
>
> *Thus, our architects need to shift their thinking away from creating the perfect end product, and instead focus on helping create a framework in which the right systems can emerge, and continue to grow as we learn more.*

Think of "framework" in the larger sense of a software architecture. We'll be creating a means and approach to implementing new features.

Newman continues to describe the evolutionary architect:

> *We should think of our role more as town planners than architects for the built environment… A town planner's role is to look at a multitude of sources of information, and then attempt to optimize the layout of a city to best suit the needs of the citizens today, taking into account future use.*
>
> *The way he influences how the city evolves, though, is interesting. He does not say, "build this specific building there"; instead, he zones a city… Rather than worrying too much about what happens in one zone, the town planner will instead spend far more time working out how people and utilities move from one zone to another…*
>
> *This is why I try to stick to the guideline that we should "be worried about what happens between the boxes, and be liberal in what happens inside."*

3 *The Strangler Application:* http://www.martinfowler.com/bliki/StranglerApplication.html

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

FIGURE 2 **Strangler vines (*Ficus watkinsiana* on *Syzygium hemilampra*, Iluka, Australia, by Peter Woodard)**

The Value of Conferences

The ideas for implementing this approach at work have been germinating for some time. Last March, I attended Midwest PHP 2016[5]. Mike Stowe, @mikegstowe[6], gave a talk, *Services in the Enterprise: How Not to Fail.* Part way through the talk I realized this might be a good approach to solving our problem of the moment. In that talk, he mentioned the book *Building Microservices*[7] by Sam Newman.

Two more speakers brought my possible solution into focus. Samantha Quiñones , @ieatkillerbees[8], shared *Building Real-time Data Pipelines* and *Manage Your Content with Elasticsearch*. Mathew Beane, @aepod[9], caught my eye with *Serious Log Crunching and Intelligence Gathering* so I sat in on *ELK: Ruminating on Logs*. He kept things creative with his diagrams of stacked ELK and log runs!

A few months later I'm here sharing our solution with you. I can't speak highly enough of the unexpected collisions of ideas which happen at our conferences. They can be our PHP community at its best.

The Single Responsibility Principle

I got it wrong. When I learned the SOLID[10] principles of object-oriented design, I knew that *S* stands for *Single Responsibility Principle*[11]. "Uncle Bob" Martin introduced the term, and Wikipedia reports Uncle Bob expressed the principle in his book as:

> *A class should have only one reason to change.*

So, naturally, I thought that's what *Single Responsibility Principle* means. But wait, there's more!

There's something I've been doing for years because it has intuitively seemed the right thing to do. Where it makes sense, I organize unit tests by *feature* because my software development tends to be feature-by-feature, not module-by-module.

On the same principle, at InboxDollars I try to put all the "business logic" code for a new feature into one folder off to the side, rather than scatter that code around the existing code base. In my experience, it is the *feature* which needs to change, as opposed to a certain web page or database table.

Uncle Bob, who introduced the term in the first place, posted an outstanding article in 2014. He explained the Single Responsibility Principle[12] is really about people and organizations. It's a great read!

His conclusion was my *aha!* moment. I'd gotten it wrong. Watch carefully as Uncle Bob explains:

> *Another wording for the Single Responsibility Principle is: Gather together the things that change for the same reasons. Separate those things that change for different reasons.*
>
> *If you think about this you'll realize that this is just another way to define cohesion and coupling. We want to increase the cohesion between things that change for the same reasons, and we want to decrease the coupling between 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 many different people care about for different reasons.*

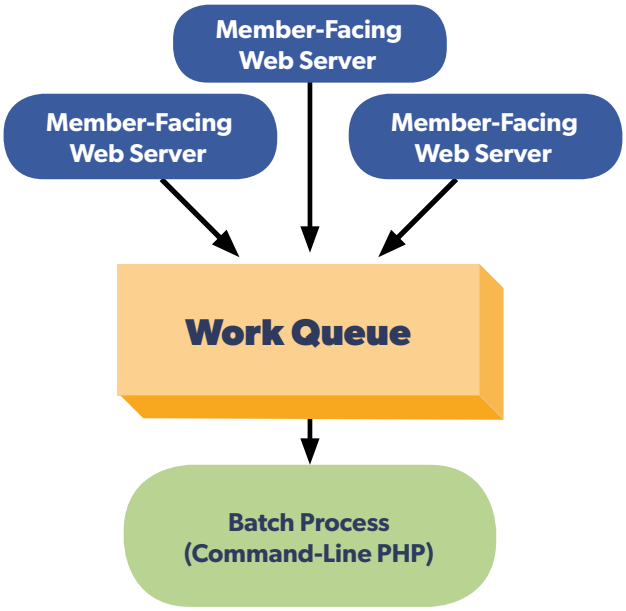
The Work Queue

At InboxDollars we've been using work queues for a year or two. That's an important consideration: several of our developers now have experience with work queue programming. Our DevOps team is likewise familiar with the queues' capacity, behavior, and performance.

If you're not familiar with using work queues, they allow you to process tasks outside of the user request. It takes quite a shift in thinking to get your head around them. We'll explain how they work in a later article in this series.

It's the work queue which allows us to offload work from the member-facing web servers and hand the work off to our batch processes.

Work queue **FIGURE 3**



For example, we log information for every web service request for later reporting and analysis. We originally had the web service request write the log information to the database during the web service request. Now, instead, we package the information and send it to the call log work queue.

5 *Midwest PHP 2016:* https://twitter.com/midwestphp
6 *@mikegstowe:* https://twitter.com/mikegstowe
7 *Building Microservices:* http://samnewman.io/books/building_microservices/
8 *@ieatkillerbees:* https://twitter.com/ieatkillerbees
9 *@aepod:* https://twitter.com/aepod
10 *SOLID:* http://phpa.me/solid-oo-design
11 *Single Responsibility Principle:* https://en.wikipedia.org/wiki/Single_responsibility_principle
12 *Single Responsibility Principle:* http://phpa.me/8th-light-srp

We have now sped up the user request in two different ways:

- We have reduced the number of database writes during the page load, which reduces the total elapsed time of this user request, and
- We moved the table out of the member-facing production database, thus reducing that database load, allowing the server to process all requests more quickly. The table now resides on a separate database server.

Our batch server has a command-line PHP (CLI) script which processes the log requests as they arrive from the work queue. The queue is large enough it can handle a "burst" of web service activity while waiting for the batch script to catch up.

Our Strangler

Let's put together our requirements, concerns, and chosen methods. We'll have to leave details for later articles, but we can lay out our starting point here.

Stay With the Familiar

Our development team has been working with CakePHP "since forever." Let's play to our strengths and continue using CakePHP as our framework. Specifically, we'll jump to CakePHP 3 and PHP 5.6.

Remember, CakePHP has breaking changes between CakePHP 2 and CakePHP 3[13]. The biggest change is in the data (ORM) layer. It disappeared! To be sure, it's been replaced with something better, but given a very large proportion of our code is in that Model layer, what matters to us is it's gone.

I'm just beginning to gain experience with CakePHP 3. The rest of our development team has none. That, obviously, is a barrier to ongoing rapid development. However, CakePHP has a code generator which will generate the Model (data) layer for us. Given we all know how CakePHP works, once we see the data layer code as generated, we can take it from there.

On the bright side, CakePHP 3, unlike CakePHP 2, is fully integrated with Composer. This means we can fully embrace the PHP ecosystem, use PSR-4 loading, and build code as Composer packages.

Most of our development team now uses PhpStorm. This gives us easy access to static analysis tools. Given our full team is familiar with the concept, we can depend on the PHP code sniffers and formatters to keep us up to date with the PSR-2 and CakePHP coding standards.

Ongoing Breaking Changes?

When dealing with a version upgrade which contains significant (to you) breaking changes, it's important to consider whether you're in a cycle of breaking change after breaking change. The Strangler Pattern helps here by creating code which expects to evolve over time.

Fortunately for us, the CakePHP core developers have stated CakePHP 4 will not be a "huge breaking release like 3.x."[14]

Microservices

At InboxDollars we usually work in terms of new features, or in changes to existing features. We don't usually make a general change to "the members controller." Instead we might, for example, tweak the member signup process and evaluate the change with a split test.

Remember, the "strangler" is not the whole tree. We're not looking for a rewrite. Instead, we are focusing on new feature work.

For a given new feature, what can be offloaded? That is, what part of the functionality can be handed off for batch processing? We want to structure the new feature code so the maximum amount of processing can be handed off. By offloading work, we should also be keeping the member-facing web servers as responsive as possible.

We are not only offloading the PHP work. We're offloading the database as well. Any data not needed for member-facing responses need not be in our primary database. Our web service call log, for example, can be on a different database server inaccessible to our member-facing servers.

We know at this point that we'll be structuring the offloaded work as microservices, but we've not described what that actually means. We'll be exploring this microservices architecture in upcoming articles.

Education

Pretty much everything we're considering is new for our development team. We've outgrown our Beanstalk job queues and are moving to RabbitMQ[15] as our messaging system.

With a single person developing our Strangler Pattern, we have the usual "hit by a bus" risk or *bus factor*. More importantly (from my standpoint, given being hit by the bus removes all concern on my part), if the other developers don't know how to use our new Strangler Pattern infrastructure, they won't. I'd feel awful if I were hit by a bus, but I'd feel worse knowing nobody was using the stuff I wrote.

To help mitigate this risk, I got permission to do a weekly demo for our development team. That's a large time investment for everyone. I tried to keep it to a half hour, but it quickly grew to an hour.

This has proven to be a useful strategy. In fact, this weekly education session is a secret part of our Strangler Pattern! We've used whiteboard sessions explaining purpose, concepts, and most importantly message flow.

We now have several features scheduled to use our new Strangler Pattern infrastructure.

13 *CakePHP 3 Roadmap,* https://github.com/cakephp/cakephp/wiki/3.0-Roadmap

14 *CakePHp 4 Roadmap,* https://github.com/cakephp/cakephp/wiki/4.0-Roadmap

15 *RabbitMQ:* http://www.rabbitmq.com

Outgrowing Beanstalk

We've been doing Producer/Consumer programming for over a year using Beanstalk as our queueing system. This approach has tremendously improved our production system. However, we've had a few difficulties where jobs got stuck in the queue and can't be removed; and where Beanstalk tipped over entirely (everything stopped).

We were not able to get much help with our issues, except to see other sites seem to have had similar problems in recent years. We found no indication Beanstalk is actively supported, by anyone. We noted Redis[16] and RabbitMQ[17], by contrast, offer Enterprise-level support.

With these difficulties in mind, I attended Matthew Turland's, @elazar[18], talk *Effective Redis for PHP Developers* at php[tek] 2016. After the talk I explained we were considering moving from Beanstalk to either RabbitMQ or Redis as our queueing system. He suggested I talk to Gary Hockin, @GeeH[19], at the PhpStorm table. Gary knows Beanstalk.

I described our situation to Gary, and he said "you have outgrown Beanstalk" and RabbitMQ is the correct answer for us. Gary walked me over and introduced me to James Titcumb, @asgrim[20], who happily answered questions for me.

Again, I can't recommend the PHP developer conferences highly enough!

Make It Look Easy

Our new Strangler Pattern methodology is this:

- Identify the parts of new feature work which can be offloaded to our batch system.
- Use our batch system architecture (which has not yet been explained here).
- Wherever practicable, use CakePHP 3 (or plain PHP) and PHP 5.6.
- Write code which may be easily kept up to date as time moves on.

Our new batch system (which I've not yet described) has infrastructure for monitoring, getting work to the batch processes, and so on. Our other developers don't need to know the batch system internals. Instead, we need to know how to write new feature code which *uses* the new batch system.

Any new system, by definition, is more difficult to work with than the current familiar system. It's crucial, therefore, to make the system as easy as possible for other developers to use.

Assuming you are one of our developers ready to create a new batch feature, here are the steps you would take:

1. Using Composer, create a new CakePHP 3 project. (See *Projects and Packages* next.)
2. Add the Composer dependencies bringing in our batch system support libraries.
3. Add the Composer commands to run the unit tests, reformat code, and check the code against the coding standards (lint check).

To make it as easy as possible for our developers to get started, I wrote a tool which does the above. You can also use it to create a "plain" Composer package, based on The PHP League's skeleton project[21]

I released our tool under an Open Source (MIT license) on GitHub: ewbarnard/StranglerStartingTool[22]. It's a teaching tool rather than a production tool. It shows how we incorporate our own packages for our own environment.

Your details and framework-of-choice will be different as you create your own Strangler Pattern implementation. Please do take ideas from my tool as to how to make the starting point as simple and automated as possible.

Projects and Packages

Our Strangler Pattern implementation takes to heart Uncle Bob's principle, *Gather together the things that change for the same reasons. Separate those things that change for different reasons.*

Our main web application is structured as a large single CakePHP 2 project. By contrast, each new Strangler feature becomes its own separate CakePHP 3 project. That project might have several different batch processes, all supporting the same feature. Separate features, therefore, become separate CakePHP 3 projects. Be sure you read Newman's *Building Microservices* as you take this approach. There are advantages, but there are pitfalls.

We have common code which now gets spread across multiple CakePHP 3 projects. Each project uses the same code to get jobs in and out of RabbitMQ, interact with the Strangler Pattern infrastructure, and so on.

These things become separate Composer-compatible packages which each project can include. I use the above-described "starter" tool to start a new project with those dependencies included.

Some of our Composer-compatible packages are plain PHP, and one is a CakePHP plugin. This gives us our starting point for each new-feature project.

Given we have common code shared by the separate projects, that does mean we have a shared dependency between those projects. Each project, in theory, should be independent of every other project. This allows each project to be deployed independently.

However, these interdependencies relate to two areas:

- Our operating environment, such as the fact we have four site domains, and three operating environments (development, staging, production), and
- Interacting with our Strangler Pattern infrastructure.

Because these interdependencies simply reflect our fundamentals of software development, we consider the common code a good solution.

16 *Redis:* https://redislabs.com

17 *RabbitMQ:* https://www.rabbitmq.com/services.html

18 *@elazar:* https://twitter.com/elazar

19 *@GeeH:* https://twitter.com/geeh

20 *@asgrim:* https://twitter.com/asgrim

21 *The PHP League's skeleton project:* https://github.com/thephpleague/skeleton

22 *ewbarnard/StranglerStartingTool:* https://github.com/ewbarnard/StranglerStartingTool

Summary

We chose the Strangler Pattern as a way to write new features with modern PHP without having to rewrite our primary web application.

We can't apply the Strangler Pattern to everything. We specifically apply it to new feature work which can be offloaded to our batch system. Any other code continues to be folded into our main web application.

We make it as easy as possible for our development team to take advantage of this pattern. We conduct a weekly demo/whiteboard session during development. This means all developers are included and educated. We created a simple tool to automate the "starting point" for each new Strangler feature.

Trust plays a large role here. As we observe our Strangler Pattern in practice over the next year, we learn to either strongly trust it or strongly distrust it. Assuming the former, we will naturally gravitate to greater use of the approach.

Will we choose to migrate parts of our primary CakePHP 2 application to CakePHP 3? I don't see it happening. On the other hand, we are aware of countless areas where we can offload existing work to the batch system. Continued growth will continue to force us to find more ways to offload work.

Looking Ahead

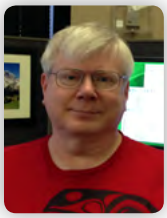
Now that we understand The Strangler Pattern and have a game plan, how do we implement it? My upcoming articles comprise a real-world case study of how we are applying this pattern at InboxDollars and discuss some of the potential pitfalls.

In *Beginning to Design for Scale With RabbitMQ* we will look at distributed processing. When your web application outgrows a single server, you begin to encounter logjams, queues filling up, and other problems of scale. Code becomes more complicated. We'll look at our InboxDollars batch system structure and design decisions.

The Rhythm of Test-Driven Development may come as a surprise. Test-Driven Development (TDD) would seem to be all about the tests. However, TDD's greatest value comes in the future. The Strangler Pattern insists software must remain agile, open to change. The Rhythm of TDD often called "red-green-refactor," provides that value.

Unit Test Design With Mockery is inevitable. When your PHP code must work through other classes, functions, APIs, and databases, those dependencies become a formidable challenge to writing your unit tests. Any time you write "real world" code you may find yourself spending an hour getting structures set up for a three-line test. Things easily get out of hand. We demonstrate strategies to use in keeping your unit test development sane.

Producer-Consumer Programming in CakePHP/RabbitMQ brings our case study full circle. We will discuss how to choose what work to offload and show you how we scaled our site to 10x capacity. We use CakePHP and RabbitMQ to show concepts which apply to any framework.



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.

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

Juggle Arrays Using Functional Callbacks

Andrew Koebbe

Arrays are a fundamental data structure in almost every programming language. We use them every day, but are we limiting ourselves to just using a handful of basic array functions and control structures? Are we missing more explicit and readable iterations of arrays? Using functional programming paradigms, we can open up a new world of simplicity and power for common array iteration strategies. It's time to get out of the `foreach` rut.

When I started developing, functions that took callbacks were like magic. I understood you could use other functions as part of these methods, but it either seemed too complicated, or I couldn't find a practical use for them. Then, I started working in JavaScript and began to use anonymous functions (or lambda expressions), a whole new world opened up to me. Suddenly, these PHP callback functions made sense; I had no idea I could do anonymous functions in PHP. It was "game on" for a bit of functional programming in PHP.

Many PHP functions that take callbacks are used on arrays and can help you develop more intentional and readable code. Before diving into these functions and learning how we can use functional paradigms, some readers may find a primer on using callbacks and anonymous functions helpful.

Primer

First, let's review the basics of callbacks and anonymous functions. Callbacks have been around since PHP 4 and the `create_function` function. They allow us to do some pretty impressive things, but at first glance it may look like some sort of Russian nesting doll trickery.

Callback Functions

There is nothing special about callback functions; what is special, are the functions that accept a callback as a parameter. Essentially, we are passing in a reference of a function(such as a string matching a function name which is in scope) to use within the calling function. One special function to understand is `call_user_func`. This function will accept a reference (like a string) to another function and execute it.

```
function myFunction () {
    return 3 + 2;
}

call_user_func('myFunction'); // returns 5;
```

The `call_user_func` executes `myFunction` and passes the return value on. This can be used to make a function with callback capabilities.

Let's say we want to make a function which will uppercase and print out the value from another function. We'll make a regular function, `ucReturn`, which will take a parameter referencing the function we need to get the return value from. Next, we'll use the `call_user_func` function to call the function from the parameter and pass through the return value, which we'll assign to a variable to uppercase.

```
function ucReturn($fn) {
    $return = call_user_func($fn);
    print strtoupper($return);
}
```

Now, we can pass in the name of the function we want to uppercase the return value.

```
function getHello() {
    return 'hello';
}

ucReturn('getHello'); // outputs: 'HELLO'
```

With traditional callbacks, we use a string of the callback function's name as the parameter. Once we understand anonymous functions, we can combine the two and go a step further.

Anonymous Functions

PHP introduced anonymous functions (also known as lambda functions) in version 5.3. At the time of this release, I was a lone dev in an IT department and had no connection to the outside PHP world (my fault), so it wasn't until the past few years that I learned of these magical little functions.

In our callback example above, we've defined a first-class function (i.e. a named function):

```
function getHello() {
    return 'hello';
}
```

This is pretty straight forward. We have a function called `getHello` which only returns the string `"hello"`.

An anonymous function is a function without a name. We can change the `getHello` function to an anonymous function

Picture of Juggling balls: William Warby from London, England (http://www.flickr.com/people/26782864@N00)

in two ways. The first way is to store this function in a variable. We can then call the function with the variable reference as if it were a standard function.

```
$helloFn = function () {
    return 'hello';
}

$helloFn(); // returns 'hello'
```

Notice the function doesn't have a name in this example. The function is directly assigned to a variable and can now be passed around like any other variable. How does this help us? Well, for this topic, they can be used as callbacks. Let's change our callback example to use an anonymous function assigned to a variable.

```
$helloFn = function () {
    return 'hello';
}

ucReturn($helloFn); // outputs: 'HELLO'
```

Rather than passing in a function's name as a string for the callback, we passed in a variable which has an anonymous function for its value. There is one more step we can take to make things more readable and intentional by using inline anonymous functions.

Inline Anonymous Functions as Callbacks

We can combine callbacks and anonymous functions by putting our anonymous callback code directly in the function call. This keeps the callback code embedded in the function call. We can see this by taking our previous callback example and putting the `getHello` function code directly into the `ucReturn` call.

```
ucReturn(function(){
    return 'Hello';
}); // outputs: 'HELLO'
```

This brings the entire logic of the task in one command. We no longer have to hunt around for the callback function, it's right there in the main function call!

Now, we can take this knowledge and use it with some of PHP's array functions to eliminate a lot of the `foreach` implementations we've used in the past.

Callback Enabled Array Functions

PHP has a number of great array functions that take callbacks which can make your code more explicit, intentional, and readable. Let's start out with the functions I find most useful.

In all of the following examples, we'll assume we're working on a hypothetical B2B system and using the `Customer` class defined in Listing 1.

array_filter

Many times an array will have more objects than are needed, certain objects are not valid for our purpose. We need some way to filter out the array elements we don't want. When we use a `foreach` loop, we have to set up an empty array before we begin looping. Then, pick out the elements we want and put them into the empty array.

LISTING 1

```
<?php

class Customer
{
    /** @var int */
    public $id;
    /** @var string */
    public $name;
    /** @var \DateTime $createdDate */
    public $createdDate;
    /** @var string */
    public $status; // 'active', 'inactive', or 'lead'

    public function __construct($id, $name,
                                \DateTime $createdDate = null,
                                $status = 'active') {
        $this->id = $id;
        $this->name = $name;
        $this->createdDate = $createdDate ?: new \DateTime();
        $this->status = $status;
    }
}

$customers = array(
    new Customer(
        1, 'Widgets International  ',
        new DateTime('2016-05-01')
    ),
    new Customer(
        20, '  Good Buys',
        new DateTime('2016-02-01'), 'inactive'
    ),
    new Customer(
        5, '  Space Padding Co.  ',
        new DateTime('2016-01-01')
    ),
    new Customer(
        60, 'Money Bags R Us',
        new DateTime('2016-06-01'), 'lead'
    ),
);
```

Let's say we have our array of customers and we want to filter the array to only "active" status customers. Using a `foreach` loop would look something like this:

```
$activeCustomer = [];
foreach ($customers as $customer) {
    if ($customer->status == 'active') {
        $activeCustomer[] = $customer;
    }
}
```

Instead of the `foreach` loop above, we can use the `array_filter` function. The `array_filter` function takes the array to be filtered and the callback function as parameters, and returns a **new** array. The callback function will take each array element as its parameter, and the return value will be evaluated as a boolean: `true` if the element should stay, `false` if the element should be removed. Let's take the above example and implement it with `array_filter`.

```
$activeCustomers = array_filter(
    $customers,
    function ($customer) {
      return $customer->status == 'active';
    }
);
```

This change took six lines of code down to three and once you're used to the syntax, it becomes easier to read. There is one caveat: `array_filter` maintains the keys from the original array, while our `foreach` example resets the keys. Keep in mind the `foreach` example can be tweaked to add the key into the loop to achieve the same result as the `array_filter` implementation. If you want the key order to be reset in the `array_filter` example, you can always use `array_values` to reset the keys.

Dude, Where Are My Keys?

In our first `array_filter` example we only have access to the `$customer` value of the array element and not its respective key. There is an optional flag which can be passed as a third parameter to `array_filter` which was added in PHP 5.6. By default, the callback function parameter is the element value, but if we set this flag to the constant `ARRAY_FILTER_USE_KEY`, that parameter will be the key and not the value.

```
$activeCustomers = array_filter(
   $customers,
   function ($key) {
      // $key is now array key from $customers
   },
   ARRAY_FILTER_USE_KEY
);
```

Perhaps we need access to both the key and value to filter our array. There's a flag for that! In this case, if we use the constant `ARRAY_FILTER_USE_BOTH` we will now have two parameters for our callback.

```
$activeCustomers = array_filter(
   $customers,
   function ($customer, $key) {
      // Have access to both the $customer AND the $key
   },
   ARRAY_FILTER_USE_BOTH
);
```

I often find myself needing to filter arrays in my day-to-day work. It's extremely nice to be able to do so in a concise and elegant way using `array_filter`.

array_reduce

If we want to take an array and end up with a single value, then `array_reduce` is the tool for the job. For example, if we want a count of the number of active clients in our array, the `foreach` implementation would look something like the following code:

```
$active = 0;
foreach ($customers as $customer) {
    if ($customer->status == 'active') {
        $active += 1;
    }
}
```

This is pretty simple. The code loops through the array and if the status matches, it adds 1 to `$active`.

The `array_reduce` function can do the same thing in a more intentional way. The first parameter of `array_reduce` is the array. The second is a callback which must accept two parameters. The first parameter of the callback is the value being carried over to each iteration as we build our aggregate (a counter in our example) and the second parameter is the array element to consider. Lastly, a final parameter on `array_reduce` is the starting value of the carry variable. We want ours to start at 0.

```
$active = array_reduce(
    $customers,
    function ($carry, $customer) {
        if ($customer->status == 'active') {
            $carry++;
        }
        return $carry;
    }, 0
);
```

Here we are more explicit with our code. There's no ambiguity of a `foreach` loop that could be doing any number of things with the array. We know we are taking an array and returning one value.

usort, uasort, and uksort

PHP has no less than thirteen different built-in functions to sort the elements of an array. THIRTEEN! Nine of them sort by the key or the entire value. But, what if we want to sort an array of objects by a property? The non-callback, non-functional method we would choose would be `array_multisort`, which is a bear to work with. The first parameter will be a new single dimensional array of scalars (or that can be cast to a scalar) which will be sorted as a reference, and the second will be a single dimensional array of scalar values which can be easily sorted. The first parameter's elements must correspond to the elements in the array you want to sort. Basically, PHP will sort the first parameter and take the resulting order of those keys and do the same reorder to the actual array we want to sort. It should be noted that `array_multisort` modifies the actual array and does not create a new one. Let's say we want to order our `$customers` array by `createdDate`.

```
$sortValues = [];
foreach ($customers as $customer) {
    $sortValues[] = $customer->createdDate;
}

array_multisort($sortValues, $customers);
```

If you want to sort in another direction or specify the type of casting for the sort, you'll have to start passing in extra flags.

```
$sortValues = [];
foreach ($customers as $customer) {
    $sortValues[] = $customer->createdDate->getTimestamp();
}

array_multisort($sortValues, SORT_DESC,
                SORT_NUMERIC, $customers);
```

This can get confusing quickly, and the need to create another reference array for the sorting is tedious. Wouldn't it be nice to just tell the sort to use a property or some other custom logic? Well, I'm glad you asked! `usort` is a function which uses a callback to compare elements however you want. And, if we use a lambda right in the function call, everything is kept together in one command.

`usort` and its siblings, `uasort` and `uksort`, take a callback that is given two parameters for you to compare and return:

- An integer less than zero, if the first parameter is less than the second.

- An integer greater than zero, if the first parameter is greater than the second.
- Zero, if the two parameters are equal.

Our customer date sort will now look like:

```
usort($customers, function ($a, $b) {
    return ($a->createdDate < $b->createdDate) ? -1 : 1;
});
```

Here we are taking customer `$a` and comparing it to customer `$b`. If `$a`'s `createDate` is less than `$b`'s then we want it to come first. This will result in an ascending sort for the array. You can see we can really do any type of comparison logic in the lambda, which makes it more powerful than most array sort functions and much more intentional than `array_multisort`.

array_map

If you have an array and need to create a new array with values based on the original array, then `array_map` is what you're looking for. For example, let's say we have our `$customers` array, but what we need is a simple array with the `id` property as the value of each element. In this scenario we might be tempted to set up an empty array, loop through the `$customers` array, and push the `id` value:

```
$customerIds = [];
foreach ($customers as $customer) {
   $customerIds[] = $customer->id;
}
```

This results in a new array with the customer IDs as the values:

```
Array
(
    [0] => 1
    [1] => 20
    [2] => 5
    [3] => 60
)
```

`array_map` allows us to pass in a function which will take each element of the array as its parameter and the return value will be used as the value for the element in the resulting array. It should be noted `array_map` **creates a new array** and does not modify the original array (i.e. it does not pass the element in by reference). If you are looking for the latter, see the next section on `array_walk`. Let's rewrite the above `foreach` implementation using an anonymous function with `array_map`.

```
$customerIds = array_map(function ($customer) {
    return $customer->id;
}, $customers);
```

In the case of `array_map`, the function is the first parameter, and the originating array is the second. While this doesn't save us a ton of code, it is much more direct since we are setting the `$customerIds` to the result of the `array_map` and not having to track down where that array is added to in a `foreach` loop. But, `array_map` has a trick up its sleeve.

I was troubled at first that the function was the first parameter when other related functions take the array first. Initially, I just chalked this up to some oversight in a PHP RFC that no one caught until it was too late. After looking closer, I realized there was more power in this function. There is a third parameter to `array_map`. A variadic, no less!

You can pass in any number of arrays to this function and the elements for each of those arrays will be passed into your call back function as additional parameters.

In the following example, we end up with a new array with the customer IDs and corresponding statuses combined.

```
$customerStatus = array_map(
   function ($customer, $status) {
       return [$customer->id, $status];
   },
   $customers,
   ['active','inactive','lead','inactive']
);
```

array_walk

Similar to `array_map`, `array_walk` will operate on each element of an array, but in `array_walk`'s case, each element is passed by reference and does not create a new array. If we wanted to clean up the white space surrounding the customer names, we would do the following in a `foreach` loop:

```
foreach ($customers as $key => &$customer) {
   $customer->name = trim($customer->name);
}
```

`array_walk` takes the array as the first parameter and a callback to perform on each element as the second. An optional third parameter to `array_walk` will be available as a third parameter in your callback.

```
array_walk($customers, function (&$customer, $key) {
   $customer->name = trim($customer->name);
});
```

As with `array_map`, we aren't shortening our code, but we are being more explicit about what we are doing in our code. We know with `array_walk` we are modifying the elements of the array, and that can help others (and our future selves) understand what we're doing here.

Conclusion

Arrays are arguably one of the most important data structures in any programming language. Particularly in PHP, we use them all the time. `foreach` has its place and is a wonderful general purpose tool. But if we use it in every iteration situation, we aren't being intentional or helping the readability of our code for those who come after us, including our future selves.

Andrew Koebbe is a software architect and web developer currently focused on PHP and Symfony. He has worked in digital advertising and medical research fields. He is a Grav CMS contributor and presenter. He enjoys dabbling in Drupal, DevOps, and Raspberry Pi and Arduino hacking. He also enjoys to spending time with his wife and two kids, sous vide cooking, and watching way too much MST3k. @andrewkoebbe

Meddling in Middleware

David Stockton

In the past year or two, middleware approaches to writing web applications in PHP have grown quickly in popularity. It's quite a bit different in terms of how you think about structuring and organizing an application compared to MVC, but it also brings a lot to the table regarding simplicity and understanding. Today we'll be looking at middleware, what it means, and how to use it.

PHP has been through some fairly major changes over the years, not just in terms of the capabilities of the language, but in the way we write applications using PHP. In the early days, it was common to write scripts "top-down," mixing code and HTML in the same script. Later, functions were extracted into other files and `included` or `required` into the page, but code and HTML still were typically intertwined. With PHP supporting OOP, MVC became a way to clearly organize and structure code where everything had its place. While PHP MVC is not really MVC[1] as far as the pattern was originally designed, the benefits for organization and separation of code and markup are quite good. Other patterns have also emerged, such as ADR[2] (Action / Domain / Responder).

The pattern we're talking about today is middleware. The terms and concepts are not new—like MVC, they've both been around since the 1960s and 70s. Middleware provides a framework for transforming a request into a response. This is typically done via a pipeline which allows middleware to flow from one piece to the next, and then back out again. It's very similar to how a standard bit of code works with one function calling the next and the next, with the language tracking where we are in a stack. On the way back out, when the inner functions return, control returns to the caller. It is similar in middleware.

Middleware Structure

In general, a piece of middleware will receive a request and a way to call the next piece of middleware in the pipeline. In some frameworks, they also receive a response object as well. If you're interested in the debate over which way is superior, feel free to google for the PHP-FIG mailing list archives.

> **Editor's Note:** *For a discussion of the two implementations, see The Middleware Awakens[3] by Ian Littman in our June 2016 issue.*

In any case, middleware will typically have a signature like one of the following:

```
function __invoke(
    Request $request, Response $response, $next
) : Response
```

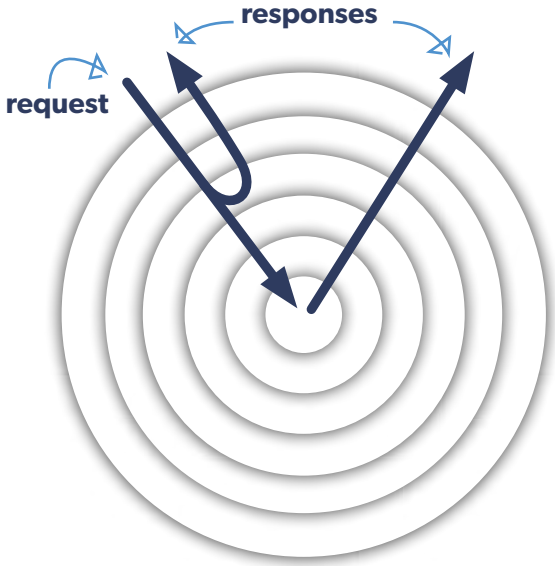
or:

```
function __invoke(
    Request $request, $next
) : Response
```

You may find frameworks which use method names like `handle` or `process` or something other than `__invoke`, but the idea is they'll receive a `Request` and a way to get to the next middleware, and they'll return a `Response`. In some frameworks, the `Response` is provided as an argument when the middleware is called.

A pipeline can be configured with a number of middleware pieces. Each middleware can be responsible for a single change or operation, or they can do more. To make things simple, though, my recommendation is not to do too much with any individual piece of middleware. As I mentioned before, the flow through the middleware typically passes in through each piece, moving to the next until the final middleware is reached. It returns a `Response` object which is passed back through the middleware in the opposite order in which they were called. This gives the original caller an opportunity to manipulate the request on the way out. In the case of an error happening, the flow can detour off into a set of error handling middleware.

Typical Middleware Flow **FIGURE 1**



1 *MVC: https://en.wikipedia.org/wiki/Model-view-controller*

2 *ADR: http://paul-m-jones.com/archives/5970*

3 *The Middleware Awakens: https://www.phparch.com/magazine/2016-2/june*

Figure 1 illustrates a couple of ways a flow may work. Consider each circle as a piece of middleware. The request comes in through the outermost or first piece of middleware which can manipulate the request in some way and then pass it to the next level. This proceeds until the final piece of middleware on the stack (the centermost circle) which then returns some kind of a response. It flows back out through each piece of middleware, each of which is given a chance to manipulate the response in some way.

Alternatively, middleware can determine it's not going to pass control on to the next piece of middleware. Imagine the third circle was a validator of some kind which determined the request could not possibly proceed. Rather than pass on a broken request until it fails, we can stop processing early by returning a `Response` or an error object. This could be done via throwing an exception or just returning a `Response` object which represents the error that needs to be addressed if the caller wants to have the request succeed. This is represented by the curve to return back out in the third circle.

FIGURE 2

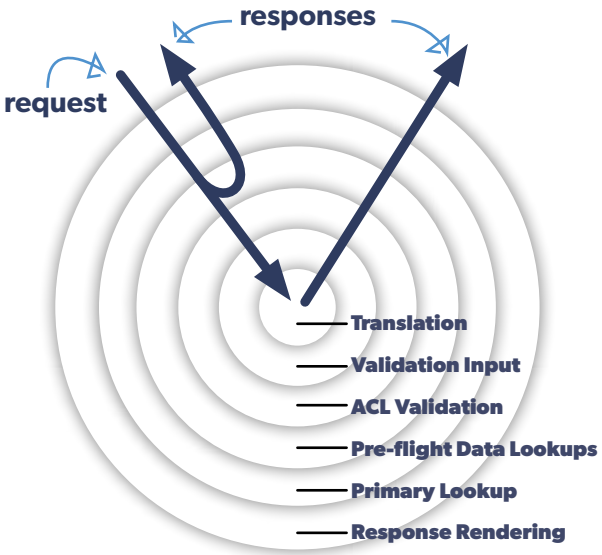


Figure 2 shows some example but feasible uses for middleware. On the outermost circle, we have a translation service. Presuming we have some magical technology which could translate any outgoing `Response`, this would be a good place for that middleware. It means on the way in it does nothing, but on the way out, whatever is returned could be translated into whatever language it needs to be. This would include everything from a fully baked and proper response, to any errors which may have happened in the subsequent levels.

In the second layer, we have input validation. If the request doesn't include the proper inputs, we can determine early on that the request will be an error. We can make that determination and return early. The response would be translated as appropriate, and we don't end up doing any extra work.

If the inputs are valid, it will pass through to the ACL (access control list) layer where we could determine if the user should be allowed to access the resource. If not, we can return early with an error indicating the user is not allowed to do whatever they are trying to do. In this case, the ACL error would return back through the *Validate Inputs* middleware, but remain unchanged until it got back to the *Translation* level.

Depending on your preferences, the ACL level could be swapped with the input validation middleware. This would allow you to kick an unauthorized user out of the application sooner, and avoid even doing the work of validating inputs. If the work we're doing is considered open or public, the ACL layer could be omitted, the same way the validation layer could be left out if there's nothing to validate. The ACL level itself could be its own middleware pipeline, splitting the work into smaller, more discrete units. This could include a layer to authenticate a user, a layer to load permissions for the user, and a layer to figure out what permissions are required for the route, and another to determine if the permissions the user has match the ones they need.

Assuming everything is good to this point, we can move on to a theoretical "pre-flight" set of data lookups which could include database or API calls in order to make the "meat" of the request possible. This could move into the main lookups or retrieval, followed by another level which would render the response as JSON or something else. This level could, again, be another pipeline which determines what type of response is expected and formats the output accordingly.

Needless to say, there are many options. Each piece of middleware can be simple and straightforward, doing only one single thing on the way to transform a request to a response. Thus, each piece of middleware can be easily unit-tested. It also means there is a significantly simpler mental model for what will happen on the path to transform a `Request` to a `Response`. There's no need to try to remember all the events or pathways code may take to get through an application.

Incremental Building

Let's imagine we're trying to build an API which will allow a user to retrieve a filtered list of United States state abbreviations. We want to ultimately be able to filter a list, as well as returning the list in a couple of different output formats, ensuring only allowed users can get these top secret state abbreviations. That's quite a few things, but we can approach each one, in turn, incrementally adding functionality by wrapping middleware in other middleware that gives our application more functionality.

To start, we need to be able to return data. We can fake this part for now (see Listing 1).

LISTING 1

```
01. <?php
02. class StateReturner
03. {
04.     public function __invoke(Request $request,
05.                              Response $response,
06.                              $next) {
07.         $list = ['CA', 'CO', 'CT'];
08.
09.         return $next(
10.             $request->withAttribute('stateList', $list),
11.             $response
12.         );
13.     }
14. }
```

The `StateReturner` class provides data to the `Request` object and passes it on. It is not responsible for figuring out how the state codes will be represented in the final output; its only job is to provide the state codes so some other piece of middleware will be able to process it, ultimately ending up with proper output.

As a side note, the code examples I'm giving are not targeted to work under any particular middleware framework, but with slight modifications in one direction or another, they theoretically could work with quite a few of them. The `Request` and `Response` objects I'm type hinting would follow closely with PSR-7's[4] `ServerRequestInterface` and `ResponseInterface`.

Suppose we add this to our pipeline as the only piece of middleware. We won't see much since nothing is yet returning a `Response` object. With only this piece of middleware, we need something else which can transform it into data that can be rendered.

```
Pipeline:
[ StateReturner::class ]
```

Let's build something, Listing 2, which will take our `stateList` and put it in the `Response`.

LISTING 2

```
01. <?php
02.
03. class JsonStateOutput
04. {
05.     public function __invoke($request, $response, $next) {
06.         $realResponse = $next($request, $response);
07.         $stateList = $request->getAttribute('stateList') ?? [];
08.         $realResponse = $realResponse->withHeader(
09.             'Content-type', 'application/json'
10.         );
11.
12.         return $realResponse->getBody()
13.             ->write(json_encode($stateList));
14.     }
15. }
```

Now, we have a piece of middleware which takes the `stateList` value from the request, JSON encodes it, and puts it into a response it returns. Even though there's not a lot of code, I feel it's worth stepping through each line to make sure we're on the same page.

Starting out, we call the next piece of middleware and store its return value (a response) in `$realResponse`. I named it that to make sure it was distinct from the passed in `$response` parameter. Additionally, I usually try to avoid changing the incoming parameter variables in my functions unless the purpose of the function is a side effect.

We don't know what else may come after this middleware, but it doesn't matter. It could be the final middleware, in which case the value of `$realResponse` will be the same as the incoming `$response` parameter. It could also be something else, or a bunch of other middleware.

Next, we retrieve the `stateList` from the `$request`. This means for this to work, the list of states must be already in place when the `JsonStateOutput` middleware is called. It should come after the `StateReturner` middleware:

```
Pipeline:
[ StateReturner::class,
  JsonStateOutput::class ]
```

We make a new `Response` object which includes the proper content type header. We then take the new `Response` object, create another new one based on that, but now with the body content added containing a JSON encoded list of "C" states.

At this point, the application will run through `StateReturner`, placing the list of states starting with `"C"` in an array stored as an attribute of the `Request`. Then, we invoke the next middleware which is the `JsonStateOutput` middleware. It, in turn, immediately passes control on to the next piece of middleware. The `JsonStateOutput` will do its job on the way back out. Once the response is returned, the `JsonStateOutput` class augments it with a body with that list of states rendered as JSON. That response is then returned to whatever piece of middleware called the `StateRunner`.

At this point, we should see a JSON list of states with names startong with "C". Each of these middleware classes we've built so far is easily unit testable.

The Order of Things

With this style of middleware, it is certainly possible to manipulate and change the response on the way in. However, I'd highly recommend avoiding that. It takes some discipline, but there's a good reason. If `JsonStateOutput` had placed the JSON in the response before calling `$next`, then the next piece of middleware could completely remove or mess up the JSON body. On the way out, though, it's another story. We're moving from least important or critical middleware to more important.

If we were to follow the other middleware pattern, specifically the signature that doesn't have the `Response` object in the incoming parameters in its signature, we wouldn't even have the option to modify the response on the way in. The pattern I feel is best for middleware is to work with the `Request` on the way in, meaning before the call to invoke `$next`, and to work with the `Response` on the way out, that is, after the call to `$next`. If your middleware is only doing one thing (which it should), then it would likely fall into one of two patterns.

```
// Incoming middleware that manipulates the Request
public function __invoke($request, $response, $next) {
   // Make changes to $request
   $newRequest = $request->withAttributes(
      'whatever', 'you want'
   );
   return $next($request, $response);
}
```

or:

```
// Outgoing middleware responsible for changing the response
public function __invoke($request, $response, $next) {
    $outgoingRequest = $next($request, $response);

    // Figure out modifications
    return $outgoingRequest->withChanges();
}
```

In the example code, `StateReturner` follows the incoming

4 *PSR-7's: http://www.php-fig.org/psr/psr-7/*

side, while `JsonStateOutput` follows the outgoing pattern. Again, rarely should a piece of middleware need to modify the incoming `Request` *and* the outgoing `Response`. I'd also argue other than making it easy to return a response without needing to instantiate it, the incoming `$response` argument should probably not be used unless you know a piece of middleware is going to be the final middleware.

Multiple Output Formats

To be able to return different formats for the data, we need to figure out what the `Request` is asking for. We could make another couple of middleware for `XmlStateOutput` and `CsvStateOutput`. Each of those could look into the request and process the "Accept" header. If it didn't match the appropriate type for each middleware, they could essentially become pass-through objects. However, this means that processing of the Accept header would be duplicated in each output rendering middleware. Instead, we can create a piece of middleware which would process the Accept header and set an attribute on the request before sending it through to the rest of the stack. Each of the output middleware could then look something like Listing 3.

LISTING 3

```
01. <?php
02. class JsonStateOutput
03. {
04.     public function __invoke($request, $response, $next)
05.     {
06.         $realResponse = $next($request, $response);
07.         // **** NEW ****
08.         if ($request->getAttribute('outputType') != 'json') {
09.             return $realResponse;
10.         }
11.         // **** END NEW ****
12.
13.         $stateList = $request->getAttribute('stateList') ?? [];
14.
15.         $realResponse = $realResponse->withHeader(
16.             'Content-type', 'application/json'
17.         );
18.
19.         return $realResponse->getBody()
20.                             ->write(json_encode($stateList));
21.     }
22. }
```

The conditional ensures JSON should be returned. Before we get to this piece of middleware, we should have another middleware which determines the output type from the `Accept` headers and sets the `outputType` attribute as appropriate.

Our pipeline could look like this:

```
Pipeline:
[ NoOutputError,
  GetOutputType,
  JsonStateOutput,
  XmlStateOutput,
  CsvStateOutput,
  HtmlStateOutput,
  StateReturner ]
```

There are still additional pieces which would need to be built. Obviously, we'd want to replace the `StateReturner` with something that can return more than three state abbreviations. If we want to ensure only authorized people can utilize our state abbreviations API, we'd need to build a set of authentication and authorization middleware, as well as potentially some to deal with ACL.

We'd need to write some code to process the query string parameters into a filter which could be used to limit the states that are returned.

We may want to build some middleware to modify the JSON output to add HAL or other metadata to the response. We would definitely want to revisit the `StateReturner` so it gets the list of state abbreviations from a database or a file. Depending on if we went with the database or a file, we may have another piece of middleware which would come into play for actually filtering that list down.

In short, we have tons of possibilities, and each one of the individual middlewares should be pretty simple. Each modification we make can be done incrementally, moving from one working state to another. Instead of the path many applications take, where new features get tacked on and slammed in wherever they appear to fit, eventually resulting in a pyramid of doom and technical debt waiting to collapse under the weight of its own sadness (thanks, Chris Pitt), we can build small, incremental, loosely coupled, easily testable bits of code.

In our example so far, we can assume there's some kind of route which points to this pipeline. In addition to route specific middleware like this, most frameworks will allow you to pipe in middleware which will automatically be executed before or after any route related middleware are executed. This would potentially be a good place to inject middleware for authentication, authorization, error handling, output rendering and any other generic and non-route or non-resource specific code we may need.

Because of this, if we build the entire application through middleware, there will be multiple endpoints, routes, and functionality. We can upgrade and add functionality across the board just as easily as adding a feature to an individual route or API.

Middleware Frameworks

There are a number of current PHP frameworks for building middleware applications. From Zend, there's the Expressive framework[5] and Stratigility[6]. Additionally, Zend Framework 3 provides middleware execution by default now. Additionally, there's Slim[7], Relay[8], Equip[9], and Stack[10]. If you're a fan of Symfony, Silex[11] provides a way to run middleware with Symfony components. If you prefer the Laravel ecosystem, Laravel[12] can do middleware as well.

In short, there's no shortage of options; you just have to choose the one that fits best with what you're familiar with or what you want to learn. Each of the frameworks brings something a little different to the table. Expressive, Slim and Stratigility appear to work in nearly the same way. Several of these are based on PSR-7, each providing their own way to handle HTTP messages.

5 *Expressive framework:* https://zendframework.github.io/zend-expressive/

6 *Stratigility:* https://github.com/zendframework/zend-stratigility

7 *Slim:* http://www.slimframework.com/docs/concepts/middleware.html

8 *Relay:* http://relayphp.com

9 *Equip:* https://equipframework.readthedocs.io

10 *Stack:* http://stackphp.com

11 *Silex:* http://silex.sensiolabs.org/doc/master/middlewares.html

12 *Laravel:* https://laravel.com/docs/5.3/middleware

Conclusion

Building applications following the middleware pattern is a great way to build complex applications while keeping the code small, simple, maintainable, and testable. The middleware pattern allows for an easy way to mentally track through and understand the flow of an application. It allows for fast execution because there's little overhead. Pieces of middleware can be reused or even automatically piped in for any or all application routes which provide a convenient way of upgrading and adding functionality with minimal effort. I'd highly recommend checking out one of the middleware frameworks or looking more into middleware and seeing how it could work for you.

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.

Automate Tasks Away With Robo the Faithful Task Runner

Matthew Setter

Tasks, maintenance, automation—no matter what you call it, it's a core part of any project we're involved in. It's also a part of a project which can, if we're not careful, suck the life out of it.

If you're not sure what I mean, as a developer, the thing we want to do most is to achieve outcomes. These can be shipping new releases and new features, or fixing bugs in existing releases or features. To do that, we do what we love most—code.

Code doesn't exist in a vacuum; like everything in life, it never has and never will. Therefore, to be able to develop there are always a number of miscellaneous, related tasks we have to take care of, whether on a regular or intermittent basis.

When we're developing, there are setup tasks, such as creating cache and log directories, starting Docker containers, cloning project source code from GitHub or Bitbucket, and so on. It can be easy to get lost, even bogged down in doing maintenance work, instead of creating, shipping, or fixing.

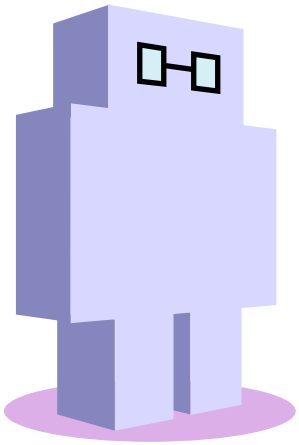
While that can be fun now and then, over the longer term it's not effective, nor contributing to the bottom line (our income). Much less if every developer on a project has to perform the same tasks manually. We need to be able to identify these tasks and put automation in place so they can be taken care of quickly and efficiently, without us having to get too involved.

At this stage, you might be wondering why I'm talking about this. You might also be thinking, why am I writing about it? It's neither a new subject nor one that's lacking in supporting tools. And, on both counts, you'd be correct. These kinds of tasks have been around since computers and software development were first invented, and we're not lacking in tools to help us.

Using Linux as our operating system of choice, we can create scripts which help us in any of the commonly available shells, such as Bash, Zsh, and Csh. If that's not your cup of tea, we can, instead, use a range of prepackaged tools, such as Capistrano[1], Phing[2], Make[3], and Ant[4].

Now, all of these are great tools; I've used each and every one of them (except Csh). Perhaps you're already familiar with some, or all, of them. Considering this column's goal is to continue expanding your awareness and knowledge of available tools and technologies, I want to let you know about another automation tool. This one I came across recently, thanks to our very own Editor-In-Chief, Oscar Merida. It's called Robo[5].

1 *Capistrano:* http://capistranorb.com
2 *Phing:* https://www.phing.info
3 *Make:* https://www.gnu.org/software/make/
4 *Ant:* http://ant.apache.org
5 *Robo:* http://robo.li



Introducing Robo–The Modern Task Runner

Robo is self-described as a:

> *"Modern Task Runner for PHP"*

It's written by the good folks at Codegyre[6], the same folks behind awesome tools such as Codeception[7] and AspectMock[8]. It's an excellent package for managing tasks which you need to take care of on a regular basis. The other reason I wanted to talk about it, is it's written in PHP.

Before we dig in too deeply, there's a lot to be said for using the best, or most focused, tool or language for the job at hand, instead of sticking to what you know. For example, use PHP for the web, use Bash, Zsh, Ruby, or Python for system tasks, and so on.

6 *Codegyre:* http://codegyre.com
7 *Codeception:* http://codeception.com
8 *AspectMock:* https://github.com/Codeception/AspectMock

But in recent times, the lines have blurred quite a lot in this regard. PHP has become so much more flexible in what it can do away from a web-centric focus. And as we're all about PHP (and its wider ecosystem), then why not choose a tool written in PHP for command-line task automation? Certainly, we'll be productive quicker by sticking with a familiar language. With that qualification out of the way, let's have a look at what it offers.

Whether you only need to perform some simple tasks, such as running Composer, cleaning or deleting cache directories, executing background tasks, or preparing an application's assets, there are a set of prepackaged tasks for Robo to do that. If you're looking for something more specific, you can do that as well.

Installing Robo

But first, we need to install Robo. We're able to do so using either Composer,

```
# Install via Composer
composer require codegyre/robo
```

or by downloading a phar file, as you can see in the commands below:

```
# Download the Phar file
wget http://robo.li/robo.phar

# Make it executable, and put it in the system path
sudo chmod +x robo.phar && mv robo.phar /usr/bin/robo
```

Now, there's one more step to go before we can begin creating tasks, which is to initialize the core Robo file. To do this, in the terminal in the root directory of your project, run `vendor/bin/robo init`. It will create a new file, called `RoboFile.php`, which looks like this:

```
<?php
/**
 * This is project's console commands configuration for \
  Robo task runner.
 *
 * @see http://robo.li/
 */
class RoboFile extends \Robo\Tasks
{
    // define public methods as commands
}
```

You can see it does nothing other than extending `\Robo\Tasks`. Now, let's get started fleshing it out, so it does something meaningful.

Creating Our First Task

Let's say we need to automate the process of getting a project set up and running for new developers on a project. To ensure they're ready to begin work, after they've cloned the project source, they need to run the following tasks:

1. Install 3rd-party PHP dependencies.
2. Install 3rd-party front-end dependencies.
3. Run the test suite to ensure everything works.

Let's step through and create a set of tasks, and bring them together in `RoboFile.php`, so you learn the basics of how Robo works. Nothing better than a practical example to learn by doing, rather than a trivial "hello, world" type example which teaches next to nothing.

Before we get started, a word on Robo and autoloading classes. Robo will look for a `vendor/autoload.php` file and load it if it is found. If you are adding it to a project that already uses `Composer` and it's autoloader, you'll be set. You can make a minimal `composer.json` file with the following to generate an autoloader to follow along with this month's code.

```
{
    "autoload": {
       "psr-4": {
         "Project\\": "src/Project"
       }
    }
}
```

Don't forget to run:

```
composer dump-autoload
```

Assuming we have a custom namespace, called `Project`, located under `/src/Project`, in that directory create a file called `SetupProject.php` which extends `BaseTask`, such that it looks like Listing 1.

LISTING 1

```
01. <?php
02.
03. namespace Project\Tasks;
04.
05. use Robo\Task\BaseTask;
06.
07. class SetupProject extends BaseTask
08. {
09.     protected $path;
10.
11.     public function __construct($path) {
12.         $this->path = $path;
13.     }
14.
15.     public function run()
16.     {
17.
18.     }
19. }
```

Next, create a second file called `loadTasks.php`, which looks like Listing 2.

LISTING 2

```
01. <?php
02.
03. namespace Project\Tasks;
04.
05. trait loadTasks
06. {
07.     protected function loadSetupProject($path)
08.     {
09.         return new SetupProject($path);
10.     }
11. }
```

FIGURE 1

```
Robo version 1.0.0-RC3

Usage:
  command [options] [arguments]

Options:
  -h, --help                            Display this help message
  -q, --quiet                           Do not output any message
  -V, --version                         Display this application version
      --ansi                            Force ANSI output
      --no-ansi                         Disable ANSI output
  -n, --no-interaction                  Do not ask any interactive question
      --simulate                        Run in simulated mode (show what would have happened).
      --progress-delay=PROGRESS-DELAY   Number of seconds before progress bar is displayed in long-running
 task collections. Default: 2s.
      --supress-messages                Supress all Robo TaskIO messages.
  -v|vv|vvv, --verbose                  Increase the verbosity of messages: 1 for normal output, 2 for mor
e verbose output and 3 for debug

Available commands:
  help                  Displays help for a command
  list                  Lists commands
 collection
  collection:builder
 project
  project:prepare
```

Then, in `RoboFile.php` add the following code in the body of `RoboFile` class:

```
use \Project\loadTasks;

public function projectPrepare()
{
    $this->loadSetupProject(__DIR__)
         ->run();
}
```

`SetupProject` is our custom task, which we'll be able to use via Robo when we're finished. In there, as we'll see shortly, we'll flesh it out so it contains all of the tasks which we need to carry out.

The `loadTasks` trait is Robo's loader convention. In it you create a method, prefixed with `load` which returns a new instance of your task object, in our case `SetupProject`. That way, when you're reading through an existing setup or perusing the source of the pre-packaged tasks, the cognitive load is greatly reduced, making it easier to quickly know what a task does, how it's loaded, and where its source files are located.

Finally, the `projectPrepare()` function in RoboFile creates a task which can be called from the command-line. If we were to run Robo now, from the command-line, we'd see output similar to the one in Figure 1. There, at the bottom, under "Available commands:" you can see we have one command, "project:prepare", listed under the heading "project."

Note how the names were auto-generated, based on our namespace (`project`) and function name (`projectPrepare`). The camel-cased name was converted to a colon-separated name (`project:prepare`). Keep this in mind as you're naming your task functions. With that done, let's start fleshing out our tasks.

Install 3rd-Party PHP Dependencies

To setup 3rd-party dependencies, we're going to use Composer, PHP's tool of choice. Specifically, we're going to use the Composer task methods[9]: `taskComposerUpdate()` and `taskComposerInstall()`. To make these available, in `SetupProject.php`, use the trait `Robo\Task\Composer\loadTasks`, and add the code below:

```
public function composer() {
   if (file_exists($this->path . '/vendor')) {
      $this->printTaskInfo('Running Composer Update');
      $this->taskComposerUpdate($composerPath)
          ->optimizeAutoloader()
          ->run();
   } else {
      $this->printTaskInfo('Running Composer Install');
      $this->taskComposerInstall()
          ->optimizeAutoloader()
          ->run();
   }

   return $this;
}
```

To call the method, add `$this->composer` to the `run()` method.

Here, you can see if Composer's `vendor` directory exists, we'll update the existing dependencies. If it doesn't exist, we'll install them instead. Admittedly, this is a bit more than what I said I'd cover. But, why not go a bit further and help to future-proof the code.

Note that before calling either task, I've called the `printTaskInfo` method. This method allows us to write out some task-specific information to the console; quite handy when there's a lot going on.

When written out, the string we provide will be prefixed with task class' name, in this case, `SetupProject`. If we didn't want that, we could also use the `say()` method. This prints out the

9 *the Composer task methods:* http://robo.li/tasks/Composer/

FIGURE 2

```
 [Project\SetupProject] Running Composer Update
 [Composer\Update] Updating Packages: /usr/bin/composer update --optimize-autoloader
Loading composer repositories with package information
Updating dependencies (including require-dev)
Nothing to install or update
Generating optimized autoload files
```

text we supply, but without any context-specific information. You can see both of these in action in Figure 2.

Install 3rd-party Front-End Dependencies

Most web applications use a range of 3rd-party, front-end dependencies, such as *AngularJS*, *jQuery*, *Font-Awesome*, and *underscore*. One of the best ways I know of to make managing them effective is to use the Bower package manager[10].

If you've never used Bower before, it's a lot like Composer. Project configuration is managed via a `bower.json` file, which follows a very similar structure to Composer's `composer.json` file, as you can see below:

```
{
  "name": "app-name",
  "version": "0.0.1",
  "dependencies": {
    "sass-bootstrap": "~3.0.0",
    "modernizr": "~2.6.2",
    "jquery": "~1.10.2"
  }
}
```

Here, we've listed the project's *name*, *version*, and *dependencies*. To avoid running this manually, such as by calling either `bower install` or `bower update`, let's add a custom Robo task.

As before, create a new function, called `bower()`, in `SetupProject.php`. In there, use the `\Robo\Task\Bower\loadTasks` trait and fill out the function, so it looks as follows:

```
public function bower()
{
   $this->taskBowerInstall()
        ->noDev()
        ->run();
}
```

Then, in `run()`, under `$this->composer()`, add `$this->bower()`. One thing worth noting, if you don't have a `bower.json` file for the project, this step is going to fail. For sakes of simplicity, use the one I've included above.

LISTING 3

```
01. <?php
02.
03. namespace Project;
04.
05. use Robo\Task\BaseTask;
06.
07. class SetupProject extends BaseTask
08. {
09.    use \Robo\Task\Composer\loadTasks;
10.    use \Robo\Task\Bower\loadTasks;
11.
12.    protected $path;
13.
14.    public function __construct($path) {
15.       $this->path = $path;
16.    }
17.    public function run()
18.    {
19.       $this->composer();
20.       $this->bower();
21.       $this->tests();
22.    }
23.
24.    public function composer() {
25.       if (file_exists($this->path . '/vendor')) {
26.          $this->printTaskInfo('Running Composer Update');
27.          $this->taskComposerUpdate()
28.             ->optimizeAutoloader()
29.             ->run();
30.       } else {
31.          $this->printTaskInfo('Running Composer Install');
32.          $this->taskComposerInstall()
33.             ->optimizeAutoloader()
34.             ->run();
35.       }
36.
37.       return $this;
38.    }
39.
40.    public function bower() {
41.       $this->taskBowerInstall()
42.          ->noDev()
43.          ->run();
44.    }
45.
46.    public function tests() {
47.       $this->taskPHPUnit()
48.          ->bootstrap('phpunit.xml.dist')
49.          ->json()
50.          ->xml()
51.          ->run();
52.    }
53. }
```

10 *the Bower package manager:* https://bower.io

Run the Test Suite

Now let's assume we have a test suite which makes use of the de facto testing standard for PHP, PHPUnit. As before, add a new function, which we'll call `tests()`. This will require the `Robo\Task\Testing\loadTasks` trait. Robo has PHPUnit support baked in! The function will look as below, when completed:

```
public function tests()
{
    $this->taskPHPUnit()
        ->bootstrap('phpunit.xml.dist')
        ->json()
        ->xml()
        ->run();
}
```

Here, what we're doing is running the PHPUnit tests, specifying `phpunit.xml.dist` in the root directory, as the test bootstrap file, and that we want to log to JSON and XML formats. Next, add `$this->tests();` to the run() method.

The complete `SetupProject` class should look like Listing 3.

Running the Tasks

With all of this setup work completed, in the terminal, from the root of your project, run `robo project:prepare`, which will work through all of the tasks which we've setup, running them in sequence.

You can see it's installed the project source, installed the front-end dependencies via Bower, and run the tests. When it was finished, it output the string "Setup Complete." It's not the most sophisticated of setups, but it illustrates how to deal with a handful of tasks which we'd likely encounter on a regular basis.

There's Far More to Robo (Than Meets the Eye)

I admit, Robo has more to offer than what I've shown you here. In addition to what I've covered, there are pre-packaged tasks for working with source repositories, working with remote servers via ssh or rsync, compiling, minimizing, and optimizing assets, even the ability to work with NPM. You can even bring a series of existing shell scripts together.

As a result, if you're new to automation, or have an existing automation setup, you can leverage it if you make the decision to migrate to Robo (or if you just want to test the waters).

In Conclusion

And that's Robo and how to use it to automate the process of getting a new developer up and running on a project. Sure, there are other tools, such as Phing, Make, or Ant, which have been around longer and are more widely known.

But don't underestimate Robo. In the brief time I've been using it, it's already turning out to be a pretty handy addition to my toolchain. Perhaps you may feel the same way soon.

It's an excellent tool, with a host of prepackaged tasks, and the ability to create fairly detailed custom tasks—ones tailor-made to your exact requirements. Give it a go and see what you think.

Matthew Setter is an independent software developer, specializing in creating test-driven applications, and technical writer http://www.matthewsetter.com/services/. *He's also editor of Master Zend Framework, which is dedicated to helping you become a Zend Framework master? Find out more* http://www.masterzendframework.com/welcome-from-phparch

Conference !== Family Reunion

Cal Evans

As I sit here and write this, Pacific Northwest PHP (#PNWPHP) is taking place. I am watching the hashtag on Twitter, and it is fun to see old friends get together, slap backs, hug, and catch up. Part of me is a bit jealous I'm not there because I know what I am missing out on. Fun times and learning for the next few days.

If you have ever been to a PHP conference, you know the drill. There is one area—usually the area with the most comfortable chairs—where everyone gathers, hangs out, shoots the breeze, and catches up with each other. Around noon, the group moves en masse to the feeding troth known as lunch, grabs a table, drops their stuff, and gets food.

The lunch conversation isn't much different from the conversation at the couch.

"Hey, whatever happened to…?"

"Remember when…?"

"OMG, do you remember that time…"

The more conferences you attend, the more of these stories you not only know, but can recite by heart for the benefit of those who haven't been around as long.

Wait. That's not *your* experience at a PHP conference? Because it's been my experience for around 9 1/2 years now. Each one I attend is roughly the same. Honestly, I can't remember the last time I sat through someone else's session unless I just needed a quiet place to catch up on my email. I'm always sitting around at "the spot."

Conferences Aren't the Same for Everyone

I really didn't realize how different the experience was for people not "in the group" until I attended a WordCamp earlier this year. (I am not dissing on WordCamps.) See, at the WordCamp, nobody knew me. I traveled the conference in total anonymity. It sucked. I was surrounded by 700 people who all use WordPress in some way, and I was alone.

It wasn't difficult to find "the spot," but I wasn't invited to come sit. There was one lunch table which was obviously "the group's" table, but they were all busy laughing about the latest "Remember when…"

OK, so I wasn't totally alone, the lovely and talented Kathy was with me, and I ran into two of the organizers, one of which was an old friend; still, I was not traveling in "the group." It was a very different experience.

It wasn't until I saw the PNWPHP tweets come across and began thinking "what a wonderful time PHP conferences are because they are like family reunions" that I got to thinking about my WordCamp experience. I began to wonder if other attendees at PHP conferences experienced what I experienced at WordCamp. If so, what we as a community can do to make this suck less?

Two Problems

First, before I go too far down this road, I want to say I am not picking on "the group." There are two problems at play here, and we need to find a solution which works for both groups, not just one or the other.

Problem One

"The group" as a whole travels around together because they are friends. Even though many of us are friends, we rarely see each other. So when we do get together in person, we want to make the most of it. Catch up, share some laughs, and sometimes even do some business. There is nothing wrong with this. Friends should be able to catch up.

Problem Two

The other problem is if "the group" only talks to each other, then new people feel ignored. They see the group, but they don't feel right in joining in on the group's festivities. Especially if the attendees in question are less outgoing and shyer. (I know right? A shy developer. Like that exists.)

This is not great.

So What Do We Do?

Honestly, I have no idea. I'm not prescribing a solution here so much as laying out the problem. There is no one, single solution which fits everyone. Everyone—members of "the group," new attendees, and conference organizers—each have to find their own solutions.

For me, one of the things I've been trying to do for the past several years is make sure when lunch rolls around I sit at a table other than the group table. I start conversations with the attendees; I try to get people talking to each other, not just to me. This works for me. I am not saying it is the end-all-be-all answer everyone should adopt.

If you're part of "the group," take a moment and reflect on what it was like to go to your first conference or one where you didn't know anyone. Make an effort to meet some new faces during the conference. Start with one lunch or one of the social activities.

I've also suggested to conference organizers that they hold a webinar the week before their conference for new attendees. Introduce them to the conference, let them know what to expect, and give them tips on getting the most out of the conference. These tips would include things like who in "the group" are friendly and can be approached and talked to. (Hint, it's everyone.) Let new attendees know how and when to approach speakers too. Please, not right before their talk! (But that's another whine.)

At ZendCon last year, I met someone who had never been to a conference before. I made it my mission to introduce them to people I knew every time I saw them. My hope was that they would feel like they had friends there too by the end of the week. This year if they come back, they will be able to walk down the hall and wave at friends they talk to online. Hopefully, they will stop to talk to someone, and I'll overhear "Hey! Remember last year when…" :)

No Really, What Do We Do?

Seriously, this is not a "YOU MUST DO THIS!" OMR[1]. I am letting those of you in "the group" know there is a problem and asking you to find your own solution. I'm also letting you know if you've ever felt like an outsider at a PHP conference, you don't need to. We are all kind of shy (Well, except for Jeremy Mikola) but most attendees are nice people. Walk up to any attendee and say hi and you can usually get a good conversation going. That's all it takes.

> *Tech conferences are where a group of introverts gather to cosplay extroverts.*
>
> *– Larry Garfield, @crell*[2]

Yes, "the group" wants to catch up with friends. Yes, they will yuk it up to the latest "Do you remember when's…" (If you don't have any, find me at the next conference, I'll teach you a couple about Ben Ramsey that are guaranteed to get laughs!) But that doesn't mean you can't be a part of it. Walk up and start listening. Laugh with them.

Make time to meet new people, and make new friends. You never know, it could be that you only think you are part of the coolest group. It could be that there is a much cooler one at the next table.

No, not that one, that one is mine… That one over there.

1 *Old Man Rant*

2 *Larry Garfield, @crell: https://twitter.com/crell/status/648016795086028800*

Past Events

September

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

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

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

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

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

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

Upcoming Events

October

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

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

Brno PHP Conference 2016
October 15, Brno, Czech Republic
https://www.brnophp.cz/conference-2016

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

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

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

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

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

November

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

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

December

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

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

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

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

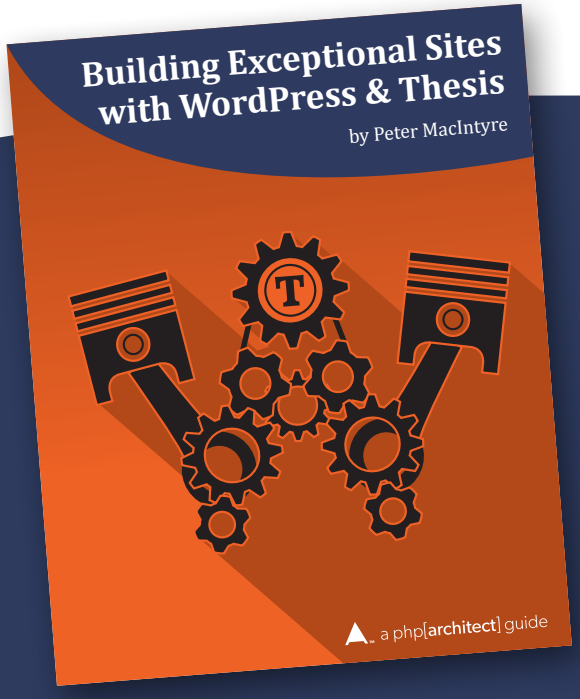
PHP UK Conference 2017
February 16–17, London, U.K.
http://phpconference.co.uk

March

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

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

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.



Two-Factor Authentication Is Just a Text Away

Chris Cornutt

In last month's article I introduced you to some of the basic concepts around two-factor authentication and how to integrate one method into your application: time-based one-time password handling. This method comes with handy benefits (like being able to use it offline) but it has one major drawback—you have to have an application installed on your device to get the codes. While a large percentage of the population has access to smart devices which can run these applications, there's a decent sized chunk that can't run them. This means potentially cutting off part of your audience just because they can't use their device that way.

What's an alternative that can be more widely used? Well, chances are if you have a phone in your possession you've already used it at least a few times today: the humble text message. Text messaging has been a part of our handy communication devices for years (decades?) now. They've evolved into a major form of communication, especially among some of the younger generations which grew up with these devices as a part of their lives. Text messages continue to be a dominant form of communication, even evolving into a communication channel for interacting with other services (so-called "chatbots"). Because they are ubiquitous, a text message is a natural fit for users with any type of devices to be able to use two-factor authentication in your service.

A Word of Warning

One quick note before I get started, though. If you read last month's column you probably remember some comments about avoiding text and SMS-based two-factor handling as prominent security professionals do not recommend it. While I do agree with some of their worries about this setup, I feel like the benefits outweigh some of the negatives in most cases. There are a lot of what-ifs with text messages and the security of their contents, but the truth is they make for a simple and easily accessible option for a wide range of users.

The Setup

Now, on to the good part—how are we going to implement and integrate this system into your application? If you remember the setup ROM last month you're already one step ahead here. The basic idea is similar:

1. The user comes in and sets up their device (here this means storing an identifier related to it).
2. They're sent an initial code to verify the device as valid and that they have access.
3. Once the code is confirmed, we redirect the user to a page which sends the new code and verifies their entry.

The difference between this SMS method and the one-time (time based) password is in how things are set up behind the scenes. There's also some more interaction involved with the SMS setup. With the application and the time-based rotating code you only need to store a single piece of information: the unique secret that identifies the user's device. This secret is then reused when the user enters the code as a "seed" for the algorithm and allows your code to match up the code it generates with the code the user enters.

In the SMS setup, this unique identifier is most likely going to be the cell number for the end user's device. You can either have the user enter this as a part of the SMS setup or as a setting already on their profile. However you get it, you'll need to save it so it can be reused later when sending the message.

> ***Note:*** *Cell numbers should be treated as personally identifiable information (PII) and should be stored as such. It's highly recommended you look into encrypting the values before storing them.*

For purposes of example, we're going to go with a setup that's only asking for the number as a part of the two-factor setup. First, we'd present the user with a simple form just asking them for their phone number. Be sure this form can *only* be accessed after the user has authenticated successfully. If you're not sure who the user is already, you shouldn't set up any additional security measures.

Once you get this number, you'll need to store it in your database (or whatever data store you're using). As mentioned above, this information should be appropriately protected when stored at rest. The idea is even if there's a vulnerability which allows for a full database dump, your user's device information couldn't be compromised and exposed. Once this device is saved, then we move to the next step— the actual verification. To finalize the two-factor device setup, your application should send a test SMS to the device and require the user to input this code. This provides an extra level of identity before finally enabling this feature. If all goes well and they enter the correct code, they're good to go on their merry way.

"But wait," you may be asking, "how do I know what the *correct* code is?" In the time-based version, it was easy: you just had the algorithm recalculate the value for the current time based on the seed. If they matched everything was happy. With the SMS version, however, it's a little trickier. Since the process involves sending a message with the user's code, we have to have something to compare it against. Now we'll need to add a way to track that.

Let's start getting into some code. Hopefully, it'll make this all clearer. In my examples, I will use the Nexmo service to handle the actual sending of the SMS messages. It's a handy web service with a great developer interface and PHP SDK which makes sending the messages simple. First, let's get a few things installed via Composer:

```
composer require vlucas/phpdotenv
composer require "nexmo/client:1.0.0-beta3"
```

This will install both the `phpdotenv` and Nexmo official PHP SDK for use in our application.

> ***Note:*** *I highly recommend using something like the* `phpdotenv` *library*[1] *for credential handling. If you haven't used it before, here's a quick summary: the library allows you to create* `.env` *files (in the INI format) which you can easily swap out per environment. This also has the added benefit of keeping hard-coded credential information out of your code and repository.*

Now that we have the software we'll need, let's get the other requirement set up: the Nexmo account. They have a trial account level which gives you enough to give the service a shot for testing. Go over there, sign up for the account[2], and come back. Don't worry; I'll wait.

As soon as you're all set up, you'll need two pieces of information to get started: the API key and the API secret. You'll also need to set up a test number to send messages from. Don't worry, you don't have to "rent" one and pay for it. They have ones the demo accounts can use, they just have a "sent by Nexmo" message after whatever you're normally sending. Once you have all that set up and those two pieces of API info, we can go back and start setting up the code. First, create a `.env` file in your directory, remembering to put it somewhere outside of the web server's document root so it's not viewable by the public:

```
NEXMO_KEY=1234abcde
NEXMO_SECRET=5678asdf1234
NEXMO_NUMBER=12148675309
```

Obviously, you'll replace those values with the ones for your own account. Then, we can move into the code for actually sending the messages. I'm going to keep it simple here and provide some snippets you can work into your application, basically as a drop in. Hopefully you're already using Composer in your application, but just in case you're not, there is a `require_once` line in here to get the autoloader pulled in, see Listing 1.

LISTING 1

```
01. <?php
02. require_once 'vendor/autoload.php';
03.
04. // To load in the .env file contents in this directory
05. $dotenv = new Dotenv\Dotenv(__DIR__);
06. $dotenv->load();
07.
08. $credentials = new \Nexmo\Client\Credentials\Basic(
09.     $_ENV['NEXMO_KEY'], $_ENV['NEXMO_SECRET']
10. );
11.
12. $client = new \Nexmo\Client($credentials);
13.
14. $client->message()->send([
15.     'to' => $to, // TBD
16.     'from' => $_ENV['NEXMO_NUMBER'],
17.     'text' => $message // TBD
18. ]);
```

That's all there is to sending the message. There are a few blanks we need to fill in on there, though. Obviously, the `$to` value is going to be the device number you stored previously, but what about the message? As a part of the message we need to generate a randomized code to send to the user. What better way to come up with that code than using the using a `random_*` function, more specifically `random_int`[3]. These functions were added in PHP 7 and make it easy to generate random cryptographically secure values. By convention, most SMS two-factor codes are made up of six digits. While that's not a requirement by any means, it does seem to make it easier for the brain to parse and recall to type into the waiting form.

The second thing we'll need to do with this code is store it. You only ever want to store the latest code which was generated for

1 `phpdotenv` *library:* https://github.com/vlucas/phpdotenv

2 *sign up for the account:* https://dashboard.nexmo.com/sign-up

3 `random_int`: http://php.net/random_int

a device. Trying to set up a "history" of codes is increasing the attack surface someone can abuse and potentially gain access to a user's account. Additionally, be sure you store the code with a *timeout* date so it doesn't last forever. This timeout should be one of the first things checked when you're trying to validate the user's input.

Now let's update our example (Listing 2) with these changes:

The code will then be sent over to the user via the Nexmo API along with the rest of the message. The user should then enter this code into the form on the application, the matching record should be pulled from the database and the two should be compared. Don't forget to check out that expiration date too! You'll notice I put a pretty short timeout on that record. In the typical SMS flow, the user will have their device at least somewhere close by as they're aware of the two-factor requirement. Because of this, I recommend keeping this timeout as short as possible. Obviously, you can tweak it based on your needs, but this is good place to start.

With all this in place the user can then enter the code, you can check for a match, and let them on through. I also advise one other thing: if there's a failure, generate a new code and let the user know it has been sent. This prevents an attacker from using an automated script which can run thousands of requests a second from guessing this code easily.

To review, here's the SMS flow overall:

- Direct an authenticated user to the two-factor setup page and have them enter their number.
- Send a verification SMS to verify the user owns the device and have them verify it.
- Every time the user logs in, require them to enter the code which was sent.
- If there's a failure, regenerate the code and send it again.

It's a relatively simple process and not too much more complicated than the time-based version from last month's article—there's just a few more moving parts to consider. It also adds a dependency on an external service for sending the SMS messages, but in my experience services like Nexmo or Twilio are robust enough to stand up to a lot of abuse.

If the time-based two-factor isn't quite your thing and you still want to be able to reach a wide set of your users, look into this SMS-based method. Remember, defense in depth suggests the more levels you can have protecting your application, the better off you'll be!

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

LISTING 2

```
<?php
require_once 'vendor/autoload.php';

// To load in the .env file contents in this directory
$dotenv = new Dotenv\Dotenv(__DIR__);
$dotenv->load();

$credentials = new \Nexmo\Client\Credentials\Basic(
    $_ENV['NEXMO_KEY'], $_ENV['NEXMO_SECRET']
);
$client = new \Nexmo\Client($credentials);

$code = random_int(000000, 999999);
$message = 'Your code is: ' . $code;

// Store it with its expire timestamp
$pdo = new \PDO(
    'mysql:dbname=myapp;host=127.0.0.1',
    $_ENV['DB_USER'],
    $_ENV['DB_PASSWORD']
);

$sth = $pdo->prepare(
    'insert into `user_two_factor` (`user_id`, `code`, `timeout`)
     values (:userId, :code, :timeout)');
$sth->execute([
    ':userId' => 1,
    ':code' => $code,
    ':timeout' => time() + 180
]);

$client->message()->send([
    'to' => $to,
    'from' => $_ENV['NEXMO_NUMBER'],
    'text' => $message
]);
```

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!

The PHP podcast where everyone chimes in.

www.phproundtable.com

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

The Frederick Web Technology Group
meetup.com/FredWebTech

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

September Happenings

PHP Releases

PHP 7.0.11:
http://php.net/archive/2016.php#id2016-09-15-1

PHP 5.6.26:
http://php.net/archive/2016.php#id2016-09-16-1

PHP 7.1.0 RC 3:
http://php.net/archive/2016.php#id2016-09-29-1

News

Medium.com: Generating Code Coverage with PHPUnit and phpdbg

In this post on his Medium page Elton Minetto shows how to generate the code coverage of your PHPunit tests with better performance using phpdbg. He shows an example of the time difference in running the tests (about 1 minute without versus 22 with XDebug). He went looking for a better way and found this post talking about using phpdbg instead. He includes the "brew" commands to get everything you'll need installed and how to use phpdbg with your coverage calls rather than XDebug. http://phpdeveloper.org/news/24379

Symfony Blog: Symfony Reaches 500 Million Downloads

The Symfony blog has posted an announcement about the major milestone the project has reached - the framework has passed 500 million downloads since the original release of the first versions of Symfony 2. The rest of the post talks about the road that lead to 500 million downloads, the pervasiveness of Symfony components and the work they've done on the Backwards Compatibility Promise and Continuous Upgrade Path. http://phpdeveloper.org/news/24446

Gary Hockin: ConfigAbstractFactory in ZendServiceManager

Gary Hockin has a post to his site today introducing you to the new `ConfigAbstractFactory` class to work with `ZendServiceManager` in Zend Framework applications. The library helps make the creation of configuration service factories easier than having to write them in code. He talks about being a fan of the "configuration over magic" approach the Zend Framework has and how, with this new library, it makes it even easier to directly link configuration files and the objects created based on their contents. http://phpdeveloper.org/news/24366

SitePoint PHP Blog: Quick Intro: PhpCompatibility for PHPCS—Are You PHP7 Ready?

The SitePoint PHP blog has a quick tutorial posted helping you get your application PHP 7 ready with the help of the PhpCompatibility "sniffs" for the widely used PHP_CodeSniffer tool. The article then introduces the PHPCompatibility set of sniffs for PHP_CodeSniffer and installing them with a "git clone" in the right Standards directory. Also included are some basics for using PHP_CodeSniffer (like the command line options) and an example of some of the output from the compatibility check. http://phpdeveloper.org/news/24445

Pascal MARTIN: Series—Introduction to PHP 7.1

Pascal Martin has made the tenth post in his series covering PHP 7.1 and how it differs from previous versions. While this series was previously mentioned there have been significant updates to the series warranting a new post. There are ten articles so far. http://phpdeveloper.org/news/24407

DotDev.co: Understanding the Laravel Service Container

The Dotdev.co blog has posted a tutorial for the Laravel users out there with the goal of helping you understand the Laravel service container, a key part of the framework's functionality and an extensible feature you can adapt to some of your own needs. The post starts with some of the basics about the container and how objects/instances are bound to it. They give an example of binding a `FooService` class in the "register" methods of providers. A code example is also included showing how to use the service you previously bound. http://phpdeveloper.org/news/24401

Master Zend Framework: How to Build a Docker Test Environment

The Master Zend Framework site continues their series covering the creation of a Docker-based testing environment in this second part highlighting the addition of testing support. Since unit tests can be run locally if need be (they shouldn't need any resources from the service if they're true unit tests) he focuses on acceptance testing. For his examples he uses the Codeception testing tool. He walks you through the setup of some simple tests based on the "home" page functionality of the Zend Expressive skeleton application. http://phpdeveloper.org/news/24450

Remi Collet: Microsoft SQL Server from PHP

In this recent post to his site Remi Collet shows you how to set up your PHP installation to allow it to work with a Microsoft SQL Server as it's data store. Several different extensions were tested as a part of making the connection to the SQL server. http://phpdeveloper.org/news/24437

Michelangelo van Dam: PHP 7 on macOS Sierra

Michelangelo van Dam has posted a "follow up" to his previous article about setting up a PHP installation on the recent versions of Mac OSX. In this new tutorial he makes some updates for the latest OS X release: Sierra. While he points out that things like XAMPP and Homebrew can be used to set the installation up, he focuses more on compiling and installing it natively. http://phpdeveloper.org/news/24439

Eleven Labs: Use the Symfony Workflow Component

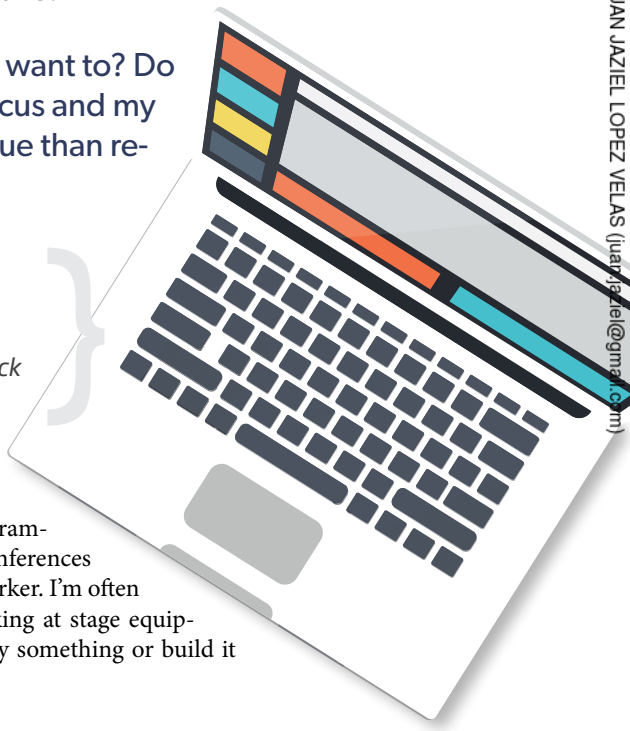
The Eleven Labs site has a new tutorial posted showing you how to use the Symfony workflow component, a component designed to help make performing a sequence of operations simpler. The tutorial starts by helping you get the Workflow component installed (via Composer) and an example configuration defining a flow for a pull request. They then show the command to generate the flow graph so you can ensure the workflow is correct. http://phpdeveloper.org/news/24455

COTS, Open Source, Build it Yourself, or Something in Between?

Eli White

As developers, this is a topic that is near and dear to our hearts. I know for myself it's a question that raises its head every time that I consider a software need I have.

I could just write the code myself and have what I need. But do I want to? Do I want the burden of maintaining new code? Is that where my focus and my is? Or are there better ways for me to spend my time making value than re-creating something that may already exist?



> *Low-level programming is good for the programmer's soul.*
>
> *–John Carmack*

Maybe you can grab a pre-built open-source solution, or buy a paid solution. But what happens if it doesn't quite meet your needs? How do you handle that? Are you left augmenting a half-done solution? Do you change your workflow to match someone else's assumptions? Do you need to throw away the money you spent because it's not good enough, or do you just hobble along with half of a solution?

Tough decisions

Honestly, I don't think there is a single answer—it's just a struggle that we always face. Luckily with the prevalence of easy to use shared libraries and extensible software, nowadays you are often able to find half-way solutions. Rarely is build-it-yourself the real answer as you can always start from somewhere.

Need a basic website and just to throw up some content on a regular basis? Maybe toss up WordPress and write a little bit of custom code to get it working. Need an eCommerce site? Well, your options can range from SaaS solutions like Shopify to full blown extensible solutions such as Magento, to using an eCommerce platform toolkit such as FoxyCart, or at the lowest level just writing your solution integrating with Stripe and Paypal.

But the fact is that no-one should be recreating the wheel. It still always leaves us in a tough situation trying to discern the right solution. I'll admit that I often spend more time trying to research the options that exist to meet a need I have than it may have taken to write the code in the first place. At the same time, if that code isn't part of my core business model itself, then I don't want to spend my time maintaining it.

Heck, this doesn't only stay in the realm of programming. I also run conferences plus I am a woodworker. I'm often debating when looking at stage equipment if I should buy something or build it from scratch.

Conclusions

This month I don't have any conclusions for you, but I'd like to hear from you. What is your make it or break it point? When you do build from scratch, versus just buy a solution? When do you customize something in-between? Take it to twitter and tweet at @phparch[1] to let us know your thought process.

I look forward to hearing what you have to say.

Eli White is the Conference Chair for php[architect] and Vice President of One for All Events, LLC. He may be currently debating some woodworking versus buy decisions for php[world] at the moment. @EliW

1 *@phparch:* https://twitter.com/phparch

Web Security 2016

from php[architect] magazine

Edited by Oscar Merida

Are you keeping up with modern security practices? The ***Web Security 2016 Anthology*** collects articles first published in php[architect] magazine. Each one touches on a security topic to help you harden and secure your PHP and web applications. Your users' information is important, make sure you're treating it with care.

This *Web Security 2016* Book includes:

- An overview of the attacks you should be familiar with and how to protect against exploits.
- Using a PHP-based Intrusion Detection System to monitor and reject requests that attempt to breach your site.
- How to protect against SQL Injection from user-supplied data by using prepared statements.
- A case study in how the Drupal security team keeps core and contributed modules safe.
- How to securely store passwords and understanding the techniques used to crack credentials.
- Using OAuth 2.0 to connect to web services and fetch information for your users without asking for a password.
- How web service security differs from traditional web application security and advice for effectively protecting one from malicious users.
- Identifying the right kind of cryptography to implement in your application and doing it correctly.

Purchase Book

http://phpa.me/web-security-2016

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